Expresso Developer's Guide

by Michael Nash

Expresso Developer's Guide

by Michael Nash

Published October 2001

Table of Contents

Chapter 1. Introduction

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.

Chapter 2. Expresso Overview

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!

Chapter 3. Developing an Expresso Application

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 dependancies 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, easily 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.

Chapter 4. Application Configuration and Initialization

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).

Expresso can then use an application's Schema object to automatically configure that application. The DBCreate servlet reads the list of all known Schema objects and can:

- Create all of the database tables required by database objects in the application and populate those tables with default values.
- Provide default security entries allowing the Admin group access to all of the newly created database object tables.
- Provide default security entries for any Controller objects in the application.
- Provide default security entries for any Job objects in the application.

Application Configuration

An Expresso application is configured by means of a number of XML files, all of which are stored in the WEB-INF/config directory by default.

expresso-config.xml

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 top-level element is the "expresso-config" object, which contains definitions that apply to the entire application. This includes 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 it's 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:

```
<class-handlers>
```

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

Nested withing the "expresso-config" element are one or more "context" elements. Each "context" element defines a different context of operation - typically including a seperate database. Settings with the context element only affect that one context.

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

It is important to check the DTD for this configuration file, and to read the comments in that DTD to understand all of the options available.

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 seperate 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

Each application has an additional configuration file used to set up the Log4j logging system for that application. The logging configuration file specifies which objects will log at which level.

Schema List Table

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.

ConfigManager

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. ConfigManager's first step on initialization is to read the expresso-config.xml file, and to create one or more "contexts" of configuration values. A context is a seperate section in expresso-web.xml, and is typiclaly 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 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 - 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. 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 deetails. 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.

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. 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.

Setup Values

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 accesed via methods in the "Setup" object in the com.jcorporate.expresso.services.dbobj package.

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.

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. 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 it's own seperate logging configuration file, e.g. expressoLogging.xml for Expresso's own internal classes, eForumLogging.xml for eForum, and so forth.

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 seperate tasks (e.g. in their own VM, running them directly as standalone java applications), or jobs queued in that context will not be processed.

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 seperate 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.

Class Handler Configuration

DBTool and DBCreate

DBTool is a standalone Java application which can be run independantly of your servlet container in order to initialie 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:

Chapter 5. Database Objects

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 it's 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

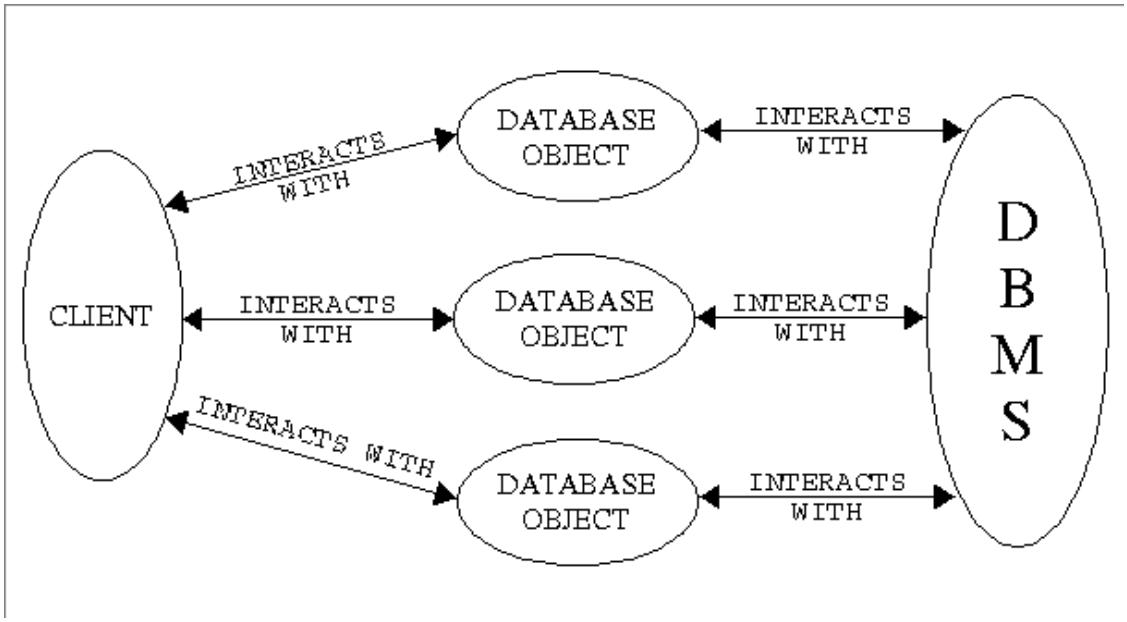
  The connection pool will automatically clean up idle connections after a certain amount of time, reducing the size of the pool to a specified minimum during periods of inactivity.

- 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.

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 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. 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 independance, automatic connection handling, declartive 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 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. 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.

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.dbobj.DBObject 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.

A DBObject must implement at least the constructors for DBObject and a setupFields method. In the setupFields method, a number of method calls are used to establish the relationship of this DBObject 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 DBObject. Lets say that the table in the database should have a unique Customer Identifier, a name and a customer type. You might name the DBObject "Customer" and specify a setupFields method like this:

```
1. public void setupFields() throws DBException {
2.   setTargetTable ("Customer");
3.   addField("CustomerID", "int", 0, false, "Customer Identifier");
4.   addField("CustomerName", "varchar", 80, false, "Customer Name");
5.   addField("CustomerType", "char", 2, false, "Customer Type");
6.   addKey("CustomerID");
7. }
```

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

Line 1 establishes the method signature which must be correct for your DBObject 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 DBObject 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 DBObject, 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 DBObject 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 DBObject but need not be unique across all tables in the database. In other words, you can have only one "CustomerID" column in this DBObject but other DBObjects 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).

- *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 DBObject 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 DBObject. 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 DBObjects assume they are accessing the default database/context as defined when Expresso is deployed. DBObjects can access any available Database/Context though, by using the setDBName(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:

```
custList.setDBName("oracle");
```

says that the custList DBObject 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 "getDBName()" call - Controller objects, for example, do this. So a DBObject being used within a Controller simply says:

```
1. custList.setDBName(getDBName());
```

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.setDBName(getDBName());
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, you can use the Expresso DBObject NEXTNUM to assist with this. For example:

```
1.   public void add() throws DBException {
2.     NextNumber myNext = new NextNumber();
3.     myNext.setDBName(getDBName());
4.
5.     long myJobNum = myNext.getNextTentative("JOB");
6.     setField("JobNumber", new String().valueOf(myJobNum));
7.
8.     super.add();
9.     myNext.getNextConfirm("JOB");
10.  } /* add() */
```

Multi-Valued Fields

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("Account Active")) {
4.       Vector myValues = newVector();
5.       myValues.addElement(newValidValue("Y", "Yes");
6.       myValues.addElement(newValidValue("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");
```

This code assumes that the Customer Type object specified exists and implements the getValues() method, which returns the vector of valid values.

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.

Secret Fields

A field in a DBObject 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:

```
1. public void setField(String fieldName, String fieldValue) throws DBException {
2.   if (fieldName.equals("Priority")) {
3.     if(!(fieldValue.equals("A") || fieldValue.equals("B") || fieldValue.equals("C")))
4.       throw new DBException ("Priority must be A, B or C");
5.     }
```

```
6.     super.setField(fieldName, fieldValue);
7.   }
8. }
```

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 javadoc for the "RE" object in the package for details on forming regular expressions.

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 object. 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.

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.

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

Special facilities have been provided for database object to generate their own unique serial numbers. For databases that provide sequential/serial number capability, this facility can be used, and for databases that do not provide this functionality a simple "next number" facility is also provided.

Default Values

Default values can be specified that populates the tables created for each database object. The "populateDefaultValues" method of each DBObject is run when DBCreate is executed, and the default values for the DBObject 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 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.

Sequential Numbers

Special facilities have been provided for database object to generate their own unique serial numbers. For databases that provide sequential/serial number capability, this facility can be used, and for

databases that do not provide this functionality a simple "next number" facility is also provided.

Using Database Objects

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

Creating an Instance

Setting a Database Context

Virtually all of the standard programs that come with Expresso will utilize the currently logged-in context for their database objects. Whenever a user is logged in, they establish, either explicitly or by default, 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 DBObjects that it uses for a particular user to use the currently logged-in context for that user. So, if the user is logged in to the "demo" database/context, all DBObjects that DBMaint utilizes will have setDBName("demo") called before they are used.

This provides the ability to have entirely seperate 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-config.xml file - each "context" element can define a JDBC section to specify database connection information 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", getDefaultValue("fieldName"));

Retrieving Records

Database objects can be used in an "aggregate" mode where a single DBObject 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 wild card 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 DBObject corresponding to a particular row in the database, you must

first initialize the object in your program:

```
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).

```
    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:

```
    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);

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

Will print Customer 1'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 DBObjects 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:

```
    /* 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 DBObject 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, lets say we want to perform some processing on all customers of type "AB" in our database. We can use code like this:

```
1.      Customer custList = newCustomer();
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:

```
1.    Iterator e = custList.searchAndRetrieveList();
2.    while (e.hasMoreElements()) {
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 anefficient 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:

```
custList.searchAndRetrieve("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.searchAndRetrieve("CustomerName Desc");
```

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

```
custList.searchAndRetrieve("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".

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 application.

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:

```
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:

```
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":

```
1.  Customer cust.List = new Customer();
2.  Customer oneCust = null;
3.  custList.setField("Processed", "N");
4.  custList.setMaxRecords(100);
5.  boolean moreRecords = true;
6.  while (moreRecords) {
7.      for (Enumeration e = custList.searchAndRetrieve().elements();
8.          e.hasMoreElements();) {
9.          oneCust = (Customer) e.nextElement();
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 effecient 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 DBObject 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 DBObject, called SecuredDBObject, is available.

By extending SecuredDBObject rather than just DBObject, your DBObject 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.

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

```
<?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.D
  </class-handlers>

  <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=""
        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>

<context name="other" notExpresso="y">
    <description>Other DB (Non-Expresso)</description>
    <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>
    <images>%context%/%expresso-dir%/images</images>
    <startJobHandler>n</startJobHandler>
    <showStackTrace>y</showStackTrace>
</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 setDBName to set its context elsewhere. In this way, the DBMaint controller can be used, even with "mapped" objects.

Security for the mpped 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 indicate this through use of the "notExpresso" flag, 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 table in the default context:

```
com.mypackagename.Employee | hr | Employee Table
com.mypackagename.Certification | hr | Certification Table
```

As you can see, the DBOTHERMAP table is used by Expresso to map DBObject class names to particular 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 Expresso 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 context or the currently logged-in context.

Important Note: DBObjects mapped to otherdb locations will work in all cases where the object is allowed 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 connection, you have the responsibility of making sure that the connection was made against the correct database. One way to do this is to initialize the db object normally (e.g. without the explicit connection) and use the "getDBName()" 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().

```
1.  DBConnectionPool myPool = null;
2.  DBConnection myConnection = null;
3.  try {
4.      myPool = DBConnectionPool.getInstance(getDBName());
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.committ();
14. } catch (DBException de) {
15.     if (myConnection != null) myConnection.rollback();
16.         throw newDBConnection(de);
17. } finally {
18.     if (myPool != null) {
```

```
19.         myPool.release(myConnection);
20.     }
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 committ updates, but should instead wait for committ() 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 committ() 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.

MultiDBObjects

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 MultiDBObjects - 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.setDBName(getDBName());
3.  myMulti.addDBObj(
        "com.jcorporate.expresso.services.dbobj.UserDBObj");
4.  myMulti.addDBObj(
        "com.jcorporate.expresso.services.dbobj.UserGroup",
            "group");
5.  myMulti.addDBObj(
        "com.jcorporate.expresso.services.dbobj.GroupMembers",
            "members");
6.  setForiegnKey("members", "UserName", "User", "UserName");
7.  setForiegnKey("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 "getDBName()" 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 "members" object's Username field must match the "user"'s object Username field, and the "members" object's GroupName field must match the "GroupName" 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 implementing 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 added shortly.

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?dbobj=com.jcorporate.expresso.core.dbobj.AutoDBObject&table=SCHEMALIST&cmd=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.

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.

Expresso must have at least one database connection defined to operate. This primary database is known as the default database context. 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 DBObject is manipulated, unless Expresso is told to "look elsewhere" in one of two ways:

Chapter 6. Using Controller Objects

MVC Architecture

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.

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.

Controller 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:

  Controller 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 Controller 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 completion of one controller permits 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 Controller without the need to program or design with JSP's a custom GUI, allowing a basic controller to be deployed very quickly, then enhanced later with a custom UI.

- XML User Interface

  An 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

  Controller 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 invokations, 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.

Controller Basics

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():

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

```
10.
11.     // The second logical block
12.     Block b2 = new Block("Links");
13.     Transition a1 = new Transition("Take action 1",
            "/servlets/ControllerServlet/?trx=.......");
14.     a1.setName("Transition");
15.     b2.add(a1);
16.     Transition a2 = new Transition("Take action 2",
            "/servlets/ControllerServlet/?trx=.......");
17.     a2.setName("Transition");
18.     b2.add(a2);
19.     Output o4 = new Output("Msg4",
            "Please choose one of the links above");
20.     b2.add(o4);
21.     addBlock(b2);
22.
23.     // The third logical block
24.     Block b3 = new Block("Form:Form1");
25.     Output o5 = new Output("Msg5",
            "Please fill in the information below");
26.     b3.add(o5);
27.     Input i1 = new Input("Name");
28.     i1.setLabel("Name");
29.     i1.setType("text");
30.     b3.add(i1);
31.     Transition a3 = new Transition("Submit",
            "/servlets/ControllerServlet?controller=......");
32.     a3.setName("Submit");
33.     b3.add(a3);
34.     addBlock(b3);
35.     ..............
```

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:

```
1.     ............
2.     String blockName = ".....";
           // Name as in the controller
3.     for (Enumeration e = controller.getBlock(blockName).getContents().elements();
4.                           e.hasMoreElements(); ) {
5.          ControllerElement oneElement = (
               ControllerElement)e.nextElement();
6.          if (oneElement instanceof Transition) {
7.            handleTransition((Transition)oneElement);
8.          } else if (oneElement instanceof Input) {
               handleInput((Input)oneElement);
9.          } else if (oneElement instanceof Output) {
10.           handleOutput((Output)oneElement);
11.         } else if (oneElement instanceof Block) {
12.           handleBlock((Block)oneElement);
13.         } else {
14.           throw new Exception("Cannot handle object of type:"
15.                               + oneElement.getClass().getName());
16.         }
17.    }
18.    ............
```

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:

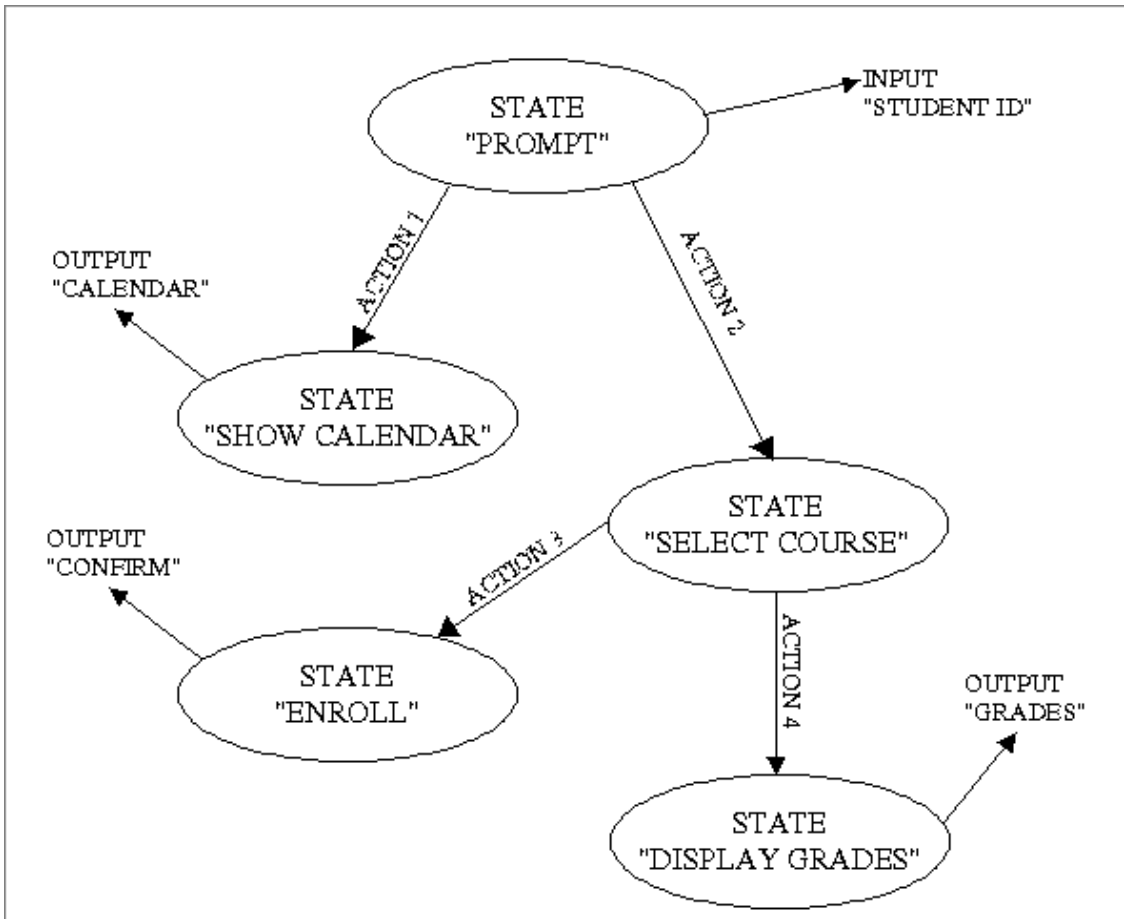
- Controller

  Controller is the base class for all Controller object, and can be directly extended for Controller that either handle their own connections or that do not connect to a database.

- DBController

  DBController extends the base Controller object and provides easy access to a database connection from the connection pool.

Controller Activity Explained



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 independant 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 private to the Controller class itself. Seperate State objects are often superior, as they allow a clean seperation 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.

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, like this:

```
1. /**
2.  * Our constructor declares the states this
    * controller supports
3.  */
4. public DBSecurityMatrix() {
5. State prompt = new State("prompt",
          "Prompt for Schema and User Group");
6. addState(prompt);
7. setInitialState("prompt");
8.
9. State dbobjmatrix = new State("dbobjmatrix",
10.         "Enter/View Database Object permissions");
11. dbobjmatrix.addParameter("SchemaClass");
12. dbobjmatrix.addParameter("GroupName");
13. addState(dbobjmatrix);
14. } /* DBSecurityMatrix() */
```

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

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:

```
1.  public Upload() {
2.          PromptBrowser browser = new PromptBrowser("browser",
```

```
          "Prompt for Upload from Browser");
3.  browser.addParameter("resource", false);
4.          addState(browser);
5.
6.  DoBrowser dobrowser = new DoBrowser("dobrowser",
        "Process Upload from Browser");
7.  dobrowser.addParameter("action");
9.          addState(dobrowser);
10.
11. } /* Upload() */
```

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:

```
(package and import statements ommitted)
1. public class PromptBrowser extends State  {
2.
3. public PromptBrowser(String stateName, String descrip) {
4.          super(stateName, descrip);
5. }
6.
7.   public void run() throws ControllerException {
8.        Controller myController = getController();
9.        String currentNumber = StringUtil.notNull(
                myController.getParameter("number"));
.... here the State will do whatever logic it needs to
10.    } /* run() */
11. } /* PromptBrowser */
```

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 DBObjects, 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 programmatically - 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 usualy 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.

Chapter 7. Database Maintenance

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 maintenace 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:

"/servlet/DBMaint?next=hr.html&dbobj=com.javacorporate.ext.hr.dbobj.Employee&cmd=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.

DBMaint: A step-by-step example

Here we present an example of building a simple application to allow maintenance of a customer, complete with security. This example follows the process we outline above, but with the actual code included for reference, and with more detail.

We'll assume that our customer maintenance form is to be a beginning for a more sophisticated sales tracking application.

1. *Define the database object*

   Let's assume for the sake of this example that you've installed Expresso in /usr/java/lib/com/javacorporate. 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).

   1. First we create the directory /usr/java/lib/com/javacorporate/ext, where we are going to define "extensions" to the javacorporate Expresso framework. The package name could just as easily be outside the javacorporate directory if so desired.
   2. Within the "ext" directory, we create a directory for our package, called "sales" (e.g. we create /usr/java/lib/com/javacorporate/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.

   3. 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.

```
1.  /*
2.   * Customer.java
3.   */

4.  package com.javacorporate.ext.sales.dbobj;

5.  import com.jcorporate.expresso.core.dbobj.*;
6.  import com.jcorporate.expresso.core.db.DBConnection;
7.  import com.jcorporate.expresso.core.db.DBException;

8.  /**
9.   * A Customer object stores information about customers for our demo
10.  * sales application.
11.  */
12. public class Customer extends SecuredDBObject {

13.   private String thisClass = new String(this.getClass().getName() + ".");


14.   /**
15.     * There are three possible constructors for SecuredDBObjects
16.     * Usually there is no need to extend them for objects we implement
17.     */
18.    public User() throws DBException {
19.      super();
20.    } /* User() */


21.    /**
22.     * This constructor is used when a connection to the database
```

```
23.     * is supplied by the calling object
24.     */
25.     public User(DBConnection theConnection) throws DBException {
26.       super(theConnection);
27.     } /* User(DBConnection) */


28.     /**
29.      * This constructor is called by the DBMaintservlet
30.      * and supplies the name of the user trying to connect to the
31.      * database object.
32.      */
33.      public User(DBConnection theConnection, String theUser) throws DBException
34.        super(theConnection, theUser);
35.      } /* User(DBConnection, String) */


36.     /**
37.      * The setupFields method does the real work of establishing the
38.      * definition of the DB object.
39.      */
40.      public void setupFields() throws DBException {

41.        /* Establish the primary database table for this object. */
42.        /* Note that there may be more than one, but this is the default */
43.        /* table */
44.        setTargetTable("CUSTOMER");

45.        /* Set a description for this object. This is a human-readable */
46.        /* string that appears at the top of forms when used with DBMaint */
47.        setDescription("Users");
```

```
48.         /* Define each of the fields in the table */
49.         /* Note that the field types used here are "internal", and can be */
50.         /* mapped to other types for the underlying relational database */
51.         addField("CustomerNumber","int", 0, false, "Customer ID Number");
52.         addField("CustomerName","varchar", 80, false, "Customer Name");
53.         addField("Phone", "varchar", 15, true, "Phone Number");
54.         addField("EMail", "varchar", 80, false, "Customer EMail");

55.         /* ... you could of course add many more fields for a production */
56.         /* object */

57.         /* addKey is called for each field in the primary key */
58.         addKey("CustomerNumber");

59.      } /* setupFields() */



60.      /**
61.       * A utility method used when generating lists of objects
62.       */
63.      public DBObject getThisDBObj() throws DBException {
64.        return (DBObject) new Customer(this.getConnection());
65.      } /* getThisDBObj() */

66. } /* Customer */
```

4. Now compile the Customer.java object and correct any errors.

2. *Create a Schema object for your new application*

   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:

```
1.  /**
2.   * Sales.java
3.   */

4.  package com.javacorporate.sales;

5.  /**
6.   * Schema object for the Sales demo application
7.   */
8.  public class Sales extends Schema {

9.   /**
10.   * Default Constructor
11.   */
12.   public Sales() throws DBException {
13.    super();

14.    /* add is called for each object to be a "member" of this schema */
15.    addDBObject("com.jcorporate.sales.Customer");

16.  } /* constructor */

17.  /* Other methods are optional in the Schema object - see the Javadoc */
18.  /* for details */

19. } /* Sales */
```

This simple Schema would be extended as you add new objects to your new application.

- Compile the Sales object and correct any errors

3. *Create the Links in HTML*

Now you create a link to your new database object using the DBMaint servlet:

- Create the web page you are going to call the maintenance function from: Let's assume it's called "sales.html".

  The overall layout of the page is unimportant to the functioning of the servlets themselves. We'll concentrate on the links that call our new object.

- On this page, create the following links:

  - Add Customers

```
1.  <a href="/servlet/DBMaint?next=sales.html
  &dbobj=com.javacorporate.ext.sales.dbobj.Customer
  &cmd=Add">Add Customer</a>
```

  - List Customers

```
1. <a href="/servlet/DBMaint?next=sales.html
&dbobj=com.javacorporate.ext.sales.dbobj.Customer
&cmd=List">List Customer</a>
```

  - Search for Customers

```
1. <a href="/servlet/DBMaint?next=sales.html
  &dbobj=com.javacorporate.ext.sales.dbobj.Customer&cmd=Search
  ">Search for Customers</a>
```

4. *Register your new Schema*

   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:

   - In your web browser, go to the Expresso pages that you downloaded from the Javacorporate site.
   - Click the "Setup" link in the left-hand column.
   - On the line entitled "Application Schema Objects" click the "plus" sign to add a new entry.
   - For the "Schema Class File" field, enter "com.javacorporate.ext.dbobj.Sales", the name of the object for the Schema we just created.
   - For the "Schema Description" field, enter "Sales Application".

5. *Run "Initialize"*

   The initialize function will create the database table for your new object, and also insert default security entries so that the "Admin" group (and the Admin user) will have full access to the newly created object.

   - Click the "Setup" link in the left-hand column of the Expresso pages.
   - Choose the "Initialize" servlet. Leave all boxes checked and click the "Run" button.
   - In a moment you will see output confirming that the tables have been initialized and that security entries have been created.

6. *Log In as Admin*

   Go to the Expresso Log In/Out page and use Login to log yourself in as Admin. As your new 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.

Now test your new database maintenance function - go to your sales.html page and try the Add, List and Search functions.

This is of course a very simple example, but one that can be the basis for a more complete application.

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:

```
1.  /DBMaint.do?dbobj=com.yourcompany.dbobj.Customer&state=List
```

will list all records (subject to a page size specified in Expresso and discussed later) in the table corresponding to the Customer DBObject. (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.

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.

Chapter 8. Developing JSP's

JSP development using Expresso leans heavily on the MVC architecture, where the components are as follows:

Expresso Tags

Extended Struts Tags

Chapter 9. Expresso Utilities

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.

Events

Expresso also defines a mechanism for defining "events" that occur within the web application. 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.

Utilities

A number of other facilities don't fall into any of the above categories, but might otherwise be useful. Some of these are:

- 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.

Chapter 10. Expresso Security

The application's security can be administered in an easy-to-use Matrix style. (using the ControllerSecurityMatrix, DBObjSecurityMatrix and JobSecurityMatrix Controller object)

Chapter 11. Expresso Component Application Deployment

Chapter 12. Internationalization

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:

```
<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 ommitted.

Chapter 13. Expresso and XML

Expresso 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 stylesheet, to produce HTML, WML, PDF, or other XML, as required. This ability is provided by the XMLViewHandler servlet.

Any Controller can be invoked vi it's URL with the "view=xml" option. "view=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 is described above. 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 XMLViewHandler object (in the com.jcorporate.expresso.core.servlet.viewhandler package) for an example of how to access Xalan programmatically.

Chapter 14. Monitoring and Verifying the Operation of Expresso Applications

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 it's performance is acceptable and does not degrade over time.

Tools exist within Expresso to verify it's own functioning and the function of any other web application. These tools consist of two types: Unit tests and Performance Tests.

Unit Tests

Expresso contains the JUnit framework - 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 below) in that they deal with individual components, making it easier to locate and correct a problem before components are assembled into a finished application.

The Schema object for each application contains a list of all of the unit tests registered for that application, and a Controller object is available to run these unit tests: com.jcorporate.expresso.ext.controller.TestController. A link to run this controller can be found on the "Applications" page in Expresso, although it can of course also be run from the command line.

Performance Tests

Conditions affecting the performance of an application can change over time. A server can get busier, a network can become more heavily loaded, a memory leak can manifest itself and so on. It is important that the performance of your application does not start off acceptable, and then as conditions change become no longer acceptable. Expresso has facilities to help you do this.

On the "Operation" page of Expresso is a link for maintenance of "Performance Tests". These entries allow you to specify a URL to be tested and to specify expected performance criteria for that URL. The URL does not actually have to be a part of an Expresso application - it can be any URL that you wish to verify is operational.

When the URL is first specified, you may not know what a reasonable performance is for that URL. The Performance Test table collects statistics as the tests are run that allow you see what the performance has been over a series of tests, so by running repeated tests at different times and viewing these statistics you can set reasonable values for the expected performance fields.

In addition to verifying that a specified URL runs and completes in a certain time, the Performance Tests table also allows you to check if the URL is returning an expected result. The "Expect String" field is used to specify a string of characters that is supposed to be a part of the output from the URL when it runs correctly. For a simple static web page, you can specify a few words of the text from the page. For a dynamic URL (such as a JSP or Servlet) you can give a term or two from the output that is expected. Be sure that the term specified would *not* occur if the page fails (e.g. avoid using HTML tags or something that might be part of an error message). In this way, the performance test is able to verify that this expected output is received, and to report a problem if it is not.

As each performance test is executed, it is timed: timing begins when the URL i s requested and ends when the input stream ends. These times are compared to st ored times to determine if the response was recieved in the expected time - the "normal" time for this test. In addition to the "normal" time, a "warning" tim e and a "max" time can be specified. Taking longer than the "max" time is consi dered an error, and the test fails. Taking longer than the normal time is repor ted as a caution, and longer than the warning time as a warning, but the test c an still succeed.

Using Performance Tests

Performance Tests Sets

Performance Tests can be organized into "sets". These sets allow a sequence of tests to be executed one after another - ideal for situations where one result depends on a previous result - e.g. where sessions are used.

HealthCheck

HealthCheck provides a Controller to run one more test sets that are specifical ly indicated as being part of the periodic health check of the system. The test set has a flag that incidates if the set is to be used by HealthCheck.

Once you have defined a series of Performance Tests, the HealthCheck utility is used to run them periodically. This can most easily be done by setting up a shell script or batch file that is executed by your operating system's task scheduler (cron, System Scheduler ,etc) on some specified schedule. For example, the script for a Linux system can look like this:

```
1.  #!/bin/sh
2.  java com.jcorporate.expresso.core.utility.ControllerRun
    configDir=/home/expresso/expresso/expresso-web/WEB-INF/config
    webAppDir=/home/expresso/expresso/expresso-web db=site
    controller=com.jcorporate.expresso.ext.controller.HealthCheck
    state=health
```

The line beginning with "java" is all one line in the actual shell script, broken across multiple lines here for clarity.

This script can then be executed, for example, every hour via the "cron" utility by specifying a crontab entry like this:

```
1.  SHELL=/bin/sh
2.  MAILTO=root
3.  0 * * * * healthcheck.sh
```

When HealthCheck runs, it reads the list of Performance Sets that are flagged for inclusion in the HealthCheck and executes them, collecting performance statistics as it goes. If any URL does not return the specified "Expect" string, or takes too long to run, it builds an email to be used to send an "Event" notification. The event code "HEALTH" is used to determine who should receive the notification, which looks like this:

```
From: support@javacorporate.com

Sent: Tuesday, March 27, 2001 12:01 PM

To: mnash@javacorporate.com

Subject: ERROR:System Health Check



HealthCheck


HealthCheck at 2001-03-27 08:00:11


For database/context 'default'



There were no failures, no warnings, and 1 caution,


CAUTION: Test 6 (Center for Expresso) for URL

'http://www.jcorporate.com/components/

  internal/Center.jsp?category=65'

responded slower than it's set normal time of 160 milliseconds.

It ran in 237 milliseconds.


>From Server:www.jcorporate.com,

  Database/context:default (Default Database)
```

The above example shows the message received when a particular URL went over it's Normal Time, but not over it's Warning time, and still executed successfully (e.g. returned the string specified).

Part of the usefulness of the HealthCheck process is that it does generate an email even if there were no warnings or cautions - this email is a "success" Event, so you can easily opt to only see a message when there *is* a problem, but the success emails arriving on schedule are a good indicator that all is well with the site. The one error condition that HealthCheck of course cannot catch is when the entire server

is down - preventing HealthCheck from running - or when the email server is unable to deliver it's warnings. If you know, however, that you should be getting an email once an hour, or whatever the appropriate schedule is, then you can take action if you notice that it has not arrived.

HealthCheck generates a system event 'HEALTH', which is emailed to all subscrib ed users automatically. The event's "success" code is true if the health check encountered no cautions, warnings, or errors, and false otherwise.

RunTests

The RunTests controller is used to execute one or more test sets. It has the ab ility to run a sequence of test sets, and to run that sequence multiple times, and optionally to run multiple threads of tests at once. This allows load on th e system to be simulated, catching performance errors and problems that only oc curr under load.

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 "ConversionPattern" 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,

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 expressoLogging.xml file, and monitor the results. Each SQL query will have it's 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 the operations faster.

Indexing

See the section on DBobjects for information about setting up Indexes for faster access on non-primary-key fields.

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.
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 . HTML is an application of SGML; to author in HTML using SGML-based authoring software, you simply need the HTML DTD.

Index