Java Back-End Web App
Tutorial Part 1: Building a Minimal App in Seven Steps

Learn how to build a back-end web application with minimal effort, using Java with Java Server Faces (JSF) as the user interface technology, the Java Persistence API (JPA) for object-to-storage mapping, and a MySQL database

Gerd Wagner <G.Wagner@b-tu.de>
Mircea Diaconescu <M.Diaconescu@b-tu.de>
Java Back-End Web App Tutorial Part 1: Building a Minimal App in Seven Steps: Learn how to build a back-end web application with minimal effort, using Java with Java Server Faces (JSF) as the user interface technology, the Java Persistence API (JPA) for object-to-storage mapping, and a MySQL database
by Gerd Wagner and Mircea Diaconescu

Warning: This tutorial may still contain errors and may still be incomplete in certain respects. Please report any issue to Gerd Wagner at G.Wagner@b-tu.de or Mircea Diaconescu at M.Diaconescu@b-tu.de.

This tutorial is also available in the following formats: PDF [minimal-tutorial.pdf]. See also the project page [http://web-engineering.info], or run the example app [http://yew.informatik.tu-cottbus.de/tutorials/java/minimalapp/] from our server, or download it as a ZIP archive file [MinimalApp.zip].

Publication date 2015-07-24
Copyright © 2014-2015 Gerd Wagner, Mircea Diaconescu

This tutorial article, along with any associated source code, is licensed under The Code Project Open License (CPOL) [http://www.codeproject.com/info/cpol10.aspx], implying that the associated code is provided "as-is", can be modified to create derivative works, can be redistributed, and can be used in commercial applications, but the article must not be distributed or republished without the authors' consent.
# Table of Contents

Foreword ................................................................................................................................... vi

1. A Quick Tour of the Foundations of Web Apps .......................................................................... 1
   1. The World Wide Web (WWW) ................................................................................................ 1
   2. HTML and XML ........................................................................................................................ 1
      2.1. XML documents ...................................................................................................................... 1
      2.2. Unicode and UTF-8 ................................................................................................................ 1
      2.3. XML namespaces ..................................................................................................................... 2
      2.4. Correct XML documents ......................................................................................................... 2
      2.5. The evolution of HTML ........................................................................................................... 3
      2.6. HTML forms ............................................................................................................................ 4
   3. Styling Web Documents and User Interfaces with CSS ............................................................ 5
   4. JavaScript - "the assembly language of the Web" ...................................................................... 6
      4.1. JavaScript as an object-oriented language .............................................................................. 6
      4.2. Further reading about JavaScript ........................................................................................... 7
2. Basic Elements of Java ............................................................................................................... 8
   1. Compared to JavaScript, what is different in Java? ................................................................. 8
   2. JavaBean Classes and Entity Classes ....................................................................................... 8
3. Building a Minimal Java Web App in Seven Steps ..................................................................... 10
   1. Step 1 - Set up the Folder Structure ......................................................................................... 11
   2. Step 2 - Write the Model Code ................................................................................................. 12
      2.1. Storing Book objects in a database table books ................................................................. 13
      2.2. Creating a new Book instance and saving it .......................................................................... 13
      2.3. Retrieving all Book instances ............................................................................................... 14
      2.4. Updating a Book instance .................................................................................................... 14
      2.5. Deleting a Book instance ..................................................................................................... 14
      2.6. Creating test data ................................................................................................................ 15
      2.7. Clearing all data .................................................................................................................. 15
   3. Step 3 - Configure and initialize the application ...................................................................... 15
      3.1. Create the EntityManager and UserTransaction objects .................................................. 16
      3.2. Configure the JPA database connection .............................................................................. 16
      3.3. Create the main template ..................................................................................................... 17
      3.4. Define the beans needed in facellet views .......................................................................... 19
      3.5. Build the WAR file and deploy it to TomEE ........................................................................ 19
   4. Step 5 - Implement the Create Use Case ............................................................................... 21
   5. Step 4 - Implement the Retrieve/List All Use Case ................................................................. 22
   6. Step 6 - Implement the Update Use Case .............................................................................. 24
   7. Step 7 - Implement the Delete Use Case ............................................................................... 25
   8. Run the App and Get the Code ............................................................................................... 26
   9. Possible Variations and Extensions ....................................................................................... 27
      9.1. Accessibility for Web Apps .................................................................................................. 27
      9.2. Using resource URLs .......................................................................................................... 27
      9.3. Offline availability ............................................................................................................... 27
10. Points of Attention ................................................................................................................ 27
    10.1. Code clarity ........................................................................................................................ 28
    10.2. Boilerplate code ................................................................................................................. 28
    10.3. Architectural separation of concerns ................................................................................ 28
4. Practice Projects ...................................................................................................................... 29
   1. Project 1 - Develop a Java Back-End App for Managing Movie Data .................................... 29
Glossary ................................................................................................................................... 30
List of Figures

3.1. The object type Book. ................................................................. 10
4.1. The object type Movie. ............................................................... 29
List of Tables

2.1. Java Visibility Level .............................................................................................................. 8
3.1. A collection of book objects represented as a table .............................................................. 13
Foreword

This tutorial is Part 1 of our series of six tutorials [http://web-engineering.info/JavaJpaJsfApp] about model-based development of back-end web applications with Java in combination with the Java Persistence API (JPA) and Java Server Faces (JSF) as a back-end platform. It shows how to build a simple web app with minimal functionality.

This tutorial provides example-based learning materials and supports learning by doing it yourself.

A distributed web app is composed of at least two parts: a front-end part, which, at least, renders the user interface (UI) pages, and a back-end part, which, at least, takes care of persistent data storage. A back-end web app is a distributed web app where essentially all work is performed by the back-end component, including data validation and UI page creation, while the front-end only consists of a web browser's rendering of HTML-forms-based UI pages. Normally, a distributed web app can be accessed by multiple users, possibly at the same time, over HTTP connections.

In the case of a Java/JPA/JSF back-end app, the back-end part of the app can be executed by a server machine that runs a web server supporting the Java EE specifications Java Servlets, Java Expression Language (EL), JPA and JSF, such as the open source server Tomcat/TomEE [http://tomee.apache.org/apache-tomee.html].

The minimal version of a Java data management application discussed in this tutorial only includes a minimum of the overall functionality required for a complete app. It takes care of only one object type ("books") and supports the four standard data management operations (Create/Read/Update/Delete), but it needs to be enhanced by styling the user interface with CSS rules, and by adding further important parts of the app's overall functionality:

• Part 2 [validation-tutorial.html]: Handling constraint validation.

• Part 3 [enumeration-tutorial.html]: Dealing with enumerations.

• Part 4 [unidirectional-association-tutorial.html]: Managing unidirectional associations between books and publishers, assigning a publisher to a book, and between books and authors, assigning authors to a book.

• Part 5 [bidirectional-association-tutorial.html]: Managing bidirectional associations between books and publishers and between books and authors, also assigning books to authors and to publishers.

• Part 6 [subtyping-tutorial.html]: Handling subtype (inheritance) relationships between object types.
Chapter 1. A Quick Tour of the Foundations of Web Apps

If you are already familiar with HTML, XML and JavaScript, you may skip this chapter and immediately start developing a minimal web application by going to the next chapter.

1. The World Wide Web (WWW)

After the Internet had been established in the 1980'ies, Tim Berners-Lee developed the idea and the first infrastructure components of the WWW in 1989 at the European research institution CERN in Geneva, Switzerland. The WWW (or, simply, "the web") is based on

- the basic Internet technologies TCP/IP and DNS,
- the Hypertext Transfer Protocol (HTTP),
- the Hypertext Markup Language (HTML) as well as the Extensible Markup Language (XML), and
- web server programs, acting as HTTP servers, as well as web 'user agents' (such as browsers), acting as HTTP clients.

2. HTML and XML

HTML allows to mark up (or describe) the structure of a human-readable web document or web user interface, while XML allows to mark up the structure of all kinds of documents, data files and messages, whether they are human-readable or not. HTML can be based on XML.

2.1. XML documents

XML provides a syntax for expressing structured information in the form of an XML document with elements and their attributes. The specific elements and attributes used in an XML document can come from any vocabulary, such as public standards or your own user-defined XML format. XML is used for specifying

- **document formats**, such as XHTML5, the Scalable Vector Graphics (SVG) format or the DocBook format,
- **data interchange file formats**, such as the Mathematical Markup Language (MathML) or the Universal Business Language (UBL),
- **message formats**, such as the web service message format SOAP [http://www.w3.org/TR/soap12-part0/]

2.2. Unicode and UTF-8

XML is based on Unicode, which is a platform-independent character set that includes almost all characters from most of the world's script languages including Hindi, Burmese and Gaelic. Each character is assigned a unique integer code in the range between 0 and 1,114,111. For example, the
Greek letter π has the code 960, so it can be inserted in an XML document as &amp;#960; using the XML entity syntax.

Unicode includes legacy character sets like ASCII and ISO-8859-1 (Latin-1) as subsets.

The default encoding of an XML document is UTF-8, which uses only a single byte for ASCII characters, but three bytes for less common characters.

Almost all Unicode characters are legal in a well-formed XML document. Illegal characters are the control characters with code 0 through 31, except for the carriage return, line feed and tab. It is therefore dangerous to copy text from another (non-XML) text to an XML document (often, the form feed character creates a problem).

### 2.3. XML namespaces

Generally, namespaces help to avoid name conflicts. They allow to reuse the same (local) name in different namespace contexts.

XML namespaces are identified with the help of a namespace URI (such as the SVG namespace URI "http://www.w3.org/2000/svg"), which is associated with a namespace prefix (such as "svg"). Such a namespace represents a collection of names, both for elements and attributes, and allows namespace-qualified names of the form prefix:name (such as "svg:circle" as a namespace-qualified name for SVG circle elements).

A default namespace is declared in the start tag of an element in the following way:

```xml
<html xmlns="http://www.w3.org/1999/xhtml">
```

This example shows the start tag of the HTML root element, in which the XHTML namespace is declared as the default namespace.

The following example shows a namespace declaration for the SVG namespace:

```xml
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  ...
  </head>
  <body>
    <figure>
      <figcaption>Figure 1: A blue circle</figcaption>
      <svg xmlns="http://www.w3.org/2000/svg">
        <circle cx="100" cy="100" r="50" fill="blue"/>
      </svg>
    </figure>
  </body>
</html>
```

### 2.4. Correct XML documents

XML defines two syntactic correctness criteria. An XML document must be well-formed, and if it is based on a grammar (or schema), then it must also be valid against that grammar.

An XML document is called well-formed, if it satisfies the following syntactic conditions:

1. There must be exactly one root element.

2. Each element has a start tag and an end tag; however, empty elements can be closed as `<phone/>` instead of `<phone></phone>`. 
3. Tags don't overlap, e.g. we cannot have

   `<author><name>Lee Hong</author></name>`

4. Attribute names are unique within the scope of an element, e.g. the following code is not correct:

   `<attachment file="lecture2.html" file="lecture3.html"/>`

An XML document is called **valid** against a particular grammar (such as a **DTD** or an **XML Schema**), if

1. it is **well-formed**,

2. and it **respects the grammar**.

### 2.5. The evolution of HTML

The World-Wide Web Committee (W3C) has developed the following important versions of HTML:

- 1997: **HTML 4** as an SGML-based language,
- 2000: **XHTML 1** as an XML-based clean-up of HTML 4,

As the inventor of the Web, Tim Berners-Lee developed a first version of HTML [http://www.w3.org/History/19921103-hypertext/hypertext/WWW/MarkUp/Tags.html] in 1990. In the following years, HTML has been used and gradually extended by a growing community of early WWW adopters. This evolution of HTML, which has led to a mess of tags and attributes, has been mainly controlled by browser vendors and their competition with each other. The development of XHTML in 2000 was an attempt by the W3C to clean up this mess, but it neglected to advance HTML's functionality towards a richer user interface, which was the focus of the WHAT working group [http://en.wikipedia.org/wiki/WHATWG] led by Ian Hickson [http://en.wikipedia.org/wiki/Ian_Hickson] who can be considered as the mastermind and main author of HTML 5 and many of its accompanying JavaScript APIs that made HTML fit for mobile apps.

HTML was originally designed as a **structure** description language, and not as a **presentation** description language. But HTML4 has a lot of purely presentational elements such as `font`. XHTML has been taking HTML back to its roots, dropping presentational elements and defining a simple and clear syntax, in support of the goals of

- device independence,
- accessibility, and
- usability.

We adopt the symbolic equation

\[
\text{HTML} = \text{HTML5} = \text{XHTML5}
\]

*stating that when we say "HTM"L or "HTML"5, we actually mean XHTML5*

because we prefer the clear syntax of XML documents over the liberal and confusing HTML4-style syntax that is also allowed by HTML5.
The following simple example shows the basic code template to be used for any HTML document:

```html
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
<meta charset="UTF-8" />
<title>XHTML5 Template Example</title>
</head>
<body>
<h1>XHTML5 Template Example</h1>
<section><h1>First Section Title</h1> ...
</section>
</body>
</html>

2.6. HTML forms

For user-interactive web applications, the web browser needs to render a user interface. The traditional metaphor for a software application's user interface is that of a form. The special elements for data input, data output and form actions are called form controls. An HTML form is a section of a document consisting of block elements that contain controls and labels on those controls.

Users complete a form by entering text into input fields and by selecting items from choice controls. A completed form is submitted with the help of a submit button. When a user submits a form, it is sent to a web server either with the HTTP GET method or with the HTTP POST method. The standard encoding for the submission is called URL-encoded. It is represented by the Internet media type application/x-www-form-urlencoded. In this encoding, spaces become plus signs, and any other reserved characters become encoded as a percent sign and hexadecimal digits, as defined in RFC 1738.

Each control has both an initial value and a current value, both of which are strings. The initial value is specified with the control element's value attribute, except for the initial value of a textarea element, which is given by its initial contents. The control's current value is first set to the initial value. Thereafter, the control's current value may be modified through user interaction or scripts. When a form is submitted for processing, some controls have their name paired with their current value and these pairs are submitted with the form.

Labels are associated with a control by including the control as a subelement of a label element ("implicit labels"), or by giving the control an id value and referencing this id in the for attribute of the label element ("explicit labels"). It seems that implicit labels are (in 2015) still not widely supported by CSS libraries and assistive technologies. Therefore, explicit labels may be preferable, despite the fact that they imply quite some overhead by requiring a reference/identifier pair for every labeled HTML form field.

In the simple user interfaces of our "Getting Started" applications, we only need three types of form controls:

1. single line input fields created with an `<input name="..." />` element,

2. push buttons created with a `<button type="button">...</button>` element, and

3. dropdown selection lists created with a select element of the following form:

```html
<select name="...">
 <option value="value1"> option1 </option>
 <option value="value2"> option2 </option>
 ...
</select>
```

An example of an HTML form with implicit labels for creating such a user interface is
In an HTML-form-based user interface (UI), we have a correspondence between the different kinds of properties defined in the model classes of an app and the form controls used for the input and output of their values. We have to distinguish between various kinds of **model class attributes**, which are typically mapped to various kinds of **input fields**. This mapping is also called **data binding**.

In general, an attribute of a model class can always be represented in the UI by a plain **input control** (with the default setting `type="text"`), no matter which datatype has been defined as the range of the attribute in the model class. However, in special cases, other types of **input controls** (for instance, `type="date"`), or other controls, may be used. For instance, if the attribute's range is an enumeration, a **select** control or, if the number of possible choices is small enough (say, less than 8), a radio button group can be used.

### 3. Styling Web Documents and User Interfaces with CSS

While HTML is used for defining the content structure of a web document or a web user interface, the **Cascading Style Sheets (CSS)** language is used for defining the **presentation style** of these web pages, which means that you use it for telling the browser how you want your HTML (or XML) rendered: using which layout of content elements, which fonts and text styles, which colors, which backgrounds, and which animations. Normally, these settings are made in a separate CSS file that is associated with an HTML file via a special **link** element in the HTML's **head**.

A first sketch of CSS [http://www.w3.org/People/howcome/p/cascade.html] was proposed in October 1994 by Håkon W. Lie [https://en.wikipedia.org/wiki/H%C3%A5kon_Wium_Lie] who later became the CTO of the browser vendor Opera. While the official CSS1 [http://www.w3.org/TR/REC-CSS1/] standard dates back to December 1996, "most of it was hammered out on a whiteboard in Sophia-Antipolis" by Håkon W. Lie together with Bert Bos in July 1995 (as he explains in an interview [http://www.w3.org/community/webed/wiki/A_Short_History_of_JavaScript]).

CSS is based on a form of rules that consist of selectors [https://docs.webplatform.org/wiki/tutorials/using_selectors], which select the document element(s) to which a rule applies, and a list of **property-value pairs** that define the styling of the selected element(s) with the help of CSS properties such as **font-size** or **color**. There are two fundamental mechanisms for computing the CSS property values for any page element as a result of applying the given set of CSS rules: inheritance and the cascade [https://docs.webplatform.org/wiki/tutorials/inheritance_and_cascade].

The basic element of a CSS layout [http://learnlayout.com/] is a rectangle, also called "box", with an inner content area, an optional border, an optional padding (between content and border) and an optional margin around the border. This structure is defined by the CSS box model [https://docs.webplatform.org/wiki/guides/the_css_layout_model].

We will not go deeper into CSS in this tutorial, since our focus here is on application logic and functionality, and not so much on page beauty.
4. JavaScript - "the assembly language of the Web"

JavaScript was developed in 10 days in May 1995 by Brendan Eich [http://en.wikipedia.org/wiki/Brendan_Eich], then working at Netscape [http://en.wikipedia.org/wiki/Netscape], as the HTML scripting language for their browser Navigator 2 (more about history [http://www.w3.org/community/webed/wiki/A_Short_History_of_JavaScript]). Brendan Eich said (at the O'Reilly Fluent conference in San Francisco in April 2015): "I did JavaScript in such a hurry, I never dreamed it would become the assembly language for the Web".

This section provides a brief overview of JavaScript, assuming that the reader is already familiar with basic programming concepts and has some experience with programming, for instance, in PHP, Java or C#.

JavaScript is a functional object-oriented programming language that can be used for

1. Enriching a web page by
   • generating browser-specific HTML content or CSS styling,
   • inserting dynamic HTML content,
   • producing special audio-visual effects (animations).

2. Enriching a web user interface by
   • implementing advanced user interface components,
   • validating user input on the client side,
   • automatically pre-filling certain form fields.

3. Implementing a front-end web application with local or remote data storage, as described in the book Building Front-End Web Apps with Plain JavaScript [http://web-engineering.info/JsFrontendApp-Book].

4. Implementing a front-end component for a distributed web application with remote data storage managed by a back-end component, which is a server-side program that is traditionally written in a server-side language such as PHP, Java or C#, but can nowadays also be written in JavaScript with NodeJS.

5. Implementing a complete distributed web application where both the front-end and the back-end components are JavaScript programs.

The current version of JavaScript is called ECMAScript 5.1, or simply "ES5", but the next version, ES6, with lots of added functionality and simplified syntaxes, is around the corner (and already partially supported by current browsers and back-end JS environments).

4.1. JavaScript as an object-oriented language

JavaScript is object-oriented, but in a different way than classical OO programming languages such as Java and C++. There is no explicit class concept in JavaScript. Rather, classes have to be defined in the form of special objects: either as constructor functions or as factory objects.
However, objects can also be created without instantiating a class, in which case they are *untyped*, and properties as well as methods can be defined for specific objects independently of any class definition. At run time, properties and methods can be added to, or removed from, any object and class. This dynamism of JavaScript allows powerful forms of *meta-programming*, such as defining your own concepts of classes or enumerations.

### 4.2. Further reading about JavaScript

Good open access books about JavaScript are

- Building Front-End Web Apps with Plain JavaScript [http://web-engineering.info/Jsf5AppBook], by Gerd Wagner
Chapter 2. Basic Elements of Java

1. Compared to JavaScript, what is different in Java?

1. **No program without a class**: Any Java program must include at least one class.

2. **No object without a class**: For creating an object, a class has to be used (or defined) for
   - defining the properties of the object's property slots
   - defining the methods and functions that can be applied to the object (and all other objects
     instantiating the class)

3. **No global variables, no global methods**: In Java, all variables and methods must be defined in the
   context of a class, which provides their name space.

4. Classes, properties and methods are defined with a **visibility level**: public, protected or
   private.

5. **Java is strongly typed**: Properties, parameters and variables must be declared to be of some type.

6. **Type parameters**: Classes and complex data structures (such as lists) can be defined with the help of
   type parameters. See, for instance, this tutorial [https://docs.oracle.com/javase/tutorial/java/generics/
   types.html].

7. **Arrays are static**: Arrays have a fixed size, which cannot be changed at run-time. See also the section
   on arrays below.

8. **Java programs must be compiled** before they can be executed.

9. **Speed**: Java is about twice as fast as optimized JavaScript.

It is important to understand that in Java, visibility level are used to define the possible levels of access:

<table>
<thead>
<tr>
<th>visibility level</th>
<th>Class</th>
<th>Package</th>
<th>Subclass</th>
<th>World</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>protected</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>n</td>
</tr>
<tr>
<td>no modifier</td>
<td>y</td>
<td>y</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>private</td>
<td>y</td>
<td>n</td>
<td>n</td>
<td>n</td>
</tr>
</tbody>
</table>

Normally, properties are defined as private, with public getters and setters, so they are only directly
accessible at the level of the class defining them.

2. JavaBean Classes and Entity Classes

A **JavaBean class** (or, simply, **bean class**) is a Java class with a default constructor where all properties
are serializable and have a get and set method (also called "getter" and "setter"). A Java bean is an
object created with the help of a bean class.
A **JPA entity class** (or, simply, *entity class*) is a JavaBean class with an `@Entity` annotation, implying that a JavaEE runtime environment (such as provided by the TomEE PLUS [http://tomee.apache.org/apache-tomee.html] web server) will take care of the persistent storage of its instances.
Chapter 3. Building a Minimal Java Web App in Seven Steps

In this chapter, we build a simple distributed Java web application with the Java Persistence API (JPA) and Java Server Faces (JSF). Such an application requires a web server (as a back-end) environment executing the Java code, but also consists of HTML, CSS and JavaScript front-end code that is executed on the user's computer.

JPA is a Java API for the management of persistent data in Java applications. It uses the Java Persistence Query Language (JPQL), a platform-independent object-oriented query language, which is heavily inspired by SQL, and its query expressions are quite similar to SQL query expressions, but they are executed in the context of JPA entity objects.

JSF is a Java specification for building component-based user interfaces for web applications. Its current version, JSF 2, by default, uses Facelets as its template technology. Other view technologies such as XUL can also be employed. In contrast, JSF 1 has used JavaServer Pages (JSP) as its default template technology.

In this tutorial, we show how to deploy and run the example app using the TomEE web server, which provides an execution environment for Java web applications. We assume that you have already installed the TomEE PLUS [http://tomee.apache.org/apache-tomee.html] web server, the MySQL [http://www.mysql.com/] DBMS, and the ANT [https://ant.apache.org/] build tool on your computer.

The purpose of our example app is to manage information about books. For simplicity, in this chapter we deal with a single object type Book, as depicted in Figure 3.1.

Figure 3.1. The object type Book.

<table>
<thead>
<tr>
<th>Book</th>
</tr>
</thead>
<tbody>
<tr>
<td>isbn  : String</td>
</tr>
<tr>
<td>title : String</td>
</tr>
<tr>
<td>year  : Integer</td>
</tr>
</tbody>
</table>

What do we need for such a data management app? There are four standard use cases, which have to be supported by the app:

1. **Create** a new book record by allowing the user to enter the data of a book that is to be added to the collection of stored book records.

2. **Retrieve** (or read) all books from the data store and show them in the form of a list.

3. **Update** the data of a book record.


These four standard use cases, and the corresponding data management operations, are often summarized with the acronym CRUD.

For entering data with the help of the keyboard and the screen of our computer, we can use **HTML forms**, which provide the **user interface** technology for web applications.
For any data management app, we need a technology that allows to store data in persistent records on a secondary storage device, such as a hard-disk or a solid state disk. JPA allows using a great number of different data storage technologies, including many SQL database management systems (DBMS) such as Oracle, MySQL and PostgreSQL. We don’t have to change much in the application code for switching from one storage technology to another. Adding the right driver implementation to our Java runtime environment, properly setting up the DBMS and changing the database access configuration is all we need to do. Below, in step 3, we explain how to set up the JPA configuration for MySQL.

1. Step 1 - Set up the Folder Structure

In the first step, we set up our folder structure for the application program code. The application name in this chapter will be "Public Library", and we will use a corresponding name for the application folder, "publicLibrary". Then we create the application structure. There are many ways to do this, like for example to use the Eclipse [http://www.eclipse.org/eclipse/] development environment (and create and configure a Dynamic Web Project). In this tutorial we show how to do it manually, so there is no need to use special tools, except ANT [http://ant.apache.org/] for being able to compile and deploy the application more easily. For your convenience, we provide an ANT script (available for download at the end of this tutorial), which is able to create the folder structure of a web application, compile it to a Web application Archive (WAR) file and then deploy it to a TomEE web server for execution. The application structure (which is also compatible with the Dynamic Web Project structure of Eclipse, so it can be imported in Eclipse) is the following:

```
publicLibrary
src
  pl
    model
    ctrl
META-INF
  persistence.xml
WebContent
  resources
    css
    media
  views
    books
WEB-INF
  lib
    templates
  faces-config.xml
  web.xml
```

This folder structure has the following parts:

1. the src folder contains the model and controller code in its subfolders `pl/model` and `pl/ctrl`, while the view/UI code is contained in the WebContent folder. The pl subfolder defines the package name for the Java application (it's a shortcut for 'public library').

2. the src/META-INF folder contains the configuration files. The most important one (and the only one we need for this application) is the persistence.xml file. It contains the configuration required to make the connection with the database. The content of this file is discussed in Section 3.2.

3. the WebContent folder contains various web resources, including template files and custom view files.

4. the WebContent/resources contains all the public available files, such as CSS and media files

5. the WebContent/views stores our custom view files for the application, so it represents the views part of the MVC paradigm. Please note that it is not strictly required to name it views, but it makes
a lot of sense to do it so, since this is what this folder represents. We will discuss in details about its content, later in this tutorial.

6. the `WebContent/WEB-INF` folder contains the used libraries (jar files) of the project (as part of the `lib` subfolder), the JSF template files for your pages (as part of the `templates` subfolder), the `faces-config.xml` file, which stores the facelets configuration data and the `web.xml` configuration file, specific to the Tomcat (TomEE) environment server used to run our application.

2. **Step 2 - Write the Model Code**

In the second step, we create the model classes for our app, using one Java source code file (with extension `.java`) for each model class. In the information design model shown in Figure 3.1 above, there is only one class, representing the object type `Book`. So, we create a file `Book.java` in the folder `src/pl/model` with the following code:

```java
@Entity
@Table(name="books")
public class Book {
    @Id private String isbn;
    private String title;
    private int year;
    // default constructor, required for entity classes
    public Book() {
    }
    // constructor
    public Book(String isbn, String title, int year) {
        this.setIsbn(isbn);
        this.setTitle(title);
        this.setYear(year);
    }
    public String getIsbn() { return isbn; }
    public void setIsbn(String isbn) { this.isbn = isbn; }
    public String getTitle() { return title; }
    public void setTitle(String title) { this.title = title; }
    public int getYear() { return year; }
    public void setYear(int year) { this.year = year; }

    public static List<Book> getAllObjects(...) {...}
    public static Book getObjectByStdId(...) {...}
    public static void add(...) {...}
    public static void update(...) {...}
    public static void destroy(...) {...}
    public static void clearData(...) {...}
    public static void createTestData(...) {...}
}
```

Notice that the model class `Book` is encoded as a JavaBean class enriched with the following JPA annotations:

1. The annotation `@Entity` specifies that the instances of this class will be stored persistently.

2. `@Table(name="books")` specifies the name of the database table to be used for storing the instances of `Book`. This annotation is optional and defaults to a table name being the same as the class name but in lower case (that is, it would be `book` if our case).

3. The `@Id` annotation is used to specify which attribute represents the standard identifier, and, as a consequence, which column of the underlying database table is used as the primary key. In our example, `isbn` is used as the standard identifier attribute, and the corresponding `isbn` column from the `books` table stores the primary key values.

In the entity class `Book`, we also define the following class-level methods:

1. A class-level method `Book.getObjectByStdId` for retrieving a specific `Book` instance from the persistent data store by means of its standard identifier.

3. A class-level method `Book.getAllObjects` for retrieving all `Book` instances from the persistent data store.


5. A class-level method `Book.destroy` for deleting a `Book` instance.

6. A class-level method `Book.createTestData` for creating a few example book records to be used as test data.


These methods are discussed in the following sections.

### 2.1. Storing `Book` objects in a database table

The instances of our entity class `Book` are special Java objects representing "entities" (or business objects), which can be serialized, or, in other words, represented and stored as records or rows of a database table. Consequently, they can be shown as a table like in Table 3.1.

#### Table 3.1. A collection of book objects represented as a table

<table>
<thead>
<tr>
<th>ISBN</th>
<th>Title</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>006251587X</td>
<td>Weaving the Web</td>
<td>2000</td>
</tr>
<tr>
<td>0465026567</td>
<td>Gödel, Escher, Bach</td>
<td>1999</td>
</tr>
<tr>
<td>0465030793</td>
<td>I Am A Strange Loop</td>
<td>2008</td>
</tr>
</tbody>
</table>

The data storage technology used in our example app is MySQL [http://www.mysql.com/]. The (My)SQL code used to create the database table schema for `books` is the following:

```sql
CREATE TABLE IF NOT EXISTS books ( 
    isbn VARCHAR(10) NOT NULL PRIMARY KEY, 
    title VARCHAR(128), 
    year SMALLINT 
); 
```

While it is also possible to create the database schema manually (with the help of CREATE TABLE statements such as the one above), we'll show later in this tutorial how to do this can be automatically generated by JPA. In both cases, the database, user and the associated rights (create, update, etc) must be done manually before the JPA application tries to connect to it (when deployed to the web server).

### 2.2. Creating a new `Book` instance and saving it

The `Book.add` method takes care of creating a new `Book` instance and saving it to a database with the help of a JPA environment's entity manager:

```java
public static void add(EntityManager em, UserTransaction ut, 
    String isbn, String title, int year) throws Exception { 
    ut.begin();
    Book book = new Book( isbn, title, year);
```
14

For saving the new object to the database, the `persist` method of the `EntityManager` object is invoked. It is responsible for creating the corresponding SQL INSERT statement and executing it.

Since in a JPA runtrim environment, database write operations are executed in the context of a transaction, our add method has a parameter `ut` of type `UserTransaction`. Before the entity manager can invoke the database write method `persist`, a transaction needs to be started with `ut.begin()`. After all write (and state change) operations have been performed, the transaction is completed (and all changes are committed) with `ut.commit()`.

### 2.3. Retrieving all Book instances

The instances of an entity class, such as `Book`, are retrieved from the database with the help of query expressed in the *Java Persistence Query Language* (JPQL [http://en.wikipedia.org/wiki/Java_Persistence_Query_Language]). JPQL queries are similar to SQL queries. They use class names instead of table names, property names instead of column names, and object variables instead of row variables.

```java
public static List<Book> getAllObjects( EntityManager em) {
    Query query = em.createQuery( "SELECT b FROM Book b", Book.class);
    List<Book> books = query.getResultList();
    return books;
}
```

In the `Book.getAllObjects` method, first a query asking for all `Book` instances is created, and then this query is executed with `query.getResultList()` assigning its answer set to the list variable `books`.

### 2.4. Updating a Book instance

For updating an existing `Book` instance we first retrieve it from the database with `em.find`, and then re-set those attributes the value of which has changed:

```java
public static void update( EntityManager em, UserTransaction ut, 
    String isbn, String title, int year) throws Exception {
    ut.begin();
    Book book = em.find( Book.class, isbn);
    if (!title.equals( book.getTitle())) book.setTitle( title);
    if (year != book.getYear()) book.setYear( year);
    ut.commit();
}
```

Notice that, when invoking the `find` method for retrieving an entity, the first argument must be a reference to the entity class concerned, so the JPA runtime environment can identify the database table from which to retrieve the entity's data. The second argument must be the value of the entity's primary key.

Notice that in the update case, we do not have to use `persist` for saving the changes. This is being automatically managed by the JPA runtime environment when we complete the transaction with `ut.commit()`.

### 2.5. Deleting a Book instance

A book entity can be deleted from the database as shown in the following example code:

```java
em.persist( book);
ut.commit();
}
```
To delete an entity from the database, we first need to retrieve it with the help of the `find` method as in the update case. Then, the `remove` method has to be invoked by the entity manager, and finally the transaction is completed with `ut.commit()`.

### 2.6. Creating test data

For being able to test our code, we may create some test data and save it in our database. We can use the following procedure for this:

```java
public static void createTestData( EntityManager em, UserTransaction ut) throws Exception {
    Book book = null;
    Book.clearData( em, ut);  // first clear the books table
    ut.begin();
    book = new Book("006251587X","Weaving the Web", 2000);
    em.persist( book);
    book = new Book("0465026567","Gödel, Escher, Bach", 1999);
    em.persist( book);
    book = new Book("0465030793","I Am A Strange Loop", 2008);
    em.persist( book);
    ut.commit();
}
```

As we can see, we just perform a succession of creating 3 instances of the `Book` entity class and saving them with the help of `persist`.

### 2.7. Clearing all data

The following procedure clears our database by deleting all rows:

```java
public static void clearData( EntityManager em, UserTransaction ut) throws Exception {
    ut.begin();
    Query deleteStatement = em.createQuery( "DELETE FROM Book");
    deleteStatement.executeUpdate();
    ut.commit();
}
```

JPA does not provide a direct method to drop the entire population of a specific class from the database. However, this can be easily obtained by using a JPQL statement as shown in the above code. The JPQL code can be read as: *delete all rows from the database table associated with the entity class `Book`*. 

### 3. Step 3 - Configure and initialize the application

In this section we show how to prepare your application for being able to run it on a TomEE Web Server. Specifically, we learn how to configure the application to connect with a specific database, how to obtain the `EntityManager` and `UserTransaction` instances required for performing database operations, such as `persist` and `remove`, and how to pack our application to a WAR file and the deploy it to a web server for execution.
3.1. Create the EntityManager and UserTransaction objects

We use a controller class for providing the code that glues the views to the model, as well as for providing other methods which do no neither belong to the model nor to the view, like getting a connection with the database server. In our example app, this class is `pl.ctrl.BookController` in the `src/pl/ctrl` folder.

JPA requires an `EntityManager` object to execute JPQL queries (with `SELECT`) and data manipulation statements (with `INSERT`, `UPDATE` and `DELETE`). Also, in order to perform most database write operations, a `UserTransaction` object is required for starting and completing transactions. In a standalone application, the programmer has to create instances of the `EntityManager` and `UserTransaction` manually, using a factory pattern as shown in the following example code fragment:

```java
EntityManagerFactory emf = Persistence.createEntityManagerFactory("MinimalApp");
EntityManager em = emf.createEntityManager();
EntityTransaction et = em.getTransaction();
```

A JPA-enabled Java web application normally runs in a runtime environment called "container" (in our case this is TomEE), which takes care of creating the `EntityManager` and `UserTransaction` objects if the right annotations are used. The code responsible for this is part of the controller class (e.g., `pl.ctrl.BookController`) since the controller is responsible for managing the database connections.

```java
public class BookController {
    @PersistenceContext( unitName="MinimalApp")
    private EntityManager em;
    @Resource() UserTransaction ut;

    public List<Book> getBooks() {...}
    public void refreshObject( Book book) {...}
    public String add( String isbn, String title, int year) {...}
    public String update( String isbn, String title, int year) {...}
    public String destroy( String isbn) {...}
}
```

A closer look at this code shows that it is sufficient to use the `@PersistenceContext` annotation and to specify the used `unitName` (discussed in the next section) for obtaining an `EntityManager` instance at runtime. Also, obtaining a `UserTransaction` instance at runtime is as simple as using the `@Resource` annotation for the user transaction reference property `ut`. Not only that the required code is short and simple, but if the database type is changed (e.g. when we switch from MySQL to an Oracle database), this code remains the same.

3.2. Configure the JPA database connection

In the previous section, discussing the `BookController` class, we have shown how to obtain the `EntityManager` and `UserTransaction` objects in order to be able to perform database operations. The `@PersistenceContext` annotation of the `EntityManager` reference property requires a `unitName`, which is just a name used for identifying the storage management configuration defined in the `src/META-INF/persistence.xml` file. In our example app this file has the following content:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="http://java.sun.com/xml/ns/persistence"
```
<p>The configuration name ("MinimalApp") is defined by the name attribute of the persistence-unit element. This is the value we have to use for the unitName property of the @PersistenceContext annotation.</p>

The persistence-unit element has three content parts:

1. One or more class elements, each one containing the full qualified name of an entity class of the app (like pl.model.Book in our example app).

2. A set of configuration property elements used for providing further configuration settings.

3. A jta-data-source element for specifying the configuration block in the config/TomEE.xml web server configuration file in the web server installation folder.

In our persistence.xml file, two configuration properties have been set:

- javax.persistence.schema-generation.database.action, with the possible values: none (default), create, drop-and-create and drop. It specifies if the database schema is to be automatically created and additionally allows to drop the existing tables before creating the new ones (with drop or drop-and-create).

- javax.persistence.schema-generation.create-source, with the possible values metadata (default), script, metadata-then-script and script-then-metadata. It specifies the source of information used to create the database schema. The value metadata enforces using the JPA annotations while the value script allows using an external DDL script for defining the schema.

The jta-data-source element of our persistence.xml file refers to the Resource element with id value "MinimalApp" in the config/TomEE.xml file, which has the following content:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<tomee>
  <resource id="MinimalApp" type="DataSource">
    <jdbcDriver>com.mysql.jdbc.Driver</jdbcDriver>
    <jdbcUrl>jdbc:mysql://localhost:3306/minimalapp</jdbcUrl>
    <userName>minimalapp</userName>
    <password>minimalapp</password>
    <jtaManaged>true</jtaManaged>
  </resource>
</tomee>
```

The Resource element contains the information required to connect with the database (i.e. username, password, access URL and the Java class name of the connection driver).

### 3.3. Create the main template

The main template, called page.xhtml, is shown below. It has two sub-templates:
1. header.xhtml defines the general header information items (such as the application name)

2. footer.xhtml defines the general footer information items (such as a copyrights notice)

Both sub-templates are included in the main template with the help of a \texttt{ui:include} element. We add all three template files to the WebContent/WEB-INF/templates folder.

The content of our HTML5-compliant main template page.xhtml is the following:

```html
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
     xmlns:ui="http://java.sun.com/jsf/facelets"
     xmlns:h="http://java.sun.com/jsf/html">
    <h:head>
        <title><ui:insert name="title">Public Library</ui:insert></title>
        <link href="#{facesContext.externalContext.requestContextPath}/resources/css/style.css"
             rel="stylesheet" type="text/css" />
    </h:head>
    <body>
        <div id="header">
            <ui:insert name="header">
                <ui:include src="/WEB-INF/templates/header.xhtml"/>
            </ui:insert>
        </div>
        <div id="main-content">
            <ui:insert name="main-content"/>
        </div>
        <div id="footer">
            <ui:insert name="footer">
                <ui:include src="/WEB-INF/templates/footer.xhtml"/>
            </ui:insert>
        </div>
    </body>
</html>
```

In the code, one can see that some HTML elements are used (e.g., title, link or div) while others like h:head and ui:insert are not HTML elements, but have been defined by JSF in different namespaces. JSF is using its own head element h:head because it allows injecting special HTML code such as script elements needed for XHR (or "AJAX") messaging.

Notice that in the main template, we have a first example of an expression using JSF's \texttt{Expression Language (EL)}, where an expression starts with \# and is encapsulated between curly brackets, like \#{expression}. Such an expression allows reading the value of a property of, or invoking a method on, a Java bean or a context object. In any case, the value of the expression will be inserted into the HTML generated from the template. The expression in our main template is the expression \#{facesContext.externalContext.requestContextPath}, which retrieves the value of the requestContextPath property of the context object faceContext.externalContext.

Our main template defines three content regions: header, main-content and footer. The header and footer regions are defined by sub-templates included with the help of the \texttt{ui:include} element.

The header.xhtml sub-template contains the following:

```
<ui:insert name="header">
    <ui:include src="/WEB-INF/templates/header.xhtml"/>
</ui:insert>
```

The footer.xhtml sub-template contains the following:

```
<ui:insert name="footer">
    <ui:include src="/WEB-INF/templates/footer.xhtml"/>
</ui:insert>
```

The main region is dynamic, and will be replaced with the content generated by a facelet view as shown below. Notice that both the file extension of template files as well as of facelet view files is \texttt{xhtml}.

A set of namespaces are used for JSF views:
Building a Minimal Java Web App in Seven Steps

• xmlns:ui="http://java.sun.com/jsf/faceslets" refers to the JSF Facelets Tag Library, used for templating reasons. One example is ui:define, used to specify on which region of a template to inject the facelet.

• xmlns:h="http://java.sun.com/jsf/html" refers to the JSF HTML Tag Library, used to UI elements which are then rendered as HTML components. For example h:inputText, which is rendered as an HTML input element.

• xmlns:f="http://java.sun.com/jsf/core" refers to the JSF Core Tag Library which specifies custom actions or elements that are independent of any particular render kit. For example, f:actionListener can be used to define a Java method which is executed when the user clicks a button.

• xmlns:p="http://xmlns.jcp.org/jsf/passthrough" provide a solution to allow additional features, like new HTML5 features, which are not part of the JSF HTML Tag Library. For example, it can be used for the type attribute of a JSF inputText element to obtain a HTML5 color widget, e.g., <h:inputText p:type="color"> renders the HTML5 element <input type="color">.

• xmlns:c="http://java.sun.com/jsp/jstl/core" refers to JSTL Core Tag Library which provide all kind of features, like dealing with loops and defining variables. For example, we can use <c:set var="isbn" value="#{book.isbn}"/> to create a variable named isbn which can be then used in the view code for conditional expressions.

• xmlns:fn="http://java.sun.com/jsp/jstl/functions" refers to JSTL Functions Tag Library and provide various utility functions, such as string converters. For example, we can use fn:toUpperCase to convert a string to its uppercase representation.

3.4. Define the beans needed in facelet views

We can create Java beans to be used in facelet views with the help of the @ManagedBean annotation, which allows defining the name of a variable for accessing the bean from the view code, typically in an EL expression. In our example app, we want to access Book beans as well as BookController beans, therefore both classes have to be annotated as follows:

@ManagedBean( name="book")
@SessionScoped
public class BookController { ... }

A bean lifetime scope can be specified with annotations. In our example the book bean is @RequestScoped, this means the instance exists as long as the HTTP request and the associated response are being processed. The bookCtrl bean is @SessionScoped, which means it is created when the session starts and destroyed when the session is closed. Other scopes are available, but we only use these two scopes.

3.5. Build the WAR file and deploy it to TomEE

In this tutorial we show how to use an ANT [https://ant.apache.org/] script for generating the structure of a Java web app, and then compile the code, build the WAR file and deploy it to a TomEE web server. One may also use Eclipse (or NetBeans or other IDEs) for doing this, but for keeping it simple we use
Building a Minimal Java Web App in Seven Steps

ANT. Our ANT script generates a folder structure, which is compatible with Eclipse, so in case you want to use Eclipse, you may simply create an Eclipse project from the existing application code.

The purpose of this section is only to show you how to use our ANT script for making your life easier. It is not intended to be an ANT tutorial, so we don't get into specific ANT details. The following ANT tasks are defined in the script:

- **create app**
  
  ```
  create app -Dappname=yourAppName -Dpkgname=yourAppPackageName
  ```

  allows creating the folder structure. Instead of `yourAppName` and `yourAppPackageName` you have to provide your app's name and its package name. In our example app, we invoke the task with `ant create app -Dappname=publicLibrary -Dpkgname=pl`.

  The script creates the folder structure, as well as the required files `src/META-INF/persistence.xml`, `WEB-INF/faces-config.xml` and `WEB-INF/web.xml`. The parameter `yourAppPackageName` is used to create the Java top level packages. If omitted, `yourAppName` is used as Java top package name instead. For the next tasks/commands you have to be sure that the ANT script file is located in the same folder as your web application folder (and not one level deeper in the web application folder). This way, one can use the same ANT script for building multiple web applications.

  Using the optional parameter, `-Dforce=true` will overwrite an existing application by first deleting the existing application folder.

  Hint: a JPA/JSF application requires a set of libraries to run. The ANT script looks for the `jar` files in a folder named `lib` located on the same folder as the script itself. The location of the jar files can be modified by editing the ANT script and setting the `lib.folder` parameter to the right folder on your computer. You can download the dependency JAR files with the link provided at the end of this tutorial.

- **build war**
  
  ```
  build war -Dappname=yourAppName
  ```

  allows building the WAR file by using `yourAppName` as file name. The resulting WAR file will be in the same folder as the ANT script file. For our example app we use the following command:

  ```
  ant war -Dappname=publicLibrary
  ```

  Hint: before being able to use this command, you have to edit the ANT script and modify the value of the `server.folder` parameter so it points to your TomEE installation folder. In case that you get compilation errors, try to copy the `mysql-connector-java-xxxxx-bin.jar` file to the `lib` folder of your TomEE installation folder. This file and some other dependency files are provided in a ZIP archive that can be downloaded with the link provided at the end of this tutorial.

- **deploy**
  
  ```
  deploy -Dappname=yourAppName
  ```

  allows deploying the WAR file associated with `yourAppName` to the TomEE web server. It automatically executes the `build war` command, which means
Building a Minimal Java
Web App in Seven Steps

the WAR file is built before the deploy. The location of the deploy folder is detected by using the server.folder property, by appending the webapps folder name. For our example app we invoke the following command: ant deploy -Dappname=publicLibrary.

Hint: we do not recommend using spaces in folder names, but if for any reason, the application name needs to contain spaces, then it has to be enclosed in double quotes, e.g. create app -Dappname="Hellow World".

4. Step 5 - Implement the Create Use Case

The Create use case involves creating a new object in main memory and then saving it to persistent storage with the help of the add method.

The corresponding add action method code from the src/pl/ctrl/BookController.java is shown below:

```java
public String add( String isbn, String title, int year) {
    try {
        Book.add( em, ut, isbn, title, year);
        // Clear the form after creating the Book record
        FacesContext facesContext = FacesContext.getCurrentInstance();
        facesContext.getExternalContext().getRequestMap().remove( "book");
    } catch ( Exception e) { e.printStackTrace(); }
    return "create";
}
```

The action method uses the Book.add method to create and save a Book instance. The add controller action method returns a string which is the name (without the extension) of the view file (found under the same folder as the view which triggered the action) which will be displayed after executing the action (i.e. create.xhtml in our case). The marked code lines are used to clear the form after creating a Book (other methods are also possible, this being the easiest one without side effects). The add method is part of the model code because the operations it performs are model specific:

```java
public static void add( EntityManager em, UserTransaction ut, String isbn, String title, int year) throws Exception {
    ut.begin();
    Book book = new Book( isbn, title, year);
    em.persist( book);
    ut.commit();
}
```

Now we need to create the view file corresponding to the create object use case. The file is WebContent/views/books/create.xhtml and it has the following content:

```html
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
     xmlns:ui="http://java.sun.com/jsf/facelets"
     xmlns:h="http://java.sun.com/jsf/html"
     xmlns:p="http://xmlns.jcp.org/jsf/passthrough">

<ui:composition template="/WEB-INF/templates/page.xhtml">
    <ui:define name="main-content">
        <h:form id="createBookForm">
            <h:panelGrid columns="2">" />
            <h:outputText value="ISBN: " />
            <h:inputText id="isbn" value="#{book.isbn}" />
        </h:form>
    </ui:define>
</ui:composition>
</html>
```
5. Step 4 - Implement the **Retrieve/List All** Use Case

This use case corresponds to the "Read" from the four basic data management use cases Create-Read-Update-Delete (CRUD).

First of all, for the list objects use case we have to add a method in the controller class (src/pl/ctrl/BookController.java file) which reads all the Book records from the database books table and then delivers this information to the view. The controller action method code is shown below:

```java
public class BookController {
    public List<Book> getBooks() {
        return Book.getAllObjects(em);
    }
    ...
}
```

The getBooks method returns a list of Book instances which are obtained by calling the static getAllObjects method of the Book model class. The BookController action methods invoke the Book model class operations (add, update, destroy) and, in addition, perform certain UI operations. The code of the Book.getAllObjects method is:

```java
public static List<Book> getAllObjects( EntityManager em) {
    Query query = em.createQuery( "SELECT b FROM Book b");
    ```
List<Book> books = query.getResultList();
return books;
}

The code is simple, and as already discussed in Section 2.3, it uses a JPQL statement to extract the Book records from the books table and create the corresponding instances of the Book model class. The EntityManager object required for being able to perform the JPQL query is parsed as parameter to Book.getAllObjects method, this object being created in the BookController controller class as discussed in Section 3.1 section.

Now, it is the time to define the facelet (view block) which is used to display a table with all the Book records found in the database. The view files corresponding to our application are located under WebContent/views/books folder. For the list objects use case, we create a file named listAll.xhtml with the following content:

```html
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
     xmlns:ui="http://java.sun.com/jsf/facelets"
     xmlns:h="http://java.sun.com/jsf/html"
     xmlns:f="http://java.sun.com/jsf/core">

<ui:composition template="/WEB-INF/templates/page.xhtml">
     <ui:define name="main-content">
<h:dataTable value="#{bookCtrl.books}" var="b">
    <h:column>ISBN: </h:column>
    <f:facet name="header">#{b.isbn} </f:facet>
    </h:column>
    <h:column>Title: </h:column>
    <f:facet name="header">#{b.title} </f:facet>
    </h:column>
    <h:column>Year: </h:column>
    <f:facet name="header">#{b.year} </f:facet>
    </h:column>
</h:dataTable>
<h:button value="Main menu" outcome="index" />
</ui:define>
</ui:composition>
</html>
```

The `ui:composition` element specify to which template file applies (i.e. `template="/WEB-INF/templates/page.xhtml"`) this view block and also specifies which view block (`<ui:define name="main-content"/>`) is replaced by this facelet at render time.

The `h:dataTable` element allows to define a table view of a set of records (i.e. Book records in our case), which is then rendered as a normal HTML5 table. Its `@value` attribute allows to specify the source of records, while the `@var` attribute allows to use a name to access the current bound value in the loop over the collection of records. The expression used in the `h:dataTable/@value` normally specifies a property name for which a getter was defined, which means that in the controller `BookController` class a property `books` is defined (optional) as well as a method named `getBooks` (this is mandatory). In this particular case it is just sufficient to define the `getBooks` method since there is no need of the `books` property in the controller class. In any case, the expression used in the template `h:dataTable/@value` does not allow to invoke a method, so we cannot call `getBooks` directly. Instead we ask to invoke the property `books` by using `#{bookCtrl.books}` expression, which internally evaluates to a call of a method named `getBooks` (a getter for the `books` property) without checking if a `books` property really exists.

The `h:button` JSF element allows to create redirect buttons. The value of the `@outcome` attribute specifies a name of a JSF view file by omitting the `.xhtml` extension (i.e. the view file name is `index.xhtml`).
6. Step 6 - Implement the **Upate** Use Case

This use case corresponds to the "Update" from the four basic data management use cases Create-Read-Update-Delete (CRUD).

As for the *create object* use case, a controller action method is defined in the `BookController` class *(src/pl/ctrl/BookController.java file)* with the following code:

```java
public String update( String isbn, String title, int year) {
    try {
        Book.update( em, ut, isbn, title, year);
    } catch ( Exception e) {
        e.printStackTrace();
    }
    return "update";
}
```

The `Book.update` takes care of updating and making persistent changes for the book entry identified by the provided `isbn` value as shown below:

```java
public static void update( EntityManager em, UserTransaction ut, 
    String isbn, String title, int year) throws Exception {
    ut.begin();
    Book book = em.find( Book.class, isbn);
    book.setTitle( title);
    book.setYear( year);
    ut.commit();
}
```

Now, we create the view where a `Book` can be selected so the user can edit the `title` and `year` properties then save the changes. The code for this view is stored in the `WebContent/views/books/update.xhtml` file which has the following content:

```xml
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:ui="http://java.sun.com/jsf/facelets"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:p="http://xmlns.jcp.org/jsf/passthrough">
    <ui:composition template="/WEB-INF/templates/page.xhtml">
        <ui:define name="main-content">
            <h:form id="updateBookForm">
                <h:panelGrid columns="2">
                    <h:outputText value="Select book: ">
                        <h:selectOneMenu value="#{bookCtrl.books}"
                            itemValue="#{b.isbn}" itemLabel="#{b.title}"/>
                    </h:selectOneMenu>
                    <h:outputText value="ISBN: " />
                    <h:inputText id="isbn" value="#{bookCtrl.books}">
                    </h:inputText>
                    <h:outputText value="Title: ">
                        <h:inputText id="title" value="#{bookCtrl.books}"
                            p:type="number" value="#{bookCtrl.books}"/>
                    </h:inputText>
                    <h:outputText value="Year: ">
                        <h:inputText id="year" p:type="number" value="#{bookCtrl.books}"
                            p:min="0" p:max="2023"/>
                    </h:inputText>
                </h:panelGrid>
                <h:commandButton value="Update"
            </h:form>
        </ui:define>
    </ui:composition>
</html>
```

In this view, a single selection list (select) element is created by using `<h:selectOneMenu>`. The `@value` attribute allows to save the value of selection by assigning it to a property of an object. In our
case, since the selected Book is identified by its isbn property value which is stored in the book.isbn property. Populating the list with book records is performed by using the f:selectItems element, which requires a list of records (using the @value attribute), as well as defining a variable name (using @var attribute) which is bound to the element of the list which is has the cursor in the loop used internally to populate the select list. The attributes @itemValue and @itemLabel are used to provide the informaton used to create the option HTML5 elements when the view is rendered, by specifying the content of the option element (i.e. the value of the @itemLabel attribute) and the value of the option/@value attribute (i.e. the value of @itemValue attribute).

In the update view, the user selects a book from the drop-down list, while the rest of the form has to be autocompleted with the details (isbn, title and year) of the selected book. The ISBN we already know, being the selection value of the drop-down list. However, we need to show also the title and year of the selected book. To do this, we use f:ajax JSFelement which allows to perform a request (an AJAX request) which results in calling a specified method. The method will get as parameter the managed book instance, and updates its title and year properties with the right values by reading them from the database. The used method is part of the BookController class and is named refreshObject. The code of this method is shown below:

```java
public void refreshObject( Book book) {
    Book foundBook = Book.getObjectByStdId( em, book.getIsbn());
    book.setTitle( foundBook.getTitle());
    book.setYear( foundBook.getYear());
}
```

It simply uses the Book.getObjectByStdId method which returns the Book instance stored in the database for the given isbn value. Then, the reference parameter of the method (i.e. book) is then updated so the title and year have the correct values. In the view code, we parse the reference of the book as value of the refreshObject method. The book properties are connected with the h:inputText and h:outputText (for isbn which must be read only) JSF elements, which means the form input elements will be display the values of the book properties. To enforce the refresh of the form (this is required, because the form updates automatically only on submit actions), so it displays the updated isbn, title and year values, the f:ajax JSF element allows to specify the form (to which the f:ajax is child of) elements to be updated, i.e. f:ajax/@render="isbn title year" in our case.

Last, the h:commandButton specifies that the action is to invoke the update action method of the BookController with the parameters isbn, title and year, so the the changes are made persistent.

7. Step 7 - Implement the Delete Use Case

This use case corresponds to the "Delete" from the four basic data management use cases Create-Read-Update-Delete (CRUD).

The corresponding destroy action method code from the src/pl/ctrl/BookController.java is presented below:

```java
public String destroy( String isbn) {
    try {
        Book.destroy( em, ut, isbn);
    } catch ( Exception e) {
        e.printStackTrace();
    }
    return "delete";
}
```
The main functionality of the controller action method is to invoke the `Book.destroy` method and provide the isbn of the `Book` entry to be deleted. The `Book.destroy` method takes care of finding the right entry (based on the given value of the standard identifier, i.e. `isbn`) as well as deleting it from the database.

```java
public static void destroy( EntityManager em, UserTransaction ut, String isbn)
    throws Exception, HeuristicRollbackException, RollbackException {
    ut.begin();
    Book book = em.find( Book.class, isbn);
    em.remove( book);
    ut.commit();
}
```

Finding the `Book` entry with the given `isbn` value is easy by using the built-in `find` method of the `EntityManager` object (see Section 2.4). Removing the `Book` entry from the database is performed by simply invoking the `remove` method of the `EntityManager` object (more details: Section 2.5).

The last thing we have to do is to create the associated view for the delete action. The view contains a drop-down list which allows to select the `Book` to be deleted, then a "Delete" button performs the deletion of the book. The code of the view (`WebContent/views/books/delete.xhtml`) file is shown below:

```html
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
     xmlns:ui="http://java.sun.com/jsf/facelets"
     xmlns:h="http://java.sun.com/jsf/html"
     xmlns:f="http://java.sun.com/jsf/core">
  <ui:composition template="/WEB-INF/templates/page.xhtml">
    <ui:define name="main-content">
      <h:form id="deleteBookForm">
        <h:panelGrid columns="2">
          <h:outputText value="Select book: " />
          <h:selectOneMenu value="#{book.isbn}">
            <f:selectItems value="#{bookCtrl.books}" var="b"
              itemValue="#{b.isbn}" itemLabel="#{b.title}" />
          </h:selectOneMenu>
        </h:panelGrid>
        <h:commandButton value="Delete"
                          action="#{bookCtrl.destroy( book.isbn)}"/>
      </h:form>
    </ui:define>
  </ui:composition>
</html>
```

As in the update object use case, a `h:selectOneMenu` JSF element is used to create and populate a drop-down list containing all the books, so we can choose which one we like to delete. Clicking on "Delete" `h:commandButton` results in performing the associated action which invokes the `destroy` action method of the controller with the `isbn` value of the selected book, thus resulting in the deletion of the `Book` entry from the database.

### 8. Run the App and Get the Code

You can run the minimal app [http://yew.informatik.tu-cottbus.de/tutorials/java/minimalapp/](http://yew.informatik.tu-cottbus.de/tutorials/java/minimalapp/) on our server or download the code [MinimalApp.zip](http://yew.informatik.tu-cottbus.de/tutorials/java/minimalapp/) as a ZIP archive file.

For running the minimal app on your computer, first download the code [MinimalApp.zip](http://yew.informatik.tu-cottbus.de/tutorials/java/minimalapp/) and edit the ANT script file by modifying the `server.folder` property value as described in Section 3.5. You may then have to stop your Tomcat/TomEE server with `bin/shutdown.bat` for Windows or `bin/shutdown.sh` for Linux. Now execute the following command in your console or terminal:

```
ant deploy -Dappname=minimalapp
```

Last, start your Tomcat web server (by using `bin/`
Building a Minimal Java Web App in Seven Steps

startup.bat for Windows OS or bin/startup.sh for Linux). Please be patient, this can take some time depending on the speed of your computer. It will be ready when the console displays the following info: INFO: Initializing Mojarra [some library versions and paths are shown here] for context '/minimalapp'. Finally, open your favorite browser and type: http://localhost:8080/minimalapp/faces/views/books/index.xhtml

For new projects, you may want to download the dependency libraries [lib.zip].

9. Possible Variations and Extensions

9.1. Accessibility for Web Apps

The recommended approach to providing accessibility for web apps is defined by the Accessible Rich Internet Applications (ARIA) standard. As summarized by Bryan Garaventa [http://www.linkedin.com/profile/view?id=26751364&trk=groups-post-b-author] in his article on different forms of accessibility [https://www.linkedin.com/grp/post/4512178-134539009], there are 3 main aspects of accessibility for interactive web technologies: 1) keyboard accessibility, 2) screen reader accessibility, and 3) cognitive accessibility.

Further reading on ARIA:
3. The ARIA Role Conformance Matrices [http://whatsock.com/training/matrices] by whatsock.com
5. W3C's ARIA overview page [http://www.w3.org/WAI/intro/aria.php]

9.2. Using resource URLs

Whenever an app provides public information about entities, such as the books available in a public library, it is desirable to publish this information with the help of self-descriptive resource URLs, such as http://publiclibrary.norfolk.city/books/006251587X, which would be the resource URL for retrieving information about the book "Weaving the Web" available in the public library of Norfolk.

9.3. Offline availability

It is desirable that a web app can still be used when the user is offline.

10. Points of Attention

The code of this app should be extended by

• adding some CSS styling for the user interface pages and
• adding constraint validation.
We show how to do this in the follow-up tutorial Building Java Web Apps Tutorial Part 2: Adding Constraint Validation [validation-tutorial.html].

We briefly discuss three further points of attention: boilerplate code, code clarity, using resource URLs and architectural separation of concerns.

10.1. Code clarity

*Any damn fool can write code that a computer can understand, the trick is to write code that humans can understand.* (Martin Fowler in After the Program Runs [http://martinfowler.com/distributedComputing/refactoring.pdf])

Code is often "unnecessarily complicated, convoluted, disorganized, and stitched together with disdain", as observed in a post by Santiago L. Valdarrama [https://blog.svpino.com/2015/04/22/the-thing-with-code-clarity-you-cant-be-proud-of-something-I-cant-read] who recommends using comments when necessary, only to explain things that can't be made clear enough, but rather make your code reveal its intention through the use of better names, proper structure, and correct spacing and indentation.

10.2. Boilerplate code

Another issue with the code of this Java example app is the repetitious *boilerplate code* needed

1. per model class for the storage management methods *add*, *update*, *destroy*, etc.;

2. per model class and property for getters, setters and validation checks.

While it is good to write this code a few times for learning app development, you don't want to write it again and again later when you work on real projects.

10.3. Architectural separation of concerns

One of the most fundamental principles of software architecture is *separation of concerns*. This principle is also the basis of the Model-View-Controller (MVC) architecture paradigm. It requires to keep the different functional parts of a software application independent of each other as much as possible. More specifically, it implies to keep the app's model classes independent of

1. the user interface (UI) code because it should be possible to re-use the same model classes with different UI technologies;

2. the storage management code because it should be possible to re-use the same model classes with different storage technologies.

In this tutorial, we have kept the model class *Book* independent of the UI code, since it does not contain any references to UI elements, nor does it invoke any view method. However, for simplicity, we didn't keep it independent of storage management code, since we have used JPA annotations, which bind the class to JPA object-to-storage mapping technology, and we have included the method definitions for *add*, *update*, *destroy*, etc., which invoke the storage management methods of a JPA environment's entity manager. Therefore, the separation of concerns is incomplete in our minimal example app.

We will show in a follow-up tutorial how to achieve a more complete separation of concerns.
Chapter 4. Practice Projects

1. Project 1 - Develop a Java Back-End App for Managing Movie Data

The purpose of the app is managing information about movies. The app deals with just one object type: Movie, as depicted in Figure 4.1 below. In the subsequent parts of the tutorial, you will extend this simple app by adding integrity constraints, enumeration attributes, actors and directors as further model classes, and the associations between them.

Notice that in most parts of this project you can follow, or even copy, the code of the book data management app, except that in the Movie class there is an attribute with range Date, so you have to use the HTML <time> element in the list objects table, as discussed in ???.

Figure 4.1. The object type Movie.

<table>
<thead>
<tr>
<th>Movie</th>
</tr>
</thead>
<tbody>
<tr>
<td>moviefd : Integer</td>
</tr>
<tr>
<td>title : String</td>
</tr>
<tr>
<td>releaseDate : Date</td>
</tr>
</tbody>
</table>

For developing the app, simply follow the sequence of seven steps described in the tutorial:

1. Step 1 - Set up the Folder Structure
2. Step 2 - Write the Model Code
3. Step 3 - Initialize the Application
4. Step 4 - Implement the List Objects Use Case
5. Step 5 - Implement the Create Object Use Case
6. Step 6 - Implement the Update Object Use Case
7. Step 7 - Implement the Delete Object Use Case

Make sure that

1. your HTML pages comply with the XML syntax of HTML5,
2. international characters are supported by using UTF-8 encoding for all HTML files,
3. your Java code complies with our Coding Guidelines [http://oxygen.informatik.tu-cottbus.de/webeng/Coding-Guidelines.html].
## Glossary

### E

| Extensible Markup Language | XML allows to mark up the structure of all kinds of documents, data files and messages in a machine-readable way. XML may also be human-readable, if the tag names used are self-explaining. XML is based on Unicode. SVG and MathML are based on XML, and there is an XML-based version of HTML. XML provides a syntax for expressing structured information in the form of an XML document with elements and their attributes. The specific elements and attributes used in an XML document can come from any vocabulary, such as public standards or user-defined XML formats. |

### H

| Hypertext Markup Language | HTML allows marking up (or describing) the structure of a human-readable web document or web user interface. The XML-based version of HTML, which is called "XHTML5", provides a simpler and cleaner syntax compared to traditional HTML. |
| Hypertext Transfer Protocol | HTTP is a stateless request/response protocol using human-readable text messages for the communication between web clients and web servers. The main purpose of HTTP has been to allow fetching web documents identified by URLs from a web browser, and invoking the operations of a backend web application program from a HTML form executed by a web browser. More recently, HTTP is increasingly used for providing web APIs and web services. |