Java EE 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

Mircea Diaconescu <M.Diaconescu@b-tu.de>
Gerd Wagner <G.Wagner@b-tu.de>

Java EE 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 Mircea Diaconescu and Gerd Wagner

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 [MinimalApp/index.html] from our server, or download it as a ZIP archive file [MinimalApp.zip].

Publication date 2017-06-01
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 vii
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 2
        2.3. XML namespaces 2
        2.4. Correct XML documents 3
        2.5. The evolution of HTML 3
        2.6. HTML forms 4
    3. Styling Web Documents and User Interfaces with CSS 6
    4. JavaScript - "the assembly language of the Web" 6
        4.1. JavaScript as an object-oriented language 7
        4.2. Further reading about JavaScript 7
    5. Accessibility for Web Apps 8
2. Java Summary 9
    1. Compared to JavaScript, what is different in Java? 9
    2. Java Bean Classes and Entity Classes 9
3. A Minimal Web App with Java EE 11
    1. Java Basics 12
    2. Step 1 - Set up the Folder Structure 13
    3. Step 2 - Write the Model Code 15
        3.1. Storing `Book` objects in a database table `books` 17
        3.2. Creating a new `Book` instance and storing it 17
        3.3. Retrieving all `Book` instances 18
        3.4. Updating a `Book` instance 18
        3.5. Deleting a `Book` instance 18
        3.6. Creating test data 19
        3.7. Clearing all data 19
    4. Step 3 - Configure the App 19
        4.1. Create the EntityManager and UserTransaction objects 20
        4.2. Configure the JPA database connection 20
        4.3. Create the main template 22
        4.4. Define the managed beans needed in facelets 24
        4.5. Build the WAR file and deploy it to TomEE 24
    5. Step 4 - Implement the *Create* Use Case 25
    6. Step 5 - Implement the *Retrieve/List All* Use Case 27
    7. Step 6 - Implement the *Update* Use Case 29
    8. Step 7 - Implement the *Delete* Use Case 31
    9. Style the User Interface with CSS 32
    10. Run the App and Get the Code 33
    11. Possible Variations and Extensions 33
        11.1. Using resource URLs 33
        11.2. Using an alternative DBMS 33
    12. Points of Attention 34
        12.1. Boilerplate code 34
        12.2. Offline availability 34
        12.3. Architectural separation of concerns 34
    13. Practice Project 35
        13.1. Managing information about movies 35

Glossary 37

List of Figures

3.1. The object type `Book`. 11
3.2. The object type `Movie`. 35

List of Tables

2.1. Java Visibility Level 9
3.1. Sample data about books 11
3.2. Java Visibility Level 12
3.3. Sample data 35

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 (**C**reate/**R**ead/**U**pdate/**D**elete), 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 [http://en.wikipedia.org/wiki/Tim_Berners-Lee] developed the idea and the first implementation of the WWW in 1989 at the European research institution CERN in Geneva, Switzerland. The WWW (or, simply, "the Web") is based on the Internet technologies TCP/IP (the *Internet Protocol*) and DNS (the *Domain Name System*). Initially, the Web consisted of

1. the *Hypertext Transfer Protocol (HTTP)*,
2. the *Hypertext Markup Language (HTML)*, and
3. web server programs, acting as HTTP servers, as well as web 'user agents' (such as browsers), acting as HTTP clients.

Later, further important technology components have been added to this set of basic web technologies:

- the page/document style language *Cascading Style Sheets (CSS)* in 1995,
- the web programming language *JavaScript* in 1995,
- the *Extensible Markup Language (XML)*, as the basis of web formats (like SVG and RDF/XML), in 1998,
- the XML-based *Scalable Vector Graphics (SVG)* format in 2001,
- the Resource Description Framework (RDF) for knowledge representation on the Web in 2004.

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. XML can also be used as the basis for defining a version of HTML that is called *XHTML*.

2.1. XML documents

XML provides a syntax for expressing structured information in the form of an *XML document* with nested *elements* and their *attributes*. The specific elements and attributes used in an XML document can come from any vocabulary, such as public standards or (private) user-defined XML formats. 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 `&#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. Many computational languages have some form of namespace concept, for instance, Java and PHP.

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:

```
<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 an SVG namespace declaration for an `svg` element embedded in an HTML document:

```
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
   ...
  </head>
  <body>
    <figure>
      <figcaption>Figure 1: A blue circle</figcaption>
      <svg:svg xmlns:svg="http://www.w3.org/2000/svg">
        <svg:circle cx="100" cy="100" r="50" fill="blue"/>
      </svg: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* with respect to that grammar, or, in other words, satisfy all rules of the 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. For instance, we cannot have

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

4. Attribute names are unique within the scope of an element. For instance, 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,
- 2014: ***(X)HTML 5*** in cooperation (and competition) with the WHAT working group [http://en.wikipedia.org/wiki/WHATWG] supported by browser vendors.

> 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. A few years later, in 1995, Tim Berners-Lee and Dan Connolly wrote the *HTML 2* [http://www.ietf.org/rfc/rfc1866.txt] standard, which captured the common use of HTML elements at that time. 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 messy set of elements and attributes (called "tag soup"), 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

> ***HTML = HTML5 = XHTML5***
>
> stating that when we say "HTML" or "HTML5", 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:

```
<!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>
```

Notice that in line 1, the HTML5 document type is declared, such that browsers are instructed to use the HTML5 document object model (DOM). In the `html` start tag in line 2, using the default namespace declaration attribute `xmlns`, the XHTML namespace URI `http://www.w3.org/1999/xhtml` is declared as the default namespace for making sure that browsers, and other tools, understand that all non-qualified element names like `html`, `head`, `body`, etc. are from the XHTML namespace.

Also in the `html` start tag, we set the (default) language for the text content of all elements (here to "en" standing for English) using both the `xml:lang` attribute and the HTML `lang` attribute. This attribute duplication is a small price to pay for having a hybrid document that can be processed both by HTML and by XML tools.

Finally, in line 4, using an (empty) `meta` element with a `charset` attribute, we set the HTML document's character encoding to UTF-8, which is also the default for XML documents.

2.6. HTML forms

For user-interactive web applications, the web browser needs to render a user interface (UI). The traditional metaphor for a software application's UI is that of a ***form***. The special elements for data input, data output and user actions are called ***form controls*** or UI **widgets**. In HTML, a `form` element

is a section of a web page consisting of block elements that contain form controls and *labels* on those controls.

Users complete a form by entering text into *input fields* and by selecting items from *choice controls*, including dropdown *selection lists*, *radio button groups* and *checkbox groups*. A completed form is submitted with the help of a *submit button*. When a user submits a form, it is normally 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 form 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 child element within 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).

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

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

2. *single line output fields* created with an `<output name="..." />` element,

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

4. *dropdown selection lists* created with a `select` element of the following form:

```
<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

```
<form id="Book">
  <p><label>ISBN: <output name="isbn" /></label></p>
  <p><label>Title: <input name="title" /></label></p>
  <p><label>Year: <input name="year" /></label></p>
  <p><button type="button">Save</button></p>
</form>
```

In an HTML-form-based data management user interface, 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 mapped to various kinds of **form fields**. This mapping is also called **data binding**.

In general, an attribute of a model class can always be represented in the user interface 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 widgets, 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 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 [https://medium.com/net-magazine/interview-with-h%C3%A5kon-wium-lie-f3328aeca8ed]).

CSS is based on a form of rules that consist of *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*.

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*.

We will not go deeper into CSS in this tutorial, since our focus here is on the logic and functionality of an app, and not so much on its 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".

JavaScript is a dynamic 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 version of JavaScript that is currently fully supported by web browsers is called "ECMAScript 5.1", or simply "ES5", but the next two versions, called "ES6" and "ES7" (or "ES 2015" and "ES 2016"), are already partially supported by current browsers and back-end JS environments. In fact, in May 2017, ES6 is fully supported in non-mobile browsers, except its important new `module` concept.

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++. In JavaScript, classes, unlike objects and functions, are not first-class citizens. Rather, classes have to be defined by following some code pattern in the form of special JS objects: either as *constructor* functions (possibly using the syntactic sugar of ES6 `class` declarations) 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 and enumerations (and other special datatypes).

4.2. Further reading about JavaScript

Good open access books about JavaScript are

- Speaking JavaScript [http://speakingjs.com/es5/index.html], by Dr. Axel Rauschmayer.
- Eloquent JavaScript [http://eloquentjavascript.net/], by Marijn Haverbeke.

- Building Front-End Web Apps with Plain JavaScript [http://web-engineering.info/JsFrontendApp-Book], by Gerd Wagner

5. 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:

1. How browsers interact with screen readers, and where ARIA fits in the mix [http://lnkd.in/kue-Q8] by Bryan Garaventa

2. The Accessibility Tree Training Guide [http://whatsock.com/training] by whatsock.com

3. The ARIA Role Conformance Matrices [http://whatsock.com/training/matrices] by whatsock.com

4. Mozilla's ARIA overview article [https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA]

5. W3C's ARIA overview page [http://www.w3.org/WAI/intro/aria.php]

Chapter 2. Java Summary

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. **Java is strongly typed**: Properties, parameters and variables must be declared to be of some type.
3. **No global variables, no global procedures**: In Java, variables, procedures and functions must be defined in the context of a class, which provides their name space.
4. **Arrays are static**: Arrays have a fixed size, which cannot be changed at run-time.
5. **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)
6. Classes, properties and methods are defined with a **visibility** level (`public`, `protected` or `private`), which restricts their accessibility.
7. **Type parameters**: Classes and complex data structures (such as lists) can be defined with the help of type parameters. See, for instance, the generics tutorial [https://docs.oracle.com/javase/tutorial/java/generics/types.html] by Oracle.
8. **Java programs must be compiled** before they can be executed.
9. **Speed**: Java is about twice as fast as optimized JavaScript.

In Java, visibility levels are used to define the possible levels of access:

Table 2.1. Java Visibility Level

	Class	Package	Subclass	World
public	y	y	y	y
protected	y	y	y	n
no modifier	y	y	n	n
private	y	n	n	n

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. Java Bean Classes and Entity Classes

A ***Java Bean class*** (or, simply, *bean class*) is a Java class with a parameter-free 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 Java bean class with an `@Entity` annotation, implying that a Java EE runtime environment (such as provided by the TomEE [http://tomee.apache.org/apache-tomee.html] web server) will take care of the persistent storage of its instances.

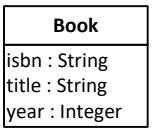
Chapter 3. Building a Minimal Web App with Java EE in Seven Steps

In this tutorial, we show how to build a simple Java EE web application using the *Java Persistence API (JPA)* for object-to-storage mapping and *Java Server Faces (JSF)* as the user interface (or "view") technology. Such an application requires a web server that supports the Java EE specifications *Java Servlets*, *Java Expression Language (EL)*, JPA and JSF, such as the open source web server Tomcat/TomEE [http://tomee.apache.org/apache-tomee.html] (pronounced "Tommy"). In addition to the Java code executed on the back-end computer that runs the TomEE web server, a Java EE web app also consists of HTML, CSS and possibly some auxiliary JavaScript code that is executed by a web browser running on the user's front-end computer. Since essentially all data processing is performed on the back-end, and the front-end only renders the user interface, we classify Java EE web applications as *back-end web apps*.

We show how to develop, deploy and run a simple example app using the **TomEE** web server, which is Apache Tomcat extended by adding basic Java EE features for providing an execution environment for light-weight Java EE web apps. We assume that you have already installed the TomEE Plume [http://tomee.apache.org/apache-tomee.html] server and MySQL [http://www.mysql.com/]. Then simply follow our installation and configuration instructions [http://web-engineering.info/JavaJpaJsfApp/Setup-Backend-for-JPA-and-JSF-Web-Applications].

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

Figure 3.1. The object type `Book`.



The following table shows a sample data population for the model class `Book`.

Table 3.1. Sample data about books

ISBN	Title	Year
006251587X	Weaving the Web	2000
0465026567	Gödel, Escher, Bach	1999
0465030793	I Am A Strange Loop	2008

What do we need for a data management app? There are four standard us e 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.
4. **Delete** 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 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 in general all we need to do. In our example application, we explain how to set up the JPA configuration for MySQL.

1. Java Basics

Compared to JavaScript, what is different in Java? We list a few observations:

1. **No program without a class**: Any Java program must include at least one class.
2. **Java is strongly typed**: Properties, parameters and variables must be declared to be of some type.
3. **No global variables, no global procedures**: In Java, variables, procedures and functions must be defined in the context of a class, which provides their name space.
4. **Arrays are static**: Arrays have a fixed size, which cannot be changed at run-time.
5. **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)
6. Classes, properties and methods are defined with a **visibility** level (`public`, `protected` or `private`), which restricts their accessibility.
7. **Type parameters**: Classes and complex data structures (such as lists) can be defined with the help of type parameters. See, for instance, the generics tutorial [https://docs.oracle.com/javase/tutorial/java/generics/types.html] by Oracle.
8. **Java programs must be compiled** before they can be executed.
9. **Speed**: Java is about twice as fast as optimized JavaScript.

In Java, visibility levels are used to define the possible levels of access:

Table 3.2. Java Visibility Level

	Class	Package	Subclass	World
public	y	y	y	y
protected	y	y	y	n
no modifier	y	y	n	n

	Class	Package	Subclass	World
private	y	n	n	n

Normally, properties are defined as private, such that they can only be assigned with a public setter, which allows to control, and possibly log, any change of the property value. Unfortunately, Java does not allow to protect only write, but not read access. Therefore, we always have to define getters along with setters, even if they are not needed.

A ***Java bean class*** (or, simply, *bean class*) is a Java class with a parameter-free 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 Java bean class with an `@Entity` annotation, implying that a Java EE 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.

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 based on SQL. JPQL query expressions look very 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 view technology. By contrast, JSF 1 has used *JavaServer Pages (JSP)* as its default view technology.

A *JSF facelet* is an XML-based template that combines static UI content items with data variables for generating an XHTML user interface page.

2. 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 is "Public Library", so it would be natural to use a corresponding name for the application folder, like "PublicLibrary", but we prefer using `MinimalApp`. 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. For your convenience, we also provide an ANT script (see our guide [http://web-engineering.info/JavaJpaJsfApp/Setup-Backend-for-JPA-and-JSF-Web-Applications] for a download link), allowing to automatically create the folder structure of the web application, compile it to a **W**eb application **Ar**chive (WAR) file and then deploy it to a TomEE server for execution. The application structure (which is compatible with the *Dynamic Web Project* structure of Eclipse, so it can be imported in Eclipse) is the following:

```
MinimalApp
  src
    pl
      m
      c
    META-INF
      persistence.xml
  WebContent
    resources
      media
```

```
      img
    views
      books
    WEB-INF
      templates
      faces-config.xml
      web.xml
```

This folder structure has the following parts:

1. The `src` folder contains the app code folder `pl`, defining the Java package name for the app (as a shortcut of 'public library'), and the folder `META-INF` with configuration files:

   a. the app code folder `pl` contains the model and controller code in its subfolders `m` and `c`, while the view/UI related code is contained in the `WebContent` folder;

   b. the most important configuration file is the `persistence.xml` file. It contains the configuration for the database connection. The content of this file is discussed in Section 4.2.

2. The `WebContent` folder contains various web resources, including template files and custom view files:

   a. the `resources` folder is used for any storing all kind of resources, such as downloadable documents, archives, but also media files, such as images, videos or sounds.

   b. the `views` folder stores our custom view files for the application, so it represents the View 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.

   c. the `WEB-INF` folder contains the extra required project libraries (a set of `jar` files, part of the `lib` subfolder, but in our case we don't need such libraries), the facelet files for the UI pages (as part of the `templates` subfolder), the `faces-config.xml` file, which stores the facelet configuration and the `web.xml` configuration file, specific to the TomEE environment server used to run our application.

We create a "Main Menu" page for our application, thus we add an `index.xhtml` file to our `views` folder with the following content:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">
<h:head>
 <meta charset="UTF-8" />
 <title>Minimal App with Java and JPA/JSF</title>
</h:head>
<body>
 <header>
  <h1>Public Library</h1>
  <h2>Minimal App with Java and JPA/JSF</h2>
 </header>
 <main>
  <menu>
   <li><h:button value="Create" outcome="create" /> a new book</li>
   <li><h:button value="Retrieve" outcome="retrieveAndListAll" />
```

```
    and list all books</li>
   <li><h:button value="Update" outcome="update" /> a book</li>
   <li><h:button value="Delete" outcome="delete" /> a book</li>
  </menu>
  <hr />
  <h:form>
   <menu>
    <li><h:commandButton value="Clear"
         action="#{bookCtrl.clearData()}" /> database</li>
    <li><h:commandButton value="Generate"
         action="#{bookCtrl.generateTestData()}" /> test data</li>
   </menu>
  </h:form>
 </main>
 <footer>
  <!-- copyright text -->
 </footer>
</body>
</html>
```

The code consists of HTML elements and JSF elements, which are discussed below.

3. Step 2 - Write the Model Code

In the second step, we create the model classes for our app, using a separate 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/m` with the following code:

```
package pl.m;

@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);
  }
  // getter and setter methods
  ...
}
```

Notice that the model class `Book` is coded as a **JPA entity class**, which is a JavaBean class enriched with the following JPA annotations:

1. The annotation `@Entity` designates a class as an entity class implying that the instances of this class will be stored persistently.

2. The annotation `@Table( name="books")` specifies the name of the database table to be used for storing the `Book` entities. 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` in our case).

3. The `@Id` annotation marks the standard identifier attribute, implying that the corresponding column of the underlying SQL database table is designated as the PRIMARY KEY. In our example, `isbn` is used as the standard identifier attribute, and the corresponding `isbn` column of the `books` table stores the primary key values.

In the entity class `Book`, we also define the following *static* (class-level) methods:

1. `Book.create` for creating a new `Book` instance.

2. `Book.retrieveAll` for retrieving all `Book` instances from the persistent data store.

3. `Book.retrieve` for retrieving a specific `Book` instance from the persistent data store by means of its standard identifier.

4. `Book.update` for updating an existing `Book` instance.

5. `Book.delete` for deleting a `Book` instance.

6. `Book.generateTestData` for creating a few example book records to be used as test data.

7. `Book.clearData` for clearing the book database table.

The signatures of these methods, which are discussed in more detail in the following subsections, are shown in the following program listing:.

```
// getter and setter methods
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;}
// CRUD data management methods
public static void create(...) {...}
public static List<Book> retrieveAll(...) {...}
public static Book retrieve(...) {...}
public static void update(...) {...}
public static void delete(...) {...}
public static void clearData(...) {...}
public static void generateTestData(...) {...}
```

The JPA architecture for data management and object-to-storage mapping is based on the concept of an *entity manager*, which provides the data management methods `persist` for saving a newly created entity, `find` for retrieving an entity, and `remove` for deleting an entity.

Since the database access operations of an entity manager are executed in the context of a **transaction**, our data management methods have 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()`.

3.1. Storing Book objects in a database table books

The instances of our entity class `Book` are Java objects representing "entities" (or *business objects*), which can be *serialized*, or, in other words, converted to *records* (or *rows*) of a database table, as shown in Table 3.1.

SQL

The *Structured Query Language* (SQL), created in the early 1970s by Donald D. Chamberlin and Raymond F. Boyce, is an ISO standard that is based on the *Relational Database* model of Edgar F. Codd and defines

1. a *Data Definition Language* (based on CREATE TABLE statements) for creating databases consisting of tables containing primitive data;
2. a *Data Manipulation Language* (based on INSERT, UPDATE and DELETE statements) for creating, updating and deleting the contents of database tables;
3. a *Query Language* (based on SELECT statements) for retrieving the information stored in database tables.

SQL is widely used in *Database Management Systems* (DBMSs) and in programming languages for persistent data storage. Despite the existence of the SQL standard, each SQL DBMS has its own dialect of SQL. Consequently, SQL code is in practice not completely portable among different DBMSs without adjustments.

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

```
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 show below how the database schema can be automatically generated by JPA. In both cases, the database setup, including a user account and the associated rights (create, update, etc), must be done manually before the JPA application can connect to it.

3.2. Creating a new Book instance and storing it

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

```
public static void create( 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();
```

```
}
```

To store the new object, the `persist` method of the given 'entity manager' is invoked. It is responsible for creating the corresponding SQL `INSERT` statement and executing it.

3.3. Retrieving all Book instances

The instances of an entity class, such as `Book`, are retrieved from the database with the help of a corresponding query expressed in the *Java Persistence Query Language* (JPQL [http://en.wikipedia.org/wiki/Java_Persistence_Query_Language]). These 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.

In the `Book.retrieveAll` 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`:

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

3.4. Updating a Book instance

To update an existing `Book` instance, first we need to retrieve it from the database, by using `em.find`, and then set those attributes the value of which has changed:

```
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 (here: `Book.class`), 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. Saving is automatically managed by the JPA runtime environment when we complete the transaction with `ut.commit()`.

3.5. Deleting a Book instance

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

```
public static void delete( EntityManager em,
    UserTransaction ut, String isbn) throws Exception {
```

```
  ut.begin();
  Book book = em.find( Book.class, isbn);
  em.remove( 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()`.

3.6. 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:

```
public static void generateTestData( 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();
}
```

After clearing the database, we successively create 3 instances of the `Book` entity class and save them with the help of `persist`.

3.7. Clearing all data

The following procedure clears our database by deleting all rows:

```
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`.

4. Step 3 - Configure the App

In this section we show how to

1. configure an app to connect with a database in a controller class,

2. obtain the `EntityManager` and `UserTransaction` instances required for performing database operations,

3. wrap an app in a WAR file and deploy it to a web server for execution.

4.1. Create the EntityManager and UserTransaction objects

A controller class contains the code that glues the views to the model, as well as all methods that 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.c.BookController` in the `src/pl/c` folder.

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

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

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

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

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

A closer look at this code shows that it is sufficient to use the `@PersistenceContext` annotation and provide a `unitName` (see 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.

4.2. Configure the JPA database connection

In the previous section, discussing the `BookController` class, we have shown how to obtain the `EntityManager` and `UserTransaction` objects required for performing 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 version="1.0" encoding="UTF-8"?>
<persistence version="2.1"
  xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <persistence-unit name="MinimalApp">
  <class>pl.m.Book</class>
  <properties>
   <!-- Request auto-generation of the database schema -->
   <property
     name="javax.persistence.schema-generation.database.action"
     value="create"/>
   <!-- Use the JPA annotations to create the database schema -->
   <property
     name="javax.persistence.schema-generation.create-source"
     value="metadata"/>
  </properties>
  <jta-data-source>jdbc/MinimalApp</jta-data-source>
 </persistence-unit>
</persistence>
```

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.

The `persistence-unit` element has three content parts:

1. One or more `class` elements, each one containing the fully qualified name of an entity class of the app (like `pl.m.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 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 version="1.0" encoding="UTF-8"?>
```

```
<TomEE>
  <Resource id="MinimalApp" type="DataSource">
    JdbcDriver com.mysql.jdbc.Driver
    JdbcUrl jdbc:mysql://localhost:3306/MinimalApp
    UserName MinimalApp
    Password MinimalApp
    JtaManaged true
  </Resource>
</TomEE>
```

The `Resource` element contains the information required to connect with the database (user name, password, access URL and the Java class name of the connection driver). Notice that in order to use a connection driver, one needs to download the jar file for that specific technology and copy it into the `lib` subfolder of the TomEE installation folder. Finally, TomEE needs to be restarted in order to load the new driver. See also our instructions page [http://web-engineering.info/JavaJpaJsfApp/Setup-Backend-for-JPA-and-JSF-Web-Applications], for additional details related to JDBC drivers, and in particular how to use the official MySQL JDBC driver "Connector/J".

4.3. 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 `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:

```
<!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>
  <meta charset="UTF-8" />
  <title>Public Library</title>
 </h:head>
 <body>
  <header>
   <ui:insert name="headerTitle"/>
   <ui:insert name="headerMenu">
    <ui:include src="/WEB-INF/templates/header.xhtml"/>
   </ui:insert>
  </header>
  <main>
   <ui:insert name="main"/>
  </main>
  <footer>
   <ui:insert name="footer">
    <ui:include src="/WEB-INF/templates/footer.xhtml"/>
   </ui:insert>
```

```
  </footer>
 </body>
</html>
```

In the code, one can see that some HTML elements are used (e.g., `title`, `header`, `main` and `footer`) while others like `h:head` and `ui:insert` are not HTML elements, but have been defined by JSF. For instance, JSF defines its own head element `h:head` for injecting special HTML code such as `script` elements that load JavaScript code for HTTP messaging between the web browser and the back-end Java program by invoking methods of the JavaScript API *XML HTTP Request (XHR)* .

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

The `header.xhtml` sub-template contains the application menu, corresponding to the CRUD operations:

```
<nav xmlns:h="http://java.sun.com/jsf/html">
  <ul>
    <li><h:link outcome="create">C</h:link></li>
    <li><h:link outcome="retrieveAndListAll">R</h:link></li>
    <li><h:link outcome="update">U</h:link></li>
    <li><h:link outcome="delete">D</h:link></li>
  </ul>
</nav>
```

The `footer.xhtml` sub-template contains the "Back to main menu" link, which we like to show on each application page, excepting the index page:

```
<h:link outcome="index" xmlns:h="http://java.sun.com/jsf/html">
  Back to main menu
</h:link>
```

Notice that the `h` namespace must be defined once again in `footer.xhtml` even if it is used in the `page.xhtml` template file, where was already defined. In the footer template component, we define the "Back to main menu" operation, which is going to be injected into each page that uses this template. The *main* region is dynamic, and will be replaced with the content generated by a facelet template.

JSF is using the following namespaces:

- `xmlns:ui="http://java.sun.com/jsf/facelets"` for the *JSF Facelets Tag Library* providing templating elements (like `ui:insert` for specifying the region of a template where to inject the facelet content).

- `xmlns:h="http://java.sun.com/jsf/html"` for the *JSF HTML Tag Library* providing JSF versions of HTML elements, which are then mapped to HTML elements. For example `h:inputText`, which is mapped to an HTML `input` element.

- `xmlns:f="http://java.sun.com/jsf/core"` for the *JSF Core Tag Library* providing 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"` for using HTML attributes in JSF HTML elements and passing them through to the generated HTML. For example, with `p:type`

in, `<h:inputText p:type="number">` an HTML5 `input` type attribute can be created in the generated HTML: `<input type="number">`.

- `xmlns:c="http://java.sun.com/jsp/jstl/core"` for the *JSTL Core Tag Library* providing all kinds 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 then be used in the view code for conditional expressions.
- `xmlns:fn="http://java.sun.com/jsp/jstl/functions"` for the *JSTL Functions Tag Library* providing various utility functions, such as string converters. For example, we can use `fn:toUpperCase` to convert a string to its uppercase representation.

4.4. Define the managed beans needed in facelets

JavaBean classes, including entity classes, can be used for creating 'managed beans' with the help of the `@ManagedBean` annotation, which allows defining the name of a variable for accessing the created bean in the view code, typically in an EL expression. In our example app, we want to access a `Book` bean as well as a `BookController` bean, therefore both classes have to be annotated as follows:

```
@Entity @Table( name="books")
@RequestScoped @ManagedBean( name="book")
public class Book { ... }

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

Notice how a lifetime scope can be specified for a managed bean with a scope annotation. 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 in our example we only need these two.

4.5. Build the WAR file and deploy it to TomEE

There are multiple ways on which we could compile the source code, create the war file and deploy it to TomEE server. For example one can use Eclipse IDE [http://www.eclipse.org/downloads/packages/] or Netbeans [https://netbeans.org/], but as well you can use ANT and create a script which takes care of all these tasks.

For your convenience, we wrote an ANT [https://ant.apache.org/] script that allows to automatically create the structure of a Java web application, compile the Java source code, build the WAR file and deploy it to a TomEE web server. Our ANT script generates an `Eclipse IDE` compatible folder structure, so in case you want to use Eclipse, simply create a project from the existing application code. The ANT installation instructions, detailed description of the available tasks as well as download links are available on our instructions page [http://web-engineering.info/JavaJpaJsfApp/Setup-Backend-for-JPA-and-JSF-Web-Applications].

For our "PublicLibrary" example application, we have used the ANT script and the following steps were needed:

- create the application folder structure: `ant create app -Dappname=publicLibrary -Dpkgname=pl`.
- compile the Java code and build the war file: `ant war -Dappname=publicLibrary`.

- deploy the application to TomEE server: `ant deploy -Dappname=publicLibrary`. This operation also builds the war file, if it was not already created using the `war` task, as shown before.

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 -Dpgkname=hw -Dappname="Hellow World"`. In this case it was required to provide the `pkgname` parameter, while "Hello World" is not a valid Java identifier, since it contains a white space.

5. Step 4 - Implement the *Create* Use Case

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

The corresponding `create` action method code from the `src/pl/c/BookController.java` is shown below:

```
public class BookController {
  ...
  public String create( String isbn, String title, int year) {
    try {
      Book.create( em, ut, isbn, title, year);
      // clear the form after saving the Book record
      FacesContext fContext = FacesContext.getCurrentInstance();
      fContext.getExternalContext().getRequestMap().remove("book");
    } catch ( Exception e) {
      e.printStackTrace();
    }
    return "create";
  }
}
```

The `BookController::create` action method invokes the `Book.create` model class method for creating and saving a `Book` instance. It returns the name of the view file found in the same folder as the view that triggered the action. This file (`create.xhtml` in our case) will be displayed after executing the action. Using the `FacesContext` object, the form is cleared after creating a `Book` instance. The code of the `create` method in `src/pl/m/Book.java` is the following:

```
public class Book {
  ...
  public static void create( 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 facelet template for the view of the *Create* use case, `WebContent/views/books/create.xhtml`. Such a facelet template essentially defines an HTML form with *data binding* and *action binding*.

Data binding refers to the binding of model class properties to form (input or output) fields. For instance, in the following facelet code fragment, the entity property `book.isbn` is bound to the form input field "isbn":

```
<h:outputLabel for="isbn" value="ISBN: ">
  <h:inputText id="isbn" value="#{book.isbn}" />
</h:outputLabel>
```

In JSF, for the `inputText` elements of a form, the `id` attribute is used with a given value, e.g., `id="isbn"`. The rendered HTML5 `input` elements have both, the `id` and the `name` attributes, and their values are obtained by using the form id and element id values separated by a colon, i.e., `id="createBookForm:isbn"` and `name="createBookForm:isbn"`.

We can also see a first example of an expression in the Java EE 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. The value of the expression is assigned to the `value` attribute of the generated HTML `input` element. The example in our JSF code above is the expression `#{book.isbn}`, which retrieves the value of the `isbn` property of the `book` bean.

Action binding refers to the binding of method invocation expressions to actionable UI elements, where the invoked methods typically are controller action methods, and the actionable UI elements typically are form buttons. For instance, in the following facelet code fragment, the method invocation expression `bookCtrl.create(...)` is bound to the form's submit button:

```
<h:commandButton value="Create"
    action="#{bookCtrl.create( book.isbn, book.title, book.year)}"/>
```

After discussing data binding and action binding, it's time to look at the complete code of the facelet:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="..." xmlns:h="..." xmlns:p="...">
 <ui:composition template="/WEB-INF/templates/page.xhtml">
  <ui:define name="headerTitle">
   <h1>Create a new book record</h1>
  </ui:define>
  <ui:define name="main">
   <h:form id="createBookForm">
    <div><h:outputLabel for="isbn" value="ISBN: ">
      <h:inputText id="isbn" value="#{book.isbn}" />
     </h:outputLabel></div>
    <div><h:outputLabel for="title" value="Title: ">
      <h:inputText id="title" value="#{book.title}" />
     </h:outputLabel></div>
    <div><h:outputLabel for="year" value="Year: ">
      <h:inputText id="year" p:type="number"
                   value="#{book.year}" />
     </h:outputLabel></div>
    <div>
     <h:commandButton value="Save"
         action="#{bookCtrl.create(
             book.isbn, book.title, book.year)}" />
```

```
    </div>
   </h:form>
  </ui:define>
 </ui:composition>
</html>
```

This facelet replaces the `main` region of the template defined in `page.xhtml`, because the `name` attribute of the `ui:define` element has been set to "main", but it also replace the `headerTitle` region of the template, which is part of the `header` and displays the current operations allowed by the page.

`h:outputLabel` elements can be used for creating form field labels, while `h:inputText` elements are used for creating HTML input elements. It is possible to specify a HTML5 type of an `input` element by using a special namespace prefix (`xmlns:p = "http://xmlns.jcp.org/jsf/passthrough"`) for the `type` attribute, enforcing it to be 'passed through'. In this way the `year` input field can be defined with type `number`, so it's rendered by the corresponding number widget in the browser.

The `h:commandButton` element allows creating submit buttons rendered as a `input` elements with `type="submit"`, and binding them to an action to be performed when the button is clicked. The value of the `action` attribute is a method invocation expression. In our *Create* use case we want that, when the button is clicked, a `Book` instance with the property values provided by corresponding form fields is created and saved.

6. Step 5 - Implement the *Retrieve/List All* Use Case

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

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

The `getBooks` method returns a list of `Book` instances which are obtained by calling the static `retrieveAll` method of the `Book` model class. The code of the `Book.retrieveAll` method is:

```
public class Book {
  ...
  public static List<Book> retrieveAll( 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 3.3, it uses a JPQL statement to retrieve the `Book` records from the `books` table and create the corresponding `Book` instances. The `EntityManager` object required for being able to perform the JPQL query is passed to `Book.retrieveAll` from the `BookController` object as discussed in Section 4.1 section.

Now it is the time to define the facelet template for displaying a table with all book records found in the database. The view template files corresponding to the model classes of our app are located in model class subfolders of the `WebContent/views/` folder. For the *Retrieve/list all books* use case, we create a file named `retrieveAndListAll.xhtml` in `WebContent/views/books/` with the following content:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:ui="..." xmlns:h="..." xmlns:f="...">
 <ui:composition template="/WEB-INF/templates/page.xhtml">
  <ui:define name="headerTitle">
   <h1>Retrieve and list all book records</h1>
  </ui:define>
  <ui:define name="main">
   <h:dataTable value="#{bookCtrl.books}" var="b">
    <h:column>
     <f:facet name="header">ISBN</f:facet>
      #{b.isbn}
    </h:column>
    <h:column>
     <f:facet name="header">Title</f:facet>
      #{b.title}
    </h:column>
    <h:column>
     <f:facet name="header">Year</f:facet>
      #{b.year}
    </h:column>
   </h:dataTable>
  </ui:define>
 </ui:composition>
</html>
```

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

The `h:dataTable` element defines a table view for a set of records, which is then rendered as an HTML table. Its `value` attribute defines a data binding to a record collection, while the `var` attribute defines a variable name for iteratively accessing records from this collection. The expression provided in the `value` attribute normally specifies a collection-valued property (here: `books`) which is accessed via a corresponding getter (here: `getBooks`) as defined in the controller class `BookController`. In this particular case it is just sufficient to define the `getBooks` method since there is no need of a `books` property in the controller class. In any case, `value` does not allow to invoke a method, so we cannot call `getBooks` directly. Instead we have to use the (possibly virtual) property `books`, which internally evaluates to a call of `getBooks` without checking if a `books` property really exists.

The `h:button` element allows to create redirect buttons. The value of the `outcome` attribute specifies a name of a facelet file (omitting the `.xhtml` extension, such that, in the example, the file name is `index.xhtml`).

7. Step 6 - Implement the *Update* Use Case

Like for *Create*, for the *Update* use case also a controller action method is defined in the `BookController` class:

```
public class BookController {
  ...
  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 saving property value changes for a book object identified by its `isbn` value as shown below:

```
public class Book {
  ...
  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 != null && !title.equals( book.title)) {
      book.setTitle( title);
    }
    if (year != book.year) {
      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, and then save the changes. The code for this view is stored in the `WebContent/views/books/update.xhtml` file which has the following content:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:ui="..." xmlns:h="..." xmlns:f="..." xmlns:p="...">
 <ui:composition template="/WEB-INF/templates/page.xhtml">
  <ui:define name="headerTitle">
   <h1>Update a book record</h1>
  </ui:define>
  <ui:define name="main">
   <h:form id="updateBookForm">
```

```
    <div><h:outputLabel for="selectBook" value="Select book: ">
      <h:selectOneMenu id="selectBook" value="#{book.isbn}">
       <f:selectItem itemValue="" itemLabel="---"/>
       <f:selectItems value="#{bookCtrl.books}" var="b"
          itemValue="#{b.isbn}" itemLabel="#{b.title}"/>
       <f:ajax listener="#{bookCtrl.refreshObject( book)}"
          render="isbn title year" />
      </h:selectOneMenu>
     </h:outputLabel></div>
    <div><h:outputLabel for="isbn" value="ISBN: ">
      <h:outputText id="isbn" value="#{book.isbn}"/>
     </h:outputLabel></div>
    <div><h:outputLabel for="title" value="Title: ">
      <h:inputText id="title" value="#{book.title}"/>
     </h:outputLabel></div>
    <div><h:outputLabel for="year" value="Year: ">
      <h:inputText id="year" p:type="number"
                   value="#{book.year}"/>
     </h:outputLabel></div>
    <div><h:commandButton value="Save Changes"
            action="#{bookCtrl.update(
              book.isbn, book.title, book.year)}"/>
    </div>
   </h:form>
  </ui:define>
 </ui:composition>
</html>
```

In this facelet template, a single selection list (that is, a single-`select` HTML element) is created with the help of the JSF element `h:selectOneMenu`, where the selection list items (the HTML `option` elements) are defined by the JSF elements `f:selectItem` or `f:selectItems`. The `value` attribute of `h:selectOneMenu` binds `book.isbn` to the value of the selected item (or `option` element). The selection list is populated with book records with the help of a `f:selectItems` element bound to `bookCtrl.books`. The attributes `itemLabel` and `itemValue` define the `option` elements' text and value.

In the update view, when the user selects a book from the selection list, the form fields are filled with the (ISBN, title and year) property values of the selected book. While the ISBN is immediately available in the view (on the front-end) as the value of the selected `option` element, the values of the `title` and `year` properties have to be fetched from the back-end database. This can be done with the help of the JSFelement `f:ajax`, which sends an HTTP request message for invoking a remote method, `bookCtrl.refreshObject`, on the back-end using XHR. This method takes the managed `book` bean, and updates its `title` and `year` properties with the current values retrieved from the database. Its code is the following:

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

```
}
```

To enforce a refresh of the HTML form after the user's selection, such that it displays the values of `isbn`, `title` and `year`, the `f:ajax` element allows specifying form fields to be updated with the `render` attribute like, in our case, `render="isbn title year"`.

Finally, the `h:commandButton` element is used for invoking the `update` action method of the `BookController` with the parameters `isbn`, `title` and `year`, for making the changes persistent.

8. Step 7 - Implement the *Delete* Use Case

For the *Delete* use case, the `delete` action method of the `BookController` invokes the `Book.delete` method by providing the ISBN of the `Book` object to be deleted:

```
public class BookController {
  ...
  public String delete( String isbn) {
    try {
      Book.delete( em, ut, isbn);
    } catch ( Exception e) {
      e.printStackTrace();
    }
    return "delete";
  }
  ...
}
```

The `Book.delete` method first retrieves the `Book` object to be deleted, and then invokes the entity manager's `remove` method on it:

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

The view for the *Delete* action provides a selection list for selecting the book to be deleted. A "Delete" button allows performing the deletion of the selected book. The code of the view in `WebContent/views/books/delete.xhtml` is as follows:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:ui="..." xmlns:h="..." xmlns:f="...">
 <ui:composition template="/WEB-INF/templates/page.xhtml">
  <ui:define name="headerTitle">
   <h1>Delete a book record</h1>
```

```
  </ui:define>
  <ui:define name="main">
   <h:form id="deleteBookForm">
    <div><h:outputLabel for="selectBook" value="Select book: ">
      <h:selectOneMenu id="selectBook" value="#{book.isbn}">
       <f:selectItems value="#{bookCtrl.books}" var="b"
          itemValue="#{b.isbn}" itemLabel="#{b.title}"/>
      </h:selectOneMenu>
     </h:outputLabel></div>
    <div><h:commandButton value="Delete"
            action="#{bookCtrl.delete( book.isbn)}"/>
    </div>
   </h:form>
  </ui:define>
 </ui:composition>
</html>
```

As in the *Update* use case, a `h:selectOneMenu` element is used to create and populate a selection list containing all the books to choose from. Clicking on the "Delete" command button results in invoking the `delete` action method of the controller with the `isbn` value of the selected book, thus resulting in the deletion of the `Book` object from the database.

9. Style the User Interface with CSS

We style the UI with the help of the HTML5 Boilerplate [https://html5boilerplate.com/] CSS and the browser style normalization file normalize.css [http://necolas.github.io/normalize.css/]. The two CSS files `main.css` and `normalize.min.css` are located in the `WebContent/resources/css` folder.

Adding the CSS to the HTML pages requires to edit the `WebContent/WEB-INF/templates/page.xhtml` file and add the `style` elements referencing the CSS files:

```
<h:head>
  <title>Public Library</title>
  <link rel="stylesheet" type="text/css"
    href="#{facesContext.externalContext.requestContextPath}/
      resources/css/normalize.min.css" />
  <link rel="stylesheet" type="text/css"
    href="#{facesContext.externalContext.requestContextPath}/
      resources/css/main.css" />
</h:head>
```

Notice that we also need to add the CSS to the `WebContent/views/books/index.xhtml` file, which is not generated with the `page.xhtml` template. The `facesContext.externalContext.requestContextPath` expression allows to get the path to `WebContent` folder. We then need to append the rest of the path, including the CSS file name, for each of the CSS files.

With JSF, there are two main ways to apply CSS styles to specific elements. For the elements created with JSF, we need to use the `@styleClass` attribute, e.g.:

```
<h:form id="createBookForm" styleClass="clearfix">
  ...
</hform>
```

For the HTML elements, we simply use the CSS classes, as in any normal HTML document, e.g.:

```
<div class="wrapper clearfix">
  ...
</div>
```

For creating advanced user interfaces, additional block elements that act as containers are used for defining paddings, margins, various background colors and other styles. Thus, in various HTML pages you may find DOM structures as shown below, where the `div` element containing the `@class` attribute is used as container for styling purposes:

```
<header>
  <div class="wrapper clearfix">
    <div class="title">
      <ui:insert name="headerTitle"/>
    </div>
    <ui:insert name="headerMenu">
      <ui:include src="/WEB-INF/templates/header.xhtml" />
    </ui:insert>
  </div>
</header>
```

10. Run the App and Get the Code

You can run the minimal app [http://web-engineering.info/tech/JavaJpaJsf/MinimalApp/] (or run the minimal app with CSS [http://web-engineering.info/tech/JavaJpaJsf/MinimalAppWithCSS/]) on our server or download the code [http://web-engineering.info/tech/JavaJpaJsf/MinimalApp.zip] as a ZIP archive file.

Follow our instructions [http://web-engineering.info/JavaJpaJsfApp/Setup-Backend-for-JPA-and-JSF-Web-Applications] to get your environment prepared for running back-end Java EE web apps with JPA and JSF.

11. Possible Variations and Extensions

11.1. 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. However, resource URLs are not supported by JSF. In the Java world, we would have to use JAX-RS, instead of JSF, for programming a web API with resource URLs. But this would imply that we need to take care of the front-end UI in a different way, since JAX-RS is a pure back-end API, not providing any support for building user interfaces. A natural option would be to use a JavaScript front-end framework, such as BackboneJS or ReactJS, for rendering the UI.

11.2. Using an alternative DBMS

Instead of *MySQL*, one can use various other database management systems for persistent storage. The following four steps are required to specify the used DBMS (only one DBMS at a time is possible):

- Configure TomEE so it uses the corresponding resource for your application. For a list of resource configuration examples used for common DBMS's, check Common DataSource Configurations [http://tomee.apache.org/common-datasource-configurations.html]. For example, if *PostgreSQL* was chosen as DBMS, then edit the `conf/tomee.xml` file and add the following code:

```
<Resource id="application-persistence-unit-name" type="DataSource">
  JdbcDriver   org.postgresql.Driver
  JdbcUrl  jdbc:postgresql://host/database-name
  UserName     dbms-username
  Password     user-password
</Resource>
```

- Copy the *jar* file corresponding to the DBMS driver implementation to the `lib` folder. After this, the TomEE server needs to be restarted.
- Install the DBMS, if not already installed. Installation instructions are usually available on the corresponding DBMS web page.
- Create the corresponding DBMS user and database to be used for your application.

12. Points of Attention

The code of this app should be extended by 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 issues.

12.1. Boilerplate code

Another issue with the code of this Java example app is the repetitious *boilerplate code* needed per entity class for the storage management methods `create`, `retrieve`, `update` and `delete`, and their wrappers in the corresponding controller classes. While it may be instructive 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.

12.2. Offline availability

It is desirable that a web app can still be used when the user is offline. However, this is not possible with a back-end web application architecture as implied by Java EE. Offline availability can only be obtained with a truly distributed architecture where essential application components can be executed both on the back-end and on the front-end.

12.3. Architectural separation of concerns

From an architectural point of view, it is important 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.

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. First, due to using JPA annotations, the model class is bound to the JPA object-to-storage mapping technology. Second, we have included the method definitions for `create`, `update`, `delete`, etc., which invoke the storage management methods of a JPA environment's entity manager, in the model class. Therefore, the separation of concerns is incomplete in our minimal app.

13. Practice Project

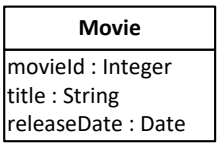
If you have any questions about how to carry out the following projects, you can ask them on our discussion forum [http://web-engineering.info/forum/JavaJpaJsfApp].

13.1. Managing information about movies

The purpose of the app to be developed is managing information about movies. The app deals with just one object type: `Movie`, as depicted in Figure 3.2 below. In the subsequent parts of the tutorial, you will extend this simple app by adding integrity constraints, enumerations, further model classes for actors and directors, 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 where an attribute with range `Date` is used, so you have to use the HTML `<time>` element in the *list objects* table.

Figure 3.2. The object type `Movie`.



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

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 *Retrieve/List All* Use Case
5. Step 5 - Implement the *Create* Use Case
6. Step 6 - Implement the *Update* Use Case
7. Step 7 - Implement the *Delete* Use Case

You can use the following sample data for testing:

Table 3.3. Sample data

Movie ID	Title	Release date
1	Pulp Fiction	1994-05-12

Movie ID	Title	Release date
2	Star Wars	1977-05-25
3	Casablanca	1943-01-23
4	The Godfather	1972-03-15

Make sure that international characters are supported by using UTF-8 encoding for all HTML files.

If you have any questions about how to carry out the following projects, you can ask them on our discussion forum [http://web-engineering.info/forum/JavaJpaJsfApp/].

Glossary

C

CRUD

CRUD is an acronym for ***C**reate*, ***R**ead/Retrieve*, ***U**pdate*, ***D**elete*, which denote the four basic data management operations to be performed by any software application.

Cascading Style Sheets

CSS is used for defining the presentation style of web pages by telling the browser how to render their HTML (or XML) contents: 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 element.

D

Document Object Model

The DOM is an abstract API for retrieving and modifying nodes and elements of HTML or XML documents. All web programming languages have DOM bindings that realize the DOM.

Domain Name System

The DNS translates user-friendly domain names to IP addresses that allow to locate a host computer on the Internet.

E

ECMAScript

A standard for JavaScript defined by the industry organization "Ecma International".

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 based on the Internet technologies TCP/IP and DNS, 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 back-end web application program from an HTML form executed by a web browser. More recently, HTTP is increasingly used for providing web APIs and web services.

I

IANA

IANA stands for *Internet Assigned Numbers Authority*, which is a subsidiary of ICANN responsible for names and numbers used by Internet protocols.

ICANN

ICANN stands for *Internet Corporation of Assigned Names and Numbers*, which is an international nonprofit organization that maintains the domain name system.

IndexedDB

A JavaScript API for indexed data storage managed by browsers. Indexing allows high-performance searching. Like many SQL DBMS, IndexedDB supports database transactions.

I18N

A set of best practices that help to adapt products to any target language and culture. It deals with multiple character sets, units of measure, keyboard layouts, time and date formats, and text directions.

J

JSON

JSON stands for *JavaScript Object Notation*, which is a data-interchange format following the JavaScript syntax for object literals. Many programming languages support JSON as a light-weight alternative to XML.

M

MathML

An open standard for representing mathematical expressions, either in data interchange or for rendering them within webpages.

MIME

A MIME type (also called "media type" or "content type") is a keyword string sent along with a file for indicating its content type. For example, a sound file might be labeled `audio/ogg`, or an image file `image/png`.

***M**odel-**V**iew-**C**ontroller*

MVC is a general architecture metaphor emphasizing the principle of separation of concerns, mainly between the model and the view, and considering the model as the most fundamental part of an app. In MVC frameworks, "M", "V" and "C" are defined in different ways. Often the term “model” refers to the app’s data sources, while the “view” denotes the app’s code for the user interface, which is based on CSS-styled HTML forms and DOM events, and the “controller” typically denotes the (glue) code that is in charge of mediating between the *view* and the *model*.

O

Object Constraint Language — The OCL is a formal logic language for expressing integrity constraints, mainly in UML class models. It also allows defining derivation expressions for defining derived properties, and defining preconditions and postconditions for operations, in a class model..

Object-Oriented Programming — OOP is a programming paradigm based on the concepts of *objects* and *classes* instantiated by objects. Classes are like blueprints for objects: they define their *properties* and the *methods*/*functions* that can be applied to them. A higher-level characteristic of OOP is *inheritance* in class hierarchies: a subclass inherits the features (properties, methods and constraints) of its superclass.

Web Ontology Language — OWL is formal logic language for knowledge representation on the Web. It allows defining vocabularies (mainly classes with properties) and supports expressing many types of integrity constraints on them. OWL is the basis for performing automated inferences, such as checking the consistency of an OWL vocabulary. Vocabularies, or data models, defined in the form of UML class models can be converted to OWL vocabularies and then checked for consistency.

P

Portable Network Graphics — PNG is an open (non-proprietary) graphics file format that supports lossless data compression.

Polyfill — A polyfill is a piece of JavaScript code for emulating a standard JavaScript method in a browser, which does not support the method.

R

Resource Description Framework — RDF is a W3C language for representing machine-readable propositional information on the Web.

S

Standard Generalized Markup Language — SGML is an ISO specification for defining markup languages. HTML4 has been defined with SGML. XML is a simplified successor of SGML. HTML5 is no longer SGML-based and has its own parsing rules.

Scalable Vector Graphics — SVG is a 2D vector image format based on XML. SVG can be styled with CSS and made interactive using JavaScript. HTML5 allows direct embedding of SVG content in an HTML document.

Slot — A slot is a name-value pair. In an object of an object-oriented program (for instance, in a Java object), a slot normally is a property-value pair. But in a JavaScript object, a slot may also consist of a method name and a method body or it may be a key-value pair of a map.

U

Unicode	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. Unicode includes legacy character sets like ASCII and ISO-8859-1 (Latin-1) as subsets. XML is based on Unicode. Consequently, the Greek letter π (with code 960) can be inserted in an XML document as π using the XML entity syntax. The default encoding of Unicode characters in an XML document is UTF-8, which uses only a single byte for ASCII characters, but three bytes for less common characters.
Uniform Resource Identifier	A URI is either a *Uniform Resource Locator (URL)* or a *Uniform Resource Name (URN)*.
Uniform Resource Locator	A URL is a resource name that contains a web address for locating the resource on the Web.
Unified Modeling Language	The UML is an industry standard that defines a set of modeling languages for making various kinds of models and diagrams in support of object-oriented problem analysis and software design. Its core languages are *Class Diagrams* for information/data modeling, and *Sequence Diagrams*, *Activity Diagrams* and *State Diagrams* (or *State Charts*) for process/behavior modeling..
Uniform Resource Name	A URN refers to a resource without specifying its location.
User Agent	A user agent is a front-end web client program such as a web browser.

W

WebM	WebM is an open (royalty-free) web video format supported by Google Chrome and Mozilla Firefox, but not by Microsoft Internet Explorer and Apple Safari.
Web Hypertext Application Technology Working Group	The *WHATWG* was established in 2004 by former employees of Apple, Mozilla, and Opera who have been unhappy with the slow progress of web technology standardization due to W3C's choice to focus on the standardization of XHTML2. Led by Ian Hickson, they developed HTML5 and related JavaScript APIs in competition and collaboration with the W3C.
World Wide Web	The WWW (or, simply, "the Web") is a huge client-server network based on HTTP, HTML and XML, where web browsers (and other 'user agents'), acting as HTTP clients, access web server programs, acting as HTTP servers.
World Wide Web Consortium	The *W3C* is an international organization in charge of developing and maintaining web standards.

X

XML HTTP Request

The XML HTTP Request (XHR) API allows a JavaScript program to exchange HTTP messages with back-end programs. It can be used for retrieveing/submitting information from/to a back-end program without submitting HTML forms. XHR-based approaches have been subsumed under the acronym "AJAX" in the past.