Umber Ballast 0.3.5 User Guide

Chapter 1 - Introduction

Umber Ballast helps create the foundation for your application. In most cases, this involves code generation of lower level object models, for example Hibernate Javabean/database bindings. The current version of Ballast ships with a DTD-to-JavaBean generator and a Hibernate bindings generator. Later releases will improve the Hibernate code generation, add new types of code generation, and introduce an extensible runtime architecture which you can tweak for your own needs.

Chapter 2 - The Generator

Ballast provides a high level interface for all processing actions, in particular code generation. The IGenerator interface has a single generate() method which accepts a BallastInput object as input and returns a BallastOutput object which contains all content produced by the generator. It throws a BallastGeneratorException if it can't finish processing the input.

```
BallastOutput generate (BallastInput input)
      throws BallastGeneratorException
```

The input and output objects are simple JavaBeans. BallastInput holds a simple array of Template objects (an inner class of BallastInput):

```
public BallastInput ()
public BallastInput (Template[] templates)
public void setTemplates (Template[] templates)
public Template[] getTemplates ()
public void addTemplate (String name, String content)
```

Templates can be added indirectly to the BallastInput with the addTemplate() method, or by instantiating them directly and passing them into the constructor or setTemplates() method. A Template is simply a named template with its text content.

```
public Template (String name, String content)
public String getContent ()
public String getName ()
```

Each generator might use the template name differently, often as the name of the output file or a java base package, so it must match what the generator expects. The content is the full text of the template (for Velocity or another templating engine used by the generator) to process. The application must load the template itself (see ResourceLoader in the Umber Core docs) before invoking the generator.

The generator wraps one or more output files within a single BallastOutput object. The getOutputNames() method retrieves the unique names of all data objects created by the generator; this also serves as a total object count. The getOutput() method can then retrieve each output data object by name. Like the template names, the output names are dependent on the generator, but will usually be the file name to save the data to. (The generator does not write any output files; it only deals with data in memory.)

```
public String[] getOutputNames ()
public Object getOutput (String outputName)
```

The constructor and addOutput() method are typically only invoked by the generator.

```
public BallastOutput ()
public void addOutput (String outputName, Object data)
```

Chapter 3 - Schema-Based Code Generation

The Umber Bellows library has a generic schema parser data model designed to handle XML DTD (and soon XML Schema) definitions. Ballast can take those schema models and pass them into an IGenerator implementation for easy template processing. Currently, Ballast has two ready-to-use schema-based generator classes: SchemaGenerator and SchemaJavaGenerator, and a TemplateRunner command-line class which wraps both for easy use.

Section 3.1 - SchemaGenerator

SchemaGenerator produces one output file for each Template object you pass in to its generate() method. When you create the generator, you must pass it an instantiated BellowsSchema object (see Umber Bellows documentation); it will use the same schema for all calls to generate().

```
public SchemaGenerator (BellowsSchema schema)
  throws BallastGeneratorException
```

The generator takes each input Template and runs its content as a Velocity template, exporting certain values (see below) from the schema as Velocity variables available to the template. It then takes the results from the Velocity engine and adds it to the BallastOutput with the template name from the BallastInput. Thus, passing three Template objects into generate() will result in three outputs with the corresponding Template names.

The generator defines three properties for each Velocity template it processes:

```
$schema -> the BellowsSchema object from the constructor
$root   -> the top level ISchemaNode object in the schema
$card   -> the SchemaCardinality object for the root node
```

See the BellowsSchema documentation for details on these objects. For convenience in your Velocity templates, the ISchemaNode objects implement an extra method not in the interface:

```
public PropertyName getNameProp ()
```

This method returns the node name wrapped in a PropertyName object (see Umber Core docs). This makes it easy to convert an element name in the schema to many different naming styles.

For example, the PropertyName.getCaseDelimitedName() method returns a version of its contents suitable for Java class names. So if your root schema node defines a "my-root" element, the following Velocity template fragment would resolve to the text "MyRoot":

```
$root.NameProp.CaseDelimitedName
```

A useful default Velocity template for converting a schema into an XML file exists in the Ballast jar file, at:

```
org/umber/ballast/schema/schema-to-xml.tmpl
```

You can load it easily with ResourceLoader. It demonstrates how to recursively traverse a BallastSchema tree and is a good starting point for writing your own templates.

Section 3.2 - SchemaJavaGenerator

The SchemaJavaGenerator is more complex than the one-to-one behavior of SchemaGenerator. The java generator is designed to handle the peculiarities of

creating a set of JavaBean object bindings for a given schema. It creates one Java source file for each element node in the schema, all rooted in a single java package.

```
public SchemaJavaGenerator (BellowsSchema schema)
  throws BallastGeneratorException
```

You pass in a template for the JavaBean source code, and the generator will iterate through the schema tree, running each ISchemaNode through the template individually. It will declare package declarations and generate proper file names for you.

Each Template object you pass into the BallastInput should have the java package as the template name. One Template corresponds to an entire package full of classes. You might use multiple Templates if you want to generate different versions of the same schema object model, each in different packages. For example, you might have a very simple data holder model, and a second more complex model with extra validation.

The BallastOutput will contain one entry per schema node, per Template in the input; the output name of each is the relative file path to save the source code. Thus, for a schema with four element definitions and an input with two Template objects, you would have eight named outputs.

So, given the DTD:

```
<!ELEMENT root (child1 | child2 | child3)>
<!ELEMENT child1 EMPTY>
<!ELEMENT child2 EMPTY>
<!ELEMENT child3 EMPTY>
```

and two templates with the names:

```
com.proj.models.one
com.proj.models.two
```

the output would contain contents with the following names:

```
com/proj/models/one/Root.java
com/proj/models/one/Child1.java
com/proj/models/one/Child2.java
com/proj/models/one/Child3.java
com/proj/models/two/Root.java
```

```
com/proj/models/two/Child1.java
com/proj/models/two/Child2.java
com/proj/models/two/Child3.java
```

In addition to the $schema, $root, and $card Velocity properties that SchemaGenerator sets, SchemaJavaGenerator defines these for each template it handles:

```
$node          -> the ISchemaNode for the current element
$className     -> Java class name for current element
$pkg           -> Java package for current element
$fullClassName -> fully qualified java class name
$file          -> java file path, incl full package
```

The default template for javabean creation resides at:

```
org/umber/ballast/schema/schema-to-java.tmpl
```

SchemaJavaGenerator has a static convenience method to convert a fully qualified Java class name into a file path for the Java file. For example, a Java class "my.proj.beans.MyRoot" would convert to "my/proj/beans/MyRoot.java".

```
public static String packageToFile (String packageName)
```

Section 3.3 - Template Runner

Although the IGenerator implementations perform a lot of work behind the scenes, they still require a fair bit of wrapper code, for loading schemas and template, writing output files, and so on. The TemplateRunner fills this gap, with a simple command-line interface for generating output from arbitrary schema, through arbitrary templates, into arbitrary output directories.

The TemplateRunner main() method takes the following mandatory command-line arguments:

```
--dtd     (DTD schema file)
--outDir  (output directory for generated files)
--format  (which generator to use: "file" or "java")
```

The input argument (--dtd) is relative to the CLASSPATH, and can be a file in the file system or a resource loaded from a jar file. The output argument (--outDir) is relative to the current working directory, typically wherever the java command is invoked (though this could be overridden, for example inside an IDE).

The --format argument determines which generator TemplateRunner will invoke for the processing: "file" uses SchemaGenerator and "java" uses SchemaJavaGenerator. This argument also affects which of the two optional output arguments you must use:

Section 3.4 - --outFile (single output file name)

If --format is "file", you must set --outFile to the output file name you want; the generated content is stored in a single file. Thus, if -outDir is set to "data-out" and --outFile is "mydata.txt", the generator will create the file data-out/mydata.txt inside the current working directory.

Conversely, if --format is "java", you must set --outPkg to a valid Java package name. All generated Java classes will be placed in a corresponding sub-directory. If --outDir is "gen-src" and --outPkg is "my.proj.beans", the generator will write all source files to the gen-src/my/proj/beans directory.

The options above will fall back on the default template for each generator (see above). The optional --tmpl argument overrides the default with a custom template.

```
--tmpl  (custom template)
```

Check the samples directory of the Ballast distribution for copies of the default templates.

Chapter 4 - Hibernate Code Generation

Ballast creates Hibernate bindings dynamically, by connecting to a live database, examining the table definitions, and invoking Jakarta Velocity templates to mass-produce Java code and Hibernate XML mappings.

An XML config file declares JDBC connection information to the database, the java package path for the generated code, the output directory, and a list of database tables to create bindings for. The XML must conform to the format below.

```
<ballast-config>
  <output-dir>gen-src</output-dir>
  <jdbc>
    <jdbc-config database-url="jdbc:hsqldb:mem:testdb"
      driver-class="org.hsqldb.jdbcDriver"
```

```
      username="sa" password=""/>
  </jdbc>

  <hibernate>
    <hibernate-config java-package="my.pkg1">
      <tables>
        <table>TABLE1</table>
        <table>TABLE2</table>
      </tables>
    </hibernate-config>
    <hibernate-config java-package="my.pkg2">
      <tables>
        <table>TABLE3</table>
      </tables>
    </hibernate-config>
  </hibernate>
</ballast-config>
```

Each config file has a single JDBC connection, a single output directory, but can have multiple database tables spread out across multiple Java packages. Each package can use the default templates, or custom templates by setting optional properties on the <hibernate-config> element.

```
<hibernate-config java-package="my.pkg3" java-pk-template="my-javabean-pk.tmpl" java-t
e="my-javabean.tmpl" xml-template="my-hibernate-xml.tmpl">
  <tables>
    <table>TABLE1</table>
    <table>TABLE2</table>
  </tables>
</hibernate-config>
```

The same database table can appear in more than one <hibernate-config> element; you might do this if you want different bindings for the same table.

To invoke the code generator with one or more XML config files, run the HibernateMain method, giving the path to each file on the command line. The path must be absolute, from the CLASSPATH root, to a file on the filesystem or to a resource inside a jar on the CLASSPATH.

The code generator can also be invoked by API, from inside your application. Create and populate a BallastConfig object and pass it to the HibernateMain.process() method:

```
BallastConfig config = ...
new HibernateMain ().process (config);
```

You can load the BallastConfig object programmatically, or you can load it from a file or resource with the ConfigFactory. This is particularly useful if you want to manage your JDBC login info at the application level, and keep it out of the filesystem.

The ConfigFactory class has four static load methods:

```
public static BallastConfig loadXmlConfig (String xml)
public static BallastConfig loadXmlConfigFromResource (String path)
  throws BallastConfigException
public static BallastConfig loadXmlConfigFromFile (File file)
  throws BallastConfigException
```

The first method loads XML in string form; the next method loads XML from a resource path (see ResourceLoader in Umber Core); the final method loads XML from the given File.

Chapter 5 - Ballast Ant Task

Another way to invoke the Ballast Hibernate generator is through the included custom Ant task, BallastAntTask. Use the Ant <taskdef> task to load it in an Ant script.

```
<target name="load-ballast">
  <taskdef classname="org.umber.ballast.util.BallastAntTask" name="ballast">
    <classpath>
      <pathelement location="${jar-dir}/umber-tools-0.3.3.jar"/>
      <pathelement location="${core-lib}/commons-logging-1.0.4.jar"/>
      <pathelement location="${core-lib}/velocity-dep-1.4.jar"/>
    </classpath>
  </taskdef>
</target>
```

Ballast config files can then be invoked from an Ant script with the <ballast> task (or whatever you put in the <taskdef> 'name' attribute). The <ballast> task can have any combination of <dtdtobean>, <dtdtofile>, and <jdbctobean> subtask elements. The following example demonstrates one of each subtask types, although in practice you can have as many of each as you like. All subtask attributes can

contain Ant variables, as in "${outdir}" below.

```
<target name="hibernate-sample" depends="load-ballast"
  description="Example Ballast task">
  <ballast>
    <dtdtofile
      dtd="my/proj/schema/sample-schema.dtd"
      template="my/proj/templates/custom-xml-template.vm"
      outdir="${outdir}/gen-xml"
      outfile="my-schema.xml"/>

<dtdtobean dtd="my/proj/schema/sample-schema.dtd" javapkg="my.proj.model" outdir="${ou
gen-src" template="my/proj/templates/custom-bean-template.vm"/>

    <jdbctobean config="${samples-dir}/sample-config.xml"/>
  </ballast>
</target>
```

The first subtask is <dtdtofile> which invokes SchemaGenerator to convert an XML DTD file into a single output file. The 'dtd' attribute specifies where on the CLASSPATH the DTD file is; the DTD can be wrapped inside a jar, or inside a directory on the file system that is accessible through the CLASSPATH. The optional 'template' attribute points to the Velocity template, also on the CLASSPATH. If not specified, the 'template' attribute falls back on the default XML-generating template schema-to-xml.tmpl. The 'outdir' attribute specifies where to write the single output file, and 'outfile' provides the output file name.

```
<dtdtofile dtd="my/proj/schema/sample-schema.dtd" outdir="${outdir}/gen-xml" outfile="
ema.xml" template="my/proj/templates/custom-xml-template.vm"/>
```

The <dtdtobean> subtask wraps SchemaJavaGenerator; its attributes are the same as <dtdtofile> except 'outfile' is replaced by 'javapkg' which holds the java package to store the generated Java classes in.

```
<dtdtobean dtd="my/proj/schema/sample-schema.dtd" javapkg="my.proj.model" outdir="${ou
gen-src" template="my/proj/templates/custom-bean-template.vm"/>
```

The <jdbctobean> subtask kicks off Ballast Hibernate code generation. Its single attribute 'config' points to a Ballast config file where it gets all information about which database to connect to and which Velocity templates to use.

```
<jdbctobean config="${samples-dir}/sample-config.xml"/>
```

Chapter 6 - Database Metadata

The Ballast config XML file can also be used to create simple text reports of JDBC metadata. The MetadataBrowser class loads the config file, logs into the database, and prints out the all the metadata it can find for each table in a <hibernate-config> tag. This includes all columns, keys, and indexes.

MetadataBrowser can be called with a resource path to the config file, or with an already loaded BallastConfig object. It sends the output to the commons logger.

```
public void dump (String configPath)
      throws BallastConfigException, BallastQueryException
public void dump (BallastConfig config)
      throws BallastConfigException, BallastQueryException
```