# Table of Contents

1. Introduction .............................................................................................................. 1  
   Welcome ...................................................................................................................... 1  
   Jcorporate’s JGroup Services ...................................................................................... 1  

2. Expresso Overview ................................................................................................. 3  
   Application Framework ............................................................................................. 3  
   Services ...................................................................................................................... 3  
   Conclusion ................................................................................................................ 3  
   Contributors ............................................................................................................. 4  

3. Developing an Expresso Application ...................................................................... 5  
   Introduction .............................................................................................................. 5  
   Planning the Application ......................................................................................... 5  
   Create the Packages ............................................................................................... 5  
   Create a Schema Object .......................................................................................... 6  
   Create any required DB Objects ............................................................................. 6  
   Create any required Controller Objects .................................................................. 6  
   Create any required Job Objects ............................................................................ 7  
   Conclusion ................................................................................................................ 7  
   Contributors ............................................................................................................. 7  

4. Application Configuration and Initialization .......................................................... 8  
   Introduction .............................................................................................................. 8  
   Steps to setup an Expresso Application .................................................................. 8  
   expresso-config.xml ............................................................................................... 8  
   struts-config.xml .................................................................................................... 11  
   Other application-specific xxx-config.xml files .................................................... 11  
   Logging Configuration ............................................................................................. 11  
   Schema List Table ................................................................................................ 12  
   Schema Object ....................................................................................................... 12  
   Initialization ............................................................................................................ 13  
   DefaultInit .............................................................................................................. 13  
   Configuration Manager .......................................................................................... 13  
   Setup Values ........................................................................................................... 14  
   LogManager ............................................................................................................ 17  
   JobHandler ............................................................................................................. 17  
   CacheManager ....................................................................................................... 17  
   DB Connection Pool Initialization ......................................................................... 17  
   Struts Initialization ............................................................................................... 17  
   Standalone Application Initialization .................................................................... 18  
   DBTool and DBcreate ............................................................................................ 18  
   DBCreate ............................................................................................................... 19  
   Introduction ............................................................................................................ 19  
   Initializing the Database ....................................................................................... 19  
   Initial Setup .......................................................................................................... 20  
   An Example of How-To Use DBCreate .................................................................. 20  
   DBTool .................................................................................................................... 21  
   Introduction .......................................................................................................... 21  
   Running DBTool .................................................................................................... 21  
   Initial Setup with DBTool ...................................................................................... 21  
   Initial Setup Values ............................................................................................... 22  
   Arguments to DBTool ............................................................................................ 22  
   Conclusion .............................................................................................................. 24  
   Contributors .......................................................................................................... 24  

5. Caching ..................................................................................................................... 26  
   Introduction .......................................................................................................... 26
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processing with JSTL</td>
<td>119</td>
</tr>
<tr>
<td>Processing with JSTL Extended Tags</td>
<td>119</td>
</tr>
<tr>
<td>Rendering the ControllerResponse with XSLT</td>
<td>120</td>
</tr>
<tr>
<td>Conclusion</td>
<td>120</td>
</tr>
<tr>
<td>Contributors</td>
<td>120</td>
</tr>
<tr>
<td>12. Expresso Taglib Descriptions</td>
<td>121</td>
</tr>
<tr>
<td>Introduction</td>
<td>121</td>
</tr>
<tr>
<td>Tag Library Descriptor</td>
<td>121</td>
</tr>
<tr>
<td>Expresso Extended Struts Tags</td>
<td>121</td>
</tr>
<tr>
<td>How to Use Struts Tags in Expresso</td>
<td>122</td>
</tr>
<tr>
<td>Expresso Bean Tag Library</td>
<td>123</td>
</tr>
<tr>
<td>Expresso Logic Tags</td>
<td>123</td>
</tr>
<tr>
<td>Expresso HTML Tags</td>
<td>124</td>
</tr>
<tr>
<td>Expresso Tag Library Summary</td>
<td>124</td>
</tr>
<tr>
<td>Example JSP code</td>
<td>125</td>
</tr>
<tr>
<td>Expresso Tags Described</td>
<td>125</td>
</tr>
<tr>
<td>Expresso Library Tag Details</td>
<td>127</td>
</tr>
<tr>
<td>ActionTag - Renders an Expresso action, and associated hidden fields</td>
<td>127</td>
</tr>
<tr>
<td>AttributeTag - Displays an attribute from an Input or Output object</td>
<td>127</td>
</tr>
<tr>
<td>Back - Displays a &quot;Back&quot; button or image to go back to a page</td>
<td>128</td>
</tr>
<tr>
<td>Block - Acts as a container for child InputTag, OutputTag, TransitionTag and Block tags</td>
<td>128</td>
</tr>
<tr>
<td>config - Defines a scripting variable based on the specified values</td>
<td>129</td>
</tr>
<tr>
<td>ContentTag - Displays an Expresso Output element's Contents field</td>
<td>129</td>
</tr>
<tr>
<td>Context - Inserts the web-app context</td>
<td>129</td>
</tr>
<tr>
<td>ElementCollection - Retrieves a collection of a controller elements that can later be iterated through using the ElementIterator tag</td>
<td>130</td>
</tr>
<tr>
<td>ElementIterator - iterates through each controller element inside a parent ElementCollection tag</td>
<td>130</td>
</tr>
<tr>
<td>ErrorMessages - Displays an Expresso Error Collection from the Message Bundle</td>
<td>131</td>
</tr>
<tr>
<td>ErrorTag - Displays an Expresso Error Collection</td>
<td>131</td>
</tr>
<tr>
<td>IfElementExists - Checks if a particular controller element exists.</td>
<td>132</td>
</tr>
<tr>
<td>IfErrorExists - Checks if the controller generated errors and stored it in the page context.</td>
<td>132</td>
</tr>
<tr>
<td>IfLoggedIn - Checks if a session is established and a valid user is logged in.</td>
<td>132</td>
</tr>
<tr>
<td>IfMemberOfGroup - Checks if the user logged in has access to the specified group</td>
<td>132</td>
</tr>
<tr>
<td>InputTag - Displays an Expresso Input Object</td>
<td>133</td>
</tr>
<tr>
<td>LabelTag - Prints the label of a controller element</td>
<td>133</td>
</tr>
<tr>
<td>Login - Tries to make the user log in</td>
<td>133</td>
</tr>
<tr>
<td>output - Defines a scripting variable based on the specified values</td>
<td>134</td>
</tr>
<tr>
<td>OutputTag - Displays an Expresso Output Element</td>
<td>134</td>
</tr>
<tr>
<td>RestrictAccess - Allows quick and dirty restriction of access to jsp pages</td>
<td>134</td>
</tr>
<tr>
<td>session - Defines a scripting variable based on the specified values</td>
<td>135</td>
</tr>
<tr>
<td>setup - Defines a scripting variable based on the specified values</td>
<td>135</td>
</tr>
<tr>
<td>TableHead - Displays a pipe-delimited string as a table header</td>
<td>136</td>
</tr>
<tr>
<td>TransitionTag - Displays an Expresso Transition Button</td>
<td>136</td>
</tr>
<tr>
<td>TransitionParamsTag - Displays the parameters with an Expresso Transition</td>
<td>137</td>
</tr>
<tr>
<td>UserName - Displays the logged in user-name</td>
<td>137</td>
</tr>
<tr>
<td>Conclusion</td>
<td>137</td>
</tr>
<tr>
<td>Contributors</td>
<td>137</td>
</tr>
<tr>
<td>13. Expresso Utilities</td>
<td>139</td>
</tr>
<tr>
<td>Utilities</td>
<td>139</td>
</tr>
<tr>
<td>Conclusion</td>
<td>139</td>
</tr>
<tr>
<td>Contributors</td>
<td>139</td>
</tr>
<tr>
<td>14. Expresso Security</td>
<td>141</td>
</tr>
<tr>
<td>Introduction</td>
<td>141</td>
</tr>
</tbody>
</table>
## Expresso Developer's Guide

### Conclusion

**Contributors**

### 19. Expresso Component Application Deployment

**Introduction**

**Before You Start**

**Installation Procedure**

**Register the Application’s Schema**

**Run DBCreate to create the Application’s Database Tables**

**Conclusion**

**Contributors**

### 20. Unit Testing

**Introduction**

**Client-side Tests**

**Client Side Testing**

**How to run this test case**

**Writing your test case guidelines**

**Schema Testing**

**Server Side Testing**

**A Quick Sample**

**Running The Test Case**

**Conclusion**

**Contributors**

### 21. Healthcheck Monitoring and Application Performance

**Introduction**

**Healthcheck Monitors Web Components**

**Healthcheck Error Reporting**

**Healthcheck Tests Performed**

**Optimizing Performance**

**Performance Hogs**

**Cache Tuning**

**Database Optimization**

**Indexing**

**Conclusion**

**Contributors**

### 22. Jobs Expresso

**Introduction**

**Job Queue**

**Job Handler**

**Job Order**

**Creating a job**

**Submitting A Job**

**Job Parameters**

**Defining the Job Parameters**

**Adding parameters to a job**

**Accessing the parameters within the Job itself**

**Job Handling and Management**

**Starting the Job Handler**

**Job Scheduling Features**

**Cron Jobs**

**Theory and Implementation**

**Some Cron parameter examples**

**Conclusion**

**Contributors**

### 23. Reporting System

**Introduction**

**The Reporting Interfaces**

**Running a Report**

**Prepare the Report**

---

### ix

---
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accessing the Report Controller</td>
<td>198</td>
</tr>
<tr>
<td>Conclusion</td>
<td>198</td>
</tr>
<tr>
<td>Contributors</td>
<td>198</td>
</tr>
<tr>
<td><strong>24. Expresso Workflow</strong></td>
<td>200</td>
</tr>
<tr>
<td>Theory</td>
<td>200</td>
</tr>
<tr>
<td>Implementation</td>
<td>201</td>
</tr>
<tr>
<td>Practice</td>
<td>202</td>
</tr>
<tr>
<td>- Defining the workflow in the Controller</td>
<td>203</td>
</tr>
<tr>
<td>- Defining the workflow at screen build time</td>
<td>203</td>
</tr>
<tr>
<td>- Defining workflow at runtime</td>
<td>203</td>
</tr>
<tr>
<td>- Workflow sample code</td>
<td>204</td>
</tr>
<tr>
<td>Conclusion</td>
<td>207</td>
</tr>
<tr>
<td>Contributors</td>
<td>207</td>
</tr>
<tr>
<td><strong>25. User Registration and Login</strong></td>
<td>209</td>
</tr>
<tr>
<td>Introduction</td>
<td>209</td>
</tr>
<tr>
<td>Express Registration</td>
<td>210</td>
</tr>
<tr>
<td>Email Validation</td>
<td>210</td>
</tr>
<tr>
<td>Registration Domains</td>
<td>210</td>
</tr>
<tr>
<td>Registration Object Map And Detailed/Extended Registration</td>
<td>211</td>
</tr>
<tr>
<td>- Default domain and registration object map</td>
<td>212</td>
</tr>
<tr>
<td>- Extended domain and registration object map</td>
<td>213</td>
</tr>
<tr>
<td>- Customization of registration</td>
<td>213</td>
</tr>
<tr>
<td>Conclusion</td>
<td>214</td>
</tr>
<tr>
<td>Contributors</td>
<td>214</td>
</tr>
<tr>
<td><strong>26. Logging</strong></td>
<td>216</td>
</tr>
<tr>
<td>Introduction</td>
<td>216</td>
</tr>
<tr>
<td>Apache Log4j Integration</td>
<td>216</td>
</tr>
<tr>
<td>Expresso Extensions</td>
<td>217</td>
</tr>
<tr>
<td>Package Organization</td>
<td>217</td>
</tr>
<tr>
<td>Configuration Mechanism</td>
<td>218</td>
</tr>
<tr>
<td>- Appender Types</td>
<td>218</td>
</tr>
<tr>
<td>- Available Appenders</td>
<td>218</td>
</tr>
<tr>
<td>- Categories</td>
<td>219</td>
</tr>
<tr>
<td>How to use Expresso Logging</td>
<td>221</td>
</tr>
<tr>
<td>How to Add logging to your own Expresso-based application</td>
<td>222</td>
</tr>
<tr>
<td>Production Capable High Speed Logging</td>
<td>222</td>
</tr>
<tr>
<td>- Appenders</td>
<td>222</td>
</tr>
<tr>
<td>- Categories</td>
<td>223</td>
</tr>
<tr>
<td>Conclusion</td>
<td>223</td>
</tr>
<tr>
<td>Contributors</td>
<td>223</td>
</tr>
<tr>
<td><strong>A. Contributing To Expresso Framework</strong></td>
<td>224</td>
</tr>
<tr>
<td>Editing The Expresso Developers Guide</td>
<td>224</td>
</tr>
<tr>
<td>DocBook, XML, and XSLT</td>
<td>224</td>
</tr>
<tr>
<td>Transforming XML to HTML</td>
<td>225</td>
</tr>
<tr>
<td>Generating A Unified Difference Patch</td>
<td>225</td>
</tr>
<tr>
<td>Working With Other Databases</td>
<td>227</td>
</tr>
<tr>
<td>- Quick Start</td>
<td>228</td>
</tr>
<tr>
<td>- Final Work</td>
<td>232</td>
</tr>
<tr>
<td>Conclusion</td>
<td>232</td>
</tr>
<tr>
<td>Contributors</td>
<td>232</td>
</tr>
<tr>
<td><strong>B. The Expresso Component Architecture [Future Direction]</strong></td>
<td>234</td>
</tr>
<tr>
<td>Introduction</td>
<td>234</td>
</tr>
<tr>
<td>Goals</td>
<td>234</td>
</tr>
<tr>
<td>Architectural Overview</td>
<td>234</td>
</tr>
<tr>
<td>- What makes a component then?</td>
<td>235</td>
</tr>
<tr>
<td>- Architectural Tradeoffs</td>
<td>235</td>
</tr>
<tr>
<td>- Component Containers and Services</td>
<td>236</td>
</tr>
<tr>
<td>Example</td>
<td>236</td>
</tr>
</tbody>
</table>
Component Lifecycle and Interfaces .............................................................. 237
ExpressoComponent Interface .................................................................. 238
ExpressoComponent Lifecycle ................................................................. 243
Startup Configuration ................................................................................. 244
Example expresso-services.xml ............................................................... 244
Component Creation .................................................................................. 246
Define the component's interface ............................................................. 246
Implement ExpressoComponent interface ............................................... 246
Write the component's metadata .............................................................. 246
Implement Lifecycle interface if you want to receive configuration and lifecycle events. ................................................................. 248
System initialization ................................................................................... 248
Required System Parameters .................................................................. 248
Logging: The odd guy out ........................................................................ 249
Enter the System Factory ......................................................................... 249
System Destruction .................................................................................... 249
To Do Items ............................................................................................. 250
Future Expansion possibilities ................................................................. 251
Conclusion .............................................................................................. 251
Conclusion .............................................................................................. 251
Contributors ......................................................................................... 252
Glossary ................................................................................................. 253
Index ...................................................................................................... 255
List of Tables

4.1. Pre-Defined Class Handlers ................................................................. 9
4.2. Setup Values .................................................................................... 15
4.3. Initializing the Database ................................................................. 19
4.4. Arguments to DBTool ............................................................... 22
8.1. Controller Objects Included with Expresso .................................. 85
12.1. Expresso Logic Tags ................................................................. 123
12.2. Expresso HTML Tags ............................................................... 124
12.3. Expresso Tags Described ......................................................... 125
12.4. Attribute Tag ............................................................................... 128
12.5. Back Attributes ........................................................................... 128
12.6. Block Attributes ........................................................................... 129
12.7. config Attributes .......................................................................... 129
12.8. Context Attributes ......................................................................... 130
12.9. DBMaint Attributes ....................................................................... 130
12.10. ElementCollection Attributes .................................................. 130
12.11. ErrorMessages Attributes ......................................................... 131
12.12. ErrorTag Attributes ...................................................................... 131
12.13. IfElementExists Attributes ........................................................ 132
12.14. IfLoggedIn Attributes ................................................................... 132
12.15. IfMemberOfGroup Attributes ...................................................... 132
12.16. InputTag Attributes ....................................................................... 133
12.17. LabelTag Attributes ...................................................................... 133
12.18. Login Attributes ........................................................................... 134
12.19. output Attributes ........................................................................ 134
12.20. OutputTag Attributes .................................................................... 134
12.21. RestrictAccess Attributes .......................................................... 135
12.22. session Attributes ......................................................................... 135
12.23. setup Attributes ........................................................................... 136
12.24. TableHead Attributes ............................................................... 136
12.25. TransitionTag Attributes ............................................................ 136
12.26. TransitionParamsTag Attributes ............................................... 137
14.1. Administrative Security ........................................................... 146
21.1. Healthcheck Functions ............................................................ 183
22.1. Job Queue Entry Parameters ..................................................... 193
22.2. Job Scheduling Parameters ......................................................... 194
26.1. Available Appenders ................................................................. 218
26.2. Categories .................................................................................. 219
A.1. JDBC Attributes ........................................................................ 230
A.2. DBCreate Failure Issues .......................................................... 231
Chapter 1. Introduction

Welcome

Welcome to the Expresso Developer's Guide! This guide provides the Java developer with the information needed to take full advantage of the Expresso Web Application Development Framework and Component Library.

It gives you in a (fairly) concise form the net results of many thousands of hours of programmer time spent working with Expresso and developing best practices for it's use and implementation.

This guide is the result of many contributions, ideas, input and work by the Expresso developer's community, and we welcome any input, suggestions, and comments on it. Our goal is to continue to improve it as Expresso itself improves, so that it can continue to serve as a valuable tool for the professional developer.

The guide can be read in sequential order, as it starts by providing an overview of the tools and capabilities embodied by Expresso, then goes on to provide the details of those tools, and how to put them quickly to use in developing your own applications.

This guide is intended to supplement the Expresso documentation, and is best used in conjunction with that documentation, particularly the JavaDoc for Expresso. The source code itself, of course, is the ultimate authority on what's in Expresso, and should be referred to as required.

Furthermore, Expresso is based on Struts. So all of the Struts documentation applies, and should be considered a prerequisite. For example, features like "tiles" (a means for sophisticated #includes and template within JSP views) and "style" (a means for redirecting output to an arbitrary JSP) are useful but documented elsewhere in Struts and associated pages.

Jcorporate's JGroup Services

Do you need knowledgeable Expresso developers to jumpstart your project or to write a software solution? JGroup, our commercial services, backs up the Expresso product with cost effective services for the corporate IT community including training, support, development and consulting services.

The developers on Expresso are the people that know the product best. We provide global services to companies around the globe. Our team is comprised of our most experienced developers on the Expresso development team. They are the best qualified people in the world to provide services on Expresso because they are the developers of Expresso. How many companies let you have their developers? So it's a very different kind of service, a very high end quality service. For more information or to request a quote, please contact sales@jcorporate.com or one of the JGroup Experts directly.

Note

Was this EDG documentation helpful? Do you wish to express your appreciation for the time expended over years developing the EDG doc? We now accept and appreciate monetary donations. Your support will keep the EDG doc alive and current. Please click the Donate button and enter ANY amount you think the EDG doc is worth. In appreciation of a $35+ donation, we'll give you a subscription service by emailing you notifications of doc updates; and donations $75+ will also receive an Expresso T-shirt. All online donation forms are SSL secured and payment can be made via Credit Card or your Paypal account. Thank you in advance.
Chapter 2. Expresso Overview

Note
If you find this EDG documentation helpful please consider DONATING! to keep the doc alive and current.

Version: Expresso 5.5
Maintainer: David Lloyd
mailto:dlloyd@jgroup.net?Subject=EDG

Expresso is an application framework with an emphasis on building robust, secure, feature-rich and reliable web applications quickly and easily.

Application Framework
To use Expresso's capabilities to their best advantage your application should be written using Expresso as it's application framework. This involves using Expresso's initialization mechanism, configuration and cache management, database objects and security systems, and it's MVC Controller and State objects with the presentation mechanism of your choice (such as JSP (with the full Struts tag library available), Webmacro, and several others).

Using Expresso is this way still leaves plenty of room for integration of other development toolkits, frameworks, and component libraries. Expresso is designed to provide extraordinary extensibility, and can integrate with many other tools easily.

While it is quite possible to use Expresso's components without using all of Expresso, the system also makes the entire construction of web applications easier when used as a whole. Expresso imposes a minimum of constraints on the applications built with it. It does not tie you in to any particular navigation method or style, or even any particular architecture. Web applications build using Expresso as their framework can use JSP's, ordinary Servlets, even Applets or stand-alone Applications as their User Interface.

In the chapters that follow, we will mostly be discussing Expresso's use as an application framework, but the individual sections that describe components like Database Objects, Controllers, Jobs, and so forth can easily be applied to their use as individual components as well.

Expresso is highly dependent on at least one relational database being available for it's operation, and is best suited to those applications in which a database is an essential part of the application.

Services
Expresso provides a number of services that assist in building web applications. We will explore these areas in the order that the developer of an Expresso application is likely to need them. Your experience of course may vary from this order.

One of the best detailed references for Expresso is the JavaDoc, which you can download along with Expresso's source code - the source code of course being the ultimate reference tool!

Conclusion
Contributors

The following persons have contributed their time to this chapter:

- Mike Rimov

Note

Was this EDG documentation helpful? Do you wish to express your appreciation for the time expended over years developing the EDG doc? We now accept and appreciate monetary donations. Your support will keep the EDG doc alive and current. Please click the Donate button and enter ANY amount you think the EDG doc is worth. In appreciation of a $35+ donation, we'll give you a subscription service by emailing you notifications of doc updates; and donations $75+ will also receive an Expresso T-shirt. All online donation forms are SSL secured and payment can be made via Credit Card or your Paypal account. Thank you in advance.

[http://www.jcorporate.com/edgdoc.html]

Copyright © 2001-2004 Jcorporate Ltd. All rights reserved.
Chapter 3. Developing an Expresso Application

Note

If you find this EDG documentation helpful please consider DONATING! to keep the doc alive and current.

Version: Expresso 5.5
Maintainer: David Lloyd
[mailto:dlloyd@jgroup.net?Subject=EDG]

This guide describes the process of developing an Expresso application.

Introduction

This guide describes the process of developing an Expresso application. Although there are many different ways to use Expresso's components, the steps described here should provide your best results.

Developing an Expresso Application consists of the following general steps:

Planning the Application

An Expresso application consists normally of a group of Database Objects (embodying the Model and persistence of the application). See the section on Database Objects for more information about Database Objects. Controller objects (providing the Controller aspects of the MVC architecture, and encapsulating all interaction with users), Job objects (handling all longer tasks and "background" jobs), and a single Schema object to tie them all together. At this point the application can be tested, using the default "view handlers".

Then the "view" of the application is created, using one of the several different UI technologies available - for example, JSP pages with Struts and the extended JSP tag library, or XML/XSL stylesheets.

Of course, the application can have many other objects that don't fall into these categories, but these are the main classes of objects that Expresso is concerned with and can help with. Please see the specific documentation in the EDG for each of these kinds of objects for where and how they can best be used.

Create the Packages

Typically an Expresso application resides in a single top-level package, with a number of sub-packages. For example, if you are creating a membership management application, you might call the package com.yourcompanyname.members. Within the base package are usually the following packages:

- dbobj: Contains the database objects for the application
- controller: Contains the Controller objects for the application
- job: Contains the job objects for the application, if any.
Of course the application may require other packages, but these are the basics.

**Create a Schema Object**

The Schema object for the application allows the entire application to be referenced by Expresso. This then allows Expresso to administer the application's security and other administration automatically when the Schema object is registered with Expresso (see Expresso's "Application" page for details on registering the Schema object with Expresso).

Usually you would create the Schema object in the "root" package for the application.

The easiest way to create your own Schema object is to copy an existing Schema class, such as the one included with eForum, which is in the com.jcorporate.eforum package, called "ForumSchema.java". You can then modify the following areas of the Schema to customize it for your application:

- **Name**: The name of the class should be changed of course, any valid class name is appropriate. For example, if your application handles a membership list, you might call your Schema object "MemberSchema".
- **Title**: The "getTitle()" method returns the title of this schema object - edit the String returned as appropriate.
- **Version and Required Versions**: Other methods in the Schema specify the version of your application, as well as the version of Expresso that is required. These methods can also be used to specify dependencies of your application on other applications - for example, if your application requires the eForum application. See the Javadoc for
- **Default Component Code**: The component code for an application defines the name of a sub-directory of the "components" directory, which is by default the location of any JSP and HTML pages associated with this application. In eForum, for example, the component code is "eforum", and the JSP's for eForum are in the expresso/components/eforum directory.
- **Message Bundle Path**: Each application may have it's own MessageBundle.properties file, which contains keys and associated message strings for each of the languages supported by the application.

**Create any required DB Objects**

Database objects are normally created in the "dbobj" sub-package of the application. An Expresso application can use as many DB objects as required, and each should be listed in the Schema object for the application, so that the DBTool and DBCreate utilities in Expresso can create the required tables automatically for you when the application is first set up on a particular database. Database objects are designed to be the persistence layer of Expresso, and should take full advantage of the facilities provided by Expresso for them, such as caching, valid values, declarative referential integrity, and built-in security. See the page in the EDG on database objects to understand how to use these objects to their best advantage.

**Create any required Controller Objects**

Controller objects should be used to encapsulate the logic of your application, especially any user-interaction required. The idea is to take advantage of things that are provided by the framework and not re-invent anything that is not required. You must design your user-interface in a presentation-independent way, thinking in terms of abstract inputs and outputs, to take best advantage of Controller objects. This allows you to take advantage of the GUI-independent user-interface abilities in Expresso.
modifying the "view" of your application even once the logic is completed.

The controller chapter is dedicated to explaining the details of Controller objects can help you create and test your Controllers easily.

Create any required Job Objects

Job objects can be created and used for any task that can run independent of the user - e.g. longer tasks that do not require user interaction, or tasks that require extensive processing of databases or other data sources and might take some time. Typically a Job object is designed to send some kind of notification of it's completion to the originating user - the methods built into the Job object make this easy to do.

The page giving details of the Job object can help you create and deploy your Jobs effectively.

Conclusion

Contributors

The following persons have contributed their time to this chapter:

- Mike Rimov
- Mike Traum (JGroup Expert)

Note

Was this EDG documentation helpful? Do you wish to express your appreciation for the time expended over years developing the EDG doc? We now accept and appreciate monetary donations. Your support will keep the EDG doc alive and current. Please click the Donate button and enter ANY amount you think the EDG doc is worth. In appreciation of a $35+ donation, we'll give you a subscription service by emailing you notifications of doc updates; and donations $75+ will also receive an Expresso T-shirt. All online donation forms are SSL secured and payment can be made via Credit Card or your Paypal account. Thank you in advance.

[http://www.jcorporate.com/edgdoc.html]
Chapter 4. Application Configuration and Initialization

Note

If you find this EDG documentation helpful please consider DONATING! to keep the doc alive and current.

Version:Expresso 5.5

Maintainer:David Lloyd
[mailto:dlloyd@jgroup.net?Subject=EDG]

Expresso contains a number of facilities designed to manage the setup and configuration data for an application, such as the information required to connect to the applications database (or databases if more than one), values required to connect to the email and other servers, and user-specific preferences.

Introduction

Expresso provides various degrees of control in both the framework and the applications, through both the schema objects and the configuration files written in XML. These configuration controls include:

- **Schema objects**: enables management of the security for all of the applications component objects
- **Configuration Manager**: used for configuration information used by all applications running in a particular context
- **Setup Values**: specific to a particular context, but are further specific to one application
- **User Preferences**: finer level of detail that are specific to a particular user.

Steps to setup an Expresso Application

There are 3 basic steps (and a few optional ones) in creating your own Expresso Application. They are:

1. Create your Schema object
2. Create your XML files in the config directory
3. Register your Schema and create/intialize the schema/database

**expresso-config.xml**

Expresso requires this configuration file in order to specify customizable settings controlling it's operation on your system. This file is called expresso-config.xml and it is found in the WEB-INF/config directory. This file contains the primary configuration information for Expresso. It is read during system startup, and it's information is available to any object in the application from the ConfigManager object.
The expresso-config.xml configuration file supplies all of the information for each of the contexts set up with Expresso on your system, and provides information for each of those contexts to connect to their appropriate database engine(s), LDAP servers, and other core information.

The expresso-config.xml that comes with Expresso is set up only for access to the built-in Hypersonic databases (the "default" and "test" contexts). In order to add access to your own databases you must edit this file and either alter one of the existing contexts or add an entirely new context. See the documentation onsite for example jdbc configuration entities for various database types under the Database-specific Installation Notes header.

The top-level element is the "expresso-config" object, which contains definitions that apply to the entire application. This top-level contains attributes and sub-elements that are applied to all contexts in Expresso. This includes properties like the log directory location where the application will write its log files, which are essential in tracking the application's operation.

Another important section of the expresso-config file is the "class-handlers" section. This section specifies the classes that are used for various implementation classes within Expresso and its applications. One of these classes, for example, is the "userInfo" implementation, which by default is supplied by the "com.jcorporate.expresso.services.dbobj.DefaultUserInfo" class.

Specifying an alternate implementation:

```xml
<class-handlers>
  <class-handler
    name="userInfo" classHandler="com.jcorporate.expresso.services.dbobj.DefaultUserInfo"/>
</class-handlers>
```

There are several classhandler already defined in Expresso. Here are a few along with their default values:

**Table 4.1. Pre-Defined Class Handlers**

<table>
<thead>
<tr>
<th>Class Handler Name</th>
<th>Default Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userInfo</td>
<td>com.jcorporate.expresso.services.dbobj.DefaultUserInfo</td>
<td>userInfo is the implementation of com.jcorporate.expresso.core.security.UserInfo. It is used for mapping login names to integer user id's and also is responsible for storing passwords and other information.</td>
</tr>
<tr>
<td>orderedCache</td>
<td>com.jcorporate.expresso.core.cache.OrderedCache</td>
<td>Ordered cache specifies a cached list where order must be preserved. It is usually backed by some sort of java.util.List implementation.</td>
</tr>
<tr>
<td>unOrderedCache</td>
<td>com.jcorporate.expresso.core.cache.UnOrderedCache</td>
<td>UnOrdered cache specifies a cached list where order must be preserved. It is usually backed by some sort of java.util.List implementation</td>
</tr>
<tr>
<td>registration</td>
<td>com.jcorporate.expresso.services.controller.SimpleRegistration</td>
<td>Registration is a controller that</td>
</tr>
</tbody>
</table>
## Class Handler Name

<table>
<thead>
<tr>
<th>Class Handler Name</th>
<th>Default Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>login</td>
<td>com.jcorporate.expresso.services.dboj.DefaultUserInfo</td>
<td>allows for an end user registration process</td>
</tr>
<tr>
<td>nextNumber</td>
<td>com.jcorporate.expresso.core.dboj.NextNumberImpl</td>
<td>login is a Controller that provides authentication capabilities for end users</td>
</tr>
<tr>
<td>cacheSynchronizer</td>
<td>N/A</td>
<td>cache Synchronizer is an object that communicates to remote machines that cache has been modified and remote caches should be cleared. Expresso itself does not contain a cacheSynchronizer component, but commercial add-ons are available through products such as Expresso Enterprise(tm)</td>
</tr>
</tbody>
</table>

Nested within the "expresso-config" element are one or more "context" elements, with various nested elements within them. Settings with the context element only affect that one context. Each "context" element defines a different context of operation - typically including a separate database. To recap, context elements contain settings that are specific to one individual context. Another example is the JDBC sub-element, which specifies the connection parameters for DBConnectionPool to connect to your JDBC data source.

Each context has a name, and a description - these are used when the context is displayed, for example, by the Login controller.

## Format

The format of the expresso-config.xml file is specified by a DTD (stored in WEB-INF/classes/com/jcorporate/expresso/core/expresso-config_5_5.dtd or a similar name for later versions), and the configuration file is validated against this DTD during startup of the system - any deviation from the expected format will cause a startup failure, and Expresso will likely not initialize. The expresso-config.xml file is organized into a hierarchy, as specified in the DTD file.

It is important to check the DTD for the expresso-config.xml configuration file, and to read the comments in that DTD to understand all of the options available. There are extensive comments in the DTD file that provide the explanation for each setup option - this file will always be your best reference as to alterations of configuration values, as it is the final word on what the valid format for expresso-config.xml is.

## Startup Problems

If you encounter startup problems with Expresso, the fault is quite possibly the expresso-config.xml file. In order to resolve the problems, examine the output in the expresso.log file in your specified log directory (by default WEB-INF/log). If there is no such file, then the log4j logging system is also not initializing - look for the standard error or standard output log for your servlet engine/application server. For Tomcat, for example, these log files would be in the logs subdirectory of a standard Tomcat installation.

This log files should explain why the loading of the configuration is failing: if it is a DTD validation ex-
ception, the element causing the problem should have been specified - correct your expresso-config.xml file and try again.

If the configuration file does not appear to be loading at all, make sure that the DefaultInit servlet is set to be run on system startup - this servlet is the one which actually calls the ConfigManager object, which performs the loading of the expresso-config.xml file itself.

**struts-config.xml**

This file provides the mappings required by the Struts frameworks from URL's to "Action" objects - in this case, to the Controller objects in Expresso (which are sub-classes of the Action object.) See the Struts documentation included with Expresso for details of the format of this file. Ordinarily, you will not need to adjust this file at all.

**Other application-specific xxx-config.xml files**

In addition to the normal struts-config.xml file, applications that have their own Controller objects can provide mappings for them in separate configuration files. For example, eForum has an eforum-config.xml file that provides mappings for it's Controllers. In this way, the core struts-config.xml file for Expresso does not need to be adjusted as you install or develop other applications, and the configurations for own applications can be easily adjusted.

**Logging Configuration**

**Setting up Explicit Log4j Initialization**

Since Expresso 5.1, Expresso uses a unified logging configuration file that is initialized and utilized quite differently from previous versions.

For servlet environments, it is a matter of setting the 'logDir' property in the Servlet context for your web.xml. Look for the configDir in existing installations and add the logDir parameter under that. This is the directory to expand expresso related macros in the log file. What the configuration system does with the logDir parameter is set the system property. 'expresso.logDir' for which log4j can expand upon. Example Snippet:

```xml
<?xml version="1.0"?>  <!-- DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN" "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
<web-app> <distributable/> <context-param>
<param-name>configDir</param-name>
<param-value>WEB-INF/config</param-value>
</context-param> <context-param>
<param-name>logDir</param-name>
<param-value>WEB-INF/log</param-value>
</context-param> ....
```

If you are using a non-servlet environment, you need to call new LogManager(String loggingConfig, String logDirectory) to initialize the log4j system with the same kind of parameters.

Step #2 is implementing the expressoLogging.xml file. For most of the public downloads, and example of this is already created. To do this, add the expresso.logDir macro to wherever you would normally put the logging directory name.

```xml
<appender name="expressoLog" class="org.apache.log4j.FileAppender"> <param name="File" value="${expresso.logDir}/expresso.log"/>
```
Fancy tidbits for initializing LogManager

There are several special instances for instantiating LogManager that may better fit your special application needs. Use them as you need or see fit.

- **System Property: expresso.logDir** Thanks to the log4j System, all log manager does is set the system property expresso.logDir. Depending on your server environment, it may be more practical to set this system property yourself. (Depending on your security settings)

- **System Property: log4j.configuration** Log4j can automatically find its own logging configuration file through this system property. You can use this directly too.

- Put **log4j.xml** in class path. If no configuration file is used, log4j automatically looks for log4j.xml in the classpath. This may be initialized as needed.

- Use **LogManager(URL logConfigFile, String logDirectory)** This is perfect if the logConfigFile is at a central configuration location or if you have it embedded in a jar file, for example. Note that when you use this method, Log4j does not watch for changes in the logging configuration file.

- **Use Other System Properties instead** Any system property can be used inside the log4j file by enclosing the system property with ${}. So, for example, under Tomcat installations you could write your log file to the catalina log directory by having:

  `<appender name="expressoLog" class="org.apache.log4j.FileAppender"> <param name="File" value="${catalina.home}/logs/expresso.log"/> <param name="Append" value="true"/> <layout class="org.apache.log4j.PatternLayout"> <param name="ConversionPattern" value="%d %-5p [%t] %C{2} (%l) - %m\n"/> </layout> </appender>`

  Notice the use of ${catalina.home} instead of ${expresso.logDir}.

Changing Logging Configuration During Runtime

As long as you don't use the last initialization method described above, log4j will watch the configuration file and automatically reload all configuration information once changed. This allows you to selectively turn on and off debugging information without restarting the server.

Schema List Table

The Schema list table by itself doesn't do much. This table gets written to after you register your schema's get registered.

Schema Object
The fundamental structure of an Expresso application is defined in the application's "Schema" object. This object extends the "Schema" class, and serves as a list of all of the other objects that make up a particular application. Expresso itself uses a Schema class (com.jcorporate.expresso.core.ExpressoSchema) to describe the classes "common" to all Expresso applications. A list of all known Schema objects for a particular installation is held in the SCHEMALIST table in the default database (which may maintained from the Setup page).

The Schema Object is method of control, allowing the application(s) to register the different parts of applications with Expresso. These parts could include database objects (dbobjects), controllers, jobs, and servlets. These different parts are integrated into Expresso to allow better management and services. Also, once integrated into the Expresso Framework, the schema objects allow the various databases in one's application to be populated (if necessary), various initial setups, and fine grained security to be applied without giving up flexibility over the rest of the framework. Every application (such as eContent, eForum, or your own application) has one "schema"ues are very flexible and can be adapted to a wide breath of situations and environments.

To create your own Schema simply extend the com.jcorporate.expresso.core.dbobj.Schema class. Then in the constructor be sure do the following:

- Add all of your controllers, dbobjects or jobs that are associated with that Schema.
- Modify/add any Setup table values
- Set the MessageBundle path (implement getMessageBundlePath())
- Define a version number with getVersion()}

For details on how to do that see the API or view ExpressoSchema.

Initialization

When the application server/servlet container that contains Expresso starts up, a number of initialization steps are triggered. These steps begin with the DefaultInit servlet and the ExpressoActionServlet servlet. These two servlets are specified to execute on system startup in the web.xml file for any Expresso application.

DefaultInit

The DefaultInit servlet begins by setting a system property to specify which XML parser is to be used during the remainder of the initialization process - this is set to the Xerces parser, and should not be changed.

DefaultInit's next step is to call the ConfigManager object to initialize itself. ConfigManager then deals with the remainder of Expresso's initialization process.

Configuration Manager

The Configuration Manager allows further configuration of the Expresso Framework as well as the applications that run under the framework. The Configuration Manager achieves this by a set of XML files under the config directory. In these configuration XML files, there are properties that applies to the entire Expresso Framework, such as the underlying database, and some that apply to certain applications. Using the Configuration Manager in conjunction with the set of both Expresso Framework XML and application XML files gives utmost flexibility in the day-to-day operations of the framework.

Every application has access to the Configuration Manager (ConfigManager), an object automatically
initialized when Expresso starts up. This object gives access to one or more sets of "properties" - typically used for configuration information used by all applications running in a particular context, such as the required information to connect to the database server. Multiple "contexts" are supported by the database manager to allow many different database contexts to be used at once, either by one or a number of applications.

A context is a separate section in expresso-web.xml, and is typically associated with a single database. A user can log in to a specific context. Most of the configuration properties held by ConfigManager are therefore specific to one context, although there are a few "system-wide" configuration values available as well.

ConfigManager's first step on initialization is to read the expresso-config.xml file, and to create one or more "contexts" of configuration values. The Configuration Manager works by reading in the XML files at Expresso's startup. ConfigManager is a singleton object - that is, there is a single instance of ConfigManager running in any given Expresso application's virtual machine, even if several different db contexts and applications are installed. On startup, the Configuration Manager (expresso/core/misc/ConfigManager.java) will open and read the express-config.xml as well as any the application xml files. Furthermore, the ConfigManager will create one or more "contexts" of the configuration values. A context is a separate section in expresso-web.xml, and is typically associated with a single database. The contexts are separate instances that a user can log into. Most of the configuration properties held by ConfigManager are therefore specific to one context, although there are a few "system-wide" configuration values available as well. Thus the ConfigManager holds both application specific and system-wide configuration values, even though the application specific values grossly outweighs the system-wide values.

Technically, ConfigManager uses the Struts Digester class to read it's XML configuration file, and any problems during this initialization process are logged to the standard output of the servlet container. The Digester class was designed by the Struts folks to map XML to java objects in a systematic fashion with predefined rules being applied along the way. Any problems during this initialization process are logged to the standard output of the servlet container - so if your system does not initialize properly, or you see ConfigurationException errors when you attempt to work with your application, examine the system output log. The Digester class provides logging through Log4J (another Apache package) to a predefined logging area (usually a file). This logging provides both record of transactions as well as a simpler debugging process. One of the advantages of Expresso, is Expresso is built upon other projects to provide a best of breed solution. For Tomcat, this is a file in the "logs" directory called "catalina.out". For other servlet containers the location and name will vary - you must check the documentation for your servlet container for details. Once the expresso-config.xml file is read, the Setup values are read from each database into their appropriate context, and the connection pool is initialized for each context. Another advantage to the ConfigManager is the flexiblity of the class. The ConfigManager supports "custom" configuration values, so you can use it's capabilities to read and maintain values that are specific to your particular application, eliminating the need to create a custom object to do this. This alleviates the need for the application developer to write any custom code to read and parse the application's values. Furthermore, once the application has been registered with the Expresso framework, the ConfigManager is at the application's disposable.

ConfigManager then becomes available to all applications running in that environment for them to request the value of any of it's configuration settings - see the JavaDoc documentation for ConfigManager and the various "Config" objects in the com.jcorporate.expresso.core.misc package for details on what configuration values are available.

**Setup Values**

's content in. The values are available to any part of the package at runtime with a simple method call to the Configuration Management object (ConfigManager), and are automatically maintained in cache or read again as required. Setup values are different from the values held by the ConfigManager in several ways. They are more application-specific - e.g. there can be a number of setup values that are only used by one particular application, and another set of simiar values that are used by another application,
whereas ConfigManager's values are available to all applications. Setup values can also be changed and re-read during execution, whereas ConfigManager's values are only read during system startup. Setup values are stored in the database for a particular context, and can be accessed via methods in the "Setup" object in the com.jcorporate.expresso.services.dbobj package. Once Expresso is set up, you can use the Database Maintenance Servlet to display and edit the list of configuration values. Setup Values can be accessed from the Expresso Framework admin page (running on one's machine of course). To view the Setup Values, click on the Setup in the left bar to get a list of all currently installed setup values. After the Setup Page finishes loading, click on Setup Values in the first table.

Some uses of Setup values are to hold the connection information for a SMTP server, so that various Expresso functions can send emails automatically, and to hold the preferences for the URL of the header of the standard frameset used by Expresso - allowing easy customization to your own frame header for your applications.

Setup values take affect as soon as they are saved.

**Table 4.2. Setup Values**

<table>
<thead>
<tr>
<th>Setup Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AdminEmail</td>
<td>Email address of Administrative / Support Contact. This value should be set to the email address of a user who will receive support queries for the web site being set up - it is used when building the &quot;verify&quot; email for email-based user authentication.</td>
</tr>
<tr>
<td>AdminName</td>
<td>Name of Administrative / Support Contact for Site webmaster. This is the name to use for the email address specified above.</td>
</tr>
<tr>
<td>BaseDir</td>
<td>Web Document Root Directory on Server. This should be set to the directory of the root context of your web server.</td>
</tr>
<tr>
<td>CompanyName</td>
<td>Name of Company - this value is also used for the &quot;verify&quot; email when email-based user authentication is utilized.</td>
</tr>
<tr>
<td>ConnTimeOut</td>
<td>Database Connection Timeout Interval seconds - this is the number of seconds that a database connection may be idle before it is considered &quot;stale&quot; and returned to the connection pool.</td>
</tr>
<tr>
<td>ContextPath</td>
<td>Context Path for Expresso - this is a very important setup value for all other Express components to operate correctly. By default Expresso is installed in the /components/expresso directory (relative to the root web directory), but may be any other value depending on your installation. For Servlet API 2.2 installations, this is the Context Path for the application context of Expresso.</td>
</tr>
<tr>
<td>DefaultGroup</td>
<td>Default User Group for Self-registered users - this value supplies the code of a security group. Users that use the self-register function of Expresso are automatically made members of this group.</td>
</tr>
<tr>
<td>EmailValidateURL</td>
<td>Full URL of Email Validation Servlet - this is the URL to use in the user-authentication emails that are sent if this function is enabled.</td>
</tr>
</tbody>
</table>
| HomePageURL    | Home Page URL - this is the URL included in the...
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTPServ</td>
<td>Web Server Host Name - this is the system host name of the server hosting this installation of Expresso.</td>
</tr>
<tr>
<td>LogLevel</td>
<td>Level of detail in Log 0=min, 9=max - you may want to initially specify a high level of detail (e.g. 9) and reduce the level once the system goes into production.</td>
</tr>
<tr>
<td>MAILFrom</td>
<td>Value for From field in Event E-Mails - this must be a valid email address on the specified email server for outgoing event emails to be operational.</td>
</tr>
<tr>
<td>MAILPassword</td>
<td>Password for sending e-mail if required - if your email server requires a username and password for outgoing emails the password must be specified here.</td>
</tr>
<tr>
<td>MAILServer</td>
<td>SMTP Server Name for sending E-Mails - the host name of the server hosting your EMail server. Expresso uses this server for sending all of it's email notifications and events.</td>
</tr>
<tr>
<td>MAILUserName</td>
<td>User Name for sending e-mail if required - if the email server requires a username and password this field specifies the username.</td>
</tr>
<tr>
<td>MaxConnections</td>
<td>Maximum Number of allowed Database Connections - the connection pool will throw an exception if more than this many connections are requested at any one moment. This is useful if there is an external limit imposed on how many connections to the database can be used - if there is no such limit, this value should be set fairly high (e.g. 20 to 30 is reasonable).</td>
</tr>
<tr>
<td>MaxJobs</td>
<td>Max Number of Server Jobs to run at once - this specifies the limit on the number of JobHandler requests that will be processed in parallel. Specifying 1 indicates a single-threaded job queue.</td>
</tr>
<tr>
<td>RequireEmailValidate</td>
<td>Require Email Validation for Account Activation - set to Y to enable to email user-authentication process.</td>
</tr>
<tr>
<td>RestartServer</td>
<td>Command Line to re-start Servlet Server - a command line on the server that will re-initialize the servlet engine, used by the HealthCheck process if the servlet engine fails.</td>
</tr>
<tr>
<td>ServletEvent</td>
<td>Servlet Exceptions Trigger SYSERROR Event? - set this field to &quot;Y&quot; if you wish to receive emails for any exceptions that users of Expresso or Expresso applications encounter. A SYSERROR event will be triggered, and the contents of the email will include a full stack trace and other details of the error. Setting of &quot;N&quot; will not email messages, and a setting of &quot;E&quot; will email all exceptions <em>except</em> SecurityExceptions.</td>
</tr>
<tr>
<td>ServletPath</td>
<td>Default path for Servlets - this is the prefix for servlets in Expresso, by default /servlet. For Servlet 2.2API installations (such as J2EE) this is the con-</td>
</tr>
</tbody>
</table>
User Preferences

At a finer level of detail even than Setup values, which are application-specific, Expresso also provides for the management of "User Preferences". These are settings that are specific to a particular user, and may be used for custom settings for that user - such as the presentation mode used to display forum listings, for example.

LogManager

The LogManager class is Expresso's interface to the Apache Log4j framework, and is used to set up logging for all applications in a particular environment. See the section regarding log4j configuration above for more information on LogManager.

JobHandler

A JobHandler thread can be configured to start automatically whenever a particular db/context is initialized. This thread runs in the same VM as the servlet engine, and handles jobs queued in that context automatically. If you configure the job handler to not be started automatically, you should run one or more JobHandler utilities as separate tasks (e.g. in their own VM, running them directly as standalone java applications), or jobs queued in that context will not be processed.

JobHandler's thread can be turned on at startup by having the following line in expresso-config.xml:

```xml
<startJobHandler>y</startJobHandler>
```

CacheManager

ConfigManager also initializes the CacheManager, which is responsible for keeping in-memory caches of database information and other data, making it available to applications for quick access, while at the same time managing the size of the cache so that it does not consume all available memory. The CacheManager also "listens" to update events for object so that the cached copy of data can be discarded if it becomes "stale", and new data read directly from the database (or other source).

DB Connection Pool Initialization

Each context also has a database connection pool initialized for it at startup time. No connections are established immediately, but the pool grows (to a maximum size specified in the Setup values) as objects request access to the database. The connection information for the pool is stored in the expresso-config.xml file.

Struts Initialization
Once the Expresso Configuration Manager has initialized, a second servlet is called during system startup to initialize the Struts framework, which Expresso uses for URL mapping and UI presentation via JSP pages.

The Struts initialization process reads, initially, the struts-config.xml file (also in the config directory) in order to determine how URL's are mapped to Controller classes, and various other configuration information for Struts. In addition, each XML file that implements the same DTD as struts-config.xml is also read - again meaning that each application can use a separate configuration file, eliminating the need to edit the supplied struts-config.xml file as you add your own application Controllers. For example, eForum uses eforum-config.xml, ePoll has epoll-config.xml, and so forth. All of these files that can be located are read during startup, and all of the appropriate path mappings are then made available to the user.

Standalone Application Initialization

It is also possible to initialize Expresso from a standalone java application - e.g. not running in a servlet environment. Examine the code in the "main" method of com.jcorporate.expresso.core.utility.JobHandler for an example of this. Struts is not available from standalone applications initialized in this way, but ConfigManager, CacheManager, DB connection pools and logging all are.

DBTool and DBcreate

DBTool is a standalone Java application which can be run independently of your servlet container. DBCreate/DBTool can help in two directions: from existing Java code, they can initialize database tables and perform other setup functions. Also, DBTool can be used to reverse-engineer java from existing DB tables. See the javadoc for DBTool, and the Ant task "generatefromdb" which calls DBTool and creates java (DBObject) files from the database specified in /config/expresso-config.xml.

To reverse engineer database tables, the first thing is to set up the connection to the database that you are trying to reverse engineer. Otherwise, there is no way for Expresso to find the tables that you want to reverse engineer. Reverse engineering presumes that the database driver has the ability to query the catalog for the database (most major databases do have this ability). So you have to figure out what to put in the file expresso-config.xml. There is likely a complete example and notes for whatever database you want to use at http://www.jcorporate.com/econtent/Content.do?state=template&template=2&resource=636&db=default

Alternatively, the expresso-config.xml that comes embedded within the download is for the database HSQL, a simple Java database. You could start with that, and just reverse engineer the database tables already created within Expresso, just to get a feeling for how to reverse engineer tables into Java classes.

Have you used Ant before? Ant is a nice cross-platform tool, and all of the jars are put in the classpath automatically when you use that tool as described below. You will have to install it separately (download from http://ant.apache.org/).

Now, here's a problem: some downloads, like Expresso-x.y-complete.zip does not include the file build.xml, and described below. "Complete" as part of the download name means that Tomcat has been integrated into the package, so nothing else is needed to run Expresso as a web server, but a "complete" download does not necessarily include source.

So if you download the corresponding package for source, expresso-x.y-src.zip, you will find build.xml there. OK, one more problem: this source package does not include libraries, so you have to copy from the "complete" download, webapps/ROOT/WEB-INF/lib/* into the corresponding directory of this source download. Clearly, these downloads are created with a single purpose, and their size prohibits adding anything that might be convenient but not absolutely necessary.

OK, if you got this far, you have a source distribution which also has libraries, and you have to make one more environmental setting: set an environmental variable for CATALINA_HOME. That is the Tomcat main directory. If you like, you can set it to the directory where you expanded the "complete" download, since Tomcat is installed into the top directory there. (The whole point is just to find ser-
vlet.jar to put into the classpath.)

From the top directory where this source download was expanded (e.g., "expressosrc"), and referring to
the directory where the "complete" download was expanded (e.g., "expressocomplete"), the commands
would be something like

```bash
# copy appropriate db config, expresso-config.xml, into directory # where src package has been
expanded cp downloads/expresso-config.xml expressosrc/expresso-web/WEB-INF/config/
# set env var for where to find
tomcat (used to find servlet.jar) export CATALINA_HOME=/home/me/expressocomplete
# go to topmost directory where you
unpacked src zip file, # where (topmost) build.xml is found cd expressosrc
# execute ant command; assumes ant is installed per instructions, # with
(ant-distrib)/bin in path ant generatefromdb # list java files created in
this directory ls *.java
```

In order to know if the Java files have been created, look in the directory. The Ant script will also give
you output that Java files are being created for various tables.

Whew, this could be a lot easier! :-) And after you get the Java files created, there's still plenty to do to
register them in a new schema in Expresso in order to use them. Despite all the work, it is still faster to
go this route when reverse engineering several tables. And of course, after you've done it once...

**DBCreate**

The DBCreate is a servlet that is built into the Expresso framework to allow easy addition of applica-
tions into the underlying database. To use the DBCreate function, first one must install the Expresso
framework and an application.

**Introduction**

DBCreate works in conjunction will DBTool. The DBCreate servlet is actually a wrapper to check that
the user can execute this function (usually the Admin user), search for the schemas (done in #8 of How-
To), and then calls various methods in DBTool. The code source of DBCreate can be found at
($Expresso Installation Directory) /src/com/jcorporate/expresso/services/servlet/DBCreate.java. Once
you have installed Expresso and have the servlet server configured, you can automatically create the
database tables that are required. You should manually create the actual database that is referred to in
your properties file (part of the dbURL is normally the database name). The name can be any valid name
for a database. Do not create any tables in the database - Expresso will do that for you. As soon as the
database exists, you can verify Expresso’s connection to it with the "Status" servlet, on the "Server"
page.

**Initializing the Database**

The DBCreate servlet is used to perform the following actions. You may select one or more functions
for this program to perform. The functions are listed below.

**Table 4.3. Initializing the Database**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create Tables</td>
<td>Creates the tables in the database for each database objects (dbobject) in the default schema and in any registered schemas. If the table already exists, no further action occurs.</td>
</tr>
</tbody>
</table>
### Initial Setup

When first setting up a server, perform the following steps:

1. Run the Initialize servlet, selecting all functions. This will create all of the required tables for the standard Jcorporate schema in your database.

2. Go to the Setup Functions page by clicking Setup on the Expresso Frameworks Table of Contents and select Database Schema Objects. Add a new Schema object definition for each of the applications you have installed, if any.

3. Come back to the Initialize servlet and run it again. Now all of the required items for each of the schemas will be created, and default security will be initialized. See the Help file for Initialize (the DBCreate servlet) for details.

Once DBCreate is complete, you should Log In as "Admin". There will initially be no password for this user, one should be set immediately using the "Change Password" servlet on the Login page. When you are finished, you can go back to the Expresso Setup page and continue with the Verify Configuration step.

### An Example of How-To Use DBCreate

It is suggested that for beginner's, the user installs the location example application (Download) as a demonstration. The following steps illustrate how to get the location demo.

1. Install the Expresso Framework
2. Download location-2.0.zip from the Expresso Examples project (to find click on Projects in the header of Jcorporate's site)
3. Unzip under the location-2.0.zip under the ROOT directory (see here, under Directory Structure Details for more info)
4. After installing the location application, start up Expresso if not done so already
5. Bring up Expresso's home page in a web browser

<table>
<thead>
<tr>
<th>Setup Default Security</th>
<th>Create initial entries for security, including an &quot;Admin&quot; user and user group, and permission for this group to access all administrative functions. Sets user security for a user named &quot;Admin&quot; to access all tables created.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create Default Config Values</td>
<td>Creates default entries in the SETUP database table for configuring your system. Create an entry for all required configuration values for all known schemas. If there are customized values already set up, this function will not overwrite them.</td>
</tr>
<tr>
<td>Perform additional Setup</td>
<td>Schema objects may optionally provide a method for additional setup to be performed. This method may create application component records, create default values in lookup tables and any other required initialize functions.</td>
</tr>
</tbody>
</table>
6. Click on the Applications (see below in the circle)
7. Find the add button for the Registered Schemes
8. Fill in the appropriate values and Click the form's button at the bottom
9. Click on Setup
10. Make sure that all the checkboxes are checked, and click run.
11. The user will see the text appear in the frame saying what the process is doing. The user will need to check the output for an errors.

**DBTool**

The DBTool program is used to create the required database tables for Expresso (and other application built with Expresso) and to set initial values for a number of important Setup values.

**Introduction**

DBTool is a standalone Java application which can be run independantly of your servlet container in order to initialise your database tables and perform other setup functions. It uses the same code as the DBCreate servlet and the two perform the same function.

- **Create tables**: One of the primary functions of DBTool is to create new tables in the selected database for all of the Expresso DBObjects. Once the Expresso tables are created, other applications can be "Registered" by adding entries to the SCHEMALIST table and then DBTool can be re-run and these applications' tables also created.

- **Default Security**: DBTool can also create default security settings for the applications of a particular database.

- **Populate Default Values**: DBTool allows you to step through the default setup values & change their value to one suitable to your system.

**Running DBTool**

The DBTool program can be run with the command line: java com.jcorporate.expresso.core.utility.DBTool configDir=<config> [arg=value] This assumes that you have Expresso on your CLASSPATH - the CLASSPATH for running DBTool should be exactly the same as the one your servlet engine is using. You can use the -classpath option on the "java" executable to set the classpath, or specify it from the environment. In this command line, you replace <config> with the directory that contains your default.properties and any other database properties. Be sure it is the directory, not the path to the default.properties file itself! Other arguments to DBTool are available to perform several other functions which are discussed later in this section.

**Initial Setup with DBTool**

To perform initial setup with DBTool, execute it with the following arguments: java com.jcorporate.expresso.core.utility.DBTool configDir=<config> setup=yes

You should first manually create the actual database that is referred to in your properties file (part of the dbURL is normally the database name). The name can be any valid name for a database. Do not create
any tables in the database - Expresso will do that for you. With the built-in Hypersonic database you
must only create the empty directory for the database to be stored in. DBTool will display a long se-
quence of output in this mode, you may wish to redirect standard output and standard error to a file for
later review. If you have problems during initialization, you can send this file to us to the listserv or for-
ums and we will try to help. This function will:

1. Create the Tables in the database for each DBOBJECT in Expresso
2. Create default entries in the SETUP database table for configuring your system
3. Create initial entries for security, including an "Admin" user and user group, and permission for
   this group to access all administrative functions.
4. Perform any additional setup and initial processing specified by custom applications.

This function can also be performed by the DBCREATE servlet, once Expresso is set up and operational.
For more information see:

- Help file (at http://yourservername:8080/expresso/help/DBCreate.html) for Initialize (the DBCREATE
  servlet) for details
- The DBCREATE section of the documentation

Review the output from the "setup=yes" step and correct any errors (for example, bad configuration file
information). You can safely re-run this step even if it completes partially - no information will be over-
written.

Initial Setup Values

Once you have created the database and the default setup values, you must specify appropriate Setup
values for your system and installation. DBTool allows you to step through the default setup values &
change their value to one suitable to your system. Particularly important are the ServletPath and Contex-
tPath for your system, but all values should be checked carefully. You can edit these setup values at any
time once Expresso is installed from the link on the setup page. Enter the config mode of DBTool by
running: java com.jcorporate.expresso.core.utility.DBTool configDir=<config> config=yes

DBTool will now prompt you for each essential setup value as discussed earlier in this chapter. DBTool
will prompt for each setup value, displaying it's current value and description. You can accept the de-
fault value by pressing RETURN or specify a new value, which will be saved to the database. Once
setup values have been set, it is best to restart your servlet server to make sure no caching is occurring,
then continue to verify your installation. Once initial setup is complete, you should log in as "Admin".
There will initially be no password for this user, one should be set immediately using the Change Pass-
word command on the Log in page. When you are finished, you can go back to the Expresso Setup page
(http://yourservername:8080/expresso/setup.jsp) and continue with the next step.

Arguments to DBTool

DBTool takes a number of optional command-line arguments that can perform many different functions.

Table 4.4. Arguments to DBTool

<table>
<thead>
<tr>
<th>Argument</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Argument</td>
<td>Value Specification</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>----------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>configDir</td>
<td>a directory</td>
<td>The directory argument indicates where the default.properties and other properties files can be found. If no dbname argument is specified, default.properties is used.</td>
</tr>
<tr>
<td>webAppDir</td>
<td>a directory</td>
<td>The directory argument to webAppDir indicates the location in which Expresso (or the Expresso-based application) is installed as a web-application. If you use the %web-app% macro in your properties file you must supply this value.</td>
</tr>
<tr>
<td>db</td>
<td>A database/context name</td>
<td>This optional property specifies a database/context name to use for subsequent processing. The db/context name is the same as the first part of the name of a .properties file in the directory indicated by configDir's value (e.g. dbname=oracle would be looking for oracle.properties in the configDir). If this value is not specified, default.properties is used for database connect information.</td>
</tr>
<tr>
<td>setup</td>
<td>yes (or any non-blank value)</td>
<td>Initial system setup mode. With this argument, DBTool will attempt to create the default tables and default setup values and security on the indicated database. Note that if this command is run after the tables are already there, it is safe - the existing tables will not be dropped. This can be particularly valuable for adding new tables as Expresso (or other applications) are updated.</td>
</tr>
<tr>
<td>config</td>
<td>yes (or any non-blank value)</td>
<td>Enter/edit Setup/configuration values</td>
</tr>
<tr>
<td>test</td>
<td>yes (or any non-blank value)</td>
<td>Invoke test mode. In test mode, DBTool will scan all registered applications (e.g. all entries in the SchemaList table), including Expresso, and perform a database &quot;verify&quot; on all database objects in those Schemas/Applications. This will report all and any referential integrity errors, and will display an exception if the definition of any database object does not match the actual database. This can be very helpful when checking for database changes in a new version of Expresso or other application.</td>
</tr>
</tbody>
</table>
### Conclusion

#### Contributors

The following persons have contributed their time to this chapter:

- Kris Thompson
- Michael Nash
- Larry Hamel (JGroup Expert)
- Mike Rimov
- Mike Traum (JGroup Expert)
- Sandra Cann
- Tino Dai

#### Note

Was this EDG documentation helpful? Do you wish to express your appreciation for the time expended over years developing the EDG doc? We now accept and appreciate monetary donations. Your support will keep the EDG doc alive and current. Please click the Donate button and enter ANY amount you think the EDG doc is worth. In appreciation of a $35+ donation, we'll give you a subscription service by emailing you notifications of doc updates; and donations $75+ will also receive an Expresso T-shirt. All online donation forms are SSL secured and payment can be made via Credit Card or your Paypal account. Thank you in advance.
Chapter 5. Caching

Note

If you find this EDG documentation helpful please consider DONATING! to keep the doc alive and current.

Version: Expresso 5.5
Maintainer: David Lloyd
[mailto:dlloyd@jgroup.net?Subject=EDG]

Expresso offers built-in caching ability, designed to be completely configurable for different deployment situations. Caching is handled via a CacheManager object, designed to maximize performance without causing out-of-memory problems by trying to hold too much in cache at once.

Introduction

The Expresso framework includes a caching system build into the framework. This caching layer can cache just about anything from dbobjects and configuration parameters to custom build caches. This caching layer provides fast lookups for frequent used data and provides an alternative for make frequent database calls. The actual class that controls the caching is called the CacheManger and call be found in the expresso tree under under expresso/core/cache/CacheManager.

The CacheManger works by providing a general-purpose class that handles both ordered caches and unordered caches. The ordered caches are made possible by storing the data as a vector of objects. The unordered caches are supported by placing the data into hashtables. Also, the CacheManger only works if the amount of free memory is above 10%. The free memory is the memory in the Java Virtual Machine, and can vary from moment to moment.

The CacheManager is fully independant of even the DBObject components, so you can use it to your heart's content independently of using Expresso. For example, there's an example in \texttt{com.jcorporate.expresso.core.dbobj.SecuredDBObject} of the security system creating it's own cache to significantly speed table lookups for security checking.

The 5.X releases includes work to the Cache system to make it more scaleable and robust.

Automatic Caching

Objects commonly accessed, such as configuration values and setup values and security data are automatically cached, but the cache is automatically updated when the corresponding item is changed in the database, so you never have to worry about "stale" data in the cache.

Database Object Caching

Database objects (usually representing data in a table in a database) can also be cached, and you can control the number of cache entries for each object, using more caching for items more frequently used by your application.

The cache manager also performs automatic memory management by keeping a monitor of the amount of memory available and used, and automatically clearing the least-frequently used caches when memory drops below a configurable threshold.
Custom Caching

Your application can use the CacheManager to cache any custom information required, allowing you to selectively improve performance without sacrificing the accuracy of data.

Clustered or Distributed Environments

Caching can even be used in a distributed environment, as the cache manager can be set up to allow synchronization with caches from other servers or contexts, guaranteeing the most current data to your users while still offering the performance benefits of the cache.

Conclusion

Contributors

The following persons have contributed their time to this chapter:

- Mike Nash
- Sandra Cann

Note

Was this EDG documentation helpful? Do you wish to express your appreciation for the time expended over years developing the EDG doc? We now accept and appreciate monetary donations. Your support will keep the EDG doc alive and current. Please click the Donate button and enter ANY amount you think the EDG doc is worth. In appreciation of a $35+ donation, we'll give you a subscription service by emailing you notifications of doc updates; and donations $75+ will also receive an Expresso T-shirt. All online donation forms are SSL secured and payment can be made via Credit Card or your Paypal account. Thank you in advance.

[http://www.jcorporate.com/edgdoc.html]

Copyright © 2001-2004 Jcorporate Ltd. All rights reserved.
Chapter 6. Database Objects

Note

If you find this EDG documentation helpful please consider DONATING! to keep the doc alive and current.

Version: Expresso 5.5
Maintainer: Malcolm Wise
mailto:Malcolm.Wise@eu.sony.com?Subject=ED

Expresso Database objects use database connections to provide a means of mapping from objects to relational database tables. Database objects are similar to (and can be deployed as - see the Expresso Enterprise Project) BMP Entity Enterprise JavaBeans. They are primarily intended to provide persistence for business objects, but frequently include business logic as well. Database objects provide methods for add, update, delete, and a number of different retrieval methods. Expresso implements CRUD a la in Grand's book "Java Enterprise Pattern" Wiley 2001. The searchAndRetrieve method integrates the search process to retrieve records with a means to access the retrieved records, and is frequently used to apply some processing to a set of records.

Introduction

Expresso is organized around a number of database objects. A database object is simply a java object whose state can be preserved by storing in a database. In this way, a database object is analogous to a table in a relational database. The difference is that in addition to attributes (or "fields" or "columns"), the database object also has methods. For example, a database object called "Report" might have an attribute called "ReportNumber", and a method called "publish". In this way the storage of data is grouped with the actions to be performed on this data.

Database objects provide a means to map relational database tables to Java objects, as well as a way to associate the actions of business objects with their data storage. Database objects are mapped to one or more database tables in the external database (for example, Oracle). These mappings define which columns in the table store which attributes in the object, what the primary key of the table(s) are, and other information required to create a database on the external database.

Note

For more research material on mapping objects to relational databases please see Scott Ambler's paper at IBM's site.

For example, an Invoice object can be defined as relating to a table "INVOICE" in the database, and the object then has an attribute "InvoiceNumber", "Customer", and all other columns in the table.

The database object, unlike the database table, then has associated actions as well - such as "AcceptPayment" or "Ship". These actions operate directly on the data stored by the object, and bind the actions and their data together.

A database object is similar to, (and soon to be compatible with) the Entity Enterprise JavaBean object. Expresso is intended to work both in the non-EJB environment as well as scale to EJB. Just as we have optional support for the 2.0 Servlet API in Expresso, we have support for both the non EJB and EJB environments. DBOObjects are more "lightweight" than Entity beans, allowing applications to scale in BOTH directions.
In Expresso there is the functionality to create a "database object" by reading the database & generating a java object that corresponds to a table. Then you can use the DBMaint servlet to automatically create an HTML form for add/update/delete/search capability on this new object (and the underlying table). This allows you to create basic database maintenance forms in a matter of minutes from an existing database.

A database object can also span multiple tables, so that it is a high-level object: for example an Invoice object might also contain data for the detail items on an invoice, which are stored in an "INVOICEDETAIL" table - rather than having many InvoiceDetail database objects, which might create undue performance load on the server. Instead, the Invoice object would contain methods for manipulating invoice details, such as "AddDetail". Database objects also make "virtual fields" easy to produce - such as a "Total" field, which is calculated on the fly, not stored in the database itself.

Database Objects Advantages

Database Objects provide a number of advantages, including:

- **Database Independence**: Database objects are built from the ground up to be database independent - they do not rely on the features of a specific underlying database to provide their functionality. This allows an application build using database objects to be ported from one database platform to another in literally moments, providing the potential for great scalability.

- **Automatic Connection Handling**:

- **Declarative Referential Integrity**: By embedding access logic directly in the database object, you can achieve the same advantages as using stored procedures for accessing a database without the platform-dependence. Referential integrity becomes database independent, and complex relationships between database objects become portable. For example, business rules can be integrated into the database object, so that all applications accessing the object are assured of following the established rules.

- **Multi-Level Validations**:

- **Avoiding embedded SQL**: Using database objects allows an application to avoid embedded SQL and it's maintenance difficulties and system dependencies. Applications simply interact with other Java objects, allowing the design of a system to be entirely object-oriented.

- **Security, and much more**: Every interaction with a database object can be secured, and the security data is easily maintainable via Expresso's built-in capabilities.

By using Database Objects in your application, you can take advantage of all of these capabilities without writing additional code.

Database objects provide a functionality that is similar to an extension to the Entity EJB specification, but do not require EJB's to be used, with the attendant complexities of an Application Server. Where EJB's are used, database objects can be implemented as Entity EJB's easily, allowing scaling both up AND down in size and complexity.

Scaling Your Applications

Expresso in a non-J2EE environment provides pre-built services and components for putting together database-based web applications. The way we approach the J2EE environment is that Expresso applications are able to scale, fairly "painlessly" from a non-J2EE environment to a J2EE environment. This actually involves very little redundancy overall, although there is certainly some overlap. In the areas of overlap, our goal is to not re-invent any wheels, but to provide app-server independant alternatives to
things that might bind an application to any single application server implementation.

For example: many application servers provide a connection pooling mechanism. So does Expresso. Expresso's connection pooling, however, is not tied to any specific implementation by a J2EE-server vendor, so your application is not tied either. It also provides a consistent set of features that your application can depend on being there that go beyond the typical services available in a J2EE compliant JDB-BC2.0 connection pool - for example, reconfigurable type mapping, error message translation, timeouts, tracing, multi-database capability and connection testing, to name a few.

In the area of database objects and entity beans, db objects give a much more "filled out" implementation of a database-aware object. They provide, for example, for db object-level referential integrity, are integrated with the automatic database maintenance servlets that are part of Expresso, can be automatically generated (bi-directionally), provide built-in methods for multi-valued fields, automatic change logging on critical objects, validation, and so forth. So, we think of entity EJB's as a mechanism for using DBObjects in a distributed environment, not as a "competitive" capability to dbobjects. DB Objects also do not have the inheritance problems that exist with Entity EJB's.

Security is another area: All J2EE servers provide some means of administering security, both for session and entity beans. Expresso's security is explicitly designed to be stored in a database, and can easily be extended to integrate with other models (e.g. LDAP integration is in the works already). It's tight integration with database objects, and it's existing thin-client administration capabilities mean again that an application can have superior functionality right away, and avoid any vendor-specific features. Expresso's security layer allows web-based administration of dynamic security that is in addition to & complementarty to EJB container's declarative (deployment-descriptor) based security.

The Transaction package (probably a bad name for it) is probably the least understood and quite possibly the most powerful area of Expresso. Again, we see session EJB's as a perfect mechanism to provide Transaction's functionality in a distributed way. The finite-state machine design of Transactions is an extension of the session bean's functionality, and is again tied into the security model. The new enhancements being done now to the transaction package provide a much easier means of tying your business logic to UI-independant interaction with the user than having to create custom JSP's or servlets to do the same thing. The way Transactions can be used with TrxActionServlet and TrxServlet to "automatically generate" user interfaces (including the new XSLT UI generation in the Expresso XML project) allows an application to be prototyped and tested without any though to the UI code at first, then customized at the UI level completely independantly of the business logic, promoting good application design (MVC architecture).

Logging: A singleton-based multithreaded logging service that is independant of any specific EJB implementation is provided by Expresso's logging classes.

Event Handling: The built-in integration of email event handling provides immediate administrative notifications to an Expresso application, whether in an EJB environment or not. This is again something that would have to be built without the framework.

Job Handling: Even in an EJB environment, Expresso's ability to "queue" jobs for handling on the server side is valuable for longer tasks than are suitable interatively. E.g. an accounting system might need to "post" thousands of entries in response to a user request. This is an ideal use of the Job object in Expresso, and applies equally well to EJB environments.

In summary J2EE provides a good environment to host applications, but it doesn't provide pre-built tools to snap business logic - Expresso helps to fill the gap between J2EE's capabilities and your finished application, while still allowing the ability to scale both up (EJB/J2EE) as well as DOWN (e.g. non-J2EE environments) with the same applications.

Feature Overview
Expresso Database Objects

Expresso Framework provides ancestor objects for Database objects (DBObject), which provide means for defining attributes (columns to be stored in the database), defining keys, specifying relationships to other objects (referential integrity) and defining default values and valid values for columns (e.g. multi-valued columns, such as Yes and No, etc).

Database objects also provide all basic operations, such as add, update, delete, search and retrieve. Methods are provided that allow objects to be searched for either by primary key or by any combination of field values (wild-card searches), and searches can be set up to return only a specific database object or a Vector of database objects. With a few lines of code you can create a database object that includes all of this functionality, allowing applications to be built extremely rapidly.

Object/Relational Mapping

Database objects map capability between Java objects and a relational database (RDBMS) in a standard an highly extensible way, so that the objects themselves can be used in your application, removing the need to embed SQL code directly into your Java applications.

Many enterprise applications require access to the data stored in multiple databases. The "Database Object Mapping" functionality in Expresso allows a developer to define a new DB context to describe a particular database, and then associate particular dbojjects with that DB context so that when that object is manipulated, it is always operating against the correct DB context.

Expresso applications can now be run with a "split context", with one database handling the control tables (such as USERLOGIN, etc.), and one database storing the application data (like a data repository, shared by a few different Expresso applications). This allows you to create different "looks" or "views" to the same data, with completely independent logins, security groups, etc. The bottom line is that now you can specify many different database contexts, and define DBObjects as belonging to one of these contexts at the schema level. After this step, the system will always look to the right database when you use that DBOject.

Automatic Generation

DBObject code can be generated from an existing database, or the database required for an applications DB objects can be automatically created, making working with DB Objects in either new or existing environments extremely straightforward.

DBMaint

A servlet is supplied with Expresso that provides immediate database-maintenance capability for any database object (in fact, for any database table through the use of AutoDBObjects). This facility uses the security defined for the object and automatically generates forms to add, update, delete and search for data in the underlying database tables.

Security

Security can be specified for any operation on any database object in a database-independant layer of Expresso, again making your security portable between database engines and immediately usable to any Expresso application, including the DBMaint servlet discussed above.

Hiding Changes

Changes made to a database table have no affect on code using the database object, as these changes only need be made to the database object itself. This decreases maintenance burden on code and en-
hances portability between different databases.

**Virtual Fields, Referential Integrity, Valid Values, Next Number**

Database objects allow fields to be defined that are calculated by any specified method (and not stored in the database). They allow referential integrity (the relationships between objects that must be maintained (e.g. invoice master/detail) to be defined in a database-independent way, they allow a set of valid values to be defined for a field and verify field validation against that list automatically, they allow serial number fields to be automatically assigned correct values, and much more. Please see the detailed documentation in the Expresso developer’s guide for details.

**MultiDBObject**

A special extension of the DBObj ect, MultiDBObject, is provided which can also span multiple tables. As it is often necessary to deal with Join relationships between tables in relational databases - the MultiDBObject exists to handle this. If a DBObj ect is analogous to a table, a MultiDBObject is analogous to a view (of joined tables). For example an Invoice object might also contain data for the detail items on an invoice, which are stored in an "INVOICEDETAIL" table - rather than having many InvoiceDetail database objects, which might create undue performance load on the server. Instead, the Invoice object would contain methods for manipulating invoice details, such as "AddDetail". Database objects also make "virtual fields" easy to produce - such as a "Total" field, which is calculated on the fly, not stored in the database itself. Many of the same operations available to DBObj ects are available to MultiDBObject - including searchAndRetrieve(), clear(), setField (although with different arguments), etc.

**Schema Objects**

Collections of database objects can be gathered together into a "Schema" - for example, all of the database objects required for the Invoicing application could be gathered into a Schema. This schema can then be used to automatically generate the database tables required by each of the database objects (as well as setting up required configuration values and reports). This makes installing the application on a new server a matter of minutes, and eliminates the need to tediously create the database tables individually.

**AutoDBObject**

The AutoDBObject is the easiest way to get access to your database tables and can be very valuable for prototyping your application. AutoDBObject can populate its fields automatically from the schema information of its target table. This allows an AutoDBObject to be instantiated and used to access a table without any coding at all!

**Server-Side Objects**

The Expresso Framework also contains server-side objects without data, similar to (and soon to be compatible with) Session Enterprise JavaBeans.

**Database Connections**

Establishing and cleaning up connections to the database can be very time-consuming to an application, slowing it’s performance. Instead of creating new connections as needed, it is better to use an existing pool of connections that are held open and available at all times.

Expresso provides facilities for managing database connection pools, abstracting the connection process.
even further than the JDBC API. A sophisticated connection pool object provides access to one or more databases in an efficient manner, providing the following facilities:

- **Multiple Connection Pools/Contexts**

  Expresso’s connection pooling capability provides access to many different connection pools, each potentially working against a separate database, even from another vendor. For example, your core Expresso tables might be stored in Oracle, but you could have an alternate connection pool to a DB2/400 database on a different server.

- **Maximum Size**

  A setup value can specify the maximum size for a connection pool. When the pool reaches this size, the connection pooling code tries to “clean up” stale connections and re-use them if possible to meet requests for new connections.

- **Timeouts**

  The connection pool is protected against inadvertent errors where the connections are not released by means of a timeout mechanism. This timeout returns a connection to the pool after a certain interval, in case the client program neglected to release the connection normally. This timeout can be overridden for long-running requests.

- **Verified Connections**

  The connection pool can optionally verify each connection before it is supplied to the client program by running a small query against the connection, thereby handling situations where the database may have closed the connection from its end and avoiding the client program being handed a closed connection. This mechanism uses both a test query and the `isClosed()` method, as `isClosed()` can give false indications in some situations. This results in an overall increase in resilience and reliability of your applications written with Expresso.

- **Automatic Cleanup**

  The connection pool will automatically clean up idle connections after a certain amount of time, reducing the size of the pool to a specified minimum during periods of inactivity.

- **Database-Specific Configurations**

  The connection pool allows database-specific options to be specified as setup options in the `expresso-config.xml` file. See the `expresso-config.xml` file documentation for details.

### Using JNDI to retrieve your connections

`DBConnectionPool` can also grab JNDI connections for use in enterprise systems. The initial setup of JNDI DataSources is dependent upon your site admin and app server. We will only specify how to actually get at that JNDI DataSource for use in the `ConnectionPool`.

To use JNDI Datasources, you will need to modify the `expresso-config.xml` to something along the lines of the following example that uses a Firebird driver from a JNDI DataSource:

```xml
<jdbc
driverType=dataSource
driver="org.firebirdsql.jdbc.FBDriver"
url="jdbc:firebirdsql:localhost/3050:c:/expresso/db/expresso.gdb"
connectFormat="1"
/>```
There are two keys here:

1. **driverType=datasource**: This tells the database connection pool that the connections will be through a JNDI Datasource rather than through the normal JDBC Driver manager

2. **<jndi/>** This tells the system to use JNDI for lookup with blank properties.

For many items you will want other attributes inside the jndi tag. Most of these are optional. Check with your appserver administrator for possible values for these attributes:

- initialContextFactory - Initial Context Factory ClassName
- providerURL - Datasource provider URL
- securityPrincipal - The principal user for security purposes
- securityCredentials - Security credentials for the client to server connection
- objectFactories - ?
- stateFactories - ?
- urlPKGPrefixes -
- dnsURL -
- authoritative - (true/false)
- batchSize - The batch size for number of records to retrieve in one roundtrip to the server
- securityProtocol - Security protocol for authenticating against the data source
- securityAuthentication
- language

## Compatibility with Other Applications

Sometimes you will need to integrate with other libraries such as the JSTL SQL tags, Reporting libraries, and others. Many of these libraries need an implementation of javax.sql.DataSource to use for grabbing new database connections. Expresso 5.3 now has code to allow you to retrieve a "dumb" datasource from the DBConnectionPool. Example code to do this is:

```java
javax.sql.DataSource dataSource = DBConnectionPool.getDataSource("default");
java.sql.Connection connection = dataSource.getConnection();
//... Use the connection here
```
Database Objects

Database objects use the database connections described above to provide a means of mapping from objects to relational database tables. Database objects are similar to (and can be deployed as - see the Espresso Enterprise Project) BMP Entity Enterprise JavaBeans. They are primarily intended to provide persistence for business objects, but frequently include business logic as well. Database objects provide methods for add, update, delete, and a number of different retrieval methods. The searchAndRetrieve method integrates the search process to retrieve records with a means to access the retrieved records, and is frequently used to apply some processing to a set of records.

Why Use Database Objects?

Database Objects provide a number of advantages, including database independence, automatic connection handling, declarative referential integrity, multi-level validations, security, and much more.

By using Database Objects in your application, you can take advantage of all of these capabilities without writing additional code.

Using database objects allows an application to avoid embedded SQL and it's maintenance difficulties and system dependencies. Applications simply interact with other Java objects, allowing the design of a system to be entirely object-oriented.

Database objects are built from the ground up to be database independent - they do not rely on the features of a specific underlying database to provide their functionality. This allows an application build using database objects to be ported from one database platform to another in literally moments, providing the potential for great scalability.

By embedding access logic directly in the database object, you can achieve the same advantages as using stored procedures for accessing a database without the platform-dependence. Referential integrity becomes database independent, and complex relationships between database objects become portable. For example, business rules can be integrated into the database object, so that all applications accessing

connection.close();
the object are assured of following the established rules.

Database objects provide a functionality that is similar to an extension to the Entity EJB specification, but do not require EJB’s to be used, with the attendant complexities of an Application Server.

Since Entity Beans can easily be molded to access the legacy data of DBObjets, it is fairly simple to migrate to Entity Beans from DBObjets if it is needed for particular tables. It should be noted, that while Entity Beans may shine under high traffic sites where transactional integrity is needed, DBObjets can be significantly faster than Entity beans since they don’t try to add all the complexities of data hiding from the client programmer as Entity Beans use.

### An Introduction To The Expresso DataObject API

Astute observers of Expresso code will have noticed the emergence of a new package in the expresso code base at: `com.jcorporate.expresso.core.dataobjects`. This is the location of the new Expresso DataObject API that will be used to refactor DBObjets into a more modular scalable entity. Its goal is to provide a unified interface to multiple data sources such as JMS providers, Databases, and even comma delimited files. The Expresso DBObject implements the DataObject interfaces. So if you learn the ways to manipulate a DataObject, you will automatically learn the ways to operate on a DBObject.

**Note**

The DataObject API takes the minimalistic approach of providing a minimum set of methods that provide, in essence, maximum flexibility, while still providing the same base services as DBObjets. While convenience methods such as `addIfNecessary()` do not exist in the API, the ready availability of the DataObject interfaces make it simple to add such methods in utility classes that can function with all dataobjects, and not just JDBC specifics.

### DataObject Core Interfaces

The following picture shows the core DataObject interfaces and their relationships to one another. The green objects are the primary interfaces, while the brown objects are what’s known as "metadata" interfaces. In other words, they provide descriptive information about the green objects, such as what data type they are, how many fields, etc.

Now let’s cover the responsibilities of each of the objects.

- **DataObject** This is the primary interface that you will be using. Every time you create a DBObject you’re in essence creating a DataObject. It provides methods to get and set field values, and methods to provide CRUD capabilities: Create, Read, Update, Delete. The Expresso specific terms are: `add()`, `find()`, `update()`, `delete()`.

- **DataField** The DataField's job is to provide a placeholder for field values in the DataObject. It also
provides methods to determine if the field value is null, if it has changed since it was created, if it has, what was the last value it had, etc. It also provides type conversions (if possible) from things like Strings to and from BigDecimal objects, or Boolean capabilities etc.

- **DataObjectMetaData** Each unique DataObject 'type' has specific data about it. For example, how many fields does it have? Which fields are the key fields? Are there any generic attributes for the DataObject? What about Master-Detail Relations? DataObjectMetadata provides methods to get at such information. It allows certain programs to generically operate on DataObjects without necessarily knowing the specifics about the object. The prime example of this is the Controller DBMaint. It provides a basic administrative interface for all DataObject implementations in the classpath of Expresso.

  You can get to the object's metadata by calling DataObject.getMetaData()

- **DataFieldMetaData** This class is similar to DataObjectMetadata in that it provides descriptive information, but in this case, it is specific to individual fields. For example, it contains information about the field's data type, field size, field name, etc. A concrete implementation of this interface is com.jcorporate.expresso.core.dbobj.DBField

  You can get to the DataFieldMetaData object by calling: DataObject.getFieldMetaData("fieldname")

### A quick note about metadata to data relationships.

One distinction about DataObjects and their corresponding metadata objects, is the number of instances created at runtime. There is only one metadata instance created for each unique 'type' of data object. For example, if you instantiate 1000 `com.jcorporate.expresso.services.dbobj.MimeTypes` objects, there will only be one corresponding Metadata object created. This is an optimization to significantly save runtime memory and greatly increase the application speed.

### Showing the bigger picture.

The following diagram shows how the DataObject API relates to several other objects. When you first see this diagram, DON'T PANIC! It is easy to have your eyes become like that of a deer in the headlights at night, but don't worry we'll go over the items one by one.
Ok, so now that you haven't listened and let your eyes glaze over anyway, let's try to tackle this diagram in small pieces.

First, start up with what you know. You've already seen DataObject, DataField, DataFieldMetaData, and DataObjectMetaData. They're the base. Now let's first focus on the other interfaces we've introduced in the diagram. They are the boxes that are green.

- **ContextNested** Expresso can work with multiple, independent databases at once. You will come across code like: `DBObject.setDataContext("default")` often. We call these independent databases *Database Contexts*. The ContextNested interface simply provides `getDataContext/setDataContext` methods. This is necessary since objects other than DataObjects may need to be sensitive to the database being currently used. If an object implements ContextNested, it means it should have `setDataContext()` called before it is ever used. If you don't know what this all means yet, don't worry. Multiple database are a fairly advanced topic, yet simple once you get a hold of it.

- **Securable** Expresso uses an integer number to represent a user. This allows for quick internal workings of Expresso and quick lookups in the databases. Any object that wants to act with the client's credentials should have the user's *uid* set. Securable simply has `setRequestingUid()`, `getRequestingUid()`, and a single access check method called `isAllowed()`. Any object implementing Securable expects to have `isAllowed()` called before any operation is done on it. If you ever need to bypass security and act as the 'superuser', then call `setRequestingUid(Securable.SYSTEM_ACCOUNT)` to set the system account for the Securable object.
Defineable

Most DBObjects in Expresso are simply differentiated between each other by their class name. But there are some data object implementations that must be defined more uniquely than just class name. A prime example of this is AutoDBObject. Every instance of AutoDBObject might be unique because the real issue is what table AutoDBObject points to. This is where Defineable comes in. Defineable simply has the methods getDefinition/setDefinition. The meaning of the definition may be interpreted differently by each object, so you need to consult each Defineable object with how to pass a definition to it. For example, AutoDBObject is defined by telling it what database table name to use. JoinedDataObject requires the path to the XML definition file.

If you understand these interfaces in the least bit, you can now understand the rest of the diagram. Let's cover it starting with the classes that are metadata classes.

Metadata:

- **DBObjectDef** Every interface must somewhere have a corresponding concrete implementation. DBObjectDef is a concrete implementation of DataObjectMetaData that is used by DBObjects.
- **DBField** Just like DBObjectDef, DBField is a concrete object that implements the DataFieldMetaData interface. If you call DBObject.getFieldMetaData("blah"), while the type returned is DataFieldMetaData, it is, underneath the interface, a DBField instance.

Now the rest of the objects are various combination of DataObject Implementations:

- **BaseDataObject** This is a blank base object that is designed to eventually provide any features common to all DataObjects. Right now it is basically blank.
- **JDBCDataObject** JDBCDataObject is a subclass of BaseDataObject that also contains methods that are very specific to JDBC type environments. Examples are getTargetTable(). This would never make sense in a comma-delimited file environment. So it is included here further down the object chain.
- **DBObject** As a subclass of JDBCDataObject, you can guess that DBObject is designed to be specific to JDBC environments. It is probably the oldest class in the Expresso framework, and can also be the most valuable. For the near-term, almost all database access classes in Expresso are derived from DBObject or SecuredDBObject in some way.
- **DefaultDataField** As mentioned a few times... *every interface must have a concrete implementation*. DefaultDataField is the DataField interface implementation that DBObject uses.
- **SecuredDBObject** Since this class derives from DBObject and implements the Securable interface, you can probably guess that this class has all the features that DBObject does, but is security conscious. Almost all DBObjects in a web environment should use a security system of access. Expresso provides the framework for that access through SecuredDBObject. SecuredDBObject checks if a particular user has Add,Update,Search, and Delete access through a role-based security matrix. If you derive your class from SecuredDBObject, you automatically get this security without one extra line of code.
- **AutoDBObject** Here we pull a bit of a twist. If you notice AutoDBObject implements the Defineable interface. This means that it is configurable in other ways than through class instances. In fact, an autoDBObject is meant to create instant access to legacy tables by reading a table's catalog and configuring itself to read the table dynamically. To use it, you instantiate the AutoDBObject and call setDefinition("TABLENAME"); and use the AutoDBObject normally from there.
UML Conclusion

So that is the basic philosophy of the DataObject API and how the different objects relate to one another. The rest of the chapter will deal with concrete examples of constructing and using DBObjets, which as you can see above, simply implement the DataObject interface and add some special functions such as setupFields(), and other items.

Creating Database Objects

This section describes all of the different options available when creating your database objects - you may not use many of them on any one database object, and the process of creating the simplest type of Database Object is described in the chapter about the "DBMaint" controller.

The most direct use of DB Objects is to write a class that extends the com.javacorporate.dbobjDBObject class. Your class then needs to implement a few simple methods to describe its relationship to a database table. A shortcut here is to use the DBTool utility to generate the code for you by reverse-engineering the database. See the documentation for DBTool for details on this. There is also an Ant target, "generatefromdb" that can be used to launch the DBTool reverse-engineering routine. It depends on the DB context specified in the /config/expresso-config.xml file.

A DBObjet must implement at least the constructors for DBObjet and a setupFields method. In the setupFields method, a number of method calls are used to establish the relationship of this DBObjet to the database and to a specific table or tables. At a minimum the setupFields object must call the setTargetTable method to specify a table in the database, and one or more addField calls to specify the fields in the table. For example, lets say we are setting up a customer information DBObjet. Lets say that the table in the database should have a unique Customer Identifier, a name and a customer type. You might name the DBObjet "Customer" and specify a setupFields method like this:

```java
public void setupFields() throws DBException {
    setTargetTable("Customer");
    addField("CustomerID", "auto-inc", 0, false, "Customer Identifier");
    addField("CustomerName", "varchar", 80, false, "Customer Name");
    addField("CustomerType", "char", 2, false, "Customer Type");
    addKey("CustomerID");
}
```

Let's examine this method, line-by-line:

*Line 1* establishes the method signature which must be correct for your DBObjet to compile.

*Line 2* specifies the "Target Table" or the table in the database that will (primarily) be associated with this database object. Note that no database name is prepended to the table name - this allows the same DBObjet to be used against any appropriate database. The content of the Expresso application will specify the actual database connected to at run-time.

*Line 3* specifies the first of the fields, or columns, of this database object and of the corresponding table.

Although it is possible for the database table to have columns that are not specified in the DBObjet, this is not recommended.

In the call to addField, we specify the following parameters:

- **Column Name**: this is the name by which this column is referred to within both the database object and the database.

  This name must match the name of the column in the database, but keep in mind that the DBObjet can be used to create the table automatically which we will explore later. We recommend the naming
convention shown here, with a leading capital letter and a capital at the beginning of each subsequent word, with no embedded spaces or underscores. Again, no prefix is specified.

Be careful when naming your columns not to use reserved words from any database system, as even though a particular word might not be reserved in your database system of choice, it is prudent not to restrict the portability of your applications. The name used must be unique within the DBOject but need not be unique across all tables in the database. In other words, you can have only one "CustomerID" column in this DBOject but other DBOjects can use "CustomerID" as well.

- **Data Type:** the string used here should specify the Expresso data type used for this field.

  This may or may not be the same data type that is used by your database - the Expresso data types are "mapped" at runtime to the appropriate type for a particular database engine. You can even define new Expresso types and map them as required to take advantage of specified database capabilities (although this again can limit portability and is not recommended for this reason).

  This particular field uses a special Expresso-only dataype called "auto-inc" that provides for unique integer numbers. The underlying column in the database table will be of type integer, but Expresso will automatically provide incremented values every time a record is added to the table.

- **Field Size:** certain data types require a field size or maximum length to be defined. Others, such as "int" typically do not. If a field size is needed, it is specified in this parameter - otherwise this parameter is left 0. An additional addField method exists to specify field sizes for fields with two size specifications; such as "float" fields, where the first size is the length of the field and the second is the number of decimal places.

- **Allow Nulls:** this boolean parameter indicates if nulls can be stored in this field: false indicates that no nulls are accepted, true indicates that null is a permitted field value.

- **Field description:** this is a short human-readable term or phrase that describes the field, often used as a column header or listings and reports. For best local language support, this description should be specified as a call to the getString method of the application's schema object which can select the appropriate name in the correct language at runtime. See Documentation on Internationalization for more details.

*Lines 4 and 5* in our listing specify additional fields in our table - although the order in which fields are added is not relevant, it does affect the sequence in which the columns are listed when a table is created from the DBOject or when using the Expresso components to list records from this table, so some logical order should be followed.

*Line 6* specifies the primary key field for this DBOject. You must specify at least one primary key field and you can specify more than one by repeated calls to addKey with each field name that makes up the primary key. Primary field keys can not allow nulls.

### Specifying Databases

So far, all of the examples we have seen of DBOjects assume they are accessing the default database/context as defined when Expresso is deployed. DBOjects can access any available Database/Context though, by using the setDataContext(String) method. This method takes as its argument the code of a database/context, which is the same as the name of the .property file for that context in Expresso. For example:

```java
custList.setDataContext("oracle");
```

says that the custList DBOject should use the context "oracle", which presumably has been defined to
connect to an Oracle DBMS.

Most other Expresso objects allow the current Database/Context of the user to be retrieved with a "get-DataContext()" call - Controller objects, for example, do this. So a DBOBJECT being used within a Controller simply says:

1. custList.setDataContext(getDataContext());

to specify that it should use the Database/Context of the Controller - this makes the whole Controller object portable across databases.

It is good practice to always set the Database Context of a DBOBJECT just after it is initialized.

**Type Mapping**

All of the java.sql.Types data types are supported, and can be mapped to any database data type for complete DBMS independence. Fields can be accessed as any appropriate Java data type with automatic conversion. Special methods make handling date/time fields easier.

**Virtual Fields**

DBOBJECTs sometimes need to present fields that are not stored in the target table. They may be calculated (such as an invoice total) or retrieved from other tables (such as a code lookup table).

For example, you could define an Invoice DBOBJECT to have a field for Customer Name, showing the name of the Customer to whom the invoice is issued but it would be bad relational design to store the customer name in the invoice table (not normalized correctly).

You can instead set up a "virtual field" for the DBOBJECT, and provide the logic to look up the value. We add a call to our setupFields method:

1. addVirtualField("CustomerName", "varchar", 30, "Customer Name");

In our previous customer example we specified that Customer has a field Customer Name. We want our virtual field to look up the proper name automatically, making it seem as though the name is stored with the invoice.

In order to do this we extend the usual getField(String) method:

1. public String getField(String fieldName) throws DBException {  
2.     if (fieldName.equals("CustomerName")) {  
3.         Customer ourCustomer = newCustomer();  
4.         ourCustomer.setDataContext(getDataContext());  
5.         ourCustomer.setField("CustomerID", getField("CustomerID"));  
6.         ourCustomer.retrieve();  
7.         return ourCustomer.getField("CustomerName");  
8.     }  
9.     return super.getField(fieldName);  
10. }

As you can see, it is important to return the value from the superclass method for all other fields.

**Read-Only Fields**

Some fields in a DBOBJECT may have values that are set only when the record is first stored, such as a
creation date or sequential serial number. In order to specify this, the setReadOnly method can be used. For example:

```
setReadOnly("CreationDate");
```

If you need a sequential serial number, such as an invoice number, then you can use the "auto-inc" data type to have Expresso automatically assigns unique values for your records when you add the record to the database.

### Multi-Valued Fields & Framework Managed Relationships

Some fields in a DBObject may have only a specific set of allowed values. A simple example might be a Yes or No type of field. The only valid values are Y or N. In order to make this restriction, we call the setMultiValued method in setupFields():

```
1. setMultiValued("AccountOpen");
```

Once you've specified that a field is multi-valued, you must enumerate the possible values by one of two methods.

In the first case, you extend the getValidValues() method to supply the values available for the field. This is most appropriate for static values, such as in our example above:

```
1. public Vector getValidValues(String fieldName)
   throws DBException {
2.   3. if (fieldName.equals("AccountOpen")) {
4.      Vector myValues = newVector();
5.      myValues.addElement(new ValidValue("Y", "Yes"));
6.      myValues.addElement(new ValidValue("N", "No"));
7.      return myValues;
8.   }   9.   return super.getValidValues(fieldName);
10. }
```

Another method of providing valid values is to specify what is called a "lookup object" for the field. This is appropriate if the values for the field come from another DBObject, such as a code lookup table. For example:

```
1. setMultiValued("CustomerType");
2. setLookupObject("CustomerType", "com.yourcompany.dbobj.CustomerType");
```

If you choose the second option then you are allowing the framework on your behalf to manage the relationships between two data objects.

By setting metadata with setLookupObject(), this code assumes that the Customer Type object specified exists and implements the getValues() method, which returns the vector of valid values. The setMultiValued() defines the meta field to be multi-valued, that is accepts a value that belongs to a group of enumerated members. The setLookupObject() defines which database object to load, search and retrieve these enumerated values. In other words it instructs Expresso to lookup the enumeration values of this field name inside another database object. The first parameter is the field name and the second name is the fully qualified java class name of the target dataobject.

An example will make this clearer. Suppose we have an EMPLOYEE and DEPARTMENT database tables, and they have the following schemas.
CREATE TABLE IF NOT EXISTS EMPLOYEE (  
  INT Employee_Id,  
  VARCHAR(25) First_Name,  
  VARCHAR(25) Last_Name,  
  CHAR(1) Sex,  
  INT Dept_Id,  
  PRIMARY KEY (Employee_Id))

CREATE TABLE IF NOT EXISTS DEPARTMENT (  
  INT Dept_Id,  
  VARCHAR(25) Dept_Name,  
  PRIMARY KEY (Dept_Id))

You can see that DEPARTMENT has a foreign key Dept_Id into the EMPLOYEE table. When presenting a user interface to insert new employee's into the database, you want to display a selection of department name rather than identity names. Likewise with the new employee's sex you want to present a readable menu. You need to write your Employee DBObject to return a vector of valid values according to the right field and also set up the lookup meta data accordingly.

package org.fooey.testapp.dbobj;  
public class Employee extends DBObject {
  // ...
  public void setupFields() {
    // ...
    setMultiValued("Sex");  
    setMultiValued("Dept_Id");  
    setLookupObject("Dept_Id", "com.fooey.testapp.dbobj.Department");
  }

  public Vector getValidValues(String fieldName) throws DBException {
    if (fieldName.equals("Sex")) {
      Vector myValues = new Vector();  
      myValues.addElement(new ValidValue("M", "Male"));  
      myValues.addElement(new ValidValue("F", "Female"));  
      return myValues;
    }
    return super.getValidValues(fieldName);
  }
  // ...
}

So we have an Employee database object with the attribute "Sex" that handles the appropriate hard-coded valid values. To complete the look up code we implement the valid value look up in the corresponding Department database object. In order to handle this automatically we override the getValues() method.

package org.fooey.testapp.dbobj;
public class Department extends DBObject {
  // ...

  public Vector getValues() throws DBException {
  // ...
  }
}
Vector myValue = this.getValuesDefault( "Dept_Id", "Dept_Name" );
return myValues;

// ...

The method getValuesDefault() in the base class DBOject performs a search and retrieve of records from the database according to criteria. The first parameter is the target field name, and the second parameter is the description field name. It is functionally equivalent writing to the SQL code SELECT Dept_Id, Dept_Name FROM DEPARTMENT, in other words calling directly searchAndRetrieveList() except that the base class method as convenience stores the results, the valid values, automatically inside Expresso's standard cache manager. Brilliant!

To this issue clear again. The code 'this.getValuesDefault( "Dept_Id", "Dept_Name" )' automatically retrieve the data from the database as if it is stored like this:

<table>
<thead>
<tr>
<th>DEPT_ID</th>
<th>DEPT_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>PAYROLL</td>
</tr>
<tr>
<td>200</td>
<td>SALES</td>
</tr>
<tr>
<td>300</td>
<td>MARKETING</td>
</tr>
<tr>
<td>400</td>
<td>RESEARCH AND DEVELOPMENT</td>
</tr>
<tr>
<td>500</td>
<td>CUSTOMER SERVICES</td>
</tr>
</tbody>
</table>

The end result is an employee record that is controllable from DBMaint that renders a drop-down list for entering the sex of the new employee, and another drop-down list for choosing a list of departments.

A shortcut exists for implementing getValues() methods where the values to be returned are simply a key field and a description. This is the getValuesDefault(String) method.

Multi-valued fields are handled by the DBMaint program in a specified manner: When records are listed, the value description is displayed instead of the value itself (e.g., "Yes" is shown rather than "Y") and during field entry, a drop-down list will be presented for the user to choose from.

Future Framework Managed Relationship Possibilities

In the current Expresso Framework 5.5 the current method of specifying the target source data object is incomplete. If the target database table has a different named column name to the one in the source database table, then the framework cannot cope automatically. For example if the employee table has a column called DEPARTMENT_ID and the department table has a column DEPT_ID

It is envisaged that a new method call will be introduced handle this requirement in the near future.

Automatic Look-up of valid values

A field in a DBOject which is set to LookupObject is expected to refer to a DataObject which implements the LookupInterface.

```java
package com.jcorporate.expresso.core.dbobj;
public interface LookupInterface {
    public Vector getValues() throws DBException;
}
```

The DBOject implements the LookupInterface. If you would rather not write code to generate valid values look-ups automatically, then you use this method signature and the interface to return a single list of
valid values. Not with this interface *LookupInterface* you can only ever return one vector list of valid values. You cannot return valid values that depend on different field names. This is what the method *getValidValues()* is designed for! So for example consider a EMPLOYEE data object, then has a look-up values in another data object ACCOUNT such as

```
package org.fooey.testapp.dbobj;

public class Employee extends DBObj {
    // ...
    public void setupFields() {
        // ...
        setLookupObject("Account_Id", "com.fooey.testapp.dbobj.Account");
    }
}
```

The account object is rather simply written as the following code. Notice that it uses the convenience *getDefaultValues()* method to retrieve valid values and cache them simultaneously.

```
package org.fooey.testapp.dbobj;

public class Account extends DBObj {
    // ...
    public Vector getValues() {
        return this.getValuesDefault("Account_Id", "Account_Description");
    }
}
```

When working with multi-values data field names, there are basically two choices.

- **Override** *getValidValues()* and provide your enumerations for the field name by returning a list collection of *ValidValue* objects. The advantage here is that you have maximum control of what constitutes a valid value, you can specify what the label and the value is. You define the source of the valid values, whether it is hard coded, or if it comes from another dataobject, or wherever. There is a further advantage you can further internationalise the valid values. This is best practice.

- **Do not override** *getValidValues()* and let the framework try to provide enumerations for the field name. The default *DBObj.getValidValues()* expects each field name to be a multivalue and it will treat each multi-valued field name as look up on another DBObj. If this condition is not met, then you will get an DBException.

The javadoc makes this clear. "Database objects should extend this method to return Vectors of ValidValue objects for multi-value fields. A specific [database] object can return its own list [collection] of ValidValues, or it can call this superclass method to use the lookup object to get the list of valid values instead."

### Secret Fields

A field in a DBObj can also be set to "secret" so that its value cannot be seen by users with only "search" ability - the field is also shown as asterisks during data entry. Password fields, for example, might use this feature.

### Field Validation
DBObjects can be set up so that only appropriate values are accepted for fields. Several features facilitate this:

- **Null/not null:**
  The boolean parameter to the addField method is a simple form of validation - a non-null value must be specified for all fields where this value is false.

- **Data-type checking:**
  The data type is also a basic form of validation - only the appropriate type of value will be accepted, even if the field is set with a setField(String, String) call.

- **Valid Values:**
  Multi-valued fields are also validated - only a value found in the list of valid values is permitted.

- **Extend setField:**
  A more specific way of validating fields is to extend the actual setField method, like this:

  ```java
  public void setField(String fieldName, String fieldValue)
  throws DBException {
    if (fieldName.equals("Priority")) {
      if (!(fieldValue.equals("A") ||
            fieldValue.equals("B") ||
            fieldValue.equals("C"))) {
        throw new DBException("Priority must be A, B or C");
      }
    }
    super.setField(fieldName, fieldValue);
  }
  ```

**Field Filtering**

For security purposes, fields are able to be “filtered” on the way in and out of the database. In the setupFields method, where the fields of a DB object are specified, you can use the setFilter method to specify a filter for a field. By default, a field will be filtered via the "standard" filter which.

Also available for use are the following filters:

**Field "Masks"**

Fields can also be validated against a regular expression. In the setupFields method, the setMask(fieldName, Mask) method can be called to specify a regular expression that is associated with a particular field. The field then must "match" this expression in order to be considered valid. See the documentation for the Jakarta ORO project for more information on setting up Regular Expressions. The project is located at http://jakarta.apache.org/oro/.

Expresso also currently provides two predefined field masks for you: DBObject.INT_MASK and DBOject EMAIL_MASK to properly check for valid integer and email address entries respectively.

**Virtual Fields**

By adding virtual fields (e.g. calculated fields) to the database object, and by using "nested" database objects, data which is in fact stored across multiple tables can be dealt with by the application as a single
For example, if a virtual field in an invoice header gives the invoice total, the application does not need to deal with the invoice detail objects in order to get the invoice total information - this is an implementation detail which is hidden by the Invoice database object.

**Master/Detail Records**

Data is often represented in related tables with a master/detail, or one-to-many relationship. DBObjects support this kind of relationship explicitly, and by using the "addDetail" method, other database objects can be declared to be detail records of a master DBObject. This allows the detail records to be maintained easily by the DBMaint controller, and supports cascading deletes - that is, detail records can be automatically deleted when the master record is deleted.

**Declarative Referential Integrity**

Database objects can be set up with referential integrity between themselves by means of simple one-line method calls in the objects themselves. These integrity constraints are then automatically verified for any operations on these objects. It is also possible to easily implement cascading deletes and updates based on referential constraints.

To programatically implement referential integrity, you can override the CheckRefs method in your own DBObject to make sure no other DBObject that might depend on you is still pointing to your data.

**Multi-Valued Fields**

Fields can be specified as multi-valued, in which case the valid values for the fields can be returned by a call to the getValues method. These valid values can be cached for performance (via the ConfigManger class), and can be retrieved from another database object, or computed as needed. Special methods make adding validation checks for both multi-valued and ordinary fields very easy. The validation is then applied to all access to the database object, ensuring data validity.

**Field Descriptions**

The database object can return extended information on it's fields, including a long description (other than the database name) and other information.

If the value of this fieldDescriptions happens to be defined in your schema's Messages-Bundle_XXXX.properties then Expresso will automatically substitute the appropriate string from the appropriate Message Bundle instead, thus providing field name i18n support.

**Lookup Objects**

Any fields in a database object that is looked up or validated against the values in another database object can have the reference recorded. The client application can then request the name of the referenced object, perhaps in order to provide a list of valid values to the client or for displaying the relationship between the objects.

**Read-Only Fields**

Fields can be set as read-only if they are only manipulated from within the database object. An example of this is a sequential number key object.

**Sequential Numbers**

Current Expresso does not, by and large support Database native auto incrementing features. One example of a class that creates it's own Sequence type is located in the demo package called DBSequence. The biggest reason that Expresso does not at this time support this is due to the fact that databases all
have their own semantics for auto incrementing types. So Expresso provides its own through the auto-
inc fields and thus the underlying "NextNumber" classes.

Default Values

Default values can be specified that populates the tables created for each database object. The "popu-
lateDefaultValues" method of each DBObj ect is run when DBCreate is executed, and the default values for the DBObj ect are added in this method.

Read-Only Fields

Fields can be set as read-only if they are only manipulated from within the database object. An example of this is a sequential number key object.

Lookup Objects

Any fields in a database object that is looked up or validated against the values in another database ob-
ject can have the reference recorded. The client application can then request the name of the referenced object, perhaps in order to provide a list of valid values to the client or for displaying the relationship between the objects.

Using Database Objects

Now that you have created the DBObj ect, you can begin using it in your programs. We will examine each of the most common operations with DBObj ects.

Creating an Instance

Setting a Database Context

Virtually all of the standard programs that come with Expresso will utilize the currently logged-in con-
text for their database objects. Whenever a user is logged in, they establish, either explicitly or by de-
fault, a "current" database/context. Usually this is done by selecting a context from the drop-down list when logging in via the Login Controller, but it can also be implied by means of a special tag in JSP pages.

DBMaint, for example, will always set any DBObj ects that it uses for a particular user to use the cur-
rently logged-in context for that user. So, if the user is logged in to the "demo" database/context, all DBObj ects that DBMaint utilizes will have setDataContext("demo") called before they are used.

This provides the ability to have entirely separate contexts with distinct databases running on the same servlet/JSP/app server, with only a single VM.

Multiple database contexts are set up by having more than one "context" sub-element in the expresso-con-
fig.xml file - each "context" element can define a JDBC section to specify database connection informa-
tion for that context.

Adding Records

Adding new records is as easy as updating: simply populate the fields of the record (especially the primary key) and call add(). If a record already exists with the same primary key, an exception will be thrown.

You can also populate default values in the fields by calling setField("fieldName", getDefault-
Value("fieldName"));
Retrieving Records

Database objects can be used in an "aggregate" mode where a single DBObj ect represents a list of records, or other database objects. This allows searching and the result sets from searches to be manipulated. Result sets can be sorted on any fields, and searches can be made on any fields, including wildcard and range criteria. The count of records retrieved by a search can be obtained without accessing the entire result set.

It is also possible to return only the keys of records retrieved, in order to reduce the size of the data that must be manipulated.

In order to retrieve a specified DBObj ect corresponding to a particular row in the database, you must first initialize the object in your program:

```java
import com.yourcompany.dbobj.Customer;
.
.
.
   Customer oneCustomer = newCustomer();
```

This initializes one instance of Customer, called oneCustomer. Now to retrieve a specific customer, we must specify a value for the key field (or fields if there were more than one key field).

```java
oneCustomer.setField("CustomerID", "1");
```

This specifies a value for the Customer ID field as "1". Note that we specify the value as a string - there are also setField methods for other types but we can always use strings - the value will be converted to the appropriate type for us.

Everything we have done so far has been in memory, no access to the database has been made until we say:

```java
oneCustomer.retrieve();
```

This will access the database (or potentially the cache - more on this later) and retrieve the appropriate record for the specified key. Now we can access other fields in the record with getField(fieldName);

```java
System.out.println( getField("CustomerName") );
```

Will print Customer I’s name.

What if we don’t know the Customer’s ID but want to locate customers based on other criteria? The retrieve() method requires that the fields that make up the primary key each have a value specified - it will throw an exception if this is not the case. (Most methods of DBObj ects may throw DBException - you must enclose the above code in a catch/try block to handle this exception or your method must also throw DBException.)

Other methods for retrieval exist for when the key retrieved is not suitable. For example:

```java
/* Erase any current values in fields */
oneCustomer.clear();

oneCustomer.setField("CustomerName", "Jones");
if (oneCustomer.find()) {
   System.out.println("Jones found!");
}
```
The find() method, if successful, returns true and populates the DBObj ect with the field values from the database for the first matching record. If the find() does not locate any records it returns boolean "false" and the fields are not populated.

Retrieving Multiple Records

DBObjects can also be used to retrieve whole sets of records rather than one at a time. For example, let's say we want to perform some processing on all customers of type "AB" in our database. We can use code like this:

```java
import java.util.*;
1. Customer custList = new Customer();
2. Customer oneCustomer = null;
3. custList.setField("CustomerType", "AB");
4. for (Iterator e = custList.searchAndRetrieveList().iterator();
   e.hasNext(); ) {
5.   oneCustomer = (Customer) e.next();
6.   /* do whatever we need to do to oneCustomer */
7. }
```

Let's again examine this code line-by-line:

*Line 1:* We initialize a new Customer DBObject, custList. Rather than being used to deal with one customer record, this DBObject is used to retrieve a whole list of Customer objects.

*Line 2:* We declare a second instance of Customer to hold each individual customer that we retrieve. We do not need to initialize this instance, so we set it to "null" for now.

*Line 3:* Here we supply the search criteria to the custList object, specifying that we will be looking for records where CustomerType equals "AB".

*Line 4:* This is a complex line. We start a "for" loop by creating an iterator and initializing this iterator to the results of the "searchAndRetrieveList()" method from custList. We could write each step of this line separately like this:

```java
1. java.util.Iterator i = custList.searchAndRetrieveList();
2. while (i.hasNext()) {
3.   oneCustomer = (Customer) e.next();
4. }
```

*Line 5:* Here we get each Customer object retrieved individually, re-using the oneCustomer object to hold each record. So, the first time through the loop oneCustomer would hold the first customer record matching the criteria. The second, the next customer, and so forth. We can then use the information in oneCustomer to perform whatever processing we need in line 6.

In this way, we can process as large a list of records as is required, while still handling the database access as inefficient single lookup. This minimizes the access to the database for greatest efficiency.

Handling Ordered Data

If it is necessary to process the records in a specific sequence, another version of searchAndRetrieve can be called with a parameter to specify sort fields. For example:

```java
custList.searchAndRetrieveList("CustomerName");
```

Retrieves the specified records in Customer Name order (ascending). To specify descending (reverse) order, specify "Desc" on the end of the string, such as:
custList.searchAndRetrieveList("CustomerName Desc");

You can also specify multiple sort field by specifying more than one field name separated by a pipe "|" symbol, like so:

custList.searchAndRetrieveList("CustomerName|CustomerID");

This specifies that the records are to be retrieved in Customer Name order but with any Customers having the same name further ordered by Customer ID.

If no sort criteria is specified, it is *NOT* safe to assume that the records will be returned in any particular order. Often a database will return records in the order they were inserted, or in key order, but you cannot count on this being the case. If you need the records in a particular order, ask for it by specifying the parameter to searchAndRetrieve.

**Handling Large Data Sets**

As a general practice, you should specify your search criteria as narrowly as possible, in order to retrieve as few records as are needed to do the specified task. You can set criteria on as many fields as are needed and all will be combined to create the resulting record set. For example:

1. custList.setField("CustomerName", "Jones");
2. custList.setField("CustomerType", "AB");

This specifies that you want customers whose name is Jones and whose type is "AB".

Alternately you can use DBObjext.setMaxRecords(int), and DBObjext.setOffset(int) to limit the number of results set. The effectiveness depends on your database and config file setup, however, even if your database does not support limitation at all, you will save significant memory so you do not have to load large datasets into memory.

**Using Ranges and Wild Cards**

You can specify more than just exact matches when using search criteria. Wild cards and ranges can also be used, the exact syntax depending on the database engine being used. See the properties file for documentation or details on setting up the appropriate wild card characters for your specific database.

For example:

custList.setField("CustomerName", "A%");

Specifies a search for all customers with a name beginning with "A".

custList.setField("CustomerName", "[A-M]%");

Specifies a search for all customers with a name beginning with "A" through "M".

Also,

custList.setField("CustomerID", "BETWEEN 1 AND 20");

Specifies a search for customers with a Customer ID between the numbers 1 and 20.
Specifying search criteria carefully can reduce the number of records processed and speed up your ap-
You may also specify that a search only return a specific number of records at a maximum. This can be helpful when you need, say, only the first 100 customers matching certain criteria:

```java
custList.setMaxRecords(100);
```

Says that the `searchAndRetrieve` will retrieve a maximum of 100 records, even if more match the criteria.

You can use the "count()" method to see how many records will match your search without actually retrieving the records themselves:

```java
1. custList.setField("CustomerName", "A%");
2. int ct = custList.count();
3. System.out.println("There are " + ct + " customers with names starting with A");
```

If it is necessary to process a very large record set, you can use a flagging technique, such as this example. Here we assume that all records have a field "Processed" that is initially set to "N":

```java
1. Customer custList = new Customer();
2. Customer oneCust = null;
3. custList.setField("Processed", "N");
4. custList.setMaxRecords(100);
5. boolean moreRecords = true;
6. while (moreRecords) {
7.    for (Iterator i = custList.searchAndRetrieveList().iterator();
8.        i.hasNext();)
9.        oneCust = (Customer) i.next();
10.       /* Process Customer */
11.       oneCust.setField("Processed", "Y");
12.       oneCust.update();
13.   }
14.   if (custList.count() == 0) {
15.      moreRecords = false;
16.   } /* if */
17. } /* while */
```

The above code will process all customers by retrieving only 100 at a time.

**Specific Fields**

For large tables (e.g. a table with many fields), it is often more efficient for some queries to request and retrieve only the necessary fields: e.g. do something like a "SELECT a, b, c FROM..." rather than "SELECT * FROM...". DBObjects support this facility, by means of the "setFieldsToRetrieve(String)" method, which takes a pipe-delimited list of the fields that should be retrieved for subsequent queries.

**Max Records**

The `setMaxRecords` method can be used to tell a database object that subsequent calls to retrieve multiple objects (such as `searchAndRetrieve`) are only expected to return a certain number of objects - this can be useful when showing only the "first n" records that match a query, or providing other "query governor" functions to prevent extremely large result sets from being processed. If the database being used supports it, this function can be used along with methods that set the starting record of the selected set to be retrieved as well, making "page by page" operations very efficient.

**Caching**
In order to improve performance, DBObjects and valid values can be "cached" or stored in memory. As memory access is many times faster than disk access (or Database access) this can result in significant performance enhancements.

Caching of DBObjects can be enabled by adding an entry to the DBObjPageLimit table, specifying a non-zero value for the Cache Limit field. This field sets the number of that DBObject that will be cached, at a maximum. The cache manager will populate the cache on a most-frequently-used basis, so performance will be enhanced over time as the item's cache fills. The best settings for the cache limits for each DBObject depends on the exact nature of your application and the memory available to your JVM. Options on the JVM's command line (-Xmx for example) can adjust the amount of memory available for caching.

In addition, valid values can be cached and this can also be a significant performance enhancement. When using the getDefaultValues method, caching will automatically be used.

The DBObject and valid value caches are affected by updates, deletions and additions, so the cache values never become out of date. This is another important reason to use only DBObject database access in your applications - if you do, the cache values for records will always be up-to-date. On the other hand, if you were to use a direct SQL update, then the values in cache would be out of date compared to the latest values in the database.

For more details on caching, see the Caching documentation.

Custom where Clauses

If you need to specify "or" relations, or other special conditions, you can use the ability of DBObjects to set a custom "where" clause for the SQL query to be executed. For example:

1. custList.setCustomWhereClause
2. ("CustomerType = "AA" OR CustomerType = "BA");

The custom where clause only effects the next query run: it is reset after each query is executed, for safety.

Updating Records

As you can see above, updating records in the database is very easy. Call update() on the object you have changed and the changes are written back to the database. Only a single record is ever updated and it is always safest to retrieve() the record first. (See section later about transaction control for information about commit and rollback operations or updates.)

Deleting Records

Like add, delete requires that the DBObject have field values for at least the primary key field. The call

oneCustomer.delete();

removes the record specified by the oneCustomer object (see the later section on referential integrity).

Delete, like update and add, only affects one record at a time.

Security

Every interaction with a database object can be secured, and the security data is easily maintainable via Expresso's built-in capabilities. This allows the security to be updated from any location, and for the changes to take effect immediately.
An extension of the basic DBObject called SecuredDBObject uses a series of tables containing user and group information to supply database security at an object level. Users are collected into groups, and these groups given only what permissions they require to appropriate database objects. Database objects that inherit from SecuredDBObject make use of this security automatically, with no further effort on the part of the developer (or the DBA).

A DBOject can be accessed by any program and can read and write to any database object that the user specified in the property file (as the database user) has permission for - often this is the entire database. In order to define a standard and database-independent way of specifying database permissions, an extension of DBOject, called SecuredDBObject, is available.

By extending SecuredDBObject rather than just DBOject, your DBOject automatically gets to take advantage of this security capability.

Using built-in maintenance functions in Expresso, authorized users (such as system administrators) organize users into groups, then give these groups any or all of 4 possible permissions on each SecuredDBObject: Add, Update, Search, Delete. You can specify, with the setUser(String) method, the user running your program and the SecuredDBObject will automatically verify each operation requested against this security information. For example:

1. Customer oneCustomer = newCustomer();
2. oneCustomer.setUser("Fred");
3. oneCustomer.setField("CustomerID", "1");
4. oneCustomer.retrieve();

the above code will succeed only if the user with user ID 1 has "Search" permission on the Customer SecuredDBObject. If he does not, the call to retrieve() throws a Security Exception.

You can also check permission before calling a method, e.g.

1. custList.isAllowed("S");

will return true if the current user is allowed Search ("S") permission, false if permission is denied. A, U, and D can also be used to check Add, Update and Delete permission respectively.

This can be used to present only the appropriate choices to the user, showing only the options that are available to that user.

**Multi-Database Capability**

A database object can be set up to be accessed from an alternate data store, e.g. another database on the same server, or another database on a different server. This is very valuable in data warehousing scenarios, or where the control data for Expresso is stored in one database and the application data in another.

In order to use this ability, a few preparations must be made: Let's say for the sake of example, we have a DB2 database called "SALES" with a table in it called "CUSTOMER" that we want to access. The "SALES" database will not contain any Expresso specific tables, these will all be kept in our MYSQL "default" database. The XML tag that should be used is called <hasSetupTable>. The body content of this tag should be boolean value such as "yes" or "no", "true" or "false". Please not the default is to assume that the database context has Expresso specific configuration tables. Please also note the order and placement of the <hasSetupTable> it comes after the <description> (checked against the Expresso Configuration DTD; Peter Pilgrim 30th September 2002).

- First, we must prepare an appropriate expresso-config.xml entry for the new "SALES" database.

```xml
<?xml version="1.0" encoding="ISO-8859-1" ?>
```
<!DOCTYPE expresso-config PUBLIC
"-//Jcorporate Ltd//DTD Expresso Configuration 4.0//EN"
"http://www.jcorporate.com/dtds/expresso-config_4_0.dtd">

<expresso-config>
  <logDirectory>%web-app%WEB-INF/log</logDirectory>
  <strongCrypto>n</strongCrypto>
  <userInfo>com.jcorporate.expresso.services.dbobj.DefaultUserInfo</userInfo>
  <servletAPI>2_3</servletAPI>

  <class-handlers>
    <class-handler name="userInfo"
      classHandler="com.jcorporate.expresso.services.dbobj.DefaultUserInfo"/>
  </class-handlers>

  <!-- This database context does have Expresso configuration tables -->
  <context name="default">
    <description>Hypersonic Database</description>
    <hasSetupTables> true </hasSetupTables>
    <jdbc
driver="org.hsqldb.jdbcDriver"
    url="jdbc:hsqldb:%web-app%WEB-INF/db/default/default"
    connectFormat="3"
    login="sa" password="letmein"
    cache="y" />
    <type-mapping>
      <java-type>LONGVARCHAR</java-type>
      <db-type>LONGVARCHAR</db-type>
    </type-mapping>
    <images>%context%/%expresso-dir%/images</images>
    <startJobHandler>n</startJobHandler>
    <showStackTrace>y</showStackTrace>
    <mailDebug>n</mailDebug>
  </context>

  <!-- This database context does NOT have Expresso configuration tables -->
  <context name="other">
    <description>Other DB (Non-Expresso)</description>
    <hasSetupTables> false </hasSetupTables>
    <jdbc
driver="org.gjt.mm.mysql.Driver"
    url="jdbc:mysql://localserver/sales"
    connectFormat="4"
    login="root"
    password=""
    cache="y" />
    <type-mapping>
      <java-type>LONGVARCHAR</java-type>
      <db-type>text</db-type>
    </type-mapping>
  </context>
</expresso-config>
As you can see, we have flagged this database as not containing the regular expresso tables by means of the <hasSetupTables> element. This prevents Expresso from trying to read from the usual setup tables in this database.

- Now we code (or generate for) the Customer database object. This object is just like any other database object, containing fields that match the columns in the "CUSTOMER" database.

- Now we register the Customer dbobject with our Schema object for our application. We can choose to specify the dbcontext name that the object will be "mapped" to at this time, with the following:

  addDBObject("com.jcorporate.expresso.services.dbobj.Event", "sales");

- Alternatively, we can leave the mapping until runtime, and make an entry for the customer object in the DBOther Map table. This allows us to change the name of the mapped context without having to recompile anything.

Once you have defined and "mapped" your DB object, you access it just like any other DB object - it will "know" that it is always to interact with the "SALES" context and will ignore any calls to setDataContext to set its context elsewhere. In this way, the DBMaint controller can be used, even with "mapped" objects.

Security for the mapped object is still read from the "current" context - e.g. whatever context the user of the object has established by logging in. DB object security is otherwise applied normally.

Database Object Mapping

Many enterprise applications require access to the data stored in multiple databases. The "Database Object Mapping" functionality in Expresso allows a developer to define a new DB context to describe a particular database, and then associate particular dbobjects with that DB context so that when that object is manipulated, it is always operating against the correct DB context.

Expresso applications can now be run with a "split context", with one database handling the control tables (such as USERLOGIN, etc.), and one database storing the application data (like a data repository, shared by a few different Expresso applications). This allows you to create different "looks" or "views" to the same data, with completely independent logins, security groups, etc. The bottom line is that now you can specify many different database contexts, and define DBObjects as belonging to one of these contexts at the schema level. After this step, the system will always look to the right database when you use that DBObject.

Here is how it is done:

- Create a new DB context, other then your default context. For examples sake, lets say we are creating a DB context called "hr" that maps to our human resources database.

  To do this, we add a "context" sub-element to the expresso-config.xml file for this new context, with appropriate db connection information in a JDBC sub-element. (See the documentation on the expresso-config.xml file for details of the format to be used for this).
You may wish this new context to not contain any of the usual Expresso database tables. You can indica-
ticate this through use of the <hasSetupTables>false</hasSetupTables> element, as described in the
expresso-config.xml file documentation and the DTD for this file. This specifies to Expresso that the
context so labelled will not be used to store any of Expresso's own tables, such as the security and
group tables, setup tables, and so forth. This is usually preferable if the context is to be used strictly
for application data, with Expresso's setup information stored in a different context.

- We create a few new DBObjects that map to tables in the hr database. For example, we create a
DBObject called "Employee" that maps to an employee table in the hr database, and we create a
DBObject called "Certification" that maps to the certification table in the hr database.

- In our schema, we define the DBObject as "belonging" to our new "hr" context. We do this by
adding the DBObject to the schema in the following manner:

1. add(Employee(), "hr");
2. add(Certification(), "hr");

- We now run DBTool (or DBCreate). DBTool will see these directives and automatically create a
DBOtherMap entry to tell Expresso to always use these two DBObjects against the "hr" DB Context.
After running DBTool, the following two entries will appear in the "DBOTHERMAP" expresso ta-
bble in the default context:

<table>
<thead>
<tr>
<th>com.mypackagename.Employee</th>
<th>hr</th>
<th>Employee Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.mypackagename.Certification</td>
<td>hr</td>
<td>Certification Table</td>
</tr>
</tbody>
</table>

As you can see, the DBOTHERMAP table is used by Expresso to map DBObject class names to par-
ticular DB Contexts. This table can be directly manipulated to change the mappings of objects to DB
Context locations, but running DBTool with the proper schema entries is generally safer and easier.

- The DBOTHERMAP table entries are read into memory when Expresso starts up. Therefore, if Ex-
presso is currently running and you have made changes to the DBOTHERMAP entries, you need to
restart Expresso for the mappings to take effect.

- When Expresso starts, you will see a message that states: "Reading otherdb mappings...", "2 otherdb
mappings found".

- These objects will now always be fetched and saved to the "hr" context, rather then the default con-
text or the currently logged-in context.

**Important Note:** DBObjects mapped to otherdb locations will work in all cases where the object is al-
lowed to create its own connection. However, if you specify a connection object that is pointed to some
database, and then create the DBObject with this explicit connection, the DBObject will operate against
THAT database connection, regardless of what is specified in the DBOTHERMAP table. This is often
the case with a multi-part transaction against the database, where an explicitly allocated DBConnection
is used to be able to use "commit" and "rollback". When you create a DBObject with an explicit connec-
tion, you have the responsibility of making sure that the connection was made against the correct data-
base. One way to do this is to initialize the db object normally (e.g. without the explicit connection) and
use the "getDataContext()" method to see what connection it allocates for itself, then use this context for
subsequent connections.

**Change Logging**

Database objects can be set up to automatically log and track changes to their data, providing automatic
audit ability for critical data without additional development.
Transaction Control

When writing sophisticated applications you will sometimes need to have transaction control, that is, the ability to perform several database operations either all successfully or not at all.

Until now, our examples with database objects have all relied on the object's ability to manage their own database connection. Other than specifying the correct Database/Context, we have allowed the database objects to request connections from the appropriate connection pool and release them automatically.

Transaction control requires us to specify a particular database connection, which in turn allows us access to the connection's commit().

Note

MySQL users need to take care to make sure the tables for which transaction control is being used actually support transactions. With other databases, Expresso will throw an error if transactions are attempted to be used but not supported. Due to technical limitations of MySQL, these errors will not be thrown.

```java
1. DBConnectionPool myPool = null;
2. DBConnection myConnection = null;
3. try {
4.   myPool = DBConnectionPool.getInstance(getDataContext());
5.   myConnection = myPool.getConnection();
6.   myConnection.setAutoCommit(false);
7.   Customer oneCustomer = newCustomer(myConnection);
8.   Invoice oneInvoice = newInvoice(myConnection);
9.   /* populate the Invoice fields */
10.  oneCustomer.setField("Balance", newBalance);
11.  oneInvoice.add();
12.  oneCustomer.update();
13.  myConnection.commit();
14. } catch (DBException de) {
15.   if (myConnection != null)
16.     myConnection.rollback();
17.   throw newDBConnection(de);
18. } finally {
19.   if (myPool != null) {
20.     myPool.release(myConnection);
21.   }
```

*Line 1 and 2* declare the connection pool and connection objects that we will be using, which we declare outside of the try/catch block so that they are available in the "catch" and "finally" blocks.

*Line 3* begins the try block. All of the operations in the block must succeed or must not be done at all. For the purposes of our example, we assume we are creating a new invoice and recording the new customer balance when the invoice is added. If the invoice cannot be added correctly, the customer should not be updated, and visa versa - else the customer's balance would not agree with the total of invoices for that customer.

*Line 4* Here we request a reference to the appropriate connection pool object from the connection pool class. We pass the Database/Context name.

*Line 5* We request a connection from the connection pool and ...

*Line 6* tell this connection that it should not automatically commit updates, but should instead wait for
commit() to be called. This effectively begins the transaction.

Line 7 and 8 We now instantiate the Customer and Invoice database objects, passing the connection object we want them to use. The objects will now use the connection passed to them rather than requesting their own connections.

Line 9 and 10 We assume the appropriate code (perhaps a method call) to populate the fields of the invoice object and compute the new customer balance. We’ll assume the new balance is stored in the variable “newBalance”. This balance is set into the Customer object in line 10 (we also assume a field called “Balance” has been defined in our Customer object).

Line 11 and 12 are the statements that the transaction logic is concerned about. The customer is updated and the new invoice is stored in the database. If either of these operations fail, they will throw a DBException and execution will continue at line 15.

Line 13 calls commit() on the connection which confirms both operations to the database.

Line 15 and 16 handle the situation where either one of the updates or some other database operation has failed. The rollback() method ensures that no partial operations are written to the database and the throw clause re-throws the exception that occurred, allowing Expresso’s error handling to deal with it appropriately.

Line 18 and 19 are executed in either the success or failure case and are extremely important: the connection is released back to the connection pool for use by other objects. Without these lines, the connection would be held forever and the connection pool would rapidly run out of available connections.

You can determine whether or not the currently connected database supports transactions by means of the "supportstransactions()" method on the DB connection pool object, allowing your application to determine at run-time if transaction control is available to use. Many JDBC drivers will throw an exception if the setAutoCommit(false) method is called and they do not support transactions. For portability, it is best to check first.

### Status Information

The database object can tell the client it’s own status, via the getStatus() call, to allow the client to determine if updates are required, if the item has been deleted, if it needs to be stored, etc.

### Indexing DBObjects

For performance consideration, you can add an index on a field(s) with the following syntax, using commas to indicate multi-column indices. Of course, primary keys already have automatic indices. Add an index only for column(s) which are not already a primary key.

```java
protected synchronized void setupFields() throws DBException {
    ...
    // use syntax addIndex( "someIndexName", "someFieldName", whetherIndexIsUnique )
    addIndex("myIndexName", MY_FIELD_NAME, false);
    addIndex("myIndexName", MY_FIELD_NAME + "," + MY_SECOND_FIELD_NAME, false);
}
```

### Database Objects in Multiple Databases

Expresso has the capability to define and maintain connection pools for multiple databases, and to tie database objects to particular data sources.

### Why Multiple Databases?
There are several cases where you would like to deal with multiple databases. The most obvious reason for development purposes is to test your application against many databases at one time without having to modify many configurations. Other reasons include having a connection to a read only data warehouse application, virtual hosting, and others.

There are two ways to set up databases. One way is to define a full Espresso-like environment where each database has its own security tables, isolated data, etc. Each data context will also have its own security as well. The other way is for Expresso to store all of its own database bookkeeping in one data context and link to some pre-defined tables used in another database. This method is especially excellent for data warehousing applications where the back-end database must not have any bookkeeping tables installed in it.

Independent Multiple Databases

Expresso must have at least one database connection defined to operate called 'default'. The default database context contains setup and configuration information that Expresso needs to operate. The default database is always the database that Expresso will assume is being used when a DBObjec is manipulated unless setDataContext() [Or the older name setDBName()] is called first either directly or through an appropriate constructor.

If you take a look at your expresso-config.xml file you'll see some information like so:

```xml
<context name="default">
  <description>Hypersonic Database</description>
  <jdbc driver="org.hsqldb.jdbcDriver">
    url="jdbc:hsqldb:%web-app%WEB-INF/db/default/default"
    connectFormat="3"
    login="sa" password="letmein"
    cache="y"
    createTableIndicies="true"
    limitationPosition=""
    escapeHandler="com.jcorporate.expresso.core.db.DoubleQuoteEscapeHandler" />
  </jdbc>
  <type-mapping>
    <java-type>LONGVARCHAR</java-type>
    <db-type>LONGVARCHAR</db-type>
  </type-mapping>
  <images>%context%/%expresso-dir%/images</images>
  <startJobHandler>y</startJobHandler>
  <showStackTrace>y</showStackTrace>
  <mailDebug>n</mailDebug>
</context>
```

As you can see, at the top of the file, the context is named default as expected. The context definition includes a definition of a JDBC connection to a hypersonic database.

```xml
<!-- Old Context Handler -->
<context name="default">
  <description>Hypersonic Database</description>
  <jdbc driver="org.hsqldb.jdbcDriver">
    url="jdbc:hsqldb:%web-app%WEB-INF/db/default/default"
    connectFormat="3"
    login="sa" password="letmein"
    cache="y"
    createTableIndicies="true"
    limitationPosition=""
    escapeHandler="com.jcorporate.expresso.core.db.DoubleQuoteEscapeHandler" />
</context>
```

Database Objects

There are several cases where you would like to deal with multiple databases. The most obvious reason for development purposes is to test your application against many databases at one time without having to modify many configurations. Other reasons include having a connection to a read only data warehouse application, virtual hosting, and others.

There are two ways to set up databases. One way is to define a full Espresso-like environment where each database has its own security tables, isolated data, etc. Each data context will also have its own security as well. The other way is for Expresso to store all of its own database bookkeeping in one data context and link to some pre-defined tables used in another database. This method is especially excellent for data warehousing applications where the back-end database must not have any bookkeeping tables installed in it.

Independent Multiple Databases

Expresso must have at least one database connection defined to operate called 'default'. The default database context contains setup and configuration information that Expresso needs to operate. The default database is always the database that Expresso will assume is being used when a DBObjec is manipulated unless setDataContext() [Or the older name setDBName()] is called first either directly or through an appropriate constructor.

If you take a look at your expresso-config.xml file you'll see some information like so:

```xml
<context name="default">
  <description>Hypersonic Database</description>
  <jdbc driver="org.hsqldb.jdbcDriver">
    url="jdbc:hsqldb:%web-app%WEB-INF/db/default/default"
    connectFormat="3"
    login="sa" password="letmein"
    cache="y"
    createTableIndicies="true"
    limitationPosition=""
    escapeHandler="com.jcorporate.expresso.core.db.DoubleQuoteEscapeHandler" />
  </jdbc>
  <type-mapping>
    <java-type>LONGVARCHAR</java-type>
    <db-type>LONGVARCHAR</db-type>
  </type-mapping>
  <images>%context%/%expresso-dir%/images</images>
  <startJobHandler>y</startJobHandler>
  <showStackTrace>y</showStackTrace>
  <mailDebug>n</mailDebug>
</context>
```

As you can see, at the top of the file, the context is named default as expected. The context definition includes a definition of a JDBC connection to a hypersonic database.

```xml
<!-- Old Context Handler -->
<context name="default">
  <description>Hypersonic Database</description>
  <jdbc driver="org.hsqldb.jdbcDriver">
    url="jdbc:hsqldb:%web-app%WEB-INF/db/default/default"
    connectFormat="3"
    login="sa" password="letmein"
    cache="y"
    createTableIndicies="true"
    limitationPosition=""
    escapeHandler="com.jcorporate.expresso.core.db.DoubleQuoteEscapeHandler" />
</context>
```
When the servlet engine now loads Expresso, Expresso will load connections to both a data context named 'test' and one named 'default'. Using this configuration, each data context will be completely independent. You are in a position to run DBCreate on your new context. Go to the setup page, and click on the "Create/Verify Database Structure & Perform Initial Setup ", and you'll see a screen similar to that shown here.
In this I have 3 different contexts defined. I can choose to run DBCreate against any of them to define what I want.

So once we've run how do we use this? An application can have a user deal with multiple database contexts by calling the function setDataContext([Database Name]); So given the config file above, you could have the following code to retrieve a list of all UserPreference settings for the database context 'test':

```java
UserPreference userPreference = new UserPreference();
userPreference.setRequestingUid(SecuredDBObject.SYSTEM.Account);
userPreference.setDataContext("test");
ArrayList allPreferences = userPreference.searchAndRetrieveList();
```

When browsing code, you may find code that uses setDBName() rather than setDataContext(). The two functions are currently equivalent, although setDBName() will be eventually phased out of use, and thus it is recommended that you call setDataContext();

When working with controllers, there is a shortcut available for setting the security context of the database object as well as the user's id. If you are within a State handler for a controller (to be discussed shortly), the above code can be shorted to:

```java
UserPreference userPreference = new UserPreference(request);
ArrayList allPreferences = userPreference.searchAndRetrieveList();
```

Where request is the ControllerRequest object handed to you by the Expresso framework.
Application Integration

There is a user table in Expresso called AppIntegration that provides the ability for a single user to have the security in one database and operate in the other database. It provides the ability to set up, for example, a sample database for onsite demonstrations that provide the same user base as the primary site.

TODO: Explain This

DBOtherMap

DBOtherMap is used in the situation where you have all your Expresso book keeping tables in one database and certain data tables in another. Using DBOtherMap, you can make both datasources appear as one single data context.

DBOtherMap Config File Setup

First off, you’ll want to set up your multiple contexts in a similar way as to above with one attribute change:

```xml
<!-- Old Context Handler -->
<context name="default">
  <description>Hypersonic Database</description>
  <jdbc driver="org.hsqldb.jdbcDriver"
    url="jdbc:hsqldb:%web-app%WEB-INF/db/default/default"
    connectFormat="3"
    login="sa" password="warl0rd"
    cache="y"
    createTableIndices="true"
    limitationPosition=""
    escapeHandler="com.jcorporate.expresso.core.db.DoubleQuoteEscapeHandler" />
  <type-mapping>
    <java-type>LONGVARCHAR</java-type>
    <db-type>LONGVARCHAR</db-type>
  </type-mapping>
  <images>%context%/%expresso-dir%/images</images>
  <startJobHandler>y</startJobHandler>
  <showStackTrace>y</showStackTrace>
  <mailDebug>n</mailDebug>
</context>

<!-- Here we add our new context with setup tables == false -->
<context name="test">
  <description>A sample test database</description>
  <hasSetupTables>false</hasSetupTables>
  <jdbc driver="org.hsqldb.jdbcDriver"
    url="jdbc:hsqldb:%web-app%WEB-INF/db/test/test"
    connectFormat="3"
    login="sa" password="warl0rd"
    cache="y"
    createTableIndices="true"
    limitationPosition=""
    escapeHandler="com.jcorporate.expresso.core.db.DoubleQuoteEscapeHandler" />
  <type-mapping>
    <java-type>LONGVARCHAR</java-type>
    <db-type>LONGVARCHAR</db-type>
  </type-mapping>
  <images>%context%/%expresso-dir%/images</images>
  <startJobHandler>n</startJobHandler>
</context>
```

Database Objects

64
Notice the `<hasSetupTables>false</hasSetupTables>` addition to the above XML configuration. This tells Expresso Framework that it isn't going to find any setup tables such as Database object security, controller security, or Jobs under the "test" database context.

**Define your Schemas and DBObjets**

Next you'll want to define your schema and dbobjects that use the other database map. To do this, you first define your DBObjet as you would normally. But you define it in the Schema a little differently using the Schema function:

```java
protected synchronized void addDBObject(String dbobjectClass,
        String otherDBName)
```

rather than `addDBObject(Class);`. The otherDBName is the context name that this dbobject will eventually map to. So a concrete example is:

```java
addDBObject("com.mycompany.myapp.dbobj.myDBObject","test");
```

**Rerun DBCreate**

Ok, now rerun DBCreate, but run it against the DEFAULT context. This is a very important point since Security entries for your new dbobject will still need to be created in the default context. Once you finish it, if you take a look at the table in the setup page with the link name of 'Database Object DB/Context Mappings' You should now see your newly created otherdbobject sitting in that table.

**Program to use your new OtherMap object**

So now you're ready to use it. How do you go about it? It's actually really simple, you just program as if you're accessing the default data context and Expresso will do the rest of the work for you. So to access your object like so:

```java
MyDBObject myobj = new MyDBObject();
myobj.setRequestingUid(SecuredDBObject.SYSTEM_ACCOUNT);
myobj.setDataContext("default");
ArrayList allPreferences = myobj.searchAndRetrieveList();
```

In the above code, Expresso will automatically access the test context because of the entry in the other db map.

**Using DataObjects in conjunction with Serialization**

Expresso DataObjects are tested for serializability in their unit tests, to allow one to serialize dataobjects "over the wire", to disk file, or whatever you need for your application. The biggest issue is that DataObjects have a lot of precomputed member variables included in their state. What a typical user wants in more simply a HashMap of the field names keyed by value. Enter the DataTransferObject in package com.jcorporate.expresso.core.dataobjects

The only thing the DTO contains is the database context, the classname of the parent, and the value of the database fields. Nothing else is stored in it. The end result is that serialization size (and thus times) is approximately 1/3 that of the corresponding DBObjet.
Getting an instance of a DataTransferObject

DataTransferObjects are not instantiated directly. Rather you use an existing DBObjekt to create a Data-Transfer Object with the following code:

```java
DBObject myDBObj = new MyDBObject();
DataTransferObject dto = myDBObj.getDataTransferObject();
```

Rebuild a DBObjekt with a DataTransferObject

Obviously, once you've sent your DataTransfer object wherever you need to, you'll want to rebuild a corresponding DBObjekt based upon the contents of the DTO. You can do that with the following code snippet:

```java
//This code is the same as that above, we create the DTO
DBObject myDBObj1 = new MyDBObject();
DataTransferObject dto = myDBObj1.getDataTransferObject();
...
//Here we reconstruct the dbobject based upon the data transfer object
//Notice that the result constructs the right kind of DBObjekt
MyDBObject myDBObj = (MyDBObject)DBObject.getThisDBObj(dto);
```

Note

As of the time of this writing, DataTransferObjects have not been officially included into the rest of the DataObject API. The feature for Serialization is completely desired, so you can rest assured that it will be included, however the actual implementation may slightly change by the time it is fully incorporated into the DataObject API.

Conclusion

Contributors

The following persons have contributed their time to this chapter:

- Larry Hamel (JGroup Expert)
- Peter Pilgrim
- Mike Nash
- Mike Rimov
- Mike Traum (JGroup Expert)
- Sandra Cann

Note

Was this EDG documentation helpful? Do you wish to express your appreciation for the time expended over years developing the EDG doc? We now accept and appreciate monetary dona-
ations. Your support will keep the EDG doc alive and current. Please click the Donate button and enter ANY amount you think the EDG doc is worth. In appreciation of a $35+ donation, we'll give you a subscription service by emailing you notifications of doc updates; and donations $75+ will also receive an Expresso T-shirt. All online donation forms are SSL secured and payment can be made via Credit Card or your Paypal account. Thank you in advance.

[http://www.jcorporate.com/edgdoc.html]

Copyright © 2001-2004 Jcorporate Ltd. All rights reserved.
Chapter 7. Specific DataObject types.

Note
If you find this EDG documentation helpful please consider DONATING! to keep the doc alive and current.

Version: Expresso 5.5
Maintainer: David Lloyd
[mailto:dlloyd@jgroup.net?Subject=EDG]

Expresso provides the database abstraction layer through the DBObject class and DataObject Interface API. But it provides many more services through extending DBObject in several different subclasses to provide additional functionality. This chapter will cover several of the classes that extend DBObject/DataObject to provide special functionality.

MultiDBObject

Note
As of Expresso 5.1, it is recommended that new developers work with JoinedDataObject due to its increased flexibility and speed over MultiDBObject.

It is often necessary to deal with Join relationships between tables in relational databases - the MultiDBObject exists to handle this. If a DBObject is analogous to a table, a MultiDBObject is analogous to a view (of joined tables).

Many of the same operations available to DBObjects are available to MultiDBObject - including searchAndRetrieve(), clear(), setField (although with different arguments), etc.

When setting up a MultiDBObject however, you do not use addField - instead, you add entire DBObjects via the addDBObject method.

Also unlike DBObject, MultiDBObject is not an abstract class - you can directly instantiate MultiDBObject objects, rather than having to subclass. For example:

1. MultiDBObject myMulti = new MultiDBObject();
2. myMulti.setDataContext(getDataContext());
3. myMulti.addDBObject("com.jcorporate.expresso.services.dbobj.UserDBObj");
4. myMulti.addDBObject("com.jcorporate.expresso.services.dbobj.UserGroup", "group");
5. myMulti.addDBObject("com.jcorporate.expresso.services.dbobj.GroupMembers", "members");
6. setForeigntKey("members", "UserName", "User", "UserName");
7. setForeigntKey("members", "GroupName", "group", "GroupName");
8. MultiDBObject oneMulti = null;
9. myMulti.setField("User", "Username", "Fred");
10. System.out.println("User Fred belongs to the following groups:");
11. for (Enumeration e = myMulti.searchAndRetrieve().elements();
    e.hasMoreElements()); {
12.    oneMulti = (MultiDBObject)e.nextElement();
13.    System.out.println(oneMulti.getField("group", "Descrip"));
14. }

Long lines above are broken into multiple lines for clarity, but need not be in your application

Line 1 instantiates the MultiDBObject that we will query.

Line 2 sets the database/context of this MultiDBObject to the db/context of whatever object we are using
MultiDBObject from within - e.g. if we are using a Controller object, the "getDataContext()" method accesses
the name of the current database/context, making sure the MultiDBObject is operating within the
same context.

Lines 3 through 5 specify the DBObjects that this object "contains" and specifies a "short" name for the
objects - for example, com.jcorporate.expresso.services.dbobj.UserDBObj is referred to by the short
name "user".

Lines 6 and 7 establish the relationships between the 3 objects by specifying a foreign key object and
field and a related primary key in another object. Two such relationships exist in our example, the "mem-
bers" object's Username field must match the "user"'s object Username field, and the "members" object's
GroupName field must match the "GroupOwner" field in the groups object.

Line 8 declares a new MultiDBObject to hold each query result now - just like DBObjects.

Line 9 sets the search criteria - you will note that the object short name, field name and value must be
specified - this lets the MultiDBObject know which field in which object is to be set with the given
value.

Lines 11 through 13 retrieve the results of the query, just like DBObjects - except you will note that the
getField call also takes the "short" name of the object to retrieve the field value from. All 3 DBObjects
have their values populated for each result item returned.

You can also use the MultiDBObject by extending it in your own "predefined" MultiDBObject, by im-
plementing the setupFields() method, just like regular DBObjects but instead of calling addField you
specify addDBObject calls and setForeignKey calls.

MultiDBObjects are currently read-only. e.g. No Update operations are supported but these will be ad-
ded shortly.

**JoinedDataObject [Since Expresso 5.1]**

**Introduction**

As you might recall near the beginning of the chapter, Expresso is in the midst of abstracting services
common to DBObjects into a series of interfaces that provide for generic access to multiple data source
types. JoinedDataObject is an example of the results of this effort.

JoinedDataObject is very similar to MultiDBObject except that it was a ground up rewrite for efficiency,
as well as its goal is to act to a client programmer just like a standard DataObject. While the conveni-
ence services available to DBObjects are often not available, the DataObject API is no less powerful.
After all, DBMaint can accept any object that implements the DataObject interface for management. By
extension, DBMaint can then manager a JoinedDataObject just like any other DBObjec.
For those that are proficient in understanding UML, a UML diagram describing the JoinedDataObject class has been included here. The details won’t be gone into here, but remember that it simply uses the same interfaces that have been defined earlier in the other UML diagrams. The implementation is just different.

Creating a Joined DataObject

Defining your Join

This guide will not get into the details of database design when it comes to how to create a join on the SQL level. Rest assured, you will not have to figure out the SQL to create the join, you only need to know the theory of what is a join and when to use it.

So let’s get into a concrete example of define your join. If you ever dig through the Download controller and its related DataObjects, you will notice that there is a relationship between the DownloadLog table and the DownloadFiles table. Specifically, the DownloadLog contains the File number of the specified table. But what if you want to view the DownloadLog AND the DownloadFile at the same time without multiple browser windows? This is where the JoinedDataObject comes into play.

Instead of creating a new subclass like you do for many other DataObjects, in a JoinedDataObject, you simply write a definition file, a special XML file, to define how the DataObjects are going to be joined together.

So let’s show a JoinedDataObject that joins DownloadLog and DownloadFiles tables based upon the file number in Download Log.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE dataobject-join PUBLIC "-//Jcorporate Ltd//DTD Expresso DataObject Join 5.1//EN" "http://www.jcorporate.com/dtds/jdbc-join_5_1.dtd">
<dataobject-join description="Download Log View">
  <dataobject className="com.jcorporate.expresso.ext.dbobj.DownloadLog"/>
  <dataobject className="com.jcorporate.expresso.ext.dbobj.DownloadFiles"/>
</dataobject-join>
```
Now let's split up the definition file bit-by-bit.

1. XML Definition and Doctype: True to any XML document, you need to give it the xml processing syntax and a Doctype. The doctype for JoinedDataObjects is `database-join`. It is included within the Expresso code distribution for fast validation during runtime.

2. `<dataobject-join/>` This is the 'root' element of the join definition. It has one attribute called 'description', which is the equivalent of `setDescription()` for DBObjects. Whatever you put here, DB-Maint will list as the description for the DataObject. It has no other functional bearing.

3. `<dataobject/>` For each table you wish to join with this database object, you need to list it here as `dataobject` elements. They have two elements. The class name of the DataObject, and an alias for the DataObject, who's purpose will be explained later.

4. `<relations/>` Following the listing of all DataObject, the relations section defines how the various DataObject are related. Usually there will be (X - 1) relations, where X is the number of DataObjects used in the Join.

5. `<foreign-key/>` Now it's time to define how the tables relate to each other. In it is the 'left' table, which is referred to as the local alias and key here, and the 'right' table, which, here is referred to as the 'foreign' alias and key. As you can see from the example, the DownloadLog field named FileNumber corresponds to the DownloadFiles' field FileNumber.

When you have a complete file, you save it somewhere in your classpath, preferably somewhere that makes sense. An example we'll use is "/org/example/myapp/dbobj"

Now on to the purpose of an alias. As mentioned several times, JoinedDataObjects implement the DataObject API. One of the methods in the DataObject interface is: `setFieldValue(String fieldName, Object o)`. We all know about the field names for various DataObjects, but how do we specify which DataObject to set the field of? MultiDBObject used the method `setField(String shortName, String fieldName, String value);` Obviously, we can't use that and conform to the API. So JoinedDataObjects expect field names to be specially formatted. They go by the format `[alias].[fieldName]`. So, for example, to refer to the file number in DownloadLog, you would type in code:

```java
setField("DownloadLog.FileNumber",4);
```

Although JoinedDataObject parses the string, to all external clients, it looks like a single data object, with weird field names.

### Instantiating and Using a Joined DataObject

#### Manual Construction

JoinedDataObjects implement `Defineable` interface. That means that when you instantiate an Defineable object, you need to call the method `setDefinition()` before you can use the object. Defineable allows Expresso to uniquely differentiate objects by setting some sort of String value for the object to configure itself with. For AutoDBObject, for example, you set the table name to attach to. JoinedDataObjects take
the path (in classpath) to the definition file. So let's use a sample:

```java
import com.jcorporate.expresso.core.dataobjects.jdbc.JoinedDataObject;
import com.jcorporate.expresso.core.dataobjects.Securable;
//...

JoinedDataObject myjoin = new JoinedDataObject();
myjoin.setRequestingUid(Securable.SYSTEM_ACCOUNT);
myjoin.setDataContext("default");
myjoin.setDefinition("/org/example/myapp/dbobj/myjoin.xml");
```

### Factory Method Construction

Expresso 5.1 now has a DataObject factory that considers all the construction order of various dataobjects including Securable and Definable. Instead of the sample above, you can instead write the code:

```java
import com.jcorporate.expresso.core.dataobjects.DataObjectFactory;
import com.jcorporate.expresso.core.dataobjects.jdbc.JoinedDataObject;
import com.jcorporate.expresso.core.dataobjects.Securable;
//...

JoinedDataObject myjoin = (JoinedDataObject)DataObjectFactory
    .createObject(JoinedDataObject.class,
                 "default", "/org/example/myapp/dbobj/myjoin.xml",
                 Securable.SYSTEM_ACCOUNT);
```

Sometimes, the single factory method is more convenience than all the methods in the first example.

### Typical Usage

Since JoinedDataObject implements DataObject API, the much of the same logic for DBObjects can also be applied to JoinedDataObjects. Examples:

- Getting a field string value:
  ```java
  String myValue = myjoin.getDataField("alias1.field1").asString();
  ```

- Setting a field value:
  ```java
  myjoin.setFieldValue("alias1.field2","abcdefg");
  ```

- Retrieving search results:
  ```java
  myJoin.setFieldValue("alias1.field3", "ab%";
  myJoin.setFieldValue("alias2.field2", "e%");
  java.util.ArrayList results = myJoin.searchAndRetrieveList("alias1.field1")
  ```

  This version searched for all records where table 1 field 3 starts with 'ab' and table 2 field 2, starts with 'e'. It then orders the results by table 1 field 1.

- Clearing loaded fields
  ```java
  myJoin.clear();
  ```

### Accessing JoinedDataObjects in DBMaint
Since Expresso 5.1, DBMaint has been rewritten to manage any DataObject, whether it is backed by a database or an in-memory hashmap. So, of course, DBMaint can now handle displaying and managing JoinedDataObjects.

Traditionally, you launch DBMaint with the 'dbobj' parameter telling it what class to instantiate and use. But of course, JoinedDataObjects use the 'Defineable' interface to uniquely retrieve their definition. To this end, the 'definition' parameter has been added to DBMaint's capability. The definition parameter is identical to that of what you set earlier when instantiating it by memory. So to list all records of the join, enter (all in one line)

```
```

Performing 'negative' security overrides

Normally, JoinedDataObject checks the security of all dbobjects involved and takes the most negative result. In other words, a security check in a JoinedDataObject only returns true if access is granted for all the joined objects.

However, sometimes you flat don't want certain actions available to the client. An example of this might be the 'Add' action. Sometimes adds on a join can be extremely fickle. While JoinedDataObject first adds the rightmost object (the last data object defined in the xml file) and then works to the left (the first data object defined in the xml file), sometimes it's too likely that the end user will muck up the relations. So Expresso provides a feature to allow you to turn off the create/read/update/delete capabilities in an override fashion. To do this, you add a new element to your join definition file called 'permissions'. An example is like so:

```
<dataobject-join description="Download Log View">
  <dataobject className="com.jcorporate.expresso.ext.dbobj.DownloadLog" alias="DownloadLog"/>
  <dataobject className="com.jcorporate.expresso.ext.dbobj.DownloadFiles" alias="DownloadFiles"/>
  <dataobject className="com.jcorporate.expresso.services.dbobj.MimeTypes" alias="MimeTypes"/>
  <distinct distinctJoin="true"/>
  <permissions add="false" read="true" update="true" delete="false"/>
  ......
</dataobject-join>
```

Take a look at the permissions line. In this case, we never want to allow an 'add' or 'delete' through this join. But don't mind allowing updates or reads. In short, we want this particular join to allow easy browsing of the underlying database tables, but not allow the user to seriously modify the underlying data. (In 'update', the user is not allowed to modify the foreign keys in an object, so referential integrity will not be destroyed)

MediaDBObject [Since Expresso 5.1]

A frequent desire when working with a database is to be able to upload and store various media into a database. But when you do this, there are several issues that might arise. How do I store the file name? How do I tell what type of file (image/video/text) the stored BLOB is? How can I provide a way with retrieving the file. To this end, Expresso 5.1 now has the MediaDBObject to assist in your file storage

Creating a MediaDBObject

Creating a MediaDBObject basically two steps: to start with, derive your class from com.jcorporate.expresso.services.dbobj.MediaDBObject. Secondly, instead of calling addField() in your setupFields function, you call addBlobField() instead.

A concrete example:
public void setupFields()
    throws DBException {
    setTargetTable("TESTMEDIA");
    setDescription("Blob Storage Test");
    setCharset("ISO-8859-1");
   addField("TestKey", "auto-inc", 0, false, "Test Table Key");
    this.addBlobField("Data", "Sample Media Field");
    addKey("TestKey");
}

The crux of the above code is the addBlobField(). In reality all it does is create three additional read-only fields for each file. You do not have to worry about these fields yourself as MediaDBObject uses and sets them automatically for proper BLOB bookkeeping. You CAN reference the fields if you so wish for your own programming needs.

- Mime Type - What is the MIME type of the file that is stored in this database field
- File Size - How big is the file that is stored in the database?
- File Name - What is the name of the file being stored in the database?

**Note**

You *can* have more than one BLOB field in a MediaDBObject.

**Programmatically using the Media DBOBJECT**

You can save your BLOB fields in the following example code:

```java
// Open the file, get its length
File f = new File(importFile);
InputStream is = new FileInputStream(f);
int fileSize = (int) f.length();

// Create an instance of your media dbobject.
MyMediaObject myObj = new MyMediaDBObject(Securable.SYSTEM_ACCOUNT);
myObj.setDataContext("default");

// Add the dbobject with key == 1
myObj.setField("key", 1);
myObj.add();

// Now we save the BLOB fields in a different 'statement'.
myObj.saveBlob("Data", is, importFile, fileSize);
```

The code to retrieve the BLOB is similar:

```java
// Open the file, get its length
File f = new File(exportFile);
OutputStream os = new FileOutputStream(f);

// Create an instance of your media dbobject.
MyMediaObject myObj = new MyMediaDBObject(Securable.SYSTEM_ACCOUNT);
myObj.setDataContext("default");

// Load the dbobject with key == 1
myObj.setField("key", 1);
```
myObj.retrieve();

// Now we retrieve the BLOB fields in a different 'statement'.
InputStream is = myObj.retrieveBlob("Data");

//
// The following code is basically the fastest way to read an input
// stream and copy it to an output stream. Error checking for
// flush and close() have been removed.
//
byte[] buf = new byte[4096]; // 4K buffer
int bytesRead;

while ((bytesRead = is.read(buf)) != -1) {
    os.write(buf, 0, bytesRead);
}

os.flush();
os.close();

Saving Time - Using DBMaint

Ok, so that may not appear to be the easiest way of dealing with it. How about if we look at DBMaint? If you haven't encountered DBMaint already, you soon will. It is saturated throughout Expresso for maintaining database tables. It has search/add/update/delete capabilities and if you take the time to add a custom UI, can save you buckets of coding time for webapp management controllers.

If you use the URL with something like:


You will notice that there is a 'File Upload' field where you added the BLOB field. You can use this to upload any file to the database and DBMaint will take care of the rest for you. Use the browse button to find a file to upload and hit 'save' record. The file will automatically be uploaded and saved.

A brief note about database compatibility with BLOB datatypes

As the Expresso development team has been exploring BLOB capabilities with various databases, we've been disappointed in performance capabilities with the various databases:

- HSQLdb - Because all BLOBs are HEX encoded, you have a practical limit of about 200K before you run out of memory.
- Oracle - We now have code that uses Oracle's native BLOB API. BLOBs should work well here.
- Most other databases - It appears that most of the time the JDBC driver loads the entire BLOB into memory before handing Expresso the InputStream to read it. This could create large memory issues if BLOBs are heavily used in conjunction with multi-megabyte BLOBs.

You have been forewarned!

AutoDBObject

The AutoDBObject is the easiest way to get access to your database tables and can be very valuable for
prototyping your application. AutoDBObject can populate its fields automatically from the schema information of its target table. This allows an AutoDBObject to be instantiated and used to access a table without any coding at all! The DBMaint servlet has a special parameter to allow an AutoDBObject to be used:

1. /DBMaint.do?dbobj=com.jcorporate.expresso.core.dbobj.AutoDBObject&definition=SCHEMALIST&state=List

This command will list (and enable editing) on the SCHEMALIST table on the current database. No coding at all is required but the user must have access to the AutoDBObject object (AutoDBObject is a SecuredDBObject).

Warning: Giving a user full access to AutoDBObject allows them read/write access to any table in the current database (or at least any tables that the database user specified in the property file has access to). It should be used with great caution, particularly in a production environment.

Row Secured DOBObjects

Recent additions to Expresso are the ability to provide DOBObjects that secure database tables row-by-row. Row Secured DOBObjects are further explained in the EDG Security Chapter

SynchronizedDOBObjects

DOBObjects and DOBObjects themselves are not thread safe. While the words "not thread safe" may inspire much great fear among many readers, it isn't nearly as bad as it sounds. Each request in a Servlet environment is handled by one thread. You don't start the request having it handled by one thread and finish the request having it handled by another. The only time you need something thread safe is if you are sharing the same instance between multiple requests, or possibly something that is being accessed on a remote machine by multiple clients. Either way, a thread-safe DOBObject is actually seldomly required.

On very rare occasions, you might want to share a DOBObject between threads. While it is more often advisable to perform some sort of manual synchronization to ensure data integrity, Expresso 5.3 now has a wrapper class called com.jcorporate.expresso.core.dataobjects.SynchronizedDOBObject that creates a thread-safe variety of DOBObjects.

SynchronizedDOBObject example

DataObject sharedObject = SynchronizedDOBObject.newInstance(mydbobj);

The concept is simple, pass the newInstance method an already instantiated instance of a non-threadsafe DOBObject or DOBObject, and it will wrap that object in a threadsafe manner. You can use the threadsafe dataobject in the same way you could use any other DOBObject. The API is conformed to.

Performance Note

SearchAndRetrieve for a SynchronizedDOBObject is not recommended for performance reasons: what happens under the hood is that the data object locks, creates a search and then copies it to the target thread. The result is much overhead in the copy, especially if you have large resultsets. Use with care!

SecurityDOBObjects
Take care with the spelling: SecurityDBObject is described here; contrast this with SecuredDBObject, which is much more commonly used, and is discussed in the Security chapter. You probably will never create a subclass of SecurityDBObject yourself—it is a framework tool embedded in the Expresso security system. The idea is to provide a flexible way to redirect authentication classes to another database context in another Expresso application. For example, consider a situation where there are two Expresso applications, and you wanted the second application to rely on the first in order to authenticate users and otherwise supply user information.

The following classes extend SecurityDBObject in Expresso:

- SecurityDBObject provides a means to control the DB context (for some or all of the classes listed above) via Setup values. Two Setup values are important for SecurityDBObject:

  - If these Setup values are empty, SecurityDBObject does nothing special. However, if these 2 Setup values contain meaningful info, subclasses of SecurityDBObject may ignore any DB context supplied (e.g., they can ignore a context like 'default' that comes from the ControllerRequest). The logic is two-fold: the Setup value SecurityDB must be filled, AND the Setup value SecurityDBObjs must contain the fully-qualified class name of all objects (from the list of SecurityDBObject subclasses listed above) that will use the 'fixed' context value in Setup value SecurityDB. That's a bit tricky, so to repeat: even though all the security classes listed above are instances of SecurityDBObject, each one will use the fixed context only if that individual class is also listed in the Setup value, SecurityDBObjs.

  - This scheme is useful in at least one case: one primary Expresso application supplying User information to a secondary Expresso application. For example, consider a case where we have a primary application 'Primary', and a secondary application 'Secondary'. Primary has standard Expresso Setup values and behavior—the Users are all described in the local database, and the Setup value 'SecurityDB' is empty. Primary knows nothing about Secondary. Secondary has two database contexts defined: "default" and "userInfoDB", where "default" context includes Setup values, but "userInfoDB" does not.

    <context name="userInfoDB">
    <description>mysql DB for user info</description>
    <hasSetupTables>false</hasSetupTables>
    ...

  - The Secondary application also has Setup values entered to indicate special context handling for two classes (in this example, these setups are added programmatically within the SecondarySchema constructor):

    addSetup(ExpressoSchema.class.getName(), SecurityDBObject.SECURITY_CONTEXT, "Database to use for User/Group Security Info", "userInfoDB");
    addSetup(ExpressoSchema.class.getName(), SecurityDBObject.SECURITY_OBJECTS,
"Database Objects that use the SecurityDB context",
DefaultUserInfo.class.getName() + " " + UserGroup.class.getName());

addSetup(ExpressoSchema.class.getName(), DObject.IS_CHECK_RELATIONAL_INTEGRITY,
"check relational integrity w/i middleware", "N");
...

As always, this sample may not be representative of your needs, so attempt this only with caution and a single-step debugger handy.

Conclusion

Contributors

The following persons have contributed their time to this chapter:

- Larry Hamel (JGroup Expert)
- Peter Pilgrim
- Mike Rimov
- Mike Traum (JGroup Expert)

Note

Was this EDG documentation helpful? Do you wish to express your appreciation for the time expended over years developing the EDG doc? We now accept and appreciate monetary donations. Your support will keep the EDG doc alive and current. Please click the Donate button and enter ANY amount you think the EDG doc is worth. In appreciation of a $35+ donation, we'll give you a subscription service by emailing you notifications of doc updates; and donations $75+ will also receive an Expresso T-shirt. All online donation forms are SSL secured and payment can be made via Credit Card or your Paypal account. Thank you in advance.

[http://www.jcorporate.com/edgdoc.html]

Copyright © 2001-2004 Jcorporate Ltd. All rights reserved.
Chapter 8. Using Controller Objects

Note
If you find this EDG documentation helpful please consider DONATING! to keep the doc alive and current.

Version: Expresso 5.5
Maintainer: David Lloyd
[mailto:dlloyd@jgroup.net?Subject=EDG]

Expresso complements the J2EE architecture by providing a framework for a clean separation of data, presentation, and application logic.

Introduction
An Expresso Controller is a Struts Action subclass. (Contrary to what may be initial expectation, a "Controller" does not therefore extend the Struts ActionServlet.) The Expresso Controller component is thus strictly part of the front end: the web tier. Controller objects provide a means for a sequence of interactions to be encapsulated in a way that makes them available to virtually any kind of user interface (e.g. servlet, JSP, Applet, Application and others). A Controller is a finite-state machine, where the flow from one state to another is directed by the Controller itself, and by the actions the user takes.

Controller objects should be used to encapsulate the logic of your application, especially any user-interaction required. The idea is to take advantage of things that are provided by the framework and not re-invent anything that is not required. You must design your user-interface in a presentation-independent way, thinking in terms of abstract inputs and outputs, to take best advantage of Controller objects. This allows you to take advantage of the GUI-independent user-interface abilities in Expresso, easily modifying the "view" of your application even once the logic is completed.

Expresso has a built-in mechanism by which you can specify security for each of your your controllers. For example, a group of users can be allowed or not allowed to access the entire controller, or can be granted permission to only specified states within a controller. This allows a single controller to be used by a wider audience of users, where all states might be available to only certain users (say, system administrators).

MVC Architecture
Keeping the elements of Model (business tier), View (presentation and user interface) and Controller separated is a central issue of good application design. The Expresso "Controller" objects make this easy when building web applications. Expresso includes a package of components for creating several types of "Controller" objects. These Controller objects encapsulate a series of interactions with the user, in a manner similar to Session EJB's (in fact a Controller can be a Session EJB in an environment where EJB's are supported). The Controller can be utilised from any kind of client: a Servlet, a JSP, an Applet, or an Application.

The Apache Jakarta project has a sub-project called "Struts" that has complimentary aims to the Controller object in Expresso. Struts is considered by many to be the defacto MVC implementation. Therefore Expresso is integrated with Expresso (as of v4.0) and brings the best of both worlds to Expresso. The struts-config.xml file provides the mappings required by the Struts frameworks from URL's to "Action" objects - in this case, to the Controller objects in Expresso (which are sub-classes of the Action object.) In addition to the normal struts-config.xml file, applications that have their own Controller objects can provide mappings for them in separate configuration files.
For example, eForum has an eforum-config.xml file that provides mappings for its Controllers. In this way, the core struts-config.xml file for Expresso does not need to be adjusted as you install or develop other applications, and the individual configurations for your own applications can be easily adjusted and kept independent.

Expresso includes a package of components for creating several types of "Controller" objects. These Controller objects encapsulate a series of interactions with the user, in a manner similar to Session EJB's (in fact a Controller can be a Session EJB in an environment where EJB's are supported). The Controller can be utilized from any kind of client: a Servlet, a JSP, an Applet, or an Application. Controllers are covered in more detail in their own chapter in this guide.

Controllers Overview

Controller objects provide a means for a sequence of interactions to be encapsulated in a way that makes them available to virtually any kind of user interface (e.g. servlet, JSP, Applet, Application and others). A Controller is a finite-state machine, where the flow from one state to another is directed by the Controller itself, and by the actions the user takes. Controllers are collections of States. A State generally corresponds to a single web page. The controller can arbitrarily encapsulate a number of states, as decided by the developer, but usually states are grouped by purpose. For example, a state that prompts for some input, and the state that handles the posting of that input, would usually be grouped together in the same controller.

Furthermore, Expresso supports chaining states together in a "wizard" or "process funnel", also known as workflow. This workflow association makes the controller the ringmaster for validation, and puts the definition of workflow into the controller. See the workflow chapter in this manual.

Controllers can be secured using the security mechanisms of Expresso - a group of users can be allowed or not allowed to access the entire Controller, or can be granted permission to only specified states within a controller. This allows a single controller to be used by a wider audience of users, where all states might be available to only certain users (say, system administrators).

Why Use Controllers?

Controllers provide a number of advantages:

- **Encapsulate business logic separately from user interface logic**: Controllers are not concerned with the user interface that presents their Inputs, Outputs and Transitions to the client - this separates the business and user interface logic cleanly, and allows each to be maintained as independently as possible, promoting good design practices.

- **Dynamic Secured Access**: Every state of a controller can be secured, and the security data is easily maintainable via Expresso's built-in capabilities. This allows the security to be updated from any location, and for the changes to take effect immediately. This makes controllers ideal for situations where a user's permissions might be updated dynamically - for example, when a customer completes a controller allowing him access to some on-line information, the successful completion of one controller could then permit him access to additional states in other controllers.

- **MVC Architecture**: Controller objects provide the "Controller" portion of the MVC architecture in a way that is portable across all types of Java environments. They can scale all the way from a Java Micro-edition environment to a complex multi-server cluster using EJB's and application servers.

- **Default User Interface**: There is a default ViewHandler implementation that is used where no "custom" view is defined, which can be used to run a controller without the need to program a custom GUI or design with JSPs, allowing a basic controller to be deployed very quickly, then perhaps enhanced later with a custom UI.
• **XML User Interface**: In additional user interface option allows controller responses to be sent via XML, optionally transformed via an XSL stylesheet. In addition to providing great UI flexibility, this mode is also very handy for debugging.

• **Session Management**: Controllers themselves do not preserve any information about their state from one invocation to another, requiring their input items and parameters to provide them with the information needed to process the next state. They can, however, use a PersistantSession object to preserve state information across invocations, and to make it available to other controller objects.

• **Test Harnesses**: Useful test harnesses exist for testing controller objects, both from the command line and from a JSP.

## Pitfalls to Avoid in Using Controllers

Controllers are generally run in a servlet context. This means that they remain resident in RAM, and multiple threads run through them. Controllers must be thread safe in this context. Thread un-safe code is the 'bogeyman' that haunts any servlet project, because the bugs are often intermittent and hard to reproduce and fix. And they happen under load, when stability is particularly valuable. Part of the EJB design is to remove multithreading as a programmer responsibility, by having pools of EJB objects and running their methods in a strictly controlled manner, cloning EJBs if necessary, avoiding concurrency. However, EJBs engender their own complexity too, and for some purposes, there can be smart applications of the concept of one object, many threads.

The following references and cautions apply to Controllers:

• *Understand servlets* [http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/Servlets5.html#102985].

• *Understand concurrency* [http://gee.cs.oswego.edu/dl/cpj/].

• Of primary concern are any data members in your Controllers, and in the supporting expresso framework. Think twice about adding data members to a controller. For example, if you need some kind of cache for efficiency, use synchronized containers like Hashtable, or use synchronized blocks for initialization of more efficient tools like the *oswego collections* [http://gee.cs.oswego.edu/dl/classes/EDU/oswego/cs/dl/util/concurrent/intro.html] (note that this jar is already available in the Expresso library).

• Static data members are even more prone to thread-safety trouble. To synchronize these, the synchronization lock must be on a static object; in particular, the "synchronized" keyword in a method signature will not suffice unless that method is a static method.

• Test code under load. Use the Morebot [http://morebot.org] tool or your own custom tool to simulate the activity of many users touching the active pages in your application. If you get reports of intermittent failures, adapt the test to focus on the pages in question in order to reproduce the failure.

## UI-Independant

The controller architecture is independant of the method used to present the outputs of the controller to the user - User Interfaces (UI) are available for several different web-application technologies, including:

• **JSP (Java Server Pages)**: This is a popular method of creating the "view" for Controller objects.

• **JSF (Java Server Faces)**: 
• **WebMacro:** This templating engine can be used to create the view as webmacro template files from Controller objects.

• **Text:** Useful for testing, ordinary command-line operation of controllers is possible.

• **XSL:** Where the UI may be a mobile device, WebTV, a PDF reader, or any of a number of other kinds of browsers, XML outputs can be generated from Controllers that can be transformed via the appropriate XSL stylesheet into WML, PDF, HTML, XHTML, XML, and many other formats. This function is part of the Expresso XML sub-project.

### Form-Handling

Special support facilities exist to make using Controller objects easier in a web forms-based environment, providing the ability to validate forms, especially against DB Objects, supplying error handling and retry abilities, and provide the ability for the user to navigate between forms easily.

### Controllers and EJBs

DBObjects are not Enterprise Java Beans. However, if you have Expresso Framework 5 downloaded on your webserver, you can work with Enterprise Java Beans if you want to. There are no restrictions. However since Expresso does not create or manipulate EJBs directly (at the moment), but instead talks to the database using DBObjects, you will have to do the extra work yourself. Briefly, EJB technology is designed to be distributed component based, transaction safe, scalable across virtual machines, and portable across J2EE application servers. This is important if you run Enterprise mission-critical applications. In detail, EJB provides the following features:

• **Distributed Component-based:** EJB is a component build technology that permits deployment across local and foreign servers.

• **Transaction safe:** J2EE specifies level of transaction control for Enterprise JavaBeans so that two separate database transactions can be undertaken without serious race conditions, double insertions, or phantom reads.

• **Scalable:** J2EE specifies that application servers can deploy EJBs across virtual machines on different CPUs and across networks. So if you buy an application server you can take advantage of clustering features, load balancing, etc. for your EJBs. (For example, if the server in London crashes, then the one in New York can take over running a bank business 24/7/365 over the web. Moreover, if you find that web business doubles overnight then you can buy more application servers and spread the work.)

• **Portable:** J2EE certified application servers can deploy EJBs that are written on another EJB server. Your mileage may vary if you use proprietary server’s features of course, but that's life.

DBObjects (in Expresso 5.X) do not have any of the above Enterprise features. However DBObjects are better than nothing if you have no Object Relational layer. DBObjects are good because you can manipulate database tables within Java, the language we all love. DBObjects are an implementation of a well-recognized software design pattern (see Mark Grand, J2EE Enterprise Patterns, Wiley 2001) For more info look at TheServerSide.com for Chuck Cavaness’ preview book chapter on “EJB and Struts”.

### Controller Basics

A Controller object can be thought of as a finite-state machine, transitioning from state to state and performing the appropriate business logic at each state. These states comprise the ”Model” of your applica-
tion - it is here that the application interacts with the database, with the user, with files and other resources, and does it's processing. So if each state represents a particular step or unit of processing, the controller can be thought of as a (finite-state) machine, and the function of transitioning from one state to another is how the controller gets it's work done.

What is a (deterministic) finite state machine? Mathematically, it is a directed graph, where the nodes are "system states" and the arrows are labelled with "user actions". Thus given any state and a user action, exactly one state results and the finite state machine has the responsibility of determining what that resultant state is. Intuitively, you pick some state, follow some user action - via an arrow out of that state - and the node at the end of that arrow tells you what state results due to that action. One state is designated as the "initial state" and some nonempty subset of states as the "final states". Intuitively, the final states are thought of as "successful states": reached via successful completions of (possibly) different user actions.

Controllers are basically a collection of "States", where each state represents a particular step or unit of processing. Thus, the controller is really a finite-state machine, and the function of transitioning from one state to another is how the controller gets it's work done.

As a controller transitions to a new state, it generates a ControllerResponse object. This object contains a group of ControllerElement objects, of 4 types:

**Inputs**

An Input object is a request from the Controller for the client to supply some information. Some additional attributes of the Input object can provide some formatting "hints", which the client may or may not use, but the actual presentation to the user is up to the UI portion of the application. Input objects may also provide "valid values", which allow the UI to present a list of values to the user for him/her to select from. Again, the nature of this list is up to the UI layer.

Normally, if a particular state of a Controller requests an input, the subsequent states will require it as a parameter.

**Outputs**

An Output object represents a response from the Controller to be presented to the user. This can be as simple as a single String, or as complex as a nested tree of items, that the UI might choose to represent as a table, for example. Every Output may have zero or more other Outputs "nested" within it, to represent structure in the returned items. For example, an Output called "invoice" might have a number of "line item" outputs nested within it.

Output's also have "Attributes", which are a group of arbitrary name/value pairs that help further describe the Output to the client application - for example, the "invoice" Output might have an attribute called "count", which might give the number of line items associated with this invoice.

**Transitions**

A Transition object represents a choice for the client to transition to a new state - only states that are appropriate given the current state and allowed given the current user's permission have action items generated for them.

**Blocks**

For easier use in JSP's (and other environments) where there are a lot of inputs, outputs and transitions, the concept of "blocks" is available to help keep the controller response organized.

A Block object's purpose is to act as a container for other ControllerElement objects such as Transition,
Input, Output and other Block objects. The idea behind a Block object is that it is a logical grouping of other ControllerElement objects for presentation purposes. Think about a HTML page which has multiple sections (not necessarily HTML frames), with each section having its own text messages, buttons/links and forms. by name, which Input, Output and Transition items were to be grouped together. With Block objects, the Controller writer would group each logical set of Input, Output and Transition objects within a Block, with each logical section of the HTML page getting allocated its own Block. The JSP writer in turn, would simple get an Enumeration of all the Blocks from the controller, iterate over each ControllerElement and then create appropriate HTML for each object. Since a Block returns each ControllerElement in the order it was put into the Block, if a JSP writer did not care, the ControllerElements could be placed in the same order that the Controller writer put them in. Of course, the JSP-writer is still free to access each ControllerElement by name as previously and then do a custom HTML presentation.

Here's an example. Let's assume that the HTML page consists of several lines of messages, followed by some links, followed by a form. In some Controller.someState():

```java
//The first logical block
Block b1 = new Block("Header"); // The parameter sets the name of the Block
Output o1 = New Output("Msg1", "This is message 1");
b1.add(o1); //Nest the action within the block
Output o2 = new Output("Msg2", "This is message 2");
b1.add(o2);
Output o3 = New Output("Msg3", "This is message 3");
b1.add(o3);
addBlock(b1); //Add the block to the controller.

// The second logical block
Block b2 = new Block("Links");
Transition a1 = new Transition("Take action 1", "/servlets/ControllerServlet?trx=.....");
a1.setName("Transition");
b2.add(a1);
Transition a2 = new Transition("Take action 2", "/servlets/ControllerServlet?trx=.....");
a2.setName("Transition");
b2.add(a2);
Output o4 = new Output("Msg4", "Please choose one of the links above");
b2.add(o4);
addBlock(b2);

// The third logical block
Block b3 = new Block("Form:Form1");
Output o5 = new Output("Msg5", "Please fill in the information below");
b3.add(o5);
Input i1 = new Input("Name");
i1.setLabel("Name");
i1.setType("text");
b3.add(i1);
Transition a3 = new Transition("Submit", "/servlets/ControllerServlet?controller=.....");
a3.setName("Submit");
b3.add(a3);
addBlock(b3);
```

In a corresponding HTML generator (whether it is another utility class, a bean to be accessed from a JSP, adapter classes for WebMacro/Freemarker, or a JSP itself), one would write something similar to:

```java
String blockName = "....."; // Name as in the controller
for (Enumeration e = controller.getBlock(blockName).getContents().elements();
e.hasMoreElements(); ) {
    ControllerElement oneElement = ( ControllerElement)e.nextElement();
```
if (oneElement instanceof Transition) {
    handleTransition((Transition)oneElement);
} else if (oneElement instanceof Input) {
    handleInput((Input)oneElement);
} else if (oneElement instanceof Output) {
    handleOutput((Output)oneElement);
} else if (oneElement instanceof Block) {
    handleBlock((Block)oneElement);
} else {
    throw new Exception("Cannot handle object of type:
        + oneElement.getClass().getName() + ");
}

One could also use the controller.getBlocks() call to get an enumeration of all the Block objects to iterate over. Note that the exact mechanics of the HTML output generation is not shown here to keep the example simple and to the point.

Types of Controller

There are several types of controllers available, all extending the base class com.javacorporate.common.controller.Controller. They each have a specific use:

- com.javacorporate.expresso.core.controller.Controller

    Controller is the base class for all Controller object, and can be directly extended for. Controllers have no built in security of their own and access to all states must be manually set.

- com.javacorporate.expresso.core.controller.DBController

    DBController extends the base controller and provides integrated security. [Probably a better name would be SecuredController, but due to a legacy issue unknown to the other, DBController it is. ]. When you run DBCreate on a Schema that has DBControllers, a security matrix is generated for each DBController which is checked against before passing the control on to the DBController implementation class itself. Most Expresso-based controllers will derive from DBController since it is usually vital in webapps to have some sort of security system built in.

- com.javacorporate.expresso.core.controller.SecureIfSetController

    The SecureIfSetController provides a compromise between Controller’s insecurity and DBController’s strict security. What happens is that SecureIfSet is available to anybody if there is no security set up in the system. An example of this would be an initial "Setup Wizard" that would allow the admin to decide how to set up the machine graphically before it went live. Once DCBCreate is run and a security system is up and running, then SecureIfSetController performs in a secure way just like DBController, thus allowing the management controllers to be still accessible for Administrators (for example), but not for everyone in the Internet.

Controller Objects Included with Expresso

The following table lists some of the important Controller classes included with Expresso. All of them belong to the com.javacorporate.expresso.services.controller package.

Table 8.1. Controller Objects Included with Expresso

<table>
<thead>
<tr>
<th>Controller Class</th>
<th>Description</th>
</tr>
</thead>
</table>

85
### Controller Activity Explained

<table>
<thead>
<tr>
<th>Controller</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CacheControl</td>
<td>Controls the CacheManager. Instead of controlling the CacheManager directly, this allows a user to clear either one or all of the caches and to display current status of the cache.</td>
</tr>
<tr>
<td>ControllerSecurityMatrix</td>
<td>Provides an easy conduit for controlling access to controllers and the states within the controllers.</td>
</tr>
<tr>
<td>DBMaint</td>
<td>A servlet/controller that enables database maintenance of a specified DBOBJECT.</td>
</tr>
<tr>
<td>DBSecurityMatrix</td>
<td>Controls access to DBOBJECTS by authorized people.</td>
</tr>
<tr>
<td>EditUserPreference</td>
<td>Allows a user to edit his/her preferences either for a single object or all of the user's own preferences.</td>
</tr>
<tr>
<td>ErrorHandler</td>
<td>Displays error to user, logs error and triggers email notification to specified list of users both local and remote.</td>
</tr>
<tr>
<td>JobSecurityMatrix</td>
<td>Allows easy administration of the security rights of user groups to get to certain Jobs and to certain methods in those Jobs (assuming the job has more than one method).</td>
</tr>
<tr>
<td>Log</td>
<td>Allows for log administration including archiving current log, starting a new empty log, and viewing of both the database log and the text log.</td>
</tr>
<tr>
<td>LoginController</td>
<td>Validates login/logouts, establishing and releasing session, and basic interaction with the registration systems</td>
</tr>
<tr>
<td>LoginListener</td>
<td>A skeleton for a developer to extend.</td>
</tr>
<tr>
<td>Navigation</td>
<td>Provides a framework for navigating around an application. It does this by using the schema object's information</td>
</tr>
<tr>
<td>QueueJob</td>
<td>Acts as a traffic cop for any named job that the user has access to queue.</td>
</tr>
<tr>
<td>Registration</td>
<td>Provides a mechanism for self registration of for new system users.</td>
</tr>
<tr>
<td>Status</td>
<td>Shows various information about the servlet systems, the back-end database connections, and server-task tasks.</td>
</tr>
<tr>
<td>ValidationController</td>
<td>Monitors and reacts to the user interaction of the URL being clicked from the validation job.</td>
</tr>
</tbody>
</table>
In the diagram above, you see a simple representation of a Controller with 5 states. Let's follow the execution of this controller to see how the interactions work:

- The Controller begins (let's say it's called from a JSP page for the purpose of our discussion) at the "prompt" state at the top. Entering this state causes the controller to generate an Input item, called "Student Id", and two Transition items. The action items indicate that the user can choose to select a course (Transition 2) or show the calendar (Transition 1) from this point (we're assuming that whatever user is logged in does in fact have permission to get to all of these states). Control then returns to the user interface.

- Let's say the user enters an appropriate Student Id and chooses Transition 2, "select course". The Controller transitions from the "prompt" state to the "select course" state. The "select course" state has a required parameter, the "Student Id" that the previous state prompted for. If this parameter is not present, the transition to the new state will not be successful (and the user should be notified in whatever manner the UI designer chooses - ideally by populating an ErrorCollection object and transitioning back to the previous state so that the user can try their entry again.). In this case, the "Student Id" was present, so the controller enters the "select course" state.

- This state produces another Input item - this time asking the user to choose from a list of courses. The Input item could be supplied with a list of valid values (in this case, the courses available to this student, for example), and might be represented by the UI as a drop-down list. The "select course" state also generates two action items, Transition 3 and Transition 4. If the user did not have permission to access state "enroll", for example, then Transition 3 would not be generated.

- Let's say the user selects a course from a list and selects Transition 4 - to transition to the "display grades" state. In this case, the state "display grades" would require two parameters: the "Student Id"
from the first state and the course selected by the second state. As both of these are present, the transition occurs, and the "display grades" state generates an Output item (probably a group of output items using nesting) that presents the requested information to the user.

This example is of course very simplistic, and a real controller would have many other connections between states, and probably more complex behaviors in each state, but it serves to illustrate the use of the Controller model. It is important to realize that this Controller could actually be used in a number of different ways - for example, from a JSP page that showed a list of students enrolled in a particular course, an authorized user could select a link that said "Show Grades". This link might invoke our example controller, but not at the initial state: It would go directly to the "display grades" state, and assuming that permissions were OK, and that the course and Student Id were supplied (as these are the required parameters to enter this state), our sample controller would respond with the same output that it provided when it was invoked in the way we describe above. This kind of "interaction" with other controllers increases the utility of the Controller model considerably, and promotes re-use without sacrificing portability or data integrity.

Of course, a Controller object typically interacts with one or more database objects (or other controllers) to do it's job - but these interactions are completely concealed within the states themselves, and the client does not need to be aware of them.

**States**

It is the states within a Controller that actually contain the logic for the controller to perform it's functions. These states can be coded simply as methods within the Controller object, or can be their own independent objects, inheriting from the "State" superclass. For simple states, it is often easier to code the state as a single method. The controller will automatically invoke the proper state if a specific naming convention is followed: the state method's name should be "run" followed by the name of the state followed by "State". For example, a state called "prompt" would look for a method called runPromptState. The state's method must take two parameters, a ControllerRequest followed by a ControllerResponse, both of which are passed to the method already initialized. The method must be declared void and should be protected to the Controller class itself. Separate State objects are often superior, as they allow a clean separation between each state.

Each state must be completely independant of any others and "thread safe" - e.g. they should not count on shared elements that are local variables to the Controller. Instead, each state simply works on it's parameters and produces it's own Outputs, Inputs and Transitions. It is common to set parameters in the Transitions between states to pass on information from one state to the next.

**Note**

Using outside classes derived from the State class does not require

**Internal States**

If states are to be coded "internal" to the Controller class, then the state objects are created internally in the constructor of the method. You then define your state handler with a particular naming convention. An example of this is below:

```java
/**
 * Our constructor declares the states this controller supports
 */
public DBSecurityMatrix() {
    State prompt = new State("prompt", "Prompt for Schema and User Group");
    addState(prompt);
}```
setInitialState("prompt");

State dbobjmatrix = new State("dbobjmatrix", "Enter/View Database Object permissions");
dbobjmatrix.addParameter("SchemaClass");
dbobjmatrix.addParameter("GroupName");
addState(dbobjmatrix);
} /* DBSecurityMatrix() */

ControllerResponse runPromptState(ControllerRequest request, ControllerResponse response)
throws ControllerException {
    return response;
}
/* runPromptState(ControllerRequest, ControllerResponse); */

In the code above, the State object is create, and then the "addState" method used to "register" that state as belonging to the controller. The logic for the state, however, is coded into methods built right into the controller object itself.

To do this, you create a special function that follows a particular naming convention. The naming convention is similar to that of the JavaBeans naming convention, but with a twist to help maintain backwards compatibility. Name your function as run<StateName with first letter capitalized>(ControllerRequest, ControllerResponse). If you keep to this convention, then Expresso will automatically find your internal state function through the use of the Java Reflection API's.

Servlet Container Specific Controllers [Expresso 5.3]

Controllers are inherently made to be able to be run outside a servlet environment. Such examples are: webservises, command line programs, and even jobs. But what if you wanted to use servlet specific features? In older versions of Expresso you would have the following code in your internal state handler:

1. ControllerResponse
    runPromptState(ControllerRequest request, ControllerResponse response)
    throws ControllerException {
2. ServletControllerRequest scr = (ServletControllerRequest)request;
3. /** Get HttpServlet Request through: scr.getHttpServletRequest() **/ ..... 4 */
    runPromptState(ControllerRequest, ControllerResponse);
}
/* runPromptState(ControllerRequest, ControllerResponse); */

Since Expresso 5.3, you can now simply change the signature of your state handler to receive ServletControllerRequest directly:

ControllerResponse runPromptState(ServletControllerRequest request, ControllerResponse response)
throws ControllerException {
    /** Get HttpServlet Request through: request.getHttpServletRequest() **/
    ..... 
} /* runPromptState(ControllerRequest, ControllerResponse); */

Expresso will only look for the ServletControllerRequest version of the function if it cannot find a ControllerRequest version of the function. So if you happen to have both, then only the ControllerRequest version will be called.
Note

This feature is not yet available for External States as described below

External States

In a more complex controller, or one with more states, it is often better to create individual "State" objects, rather than the "internal states" methodology discussed above. Again the superclass handles the transitioning to the correct state, dynamically invoking the appropriate State object's "run()" method when each state is entered.

When using "external" state objects, the declaration of the Controller is a bit different:


In the listing above, PromptBrowser and DoBrowser are objects defined external to the Controller class that extend the State object. For example, PromptBrowser could look like this:

```java
public class PromptBrowser extends State {
    public PromptBrowser(String stateName, String descrip) {
        super(stateName, descrip);
    }

    public void run() throws ControllerException {
        Controller myController = getController();
        String currentNumber =
            StringUtil.notNull(myController.getParameter("number")); // here whatever
    } /* run() */
}
```

In the listing above, the PromptBrowser class extends State, defining it as one of the possible states for a Controller, and implements the "run()" method. The "run()" method is called by the superclass of the controller object whenever the Controller transitions into the specified state - it is where all of the work of the state will get done, where the Inputs, Outputs, and Transitions are generated, etc.

In order for a State object to have access to it's Controller's parameters, session, and other information, it has available to it the "getController()" method. This method returns a Controller object - the Controller object that contains this state, specifically. This allows the state to use "getParameter(name)" and other methods to interact with the controller itself, and to use the Controller's "add" methods to return it's Inputs, Outputs and Transitions as required. At the end of the "run()" method, control returns to the Controller superclass, which then processes the new Inputs, Outputs and Transitions appropriately.

As a part of it's processing, a State object often uses one or more DBObjets, and of course it can also submit jobs, access other controllers, and generally perform whatever processing is required of it.

Transitioning
Controller sometimes also need to transition from one state to another programatically - in other words, not in response to a user clicking a button or selecting a link, but in response to logic within a state. This can be done in one of several ways:

**Transitioning in an internal state method**

From a state method within a controller it is very easy to transition to another state in the same Controller. Simply call: transition("newstate", req, res);, where "newstate" is the name of the new state (the state name, not the method name), and req and res are the ControllerRequest and ControllerResponse objects that were passed in to the method originally. It is important also to call "return" immediately after the transition, as you usually don't want to do any further processing in the current state.

**Transitioning in an external State object**

In an external state object, the transition call is identical to the transition method from within a controller. Again, the first argument must be the name of another state within the same controller, and you should still call "return" immediately afterwards.

**Transitioning to another Controller**

Transitioning into another state is done via the Transition object. You prepare a transition object in exactly the same way as you would if you were going to add the transition to the response object for the user to click. For example:

1. Transition t = new Transition();
2. t.setName("goSomewhere");
3. t.addParameter("controller", "com.something.SomeOtherController");
4. t.addParameter("state", "someState");
5. t.addParameter("otherParameter", "etc");
6. t.transition(req, res);
7. return;

In the code above, we prepare a transition, including a reference to another controller "SomeOtherController" and a specific state of that controller. Once the transition is called, we return from the state method.

**Extending Controllers**

Expresso has a very good framework to develop your web applications, however, there are many ways to extend it's boxed functionality to better suit your application's needs.

Expresso includes many pre-built controllers that address many basic functionality needs: Login, Registration, Download, Database Maintenance. However, often we need to modify the capabilities of these controllers to better fit our own needs. Specifically we'll talk about providing custom User Interfaces easily and how to change the default behavior of a controller.

**Background information**

As of Expresso 5.0, developers can easily derive their own controllers from existing controllers. The mechanics of this is a rewrite of the reflection mechanism that Expresso controllers utilize to call various states. Now if you do not override any particular state, Expresso's controller walks up the inheritance hierarchy looking for other classes that implement the requested state. If an object in the hierarchy is found to implement the state, it will invoke that state and return. In the past, if your own controller didn't override EVERY state, Expresso's reflection mechanism would fail.
Now, you have the ability to derive your own controller and ONLY override the states where you want to change the default behavior of the base controller.

Extending the User Interface

One of the annoying aspects of dealing with frameworks like Expresso is the fact that normally to change the user interface, you’d have to change the default jsp pages that ship with Expresso. Although this is fine initially, with each release of Expresso, you would have to re-merge your changes. This problem is exacerbated by anybody working with a CVS copy, where CVS will mark any changed files and try to merge any updates, thus preventing any effective development with CVS snapshots.

The solution is to create your own controller, and then create your own JSP pages based upon this new controller.

Step 1 - Derive your own Controller Class

The first step is to derive your own controller. We will use Expresso's own login controller as a code example. Here's all that is necessary to create your own controller.

```java
package com.jcorporate.myproject.controller;
import com.jcorporate.expresso.core.controller.Controller;
import com.jcorporate.expresso.core.controller.ControllerException;
public class MyLoginController extends com.jcorporate.expresso.services.controller.LoginController {
    public MyLoginController () {
        super();
        setSchema(com.jcorporate.myproject.MyProjectSchema.class);
    }
    public String getTitle() { return "Sample Controller Login"; }
}
```

All state handlers use the base class, and since, for this example, we want the default behavior, we won't be changing anything there.

Step2 - Make your own struts-config file entries

Now that you have your own controller, you must create your own configuration file so that the underlying controller/Struts framework can be aware of your new controller and what files/servlets should do the rendering of your controller's output. Here's myproject-config.xml:

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE struts-config PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 1.0//EN"
    "http://jakarta.apache.org/struts/dtds/struts-config_1_0.dtd">
<struts-config>
    <action-mappings>
        <action path="/MyAppLogin"
            type="com.jcorporate.myproject.controller.MyLoginController"
            name="default" scope="request" validate="false">
            <!-- This sample only changes the promptLogin screen -->
            <forward name="promptLogin"
                path="/expresso/components/myapp/login.jsp"/>
            <forward name="processLogin"
                path="/expresso/jsp/register/redirect.jsp"/>
            <forward name="processLogout"
                path="/expresso/jsp/register/logout.jsp"/>
            <forward name="promptChangePassword"
```
Notice that we've changed the promptLogin state to forward to a different JSP, assuming that your app is conforming to standard layout of an Expresso-based component. This configuration file is placed in your Expresso's config directory and will be automatically picked up by the configuration system.

Now, as your app progresses, you can one by one change the state to forward to jsps that you have worked up with your own application's look and feel. And you haven't modified the default pages one iota!

**Step 3- Copy Message Resources [Pre Expresso 5.0.4 Only]**

**Note**

Expresso 5.0.4 has a new mechanism for finding messages that makes this step unnecessary. Simply write in your own MessagesBundle any keys that you wish to 'overwrite' and any messages you wish to add, and you can skip the rest of this step.

If the controller you are deriving from is hooked into a different schema, you will want to find the messages that it uses and add them to your own schema. If you don't then the superclass controller will attempt to look in your MessagesBundle.properties and will throw an exception if it can't find what it's looking for. If the superclass controller relies heavily on Messages (which as time goes on, most controllers will rely more and more on Messages) this portion can be tedious at best, and we are working on a better resolution for this.

**Step 4 - Login Controller Only**

There are various parts of Expresso that need to know where to send the login requests. So to accomplish this, we need to define a Class Handler for login controller. Note that this step is also only necessary if you're using the <loginTag> if you aren't, or you aren't dealing with LoginController, you can skip this step.

To do this, you need to add a class handler entry in your expresso-config.xml file. Insert the code outlined below RIGHT before your first <context> entry.

```
<class-handler
  name="login"
  classHandler="com.jcorporate.myproject.controller.MyLoginController"/>
```

**Extending the default controller behaviors.**

Any object oriented course teaches that you should be able to derive an object and override it's behavior.
Expresso Controllers are no different. State handlers in Expresso should be of protected visibility, so you should be able to override any state handler to perform what you wish. We have a program listing that shows that kind of behavior:

```java
package com.jcorporate.myproject.controller;
import com.jcorporate.expresso.core.controller.*;
public class MyLoginController
    extends com.jcorporate.expresso.services.controller.LoginController {
    public MyLoginController () {
        super();
    }
    public String getTitle() {
        return "Sample Controller Login";
    }
    /**
     * Overridden getRegistraitonController()
     */
    public Controller getRegistrationController() throws ControllerException {
        return ConfigManager.getControllerFactory().
            getController(MyRegistrationController.class.getName());
    }
    /**
     * Overridden state that simply adds an additional output to the response.
     */
    protected void runPromptLoginState(ControllerRequest request,
        ControllerResponse response) throws ControllerException {
        super.runPromptLoginState(request,response);
        response.add(new Output("myaddition","This is my added text message");
    }
}
```

Now what you see in the above code, we’ve overridden the runPromptLogin state in such a way as the normal promptLoginState is run, but we add our own Outputs after the fact. Oftentimes this can be the easiest way to accomplish a simple extension since in your jsp page layout you can always not render any ControllerElements (Inputs, Outputs, Transitions, Blocks) that you don’t want to see on the screen. If, however, you, wish for more radical differences in your own work, you can always omit the call to super() and perform whatever behavior you need.

So what about the functions getRegistrationController() and getLoginController()?? Well, it turns out that LoginController is a strange beast compared to a normal controller. The reason being is that Login and Registration are fairly tightly coupled classes at this point in time. If you derive your own Registration class, for example. Login will need to know what Class to forward registration requests to. By overriding this method, we're now causing our controller’s links to point to our own derived Registration Controller. Most other controllers will not need this done.

**Workflow**

Expresso supports chaining states together in a "wizard" or "process funnel", also known as workflow. See the workflow description at the JCorporate site. [http://www.jcorporate.com/econtent/Content.do?state=resource&resource=779].

**Conclusion**

**Contributors**
The following persons have contributed their time to this chapter:

- Larry Hamel (JGroup Expert)
- Mike Rimov
- Mike Traum (JGroup Expert)
- Tino Dai
- Sandra Cann

**Note**

Was this EDG documentation helpful? Do you wish to express your appreciation for the time expended over years developing the EDG doc? We now accept and appreciate monetary donations. Your support will keep the EDG doc alive and current. Please click the Donate button and enter ANY amount you think the EDG doc is worth. In appreciation of a $35+ donation, we'll give you a subscription service by emailing you notifications of doc updates; and donations $75+ will also receive an Expresso T-shirt. All online donation forms are SSL secured and payment can be made via Credit Card or your Paypal account. Thank you in advance.


Copyright © 2001-2004 Jcorporate Ltd. All rights reserved.
Chapter 9. Database Maintenance

Note

If you find this EDG documentation helpful please consider DONATING! to keep the doc alive and current.

Version: Expresso 5.5

Maintainer: Geeta Ramani
[mailto:geeta.ramani@cmpco.com?Subject=EDG]

Expresso includes a powerful controller object for database maintenance. This controller can present add, update, search and delete options to authorized user without having to write any code beyond the underlying database object (or not even that, when using “AutoDBObject” classes).

This eliminates the need to create custom table maintenance code in your application, yet still retains complete flexibility of presentation, as we will see.

There are many examples of the use of the database maintenance Controller in Expresso itself.

The DBMaint controller is designed to allow easy extensibility. This is done by making each of the commands that DBMaint responds to an extensible object in itself. These objects are stored in the com.jcorporate.services.controller.dbmaint package, and may be extended for custom functionality if required.

Using DBMaint

In order to use DBMaint you first must define a secured database object (SecuredDBObject). This is in practice very simple:

- Decide which package your new object will reside in. It should not be within the directories used by Expresso. We recommend a package name of the form "com.javacorporate.ext.application.dbobj", where "application" is the application you are working to create - for example, if it is a Human Resources application, you might choose "hr", so the package name would be "com.javacorporate.ext.hr.dbobj". The object name should reflect it's intended purpose: let's say you are creating an object to maintain an Employee database, you might name it "Employee".

- Copy an existing SecuredDBObject definition out of Expresso to use as a template - saves some typing! We recommend the object com.javacorporate.common.dbobj.User as a nice simple example.

- Modify the copied object, replacing all instances of the name "User" with the name "Employee", modifying the package definition, and of course the target table and field definitions to match the table you wish to have in your database. See the JavaDoc documentation for the SecuredDBObject object for details.

- You may at this point decide to manually create the table to hold the Employee object - there is a way to have Expresso do this for you, discussed in the section on Schema objects.

- Compile your new object so that it's class file is available.

Now you create a link to your new database object using the DBMaint servlet:
• Select the web page you are going to call the maintenance function from: Let's assume it's called "hr.html".

• On this page, create a link for the List, Add, and Search functions for your new object - you may want to use the images and format that are used within Expresso - these can be copied from a page such as "server.html" in the /components/expresso directory of the HTML download.

• Each link should have the following form: "/DB-Maint.do?dbobj=com.javacorporate.ext.hr.dbobj.Employee&state=Add"

• The last parameter (cmd) can have one of three values:
  • Add: Bring up a full-screen form to accept a new entry for this database object
  • List: List all existing records database object
  • Search: Bring up a form to accept search criteria for locating one or more records for this object

  Of course, the List option must be used with caution if the number of records is likely to get large (there is a facility to limit the number of records shown and to provide pagination, however). There is also an Update argument, see the JavaDoc for the DBMaint object for details.

• Create a database object security entry for your new object: See the help file for "Database Object Security" for information on how to do this.

That's it - you now have a fully functional, secured database maintenance servlet.

**Using DBMaint**

Once you have coded your SecuredDBObject, you can immediately take advantage of the "Automatic" database maintenance capability of Expresso - the DBMaint Controller. This Controller allows you to provide to your users, Add, Update, Delete and Search capabilities for any SecuredDBObject, without any coding at all!

The servlet works on any SecuredDBObject, specified as a parameter, and can perform any of a number of different commands. For example:


will list all records (subject to a page size specified in Expresso and discussed later) in the table corresponding to the Customer DBObjec. (Assuming the user issuing the request has Search permission on Customer.)

The list of options for "state" (which is extensible by coding new objects for the com.jcorporate.expresso.services.controller.dbmaint package) includes:

• **Search**: Present a query form for the user to enter search criteria to look up records.

• **Add**: Present a blank form for the user to enter a new record.

• **Update**: (requires "key" parameter as well) Presents an existing record for the user to update.

• **List**: Lists the records (or the current query results) and allows the user to select a record for update.
SearchList: Given special parameters for a given key, display all records that match the given criteria. You do this by matching the parameter field names to the DBObject field. For example, if you add the parameters &id=10 to the parameter, then if you had a field named id, it would set that field name to 10 and perform a search against the database.

A classic example of this is displaying all setup items for your schema:

```
/DBMaint.do?dbobj=com.jcorporate.expresso.services.dbobj.Setup
&SchemaClass=com.jcorporate.expresso.core.ExpressoSchema&state=SearchList
```

The URL would normally appear all on one line. The results would be a listing of all setup codes belonging to the Expresso Schema. Substitute the SchemaClass parameter value for your own schemas to have the same result.

All of the forms present icons for the user to move from mode to mode, making DBMaint a complete database maintenance application for your DBObject.

Expresso's own administrative pages provide good examples of the use of DBMaint. See the Expresso Users Guide for details of DBMaint's operation.

DBMaint and Defineable Objects

As introduced in the earlier chapters on DBObjects, as of Expresso 5.1, DBMaint is designed to cope with any class implementing the DataObject interfaces. Some of these objects such as JoinedDataObject and AutoDBObject also need other parameters to properly instantiate other than the class name. These classes implement the `com.jcorporate.expresso.core.dataobjects.Definable` interface. The way to get the name of the definition to the Defineable object is added the `definition` parameter. An example is:

```
/DBMaint.do?dbobj=[objectname]&definition=[definitionName]&state=List
```

The actual value to insert as the definition name varies depending on the object. For example, AutoDBObject uses database table names for definition names, JoinedDataObject uses the path to the xml file describing the join. Check the javadocs for the class you wish DBMaint to use for information on using the definition.

A Step-By-Step Example

This is an example of building a simple application to allow maintenance of a customer database, complete with security. The application will use a customer maintenance form as a beginning for a more sophisticated sales tracking application.

Directory Structure

A word first about directory structure. Assume that the root of the Expresso directory is at $Expresso_dir. So for example, under this directory would be folders like "expresso" (containing the html/jsp pages for the Expresso application itself like "toc.jsp" and "frame.jsp", an "images" folder and so on.) $Expresso_dir would also contain the WEB-INF folder, within which you should have the usual "classes" folder, the "lib" folder and so on. Drilling down into the classes folder, you should see "com/jcorporate/expresso" with further subfolders. These, along with the lib archives, hold most (or all?) the classes that do the main work in setting up the Expresso framework.

Classes
The classes for the sales application that we will build will belong to a package called com.jcorporate.ext.sales.dbobj. So first create a directory called "ext" in $Expresso-dir/WEB-INF/classes/com/jcorporate. Then within "ext", create a directory called "sales" and within "sales" yet another directory called "dbobj". The (two) classes that we write for this application will be placed in this directory: $Expresso_dir/WEB-INF/classes/com/jcorporate/ext/sales/dbobj.

**jsps**

The jsps for the sales application will be held in $Expresso-dir/expresso/components/sales. If you follow this convention, when we are done, you will get an clickable icon for the Sales Application on your Expresso homepage left-hand nav bar. The jsps needed for maintaining the Sales database will be held in $Expresso-dir/expresso/components/sales/dbmaint/.

You may also at this time want to look inside the folder $Expresso-dir/expresso/. The jsps and html pages for the apps that Expresso itself uses (like Login, Dbmaint, etc.) are stored in this directory. For example the dbmaint jsps that Expresso uses are in the subdirectory $Expresso-dir/expresso/jsp/dbmaint. Further, jsps for eforum, ePoll, etc. are held in $Expresso-dir/expresso/components/eforum and $Expresso-dir/expresso/components/epoll.

**Note**

You could of course choose to not follow this directory structure. The important thing to know is that the classes you write should be within the WEB-INF/classes folder (in some logical order) because we want to stay within the same context as is Expresso. For a detailed discussion about this, read the thread entitled "EXPRESSO DEVELOPER'S GUIDE CHAPTER ZERO" in the "Expresso discussions about features, installation, direction & ideas" group of the Expresso forum.

Also note that should you change your directory structure, you will have to appropriately change the code below as well as the name of the Schema class in step 6 below.

**Example Step-by-Step**

1. **Define a database object, "Customer" by creating a Customer.java as follows:**

   Let's assume for the sake of this example that you've installed Expresso in /usr/java/lib/com/jcorporate. Your CLASSPATH would then have to include /usr/java/lib. (Of course, the CLASSPATH for your servlet engine also has to include this directory - see the installation section for details).

   A. First we create the directory /usr/java/lib/com/jcorporate/ext, where we are going to define "extensions" to the Jcorporate Expresso framework. The package name could just as easily be outside the jcorporate directory if so desired.

   B. Within the "ext" directory, we create a directory for our package, called "sales" (e.g. we create /usr/java/lib/com/jcorporate/expresso/ext/sales). Inside this we create another directory, called "dbobj" to hold our database objects.

   This long directory hierarchy is fairly typical to Java applications and helps keep packages well organized - again, we are only using these particular names in this example, if you want a different organization, that's OK too.

   C. Within the ext/dbobj directory, we create our database object, called "Customer.java". The following code defines our object:
The comments within the source help describe the purpose of each section in the Code for Customer.java:

```java
package com.jcorporate.ext.sales.dbobj;
import com.jcorporate.expresso.core.dbobj.*;
import com.jcorporate.expresso.core.db.DBConnection;
import com.jcorporate.expresso.core.db.DBException;

/**
 * A Customer object stores information about customers for our demo
 * sales application.
 */
public class Customer extends SecuredDBObject {
    private String thisClass = (this.getClass().getName() + ".");

    /**
     * There are three possible constructors for SecuredDBObjects
     * Usually there is no need to extend them for objects we implement
     */
    public Customer () throws DBException {
        super();
    } /* Customer() */

    /**
     * This constructor is used when a connection to the database
     * is supplied by the calling object
     */
    public Customer(DBConnection theConnection) throws DBException {
        super(theConnection);
    } /* User(DBConnection) */

    /**
     * This constructor is called by the DBMaintServlet
     * and supplies the name of the user trying to connect to the
     * database object.
     */
    public Customer(DBConnection theConnection, String theUser) throws DBException {
        super(theConnection, theUser);
    } /* Customer(DBConnection, String) */

    /**
     * The setupFields method does the real work of establishing the
     * definition of the DB object.
     */
    public void setupFields() throws DBException {
        /* Establish the primary database table for this object. */
        /* Note that there may be more than one, but this is the default */
        /* table */
        setTargetTable("CUSTOMER");

        /* Set a description for this object. This is a human-readable */
        /* string that appears at the top of forms when used with DBMaint */
        setDescription("Customers");
    }
}
```
Database Maintenance

/* Define each of the fields in the table */
/* Note that the field types used here are "internal", and can be */
/* mapped to other types for the underlying relational database */
addField("CustomerNumber","int", 0, false, "Customer ID Number");
addField("CustomerName","varchar", 80, false, "Customer Name");
addField("Phone", "varchar", 15, true, "Phone Number");
addField("EMail", "varchar", 80, false, "Customer EMail");

/* ... you could of course add many more fields for a production */
/* object */
/* addKey is called for each field in the primary key */
addKey("CustomerNumber");

} /* setupFields() */

/**
* A utility method used when generating lists of objects
*/
public DBObj getThisDBObj() throws DBException {
    return (Customer) new Customer();
} /* getThisDBObj() */

} /* Customer */

2. **Now compile the Customer.java object and correct any errors.**

If you have trouble compiling, remember that the CLASSPATH would have to include
$Expresso-dir/WEB-INF/classes, the archives struts.jar, junit.jar, activation.jar, and log4j-core.jar
in the $Expresso-dir/lib directory, as well as /common/lib/servlet.jar in the expresso root directory.
(If you did not install the -complete bundle, you may need to download some of the jar files needed
and/or find their exact location in your set-up.)

3. **Create a Schema object "Sales" by creating a Sales.java:**

The Schema object identifies the whole application and it's required DBObjects to the Expresso
framework. It enables you to use Expresso's automatic facilities for creating tables and configuration
values to your advantage, saving work when setting up your application.

In the same "dbobj" directory, create a java file "Sales.java" as below. This simple Schema would be extended as you add new objects to your new application.

**Code for Sales.java:**

```java
/**
* Schema object for the Sales demo application
*/
package com.jcorporate.ext.sales.dbobj;
import com.jcorporate.expresso.core.dbobj.*;
import com.jcorporate.expresso.core.db.DBException;

public class Sales extends Schema {
    /**
*/
```
* Default Constructor
*/
public Sales() throws DBException {
    super();
    /* add is called for each object to be a "member" of this schema */
    addDBObject("com.jcorporate.ext.sales.dbobj.Customer");
} /* constructor */

/** *
* Return the path, relative to the classpath, of the MessageBundle file for
* this schema. For example, the Expresso schema
* (com.jcorporate.expresso.core.ExpressoSchema) returns
* "com/jcorporate/expresso/core", as this is where it's message files are
* @return */
public String getMessageBundlePath(){
    return ("com/jcorporate/ext/sales/dbobj");
} /* getMessageBundlePath() */

public String getDefaultDescription(){
    return ("Example Sales Schema");
}

public String getDefaultComponentCode(){
    /* The framework finds the jsp pages here (within the "components" subdirectory) */
    return ("sales");
} /* Other methods are optional in the Schema object - see the Javadoc */
/* for details */
} /* End Sales Schema*/

4. Again, compile the Sales.java object and correct any errors.

5. Next, we create and configure the jsp's for this application.

   There are three things you need to do.

   - First create a subdirectory called "sales" in $Expresso-dir/expresso/components/ and save frame.jsp as below in it:

     **Code for frame.jsp:**

     

```jsp
<%@ page language="java" %>
<%@ page import="com.jcorporate.expresso.core.misc.StringUtil" %>
<%@ page import="com.jcorporate.expresso.core.misc.ConfigManager" %>
<%@ page import="com.jcorporate.expresso.services.dbobj.Setup" %>
<%@ page import="com.jcorporate.expresso.core.db.DBException" %>
<%@ page import="com.jcorporate.expresso.core.jsdkapi.GenericSession" %>
<%@ page import="com.jcorporate.expresso.core.servlet.CheckLogin" %>
```
<html>
<head>
<meta http-equiv="cache-control" content="no-cache">
<meta http-equiv="expires" content="0">
<meta name="pragma" CONTENT="no-cache">
<title>Sales App Home Page</title>
</head>
<body bgcolor="WHITE" link="#0000cc" vlink="#0000cc">
<% String db = "";
    db = GenericSession.getAttributeString(request, "db");
    if (db == null || db.equals("")) {
        db = "default";
    }
%>
String contextPath = StringUtil.notNull(Setup.getValue(db, "ContextPath"));%
<%> 
<center>
<table border=1 cellpadding="4" cellspacing="0">
<tr><th>Maintain the Customer table in the Sales database</th></tr>
<tr><td>
<a href="<%= contextPath %>/DBMaintCustomer.do?dbobj=com.jcorporate.ext.sales.dbobj.Customer&state=Add">Add a Customer!</a></td></tr>
<tr><td>
<a href="<%= contextPath %>/DBMaintCustomer.do?back=&fields=&dbobj=com.jcorporate.ext.sales.dbobj.Customer&state=List">Manage Customers!</a></td></tr>
<tr><td>
</table>
</center>
</body>
</html>

The main thing to notice here is the structure of the links: For example, reference to the link
"<%= contextPath %>/DBMaintCustomer.do?dbobj=com.jcorporate.ext.sales.dbobj.Customer&state=Add" says that the DBMaintCustomer is being invoked to act on the dbobject called "Customer" with a state of "Add". Further on, you will see that we will edit the strus-config.xml file to say that when we invoke DBMaintCustomer, we actually mean DBMaint (the "type" attribute for the "action" DBMaintCustomer will explicitly point to the class "com.jcorporate.expresso.services.controller.DBMaint"). The source of this class will tell you that the "states" which are allowed include "Add", "List", "Search", etc. - which is why this link as shown works.

- Second, create a subdirectory "dbmaint" in $Expresso-dir/expresso/components/sales.

(Refer to the note above regarding directory structure.) Copy "update.jsp", "search.jsp", "list.jsp", "add.jsp" and header.inc and tabs.inc from $Expresso_dir/expresso/jsp/dbmaint/ into $Expresso-dir/expresso/components/sales/dbmaint.

- Finally, you have to edit the struts-config.xml in the $Expresso-dir/config directory.
(Make a backup first!!) Scroll down to the "Action Mapping Definitions" portion and add the following below the "Database Maintenance" portion for the "DbMaint" (You could just cut and paste the DBMaint portion and then edit the path values):

```xml
<!-- Customer Database Maintainance for Sales Application -->
<action path="/DBMaintCustomer"
  type="com.jcorporate.expresso.services.controller.DBMaint"
  name="default"
  scope="request"
  validate="false">
  <forward name="Update" path="/expresso/components/sales/dbmaint/update.jsp" />
  <forward name="Search" path="/expresso/components/sales/dbmaint/search.jsp" />
  <forward name="SearchList" path="/expresso/components/sales/dbmaint/list.jsp" />
  <forward name="List" path="/expresso/components/sales/dbmaint/list.jsp" />
  <forward name="Add" path="/expresso/components/sales/dbmaint/add.jsp" />
</action>
<!-- End Customer Database Maintainance for Sales Application -->
```

6. **Reload struts-config.xml**

   Just go to: http://<hostname>:<port>/expresso/admin/reload.do (Expresso will out-do itself by displaying a page which says "OK". :-)

7. **Register your new Schema as follows:**

   Now you must tell the Expresso framework about the existence of your new schema, so that the Initialize function can correctly access your Schema object as follows:

   - In your web browser, go to the Expresso pages that you downloaded from the jcorporate site (i.e. yourserver/expresso/frame.jsp).
   - Click the "Applications" link in the left-hand column. Scroll down to the bottom of the page.
   - On the line entitled "Application Schema Objects" click the "plus" sign to add a new entry.
   - For the "Schema Class File" field, enter "com.jcorporate.ext.sales.dbobj.Sales", the name of the object for the Schema we just created. Careful with case/spelling/path!! Mistakes are easily made here and Expresso will not notify you - things will just not work!! Further, if you chose a different directory structure fom the one mentioned in this document, your schema class file should reflect that change.
   - For the "Schema Description" field, enter "Sales Application". For component code, enter "sales"

8. **Run the DBCreate servlet as follows:**

   This step will create the database table for your new (Customer) object, and also insert default security entries so that the "Admin" group (and the Admin user) will have full access to the newly created table.

   - Go back to your expresso/frame.jsp in your browser window.
   - Click the "Setup" link in the left-hand column.
• Click on "Create/Verify Database Structure and Perform Initial Setup". On the resulting page, leave all boxes checked and click the "Run" button.

• In a moment or two, you will see output confirming that the Customer table has been initialized and that security entries have been created.

• When you are done admiring the messages on the screen, click on "Setup" in the left navigational frame.

9. **Log In as Admin**

   Click on the "Login" icon on top right of the page to log yourself in as Admin using a user name and password of "Admin" and "" respectively. As your new Schema object is only accessible as Admin at this point, that is who you must be in order to test it out. Later of course you can add security for other groups/users (described in the next step).

10. **Secure the Sales Schema as follows:**

   • Click on the "Security" link on the left hand navigational frame. Then click on the "Administer Database Object Security" link in the table that appears. If all the steps above went through correctly, you should see "Sales Application" as well as "General" against the dropdown list labelled "Choose Schema". If you don't see the Sales schema listed, you need to recheck your work so far because none of the subsequent steps will work.

   • Choose the "Sales Application" schema, and click on "Set Security".

   • The resulting page should have a table with exactly one entry: Customers. Check all the boxes and click on "Update".

11. **Finally check the application as follows:**

   • Bring up your Expresso Home Page (http://<hostname>:<port>/expresso/frame.jsp) on your browser window. If you followed the suggested directory structure, you should see a link on your Expresso pages (frame.jsp) to the Sales application labelled "Sales Application". (You may have to reload/restart the servlet engine!.) You may of course also directly access the Sales App with a uri which is something like: http://www.yourcompany.com/expresso/components/sales/frame.jsp or http://www.yourcompany.com:8080/expresso/components/sales/frame.jsp.

   • Now pick your favourite administrative task, and from here on life's a bed of roses!

**Note**

When you add columns to the Customer table be careful to follow the table column data types: the error messages that appear if you don't are not particularly helpful!). The following four-tuple, for example, should work: 1234, Geeta Ramani, 800-123-4567 and geeta.ramani@cmpco.com.

12. **This completes the DBMaint-Step-by-Step example.**

   You can extend it as follows:

   • Create more tables by writing appropriate dbObject classes (one for each table you want cre-
• Then add these classes to the $Expresso_dir/WEB-INF/classes/com/jcorporate/ext/sales/dbobj
folder.

• Rewrite the schema Sales.java so that all these objects are added in the constructor.

• Recompile everything.

• Then go to the "Applications" link, scroll down to the bottom and click on the "Application
Schema Objects" link. You should see a page listing the Sales schema that we had created earli-
er), Click on it and in the subsequent page click on "Delete". This "delete" step is most probably
NOT neccessary, but I haven't tested it.

• You can now carry out steps 7 through 11 and you should be done!

13. **A Final Note.**

Of course you should be able to give the jsps whatever look and feel you want by editing the ap-
propriate "views" in the /sales/dbmaint folder. A biggie here is to just modify the default.css
stylesheets. You can get some amazing results just by tweaking here. This way, you automatically
get the color scheme that fits your application. If you don't care about changing the "default" look
and feel that Expresso has already provided, you may cut short your work by replacing Step 5
above as follows:

• Change the links in $Expresso-dir/expresso/components/sales/frame.jsp to point instead to DB-
Maint.do?dbobj=com.jcorporate.ext.sales.dbobj.Customer&state=Add, etc.

• You don't need to create any new jsps.

• Don't bother to edit struts-config.xml.

Carry on with Steps 7 through 11 and you should be done.

**Conclusion**

**Contributors**

The following persons have contributed their time to this chapter:

• Geeta Ramani

• Mike Rimov

• Mike Traum (JGroup Expert)

**Note**

Was this EDG documentation helpful? Do you wish to express your appreciation for the time
expended over years developing the EDG doc? We now accept and appreciate monetary dona-
tions. Your support will keep the EDG doc alive and current. Please click the Donate button
and enter ANY amount you think the EDG doc is worth. In appreciation of a $35+ donation, we'll give you a subscription service by emailing you notifications of doc updates; and donations $75+ will also receive an Expresso T-shirt. All online donation forms are SSL secured and payment can be made via Credit Card or your Paypal account. Thank you in advance.

[http://www.jcorporate.com/edgdoc.html]

Copyright © 2001-2004 Jcorporate Ltd. All rights reserved.
Chapter 10. Database Connection Pooling

Note

If you find this EDG documentation helpful please consider DONATING! to keep the doc alive and current.

Version: Expresso 5.5
Maintainer: David Lloyd
mailto:dlloyd@jgroup.net?Subject=EDG

Establishing and cleaning up connections to the database can be very time-consuming to an application, slowing it's performance. Instead of creating new connections as needed, it is better to use an existing pool of connections that are held open and available at all times. Expresso provides facilities for managing database connection pools, abstracting the connection process even further than the JDBC API. A sophisticated connection pool object provides access to one or more databases in an efficient manner.

Introduction

Expresso offers a fully Java package for handling multiple concurrent database connections for handing out and returning database connections from a configurable pool of connections. This completely avoids the overhead required in establishing a new database connection, typically around 1 to 2 seconds, by reusing a collection of pre-established connections. This profoundly improves the performance for database-intensive applications where HTML pages and forms are either stored directly in a database or created on-the-fly from data or procedures stored in a database.

Expresso's connection pooling handles advanced features such as: minimum and maximum pool size, connection timeouts (with override for long running tasks), unique connection ID's and tracing, multiple database capability, even across different database types, and the ability to use virtually any JDBC driver without any coding required.

Note

Scales well! Handles things like connections not being released from servlets on a timely basis, maximum numbers of connections, and a few other things you may run into when you're scaling to support larger groups of users.

Why Is a Database Connection Pool Used?

One of the advantages over CGI is that a Servlet can keep information between requests and share common resources. One common use of this feature is a database connection pool.

Connection pooling is a technique used to avoid the overhead of making a new database connection every time an application or server object requires access to a database. A dynamic web site often generates HTML pages from information stored in a database. Each request for a page results in a database access. The database access itself is not the bottleneck, but setting up a new connection for each request often is. A database connection pool avoids this bottleneck.

The overhead time for establishing a database connection is typically around 1 to 3 seconds. This is the
time it takes to locate the database server, establish a communication channel with it, exchange information. For many web applications the database connect time can become the dominant factor in its usability, especially if it is used over the internet versus a corporate network.

Expresso is scalable; it uses a smart connection pool that holds connections open, and will open as many as it either needs or is allowed (e.g. you set a max) and hand them out to separate invocations of the same servlet when many hits arrive at once.

EJB Server Connection Pools

Jcorporate's connection pooling can work with the connection pooling provided by Enterprise JavaBean application servers, if the database objects that make up the system being implemented are created as entity EJB's. The EJB server's connection pooling then handles the pooling of connections for the entity EJB's, and the application framework's connection pool is used for all other database accesses, ensuring the entire system maintains high performance.

Expresso Implementation

With security handled at the application level with the Expresso Framework and other components, it is practical to use "connection pooling" e.g. holding a number of connections open even after they have been used in order to supply them to another process that required a connection.

It's something like holding a number of phone lines open rather than dialing every time a connection is required! This facility is automatically provided by the framework's connection pool, which any program in the system can access. In addition, database objects can request that their data be cached, for even faster read access - this is particularly helpful for objects such as the configuration values, which might be read many times without being written or changed. These two facilities ensure that the application framework scales up well.

Database and Application Server Independent

You are not tied to any one database or application server company which means you can make changes to the software being utilized on your server as needed.

Jcorporate connection pooling implements a sophisticated database-independent type of connection pool that can handle multiple simultaneous databases and database connection pooling with sophisticated management.

While any database that can be accessed via JDBC can be used, a number of popular database engines have been tested with the system - see the system requirements documentation for recommendations of database engines to use with the system.

DB Pooling Management

Database connections are expensive in terms of performance to make & re-make, so rather than making and breaking connections as required, a "pool" of database connections is maintained by the system on the server. When a servlet needs a database connection, it simply requests an available one from the pool - if none is available, a new one is created & added to the pool.

The connection pool not only grows to specified limits, but also contracts as required, closing connections that have not been used for a specified time. This avoids taking up system resources on simply holding connections that are not currently required. This also handles databases which "time-out" their connections, and prevents handing a "stale" connection to an application object. Status servlets allow the state of the connection pool to be viewed at any time, and every connection has a description that identi-
fies it's current and last use, as well as the last time it was accessed.

Setup variables allow a maximum size to be set for the pool in order to limit the amount of traffic to the database if required.

Connection pooling means that a single database login & password should be allocated for access by the entire system - security is handled as described below, so this one single login/password should have full access to the database. Individual users are typically granted no access to the database at all, eliminating any security issues that might arise from allowing users access to the database directly (e.g. via ODBC).

**Queued Connections**

Connections can also be "queued" - if there is a specified maximum number of connections to the database (due to limitations on either the database itself or for performance reasons), then objects requesting a connection can be "put on hold" for a few moments until another object releases its connection. The "waiting time" can also be configured to avoid unacceptable delays. In addition, background and low-priority jobs can be suspended or queued when interactive requests are waiting for connections.

Database object security means that maintaining multiple connections for multiple users is not required, further reducing the demands on the system and improving overall performance.

**Database Connections**

Most servlets in the system utilize a connection to one or more databases. These databases may be one of a number of different types, and are connected via the JDBC API and a suitable JDBC driver.

Each database context also has a database connection pool initialized for it at startup time. No connections are established immediately, but the pool grows (to a maximum size specified in the Setup values) as objects request access to the database. The connection information for the pool is stored in the expresso-config.xml file.

**Conclusion**

**Contributors**

The following persons have contributed their time to this chapter:

- Sandra Cann

**Note**

Was this EDG documentation helpful? Do you wish to express your appreciation for the time expended over years developing the EDG doc? We now accept and appreciate monetary donations. Your support will keep the EDG doc alive and current. Please click the Donate button and enter ANY amount you think the EDG doc is worth. In appreciation of a $35+ donation, we'll give you a subscription service by emailing you notifications of doc updates; and donations $75+ will also receive an Expresso T-shirt. All online donation forms are SSL secured and payment can be made via Credit Card or your Paypal account. Thank you in advance.

[http://www.jcorporate.com/edgdoc.html]
Copyright © 2001-2004 Jcorporate Ltd. All rights reserved.
Chapter 11. Developing The View in the MVC

Note
If you find this EDG documentation helpful please consider DONATING! to keep the doc alive and current.

Version: Expresso 5.5
Maintainer: Mike Traum
[mailto:mtraum@jgroup.net?Subject=EDG]

Introduction

Expresso relies heavily on the Struts MVC framework for its MVC architecture. As such it has quite a bit of view independence. Although we will be primarily focusing on JSP development, the first part of this chapter would apply to other developers who wish to use development systems such as FreeMarker or Velocity. For a specific chapter on how Struts integrates with Velocity, the book "Struts In Action" by Ted Husted et al. is recommended.

Order of execution to a request.

This is an important aspect to understand how your system gets to the jsp page. This is the order of execution when a web browser makes a request to the application server:

- **Web Browser Request** The web browser fetches a URL. Let's say for the sake of this example, the url is: http://localhost:8080/Example.do

- **Servlet Engine Routing** The application server looks at the request URL and sees that any request URL ending in .do should be sent to the ExpressoActionServlet (which is slightly modified from the Struts ActionServlet but for all purposes of this discussion, the two are equal). So the request is sent to the ActionServlet.

- **ActionServlet Routing** Now the ActionServlet takes a look in its own configuration files and looks for any matches to /Example.do. If it fails to find one, it will return an error to the web browser about unable to locate resource. Once it finds the match that is specified in the struts-config.xml file, the ActionServlet will load the classname specified in the config file and pass that on to the particular class. For the purposes of this discussion, we'll assume that the class is derived from com.jcorporate.expresso.core.controller.DBController or com.jcorporate.expresso.core.controller.Controller

- **Controller State Handler Routing** By the time you've hit this paragraph you probably get the idea that in any http request to an application server, there is a lot of 'hand offs' that go on! The Expresso Controller examines the http parameter called 'state' and uses the information it gleans from there to invoke via java reflection either the actual function within the controller or the class derived from com.jcorporate.expresso.core.controller.State

- **Controller State Handler Work** Finally! We've come to the step were some work is done! This is where you generate your Inputs, Outputs, Transitions and Blocks and add them to your ControllerRe-
quest. This particular step was discussed in the chapter on Controllers.

- **Controller Saves Response** Now that you're done, the Controller base class saves the entire controller response onto the Servlet Request Context under the name "controllerResponse". This part is important because this is the name of the 'javabean' you're going to be accessing while building your actual html page.

- **Action Servlet Forwards To The Renderer** Action servlet sees that the Controller has successfully executed and it now looks in its configuration again to see where to hand off control. Let's say for the sake of discussion, you have the forward set to "/expresso/components/example/mySample.jsp" In this case, it will hand off control to the jsp to finish the job. Remember that a JSP is actually a special form of a java servlet, and is thus a running 'program' so to speak. Now again, it must be emphasized that the forward does not necessarily have to be sent to a jsp. For example, it can be sent to a servlet for special post-processing or rendering, or forward to a Velocity template for straight rendering as an html page.

**How to get at your data.**

In a typical Expresso data rendering situation, most of your data will be stored in the ControllerResponse object. As mentioned before, the ControllerResponse object basically contains ControllerElements and Error Collections. Below is a UML diagram highlighting the relationship with various ControllerElements and ControllerResponse. It also highlights some of the most common access methods used in view programming.

The basic principle behind getting at the controller elements you generated is the fact that the Controller class saves the controllerRequest object in the request context under the name `controllerRequest`. Whether you use Velocity, JSP, or raw servlets, you will need to access the ControllerResponse object by looking in the request scope and getting the object by that name. The actual implementation of how you do this, for obvious reasons varies drastically. Further on we'll talk about just how to do that in a JSP environment using two different tag libraries.

**Understanding JavaBean properties naming conventions**
Before we get into the actual taglibs, a quick note should be made for those of you that have never dealt much with JavaBean terminology. Consider the following code:

```java
class myBean {
    //Standard Read only property named 'var1'
    private int var1 = 233;

    //Standard Read/Write property named 'var2'
    private String var2 = "Hello World!";

    //Mapped Property named 'values'
    private Map values = new HashMap();

    /**
     * Default Constructor
     */
    public myBean() {
    }

    /**
     * Getter property for var1
     */
    public int getVar1() {
        return var1;
    }

    /**
     * Getter property for var2
     */
    public String getVar2() {
        return var2;
    }

    /**
     * Setter property for var2
     */
    public void setVar2(String newValue) {
        var2 = newValue;
    }

    /**
     * Getter property for mapped property 'values'
     */
    public String getValues(String key) {
        return (String)values.get(key);
    }

    /**
     * Setter property for mapped property 'values'
     */
    public String setValues(String key, String newValue) {
        values.put(key, newValue);
    }
}
```

Notice the naming pattern for the get/set values. You take the name of the variable, prepend 'get' or 'set' to it and then capitalize the first letter of the value. Java uses this naming convention to find what it calls properties for JavaBeans. What you tell many java libraries, for example, is to set the property var2 to "abcdefg". The underlying tools figure out that the function it needs to call is setVar2("abcdefg").
naming method and corresponding method references are used extensively in jsp tag libraries.

The odd duck out that I want to mention is the so-called "Mapped Properties", which in the example above was called "values". While the Struts tag library can reference Mapped Properties, the Java Standard Tag Library (JSTL) does not. However, a workaround can be made by exposing the underlying Map object, which JSTL can work with. To add this compatibility layer, simply add the method:

```java
public Map getValuesMap() {
    return Collections.unmodifiableMap(values);
}
```

and you now have a read only Map that JSTL can work with but not mess up the rest of your bean state.

## Rendering ControllerResponse With JSP

Although we can use `<jsp:usebean property="controllerResponse">`, this gets troublesome when dealing with any controllerResponse of sufficient complexity. For the rest of the chapter we will focus on two methods of writing the JSP pages: The Java Standard Tag Library (JSTL), or the Struts Tag Library.

### Which tag library is better: Struts or JSTL?

The answer is that famous ambiguous phrase: "It Depends."

JSTL has advantages in that it is now an official JSR-based specification and will continue to grow in popularity as 'the' way of working with jsp pages.

However, Struts tags have the advantage in that they support Servlet API 2.2 compliant application servers. JSTL requires Servlet API 2.3, and as such can easily eliminate many houses that rely on older application servers.

Finally, it may depend on what you've been trained in. Many people who work with Expresso do so with Struts experience, and therefore, they can easily be up to speed with Expresso JSP writing in much less time than it would take to learn JSTL.

If you are completely new to both and you have a Servlet API 2.3 compliant container (such as Jakarta Tomcat 4.0 or greater), the author recommends that you would be better off if you learn JSTL since it will eventually become a widely used specification. To get your feet wet, it is recommended that you take a look at "JSTL in Action" by Shawn Bayern, published by Manning Press. Shawn is implementation lead for the Jakarta JSTL implementation and is a member of the JCP Committee on JSTL. The book is amazingly easy to read and does not require previous experience with Struts, Expresso, or anything like that.

## A Common Controller

Before we get started in this introduction, we need a sample ControllerResponse to work with from the JSTL side. Our goal is to output the contents of the ControllerResponse onto a jsp page.

```java
public Controller myController {
    //declarations here
    public ControllerResponse runMyState(ControllerRequest request, ControllerResponse response) {
        //Add an output named 'hello' to the response
        response.add(new Output("hello","This is a rather unusual Hello World"));
    }
}
```
Processing With Struts Tags

A sample JSP code below will show how to access all the various items in the ControllerResponse. Formatting is kept out to keep the code short. Remember that with bean properties, if you see somebody reading the property 'content', that under the hood, the function getContent() is being called.

```jsp
<%@ taglib uri="/WEB-INF/tld/struts-logic.tld" prefix="logic"%>
<%@ taglib uri="/WEB-INF/tld/struts-html.tld" prefix="html"%>
<%@ taglib uri="/WEB-INF/tld/struts-bean.tld" prefix="bean"%>

<html>
<head>
<title>Hello World With Twist</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
</head>
<body>

<%-- Write Output Hello
---%>
Developing The View in the MVC

Processing with Struts Extended Tags

The biggest limitation that Struts has is that when dealing with Expresso ControllerElements, you often have to perform multiple <bean:define> tags before you can write the actual code. The Struts Extended
Tags included with Expresso are designed to reduce some of the coding work and still retain the same feel as the original Struts tags.

Syntax-wise, the extension tags are also significantly different. The tags were originally written before Struts had the ability to access items such as 'Mapped Properties'. As such, the syntax conforms more like XPath syntax.

- Nesting is performed with '/' instead of periods.
- ControllerElement attributes are accessed through the at sign (@).
- Properties are accessed through periods.

Now let's look at a jsp page written entirely in Struts-Extension tags, that has the identical functionality as the pure-struts tags. Notice that his example uses the same prefix, but they point to the expresso-extension tag libraries instead. You can mix Struts extension tags and pure struts tags in the same page. Usually, the author assigns struts tag libraries to have the struts extension to them. So 'logic' becomes 'struts-logic', 'bean' becomes 'struts-bean' etc.

```jsp
<%@ taglib uri="/WEB-INF/tld/expresso-logic.tld" prefix="logic"%>
<%@ taglib uri="/WEB-INF/tld/expresso-html.tld" prefix="html"%>
<%@ taglib uri="/WEB-INF/tld/expresso-bean.tld" prefix="bean"%>

<html>
<head>
<title>Hello World With Twist</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
</head>
<body>

<%-- Write Output Hello --%>
<h1><bean:write property="hello"/></h1>

<%-- Iterate over all of aBlock and for each output retrieved, write the message and the cell they're from. --%>
<logic:iterate id="eachPrisoner" property="aBlock"/>
<p><b>Message: </b> <bean:write name="eachPrisoner"/>
From Cell:<bean:write name="eachPrisoner" property="@CellNumber"/></p>

</logic:iterate>
<hr/>
<h1>Sample Form</h1>

<%-- Create a Sample form with one input and one button --%>
<html:form action="/myapp/MyForm.do" method="post"/>

<%-- Create the button --%>
<html:text property="input1"/>

Create the Submit label. The Expresso extension automatically create any and all hidden parameters.
```
You may notice in this example, that many pieces are calculated automatically for you. For example, unless you specify the tag's 'name', it is automatically assumed that you are talking about the object named 'controllerResponse'. Secondly, the XPath-like tags are smart enough to know that if it comes across an output, bean:write is talking about writing the contents of the output. So significant typing can be saved using the Struts-extension tags rather than pure struts tags. But sometimes the logic in dealing with the struts-extension tags can be rather confusing, especially when dealing with nested elements. So sometimes, the pure struts tags or JSTL tags make more sense.

Processing with JSTL

JSTL has its own advantages and detractions. Among its advantages are:

- Standards Based. The JSTL expression language is part of the JSP 2.0 specification.
- Powerful Expression Language. The JSTL expression language has great flexibility, and provides a viable alternative to including scriptlets in your JSP pages.
- Very friendly with Visual HTML builders. Include <c:out> tags inside the appropriate items in a standard html page, and programs such as Dreamweaver will be quite happy.

At the same time, its detractions are:

- Very Verbose. JSTL expressions are much more verbose than, for example, the struts extension tags.
- Incapable of Mapped Properties. Unlike the Struts 1.1 tags, JSTL cannot handle mapped properties in a bean. To be compatible with JSTL, you must include methods to expose an entire underlying java.util.Map interface.
- Slower performance. JSTL expressions are much more complicated than Struts-extensions, and are thus likely to lead to slower performance. This penalty, of course, will tend to become less of a factor as time goes on as more people analyze the JSTL-EL and optimize its performance.

Processing with JSTL Extended Tags

Like the basic Struts tags, the JSTL tags are limited when dealing with Expresso ControllerElements: you often have to perform multiple <c:out> tags before you can write the actual code. The JSTL Extended Tags included with Expresso are designed to reduce some of the coding work and still retain the use of the JSTL expression language.

Details TBA
Rendering the ControllerResponse with XSLT

As of Expresso 5.1, XSL stylesheets are first class citizens. To have a response processed with an XML stylesheet, simply add the location of the stylesheet to your 'struts-configuration'. The following listing gives an example of this.

```
<action path="/SampleController" type="org.example.myapp" name="default" scope="request" validate="false">
    <!-- this state forwards to a jsp -->
    <forward name="start" path="/expresso/components/myapp/start.jsp"/>
    <!-- this state forwards to a stylesheet for processing -->
    <forward name="process" path="/expresso/components/myapp/process.xslt"/>
</action>
```

Conclusion

Contributors

The following persons have contributed their time to this chapter:

- Larry Hamel (JGroup Expert)
- Mike Rimov
- Mike Traum (JGroup Expert)

Note

Was this EDG documentation helpful? Do you wish to express your appreciation for the time expended over years developing the EDG doc? We now accept and appreciate monetary donations. Your support will keep the EDG doc alive and current. Please click the Donate button and enter ANY amount you think the EDG doc is worth. In appreciation of a $35+ donation, we'll give you a subscription service by emailing you notifications of doc updates; and donations $75+ will also receive an Expresso T-shirt. All online donation forms are SSL secured and payment can be made via Credit Card or your Paypal account. Thank you in advance.


Copyright © 2001-2004 Jcorporate Ltd. All rights reserved.
Chapter 12. Expresso Taglib Descriptions

Note

If you find this EDG documentation helpful please consider DONATING! to keep the doc alive and current.

Version: Expresso 5.5

Maintainer: Turgay Zengin

[mailto:turgay_zengin@hotmail.com?Subject=EDG]

Introduction

Expresso Taglibs is a custom tag library. Expresso of course extends the Struts and JSTL, including an Action, Input, and Error tag that make page design much easier and cleaner.

These tags are fairly simple and work well. Give them a try. You will find that after you create one page with them, you will never go back to the old way. They make a HUGE difference!

Tag Library Descriptor

Also included is the tag library descriptor, expresso.tld, under the /WEB-INF/ directory in CVS. If you have questions about which options are required, optional, etc., just scan the expresso.tld. This expresso.tld should be placed in your WEB-INF/ directory. Then, edit your web.xml file to include this:

```
<taglib>
  <taglib-uri>/expresso</taglib-uri>
  <taglib-location>/WEB-INF/expresso.tld</taglib-location>
</taglib>
```

This is also done in the web.xml file in CVS.

Quick usage: Include this at the top of the page:

```
<%@ taglib uri="/expresso" prefix="expresso" %>
```

and then the tags wherever you want them.

Important Notes

The custom tags make some assumptions about how your jsp page is defined. One important assumption is that your controller is called "controller" in the jsp page (jsp:useBean id="controller" scope="session" class="myControllerClass" /). If you have used an id in your usebean tag that is something other then "controller", then you need to specify the controller name to each of your expresso tags as an argument, like: <expresso:InputTag name="expresso.input.first_name" controller="mycontrollername" />

Expresso Extended Struts Tags
Expresso has extended a number of Struts Tags.

**How to Use Struts Tags in Expresso**

The "extended" Struts tags implemented in the com.jcorporate.expresso.ext.struts.taglib.* packages all use a consistent meaning for the "name" and "property" attributes for the tags that are somewhat different than the normal Struts meanings.

1. **Name Attribute**: The name attribute for the extended tags refers to a specific ControllerElement in the current session. Such an element can be defined by means of the tag or the tag. For the "input field" related tags (e.g. text,textarea,hidden,file,checkbox,password and select), the ControllerElement being referred to should be an "Input" object. For example: `<logic:iterate id="oneInput" property="inputs"/> <html:text name="oneInput"/>`  `<logic:iterate>` Would iterate over each "input" element in the ControllerResponse and display this input as a text field. Note that the "text" tag does *not* require the "property" attribute, unlike the corresponding Struts tag that it inherits from.

2. **Property Attribute**: The "property" attribute can take one of several "forms" for the extended tags, these being:

   - **Directly Supported Property**: Each type of ControllerElement supports a few "built-in" properties. When referring to a specific element (e.g. via the "name" parameter, or via the "Reference to Element" method described below) this property name can be used directly. For example: `<bean:write name="oneInput" property="name"/>`  `<bean:write>` Would write the element name of the input element on the JSP page.

   - **"Special" Name**: Several special names can be used to refer to entire groups of elements, for convenience:

     - inputs: All Input elements in the ControllerResponse
     - outputs: All Output elements in the ControllerResponse
     - blocks: All Block elements in the ControllerResponse
     - transitions All Transition elements in the ControllerResponse

   - **Attribute**: ControllerElements can have additional meta-data stored with them by the State that generates them: This might take the form of "formatting hints" (such as "table" to indicate that a block should be rendered as a table), or any other useful information specific to the Controller. These attributes can be referenced by prefixing their name with the "@" symbol, so, for example: `<bean:write name="oneOutput" property="@specialAttribute"/>`  `<bean:write>` Would print the value of the attribute "specialAttribute" of the specified Output element. This attribute would have been set via the setAttribute(String, String) method by the Controller or State generating the output.

3. **Reference to Element**: Use of the property attribute in a tag *without* the name attribute refers to an element directly - for example: `<html:text property="customerName"/>`

   Creates an input field of type "text" from the element called "customerName" (customerName must be of type "Input"). In this case, customerName is not nested within any other input. If it were nested, for example, in a block stored as a session bean called "bigBlock" (say by an iterate or bean:define tag), then the following syntax would refer to it:
4. **Nested Reference to Element**: A nested reference to an element provides the means to refer to elements nested with other elements by name. For example, if an output "customer" has a nested output called "address", you can refer to this element with a property value of "customer/address". Nested references can be combined with property or attribute references, so for example:

- \( a/b.label \) Refers to the "label" property of the element called "b" nested within the element called "a".
- \( a/b@something \) Refers to the \*attribute\* of the element b, found nested within the element "a"

---

### Expresso Bean Tag Library

This tag library defines Expresso-aware bean tags, allowing easy access to write contents of Output objects and various attributes of other ControllerElements found in the ControllerResponse.

[home/expresso/expresso-web/WEB-INF/classes/com/jcorporate/expresso/ext/struts/taglib/bean/ExWriteTag.java](home/expresso/expresso-web/WEB-INF/classes/com/jcorporate/expresso/ext/struts/taglib/bean/ExWriteTag.java)

### Expresso Logic Tags

This tag library contains extensions of a few of the Struts "logic" tags, making them "aware" of Expresso's ControllerResponse object.

**Table 12.1. Expresso Logic Tags**

<table>
<thead>
<tr>
<th>Tag Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExNotPresentTag</td>
<td>/home/expresso/expresso/expresso-web/WEB-INF/classes/com/jcorporate/expresso/ext/struts/taglib/logic/ExNotPresentTag.java</td>
</tr>
<tr>
<td>ExPresentTag</td>
<td>/home/expresso/expresso/expresso-web/WEB-INF/classes/com/jcorporate/expresso/ext/struts/taglib/logic/ExPresentTag.java</td>
</tr>
</tbody>
</table>
Expresso Taglib Descriptions

<table>
<thead>
<tr>
<th>Tag Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExIterateTag</td>
<td>/home/expresso/expresso/expresso-web/WEB-INF/classes/com/jcorporate/expresso/ext/struts/taglib/logic/ExIterateTag.java</td>
</tr>
</tbody>
</table>

For example, the following page will display every Output in the current ControllerResponse, one per line:

```jsp
<%@ page language="java" %>
<%@ taglib uri="/WEB-INF/expresso-bean.tld" prefix="bean" %>
<%@ taglib uri="/WEB-INF/expresso-html.tld" prefix="html" %>
<%@ taglib uri="/WEB-INF/expresso-logic.tld" prefix="logic" %>
```

**Expresso HTML Tags**

This tag library extends selected Struts HTML tags, creating Expresso-aware HTML tags, particularly tags used to produce various types of input fields for use in HTML forms. The package extends the following tags:

**Table 12.2. Expresso HTML Tags**

<table>
<thead>
<tr>
<th>Tag Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExBaseFieldTag</td>
<td>An extension of BaseFieldTag to create Expresso-aware versions of tags that inherit from this class.</td>
</tr>
<tr>
<td>ExCheckboxTag</td>
<td>Tag for rendering Checkbox input fields from Input elements.</td>
</tr>
<tr>
<td>ExSelectTag</td>
<td>Tag for rendering Select lists (drop-down lists) from Input elements.</td>
</tr>
<tr>
<td>ExSubmitTag</td>
<td></td>
</tr>
<tr>
<td>FileTag</td>
<td></td>
</tr>
<tr>
<td>HiddenTag</td>
<td></td>
</tr>
<tr>
<td>PasswordTag</td>
<td></td>
</tr>
<tr>
<td>TextTag</td>
<td></td>
</tr>
</tbody>
</table>

For each of the above tags, the attributes and usage of the tag is identical to the Struts tag that it extends, with one exception: the "property" attribute, instead of referring to a property of the current ActionForm bean, refers instead to an Input element of the current ControllerResponse. The property field can contain an XPath-like "controller element path", as described here, to refer to nested ControllerElements and sub-properties of those elements.

**Expresso Tag Library Summary**

The Expresso Tag Library contains a number of JSP custom tags to help build JSP pages easily, and
eliminate the Java usage in JSP pages. All tags are currently in the com.jcorporate.expresso.ext.taglib package. Many of these tag are being depreciated in favor of JSTL. They are included in this documentation for reference to help you transition to JSTL.

Example JSP code

The following JSP will try to illustrate the tags in the library:

1. <font color="red">
2. <expresso:ErrorTag/>
3. </font>
4. <table class="jc-default" border="0" align="center">
5. <expresso:TableHead value="ID|Name|Start Date|End Date|Duration|Command"/>
6. <expresso:ElementCollection type="block">
7. <expresso:ElementIterator>
8. <expresso:ElementCollection type="output">
9. <expresso:ElementIterator>
10. <expresso:OutputTag name="xxx">
11. <tr>
12. <td><expresso:AttributeTag name="CourseId" /></td>
13. <td><expresso:AttributeTag name="CourseName" /></td>
14. <td><expresso:AttributeTag name="CourseStartDate" /></td>
15. <td><expresso:AttributeTag name="CourseEndDate" /></td>
16. <td><expresso:AttributeTag name="CourseDuration" /></td>
17. </expresso:OutputTag>
18. </expresso:ElementIterator>
19. <expresso:ElementCollection>
20. <expresso:ElementCollection type="transition">
21. <expresso:ElementIterator>
22. <html:form action="/training/TrainingController.do&state=promptUpdateCourse" method="POST">
23. <td><expresso:TransitionTag name="promptUpdateCourse" /></td>
24. </html:form>
25. </expresso:ElementIterator>
26. </expresso:ElementCollection>
27. </tr>
28. </expresso:ElementIterator>
29. </expresso:ElementCollection>
30. </table>

This example will: Display the Expresso Error Collection if it exists (Lines 1-3) Construct and display a table header with header names ID, Name, Start Date, End Date, Duration, Command (Lines 4,5) Iterate through all Block objects found in the ControllerResponse (Lines 6,7) --->Iterate through all Output objects in the current Block (Lines 8,9) ---------->Use an OutputTag to help OutputAttribute tags (Line 10) - -----------Display the various attributes of the current Output object (Lines 12-16) ----->Iterate through all Transition objects found in the ControllerResponse (Lines 20,21) ---------->Display the current Transition as a submit box (Lines 22-24)

Expresso Tags Described

The following outlines the Expresso Tags. We'll be adding another column to describe how JSTL replaces some of these tags.

Table 12.3. Expresso Tags Described
<table>
<thead>
<tr>
<th>Tag Name</th>
<th>Description</th>
<th>If Deprecated then by</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActionTag</td>
<td>Renders an Expresso action, and associated hidden fields</td>
<td></td>
</tr>
<tr>
<td>AttributeTag</td>
<td>A tag to display an Attribute from an Expresso Input or Output tag.</td>
<td></td>
</tr>
<tr>
<td>Back</td>
<td>Displays a &quot;Back&quot; button or image to go back to a page</td>
<td></td>
</tr>
<tr>
<td>Block</td>
<td>A tag to act as a container for child InputTag, OutputTag, TransitionTag and Block tags. Directly related to the Expresso Block controller element.</td>
<td></td>
</tr>
<tr>
<td>config</td>
<td>Defines a scripting variable out of the ConfigManager object</td>
<td></td>
</tr>
<tr>
<td>ContentTag</td>
<td>A tag to display an Expresso Output Element's Contents Field.</td>
<td></td>
</tr>
<tr>
<td>Context</td>
<td>A tag to insert the web-app context, optionally including the expresso-dir and the path to an application's files</td>
<td></td>
</tr>
<tr>
<td>DBDescription</td>
<td>A tag to display the description text for the current DB context</td>
<td></td>
</tr>
<tr>
<td>DBMaint</td>
<td>A tag to include a row with calls to DBMaint for a dboject</td>
<td></td>
</tr>
<tr>
<td>ElementCollection</td>
<td>A tag that retrieves a collection of a controller elements that can later be iterated through using the ElementIterator tag</td>
<td></td>
</tr>
<tr>
<td>ElementIterator</td>
<td>A tag to iterate through each controller element inside a parent ControllerCollection tag</td>
<td></td>
</tr>
<tr>
<td>ErrorMessages</td>
<td>A tag to display error messages based on their key's in the MessageBundle.properties</td>
<td></td>
</tr>
<tr>
<td>ErrorTag</td>
<td>A tag to display an Expresso Error Collection as a Unordered List (UL)</td>
<td></td>
</tr>
<tr>
<td>IfElementExists</td>
<td>A tag to check if a particular controller element exists as part of the parent: controller-response, block or element-iterator tags.</td>
<td></td>
</tr>
<tr>
<td>IfErrorExists</td>
<td>A tag to check if the controller generated errors and stored it in the page context.</td>
<td></td>
</tr>
<tr>
<td>IfLoggedIn</td>
<td>A tag to check if a session is established and a valid user is logged in.</td>
<td></td>
</tr>
<tr>
<td>IfMemberOfGroup</td>
<td>This tag checks the currently logged in user for membership into a specified security group.</td>
<td></td>
</tr>
</tbody>
</table>
### Expresso Library Tag Details

Below describes each tag and how it is used.

**ActionTag - Renders an Expresso action, and associated hidden fields**

```xml
<expresso:ActionTag name="refreshCompany" /> - renders the expresso action button with that name.
```

This tag also takes care of the hidden form field that action tags rely on to pass parameters. This action tag allows seamless changes between GET and POST form actions. All three tags have no body (no closing tag). These simplest examples will work for most all cases, but there are many options you can set to override the default behaviors. Sufficient code and info has been included here to cover three out of four common expresso cases (Output is missing...this is easy in its simple case, but the nested cases may be a little tricky to create as tags).

**AttributeTag - Displays an attribute from an Input or Output object**

The AttributeTag can be used to display attributes from Input or Output objects. These attributes are the ones that you put in Input or Output objects by `out1.setAttribute(String attrName, String value)` in your States. The AttributeTag can either be used stand-alone, in which case all the arguments are required, or nested within an OutputTag or InputTag, in which case only the attribute name must be specified. If no valid Input or Output objects can be found, a JspTagException will be thrown, saying "A controllerElement was not specified for this attribute tag, or a parent tag did not supply a controllerElement." Examples:
1. `<expresso:AttributeTag controllerElement="product" name="product_desc" type="OUTPUT" />
   Will display the value of the "product_desc" attribute, contained in the "product" Output object.

2. `<expresso:OutputTag name="product">
   <expresso:AttributeTag name="product_desc"/>
   </expresso:OutputTag>` This one does the same thing as the 1st example but this time the Output-Tag must be enclosed in a Block or ElementIterator tag

### Table 12.4. Attribute Tag

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Specifies the name of the attribute whose value will be displayed (If the attribute is not found, &quot;ATTRIBUTE NOT FOUND&quot; is displayed. [Required] [RT Expr]</td>
</tr>
<tr>
<td>controllerElement</td>
<td>If supplied, specifies the name of the ControllerElement object holding the attribute. If not supplied, the tag will try to use the enclosing InputTag or OutputTag, if there is any. [RT Expr]</td>
</tr>
<tr>
<td>type</td>
<td>If supplied, specifies the type of the ControllerElement (Must be &quot;INPUT&quot; or &quot;OUTPUT&quot;). If not supplied, the tag will put the correct value according to the enclosing InputTag or OutputTag, if there is any. [RT Expr]</td>
</tr>
</tbody>
</table>

### Back - Displays a "Back" button or image to go back to a page

Constructs an anchor to go to a specified URL, to go to the previous page, or to refresh the current page.

### Table 12.5. Back Attributes

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>page</td>
<td>Specifies the page to go to. If null, a hyperlink to go back to the previous page is constructed. If &quot;current&quot;, a hyperlink to refresh the current page is constructed, if something else, a hyperlink to go to the desired URL is constructed. [RT Expr]</td>
</tr>
<tr>
<td>image</td>
<td>Defaults to expresso-images-dir/FingerLeft.gif [RT Expr]</td>
</tr>
<tr>
<td>key</td>
<td>Title will be read from the locale file if supplied. [RT Expr]</td>
</tr>
<tr>
<td>title</td>
<td>Title to be displayed. [RT Expr]</td>
</tr>
</tbody>
</table>

### Block - Acts as a container for child InputTag, Output-Tag, TransitionTag and Block tags

The Block tag is used to provide a container tag for other tags. Just like a Block object being a container
for ControllerElement objects. If there is a container for a Block tag, it must either be an ElementIterator tag, or another Block tag. Examples: <expresso:Block name="courseDataInput"> <tr> <td> <expresso:InputTag name="CourseName" /> </td> <td> <expresso:InputTag name="CourseDuration" / > </td> </tr> </expresso:Block> The Block tag helps the InputTag tags to retrieve the Block object and then execute Block.getInput()

### Table 12.6. Block Attributes

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>The name of the block that we want to act as a container for child tags. [Required] [RT Expr]</td>
</tr>
</tbody>
</table>

**config - Defines a scripting variable based on the specified values**

This tag puts the specified value to the desired scope. If "value" is not supplied, gets the value as ConfigManager.getProperty(). If "value" is supplied, this is used to be placed in the desired scope.

### Table 12.7. config Attributes

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>Specifies the name of the scripting variable that will be made available. [Required] [RT Expr]</td>
</tr>
<tr>
<td>property</td>
<td>Specifies which property to get from ConfigManager. [Required][RT Expr]</td>
</tr>
<tr>
<td>db</td>
<td>If supplied, specifies the database definition to get configuration parameter from. [RT Expr]</td>
</tr>
<tr>
<td>toScope</td>
<td>One of &quot;page&quot; (default if not supplied), &quot;request&quot;, &quot;session&quot;, &quot;application&quot;. [RT Expr]</td>
</tr>
<tr>
<td>type</td>
<td>? [RT Expr]</td>
</tr>
<tr>
<td>value</td>
<td>If supplied, this will be placed in the desired scope. [RT Expr]</td>
</tr>
</tbody>
</table>

**ContentTag - Displays an Expresso Output element's Contents field.**

ContentTag must be nested within an OutputTag tag. The ContentTag has no body, and simply displays the content from the parent OutputTag (Output.getContent()). Examples: <expresso:OutputTag name="product"> <expresso:ContentTag /></br> </expresso:OutputTag> Gets the Output object from the OutputTag tag, and then displays Output.getContent(). No tag attributes are needed.

**Context - Inserts the web-app context**

Writes the context path and/or the "expresso" directory. Optionally figures out context paths for specific schema's as well. Examples: <expresso:Context> Writes: /expresso/expresso/components/training for example
Table 12.8. Context Attributes

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>schema</td>
<td>The specific schema whose context path we want to get. [RT Expr]</td>
</tr>
</tbody>
</table>

**DBDescription - Displays the description text for the current DB context**

Retrieves the CurrentLogin object (if any) from the pageContext object, and displays ConfigManager.getContext(currentLogin.getDBName()).getDescription(). If there is no CurrentLogin, "default" is used for the dbName. Examples: <expresso:DBDescription> Displays "Oracle Database", for example. No tag attribute is needed.

**DBMaint - Includes a row with calls to DBMaint for a dbobject**

Provides a row for a table that contains links to the DBMaint controller providing add/update/delete functionality for a specific DB object.

Table 12.9. DBMaint Attributes

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbobj</td>
<td>The DObject class. [Required] [RT Expr]</td>
</tr>
<tr>
<td>label</td>
<td>Label to be displayed. [RT Expr]</td>
</tr>
<tr>
<td>help</td>
<td>The URL for a help file about this DObject. [RT Expr]</td>
</tr>
</tbody>
</table>

**ElementCollection - Retrieves a collection of a controller elements that can later be iterated through using the ElementIterator tag**

This tag provides an Enumeration object to an ElementIterator tag, providing access to the desired ControllerElements. The ElementIterator may iterate through all the elements found in the ElementCollection.

Table 12.10. ElementCollection Attributes

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>What type of objects do we want to get? Type should be one of &quot;block&quot;, &quot;input&quot;, &quot;output&quot;, &quot;transition&quot; or &quot;nested&quot;. If something else is supplied, the tag will do nothing. [Required] [RT Expr]</td>
</tr>
</tbody>
</table>
ElementIterator - iterates through each controller element inside a parent ElementCollection tag

This tag can only be used nested in an ElementCollection tag. It will iterate through all the elements found in the ElementCollection, and provide each element to the nested tags. No tag attributes are needed.

ErrorMessages - Displays an Expresso Error Collection from the Message Bundle

This tag is a lot like the ErrorTag in that it looks for an Expresso ErrorCollection in the page request. If an errorCollection is found, the tag iterates through the errors and displays them as an unordered (bulleted) list, however this tag goes one step further, it will display the text from the Message Bundle. This is done by adding the Message Bundle key for the string that is associated with that error. Examples: In your JSP <expresso:ErrorMessages /> Now in the java code you simple put ErrorCollection errors = new ErrorCollection(); errors.addError( "error.username.required" ); Then make sure that this maps to the MessageBundle associated with controller (done in the Schema class) error.username.required=<li>Username is required

Table 12.11. ErrorMessages Attributes

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the ErrorCollection object. If not supplied, defaults to ErrorCollection.ERRORCOLLECTIONKEY [RT Expr]</td>
</tr>
<tr>
<td>property</td>
<td>If supplied, the tag will display the set of error messages related to a specific property, if not supplied, the tag will display all error messages. [RT Expr]</td>
</tr>
</tbody>
</table>

ErrorTag - Displays an Expresso Error Collection

This tag looks for an Expresso ErrorCollection in the page request. If an errorCollection is found, the tag iterates through the errors and displays them as an unordered (bulleted) list. An errorCollection object is generally inserted into the page request by calling controller.saveErrors(errorCollection), which saves the errorCollection under the proper parameter name. If no errorCollection is found in the page request, this tag does nothing, so it is safe to include this on all of your jsp pages if desired. The ErrorTag renders an Expresso ErrorCollection on a page. The error list is rendered as an html unordered (bulleted) list. The errors are displayed in the order they were saved to the ErrorCollection. Examples: <expresso:ErrorTag /> <expresso:ErrorTag name="myErrors" property="validationErrors" /> 

Table 12.12. ErrorTag Attributes

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the ErrorCollection object. If not supplied, defaults to ErrorCollection.ERRORCOLLECTIONKEY [RT Expr]</td>
</tr>
<tr>
<td>property</td>
<td>If supplied, the tag will display the set of error messages related to a specific property, if not supplied, the tag will display all error messages. [RT Expr]</td>
</tr>
</tbody>
</table>
**IfElementExists - Checks if a particular controller element exists.**

Tries to find the named element in the current ControllerResponse, ElementIterator or Block, whichever is appropriate. If finds, the body of the tag will be processed. "inverse" attribute can be used to mean "If Element Does Not Exist"

Table 12.13. IfElementExists Attributes

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the element to be checked. [Required] [RT Expr]</td>
</tr>
<tr>
<td>type</td>
<td>The type of the element to be checked (Must be &quot;block&quot;, &quot;input&quot;, &quot;output&quot;, or &quot;transition&quot;) [Required] [RT Expr]</td>
</tr>
<tr>
<td>inverse</td>
<td>If &quot;true&quot;, the tag will mean &quot;IfElementDoesNotExist&quot; [RT Expr]</td>
</tr>
</tbody>
</table>

**IfErrorExists - Checks if the controller generated errors and stored it in the page context.**

If finds the Expresso ErrorCollection in the page context, then the body of the tag will be processed. No tag attributes are needed.

**IfLoggedIn - Checks if a session is established and a valid user is logged in.**

If there is a valid user logged in, the body of the tag will be processed. "inverse" attribute can be used to do something if there is no valid login.

Table 12.14. IfLoggedIn Attributes

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inverse</td>
<td>If this attribute is set to true, the tag will mean &quot;IfNotLoggedln&quot;. [RT Expr]</td>
</tr>
</tbody>
</table>

**IfMemberOfGroup - Checks if the user logged in has access to the specified group**

Tag checks to see if the user in the specified groupname is apart of that security group.

Table 12.15. IfMemberOfGroup Attributes
InputTag - Displays an Expresso Input Object

<expresso:InputTag name="expresso.input.first_name" /> - renders the expresso input with that name. This tag can currently render the simplest html input box, along with all of the associated directives such as displayLength, type, etc. The tag will also render pulldown-style select boxes, and will ensure that the default value will be selected when the select box is rendered. Currently, the Expresso input must be defined as type = "LISTBOX" to be rendered as a select box. Produces html code for a <html input>. The type will tell the tag what html input to produce(listbox, radio boxes, input box, check boxes, hidden). This tag must be contained in one of: ControllerResponse (top level), Block tag, or ElementIterator tag.

Table 12.16. InputTag Attributes

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the Input object. [Required] [RT Expr]</td>
</tr>
<tr>
<td>type</td>
<td>Must be one of: listbox, checkbox, checkbox-vertical, radio, radio-vertical, text, textarea, hidden, readonly. The tag will produce the correct type of html input. If no type is supplied, a normal input box is produced. [RT Expr]</td>
</tr>
<tr>
<td>size</td>
<td>Size of the input to be produced. [RT Expr]</td>
</tr>
<tr>
<td>maxlength</td>
<td>The maximum length for input boxes. [RT Expr]</td>
</tr>
<tr>
<td>showlabel</td>
<td>If &quot;true&quot;, the label of the Input object will be displayed. [RT Expr]</td>
</tr>
</tbody>
</table>

LabelTag - Prints the label of a controller element

Displays the specified element's label (with element.getLabel()).

Table 12.17. LabelTag Attributes

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the element whose label will be displayed. [Required] [RT Expr]</td>
</tr>
<tr>
<td>type</td>
<td>Type of the element (&quot;input&quot;, &quot;output&quot;, &quot;transition&quot; or &quot;block&quot;). [RT Expr]</td>
</tr>
</tbody>
</table>

Login - Tries to make the user log in

Tag to transition between database contexts, either logging the user into a new context (if possible with
cookies, etc), or taking them to the login screen if it is not possible.

Table 12.18. Login Attributes

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>db</td>
<td>If supplied, tries to log in to this database context. [RT Expr]</td>
</tr>
<tr>
<td>forward</td>
<td>If &quot;false&quot;, tries to to logInAsNone(request,db) [RT Expr]</td>
</tr>
</tbody>
</table>

output - Defines a scripting variable based on the specified values

This tag puts the specified value to the desired scope. Must be nested within an OutputTag tag. If "value" is not supplied, gets the value as getContent() from the enclosing OutputTag. If "value" is supplied, this is used to be placed in the desired scope.

Table 12.19. output Attributes

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>Specifies the name of the scripting variable that will be made available. [Required] [RT Expr]</td>
</tr>
<tr>
<td>toScope</td>
<td>One of &quot;page&quot; (default if not supplied), &quot;request&quot;, &quot;session&quot;, &quot;application&quot;. [RT Expr]</td>
</tr>
<tr>
<td>type</td>
<td>? [RT Expr]</td>
</tr>
<tr>
<td>value</td>
<td>If supplied, this will be placed in the desired scope. [RT Expr]</td>
</tr>
</tbody>
</table>

OutputTag - Displays an Expresso Output Element

OutputTag is a tag that by itself does not display anything. The OutputTag must have a start and end tag. Between these tags, the ContentTag can be used to place the content from the Output, and the AttributeTag can be used to place attributes of the Output. The OutputTag can also be used to fetch out nested outputs, if one OutputTag is nested within another one.

Table 12.20. OutputTag Attributes

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the element to be used in the nested ContentTag and AttributeTag tags. Will be automatically set if nested in an ElementIterator tag. [Required] [RT Expr]</td>
</tr>
</tbody>
</table>

RestrictAccess - Allows quick and dirty restriction of access to jsp pages
Provides an easy way to restrict access to particular pages. If the user is not found to be allowed to access this page, s/he will be forwarded to a specified page, or a JspTagException saying "Access to this page is denied". This tag allows quick access restriction to a jsp page. Example Usage: 

```<expresso:RestrictAccess allowedUsers='Admin,scann' allowedGroups='Admin,Everyone />' or
<expresso:RestrictAccess denyURL='join.jsp'/> or <expresso:RestrictAccess/>```

**Table 12.21. RestrictAccess Attributes**

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>allowedUsers</td>
<td>A comma delimited list of user names that are allowed to access this page (Case Sensitive). [RT Expr]</td>
</tr>
<tr>
<td>allowedGroups</td>
<td>A comma delimited list of groups that are allowed to access this page, also case sensitive. [Note that you can have BOTH allowedUsers and allowedGroups]. [RT Expr]</td>
</tr>
<tr>
<td>denyURL</td>
<td>URL to forward to, if the user is not allowed to access this page. The URL to forward to if access is denied. If this attribute is not supplied, then a JspTagException is thrown with the message 'Access to this page is denied'. If no allowedUsers or allowedGroups are given, then as long as the person is logged in as ANYTHING other than NONE, they'll be given access to the page. [RT Expr]</td>
</tr>
</tbody>
</table>

**session - Defines a scripting variable based on the specified values**

This tag puts the specified value to the desired scope. If "value" is not supplied, gets the value as page-Context.getRequest().getAttribute(property). If "value" is supplied, this is used to be placed in the desired scope. (TODO: should be getSession??)

**Table 12.22. session Attributes**

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>Specifies the name of the scripting variable that will be made available. [Required] [RT Expr]</td>
</tr>
<tr>
<td>property</td>
<td>Specifies which property to get from session scope. [Required][RT Expr]</td>
</tr>
<tr>
<td>toScope</td>
<td>One of &quot;page&quot; (default if not supplied), &quot;request&quot;, &quot;session&quot;, &quot;application&quot;. [RT Expr]</td>
</tr>
<tr>
<td>type</td>
<td>? [RT Expr]</td>
</tr>
<tr>
<td>value</td>
<td>If supplied, this will be placed in the desired scope. [RT Expr]</td>
</tr>
</tbody>
</table>

**setup - Defines a scripting variable based on the specified values**
This tag puts the specified value to the desired scope. If "value" is not supplied, gets the value as Setup.getValueRequired(). If "value" is supplied, this is used to be placed in the desired scope.

Table 12.23. setup Attributes

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>Specifies the name of the scripting variable that will be made available. [Required] [RT Expr]</td>
</tr>
<tr>
<td>property</td>
<td>Specifies which property to get from Setup. [Required][RT Expr]</td>
</tr>
<tr>
<td>db</td>
<td>If supplied, specifies the database definition to get setup parameter from. [RT Expr]</td>
</tr>
<tr>
<td>toScope</td>
<td>One of &quot;page&quot; (default if not supplied), &quot;request&quot;, &quot;session&quot;, &quot;application&quot;. [RT Expr]</td>
</tr>
<tr>
<td>type</td>
<td>? [RT Expr]</td>
</tr>
<tr>
<td>value</td>
<td>If supplied, this will be placed in the desired scope. [RT Expr]</td>
</tr>
</tbody>
</table>

TableHead - Displays a pipe-delimited string as a table header

Parses apart a pipe-delimited string into a table heading row. The pipe-delimited string is a convenience used by a number of Expresso controllers to pass the table headings required for various tables to the view. If the keys attribute is used instead of "value", the headers will be read from the appropriate MessagesBundle.properties file.

Table 12.24. TableHead Attributes

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>Values separated by a pipe - these don't get &quot;translated. [Required] [RT Expr]</td>
</tr>
<tr>
<td>keys</td>
<td>Keys into the language file - these are translated for display. [RT Expr]</td>
</tr>
</tbody>
</table>

TransitionTag - Displays an Expresso Transition Button

Displays a submit button with the information contained in the current Transition element. If "value" is not supplied, the label of the button will come from element.getLabel().

Table 12.25. TransitionTag Attributes

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the Transition element to be displayed. [Required] [RT Expr]</td>
</tr>
<tr>
<td>ClassValue</td>
<td>If supplied, will be used to render the class value for the button if you use Cascading Style Sheets.</td>
</tr>
</tbody>
</table>
OnClick If supplied, will be used to render the onclick parameter for javascript submit box. [RT Expr]

### TransitionParamsTag - Displays the parameters with an Expresso Transition

Writes: `element.getParamString()`. This tag does not construct the submit button therefore makes for a great tag for creating hrefs. `<a href="/Login.do?<expresso:TransitionParamsTag name="promptFirstTime" />">I don't have a Username</a>`

#### Table 12.26. TransitionParamsTag Attributes

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the Transition element whose parameters are to be displayed. [Required] [RT Expr]</td>
</tr>
<tr>
<td>value</td>
<td>? [RT Expr]</td>
</tr>
</tbody>
</table>

### UserName - Displays the logged in user-name

If there is a valid login, displays the username. If not, displays "Not Logged In". No attribute required.

```
nbb,m ,
```

### Conclusion

For additional technical information about how to use taglibs, please refer to any of the below resources:

- Javadocs

### Contributors

The following persons have contributed their time to this chapter:

- Sandra Cann
- Turgay Zengin

### Note

Was this EDG documentation helpful? Do you wish to express your appreciation for the time expended over years developing the EDG doc? We now accept and appreciate monetary donations. Your support will keep the EDG doc alive and current. Please click the Donate button.
and enter ANY amount you think the EDG doc is worth. In appreciation of a $35+ donation, we'll give you a subscription service by emailing you notifications of doc updates; and donations $75+ will also receive an Expresso T-shirt. All online donation forms are SSL secured and payment can be made via Credit Card or your Paypal account. Thank you in advance.

[http://www.jcorporate.com/edgdoc.html]

Copyright © 2001-2004 Jcorporate Ltd. All rights reserved.
Chapter 13. Expresso Utilities

Note

If you find this EDG documentation helpful please consider DONATING! to keep the doc alive and current.

Version: Expresso 5.5

Maintainer: David Lloyd
[mailto:dlloyd@jgroup.net?Subject=EDG]

Expresso also provides a number of "Utility" objects - general purpose objects used in many areas of Expresso itself and useful in the applications you develop with Expresso.

Utilities

- File manipulation: Utilities for copying and moving files, and performing certain manipulations on file names.
- String Utilities: Some utilities for more complex manipulation of strings than the standard Java String object allows, including checking of strings for null values and blank values.
- Logging: Classes to interface with the Apache Log4j project, allowing completely customizable multi-level logging to files, database, and other destinations with low impact on performance.
- Operating System Processes: Classes to wrapper a call to an external executable, and to wait for a specified timeout interval for it to finish.

Conclusion

Contributors

The following persons have contributed their time to this chapter:

- Mike Rimov

Note

Was this EDG documentation helpful? Do you wish to express your appreciation for the time expended over years developing the EDG doc? We now accept and appreciate monetary donations. Your support will keep the EDG doc alive and current. Please click the Donate button and enter ANY amount you think the EDG doc is worth. In appreciation of a $35+ donation, we'll give you a subscription service by emailing you notifications of doc updates; and donations $75+ will also receive an Expresso T-shirt. All online donation forms are SSL secured and payment can be made via Credit Card or your Paypal account. Thank you in advance.
Chapter 14. Expresso Security

Note

If you find this EDG documentation helpful please consider DONATING! to keep the doc alive and current.

Version: Expresso 5.5
Maintainer: David Lloyd
[mailto:dlloyd@jgroup.net?Subject=EDG]

Expresso allows a visitor to the site to register and become a member of the community. It is focused directly on creating login information and includes security functionality. Expresso allows for administration of sophisticated security that is in addition to any security provided by your choice of application server product, and portable across all deployment environments.

Introduction

User management is the route that we took with Expresso. Expresso’s security framework came into being long before container-based security became a possibility. Though now we also have contained managed security integrated into Expresso.

This release also includes a major usability refactoring of our self-registration code. As you will see a generic self-registration system is definitely not a trivial task.

Security is applied at several levels in the package:

- Login Security
- Database Object Security
- Application Security

To administer security in your application, use the Expresso Administrative Menu and select the Security selection. This page allows the system administrator to control what users are allowed to access what functions. Users can be registered, assigned to groups, and those groups granted appropriate security access. Each application is then assigned a list of groups which are allowed to access it, and every database object is assigned security by group and method (add/update/delete/search) as well. To use Expresso with no security restrictions: Override isAllowed() for your data entry controller to return true for all situations; otherwise, you can just derive directly from Controller instead of DBController to not use any security systems.

Here are some helpful hints to follow when starting to use Expresso Security:

- Make a copy of the existing Login and Registration controllers to your code base (or modify the existing ones)
- Modify/add mappings in struts-config.xml or your own config file to change the use of the /Login.do and /Registration.do mappings (if you wanting to change the look and feel, change the associated JSPs also).
- Be careful of the inner-dependencies between LoginController and RegistrationController. In some
instances, the transitions are called by class name, so you'll have to update the location of the con-
trollers if you migrate the controllers into your packages.

- Also see section, Tips for Securing a Production Expresso Environment.

**Security Architecture Overview**

Expresso security includes Authentication with every access, and will include Authorization tests as well, if the application has required them.

Authentication involves a login with username and password to ascertain identity. See com.jcorporate.expresso.services.controller.LoginController for details. Identity is stored in a persistent session, in the session provided by the Container servlet engine. On the login screen, the user is given a checkbox to "remember" the login. If the user checks this box, a 30-day-duration cookie is written to the user's browser with the username and password. Clearly, this cookie is appropriate only for relatively low-stakes security. If your application should not allow such a cookie to be written, remove the checkbox from the GUI. During repeated access, first the session is checked, and if that Authentication fails, the cookie is attempted. See com.jcorporate.expresso.core.servlet.CheckLogin for details. Also, if a user tries to access a page and fails because of a lack of login, Expresso from version 5.0 will automatically reroute the request to a login page, and subsequent to a successful login, reroute to the user's original destination.

Authorization checks pertain to every user access if the Expresso application subclasses (Row)SecuredDBObject and DBController. In other words, if your application requires Authorization checks, then subclass either SecuredDBObject or RowSecuredDBObject for your objects that need authorization, and subclass DBController for controllers for controllers in your application, rather than the unsecured DBOject and Controller superclasses. Most of the Expresso administrative code subclasses SecuredDBObject and DBController for this reason. Authorization checks occur whenever a user attempts to access a page which in turn accesses a DBController method, and usually thereby accesses a (Row)SecuredDBObject.

Authentication can be administered via the Security pane, User Information link where new users can be added, and their membership within groups administered. Expresso employs the concept of group membership for all security permissions. A group is enabled, and therefore all members of the group are enabled, and vice-versa.

Authorization can be administered via the Security pane, and links: 'Administer Database Object Security', 'Administer Controller Security' and 'Administer Job Security'. These present a matrix of permissions, where a group of users is given permissions per SecuredDBObject. There are typically 4 types of permissions: search, add, update, and delete. These are directly analogous to the ability to execute SQL commands 'select', 'insert', 'update', and 'delete' against the table associated with a given SecuredDBObject. Administration of row-based permissions is typically handled by the application, showing row permissions to users just as it displays row content.

**User Authentication and Authorization**

User management is the route that we took with Expresso. Expresso's security framework came into being long before container-based security became a possibility. Though now we also have contained managed security integrated into Expresso.

Version 5.0 also included a major usability refactoring of our self-registration code. As you will see a generic self-registration system is definitely not a trivial task.

The user security features in Expresso include:
• Allows a user to register and obtain a username and password
• Provides required fields checking facilities on the quality of the registration
• Mechanism to extend default user fields in Registration Process
• Checks for confirmation of password before making the account active
• Confirms registration by sending a welcome email to the user including the username and password. This email is user-definable, which allows for customizable message
• Method to approve membership before joining fully
• Alternative mechanism to log in with the user's email and password
• Function for users to change their password once they have logged in
• Function for users to update their user profile such as address and contact information
• Function for user to log out
• Mechanism to verify the identity of users by email if a user forgets their login and/or password - and to email the user their login and a new password
• User's Address and contact information is available for contact member directory
• Ability to add Users to User Groups which is used to define user roles.
• Facility to grant Database Object Security to User Groups.
• Provides user authentication for all member-related services
• Provides user membership that can be extended with additional components. i.e. by other applications such as eContent's ability to protect resource access and prevent non-members seeing content, or through custom development of Expresso based applications

If you want to have to avoid the Expresso Login mechanism, then it boils down to this. Servlets depend on a "session" linking the client to the server. Expresso login mechanism simply stores a username in the session, after authenticating it. So, if you need to use some other login mechanism, what'd you'd do is write a class that intercepts a new session being created, and then set the username variable just like Expresso's Login controller does.

Authentication

The first layer of security in Expresso is the Authentication - this is the process of identifying the user and verifying that the user is in fact who they say they are: At it's simplest level this is done via a user name/password login process, but is "pluggable" to allow more sophisticated authentication schemes, integration with LDAP servers, etc to easily be added.

If appropriate, authentication information can be stored on the client as a cookie to enable the user to bypass the manual login process when they next visit the application.

Login requires a login name and password for all secured functions. Some functions may be unsecured - that is, any user may access them without logging in, but ordinarily most functions will require a login. If the user goes directly to the URL of a function that requires a login, but has not yet logged-in, they will be presented with the login screen, and upon a successful login will continue on to the function they originally selected.
A successful login establishes a session - a session is maintained until the user either logs out, leaves their browser (e.g. exits the browser application), or until the server is restarted for some reason. (Sessions will also time out after a period of inactivity).

Expresso also has an **auto-redirect feature** to redirect the user to where they logged in from.

## Authorization

Once a user is identified and associated with a session, their authorization is determined by identifying which user groups a user belongs to. Similar to user groups in the Unix environment, or to "roles" as defined by many application servers, groups can however be nested within other groups to make management of extremely large user communities very efficient. Permission is then granted at any of several levels to the group, and thus to the members of the group. Permissions can be assigned at:

1. **Database Object Level**: Assigning permission to add/update/delete/search for any database object (table) in the application.

2. **Controller Level**: Allowing access to all or selected "states" of each Controller in the application, controlling precisely which functions specific groups of users are allowed to perform.

3. **Job Level**: Allowing access to specific background jobs.

4. **Custom level**: Allowing access at a customized level for a specific application: for example, allowing users to subscribe to different discussion groups for a forum application (such as eForum).

## Servlet Engine Container Authentication

CheckLogin has been modified to allow for Expresso to get the currently logged in user as far as the container is concerned and hook into it's own Security system this way. Expresso now will utilize getUser() from the servlet engine to log the user in that way. It uses standard Servlet API's so it shouldn't be a problem for various Servlet engines. The roles the user receives are still Expresso-based.

As of Expresso version 5.0, we have successfully tested with integration with Tomcat's servlet engine authentication. We go about this simply by checking the user name with the Session.getUser() call to the standard Servlet API. We are in the process of getting more data about container compatibility, but if you require Container-based authentication, trying Tomcat may assist you in finding what you need. Even with container security integration, we will keep the current database tables as the default implementation (although pluggable security matrices are in the works) since they've been SO convenient and nice to work with. As of this writing, sometimes container integration can be a major hassle, especially when switching from container to container. Staying container independent at this phase, and slowly integrating in as container spec conformance comes about seems like an appropriate plan.

## Filtering

Field data entered into an application can have "filters" applied to it automatically to avoid attacks on system security by embedding information in the entered data (such as Javascript or other scripting languages).

## LDAP Integration

A LDAP component will be available as part of the Expresso Enterprise which includes commercial extensions to Expresso. Individual components of Expresso Enterprise may be purchased separately. A release of Expresso Enterprise will be available sometime in the next quarter. Enables a Tomcat based Ex-
presso solution to handle single sign on (SSO) for all types of web content, from jsp to html/doc/etc, using an LDAP-server. It also enables SSO for multiple web applications under a single Tomcat instance. Users: Can change their password transparently using regular Expresso functionality. Admin: Uses export tool after setting up or changing groups. User information is handled transparently. With some minor coding this can be made to work under any J2EE compliant web container.

SSL and Web Applications

As of Expresso version 5.0, the ability to utilize the ability to enter and exit SSL pages has been automatically added. In this section we cover the basic installation concepts as it applies basic expresso configuration, and then we continue with how to utilize these new features in your own code.

A Super Quick Introduction To SSL

As of Expresso 5.0, the ability to utilize the ability to enter and exit SSL pages has been automatically added. Expresso can now set various states in a controller as secure, by simply in the constructor, call myState.setSecure(true). If this is true, then Expresso will automatically redirect to SSL protocol (or back) without losing session! This is the most critical component in an eCommerce application. The best part about it is that this is backwards compatible. If you do not set useSSL in your context configuration, then even if a state is ‘secure’ then Expresso will NOT attempt to use SSL.

Why SSL? SSL is an encrypted protocol that allows you to pass data between the webserver and web client without the myriad of computers sitting between you and the webserver being able to find out what communication took place. This is extremely important for any situation where you may have information that you do not want other people to steal just by listening in on your web transaction. Examples include Credit Card information, passwords, personal registration information, and many other possibilities.

Why isn't everything done over SSL then? The biggest answer is price. To set up a secure webserver you have to do several things. First is acquire a digital certificate from a certificate authority such as http://www.verisign.com or http://www.thawte.com/ These cost money for each server you run this one. The more important aspect, however, is that encrypted connections require A LOT more CPU power than a standard non-encrypted password. If a site is completely encrypted, it will be must less responsive than a non-encrypted site.

Therefore, before you really use SSL, you need to decide where in your application you require it. Examples are login pages, online store checkout systems. For the latter example, for instance, you wouldn't necessarily want to encrypt all data between you and the customer concealing what was in their shopping card. But you WOULD want to encrypt the page where the customer includes the credit card.

One final consideration when setting up Expresso security is that you will want both the page that prompts for the secure information AND the page that processes that secure information encrypted over SSL. Otherwise Expresso’s features will be unable to automatically compensate for the user’s data entry.

Configuring Expresso to use SSL

Configuring your application server.

The steps to configure your application can vary widely. The basic steps are to acquire the appropriate cryptographic libraries and then set up your application server to utilize these libraries.

If you are using Expresso complete or the Tomcat servlet engine, you can download the JSSE cryptographic library at http://java.sun.com/products/jsse/ or if you are running under the JDK version 1.4 or later, JSSE will automatically be included in your system.
You can then configure your servlet engine by following the instructions as provided at the Jakarta Apache site: http://jakarta.apache.org/tomcat/tomcat-4.0-doc/ssl-howto.html. When you are finished configuring your system, you can test it by attempting to go to the link: https://localhost<port number>/ If the Tomcat home or Expresso home page comes up, you've successfully installed SSL onto your web-server.

Setting up the Expresso configuration file

To utilize SSL in your system you need to configure the expresso-config.xml file in two places. The first is to add the following to your <expresso-config> context. After the encryptMode tag, add the following tags:

```
<httpPort>80</httpPort> <sslPort>443</sslPort>
```

These define the ports that your webserver is configured for SSL and nonSSL traffic. For example, on a default Tomcat installation httpPort=8080 and sslPort=8443. Please consult your application server's documentation for the proper numbers to add to this location.

The second location is within the context tag. To do this, add the following attribute to the context tag that you wish to secure with SSL:

```
useSSL="true"
```

Note that you can define which database contexts use SSL at all and which do not. This gives you an added fine-grain access capability within Expresso.

Coding Expresso to utilize SSL

We've talked a bit about deciding what to secure within your web application. All that you have to do to secure your application once you've decided which controller states is to modify your controller's constructor to utilize secured states. In your constructor once you have created your state with the new operator, secure it with the single line of code: myState.setSecure(true); And that's it! You can test your setup now to make sure you are utilizing SSL capabilities as expected.

Application Security

Any object (servlet or otherwise) can be registered as an application. Applications then have security applied to them in a manner very similar to that of database objects. In addition, applications may have a more detailed level of security, called application methods: For example, if a user is allowed to use the ledger posting application, they might be able to post only their own transactions, or to post any transactions. These "methods" are application specific. The Expresso Framework provides a complete system for applying all of these security facilities, ready to add your own application-specific objects into.

Administrative Security

Any of the table maintenance/administrative functions can be secured. For all table maintenance functions, security is achieved by granting group access to certain database objects, as required for the particular function. Database Object Security can be granted in any of the following degrees:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>Add new records to the table</td>
</tr>
<tr>
<td>Update</td>
<td>Update existing records</td>
</tr>
</tbody>
</table>

Table 14.1. Administrative Security
Once appropriate administrative privileges have been granted to a group, it is then straightforward to add administrative users to that group as required.

A special group called "Everybody" is recognized automatically by the system. This group can be used to grant global permissions, such as universal ability to read the job queue, for example. Use extreme caution in granting access to the Everybody group.

For a user to access the privileges assigned to his/her group, they must log in (separate from their regular network login). A user who has not yet logged in is assigned the user login name "Anonymous", which is a member of the "Everybody" group only.

A login is valid until the web browser is closed or until the servlet-runner process is restarted. Passwords may be set explicitly via the user administration program, or may be changed by the individual user on the Login web page using the appropriate form.

Logging in is only necessary for users not already authenticated by a system logon (i.e. Unix logon or NT domain users).

### Database Security

All database objects (which, as we discussed, may or may not have a one-to-one relationship with database tables) have security attributes. A group of users can be granted permission to any combination of Search, Add, Update or Delete security.

This security information is, for performance reasons, read into a cache when the first database connection is made & read from that cache until a change requires it to be re-read. Changing any of the user/group or security tables will cause security to be re-read, and this re-reading takes place as a background thread until it is complete - then the old security information is replaced in a single operation - this prevents partially-read security information causing problems.

Expresso supports row-level authorization. RowSecuredDBObject is the key class. By subclassing RowSecuredDBObject, an application can have authorization checks that vary per row.

The checks within SecuredDBObject correspond roughly to permission checks for an entire database table: users either have access to the entire table or not.

Below is an example of how to programmatically set security for all DBObjects and Controllers in a given schema. You can just override, in your "MySchema.java", the method otherSetup() in superclass Schema.java. This method is run automatically during dbcreate (the "setup" initialization phase that can be run at any time from the admin panel). The example below gives full permissions for any logged in user.

```java
public synchronized void otherSetup(String dbName) throws DBException { // add security for logged-in users
    int numSecs = sec.getNumSecuredObjects();
    for (int i = 0; i < numSecs; i++) {
        SecuredDBObject sec = (SecuredDBObject) DB.newClassObject(SecuredDBObject.class);
        sec.setField(ControllerSecurity.GROUP_NAME, UserGroup.ALL_USERS_GROUP);
        sec.addIfNeeded();
    }
}
```

Checks by RowSecuredDBObject provide row-level authorization, providing UNIX-style permissions for the row’s owner, group, and others. Thus a user can have access to one row, but not another, within the same table. See more information on RowSecuredDBObject the subsection below.

The checks within DBController correspond roughly to pages, since the central "Run" methods of a DB-Controller are typically one method per page. A user either has access to the page (the DBController
Encryption

Both basic and sophisticated encryption is available for securing any data within the system, so that only authorized users can access the specified data. This deals with the issue of users gaining access to the database server directly and attempting to bypass application security in this manner.

Cross Site Scripting Countermeasures

Expresso uses a two-phase (input and output) filtering mechanism. An EscapeHandler does filtering on DB insertion side. Raw text is saved EXCEPT for what is replaced to accommodate a given DB. For example, DoubleQuoteEscapeHandler.java removes double quotes ("), and escapes single quotes. FilterManager does filtering on the output side. There are ‘static’ filters and ‘instance’ filters, which persist in metadata, and only affect a single DBObj filter, respectively.

FilterManager

FilterManager is a singleton class that does translation/filtering of strings when you call getField(). That is, on the way FROM the database, strings are filtered. One of the most common filters is the HtmlFilter. It filters out HTML tags that might be entered by users. A malicious user can enter codes which get returned to a web browser, and these codes can cause the web browser to act on them in a way contrary to what the site author intended—it could be a breach of security. For more on these see: http://www.cert.org/tech_tips/malicious_code_mitigation.html; Understanding Malicious Content Mitigation for Web Developers.

All Filters implement the 3 methods in the Filter interface, which typically have the following implementations (although your Filter can, of course, do whatever you like):

// Replace control characters with appropriate values, protect against XSS attack

Setting Filter Characteristics in DBObj

By default, the filter class is HtmlFilter.java, and the method is set to 'standardFilter'. To set the filter method for a field in your DBObj, use setStringFilter(fieldname, filtername) within your setupFields() of the object, like

protected synchronized void setupFields() throws DBException { ... setStringFilter("myfield", FilterManager.RAW_FILTER); ... }

(There is also DBObj.setStringFilterOnAll() if you want to have the same filter on all fields. Call this AFTER adding all fields in setupFields().)

Note that setStringFilter() sets the filter PERMANENTLY FOR ALL INVOCATIONS for your class because it sets the "metadata" information, which is held statically. If you need some special filtering, on a permanent basis, a different Filter class can be set by getting the metadata and setting DBField.setFilterClass() like

protected synchronized void setupFields() throws DBException { ... fieldMeta = (DBField) getMetaData().getFieldMetadata("myfield"); fieldMeta.setFilterClass(HtmlPlusURLFilter.class); ... }

In order to set the string filter for just an instance, you can use DBObj.setFilterClass(Filter); note the contrast: above was DBField, which is permanent in metadata, and here we are talking about DBObj.setFilterClass(Filter), which only pertains to the instance, the DBObj instance explicitly referenced. For example, let's say you have a field which is edited (after initial storage) as a text area, and you want its contents to show up for editing with regular line feeds, instead of <br> for breaks in the text area. You can do the following in some controller which creates the Input for the text area:
Filter old = existing.setFilterClass(new RawFilter()); summaryStr = existing.getField("MyField"); // restore existing.setFilterClass(old);

DBObject.setFilterClass(Filter) changes the Filter class on the instance, but not the filter method, which is specified on the metadata statically. Here we have used RawFilter(), which returns the raw text, no matter what filter method is called on the Filter, no matter what field is accessed. Again, the Filter class applies to all fields. There is currently no way to specify a method name in a per-instance way, or to specify a per-field filter per instance, but from the usage above, such fine-grained approaches may not be necessary.

In addition to HtmlFilter and RawFilter, there is XmlFilter, suitable for generating XML (if you are not using XSL for that), and also HtmlPlusURLFilter. HtmlPlusURLFilter subclasses HtmlFilter, adding an ability to recognize user-entered text which begins with 'http://' or 'www.', etc., and automatically create an anchor (<a>) tag from it.

SQL Injection Countermeasures

Expresso filters and validates all field values that are getting sent to a database. To do this, we put the code through a database escape handler. The escape handler's job is to filter various escape codes and put the appropriate literal value back into the database SQL. Current implementations of the escape handler are:

- DefaultEscapeHandler - Used by databases such as Postgresql. Escapes ' as \\
- DoubleQuoteEscapeHandler - Most common filter. Escapes ' as \\
- MySQLEscapeHandler - mySQL specific escape handler. Performs several escape sequences

The particular escape handler is set in the jdbc section of the expresso-config.xml file.

RowSecuredDBObject

Checks by RowSecuredDBObject provide row-level authorization, providing UNIX-style permissions for the row's owner, group, and others. RowSecuredDBObject extends the idea of group permissions beyond the UNIX-style permissions, in that RowSecuredDBObject allows multiple groups to have different permissions. For example, imagine a row where the owner of a row has read/write/administrate permissions, "others" have just read permissions, group A has read/write permissions, and group B has read permission. A person who is a member of both group A and group B will have write permissions for the row. In a sample application, users could be members of different groups, and could read each other's rows but could not edit row data that their group had not created. This happens without much effort because the default permissions for a row make it writable by the primary group (see User.getPrimaryGroup()) of a user, but only readable by others. Furthermore, when getting a list of rows from a search, only the rows that are readable are returned to the requesting user.

Fundamentally, the idea is that you don't have to do much extra. You establish your groups and group memberships, and then the objects that these groups create have default permissions, allowing reading by others, but writing only by group members. Here is a code snippet for using permissions to determine what to show:

    // add link to edit if we have privileges if (myObject.canRequesterWrite()) { Transition trans = new Transition( ... matrix", MatrixAction.class, PROMPT_EDIT_SCORING_MATRIX); trans.addParam(Node.NODE_ID, nodeId); response.add(trans); }

Cookie Encryption

If you use Persistent.setClientAttribute() all cookies sent to the client are encrypted. This significantly increases the ability to guard against cracking the contents of the cookies to reveal items such as pass-
words.

Note that encryption does not prevent an attack called "replay". In the replay attack, a malicious user grabs a cookie, doesn't crack it's contents, but does send it back to the web server to, for example, gain access as the user of the encrypted cookie. For this reason, sometime's it's better to prevent setting of cookies at login for high security websites.

Tips for Securing a Production Expresso Environment

Expresso has many security features that enable rock-solid deployment options. However, it should always be kept in mind by all system administrators the number one rule about security:

*Security Is a Process*

What this means is that any application server should be routinely audited, and there will always be new security enhancements available. Routine updates, audits and patches will always be needed. If a company tells you that they have bullet-proof, unpenetrable security..... They're Lying! This list is the start of a checklist to help you secure your Expresso environment:

- **Run your application under the most recent version of the JVM as possible**: There are always bugfixes that are included in every new release, and some old bugs may be able to be exploited by a hacker.

- **Install strong encryption**: Strong encryption helps prevent attackers from stealing the actual passwords that you may use to log into your system. To get this going, you must set strongCrypto="Y" in the expresso-config.xml file. Then download and install the bouncy castle cryptographic provider. This cryptography library can be found here [http://www.bouncycastle.org/].

- **Run your expresso setup under a java security policy**: As of JDK version 1.2, Java has some extremely fine-grained security possibilities. Things to do with the policy include but are not limited to:

  - Limiting the classes that can access to the file system to only those necessary. This will mainly be limited to the configuration classes and that should be it.

  - Limiting the classes that can make network connection. You also have the ability to determine what ip address you will allow connection from. This is very good if you are running your application server as dependant on another web server (See the Apache Jakarta project [http://jakarta.apache.org/] for what I mean by this). By and large, you'll only want connections coming in directly from the web server. You'll also want to make sure that only the JDBC client classes, and the classes responsible for connections to the webserver are the only classes allowed to make outbound connections. This way a hacker cannot add a a small class that sends passwords directly to his own site. See your JDK documentation for more information on creating a policy file.
• **Secure the file system:** Make sure that the only users that have access to the Application server directories are root for full control, and the account that the Application Server is going to run under for read only and execute control. If you have some directories where the Application Server is going to write files, open that directory, but remember, wherever a file can be written to your hard drive is a point of entry that a hacker may come in through. So keep the file system permissions are limited. Also make sure that you have completely restricted access to the Expresso config directory.

• **Secure the database JDBC drivers:** Depending on what machine the database is located on, this may be different than the "Run under a security policy" step. Make sure that the db drivers only accept connections from certain sources, and block out all other connection attempts.

• **Employ Firewalls between the internet and your database:** It's embarrassing if your web server gets defaced..... It's a legal problem if your entire credit card database gets read by a hacker.... Protect your database with every available means.

• **Run the Expresso database user with restricted privileges:** Although Expresso does a fine job of saying what users can and can't add/update/delete records. It would be possible to steal the Expresso database config file, log directly into the database and issue a command something like this: DROP TABLE CUSTOMER_ORDERS; COMMIT; So make sure that there are only permissions granted to the Expresso app that coincide with what is absolutely necessary to run.

• **Remove all unnecessary services:** Some servlets and services may not be used in a production environment. For example, the runSQL servlet won't probably be used directly. Or maybe your application doesn't use the file download features. Remove every service on your application server that is not required for the web site to function properly.

• **Remove all default passwords:** The Openhack Project [http://www.eweek.com/article2/0,3959,741388,00.asp] got cracked simply because somebody forgot to change a default password that comes with Oracle. Make sure you change all passwords for default user accounts. Or better yet, create an Administrative account under a different name, and then delete the default user accounts all together. That way a hacker _has_ to start from scratch.

• **Run security monitoring applications:** Most determined hackers will eventually get in. Its just a matter of time. Its best to have a series of Intrusion detection systems in place. This way you can get notified if suspicious activity is going on. Or if you have the budget for it, check out the monitoring service that Counterpane Internet Security [http://www.counterpane.com/] offers.

• **Run External Audits:** It's next to impossible to see your own mistakes. Have a security firm (or two) do at least a basic audit of your webservers system. This will at least catch some of the more common mistakes you might make in your installation.

• **Make sure that strong crypto is used with the library:** By default this is turned on, but unless its very necessary, please don't turn it off.

• **Don't use Windows 95 or Windows 98 for your server!!!!!** I'm sorry that this even has to go into this article, but many people think that Personal Web Server / Win98 is secure enough for production use.. THIS IS COMPLETELY UNTRUE! People can gain complete control of a Win95/98 system faster than you can say "What Happened?" If you have to use windows for your server machine, at the very least use Windows 2000, or better yet, NT4 with Service Pack 6. [I know that NT4 is older, but MS added a number of untested features into Win2K that probably won't be ironed out security wise until Service Pack #2 or #3. Once there's enough service packs out there... go for it and ditch NT4]

I repeat that security will always be a process. You will always be having to modify things in your server to make things more secure, even if its just to apply the latest security patches available from various sources. But this document should give you a decent starting point of what to deal with when using a production security environment. If you are unfamiliar with creating production-quality secure ma-
chines, please do a search on the internet for things like "Security FAQ" and check out books on security to get a better idea of what you're guarding yourself against, and how best to do it.

Conclusion

Here are a few areas of enhancements coming in new releases:

- The ability to export security settings so you can set all the security settings you want on your test environment, upload the file to your production environment when you're ready to go online and let 'er rip. Probably displaying the current security settings in an xml file would be most doable...
- You'll see integration with JAAS coming up...

Contributors

The following persons have contributed their time to this chapter:

- Larry Hamel (JGroup Expert)
- Mike Rimov
- Mike Traum (JGroup Expert)

Note

Was this EDG documentation helpful? Do you wish to express your appreciation for the time expended over years developing the EDG doc? We now accept and appreciate monetary donations. Your support will keep the EDG doc alive and current. Please click the Donate button and enter ANY amount you think the EDG doc is worth. In appreciation of a $35+ donation, we'll give you a subscription service by emailing you notifications of doc updates; and donations $75+ will also receive an Expresso T-shirt. All online donation forms are SSL secured and payment can be made via Credit Card or your Paypal account. Thank you in advance.

[http://www.jcorporate.com/edgdoc.html]

Copyright © 2001-2004 Jcorporate Ltd. All rights reserved.
Chapter 15. Expresso Event System

**Note**

If you find this EDG documentation helpful please consider DONATING! to keep the doc alive and current.

**Version:** Expresso 5.5

**Maintainer:** David Lloyd

[mailto:dlloyd@jgroup.net?Subject=EDG]

*Expresso Event Notification is a subsystem that handles letting key users know when important events are completed, or when problems have occurred. Utilizing the standard JavaMail API and the Java Activation Framework, Expresso provides easy access from your application to email.*

**Introduction**

The event-driven processing feature allows any system event (e.g. the publication of a report, a system error, a server being restarted) to trigger email notifications to be sent to a specified list of users (both local and remote (internet)). Any message or attachment can be included with the notification, and an event can be set to trigger other events if required (e.g. a data load might trigger a report to be run). The list of messages and users to be notified is maintained in a series of database tables. When a server-side task, such as loading data into a data warehouse, is completed, it triggers an event with a specific code. The event is then broadcast to a list of specified users that are subscribed to receive the event.

The goal of the Expresso event system is to send email messages to a specified number of people when a specified event occurs. The configuration of the Event system is done through database tables, while the triggering of an Event can take place in one line of code. Events are always asynchronous, and depending on how the system administrator sets up the website, when events are triggered, it is possible that nothing will happen.

An example of an event is a system error - this is a pre-defined event that comes with the system. This event can then have a list of users associated with it that get notified when the event occurs, in the case of a system error, you would probably want to notify your system administrator.

The developer then simply has to specify that an event is triggered and the appropriate notifications are sent out automatically by the system.

**Event Conditions: Success or Unsuccessful**

Every event is either "successful" (such as a load completing correctly) or "unsuccessful" (such as a load aborting due to a database problem). Successful events can have a different group of users from unsuccessful events - for example, a number of people might want to know that the data load is completed, but you might want to tell the system administrator only if the load fails. Of course, the same people can also be notified of both success and failure if so desired.

**Message Content**

The event e-mail contains a message from the server-side object, which can give details about the event (for example, how many records got loaded, or what the exact error message from the database was in the case of a failure).
Event Configuration

The Event configuration tables can be reached through the Operations page from the main Expresso frame. The two tables are:

- Event Definitions - This table contains a list of all defined events. An example of a defined Event is SYS_ERROR
- Event Email Recipients - This table contains data on who to send email notices to when a particular event triggers

Event Definitions and Event Email Recipients are connected with a Master/Detail relationship.

The Event Definitions table is fairly straightforward. The two fields are event name and event description. A pre-built example is SYSERROR which has the description of "System Error." The name SYSERROR is the name of the event that gets triggered in code as will be demonstrated shortly.

The Event Email Recipients contains the rest of the Event configuration information. When adding a new recipient, you decide on who to send the notification to, what event, and why. For SYSERROR, for example, you could select "System Error." from the dropdown box for Event code, and 3 for the user ID (Usually Administrator).

Now the last part of the Event Email Recipient is a little bit different. Each Event has a status associated with it: Success or Failure. For example, if a user successfully downloaded a file, the download controller could trigger a DOWNLOAD event with a success status, whereas if the connection died for some reason, the download controller could trigger a DOWNLOAD event with failure status. Each of these statuses is configurable with who gets what notifications. For example, you could send a DOWNLOAD success event to an automated mail archival system for statistical reference, while you would want to send failure events to the administrators so they can see if there are an unusual number of failures occurring.

---

**Triggering an event from code**

You can trigger an event from code with very little code:

```java
import com.jcorporate.expresso.core.misc.EventHandler;
EventHandler.Event("default", "SYSERROR", "Successful Test Message", true);
```
EventHandler.Event is an asynchronous call. If you watch the runtime logs, you will see the mail going out a few seconds after the call. The parameters for the call are:

1. The data context (or db name) in this case default
2. The name of the event to fire as defined in the Event Definitions Table
3. The message to dispatch when the event is fired. This will be the text of the email message that the Event dispatches.
4. The final parameter is set to true if this is a 'success' message, and false if it is a 'failure' message.

**EMail Integration**

The framework has email integration as part of its base feature set. Instead of the application programmer having to develop his/her own email notification system, there is a base level email integration built right in. From that base email class, there are several different classes that make use of the email notification class.

Some facilities provided for email include:

- **Event notifications:** Discussed above, Expresso's event notification mechanism uses email by default to inform users when specific events, including system errors, occur.
- **User notifications:** The ability to send email to particular users.
- **Login Verification:** An optional portion of the registration process allows users to be emailed in order to verify their email address before authorizing their account for login, assuring your user data is up-to-date.
- **Attachments:** An easy mechanism is provided to attach one or more files to any email being sent to a user, the contents of which can be created by your custom application.

**Event notifications**

The Job class, a class to provide cron or batch like functions, has a mechanism for emailing certain parties that can be customized to alert the certain recipients when the job has started, when the job has finished, and/or when the job encounters in error.

**User Notifications**

There are times that an email needs to be sent to particular users and to those users explicitly. The EventHandler and UserInfo objects can work together to send an email to the user. An example of this is in com.jcorporate.eforum.dbobj.ForumMessage where it sends an email to everyone subscribed to get email in a forum.

EventHandler.notify(dataContext, // the database context
tSubscribe.getFieldInt(ThreadSubscribe.FLD_UID), // the user id to send email to msgSubject, // the email subject message); // the email message

**Login Verification**
The User, DefaultUserInfo, and UserLDAP classes, which handles the registration and manipulation of users within the Expresso framework, has an email conduit already setup. This email conduit allows verification of the user, deterring spammers from registering, and keeping the database up-to-date.

**Attachments**

The [User](#) and [DefaultUserInfo](#) classes also have an email attachment feature. This makes it easy to attach one or more attachments to any email being sent to a user. As with most features in Expresso, this can be customized to the specific application that is being developed.

**Conclusion**

**Contributors**

The following persons have contributed their time to this chapter:

- David Lloyd (JGroup Expert)
- Mike Rimov
- Mike Traum (JGroup Expert)
- Tino Dai
- Sandra Cann

**Note**

Was this EDG documentation helpful? Do you wish to express your appreciation for the time expended over years developing the EDG doc? We now accept and appreciate monetary donations. Your support will keep the EDG doc alive and current. Please click the Donate button and enter ANY amount you think the EDG doc is worth. In appreciation of a $35+ donation, we'll give you a subscription service by emailing you notifications of doc updates; and donations $75+ will also receive an Expresso T-shirt. All online donation forms are SSL secured and payment can be made via Credit Card or your Paypal account. Thank you in advance.

[http://www.jcorporate.com/edgdoc.html]

Copyright © 2001-2004 Jcorporate Ltd. All rights reserved.
Chapter 16. Internationalization

Note

If you find this EDG documentation helpful please consider DONATING! to keep the doc alive and current.

Version: Expresso 5.5
Maintainer: David Lloyd
[mailto:dlloyd@jgroup.net?Subject=EDG]

There are a number of facilities in Expresso that facilitate developing applications that are fully internationalized. Each Expresso application may have one or more MessagesBundle.properties files. Expresso itself has a collection of these files, one for each language supported: for example, the french translation file is MessagesBundle_fr.properties, following the standard Java conventions for local language files. In order to create an internationalized application, all messages and strings displayed to the user (or used in emails or log messages) should be stored in the local language files.

Locales

Every time a user logs in to Expresso or an Expresso application, a locale is established for the user. This locale is determined by first examining the user preference options for the user. If there are no preference values specified for the user, then the user’s browser settings are used to determine a language preference. If still no language can be determined, then the defaults set for the application are used, if any. If no such default was specified, english will be used.

If the user changes language preference while logged in, they must log in again (either explicitly, via the Login controller, or implicitly via a cookie) in order for the change to take affect.

Character Sets

A specific character set can be specified for each language. In the local language file for each application and supported language, the "charset" key is specified with a character set as it's string - this character set is applied to each page that includes the <html:html> tag automatically. In this way, the character set is determined before all other strings on the page are displayed.

Internationalization and DB Objects

Every instance of the SecuredDBObject class has the capability to support internationalization easily as well. When the setDescription(String) method is called, the String specified is assumed to be a key for the local language file. If no such key is found, the string itself is used as the description - this means you can use the description directly instead of a key if you do not need to support internationalization.

The same is true of the descriptions for each field, as specified with the "addField" method within the "setupFields" method of each DB Object. The field description is assumed to be a key within the language file, and if the key is not found, the string specified is used directly.

Additional strings and messages can be internationalized within a SecuredDBObject. You can use the getString methods within the SecuredDBObject to access strings, with or without arguments.
Internationalization in Controllers

Controllers provide easy access to internationalization via getString methods - these methods can be used to specify a key into the appropriate local language file, and optionally to provide arguments to this string (as described in the Javadoc of the Messages class in the com.jcorporate.expresso.core.i18n package.

Internationalization in JSP's

Expresso provides an extended version of the Struts message tag in order to make it easier to specify strings to be used in JSP pages. The extended tag takes an additional optional attribute: schema. The schema attribute specifies the schema class file of the application whose message file is to be used to look up the specified key. For example, to access the string with the key "description" in the eForum language files:

```xml
<bean:message key="description" schema="com.jcorporate.eforum.ForumSchema"/>
```

Once the message tag has been used with the schema attribute once, subsequent uses of the tag on that same page will default to the same schema file. Make sure, however, that the first tag is not in a conditional section - otherwise it may not set the schema default properly for subsequence calls to the tag. You can of course always use the schema attribute explicitly each time, but this gets verbose in a hurry.

In addition to the message tag, the TableHead tag in the "expresso" tag library also directly supports internationalization: instead of the usual "value" attribute, you can use the "keys" attribute instead - in this case, each of the values specified in the string will be considered a key to be looked up in the language file: so, instead of value="one|two|three" simply resulting in the table headers of "one", "two", and "three", keys="one|two|three" will look up each of those values as keys, and use the resulting strings as the table headers.

The DBMaint tag will use the local language version for the description of the database object if the "label" attribute is omitted.

ISO Valid Values

Expresso supports internationalised valid values in the class ISOValidValue which is a subclass of ValidValue. Both classes belong to the com.jcorporate.expresso.core.dbobj package. The ValidValue class is a simple label value Java Bean, that is it has a label and description attribute. The ISOValidValue class has the same attributes but supports internationalisation of the label attribute. In particular this class has canonisation support. This class has method to convert the `description' field to an i18n value from a resource bundle. Both classes support accessor and mutator method as well as default constructors.

```java
public ISOValidValue(); // Default constructor
public ValidValue();

public String setValue() ; // Accessor
public String setDescription();

public void setValue( String v) ; // Mutator
public void setDescription( String d );
```

The ISOValidValue supports canonisation. In order to do conversion of the `description' field with a resource bundle the `Schema' class name and `Locale' must be supplied, as well as `prefix'. The prefix parameter avoids possible namespace conclusions with the description attribute which we would use as key into the resource bundle. The most obvious candidate for a prefix string is a fully qualified class name.
public void canonize( String schemaClass, Locale locale, String prefix )

This is method will attempt to convert the `description' attribute into a localised value string from a message resource bundle. If the `prefix' parameter is not null then the `description' attribute is prepended with a full stop "." to the original description string to make a new look up key. Otherwise if the `prefix' parameter is null then original description is the look up key. The key value is used to look up the description from a resource bundle.

The method is equivalent to functionally calling `Messages.getString( String schemaClass, Locale locale, key, new Object[0] )'. If the key is found in the resource bundle then the value string becomes the new description.

So we if have the following in enumerated values

```java
SORT_BY_DATE
SORT_BY_SCORE
```

and I have a Java class `org.acme.test.SortOrder' then I might have use the class name as a prefix which results in a look up like this:

```java
org.acme.test.SortType.SORT_BY_DATE
org.acme.test.SortType.SORT_BY_SCORE
```

All I have to do is place these keys inside a resource bundle to internationalise an application (e.g German).

```java
org.acme.test.SortType.SORT_BY_DATE = Sortiert von Datum
org.acme.test.SortType.SORT_BY_SCORE = Sortiert von Punkten
```

This is exactly what we mean by canonisation or conversion. There are more additional convenience `ValidValue' APIs

```java
public void canonize( String schemaClass, Locale locale, String prefix )
    public void canonize( String schemaClass, RequestContext context, String prefix )
```

The `ISOValidValue` class also has constructors that will automatic canonise the description upon initialisation. These are convenience functions and because they are constructors there are thread safe.

```java
public ISOValidValue(
    String schemaClass, Locale locale, String prefix,
    String label, String description )
    public ISOValidValue(
        String schemaClass, RequestContext request, String prefix,
        String label, String description )
```

We can use the `ISOValidValue` object in our own `getValidValues` method in a DBObject to generate hard-coded look up values.

```java
public Vector getValidValues( String field ) {
    if ( field.equals( "Fruits" )) {
        Vector list = new Vector();
        String schemaClass="com.my.own.schema";
        String prefix = this.getClass().getName();
        list.add( new ISOValidValue( schemaClass, requestContext, prefix, "1", "Apple" ));
        list.add( new ISOValidValue( schemaClass, requestContext, prefix, "2", "Pear" ));
    }
    return list;
}
```
list.add( new ISOValidValue( schemaClass, requestContext, prefix, "3", "Orange" ));
list.add( new ISOValidValue( schemaClass, requestContext, prefix, "4", "Banana" ));
return list;

else
    return super.getValidValues( field );
}

Or we can utilise the automatic lookup feature in the DBObject class, which has an alternative method to retrieve valid values that uses the newer ISOValidValue classes. This method is called getISOValuesDefault().

    public Vector getValidValues( String field ) {
        if ( field.equals( "Product_Id" )) {
            Vector list = this.getISOValuesDefault( "Product_Id", "Product_Desc" );
            return list;
        }
        else
            return super.getValidValues( field );
    }

In the DBObject.getISOValuesDefault method, valid values are automatically cached by key name. The cacheName is built using a cache key using only the fully qualified java class name of the DBObject. For internationalisation the cache name needs to have the locale specifically added to it. So that the valid values are cached for each language and territory.

    String cacheName = this.myClassName + locale.getLanguageCode() +"_"+locale.toString() + ".validValues";

So that we get some cache names like this:

`com.acme.test.SortType.en_gb.validValues`
`com.acme.test.SortType.fr_fr.validValues`
`com.acme.test.SortType.fr_ca.validValues`
`com.acme.test.SortType.es_es.validValues`

In the DBObject.getISOValuesDefault method the fully qualified class name is used as the `prefix' to get the valid value as described above. Also the schema class name and the locale is retrieved from the DBObject, because it has these attributes already stored. So the canonisation for the product type might be

# (English)
    org.acme.test.Product.1 = 1300RPM Washing Machine
    org.acme.test.Product.2 = Portable Television

# (German)
    org.acme.test.Product.1 = 1300RPM Waeshmaschine
    org.acme.test.Product.2 = portierbares Fehershen

And that is all there is to internationalisation of valid values.

Conclusion
Contributors

The following persons have contributed their time to this chapter:

- Peter Pilgrim
- Mike Rimov
- Mike Traum (JGroup Expert)

Note

Was this EDG documentation helpful? Do you wish to express your appreciation for the time expended over years developing the EDG doc? We now accept and appreciate monetary donations. Your support will keep the EDG doc alive and current. Please click the Donate button and enter ANY amount you think the EDG doc is worth. In appreciation of a $35+ donation, we'll give you a subscription service by emailing you notifications of doc updates; and donations $75+ will also receive an Expresso T-shirt. All online donation forms are SSL secured and payment can be made via Credit Card or your Paypal account. Thank you in advance.

[http://www.jcorporate.com/edgdoc.html]

Copyright © 2001-2004 Jcorporate Ltd. All rights reserved.
Chapter 17. Asynchronous Request Processing Framework

Introduction

There are times when you need to process a task that is going to take a long time. Examples are processing credit cards, performing reporting tasks, retrieving a user record from remote databases. For each of these items, it is desirable to give the user immediate feedback of some sort, while the task continues. Or if the task has progressive results, it would be even more desirable to display those results as they become available. Imagine how much nicer it would be to work with DBCreate if you could see the tables being created on the browser screen instead of waiting for 30 seconds before you saw any result at all!

This is where the Async Processing Framework comes into being. Although the default setup uses a servlet to provide lifecycle events, it is certainly not restricted to a servlet environment. The interfaces provided are quite generic and can be adapted to nearly any situation such as JMS message queuing, RMI dispatching to remote machines, or even command line multithreading tasks.

How it works

The Big Picture

Let's talk about the basic flow of events on how the Async process model works. Below is a sequence
So here's a step by step walkthrough of the diagram:

1. [Factory method] The Controller wants to have a process performed asynchronously. So it calls AsyncProcessorFactory.getAsyncProcessor() to get an instance of the processor.

2. The Controller then constructs an object that implements the AsyncProcess interface and hands that to the processor by calling AsyncProcessor.addToQueue()

3. The AsyncProcessor queues the process and returns an AsyncTicket object. Think of AsyncTicket as a claim ticket.

4. When the AsyncProcessor has available slots, it called the queued object's process() method.

5. After a period of time, the Controller will want to see how its AsyncProcess is doing. So it calls AsyncProcessor.getStatus() by giving the AsyncProcessor the AsyncTicket it received originally by calling addToQueue()

6. The AsyncProcessor now checks the status of the running process. It then builds an AsyncProcessResult object and returns it to the controller.

7. If the status of the process was that it was finished, the AsyncProcessor adds the result of the process into the AsyncProcessResult object and then removes all references to that particular ticket. Any future claims on the ticket will throw an Exception.

8. The controller then, based upon the object's status either renders the Status object or the Result object. [Or possibly both?]

9. If an exception was thrown by the process, it is stored in the AsyncProcessResult object too, and the controller can then decide what to do with the exception.

Now let's reiterate what is done strictly from the controller standpoint. This is important as it will affect the general flow of your app while the process is running:
1. The controller gets the async processor and queues the process.

2. The controller checks to see if the process is done.

3. If the process is not done, then the controller sends the client to a status page that automatically refreshes after, for example, 5 seconds. The ticket should be saved in the user's session.

4. Now that the page refreshes, the controller once again checks to see if the process is done. If not, repeat the previous step.

5. If the process is done, then retrieve the result and display the result.

6. If the process threw an exception, then the exception should be handled appropriately.

**A Real World Example**

Time to put this in a plain English use case. Joe wants to order "Awesome Book" from Bookazon.com. He's navigated the checkout system and now wants to complete his order. Bookazon's back end for credit card processing is a bit old, however. It has a bank of modems hooked to a serial port back to the processing servers. Each time a card is validated, one of the modems has to dial out to the validation server and validate the results. Obviously this is going to take a few seconds, and we all know what fidgety end users do, especially when "Awesome Book" and their credit card is on the line!

This is where the async processing system can come to the rescue. "Mack Programmer" implements the entire dialing and validation sequence in a AsyncProcess implemented object. So what happens is the following sequence of events. Each physical even is cross-referenced to a step in the previous list.

1. The checkout controller creates an order processor process and submits it to the processing queue. [Step 1]

2. Since the dialout takes a long time, we don't even try to see if the process is complete right after it has been submitted. [Skip step 2] So instead we display a fancy "Your order is being processed, please please please please don't hit the back button!" page. Instead of sitting there fidgeting thinking of hackers and all that proverbial trash, Joe is nicely occupied with the message and fancy Javascript animations on the page. Much less temptation to hit the back button! [Step 3]

3. After a couple of seconds, the browser refreshes to see how the process is going. Since dialing takes a while, perhaps we'll want to have alternate screens that come up for more eye candy... [Step 4]

4. After a couple of refreshes, the authorization code finally comes through! We send Joe onto to the 'success' page and display his order. Joe is happy, we're happy. [Step 5]

5. If Joe blew it on the credit card number, then we can show the authorization failed message and prompt for a re-input of the card. Joe now may not be happy, but we're still happy because Joe didn't hit the back button while processing was taking place! [Step 6]

**The Details**

As you can see, there are quite a few objects interacting here to make this whole dance take place. The following is a UML static structure diagram:
The color coding of the UML diagram is this:

- Yellow: These are the interfaces that you will be interacting with.
- Green: Concrete classes you will use. In this case, it is mainly the static factory method in the Processor factory class that is used
- Blue: Your own processor class that you will need to implement
- White: Expresso default implementations of the interfaces. You will not directly instantiate or interact with these objects themselves. They are mainly there for reference and assistance for spelunking the depths of Expresso code

Implementing Your Own Async Process

Now that plenty of theory has been covered, let us turn to the task of creating an asynchronous object.

Details TODO

Conclusion

Contributors

The following persons have contributed their time to this chapter:

- Mike Rimov

Note

Was this EDG documentation helpful? Do you wish to express your appreciation for the time expended over years developing the EDG doc? We now accept and appreciate monetary donations. Your support will keep the EDG doc alive and current. Please click the Donate button
and enter ANY amount you think the EDG doc is worth. In appreciation of a $35+ donation, we'll give you a subscription service by emailing you notifications of doc updates; and donations $75+ will also receive an Expresso T-shirt. All online donation forms are SSL secured and payment can be made via Credit Card or your Paypal account. Thank you in advance.

[http://www.jcorporate.com/edgdoc.html]

Copyright © 2001-2004 Jcorporate Ltd. All rights reserved.
Chapter 18. Expresso and XML

Note

If you find this EDG documentation helpful please consider DONATING! to keep the doc alive and current.

Version:Expresso 5.5
Maintainer:David Lloyd
[mailto:dlloyd@jgroup.net?Subject=EDG]

Expresso contains a number of XML capabilities that can be used in building your application. This chapter explores these briefly, and how then can be applied.

XML Configuration Files

The configuration files that specify all of the options for Expresso at system startup are in XML format, and these files can be used to specify additional custom properties for your own application. See the "custom-property" element description in the DTD for expresso-config.xml for details. (This DTD can be found in the WEB-INF/classes/com/jcorporate/expresso/core directory.

XML Data Import/Export

A controller object called XMLController can import data into any DBObject from an XML format, and export data from the DBObject into an XML format file. This controller is accessed from the "Operations" page in Expresso.

XML Output from Controllers

All Controller objects can produce their inputs, outputs and transitions in XML format. This can be used when debugging an application, to examine the elements produced by a particular controller. This XML can also be passed through a specified XSL stylesheets, to produce HTML, WML, PDF, or other XML, as required. This ability is provided by the XMLViewHandler servlet.

Any Controller can be invoked via it's URL with the "style=xml" option. "style=xml" in conjunction with "xsl=none" will produce the XML output directly to the browser. Any other "xsl=..." option will use the specified stylesheet for transformation. If no XSL is defined, the XSL to be used will be determined by the Controller/XSL mappings (defined from the "Setup" page) in the current context.

XSL Stylesheets

The use of XSL stylesheets with Controller outputs can be invoked according to a URL parameter, as described above. Furthermore, the configuration file can specify an xsl transformation. That is, instead of specifying a forward for a JSP, a forward can be specified for a .xsl file. For example, considered the action mapping below, from an entry in a fictitious configuration file "myProject-config.xml":

```
<action path="/MyController" type="com.jcorporate.myproject.controller.MyController"
name="default" scope="request" validate="false">
```
<!-- This method is rendered by a JSP -->
<forward name="someMethod" path="/myProject/jsp/register/status.jsp"/>

<!-- you can use XSLT transforms here too. any path which ends with .xsl or .xslt will be handled appropriately. -->
<forward name="someOtherMethod" path="/myProject/xsl/blah.xsl"/>
</action>

In this configuration file, the .xsl forward will be interpreted by Expresso to mean that the forward will be remapped to "xml", which by default is /servlet/XMLViewHandler, and the xsl information is automatically added to the request with a parameter. Note that the mime type for XSL transformations is assumed to be HTML. If you wish to transform from XML to XML, add a parameter to the request as follows:

```java
class MyController ...
    protected void runSomeOtherMethodState(ControllerRequest request, ControllerResponse response)
        throws ControllerException {
        ...  
        // set mimetype of result, since xsl is usually translated to html, but in this case we will be outputting xml
        request.setParameter("mime", "text/xml");
    }
```

Expresso incorporates and includes the Apache Xalan XSL processor, and this processor can be accessed directly to perform custom XSL transformations as required. See the source code for the XM-LViewHandler object (in the com.jcorporate.expresso.core.servlet.viewhandler package) for an example of how to access Xalan programmatically.

## Conclusion

### Contributors

The following persons have contributed their time to this chapter:

- Larry Hamel (JGroup Expert)
- Mike Rimov

### Note

Was this EDG documentation helpful? Do you wish to express your appreciation for the time expended over years developing the EDG doc? We now accept and appreciate monetary donations. Your support will keep the EDG doc alive and current. Please click the Donate button and enter ANY amount you think the EDG doc is worth. In appreciation of a $35+ donation, we'll give you a subscription service by emailing you notifications of doc updates; and donations $75+ will also receive an Expresso T-shirt. All online donation forms are SSL secured and payment can be made via Credit Card or your Paypal account. Thank you in advance.
Chapter 19. Expresso Component Application Deployment

Note

If you find this EDG documentation helpful please consider DONATING! to keep the doc alive and current.

Version: Expresso 5.5

Maintainer: Marc Schipperheyn

This document describes how to install a "component application" into your Expresso environment and get it running on your system.

Introduction

This document describes how to install a "component application" into your Expresso environment and get it running on your system. This "component application" is an application written to use Expresso, such as eForum, ePoll, eSearch, among many others. This procedure assumes you already have Expresso operational and have verified that it is running properly. If you're not sure that this is so, please check the Expresso installation instructions first.

Before You Start

You should have a single .jar file that contains your application, such as eforumXXXX.jar, where XXXX is the version number of the application to be installed. You need to determine the directory into which Expresso was installed: this is the web application directory where the Expresso .war file was un-jar-ed during Expresso's installation. You should be able to see the files error.jsp, frame.jsp, toc.jsp, etc in this directory, and it should contain sub-directories such as doc, jsp, WEB-INF, and so forth.

Installation Procedure

Install the Files

1. Copy the .jar file referred to above to the web application directory where Expresso was installed.

2. Un-jar the jar file by issuing the following command: jar xvf eforum1-20.jar Replacing of course the eforum1-20.jar name with the name of the appropriate jar file. As the .jar expands, you should see a number of files extracted. These files go into the following directories within the Expresso installation:

   - **components**: The installation should create a new sub-directory within the "components" sub-directory of the expresso installation. For example, if you have Expresso installed at /usr/expresso, then the eForum application discussed above should create /usr/expresso/components/eforum.

   - **WEB-INF/classes**: The .class files (and, optionally, the .java files if you are installing with source code) for the application will be installed in the appropriate sub-directories within the WEB-INF/classes directory in the Expresso web application. For example, eForum will install
it's files in WEB-INF/classes/com/jcorporate/eforum, creating several sub-directories.

- **config/XXXLogging.xml**: The new application will also probably install a file in the "config" sub-directory to define it's logging setup for Expresso to use with Log4j (the logging sub-system integrated with Expresso). If "config" is not the configuration directory your installation is using (which is should *not* be for a production system), you should move or copy this file to the configuration directory (this is the same directory in which is found expressoLogging.xml, default.properties, and other setup files).

3. If you have a separate database for your application, you need to set it up in the expresso-config.xml file. Refer to the database specific Installation Notes for more information.

# Register the Application's Schema

Usually, you will want to register your Schema in your own database, as opposed to the default Expresso database (Hypersonic). In order to do this you must (in the following order):

1. Set up your database in the expresso-config.xml file.
2. Create the Expresso tables in your database:
   - Open the Expresso Administrator.
   - Go to Setup and click on Create/Verify Database Structure & Perform Initial Setup.
   - Select your newly created context/database in the dropdown box.
   - Click Run.
3. Make sure that the value for hasSetupTables in the expresso-config.xml file for your database is set to 'y'.
   - If it was set to 'n', you'll have to restart Expresso before continuing.
4. Click Login and select your your application from the dropdown box.

Next you must "register" the application's Schema object with Expresso. This allows Expresso to manage and set up the application for you, as the Schema object lists all of the other components contained in the application.

1. Start your web or application server if required.
2. Start your web browser, and go to the appropriate URL for your Expresso installation.
3. Log in as the "Admin" user. If you have not set a password, the password can be left blank.
4. Select the "Applications" page, and at the bottom of this page select the "add" icon for Application Schema Objects
5. Enter the following information for the newly installed application:

- **Schema Class**: Enter the class name (not the file pathname) of the Schema class for the new application. For example, for eForum the name would be com.jcorporate.eforum.ForumSchema. Make sure this name is on the first line of the field, and do not follow it with a return or spaces.

- **Schema Description**: Enter a short descriptive title for this application. This title is used in the list of applications provided by Expresso at the left of all Expresso pages.

- **Component Code**: This code gives the name of the directory within the /components subdirectory of Expresso that contains the HTML and JSP pages for the new application component. For example, "eforum" is the component code for eForum. Note that the component code is usually all lowercase, even though the name of the application may not be. Expresso uses this code to create a link to the "frame.jsp" file of the newly installed application for the menu at the left side of Expresso pages.

**Run DBCreate to create the Application's Database Tables**

Once the application is registered correctly, you should run the DBCreate servlet (or the DBTool standalone application) to initialize the new application. This will do several things:

1. Create the database tables required by the application's DB objects.
2. Create default entries for those tables, if any are specified by the application.
3. Create default security (allowing the group "Admin" permission) to all of the applications Jobs, Controller and DB Objects.
4. Perform any custom setup defined by the application.

**Conclusion**

Once these steps are completed successfully, you are ready to use the new application. If you refresh the left-hand frame from Expresso, you should see a link to the new application in the list of installed applications.

**Contributors**

The following persons have contributed their time to this chapter:

- Marc Schipperheyn
- Sandra Cann

If you find any trouble spots with this document, or something you'd like to see clarified please contact the maintainer of this document.
Note

Was this EDG documentation helpful? Do you wish to express your appreciation for the time expended over years developing the EDG doc? We now accept and appreciate monetary donations. Your support will keep the EDG doc alive and current. Please click the Donate button and enter ANY amount you think the EDG doc is worth. In appreciation of a $35+ donation, we'll give you a subscription service by emailing you notifications of doc updates; and donations $75+ will also receive an Expresso T-shirt. All online donation forms are SSL secured and payment can be made via Credit Card or your Paypal account. Thank you in advance.

[http://www.jcorporate.com/edgdoc.html]

Copyright © 1999-2004 Jcorporate Ltd. All rights reserved.
Chapter 20. Unit Testing

Note

If you find this EDG documentation helpful please consider DONATING! to keep the doc alive and current.

Version: Expresso 5.5

Maintainer: David Lloyd
[mailto:dlloyd@jgroup.net?Subject=EDG]

Expresso builds upon the JUnit testing framework to provide an automated unit testing environment.

Introduction

Expresso contains the JUnit framework [http://junit.sourceforge.net/#Documentation] - a unit testing framework that allows the individual components of Expresso to be tested and verified as functioning correctly. Expresso applications can (and should) also include a number of unit tests. Unit tests differ from Performance Tests (covered in the HealthCheck chapter) in that they deal with individual components, making it easier to locate and correct a problem before components are assembled into a finished application.

It was necessary to extend JUnit because Expresso applications expect certain system configurations to be available at any given time. These include database configuration, as well, as web-application location (which would normally be gleaned off of the servlet container). Further extensions were needed to be able to run test cases within a servlet container.

To accomplish this, we branched into two areas:

- **Client Side Testing**: involves any areas that can be run without a servlet container. There are a surprising large number of subsystems that just need the underlying Expresso database framework, and not the servlet containers. This also contains a special case of Schema Testing that is covered below too.

- **Server Side Testing**: involves testing of Servlets and Controllers.

You can use your IDE, like Eclipse or Idea or JBuilder, to run unit tests. Set up VM arguments as:

```
-Djunit.argv.configDir=./WEB-INF/config
-Djunit.argv.webAppDir=./
-Djunit.argv.testContext=default
-Djunit.argv.logDir=/home/tomcat/logs
-Djunit.argv.loggingConfiguration=./WEB-INF/config/expressoLogging.xml
```

Then choose a test suite to run. The do-everything test is: com.jcorporate.expresso.core.ExpressoTestSuite, which will run all JUnit tests.

Also, you can use the standard JUnit user interfaces for running unit tests such as the TextUI or SwingUI classes. For more information on JUnit, check out JUnit's Web Site [http://www.junit.org/]

Client-side Tests
Expresso web applications can be tested via free tools like httpunit [http://httpunit.org], which offer a browser-like client to test individual pages. Built on top of httpunit, Morebot [http://morebot.org] has special support for Expresso, and can provide a cursory test the majority of your application by "spidering" (following links) it after logging in. Morebot can dispatch several of these tests simultaneously to serve as a load test for your server.

Client Side Testing

To provide a framework that we can perform client side testing with all standard expresso services such as logging, database connection management, and many others we provide the following class:

```java
public class SampleTestSuite extends EspressoTestCase {
    public SampleTestSuite(String name) throws Exception {
        super(name);
    }

    public static void main(String[] args) throws java.lang.Exception {
        //Set the system properties we need
        junit.textui.TestRunner.run (suite());
    }

    public static junit.framework.Test suite() {
        return new TestSuite(SampleTestSuite.class);
    }

    public void test1() {
        //Do your testing code here
    }

    public void test2() {
        //Do more testing here
    }

    public void test3() {
        //You can do more testing here
    }

    /* ....Do as many test cases as you want, all with the public void signature */
}
```

How to run this test case

To run the test case, you need to set the following Java Virtual Machine parameters to:

```
junit.argv.configDir= junit.argv.webAppDir=
```

Usually in the command line you'll do this by having the command line:

```
-Djunit.argv.configDir= -Djunit.argv.webAppDir= -Djunit.argv.logDir=
```
The JUnit framework will automatically use Java Introspection to run all the public(void) test cases. See the JUnit documentation on how to use the framework to signal if a test passed or failed. Also, see the class: com.jcorporate.expresso.core.misc.CookieTests to see a fully functional sample of integration with JUnit Framework.

All expresso test case classes will play well with the JUnit test runners. So if you wish to run a text-based only test case, you can run junit.textui.TextRunner. If you wish to run within a nice graphical environment, you can run your test suite with:

To run all test cases within a graphical environment, run the class:

junit.swingui.TestRunner use the UI to pick the class you wish to test and let her run.

The -c will determine which test suite to run. You can use your own test cases, or use the class sample above to run all Expresso test cases at once.

Writing your test case guidelines
There are a couple of necessary guidelines to follow when writing your own test cases:

- Make sure that your tests are fully automated. This will indeed pretty much take the most work in creating your test cases, but without that automation, it is ***IMPOSSIBLE*** to spot any minor errors that creep into your code while refactoring. It cannot be stressed how much your time will be saved in the long run if you can think of ways to automatically check for proper behavior.

- Whenever you need the expresso environment properly running, make sure you derive from ExpressoTestCase.

- Always include a main() function. This allows for extremely quick testing while you're building your classes and tests.

### Schema Testing

When we're first creating our application, we want to make sure it runs correctly, right? But as we're developing our schema, and creating out databases, we have to each time go through the monotonous task of dropping a database, recreating it, starting up Expresso, running DBCreate, etc. This quickly becomes quite a chore. So we've created a special test case for testing Schema Creation. Here's all the code you need. The following example is borrowed directly from eForum and is complete:

```java
import junit.framework.*;
import com.jcorporate.expresso.services.test.*;
import com.jcorporate.expresso.core.utility.DBToolTests;

public class SchemaTests extends DBToolTests {
    public SchemaTests(String testName) throws Exception {
        super(testName);
    }

    public static void main(String[] args) throws Exception {
        // Set the system properties we need
        junit.textui.TestRunner.run (suite());
    }

    public static junit.framework.Test suite() throws Exception {
        return DBToolTests.suite();
    }

    protected void setUp() throws Exception {
        // System must be initialized prior to instanatiating the schema instance
        TestSystemInitializer.setUp();
        Class c = Class.forName("com.jcorporate.eforum.ForumSchema");
        schemaList.add(c.newInstance());
        super.setUp();
    }
}
```

That's it! To make this code work for your schema all you do is change the ONE line in setup() that loads ForumSchema to use your own schema class. When this class is run, it will make sure that all tables are removed from the Test database, and then attempt to create all schemas (Including Expresso's) in the test database. It will also run separately populateDefaultValues() and setupDefaultSecurity() so you can make sure that everything is cooperating.

This kind of testing is an extremely helpful and powerful tool for testing your schemas as you're building them.
Server Side Testing

The bulk of your web application is going to be logic within an Expresso Controller object, or possibly a standard Servlet. Unfortunately, standard JUnit client-side testing procedures definitely fall flat in being able to cope in this area. Enter Apache’s Cactus project.

Cactus was designed to provide the unification of JUnit’s client side testing API and in-container server-side testing. What it does is initiate a test case on the client side so you can stuff all the HTTP request parameters with everything you need such as state to request, login cookies, etc. The test case then serializes itself to the server, where a specially designed servlet loads your test class and executes the code on the server side. The result is then sent back to the client side so you can examine cookies sent back to client or other results.

Here is an animation created by the Apache Cactus documentation project that highlights what was said above.

If you are writing a servlet, the Cactus documentation will be sufficient to get you started on a servlet. Even if you’re writing a controller test case, it is strongly recommended that you browse through the Cactus website to get an idea of how a Cactus test case is written since the controller test cases are extensions to a standard Cactus test.

If you are writing a controller test harness, there are quite a few things to consider such as having a running underlying database, negotiating security, and parsing the controller response. To assist in this, we’ve provided the classes:

```
com.jcorporate.expresso.services.test.ControllerTestCase
com.jcorporate.expresso.services.test.ControllerTestSuite
```

A Quick Sample

Below is a sample showing a simple usage of a controller test case (suite). All comments with a number are footnoted/explained in the area below the code sample.

```
import com.jcorporate.expresso.services.test.*;
import com.jcorporate.expresso.core.controller.*;
import org.apache.commons.cactus.*;
import org.apache.commons.cactus.util.*;
```
import junit.framework.*;
import org.w3c.dom.*;
import java.net.HttpURLConnection;
import java.io.IOException;
import javax.servlet.*;
import java.util.*;

public class SampleControllerTest extends ControllerTestCase {
    public SampleControllerTest(String name) {
        super(name, "com.jcorporate.expresso.services.DBSecurityMatrix"); //1
    }

    public static void main(String[] args) throws Exception {
        junit.textui.TestRunner.run(suite()); //2
    }

    public static TestSuite suite() throws Exception {
        ControllerTestSuite cts = new ControllerTestSuite(); //3
        cts.addReadOnlySchemaDependency("com.jcorporate.expresso.core.ExpressoSchema"); //4
        cts.addTestSuite(SampleControllerTest.class);
        return CTS;
    }

    /* Executed on the client side of the web request */
    public void beginPromptState(WebRequest theRequest) throws Exception {
        super.logIn(theRequest); //5
        super.setupParameters("prompt", theRequest); //6
    }

    /* Executed on the server side of the web request */
    public void testPromptState() throws Exception {
        ControllerResponse response = super.controllerProcess(); //7
        assertTrue("Got a null response", response != null);
        assertTrue("Title returned from the controller state.",
                response.getTitle().length() > 0); //8
    }

    /*....Do as many test cases as you want, all with the public void signature */
}

Code explanations:

1. The second parameter to the ControllerTestCase's constructor is the name of the class you wish to test. In this case it's Expresso's own DBSecurityMatrix controller.

2. By writing a main method that calls JUnit's Text Test Runner, you have the ability to test out your classes and test cases by just running this class.

3. The controller test suite is responsible for having a proper underlying test database running. If one doesn't exist, it will create it for you automatically, (including schemas), and if you need it to be erased when you're done, it will do that too.

4. This line(s) tells the test suite what schemas you must have up and running for your test case to
work properly. To add more schemas, simply repeat the call for other schemas. Use addReadOnlySchemaDependency() if you DO NOT do any modifications to the underlying databases by running your controller. This will tell the system to not delete your created test database at the end of a run since it's still in pristine state. Call addSchemaDependency() if your controller modifies the underlying database in any way.

5. super.logIn() set’s the appropriate cookies for a user Admin with a blank password. This is what is created by default in a test database.

6. The first parameter of setupParameters() is the name of the state that you will be testing with this test run. In this case it’s "prompt". The function also sets the appropriate parameters for the Controller to return an XML formatted output.

7. The first confusing thing to know is that testXXXXXX() RUNS ON THE SERVER. Everything up to this point has been running on the client side. To process the controller, however, your job is simple, just call the ControllerTest case's controllerProcess() function and the testing framework will call your controller state automatically. This function will return a ControllerResponse object back to you, in effect, the data that the controller generated.

8. Once you get the ControllerResponse object you can walk through it's Inputs, Outputs, Blocks and Transitions to make sure they are as you would expect them to be. Use JUnit's assertTrue() and fail() functions to test if things were properly sent back.

You can take a look at the class com.jcorporate.expresso.services.controller.test.DBSecurityMatrixTests for a complete working controller test case that tests a couple of states including getting the results of a form POST.

Running The Test Case

To run the Cactus tests you need the following software pieces.

- **A database capable of dealing with more than one virtual machine connection.** Sadly, this eliminates the possibility of using the Hypersonic Database that comes with Expresso as-is. There is a driver out there that uses RMI that allows more than one connection, but it is in a separate download available on the Internet.

- **A Servlet API 2.3 compliant servlet container:** This includes containers such as Orion and the Tomcat 4 bundle that is included with the Expresso full download. If you only have a Servlet API 2.2 compliant container such as Tomcat 3, you need to separately download and install the Cactus jar for servlet api 2.2. Such downloads can be found at: https://jakarta.apache.org/commons/cactus/downloads.html

Configuring Cactus

There is one step that has to be done for Cactus to run properly on your system.

- Locate the file cactus.properties located in the WEB-INF/classes directory of your installation. Change the URL to the Cactus redirector servlet depending on your system. For most cases the URL will be very similar to what is listed already in the file. This allows Cactus to know where to send it's test case requests.
Running the test cases

- Before you can run your test cases. You must start your servlet engine. Otherwise the test case will fail when it goes to perform the server side code.

- Run the class you are testing with the following Virtual Machine parameters:

  -Djunit.argv.configDir=
  -Djunit.argv.webAppDir=
  -Djunit.argv.logDir=

  Be sure these parameters have the proper values set.

You can run any JUnit test runner for a controller test case including SwingTestRunner for graphical test suites.

Conclusion

It should be noted that to follow Extreme Programming's (XP) unit testing strategies, you will want to chain your test cases together. Check the example of com.jcorporate.expresso.core.ExpressoTestSuite for a clear example of this usage.

Contributors

The following persons have contributed their time to this chapter:

- Larry Hamel (JGroup Expert)
- Mike Nash
- Mike Rimov
- Mike Traum (JGroup Expert)
- Sandra Cann

Note

Was this EDG documentation helpful? Do you wish to express your appreciation for the time expended over years developing the EDG doc? We now accept and appreciate monetary donations. Your support will keep the EDG doc alive and current. Please click the Donate button and enter ANY amount you think the EDG doc is worth. In appreciation of a $35+ donation, we'll give you a subscription service by emailing you notifications of doc updates; and donations $75+ will also receive an Expresso T-shirt. All online donation forms are SSL secured and payment can be made via Credit Card or your Paypal account. Thank you in advance.

[http://www.jcorporate.com/edgdoc.html]

Copyright © 2001-2004 Jcorporate Ltd. All rights reserved.
Chapter 21. Healthcheck Monitoring and Application Performance

Note

If you find this EDG documentation helpful please consider DONATING! to keep the doc alive and current.

Version: Expresso 5.5
Maintainer: David Lloyd

Now that you have your application developed and deployed, what tools are available for you to monitor and maintain your application? You want to ensure that the individual components of Expresso and your custom components are operating correctly, to ensure that your applications stays operational and available to users, and to ensure that its performance is acceptable and does not degrade over time. Tools exist within Expresso to verify its own functioning and the function of any other web application. Expresso Health Check utility provides periodic system and performance tests as means to monitor the various components of a web and/or intranet and to ensure that they remain up and operational at all times. It is designed to be run by a system scheduler.

Introduction

Expresso Health Check utility provides a means to monitor the various components of a web and/or intranet and to ensure that they remain up and operational at all times. It is designed to be run by a system scheduler (cron, ROBOT, NT Scheduler) at short intervals (e.g. once every 30 minutes).

This chapter also covers the area of application performance.

Healthcheck Monitors Web Components

Designed to be run on a timer on the server. It checks that:

1. Web services are running
2. The database is up
3. The ORB is operational
4. Expresso Framework is running correctly
5. The ServletRunner is operational
6. Pilot/PIP is operation (if applicable at site)
7. Email services are operational.
8. Performance is within acceptable range for specified urls

Tests may be defined which execute a specific URL and measure how long it takes to respond, as well
as checking for a specific response. This helps ensure the system is operating correctly.

**Healthcheck Error Reporting**

All except the last item on the above list, give a warning message in email, the last one just gets a big warning on the console & a message in the log. Health Check is an essential utility for large sites when each of these services may be on a different server.

If any of these checks fails, the HealthCheck utility stops what it is doing and attempts to notify the system administrator by one of several means:

- It logs a message into the server log (red)
- It attempts to send an email to the SYSERROR event list of users
- It displays a prominent message on the console (in case all other notifications fail)

Health Check can also be monitored real-time by a special System Status/Health Check Servlet web page, which automatically updates itself at a user-specified interval (e.g. every 30 seconds). It performs all of the selected health check functions, and shows a check mark for every successful check. It also shows a distinctive “warning” icon for any function that fails, along with a detailed description of the problem & an audio warning (so the system administrator is alerted even if not actually watching the page). This page can be left up on a console for a continuous monitoring of the overall system health at all times.

**Healthcheck Tests Performed**

HealthCheck performs the following tests:

- Verifies that the control database is operational by connecting to it & reading setup values
- Verifies that the Object Request Broker is operational by connecting to it
- Verifies that the Application Framework is operational by connecting to it
- Verifies that the SMTP email server is operational by sending a test e-mail to a fictional user
- Verifies that the servlet runtime system is operational by running a special status servlet Using our extensibility philosophy, Health Check can be configured to check other system functions, i.e. could verify that PIP/Pilot is operational by requesting status.

To Use HealthCheck, select the "Operation" menu choice in Expresso's Admin menu. Scroll down to the heading, "Periodic System and Performance Tests." There you will see a list of functions like the below:

<table>
<thead>
<tr>
<th>HealthCheck Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run Performance Tests</td>
<td></td>
</tr>
<tr>
<td>Run HealthCheck</td>
<td></td>
</tr>
<tr>
<td>Performance Test Definitions</td>
<td></td>
</tr>
<tr>
<td>Performance Test Stats</td>
<td></td>
</tr>
</tbody>
</table>
Optimizing Performance

It is important to concentrate your efforts on the areas where the largest gain can be experienced - in order to do this, you must understand where the "hot spots" are in your application. Expresso can help you do this, and then help you perform more extensive analysis of those "hot spots" to make sure they are running as quickly as possible.

Performance Hogs

There are a number of things that can slow down an Expresso application. By checking this list carefully before beginning any more intensive performance tuning, you can verify that you are not "driving with the parking brake on" and can often solve performance problems immediately.

Some of the items to watch for are:

- **Logging:**

  Log4j, which is the integrated logging mechanism built into Expresso, is a very capable package, and it's performance impact is very small compared to many logging methods. Still, the fastest logging is no logging at all, and careful adjustment of the expressoLogging.xml file, and the corresponding file for your custom application can result in a large performance benefit.

  The first logging issue to look for is of course to select the appropriate logging level for your running application. If you are still testing and debugging, then log priorities of "debug" for at least some objects might be appropriate, but in a production environment you should need no more than "info", and preferably only "warn". One of the easiest ways to see if you have something set to too detailed a level is to read the log file, looking for "debug" or "info" messages that you do not really need to see in production - then find out why these messages are still enabled and edit the xml configuration file accordingly.

  Another important logging performance issue is the format of the logging message: the "Conversion-Pattern" parameter that can be supplied when setting up a logging channel can have a significant impact on performance. During testing and debugging, it is often very useful to be able to see the exact class, method, and line number that a log message originated from. Unfortunately, during production the processing required to produce this information is very expensive, and can be the cause of a performance issue. Review the log4j documentation included with Expresso for details on how to specify less "expensive" patterns for logging entries for production systems.

- **dbTest on:**

  In the property file, the dbTest property supplies an SQL statement that is used to verify if a database connection is "alive" (e.g. connected to the database and operational) before passing the connection back from the connection pool when a new connection is requested. This is helpful in situations where the connection could time out or otherwise become unavailable while the connection pool is still maintaining the connection - for example, an external data warehouse connection that is only used infrequently. In normal production, however, this option can be expensive, particularly if the query being specified is not very fast on that particular database. For example, if you specified a dbTest of "select * from USERLOGIN" you would certainly notice a severe performance decrease, as the entire USERLOGIN table would be retrieved every time a new connection was requested (which is a great many times).

  Ideally, leave out the dbTest property where it's not needed, or at least be sure that it is a fast opera-
tion for the specific database.

- **dbCache off:**

  In the properties file, the `dbCache` property is by default "y", e.g. database object caching is enabled. It is possible to specify `dbCache=n`, which disables caching for database objects entirely, and as a result does not take advantage of caching as a performance boost for database accesses. Be sure you leave out "dbCache=n" for best production performance.

- **Bad Setup Values:**

  For best performance, it is important to review your Setup values for correctness. For example, if you should have specified an email server that is incorrect our outside your local network, your application could be spending a lot of fruitless CPU cycles trying to attach to this server to send email notifications.

  The best way to catch such problems is to review the log, looking for any unusual errors or warnings and tracking them back to their source. Often the culprit will be a bad Setup value.

### Cache Tuning

Unless you have specifically turned off the option to do so, (see the properties file documentation) Expresso as it runs will collect information about the effectiveness of the DB Object caching system. This information includes how many read operations (retrieve(), searchAndRetrieve(), find() and so forth) have been made for a particular DB object (in a given db/context), and how many of these reads were able to use the cache to find the record they were looking for. The idea of course, is that generally speaking the more often the record can be read from cache, instead of being retrieved from the database, the faster your application will run.

The "Status" servlet is used to retrieve this information, which also computes the percentage of "hits" to the database object that were supplied by the cache - what you're looking for in this listing is a large number of total hits with a low percentage (or zero percentage) of hits to the cache. This indicates a `DBObject` that is heavily used by read operations in your application, but which is not caching very much. The listing from status also shows you the size of the currently specified cache for this database object - if it is zero, then you may not have an entry for the object in the "Database Object Page Limits" table (accessed from the "Setup" page in Expresso).

There are a number of other factors you should consider when tuning caching for database objects:

- **Available Memory:**

  Caching can use substantial amounts of memory, so you should check the amount of available memory to your application before increasing the cache sizes for objects too much. You can see a "snapshot" of current memory usage from the status servlet, but you may wish to read the log to see what memory usage has been over time. You can adjust the logging detail level for the "core.cache.CacheManager object" to see periodic memory reports. If you see a number of instances where the cache manager is having to clear caches in order to get more memory (e.g. to bring it over the specified minimum - again see the property file documentation for details), then you probably cannot increase caching much without actually slowing down the application.

  Be sure that you are launching your java virtual machine (or your applications server) with the appropriate options to make enough memory available to your application - just because you physically have enough memory in your system doesn't mean that your JVM has use of this memory. See the -X options for your Java runtime for details on how to increase this limit.
• **Cached Items:**

Some database objects may show high activity, and low cache hits, but still not be a problem. This is particularly true of items that are *always* cached by Expresso, such as Setup values, security entries for db objects and controllers, and a few others.

• **Write Patterns:**

The effectiveness of caching is also influenced by the patterns in which your application writes to the database - if a particular db object is changed, it's cache entry is removed (so that stale data is not read) and it must be read from the database again the next time it is accessed. If you have an object that has a lot of cache set up for it, but is still not caching very successfully, it may be that this object is being written to very frequently.

This is more of an application design issue than something that can be directly addressed by cache tuning.

By careful manipulation of the records in the "Database Object Page Limit" table, you can make best use of your available memory via db object caching. This is often the single most important optimization step you can take.

**Database Optimization**

If all of the above optimization and performance tuning techniques indicate that there is a problem with database access slowing down the system (despite tuning the db cache), then it is sometimes required to analyze the database queries themselves in order to determine which queries are causing the problem.

The DBConnection object, which is responsible for executing the actual SQL queries built by DB objects, can help perform this analysis: turn on the logging priority for this object to "debug" using the espressoLogging.xml file, and monitor the results. Each SQL query will have its elapsed time recorded, and you can use this information to pick out the worst offenders and either modify your application to perform better queries, or use database indices to make them faster.

**Indexing**

See the section on DObjects for information about setting up Indexes for faster access on non-primary-key fields.

**Conclusion**

**Contributors**

The following persons have contributed their time to this chapter:

• Larry Hamel (JGroup Expert)
• Mike Nash
• Mike Rimov
• Mike Traum (JGroup Expert)
Note

Was this EDG documentation helpful? Do you wish to express your appreciation for the time expended over years developing the EDG doc? We now accept and appreciate monetary donations. Your support will keep the EDG doc alive and current. Please click the Donate button and enter ANY amount you think the EDG doc is worth. In appreciation of a $35+ donation, we'll give you a subscription service by emailing you notifications of doc updates; and donations $75+ will also receive an Expresso T-shirt. All online donation forms are SSL secured and payment can be made via Credit Card or your Paypal account. Thank you in advance.

[http://www.jcorporate.com/edgdoc.html]

Copyright © 2001-2004 Jcorporate Ltd. All rights reserved.
Chapter 22. Jobs Expresso

Note

If you find this EDG documentation helpful please consider DONATING! to keep the doc alive and current.

Version: Expresso 5.5
Maintainer: David Lloyd
[mailto:dlloyd@jgroup.net?Subject=EDG]

The Job Handler component is a server-side thread (can also be run in a separate VM if desired) that runs tasks according to entries in a job queue. These jobs may be run sequentially (single threaded) or in parallel (multi threaded), depending on the nature of the job.

Introduction

As a server side process, a job can be queued by means of a user request (by a servlet) or by means of a simple batch file, which can be triggered by an operating system scheduler to provide scheduled jobs - for example a certain job to run a report might be triggered every night at 10PM. Jobs typically send notice of their results via the event mechanism, and system error events are automatically triggered if a problem occurs.

Jobs are submitted to the queue either by:

- user interaction (e.g. manually requesting a job)
- timed processes (e.g. cron or NT scheduler)
- or by other jobs (such as events triggering a data load or report run).

Job control is similar to the same mechanism used so successfully on mainframes for many years. It can be used for any function which takes more time than is practical for an interactive session, such as a complex calculation, a data load from a legacy system, or the running of a group of long reports or analyses.

Job Queue

The basic mechanism involved is a "job queue". A job is queued by creating a new entry in the job queue and marking it as "available". A job can also have parameters associated with it to control it's functions, much like arguments to a method.

Job Handler

There are one or more "job handler" objects running on the server side, waiting for jobs to enter the queue - when they do, a job handler picks up the job & begins to process it - which job handler does this can be controller so that different jobs run on different systems, for performance reasons if so desired. Each job handler may process either a single job or can run a set number of jobs in parallel, and each job can be designated as "single-threaded" (e.g. should not be run in parallel with other jobs) or "multi-threaded".
The job-handler objects can also be implemented as either stateful or stateless Session EJB’s, allowing business logic to be developed with all of the advantages of Enterprise JavaBeans.

The Job Handler is a separate thread (which can be run as a thread in your application/servlet server JVM, or as a separate Java application (not a servlet)) that runs on a server system (usually the same system as your servlet or application server, but not necessarily). The Job Handler constantly monitors the Job Queue, and when Jobs are detected it starts the requested job, monitor’s it’s processing, and sends email notifications when it is complete. This allows your application to request tasks that would take longer than a few moments as a background job.

**Job Order**

Jobs are handled in priority order, and a job can be suspended and it’s priority changed even once it has begun running. Certain jobs can be set to process at only specified times, deferring process-intensive tasks until off-peak hours if so desired. Jobs frequently trigger event notifications when they are complete or when problems occur.

**Creating a job**

Expresso Framework has the ability process tasks asynchronously. You might want to a task that takes a considerable amount of time to execute, for example, to search an entire database, to calculate the payroll of all employees in your company, or to index web pages for a search engine. The list of potential long running processes is endless. It does make sense to deploy these long process over the web directly, but rather active them indirectly using a job handler. Expresso Framework has a built-in job handler.

To create simple job you must extend the standard class `com.jcorporate.expresso.core.job.Job` and implement the method `run`.

```java
package org.fooey.testapp.job;
import com.jcorporate.expresso.core.job.Job;
import com.jcorporate.expresso.services.dbobj.JobQueue;
import com.jcorporate.expresso.services.dbobj.JobQueueParam;
public class PrimeNumberSearch extends Job {
    public PrimeNumberSearch() { super(); } /**
```

189
Return a string for the security administration screens. */ public String
getTitle() { return new String("Prime Number Search"); } /** The run methods should process the long running task */ public void run() { try {
doCalculation(); finish("Successfully calculation in
database:"+getDBName()+"'); } catch (Exception e) {
finish("Abnormal termination of calculation", e); } } } The run() should be very familiar to programmers who have experience of multi-thread Java applications. In fact the Job is a direct sub-class of java.lang.Thread! If you want to send feedback to a user, then you call one of the following finish methods.

* finish( String msg ) - this method accepts a message parameter.
* finish( String msg, Exception e ) - this method accepts a message parameter, and a second parameter for reporting an exception.

CAVEAT EMPTOR: Make sure that you have already set up a valid email address for sending error messages. This is because asynchronous jobs are launch as separate threads, and all communication back to the user is either through an email or through a log file. If the Expresso Framework cannot send email about the success or failure of a job then the job handler will fail over continuously in version 5.0 at least. You can set up the email address by going to Administration pages and defining a valid email address for the setup code MAILFrom. For example set up the email address to "yourusername@localhost".

**Submitting A Job**

To submit a job you need to be familiar with JobQueue objects. Job queue object are database objects, you create them with system privileges like so.

```java
JobQueue jq = new
JobQueue(SecuredDBObject.SYSTEM_ACCOUNT);
```

Once you create a job queue object you must associate it with an user id, the job class, and a status.

```java
jq.setField(JobQueue.FLD_UID, requestContext.getUid());
jq.setField(JobQueue.FLD_JOBCODE,
org.fooey.testapp.job.PrimeNumberSearch.class.getName());
jq.setField(JobQueue.FLD_STATUS_CODE, JobQueue.JOB_STATUS_NEW);
```

As with an DBOObject you need to insert into the data store, so likewise with a job queue.

```java
jq.add();
```

When the job queue object is added to the database, because the status has been set to JobQueue.JOB_STATUS_NEW, then the internal Expresso job handler will not active it immediately. In order to active the job queue then you must set the status to JobQueue.JOB_STATUS_AVailable, then the handler will pick it up on its next scheduled loop and execute it.

```java
jq.setField(JobQueue.FLD_STATUS_CODE,
JobQueue.JOB_STATUS_AVAILABLE); jq.update();
```
Why have these two difference status levels? The answer is that job queue may have associated job queue parameters. When you create job queue parameter you need to already have a job queue in existence. Therefore there has be intermediate state between a new creation and activation.

**Job Parameters**

Many jobs need parameters of some sort to execute. For example, the job described above, might want to have prime numbers calculated up to x many digits. You can use the parameters defined in the job to customize your job for specific tasks, and to provide a means to reuse the job code with different parameters.

**Defining the Job Parameters**

You can define a job parameters in the constructor of the job if you want to be able to set values via the Job GUI in the admin pages, or retrieve a list of all possible parameter names. The first argument is the real name of the parameter. The second is the 'friendly' name that is displayed by JobQueue UI.

```java
public class PrimeNumberSearch extends Job {
    public static final DIGIT_PARAM = "digits";

    public PrimeNumberSearch() { // optional: this allows job and UI to know about
        param, for setting value via UI, // or returning via
        "getParameterNamesAndDescriptions()"
        addParameter(DIGIT_PARAM,
        "Number of digits in prime to find"); } ...
    }
```

**Adding parameters to a job**

In the method below, we queue a job just like you would with a normal JobQueueEntry. But we add parameters, name/value pairs, that are associated with the job.

For example:

```java
//Create a new
connection and turn off autocommit for a transaction
DBConnection connection = DBConnectionPool.getInstance("default").getConnection();
connection.setAutoCommit(false); try { // Create the jobqueue entry and add
jq = new JobQueue(SecuredDBObject.SYSTEM_ACCOUNT); // it as before
jq.setField(JobQueue.FLD_UID, requestContext.getUid());
jq.setField(JobQueue.FLD_JOBCODE, org.fooeys.testapp.job.PrimeNumberSearch.class.getName());
jq.setStatus(JobQueue.JOB_STATUS_AVAILABLE); // Set the connection so it will use the same transaction system // as the
other dbobjects
jq.setField(JobQueue.FLD_JOBNUMBER); // Likewise set the connection so it's part of the same transaction
jq.setField(SecuredDBObject.SYSTEM_ACCOUNT);
jobParam.setField(JobQueueParam.FLD_JOB_NUMBER, jq.getField(JobQueue.FLD_JOBNUMBER)); // Arbitrary param number... just must
be unique for each parameter
jobParam.setField(JobQueueParam.FLD_PARAM_NUMBER,1); // Set the
parameter code in this case digits
jobParam.setField(JobQueueParam.FLD_PARAM_CODE, PrimeNumberSearch.DIGIT_PARAM); // Set the digits parameter to 85
jobParam.setField(JobQueueParam.FLD_PARAM_VALUE,85); // Now add
the object and commit the entire transaction
jobParam.addOrUpdate();
connection.commit(); } catch (DBException ex) { // Rollback the transaction
```
if we have an exception connection.rollback(); } finally { // Finally release the entire database connection if (connection != null) {
    connection.release(); } }

Accessing the parameters within the Job itself

You use the function getJobParameter(name) to access the actual parameters that are passed to the job itself.

    public void run() { try {
        String digitParam = (String)getJobParameter(PrimeNumberSearch.DIGIT_PARAM); if (digitParam == null ) throw new IllegalArgumentException("Digits parameter must be > 0"); int numDigits = Integer.parseInt(digitParam ); if (numDigits <= 0) throw new IllegalArgumentException("Digits parameter must be integer > 0"); doCalculation(numDigits); finish("Successfully calculation in database: " + getDBName()+ "]"); } catch (Exception e) { finish("Abnormal termination of calculation", e); } }

Job Handling and Management

Starting the Job Handler

Normally the JobHandler is run as a separate thread in your application server's JVM by adding the 'startJobHandler=y' property to your default.properties (or other property) file.

In order to start the Job Handler in a separate JVM, you must issue a command line on your server system. It is almost always convenient to package up this command line into a batch or shell script file (depending on your operating system), so that it can be run again easily.

An example Job Handler shell script is shown here (the text is all one line in the shell script, but broken into multiple lines here for readability):

```
/usr/java/jdk1.2.2/bin/java -classpath /usr/orion/orion.jar:/usr/java/lib/mail.jar:/usr/java/lib/activation.jar:/usr/java/lib:/usr/java/jdk1.2.2/lib/tools.jar com.jcorporate.expresso.core.utility.JobHandler configDir=/usr/expresso/config webAppDir=/usr/web-apps/expresso
```

This example uses the JDK1.2.2 runtime and the servlet API libraries in the Orion Application Server, assumes that "default.properties" is in the specified configDir directory. Running this script produces output to the standard output, so it is recommended that it be run with the nohup command, or that its output be redirected.

This script can of course be altered for the specific directories and libraries that you are using, and broken into multiple lines if desired.

The Job Handler is ordinarily left running whenever the server is up, so that Jobs receive immediate consideration based on their priorities.

Job Scheduling Features

Submit New Job
// allocate empty slot for new job
JobQueue oneJob = new JobQueue(); // this job should be executed on "Unix" systems
Only. // the valid options are: MSWIN, Unix, any or Specific OS as returned
by System.getProperty("java.os")
oneJob.setJobOSName("Unix"); // execute this job at specified by
cron parameters time. oneJob.setJobCronParams(minute, hour, dayOfMonth,
month, dayOfWeek, year); // set job code to execute
oneJob.setJobCode("com.jcorporate.expresso.utils.TestJob"); // set
the owner of this job
oneJob.setUserName("root") // job is ready for
execution oneJob.setJobStatus(JobQueue.JOB_STATUS_AVAILABLE) // Add
parameters associated with this job (same as before) // submit job
oneJob.add();

Controlling jobs

JobHandlerControl jobControl = new
JobHandlerControl(); String jobID = "10"; // stop job
jobControl.setCmdStopJob(jobID); jobControl.add(); // restart job
jobControl.setCmdRestartJob(jobID); jobControl.add();

Cron Jobs

Theory and Implementation

When the JobHandler searches for jobs in the queue, it only picks up jobs that are set to status =
JOB_STATUS_AVAILABLE and server id = 0. I believe the server id is a reference to the jobhandler
in jobhandlerregistry table. If you have a job you want initiated on every startup, be sure to set server id
to zero (0) (which will happen automatically if the Job is created with the standard constructor).

The jobhandler does the following:

• Jobhandler looks for jobs in the queue that are serverid=0, code=available and runs them or submits
them to crontab, then sets the status to R (running). I think this means that if you change the crontab
params after the job is running it won’t do anything since it’s already been submitted to crontab.

• When the crontab daemon thinks it time to run, the job runs

• The job gets set to C for "Complete"

Each job can be executed in a cron-like manner. The ability to start a repeating job through the "Queue
Any Job" controller does not exist. Instead you have to manually create the entry with some additional
fields filled out. To do this, go to: Applications -> Job Queue Entries. The easiest way is to pick out a
job that you have already executed once that you wish to submit to the cron manager.

In the entries parameters enter the following values:

Table 22.1. Job Queue Entry Parameters

<table>
<thead>
<tr>
<th>Entry Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Job Serial Number</td>
<td>Leave Alone Since it is auto-assigned</td>
</tr>
<tr>
<td>Requested By User</td>
<td>The login name of the user that will be requesting</td>
</tr>
</tbody>
</table>
**Entry Name** | **Value**
---|---
Job Code | The classname of the class derived from com.jcorporate.expresso.core.job.Job that will be run when the job executes.
Status Code | Set to "Available"
Job Priority | Best to leave at the default value
Handling Server | Set to "0" Job servers will only pick up items that are not associated with any server yet.
Required Job OS Name | Leave to any. Although you can set the value to what the JVM gives as the operating system if you have special job needs. For example, if you have a job server on a Linux box, and a job server on the Windows box, and you need the job to use Windows Automation to build an Excel spreadsheet, then you could assign the particular job to require the Windows system identifier. Note, however, that this is an untested feature.
Cron Parameters | Enter six comma-delimited integers. The actual values will be set up as defined later. Leave blank if you do not what this job submitted to the Crontab manager.

Click "Add Record" or "Update Record" to save the job to the database. In approximately 30 seconds, the JobHandler will pick up the job. If there are cron parameters associated with it, it will be sent to the crontab for time management.

**Meaning of the cron parameters**

Eddie Lewis

The setJobCronParams method as was mentioned earlier, should be 6 comma-delimited integer values like so: -1,-1,-1,-1,-1,-1. Each item corresponds to a different time slot. The meanings are:

**Table 22.2. Job Scheduling Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>minute</td>
<td>the 'minute' number when the cron should be executed. Standard useful values are 0-59. Set to zero if you want the cron to execute at the top of the hour.</td>
</tr>
<tr>
<td>hour</td>
<td>the 'hour' number when you want the cron to execute. Values are -1 to 23. -1 if you want the job to execute every hour. 0 if you want the job to execute at midnight.</td>
</tr>
<tr>
<td>dayOfMonth</td>
<td>Day of the month to execute (-1 if you don't want this parameter considered). The attribute is exclusive of day of week. Allowed values are: 1-31. java.util.Calendar constants can be used. Be careful of 31 when dealing with months that have less than 31 days as unexpected results can occur.</td>
</tr>
<tr>
<td>month</td>
<td>Which month should the job be run? Values are</td>
</tr>
</tbody>
</table>

194
Note

As you may have noticed, the cron parameters here are absolute. The Job queueing API currently does not allow for items such as "Execute every 20 minutes". Instead you would have to create 3 job entries, one that execute at the top of the hour, one at 20 minutes, and one at 40 minutes.

Cron jobs which are specified to repeat have the special quality of resetting themselves to run again if they have already run successfully during the lifetime of the current JobHandler. That is, after the job is done, even though its status is set to C ("complete"), the job will be rerun at the appropriate (repeating) date and time. However, if you restart the server, you need to make sure, during initialization, that any repeating cron job has its status set to "available" and server = 0.

Some Cron parameter examples

- Execute at midnight every day: 0,0,-1,-1,-1,-1
- Execute at half past every hour: 30,-1,-1,-1,-1,-1
- Execute at December 31, 11:59 every year: 59,23,31,11,-1,-1
- Execute at 3:00 a.m., the first of every month: 0,3,-1,-1,-1
- Execute at June 1, 2:30 a.m., 2006 AD only: 30,2,1,5,-1,2006
- Execute all the time: -1,-1,-1,-1,-1,-1

Conclusion

Contributors

The following persons have contributed their time to this chapter:

- Larry Hamel (JGroup Expert)
- Peter Pilgrim
- Mike Rimov
- Mike Traum (JGroup Expert)
• Sandra Cann

Note

Was this EDG documentation helpful? Do you wish to express your appreciation for the time expended over years developing the EDG doc? We now accept and appreciate monetary donations. Your support will keep the EDG doc alive and current. Please click the Donate button and enter ANY amount you think the EDG doc is worth. In appreciation of a $35+ donation, we'll give you a subscription service by emailing you notifications of doc updates; and donations $75+ will also receive an Expresso T-shirt. All online donation forms are SSL secured and payment can be made via Credit Card or your Paypal account. Thank you in advance.

[http://www.jcorporate.com/edgdoc.html]

Copyright © 2001-2004 Jcorporate Ltd. All rights reserved.
Chapter 23. Reporting System

Introduction

Nearly any J2EE application worth its salt will have some sort of reporting requirements. This is not something Expresso has left unthought of! Expresso 5.3 introduces a series of interfaces that provide for a generic reporting framework. It also provides an implementation of this framework that outputs raw html. In this chapter we will study the interfaces more closely, and show how Expresso can assist you in your reporting needs.

Note

While any developer can create their own interfaces to their preferred reporting engine using the details provided in this chapter. You may want to consider purchasing a license of JCorporate's eContent content management framework. eContent takes the basic Expresso interfaces and provides: Per-report security, multiple export formats such as PDF, XML, Excel, and Html; and integration with popular reporting engines that only require xml definitions rather than manual Java code.

The Reporting Interfaces

The reporting interfaces that come with Expresso are quite simple:
A super quick explanation of the classes is as follows:

- ExpressoReport: This is the base interface for all the reports.
- MultiFormatReport: This is the interface for report classes that can output several formats. For example, some reporting engines can export to CSV, Excel, PDF, or HTML formats. So if the report engine you’re tying in has these kinds of capabilities, then you will want to implement this interface.
- ReportPage: This is a concrete implementation of Expresso report that has it's roots from many years ago. It can only output to HTML and stores all the elements in memory. So while it is not advised for serious reporting, it can certainly work for basic usage.

While these explanations may not give you a lot to go on, please check the javadocs on these interfaces as it has been attempted to show users what each method is supposed to do.

Running a Report

Prepare the Report

To prepare your report implementation you will need to add it to the Schema. Do this through the schema’s addReport(Class) method in your Schema’s constructor. Use it the same way you use addD-BOBJECT() or addController().

```
addReport(org.example.myapp.DonationsReport.class);
```

Accessing the Report Controller

Expresso has a simple controller implementation that allows you to run a report. The controller’s typical location is [context path]/ExpressoReport.do. Start at that location and it will lead you through the steps to run the report you have created.

If you are interested in creating your own report runner, check out the code to ExpressoReport.do. It is com.jcorporate.expresso.ext.controller.ReportServer. Check the source code for an example on how to drive the report and grab the output of the report.

Conclusion

Contributors

The following persons have contributed their time to this chapter:

- Mike Rimov

Note

Was this EDG documentation helpful? Do you wish to express your appreciation for the time expended over years developing the EDG doc? We now accept and appreciate monetary donations. Your support will keep the EDG doc alive and current. Please click the Donate button.
and enter ANY amount you think the EDG doc is worth. In appreciation of a $35+ donation, we'll give you a subscription service by emailing you notifications of doc updates; and donations $75+ will also receive an Expresso T-shirt. All online donation forms are SSL secured and payment can be made via Credit Card or your Paypal account. Thank you in advance.

[http://www.jcorporate.com/edgdoc.html]

Copyright © 2001-2004 Jcorporate Ltd. All rights reserved.
Chapter 24. Expresso Workflow

Note

If you find this EDG documentation helpful please consider DONATING! to keep the doc alive and current.

Version: Expresso 5.5
Maintainer: David Lloyd
[mailto:dlloyd@jgroup.net?Subject=EDG]

Warning: The workflow functionality in Expresso is still an alpha-quality feature. A few users have developed and use this feature, but it has not been tested well and definitely will contain some bugs.

Theory

In addition to being able to use transitions to transfer control between states and controllers, Expresso offers a basic form of workflow and allows for quite simple linear flows between states in a wizard like fashion. Each workflow state may also be re-useable in other controllers. This means that the workflow through the controller is defined at a higher level and not in the states themselves.

This approach expects controller objects to only perform 'controller' functions as described in the MVC design pattern. This should limit the controller to 'adapt' and 'mediate' type of actions. Thanks to the Controller class' support for the HTTP protocol, there should be little need for web applications to provide their own 'adapting' code. This is because Expresso provides abstractions such as ControllerRequest and PersistentSession objects that have already done the adapting work between the HTTP and Java worlds.

Keeping the mediation code away from state classes allows these objects to focus on 'model' functions as described in MVC. In particular the state classes are 'commands' as defined in the Command design pattern. The state classes accept any required input via a StateForm at the time they are invoked, perform their 'business' logic and return any results via the state's form and/or the ControllerResponse object. If the business logic that needs to be performed is fairly complex then the state should act as a type of adapter. This is adapting between a user interaction and the classes used to support that user request (note: this is NOT like the adapting role of the Controller from HTTP to Java). The state would be responsible to take the single user request and adapt it into many calls onto supporting custom business 'model' objects. The state class (and any supporting model objects) should rarely include any mediation or workflow logic otherwise this reduces the potential for reuse in different application configurations. Therefore a properly coded state will be ignorant about who called it as well as who it should call once complete.

The new features were designed with the following intent: A controller should relate to a 'use-case'. This is a single business function such as 'Register to a website' or 'Add a new widget to the inventory'. These use cases could comprise either a single user interaction or many user interactions/screens. A state relates to a single user interaction such as 'Enter address information' where the address information fits on a single screen (Note: since we kept mediation logic out of the state, this same state could be used both within a wizard-style registration and as part of a stand-alone screen to revise address information).

When a controller is coded, states are added to the controller in one of 3 ways:
Prompt states are those which build the contents of a screen using Input, Output, Block and Transition objects. The Controller will add its own transition objects based on the 'position' of the prompt state within the controller these are the Next and Previous buttons common in wizards. When errors are generated in a prompt state's associated handle state, the prompt state will be invoked by default assuming the handle state has not overridden this at runtime.

Handle states accept user input from a state form and usually perform validation on that data. These states are normally directly invoked as a result of a user action however they will also be invoked as a result of a 'final' state execution. Restricting these states to validation-type logic allows them to be reused more easily (eg. Within a wizard or stand-alone). Even in a single-screen controller it is a good practice to separate the validation from the commit processing using a final state.

Note
Keeping the handle state ignorant of its associated prompt state allows it to be reused more readily. This can be useful for example, when automating a number of user tasks in a batch or proxy mode.

Final states are the last state to be invoked in a controller. They usually contain the guts of the processing involved in a use-case. This could include database updates or any other processing that needs to occur once all validation has succeeded. Before a final state is executed, all handle states within that controller are rerun and any validation errors cause the appropriate prompt state to be redisplayed. This is useful for wizard-style screen flow.

Structuring your states in this fashion allows a use-case to be fairly easily reconfigured so that it can be used via different user interfaces. In a WAP interface for example, a new controller would be created with states defined at a lower level of granularity. The final state could then be reused to complete the use-case operation.

Data collected through the wizard process may also be conveniently held in a ControllerForm which may be given request or session level scope. See the section on session data persistence for a detailed explanation.

Implementation
The new features provide for the concept of both a state and controller form. The controller form holds all the data required by the final state to complete the use-case. The state form holds only the data required by the individual states to perform their duties. Controller forms should inherit from com.jcorporate.expresso.core.controller.ControllerForm and state forms should implement the com.jcorporate.expresso.core.controller.StateForm interface.

The state form defines the contract between the controller and the state. A state's form is specified using the addStatePairing() method as described below. It is populated from the controller form just prior to invoking the state. A new method has been defined within the State class called 'perform()'. This method replaces the existing run() method and should therefore be overridden to define state logic. This method has an additional parameter of type StateForm. Once the perform() has completed, the updated state form data is put back into the controller form. In a wizard setup, the attributes of the state form would be made up of a subset of the attributes on the controller form - this helps isolate the state from the other states in the controller. In a single screen controller, the state form would be the actual controller form.

The controller form holds all the data required to perform the use-case. For wizard controllers, the form would normally be kept in session scope as defined in the Struts config file. In these controllers, the controller form will accumulate more data as the user progresses through the screens. In single screen situations, the controller form is used as the state form.
Struts calls an ActionForm's reset() method every time it is populated from a user request. This would cause our data to be wiped out so the reset() method should not be overridden. Instead the resetController() method should be overridden. This method is only called when the initial state of a controller is invoked. The initial state is automatically defined when the controller's states are added in the controller constructor.

**Practice**

Basically, the workflow allows you to build a sequence of screens with Previous, Next and Finish buttons. Validation and error handling management to the correct states also occurs. A special transition called `enableReturnToSender()` allows for a wizard to essentially call another wizard and continue where it left off.

An example of a state flow diagram:

There are three different types of states used in Expresso workflow:

1. Prompt states: These are typically used to input user information.
2. Handle (validate) states: These are used to validate the input entered in the prompt state.
3. Final states: This is always the last state called in the wizard and usually is responsible for performing the final actions on the data collected in previous states.

The sequence of transitions between workflow states can be defined in three different places, namely:

1. In the Controller (preferred): The state sequence is defined in the Controller object, externally to the states.
2. At screen build time: The state sequence is defined at screen build time.
3. At runtime: The state itself can decide the sequence. This is not the preferred approach as remember we are trying to build re-usable states here.

**Defining the workflow in the Controller**

Prompt states are paired off with handle (validate) states using the `addStatePairing()` function. The sequence in which the `addStatePairing()` function is called essentially defines the state transition sequence or routing.

For example the following code in the Controller constructor will create three wizard screens/states that can be navigated backwards and forwards from state 1 through 3.

```java
addStatePairing(promptState1, handleState1, null);
addStatePairing(promptState2, handleState2, null);
addStatePairing(promptState3, handleState3, null);
addFinalState(handleFinalState);
```

It is not possible to achieve sequences other than a simple linear sequence using a controller definition. If your wizard requires multiple paths, then you need to use either "screen build time" or "runtime" transitions.

Errors defined in the handle states will cause a return to the prompt state with appropriate error.

A special transition can be used to route all security authorization failures to a new state (for example a login state).

```java
setControllerSecurityTransition(oneTransition);
```

or

```java
setControllerChainingTransition(oneTransition);
```

By setting the return-to-sender it is possible to return to the original place in the first wizard once the called wizard completes.

```java
oneTransition.enableReturnToSender(null);
```

**Defining the workflow at screen build time**

Divergent transitions can be coded into the prompt states to show additional buttons for state paths. These transitions may also be defined using the `enableReturnToSender()` function. Obviously these transitions are built into the State code and will appear in all wizards that choose to reuse the state code, and hence are not as favorable as Controller definitions.

**Defining workflow at runtime**
In this case the transition is coded into the handle states and usually depends on testing of some user input. For example:

```java
if (addDeviceForm.getSMSCarrier().equals("MTS")) {
    oneTransition = new Transition();
    oneTransition.setState("prompt1");
    oneTransition.setControllerObject("com.xyz.controller.ActivateMTS");
    oneTransition.enableReturnToSender(response);
    setErrorTransition(oneTransition);
    response.addError("This message will be displayed on ActivateMTS");
}
```

This piece of code will cause the wizard to branch off to the com.xyz.controller.ActivateMTS controller if the form field SMSCarrier in the StateForm is equal to "MTS".

**Workflow sample code**

Authors note: This code was taken from Aime Bazin's (the original submitter of the workflow module) readme.txt file from his original submission to the Expresso opensource mailing list. I have filled in some of the missing bits that Aime didn't feel were necessary but that tripped me up. The code has not been compiled by myself and may contain errors.

This sample code uses a ControllerForm to store user input data for the session. The ControllerForm bean is defined as follows:

```java
// Standard form bean with getter and setter methods
// This form is listed in struts-config.xml as a Struts ActionForm
// and associated with the AddDevice.do action
public class AddDeviceForm extends ControllerForm {
    private String smsCarrier;

    public void setSMSCarrier(String val) {
        smsCarrier = val;
    }

    // This method is only called upon entry to a controller so this is
    // where the wizard values can all be reset
    public void resetController() {
        setSMSCarrier(null);
    }
}
```

This bean has to be defined in the struts-config.xml file as follows:

```xml
<form-beans>
    <form-bean name="addDeviceForm" type="com.xyz.controller.AddDeviceForm"/>
</form-beans>
```

Here's the Controller code:

```java
//
// Here the 'use-case' is AddDevice
```
public class AddDevice extends Controller {

    public AddDevice() throws NonHandleableException {
        super();

        State promptState = null;
        State handleState = null;

        promptState = new PromptAddDevice("prompt1", "Add Device");
        handleState = new ValidateAddDevice("handle1", "Add Device");

        // Null in addStatePairing() indicates that a controller form
        // should be used
        addStatePairing(promptState, handleState, null);

        // More addStatePairing calls could go here if ever the AddDevice
        // function needs to split into multiple screens (ie wizard).
        handleState = new CompleteAddDevice("handleFinal", "Save");
        addFinalState(handleState);

        // The next few lines will cause controller authorization failures
        // to route to the SignIn controller.
        // The return-to-sender is activated to return back to
        // the State that caused the authorization failure.
        Transition oneTransition = new Transition();
        oneTransition.setState("prompt1");
        oneTransition.setControllerObject("com.xyz.controllerSignIn");

        try {
            // Will cause return to AddDevice on SignIn completed
            oneTransition.enableReturnToSender(null);
        } catch (ControllerException ce) {
            throw new NonHandleableException(ce);
        }

        setControllerSecurityTransition(oneTransition);

        // To use controller chaining call:
        // setControllerChainingTransition(oneTransition)
    }

    The controller also has to be registered in struts-config.xml as follows:

    <action path="/AddDevice"
        type="com.xyz.controller.AddDevice"
        name="addDeviceForm"
        scope="session"
        validate="false">
    </action>

    Note that the action name refers to the form bean (ControllerForm) we defined above, and that the scope
    is set to 'session'. The session scope will keep the values stored in the ControllerForm intact for the dura-
    tion of the session across all states. Setting scope='request' will only keep the form bean values valid
    across a single prompt/handle state transition.

    Note also that there are no action forwards defined in our example so all views will be rendered by the
default view handler.
[Authors note: It would be nice to have an example of a JSP that implements the action forwards]

Code for the sample prompt state:

```java
public class PromptAddDevice extends State {
    //
    // The 'perform' method is overridden to provide State logic.
    //
    // The StateForm class is a new interface for the new features.
    //
    public void perform(StateForm stateForm,
            ControllerRequest request,
            ControllerResponse response)
        throws ControllerException, NonHandleableException {
        super.perform(stateForm, request, response);

        AddDeviceForm addDeviceForm = (AddDeviceForm) stateForm;

        // Add Blocks, Inputs, Outputs, Transitions here...standard stuff.
        // Use stateForm to populate controller elements.
        Input I = new Input("SMSCarrier");
        Response.addInput(I);

        // This will add a link/button on the AddDevice page that will take
        // user to AuthorizeDevice page and when that controller completes
        // normally, flow returns to this prompt state again.
        Transition oneTransition = new Transition();
        oneTransition.setState("prompt1");
        oneTransition.setControllerObject("com.xyz.controller.AuthorizeDevice");
        oneTransition.setName("promptAuthorizeDevice");
        oneTransition.setLabel("Authorize Device");
        oneTransition.enableReturnToSender(response);

        // To flow to another state after AuthorizeDevice is completed replace
        // the enableReturnToSender(response) with:
        // oneTransition.addParam(Controller.CTL_SUCC_CTL,
        // "com.xyz.controller.AnotherState");
        // oneTransition.addParam(Controller.CTL_SUCC_STATE, "prompt1");
    }
}
```

Note that when populating a response with input fields that will store values in the ControllerForm. The exact name of the bean property must be used, in our case SMSCarrier. This way the default renderer will automatically fill in the ControllerForm from the Input data entered by the user.

This code also shows how to use screen time build transitions.

Code for the sample handle state:

```java
public class ValidateAddDevice extends State {
    // Validate the form.
    // If any errors are generated here then by default the
    // associated promptState will be called. This can be overridden for
    // certain runtime conditions/errors.
    //
    public void perform(StateForm stateForm,
            ControllerRequest request,
            ControllerResponse response)
        throws ControllerException, NonHandleableException {
        super.perform(stateForm, request, response);

    }
}
```
AddDeviceForm addDeviceForm = (AddDeviceForm) stateForm;

//Example of 'State Run' routing
if (addDeviceForm.getSMSCarrier().equals("MTS")) {
    oneTransition = new Transition();
    oneTransition.setState("prompt1");
    oneTransition.setControllerObject("com.xyz.controller.ActivateMTS");
    oneTransition.enableReturnToSender(response);
    //If no msg/error is added then use setSuccessTransition(oneTransition)
    setErrorTransition(oneTransition);
    response.addError("This message will be displayed on ActivateMTS");
} else {
    //This will route back to prompt state so user can correct input
    response.addError("Invalid Carrier.");
}
}

And lastly the final state

public class CompleteAddDevice extends State {

    public void perform(StateForm stateForm,
                        ControllerRequest request,
                        ControllerResponse response)
        throws ControllerException, NonHandleableException {
        super.perform(stateForm, request, response);

        // This is the final state so use the controller form
        AddDeviceForm controllerForm = (AddDeviceForm) stateForm;

        // Perform any database updates here
    }
}

Conclusion

Contributors

The following persons have contributed their time to this chapter:

- Anonymous Contributor
- David Lloyd (JGroup Expert)

Note

Was this EDG documentation helpful? Do you wish to express your appreciation for the time expended over years developing the EDG doc? We now accept and appreciate monetary donations. Your support will keep the EDG doc alive and current. Please click the Donate button and enter ANY amount you think the EDG doc is worth. In appreciation of a $35+ donation, we'll give you a subscription service by emailing you notifications of doc updates; and dona-
tions $75+ will also receive an Expresso T-shirt. All online donation forms are SSL secured and payment can be made via Credit Card or your Paypal account. Thank you in advance.

[http://www.jcorporate.com/edgdoc.html]

Copyright © 2001-2004 Jcorporate Ltd. All rights reserved.
Chapter 25. User Registration and Login

Note

If you find this EDG documentation helpful please consider DONATING! to keep the doc alive and current.

Version: Expresso 5.5
Maintainer: David Lloyd
[mailto:dlloyd@jgroup.net?Subject=EDG]

Expresso provides a sophisticated and customizable solution for registration of users into the Expresso framework and, therefore, the Expresso-based applications. This document explains the various features available with the Expresso registration system and details the procedures for customizing the features.

Introduction

In general, the process is thus. A user visits your site and finds one (or more) links to register on the site. Clicking on the registration link takes the user to a screen where they are prompted for the bare minimum information required for "express" registration. Once the user fills-in the form and registers, based on site properties, she is optionally taken to a screen for "detailed" registration. Once she completes all required entries, she is considered "fully" registered and can proceed to the login screen and actually login to the system.

The functions described below are provided by two of Expresso's controllers. The login function, including express registration, are handled by the com.jcorporate.expresso.services.controller.LoginController class, mapped to the Struts /Login action. The extended registration functions are handled by the com.jcorporate.expresso.services.controller.Registration class, mapped to the Struts /Register action. To see the various states (Expresso) or forwards (Struts) that these two controllers define, and to find out which JSPs handle the presentation, see struts-config.xml.

The login controller provides functions to register a user, validate the newly registered user's email address, login/logout users, change the user's password, and send email containing the user's password if the password is lost or forgotten. The registration controller provides a menu of the types of registration information required from the user for extended registration. It allows the user to add, update or, optionally, to delete the information.

- Express Registration
- Email Validation
- Registration Domains
- Registration Object Map and Detailed/Extended Registration
- Default Domain and Registration Object Map
- Extended Domain and Sample registration Objects
- Customization of Registration
If you decide to write your own registration and login controllers, all Expresso needs for its core functionality is: login-name, uid (integer) and email-address, and that the user be put in a user-group. If you write your own login controller, then even the password is optional since your controller will provide the password checking/email-validation/change-password/etc. functions on its own.

**Express Registration**

Express registration deals with getting the bare minimum information required to identify and authenticate an user on the site. As such, it requires a "Login Name" and a password. In addition, to communicate with the user, an email address is required. The site has control over whether the user specifies the "Login Name" she prefers, or alternatively, if the email address is used also as the "Login Name". As detailed below, the site has also a choice in specifying whether the user specifies her password, or alternatively, if Expresso generates a random initial password for the user. In either case, the user has a choice to change the password after the registration process is complete.

If extended registration is specified for a domain, then, upon completion of express registration the user is temporarily placed in a Expresso user group called "NotReg - Users Not Fully Registered Yet". The registration controller, when all extended information is provided, moves the user into the proper user group for the domain. When the user completes the registration the Admin is emailed an email that provides a link to access the registration information and approve or deny the registration.

**Email Validation**

How do you know that the email the user provided, in fact, really does belong to the user? The way Expresso does this is to send an email containing an authorization code to the email address provided by the user at registration. Once the user clicks on the link provided, Expresso will validate the user with the assumption that if the user is able to access email sent to that address, then the user owns that email address (it's your call whether this is a valid assumption or not). Once validated, in the securest form, only then is an email sent to the user containing her password.

Of course, there are various levels of customization in the authentication process. The site can choose whether the user provides the password, or whether Expresso generates one for the user. The site can choose whether an authentication step, as detailed above, is used before the password is emailed to the user. Or, the site can choose simply to assume that the user provided a valid email address to begin with and to simply send the password to that email address. In all cases, the password is stored internally in Expresso in encrypted form and the cleartext is not only never displayed on the browser, it is simply not available.

**Registration Domains**

Do you have requirements for different sorts of registration information based on the type of user? For instance, on your site, you might have a type of user that is simply interested in using the forums. For these users, maybe you decided that only express registration is sufficient (i.e. all you need is a login name, a password and an email address). The next step up maybe someone that will buy something, and so you need contact information in addition to express registration. Finally, you might want to have suppliers register on your site and you might have requirements for completely different types of registration information that they need to provide.

Expresso provides a way to easily handle these situations without having to write a whole bunch of custom controllers. All you need is to provide different Registration Domains, one each for each type of user. Registration domains are implemented as a regular Expresso DBOject, in com.jcorporate.expresso.services.dbobj.RegistrationDomain. As such, you can use the DBMaint controller to manipulate the records in RegistrationDomain. In fact, a link to do that is already provided to you on the Expresso "Security" page.
So, what are the various fields in a registration domain?

- **Name**: This is the name of the domain. This is what you specify in the "domain=" parameter to the Login controller to point a registration screen to be built for this domain.

- **Description**: A short and sweet description for this domain.

- **Default Group**: The name of an Expresso user security group, as defined in com.jcorporate.expresso.services.dbobj.UserGroup. Once registered, the user is made a member of this group.

- **Registration Controller Class Name**: If extended registration is required, the name of the controller class that will handle the extended registration process. The default registration controller is com.jcorporate.expresso.services.controller.Registration.

- **Registration Required Prior to Login?**: A "Y"/"N" field that determines whether this domain requires the user to go through extended registration or not.

- **Approval required prior to Login?**: A "Y"/"N" field that specifies that after the user completes registration, a site administrator has to approve the registration before the user can log in.

- **User specifies password?**: A "Y"/"N" field that specifies whether the user specifies their preferred password or whether Expresso randomly generates a password for the user.

- **Validate email?**: A "Y"/"N" field that specifies whether an authentication code/link is sent to the user's email address to verify correct email address before the password is sent.

- **Email as login name?**: A "Y"/"N" field that specifies whether the email address of the user is also used as the login name or whether the user has to specify a login name.

### Registration Object Map And Detailed/Extended Registration

If a registration domain specifies extended registration, then the question arises as to which set of DBObjects are to be used for the user to provide registration information for? That's where the registration object map comes in. Again, it is another plain, vanilla Expresso DBObject, in com.jcorporate.expresso.services.dbobj.RegistrationObjectMap. As such, you can use the DBMaint controller to manipulate the records in RegistrationObjectMap. In fact, a link to do that is already provided to you on the Expresso "Security" page.

A registration object map provides a one-to-many relationship between a RegistrationDomain to DBObjects. Put another way, for each domain defined in RegistrationDomain, one or more entries are provided for DBObjects in RegistrationObjectMap, each tied back to the RegistrationDomain by its DomainID. Note that the DBObjects are completely independent of the domain, i.e. a DBObject can be reused for as many different domains as needed. Here are the fields in RegistrationObjectMap:

- **Registration Domain ID**: The ID of the domain for which this DBObject will be used (the same DBObject can be repeated for multiple domains).

- **Registration Object Class Name**: The name of the DBObject class that is to be used. The only requirements are that the class be derived from com.jcorporate.expresso.core.dbobj.DBObj and that it define one field to store a reference back to the "ExpUid" user-id field in com.jcorporate.expresso.core.security.User/UserInfo.
• **Uid Field Name**: The name of the field that is to store the reference to ExpUid. Minimum records per user - The minimum number of records of this DBObject that the user has to supply before registration is considered complete. A 0 means that this information is optional.

• **Maximum records per user**: The maximum number of records of this DBObject that the user has to supply before registration is considered complete. A 0 means that there is no upper bound.

• **Fields in auto-generated forms**: A comma-separated list of field names to display on pages automatically generated by the registration controller. If the fieldname is preceded by a "!", then that denotes that the field is marked for no-display. A "+" character is pre-fixed to a fieldname for display in listing multiple records of that DBObject, so that a particular record can be selected for Update/ Delete. Display order in menu - If the domain has multiple DBObjects in the object map, then this field determines the order this database will be listed in the menu generated by the registration controller.

• **User can edit**: A "Y"/"N" field that specifies whether the user can alter the information he/she entered.

• **User can delete**: A "Y"/"N" field that specifies whether the user can delete the registration records.

**Default domain and registration object map**

Expresso comes pre-configured for a domain called "default". If no "regDomain=" parameter is supplied when the Login controller is invoked, or if "regDomain=default" is supplied, then the default domain is used for user registration. With this domain, the following behavior is seen:

• Default Group - Everybody

• Registration Controller - com.jcorporate.expresso.services.controller.Registration

• Registration required prior to login - No

• Approval required prior to login - No

• User specifies password - Yes

• Validate email - No

• Email as login name - No

The corresponding registration domain map entry has the following characteristics:

• Registration Domain ID - default

• Registration object class name - com.jcorporate.expresso.services.dbobj.RegisteredUser

You'll note that RegisteredUser does not have a Uid key field as required by the registration controller. However, for backwards compatibility, the registration controller allows an exception in only this one specific case

• Uid Field Name - ExpUid

• Minimum records per user - 1

• Maximum records per user - 1
Extended domain and registration object map

Expresso also comes pre-configured for a domain called "extended". This domain is primarily intended as an example of how to deal with multiple registration databases.

The extended domain also comes with some sample registration objects in the package com.jcorporate.expresso.ext.dbobj.regobj. These DBObjects are Person, Address, Phone, Contact. Please note that these classes are not meant to be of use to every site. These are example objects. The point is that Expresso does not care at all what registration objects are used, or even if any are used. It is up to the site to either use these sample objects or define new ones to suit local needs.

Having read this document, the best way to see how the extended domain is setup is to go to Expresso's Security page and to list the records in the Registration Domain and Registration Object Map records.

To get a taste of how it all works, go to Expresso's Security page and using the "Users" link, modify a test-user's registration domain to "extended". Then login as the test-user and from the Login page click on "Update Registration".

Customization of registration

There are several ways Expresso's behavior in this regard can be customized:

- The "default" registration domain, is after all, a record in a DBOject. One quick way to modify the registration behavior is to use DBMaint to mmodify the various fields of the "default" domain. The default registration domain is used in this way:

  <a href="<%=contextPath%>/Login.do?state=promptRegister">Register</a>

In this case, since no registration domain is specified using the "regDomain" parameter, the Login controller will check to see if the Setup database contains a value for "defaultRegDomain". If a value is found, that will be used for the name of the registration domain. Otherwise, the name "default" will be used.

If I wanted a registration process that does not requires the extra information supplied by com.jcorporate.expresso.ext.dbobj.RegisteredUser then... login as Admin, go to Security, Registration Object Map, edit default, and set the minRecords and maxRecords to zero (the default is one). Or creating your own registration object is a good course of action in the long run.

- Another way is to create a new registration domain. A typical way might be to do this in the other-Setup(...) method of your applications Schema class. The advantage of doing this is that it leaves the Expresso defaults alone. Here’s a fragment of sample code to achieve this:

  ......
  /**< Create the domain */
  RegistrationDomain rd = new RegistrationDomain();
  rd.setDBName(getDBName());

  // More code...

  rd.setRegDomain("newDomain");
  rd.setMinRecords(0);
  rd.setMaxRecords(0);
  rd.setRegDomainExpiryDays(0);
  // More code...

  RegistrationDomainDao rdDao = new RegistrationDomainDao();
  rdDao.addRegDomain(rd);

  // More code...


rd.setField("Name", "myCustomDomain");
if (!rd.find()) {
    rd.setField("Description", "My Valued Customers");
    rd.setField("ReqRequired", "Y");
    rd.setField("GroupName", "myValuedCustomers"); // Assume this group indeed exists
    rd.setField("ControllerClass", "com.jcorporate.expresso.services.controller.Registration");
    rd.add();
}

/* Map a DBObj to the domain */
RegistrationObjectMap rdm = new RegistrationObjectMap();
rdm.setDBName(getDBName());
rdm.setField("RegDomId", rd.getField("RegDomId"));
rdm.setField("RegObj", "com.mycompany.dbobj.MyCustomer");
rdm.setField("UidField", "CustomerId");
if (!rdm.find()) {
    rdm.add();
}

/* Map another DBObj to the same domain */
rdm.clear();
rdm.setField("RegDomId", rd.getField("RegDomId"));
rdm.setField("RegObj", "com.mycompany.dbobj.MyCustomerAddr");
rdm.setField("UidField", "CustomerId");
if (!rdm.find()) {
    rdm.add();
}

• The login/registration JSP's must be modified to take advantage of the custom registration domain(s). The JSPs are in /expresso/jsp/register/*.jsp. In particular, the links to the registration controller are defined in the file header.inc. The registration domain name must now be supplied to the Login controller as follows:

    <a href="<%=contextPath%>/Login.do?state=promptRegister&regDomain=myCustomDomain">Register As A Customer</a>  
    <a href="<%=contextPath%>/Login.do?state=promptRegister&regDomain=myCustomDomain2">Register As A Supplier</a>

• If you'd rather not change the Expresso-supplied JSPs, then another option might be to copy the JSPs over to another directory and then modify the "forward" definitions for the "Login" action in struts-config.xml.

Conclusion

Contributors

The following persons have contributed their time to this chapter:

• Shash Chatterjee
• Sandra Cann
Note

Was this EDG documentation helpful? Do you wish to express your appreciation for the time expended over years developing the EDG doc? We now accept and appreciate monetary donations. Your support will keep the EDG doc alive and current. Please click the Donate button and enter ANY amount you think the EDG doc is worth. In appreciation of a $35+ donation, we'll give you a subscription service by emailing you notifications of doc updates; and donations $75+ will also receive an Expresso T-shirt. All online donation forms are SSL secured and payment can be made via Credit Card or your Paypal account. Thank you in advance.

[http://www.jcorporate.com/edgdoc.html]

Copyright © 2001-2004 Jcorporate Ltd. All rights reserved.
Chapter 26. Logging

Note

If you find this EDG documentation helpful please consider DONATING! to keep the doc alive and current.

Version: Expresso 5.5
Maintainer: David Lloyd
[mailto:dlloyd@jgroup.net?Subject=EDG]

One of the main advantages of linux and other multi user systems is the logging ability. Expresso integrates with Logging standards to create logs, thereby documenting and storing certain system or user activity. This document seeks to describe the logging mechanism used within Expresso. It will cover basic requirements, rationale behind various decisions and a detailed walkthrough of the various logging categories and the purpose of each category.

Introduction

One of the main advantages of linux and other multi user systems is the logging ability. What is logging exactly? It is merely a process of creating logs, documenting and storing certain system or user activity for various reasons. Entire process and idea of logging probably came from programming, where it was useful to record events, such as program behaviour for further development, and to collect such information for future diagnostic and improvement. Same is with logging on the system level. It is, to say the least, useful to log all activities and events that took place on your system for future analysis. From a security point of view, logging serves as a proof of malicious activity or as an indicator of a potential one.

Any framework needs logging facilities to assist in the development of applications on top of the framework. The basic requirements of this logging facility is outlined below:

• Such a logging facility will need to be able to log to multiple targets such as a database, file, socket, etc.

• The facility needs to be able to log different types of unforeseen messages.

• Various types of logging messages should be able to be turned off and on without having to modify the original source code.

Apache Log4j Integration

Expresso is integrated with the Apache Log4j API [???]. Log4j is an open source project based on the work of many authors. It allows the developer to control which log statements are output with arbitrary granularity. It is fully configurable at runtime using external configuration files. Best of all, log4j has a gentle learning curve. To achieve these requirements. We have selected the logging system log4j for the following reasons:

• Good hierarchy of categories allows for extremely flexible logging options.

• Highly tested and recommended from a wide variety of sources.
• Configurable with a single file.
• Fast.
• Allows for multiple targets for logging.
• Large amount of industry support.

The LogManager class is Expresso’s interface to the Apache Log4j framework, and is used to set up logging for all applications in a particular environment. When ConfigManager first calls LogManager during system startup, all files with names of the form "*Logging.xml" in the configuration directory are read, and logging channels are then available for each of the channels defined in these files. This allows each application to have its own separate logging configuration file, e.g. espressoLogging.xml for Expresso's own internal classes, eForumLogging.xml for eForum, and so forth.

The primary means for monitoring the intranet system is via a log. All activities run and monitored by Expresso are logged into an HTML server log, accessible via any web browser. This allows the server to be monitored remotely as required, for completely remote system administration and operation. Most tasks are triggered by the application framework, and it writes status notifications, warnings and error messages to the log. Multiple logs can be set up for separate facilities, or one master log can be used for centralized collection of log data. For more information please refer to the Log4j project [???] and the Log4j manual. [??]

Expresso Extensions

To have Log4J fit into Expresso with the best capabilities, there will have to be some additions made other than simply using the straight logging code:

• **Categories**: Expresso will come predefined with a set of categories that will be useful to both the framework developer, and the framework user. They will include categories such as SQL tracing, ServletRequest Logging, Security Logging, etc. The categories will be detailed later on in this document.

• **Appenders**: Expresso currently does the bulk of its event logging to a database table. As such, a DBAppender needs to be created.

• **static final traceDebug**: A global static final integer will be used to filter out all debug messages. This way, with a single line of source code modified, all debug statements will be optimized out of the source code providing for faster, smaller source code.

Package Organization

All logging operations will be encapsulated within the com.jcorporate.expresso.core.logging package. It needs to be located there instead of within the services subtree because the logging code will be integrated throughout the entire framework and will not be easily separated out. Package organization will be:

• com.jcorporate.expresso.core.logging will contain the global debug switch file, the logging configuration properties file, the static initializer for the logging system, and any other miscellaneous files that would not fit any of the other subpackages.

• com.jcorporate.expresso.core.logging.appenders will initially contain the DBAppender as well as any other appenders that may be added at a later time. Examples include an HtmlCommentAppender to dump logs into the servlet output stream, and JMSAppender to dump log activity into a Java Mes-
sage Queue.

Configuration Mechanism

Log4j supports both properties files and XML configuration files. Due to the potentially complex configuration options of the categories, it is recommended that an XML configuration file be used.

The second configuration mechanism will be the class com.jcorporate.expresso.core.logging.debugLogging which will contain the public static final boolean member variable "debug" Which will be used to turn on/off any references to debug logging. The EspressoLog class will also contain the factory method to initialize logging throughout the framework at startup time.

Finally, the expresso config file will need to point to a directory where events will be logged to.

Appender Types

The appender types are the following:

- **DBAppender**: This class will provide the functionality to log system events to the database table syslog. It will take care of the appropriate color coding necessary, and will seek to provide the same functionality that currently exists in Expresso Logging.

- **HTMLCommentAppender**: This class will allow for embedding log statements as comments within an html page returned by a servlet. This will assist in rapid debugging and is easily turned off for a secure production environment.

- **SessionAppender**: This appender will store the log information into the session. The servlets can have a javascript window open up which calls a servlet that will dump the session log into a web page.

- **JMSAppender**: Expresso J2EE Only This will give the ability of logging to be dumped to a Message Queue Server. This gives the framework secure remote monitoring ability as the JMS messages will be distributed over SSL instead of in the clear. It is recommended that this appender be donated to the log4j initiative to provide better debugging / iterations as time goes on, and this appender is listed in the “TODO” list for upcoming features of log4j anyway.

Available Appenders

This section details the appenders that have been set up in the default configuration file. If you wish to change how a category is logged, please check here for names of the default appenders.

<table>
<thead>
<tr>
<th>Designation</th>
<th>Description</th>
<th>Appender Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>expressoLog</td>
<td>The log file that expresso uses to commit any special warnings/errors that would not normally be able to be sent to the database. The expressoLog filename/location is set in the Expresso Configuration file with the entry</td>
<td>FileAppender</td>
</tr>
</tbody>
</table>
Categories

This section is by far the most detailed portion of this document. Categories set up in a tree-like fashion. The overall goal is to provide good grouping by functionality as well as class hierarchy. For each category listed, there will be the following information provided:

**Note**

Note, the chosen appenders and priorities are only defaults. They may be changed at any time by the configuration file.

<table>
<thead>
<tr>
<th>Designation</th>
<th>Parent</th>
<th>Description</th>
<th>Default Appenders</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>espresso</td>
<td>&lt;root&gt;</td>
<td>The base category for all log messages from the framework and any other derived works.</td>
<td>espressoLog</td>
<td>Error</td>
</tr>
<tr>
<td>espresso.core</td>
<td>espresso</td>
<td>The base category for dealing with events/ logging in Expresso's core services.</td>
<td>espressoLog</td>
<td>warn</td>
</tr>
<tr>
<td>espresso.services</td>
<td>espresso</td>
<td>The base category for dealing with events/ logging in Expresso's added services.</td>
<td>FileAppender</td>
<td>Error</td>
</tr>
<tr>
<td>espresso.core.security</td>
<td>espresso.core</td>
<td>The base category for all security related messages such as failed login attempts, failed ACL requests, failed JVM security context errors, as well as other security checks.</td>
<td>expressLog dbLog</td>
<td>Info</td>
</tr>
<tr>
<td>espresso.core.security.Administrative</td>
<td>espresso.core.security</td>
<td>This category is active for all logged in super-</td>
<td>expressLog dbLog console</td>
<td>Info</td>
</tr>
</tbody>
</table>
visors. All actions that the supervisor performs are then traced. Examples include login, event-clearing, addition of users, etc. All servlets should report the actions taken by an Administrative session.

<table>
<thead>
<tr>
<th>Category</th>
<th>Base Category</th>
<th>Description</th>
<th>Logger</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>expresso.core.db</td>
<td>espresso.core</td>
<td>The base category for all db log messages from the framework.</td>
<td>expressoLog</td>
<td>Warn</td>
</tr>
<tr>
<td>expresso.db.SQL</td>
<td>espresso.core.db</td>
<td>The category for tracing all SQL calls made to underlying database objects.</td>
<td>expressoLog</td>
<td>Error</td>
</tr>
<tr>
<td>expresso.core.cache</td>
<td>espresso.core</td>
<td>The base category for any caching messages, tracing.</td>
<td>expressoLog</td>
<td>Warn</td>
</tr>
<tr>
<td>expresso.core.jsdkapi</td>
<td>espresso.core</td>
<td>Tracking all servlet-api calls</td>
<td>dbLog console</td>
<td>Warn</td>
</tr>
<tr>
<td>expresso.core.jobhandler</td>
<td>espresso.core</td>
<td>For tracking all job related information.</td>
<td>dbLog</td>
<td>Warn</td>
</tr>
<tr>
<td>expresso.core.servlet</td>
<td>espresso.core</td>
<td>The base category for any servlet-related logging messages. Individual servlets should create their own logging categories as a subcategory of expresso.servlet.</td>
<td>dbLog</td>
<td>Warn</td>
</tr>
<tr>
<td>expresso.core.servlet.controller</td>
<td>expresso.core.servlet</td>
<td>The category for tracking state listing and changes within the controller package.</td>
<td>dbLog</td>
<td>Warn</td>
</tr>
<tr>
<td>expresso.services.dbTool</td>
<td>expresso.services</td>
<td>Category for tracking all uses of DB-Tool.</td>
<td>expressoLog</td>
<td>Info</td>
</tr>
<tr>
<td>expresso.services.servlet</td>
<td>expresso.services</td>
<td>The base category for any servlet-related logging messages. Individual servlets should create their own logging categories as a subcategory of expresso.servlet. This covers all servlets that are not part of</td>
<td>dbLog</td>
<td>Warn</td>
</tr>
</tbody>
</table>
How to use Expresso Logging

At first, the number of possibilities for log4j may seem bewildering. However, to use it within your own work with Expresso is quite simple. Here's an example code snippet within your class:

```java
import org.apache.log4j.Category
...
 LogManager.instantiate();
 static Category log = Category.getInstance("expresso.ext.controller");
...

log.error("This will log if the framework is set to log errors");
log.warn("This will appear if the framework is set to log errors and warnings");
log.info("Information message: Will only appear if framework is set to log errors, warnings, and information");
if (log.isDebugEnabled()) {
    log.debug("This is a debug message. Wrap it in the isDebugEnabled() to speed performance");
}
```

What this code does:

- **LogManager.instantiate()**: You call this before ever attempting to get a Category instance. The log manager handles loading the logging XML configuration files. If the system is already initialized, the function will return without doing anything. You usually set up the Category variable to be a private static attribute that only gets initialized when the object instantiates.

- **Category.getInstance()**: This function loads the actual "logging category" that you wish to use. What the categories do will depend on the hierarchy set up. You may browse through the WEB-INF/config/expressoLogging.xml file in the config directory to choose which category you wish to use. If a category doesn't exist, then you can create a category that will inherit all the attributes of the parent category by simply adding to the parent category name: Example - Category.getInstance("expresso.ext.controller.download") will create a category that will have "expresso.ext.controller" as a parent and will do everything that expresso.ext.controller will do configuration-wise.

- **log.error()/log.warn()/log.info()/log.debug()**: These are the gut logging functions. The string they are supposed to log may or may not appear depending on how the logging configuration file is set up. In this case, if expresso.ext.controller's priority is set to error only, then only log.error() messages will actually produce any output. If its priority is set to "debug", then all messages will appear.

- **log.isDebugEnabled()**: Log4J was designed for performance. As such, the first thing it does is de-
How to Add logging to your own Expresso-based application

You can easily add logging to your own application. To do this, you:

• **Design your category hierarchy**: You'll want to think about how to create your category hierarchies. In Expresso, for example, we have a split hierarchy for each major branch: ext, core, and services. You should look at the eforumLogging.xml, and ePollLogging.xml to get an idea of how you can set up your own hierarchies.

• **Design your appender hierarchy**: Each category can be set up to be dumped to different locations. You should consider what should go where. Explore the available log4j appenders including our own supplied DBAppender to decide where you want things to go. Again, we encourage you to check out the logging configuration files to get an idea of how to set this up.

• **Create your own logging xml file**: After the design decisions are made, you'll want to create your own logging xml file and put it in your configuration directory. Expresso's LogManager will automatically load all xml files in the configuration directory and merge the categories into the existing expressoLogging.xml category tree. Please note: Do not define your own <root> node - Expresso already defines its own root node, and your own root will conflict with the rest of the system.

• **Add logging code as detailed above**: After your own logging.xml file is created, you can then add logging code to your own application.

We recommend that you study the Log4J manual included with the expresso distribution for a full feeling of the power of this logging framework.

Production Capable High Speed Logging

Logging can be a serious performance drain if it is set up incorrectly - so it is important to have logging configured for your production environment for performance. Chief Points are:

**Appenders**

The following persons have contributed their time to this chapter:

• **Remove all debug-information generation**: But using a ConversionPattern as shown below:

  ```xml
  <param name="ConversionPattern" value="%d %-5p [%t] - %m
"/>
  ```

  We remove all debug information from the logging event. For example, stack traces will not be shown, nor will calling file location be displayed. This greatly improves the efficiency of the logging event.

• **Move logging to files, syslog or NT Event Log**: Database logging can be expensive on the target database. It is better to log to a file at a specified location rather than to a database server. Since db-
Logging

Log is the only appender using the database. Change that value and all categories using that append-er will now log to a file instead of the database. <appender name="dbLog" class="org.apache.log4j.RollingFileAppender"> <layout class="org.apache.log4j.PatternLayout"> <param name="ConversionPattern" value="[%t] - %m"/> </layout> </appender>

Note

this example uses the RollingFileAppender - This allows the log to get rolled over whenever it reaches 10 Mb in size. Perfect for a long term production environment. Other options to invest-igate are: SyslogAppender, DailyRollingFileAppender, and NTEventLogAppender. Check the log4j javadocs for more information on these classes.

Categories

Move category priorities to warn or error only. By setting the categories to log log.error() and maybe log.warn() you effectively eliminate all the overhead of logging various categories unless something ser-ious happens. Here's an example: <category name="expresso.core" additivity="false"> <priority value="error"/> <appender-ref ref="expressoLog"/> </category> This sets all espresso.core and subcat-egories to only log on Errors. Note that any hierarchies that have additivity="false" must also be set be-cause their parent categories have no effect on them.

Conclusion

Contributors

The following persons have contributed their time to this chapter:

• Michael Nash
• Sandra Cann

Note

Was this EDG documentation helpful? Do you wish to express your appreciation for the time expended over years developing the EDG doc? We now accept and appreciate monetary dona-tions. Your support will keep the EDG doc alive and current. Please click the Donate button and enter ANY amount you think the EDG doc is worth. In appreciation of a $35+ donation, we'll give you a subscription service by emailing you notifications of doc updates; and dona-tions $75+ will also receive an Expresso T-shirt. All online donation forms are SSL secured and payment can be made via Credit Card or your Paypal account. Thank you in advance.

Copyright © 2001-2004 Jcorporate Ltd. All rights reserved.
Appendix A. Contributing To Expresso Framework

Editing The Expresso Developers Guide

DocBook, XML, and XSLT

In order successfully modify the EDG you need to learn about DocBook [http://docbook.org/tdg/en/html/docbook.html]. DocBook consists of two parts:

- The Docbook DTD; the grammar rules contains in a Document Type Definition file.
- The Docbook XSL stylesheets - extensible mark-up stylesheet language files, special template files that describe how to convert raw XML to printable format for presentation e.g. HTML for the web browsers.

Since Docbook has its roots in SGML (Standard General Mark-up Language). SGML is a pre-cursor to XML that originated in the 1970s. You will see many SGML references on docbook websites, but you can safely ignore them. In order to modify the developer guide you have to located. It can be found in the web container under `${CONTAINER_HOME}/webapps/expresso/expresso/doc/edg/edg.xml`.

The file is a DocBook XML file in order to convert the raw data to a usual output medium you simply have to locate the the appropriate stylesheets. They exist for:

- HTML - hypertext mark-up language
- FOP - formatted object protocol (e.g Adobe PDF)
- RTF - Rich Text Format
- XHTML - Extensible Mark-up well formed HTML
- HTML-Help - A Microsoft Windows Html Help file (.chm)

Most people are interested in converting DocBook XML to HTML in order to create the EDG as an HTML file. Here is the line in the `build.xml` file that does that execution, it should help you run it from the command line:

```xml
<java classname="org.apache.xalan.xslt.Process" classpath="${buildclasspath}" maxmemory="64m" failonerror="true" fork="true">
</java>
```

Essentially one uses the commands above to generate other types of document such as Rich Text Format or Portable Document Format. Here are some other links of interest for dealing with DocBook:
Transforming XML to HTML

In the CVS repository change to the EDG directory. This should be the directory that contains `edg.xml'. NB: The example assumes a compatible Linux or UNIX operating system. The CVS repository is stored in Mr Tony Blair's home directory at `/home/blairt/proj/expresso'. So we change to the EDG directory.

% cd /home/blairt/proj/expresso/expresso-web/expresso/doc/edg

Use a XML Stylesheet Language Template (XSLT) engine to transform the DocBook XML to a well formed HTML document `edg.html'. In this example we will use Apache's Xalan to perform this. The stylesheets are located in subdirectory `docbook-xsl/html' of the EDG directory. You may also need to set the classpath to find the Xalan JAR file.

% java -classpath /home/blairt/proj/lib/WEB-INF/lib/xalan.jar:$CLASSPATH \
org.apache.xalan.xslt.Process \
-IN edg.xml -xsl \
docbook-xsl/html/docbook.xsl \ 
-OUT edg.html

In order save a lot of typing I would put the above command in a Shell script. Here is an example of a such a quick-fix shell script

#!/bin/sh
# Shell script `Convert-EDG-Xml2Html'
# Peter Pilgrim Mon Sep 30 17:35:50 BST 2002
myname=`basename $0`
JAVA_CMD=java
XALAN_JAR=${HOME}/proj/lib/WEB-INF/lib/xalan.jar
echo "$myname: Transforming Expresso Developer Guide from XML to HTML."
cd ${HOME}/proj/expresso/expresso-web/expresso/doc/edg

(set -x verbose; ${JAVA_CMD} -classpath ${XALAN_JAR}:${CLASSPATH} \ 
org.apache.xalan.xslt.Process \ 
-IN edg.xml -xsl \ 
docbook-xsl/html/docbook.xsl \ 
-OUT edg.html )
status=$?
if [ $status -ne 0 ]; then
echo "$myname: XSLT Transformation failed (error status:$status)" 1>&x003E;&amp;
exit 1
fi
# End-of-script

Generating A Unified Difference Patch

A unified diff patch is the format that is used by Linux Kernel developers. This is a special type that pre-
serves the differences between source files. On UNIX and Linux a context diff file is created with the
diff command.

```
% diff -u acme.java.org acme.java > acme.java.diff.peter
```

Here a sensible developer backed up the original source and then made some modifications to a copy file. He, then, created a patch, a unified diff file. He sent the patch files to the lead developer who if she
is also running a UNIX system can applied the patch. The lead developer will apply the patch like this:

```
% patch -p0 < acme.java.diff.peter
```

Often a contributor has made a lot of changes across many files. On UNIX system a shell script can help reduce the work of creating patches. Here is a shell script that helps to generate unified diff patch. This scripts searches for files that suffixed by a particular extension, say `.orig'. The script creates patches, unified diffs, for each of these files automatically.

```bash
#!/bin/sh
# Program `generate-patch' Mon Sep 30 02:08:40 BST 2002

SysWarn ()
{
    echo "${myname}: *WARNING*: $*" 1>&2
}

SysError ()
{
    echo "${myname}: *ERROR*: $*" 1>&2
    exit 1
}

myname=`basename $0`
if [ -z $1 -o -z $2 ]; then
    cat <<-XEOF
    USAGE: $myname <DIRECTORY> <DIFF-EXTENSION>
    Generates a software patch for any development application.
    This program prints the diff to the standard output.
    e.g.
    % $myname com/xenonsoft .orig
    XEOF
fi

directory=$1
extension=$2
if [ ! -d $directory ]; then
    SysError "no such directory: \`$directory'"
fi
if [ ! -r $directory ]; then
    SysError "cannot read directory: \`$directory'"
fi

find $directory \( -name "*$extension" -o -name "*.$extension" \) -type f -print | while read file1
  do
    # e.g file1=acme.java.org => file2=acme.java
    file2=`echo $file1 | sed 's!$extension$!!'
    if [ ! -f $file2 ];then
```
The script is typically used like this. I am assuming UNIX again.

% cd /home/blairt/proj/expresso
% generate-patch \ expresso-web/WEB-INF/src/com/jcorporate/core/dbobj \ .orig > dbobj.diff.peterp-20020930

If you have a working CVS repository on your system then you can edit the diff line in the script file, so that you get difference up-to-date and accurate from CVS. Simply replace the the diff with the following

cvs diff -u $file2

The information in the section is distilled at Jakarta Apache Bug Submission web page [http://jakarta.apache.org/site/bugs.html]

**Working With Other Databases**

Due to natural testing habits, Expresso works best with the databases the core development team works with the most. These tend to be:

- MySQL [http://www.mysql.com/]
- SapDB [http://www.sapdb.org/]
- PostgreSQL [http://www.postgresql.org/]

Other databases that are frequently used by community members but may not have up to date documentation or might have some testing issues are:

- Interbase [http://www.borland.com/interbase]
- Oracle [http://www.oracle.com/]
- DB2 [http://www-3.ibm.com/software/data/db2/]
- Firebird [http://www.firebirdsql.org/]
- HSQL (Formerly Hypersonic) [http://hsqldb.sourceforge.net/] This database gets quite a bit more testing than the others because Expresso is distributed with this database. However, since it isn't used during the normal development process, it isn't as thoroughly tested as the ones listed above.
So what do you do if Expresso doesn't directly support your native database? Not a problem! The following section will help guide you through the process of how to port Expresso to your database. We will include sections on how to get a quick "Up And Running" and then later show you how to test the system so that you know that your configuration works for all Expresso-supported data types.

Before we proceed, let us encourage you that porting to a database is seldom a really painful process. Perhaps a field will need to be renamed, or perhaps the CREATE TABLE statement will have to be slightly modified, but in most cases, you won't have to modify Expresso's code at all.

**Note**

If you get another database up and running, *please please please please* (Have we made it clear?) submit your patches and or supporting configuration file to the community! You will save yourself A LOT of work in the long run as contributing database documentation will get more people using that database, and thus more eyes on the code to spot bugs.

[End "Soap Box"]

**Quick Start**

The database porting process can, in a short diagram, be summed up like this:
Creating your initial database setup

You will need to create a skeletal jdbc entry in the database to get started. To find the entries you want, take a look at the database JDBC documentation. First off, a sample of a skeletal configuration:

```xml
<context name="hypersonic" active="true">
  <description>Hypersonic Database</description>
  <jdbc>
    driver="org.hsqldb.jdbcDriver"
    url="jdbc:hsqldb:%web-app%WEB-INF/db/default/default"
    connectFormat="3"
    login="sa"
    password=""
    cache="y"
    createTableIndices="true"
    limitationPosition=""
    escapeHandler="com.jcorporate.expresso.core.db.DoubleQuoteEscapeHandler"
  </jdbc>
</context>
```
Most of the items can be left as is, but the key would be the entries for the <jdbc> tag. Here is where you
can get that information from your documentation.

**Table A.1. JDBC Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>What To Set It To</th>
</tr>
</thead>
<tbody>
<tr>
<td>driver</td>
<td>Look at your driver docs. The value in Class.forName(&quot;database driver class&quot;); will be what you want to set this value to.</td>
</tr>
<tr>
<td>url</td>
<td>Look for the URL the database driver is connecting to. Put that value here.</td>
</tr>
<tr>
<td>connectFormat</td>
<td>Most databases will work well with &quot;1&quot; Try other varieties if you have connection issues.</td>
</tr>
<tr>
<td>login</td>
<td>Depends on your database. The connection account Expresso should use for the database</td>
</tr>
<tr>
<td>password</td>
<td>Depends on your database. The connection password Expresso should use for the database</td>
</tr>
<tr>
<td>cache</td>
<td>Usually set to &quot;true&quot;</td>
</tr>
<tr>
<td>createTableIndices</td>
<td>Set to &quot;true&quot; unless your database has weird index creation routines and DBCreate fails on index creation. In this case you'll need to create database indices manually.</td>
</tr>
<tr>
<td>limitationPosition</td>
<td>Leave Blank</td>
</tr>
<tr>
<td>escapeHandler</td>
<td>In most cases set to: com.jcorporate.expresso.core.db.DoubleQuoteEscapeHandler. This is for filtering the SQL strings against sql injection attacks so characters like &quot;&quot; will work in a database query properly. If you have trouble with database values, you MIGHT want to set it to:</td>
</tr>
<tr>
<td>Attribute</td>
<td>What To Set It To</td>
</tr>
<tr>
<td>-----------------------</td>
<td>----------------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>com.jcorporate.expresso.core.db.DefaultEscapeHandler</td>
</tr>
<tr>
<td>skipText</td>
<td>Leave to true. What this does is exclude 'text' (not varchar) fields from searches. This is best for database efficiency even if your database does support searches in long text fields.</td>
</tr>
</tbody>
</table>

### Launching and running DBCreate

Now launch your app server. Go to the expresso home page (expresso/frame.jsp), click on “Setup” and run 'Create and Verify Database' this is otherwise known as DBCreate. Have all boxes checked and run it.

### Troubleshooting

If DBCreate fails, take a close look at the error message. In particular, take a look at the generated SQL Expresso claims to have attempted to execute against the database. Once you isolate the issue that is causing the problems, take a look at the solutions listed below.

#### Table A.2. DBCreate Failure Issues

<table>
<thead>
<tr>
<th>Symptoms</th>
<th>Cause</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weird database types. Example:</td>
<td>Incorrect Database Type Mappings</td>
<td>Add a new database type mapping to the config file.</td>
</tr>
<tr>
<td>Identity instead of Integer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Invalid Primary Key Generation</td>
<td>Expresso is generating incorrect primary keys that your database</td>
<td>Modify TableCreator code to create statements compatible with your database.</td>
</tr>
<tr>
<td>Statements.</td>
<td>doesn't understand.</td>
<td></td>
</tr>
<tr>
<td>Create statement is failing,</td>
<td>Check a list for your database of reserved words. Chances are either</td>
<td>Rename the offending fields or table name to something more innocuous and submit a patch to the Expresso community.</td>
</tr>
<tr>
<td>everything looks ok in the</td>
<td>the table name or the field names are reserved for your database</td>
<td></td>
</tr>
<tr>
<td>statement.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Most of the time you will need to simply adjust the type mappings on your database. To add an entry, take a look at the DBObjecet class that the create is failing on. Find out what type it is supposed to be and add a line to your config mapping that will look similar to this:

```xml
<type-mapping>
  <java-type>TINYINT</java-type>
  <db-type>SMALLINT</db-type>
</type-mapping>
```

On rare occasions, you might have to modify source code. The following are the most typical scenarios that may happen:

- Reserved Table or Field Name When porting to a new database, often table names or field names will be reserved keywords in your database. If it is, you will have to rename either the table or the
field name. Do this via a global search and replace. Submit the changes to the Expresso community so that we can have the changes included in the next major revision. To deemphasize the impact this may have, the best practice in Expresso is to use string constants in naming database field names. However, there are several areas where this switchover has not yet been completed.

- Invalid Primary Key Generation There have been several databases that have unique ways of generating primary key statements. The class com.jcorporate.expresso.core.db.TableCreator is the location where all this code is done. To patch it for your particular database, add a new if(drivername == "com.yourdatabase.Driver") { [create primary key] } type of statement.

Final Work

DBObject Unit Test

Now that you have DBCreate working, you're probably about 80% of the way to a completely correct configuration. Expresso itself doesn't use all the supported database types in its tables. Therefore, you'll want to run some of the more thorough tests to make sure that your configuration is fully working with your database.

To do this, set up a Unit Test run. To do this, locate the article about Unit Testing in the Expresso documentation bundle. The test you're going to run is com.jcorporate.expresso.core.dbobj.tests.DBOBJECTTest. Run the test, (The unit test has a main[] function, so you can just "Run" the class and thus perform the test) and perform the same process you did as in the quick start. Once you have all the database unit tests passing you have will have successfully mapped all supported Expresso unit test types!

BLOB Tests

The following section is optional. Your database may or may not support Large Objects (LOBS). So the following test is optional depending on your needs.

Set up another unit test just like you did in the previous DBObjectTest. This time run the class com.jcorporate.expresso.core.dataobjects.jdbc.tests.LobFieldTest. This will get you an idea if the database at least supports some sort of Large Objects.

Note

Once this passes, your database still might not support LOBs directly! Test your database by trying to insert an object the size of the maximum object size you wish to use in your deployment. Make sure your app won't run out of memory inserting it and make sure the database driver doesn't corrupt the LOB in any way. You may need custom code at this point if your tests fail. We did such code for the Oracle database. Take a look at com.jcorporate.expresso.core.dataobjects.jdbc.LOBField to see if you can modify it to support your database. Again, submit any patches to us for inclusion in later releases!

At this point, you have reasonable assurance that your database configuration file is working correctly. Please submit your new config file With passwords, usernames, and machine names stripped to the Expresso community for inclusion in the future documentation.

Conclusion

Contributors
The following persons have contributed their time to this chapter:

- Mike Rimov
- Mike Traum (JGroup Expert)

**Note**

Was this EDG documentation helpful? Do you wish to express your appreciation for the time expended over years developing the EDG doc? We now accept and appreciate monetary donations. Your support will keep the EDG doc alive and current. Please click the Donate button and enter ANY amount you think the EDG doc is worth. In appreciation of a $35+ donation, we'll give you a subscription service by emailing you notifications of doc updates; and donations $75+ will also receive an Expresso T-shirt. All online donation forms are SSL secured and payment can be made via Credit Card or your Paypal account. Thank you in advance.

[http://www.jcorporate.com/edgdoc.html]
Appendix B. The Expresso Component Architecture [Future Direction]

Introduction

Note

This chapter is a discussion of a future architectural concept for Expresso. If you are just learning Expresso, then you can skip this chapter without missing out on any of Expresso's features. If you wish to participate in the direction that Expresso develops in the future, then this chapter is for you! Please post your comments about the design and code to the Expresso forums or OpenSource mailing list.

When you read this, you may be thinking to yourself. "Why are we doing this? What is wrong with the current architecture." So before we get into the specific, let's discuss the reasons for going into this refactoring effort in the first place.

First off, Expresso has grown huge. We need a way to refactor the system so that a minimalist application can exist without 8 mb of object code. We also need to provide a way that people can understand Expresso piece by piece without feeling overwhelmed.

The second problem that has readily come apparent as I've come aboard is that Expresso has MANY interdependencies. There is absolutely no way to take one part of Expresso and replace it with something else, or worse yet, try to remove a piece all together. As the developers, we need a way to separate and divide the framework so that we can more easily change code without fear of breaking other areas in the framework.

Goals

Some basic goals are:

1. Provide a clean way to componentize pieces of Expresso
2. Provide a basic set of interfaces that allow people to plug in their own components in many areas of Expresso.
3. Provide the ability to manage the components in a consistent way, including provide an ability to eventually interface well with Java Management Extensions.
4. All meta-data about the component and its settings are externalized into xml files... component definitions if you will. However, this will be optional to give maximum flexibility to the component writer. Some people hate writing XML definitions, and this will provide them that capability. However, for Expresso proper, we will encourage the use of XML component definitions.

Architectural Overview

The basic architecture will be similar to JBoss in the sense that there will only be a 'microkernel' left as the core, and the rest of the framework will then become services and extras sitting on top of the microkernel. After some deliberation, I decided to not go with the concept that everything is an MBean like JBoss does. The reasons are at the time:
• Would require yet another 3rd party library, to develop with such as MX4J or Sun's reference imple-
mentation.

• Implementing our own high performance version of JMX would take significant time.

• The author, at the time of this writing, is strongly suspicious of the possibility of opening up security
holes by exposing everything to potential remote management through JMX. This may or may not
turn out to be true, but as far as security goes, paranoia pays off.

Instead, we'll make our own component definition and metadata definitions, immediately provide a con-
troller management interface, and in the near future provide a bridge to allow exposure of all the Ex-
presso components to JMX management if the web site operator so chooses. However, using a standard
controller, web site operators will still be able to have the full configuration power that a JMX user
would have access to.

We'll accomplish this through describing the components through XML files. This way, we can later on
provide transformation sheets for

• Upgrade Paths: Allow easy and automatic upgrade by applying various stylesheets against the vari-
ous xml documents.

• JMX: Projects such as Apache Modeler take XML definitions and construct MBeans out of them. By
using a platform neutral XML file that contains basically the same information, we can provide a
stylesheet to perform on-the-fly transformation and interfacing with Apache Modeller to provide in-
stant JMX capabilities.

What makes a component then?

Some things are easy to spot as a component. Expresso's Cache or DBCConnectionPools are obvious can-
didates for componentization. Expresso's Schema class (and its derivatives) has traditionally been con-
sidered a component, but it seems like it would have different needs than a standard component. The
proposed architecture actually unifies Schema as just another component with a special section in its
metadata definition.

Architectural Tradeoffs

Of course, with any architecture there are a series of tradeoffs. It is the goal here to make the adjust-
ments:

• When judging component simplicity vs ComponentManager complexity, we will go with the side of
component simplicity. The basic goal is to allow a component writer to make a component as simple
as possible. By doing this, we will provide a baseline of interfaces, and provide further interfaces to
provide additional basic functionality if the component writer so chooses.

• Simplicity vs Extensibility: The component manager interface should try to err on the side of extens-
ibility even at the cost of its own implementation being a little more complex. This will allow folks
to plug in the Expresso runtime kernel into a wider variety of applications. However, care should be
made to not make things so complicated that people cannot tackle the learning curve. If folks are not
understanding this architecture, then we need to find a compromise to make things more understand-
able.

• Although the use of System properties to provide config information will by and large not be encour-
egated, it will be possible to pass int system properties where any command line parameter would nor-
mally exist.

- Servlet vs. Generic Container: The Expresso Runtime will emphasize use over a wide variety of systems including AWT and command line programs. Although an equivalent to DefaultInit will exist in Servlet form, there will also be a command line initialization capability as well. If it turns out that there are issues that need to be resolved for the servlet environment to run properly, we will discuss these and modify this design.

- Statics vs Non Statics. The runtime framework will discourage the use of static variables. The use of component containers and locators will take the place of static objects such as singletons.

**Component Containers and Services**

To get a reference to a component, the programmer will need to look up a reference to the component. This will be similar to the Internet’s DNS system or a JNDI server. The concept is nearly identical to the Apache Avalon concept of a Service manager. However, one of the major concepts that the Expresso component system will encourage is the fact that each component container can serve as its own service locator in a hierarchical manner as well as contain subcomponents. Whenever a component container wants to find a service, it queries its parent component for the service it wants to locate, which then first checks itself for the requested service, and then checks the parent for the service. The request will then be delegated up the hierarchy of service locators.

The next concept behind this will turn the current Expresso configuration system on its ear. Each component contains various ‘setup values’. Similar to the concept of the Setup table. Locating setup values will be identical to locating services. A hierarchical search takes place looking for the requested setup value. This is important in that it'll dictate a lot of the program organization: *Any class that requires a setup value must be either itself a component, or explicitly contained by a component container.* Yes, there will be a few global setup values, but by and large these will be discouraged. This will be in fact similar to the fact that EJBs only store their own setup values in a JNDI directory.

**Example**

Ok, so enough of the hand waving. Let's look at a real sample:
Here the box labelled Service Locator would be the analogous to a root level Domain Server. Everything eventually percolates to the root server. All boxes in the diagram are components that are also acting as containers and all circles are components that do not have subcomponents. If Schema1 wants to get an instance of cache, it would call getService("CryptoManager") and retrieve a handle to an instance of the system CryptoManager. However, the call to getService("Cache"), would be the same, only that it returns the Cache that is associated with the Context that Schema2 resides in. Which brings us to another point: Every Context is a component (and a component container) too! It is just a specialized component container that has its own custom services. You will see that any current component in Expresso that has getInstance("ContextName") will be replaced with myContainer.getService(), and the appropriate instance for the current Data context will be returned.

**Component Lifecycle and Interfaces.**
The Expresso Component Architecture

ExpressoComponent Interface

Before we begin, we’ll show a diagram with the various interfaces and classes that participate in the component class creation.

ExpressoComponent

The basic building blocks here are the Containable interface and the Component interface. Containable denotes that an object can contain components. An ExpressoComponent interface denotes the most basic set of methods needed to allow a component to interact with the rest of the system. There are other interfaces available [as can be denoted by all the yellow interfaces in the UML diagram above], but they all add additional optional functionality. The interface for ExpressoComponent is as follows:

We will address the ideas of these components one at a time.

ExpressoComponent

The basic building blocks here are the Containable interface and the Component interface. Containable denotes that an object can contain components. An ExpressoComponent interface denotes the most basic set of methods needed to allow a component to interact with the rest of the system. There are other interfaces available [as can be denoted by all the yellow interfaces in the UML diagram above], but they all add additional optional functionality. The interface for ExpressoComponent is as follows:
public interface ExpressoComponent {

    /**
     * Retrieve the metadata for this component.
     * @return Metadata for the file. This can then be used with
     * <code>getInputStream()</code> to read in and parse the data.
     */
    public ComponentMetadata getMetaData();

    /**
     * Retrieve the parent component [if any]. May return null.
     * @return ExpressoComponent or null
     */
    public Containable getParent();

    /**
     * Called by the parent upon initial Expresso component creation so that the
     * child component can find it's container.
     * @param parent the 'containing' component.
     */
    public void setParent(Containable parent);

    /**
     * Retrieve a property of the component.
     * @param propertyName the name of the property to retrieve
     * @return java.lang.Object The type of the property depends on the type
     * defined in the metadata description file.
     */
    public Object getProperty(String propertyName);

    Method Details

    • getMetaData() This method simply returns the component metadata class. This class is simply a
      JavaBean containing a description of manageable methods, configuration values, etc. If a component
      writer chooses to not implement the interface Describable to provide for XML-capable metadata,
      then it would be up to the component to create the component metadata class. However, this is a
      fairly trivial thing to do. If the component writer implements Describable then the framework will
      construct the ComponentMetadata object for the component automatically.

    • getParent() Every component must nest within a container so it can be located, queried, etc. The ba-
      sic implementation should contain a private Containable variable that is set for the component auto-
      matically when the component is created.

    • setParent() This method would be called automatically by the system factory as it builds and nests
      ExpressoComponents. Other than returning the container with the getParent(), the component basic-
      ally does not have to do anything with this variable as far as framework management goes. The com-
      ponent itself would query its Container to other components and other setup properties, however.

    • getProperty() Each component has its own set of properties, and this allows the component to get its
      own setup values through this method. If the component writer does not implement any lifecycle
      events, then it will be responsible for its own configuration, although it could certainly provide read-
      only variables. However, this method's use will be limited if the component writer does not imple-
      ment lifecycle events.

    Containable
The Interface Containable denotes that this object can contain Expresso components. Each Expresso-Component in fact, will have a Containable object as its parent. The interface definition for this class is:

```java
public interface Containable {
    /**
     * Locates an Expresso Service for use by a client.
     * @param componentName the name of the service to locate.
     * @return ExpressoService.
     * @throws IllegalArgumentException if the service cannot be found.
     * @throws IllegalStateException if the service exists, but is not in a
     * 'runnable' state due to some configuration error or other unforeseen
     * issue.
     */
    public ExpressoComponent locateComponent(String componentName);

    /**
     * Query the container to see if a particular service name is installed
     * in the system.
     * @param componentName the name of the component to query for.
     * @return true if the service is installed and running.
     */
    public boolean isComponentExists(String componentName);

    /**
     * To register the component for control by the Component Manager. This will
     * in essense transfer the control of the service to the Component Manager.
     * This will often be called by the Configuration Bootstrap system.
     * @param newComponent the component to install
     */
    public void addComponent(ExpressoComponent newComponent);

    /**
     * Removes a component from this container.
     * @param componentName The name of the component to remove.
     */
    public void removeComponent(String componentName);

    /**
     * Install a component into the system. If newComponent implements <code>
     * installable</code> then it shall be installed. After that, the component
     * is added.
     * @param newComponent An instance of the component to install.
     * @param log a Logger-like interface to a component that records the process
     * of the installation including any errors, etc.
     */
    public void installComponent(ExpressoComponent newComponent, InstallLog log);

    /**
     * Uninstall the component. If the component implements <code>
     * installable</code> then it shall be uninstalled. After that, it shall
     * be removed.
     * @param componentName the name of the component to uninstall
     * @param log a Logger-like interface to a component that records the process
     * of the installation including any errors, etc.
     */
    public void uninstallComponent(String componentName, InstallLog log);

    /**
     * Retrieves a list of instances of all contained ExpressoComponents. Use
     * this for iterating through the components of a current 'context'. Do not
     */
```
* attempt to modify the map given. Either add or remove a component through
* the addComponent or removeComponent methods.
* @return Read only map of the components.
*/
public Map getChildComponents();
}

As you can see, it provides basic management capabilities for iterating container components, locating
components, etc.

**Describable**

The Describable interface is for allowing a component's metadata to be described by an XML file. Its
definition is fairly straightforward:

```java
public interface Describable {
    /**
     * Get the location of the metadata.
     * @return the URL for the metadata
     */
    public URL getMetadataLocation();

    /**
     * Sets the component metadata for component. Usually the external
     * 'factory class' that parsed the metadata would then set a ComponentMetadata
     * object for this component.
     * @param metadata
     */
    public void setMetaData(ComponentMetadata metadata);
}
```

**Method Details**

- **getMetadataLocation()** All the component has to do is return the resource location as a URL. This
can be either through the Class.loadResource() method or for highly specialized applications, the
component writer could conceivably retrieve the component metadata through a central web server.
Although this method won't be used in Expresso itself, it is available to the component writer if so
desired.

- **setMetaData()** This method will be called by the system factory that is building the component. The
system factory will parse the metadata xml file and then with the ComponentMetadata bean success-
fully constructed, will associate the component with the constructed metadata class.

**Installable**

The interface Installable represents any component that when it is first installed into the system needs to
perform some specific initialization tasks. Examples of this are a Schema object that needs tables and se-
curity set up when it is first installed. The interface definition would look like so:

```java
public interface Installable {
```
/**
 * Called when the Service Manager installs this component
 * @param dataContext the DataContext (aka dbname) to install into
 */
public boolean install(InstallLog log);

/**
 * Called when the Service Manager uninstalls this component
 * @param dataContext the DataContext (aka dbname) to install into
 */
public void uninstall(InstallLog uninstallLog);

Method details

- install() This is the function that installs the actual component. It logs its progress through the InstallLog object. If an error occurs, then it logs the error in InstallLog and returns false, at which point the component container will call uninstall() to remove any components that were partially installed.

- uninstall() This is the function that uninstalls the component, like install(), it will log its progress. The uninstaller() should be aware of the possibility of partially installed components. Therefore, for example, if an expected table is missing, don't quit, just log the fact that the table was missing, and continue uninstallation.

Note about InstallLog. The InstallLog interface is basically identical to log4j in that it has debug(), info(), warn(), and error(). However, InstallLog will probably NOT be a log4j log, rather the actual implementation will capture the output and print it to screen, such as DBCreate currently does. However, it provides a good way of abstracting the actual output to be able to write it to a console, AWT window, or HTML page.

ComponentLifecycle

Most components will want to implement this interface to be able to be configured by the Component-Manager as well as provide initialization and deinitialization services.

A brief note about nesting limits

Although a component could theoretically have unlimited levels of nesting, the current implementation of Apache Digester does not support recursive rules. As such, Expresso dynamically creates rules for Digester to parse nested components up to twenty levels deep. In most cases, this should not affect the designs of most web applications, but it is still a limitation to be noted if you are heavily leaning on nested components for your application design.

How many object instances?

This architecture is kind of a spinoff of the Singleton pattern, first defined by Erich Gamma, et al. The rule of thumb is this: Only one instance of a component will exist with respect to itself and its children components. What does this mean? Looking at the diagram, above, if I call locateComponent("Cache") from Schema1, I'll get the cache component for my data context/ While other data contexts may have other cache components, if I exist inside the context component, I'll only ever see one Cache component ever. So if a component is globally accessible, then there will indeed be only one instance. If the component differs per context (such as DBConnectionPool), then there will be one DBConnectionPool per data context.
ExpressoComponent Lifecycle

Unless the component that is going to be built has absolutely no manageable configuration values, does not need any configuration whatsoever, and is completely thread safe with no static variables, then a component is going to need some sort of lifecycle management. The ComponentLifecycle interface will need to be implemented by the component. The Lifecycles, to be summed up, are:

- **init** Called after the constructor, but before configuration. Allows the component to run any basic construction needs.

- **configure** Called to configure the component.

- **reconfigure** Called to re-configure the component with the servlet engine still running. The component may have to essentially shut down and restart, but it gives us great runtime flexibility to be able to reconfigure without an entire system restart.

- **destroy** Called when the component should release all resources and consider itself dead. This event may or may not occur at program termination, but may be called for a context reload, or many other situations. [Unused Service passivation comes to mind]

The semantics of the tree structure are as follows: For init, configure, and reconfigure, a child component is guaranteed that the parent will have already completed its lifecycle call. For example, if the child depends on a parent setup value, the child is guaranteed that it will be set when it receives a configure event.

The only difference is the destroy() event, which is performed in reverse: Children are destroyed before the parent is destroyed.

Below is the full code for the interface:

```java
public interface ComponentLifecycle {

    /**
     * Initialize the component, this is called before the component receives any configuration information.
     */
    public void initialize();

    /**
     * Configure the service. This is where any parameter settings will take place.
     * @param newConfig a read only dynabean containing all the needed configuration.
     * @throws IllegalArgumentException if the configuration is improper.
     */
    public void configure(Configuration newConfig);

    /**
     * Reconfigures the service during runtime without having to restart the container.
     * @param newConfig a read only dynabean containing all the needed configuration.
     * @throws IllegalArgumentException if the configuration is improper.
     */
    public void reconfigure(Configuration newConfig);

    /**
     * Called upon destruction of the service. This may or may not have anything to do with container shutdown or reloading.
     */
    public void destroy();
}
```
You might ask yourself what the Configuration object is. Well its actually pretty simple, it's an interface to an underlying Dynabean, but all the read methods have been removed. This helps enforce the contractual obligation that a component does not modify its own configuration.

In addition to basic lifecycle events, We'll also provide the 'startable' interface to indicate that the component or service can be started and stopped. stop() will be called before destroy() and start() will always be called after configure. Its full text is listed here

```java
public interface Startable {
    /**
     * Called when the system wants the service to start. This, like any of the
     * other lifecycle events may
     */
    public void start();

    /**
     * Called when the system wants the service to stop. Stop should attempt to
     * release all resources, so that a component can virtually "cold start" when ( *
     * start is called again.
     */
    public void stop();
}
```

An example use of this is a Job Server. If it is stopped, then it will not give job handlers any new jobs to perform. When it is started again, it will begin handing out jobs again. On normal startup and shutdown, start() is called after configure(), and stop() is called before destroy(). However, through an administrative interface, start() and stop() can be called any number of times during a component's lifecycle.

## Startup Configuration

Obviously all this kind of configuration is going to need some sort of boot strapping process. Whatever factory is combining all these components will have to find out about them, and construct the necessary components to make the running structure like the diagram listed above. This is where the expresso-services.xml file comes into play.

The system factory will have to either be fed an expresso-services.xml file location, which for servlet usage, can easily be sent in via the servlet parameters to the DefaultInit servlet. Or if no parameter is fed in, it will try to locate the expresso-services.xml file in the classpath through a loadResource() call. The second method will be best used for standalone applications rather than servlet environments.

The System Factory will then go through the expresso-services.xml file and use it to instantiate all Expresso components and subcomponents. The System factory would also load the component metadata and call first init() and then configure() for each of the components.

### Example expresso-services.xml

Here we list an example expresso-services.xml file. This example will be MUCH larger in real life. For example, every item in the Setup table would be included in each context for each Schema. However, with proper GUI, this won't be a daunting task at all IMO.
<expresso-services>
 <!-- Globally Defineable components. Put here anything that does NOT care about
context it operates in. -->
  <component name="CryptoManager" class-name="com.jcorporate.expresso.core.security.CryptoManager">
    <property name="strongCrypto" value="y"/>
    <property name="cryptoKey" value="[random string of base64 encoded values could go here]"/>
    <property name="ecryptMode" value="AES"/>
  </component>
  <component name="SSLRedirector" class-name="com.jcorporate.expresso.core.controller.">
    <property name="httpPort" value="8080"/>
    <property name="sslPort" value="443"/>
  </component>
  <!-- Globally definable setup values -->
  <property name="dummy" value="whatever"/>
  <component name="default" class-name="com.jcorporate.expresso.core.components.DataContext">
    <property name="active" value="true|false"/>
    <property name="hasSetupTables" value="true"/>
    <property name="description" value="Default Database"/>
    <property name="dbConfig" value="hypersonic"/>
    <property name="useSSL" value="true|false"/>
    <property name="startJobHandler" value="true|false"/>
    <!-- Don't want to cache things? Don't install this component! There will be officially one
    CacheManager instance per Context component -->
    <component name="Cache" class-name="com.jcorporate.expresso.core.cache.CacheManager">
      <!-- Whatever settings we want here -->
    </component>
  </component>
  <!-- The number of components will grow as we manage to componentize other components -->
  <component name="email" class-name="whatever the class-name is">
    <property name="mailDebug" value="true|false"/>
    <property name="smtpServer" value="mail.jcorporate.com"/>
    <property name="smtpPassword" value=""/>
  </component>
  <component name="Persistance" class-name="com.jcorporate.expresso.core.components.DefaultContainer">
    <!-- Notice that driver name, type mappings, connect format, etc are NOT listed here. We'll
    Move these to a 'driver' directory in the Expresso code base, so once and for all, we start
    officially handling the 'non-variable stuff' and save people a lot of non-essential configuration
    options. We find this by the dbConfig -->
    <component name="DBConnectionPool" class-name="com.jcorporate.expresso.core.db.DBConnectionPool">
      <property name="url" value="[URL to database]"/>
      <property name="login" value="[login name]"/>
      <property name="password" value="[password]"/>
      <property name="minConnections" value="3"/>
      <property name="maxConnections" value="24"/>
    </component>
    <!-- New component that takes the place of Schema class to do the type mapping to
    a database -->
    <component name="type-mapper" class-name="com.jcorporate.expresso.core.db.TypeMapper">
      <!-- Type mapper would use the Context level variable dbConfig to know what to load
      -->
    </component>
  </component>
  <!-- All the properties will be located here, BUT all the basic metadata, like what controllers,
what dbobjects, etc will NOT be listed here. -->
  <component name="ExpressoSchema" class-name="com.jcorporate.expresso.core.dbobj.XMLSchema">
    <property name="metaData" value="/com/jcorporate/expresso/core/ExpressoSchema.xml"/>
    <property name="whatever" value="whatever"/>
  </component>
  <!-- And so on for each Schema... each Schema is a component -->
  <component name="" class-name="">
  </component>
  <!-- The following example would show how we could map a certain number of -->
</expresso-services>
If you notice, each component gives itself a service name, and an implementation class that it is associated with. After that, it defines the current settings of its own read/write properties. And finally, if it contains any nested components, it will do so after that.

**Component Creation**

TODO: Flesh this out.

**Define the component's interface.**

If you want your component to be interchangeable, then you will want to define an interface to your component first. For example the system cache should be a Cache interface rather than a static CacheManager class. The client that is using the cache component would then write:

```
Cache systemCache = (Cache)mySchema.locateComponent("Cache");
```

If you keep your component's usage faithful to the interface, then you are well on your way to making sure your component can be interchangeable.

**Implement ExpressoComponent interface**

If you look at the ExpressoComponent interface, you will notice that the algorithms behind each of the methods are nearly identical. For example, locateComponent() is going to first check out its own internal Map (HashMap or otherwise) and then call super.locateComponent(). To alleviate coder fatigue, There will be an ExpressoComponentImpl class that will provide a default implementation of this interface. Classes that cannot derive from ExpressoComponentImpl are encouraged to aggregate ExpressoComponentImpl, and forward all standard component calls to the default implementation. For worst case scenario, the user will need to write about 10 lines of code, one line for each method in the interface to call the equivalent method in ExpressoComponentImpl.

**Write the component's metadata**

The component's metadata will include all setup values necessary to run, a definition of all subcomponents, any schema relations, version information, and descriptions. A minimal description would be as small as:

```
<component-metadata name="sample1" load-on-startup="false" class="com.sample.MySampleComponent">
    <description>My Sample Component</description>
    <version-info>
        <major-version>1</major-version>
    </version-info>
</component-metadata>
```

and that is about it. This can be seriously expanded into the full fledged schema component listed below:
The Expresso Component Architecture

[Future Direction]

Note that the component has all the metadata necessary for an Administrative application to render a form to administer all config values for the component. java.lang.String properties would be text boxes, java.lang.Boolean properties would be checkboxes, etc. Furthermore, properties with property-valid-value sublistings would be rendered as a dropdown box. This allows for extensive component config-
uration validation before the changes are applied.

You'll notice another section called 'method-list'. This would list all methods that could be called be the administrative app. Examples could be: 'purge job queue' or 'remove old validation entries'. They would be rendered as buttons from the admin page.

**Implement Lifecycle interface if you want to receive configuration and lifecycle events.**

The Configuration Interface's rules are simple:

```java
Boolean booleanProperty = (Boolean)newConfig.get("myBooleanProperty");
```

Since the component knows about what it requires type-wise, as per its metadata, then it knows how to case the property that it is getting.

Init is called after the component's constructor, but before any configuration. The component writer will often put any code that would often go in the constructor into the init() method.

Reconfigure's logic is similar but it will tend to be more complex. The component will most likely stop everything that it is doing before reconfiguring itself. However, sophisticated reconfiguration will check which fields have changed (if any) and dynamically adjust itself to the new configuration without stopping. The choice is up to the programmer.

Destroy is simple: destroy all subcomponents first, and then release all the component's own resources.

**System initialization**

**Required System Parameters**

To bootstrap the Expresso component system, the component system will need two parameters:

1. Location of the expresso-services.xml file. If no parameter is configured, then SystemFactory will first look for the system property "expresso.services". If it does not find that then it will attempt to locate in the classpath an expresso-services.xml file. This initialization tactic is identical to that used by Log4j and allows easier use of the component architecture in self-contained jar file where you don't want the user to have to enter a ton of command line options.

2. Directory to use for logging. Like expresso-services, we will look for the expresso.logDir system property if this parameter is not provided. If this system parameter doesn't exist, we will go ahead anyway since it is possible for log4j configuration files to have absolute paths for logging that do not require a logging directory. This should give the developer the maximum flexibility in working with the logging configuration.

3. Logging Configuration file. Like the others, you can use the log4j.configFile parameter to pass the location of the config file. If you pass a filename into the system, you gain the added ability that the ExpressoRuntime initializes log4j using DomConfigurator.ConfigureAndWatch(), so you can adjust log4j.xml config files at runtime without even doing a context reload!

As discussed above, these can be acquired in several ways::

- Command Line: Pass in those two variables via java virtual machine properties
• Command Line: Pass in those two variables via command line args. [A quick note here, it is planned to remove the /webAppDir= parameter because of several incompatibilities with a few application servers that rely on keeping the war file compressed.]

• Servlet Environment: Pass in those two variables servlet initialization properties

• Either Environment: Load the configuration via some properties file located by the classloader

Logging: The odd guy out

Logging is an interesting system because it has to be the VERY first thing initialized, at startup, and it has to be running while everything else is destroyed. So currently, I'm looking at the following methodology:

Initialize logging first... even before Digesting the config file. [Digester needs a working logging system] By retrieving the LogDirectory in the command line rather than in the config file, it allows us to set up appropriate logging this way.

Never deinitialize it! The only time Logging should be de-initialized is when the LogManager detects a classloader change. [ie a servlet context reload]. At that point, it would re-initialize logging. But we will leave the de-initialization to the log4j package itself. This is the current plan after experience so far in dealing with classloading and log4j. It sill may change some as experience proves or disproves the author's theories. Log4j can always be moved outside the context classloader if ClassLoading is a problem.

Enter the System Factory

The entire job of the System Factory is to take the initialization values, and build a Global runtime container for the rest of Expresso. The following snippet shows how to build the global container from within the new init servlet:

```java
/**
 * Initialize the Expresso runtime system.
 * @param sc The ServletConfig. The Servlet config in web.xml needs to
 * have 3 parameters set: loggingDirectory, espressoConfig, and loggingConfig.
 * @throws javax.servlet.ServletException upon instantiation error
 */
public void init(ServletConfig sc) throws javax.servlet.ServletException {
    super.init(sc);
    long startTime = System.currentTimeMillis();
    loggingDirectory = sc.getInitParameter("loggingDirectory");
    espressoServicesConfig = sc.getInitParameter("expressoConfig");
    loggingConfig = sc.getInitParameter("loggingConfig");
    root = SystemFactory.buildExpressoComponentSystem(expressoServicesConfig,
            loggingConfig, loggingDirectory);

    log = Logger.getLogger(RuntimeInitServlet.class);
    long endTime = System.currentTimeMillis();
    log.info("Completed initialization in " + (endTime-startTime)/1000 + " seconds");
}
```

As can be seen, the initialization can easily be ported to a command line environment, or anything else the Expresso developer can conceive.

System Destruction
Any server engine must also be able to shut itself down in a graceful manner. The global container implements the Lifecycle events and therefore, you can call the destroy() event to trigger a graceful shutdown of the Expresso runtime system. Below is the code for the init servlet's destroy method:

```java
/**
 * Destroys the servlet and the Global Expresso runtime container.
 */
public void destroy() {
    if (globalContainer != null) {
        globalContainer.destroy();
    }
}
```

Graceful system destruction is also vital for context reloading since the entire runtime must be destroyed and then reloaded.

## To Do Items

Other sections still must be done

1. Error Handling: What are the defined Exceptions and how are they dealt with?

2. Full Implementation: Currently classes that properly digest the expresso-services.xml and metadata xml are in place. Work is ongoing to finalize the Global Container implementation and System Factory.

3. Backwards Compatibility: Since we need to keep ConfigManager, Setup, SchemaList around for another version, how are we to map these classes to the new version. The current train of thought is to map Setup to Schema-Level properties, ConfigManager to various level properties as known by the default system layout and SchemaList to walk the component hierarchy to locate Schemas. However, this is still in 'think tank' stage and has not yet been cemented.

4. Either rearranging expresso.core OR adding expresso.kernel package to signify this as the 'crux' of the Expresso package with other things being secondary. [For example, persistence would actually be still a service]

5. I want the new component manager GUI to be able to work with no expresso-services file installed at all! I want the GUI to be able to add contexts and components piece by piece. [Of course the default distribution will have some basic components available and preinstalled] This might lead to a pure struts Action class since DBOject configuration and persistance may or may not exist yet. This is similar to the fact that DBCreate currently is running as a Servlet since Expresso Security is not necessarily set up. If any sort of database capability is set up, then we will use it from that point forward.

6. We need a full analysis of what 3rd party jars will be required for the Expresso runtime kernel to function properly. The goal of the analysis would be to reduce the number of jars required for a minimal command-line application. By minimal we also mean no servlet environment, and no configuration manager. Currently we're looking at:

   a. Oswego Concurrency Library: This allows us to have much higher performance component lookups that are thread safe.

   b. Commons Digester: Required for parsing the configuration files

   c. Commons Logging: Required by Digester
d. JAXP compliant parser: Our goal is to be Xerces independent if so desired.

e. Commons Beanutils: Needed for the introspection of the components and invocation of the configuration handlers. For a minimalist app, this MIGHT end up being optional.

f. Log4j: Needed for logging, of course

However, this list has not been fully realized yet since there are still implementation issues to overcome.

7. J2EE App Server Integration: This is the ideal time to also check to make sure that the basic architecture will work with the rest if the current J2EE standards. For example, a version of the Global-Container could be created that runs under JNDI rather than its own internal hashtables. Instances of components could be retrieved from the JNDI server this way. Although there would be a performance penalty, this would be a good method to make sure we can expand into the EJB world. Before Expresso 5.1 is released, there should at least be a ‘game plan’ for how this framework could be run and used from within an EJB server.

8. Remote Objects and Clustering: The final architecture should work well in a clustered and remote object environment. Although the architecture itself is not particularly concerned with clustering aspects, it should be determined that it would be used to access remote and clustered services.

Future Expansion possibilities

This model with metadata has many possibilities in the future. Examples are:

- Adding the full DBOject metadata in the dbobject tag in the schema tag.
- Defining regular expressions that could be used for standardized field and parameter validation. This would often be defined on a Schema-wide basis.
- Full JMX integration. The use of Apache modeler would make dynamic MBeans relatively automatic. All the design team has to do is write a single bridge class to translate the digested XML file into MBean metadata.
- Full database relations: Inside the schema tag, one could define how all the various dbobjects are interrelated. Foreign key relationships could be well defined here.
- Component Event Listeners: We should be able to allow components to listen to events that occur with other components. This way, things like cached setup values can be reread. An example of this would be to have a component get notified whenever the Data Context level setup values get changed.

Conclusion

It is hoped that this article will give you a clear enough picture about where the development team is steering Expresso towards its 5.1 release. Feedback can be directed directly to the author at rimovm@centercomp.com or to the JCorporate opensource mailing list.
Contributors

The following persons have contributed their time to this chapter:

- Mike Rimov
- Mike Traum (JGroup Expert)

Note

Was this EDG documentation helpful? Do you wish to express your appreciation for the time expended over years developing the EDG doc? We now accept and appreciate monetary donations. Your support will keep the EDG doc alive and current. Please click the Donate button and enter ANY amount you think the EDG doc is worth. In appreciation of a $35+ donation, we'll give you a subscription service by emailing you notifications of doc updates; and donations $75+ will also receive an Expresso T-shirt. All online donation forms are SSL secured and payment can be made via Credit Card or your Paypal account. Thank you in advance.
Glossary

ASCII
(American Standard Code for Information Interchange) This standard character encoding scheme is used extensively in data transmission.

ANSI
(American National Standards Institute) This group is the U.S. member organization that belongs to the ISO, the International Organization for Standardization.

BLOB
(Binary Large Object) A term for a binary file stored as part of a database record.

CLOB
(Character Large Object) A term for a large file of characters stored as part of a database record.

DTD
(Document Type Definition) A DTD is the formal definition of the elements, structures, and rules for marking up a given type of SGML document. You can store a DTD at the beginning of a document or externally in a separate file.

HTML
(HyperText Markup Language) This is the format of files published on the web. HTML is an application of SGML; to author in HTML using SGML-based authoring software, you simply need the HTML DTD.

HTTP
(HyperText Transfer Protocol) The method by which our computers talk to one another when communicating over the world wide web. There are many protocols available, but this is preferred for retrieving web pages and interacting with servlet engines.

Job
An Expresso Job is a task that is performed asynchronously. Instead of the web browser seeing the results, the results are emailed to the person that queued the job.

MVC
Acronym for Model-View-Controller. MVC was first widely defined in the book "Design Patterns" by Erich Gamma et al. It defines a separation of concerns in a program where the model defines the internal data structures of the program, the view defines how the model is rendered to the user, and the controller performs the actual actions in the program that affect the model.

Servlet
A Java(TM) language program set up to interact along with a specially designed webserver to provide an interactive web page.

SSL
(Secure Sockets Layer) An encryption mechanism that works along-
side HTTP to provide an HTTP-like capability but with the data being sent back and forth protected from electronic eavesdropping.
Index

B

BETWEEN
SQL Keyword, use of, 52

C

Cache
Tuning, 185
Caching
Of Database Objects, 53
Change Logging
With Database Objects, 58
Client-side
Testing
Load, 174
COMMIT, 59
Controller (in MVC)
Role of Controller object in, 112
Cross Site Scripting Countermeasures
XSS, 148

D

Database Objects
Advantages of, 35
Automatic
AutoDBObject, 75
Capabilities Of, 36
Example of, 40
Generation of Tables for, 40
Operations using, 49
Retrieving multiple instances of, 51
Security for, 55
Using, 36
Using ranges and wild cards with
Searching for records, 52
DBMaint
Using, 97

E

Entity EJB
Relationship of Database Objects to, 36
Expresso XML Project
XML, 7

F

Fields
Description, 41
Multi-valued, 48
Read-Only, 42, 48, 49
Secret, 46
Sequential, 43
Sequential valued, 49
Validation of, 46
Virtual
Calculated, 42
Filtering
String filtering, 148

J

JoinedDataObject
Database Join, 69
JUnit
Unit Testing
Testing, 174

L

Logging
Affect on performance, 184
Lookup
Automatic, 45

M

Mapping
Of Database Types, 42
MediaDBObject
BLOB Fields
Images, 73
Model (in MVC)
Role of Controller object in, 5
MultiDBObject
Database Join, 68
MVC
Role of Controller object in, 5, 112

P

Performance
Affect of Logging on, 184
Problems to watch for, 184

R

Ranges
Using with Database Objects, 52
Records
Adding new, 49
Deleting, 54
Updating, 54
Referential Integrity, 48
ROLLBACK, 59
RowSecuredDBObject
Authorization, 149

S

Schema Object
Creating, 6
Location of, 6
Searching
For records using Wild Cards, 52
Limiting records retrieved, 53
Security
For Database Objects, 55
SecurityDB, 77
SecurityDBObjects, 76
SecurityDBObjs, 77
Session EJB
   Relationship to Controller objects, 80
Sort, 51

T

Thread Safety
   Pitfalls, 81
Transaction Control, 59
Type Mapping, 41, 42

V

Virtual Fields, 42

W

WHERE Clause
   Using custom, 54
Wild Cards
   Using with Database Objects, 52
wizard
   process funnel, 94
workflow
   work flow, 94