Neutrino Server Developer Guide

SUBATOMIC
SYSTEMS INCORPORATED

180 Pacific Avenue
San Francisco, CA 94111
info@subatomicsystems.com

Getting Started

The Neutrino Server is a Java Server API compliant application which is very similar in scope and behaviour to the popular Rails framework. Server actions are accessed via REST URLs and by default return JSON.

Facilities

Reflection-based Controller & Action Dispatch

Neutrino Server uses reflection to determine whether a controller is available to handle a request and whether that controller has a matching action defined.

Database Abstraction

Neutrino Server offers a very simple database abstraction layer to make models very easy to implement.

Scheduled Tasks

Neutrino Server offers a database-driven way of performing deferred or recurring tasks.

Email Queue

Neutrino Server offers a multi-threaded outgoing email queue which is configured via a database table.

Project Layout

project/server/

Home directory for the server environment. Typical things to find here are the build.xml file which controls the build process, the war file for the binary form of the application, and the following directories.

project/server/app

Repository for application specific server-side source code.

project/server/app/config

This directory usually contains one source file, Routes.java, which describes routes which override the default route determination algorithm.

project/server/app/controllers

Source files for application controllers, subclasses of NeutrinoController.

project/server/app/db

SQL files required to construct the database schema, and populate it with any initial fixtures.

project/server/app/models

Source files for application models.

project/server/tomcat

Tomcat environment managing the server.

project/server/web

Root of the actual application itself. Index.html is located here, as are static resources (such as neutrino.js, etc).

project/server/web/WEB-INF

Root of the dynamic web application. Server configuration lives here, as do libraries on which the application depends, and class files for the application code itself.

Build System

Implementing Controllers

Application specific controllers are implemented by implementing a Java class. The source file for this class should live in server/app/controllers, should be in the "controllers" package, and should extend com.subatomicsystems.neutrino.server.NeutrinoController. The class must be named appropriately for the URL pattern it is to handle - for example, a controller implementing the pattern "/movies/*" should be called MoviesController. It is not necessary to implement a constructor in this class, but if one is necessary, it should take no parameters - otherwise the ControllerManager will be unable to instantiate it. Other public methods in this class which take no parameters and return Object are presumed to be actions and available via REST URLs, so care should be taken with architecting Controllers.

Within Controller methods, the following protected instance members are available to aid request processing --

connection

A java.sql.Connection which represents the Controller's connection to the database. Neutrino Database and Record facilities require an instance of Controller to operate. This instance is checked out of the Controller pool prior to Controller invocation and checked in thereafter. A transaction is started prior to invocation and is committed if the Controller proceeds without throwing an exception, and is rolled back otherwise.

request

A javax.servlet.http.HttpServletRequest representing the request which is responsible for the invocation of the Controller.

response

A javax.servlet.http.HttpServletResponse representing the response for which the Controller is responsible for formulating.

session

A javax.servlet.http.HttpSession representing the session associated with the request, if any.

uploads

A List<Map<String, Object>> representing HTTP file uploads accompanying the request, if any. The Map contains the following fields --

name (String) - name of the uploaded file
field_name (String) - HTTP form field name
content_type (String) - content type of the uploaded file
file (File) - temporary file holding the contents of the uploaded file

parameters

A Map<String, String>> representing the HTTP parameters accompanying the request, if any.

action

String containing the name of the Action which resulted in the method being invoked.

controllerKey

String containing the key to the Controller which resulted in the method being invoked. For example, for the MoviesController, the controller key would be "movies".

renderResponse

boolean determining whether Neutrino converts the return value from the Action into JSON and sends it back to the client. If this is set to false via the *setRenderResponse()* call or directly, then the Action is presumed to be sending content directly.

server

String containing the scheme, hostname, and port number of the server, in URL format, to which the request was sent.

The following example is a very simple controller --

```
package  controllers;

import com.subatomicsystems.neutrino.server.NeutrinoController;

public class MoviesController
extends NeutrinoController
{
	public Object
	get ()
	throws Exception
	{
		return Movie.getByID (this.connection, this.parameters.get ("id")
);
	}
}
```

Caveat to adding controllers

One caveat with adding controllers is that in order to get the NeutrinoDispatcher to respond to top-level URL elements, it has to be "mounted" on that URL stem in the Tomcat configuration. Otherwise, it's not possible to pass everything else off to the regular Tomcat file request dispatcher.

The following example associates the NeutrinoDispatcher with the /movies stem so that requests for all matching URLs are forwarded to MoviesController via the Neutrino dispatcher.

```
<!-- this section is usually already here, included for clarity →
<servlet>
      <servlet-name>NeutrinoDispatcher</servlet-name>
      <servlet-
class>com.subatomicsystems.neutrino.server.NeutrinoDispatcher</servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>NeutrinoDispatcher</servlet-name>
  <url-pattern>/movies/*</url-pattern>
</servlet-mapping>
```

Using Standard Controllers

The base Neutrino distribution arrives with one controller, the ConfigController, that is free to use with applications. For security reasons, this and any future included controllers are *not* in the "controllers" package which is exposed to the request dispatcher. Therefore in order to use it, developers must either subclass it, or embed its actions in another controller.

Subclassing standard controllers

This approach works well if the intent is just to expose a standard controller to the dispatcher, which is mostly the case.

```
// ConfigController.java

// the "controllers" class is exposed to the request dispatcher
package controllers;

// this exposes all of the standard controller's actions
// which may or may not be appropriate or desired
public class ConfigController
extends com.subatomicsystems.neutrino.server.ConfigController
{
}
```

Embedding actions

Standard controllers are actually just shells which forward requests on to corresponding "action" classes, which actually implement the functionality. This permits controllers to easily perform compound operations involving multiple controllers, and also avoid implicitly making their superclass's actions visible.

```
// the standard ConfigController
package com.subatomicsystems.neutrino.server;
```

```
public class ConfigController
extends NeutrinoController
{
	public
	ConfigController ()
	{
		this.configActions = new ConfigActions (this);
	}

	public Object
	get ()
	throws Exception
	{
		Object config = this.configActions.get ();

		// could now use the return in a call off another action
		// or do further processing on it here, etc

		return config;
	}

	// remainder of actions here...

	private ConfigActions
	configActions = null;
}
```

Implementing Models

Neutrino Server's database abstraction facility would prefer you to implement a database-backed model using a Plain Old Java Class. The source file for this class should live in server/app/models, should be in the "models" package, and should extend com.subatomicsystems.neutrino.server.Record.

The Record superclass, in conjunction with the global Database instance, make the issue of dealing with JDBC easier while adding little to no overhead. There is no obligation to use Record while implementing an application running on Neutrino Server, but Database does a couple of things which are useful to any JDBC application. First, a Connection instance is checked out from a pool prior to action invocation, made available to the action, and checked back in thereafter. Second, a transaction is maintained per action invocation, which is committed if the action completes without throwing an exception, and rolled back otherwise.

Record inherits from HashMap<String, Object>, and column access is achieved via the get() method, obviating the need for implementors to define getter and setter methods. Of course, if a particular accessor is used frequently, then it may make sense to actually implement it discretely. The getID() provided in the Record base class typifies this.

Record defines two abstract methods which must be implemented by a model class in order to satisfy the interface:

getColumnNames()
As Java classes modelling database records are freeform maps, Record cannot determine the record structure without asking the subclass. In order to avoid pollution and errors resulting from assuming everything in the map is mapped to a table column, the subclass is asked to provide names for the table's columns. This list takes the form of Map<String, Object> so as to provide a future facility for attaching (for example) framework-managed validators and the like to the column entries.

getTableName()
Record cannot determine the name of the database table underlying the model, therefore the subclass has to provide it. Note that currently a Neutrino Server model can only map to one database table.

With these two methods in place, Record and Database together provide CRUD functionality without any further code. Record provides getByID(), getAll(), and deleteByID() to get started retrieving records, but conventionally, subclasses then provide more appropriate model-specific facilities.

Database helps with retrieval, too. As records are just maps, Database can populate any record without knowing anything about the role of the model or indeed what type its subclass is. getOne() and getMany() take PreparedStatements and a RecordFactory, and then return populated Record instances made by the factory (so that the type is maintained).

Other methods in Record can be overridden to provide more functionality. Prior to a Record being inserted, it calls validateForInsert() to ensure that the new record looks OK. Similar with validateForUpdate(). Overriding these methods allows highly granular control over validation.

Similarly, adorn() is called after a record is retrieved from the database. If the subclass needs to change anything in the instance before it goes back to the client -- like for example removing sensitive fields or adding convenience representations -- then it can override adorn() and be called to do so at a convenient time.

Record Factories

Neutrino Server tries hard to make its standard Controller and any future provided Controllers easily integratable into custom applications. However, standard Controllers make use of standard Models, and it should be possible for custom applications to extend the standard Models with properties of their own. This is where Record Factories come in.

Models are not directly instantiated by the code manipulating them. Instead, a reference to a RecordFactory is passed to each Record or Database API call, and the factory is then asked to instantiate a Record of the appropriate type. Conventionally, the Controller in charge of handling the current request is also the RecordFactory responsible for creating Records pertaining to that request. Indeed, NeutrinoController implements RecordFactory and knows how to create Records for all the standard types.

For custom Records and extended standard Records, it is recommended to implement an application-specific NeutrinoController subclass, and have all the other application Controllers extend it. In that class, override createRecord() to construct any custom Records, and call NeutrinoController in turn for the standard ones.

Alternatively, for smaller applications with perhaps only one custom Controller, implement RecordFactory directly within that Controller.

The following code fragment illustrates how to implement RecordFactory within a custom Controller superclass.

```
// NeutrinoController implements RecordFactory already
// so we don't need to state it here
public class MyAppController
extends NeutrinoController
{
  // RECORDFACTORY IMPLEMENTATION

  public Record
  createRecord (String inRecordType)
  {
    Record record = null;

    if (inRecordType.equals ("user"))
    {
      // make our User record, not the standard one
      return new models.User ();
    }
    else
    {
      // let the standard Controller handle it
      record = super.createRecord (inRecordType);
    }

    return record;
  }
}
```

Scheduling Tasks

Neutrino Server incorporates a facility called Task Manager whereby tasks can be scheduled to be run after a specified period, with optional repetitions.

Configuration

Task Manager requires the existence of the tasks database table to operate. SQL to add the tasks table to the database can be found in app/db/fixtures-base.sql in the application template. Note that tasks are persisted in the database so that server restarts don't affect anything.

Control

Neutrino Dispatcher starts Task Manager automatically when the server starts up. If the tasks table is not present in the database, then it will fail non-destructively to initialise, leaving messages in the server log. It is not recommended for application code to start or stop Task Manager.

The Java class representing Task Manager is a singleton, therefore it is accessed and instantiated by the same method --

```
// get the solitary TaskManager instance
// and instantiate it if it is not already made
TaskManager taskManager = TaskManager.getInstance ();
```

By default, Task Manager checks for due tasks every minute. If application code really needs Task Manager to check the queue at a specific moment, then it is allowed to call *run()* manually.

Running Tasks

Task Manager persists tasks as Java class names and parameters. When the time comes to run a task, it is instantiated from its class name and then configured via its stored parameter set. Note that tasks *must* inherit from TaskBase (com.subatomicsystems.neutrino.server.TaskBase) in order to successfully execute. Once a task has successfully completed, Task Manager checks its "period" property. If this is zero, then the task is presumed to be a one-off, or presumed to have itself determined that its periodicity has ended, and the task is then deleted from the database.

The following code fragment shows how to schedule a task to run in five minutes, then every minute after that.

```
// assuming that myapplication.Task inherits from TaskBase
String      className = "myapplication.Task";

Map<String, String> parameters = new HashMap<String, String> ();
parameters.put ("param1", "value1");
parameters.put ("param2", "value2");

long  now = System.currentTimeMillis ();
long  initialTime = now + (5 * 60 * 1000);
int   period = 60 * 1000;

TaskManager taskManager = TaskManager.getInstance ();
int   taskID = taskManager.addTask
  (className, parameters, initialTime, period);
```

In order to cancel a scheduled task, call *cancelTask()*, passing the database connection and the task ID which is returned from *addTask()*.

Sending Email

Neutrino Server incorporates a facility called Postal Service, which is a multi-threaded email sender based on the standard JavaMail and ExecutorService facilities included as standard within JSE.

Configuration

Postal Service requires configuration to do its job and is configured via the Configuration database table. The following is a list of suggested Configuration entries (for Google mail) with their default values. SQL to add the configuration table and suggested entries to it can be found in app/db/fixtures-base.sql in the application template.

Domain	*Name*	*Default Value*

mail	mail.transport.protocol	smtp
mail	mail.smtp.host	smtp.gmail.com
mail	mail.smtp.auth	true
mail	mail.smtp.port	465
mail	mail.smtp.socketFactory.port	465
mail	mail.smtp.socketFactory.class	javax.net.ssl.SSLSocketFactory
mail	mail.smtp.socketFactory.fallback	false
mail	mail.smtp.username	(no default)
mail	mail.smtp.password	(no default)

Control

If an attempt to send a message occurs before the Postal Service has been started, then it will start itself, but it can be controlled directly if required. The Java class representing the Postal Service is a singleton, therefore it is accessed and instantiated by the same method --

```
// get the solitary PostalService instance
// and instantiate it if it is not already made
PostalService     service = PostalService.getInstance ();
```

Once a reference to the Postal Service is obtained, it can be started and stopped using *start()* and *stop()* methods. In addition, the number of transmission threads can be changed from the default of 5 using the *setNumTransmissionThreads()* method.

Sending Mail

Once a reference to the Postal Service is obtained, mail can be scheduled for transmission by passing the relevant details it to *send()* --

```
public void
sendMessage
  (String inAddress, String inScreenName, String inSubject, String inBody)
{
  // populate the sender from configuration
  // so that it matches the configured sending credentials, etc
  Configuration config = Configuration.getInstance ();

  String from = config.get ("mail", "mail.from.address");
  String fromName = config.get ("mail", "mail.from.name");
  InternetAddress fromAddress = new InternetAddress (from, fromName);
  // build the destination address from the parameters
  InternetAddress toAddress = new InternetAddress (inAddress, inScreenName);

  // queue the message for transmission
  PostalService.getInstance ().send
    (fromAddress, toAddress, inSubject, inBody);
}
```

Note that as the process is asynchronous, there is no way of detecting failure other than monitoring the reply address for mail system notifications.

Server Reference