

TorqueBox

The Ruby Application Platform

1.0.0.Beta20

by JBoss by Red Hat

1. What is TorqueBox?	1
1.1. Built upon JBoss AS	1
1.2. Built upon JRuby	1
1.3. Open-Source	1
2. Differences between TorqueBox & Traditional Ruby	3
2.1. The "application platform" concept	3
3. Installation	5
3.1. Installation using Complete Binary Distribution	5
3.1.1. Ensure you have Java 6	5
3.1.2. Get the latest version of TorqueBox binary package	5
3.1.3. Unzip it somewhere handy	5
4. Web Applications	7
4.1. Rack Applications	7
4.1.1. Applications with <code>config.ru</code>	7
4.1.2. Rack deployment descriptor (<code>*-rack.yml</code>)	7
4.2. Ruby-on-Rails Applications	9
4.2.1. Preparing your Rails application	10
4.2.2. Deploying	12
4.3. Controlling the TorqueBox Server	17
4.3.1. Using Rake	17
5. Scheduled Jobs	19
5.1. What Are Scheduled Jobs?	19
5.2. Ruby Job Classes	19
5.3. <code>TorqueBox::Jobs::Base</code> module	20
5.3.1. Logging	20
5.4. Scheduling Jobs	21
5.4.1. Ruby-on-Rails: <code>\$RAILS_ROOT/config/jobs.yml</code>	21
5.4.2. <code>jobs.yml</code> Format	21
6. Messaging	23
6.1. Message Broker	23
6.1.1. Basics	23
6.1.2. HornetQ	23
6.1.3. Producers and Consumers	23
6.1.4. Queues and Topics	23
6.2. Deploying Destinations	24
6.2.1. Deployment Styles	24
6.2.2. Deployment Descriptors	25
6.3. Ruby Consumers	26
6.3.1. Low-level message consumption	26
6.3.2. Higher-level message consumption	26
6.3.3. Connecting Consumers to Destinations	28
6.4. Ruby Producers	29
6.4.1. Create a client	30
6.4.2. Using the client	31

- 6.4.3. Sending a text message 31
- 6.4.4. Sending an object message 31
- 6.4.5. Tasks 31
- 7. Ruby Runtime Pooling** 35
 - 7.1. Control thread concurrency using `pooling.yml` 35
 - 7.1.1. Pooling for web requests 35
- 8. Cryptography** 37
 - 8.1. Cryptographic Key Stores 37
 - 8.2. Configure Key Stores: `crypto.yml` 37
 - 8.2.1. Ruby-on-Rails and `crypto.yml` 37
 - 8.3. Managing Key Stores 38
 - 8.3.1. Create a New Key Store 38
 - 8.3.2. Create a New Key Pair 38
 - 8.3.3. Inspect the Key Store 39
- 9. SOAP Endpoints** 41
 - 9.1. Introduction to SOAP & Data-Binding 41
 - 9.2. Ruby Endpoints 41
 - 9.2.1. Basic Implementation & Configuration 42
 - 9.2.2. Operation Implementation 44
 - 9.3. Data-Binding 45
 - 9.3.1. Basics 46
 - 9.3.2. Collections 46
- 10. Telecom (SIP and Media)** 49
 - 10.1. Making phone calls 49
 - 10.2. Handling phone calls (Ruby Telco Classes) 50
 - 10.3. Media Support 51
- 11. Capistrano Support** 55
 - 11.1. Capistrano 55
 - 11.2. App-Server separate from App 55
- A. GNU Lesser General Public License version 3 57
- B. MIT License 61

What is TorqueBox?

TorqueBox provides an enterprise-grade environment that not only provides complete Ruby-on-Rails compatibility, but also goes beyond the functionality offered in traditional Rails environments.

1.1. Built upon JBoss AS

Instead of building a Ruby Application Platform from the ground-up, TorqueBox leverages the existing ninja-grade functionality JBoss has been shipping for years in the JBoss Application Server. JBoss AS includes high-performance clustering, caching and messaging functionality. By building Ruby capabilities on top of this foundation, your Ruby applications gain more capabilities right out-of-the-box.

1.2. Built upon JRuby

JRuby is a fast, compliant implementation of the Ruby language upon the Java Virtual Machine. Pure Ruby applications run un-modified within the JRuby interpreter. By binding JRuby to the components within JBoss, their functionality is exposed in a manner suitable to Rubyists.

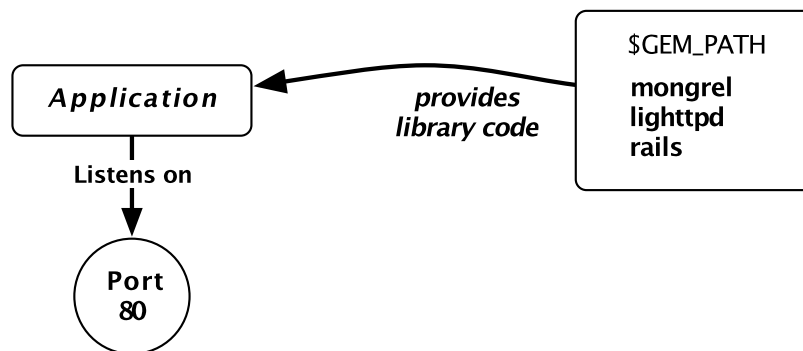
1.3. Open-Source

TorqueBox is a product of the JBoss Community, and is completely open-source software. TorqueBox is licensed under the LGPL. You may download the binaries or the source-code, modify it if you desire, and use it, even for profit, without any licensing costs.

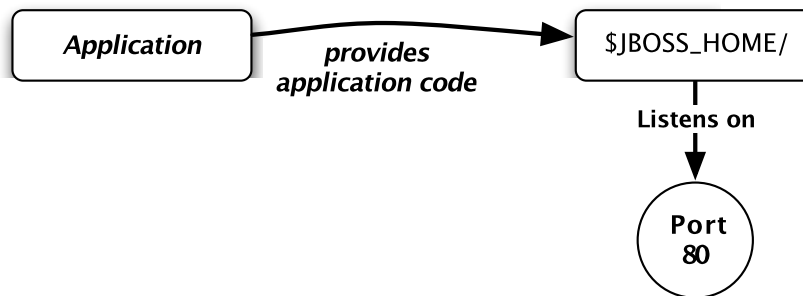
Differences between TorqueBox & Traditional Ruby

2.1. The "application platform" concept

Traditionally, Ruby applications were responsible for their services from the ground-up. You literally ran the application. It would import support libraries to handle HTTP listening, for example.



An application platform provides the foundations for any and all application functionality. The deliverable application itself does not need to handle the networking layers, the messaging facilities or the clustering logic. This is provided to the application "for free".



Installation

3.1. Installation using Complete Binary Distribution

The latest Complete Binary Distribution contains:

- The TorqueBox server, ready-to-run
- A complete JRuby installation

3.1.1. Ensure you have Java 6

TorqueBox requires Java JDK 6.

To determine which version, if any, is installed on your system, at a command-line, attempt to run the `java` command with the `-version` argument.

```
$ java -version
java version "1.6.0_07"
Java(TM) SE Runtime Environment (build 1.6.0_07-b06-153)
Java HotSpot(TM) 64-Bit Server VM (build 1.6.0_07-b06-57, mixed mode)
```

If the version is at least 1.6, your version of Java is sufficient.

If you have no Java installed, or a version less than 1.6, you'll need to install a Java Development Kit. For many systems, it is easy to install the open-source OpenJDK.

For installation on Ubuntu, Fedora, OpenSuse, or Debian, please refer to the [installation instructions provided](http://openjdk.java.net/install/) [http://openjdk.java.net/install/] by the OpenJDK project. If you find a `java` on your system, ensure that it is not actually `gcj`. The `gcj` is insufficient for running the TorqueBox server.

For Apple OSX systems, Apple provides a JDK version 6.

3.1.2. Get the latest version of TorqueBox binary package

You can obtain the latest version of TorqueBox from the TorqueBox repository. As of this writing, the latest version is 1.0.0.Beta20.

<http://repository.torquebox.org/maven2/releases/org/torquebox/torquebox-bin/1.0.0.Beta20/torquebox-bin-1.0.0.Beta20.zip>

3.1.3. Unzip it somewhere handy

We'll install TorqueBox under your user's `$HOME` directory.

```
$ unzip torquebox-bin-1.0.0.Beta20.zip
```

Chapter 3. Installation

```
$ cd torquebox-1.0.0.Beta20
```

Before using the TorqueBox server, you must set up your environment. To make it easier to upgrade without having to reconfigure your environment, it is useful to create a symlink to the versioned directory produced when you unpackaged the distribution.

```
$ ln -s torquebox-bin-1.0.0.Beta20 torquebox-current
```

Next, `$TORQUEBOX_HOME`, `$JBOSS_HOME` and `$JRUBY_HOME` need to be set, and adjusting your `$PATH` will make working with the package easier. You can either run the following commands each time on the command-line, or add them to your `.bash_profile`.

First, the various `$X_HOME` variables are set so that each subsystem can find its supporting files.

```
export TORQUEBOX_HOME=$HOME/torquebox-current
export JBOSS_HOME=$TORQUEBOX_HOME/jboss
export JRUBY_HOME=$TORQUEBOX_HOME/jruby
```

Next, we make sure that JRuby's binaries are first in our executable `$PATH`, before any previously-installed Ruby packages.

```
export PATH=$JRUBY_HOME/bin:$PATH
```

By doing this, commands such as `rake`, `gem`, and `rails` will load from the TorqueBox-provided JRuby installation.

Web Applications

TorqueBox provides support for several popular Ruby web application frameworks. Generally speaking, TorqueBox integrates with web applications at the level of the Rack specification.

Each application ends up running within TorqueBox in the same fashion that any typical Java web-application does. The Java-based server bridges to Ruby/Rack-based applications through the Java Servlet facility.

4.1. Rack Applications

Rack is a specification which describes how web server engines can integrate with additional logic written in Ruby. Rack is akin to CGI or the Java Servlets Spec in terms of goals and functionality.

4.1.1. Applications with `config.ru`

TorqueBox currently supports general `config.ru`-based applications. In your application's directory, your Rack application can be booted from a file named `config.ru` that you provide. You may override this name using the `rackup` parameter, [described below](#). The Ruby runtime provided to your application is quite rudimentary. If you desire to use RubyGems or other libraries, it is up to you to require the necessary files (for instance, `require 'rubygems'`).

```
app = lambda{|env|
  [ 200,
    { 'Content-Type' => 'text/html' },
    'Hello World'
  ]
}
run app
```

4.1.2. Rack deployment descriptor (`*-rack.yml`)

To customize some of the aspects of deployment, instead of using the Rake tasks, you may manually create a *deployment descriptor*. A deployment descriptor is a small text file that is placed in the `deploy/` directory of the server in order to have the application deployed.

4.1.2.1. Location & Naming

The deployment descriptor needs to be placed within the `deploy/` directory of the AS configuration in use. If you are using the default configuration, the path would be:

```
$JBOSS_HOME/server/default/deploy/
```

The descriptor is a YAML file, and must end with the suffix of `-rack.yml`. The prefix is arbitrary, but is usually some form of your application's name.

`$JBOSS_HOME/server/default/deploy/myapp-rack.yml`

4.1.2.2. Contents of the descriptor

The descriptor has 2 main sections:

1. General application configuration
2. Web-specific configuration

4.1.2.2.1. General Application Configuration

The application section describes the `RACK_ROOT` and `RACK_ENV` for the deployed application. Under traditional (mongrel, lighttpd) deployments, this information is picked up through the current working directory or environment variables. Since the TorqueBox Server runs from a different location, the current working directory has no meaning. Likewise, as multiple applications may be deployed within a single TorqueBox Server, a single global environment variable to set `RACK_ENV` is nonsensical.

Example 4.1. Application configuration in `*-rack.yml`

```
application:  
  RACK_ROOT: /path/to/myapp  
  RACK_ENV: development
```

4.1.2.2.1.1. Rackup Config File

By default, the TorqueBox server will look for a rackup config file named `config.ru`. You can override the default name using the `rackup` config parameter.

Example 4.2. Application configuration in `*-rack.yml`

```
application:  
  RACK_ROOT: /path/to/myapp  
  RACK_ENV: development  
  rackup: your_app_name.ru
```

4.1.2.2.2. Web-specific configuration

Traditional Rails applications are deployed individually, without respect to hostnames or context-path. Running under TorqueBox, you may host several apps under a single host, or multiple apps under different hostnames.

Both the virtual-host and context-path configuration are nested under the `web` section.

4.1.2.2.2.1. Virtual Hosts

Virtual hosts allow one application to respond to *www.host-one.com*, while another running within the same JBoss AS to respond to *www.host-two.com*. If no host is specified, then the application will respond to all requests directed at the TorqueBox Server.

Example 4.3. Virtual host configuration in *-rack.yml

```
web:  
  host: www.host-one.com
```

4.1.2.2.2.2. Context paths

In addition to virtual hosts, applications within a single TorqueBox Server may be separated purely by a *context path*. For a given host, the context path is the prefix used to access the application. Traditional Rails apps respond from the top of a site. By using a context path, you can mount applications at a location under the root.

For example, **http://www.host-one.com/app-one/** could point to one application, while **http://www.host-one.com/app-two/** could point to another separate application.

Example 4.4. Context path configuration in *-rack.yml

```
web:  
  context: /app-one
```

The context path and virtual host configurations can be used at the same time, if desired.

Example 4.5. Virtual host with context path configuration in *-rack.ymls

```
web:  
  host: www.mycorp.com  
  context: /app-one
```

4.2. Ruby-on-Rails Applications

TorqueBox provides an enterprise-grade environment that not only provides complete Ruby-on-Rails compatibility, but also goes beyond the functionality offered in traditional Rails environments.

Ruby-on-Rails (also referred to as "RoR" or "Rails") is one of the most popular Model-View-Controller (MVC) frameworks for the Ruby language. It was originally created by David Heinemeier

Hansson at [37signals](http://37signals.com/) [http://37signals.com/] during the course of building many actual Ruby applications for their consulting business.

Rails has straight-forward components representing models, views, and controllers. The framework as a whole values convention over configuration. It has been described as "opinionated software" in that many decisions have been taken away from the end-user.

It is exactly the opinionated nature of Rails that allows it to be considered a simple and agile framework for quickly building web-based applications. Additionally, since Ruby is an interpreted language instead of compiled, the assets of an application can be edited quickly, with the results being immediately available. In most cases, the application does not need to be restarted to see changes in models, views or controllers reflected.

4.2.1. Preparing your Rails application

While TorqueBox is 100% compatible with Ruby-on-Rails, there are a few steps that must be taken to ensure success.

4.2.1.1. Using the application template

You can use the included application template to setup a new Rails application or modify an existing one to work with TorqueBox.

4.2.1.1.1. Creating a new Rails application

To create a new Rails application using the template, simply use the `-m` parameter when you execute the `rails` command.

```
$ rails -m $TORQUEBOX_HOME/share/rails/template.rb
```

This will perform the necessary setup to quickly get started with TorqueBox.

4.2.1.1.2. Applying template to an existing application

To apply the template to an existing application, simply use the `rails:template` rake task.

```
$ rake rails:template LOCATION=$TORQUEBOX_HOME/share/rails/template.rb
```

4.2.1.2. Manually configuring an application

4.2.1.2.1. Include the JDBC Gems for Database Connectivity

ActiveRecord applications deployed on TorqueBox benefit from using the Java-based JDBC database drivers. These drivers are provided as a handful of gems which you may include into your application through `config/environment.rb`.

JDBC gems for many popular databases is pre-installed with the TorqueBox server. You simply must reference the `activerecord-jdbc-adapter` from your `environment.rb` within the `Rails::Initializer.run` block.

```
Rails::Initializer.run do |config|

  config.gem "activerecord-jdbc-adapter",
    :lib=>'jdbc_adapter'

end
```

All databases will require inclusion of the `activerecord-jdbc-adapter`. No other gems need to be required or loaded, since ActiveRecord will perform further discovery on its own. Database gems supporting Derby, H2, HSQLDB, MySQL, PostgreSQL, and SQLite3 are supplied in the TorqueBox binary distribution.

4.2.1.2.2. Include the TorqueBox Ruby packages

In order to gain access to the advanced features of TorqueBox, you must include the TorqueBox packages into your project. The TorqueBox gems are also included into your application through `config/environment.rb`.

```
Rails::Initializer.run do |config|

  config.gem "torquebox-gem"
  config.gem "torquebox-rails"

end
```

4.2.1.2.2.1. `torquebox-gem`

The `torquebox-gem` provides access to advanced functionality, such as scheduled jobs and task queues.

4.2.1.2.2.2. `torquebox-rails`

The `torquebox-rails` provides additional `rake` tasks to your application. These tasks assist in deploying, undeploying, and executing the TorqueBox server. To make these tasks available to your project, you also need to add a `require` statement to the Rakefile at the top of your application.

```
require 'torquebox/tasks'
```

Invoking `rake -T` will show the tasks now available within your project.

```
rake torquebox:rails:check_frozen      # Check for that Rails has been
f...
rake torquebox:rails:deploy            # Deploy the Rails app
rake torquebox:rails:undeploy          # Undeploy the Rails app
rake torquebox:server:check            # Check your installation of the
...
rake torquebox:server:run              # Run TorqueBox server
```

4.2.1.2.3. Eliminate or replace "native" Gems

"Native" gems that rely upon machine-specific compiled code do not function with JRuby and TorqueBox. You must replaced these gems with pure-Ruby or pure-Java implementations. In the future, native gems using the FFI facilities will be usable.

4.2.2. Deploying

The TorqueBox Application Server is capable of serving many applications simultaneously. To add your application to the server, you must *deploy* it.

4.2.2.1. Deploy using the Rake tasks

The TorqueBox-Rails support package includes Rake tasks to deploy and undeploy your application from an instance of the TorqueBox Server.

First, the variable `$JBOSS_HOME` must be set to the path of the top of your JBoss installation as described in [Chapter 3, Installation](#)

```
$ export JBOSS_HOME=/path/to/torquebox/jboss
```

If you're using any configuration other than `default`, you must also set `$JBOSS_CONF`.

```
$ export JBOSS_CONF=web
```

Once these variables are set, you may perform a default deployment using the `jboss:rails:deploy` task.

```
$ rake torquebox:rails:deploy
```

To undeploy your application, the `jboss:rails:undeploy` task is available

```
$ rake torquebox:rails:undeploy
```

The TorqueBox Server does not need to be running for these commands to work.

By default, these tasks deploy your application to root of your TorqueBox Server's web space, without any virtual host configuration. To access the application once deployed, you should use your browser to access `http://localhost:8080/`.

When the application is deployed, a deployment descriptor is written to the `$JBOSS_HOME/server/$JBOSS_CONF/deploy/` directory with a filename based upon the directory name of your `$RAILS_ROOT`.

For instance, if your application was deployed from `/Users/bob/myapp/`, the deployment descriptor would be named `myapp-rails.yml`.

Rewriting or simply updating the last-modified time (using a command such as `touch`) of this descriptor will cause the TorqueBox server to redeploy the application. The `torquebox:rails:deploy` task simply emits this file.

Removing the descriptor will cause the TorqueBox server to undeploy the application. This is what the `torquebox:rails:undeploy` task does.

4.2.2.1.1. Deploying a non-root context

By default, the `torquebox:rails:deploy` task will attach your application to the root context. If you would rather deploy to a non-root context, you may provide it as an argument to the task invocation.

```
$ rake torquebox:rails:deploy['/my-application']
```

The root of your application would then be accessible at `http://localhost:8080/my-application`.

4.2.2.2. Deploy using a descriptor

To customize some of the aspects of deployment, instead of using the Rake tasks, you may manually create a *deployment descriptor*. A deployment descriptor is a small text file that is placed in the `deploy/` directory of the server in order to have the application deployed.

4.2.2.2.1. Location & Naming

The deployment descriptor needs to be placed within the `deploy/` directory of the AS configuration in use. If you are using the default configuration, the path would be:

```
$JBOSS_HOME/server/default/deploy/
```

The descriptor is a YAML file, and must end with the suffix of `-rails.yml`. The prefix is arbitrary, but is usually some form of your application's name.

```
$JBOSS_HOME/server/default/deploy/myapp-rails.yml
```

4.2.2.2.2. Contents of the descriptor

The descriptor has 2 main sections:

1. General application configuration
2. Web-specific configuration
3. Sip-specific configuration

4.2.2.2.1. General Application Configuration

The application section describes the `RAILS_ROOT` and `RAILS_ENV` for the deployed application. Under traditional (mongrel, lighttpd) deployments, this information is picked up through the current working directory or environment variables. Since the TorqueBox Server runs from a different location, the current working directory has no meaning. Likewise, as multiple applications may be deployed within a single TorqueBox Server, a single global environment variable to set `$RAILS_ENV` is nonsensical.

Example 4.6. Application configuration in *-rails.yml

```
application:  
  RAILS_ROOT: /path/to/myapp  
  RAILS_ENV:  development
```

4.2.2.2.2. Web-specific configuration

Traditional Rails applications are deployed individually, without respect to hostnames or context-path. Running under TorqueBox, you may host several apps under a single host, or multiple apps under different hostnames.

Both the virtual-host and context-path configuration are nested under the `web` section.

4.2.2.2.2.1. Virtual Hosts

Virtual hosts allow one application to respond to `www.host-one.com`, while another running within the same JBoss AS to respond to `www.host-two.com`. If no host is specified, then the application will respond to all requests directed at the TorqueBox Server.

Example 4.7. Virtual host configuration in *-rails.yml

```
web:  
  host: www.host-one.com
```

4.2.2.2.2.2. Context paths

In addition to virtual hosts, applications within a single TorqueBox Server may be separated purely by a `context path`. For a given host, the context path is the prefix used to access the application. Traditional Rails apps respond from the top of a site. By using a context path, you can mount applications at a location under the root.

For example, `http://www.host-one.com/app-one/` could point to one application, while `http://www.host-one.com/app-two/` could point to another separate application.

Example 4.8. Context path configuration in `*-rails.yml`

```
web:
  context: /app-one
```

The context path and virtual host configurations can be used at the same time, if desired.

Example 4.9. Virtual host with context path configuration in `*-rails.ymls`

```
web:
  host: www.mycorp.com
  context: /app-one
```

4.2.2.2.3. SIP-specific configuration

The sip configuration section allows you to define the apname (application name) of the SIP Servlets application (mandatory) and the name of the class that will handle the SIP messages :

Example 4.10. SIP configuration in `*-rails.yml`

```
sip:
  rubycontroller: SipHandler
```

4.2.2.2.3. Complete `*-rails.yml` Deployment Descriptor Example

```
application:
  RAILS_ROOT: /path/to/myapp
  RAILS_ENV: development
web:
  host: www.mycorp.com
  context: /app-one
sip:
  rubycontroller: SipHandler
```

4.2.2.3. Deployment using a partial descriptor

While the `myapp-rails.yml` descriptor may contain all the abovementioned information, you may also deploy using a combination of descriptor and additional files in your `config/` directory. The bare minimum required in the descriptor is the `RAILS_ROOT` setting for the `application:` block.

The primary purpose is to allow an application to be fully self-contained, specifying all vital production information within the same codebase as the rest of the application. This is useful for Capistrano-based deployments.

4.2.2.3.1. `config/web.yml`

The entire `web:` section of `myapp-rails.yml` may be supplied from within the application's own `config/web.yml` file, if present. If both `myapp-rails.yml` and `config/web.yml` exist, the `myapp-rails.yml` configuration values take precedent.

This allows for typical values to be placed in `config/web.yml` for production deployment, while allowing developers to override them through `myapp-rails.yml` during development deployments.

Example 4.11. `config/web.yml`

```
host: torquebox.org
context: /
```

4.2.2.3.2. `config/rails-env.yml`

If present in your application, `config/rails-env.yml` can specify the default value for `RAILS_ENV`. A value specified in `myapp-rails.yml` deployment descriptor may still override the value provided by the application's own `config/rails-env.yml`.

Example 4.12. Example `config/rails-env.yml`

```
RAILS_ENV: production
```

Similar to `config/web.yml`, usage of `config/rails-env.yml` can simplify production deployment by providing the correct value from within the app.

4.2.2.4. Deployment using a bundle

Rails applications may be deployed as atomic *bundles*. A bundle is simply an archive of the application's directory. The TorqueBox server deploys bundles created with the Java `jar` tool. Rake tasks are provided to assist with the creation and deployment of bundles.

4.2.2.4.1. Creating a bundle

The `torquebox:rails:bundle` rake task may be used to create a bundle of the application. The task invokes the Java `jar` commandline tool to bundle up the project directory, *excluding* `tmp/` and `log/` directories.

```
$ rake torquebox:rails:bundle
```

The resulting bundle will be placed at the root of the application, as a file named `myapp.rails`. To inspect the contents, you may use the `jar` tool.

```
$ jar tf
myapp.rails
META-INF/
```

```
META-INF/MANIFEST.MF
app/
app/controllers/
app/controllers/application_controller.rb
...
```

4.2.2.4.2. Deploying a bundle

To deploy a bundle, simply copy it to the `deploy/` directory of the server. The `torquebox:rails:deploy:bundle` rake task may be used to both create and deploy a bundle.

```
$ rake torquebox:rails:deploy:bundle
```

If you wish to deploy manually, a command similar to the following may be used

```
$ rake torquebox:rails:bundle
$ cp myapp.rails $JBoss_HOME/server/default/deploy/
```

If you redeploy a bundle, the server will remove the previous version, and hot-redeploy the bundle just copied.

4.3. Controlling the TorqueBox Server

4.3.1. Using Rake

To start the TorqueBox server from within your Rails application, the Rake task `torquebox:server:run` is provided.

```
$ rake torquebox:server:run
TorqueBox Server OK:
/Users/bob/oddthesis/jboss/jdk1.6/current/server/default
=====

JBoss Bootstrap Environment
```

To stop the server, simply interrupt the terminal using `control-c`. While the server is running, applications may be repeatedly deployed and undeployed.

Scheduled Jobs

5.1. What Are Scheduled Jobs?

Scheduled jobs are simply components that execute on a possibly-recurring schedule instead of in response to user interaction. Scheduled jobs fire asynchronously, outside of the normal web-browser thread-of-control. Scheduled jobs have full access to the entire Ruby environment. This allows them to interact with database models and other application functionality.

5.2. Ruby Job Classes

Each scheduled job maps to exactly one Ruby class. The path and filename should match the class name of the job contained in the file.

File name	Class name
mail_notifier.rb	MailNotifier
mail/notifier.rb	Mail::Notifier

Example 5.1. Skeleton scheduled job class (`mail/notifier.rb`)

```
module Mail
  class Notifier

    # implementation goes here

  end
end
```

Each job class should implement a no-argument `run()` method to perform the work when fired.

Example 5.2. Scheduled job implementation (`mail/notifier.rb`)

```
module Mail
  class Notifier

    def run()

      # perform work here

    end

  end
end
```

```
end
end
```

From within the class's `run()` method, the full application environment is available.

5.3. `TorqueBox::Jobs::Base` module

The optional `TorqueBox::Jobs::Base` module provides access to helpful functionality. It is not required to be included in your Ruby job class. By including it, though, extra functionality is introduced into your classes.

Example 5.3. Using `TorqueBox::Jobs::Base` module

```
require 'torquebox/jobs/base'

module Mail
  class Notifier
    include TorqueBox::Jobs::Base

    def run()

      # Use the logger provided by TorqueBox::Jobs::Base
      log.info( "Executing the job" )

      # perform work here
    end
  end
end
```

5.3.1. Logging

To gain access to a logging device, the `log()` method is available. Messages of various level can be logged.

Method	Use
<code>trace()</code>	Tracing program execution
<code>debug()</code>	Development-time debug information
<code>info()</code>	Information messages for the user
<code>warn()</code>	Warnings for the user
<code>error()</code>	Severe errors during execution

The log messages will be logged in the normal `server.log`.

```
10:02:35,074 INFO [Notifier] Executing the job
10:02:40,074 INFO [Notifier] Executing the job
```

5.4. Scheduling Jobs

The job schedule defines the time(s) that a job should execute. This may be defined to be single point in time, or more often, as recurring event. The job schedule is defined with a file named `jobs.yml`.

5.4.1. Ruby-on-Rails: `$RAILS_ROOT/config/jobs.yml`

For Ruby-on-Rails applications, the schedule should be placed at `$RAILS_ROOT/config/jobs.yml`.

5.4.2. `jobs.yml` Format

Within `jobs.yml`, a block of information is provided for each job. The block starts with arbitrary name for the job. Each block must also define the job class and the schedule specification. Optionally a description may be provided.

Example 5.4. Example `jobs.yml`

```
mail.notifier
job:      Mail::Notifier
cron:     0 */5 * * * ?
description: Deliver queued mail notifications
```

The cron attribute should contain a typical crontab-like entry. It is composed of 7 fields (6 are required).

Seconds	Minutes	Hours	Day of Month	Month	Day of Week	Year
0-59	0-59	0-23	1-31	1-2 or JAN-DEC	1-7 or SUN-SAT	1970-2099 (optional)

For several fields, you may denote subdivision by using the forward-slash (/) character. To execute a task every 5 minutes, `*/5` in the minutes field would specify this condition.

Spans may be indicated using the dash (-) character. To execute a task Monday through Friday, `MON-FRI` should be used in the day-of-week field.

Multiple values may be separated using the comma (,) character. The specification of `1,15` in the day-of-month field would result in the job firing on the 1st and 15th of each month.

Either day-of-month or day-of-week must be specified using the ? character, since specifying both is contradictory.

Messaging

6.1. Message Broker

6.1.1. Basics

A message broker is simply a service that facilitates asynchronous message-passing between application components.

6.1.2. HornetQ

TorqueBox integrates the JBoss HornetQ message broker technology. It is automatically available to you, with no additional configuration required to start the messaging service. HornetQ supports clustered messaging, to allow for load-balancing, failover, and other advanced deployments.

6.1.3. Producers and Consumers

While some components may be purely a producer, or purely a consumer, the roles are not mutually exclusive. Many components may rightly play multiple roles, consuming and producing messages in the act of performing its actions.

6.1.3.1. Producers

Any component or client code that creates messages and gives them to the message broker for delivery is considered a *producer*. Generally speaking, the producer does not know the details of the destination.

6.1.3.2. Consumers

Any component that waits for messages to be delivered to it by the message broker is consider a *consumer*. A consumer is unaware of the producer and any other consumers, potentially.

6.1.4. Queues and Topics

The message broker supports two different delivery semantics, referred to as *queues* and *topics*, which are specialization of the generic idea of *destinations*.

6.1.4.1. Destinations

A destination represents a place a message producer sends messages. From the producer's point-of-view, every destination, whether a queue or a topic, is simply a place to deposit messages. From the consumer's point-of-view, every destination, whether a queue or a topic, is simply a place to pick up messages.

The specific semantics of a destination, while important in the design of application components, does not necessary affect how individual producers or consumers are created. The topology of

how messages flow between them is defined outside of the scope of any individual producer or consumer.

In all cases, a destination may be fed messages from multiple producers.

6.1.4.2. Queues

A queue is a destination that allows multiple consumers to withdraw messages from it, but each message is delivered to exactly one consumer. Imagine a bank with multiple customer service windows, but a single line. As each bank teller becomes available, the next person in line goes to the next open window. In this scenario, the window tellers are the consumers, and the line of customers waiting service represent the queue of messages.

While a single consumer is allowed to consume each message, multiple producers may be responsible for placing messages into the queue.

6.1.4.3. Topics

A topic is a destination that sends a copy of a message to all interested consumers. Presentations at conferences may be thought of as topics. Each audience member has expressed interest in a particular topic, and the message producers are the presenters at the front of the hall.

Multiple consumers receive the same message, and multiple producers may be adding messages to the topic.

6.2. Deploying Destinations

Queues and topics (collectively known as destinations) may be deployed with with your application, or separate from your application. Additionally, various parts of your application may also implicitly deploy and use some destinations.

Each method has advantages and disadvantages involving the expectations of your application and its interaction with resources outside the scope of the application.

6.2.1. Deployment Styles

6.2.1.1. Deploying destinations with your application

If you decide to deploy your queues and topics with your application, you automatically align their lifecycle to the deployment cycle of your application. If you undeploy your application, your queues and topics will also disappear, and be unable to receive messages. If the queues are used only internally to your application, and short lifespan semantics are useful to you, deploying destinations with your application reduces deployment steps and moving parts.

6.2.1.2. Deploying destinations apart from your application

If you deploy destinations separate and apart from your application, they become long-lived first-class component citizens in your environment. Applications may be deployed and undeployed,

while the destinations continue to function, accepting and processing messages to the best of their ability.

If the consumers to a destination are offline, the destination may persist and store any unhandled messages until a consumer re-attaches.

The downside is that by making destinations first-class top-level components of your environment, you must also manage, deploy and undeploy them separate from any app, creating additional work.

6.2.2. Deployment Descriptors

Deploying queues and topics is as easy as creating simple YAML files, and placing them either in your application's `config/` directory, or in the application server's `deploy/` directory, depending on which deployment style you've chosen. Both methods may be combined. Server-deployed destinations and application-deployed destinations are indistinguishable once deployed.

6.2.2.1. `queues.yml`

To deploy queues, a simple YAML file is required, simply naming the queue, and providing additional configuration parameters. Currently, no additional configuration parameters are allowed.

Example 6.1. `queues.yml`

```
/queues/my_queue:  
  
/queues/my_other_queue:
```

The name of the queue will be used when registering the queue in the naming-service, and is used to discover the queue for attaching consumers and producers.

By convention, queues are named with the prefix of `/queues`.

6.2.2.2. `topics.yml`

To deploy topics, a simple YAML file is required, simply naming the topic, and providing additional configuration parameters. Currently, no additional configuration parameters are allowed.

Example 6.2. `topics.yml`

```
/queues/my_topic:
```

```
/queues/my_other_topic:
```

The name of the queue will be used when registering the topic in the naming-service, and is used to discover the topic for attaching consumers and producers.

By convention, topics are named with the prefix of `/topics`.

6.3. Ruby Consumers

Message consumers may be implemented in Ruby and easily attached to destinations. A Ruby consumer may either interact at the lowest JMS-level, or take advantage of higher-level semantics.

6.3.1. Low-level message consumption

For the lowest-level implementation of a Ruby consumer, the class must simply implement `on_message(msg)` which receives a `javax.jms.Message` as its parameter. Admittedly, this gets quite a lot of Java in your Ruby, but it's available if needed.

Example 6.3. Low-level message consumer

```
class MyLowConsumer
  def on_message(msg)
    # manipulate the javax.jms.Message here
  end
end
```

6.3.2. Higher-level message consumption

If you know the producers of messages will be using the TorqueBox client API (or following the correct conventions, at least), you may wish to take advantage of higher-level messaging semantics, and easier APIs.

For instance, message consumers may be triggered with either simple text, or a complex Ruby object graph as the message payload, instead of having to directly manipulate an instance of `javax.jms.Message`.

Using higher-level messaging semantics requires the cooperation of both the producer and consumer of messages.

6.3.2.1. Text-based messages

For simple text-based messages, a consumer may instead implement an `on_text(text)` method which will receive a string payload of the message.

Example 6.4. Text-based message consumer

```
class MyTextConsumer
  def on_text(text)
    # process the string `text` payload
  end
end
```

6.3.2.2. Object-based message

Consumers may also easily treat the message payload as a marshalled Ruby object graph. By implementing `on_object(obj)` method, the consumer will receive the unmarshalled Ruby object without additional work.

Example 6.5. Object-based message consumer

```
class MyObjectConsumer
  def on_object(obj)
    # work with Ruby object `obj` payload
  end
end
```

6.3.2.3. Mixed consumption

In the event you want the best of both worlds, both `on_text(...)` and `on_object(...)` may take a second parameter, through which the original `javax.jms.Message` will be passed. This can be useful if your consumer needs to generate a response back to a temporary reply-to destination.

Example 6.6. Mixed consumption

```
class MyMixedConsumer
  def on_object(obj, message)
    # work with Ruby object `obj` payload
    # AND have access to the `message` javax.jms.Message object
  end
end
```

6.3.3. Connecting Consumers to Destinations

You can connect consumers hosted within a TorqueBox-based application, or in external scripts. The method for each environment is similar, but slightly different, since TorqueBox-hosted consumers get more baked-in support from the container.

6.3.3.1. Connecting consumers within TorqueBox

To connect consumers within a TorqueBox-deployed application, you typically need to create a `consumers.rb` file, such as `config/consumers.rb` for Rails applications.

This file should evaluate to a container configuration object. A simple, no-op `consumers.rb` configuration looks like

```
TorqueBox::Messaging::Container::Config.create {  
  
}
```

Within the block, you may configure general connection information, such as the naming-service URL.

Example 6.7. Configuring naming-service within `consumers.rb`

```
TorqueBox::Messaging::Container::Config.create {  
  
  naming_provider_url 'jnp://some.other.host.com:1099/'  
  
}
```

Additionally, you may create a `consumers` sub-block, then use the `map` method to link destinations with consumers.

Example 6.8. Mapping destinations to consumers

```
TorqueBox::Messaging::Container::Config.create {  
  
  consumers {  
    map '/queues/my_queue', 'MyConsumer'  
  }  
}
```

Note that the consumer name should be a string and not a constant in this configuration file, due to load-time constraints.

6.3.3.1.1. Connecting Consumers outside of TorqueBox

You may also connect consumers outside of TorqueBox. The configuration is very similar, but your script must also launch the container that manages the consumers and their lifecycle.

Assuming you have the full array of required Ruby gems, any script can host consumers.

Example 6.9. Consumer hosting outside of TorqueBox

```
#!/usr/bin/env jruby

require 'rubygems'
require 'torquebox-messaging-container'

class MyConsumer
  attr_accessor :session

  def on_message(msg)
    puts "Received: #{msg.text}"
  end
end

container = TorqueBox::Messaging::Container.new {

  naming_provider_url 'jnp://localhost:1099/'

  consumers {
    map '/queues/foo', MyConsumer
  }
}

container.start
container.wait_until( 'INT' )
```

Notice that the consumer may be specified with a constant in this case, since the constant is definitely in-scope.

6.4. Ruby Producers

While consuming messages is easy with Ruby objects, producing them is just as simple.

The client library, usable outside of the core TorqueBox server even, is available as a Ruby gem named `torquebox-messaging-client`. A client is created, and can then be used to send either bare `javax.jms.Message` objects, or higher-level text and Ruby object graph messages.

6.4.1. Create a client

A client represents a configured connection capable of locating and using destinations managed by the message broker. To accomplish this, it needs to be able to connect to the naming-service where the queues and topics are registered. By default, this is `jnp://localhost:1099/`, which indicates it should connect through the Java Naming Protocol to the server running on port 1099 of the current host.

This is the default case for when you are running the TorqueBox server, as it provides a naming-service used by the components. If, by chance, you have bound the server to a non-localhost port, or are attempting to reach it from an external machine, you will need to adjust this connection parameter.

6.4.1.1. Create a client within TorqueBox

Within an application running inside TorqueBox, you may simply create a client with the simple no-argument version of `connect()`, and begin using it to communicate with the message broker.

Example 6.10. Client within TorqueBox

```
TorqueBox::Messaging::Client.connect do |client|
  # use `client` here
end
```

6.4.1.2. Create a client in external scripts

You may connect a client from any external script also, provided you have all the appropriate gems and JRuby installed. This is easy if you use the JRuby distribution that ships with TorqueBox.

Simply require `rubygems`, and `torquebox-message-client`, then optionally provide additional configuration information to the `connect(...)` method.

Example 6.11. Client from external script

```
#!/usr/bin/env jruby

require 'rubygems'
require 'torquebox-messaging-client'
```

```
Torquebox::Messaging::Client.connect(
  :naming_provider_url=>'jnp://some.other.host.com:1099/' ) do |client|
  # use `client` here
end
```

6.4.2. Using the client

Within the client block, your code can perform the role of a producer by sending messages to destinations using the `send(...)` method.

6.4.3. Sending a text message

To send a high-level text message, which a consumer may receive using `on_text(text)`, the `:text` option may be used.

Example 6.12. Sending a text message

```
client.send( '/queues/my_queue',
  :text=>'PING' )
```

6.4.4. Sending an object message

To send a high-level object message, which a consumer may receive using `on_object(obj)`, the `:object` option may be used.

Example 6.13. Sending an object message

```
client.send( '/queues/my_queue',
  :object=>{ :time=>Time.now,
  :crazy=>:legs } )
```

6.4.5. Tasks

A special case of message queues is to execute some task asynchronously. In this case, the messaging aspect happens to just be an implementation detail.

6.4.5.1. Task Classes

Task classes simply implement one or more methods that take a single object payload. It is often advisable to use a Ruby `Hash` for the payload parameter.

The method will be called asynchronously by the server when an object has been enqueued to it.

Example 6.14. Example task class

```
class EmailerTask
  def send_welcome(payload)
    address = payload[:address]
    # send welcome email to the user
  end

  def send_password_reset(payload)
    address = payload[:address]
    # send password-reset email to the user
  end
end
```

In Rails applications, these tasks should be placed in the `app/tasks/` directory, and be named with a suffix of `_task.rb` on the file, and a suffix of `Task` on the class name.

6.4.5.2. Task Queues

Each task discovered and deployed will also have a matching queue deployed. This queue is not supposed to be directly accessed, and is rather simply an implementation detail of the asynchronous task system.

6.4.5.3. Enqueue a task

Tasks can be enqueued from within an application running inside TorqueBox, or from external scripts.

6.4.5.4. Enqueuing from within TorqueBox

From code running within the TorqueBox server, simply calling the class-level `enqueue(...)` method on `TorqueBox::Messaging::Tasks` will result in triggering the task consumer on the other end of the queue.

Example 6.15. Enqueuing a task within TorqueBox

```
TorqueBox::Messaging::Tasks.enqueue(
  'emailer#send_password_reset',
  { :time=>Time.now, :user=>22 }
)
```

The `enqueue(...)` method takes two parameters. The first is a string which specifies the task and the method upon it to execute. The second parameter is the object payload to hand to the task method.

6.4.5.5. Enqueuing from external scripts

From scripts running outside of the TorqueBox server, you may still enqueue objects to the tasks running within it. In the case where the external script is running on the same host as the TorqueBox server, you may use the same simple invocation style as above.

You must include the `torquebox-messaging-tasks` gem in your script.

Example 6.16. Enqueuing a task from external scripts, same host

```
#!/usr/bin/env jruby

require 'rubygems'
require 'torquebox-messaging-tasks'

TorqueBox::Messaging::Tasks.enqueue(
  'emailer#send_password_reset',
  { :time=>Time.now, :user=>22 }
)
```

In the event you need to configure a tasks client to communicate with another, external message broker, you may use the `TorqueBox::Messaging::Tasks::Client` class, and connect it similar to connecting a generic messaging client.

Example 6.17. Enqueuing a task from external scripts, different host

```
#!/usr/bin/env jruby

require 'rubygems'
require 'torquebox-messaging-tasks'

TorqueBox::Messaging::Tasks::Client.connect(
  :naming_provider_url=>'jnp://some.other.host.com:1099/' ) do |client|

  client.enqueue(
    'emailer#send_password_reset',
    { :time=>Time.now, :user=>22 }
  )
end
```

Ruby Runtime Pooling

7.1. Control thread concurrency using `pooling.yml`

Since the TorqueBox platform supports Rails 2.2+, which are threadsafe, the default mode of operation is to share a single Ruby runtime across threads.

If your application is not designed to be threadsafe, you can instead pool the runtimes allowing a single-threaded model. Typically, if your application creates and uses global variables to manage state for a single request, you may have problems with the default multithreaded behavior.

To enable pooling-mode instead of shared multithreading you need to add a YAML file at `$RAILS_ROOT/config/pooling.yml`. This file is optional, and only required if you wish to enable pooling.

7.1.1. Pooling for web requests

A section named `web` defines the pooling configuration for all regular web requests. This does not include SOAP endpoints. SIP requests do participate in the web pool, though.

Web pooling is defined by creating a section named `web`, and specifying `min` and `max` parameters for the pool. The minimum number of runtimes will be created before your application starts. As the available items in the pool are exhausted, new ones will be asynchronously created, up to the maximum specified.

Currently there is no reaping performed on the pool reduce its size.

Example 7.1. Example `config/pooling.yml`

```
web:
  min: 2
  max: 30
```


Cryptography

8.1. Cryptographic Key Stores

Cryptographic key stores may be configured for the application and used by any service requiring key material. Keystores themselves are encrypted to secure the key material itself.

8.2. Configure Key Stores: `crypto.yml`

A file named `crypto.yml` should contain a section for each keystore you wish to make available within your application.

Each key store defined within `crypto.yml` must provide a path to the store, along with the password required to access the store.

Example 8.1. Example `crypto.yml`

```
truststore:
  store: truststore.jks
  password: foobar

keystore:
  store: keystore.jks
  password: foobar
```

Each keystore is labelled with an identifier used to access it from other services. While any number of key stores may be configured with arbitrary identifiers, many services look for specific key stores by default.

Some services, such as SOAP Endpoints attempt to use a key store named `truststore` if inbound signature verification is enabled. Likewise, the key store named `keystore` is the default for signing outbound responses.

If the path to the keystore in the `store` parameter is an absolute path, it will be used as specified. If the path is relative, the application framework is responsible for defining those semantics.

8.2.1. Ruby-on-Rails and `crypto.yml`

To use cryptographic key stores from your Ruby-on-Rails application, place `crypto.yml` in your application's `config/` directory. Relative paths to key stores specified in the `store` parameter are relative to the directory `$RAILS_ROOT/auth/`.

```
RAILS_ROOT/
  config/
    crypto.yml
  auth/
```

```
truststore.jks
keystore.jks
```

8.3. Managing Key Stores

Key stores are kept in the Java Key Store format and managed using the `keytool` command-line utility that ships with the JDK.

8.3.1. Create a New Key Store

When you generate or import your first key into the key store, it will be created as needed. The path to the key store is specified using the `-keystore` option.

```
$ keytool -keystore auth/keystore.jks
```

When a key store is first created while importing or generating a new key, you will be prompted to provide and verify the password to protect the entire keystore. This is the same password you provide through `crypto.yml`.

```
$ keytool -keystore auth/keystore.jks
Enter keystore password:
Re-enter new password:
```

8.3.2. Create a New Key Pair

To create a new key pair to use for signing or encrypting functions, the `-genkey` option is used. The `keytool` utility will prompt you for several pieces of information. Each key is identified by an *alias*. By default `-genkey` attempts to create a key named `mykey`. The `-alias` option may be used to provide a different alias to register the key.

```
$ keytool -keystore truststore.jks -genkey -alias bobmcwhirter
Enter keystore password:
What is your first and last name?
  [Unknown]:  Bob McWhirter
What is the name of your organizational unit?
  [Unknown]:  TorqueBox
What is the name of your organization?
  [Unknown]:  JBoss
What is the name of your City or Locality?
  [Unknown]:  Wytheville
What is the name of your State or Province?
  [Unknown]:  Virginia
What is the two-letter country code for this unit?
  [Unknown]:  US
Is CN=Bob McWhirter, OU=TorqueBox, O=JBoss, L=Wytheville, ST=Virginia, C=US
correct?
  [no]:  yes
```

```
Enter key password for <bobmcwhirter>
    (RETURN if same as keystore password):
Re-enter new password:
$
```

8.3.3. Inspect the Key Store

The `-list` option may be used to list the contents of a key store.

```
$ keytool -keystore truststore.jks -list
Enter keystore password:

Keystore type: JKS
Keystore provider: SUN

Your keystore contains 2 entries

bobmcwhirter, May 13, 2009, PrivateKeyEntry,
Certificate fingerprint (MD5):
 E1:73:2B:34:26:80:CA:55:7C:E8:BC:C6:A2:F6:D0:10
mykey, May 13, 2009, PrivateKeyEntry,
Certificate fingerprint (MD5):
 24:3E:29:E2:6A:5F:AA:24:A2:F3:25:68:B6:6E:92:FF
```

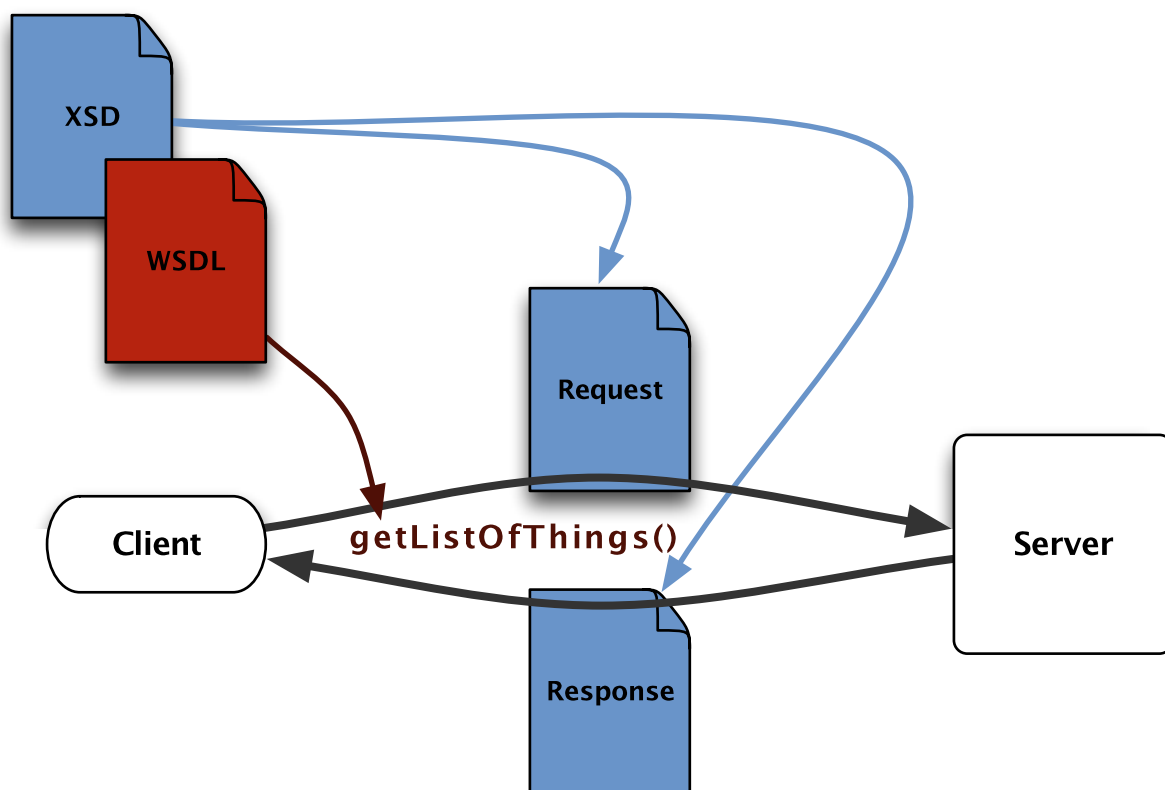
SOAP Endpoints

9.1. Introduction to SOAP & Data-Binding

Simple Object Access Protocol, or SOAP, is one component of many service-oriented architecture projects. A SOAP service is defined using a *Web Services Description Language*, or WSDL document. The WSDL document is consumed by both the server and clients as the contract of communication.

Clients invoke operations against the server, optionally involving cryptographic signatures or encryption. Servers respond to the operations synchronously, and produce a response document.

Both the request and response documents are defined using XML Schema Definition language (XSD). Advanced SOAP frameworks use these document descriptions to provide language-native *data-bindings* to facility reading and writing these documents.



9.2. Ruby Endpoints

Ruby endpoints may only be used if you already have (or are willing to write) a WSDL service definition document. The Ruby endpoints functionality will not generate a WSDL by inspecting your class. If you do not already have a WSDL, we suggest you take a RESTful strategy for web-services.

9.2.1. Basic Implementation & Configuration

Each SOAP service is implemented as a single Ruby class. Each operation provided by the service maps to a similarly-named method on the class.

In order to make SOAP service development easier, include the `TorqueBox::Endpoints::Base` module in your implementation.

Example 9.1. Basic SOAP implementation `amazon/ec2_endpoint.rb`

```
require 'torquebox/endpoints/base'

module Amazon
  class Ec2Endpoint

    include TorqueBox::Endpoints::Base

    # method per operation

  end
end
```

9.2.1.1. Endpoints with Ruby-on-Rails: `$RAILS_ROOT/app/endpoints/`

In Ruby-on-Rails applications, the `$RAILS_ROOT/app/endpoints/` directory should contain your WSDL and your Ruby implementation classes.

9.2.1.2. Configuration

The implementation needs to be configured to link up the services it provides with particular WSDL definition and the operations contained within it.

If the WSDL is present in the same directory, and is named the same as the class, minus the `_endpoint` suffix, it will implicitly be used. For example, `amazon/ec2.wsdl` would be used for `amazon/ec2_endpoint.rb` unless otherwise specified using the `wSDL_location` configuration parameter.

Other configuration parameters include `target_namespace`, `port_name`, and `security`. Configuration is accomplished at the class level using an `endpoint_configuration` block.

Example 9.2. Service configuration of a Ruby Endpoint

```
require 'torquebox/endpoints/base'

module Amazon
  class Ec2Endpoint
```

```
include TorqueBox::Endpoints::Base

endpoint_configuration do
  wsdl_location      'public/system/wsdl/ec2.wsdl'
  target_namespace  'http://ec2.amazonaws.com/doc/2008-12-01/'
  port_name          'AmazonEC2'
end

# method per operation

end
end
```

Security on the service may be configured using a `security` block within the `endpoint_configuration` block. The security block is broken into two sections: `inbound` and `outbound`.

Example 9.3. Security configuration of a Ruby Endpoint

```
require 'torquebox/endpoints/base'

module Amazon
  class Ec2Endpoint

    include TorqueBox::Endpoints::Base

    endpoint_configuration do
      wsdl_location      'public/system/wsdl/ec2.wsdl'
      target_namespace  'http://ec2.amazonaws.com/doc/2008-12-01/'
      port_name          'AmazonEC2'

      security do
        inbound do
          verify_timestamp
          verify_signature
        end
      end
    end

    # method per operation

  end
end
```



Note

Changes to the `endpoint_configuration` block are never re-parsed after your application is first deployed. A redeployment will be required to enable or disable security, or to alter the WSDL location or service bindings.

9.2.1.2.1. Inbound Security

Inbound security supports the follow directives:

- `verify_timestamp` to verify that the message is timely, and not a potential reply-attach in progress.
- `verify_signature` to verify the cryptographic signatures of inbound requests.

When verifying signatures, the security conduit will use the *cryptographic key store* named `truststore` by default. To use a different trust store, use the `truststore` option, and provide the identifier to a configured trust store.

```
security do
  inbound do
    truststore :my_other_truststore
    verify_timestamp
    verify_signature
  end
end
```

9.2.1.2.2. Outbound Security

Outbound security is currently unsupported.

9.2.2. Operation Implementation

Each operation defined by the WSDL may be mapped to a method on the implementation class. Appropriate conversion of `camelCaseNames` to `underscored_names` is performed. For example, the operation `DescribeInstances` would become the method `describe_instances()`.

This fragment of WSDL:

```
<operation name="DescribeInstances">
  <soap:operation soapAction="DescribeInstances" />
  ...
</operation>
```

would map to this method in the Ruby endpoint implementation class:

```
def describe_instances
  # implementation goes here
end
```

Within each operation method, the `request` method provides access to the data-bound request document. See [Data-Binding](#) for more information about how data-binding operates. The return value from the method converted to the appropriate SOAP response document and returned to the caller.

To create an appropriate response object, the `create_response` factory method is available. The object returned matches the default response type for the called operation.

Additionally, if security on inbound messages is enabled, and signature on the request has been verified, the `principal` method will provide access to the X.509 identity of the caller.

```
def describe_instances

  response = create_response

  request.instancesSet.each do |instance_id|
    log.info( "requesting information about instance #{instance_id} by
  #{principal}" )
    reservation_info = response.reservationSet.create
    reservation_info.ownerId = ...
  end

  return response

end
```



Caution

The entire design of the operation methods may be subject to change, pending input from the community. This is a tentative design only.

9.3. Data-Binding

Data-binding is the facility to map a data structure defined by XML Schema Description language into reasonable language-natural class definitions. It's much easier working with object trees with attributes than attempting to walk and produce XML documents.

When the WSDL is deployed by TorqueBox, the schema within is analyzed and Ruby classes are generated to support data-binding. This allows the Ruby endpoint implementation class to work purely in terms of *objects* instead of *documents*.

9.3.1. Basics

In general, XSD describes a straight-forward tree or graph of objects. Each type becomes a Ruby type, while each element becomes an attribute of some parent object. Primitive types, such as strings, booleans, or numerics are mapped to native Ruby types and vice-versa.

9.3.2. Collections

For XSD types that act purely as collections, containing repetition of a single element, the native Ruby Array is extended to provide intuitive access to the elements.

For example, this schema defines a DescribeKeyPairs type, which is seen as a <DescribeKeyPairs> element.

```
<xs:element name="DescribeKeyPairs" type="tns:DescribeKeyPairsType" />

<xs:complexType name="DescribeKeyPairsType">
  <xs:sequence>
    <xs:element name="keySet" type="tns:DescribeKeyPairsInfoType" />
  </xs:sequence>
</xs:complexType>

<xs:complexType name="DescribeKeyPairsInfoType">
  <xs:sequence>
    <xs:element name="item" type="tns:DescribeKeyPairsItemType"
      minOccurs="0" maxOccurs="unbounded" />
  </xs:sequence>
</xs:complexType>

<xs:complexType name="DescribeKeyPairsItemType">
  <xs:sequence>
    <xs:element name="keyName" type="xs:string" />
  </xs:sequence>
</xs:complexType>
```

And example XML document matching this would look like the following.

```
<DescribeKeyPairs>
  <keySet>
    <item>
      <keyName>key-1</keyName>
    </item>
    <item>
      <keyName>key-2</keyName>
    </item>
  </keySet>
</DescribeKeyPairs>
```

Using intelligent collection data-binding, the item element is not transliterated into the Ruby class. The <keySet> element translates into `Array` of `DescribeKeyPairsItemType` objects.

Navigating from a root `DescribeKeyPairsType` object to the key names works intuitively.

```
root.keySet.each do |item|
  puts "Describe key #{item.key}"
end
```


Telecom (SIP and Media)

Telecom support allows to build powerful converged VoIP applications by providing means for an application to handle/send SIP Messages. By including a [JSR 309](http://jcp.org/en/jsr/detail?id=309) (Media Server Control API) implementation, such as the [Mobicents JSR 309 implementation](http://www.mobicents.org/mms-jsr309-main.html) in your application, you can also control a Media Server remotely and so implement IVR, PBX, Call centers, Conference kind of applications.

Note that Mobicents Sip Servlets 1.0 is needed to be able to use this feature and that it is highly recommended to have knowledge about SIP ([SIP Servlets 1.1 Java Specification](http://jcp.org/en/jsr/detail?id=289)) and Media ([Media Server Control API Java Specification](http://jcp.org/en/jsr/detail?id=309)) to use it effectively.

10.1. Making phone calls

To be able to initiate the setup of a call from your TorqueBox application here is what you need to do in your controller :

Example 10.1. Code Excerpt to set up a call

```
# get the sip factory from the servlet context
@sip_factory = $servlet_context.get_attribute('javax.servlet.sip.SipFactory')
puts @sip_factory

# create a new sip application session
@app_session = request.env['servlet_request'].get_session().get_application_session();

# create a new sip servlet request to start a call to the sip phone with from header equals
to "sip:my_jruby_app_rocks@mobicents.org" and the to header equals to the sip_uri from the
complaint variable
@sip_request      =      @sip_factory.create_request(@app_session,      'INVITE',
'sip:my_jruby_app_rocks@mobicents.org', @complaint.sip_uri);

# actually sending the request out to the sip phone
@sip_request.send();
```

This piece of code will create a SIP Request and send it out. But when the user will answer OK, you need to have some code to handle the response and this can be done by defining your own SIP controller

10.2. Handling phone calls (Ruby Telco Classes)

In your Ruby or Rails application, all SIP classes should be placed under `$RAILS_ROOT/app/sip/`. No special naming convention is required, but the class name must match the path to the file containing it.

File name	Class name
<code>\$RAILS_ROOT/app/sip/telco_handler</code>	<code>SipHandler</code>
<code>\$RAILS_ROOT/app/sip/telco/handler</code>	<code>Telco::Handler</code>

Additionally, each Telco class must descend, at some point, from `TorqueBox::Sip::Base`.

Example 10.2. SIP Telco class example (`telco/handler.rb`)

```
require 'torquebox/sip/base'
  module Telco
  class Handler < TorqueBox::Sip::Base
  # Handle INVITE request to setup a call by answering 200 OK
  def do_invite(request)
    request.create_response(200).send
  end

  # Handle BYE request to tear down a call by answering 200 OK
  def do_bye(request)
    request.create_response(200).send
  end

  # Handle REGISTER request so that a SIP Phone can register with the application by answering
  200 OK
  def do_register(request)
    request.create_response(200).send
  end

  # Handle a successful response to an application initiated INVITE to set up a call (when a new
  complaint is filed through the web part) by send an acknowledgment
  def do_success_response(response)
    response.create_ack.send
  end
  end
  end
end
```

Note that this is an example but the handler class can extend all the `do_XXX` methods as defined per the Sip Servlets 1.1 Java Specification.

To help you getting started with you can play with and checkout the code of our [sample application demonstrating Announcement and DTMF Detection](http://www.mobicens.org/mss-pure-jruby-telco.html) [http://www.mobicens.org/mss-pure-jruby-telco.html]

10.3. Media Support

Mobicens Sip Servlets (and its bundled Media Server) and Torquebox allows you to include Media in your applications as well now.

This allows you to control a remote Media Server using the MGCP protocol and let you create Telco Applications such as IVR, Conferences, Announcement, DTMF recognition, Call Centers, PBX, ...

To be able to use the Mobicens JSR 309 implementation, you may want to create a maven pom.xml at the root directory of your JRuby application containing the following :

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-
v4_0_0.xsd">
<modelVersion>4.0.0</modelVersion>
<parent>
<groupId>org.test</groupId>
<artifactId>torquebox-media-example</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>jar</packaging>
<name>Torquebox Media Demo Application</name>
<url>http://www.mobicens.org/mss-pure-jruby-telco.html</url>

<dependencies>
<!-- media dependencies -->
<dependency>
<groupId>org.mobicens.external.jsr309</groupId>
<artifactId>mscontrol</artifactId>
<version>0.3</version>
</dependency>

<dependency>
<groupId>org.mobicens.jsr309</groupId>
<artifactId>mobicens-jsr309-impl</artifactId>
<version>2.0.0.BETA2</version>
<scope>runtime</scope>
</dependency>

<dependency>
```

```
<groupId>org.mobicents.servlet.sip</groupId>
<artifactId>sip-servlets-spec</artifactId>
<version>1.1.11-SNAPSHOT</version>
</dependency>
</dependencies>

<build>
<plugins>
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-dependency-plugin</artifactId>
<executions>
<execution>
<id>copy-dependencies</id>
<phase>package</phase>
<goals>
<goal>copy-dependencies</goal>
</goals>
<configuration>
<outputDirectory>/opt/torquebox/builder/torquebox/docs/en-US/lib/java</outputDirectory>
<includeScope>runtime</includeScope>
</configuration>
</execution>
</executions>
</plugin>
</plugins>
</build>

<repositories>
<repository>
<id>JbossRepository</id>
<name>Jboss Repository</name>
<url>http://repository.jboss.org/maven2</url>
<snapshots>
<enabled>>true</enabled>
</snapshots>
<releases>
<enabled>>true</enabled>
</releases>
</repository>
</repositories>
</project>
```

and type the following command in a shell

```
mvn clean install
```

or copy the dependencies mentioned in the maven pom above from the jboss repository located at <http://repository.jboss.org/maven2> in the following directory of their JRuby application *lib/java*

The next step is to start using the JSR 309 API to control the Media Server and build Media applications, you can read the specification at this address : <http://jcp.org/en/jsr/detail?id=309>

To help you getting started with you can play with and checkout the code of our [sample application demonstrating Announcement and DTMF Detection](http://www.mobicens.org/mss-pure-jruby-telco.html) [<http://www.mobicens.org/mss-pure-jruby-telco.html>]

Capistrano Support

11.1. Capistrano

Capistrano is a deployment tool to assist in moving code from a repository to a production server.

11.2. App-Server separate from App

The TorqueBox server in production is a separate entity from the applications being deployed upon it. When deploying with mongrel or other traditional Ruby solutions, part of the deployment normally includes restarting the server. TorqueBox does not need to be restarted nearly as often.

Generally the TorqueBox server is installed in a system-wide location since it may support multiple applications simultaneously. For install, you may simply unpackage and install it under `/opt/torquebox-server`.

The TorqueBox server can be managed through a normal `init.d/` script, or through Daniel Bernstein's `daemontools`. The Capistrano support provided by TorqueBox can work with either process for managing the server, when necessary.

Either way, two settings in your application's `deploy.rb` are required to be able to deploy to a remote TorqueBox server.

```
set :jboss_home,      "/opt/torquebox-server/jboss"  
set :jboss_config,   :default
```

These should point to the same location as `$JBOSS_HOME` on the server. The `:jboss_config` matches the `$JBOSS_CONF`, and is only required if a configuration other than default is desired.



Note

Capistrano support is still evolving and will be fully documented once it settles.

Appendix A. GNU Lesser General Public License version 3

Version 3, 29 June 2007

Copyright 2009 JBoss Middleware, a division of Red Hat, Inc. <http://jboss.com/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

This version of the GNU Lesser General Public License incorporates the terms and conditions of version 3 of the GNU General Public License, supplemented by the additional permissions listed below.

0. Additional Definitions.

As used herein, this License refers to version 3 of the GNU Lesser General Public License, and the GNU GPL refers to version 3 of the GNU General Public License.

The Library refers to a covered work governed by this License, other than an Application or a Combined Work as defined below.

An Application is any work that makes use of an interface provided by the Library, but which is not otherwise based on the Library. Defining a subclass of a class defined by the Library is deemed a mode of using an interface provided by the Library.

A Combined Work is a work produced by combining or linking an Application with the Library. The particular version of the Library with which the Combined Work was made is also called the Linked Version.

The Minimal Corresponding Source for a Combined Work means the Corresponding Source for the Combined Work, excluding any source code for portions of the Combined Work that, considered in isolation, are based on the Application, and not on the Linked Version.

The Corresponding Application Code for a Combined Work means the object code and/or source code for the Application, including any data and utility programs needed for reproducing the Combined Work from the Application, but excluding the System Libraries of the Combined Work.

1. Exception to Section 3 of the GNU GPL.

You may convey a covered work under sections 3 and 4 of this License without being bound by section 3 of the GNU GPL.

2. Conveying Modified Versions.

If you modify a copy of the Library, and, in your modifications, a facility refers to a function or data to be supplied by an Application that uses the facility (other than as an argument passed when the facility is invoked), then you may convey a copy of the modified version:

- a. under this License, provided that you make a good faith effort to ensure that, in the event an Application does not supply the function or data, the facility still operates, and performs whatever part of its purpose remains meaningful, or
- b. under the GNU GPL, with none of the additional permissions of this License applicable to that copy.

3. Object Code Incorporating Material from Library Header Files.

The object code form of an Application may incorporate material from a header file that is part of the Library. You may convey such object code under terms of your choice, provided that, if the incorporated material is not limited to numerical parameters, data structure layouts and accessors, or small macros, inline functions and templates (ten or fewer lines in length), you do both of the following:

- a. Give prominent notice with each copy of the object code that the Library is used in it and that the Library and its use are covered by this License.
- b. Accompany the object code with a copy of the GNU GPL and this license document.

4. Combined Works.

You may convey a Combined Work under terms of your choice that, taken together, effectively do not restrict modification of the portions of the Library contained in the Combined Work and reverse engineering for debugging such modifications, if you also do each of the following:

- a. Give prominent notice with each copy of the Combined Work that the Library is used in it and that the Library and its use are covered by this License.
- b. Accompany the Combined Work with a copy of the GNU GPL and this license document.
- c. For a Combined Work that displays copyright notices during execution, include the copyright notice for the Library among these notices, as well as a reference directing the user to the copies of the GNU GPL and this license document.
- d. Do one of the following:

1. Convey the Minimal Corresponding Source under the terms of this License, and the Corresponding Application Code in a form suitable for, and under terms that permit, the user to recombine or relink the Application with a modified version of the Linked Version to produce a modified Combined Work, in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.
 2. Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (a) uses at run time a copy of the Library already present on the users computer system, and (b) will operate properly with a modified version of the Library that is interface-compatible with the Linked Version.
- e. Provide Installation Information, but only if you would otherwise be required to provide such information under section 6 of the GNU GPL, and only to the extent that such information is necessary to install and execute a modified version of the Combined Work produced by recombining or relinking the Application with a modified version of the Linked Version. (If you use option 4d0, the Installation Information must accompany the Minimal Corresponding Source and Corresponding Application Code. If you use option 4d1, you must provide the Installation Information in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.)

5. Combined Libraries.

You may place library facilities that are a work based on the Library side by side in a single library together with other library facilities that are not Applications and are not covered by this License, and convey such a combined library under terms of your choice, if you do both of the following:

- a. Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities, conveyed under the terms of this License.
- b. Give prominent notice with the combined library that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

6. Revised Versions of the GNU Lesser General Public License.

The Free Software Foundation may publish revised and/or new versions of the GNU Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library as you received it specifies that a certain numbered version of the GNU Lesser General Public License or any later version applies to it, you have the option of following the terms and conditions either of that published version or of any later version published by the Free Software Foundation. If the Library as you received it does not specify a version number of the GNU Lesser General Public License, you

may choose any version of the GNU Lesser General Public License ever published by the Free Software Foundation.

If the Library as you received it specifies that a proxy can decide whether future versions of the GNU Lesser General Public License shall apply, that proxy's public statement of acceptance of any version is permanent authorization for you to choose that version for the Library.

Appendix B. MIT License

Copyright 2009 JBoss Middleware, a division of Red Hat, Inc. <http://jboss.com/>

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

