Developing RESTful Services with JAX-RS 2.0, WebSockets, and JSON

A complete and practical guide to building RESTful Web Services with the latest Java EE7 API

Masoud Kalali
Bhakti Mehta

Developing RESTful Services with JAX-RS 2.0, WebSockets, and JSON

A complete and practical guide to building RESTful Web Services with the latest Java EE7 API

Masoud Kalali

Bhakti Mehta

[PACKT] enterprise
PUBLISHING
professional expertise distilled

BIRMINGHAM - MUMBAI

Developing RESTful Services with JAX-RS 2.0, WebSockets, and JSON

Credits

Authors

Masoud Kalali

Bhakti Mehta

Reviewers

Anthony Dahanne

Jitendra Kotamraju

Arvind Maheshwari

Acquisition Editors

Antony Lowe

Erol Staveley

Lead Technical Editor

Sharvari Tawde

Technical Editors

Vrinda Nitesh Bhosale

Amit Shetty

Project Coordinator

Amey Sawant

Proofreader

Stephen Copestake

Indexer

Tejal R.Soni

Graphics

Ronak Dhruv

Yuvraj Mannari

Production Coordinators

Prachali Bhiwandkar

Kyle Albuquerque

Cover Work

Prachali Bhiwandkar

About the Authors

Masoud Kalali has been working on software development projects since 1998, which gives him a broad perspective on software development in general and changes in the software development landscape in the past 1.5 decades. Masoud has experience with a variety of technologies (.NET, J2EE, CORBA, and COM+) on diverse platforms (Solaris, Linux, and Windows). He has a masters degree in Information Systems with a bachelor degree in Software Engineering.

Masoud has authored a fair number of articles and other types of material, including several articles at Java.net and Dzone. He is the author of multiple refcardz, published by Dzone, including but not limited to *Using XML in Java* (`http://refcardz.dzone.com/refcardz/using-xml-java`) and Security and GlassFish v3 (`http://refcardz.dzone.com/refcardz/getting-started-glassfish`) refcardz. Masoud is one of the founding members of NetBeans Dream Team (`http://wiki.netbeans.org/NetBeansDreamTeam`) and a GlassFish community spotlighted developer (`https://glassfish.java.net/public/developers.html`). Masoud is the author of *GlassFish Security* (`http://www.packtpub.com/glassfish-security/book`) that was published in 2010, covering GlassFish v3 security and Java EE 6 security.

Masoud's main area of research and interest includes service-oriented architecture and large-scale systems development and deployment. In his spare time he enjoys photography, mountaineering, and climbing.

Masoud's Twitter handle is `@MasoudKalali` if you want to know what he is up to.

> I should acknowledge my family's support and encouragement as well as Bhakti's patience with me during the course of developing this book. Reviewers, including Anthony Dahanne, Jitendra Kotamraju, and Arvind Maheshwari, played a vital role in developing this book and deserve special thanks. At the end I should acknowledge the role that the Packt Publishing team, including but not limited to Amey Sawant, Sharvari Tawde, and Parita Khedekar, played in concluding this project.

Bhakti Mehta is a Senior Technology Professional with over 12 years of experience in architecting, designing, and implementing Software Solutions on top of Java EE and other related technologies. On the platform level, she is well experienced in different areas of GlassFish Application Server and Java EE specifications.

Bhakti is experienced in developing open source software and working with open source communities and customers. She is a member of the GlassFish team at Oracle. Bhakti's primary areas of interest are server-side technologies, XML, Web Services, Java EE, and Cloud. She has a bachelors degree in Computer Engineering and a masters degree in Computer Science.

Bhakti is a regular speaker in various conferences along with having articles and enterprise tech tips at different portals. Her tweets can be followed at `@bhakti_mehta`.

About the Reviewers

Anthony Dahanne has been a Java software developer since 2005. His favorite topics are Android, building tools, Continuous Integration, Web Services, and, of course, core Java development. In his spare time, he's hacking on some open source Android app (G2Android, ReGalAndroid, and so on); he also contributes from time to time to build/IDE plugins usually involving Maven and Eclipse.

You can meet him at one of the many Java-related user group gathering in Montréal (Android Montréal, Montréal JUG, and Big Data Montréal). Working at Terracotta, he's currently implementing the REST management interface for Ehcache.

> I would like to thank Guilhem De Miollis for his time spent reviewing the book and even suggesting some topics, my colleagues at the Interfaces team at Terracotta for always taking the time to share their deep Java knowledge with me, and finally my beloved wife Isabelle for her patience and help to make this book happen.

Jitendra Kotamraju, a principal member of the technical staff at Oracle, is the JSON Processing specification lead and one of the key engineers behind GlassFish. Before leading the JSON Processing project, he was in charge of both the specification and implementation of JAX-WS 2.2. Currently, he is also implementing various web technologies such as Server-sent Events (SSE) and WebSocket in GlassFish.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`http://PacktLib.PacktPub.com`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Instant Updates on New Packt Books

Get notified! Find out when new books are published by following `@PacktEnterprise` on Twitter, or the *Packt Enterprise* Facebook page.

Table of Contents

Preface

Over the years, we have seen several revolutions, and shifts in paradigms spanning from mainframes to x86 farms, from heavyweight methodologies to lightweight agile methods, and from desktop and thick clients to thin, rich, and highly available web applications and ubiquitous computing.

With the advancements and changes in the technology landscape going towards smaller, more portable, and lightweight devices and the devices being used widely for day-to-day activities, the need to push the computation from the client machines to the backend grows to an even more prominent one. This also brings forth opportunities and challenges involved in developing applications with near real-time or real-time event and data propagation between servers and clients; that is where HTML 5 provides developers with the standards, API, and flexibility required to achieve the same result in web applications that is achievable in thick desktop applications.

The communication between clients and the servers has turned to the most fundamental subject both in terms of quantity, content, interoperability, and scalability of these interactions. The XML era, long waiting requests, single browser, and single device compatibility is over; instead the era of devices, multiple clients, and browsers, from very small devices capable of only processing text over HTTP to mammoth scale machines processing almost any kind of content has begun. With this said, producing the content and accepting the content along with the ability to switch between older and newer protocols has turned into an obvious must.

Java EE 7 comes with more emphasis on these emerging (and dominating) requirements; support for HTML5, more asynchronous communication/ invocation-capable components, and support for JSON as one of the data formats have arrived to help developers with resolving the technical requirements and giving the developers ample time to work on the business requirements' implementation of their systems.

This book is an attempt to provide the avid technologists with an overview of what Java EE is in general and Java EE 7 in particular, as a technology platform, provides for developing lightweight, interactive applications based on HTML5 deployable in any Java EE compatible container.

What this book covers

Chapter 1, *Building RESTful Web Services Using JAX-RS*, starts with the basic concepts of building RESTful Web Services and covers JAX-RS 2.0 API, detailing the different annotations, Providers, `MessageBodyReader`, `MessageBodyWriter`, Client API, and Bean Validation support in JAX-RS 2.0.

Chapter 2, *WebSockets and Server-sent Events*, discusses the different programming models for sending near real-time updates to clients. It also covers WebSockets and Server-sent Events, the JavaScript and Java API for WebSockets and Server-sent Events. This chapter compares and contrasts WebSockets and Server-sent Events and shows the advantages of WebSockets to reduce unnecessary network traffic and improve the performance.

Chapter 3, *Understanding WebSockets and Server-sent Events in Detail*, covers the Java EE 7 API for WebSockets, Encoders and Decoders, the Client API, how to send different kinds of messages with WebSockets using blobs, and `ArrayBuffers`. It teaches how to secure a WebSockets-based application. It outlines the best practices for WebSockets and Server-sent Events-based applications.

Chapter 4, *JSON and Asynchronous Processing*, covers the Java EE 7 JSON-P API for parsing and manipulating JSON data. It also discusses the new NIO API introduced in Servlet 3.1 specification. It teaches how to use the JAX-RS 2.0 API for asynchronous request processing to improve scalability.

Chapter 5, *RESTful Web Services by Example*, covers two real-life examples of RESTful Web Services. It covers an event notification sample based on the Twitter Search API, how the server can push the data to clients as and when events occur. A library application ties the different technologies covered in the above chapters together.

What you need for this book

To be able to build and run samples provided with this book you will need:

1. Apache Maven 3.0 and higher. Maven is used to build the samples. You can download Apache Maven from `http://maven.apache.org/download.cgi`

2. GlassFish Server Open Source Edition v4.0 is the free, community-supported Application Server providing implementation for Java EE 7 specifications. You can download the GlassFish Server from `http://dlc.sun.com. edgesuite.net/glassfish/4.0/promoted/`

Who this book is for

This book is a perfect reading source for application developers who are familiar with Java EE and are keen to understand the new HTML5-related functionality introduced in Java EE 7 to improve productivity. To take full advantage of this book, you need to be familiar with Java EE and have some basic understanding of using GlassFish application server.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "The request, which is sent to the JAX-RS resource, is a `POST` request with `app/ library/book/` as the target URI."

A block of code is set as follows:

```
@GET
@Path("browse")
public List<Book> browseCollection() {
   return bookService.getBooks();     }
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
@GET
@Path("browse")
public List<Book> browseCollection() {
   return bookService.getBooks();     }
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "When a user clicks on the **Hold** button on the HTML page".

> Warnings or important notes appear in a box like this.

> Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
Building RESTful Web Services Using JAX-RS

There are various ways to implement communication between heterogeneous applications. There are standards focusing on web services based on **SOAP**, **WSDL**, and WS* specifications; alongside these standards there is an emerging lightweight solution based on plain HTTP referred to as **Representational State Transfer** (**REST**).

REST is identified by the principles of addressable resources, constrained interfaces using HTTP verbs, representation, and statelessness.

The key principles of REST are:

- Associating IDs to resources
- Using standard HTTP methods
- Multiple formats of data sent by a resource
- Statelessness

This chapter starts with the basic concept of building **RESTful Web Services** using the **JAX-RS** 2.0 API and covers the following sections:

- Getting started with JAX-RS 2.0
- Converting POJOs to RESTful endpoints using JAX-RS 2.0 annotations
- `@Produces`, `@Consumes` annotations
- Client API for JAX-RS 2.0
- Sample showing all verbs
- Custom entity providers for **serializing** and **deserializing** user defined classes using JAX-RS
- Utilizing the Bean Validation API for validation with JAX-RS 2.0

Understanding REST

The REST architectural style is based on request and response messages transferred between clients and servers without any of the participating node keeping track of the state of previous sessions..

REST uses nouns and verbs for readability. Resources are identified in requests. The representation of the resource that is sent to the client depends on the request and how the server sends the data.

RESTful Web Services

A RESTful Web Service is a service whose interface and accessing mechanism are aligned with the REST principles . The URIs identify the resources. For example, a RESTful resource for a book can be identified as `http://foo.org/book`.

A resource for a book identified by ISBN could be `http://foo.org/book/isbn/1234459`. This shows a human-readable URI that is easy to understand and identify.

A client has enough metadata of a resource to modify or delete it as long as it is authorized to do so. To get a resource the client would send a HTTP `GET` request. To update the resource the client would send a `PUT` request. To delete a resource the client would send a `DELETE` request. To create a new resource, and for arbitrary processing, the client sends a HTTP `POST` request. The next section covers these verbs in more detail.

Verbs in REST

Some of the requests used in REST are as follows:

- `GET`: The `GET` request retrieves a representation of a resource from server to client
- `POST`: The `POST` request is used to create a resource on the server based on the representation that the client sends
- `PUT`: The `PUT` request is used to update or create a reference to a resource on server
- `DELETE`: The `DELETE` request can delete a resource on server
- `HEAD`: The `HEAD` requests checks for a resource without retrieving it

The next section will introduce the notion of safety and **idempotence**, two important terms associated with REST.

Safety and idempotence

When it comes to REST, a safe method, by definition, is a HTTP method that does not modify the state of the resource on the server. For example, invoking a `GET` or a `HEAD` method on the resource URL should never change the resource on the server. `PUT` is considered not safe since it usually creates a resource on the server. `DELETE` is also considered not safe since it will delete the resource on the server. `POST` is not safe since it will change the resource on the server.

Idempotent method is a method that can be called multiple times yet the outcome will not change.

`GET` and `HEAD` are idempotent, which means that even though the same operation is done multiple times the result does not vary. `PUT` is idempotent; calling the `PUT` method multiple times will not change the result and the resource state is exactly the same.

`DELETE` is idempotent because once the resource is deleted it is gone, and calling the same operation multiple times will not change the outcome.

In contrast, `POST` is not idempotent and calling `POST` multiple times can have different outcomes.

> The idempotence and safety of the HTTP verbs are a convention, meaning that when someone is using your API they will assume that `GET/PUT/POST/DELETE` have the same idempotency characteristics that are previously described; and the implementation of the business logic behind each verb should support these characteristics.

The response sent by the server could be in XML, JSON, or any other MIME type as long as the server supports the requested format. In case the server cannot support the requested MIME type, it can return with a status code of 406 (not acceptable).

When we are developing with RESTful principles in mind, each message should have enough information to let the server understand the purpose of the message and how to process that message, to produce the response the message is meant for, and finally to ensure visibility and statelessness.

Summarizing, these are the components of RESTful Web Services:

- **Base URI**: The base URI for the Web Service `http://foo.com/bar`
- **Media type**: The media type supported by the Web Service
- **Methods**: The HTTP methods such as `GET`, `PUT`, `POST`, and `DELETE`

Introduction to JAX-RS

The **Java API for Representational State Transfer** (**JAX-RS**) specification defines a set of Java APIs for building web services conforming to the REST style.

This specification defines how to expose POJOs as web resources, using HTTP as the network protocol. Applications using these APIs can be deployed to an application server in a portable manner.

Some of the key features that are introduced in the JAX-RS 2.0 specification are as follows:

- Client API
- Server side asynchronous support
- **Bean Validation** support

In the subsequent sections we will cover the following topics in relation to JAX-RS 2.0:

- Converting POJOs to RESTful resources
- More on JAX-RS annotations
- Client API for JAX-RS
- Entities in JAX-RS
- Custom entity providers in JAX-RS
- Using the Bean Validation API with JAX-RS

Converting POJOs to RESTful resources

A resource class is a POJO that uses the JAX-RS annotations. A resource class needs to have at least one method annotated with `@Path` or a request method. Resources are our so-called web services and incoming requests target these resources.

Steps to convert POJOs to RESTful endpoints:

1. Define a root resource as identified by a URI
2. Define the methods for the resource
3. Define the MIME types
4. Define the Application subclass
5. Define the subresources

Defining a root resource as identified by a URI

JAX-RS provides very rich client and server APIs that work on any Java EE application server. Using JAX-RS API, any POJO can be annotated to build the RESTful resources. Begin with a simple POJO `BookResource` and annotate it with the JAX-RS APIs.

```
@Path("books")
public class BooksResource {
}
```

This is a root resource class, which is annotated with `@Path annotation`. The value `"books"` will indicate that the resource will be available at a location similar to the following URI `http://host:port/appname/books`.

Later on we add the methods to this resource so that, when a request with `GET`, `PUT`, and so on hits this resource, a particular method in the class is invoked to produce the response.

Defining the methods for the resource

To add a method to this resource, we annotate the method with `@GET`, `@PUT`, `@DELETE`, or `@HEAD`. In the following example, we chose to annotate using a `@GET annotation`:

```
@GET
public String getGreeting() {
  return "Hello from Book resource"
}
```

The `@GET` annotation specifies that the `getGreeting()` method handles the `HTTP GET` requests.

Defining the MIME types

To specify the MIME type that can be handled by the resource, we should annotate the resource method with `@Produces` and `@Consumes`:

```
@Produces("text/plain")
@GET
public String getGreeting() {
  return "Hello from Book resource"
}
```

The `@Produces` specifies that the media type this method will produce is `"text/plain"`. Support for other media types, and how to map from Java to a specific format and vice versa, is covered in detail in the entity provider's section. Thus, this is the initial introduction to having a first JAX-RS resource ready. The next section covers the details of the `Application` subclass.

Defining the Application subclass

The `Application` class is a portable way to configure application-level details such as specifying the name, and registering various components of a JAX-RS application. This includes the different JAX-RS resources and the JAX-RS providers in the application.

Similarly, application-wide properties can be set using a subclass of `Application`. The `Application` subclass should to be placed in either in `WEB-INF/classes` or `WEB-INF/lib` in a WAR file. Application class has the following methods that can be overridden:

```
public Set<Class<?>> getClasses() ;
public Map<String, Object> getProperties();
public Set<Object> getSingletons();
```

Here is an example of a subclass of Application for our case:

```
@ApplicationPath("/library/")
public class HelloWorldApplication extends Application {
@Override
  public Set<Class<?>> getClasses() {
    Set<Class<?>> classes = new HashSet<Class<?>>();
    classes.add(BooksResource.class);
    return classes;
  }
}
```

In this code we create a `HelloWorldApplication`, which is a subclass of `javax.ws.rs.core.Application`. With Servlet 3.0 there is no need of a `web.xml` file and the servlet container uses the value specified in the `@ApplicationPath` as the servlet mapping. The `getClasses()` method of the `Application` class is overridden to add `BooksResource.class`.

A basic JAX-RS resource is now ready to use. When the sample is deployed to an application server such as GlassFish, you can use curl to send a request.

Here is an example on how to send a `curl -X GET` request:

```
curl -X GET http://localhost:8080/helloworld/books
```

The output in the terminal window should be:

Hello from book resource

Chapter 5, RESTful Web Services by Example, will show how to use the Application class in a `web.xml` file.

Defining the subresources

Resource classes can partially process some part of the request and provide another subresource to process the remaining part of the request.

For example, here is a snippet of a root resource `Library` and another resource `Book`.

```
@Path("/")
public class Library {

  @Path("/books/{isbn}")
  public Book getBook(@PathParam("isbn") String isbn){
    //return book
  }
}

public class Book {
  @Path("/author")
  public String getAuthor(){
  }
}
```

Subresource locators are resource methods that have `@Path` annotation but no HTTP methods.

In the preceding example, `Library` is a root resource as it is annotated with `@Path`. The method `getBook()` is a subresource locator whose job is to provide an object that can process the request.

The `@PathParam` is an annotation that allows you to map URI path fragments in the method call. In this example, the `isbn` URI parameter is passed to provide information about the book.

If a client sends a request using the URI:

```
GET /books/123456789
```

The `Library.getBook()` method will be invoked.

If a client sends a request using the URI:

```
GET /books/123456789/author
```

The `Library.getBook()` method will be invoked first. A `Book` object is returned and then the `getAuthor()` method is invoked.

More on JAX-RS annotations

The `@Produces` annotation is used to define the type of output the method in the resource produces. The `@Consumes` annotation is used to define the type of input, the method in the resource consumes.

Here is a method in a resource for a `POST` request:

```
@POST
@Consumes(MediaType.APPLICATION_XML)
@Produces(MediaType.APPLICATION_XML)
public Response addBook(Book book) {
  BooksCollection.addBook(book);
  return    Response.ok(book).
  type(MediaType.APPLICATION_XML_TYPE).build();
}
```

As shown in this snippet we have the `@POST` annotation that indicates this method accepts `POST` request.

The `@Produces(MediaType.APPLICATION_XML)` indicates that the `"application/xml"` media type is produced by the `addBook()` method of this resource.

The `@Consumes(MediaType.APPLICATION_XML)` indicates that the `"application/xml"` media type is consumed by the `addBook()` method of this resource.

The `Response.ok(book)` method builds an ok response of the type `MediaType.APPLICATION_XML_TYPE`

Other supported media types `@Produces` and `@Consumes` are `"text/xml"`, `"text/html"`, `"application/json"`, and so on.

If there is no media type specified in the `@Produces` or `@Consumes` annotations, support for any media type is assumed by default.

Here is a snippet of code that shows the @DELETE annotation.

```
@DELETE
@Path("/{isbn}")
public Book deleteBook(@PathParam("isbn")String isbn) {
   return BooksCollection.deleteBook(isbn);
}
```

The @PathParam annotation allows you to map the URI path fragments in the method call. In this example, the isbn URI parameter is passed to provide information about the book.

The ISBN uniquely identifies the Book resource so that it can be deleted.

The following table summarizes important JAX-RS 2.0 annotations included in Java EE 7 and used throughout this book.

Annotation	Description
@Path	To annotate a POJO with the resource path it represents. For example, @Path("books") or to annotate a subresource that is a method in the annotated class.
@Produces	To specify the output type that the resource produces, or in a narrower scope the type of output that a method in a resource produces. For example:
	@Produces(MediaType.APPLICATION_JSON).
@Consumes	To specify the type of input that the resource consumes, or in a narrower scope the type of input that a method in a resource consumes. For example:
	@Consumes (MediaType.APPLICATION_JSON).
@GET, @POST, @DELETE, and so on	To map the HTTP methods to methods in the resource representing class. For example, @GET can be placed on getBook method.
@PathParam	To specify the mapping between query parameter names and method. For example:
	getBook(@PathParam("isbn") String isbn).
@ApplicationPath	Identifies the application path that serves as the base URI for all resource URIs provided by path. For example, @ApplicationPath("library") for the library application.
@Context	Can be used to inject contextual objects such as UriInfo, which provides contextual request-specific information about the request URI. For example:
	getBook(@Context UriInfo uriInfo,

Chapter 5, RESTful Web Services by Example, covers the different JAX-RS APIs in detail and ties them together with other Java EE APIs to build a real-world library application.

The Client API for JAX-RS

JAX-RS 2.0 provides a rich client API to access the web resources. Here is the code on how to use the client API for the `BooksResource` we built earlier:

```
Client client = ClientBuilder.newClient();
WebTarget target = client.target(URI);
```

The default instance of the `javax.ws.rs.client.Client` object can be obtained using the `ClientBuilder.newClient()` API. The `BooksResource` can be identified by URI. The `WebTarget` object is used to build the URI.

```
String book = target.request().get(String.class);
```

The `target.request().get(String.class)` method builds an HTTP `GET` request and gets an object of type `String` in the response. More samples of the client API with other verbs are shown in the next section.

Entities in JAX-RS

The main part of an HTTP interaction consists of the request and response entities. Entities are also referred to as the payload or message body in some contexts.

Entities are sent via a request, usually an HTTP `POST` and `PUT` method is used, or they are returned in a response, this is relevant for all the HTTP methods. The `Content-Type` HTTP header is used to indicate the type of entity being sent. Common **content types** are `"text/plain"`, `"text/xml"`, `"text/html"`, and `"application/json"`.

Media types are used in the `Accept` header to indicate what type of resource representation the client wants to receive.

The following snippet shows how to use the client API to create a `POST` request. This invocation takes an entity for a user-defined class `Book` and a `MediaType. APPLICATION_XML_TYPE` parameter.

Here is the client code to invoke the `POST` method:

```
Response response = target.request()
post(Entity.entity(new Book("Getting Started with RESTful Web
  Services","111334444","Enterprise Applications"),
    MediaType.APPLICATION_XML_TYPE));
```

In the preceding snippet, the `WebTarget#request()` method returns a
`Response` object.

Here is the client API code to invoke the `delete` method:

```
response = target.path("111334444")
request( MediaType.APPLICATION_XML_TYPE)
.delete();
```

The next section will show how the entity providers that implement the JAX-RS API
map to and from Java types request and response entities.

Custom entity providers in JAX-RS

JAX-RS enables developers to add custom entity providers to the application.
The custom entity providers can be used for dealing with user-defined classes
in the requests as well as responses.

Adding a custom entity provider provides a way to **deserialize** user-defined classes
from the message bodies and **serialize** any media type to your user specific class.

There are two types of entity providers:

- `MessageBodyReader`
- `MessageBodyWriter`

Using the `@Provider` annotation, application-specific provider classes can be
discovered. Entity providers provide mapping between the representation and
associated type. There is a sample included with the book that demonstrates
the use of entity providers.

MessageBodyReader

An application can provide an implementation of the `MessageBodyReader` interface
by implementing the `isReadable()` method and the `readFrom()` method to map the
entity to the desired Java type.

---

[17]

The following figure shows how the `MessageBodyReader` reads an `InputStream` object and converts it to a user-defined Java object.
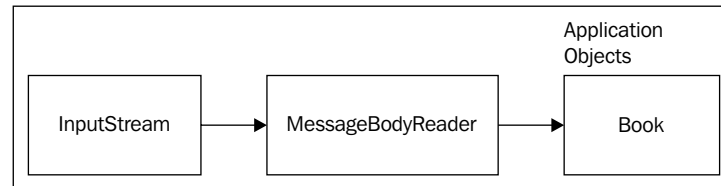
The following code shows how to provide an implementation of `MessageBodyReader` and uses **Java Architecture for XML Binding (JAXB)** with JAX-RS. JAXB provides a fast and convenient way to bind XML schemas and Java representations, making it easy for Java developers to incorporate the XML data and processing functions in Java applications. As a part of this process, JAXB provides methods for **unmarshalling** (reading) XML instance documents into Java content trees, and then **marshalling** (writing) Java content trees back into XML instance documents.

Here is a JAXB root element called `Book`. `Book` has properties such as name and ISBN.

```
@XmlRootElement
public class Book {
  public String name;
  public String isbn;
  public String getName() {
    return name;
  }
  public String getIsbn() {
    return isbn;
  }
  public Book(String name, String isbn) {
    this.name=name;
    this.isbn=isbn;
  }
  //JAXB requires this
  public Book() {

  }
}
```

The `MessageBodyReader` implementation class can provide support to read from an `inputStream` object and convert to the `Book` object. The following table shows the methods that need to be implemented:

[18]

Method of MessageBodyReader	Description
isReadable()	To check if the MessageBodyReader class can support conversion from stream to Java type.
readFrom()	To read a type from the InputStream.

Here is the code for `SampleMessageBodyReader` class that is the implementation of the `MessageBodyReader` interface:

```
@Provider
public class SampleMessageBodyReader implements
MessageBodyReader<Book> {
}
```

The `@Provider` annotation indicates that this is a provider and the implementing class can also use `@Produces` and `@Consumes` annotations to restrict the media types they support.

Here is the implementation of `isReadable()` method:

```
public boolean isReadable(Class<?> aClass, Type type, Annotation[]
  annotations, MediaType mediaType) {
  return true;
}
```

The `isReadable()` method returns `true` to indicate that this `SampleMessageBodyReader` class can process the `mediaType` parameter.

This is an implementation of the `readFrom()` method of the `SampleMessageBodyReader` class. The `mediaType` parameter can be checked here and different actions can be taken based on the media type.

```
public Book readFrom(Class<Book> bookClass, Type type,
  Annotation[] annotations,
MediaType mediaType,
MultivaluedMap<String, String> stringStringMultivaluedMap,
InputStream inputStream) throws IOException,
  WebApplicationException {
  try {

    Book book = (Book)unmarshaller.unmarshal(inputStream) ;
    return book;
  } catch (JAXBException e) {
    e.printStackTrace();
  }
```

[19]

```
      return null;
    }
  }
```

The `book` object, which is the method's return value, is then unmarshalled using JAXB **Unmarshaller** using the provided `inputStream` object as the parameter.

MessageBodyWriter

The `MessageBodyWriter` interface represents a contract for a provider that supports the conversion from a Java type to a stream.

The following figure shows how `MessageBodyWriter` can take a user-defined class, `Book`, and marshal it to an `outputStream` object.
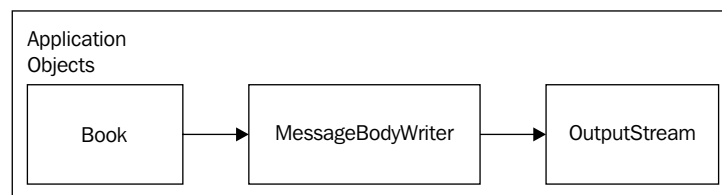
The following table shows the methods of `MessageBodyWriter` that must be implemented along with a short description of each of its method.

Method of MessageBodyWriter	Description
`isWritable()`	To check if the `MessageBodyWriter` class can support the conversion from the specified Java type.
`getSize()`	To check the length of bytes if the size is known or -1.
`writeTo()`	To write from a type to the stream.

Here are the methods of the `MessageBodyWriter` interface that need to be implemented:

```
public boolean isWriteable(Class<?> aClass, Type type,
  Annotation[] annotations, MediaType mediaType) {
    return true;
}
```

The `isWritable()` method of the `MessageBodyWriter` interface can be customized to check if this implementation of `MessageBodyWriter` supports the type or not.

```
    public long getSize(Book book, Class<?> aClass, Type type,
Annotation[] annotations, MediaType mediaType) {
        return -1;
    }
```

The `getSize()` method is called before the `writeTo()` method to ascertain the length of bytes in the response.

```
public void writeTo(Book book,
Class<?> aClass,
Type type, Annotation[] annotations,
MediaType mediaType,
MultivaluedMap<String, Object> map,
OutputStream outputStream) throws
IOException, WebApplicationException {
  try {

    Marshaller marshaller = jaxbContext.createMarshaller();
    marshaller.marshal(book, outputStream);
  } catch (Exception e) {
    e.printStackTrace();
  }
}
```

The `writeTo()` method marshals the `Book` to the `Outputstream`.

> **Tips for debugging errors with** `MessageBodyReader`
> **and** `MessageBodyWriter`**:**
>
> - Look for the `@Provider` annotation. `MessageBodyReader` implementation class and `MessageBodyWriter` implementation class need the `@Provider` annotation.
> - Confirm if the implementation classes of `MessageBodyReader` and `MessageBodyWriter` interfaces are added in the `getClasses()` method of the Application subclass.
> - Check if the implementation of `MessageBodyReader.isReadable()` method returns `true`.
> - Check if the implementation of `MessageBodyWriter.isWritable()` method returns `true`.
> - Confirm the `MessageBodyWriter.getSize()` method is `-1` if the size of response is unknown or set it to the right value if the size is known.

This is how the client looks:

```
Client client = ClientBuilder.newClient();
client.register(MessageBodyReaderWriter.class).register
  (BooksResource.class);
Response response = target
.request()
.post(Entity.entity(new Book("Getting Started with RESTful Web
  Services","13332233"), MediaType.APPLICATION_XML_TYPE));

Book  = response.readEntity(Book.class);
```

The `client.register()` method is used to register the `MessageBodyReaderWriter.class` and `BooksResource.class`.

The application class, `Book` is extracted from the response using `response.readEntity(Book.class)`.

Using the Bean Validation API with JAX-RS

Validation is the process of verifying that the given inputs are complying with the defined constraints. The Bean Validation specification defines the API to validate **JavaBeans**. This section shows how to validate the JAX-RS 2.0 resources using the Bean Validation API.

Validation can be used to ensure that fields in the JAX-RS resources follow certain constraints. For example, to check that a field is not `null` or if the ISBN follows a pattern. Using Bean Validation, a user can write custom validators and annotate the JAX-RS resources and their components using the custom validators.

The sample included along with the book will show how to use Bean Validation with JAX-RS 2.0 resources.

Here is a code snippet showing how to enforce validation along with defining a constraint and adding a user-defined message to it:

```
@Path("books")
@ValidateOnExecution(ExecutableType.GETTER_METHODS)
public class BooksResource {

  @GET
  @Path("{isbn}")
  @Consumes(MediaType.APPLICATION_XML)
  @Produces(MediaType.APPLICATION_XML)
  @NotNull(message="Book does not exist for the
```

```
  ISBN requested")
public Book getBook(
@PathParam("isbn")String isbn)      {
   return BooksCollection.getBook(isbn);

 }
}
```

The `@ValidateOnExecution` annotation can be used to selectively enable and disable the validation. In this snippet, the `getBook()` method gets validated because the `@ValidateOnExecution` annotation enables the validation for the `ExecutableType.GETTER_METHODS` value.

When the sample code is executed, if the book value is not null then, the book object is returned. If the book value is null, there is a validation error with a message shown on the screen as `"Book does not exist for the ISBN requested"`. This is the message that is provided with the `@NotNull` annotation shown previously.

Enabling validation in the application

Getting validation errors from the response is not enabled by default. The sample included in the book will demonstrate how to get the validation errors from the response. The user needs to set `BV_SEND_ERROR_IN_RESPONSE` property to Boolean value `true` using `Application` class by overriding the `getProperties()` method.

Here is the `getProperties()` method of the `Application` subclass.

```
@override
public Map<String,Object> getProperties() {
  Map<String,Object> properties = new HashMap<String,Object>() ;
  properties.put(ServerProperties.BV_SEND_ERROR_IN_RESPONSE,
    true);
  return properties;
}
```

The `getProperties()` method returns the `Map<String,Object>` object with the String property `ServerProperties.BV_SEND_ERROR_IN_RESPONSE` set to the Boolean value `true`.

> **Downloading the example code**
>
> You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

---

[23]

Reading validation errors from the response

After the application class is configured to set the String property
`ServerProperties.BV_SEND_ERROR_IN_RESPONSE` to the Boolean value `true`,
the following code in the servlet class will read the validation errors from
the response.

This is how the code looks on the client side:

```
List<ValidationError> errors = response.readEntity(new
    GenericType<List<ValidationError>>() {});
```

The `response.readEntity()` method takes a list of `GenericType<ValidationError>`
parameters. From the `List<ValidationError> errors`, returned by the `response.`
`readEntity()` method, we can extract the validation error and get the validation
message. On running the sample, the following message will be shown:

"There was 1 error when validating the request

Book does not exist for the ISBN requested"

Summary

This chapter started with a brief introduction to REST and the key principles of
RESTful Web Services development, followed by converting a POJO to a JAX-RS
resource, a RESTful endpoint along with discussing different HTTP verbs and
their use.

After the introduction, the chapter dives deeper into the JAX-RS API by introducing
the client API to send requests to the resources developed using the JAX-RS APIs.
We also covered customizing the entity providers to produce different output
formats using `MessageBodyReader` and `MessageBodyWriters`. We learned how to
validate JAX-RS 2.0 resources using Bean Validation.

In the next chapter, we will cover the different polling techniques, compare and
contrast them with Server-sent events (SSE) and **WebSockets**, followed by a closer
look at how Java EE 7 provides support for SSE and WebSockets.

2
WebSockets and Server-sent Events

With the advancements in web architecture and emerging platforms, which can provide real-time or near real-time information, the necessity of having an effective way of communicating these updates to clients caused the urge of introducing new programming models and new standards that make use of this real-time information easier for the consumer side of the system, also known as clients (mostly web browsers).

In this chapter we will cover the following topics:

- The programming models and solutions that can be used to address near real-time update transfer to clients
- Using Server-sent Events (SSE)
- Using WebSockets

Different snippets are included in this chapter but complete samples which shows these snippet in actions are included as part of the book's source code download bundle.

The programming models

In this section we will cover the different programming models that emerged to address the near real-time updating of the client view based on the updates that are produced by the server.

Polling

As mentioned before, HTTP, which is the foundation of communication over the Internet, uses a simple request/response model in which a request either timeouts or get a response back from the server. The response can be the actual response the request was intended for or it can be an error message, underneath one of the standard error status codes. The client always initiates the communication; the server cannot initiate a communication channel without receiving a request from a client to send back a response.

So, basically, to update the client it is required to check for the new updates on the server and if an update is available the client can react to the update and, for example, change a text to denote that a book that was not available is available for borrowing now or a show a new image, or to perform any other action that maps to the response received from the server.

Sending periodical requests to a server to check for updates is called **polling**. It does not scale to hundreds of thousands of clients and thus it cannot be an effective programming model to handle the massive client numbers of today's applications. In the polling model the response does not necessarily includes updates generated in the server but rather it may just be a `200 OK` response without any particular updates for the client to use. In this model, tens of requests may receive nothing but `200 OK` without any meaningful update for the client, which means these tens wasted the resources in vain. Of course, this model is useful if the number of clients is limited and if severe compatibility issues exist that prevent the clients to update to newer versions, for example, very old browsers. The following diagram shows the polling model:
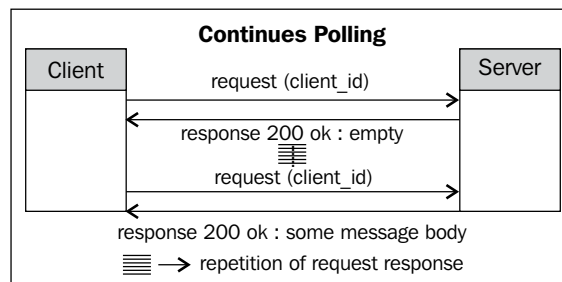
The polling-based model is enriched with JavaScript in the client side; the browsers, to update the view without changing the page and thus list available books in a library application, can change without the user refreshing the page. The following code snippet shows the server side of a polling pair written in Java:

```java
public class PollingServlet extends HttpServlet {

    @Override
```

```
    protected void doGet(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException {
        response.setContentType("text/plain");
        response.setCharacterEncoding("UTF-8");
        response.getWriter().write((new Date()).toString());
    }
}
```

The preceding sample does not use JSON for data format or any of the new Async functionalities introduced in Java EE 7 and Servlet 3.1 which are discussed in *Chapter 3, Understanding WebSockets and Server-sent Events in Detail* and *Chapter 4, JSON and Asynchronous Processing*, but rather shows how the basics work. The Servlet writes the current date on the response for any GET request. The servlet is mapped to PollingServlet path in the web.xml file.

The following code snippet shows how we can use JavaScript to perform a request in the background, get the response, and update content of a div element of the HTML page by manipulating the equivalent DOM tree element.

```
<html>
 <head>
   <title>Ajax Continues Polling</title>
     <script>
       function startUpdating() {
         req = new XMLHttpRequest();
         req.onreadystatechange = function() {updateDiv();};
         req.open("GET", "/PollingServlet", true);
         req.send(null);
         results = req.responseText;
       }

       function updateDiv(){
         results = req.responseText;
         document.getElementById("dateDiv").innerHTML = results;
         setTimeout("startUpdating()", 5000);
       }
     </script>
  </head>
  <body onload="startUpdating()">
    <p>polling the time from server:</p>
    <div id="dateDiv"></div>
  </body>
</html>
```

The HTML page in the preceding snippet is the simplest possible form; it does not check whether the response for each request is OK neither does it check whether the code is being executed in IE or a non-IE browser for the sake of simplicity.

Now, the startUpdating function is invoked when the page is loaded, the function sends a request to the Servlet shown before and invokes the updateDiv function to update the GUI and then schedule another invocation of it after 5 seconds.

The limitations of the polling model can be summarized as follows:

- It is the client that performs polling and there is no service pushing involved
- It is resource-consuming as many requests will result in a response that does not have any effective update for the client
- Long intervals between requests may yield an outdated client view while short intervals overload the servers

Long polling

With the limitation of the polling model in mind, a new programming model emerged in which a request either timeouts or carries back useful updates to the client. In this model a request is sent to the server and the request is set to a timeout after a very long time so that the cost of handshake and request initiation reduces as much as possible by a reduction in the number of requests and responses in a fixed period of time. A response is only sent back if there are some updates in the server, which the client should receive. When such an update becomes available in the server, the server sends back the response with the update and client initiates another request after consuming the update it received. The benefit of this model is fewer numbers of requests compared to polling, which reduces the resource consumption and increases the scalability. The following diagram shows the **long polling** model.
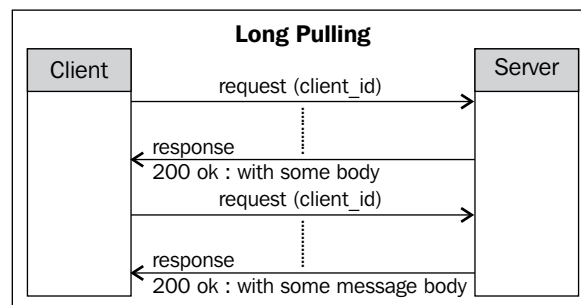
Long polling clients and XMLHttpRequest

The `XMLHttpRequest` object is available as part of the JavaScript in-browser API to facilitate the interaction of the JavaScript part of a web page with the web server. It can be used to send a request to the server and receive the response without refreshing the page after the page is loaded; for example, to update the list of available books after the `available.html` page is loaded.

The functionality of the `XMLHttpRequest` object is categorized in events, methods, and properties. The important properties, events, and important methods are discussed shortly.

The values of properties change when an event is fired and when methods are invoked. Checking the property values makes it possible to evaluate the current state of the `XMLHttpRequest` object or to handle the response.

- `readyState`: Stores the current state of the ongoing request. The table after the list shows different values of the `readyState` attribute.

- `responseText`: Returns the response text.

- `responseXML`: Returns a DOM object representing the response data. The assumption is that the response text is a valid XML document. The XML document can be traversed using standard DOM parser methods.

- `status`: Shows the status code of the request; for example, `200`, `404`, and so on.

- `statusText`: Human readable text equivalent of request status code. For example `"OK"`, `"Not Found"`, and so on.

The readyState value and textual equivalent	Description
`0` (`UNINITIALIZED`)	The `XMLHttpRequest` object is created, but not opened.
`1` (`LOADING`)	The `XMLHttpRequest` object is created, the open method is called but no request is sent.
`2` (`LOADED`)	The send method is called, no response yet.
`3` (`INTERACTIVE`)	The send method is called, some data was received but the response is not concluded yet.
`4` (`COMPLETED`)	Response is concluded and the entire message is received. Message content is available in the `responseBody` and `responseText` properties.

Each event can have an associated method, which is invoked when the event is fired. The sample code afterward shows how these events can be used.

- `onreadystatechange`: This event is fired when the state of a request initiated by this `XMLHttpRequest` instance is changed. The state change is communicated via the `readyState` property.

- `ontimeout`: This event is fired when a request initiated with this `XMLHttpRequest` instance is timed out.

After invocation of each method the value of the relevant properties will change.

- `abort`: This aborts the current request of the `XMLHttpRequest` instance. The `readyState` value is set to `0`.

- `open`: This prepares a request by setting the method, URL and security credentials.

- `send`: This sends the request that is prepared by the `open` method.

Chunked transfer encoding

Another possible way of using the long polling model is to use the message body streaming to send chunks of data and update events, when those chunks are available in the server and ready to be consumed by the developer. In the message body streaming model the server does not close the response but rather keeps it open and sends the update events to client as they are produced in the server. The message body streaming involves using the chunked transfer encoding which is a `HTTP/1.1` feature.

The chunked transfer encoding can be used to send many chunks of data as part of the response body, which is opened as a stream. The chunks can be JavaScript tags which are loaded in the hidden `iframe` and executed in the order of arrival. The execution of arriving scripts can cause the view to update or to trigger any other action that is required. The following diagram shows the long polling in action.
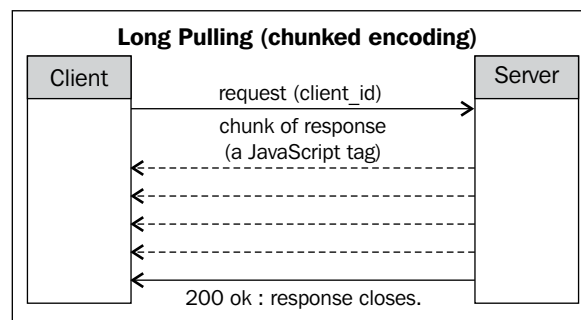
In the preceding diagram the client sent a request along with the `client_id` value to the server and the server started sending chunks of responses when some updates are available to be sent to the client. The updates are sent as JavaScript tags which are then executed in the client's browser to update the GUI.

The limitations of the long polling model can be summarized as follows:

- One-way communication
- No standard data format or message format when used in chunked transfer encoding mode
- One response per request when no `iframe` technique is used
- Each connection initiation has an initiation cost
- No caching between the clients and server, which impacts server performance instead of reading content from the cache

Emerging standards

With the emergence of requirements and solutions for those requirements, standards emerged to ensure compatibility between different the layers, applications, and components that form a solution; asynchronous communication, and especially event propagation between clients and servers, is one.

Server-sent Events

Server-sent Events (**SSE**), or sometimes simply referred to as **EventSource**, is an HTML5 browser API that makes event pushing between server and client available to web application developers. The SSE component provides a structured mechanism for having a capability similar to long polling without some of the long polling drawbacks. As it is an HTML5 component, the browser should support HTML5 SSE to be able to take advantage of this API.

The SSE kernel includes `EventSource` and `Event`.

`EventSource` is the API that provides the client with the means of subscribing to an event source, which can be a Servlet or anything of that sort. After subscription, which is nothing more than opening the connection to the URL, events are sent to the client in the order that they are produced and in the client the event listener can react to the events, for example by updating a chat window, changing a graph, or updating the list of available books to borrow or list people that are interested in the subject that the event URL is meant for.

The SSE anatomy

Before we go deep into the API and see how the API works, it is good to look more closely at the characteristics of SSE and how SSE works. The following diagram shows SSE in action, which closely resembles the chunked encoding diagram shown in the preceding section. The question may a rise: what makes SSE better than long polling if both of them work similarly when it comes to request, response, and message content?

Structured response messages
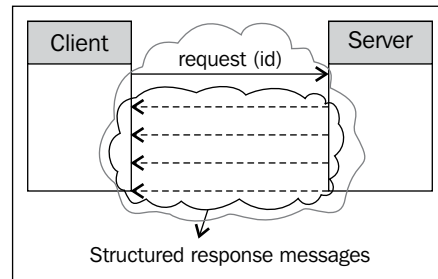
With SSE the events are plain text messages sent from the server to the clients after the client opens the initial request meaning that it does not require to be a collection of JavaScript tags that need to be executed in the client side to update something but rather it can be data that can be consumed in the client side by the event listener and event listener can interpret and react to the received event.

The second difference is the message format; SSE defines a message format for the events that are sent from the server to the clients. The message format is composed of a plain text line-separated stream of characters. Lines that carry the message body or data start with `data:` and lines that carry some **Quality of Service** (**QoS**) directives start with the QoS attribute name followed by a colon and then the QoS attribute's value, `directive: value`. The standard format makes it possible to develop generic libraries around SSE to make software development easier. The following snippet shows a sample message that can indicate a new dot in a graph. When the client receives the message it can draw the new dot on the graph to show a change in the data the graph is being constructed from. The following sample data shows a multiline message in which each line is separated from the next using `\n` and end of message is marked with `\n\n`.

```
data: Position: 75,55\n\n
data: Label: Large increase\n\n
data: Color: red\n\n
```

It is possible to develop the server component of a SSE solution using a servlet and the client side can be developed using either JavaScript or Java API. The Java API to consume SSE events is part of the Java EE 7 provided by means of JAX-RS 2.0. The next two sections go into details of the client side API and also the server side component of the solution, which is a servlet.

As mentioned earlier, in addition to the actual message or message body each SSE message can carry some directive, which instructs the browser or SSE-compatible client on some of the QoS attributes of the interaction. Some of these QoS directives are discussed next.

> The reference implementation of JAX-RS 2.0 is done under the Jersey 2.0 project. The Jersey project is located at `http://jersey.java.net/` with extensive documentation.

Associating an ID with an event

Each SSE message can have a message identifier, which can be used for a variety of purposes; one use of the message ID standard usage is to keep track of the messages that the client has received. When a message ID is used in SSE, the client can supply the last message ID as one of the connection parameters to instruct the server to resume form and specific message onward. Of course the server should implement proper the procedure for resuming a communication from where a client requests it.

An example message format with message ID can be as shown in the following code snippet:

```
id: 123 \n
data: single line data \n\n
```

Connection loss and reconnecting retries

Browsers that support SSE, which are listed early in this section, can try reconnecting to the server in case the connection between browser and server is severed. The default retry interval is `3000` milliseconds but it can be adjusted by including the `retry` directive in the messages that the server sends to the client. For example, to increase the retry interval to `5000` milliseconds the SSE message that the server sends can be similar to the following code snippet:

```
retry: 5000\n
data: This is a single line data\n\n
```

Associating event names with events

Another SSE directive is the event name. Each event source can generate more than one type of event and the client can decide how to consume each event type based on what event type it subscribes to. The following snippet shows how event name directives incorporate into the message:

```
event: bookavailable\n
data: {"name" : "Developing RESTful Services with JAX-RS 2.0,
WebSockets and JSON"}\n\n
event: newbookadded\n
data: {"name" :"Netbeans IDE7 Cookbook"}\n\n
```

Server-sent Events and JavaScript

The major SSE API that is considered the foundation of SSE in the client side for JavaScript developers is the `EventSource` interface. The `EventSource` interface contains a fair number of functions and attributes but the most important ones are listed as follows:

- **The addEventListener function**: To add an event listener to handle the incoming events based on event type.
- **The removeEventListener event function**: To remove an already registered listener.
- **The onmessage event function**: It is invoked on message arrival. There is no custom event handling available when using the `onmessage` method. Listeners manage the custom event handling.
- **The onerror event function**: It is invoked when something goes wrong with the connection.
- **The onopen event function**: It is invoked when a connection is opened.
- **The close function**: It is invoked when a connection is closed.

The following snippet shows how to subscribe for different event types omitted by one source. The snippet assumes that the incoming messages are JSON-formatted messages. The `'bookavailable'` listener uses a simple JSON parser to parse the incoming JSON and then will use that to update the GUI while the `'newbookadded'` listener uses the reviver function to filter out and selectively process the JSON pairs.

```
var source = new EventSource('books');
source.addEventListener('bookavailable', function(e) {
  var data = JSON.parse(e.data);
  // use data to update some GUI element...
}, false);
```

```
source.addEventListener('newbookadded', function(e) {
  var data = JSON.parse(e.data, function (key, value) {
    var type;
    if (value && typeof value === 'string') {
return "String value is: "+value;
    }
    return value;
});
}, false);
```

Before we move to WebSockets as another emerging technology let's take a look at the following paired server and client, which are written as a Java EE Servlet and JavaScript to see how SSE works:

Servlet's `processRequest` function look like the following snippet:

```
    protected void processRequest(HttpServletRequest request,
HttpServletResponse response)
            throws ServletException, IOException {
                response.setContentType("text/event-stream");
                response.setCharacterEncoding("utf-8");

        PrintWriter out = response.getWriter();
        while(true){
                Date serverDate = new Date();
                out.write( "event:server-time\n");
                out.write( "data:<b>Current Server Time is:" +
serverDate.toString() +"</b>\n\n");
                out.flush();
                try {
                        Thread.sleep(1000);
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
        }
    }
```

The preceding Servlet writes out the current date every one second and, if the browser hits the Servlet's URL, the output should be similar to the following figure.
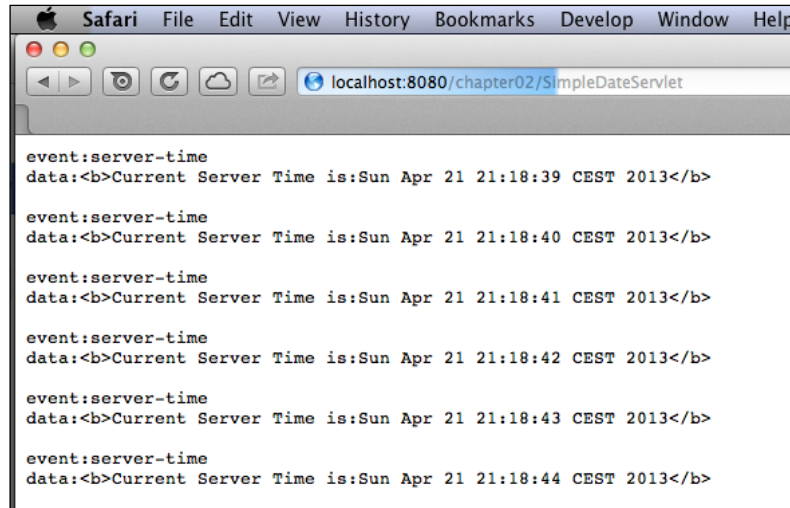
And the JSP page in the same web application look like the following:

```jsp
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
  <head>
    <title>JSP Page With SSE EventSource</title>
      <script type="text/JavaScript">
        function startSSEConnection(){
        var source = new EventSource('SimpleDateServlet');
        source.addEventListener("server-time", function(event){
         document.getElementById("server-time").innerHTML=event.data;
         },false);
       }

      </script>

      </script>
    </head>
    <body onload="startSSEConnection();">
      <div id="server-time">[Server-Time]</div>
    </body>
</html>
```
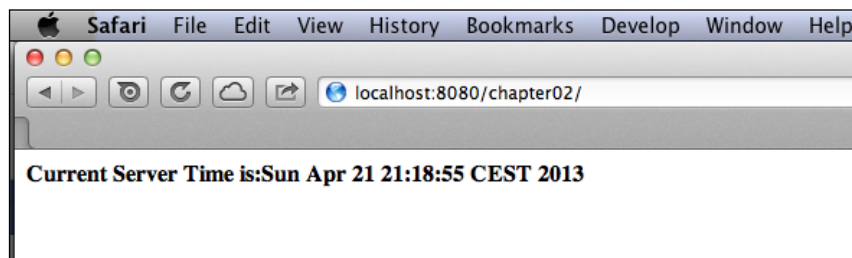
Checking the JSP page's URL will show an output similar to the following screenshot. As you can see, the Servlet's output messages are shown with formatting as it is specified in the JavaScript code in the JSP page:

More complete and advanced examples are included in *Chapter 3*, *Understanding WebSockets and Server-sent Events in Detail* and *Chapter 5*, *RESTful Web Services by Example*. The complete code for the preceding example is included in the book's code bundle.

WebSockets

The WebSockets component of HTML5 adds a brand new method for interaction between clients and servers to address the scalability and flexibility required for modern web-scale applications by introducing a full duplex event-based communication channel between clients and servers. After being initiated by the client, the server can send binary and textual data concerning the client over the channel and the client can, without reinitiating a connection, send messages to the server. The event source and event subscription model discussed in the *Server-sent Events* section is available in WebSockets as well.

WebSocket handshake

There is an optional handshake request-response devised to let the applications switch to WebSockets when required. In a sample scenario shown below, the client requests the protocol upgrade to WebSockets by sending the upgrade request header to the server. If the server supports the upgrade the response will include the protocol upgrade as shown afterward.

The client request to upgrade to WebSockets looks as shown in the following code snippet:

```
GET /demo HTTP/1.1
Host: mybookstoresample.com
Connection: Upgrade
Upgrade: WebSocket
Origin: http://mybookstoresample.com
```

And the server response handshake can look as shown in the following snippet:

```
HTTP/1.1 101 WebSocket Protocol Handshake
Upgrade: WebSocket
Connection: Upgrade
```

After the handshake is completed the communication between the client and server occurs over a bidirectional socket. The WebSockets wire level communication protocol is different than HTTP wire protocol and because of that it is possible that intermediate servers like proxy servers or cache servers are not capable or intercepting and processing the WebSockets messages as they do with HTTP messages.
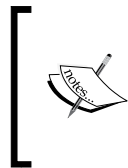
> In *Chapter 3*, *Understanding WebSockets and Server-sent Events in Detail* in the WebSockets section you can learn more details on the WebSockets client and server implementation along with details on protocol upgrade. *Chapter 5*, *RESTful Web Services by Example*, includes complete sample applications that further dive into using WebSockets.

Browser and JavaScript support for WebSockets

New versions of major web browsers support WebSockets and using WebSockets in the client side just involves creating a WebSockets object and then setting different listeners and event handlers for different events. The following list shows the important functions and attributes of the `WebSocket` class:

- **The constructor**: To initialize the `WebSocket` object, the resource URL is enough to be passed to the `WebSocket` constructor
- **The send function**: The `send` function can be used to send a message to the server's specified URL during the object construction.
- **The onopen event function**: This function is invoked when the connection is created. The `onopen` handles the `open` event type.
- **The onclose event function**: The function is invoked when the connection is being closed. The `onclose` handles the `close` event type.

- **The onmessage event function**: When a new message arrives, the `onmessage` function is invoked to handle the `message` event.

- **The onerror event function**: The function is invoked to handle the `error` event when an error in the communication channel occurs.

- **The close function**: To close the communication socket and end the interaction between the client and the server.

A very basic example of using the JavaScript WebSocket API is shown below:

```
//Constructionof the WebSocket object
var websocket = new WebSocket("books");
//Setting the message event Function
websocket.onmessage = function(evt) { onMessageFunc(evt) };
//onMessageFunc which when a message arrives is invoked.
function onMessageFunc (evt) {
//Perform some GUI update depending the message content
}
//Sending a message to the server
websocket.send("books.selected.id=1020");
//Setting an event listener for the event type "open".
addEventListener('open', function(e){
        onOpenFunc(evt)});

//Close the connection.
websocket.close();
```

An example server-side component, a WebSockets endpoint, is shown in the following code snippet:

```
@ServerEndpoint(
decoders = BookDecoder.class,
encoders = BookEncoder.class,
path = "/books/")
public class BooksWebSocketsEndpoint {
@OnOpen
public void onOpen(Session session) {
            }

@OnMessage
public void bookReturned(Library.Book book, Session session) {
                }

@OnClose
public void onClose(Session session){
sessionToId.remove(session);
}
}
```

Details of how the implementation of a WebSockets endpoint looks is included in *Chapter 3, Understanding WebSockets and Server-sent Events in Detail* and *Chapter 5, RESTful Web Services by Example*.

Java EE and the emerging standards

Java EE has always been an adopter of emerging standards and features and capabilities, which were required by the Java EE community. Starting from Java EE 6, Java EE spec leads focused their attention on the emerging standards and in Java EE 7 full support for HTML5, SSE and WebSockets is included in the spec; thus any Java EE application server can host a WebSockets, SSE, and HTML5-oriented application without any compatibility issue at the server side.

Java EE and Server-sent Events

For the SSE, which is an HTML5 browser API component, the server side can be a Servlet that produces SSE messages according to the SSE message format or it can be a SSE resource which is POJO annotated with `@Path`. In the client side, JavaScript can be used as the standard in-browser API to consume the SSE events or it can be developed using the SSE client side API introduced in Jersey 2.0 if a Java-based client is required.

The following table shows the important classes and interfaces that are entry points to SSE APIs included in Jersey 2.0:

Class	Description
Broadcaster	Used for broadcasting SSE to multiple `EventChannel` instances.
OutboundEvent	This is the outgoing event class to send the Server-sent Events. An `OutboundEvent` can have id, name, date, and comment associated with it.
EventChannel	This is the outgoing event message channel. When returned from resource method, the underlying connection is kept open and the application is able to send events. One instance of this class corresponds with exactly one HTTP connection.
EventSource	This is the client for reading and processing Server-sent `InboundEvents`
InboundEvent	This represents an incoming event.
ClientFactory	This is the main entry point to the client API used to bootstrap client instances. For example: ```\nClient client = ClientFactory.newClient();\nWebTarget webTarget= client.target(new\nURI(TARGET_URI)) ;\n```

Class	Description

Client	Client is the main entry point to the fluent API used to build and execute client requests in order to consume responses returned.
	
	`Client client = ClientFactory.newClient();`
	`WebTarget webTarget= client.target(new URI(TARGET_URI)) ;`

ResourceConfig	This encapsulates the configuration for configuring a web application.

The following table shows important annotations included in Java EE 7 and used throughout this book for developing SSE applications:

Annotation	Description

@Path	To annotate a POJO with the resource path it represent. For example `@Path("books")` or to annotate a sub-resource which is a method in the annotated class. For example `getBook` with related parameters for that method along with validation expression for the method parameters. For example:
	
	`@Path("{id: ^\d{9}[\d|X]$}")`
	
	`getBook(@PathParam("id") String isbn10)`

@Produces	To specify the type of output that the resource produces or in a narrower scope the type of output that a method in a resource produces. For example: `@Produces(MediaType.APPLICATION_JSON)`
@Consumes	To specify the type of input that the resource consumes or in a narrower scope the type of input that a method in a resource consumes. For example: `@Consumes (MediaType.APPLICATION_JSON)`
@GET @POST @DELETE	To map the HTTP methods to methods in the resource representing class. For example `@GET` can be placed on the `getBook` method
@PathParam	To specify the mapping between the query parameter's name and method. For example: `getBook(@PathParam("id") String isbn10)`
@ApplicationPath	Identifies the application path that serves as the base URI for all resource URIs provided by `@Path`. For example, `@ApplicationPath("library")` for the library application.
@Context	This can be used to inject contextual objects such as `UriInfo`, which provides contextual request-specific information about the request URI. For example: `getBook(@Context UriInfo uriInfo)`

Chapter 3, *Understanding WebSockets and Server-sent Events in Detail*, is dedicated to annotations; it explains how to use these annotations and more advanced features of Server-sent Events and *Chapter 5*, *RESTful Web Services by Example*, includes complete examples of how Server-sent Events and WebSockets work in real use cases.

Java EE and WebSockets

In Java EE 7, there is a new JSR to support WebSockets in Java EE container. JSR-356 defines the requirements and the API that a Java EE application server provides to develop WebSockets-based applications. The important annotations provided for WebSockets development are included in the following table:

Annotation	Description
@ClientEndpoint	A class-level annotation that is used to denote that a POJO is a WebSocket client for instructing the server to deploy it as a managed component of that type.
@OnClose	A method-level annotation to decorate a Java method that requires to be called when a WebSocket session is closing.
@OnError	A method-level annotation to decorate a Java method that requires to be called in order to handle connection errors.
@OnMessage	A method-level annotation to mark a Java method as WebSocket message receiver.
@OnOpen	A method level annotation to decorate a Java method that should be called when a new WebSocket session is open.
@PathParam	To specify the mapping between the query parameter's name and method. For example: `getBook(@PathParam("id") String isbn10)`
@ServerEndpoint	A class-level annotation that declares the class it decorates is a WebSocket endpoint that will be deployed and made available in the URI-space of a WebSocket server. For example: `@ServerEndpoint("/books ");  public class Books {…}`

The following table shows the important classes and interfaces that are used throughout the book when WebSockets is discussed:

Class	Description
`Encode(and subintefaces and subclasses)`	Defines how to map a WebSocket message to a Java object.
`Decoder(and subintefaces and subclasses)`	Defines how to map a Java object to a WebSocket message.
`Session`	A WebSocket session represents a conversation between two WebSocket endpoints. As soon as the WebSocket handshake completes successfully, the WebSocket implementation provides the endpoint with an open WebSocket session.

Comparison and use cases of different programming models and standards

The following table shows a comparison and conclusion of how the three major techniques and standards described in this chapter compare to each other:

Subject	SSE	WebSockets	Long polling
Error handling	Build-in support for error handling	Build-in support for error handling	Almost no error handling in case of chunked transfer
Performance	Usually results are better than long polling and inferior to WebSockets	Best performance result compared to the other two solutions	Small CPU resource but idle process/ thread per client connection, limits scalability and extensive memory usage
Browser support1,2	Firefox, Chrome, Safari, Opera	For RFC 6455: IE 10, Firefox 11, Chrome 16, Safari 6, Opera 12.10	All current browsers support this

Subject	SSE	WebSockets	Long polling
Browser performance	Built-in support in browser, small amount of resources	Built-in support in browser, small amount of resources	Complicated to get the performance right specially with lots of JavaScripts and possible memory leaks
Communication channel	HTTP unidirectional	WebSockets bidirectional	HTTP unidirectional
Implementation complexity	Easy	Requires server with WebSockets support	Easiest

For more details visit `http://en.wikipedia.org/wiki/WebSocket#Browser_support` and `http://en.wikipedia.org/wiki/Server-sent_events#Web_browsers`.
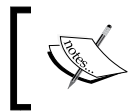
> Reading the *Memory leak patterns in JavaScript* article available at `http://www.ibm.com/developerworks/web/library/wa-memleak/` is recommended to avoid JavaScript memory leaks pitfalls.

The following list shows which types of use case match with one of the programming models and standards:

- Long polling: When compatibility is an issue and browsers are not up-to-date (usually for enterprise users who stick with approved versions of software for many years)

- **SSE**: When the communication is one way and server requires sending events to browser so browser can update some GUI elements. It provides error handling and structured message format advantage over long polling. Sample use cases include:
    - A chart that updates in real-time
    - A newsreader that shows the latest headlines
    - Stock tickers reader

- WebSockets: When full duplex, bi-directional communication is required between the client and the server. Some sample applications are as follows:
    - A chat application
    - A real-time interactive multiuser charting and drawing application
    - Multiuser browser-based games

- WebSockets provide all benefits and advantages of SSE with some disadvantages that are listed as follows:

  ° The wire protocol is different so some intermediate servers, such as proxy servers, may not be able to intercept and interpret the messages.

  ° If a browser does not support WebSockets there is no way to make the browser handle the communication while in the case of SSE the browser can use JavaScript libraries to handle SSE communication, polyfilling the browser. For example, **Remy Polyfill**.

  ° Lack of support for event ID.

> A good write-up to further understand the Polifill can be found at `http://remysharp.com/2010/10/08/what-is-a-polyfill/`.

Summary

This chapter was the opening door to the whole world of asynchronous Web by introducing the fundamental concepts involving in web architecture and going forward with the evolution of the basic request response model to polling, long polling, Server-sent Event, and WebSockets.

In the next chapter WebSockets and Server-sent Events are covered in details. *Chapter 5*, *RESTful Web Services by Example*, has complete sample application developed using WebSockets and Server-sent Events.

3
Understanding WebSockets and Server-sent Events in Detail

WebSocket is one of the most promising features that HTML5 has to offer. As covered in *Chapter 2*, *WebSockets and Server-sent Events*, the traditional request-response model incurred an overhead due to the HTTP headers. With WebSockets, once the initial handshake is done the client and server or peers can communicate directly without the use of headers. This reduces the network latency and gives a reduction in HTTP header traffic.

Chapter 2, *WebSockets and Server-sent Events*, also introduced Server-sent Events and provides a comparison between SSE and WebSockets.

Server-sent Events define an API where the server communicates and pushes events to the clients as they occur. It is a one-directional communication from the server to the client and has more benefits as compared to traditional polling and long polling techniques.

This chapter covers advanced concepts of WebSockets and Server-sent Events and covers the following sections:

- Encoders and decoders in Java API for WebSockets
- Java WebSockets Client API
- Sending different types of data such as Blob and Binary using Java API for WebSockets
- Security and WebSockets
- Best practices for WebSockets-based applications
- Developing Server-sent Events clients using Jersey API
- Best practices for Server-sent Events

Encoders and decoders in Java API for WebSockets

As seen in the previous chapter, the class-level annotation `@ServerEndpoint` indicates that a Java class is a WebSocket endpoint at runtime. The value attribute is used to specify a URI mapping for the endpoint. Additionally the user can add encoder and decoder attributes to encode application objects into WebSocket messages and WebSocket messages into application objects.

The following table summarizes the `@ServerEndpoint` annotation and its attributes:

Annotation	Attribute	Description
`@ServerEndpoint`		This class-level annotation signifies that the Java class is a WebSockets server endpoint.
	`value`	The value is the URI with a leading `'/.'`
	`encoders`	Contains a list of Java classes that act as encoders for the endpoint. The classes must implement the Encoder interface.
	`decoders`	Contains a list of Java classes that act as decoders for the endpoint. The classes must implement the Decoder interface.
	`configurator`	The configurator attribute allows the developer to plug in their implementation of `ServerEndpoint.Configurator` that is used when configuring the server endpoint.
	`subprotocols`	The sub protocols attribute contains a list of sub protocols that the endpoint can support.

In this section we shall look at providing encoder and decoder implementations for our WebSockets endpoint.
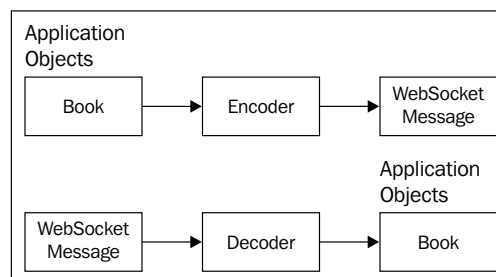
The preceding diagram shows how encoders will take an application object and convert it to a WebSockets message. Decoders will take a WebSockets message and convert to an application object. Here is a simple example where a client sends a WebSockets message to a WebSockets java endpoint that is annotated with @ServerEndpoint and decorated with encoder and decoder class. The decoder will decode the WebSockets message and send back the same message to the client. The encoder will convert the message to a WebSockets message. This sample is also included in the code bundle for the book.

Here is the code to define ServerEndpoint with value for encoders and decoders:

```
@ServerEndpoint(value="/book", encoders={MyEncoder.class}, decoders =
{MyDecoder.class} )
public class BookCollection {
    @OnMessage
    public void onMessage(Book book,Session session) {

        try {
session.getBasicRemote().sendObject(book);
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }

    @OnOpen
        public void onOpen(Session session) {
            System.out.println("Opening socket" +session.
getBasicRemote() );
        }

        @OnClose
        public void onClose(Session session) {
            System.out.println("Closing socket" + session.
getBasicRemote());
        }
}
```

In the preceding code snippet, you can see the class BookCollection is annotated with @ServerEndpoint. The value=/book attribute provides URI mapping for the endpoint. The @ServerEndpoint also takes the encoders and decoders to be used during the WebSocket transmission. Once a WebSocket connection has been established, a session is created and the method annotated with @OnOpen will be called. When the WebSocket endpoint receives a message, the method annotated with @OnMessage will be called. In our sample the method simply sends the book object using the Session.getBasicRemote() which will get a reference to the RemoteEndpoint and send the message synchronously.

Encoders can be used to convert a custom user-defined object in a text message, `TextStream`, `BinaryStream`, or `BinaryMessage` format.

An implementation of an encoder class for text messages is as follows:

```java
public class MyEncoder implements Encoder.Text<Book> {
    @Override
        public String encode(Book book) throws EncodeException {
            return book.getJson().toString();
        }
}
```

As shown in the preceding code, the encoder class implements `Encoder.Text<Book>`. There is an encode method that is overridden and which converts a book and sends it as a JSON string. (More on JSON APIs is covered in detail in the next chapter)

Decoders can be used to decode WebSockets messages in custom user-defined objects. They can decode in text, `TextStream`, and binary or `BinaryStream` format.

Here is a code for a decoder class:

```java
public class MyDecoder implements Decoder.Text<Book> {
    @Override
    public Book decode(String string) throws DecodeException {
            javax.json.JsonObject jsonObject = javax.json.Json.
createReader(new StringReader(string)).readObject();
        return new Book(jsonObject);
    }
    @Override
    public boolean willDecode(String string) {
        try {
            javax.json.Json.createReader(new StringReader(string)).
readObject();
            return true;
        } catch (Exception ex) { }
        return false;
        }
```

In the preceding code snippet, the `Decoder.Text` needs two methods to be overridden. The `willDecode()` method checks if it can handle this object and decode it. The `decode()` method decodes the string into an object of type `Book` by using the JSON-P API `javax.json.Json.createReader()`.

The following code snippet shows the user-defined class `Book`:

```
public class Book {
    public Book() {}
    JsonObject jsonObject;
    public Book(JsonObject json) {
        this.jsonObject = json;
    }
    public JsonObject getJson() {
        return jsonObject;
    }
    public void setJson(JsonObject json) {
        this.jsonObject = json;
    }

    public Book(String message) {
        jsonObject = Json.createReader(new StringReader(message)).
readObject();
    }
    public String toString () {
        StringWriter writer = new StringWriter();
        Json.createWriter(writer).write(jsonObject);
        return writer.toString();
    }
}
```

The `Book` class is a user-defined class that takes the JSON object sent by the client. Here is an example of how the JSON details are sent to the WebSockets endpoints from JavaScript.

```
var json = JSON.stringify({
                "name": "Java 7 JAX-WS Web Services",
            "author":"Deepak Vohra",
            "isbn": "123456789"
            });
function addBook() {
            websocket.send(json);
        }
```

The client sends the message using `websocket.send()` which will cause the `onMessage()` of the `BookCollection.java` to be invoked. The `BookCollection. java` will return the same book to the client. In the process, the decoder will decode the WebSockets message when it is received. To send back the same `Book` object, first the encoder will encode the `Book` object to a WebSockets message and send it to the client.

The Java WebSocket Client API

Chapter 2, *WebSockets and Server-sent Events*, covered the Java WebSockets client API. Any POJO can be transformed into a WebSockets client by annotating it with `@ClientEndpoint`.

Additionally the user can add encoders and decoders attributes to the `@ClientEndpoint` annotation to encode application objects into WebSockets messages and WebSockets messages into application objects.

The following table shows the `@ClientEndpoint` annotation and its attributes:

Annotation	Attribute	Description
`@ClientEndpoint`		This class-level annotation signifies that the Java class is a WebSockets client that will connect to a WebSockets server endpoint.
	`value`	The value is the URI with a leading /.
	`encoders`	Contains a list of Java classes that act as encoders for the endpoint. The classes must implement the encoder interface.
	`decoders`	Contains a list of Java classes that act as decoders for the endpoint. The classes must implement the decoder interface.
	`configurator`	The configurator attribute allows the developer to plug in their implementation of `ClientEndpoint.Configurator`, which is used when configuring the client endpoint.
	`subprotocols`	The sub protocols attribute contains a list of sub protocols that the endpoint can support.

Sending different kinds of message data: blob/binary

Using JavaScript we can traditionally send JSON or XML as strings. However, HTML5 allows applications to work with binary data to improve performance. WebSockets supports two kinds of binary data

- Binary Large Objects (`blob`)
- `arraybuffer`

A WebSocket can work with only one of the formats at any given time.

Using the `binaryType` property of a WebSocket, you can switch between using blob or `arraybuffer`:

```
websocket.binaryType = "blob";
// receive some blob data

websocket.binaryType = "arraybuffer";
// now receive ArrayBuffer data
```

The following code snippet shows how to display images sent by a server using WebSockets.

Here is a code snippet for how to send binary data with WebSockets:

```
websocket.binaryType = 'arraybuffer';
```

The preceding code snippet sets the `binaryType` property of `websocket` to `arraybuffer`.

```
websocket.onmessage = function(msg) {
        var arrayBuffer = msg.data;
        var bytes = new Uint8Array(arrayBuffer);

        var image = document.getElementById('image');
        image.src = 'data:image/png;base64,'+encode(bytes);
    }
```

When the `onmessage` is called the `arrayBuffer` is initialized to the `message.data`. The `Uint8Array` type represents an array of 8-bit unsigned integers. The `image.src` value is in line using the data URI scheme.

Security and WebSockets

WebSockets are secured using the web container security model. A WebSockets developer can declare whether the access to the WebSocket server endpoint needs to be authenticated, who can access it, or if it needs an encrypted connection.

A WebSockets endpoint which is mapped to a `ws://` URI is protected under the deployment descriptor with `http://` `URI` with the same `hostname,port` path since the initial handshake is from the HTTP connection. So, WebSockets developers can assign an authentication scheme, user roles, and a transport guarantee to any WebSockets endpoints.

We will take the same sample as we saw in *Chapter 2*, *WebSockets and Server-sent Events*, and make it a secure WebSockets application.

Here is the `web.xml` for a secure WebSocket endpoint:

```xml
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://
java.sun.com/xml/ns/javaee/web-app_3_0.xsd">

    <security-constraint>
        <web-resource-collection>
            <web-resource-name>BookCollection</web-resource-name>
            <url-pattern>/index.jsp</url-pattern>
            <http-method>PUT</http-method>
            <http-method>POST</http-method>
            <http-method>DELETE</http-method>
            <http-method>GET</http-method>
        </web-resource-collection>
            <user-data-constraint>
            <description>SSL</description>
            <transport-guarantee>CONFIDENTIAL</transport-guarantee>
        </user-data-constraint>
    </security-constraint>
</web-app>
```

As you can see in the preceding snippet, we used `<transport-guarantee>CONFIDENTIAL</transport-guarantee>`.

The Java EE specification followed by application servers provides different levels of transport guarantee on the communication between clients and application server. The three levels are:

- **Data Confidentiality (CONFIDENTIAL)**: We use this level to guarantee that all communication between client and server goes through the SSL layer and connections won't be accepted over a non-secure channel.

- **Data Integrity (INTEGRAL)**: We can use this level when a full encryption is not required but we want our data to be transmitted to and from a client in such a way that, if anyone changed the data, we could detect the change.

- **Any type of connection (NONE)**: We can use this level to force the container to accept connections on HTTP and HTTPs.

The following steps should be followed for setting up SSL and running our sample to show a secure WebSockets application deployed in Glassfish.

1. Generate the server certificate:

   ```
   keytool -genkey -alias server-alias -keyalg RSA -keypass changeit
   --storepass changeit -keystore keystore.jks
   ```

2. Export the generated server certificate in `keystore.jks` into the file `server.cer`:

```
keytool -export -alias server-alias -storepass changeit -file
server.cer -keystore keystore.jks
```

3. Create the trust-store file `cacerts.jks` and add the server certificate to the trust store:

```
keytool -import -v -trustcacerts -alias server-alias -file server.
cer  -keystore cacerts.jks -keypass changeit -storepass changeit
```

4. Change the following JVM options so that they point to the location and name of the new keystore. Add this in `domain.xml` under `java-config`:

```
<jvm-options>-Djavax.net.ssl.keyStore=${com.sun.aas.instanceRoot}/
config/keystore.jks</jvm-options>
        <jvm-options>-Djavax.net.ssl.trustStore=${com.sun.aas.
instanceRoot}/config/cacerts.jks</jvm-options>
```

5. Restart GlassFish. If you go to `https://localhost:8181/helloworld-ws/`, you can see the secure WebSocket application.

6. Here is how the the headers look under Chrome Developer Tools:
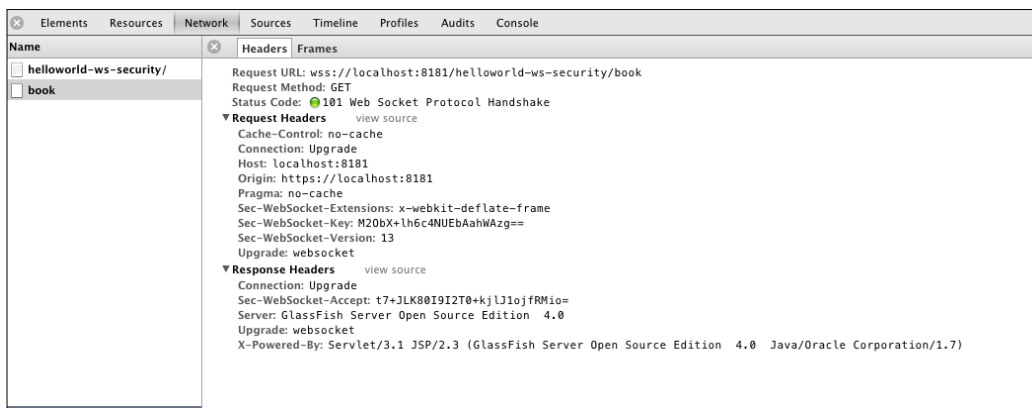
7. Open Chrome Browser and click on **View** and then on **Developer Tools**.

8. Click on **Network**.

9. Select **book** under element name and click on **Frames**.

As you can see in the preceding screenshot, since the application is secured using SSL the WebSockets URI, it also contains `wss://`, which means WebSockets over SSL.

So far we have seen the encoders and decoders for WebSockets messages. We also covered how to send binary data using WebSockets. Additionally we have demonstrated a sample on how to secure WebSockets based application. We shall now cover the best practices for WebSocket based-applications.

Best practices for WebSockets based applications

This section will cover best practices for WebSockets based applications. The following topics will be covered:

- Throttling the rate of sending data
- Controlling the maximum size of the message
- Working with proxy servers and WebSockets

Throttling the rate of sending data

After the WebSocket connection is opened, messages can be sent using the send function.

WebSockets have a `bufferedAmount` attribute that can be used to control the rate of sending data. Using the `bufferedAmount` attribute you can check the number of bytes that have been queued but not yet sent to the server.

Here is a snippet to test for the `bufferedAmount` attribute of WebSocket.

```
// This snippet checks for amount of data buffered but not sent yet
// in case it is less than a predefined THRESHOLD the webSocket
// can send the data

if (webSocket.bufferedAmount < THRESHOLD)
    webSocket.send(someData);
};
```

This can be done periodically using the `setInterval` function. As you can see, the developer can periodically check for the `bufferedAmount` attribute to see if the number of bytes in the queue to be sent to the server exceeds some threshold. In that case it should delay sending messages. Once the buffered amount is less than the threshold it should send more messages.

This is a good practice to check for the `bufferedAmount` and then send data.

Controlling the maximum size of the message

The `maxMessageSize` attribute on the `@OnMessage` annotation in Java class annotated with `@ServerEndpoint` or `@ClientEndpoint` allows the developer to specify the maximum size of message in bytes that can be handled by the `ClientEndpoint` or `ServerEndpoint`.

If the incoming message exceeds the maximum size then the connection is closed. This is a good practice to control the maximum size of a message so that the client does not deplete its resources while trying to handle a message, which it can't process.

Working with proxy servers and WebSockets

Chapter 2, *WebSockets and Server-sent Event*, covered how the WebSocket upgrade handshake looks. Not all proxy servers may support WebSockets; thus, proxy servers may not allow unencrypted WebSocket traffic to flow through. The clients use a `CONNECT` call which would never be allowed. The correct approach would be to send the request over https on the standard port `443`. Here is an example of the HTTP Connect sent by the browser client to `foo.com` on port `443`.

```
CONNECT: foo.com:443
Host: foo.com
```

Since the traffic is encrypted there is a greater chance to pass through the proxy server. Then the `CONNECT` statements will work and there will be an end-to-end encrypted tunnel for WebSockets.

The following diagram shows how clients can send HTTPS requests which get past the proxy server and firewall; the WebSocket secure scheme will work:
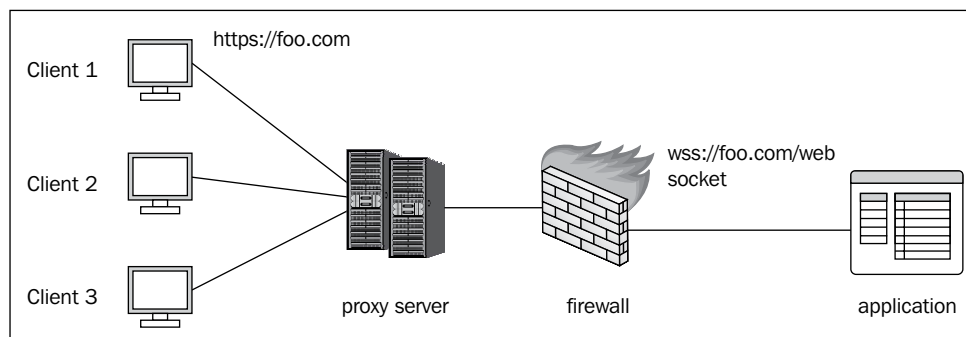
It is a good practice to use WebSocket-based applications with SSL so that the Proxy server does not impede WebSocket communication.

Server-sent Events

We covered Server-sent Events in *Chapter 2*, *WebSockets and Server-sent Events*, and compared and contrasted client/server polling alternatives as well as WebSockets. In this chapter we will cover more advanced topics such as developing a Server-sent Events client using Jersey API and best practices for Server-sent Events.

Developing a Server-sent Event client using Jersey API

Chapter 2, *WebSockets and Server-sent Events*, gave a brief introduction to the Server-sent Events and JavaScript API. In this chapter we will cover the Java Client API for Server-sent Events, which is provided by Jersey. Jersey is an implementation of JAX-RS 2.0. In addition to the features of the JAX-RS 2.0 specification, Jersey has provided support for Server-sent Events.

`EventSource` is the class for reading `InboundEvents`:

The following snippet shows how to use the `EventSource` API:

```
    WebTarget webTarget = client.target(new URI(TARGET_URI));
EventSource eventSource = new EventSource(webTarget) {
@Override
public void onEvent(InboundEvent inboundEvent) {
    System.out.println("Data " + inboundEvent.getData(String.class);

}
```

The `EventSource` object is created with `WebTarget`. We covered `WebTarget` in *Chapter 1*, *Building RESTful Web Services using JAX-RS*.

When a Server-sent Event is received by the client the `onEvent()` method of the `EventSource` is invoked. The `InboundEvent` has the `getData()` method that takes the `String.class` that is the type of the message data. You can add any custom defined class here. The JAX-RS `MessagebodyReader` will be used to read the type of the message. Thus you can see the similarity in the code between using JavaScript API and the Jersey Client API. *Chapter 5*, *Restful Web Services by Example*, will show a complete example using the Server Sent Event Jersey Client API.

Best practices for applications based on Server-sent Events

The following chapter covers the best practices for applications based on Server-sent Events. The following topics will be covered:

- Checking if the event source's origin is as expected
- Working with proxy servers and Server-sent Events
- Handling fault tolerance for Server-sent Events

Checking if the event source's origin is as expected

The following snippet shows how to check for the origin of the event source so that it matches the application's origin.

```
if (e.origin != 'http://foo.com') {
alert('Origin was not http://foo.com');
return;
```

An event stream from an origin distinct from the origin of the content consuming the event stream can result in information leakage. When the events are obtained from the server, it is good practice to check for the events originator to see if it is as expected.

Working with proxy servers and Server-sent Events

Proxy servers can drop HTTP connections after a short timeout. To avoid such dropped connections it may be a good idea to send a comment periodically.

This is how a comment is sent using Server-sent Events.

```
: this is a comment

OutboundEvent event =  new OutboundEvent.Builder().comment("this is a
comment").build();
```

The `Outboundevent.Builder` API will send a comment to the client.

The comment will fire nothing yet will make sure that connections do not get dropped between client and server.

---

[59]

Handling fault tolerance for Server-sent Events

Chapter 2, *WebSockets and Server-sent Events*, covered how you can associate IDs with events. The server can send event ids with events by using the following snippet:

```
id: 123\n
data : This is an event stream \n\n
```

The client keeps the connection alive and tries to reconnect if the connection is dropped. Setting an ID lets the browser keep track of the last event fired so when the connection between the client and server is dropped, on reconnect by the client to the server the **Last-Event-ID** will be sent back to the server. This ensures the client does not miss any messages. The server can then send events that occur after the Last-Event-ID.

The server may need a message queue to keep track of the different clients connected, check for reconnections, and send messages based on the Last-Event-ID.

Summary

In this chapter we looked at advanced topics for WebSockets and Server-sent Events. We demonstrated with code snippets how to use encoders and decoders and how to receive different kinds of data using WebSockets. We also demonstrated a sample that showed how WebSockets will work with SSL so that when working with proxy servers, the communication is encrypted.

We also discussed best practices for implementing Server-sent Events and WebSockets. We learned how to ensure messages are not lost in Server-sent Events by associating IDs with events. We covered the Jersey Client API for Server-sent Events.

In the next chapter, we will cover more advanced topics such as JSON API in Java EE and the aspects of asynchronous programming to improve scalability with respect to various Java EE specifications such as JAX-RS 2.0, EJB and Servlets.

4

JSON and Asynchronous Processing

This chapter covers a brand new JSR, The JSR 353: Java API for JSON Processing `http://jcp.org/en/jsr/detail?id=353`, and related APIs along with some updates in different services and components in Java EE that provide better support for asynchronous interaction between different components of a system. The following list shows an itemized list of topics that are covered in this chapter:

- Producing, parsing and manipulating JSON data using Java
- Introducing NIO API in Servlet 3.1
- New features in JAX-RS 2.0

Producing and parsing JSON documents

JSON format was introduced as a replacement for the XML format when the extensibility and verbosity of XML were not required and thus to lift the resource consumption of complex XML processing to let smaller devices consume streams of data or data packets produced by different services they needed to interact with.

Before the Java EE 7 specification there was no standard API to process JSON documents in Java but rather there were some open source projects such as **google-gson**, `https://code.google.com/p/google-gson` and **Jackson,** `http://jackson.codehaus.org` to manipulate JSON documents. With Java EE 7 and the addition of JSON-P to the arsenal, a standard API is added to Java EE to let the developers manipulate JSON documents in a standard fashion similar to XML processing of APIs.

The JSON-P API provides two parsing methods to parse JSON documents, the same two models that are available for parsing XML documents. The streaming event-based parsing and the object model tree parsing which are explained in the next two sections.

An overview of JSON API

The following table shows the important API segments of JSON-P along with a brief description of each class. The JSON API-related classes are placed under the `javax.json` package. The follow-up sections cover how each one of these can be used.

Class	Description and use
`JsonParser`	A Pull parser to parse JSON objects using event model.
`JsonGenerator`	A JSON stream writer to write JSON objects to an output source such as `OutputStream` and `Writer` in a streaming manner.
`JsonBuilder`	Builds `JsonObject` and `JsonArray` Programmatically
`JsonReader`	Reads `JsonObject` and `JsonArray` from input source
`JsonWriter`	Writes `JsonObject` and `JsonArray` to output source
`JsonObject and JsonArray`	To store `JSONObject` and array structure
`JsonString and JsonNumber`	To store string and numerical values

The `JSONObject` is the entry point to the entire JSON API arsenal. For each one of the following objects JSON API provides a factory method as well as a creator to create them. For example, `Json.createParser` and `Json.createParserFactory` can be used to create a JSON parser. The factory can be configured to produce customized parsers or, when more than one parser is required, to reduce the performance overhead of creating the parsers while the `createParser` overloads can be used to create a JSON parser with a default configuration.

Manipulating JSON documents using the event-based API

The event-based API is best used when a one way, going forward, parsing or producing of JSON documents is required. The event-based API works similar to **StAX** parsing of XML documents but in a much simpler (due to the JSON format being much simpler) fashion.

Producing JSON documents

Producing JSON documents using the event-based API is most suitable when a stream of events is arriving and they require transforming to JSON format for another processor that consumes JSON format.

The following sample code shows how to generate JSON output using `JsonGenerator`:

```
public static void main(String[] args) {
        Map<String, Object>configs = new HashMap<String, Object>(1);
configs.put(JsonGenerator.PRETTY_PRINTING, true);
JsonGeneratorFactory factory = Json.createGeneratorFactory(configs);
JsonGeneratorgenerator = factory.createGenerator(System.out);

generator.writeStartObject()
            .write("title", "Getting Started with RESTful Web
Services")
                .write("type", "paperback")
                .write("author", "Bhakti Mehta, Masoud Kalali")
                .write("publisher", "Packt")
                .write("publication year", "2013")
                .write("edition",  "1")
        .writeEnd()
        .close();
    }
```

Executing the preceding code produces the following content in the standard output:

```
{
    "title":" Getting Started with RESTful Web Services",
        "type":" paperback",
        "author":" Bhakti Mehta, Masoud Kalali",
        "publisher":"Packt",
        "edition":"1"
}
```

In the beginning of the code, the properties object that is created can be used to add directives on what behaviors are expected from the `JsonGenerator` object. The directives that can be specified differ from implementation to implementation but here the `JsonGenerator.PRETTY_PRINTING` is used to ensure that the resulting JSON document is formatted and human-readable.

> The `JsonParser`, `JsonGenerator`, `JsonReader`, `JsonWriter` can be used in Automatic Resource Management blocks, for example:
> ```
> try (JsonGenerator generator = factory.
> createGenerator(System.out);)
>         {
>          }
> ```

Parsing JSON documents

Assuming that the result of the previous sample is saved to a file named `output.json`, the following snippet can be used to parse the `output.json` using stream parser.

```java
FileInputStreambooksInputfile = new FileInputStream("output.json");
JsonParser parser = Json.createParser(booksInputfile);
            Event event = null;
            while(parser.hasNext()) {
                event = parser.next();
                if(event == Event.KEY_NAME&&"details".equals(parser.
getString())) {
                    event = parser.next();
                    break;
                }
            }
            while(event != Event.END_OBJECT) {
                switch(event) {
                    case KEY_NAME: {
                        System.out.print(parser.getString());
                        System.out.print(" = ");
                        break;
                    }
                    case VALUE_NUMBER: {
                        if(parser.isIntegralNumber()) {
                          System.out.println(parser.getInt());
                        } else {
                          System.out.println(parser.getBigDecimal());
                        }
                      break;
                    }
                    case VALUE_STRING: {
                        System.out.println(parser.getString());
                        break;
                    }
```

```
                    default: {
                    }
                }
            event = parser.next();
        }
```

The event types that should be processed when parsing a JSON document are listed as follows:

- `START_ARRAY`: Indicating start of an array in the JSON document
- `START_OBJECT`: Indicating start of an object
- `KEY_NAME`: Name of the key
- `VALUE_STRING`: When the key's value is string
- `VALUE_NUMBER`: When the value is number
- `VALUE_NULL`: If the value is null
- `VALUE_FALSE`: If value is Boolean false
- `VALUE_TRUE`: If value is Boolean true
- `END_OBJECT`: End of an object is reached
- `END_ARRAY`: End of an array is reached

Manipulating JSON documents using the JSON object model

The document object model for parsing JSON provides the same flexibilities and limitation that XML DOM parsing provides. The list of flexibilities includes but not limited to, forward and backward traversing and manipulating the DOM tree; the disadvantages or the tradeoff are on the parser speed and memory requirement.

Generating JSON documents

The following sample code shows how to generate JSON documents using the builder API and later on writing the produced object to standard output:

```
Map<String, Object>configs = new HashMap<String, Object>();
JsonBuilderFactory factory = Json.createBuilderFactory(configs);
JsonObject book= factory.createObjectBuilder()
.add("title", "Getting Started with RESTful Web Services")
.add("type", "paperback")
.add("author", "Bhakti Mehta, Masoud Kalali")
```

```
.add("publisher", "Packt")
.add("publication year", "2013")
.add("edition", "1")
.build();
configs.put(JsonGenerator.PRETTY_PRINTING, true);
JsonWriter writer = Json.createWriterFactory(configs).
createWriter(System.out);
writer.writeObject(book);
```

Note the use of configuration properties passed when the `JsonBuilderFactory` is created. Depending on the JSON API implementation, different configuration parameters can be passed to the factory to produce customized `JsonBuilder` objects.

The resulting JSON output looks like:

```
{
    "title":" Getting Started with RESTful Web Services",
        "type":" paperback",
        "author":" Bhakti Mehta, Masoud Kalali",
        " publisher ":" Packt",
        " edition":"1"
}
```

Parsing JSON documents

Parsing a `JSONObject` using the object model is straightforward and starts with creating a reader object and reading the input file/document into a `JSONObject`. After having access to the `JSONObject`, it is possible to traverse over the primitive and array attributes of the `JSONObject`.

```
Map<String, Object>configs =
new HashMap<String, Object>(1);
JsonReader  reader =
Json.createReader(new FileInputStream("book.json"));
JsonObject book=reader.readObject();
        String title = book.getString("title");
int edition = book.getString("edition");
```

As the sample code shows, reading each attribute of the `JSONObject`is performed through typed getters; for example `getString`, `getInt`, `getNull`, `getBoolean`, and so on.

When to use the streaming versus the object API

The streaming event based API is useful when you are manipulating large JSON documents, which you do not want to store in memory. The object model API is useful in the case when you have navigated between different nodes of the JSON document.

Introducing Servlet 3.1

The Java EE 7 specification brings along an updated specification for Servlet API, which addresses some of the community-requested and industry-required changes including but not limited to the following list of changes:

- Addition of the NIO API to servlet specification
- Adding new protocol upgrading support for WebSockets, and so on

The next two sections cover the details of these changes and how they can be used.

NIO API and Servlet 3.1

Servlet 3 introduced async processing of incoming requests in which a request could be placed in a processing queue without a thread being bound to the request until the request processing is finished. In Servlet 3.1, another forward step made forward in which receiving the request data writing back the response can be done in a non-blocking, callback-oriented manner.

Introducing ReadListener and WriteListener

The two listeners are introduced to allow developers to basically receive notification when there is incoming data available to read rather than blocking until the data arrives and to be receiving notification when it is possible to write output without being blocked.

The `ReadListener` interface, which provides callback notification on availability of data in request's `InputStream` code, is shown in the following listing, a simple interface with three methods, which are described after the code snippet.

```
public interface ReadListener extends EventListener {
 public void onDataAvailable(ServletRequest request);
 public void onAllDataRead(ServletRequest request);
 public void onError(Throwable t);
}
```

- `onDataAvailable`: Invoked when all data for the current request has been read.

- `onAllDataRead`: Invoked by the container the first time it is possible to read data.

- The `onError`: Invoked when an error occurs processing the request.

The `WriteListener`, which provides callback notification when it is possible to write data in the Servlet's `OutputStream`, is a simple two methods interface that is shown in the following snippet and described afterward.

```
public interface WriteListener extends EventListener {
public void onWritePossible(ServletResponse response);
public void onError(Throwable t);
}
```

The container invokes the `onWritePossible` method when it is possible to write in the Servlet's `OutputStream`.

The `onError` is invoked when writing in the Servlet's `OutputStream` encounters an exception.

The sample code on how these two listeners can be used is included at the end of Servlet 3.1 introduction section.

Changes in the Servlet API interfaces

There are some changes in the Servlet API to make it possible to use the newly introduced interfaces. These changes are as follow:

- In the `ServletOutputStream` interface:

    ° `isReady`: This method can be used to determine if data can be written without blocking

    ° `setWriteListener`: Instructs the `ServletOutputStream` to invoke the provided `WriteListener` when it is possible to write

- In the `ServletInputStream` interface

    ° `isFinished`: Returns true when all the data from the stream has been read else it returns false

    ° `isReady`: Returns true if data can be read without blocking else returns false

    ° `setReadListener`: Instructs the `ServletInputStream` to invoke the provided `ReadListener` when it is possible to read

Now it is time to see how the non-blocking Servlet 3.1 API works. The following snippet shows an Async Servlet, which uses the non-blocking APIs:

```
@WebServlet(urlPatterns="/book-servlet", asyncSupported=true)
public class BookServlet extends HttpServlet {
    protected void doPost(HttpServletRequestreq, HttpServletResponse
res)
            throws IOException, ServletException {
AsyncContext ac = req.startAsync();
ac.addListener(new AsyncListener() {
            public void onComplete(AsyncEvent event) throws
IOException {
event.getSuppliedResponse().getOutputStream().print("Async Operation
Completed");
            }
            public void onError(AsyncEvent event) {
System.out.println(event.getThrowable());
            }
            public void onStartAsync(AsyncEvent event) {
System.out.println("Async Operation Started");
            }
            public void onTimeout(AsyncEvent event) {
System.out.println("Async Operation Timedout");
            }
        });
ServletInputStream input = req.getInputStream();
ReadListenerreadListener =  new ReservationRequestReadListener(input,
res, ac);
input.setReadListener(readListener);
    }
}
```

The code starts with declaring the Servlet and enabling asynchronous support by specifying the `asyncSupported=true`in the `@WebServlet` annotation.

The next step is to set the `AsyncListener` to handle the `AsyncEvents`. Invoking one of the `AsyncContext` sets the `AsyncListener` for `AsyncContext#addListner` overloads. The listener will receive the `AsyncEvents` when an asynchronous invocation of the Servlet is completed successfully or ended with timeout or error. Multiple listeners can be registered and listeners receive the events in the same order they are registered.

The last part of the code sets the `readListener` for the servlet to a `ReadListener` implementation included below. When the `ReadListener`is set, reading the incoming requests is delegated to the `ReservationRequestReadListener`.

```
class ReservationRequestReadListener implements ReadListener {
    private ServletInputStream input = null;
    private HttpServletResponse response = null;
    private AsyncContext context = null;
    private Queue queue = new LinkedBlockingQueue();

ReservationRequestReadListener(ServletInputStream in,
HttpServletResponse r, AsyncContext c) {
this.input = in;
this.response = r;
this.context = c;
    }

    public void onDataAvailable() throws IOException {
StringBuildersb = new StringBuilder();
int read;
byte b[] = new byte[1024];
while (input.isReady() && (read = input.read(b)) != -1) {
String data = new String(b, 0, read);
sb.append(data);
        }
queue.add(sb.toString());
    }

public void onAllDataRead() throws IOException {
performBusinessOperation();
ServletOutputStream output = response.getOutputStream();
WriteListenerwriteListener = new ResponseWriteListener(output, queue,
context);
output.setWriteListener(writeListener);
    }

public void onError(Throwable t) {
context.complete();
    }
}
```

The `ReservationRequestReadListener.onDataAvailable` is invoked by the container when there is data to read and when reading the data is finished the `onAllDataRead` is invoked. The `onAllDataRead` performs the business operation on the available data and the set the `ResponseWriteListener`, which writes the data, is stored in the queue back to the client. The `ResponseWriteListener` is shown in the following listing:

```
class ResponseWriteListener implements WriteListener {
    private ServletOutputStream output = null;
    private Queue queue = null;
    private AsyncContext context = null;

ResponseWriteListener(ServletOutputStreamsos, Queue q, AsyncContext c)
{
this.output = sos;
this.queue = q;
this.context = c;
    }

    public void onWritePossible() throws IOException {
        while (queue.peek() != null &&output.isReady()) {
            String data = (String) queue.poll();
            output.print(data);
        }
        if (queue.peek() == null) {
            context.complete();
        }
    }

    public void onError(final Throwable t) {
            context.complete();
            t.printStackTrace();
    }
}
```

When the writing operation is finished either normally or fatally, the context needs to be closed for this operation using the `context.complete()` method.

More changes in Servlet 3.1

In addition to non-blocking IO inclusion, Servlet 3.1 brings in support for protocol upgrade in order to support the new WebSockets API. The addition of the `upgrade` method to `HttpServletRequest` allows developers to upgrade the communication protocol to other protocols, if supported by the container.

When an upgrade request is sent, the application decides to perform the upgrader, the `HttpServletRequest#upgrade(ProtocolHandler)` is invoked, and application prepares and sends an appropriate response to the client as usual. At this point the web container unwinds all the servlet filters and marks the connection to be handled by the protocol handler.

New features in JAX-RS 2.0

JAX-RS 2.0 brings in several new features aligned with other lightweight and async processing features provided in other components. The new features include the following:

- Client API
- Common configuration
- Asynchronous processing
- Filters/interceptors
- Hypermedia support
- Server-side content negotiation

From this list of features, this section covers asynchronous processing and also the relevance of asynchronous processing to filters/interceptors.

Asynchronous request and response processing

Asynchronous processing is included in both client and server side APIs of JAX-RS 2.0 to facilitate asynchronous interaction between client and server components. The following list shows the new interfaces and classes added to support this feature:

- Server side:
  - `AsyncResponse`: An injectable JAX-RS asynchronous response that provides the means for asynchronous server side response processing.
  - `@Suspended`: `@Suspended` instructs the container that the HTTP request processing should happen in a secondary thread.
  - `CompletionCallback`: A request processing callback that receives request processing completion events.
  - `ConnectionCallback`: Asynchronous request processing lifecycle callback that receives connection-related asynchronous response lifecycle events.

- Client side:

  - `InvocationCallback`: Callback that can be implemented to receive the asynchronous processing events from the invocation processing

  - `Future`: Allows the client to poll for completion of the asynchronous operation or to block and wait for it

> The `Future` interface introduced in Java SE 5 provides two different mechanism to get the result of an asynchronous operation: first by invoking the `Future.get`(...) variants, which block until the result is available or a timeout occurs; the second way is to check for the completion by invoking the `isDone()` and `isCancelled()`, which are Boolean methods returning the current status of the Future.

The following sample code shows how an asynchronous resource can be developed using JAX-RS 2 API:

```
@Path("/books/borrow")
@Stateless
public class BookResource {
  @Context private ExecutionContextctx;
  @GET @Produce("application/json")
  @Asynchronous
  public void borrow() {
        Executors.newSingleThreadExecutor().submit( new Runnable() {
        public void run() {
        Thread.sleep(10000);
        ctx.resume("Hello async world!");
        } });
        ctx.suspend();
        return;
  }
}
```

`BookResource` is a stateless session bean which has a method `borrow()`. This method is annotated with `@Asynchronous` annotation, which will work in the fire-and-forget manner. When the resource is requested through the `borrow()` method's resource path, a new thread is spawned to work on preparing the request's response. The thread is submitted to the executor for execution and the thread processing the client request is released (via `ctx.suspend`) to process other incoming requests. When the worker thread, created to prepare the response, is done with preparing the response, it invokes the`ctx.resume`, which lets the container know the response is ready to be sent back to the client. If the`ctx.resume` is invoked before the `ctx.suspend` (the worker thread has prepared the result before the execution reaching the `ctx.suspend`) the suspension is ignored and the result will be sent to the client.

Same functionality can be achieved using the `@Suspended` annotation that is shown in the following snippet:

```
@Path("/books/borrow")
@Stateless
public class BookResource {
  @GET @Produce("application/json")
  @Asynchronous
  public void borrow(@Suspended AsyncResponsear) {
  final String result = prepareResponse();
  ar.resume(result)   }
}
```

Using `@Suspended` is cleaner as it does not involve use of `ExecutionContext` method to instruct container to suspend and then resume the communication thread when the worker thread, aka the `prepareResponse()` method in this case, is finished. The client code to consume the asynchronous resource can use the callback mechanism or polling at the code level. The following code shows how to use polling via `Future` interface:

```
Future<Book> future = client.target("("books/borrow/borrow")
                .request()
                .async()
                .get(Book.class);
try {
    Book book = future.get(30, TimeUnit.SECONDS);
} catch (TimeoutException ex) {
  System.err.println("Timeout occurred");
}
```

The code begins with forming the request to the book resource and then the `Future.get(...)` blocks until the response is back from the server or the 30 seconds timeout reaches.

Another API for the asynchronous client is to use the `InvocationCallback`.

Filters and interceptors

The filters and interceptors are two new concepts added to JAX-RS 2.0 that allow developers to intercept incoming and outgoing requests and responses as well as operating at stream level on the incoming and outgoing payloads.

The filters work the same way as Servlet filters work and provide access to inbound and outbound messages for tasks such as authentication/logging, auditing, etc. while interceptors can be used to perform dumb operations on payload such as compressing/decompressing the outgoing responses and incoming requests.

Filters and interceptors are asynchronous-aware, meaning that they can handle both synchronous and asynchronous communications.

Asynchronous processing in EJB 3.1 and 3.2

Before Java EE 6 the only asynchronous processing facility in Java EE was **JMS** (**Java Message Service**) and **MDBs** (**Message Driven Beans**) in which a session bean method could send a JMS message to describe a request and then let an MDB process the request in an asynchronous manner. Using the JMS and MDBs the session bean method could return immediately and the client could check for the request completion using the reference returned by the method for the long running operation being handled by some MDBs.

The above solution works well, as it has worked for a decade now, but it is not easy to use and that was the reason for Java EE 6 to introduce the `@Asynchronous` annotation to annotate a method in a session bean or the whole session bean class as asynchronous. The `@Asynchronous` can be placed on a class to mark all the methods in that class as asynchronous or on a method to mark that particular method as asynchronous.

There are two types of asynchronous EJB invocation which are explained as follows:

- In the first model the method returns `void` and there is no container-provided standard mechanism to check the result of the method invocation. This is referred to as a **fire-and-forget** mechanism.

- In the second model, the container provides a mechanism to check back the result of the invocation using a `Future<?>` object returned from the method invocation. This mechanism is referred to as **invoke-and-check-later**. Note that `Future` is part of the Java SE concurrency package. Having the Future object returned from the method, the client can check the result of invocation by using different Future methods such as `isDone()` and `get(…)`.

Before we dive down into sample codes or use the `@Asynchronous` it is worth mentioning that, in Java EE 6, the `@Asynchronous` was only available in full profile while in Java EE 7 the annotation is added to the web profile as well.

Developing an asynchronous session bean

The following listing shows how to use the invoke-and-check-later asynchronous EJB methods:

```
@Stateless
@LocalBean
public class FTSSearch {

    @Asynchronous
    public Future<List<String>> search(String text, intdummyWait) {
        List<String> books = null;
        try {
           books= performSearch(text,dummyWait);
        } catch (InterruptedException e) {
            //handling exception
        }
        return new AsyncResult<List<String>>(books);
    }
    private List<String>performSearch(String content, intdummyWait)
throws InterruptedException{
Thread.sleep(dummyWait);
return Arrays.asList(content);
    }
}
```

`@Stateless` and `@LocalBean` are self-explanatory; they mark this class as a stateless session bean with a local interface.

The `search` method is annotated with `@Asynchronous` and this tells the container that the method invocation should happen in a separate detached thread; when the result is available the returned Future object's `isDone()` returns true.

The `search` itself invokes a presumably long running method, `performSearch`, to get the result of the long running search operation the client has requested.

Developing a client servlet for the async session bean

Now that the stateless session bean is developed it is time to develop a client that accesses the session bean's business method. In this case the client is a Servlet, which is included in the following code without some of the boilerplate codes:

```
@WebServlet(name = "FTSServlet", urlPatterns = {"/FTSServlet"})
public class FTSServlet extends HttpServlet {
```

```
    @EJB
FTSSearchftsSearch;

    @Override
    protected void doGet(HttpServletRequest request,
HttpServletResponse response)
            throws ServletException, IOException {
        Future<List<String>>wsResult = ftsSearch.search("WebSockets",
5000);
        Future<List<String>>sseResult = ftsSearch.search("SSE", 1000);

        while (!sseResult.isDone()) {
            try {
Thread.sleep(500);
                //perform other tasks... e.g. show progress status
            } catch (InterruptedException ex) {
Logger.getLogger(FTSServlet.class.getName()).log(Level.SEVERE, null,
ex);
            }
        }


response.setContentType("text/html;charset=UTF-8");
PrintWriter out = response.getWriter();
        try {
            /* TODO output your page here. You may use following
sample code. */
out.println("<!DOCTYPE html>");
out.println("<html>");
out.println("<head>");
out.println("<title>Servlet d</title>");
out.println("</head>");
out.println("<body>");
out.println("<h1>SSE Search result: " + sseResult.get().get(0) + "</
h1>");
            while (!wsResult.isDone()) {
                try {
Thread.sleep(500);
                } catch (InterruptedException ex) {
Logger.getLogger(FTSServlet.class.getName()).log(Level.SEVERE, null,
ex);
                }
            }
out.println("<h1>WS Search result: " + wsResult.get().get(0) + "</
h1>");
out.println("</body>");
```

```
out.println("</html>");
        } catch (InterruptedException ex) {
Logger.getLogger(FTSServlet.class.getName()).log(Level.SEVERE, null,
ex);
        } catch (ExecutionException ex) {
Logger.getLogger(FTSServlet.class.getName()).log(Level.SEVERE, null,
ex);
        } finally {
out.close();
        }
    }

}
```

Starting from the top, we have the servlet declaration annotations, injection of the stateless EJB, and the `get` method's implementation.

The get in the `get` method's implementation invokes the EJB's `search` method while passing two different `dummyTime` to simulate the wait. Between invocations of the `search` method till the Future object's `isDone` returns true, the client code can perform other required operations.

Now that the invoke-and-check-later model is described we can discuss the other Asynchronous EJB invocation model in which the EJB business methods return `void` and there is no container-provided way to check the result. We usually use these methods to trigger a long-running task that the current thread does not need to wait for it to be finished.

An example of this case is when a new e-book is added to the library and the full text search index needs to be updated to include the new book. In such a case the procedure that adds the book can invoke a `@Asynchronous` EJB method to index the book during the book's registration and after it is uploaded to the server's repository. In this way the registration process does not need to wait for the FTS indexing to complete while the FTS indexing starts right after the book is added to the library.

Summary

This chapter, the final chapter before showing you some real-world examples, discusses JSON processing, Asynchronous JAX-RS resources, which can produce or consume JSON data along with discussing the new NIO support in the Servlet 3.1. As `@Asynchronous` EJB is now included in Java EE 7's Web profile we discussed that feature along with other new features that are introduced in Java EE7. The next chapter shows real-world examples on how these technologies and APIs can be used together to form a solution.

5
RESTful Web Services by Example

The APIs and technologies introduced and discussed in the previous chapters are suitable for different types of projects and use cases. This chapter covers how these APIs and technologies can fit into solutions and case-oriented software systems.

After a brief introduction of what the application is supposed to do, we will break it down and focus on every component and technology used. So bring on the extra cup of mocha and join the fun.

This chapter will cover the following two samples:

- Event notification application showing Server-sent Events, **Async Servlet**, JSON-P API, and JAX-RS based on the **Twitter Search API**

- Library application showing JAX-RS API, WebSockets, JSON-P API, and asynchronous JAX-RS resources to form an end-to-end solution

Event notification application

The Twitter-based application is the first sample application that will demonstrate a HTML5-based application developed on top of Server-sent Events, JAX-RS 2.0 API, Asynchronous Servlet, and the Twitter Search API together, to dynamically update a page with more search results periodically.

The build system used for the sample application is **Maven** and the sample can be deployed in any Java EE 7-compatible application server notably GlassFish v4.0, an open source reference implementation of the Java EE 7 specification.
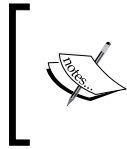
> Apache Maven is a build management tool. More information about Maven can be found at `http://maven.apache.org` and more information about GlassFish can be found at `https://glassfish.java.net/`

The project's layout

The project's directory layout follows the standard Maven structure, which is briefly explained in the following table:

Source code	Description
`src/main/java`	This directory contains all the sources required by the library application.
`src/main/webapp`	This directory contains the JavaScript files, html files, and `WEB-INF/web.xml` file.

The event notification GUI

The event notification application consists of one screen, which serves as the vehicle for showing dynamic updates based on the Twitter feeds. The screen is shown in the following screenshot:
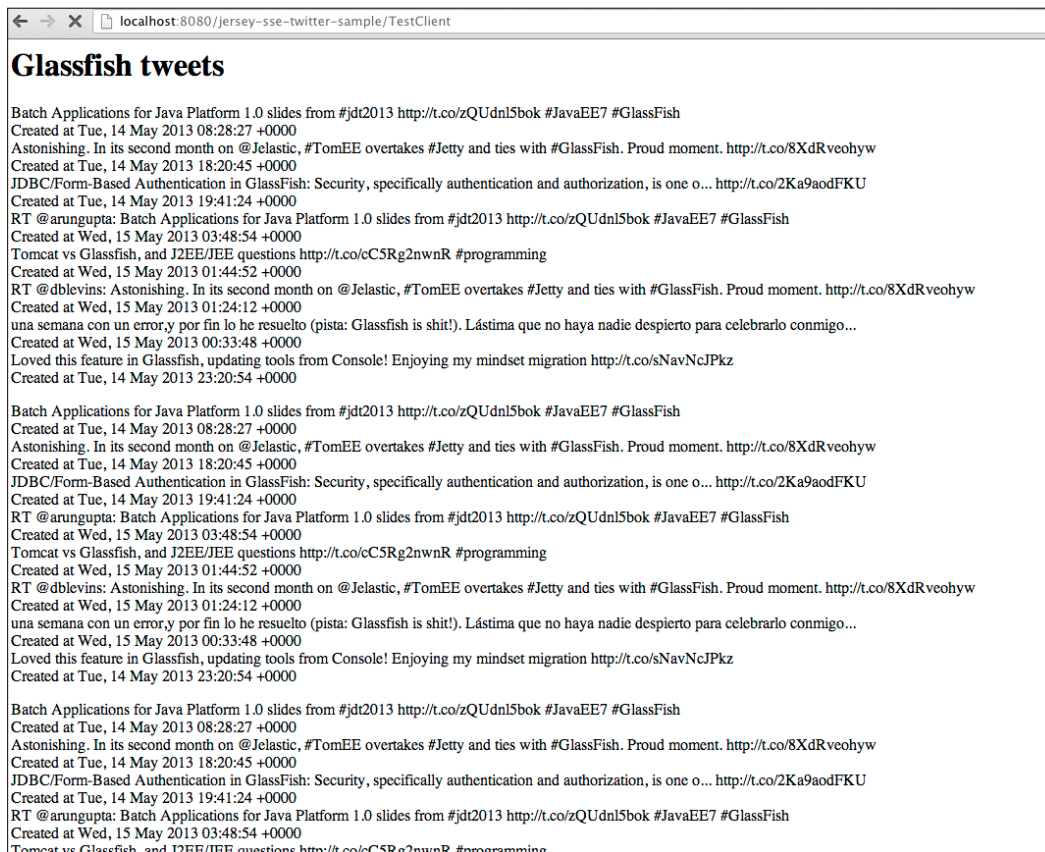
The application is a basic sample showing updates as the events carrying the updates happen and are received. This could be a newer tweet coming in, or Facebook friends' updates, or any other type of events consumable by any of the Java EE-managed components. The key point is, once the channel of communication is established with the server, it is up to the server to keep sending updates as they occur. The client does not poll for updates.

In this sample, when the servlet is loaded there is an **EJB timer**, which is run every 10 seconds and activates a CDI bean that uses the Twitter Search API to get new tweets. The Twitter Search API returns the tweets in JSON format. This tweet information is then sent to the client using Server-sent Events support with JAX-RS. On the client side the JSON data is parsed to display certain information on the screen.

A detailed look at the event notification application

After an initial introduction to what the application is supposed to do, let's dissect it further and study each individual component that builds this application.

This is the order in which the details of the application will be covered:

- The `web.xml`
- The implementation of the Application class
- The JAX-RS resource used in the application
- The Asynchronous Servlet client used by the application
- The EJB that interacts with the Twitter Search API

The web.xml

To set up the application, configure the servlet deployment descriptor `web.xml` as follows:

```
<display-name>jersey-sse-twitter-sample</display-name>

<servlet>
  <servlet-name>Jersey application</servlet-name>
  <servlet-class>org.glassfish.jersey.servlet.ServletContainer
    </servlet-class>
  <init-param>
    <param-name>javax.ws.rs.Application</param-name>
    <param-value>org.glassfish.jersey.sample.sse.MyApplication
      </param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
  <async-supported>true</async-supported>
</servlet>
<servlet-mapping>
  <servlet-name>Jersey application</servlet-name>
  <url-pattern>/*</url-pattern>
</servlet-mapping>
```

`MyApplication` is a subclass of `javax.ws.rs.Application`. It is used to register the JAX-RS resource so that it is known the JAX-RS API.

The `async-supported` element is set to `true` to indicate that the servlet supports asynchronous processing.

The implementation of the Application class

Here is the implementation of the `Application` subclass:

```
public class MyApplication extends Application {

  Set<Class<?>> classes = new HashSet<Class<?>>() {
    { add(ServerSentEventsResource.class);
      add(SseFeature.class);
    }
  };

  @Override
  public Set<Class<?>> getClasses() {
    return classes;
  }
}
```

The `getClasses()` method is overridden to return the:

- `ServerSentEventsResource.class`
- `SseFeature.class`

The `ServerSentEventsResource` class is a simple JAX-RS that sends the JSON data from the Twitter Search API as Server-sent Events. We shall look at the `ServerSentEventsResource` in more detail in the next section.

The `SseFeature.class` is an implementation provided by **Jersey** to support the `ServerSentEvents` feature. It will ensure the data is of the media type `"text/event-stream"`.

> To enable Server-sent Events feature, add `SseFeatures.class` to the list of classes returned by the `getClasses()` method in the implementation of the `javax.ws.rs.Application`.

The JAX-RS resource used by the application

Here is the source code of the `ServerSentEventsResource.java`. This is a simple POJO, annotated with `@Path` to identify the URI of the resource.

```
@Path("twittersse")
public class ServerSentEventsResource {

  static EventOutput eventOutput = new EventOutput();
```

```
@GET
@Produces(SseFeature.SERVER_SENT_EVENTS)
public EventOutput getMessage() {
  return eventOutput;
}

@POST
@Consumes(MediaType.TEXT_PLAIN)
public void sendMessage(String message) throws IOException {
  eventOutput.write(new OutboundEvent.Builder().name(
    "custom-message").data(String.class, message).build());
}
}
```

The `EventOutput` class is a channel that provides the outbound Server-sent Events. When we return the `EventOutput` object from the `getMessage()` method, the Jersey implementation keeps the connection open so that the Server-sent Events can be sent. One instance of this class corresponds with exactly one HTTP connection.

The `sendMessage()` method writes the message using the `eventOutput.write()` method. To write Server-sent Events, we use the `OutboundEvent.Builder()` method. A name `"custom-message"` is passed to this `OutboundEvent.Builder()` method and then we pass the message object to the `build()` method. The message object contains the tweets-related information for our sample.

Additionally, `OutboundEvent.Builder().id(id)` can be used to associate an ID with a Server-sent Event which is not covered previously.

The Asynchronous Servlet client used by the application

In normal request response scenarios, a thread is kept running for each request till the response becomes available. This turns into a bottleneck in cases when the backend is taking a long time to process the requests, and the thread processing the request waits for the backend to finish preparing the required response and thus cannot take on any new incoming request.

One way to solve this would be to save the request in a centralized queue and send the request as the threads are available to process the request. Calling the `startAsync()` method stores the request/response pair in a queue, the `doGet()` method returns, and the calling thread can be recycled.

The following section discusses these concepts of asynchronous request processing with servlets.

Here is the code of the Servlet client for the application:

```
@WebServlet(name = "TestClient", urlPatterns = {"/TestClient"},
asyncSupported = true)
public class TestClient extends HttpServlet {

    private final static String TARGET_URI =
        "http://localhost:8080/jersey-sse-twitter-sample/twittersse";
```

This is a Servlet with urlPatterns={"/TestClient"} and the async-supported attribute set to true. The async-supported attribute instructs the container that this servlet process the incoming requests asynchronously and thus the container should make the necessary modification in request allocation of the processing threads.

The next snippet shows the implementation of the service() method that can handle the GET and POST requests:

```
/**
* Processes requests for both HTTP
* <code>GET</code> and
* <code>POST</code> methods.
*
* @param request  servlet request
* @param response servlet response
* @throws ServletException if a servlet-specific error occurs
* @throws IOException       if an I/O error occurs
*/
@Override
protected void service(final HttpServletRequest request, final
  HttpServletResponse response)
throws ServletException, IOException {
  response.setContentType("text/html;charset=UTF-8");

  try {

    final AsyncContext asyncContext = request.startAsync();
    asyncContext.setTimeout(600000);
    asyncContext.addListener(new AsyncListener() {

      @Override
      public void onComplete(AsyncEvent event) throws IOException
        {
```

```
        }

        @Override
        public void onTimeout(AsyncEvent event) throws IOException {
          System.out.println("Timeout" + event.toString());
        }

        @Override
        public void onError(AsyncEvent event) throws IOException {
          System.out.println("Error" + event.toString());
        }

        @Override
        public void onStartAsync(AsyncEvent event) throws
          IOException {
        }
      });


      Thread t = new Thread(new AsyncRequestProcessor(asyncContext));
      t.start();

      } catch (Exception e) {
      e.printStackTrace();
    }

  }
```

In the preceding snippet, an instance of the `AsyncContext` object is obtained by invoking the `request.startAsync()` method.

The `asyncContext.setTimeout(60000)` method indicates a timeout in milliseconds for the asynchronous operations of the servlet.

An implementation of an `AsyncListener` interface is added to the asynchronous context using the `asyncContext.addListener()` method.

After the `startAsync()` method is called on the request, an `AsyncEvent` object is sent to the implementation of the `AsyncListener` interface as the operation completes, there is an error, or the operation times out. As shown previously, we have an implementation of the `AsyncListener` interface that can implement the following methods `onComplete()`, `onError()`, `onTimeOut()`, or `onStartAsync()`.

The `AsyncRequestProcessor` class shown in the following code is the `Runnable` instance of the thread, which does the actual work. The `AsyncRequestProcessor` class registers the `EventSource` object to listen for the Server-sent Events, which are sent by the JAX-RS `ServerSentEventsResource.java`, covered earlier. As events occur the `onEvent()` callback is triggered and JSONP is used to parse the events.

```java
class AsyncRequestProcessor implements Runnable {

  private final AsyncContext context;

  public AsyncRequestProcessor(AsyncContext context) {
    this.context = context;
  }

  @Override
  public void run() {
    Client client = ClientBuilder.newClient();
    context.getResponse().setContentType("text/html");
    final javax.ws.rs.client.WebTarget webTarget;
    try {
      final PrintWriter out = context.getResponse().getWriter();
      webTarget = client.target(new URI(TARGET_URI));
      out.println("<html>");
      out.println("<head>");
      out.println("<title>Glassfish SSE TestClient</title>");
      out.println("</head>");
      out.println("<body>");
      out.println("<h1>");
      out.println("Glassfish tweets");
      out.println("</h1>");
      // EventSource eventSource = new EventSource(webTarget,
        executorService) {
        EventSource eventSource = new EventSource(webTarget) {
          @Override
          public void onEvent(InboundEvent inboundEvent) {
            try {
              //get the JSON data and parse it
              JSONObject jsonObject = JSONObject.fromObject
                (inboundEvent.getData(String.class,
              MediaType.APPLICATION_JSON_TYPE));
              //get the JSON data and parse it
              JsonReader jsonReader = Json.createReader (new
                ByteArrayInputStream(inboundEvent.getData
                  (String.class,
```

```
                MediaType.APPLICATION_JSON_TYPE).getBytes()));
                JsonArray jsonArray = jsonReader.readArray();
                for (int i = 0; i <jsonArray.size(); i++) {
                  JsonObject o = ((JsonObject)
                    jsonArray.getJsonObject(i)) ;
                  out.println( o.get("text"));
                  out.println("<br>");
                  out.println("Created at " +
                    o.get("created_at"));
                  out.println("<br>");

                }
                out.println("</p>");
                out.flush();
              } catch (IOException e) {
                e.printStackTrace();
              }
            }
          };
        } catch (Exception e) {
          e.printStackTrace();
        }
      }
    }
  }
```

As seen in the preceding code, we use the **JSR 353 Java API for JSON Processing** to create a `JSonReader` object from the `inboundEvent#getData()` method. The `JSONArray` object is returned by the `jsonReader.readArray()` method. The `JsnObject` objects are read from the array and the tweet information is displayed.

The EJB that interacts with the Twitter Search API

Here is the code for the EJB that will invoke the Twitter Search API. This EJB has a timer that will periodically call the Twitter Search API to get tweets for GlassFish and get the results in the JSON format.

```
    @Stateless
    @Named
    public class TwitterBean {
    }
```

The `@Stateless` annotation indicates this is a stateless session bean.

---

[88]

Twitter v1.1 API uses **OAuth** to provide authorized access to its API. Twitter offers applications the ability to issue authenticated requests on behalf of the application itself (as opposed to on behalf of a specific user). For more on **OAuth** please check `https://dev.twitter.com/docs/api/1.1/overview`.

To run this demo, you will need to have a Twitter account and create an application based on information specified in this following link: `https://dev.twitter.com/docs/auth/oauth`. Please see the `Readme.txt` with the sample for instructions on how to run the sample.

The following code uses the **twitter4j** API from `http://twitter4j.org/en/index.html` to integrate Java and the Twitter API.

Here is the code that will connect to `SEARCH_URL` and get the tweets

```
/**
* Since twitter uses the v1.1 API we use twitter4j to get
* the search results using OAuth
* @return a JsonArray containing tweets
* @throws TwitterException
* @throws IOException
*/
public JsonArray getFeedData() throws TwitterException, IOException {

  Properties prop = new Properties();

  //load a properties file
  prop.load(this.getClass().getResourceAsStream
    ("twitter4j.properties"));

  //get the property value and print it out
  String consumerKey = prop.getProperty("oauth.consumerKey");
  String consumerSecret=
    prop.getProperty("oauth.consumerSecret");
  String accessToken = prop.getProperty("oauth.accessToken");
  String accessTokenSecret =
    prop.getProperty("oauth.accessTokenSecret");
  ConfigurationBuilder cb = new ConfigurationBuilder();
  cb.setDebugEnabled(true)
  .setOAuthConsumerKey(consumerKey)
  .setOAuthConsumerSecret(consumerSecret)
  .setOAuthAccessToken(accessToken)
  .setOAuthAccessTokenSecret(accessTokenSecret);
```

```
TwitterFactory tf = new TwitterFactory(cb.build());
Twitter twitter = tf.getInstance();
Query query = new Query("glassfish");
QueryResult result = twitter.search(query);
JsonArrayBuilder jsonArrayBuilder  =
  Json.createArrayBuilder();
for (Status status : result.getTweets()) {
  jsonArrayBuilder
  .add(Json.createObjectBuilder().
  add("text", status.getText())
  .add("created_at", status.getCreatedAt().toString()));
}
return jsonArrayBuilder.build() ;
}
```

The preceding code reads the `twitter4j.properties` and creates a
`ConfigurationBuilder` object with the `consumerKey`, `consumerSecret`,
`accessToken`, and `accessTokenSecret` keys. Using the `TwitterFactory`
API an instance of Twitter object is created. A `Query` object to send the search
request to Twitter with the keyword `"glassfish"` is created. The `twitter.search`
returns tweets that match a specified query. This method calls `http://search.`
`twitter.com/search.json`.

Once the `QueryResult` object is obtained, the `JsonArrayBuilder` object is used to
build the JSON object containing the results. For more information on twitter4j API
please check `http://twitter4j.org/oldjavadocs/3.0.0/index.html`.

The EJB bean has an additional method that will invoke the **EJB timer**. Here is the EJB
Timer code that will send these tweets, which are obtained from the Twitter Search
API, to the REST Endpoint `ServerSentEventsResource` using the `POST` method.

```
private final static String TARGET_URI =
  "http://localhost:8080/jersey-sse-twitter-sample";

@Schedule(hour = "*", minute = "*", second = "*/10")
public void sendTweets() {

  Client client = ClientBuilder.newClient();
  try {
    WebTarget webTarget= client.target(new URI(TARGET_URI)) ;
    JsonArray statuses = null;

    statuses = getFeedData();
```

```
      webTarget.path("twittersse").request().
        post(Entity.json(statuses));
    }(catch Exception e) {
      e.printStackTrace();
    }
  }
```

The `@Schedule` annotation is used to schedule fetching tweets every 10 seconds.
The EJB specification has more details on usages of `@Schedule`. The `JsonArray`
object statuses get the feeds from the `getFeedData()` method that was covered
in the earlier section.

`WebTarget` is created with the `TARGET_URI` that is the URL `http://localhost:8080/`
`jersey-sse-twitter-sample` where the application is deployed.

The `webTarget.path("twittersse")` method points to the location of the
`ServerSentEventsResource` class covered earlier that is the REST resource.

Using the `request().post(Entity.text(message))` method the tweets that are
obtained from the Twitter Search API are sent as a Text Entity.

This is the sequence of events:

1. The user deploys the application and invokes the Servlet client from this URL
   `http://localhost:8080/jersey-sse-twitter-sample`.
2. The EJB timer gets scheduled every 10 seconds.
3. The EJB timer will invoke the Twitter Search API to get the tweets for
   `"glassfish"` in JSON format.
4. The EJB timer sends the data obtained in step to the JAX-RS
   `ServerSentEventsResource` class using the `POST` request.
5. The JAX-RS resource `ServerSentEventsResource` opens the `EventOutput`
   channel, which is the outbound channel for the Server-sent Events.
6. The Servlet client in step 1 has the `EventSource` object open that is listening
   for the Server-sent Events.
7. The Servlet client uses JSON-P API to parse the Twitter feeds.
8. Finally the tweets are shown in the browser.

The library application

The library application is a simple, self-contained, real-life-based application that demonstrates HTML5 technologies such as WebSockets and shows how to use JAX-RS verbs, how to write data using JSON-P API, and how to take advantage of the asynchronous aspect of processing the resources. To stay on track the application contains the components that describe the preceding technologies using a simple GUI and does not have fancy dialog boxes or very complicated business logic.

How the application is deployed

The build system used for the sample application is Maven and the sample can be deployed in any Java EE 7-compatible application server, notably GlassFish v4.0, which is an open source reference implementation of Java EE specification.

The project's layout

The project's directory layout follows the standard Maven structure, which is briefly explained in the following table:

Source code	Description
`src/main/java`	This directory contains all the sources required by the library application.
`src/main/webapp`	This directory contains the JavaScript files, HTML files, and the `WEB-INF/web.xml` file.

The library application GUI

The library application consists of one screen that serves as the vehicle for showing different rendering of the data and forms for gathering inputs. The screen is shown in the following screenshot:

Using the screen, a user can do the following operations:

1. Browse the collection of books.

2. Search for a book.

3. Checkout a book.

4. Return a book.

5. Place hold on a book.

The following table shows the action taken by a user, the details of what happens behind the scenes, and the API and technologies involved in processing the requests:

Action	API and technology used
Browse the collection of books	This task uses the JAX-RS GET verb to get the collection of books in the library. It uses the JSON-P API to write the data in JSON format. We use an implementation of JAX-RS `MessageBodyWriter` class, which knows how to serialize a custom class to JSON output.
Borrow a book	When a book is checked out from the library it reduces from the collection of books, which the library has. This task demonstrates the use of the JAX-RS verb DELETE and deletes the book from the collection.

Action	API and technology used
Return a book	When a book is returned to the library it will be added to the collection of books that the library has. This task demonstrates the use of the JAX-RS verb POST and adds the book to the collection.
Place hold on a book	When a book is placed on hold, the library application should notify other users currently having the book to return it. Once the book is returned, a notification should be sent to the user requesting the book. This is an asynchronous operation. This task demonstrates the use of asynchronous processing of the JAX-RS resources.

Application interaction monitoring

There is a pane that will show what is the query, which was sent to the endpoint. Additionally, we will show the output returned by the endpoint.

A detailed look at the library application

After an initial introduction to what the application is supposed to do, let's dissect it further and study each individual component that builds this application.

The following is the order in which the details of the application will be covered:

- The web.xml
- The Application subclass implementation in our application
- The JAX-RS Entity Providers used in our application
- The HTML page
- JavaScript snippets and JAX-RS resource methods for the following functions:

    ° Browsing the collection of books
    ° Searching for a book
    ° Checking out a book
    ° Returning a book
    ° Placing a hold on the book

The web.xml

To set up the application, configure the servlet deployment descriptor `web.xml` as follows:

```
<servlet>
  <servlet-name>org.sample.library.BookApplication</servlet-name>
  <init-param>
    <param-name>
      javax.json.stream.JsonGenerator.prettyPrinting</param-name>
    <param-value>true</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>

<welcome-file-list>
  <welcome-file>
    index.html
  </welcome-file>
</welcome-file-list>
<servlet-mapping>
  <servlet-name>org.sample.library.BookApplication</servlet-name>
  <url-pattern>/app/*</url-pattern>
</servlet-mapping>
```

In the preceding snippet, we defined a servlet to take the subclass of the JAX-RS Application `BookApplication`. The URL pattern is `/app/*`.

The Application subclass

Here is the snippet of the `BookApplication` class, which is mentioned in the `web.xml` description.

Chapter 1, Building RESTful Web Services Using JAX-RS, covered the Application class in detail. It is used to register the JAX-RS resources and specialized Entity Providers.

```
public class BookApplication extends Application {

  @Override
  public Set<Class<?>> getClasses() {
    Set<Class<?>> classes = new HashSet<Class<?>>();
    classes.add(BooksResource.class);
    classes.add(BookCollectionWriter.class);
```

```
        classes.add(BookWriter.class);
        return classes;
    }
}
```

The BookApplication class extends the JAX-RS Application class. In the getClasses() method implementation, the following are registered:

- BookResource.class
- BookCollectionWriter.class
- BookWriter.class

The BookResource class is covered in detail in the next few sections, with every function of the JavaScript; the corresponding method of the BookResource class will be explained.

The BookCollectionWriter class is an implementation of a MessageBodyWriter interface, which takes a List<Book> object and serializes it to the JSON format. To produce the application/json encoded output, the BookCollectionWriter class uses JSON-P API.

The BookWriter class provides the facility to serialize the user-defined Book class, which is shown in the following section. The Book class has fields such as name of the book, author, and ISBN. Using this BookWriter class it is possible to convert this Book class into a format specified in the resource, for example, "text/plain" or "application/json".

JAX-RS Entity Provider: BookCollectionWriter

Similar to the BookWriter class covered in the earlier section there is a class called BookCollectionWriter in the sample; this is used to serialize a list of books. Here is an implementation of the writeTo() method in the BookCollectionWriter class:

```
@Override
public void writeTo(List<Book> books, Class<?> type, Type
  genericType, Annotation[] annotations, MediaType mediaType,
    MultivaluedMap<String, Object> httpHeaders, OutputStream
      entityStream) throws IOException, WebApplicationException {
  StringWriter writer = new StringWriter();
  if (mediaType.equals(MediaType.APPLICATION_JSON_TYPE)) {
    JsonGenerator generator = Json.createGenerator(writer);
    Map<String, Object> configs;
    configs = new HashMap<String, Object>(1);
```

```java
        configs.put(JsonGenerator.PRETTY_PRINTING, true);

        generator.writeStartArray();
        for (Book book: books) {
          generator.writeStartObject()
          .write("Name", book.getName())
          .write(" ISBN", book.getIsbn())
          .write("Author",book.getAuthor()) .writeEnd();

        }
        generator.writeEnd();
        generator.close();
        entityStream.write(writer.toString().getBytes());
      } else if (mediaType.equals(MediaType.TEXT_PLAIN_TYPE)) {
        StringBuilder stringBuilder = new StringBuilder("Book ");
        for (Book book: books) {
          stringBuilder.append(book.toString()).append("\n");
        }
        entityStream.write(stringBuilder.toString().getBytes());

      }
    }
```

The preceding code does media type filtering; if the `mediaType` parameter equals `MediaType.APPLICATION_JSON_TYPE`, then it uses the JSON-P API to create a `JsonGenerator` object. Using the `writeStartArray()` and `writeStartObject()` methods of `JsonGenerator` class, the array of JSON objects is written.

If the `mediaType` parameter equals `MediaType.TEXT_PLAIN_TYPE`, then a String representation of the books is returned.

The HTML page

As you may recall, when the application is launched in the browser, you will see the `index.html` screen. Let's take a peek at the source code of the `index.html` file:

```html
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html;
      charset=UTF-8">
    <title>Library App</title>
    <script src="main.js">
    </script>
```

```
    </head>
    <body>
      <h1 id="helloMessage">
      </h1>

      Please enter the following details:
      <p>
        Book Name:
        <input type="text" value="Game of thrones" id="bookName"/>
      </p>
      <br>
      <button onclick="search_onclick()">Search</button>
      <button onclick="checkout_onclick()">Checkout</button>
      <button onclick="return_onclick()">Return</button>
      <button onclick="hold_onclick()">Hold</button>
      <button onclick="browse_onclick()">Browse Collection</button>

      <h2>Book Information</h2>
      <h3>JAX-RS query sent by the Application:</h3>

      <div id="query" style="border: 1px solid black; color: black;
height: 6em; width: 80%"></div>
      <h3>Output from the JAX-RS query</h3>
      <div id="output" style="border: 1px solid black; color: black;
height: 18em; width: 80%"></div>
    </body>
</html>
```

This is standard HTML, which uses an external JavaScript file called `main.js` to import the following functionalities:

- Browsing the collection of books
- Searching for a book
- Checking out a book
- Returning a book
- Placing hold on a book

The highlighted `div` elements `query` and `output` show the JAX-RS query and the output on the page. Every button has an `onclick()` event associated with it that calls a function in JavaScript. Each function shall be covered in detail in the next section.

Browsing the collection of books

When a user clicks on the **Browse Collection** button on the HTML page, the input is checked and then the `sendBrowseRequest()` function is called in the JavaScript.

Using JavaScript

Here is the snippet of the `sendBrowseRequest()`:

```
function sendBrowseRequest( ) {
  var req = createRequest(); // defined above
  // Create the callback:
  req.onreadystatechange = function() {
    if (req.readyState == 4) {
      document.getElementById("query").innerHTML="GET
        app/library/books" ;
      document.getElementById("output").innerHTML=
        req.responseText;
    }
  }
  req.open("GET","app/library/books" ,true);
  req.send(null);
}
```

The `createRequest()` function is used to create an `XMLHttpRequest` object as covered in *Chapter 2*, *WebSockets and Server-sent Events*. The request, which is sent to the JAX-RS resource, is a `GET` request with the URI `/app/library/books` (we shall cover the JAX-RS resource in the next section). When the value of the `XMLHttpRequest` object's `readyState` is `4`, it means that the response is complete and we can get the data. In our sample, we display the `responseText` using the snippet `document.getElementById("output").innerHTML=req.responseText;`.

The JAX-RS resource method for the GET request

Here is the snippet for the `GET` request:

```
@GET
@Path("books")
@Produces({MediaType.TEXT_PLAIN, MediaType.APPLICATION_JSON})
public List<Book> browseCollection() {
  return bookService.getBooks();
}
```

This is a very simple method that will use the `BookCollectionWriter` class that we covered earlier to output the `List<Book>` objects in JSON format or in the plain text format.

Searching for a book

When a user clicks on the **Search** button on the HTML page,
the `sendSearchWSRequest()` function is called in the JavaScript.

Using JavaScript

The `sendSearchWSRequest()` function demonstrates the WebSockets functionality
in our application. A WebSocket URI is initializing in the JavaScript as follows:

```
var wsUri = "ws://localhost:8080/libraryApp/app/websockets";
function sendSearchWSRequest(book) {
  websocket.send(book);
  console.log("Searching for: " + book);
}
```

The `sendSearchWSRequest()` function uses the WebSocket JavaScript API to send
the string `book` name to the `BookWebSocket` class shown in the following section.

The WebSockets endpoint

Here is the snippet for the WebSockets's `ServerEndpoint` annotated
class `BookWebSocket`:

```
@ServerEndpoint(value="/app/websockets")
public class BookWebSocket {
  @OnMessage
  public String searchBook(String name) {
    return "Found book " + name;
  }
}
```

The `BookWebSocket` is a POJO that is annotated with `@ServerEndpoint` and initialized
to a URI of `/app/websockets`. The `@OnMessage` annotation on the `searchBook()`
method will cause this method to be invoked when the WebSockets server endpoint
receives the message. The WebSockets endpoint simply returns back a string with the
name of the book for the case of simplicity of the sample.

Checking out a book

When a user clicks on the **Checkout** button on the HTML page, the input is checked
and then the `sendCheckoutRequest()` function is called in the JavaScript.

Using JavaScript

Here is the snippet of the `sendCheckoutRequest()` function:

```
function sendCheckoutRequest( book) {
  var req = createRequest(); // defined above
  ;
  // Create the callback:
  req.onreadystatechange = function() {

    if (req.readyState == 4) {
      document.getElementById("query").innerHTML=
        "DELETE app/library/book/" + encodeURI(book.trim());
      document.getElementById("output").innerHTML
        =req.responseText;

    }
  }
  req.open("DELETE","app/library/book/" + book,true);
  req.send(null);
}
```

The request, which is sent to the JAX-RS resource, is a DELETE request placed on the /app/library/book/ URI. We will cover the JAX-RS resource in the next section.

The JAX-RS resource method for the DELETE request

Here is the code snippet for the DELETE request:

```
@DELETE
@Path("book/{name}")
@Produces({MediaType.TEXT_PLAIN })
@Consumes({MediaType.TEXT_PLAIN })
public Book checkoutBook(@PathParam("name") String nameOfBook) {
  return bookService.deleteBook(nameOfBook);

}
```

This is a very simple method that will delete the book if it exists in the collection and send back the book details using the `BookWriter` class covered earlier.

Returning a book

When a user clicks on the **Return** button on the HTML page, the input is checked and then the `sendReturnRequest()` function is called in the JavaScript.

Using JavaScript

Here is the snippet of the `sendReturnRequest()` function:

```
function sendReturnRequest( book) {
  var req = createRequest(); // defined above
  ;
  // Create the callback:
  req.onreadystatechange = function() {

    if (req.readyState == 4) {
      document.getElementById("query").innerHTML=
        "POST app/library/book/" + encodeURI(book.trim());
      document.getElementById("output").innerHTML=
        req.responseText;

    }
  }
  req.open("POST","app/library/book/" + book,true);
  req.send(null);
}
```

The request, which is sent to the JAX-RS resource, is a POST request with `app/library/book/` as the target URI.

The JAX-RS resource method for the POST request

Here is the code snippet for the POST request:

```
@POST
@Path("book/{name}")
@Produces({MediaType.TEXT_PLAIN })
@Consumes({MediaType.TEXT_PLAIN })
public String returnBook(@PathParam("name") String nameOfBook)       {

  return "Successfully returned Book " + nameOfBook;
}
```

Placing a hold on a book

When a user clicks on the **Hold** button on the HTML page, the input is checked and then the `sendHoldRequest()` function is called in the JavaScript.

Using JavaScript

Here is the snippet of the `sendHoldRequest()` function:

```javascript
function sendHoldRequest( book) {
  var req = createRequest(); // defined above
  ;
  // Create the callback:
  req.onreadystatechange = function() {

    if (req.readyState == 4) {
      document.getElementById("query").innerHTML=
        "POST app/library/hold/" + encodeURI(book.trim());
      document.getElementById("output").innerHTML=
        req.responseText;

    }
  }
  req.open("POST","app/library/hold/" + book,true);
   req.send(null);

}
```

A `POST` request is sent to the JAX-RS resource located at the `app/library/hold/` URI. The resource is described in the next section.

The JAX-RS resource method for the asynchronous POST request

Here is the JAX-RS resource method that places a hold on a book. This is an asynchronous resource, which is covered in *Chapter 4*, *JSON and Asynchronous Processing*:

```java
/**
 * Asynchronously reply to placing a book on hold after sleeping for
sometime
 *
 */
@POST
@Produces({MediaType.TEXT_PLAIN})
@Path("hold/{name}")
public void asyncEcho(@PathParam("name") final String name,  @
Suspended final AsyncResponse ar) {
  TASK_EXECUTOR.submit(new Runnable() {
```

```
    public void run() {
      try {
        Thread.sleep(SLEEP_TIME_IN_MILLIS);
      } catch (InterruptedException ex) {
        ar.cancel();
      }
      ar.resume("Placed a hold for " + name);
    }
  });
}
```

The parameter `ar` of type `AsyncResponse` is similar to the `AsyncContext` class in the Servlet 3.0 specification and facilitates asynchronous request execution. In this example the request is suspended for a specific duration and the response is pushed to the client with the `AsyncResponse.resume()` method.

The Singleton EJB BookService

Here is the code for the Singleton EJB, which stores the details about the books:

```
@Singleton
public class BookService {

  private static final HashMap<String,Book> books =
    new HashMap<String,Book>();

  public static void addBook(Book book) {
    books.put(book.getName(), book);
  }

  public static int getSize() {
    return  books.size();
  }

  public static Book deleteBook(String isbn) {
    return books.remove(isbn);
  }


  public static List<Book> getBooks() {
    return new ArrayList<Book>(books.values());
  }
```

```
public BookService() {
  // initial content
  addBook( new Book("Java EE development using GlassFish
    Aplication Server","782345689","David Heffinger"));
  addBook( new Book("Java 7 JAX-WS Web Services",
    "123456789","Deepak Vohra"));
  addBook( new Book("Netbeans IDE7 CookBook","2234555567",
    "Rhawi Dantas"));
  addBook( new Book("Getting Started with RESTful WebServices",
    "11233333","Bhakti Mehta, Masoud Kalali"));

}
}
```

Thus, we have seen a detailed view of the library application, which uses the different JAX-RS 2.0, WebSockets, and JSON-P APIs.

Summary

This chapter covered two real life examples of RESTful Web Services. At the beginning, using the event notification sample, we demonstrated how to use Server-sent Events with asynchronous processing of servlets and how the server can push data to the clients as and when the events occur.

Continued on, in the library application we covered the JAX-RS API and also the custom message body readers, writers. We also demonstrated the use of JSON-P API. The library application showed how to use WebSockets from a JavaScript client and send messages to WebSockets endpoints.

Index

PACKT enterprise
PUBLISHING

professional expertise distilled

Thank you for buying
Developing RESTful Services with JAX-RS 2.0, WebSockets, and JSON

About Packt Publishing

Packt, pronounced 'packed', published its first book "Mastering phpMyAdmin for Effective MySQL Management" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: www.packtpub.com.

About Packt Enterprise

In 2010, Packt launched two new brands, Packt Enterprise and Packt Open Source, in order to continue its focus on specialization. This book is part of the Packt Enterprise brand, home to books published on enterprise software – software created by major vendors, including (but not limited to) IBM, Microsoft and Oracle, often for use in other corporations. Its titles will offer information relevant to a range of users of this software, including administrators, developers, architects, and end users.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.
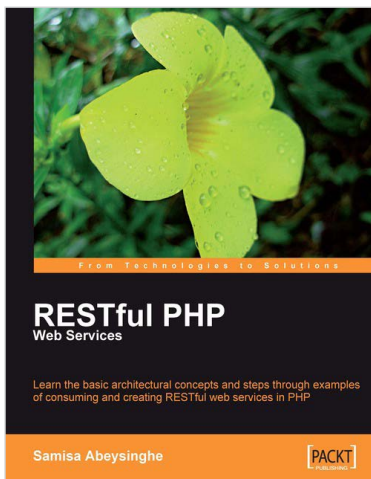
RESTful Java Web Services

ISBN: 978-1-84719-646-0   Paperback: 256 pages

Master core REST concepts and create RESTful web services in Java

1. Build powerful and flexible RESTful web services in Java using the most popular Java RESTful frameworks 2to date (Restlet, JAX-RS based frameworks Jersey and RESTEasy, and Struts 2)

2. Master the concepts to help you design and implement RESTful web services

3. Plenty of screenshots and clear explanations to facilitate learning

RESTful PHP Web Services

ISBN: 978-1-84719-552-4   Paperback: 220 pages

Learn the basic architechtural concepts and steps through examples of consuming and creating RESTful web service in PHP

1. Get familiar with REST principles

2. Learn how to design and implement PHP web services with REST

3. Real-world examples, with services and client PHP code snippets

4. Introduces tools and frameworks that can be used when developing RESTful PHP applications

Please check **www.PacktPub.com** for information on our titles

Java 7 JAX-WS Web Services

ISBN: 978-1-84968-720-1   Paperback: 64 pages

A practical, focused mini book for creating Web Services in Java 7

1. Develop Java 7 JAX-WS web services using the NetBeans IDE and Oracle GlassFish server

2. End-to-end application which makes use of the new clientjar option in JAX-WS wsimport tool

3. Packed with ample screenshots and practical instructions

Spring Web Services 2 Cookbook

ISBN: 978-1-84951-582-5   Paperback: 322 pages

Over 60 recipes providing comprehensive coverage of practical real-life implementation of Spring-WS

1. Create contract-first Web services

2. Explore different frameworks of Object/XML mapping

3. Secure Web Services by Authentication, Encryption/Decryption and Digital Signature

4. Learn contract-last Web Services using Spring Remoting and Apache CXF

5. Implement automated functional and load testing

Please check **www.PacktPub.com** for information on our titles