HornetQ REST Interface

2.2.5.Final

Preface

Commercial development support, production support and training for RESTEasy and HornetQ is available through JBoss, a division of Red Hat Inc. (see http://www.jboss.com/).

Introduction

The HornetQ REST interface allows you to leverage the reliability and scalability features of HornetQ over a simple REST/HTTP interface. Messages are produced and consumed by sending and receiving simple HTTP messages that contain the content you want to push around. For instance, here's a simple example of posting an order to an order processing queue express as an HTTP message:

```
POST /queue/orders/create HTTP/1.1
Host: example.com
Content-Type: application/xml

<order>
  <name>Bill</name>
  <item>iPhone 4</item>
  <cost>$199.99</cost>
</order>
```

As you can see, we're just posting some arbitrary XML document to a URL. When the XML is received on the server is it processed within HornetQ as a JMS message and distributed through core HornetQ. Simple and easy. Consuming messages from a queue or topic looks very similar. We'll discuss the entire interface in detail later in this docbook.

1.1. Goals of REST Interface

Why would you want to use HornetQ's REST interface? What are the goals of the REST interface?

- Easily usable by machine-based (code) clients.
- Zero client footprint. We want HornetQ to be usable by any client/programming language that has an adequate HTTP client library. You shouldn't have to download, install, and configure a special library to interact with HornetQ.
- Lightweight interoperability. The HTTP protocol is strong enough to be our message exchange protocol. Since interactions are RESTful the HTTP uniform interface provides all the interoperability you need to communicate between different languages, platforms, and even messaging implementations that choose to implement the same RESTful interface as HornetQ (i.e. the *REST-** [http://rest-star.org] effort.)
- No envelope (i.e. SOAP) or feed (i.e. Atom) format requirements. You shouldn't have to learn, use, or parse a specific XML document format in order to send and receive messages through HornetQ's REST interface.

- Leverage the reliability, scalability, and clustering features of HornetQ on the back end without sacrificing the simplicity of a REST interface.

Installation and Configuration

HornetQ's REST interface is installed as a Web archive (WAR). It depends on the *RESTEasy* [http://jboss.org/resteasy] project and can currently only run within a servlet container. Installing the HornetQ REST interface is a little bit different depending whether HornetQ is already installed and configured for your environment (i.e. you're deploying within JBoss 6 AppServer) or you want the HornetQ REST WAR to startup and manage the HornetQ server.

2.1. Installing Within Pre-configured Environment

The section should be used when you want to use the HornetQ REST interface in an environment that already has HornetQ installed and running, i.e. JBoss 6 Application Server. You must create a Web archive (.WAR) file with the following web.xml settings:

```
<web-app>
   <listener>
     <listener-class>org.jboss.resteasy.plugins.server.servlet.ResteasyBootstrap</listener-class>
   </listener>

   <listener>
           <listener-class>org.hornetq.rest.integration.RestMessagingBootstrapListener</listener-
class>
   </listener>

   <filter>
      <filter-name>Rest-Messaging</filter-name>
      <filter-class>
         org.jboss.resteasy.plugins.server.servlet.FilterDispatcher
      </filter-class>
   </filter>

   <filter-mapping>
      <filter-name>Rest-Messaging</filter-name>
      <url-pattern>/*</url-pattern>
   </filter-mapping>

</web-app
```

Within your WEB-INF/lib directory you must have the hornetq-rest.jar file. If RESTEasy is not installed within your environment, you must add the RESTEasy jar files within the lib directory as well. Here's a sample Maven pom.xml that can build your WAR for this case.

```
<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>
   <groupId>org.somebody</groupId>
   <artifactId>myapp</artifactId>
   <packaging>war</packaging>
   <name>My App</name>
   <repositories>
      <repository>
         <id>jboss</id>
         <url>http://repository.jboss.org/nexus/content/groups/public/</url>
      </repository>
   </repositories>


    <build>
        <plugin>
           <groupId>org.apache.maven.plugins</groupId>
           <artifactId>maven-compiler-plugin</artifactId>
           <configuration>
              <source>1.6</source>
              <target>1.6</target>
           </configuration>
        </plugin>
      </plugins>
   </build>
   <dependencies>
      <dependency>
         <groupId>org.hornetq.rest</groupId>
         <artifactId>hornetq-rest</artifactId>
         <version>2.2.5.Final</version>
      </dependency>
   </dependencies>
</project>
```

2.2. Bootstrapping HornetQ Along with REST

You can bootstrap HornetQ within your WAR as well. To do this, you must have the HornetQ core and JMS jars along with Netty, Resteasy, and the HornetQ REST jar within your WEB-INF/lib. You must also have a hornetq-configuration.xml, hornetq-jms.xml, and hornetq-users.xml config

files within WEB-INF/classes. The examples that come with the HornetQ REST distribution show how to do this. You must also add an additional listener to your web.xml file. Here's an example:

```
<web-app>
   <listener>
     <listener-class>org.jboss.resteasy.plugins.server.servlet.ResteasyBootstrap</listener-class>
   </listener>


   <listener>
      <listener-class>org.hornetq.rest.integration.HornetqBootstrapListener</listener-class>
   </listener>

   <listener>
           <listener-class>org.hornetq.rest.integration.RestMessagingBootstrapListener</listener-
class>
   </listener>

   <filter>
      <filter-name>Rest-Messaging</filter-name>
      <filter-class>
         org.jboss.resteasy.plugins.server.servlet.FilterDispatcher
      </filter-class>
   </filter>

   <filter-mapping>
      <filter-name>Rest-Messaging</filter-name>
      <url-pattern>/*</url-pattern>
   </filter-mapping>

</web-app>
```

Here's a Maven pom.xml file for creating a WAR for this environment. Make sure your hornetq configuration files are within the src/main/resources directory so that they are stuffed within the WAR's WEB-INF/classes directory!

```
<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>
   <groupId>org.somebody</groupId>
```

```
<artifactId>myapp</artifactId>
<packaging>war</packaging>
<name>My App</name>
<repositories>
   <repository>
      <id>jboss</id>
      <url>http://repository.jboss.org/nexus/content/groups/public/</url>
   </repository>
</repositories>
<build>
      <plugin>
         <groupId>org.apache.maven.plugins</groupId>
         <artifactId>maven-compiler-plugin</artifactId>
         <configuration>
            <source>1.6</source>
            <target>1.6</target>
         </configuration>
      </plugin>
   </plugins>
</build>
<dependencies>
   <dependency>
      <groupId>org.hornetq</groupId>
      <artifactId>hornetq-core</artifactId>
      <version>2.1.1.GA</version>
   </dependency>
   <dependency>
      <groupId>org.jboss.netty</groupId>
      <artifactId>netty</artifactId>
   </dependency>
   <dependency>
      <groupId>org.hornetq</groupId>
      <artifactId>hornetq-jms</artifactId>
      <version>2.1.1.GA</version>
   </dependency>
   <dependency>
      <groupId>org.jboss.spec.javax.jms</groupId>
      <artifactId>jboss-jms-api_1.1_spec</artifactId>
      <version>1.0.0.Beta1</version>
   </dependency>
   <dependency>
      <groupId>org.hornetq.rest</groupId>
      <artifactId>hornetq-rest</artifactId>
      <version>2.2.5.Final</version>
```

```
      </dependency>
      <dependency>
         <groupId>org.jboss.resteasy</groupId>
         <artifactId>resteasy-jaxrs</artifactId>
         <version>2.0.1.GA</version>
      </dependency>
      <dependency>
         <groupId>org.jboss.resteasy</groupId>
         <artifactId>resteasy-jaxb-provider</artifactId>
         <version>2.0.1.GA</version>
      </dependency>
   </dependencies>
</project>
```

2.3. REST Configuration

The HornetQ REST implementation does have some configuration options. These are configured via XML configuration file that must be in your WEB-INF/classes directory. You must set the web.xml context-param `rest.messaging.config.file` to specify the name of the configuration file. Below is the format of the XML configuration file and the default values for each.

```
<rest-messaging>
   <server-in-vm-id>0</server-in-vm-id>
   <use-link-headers>false</use-link-headers>
   <default-durable-send>false</default-durable-send>
   <dups-ok>true</dups-ok>
   <topic-push-store-dir>topic-push-store</topic-push-store-dir>
   <queue-push-store-dir>queue-push-store</queue-push-store-dir>
   <producer-session-pool-size>10</producer-session-pool-size>
   <session-timeout-task-interval>1</session-timeout-task-interval>
   <consumer-session-timeout-seconds>300</consumer-session-timeout-seconds>
   <consumer-window-size>-1</consumer-window-size>
</rest-messaging
```

Let's give an explanation of each config option.

server-in-vm-id
: The HornetQ REST impl uses the IN-VM transport to communicate with HornetQ. It uses the default server id, which is "0".

use-link-headers

By default, all links (URLs) are published using custom headers. You can instead have the HornetQ REST implementation publish links using the *Link Header specification* [http://tools.ietf.org/html/draft-nottingham-http-link-header-10] instead if you desire.

default-durable-send

Whether a posted message should be persisted by default if the user does not specify a durable query parameter.

dups-ok

If this is true, no duplicate detection protocol will be enforced for message posting.

topic-push-store-dir

This must be a relative or absolute file system path. This is a directory where push registrations for topics are stored. See Chapter 6.

queue-push-store-dir

This must be a relative or absolute file system path. This is a directory where push registrations for queues are stored. See Chapter 6.

producer-session-pool-size

The REST implementation pools HornetQ sessions for sending messages. This is the size of the pool. That number of sessions will be created at startup time.

session-timeout-task-interval

Pull consumers and pull subscriptions can time out. This is the interval the thread that checks for timed-out sessions will run at. A value of 1 means it will run every 1 second.

consumer-session-timeout-seconds

Timeout in seconds for pull consumers/subscriptions that remain idle for that amount of time.

consumer-window-size

For consumers, this config option is the same as the HornetQ one of the same name. It will be used by sessions created by the HornetQ REST implementation.

HornetQ REST Interface Basics

The HornetQ REST interface publishes a variety of REST resources to perform various tasks on a queue or topic. Only the top-level queue and topic URI schemes are published to the outside world. You must discover all over resources to interact with by looking for and traversing links. You'll find published links within custom response headers and embedded in published XML representations. Let's look at how this works.

3.1. Queue and Topic Resources

To interact with a queue or topic you do a HEAD or GET request on the following relative URI pattern:

```
/queues/{name}
/topics/{name}
```

The base of the URI is the base URL of the WAR you deployed the HornetQ REST server within as defined in the Installation and Configuration section of this document. Replace the `{name}` string within the above URI pattern with the name of the queue or topic you are interested in interacting with. For example if you have configured a JMS topic named "foo" within your `hornetq-jms.xml` file, the URI name should be "jms.topic.foo". If you have configured a JMS queue name "bar" within your `hornetq-jms.xml` file, the URI name should be "jms.queue.bar". Internally, HornetQ prepends the "jms.topic" or "jms.queue" strings to the name of the deployed destination. Next, perform your HEAD or GET request on this URI. Here's what a request/response would look like.

```
HEAD /queues/jms.queue.bar HTTP/1.1
Host: example.com

--- Response ---
HTTP/1.1 200 Ok
msg-create: http://example.com/queues/jms.queue.bar/create
msg-pull-consumers: http://example.com/queues/jms.queue.bar/pull-consumers
msg-push-consumers: http://example.com/queues/jms.queue.bar/push-consumers
```

The HEAD or GET response contains a number of custom response headers that are URLs to additional REST resources that allow you to interact with the queue or topic in different ways. It is important not to rely on the scheme of the URLs returned within these headers as they are an implementation detail. Treat them as opaque and query for them each and every time you initially interact (at boot time) with the server. If you treat all URLs as opaque then you will be isolated from implementation changes as the HornetQ REST interface evolves over time.

3.2. Queue Resource Response Headers

Below is a list of response headers you should expect when interacting with a Queue resource.

msg-create
: This is a URL you POST messages to. The semantics of this link are described in Chapter 4.

msg-create-with-id
: This is a URL template you POST message to. The semantics of this link are described in Chapter 4.

msg-pull-consumers
: This is a URL for creating consumers that will pull from a queue. The semantics of this link are described in Chapter 5.

msg-push-consumers
: This is a URL for registering other URLs you want the HornetQ REST server to push messages to. The semantics of this link are described in Chapter 6

3.3. Topic Resource Respones Headers

Below is a list of response headers you should expect when interacting with a Topic resource.

msg-create
: This is a URL you POST messages to. The semantics of this link are described in Chapter 4.

msg-create-with-id
: This is a URL template you POST messages to. The semantics of this link are described in Chapter 4.

msg-pull-subscriptions
: This is a URL for creating subscribers that will pull from a topic. The semantics of this link are described in Chapter 5.

msg-push-subscriptions
: This is a URL for registering other URLs you want the HornetQ REST server to push messages to. The semantics of this link are described in Chapter 6.

Posting Messages

This chapter discusses the protocol for posting messages to a queue or a topic. In Chapter 3, you saw that a queue or topic resource publishes variable custom headers that are links to other RESTful resources. The `msg-create` header is the URL you post messages to. Messages are published to a queue or topic by sending a simple HTTP message to the URL published by the msg-create header. The HTTP message contains whatever content you want to publish to the HornetQ destination. Here's an example scenario:

1. Obtain the starting `msg-create` header from the queue or topic resource.

```
HEAD /queues/jms.queue.bar HTTP/1.1
Host: example.com

--- Response ---
HTTP/1.1 200 Ok
msg-create: http://example.com/queues/jms.queue.bar/create
msg-create-with-id: http://example.com/queues/jms.queue.bar/create/{id}
```

2. Do a POST to the URL contained in the `msg-create` header.

```
POST /queues/jms.queue.bar/create
Host: example.com
Content-Type: application/xml

<order>
  <name>Bill</name>
  <item>iPhone4</name>
  <cost>$199.99</cost>
</order>

--- Response ---
HTTP/1.1 201 Created
msg-create-next: http://example.com/queues/jms.queue.bar/create/002
```

   A successful response will return a 201 response code. Also notice that a `msg-create-next` response header is sent as well. You must use this URL to POST your next message.

3. POST your next message to the queue using the URL returned in the `msg-create-next` header.

```
POST /queues/jms.queue.bar/create/002
Host: example.com
Content-Type: application/xml

<order>
  <name>Monica</name>
  <item>iPad</item>
  <cost>$499.99</cost>
</order>

--- Response --
HTTP/1.1 201 Created
msg-create-next: http://example.com/queues/jms.queue.bar/create/003
```

Continue using the new `msg-create-next` header returned with each response.

It is *VERY IMPORTENT* that you never re-use returned `msg-create-next` headers to post new messages. This URL may be uniquely generated for each message and used for duplicate detection. If you lose the URL within the `msg-create-next` header, then just go back to the queue or topic resource to get the msg-create URL.

4.1. Duplicate Detection

Sometimes you might have network problems when posting new messages to a queue or topic. You may do a POST and never receive a response. Unfortunately, you don't know whether or not the server received the message and so a re-post of the message might cause duplicates to be posted to the queue or topic. By default, the HornetQ REST interface is configured to accept and post duplicate messages. You can change this by turning on duplicate message detection by setting the `dups-ok` config option to `false` as described in Chapter 3. When you do this, the initial POST to the msg-create URL will redirect you, using the standard HTTP 307 redirection mechanism to a unique URL to POST to. All other interactions remain the same as discussed earlier. Here's an example:

1. Obtain the starting `msg-create` header from the queue or topic resource.

```
HEAD /queues/jms.queue.bar HTTP/1.1
Host: example.com

--- Response ---
HTTP/1.1 200 Ok
msg-create: http://example.com/queues/jms.queue.bar/create
msg-create-with-id: http://example.com/queues/jms.queue.bar/create/{id}
```

2. Do a POST to the URL contained in the `msg-create` header.

```
POST /queues/jms.queue.bar/create
Host: example.com
Content-Type: application/xml

<order>
  <name>Bill</name>
  <item>iPhone4</name>
  <cost>$199.99</cost>
</order>

--- Response ---
HTTP/1.1 307 Redirect
Location: http://example.com/queues/jms.queue.bar/create/001
```

   A successful response will return a 307 response code. This is standard HTTP protocol. It is telling you that you must re-POST to the URL contained within the `Location` header.

3. re-POST your message to the URL provided within the `Location` header.

```
POST /queues/jms.queue.bar/create/001
Host: example.com
Content-Type: application/xml

<order>
  <name>Bill</name>
  <item>iPhone4</name>
  <cost>$199.99</cost>
</order>

--- Response --
HTTP/1.1 201 Created
msg-create-next: http://example.com/queues/jms.queue.bar/create/002
```

   You should receive a 201 Created response. If there is a network failure, just re-POST to the Location header. For new messages, use the returned `msg-create-next` header returned with each response.

4. POST any new message to the returned `msg-create-next` header.

```
POST /queues/jms.queue.bar/create/002
```

```
Host: example.com
Content-Type: application/xml

<order>
  <name>Monica</name>
  <item>iPad</name>
  <cost>$499.99</cost>
</order>

--- Response --
HTTP/1.1 201 Created
msg-create-next: http://example.com/queues/jms.queue.bar/create/003
```

If there ever is a network problem, just repost to the URL provided in the `msg-create-next` header.

How can this work? As you can see, with each successful response, the HornetQ REST server returns a uniquely generated URL within the msg-create-next header. This URL is dedicated to the next new message you want to post. Behind the scenes, the code extracts an identify from the URL and uses HornetQ's duplicate detection mechanism by setting the `DUPLICATE_DETECTION_ID` property of the JMS message that is actually posted to the system.

An alternative to this approach is to use the `msg-create-with-id` header. This is not an invokable URL, but a URL template. The idea is that the client provides the `DUPLICATE_DETECTION_ID` and creates it's own `create-next` URL. The `msg-create-with-id` header looks like this (you've see it in previous examples, but we haven't used it):

```
msg-create-with-id: http://example.com/queues/jms.queue.bar/create/{id}
```

You see that it is a regular URL appended with a `{id}`. This `{id}` is a pattern matching substring. A client would generate its `DUPLICATE_DETECTION_ID` and replace `{id}` with that generated id, then POST to the new URL. The URL the client creates works exactly like a `create-next` URL described earlier. The response of this POST would also return a new `msg-create-next` header. The client can continue to generate its own DUPLICATE_DETECTION_ID, or use the new URL returned via the `msg-create-next` header.

The advantage of this approach is that the client does not have to repost the message. It also only has to come up with a unique `DUPLICATE_DETECTION_ID` once.

4.2. Persistent Messages

By default, posted messages are not durable and will not be persisted in HornetQ's journal. You can create durable messages by modifying the default configuration as expressed in Chapter 2 so that all messages are persisted when sent. Alternatively, you can set a URL query parameter

called `durable` to true when you post your messages to the URLs returned in the `msg-create`, `msg-create-with-id`, or `msg-create-next` headers. here's an example of that.

```
POST /queues/jms.queue.bar/create?durable=true
Host: example.com
Content-Type: application/xml

<order>
  <name>Bill</name>
  <item>iPhone4</item>
  <cost>$199.99</cost>
</order>
```

4.3. Expiration and Priority

You can set he expiration and the priority of the message in the queue or topic by setting an additional query parameter. The `expiration` query parameter is an integer expressing the time in milliseconds until the message should be expired. The `priority` is another query parameter with an integer value between 0 and 9 expressing the priority of the message. i.e.:

```
POST /queues/jms.queue.bar/create?expiration=30000&priority=3
Host: example.com
Content-Type: application/xml

<order>
  <name>Bill</name>
  <item>iPhone4</item>
  <cost>$199.99</cost>
</order>
```

Consuming Messages via Pull

There are two different ways to consume messages from a topic or queue. You can wait and have the messaging server push them to you, or you can continuously poll the server yourself to see if messages are available. This chapter discusses the latter. Consuming messages via a pull works almost identically for queues and topics with some minor, but important caveats. To start consuming you must create a consumer resource on the server that is dedicated to your client. Now, this pretty much breaks the stateless principle of REST, but after much prototyping, this is the best way to work most effectively with HornetQ through a REST interface.

You create consumer resources by doing a simple POST to the URL published by the `msg-pull-consumers` response header if you're interacting with a queue, the `msg-pull-subscribers` response header if you're interacting with a topic. These headers are provided by the main queue or topic resource discussed in Chapter 3. Doing an empty POST to one of these URLs will create a consumer resource that follows an auto-acknowledge protocol and, if you're interacting with a topic, creates a temporty subscription to the topic. If you want to use the acknowledgement protocol and/or create a durable subscription (topics only), then you must use the form parameters (`application/x-www-form-urlencoded`) described below.

autoAck
: A value of `true` or `false` can be given. This defaults to `true` if you do not pass this parameter.

durable
: A value of `true` or `false` can be given. This defaults to `false` if you do not pass this parameter. Only available on topics. This specifies whether you want a durable subscription or not. A durable subscription persists through server restart.

name
: This is the name of the durable subscription. If you do not provide this parameter, the name will be automatically generated by the server. Only usable on topics.

selector
: This is an optional JMS selector string. The HornetQ REST interface adds HTTP headers to the JMS message for REST produced messages. HTTP headers are prefixed with "http_" and every '-' charactor is converted to a '$'.

5.1. Auto-Acknowledge

This section focuses on the auto-acknowledge protocol for consuming messages via a pull. Here's a list of the response headers and URLs you'll be interested in.

msg-pull-consumers
: The URL of a factory resource for creating queue consumer resources. You will pull from these created resources.

msg-pull-subscriptions
: The URL of a factory resource for creating topic subscription resources. You will pull from the created resources.

msg-consume-next
: The URL you will pull the next message from. This is returned with every response.

msg-consumer
: This is a URL pointing back to the consumer or subscription resource created for the client.

5.1.1. Creating an Auto-Ack Consumer or Subscription

Here is an example of creating an auto-acknowledged queue pull consumer.

1. Find the pull-consumers URL by doing a HEAD or GET request to the base queue resource.

```
HEAD /queues/jms.queue.bar HTTP/1.1
Host: example.com

--- Response ---
HTTP/1.1 200 Ok
msg-create: http://example.com/queues/jms.queue.bar/create
msg-pull-consumers: http://example.com/queues/jms.queue.bar/pull-consumers
msg-push-consumers: http://example.com/queues/jms.queue.bar/push-consumers
```

2. Next do an empty POST to the URL returned in the `msg-pull-consumers` header.

```
POST /queues/jms.queue.bar/pull-consumers HTTP/1.1
Host: example.com

--- response ---
HTTP/1.1 201 Created
Location: http://example.com/queues/jms.queue.bar/pull-consumers/auto-ack/333
msg-consume-next:  http://example.com/queues/jms.queue.bar/pull-consumers/auto-ack/333/
consume-next-1
```

   The `Location` header points to the JMS consumer resource that was created on the server. It is good to remember this URL, although, as you'll see later, it is transmitted with each response just to remind you.

Creating an auto-acknowledged consumer for a topic is pretty much the same. Here's an example of creating a durable auto-acknowledged topic pull subscription.

1. Find the `pull-subscriptions` URL by doing a HEAD or GET request to the base topic resource

```
HEAD /topics/jms.topic.bar HTTP/1.1
Host: example.com

--- Response ---
HTTP/1.1 200 Ok
msg-create: http://example.com/topics/jms.topic.foo/create
msg-pull-subscriptions: http://example.com/topics/jms.topic.foo/pull-subscriptions
msg-push-subscriptions: http://example.com/topics/jms.topic.foo/push-subscriptions
```

2. Next do a POST to the URL returned in the `msg-pull-subscriptions` header passing in a `true` value for the `durable` form parameter.

```
POST /topics/jms.topic.foo/pull-subscriptions HTTP/1.1
Host: example.com
Content-Type: application/x-www-form-urlencoded

durable=true

--- Response ---
HTTP/1.1 201 Created
Location: http://example.com/topics/jms.topic.foo/pull-subscriptions/auto-ack/222
msg-consume-next:  http://example.com/topics/jms.topic.foo/pull-subscriptions/auto-ack/222/consume-next-1
```

The `Location` header points to the JMS subscription resource that was created on the server. It is good to remember this URL, although, as you'll see later, it is transmitted with each response just to remind you.

5.1.2. Consuming Messages

After you have created a consumer resource, you are ready to start pulling messages from the server. Notice that when you created the consumer for either the queue or topic, the response contained a `msg-consume-next` response header. POST to the URL contained within this header to consume the next message in the queue or topic subscription. A successful POST causes the server to extract a message from the queue or topic subscription, acknowledge it, and return it to the consuming client. If there are no messages in the queue or topic subscription, a 503 (Service Unavailable) HTTP code is returned.

Warning

For both successful and unsuccessful posts to the msg-consume-next URL, the response will contain a new msg-consume-next header. You must ALWAYS use this new URL returned within the new msg-consume-next header to consume new messages.

Here's an example of pulling multiple messages from the consumer resource.

1. Do a POST on the msg-consume-next URL that was returned with the consumer or subscription resource discussed earlier.

```
POST /queues/jms.queue.bar/pull-consumers/consume-next-1
Host: example.com

--- Response ---
HTTP/1.1 200 Ok
Content-Type: application/xml
msg-consume-next: http://example.com/queues/jms.queue.bar/pull-consumers/333/consume-next-2
msg-consumer: http://example.com/queues/jms.queue.bar/pull-consumers/333

<order>...</order>
```

   The POST returns the message consumed from the queue. It also returns a new msg-consume-next link. Use this new link to get the next message. Notice also a msg-consumer response header is returned. This is a URL that points back to the consumer or subscription resource. You will need that to clean up your connection after you are finished using the queue or topic.

2. The POST returns the message consumed from the queue. It also returns a new msg-consume-next link. Use this new link to get the next message.

```
POST /queues/jms.queue.bar/pull-consumers/consume-next-2
Host: example.com

--- Response ---
Http/1.1 503 Service Unavailable
Retry-After: 5
msg-consume-next: http://example.com/queues/jms.queue.bar/pull-consumers/333/consume-next-2
```

In this case, there are no messages in the queue, so we get a 503 response back. As per the HTTP 1.1 spec, a 503 response may return a Retry-After head specifying the time in seconds that you should retry a post. Also notice, that another new msg-consume-next URL is present. Although it probabley is the same URL you used last post, get in the habit of using URLs returned in response headers as future versions of HornetQ REST might be redirecting you or adding additional data to the URL after timeouts like this.

3. POST to the URL within the last `msg-consume-next` to get the next message.

```
POST /queues/jms.queue.bar/pull-consumers/consume-next-2
Host: example.com

--- Response ---
HTTP/1.1 200 Ok
Content-Type: application/xml
msg-consume-next: http://example.com/queues/jms.queue.bar/pull-consumers/333/consume-next-3

<order>...</order>
```

5.1.3. Recovering From Network Failures

If you experience a network failure and do not know if your post to a msg-consume-next URL was successful or not, just re-do your POST. A POST to a msg-consume-next URL is idempotent, meaning that it will return the same result if you execute on any one msg-consume-next URL more than once. Behind the scenes, the consumer resource caches the last consumed message so that if there is a message failure and you do a re-post, the cached last message will be returned (along with a new msg-consume-next URL). This is the reason why the protocol always requires you to use the next new msg-consume-next URL returned with each response. Information about what state the client is in is embedded within the actual URL.

5.1.4. Recovering From Client or Server Crashes

If the server crashes and you do a POST to the msg-consume-next URL, the server will return a 412 (Preconditions Failed) response code. This is telling you that the URL you are using is out of sync with the server. The response will contain a new msg-consume-next header to invoke on.

If the client crashes there are multiple ways you can recover. If you have remembered the last msg-consume-next link, you can just re-POST to it. If you have remembered the consumer resource URL, you can do a GET or HEAD request to obtain a new msg-consume-next URL. If you have created a topic subscription using the name parameter discussed earlier, you can re-create the consumer. Re-creation will return a msg-consume-next URL you can use. If you cannot do any of these things, you will have to create a new consumer.

The problem with the auto-acknowledge protocol is that if the client or server crashes, it is possible for you to skip messages. The scenario would happen if the server crashes after auto-acknowledging a message and before the client receives the message. If you want more reliable messaging, then you must use the acknowledgement protocol.

5.2. Manual Acknowledgement

The manual acknowledgement protocol is similar to the auto-ack protocol except there is an additional round trip to the server to tell it that you have received the message and that the server can internally ack the message. Here is a list of the respone headers you will be interested in.

msg-pull-consumers
: The URL of a factory resource for creating queue consumer resources. You will pull from these created resources

msg-pull-subscriptions
: The URL of a factory resource for creating topic subscription resources. You will pull from the created resources.

msg-acknowledge-next
: URL used to obtain the next message in the queue or topic subscription. It does not acknowledge the message though.

msg-acknowledgement
: URL used to acknowledge a message.

msg-consumer
: This is a URL pointing back to the consumer or subscription resource created for the client.

5.2.1. Creating manually-acknowledged consumers or subscriptions

Here is an example of creating an auto-acknowledged queue pull consumer.

1. Find the pull-consumers URL by doing a HEAD or GET request to the base queue resource.

```
HEAD /queues/jms.queue.bar HTTP/1.1
Host: example.com

--- Response ---
HTTP/1.1 200 Ok
msg-create: http://example.com/queues/jms.queue.bar/create
msg-pull-consumers: http://example.com/queues/jms.queue.bar/pull-consumers
msg-push-consumers: http://example.com/queues/jms.queue.bar/push-consumers
```

2. Next do a POST to the URL returned in the `msg-pull-consumers` header passing in a `false` value to the `autoAck` form parameter .

```
POST /queues/jms.queue.bar/pull-consumers HTTP/1.1
Host: example.com
Content-Type: application/x-www-form-urlencoded

autoAck=false

--- response ---
HTTP/1.1 201 Created
Location: http://example.com/queues/jms.queue.bar/pull-consumers/acknowledged/333
msg-acknowledge-next: http://example.com/queues/jms.queue.bar/pull-consumers/acknowledged/333/acknowledge-next-1
```

The `Location` header points to the JMS consumer resource that was created on the server. It is good to remember this URL, although, as you'll see later, it is transmitted with each response just to remind you.

Creating an manually-acknowledged consumer for a topic is pretty much the same. Here's an example of creating a durable manually-acknowledged topic pull subscription.

1. Find the `pull-subscriptions` URL by doing a HEAD or GET request to the base topic resource

```
HEAD /topics/jms.topic.bar HTTP/1.1
Host: example.com

--- Response ---
HTTP/1.1 200 Ok
msg-create: http://example.com/topics/jms.topic.foo/create
msg-pull-subscriptions: http://example.com/topics/jms.topic.foo/pull-subscriptions
msg-push-subscriptions: http://example.com/topics/jms.topic.foo/push-subscriptions
```

2. Next do a POST to the URL returned in the `msg-pull-subscriptions` header passing in a `true` value for the `durable` form parameter and a `false` value to the `autoAck` form parameter.

```
POST /topics/jms.topic.foo/pull-subscriptions HTTP/1.1
Host: example.com
Content-Type: application/x-www-form-urlencoded
```

```
durable=true&autoAck=false

--- Response ---
HTTP/1.1 201 Created
Location: http://example.com/topics/jms.topic.foo/pull-subscriptions/acknowledged/222
msg-acknowledge-next: http://example.com/topics/jms.topic.foo/pull-subscriptions/acknowledged/222/consume-next-1
```

The `Location` header points to the JMS subscription resource that was created on the server. It is good to remember this URL, although, as you'll see later, it is transmitted with each response just to remind you.

5.2.2. Consuming and Acknowledging a Message

After you have created a consumer resource, you are ready to start pulling messages from the server. Notice that when you created the consumer for either the queue or topic, the response contained a `msg-acknowledge-next` response header. POST to the URL contained within this header to consume the next message in the queue or topic subscription. If there are no messages in the queue or topic subscription, a 503 (Service Unavailable) HTTP code is returned. A successful POST causes the server to extract a message from the queue or topic subscription and return it to the consuming client. It does not acknowledge the message though. The response will contain the `acknowledgement` header which you will use to acknowledge the message.

Here's an example of pulling multiple messages from the consumer resource.

1. Do a POST on the msg-acknowledge-next URL that was returned with the consumer or subscription resource discussed earlier.

```
POST /queues/jms.queue.bar/pull-consumers/consume-next-1
Host: example.com

--- Response ---
HTTP/1.1 200 Ok
Content-Type: application/xml
msg-acknowledgement: http://example.com/queues/jms.queue.bar/pull-consumers/333/acknowledgement/2
msg-consumer: http://example.com/queues/jms.queue.bar/pull-consumers/333

<order>...</order>
```

The POST returns the message consumed from the queue. It also returns a `msg-acknowledgement` link. You will use this new link to acknowledge the message. Notice also a `msg-consumer` response header is returned. This is a URL that points back to the consumer

or subscription resource. You will need that to clean up your connection after you are finished using the queue or topic.

2. Acknowledge or unacknowledge the message by doing a POST to the URL contained in the `msg-acknowledgement` header. You must pass an `acknowledge` form parameter set to `true` or `false` depending on whether you want to acknowledge or unacknowledge the message on the server.

```
POST /queues/jms.queue.bar/pull-consumers/acknowledgement/2
Host: example.com
Content-Type: application/x-www-form-urlencoded

acknowledge=true

--- Response ---
Http/1.1 200 Ok
msg-acknowledge-next:        http://example.com/queues/jms.queue.bar/pull-consumers/333/
acknowledge-next-2
```

Whether you acknowledge or unacknowledge the message, the response will contain a new msg-acknowledge-next header that you must use to obtain the next message.

5.2.3. Recovering From Network Failures

If you experience a network failure and do not know if your post to a `msg-acknowledge-next` or `msg-acknowledgement` URL was successful or not, just re-do your POST. A POST to one of these URLs is idempotent, meaning that it will return the same result if you re-post. Behind the scenes, the consumer resource keeps track of its current state. If the last action was a call to `msg-acknowledge-next`, it will have the last message cached, so that if a re-post is done, it will return the message again. Same goes with re-posting to `msg-acknowledgement`. The server remembers its last state and will return the same results. If you look at the URLs you'll see that they contain information about the expected current state of the server. This is how the server knows what the client is expecting.

5.2.4. Recovering From Client or Server Crashes

If the server crashes and while you are doing a POST to the `msg-acknowledge-next` URL, just re-post. Everything should reconnect all right. On the other hand, if the server crashes while you are doing a POST to `msg-acknowledgement`, the server will return a 412 (Preconditions Failed) response code. This is telling you that the URL you are using is out of sync with the server and the message you are acknowledging was probably re-enqueued. The response will contain a new `msg-acknowledge-next` header to invoke on.

As long as you have "bookmarked" the consumer resource URL (returned from `Location` header on a create, or the `msg-consumer` header), you can recover from client crashes by doing a GET or

HEAD request on the consumer resource to obtain what state you are in. If the consumer resource is expecting you to acknowledge a message, it will return a `msg-acknowledgement` header in the response. If the consumer resource is expecting you to pull for the next message, the `msg-acknowledge-next` header will be in the response. With manual acknowledgement you are pretty much guaranteed to avoid skipped messages. For topic subscriptions that were created with a name parameter, you do not have to "bookmark" the returned URL. Instead, you can re-create the consumer resource with the same exact name. The response will contain the same information as if you did a GET or HEAD request on the consumer resource.

5.3. Blocking Pulls with Accept-Wait

Unless your queue or topic has a high rate of message flowing though it, if you use the pull protocol, you're going to be receiving a lot of 503 responses as you continuously pull the server for new messages. To alleviate this problem, the HornetQ REST interface provides the `Accept-Wait` header. This is a generic HTTP request header that is a hint to the server for how long the client is willing to wait for a response from the server. The value of this header is the time in seconds the client is willing to block for. You would send this request header with your pull requests. Here's an example:

```
POST /queues/jms.queue.bar/pull-consumers/consume-next-2
Host: example.com
Accept-Wait: 30

--- Response ---
HTTP/1.1 200 Ok
Content-Type: application/xml
msg-consume-next:     http://example.com/queues/jms.queue.bar/pull-consumers/333/consume-next-3

<order>...</order>
```

In this example, we're posting to a msg-consume-next URL and telling the server that we would be willing to block for 30 seconds.

5.4. Clean Up Your Consumers!

When the client is done with its consumer or topic subscription it should do an HTTP DELETE call on the consumer URL passed back from the Location header or the msg-consumer response header. The server will time out a consumer with the value configured from Chapter 2.3, so you don't have to clean up if you dont' want to, but if you are a good kid, you will clean up your messes. A consumer timeout for durable subscriptions will not delete the underlying durable JMS subscription though, only the server-side consumer resource (and underlying JMS session).

Pushing Messages

You can configure the HornetQ REST server to push messages to a registered URL either remotely through the REST interface, or by creating a pre-configured XML file for the HornetQ REST server to load at boot time.

6.1. The Queue Push Subscription XML

Creating a push consumer for a queue first involves creating a very simple XML document. This document tells the server if the push subscription should survive server reboots (is it durable). It must provide a URL to ship the forwarded message to. Finally, you have to provide authentication information if the final endpoint requires authentication. Here's a simple example:

```
<push-registration>
  <durable>false</durable>
  <selector><![CDATA[
       SomeAttribute > 1
     ]]>
  </selector>
  <link rel="push" href="http://somewhere.com" type="application/json" method="PUT"/>
</push-registration>
```

The `durable` element specifies whether the registration should be saved to disk so that if there is a server restart, the push subscription will still work. This element is not required. If left out it defaults to `false`. If durable is set to true, an XML file for the push subscription will be created within the directory specified by the `queue-push-store-dir` config variable defined in Chapter 2. (`topic-push-store-dir` for topics).

The `selector` element is optional and defines a JMS message selector. You should enclose it within CDATA blocks as some of the selector characters are illegal XML.

The `link` element specifies the basis of the interaction. The `href` attribute contains the URL you want to interact with. It is the only required attribute. The `type` attribute specifies the content-type ofwhat the push URL is expecting. The `method` attribute defines what HTTP method the server will use when it sends the message to the server. If it is not provided it defaults to POST. The `rel` attribute is very important and the value of it triggers different behavior. Here's the values a rel attribute can have:

destination
: The href URL is assumed to be a queue or topic resource of another HornetQ REST server. The push registration will initially do a HEAD request to this URL to obtain a msg-create-with-id header. It will use this header to push new messages to the HornetQ REST endpoint reliably. Here's an example:

```
<push-registration>
  <link rel="destination" href="http://somewhere.com/queues/jms.queue.foo"/>
</push-registration>
```

template
: In this case, the server is expecting the link element's href attribute to be a URL expression. The URL expression must have one and only one URL parameter within it. The server will use a unique value to create the endpoint URL. Here's an example:

```
<push-registration>
  <link rel="template" href="http://somewhere.com/resources/{id}/messages" method="PUT"/
>
</push-registration>
```

In this example, the {id} sub-string is the one and only one URL parameter.

user defined
: If the rel attributes is not destination or template (or is empty or missing), then the server will send an HTTP message to the href URL using the HTTP method defined in the method attribute. Here's an example:

```
<push-registration>
  <link href="http://somewhere.com" type="application/json" method="PUT"/>
</push-registration>
```

6.2. The Topic Push Subscription XML

The push XML for a topic is the same except the root element is push-topic-registration. (Also remember the `selector` element is optional). The rest of the document is the same. Here's an example of a template registration:

```
<push-topic-registration>
  <durable>true</durable>
  <selector><![CDATA[
      SomeAttribute > 1
    ]]>
  </selector>
  <link rel="template" href="http://somewhere.com/resources/{id}/messages" method="POST"/>
</push-topic registration>
```

6.3. Creating a Push Subscription at Runtime

Creating a push subscription at runtime involves getting the factory resource URL from the msg-push-consumers header, if the destination is a queue, or msg-push-subscriptions header, if the destination is a topic. Here's an example of creating a push registration for a queue:

1. First do a HEAD request to the queue resource:

```
HEAD /queues/jms.queue.bar HTTP/1.1
Host: example.com

--- Response ---
HTTP/1.1 200 Ok
msg-create: http://example.com/queues/jms.queue.bar/create
msg-pull-consumers: http://example.com/queues/jms.queue.bar/pull-consumers
msg-push-consumers: http://example.com/queues/jms.queue.bar/push-consumers
```

2. Next POST your subscription XML to the URL returned from msg-push-consumers header

```
POST /queues/jms.queue.bar/push-consumers
Host: example.com
Content-Type: application/xml

<push-registration>
  <link rel="destination" href="http://somewhere.com/queues/jms.queue.foo"/>
</push-registration>

--- Response ---
HTTP/1.1 201 Created
Location: http://example.com/queues/jms.queue.bar/push-consumers/1-333-1212
```

   The Location header contains the URL for the created resource. If you want to unregister this, then do a HTTP DELETE on this URL.

Here's an example of creating a push registration for a topic:

1. First do a HEAD request to the topic resource:

```
HEAD /topics/jms.topic.bar HTTP/1.1
Host: example.com

--- Response ---
```

```
HTTP/1.1 200 Ok
msg-create: http://example.com/topics/jms.topic.bar/create
msg-pull-subscriptions: http://example.com/topics/jms.topic.bar/pull-subscriptions
msg-push-subscriptions: http://example.com/topics/jms.topic.bar/push-subscriptions
```

2. Next POST your subscription XML to the URL returned from msg-push-subscriptions header

```
POST /topics/jms.topic.bar/push-subscriptions
Host: example.com
Content-Type: application/xml

<push-registration>
  <link rel="template" href="http://somewhere.com/resources/{id}"/>
</push-registration>

--- Response ---
HTTP/1.1 201 Created
Location: http://example.com/topics/jms.topic.bar/push-subscriptions/1-333-1212
```

   The Location header contains the URL for the created resource. If you want to unregister this, then do a HTTP DELETE on this URL.

6.4. Creating a Push Subscription by Hand

You can create a push XML file yourself if you do not want to go through the REST interface to create a push subscription. There is some additional information you need to provide though. First, in the root element, you must define a unique id attribute. You must also define a destination element to specify the queue you should register a consumer with. For a topic, the destination element is the name of the subscription that will be reated. For a topic, you must also specify the topic name within the topic element.

Here's an example of a hand-created queue registration. This file must go in the directory specified by the queue-push-store-dir config variable defined in Chapter 2:

```
<push-registration id="111">
  <destination>jms.queue.bar</destination>
  <durable>true>
  <link rel="template" href="http://somewhere.com/resources/{id}/messages" method="PUT"/>
</push-registration>
```

Here's an example of a hand-created topic registration. This file must go in the directory specified by the topic-push-store-dir config variable defined in Chapter 2:

```
<push-topic-registration id="112">
  <destination>my-subscription-1</destination
  <durable>true</durable>
  <link rel="template" href="http://somewhere.com/resources/{id}/messages" method="PUT"/>
  <topic>jms.topic.foo</topic>
</push-topic-registration>
```

6.5. Pushing to Authenticated Servers

Push subscriptions only support BASIC and DIGEST authentication out of the box. Here is an example of adding BASIC authentication:

```
<push-topic-registration>
  <durable>true</durable>
  <link rel="template" href="http://somewhere.com/resources/{id}/messages" method="POST"/>
  <authentication>
    <basic-auth>
      <username>guest</username>
      <password>geheim</password>
    </basic-auth>
  </authentication>
</push-topic registration>
```

For DIGEST, just replace basic-auth with digest-auth.

For other authentication mechanisms, you can register headers you want transmitted with each request. Use the header element with the name attribute representing the name of the header. Here's what custom headers might look like:

```
<push-topic-registration>
  <durable>true</durable>
  <link rel="template" href="http://somewhere.com/resources/{id}/messages" method="POST"/>
  <header name="secret-header">jfdiwe3321</header>
</push-topic registration>
```

Creating Destinations

You can create a durable queue or topic through the REST interface. Currently you cannot create a temporary queue or topic. To create a queue you do a POST to the relative URL /queues with an XML representation of the queue. The XML syntax is the same queue syntax that you would specify in hornetq-jms.xml if you were creating a queue there. For example:

```
POST /queues
Host: example.com
Content-Type: application/hornetq.jms.queue+xml

<queue name="testQueue">
  <durable>true</durable>
</queue>

--- Response ---
HTTP/1.1 201 Created
Location: http://example.com/queues/jms.queue.testQueue
```

Notice that the Content-Type is application/hornetq.jms.queue+xml.

Here's what creating a topic would look like:

```
POST /topics
Host: example.com
Content-Type: application/hornetq.jms.topic+xml

<topic name="testTopic">
</topic>

--- Response ---
HTTP/1.1 201 Created
Location: http://example.com/topics/jms.topic.testTopic
```

Securing the HornetQ REST Interface

8.1. Within JBoss Application server

Securing the HornetQ REST interface is very simple with the JBoss Application Server. You turn on authentication for all URLs within your WAR's web.xml, and let the user Principal to propagate to HornetQ. This only works if you are using the JBossSecurityManager with HornetQ. See the HornetQ documentation for more details.

8.2. Security in other environments

To secure the HornetQ REST interface in other environments you must role your own security by specifying security constraints with your web.xml for every path of every queue and topic you have deployed. Here is a list of URI patterns:

Table 8.1.

/queues	secure the POST operation to secure queue creation
/queues/{queue-name}	secure the GET HEAD operation to getting information about the queue.
/queues/{queue-name}/create/*	secure this URL pattern for producing messages.
/queues/{queue-name}/pull-consumers/*	secure this URL pattern for pulling messages messages.
/queues/{queue-name}/push-consumers/*	secure this URL pattern for pushing messages.
/topics	secure the POST operation to secure topic creation
/topics/{topic-name}	secure the GET HEAD operation to getting information about the topic.
/topics/{topic-name}/create/*	secure this URL pattern for producing messages.
/topics/{topic-name}/pull-subscriptions/*	secure this URL pattern for pulling messages messages.
/topics/{topic-name}/push-subscriptions/*	secure this URL pattern for pushing messages.

Mixing JMS and REST

The HornetQ REST interface supports mixing JMS and REST producres and consumers. You can send an ObjectMessage through a JMS Producer, and have a REST client consume it. You can have a REST client POST a message to a topic and have a JMS Consumer receive it. Some simple transformations are supported if you have the correct RESTEasy providers installed.

9.1. JMS Producers - REST Consumers

If you have a JMS producer, the HornetQ REST interface only supports ObjectMessage type. If the JMS producer is aware that there may be REST consumers, it should set a JMS property to specify what Content-Type the Java object should be translated into by REST clients. The HornetQ REST server will use RESTEasy content handlers (MessageBodyReader/Writers) to transform the Java object to the type desired. Here's an example of a JMS producer setting the content type of the message.

```
ObjectMessage message = session.createObjectMessage();
message.setStringProperty(org.hornetq.rest.HttpHeaderProperty.CONTENT_TYPE,
 "application/xml");
```

If the JMS producer does not set the content-type, then this information must be obtained from the REST consumer. If it is a pull consumer, then the REST client should send an Accept header with the desired media types it wants to convert the Java object into. If the REST client is a push registration, then the type attribute of the link element of the push registration should be set to the desired type.

9.2. REST Producers - JMS Consumers

If you have a REST client producing messages and a JMS consumer, HornetQ REST has a simple helper class for you to transform the HTTP body to a Java object. Here's some example code:

```
public void onMessage(Message message)
{
   MyType obj = org.hornetq.rest.Jms.getEntity(message, MyType.class);
}
```

The way the `getEntity()` method works is that if the message is an ObjectMessage, it will try to extract the desired type from it like any other JMS message. If a REST producer sent the message, then the method uses RESTEasy to convert the HTTP body to the Java object you want. See the Javadoc of this class for more helper methods.