Developing Secure Traditional Web Applications – Part 1/3

Prof. Dr. Marc Rennhard, Dr. Stephan Neuhaus
Institut für angewandte Informationstechnologie InIT
ZHAW School of Engineering
rema | neut @zhaw.ch

There are several valuable resources for the content of this chapter:

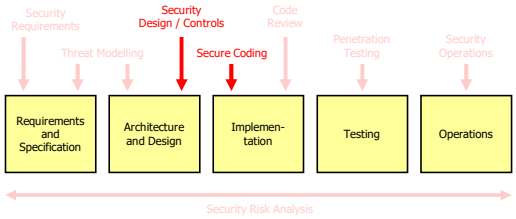
- **Jakarta EE API Specification**: *https://jakarta.ee/specifications/platform/9/apidocs*
- **Jakarta EE documentation**: A very thorough introduction to Jakarta EE can be found here: *https://eclipse-ee4j.github.io/glassfish/documentation*

Content

- Introduction to Jakarta EE based traditional web applications
- The Marketplace application – which serves as an example throughout the entire chapter
- Various security mechanisms to secure web applications, covering:
  - Information leakage prevention
  - Data Sanitation (XSS prevention)
  - Secure database access (SQL injection prevention)
  - Access control and authentication
  - Session handling
  - CSRF and forced browsing prevention
  - Input validation

Goals

- You understand what must be considered in general with respect to security when developing traditional web applications
- You understand the difference between declarative and programmatic security and can apply these concepts correctly in your applications
- You understand the different concepts (technologies and methods) that can be used to secure traditional web applications and can apply them appropriately to secure your own applications
- You can develop secure traditional web applications using the Jakarta EE technology
- Security activities covered in this chapter:



Introduction

Developing Secure Traditional Web Applications (1)

- This chapter is about developing secure traditional web applications
  - With «developing», we mean here choosing appropriate security controls and implementing them correctly – which is why this chapter covers the security activities Security Design / Controls and Secure Coding
- With «traditional» web applications, we mean monolithic web applications where most of the code runs server-side and where the server mainly serves HTML documents to the client
  - This is in contrast to more modern Single Page Web Applications that use REST based microservices and lots of JavaScript in the browser
- While traditional web applications have some drawbacks (e.g., user experience, flexibility, scalability,...), it's still important that you understand how to develop them securely, because:
  - The world is full of such applications and they won't go away quickly
  - There are still plenty of new web applications developed that follow the traditional architecture for various reasons:
    - If the UI is relatively simple, traditional web applications work very well
    - Well understood technology, lots of know-how in companies
    - Good frameworks and development tools, easier with respect to security,...
  - If you understand how to secure traditional web applications, you have a good basis to secure modern single page applications (see next chapter!)

Developing Secure Traditional Web Applications (2)

- As the example technology to develop secure traditional web applications, we are using Jakarta EE (Enterprise Edition)
  - Note: Originally, this was Java EE, but Java EE was moved from Oracle to the Eclipse Foundation in 2018, where it will be developed further under the name Jakarta EE
- We could also use other technologies (ASP.NET, Ruby, Python/Django, PHP, Node.js,…), but there are several arguments for using Jakarta EE:
  - It's an established, very mature and widely used technology
  - It provides several good built-in security mechanisms and therefore, it's well suited to develop secure web applications
  - You are already familiar with Java SE
    - You don't need Jakarta EE knowledge as a prerequisite, as the basics will be introduced in this chapter
  - Jakarta EE is not only well suited to develop traditional web applications, but also to develop REST based microservices that are used in the context of modern single page applications
    - In fact, we will reuse a significant portion of the code in the next chapter

Developing Secure Traditional Web Applications (3)

- Note however that Jakarta EE is «just the example technology» we are using here
  - The primary goal of the chapter is to learn in general what must be considered when developing secure web applications
  - This means that once you have learned to secure a Jakarta EE web application, you should be able to transfer what you've learned to other technologies and programming languages

- Another important goal is that you learn to use the security mechanisms offered by a modern framework in the correct way, which means:
  - You must know the security mechanisms that are automatically enabled and understand their benefits and limitations
  - You must understand the additional security mechanisms provided by the framework, but which must actively be used by the developer
  - This will prevent that you reinvent the wheel and will significantly increase the probability that you get a secure application

Jakarta EE as the Example Technology

It's important to realize that this chapter is mainly about learning the **concepts** how web applications can be secured. These concepts are always more or less the same, no matter what underlying technology you are using. This implies that once you have learned these concepts in this chapter - using Jakarta EE as the example technology, then you shouldn't have any problems to transfer this knowledge to any other technology that you are using to develop web applications.

What differs of course are the specific details how the concepts are then implemented in different technologies. Some technologies provide very little «out of the box» to support implementing secure applications, which means you have to do a lot on your own (or use security libraries, if they are available) and this obviously increases the probability that security vulnerabilities creep in. Other technologies (Jakarta EE is certainly one of them) provide a mature basis to implement and/or configure security mechanisms, which increases the probability you end up with a reasonably secure application. But again: The concepts to secure the application are basically always the same, independent of the technology, and you therefore must always study and understand the technology you are using to make the right decisions about how to design and implement specific security controls.

Jakarta Enterprise Edition

- Java EE was first released in 1999, current version 9 (Jakarta EE 9)
  - Jakarta EE is based on the Java Standard Edition (Java SE)
- Jakarta EE provides several components to develop traditional and modern web applications and we will use the following here:
  - Jakarta Server Faces (JSF) together with the Unified Expression Language (EL) for the view of the web application
  - CDI Beans and Enterprise Jakarta Beans (EJB) for the model and business logic
  - Jakarta Persistence API (JPA) for database access
  - Jakarta EE Security API for authentication
  - Bean Validation framework for input validation
  - Jakarta API for RESTful Web Services (JAX-RS) for REST based web services

Will be used from the beginning

Will be introduced later in this chapter

Will be used in the next chapter

- To run a Jakarta EE application, an application server is required
  - We are using Payara, which is based on the open source GlassFish server

CDI Beans

CDI stands for «Context and Dependency Injection».

GlassFish / Payara

GlassFish is the open source Jakarta EE reference implementation: *https://javaee.github.io/glassfish*

Payara is based on GlassFish: *https://www.payara.fish*

Declarative and Programmatic Security

- Most modern web frameworks / technologies – including Jakarta EE – allow the developer to use declarative or programmatic security (or both)
- Declarative Security
  - Declarative security is based on security mechanisms that are built-in and provided by the framework / technology
  - To use them, they have to be configured, which is done either outside of the code or in metadata in the code (but no «real» code is needed)
  - With Jakarta EE applications, this is done either in the deployment descriptor (*web.xml*) or with annotations
  - The configurations are enforced during runtime by the application server
  - Relatively easy to configure in a secure way, but allows only relatively coarse-grained security and is limited to what it offers
- Programmatic Security
  - Security functions are directly programmed in the code
  - More complex, more error-prone, but also more flexible
- Best practice
  - If possible, use declarative security whenever possible and supplement it with programmatic security when needed

Declarative and Programmatic Security

Most modern web frameworks / technologies allow the developer to use declarative or programmatic security (or both). In general, declarative security is based on security mechanisms that are built-in and provided by the framework and that have «just to be used the right way». While not trivial, this is in general the more secure approach as the probability of making security-relevant errors is significantly smaller compared to programmatic security, where the security mechanisms are implemented directly in the source code by the developer. The downside of declarative security is that it is (obviously) limited to the security mechanisms that are offered by the technology. This means that if you want to use security controls that are not provided by the underlying technology, you'll still have to implement them on your own using programmatic security.

In practice, one should always use declarative security mechanisms offered by the used technology as a basis (as this increases the probability that you end up with a secure application) and supplement this with programmatic security when needed. This is also what we are going to do in this chapter with Jakarta EE.

The Marketplace Application

Marketplace V01

This subchapter uses the first version of the Marketplace application: Marketplace_v01.

The Marketplace Application

- To discuss the various security concepts, an example application is used throughout the entire chapter: The Marketplace Application
- It's a relatively simple Jakarta EE web application based on Jakarta Server Faces (JSF) and CDI beans, but it's well-suited to demonstrate many security problems and fitting solutions
- In this subchapter, the technical details and parts of the source code of the first version of the Marketplace application (V01) will be explained to get an overview of the functionality
  - This also serves as an introduction to Jakarta EE technology (especially of JSF and CDI beans)
- In the following subchapters, the Marketplace application will be extended with additional functionality, creating each time a new version (V02, V03,...)
- In the next chapter, we will add a REST based web service API as a basis for a modern single page application

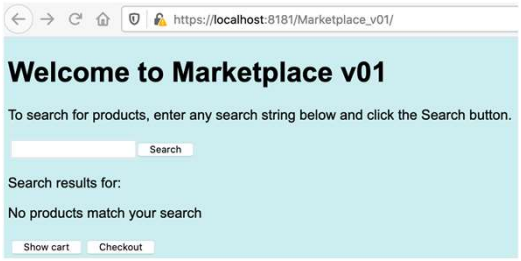
Running the Marketplace Application

Overall, 10 different versions of the Marketplace application will be used in this chapter, starting with Marketplace V01 and ending with Marketplace V10. The title slide of each subchapter contains (in the notes section) the specific version that is used in the subchapter. For instance, this subchapter uses the first version, Marketplace V01, as listed in the notes section of the previous slide.

To run any of the Marketplace versions in this chapter on the Ubuntu image that is used for the lab exercises, do the following (of course, you can also do this on your own system, provided you have the necessary software packages installed (IDE, Java, Payara, MySQL,...):

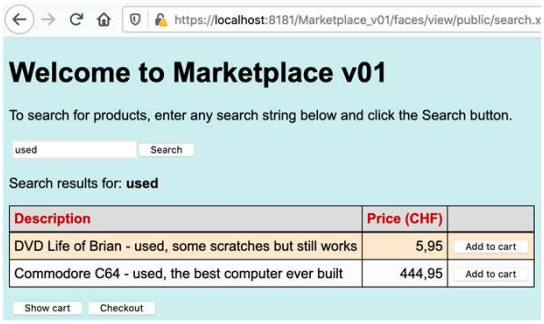
- Download *Marketplace.zip* and *Marketplace.sql* from OLAT.
- Move the files to an appropriate location (e.g., into a directory *securitylabs* in the home directory */home/user*).
- To create the database schema and the technical user used by the Marketplace application to access the database, do the following (this can be repeated at any time to reset the database):
  - Open *MySQL Workbench*.
  - Click on the left on *Local Instance 3306* and enter root as password.
  - Choose *Open SQL Script...* in the menu File and select the downloaded file *Marketplace.sql*.
  - Click the Execute icon.
- Unzip *Marketplace.zip*. This creates a directory *Marketplace*, which contains directories *Marketplace_v01*, *Marketplace_v02* etc. that correspond to the different *Marketplace* versions that are used throughout this chapter. Each directory contains a Jakarta EE project based on Maven and should be importable in any modern IDE. The remainder assumes you are using NetBeans, which is installed on the Ubuntu image.
- Start NetBeans and open the project that corresponds to the Marketplace version you want to run.
- In the following, it is assumed you want to run the first version, i.e., *Marketplace_v01*. To build the project, right-click *Marketplace_v01* in the Projects tab and select Clean and Build.
- To run *Marketplace_v01*, right-click *Marketplace_v01* in the *Projects* tab and select *Run*. If the application server (Payara) is not yet running, it will be started, which takes some time.
- Under the *Services* tab and expanding *Servers*, the Payara server is listed. Sometimes, it may be necessary to restart Payara, do this with a right-click and Restart. Under *Applications*, you can also undeploy applications if necessary.
- The application is reached with *https://localhost:8181/Marketplace_v01*.

Marketplace – Functionality (1)

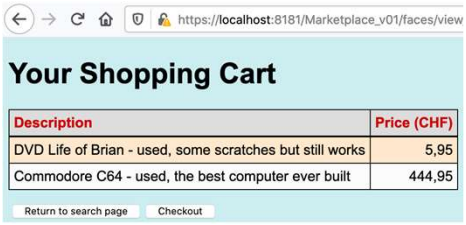


- The search page allows to search for products
  - Clicking *Search* lists all products matching the search criteria (if any)

- The search results are displayed on the same page
  - The entered search string (if any) is also displayed
  - Clicking *Add to cart* inserts a product into the shopping cart

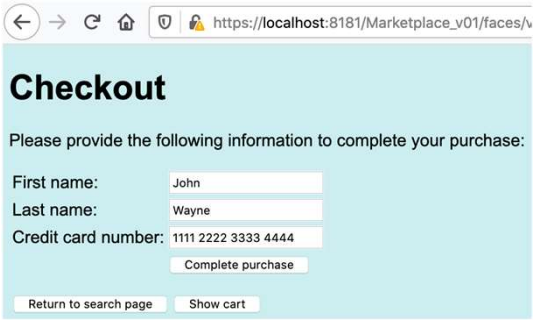


Marketplace – Functionality (2)



- The cart page shows the products that have been put into the car
  - Clicking *Checkout* results in being forwarded to the checkout page

- The checkout page requires the user to enter the name and credit card information
  - Clicking *Complete purchase* completes the purchase



Marketplace – Data Model (1)

- MySQL is used as the database
- The database schema is called *marketplace* and currently consist of two tables (more tables will be introduced later)
- Table *Product* contains the offered products

ProductID	Code	Description	Price	Username
1	0001	DVD Life of Brian - used, some scratches but sti…	5.95	donald
2	0002	Ferrari F50 - red, 43000 km, no accidents	250000.00	luke
3	0003	Commodore C64 - used, the best computer eve…	444.95	luke
4	0004	Printed Software-Security script - brand new	10.95	donald

- Table *Purchase* contains an entry for each completed purchase

PurchaseID	Firstname	Lastname	CreditCardNumber	TotalPrice
1	Ferrari	Driver	1111 2222 3333 4444	250000.00
2	C64	Freak	1234 5678 9012 3456	444.95
3	Script	Lover	5555 6666 7777 8888	10.95
4	John	Wayne	1111 2222 3333 4444	250005.95

Product Table

Column *Username* contains the user (e.g., a product manager) who maintains the product. For now, this is not important, but this will be used later.

Marketplace – Data Model (2)

- There's a database user *marketplace*, which is the technical user used by the application to access the database
  - This user has only the necessary rights needed by the application (principle of least privilege)

```
CREATE USER 'marketplace'@'localhost' IDENTIFIED BY 'marketplace';

GRANT SELECT         ON `marketplace`.* TO 'marketplace'@'localhost';
GRANT INSERT, DELETE ON `marketplace`.`Product` TO 'marketplace'@'localhost';
GRANT INSERT, DELETE ON `marketplace`.`Purchase` TO 'marketplace'@'localhost';
```

  - Some of these rights (e.g., INSERT and DELETE on table *Product*) are not required yet, but will be needed later when the application is extended

Minimal Rights

Using a database user with minimal rights can limit damage in case of an SQL injection, as the attacker cannot execute any commands (e.g., DROP, ALTER…) that have not been granted to the database user used by the application.

And obviously, the password «marketplace» is a poor choice, but is fine for demonstration purposes...

Jakarta Server Faces Fundamentals

- When writing a JSF application, one mainly develops the following:
  - Facelets: XHTML files that correspond to the web pages
  - Backing Beans: Java classes that store the state of a Facelet (web page) and provide functionality for the Facelets (business logic, model)
- Facelets can use the backing beans to read and write their attributes (instance variables) and to call methods provided by them
  - Obviously, Facelets cannot directly be delivered to the browser, as the browser does not understand their interactions with backing beans
  - So there must be a way to create HTML documents from Facelets / backing beans and to map received HTTP requests to backing bean attributes / methods
- This is handled by the Faces Servlet, which is provided by Jakarta EE:
  - The Faces Servlet creates HTML documents based on the Facelets and the current state of the backing bean(s) that are used by the Facelets
  - If a Facelet writes backing bean attributes and/or calls backing bean methods, the Faces Servlet maps this to HTML forms in the generated HTML document
  - When receiving a corresponding POST request, the Faces Servlet updates the backing bean attributes (based on parameters) and calls the appropriate method

Backing Beans and further Classes

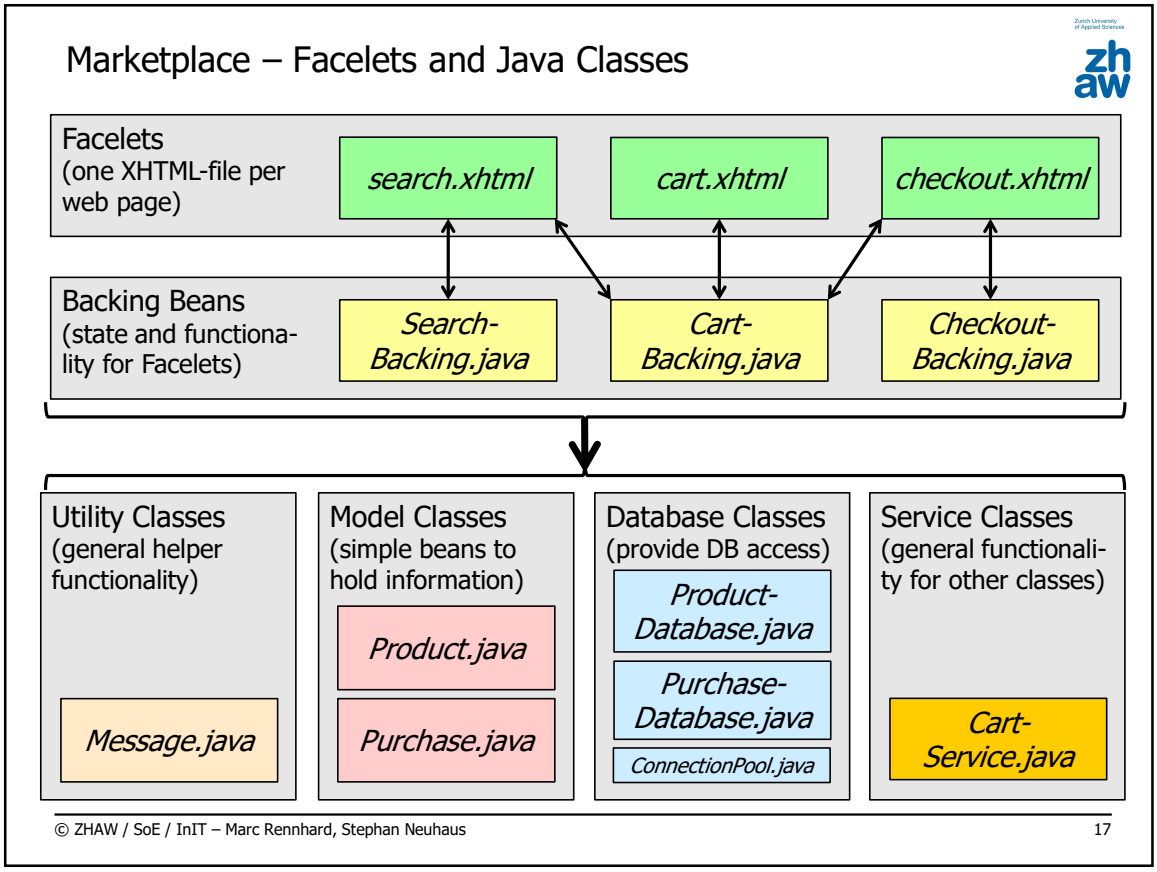
Usually, one does not put the entire model / business logic code into the backing beans. Instead, additional classes are used for better modularization and re-use of the code. But for the Facelets, this is transparent as they are only communicating with the backing beans and «don't care» how the backing beans use further classes or not.

Faces Servlet

This servlet is the central gatekeeper of a JSF application. It's provided by Jakarta EE and «just works», so you don't have to actively take care of it. «Central gatekeeper» means that it receives the requests from the browser and creates the responses for the browser.

The main task of the Faces Servlet is to translate Facelets and backing beans (which are used on the server side of the application) to HTML pages (that can be served to the browser) and to perform the right actions in the backing beans (by setting attributes and calling methods) when receiving a subsequent request from the browser. Specifically, this means the following:

- When creating an HTML page from a Facelet, the Faces Servlet includes state information from the backing beans(s) into the generated HTML page (if the Facelet contains references to such state information).
- In addition, when creating an HTML page from a Facelet, the Faces Servlet maps write accesses of backing bean attributes and calls of backing bean methods to HTML forms in the HTML page (if the Facelet includes such write accesses or method calls). These forms and the resulting POST requests that are sent when the user submits such a form are identified in a way (using a hidden field, see later in this chapter when the ViewState is discussed) so that the Faces Servlet can unambiguously associate a POST request it receives with the corresponding call of a backing bean method (that was defined in the Facelet).
- When receiving such a POST request, the Faces Servlet first uses the request parameters (if any) to update the attributes (i.e., the state) of the backing bean(s) and then calls the corresponding backing bean method.



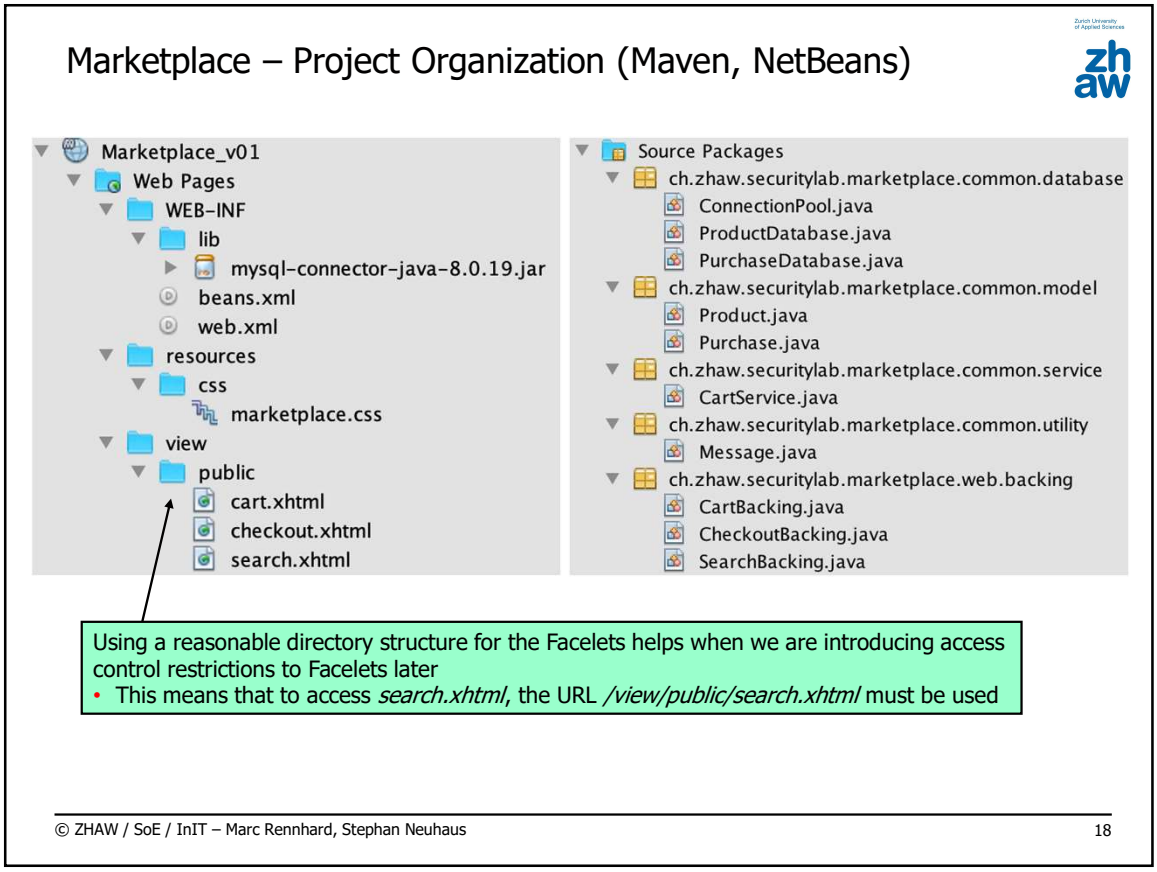
Facelets and Java Classes

The following Facelets and Java Classes exist in the base version of the Marketplace application:

- Three Facelets that correspond to the search, cart and checkout pages.
- A backing bean class per Facelet to store the state of the Facelets and to provide further functionality (methods) for them.
- A service class *CartService.java* to hold the content of the shopping cart.
- Three database classes. *ProductDatabase.java* and *PurchaseDatabase.java* provide methods to access the tables *Product* and *Purchase*. *ConnectionPool.java* provides access to the underlying connection pool (maintained by the application server, see later).
- Two model classes *Product.java* and *Purchase.java* (simple JavaBeans) to hold information about a product or a purchase.
- A utility class *Message.java* to assist message passing between components (e.g., from a backing bean to a Facelet to display a success or error message).

Facelets and Backing Beans

Often, there's a 1:1 relationship between a Facelet and a backing bean, in the sense that a Facelet frequently uses exactly one backing bean. However, that's not always the case and a Facelet can basically use any number of backing beans. In the Marketplace application, for example, the Facelet *search.xhtml* uses backing beans *SearchBacking.java* and *CartBacking.java*.



Maven & NetBeans

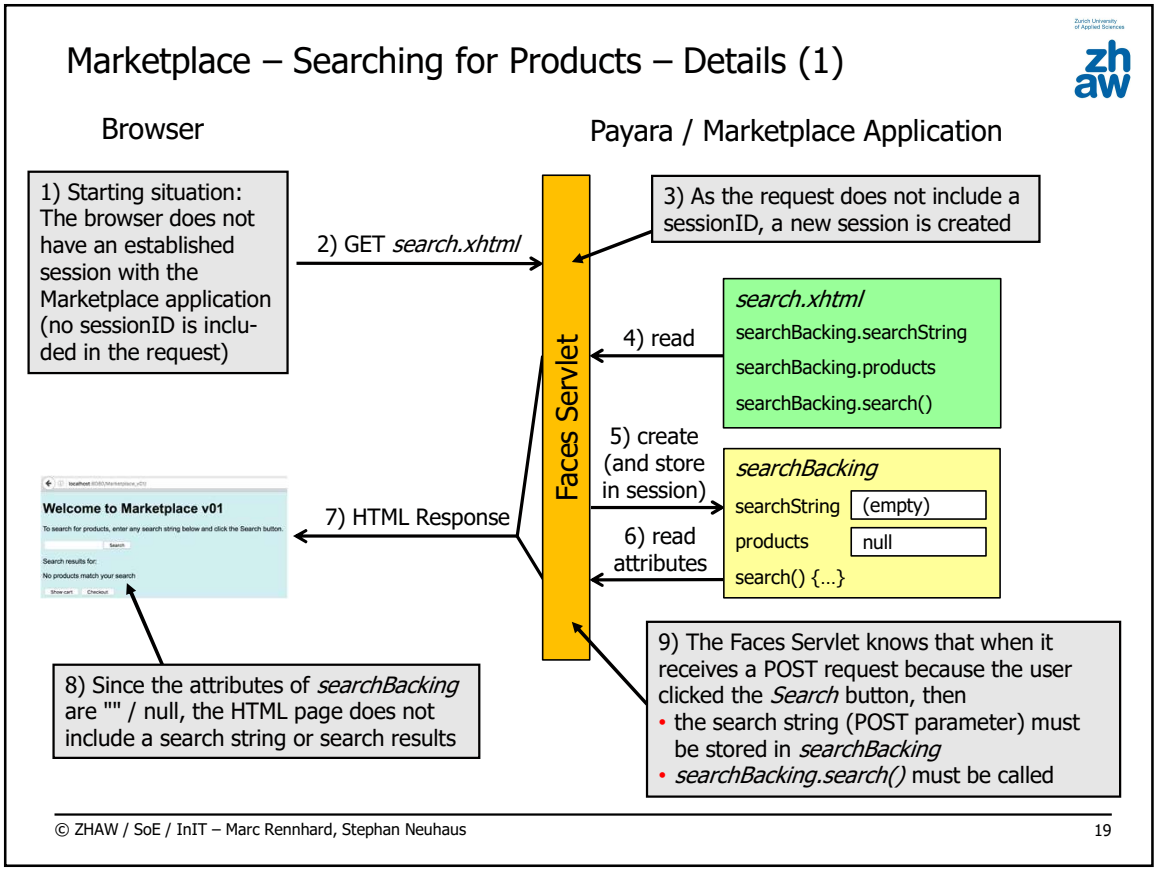
The slide shows the project organization of the Marketplace application as displayed when using NetBeans and when the project is a Maven project. In other IDEs, this looks differently, but of course, the same files should be shown.

web.xml

web.xml contains core configuration data for the entire web application (details see later).

Database Connector

To connect to the database, the database connector jar file (which is basically the JDBC database driver, here*: mysql-connector-java-8.x.y.jar*) must be added to the *lib* directory below *WEB-INF* in the Maven project. It could also be installed directly in the Payara server, but by including it in the Maven project, no special configuration in Payara is needed.

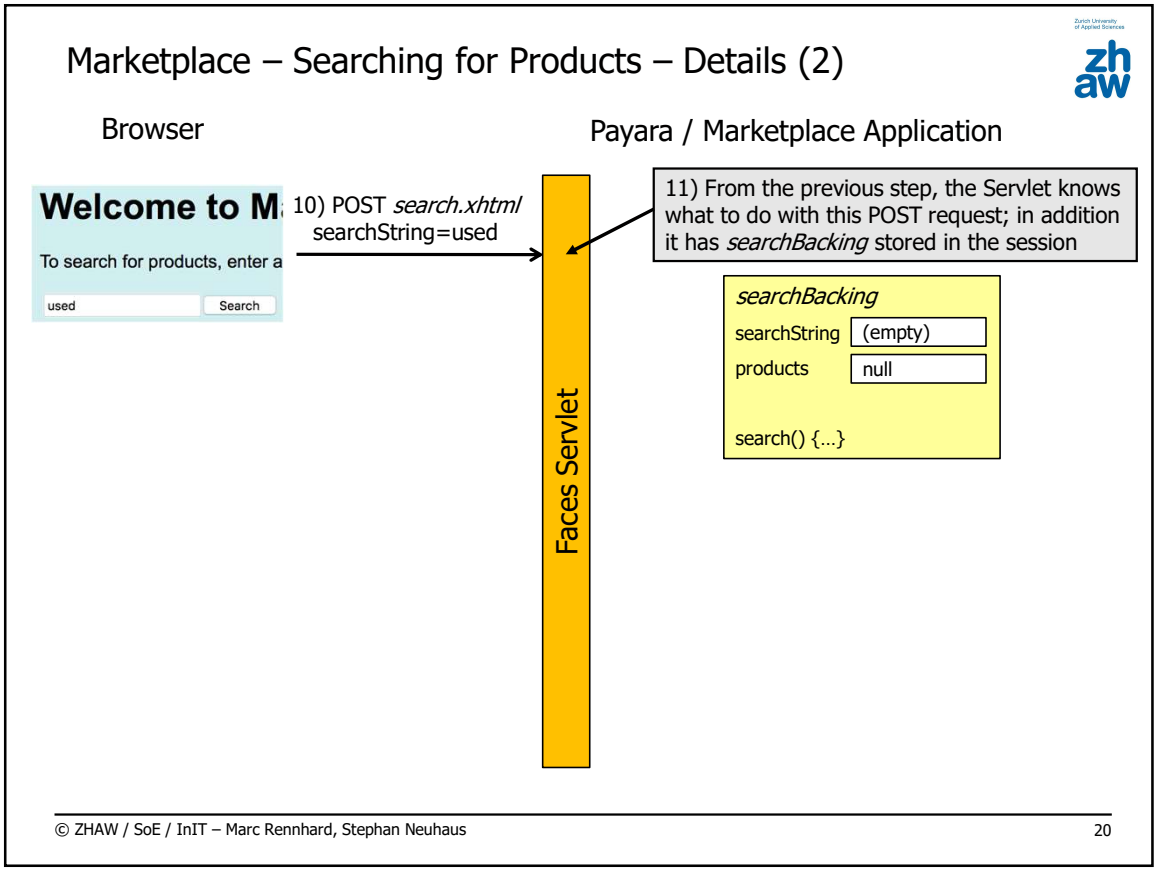


Searching for Products

This slide and the following two slides show in detail what happens when the user is searching for products. In particular, it shows how Facelets and backing beans are «working together» within the web application, how HTML pages are created based on Facelets and backing beans, how POST requests done by the browser result in state changes and method calls in the backing beans within the web application, and how the Faces Servlet is the central component that orchestrates all of this. Other functions provided by the Marketplace application, e.g., putting a product into the shopping cart or checking out work basically in the same way, which means that the process described here is representative for all functions provided by the application.

This slide shows the steps when the browser is started, has no valid session, and the browser requests the entry page (the search page). Below, additional information to the steps 1-9 illustrated above are given.

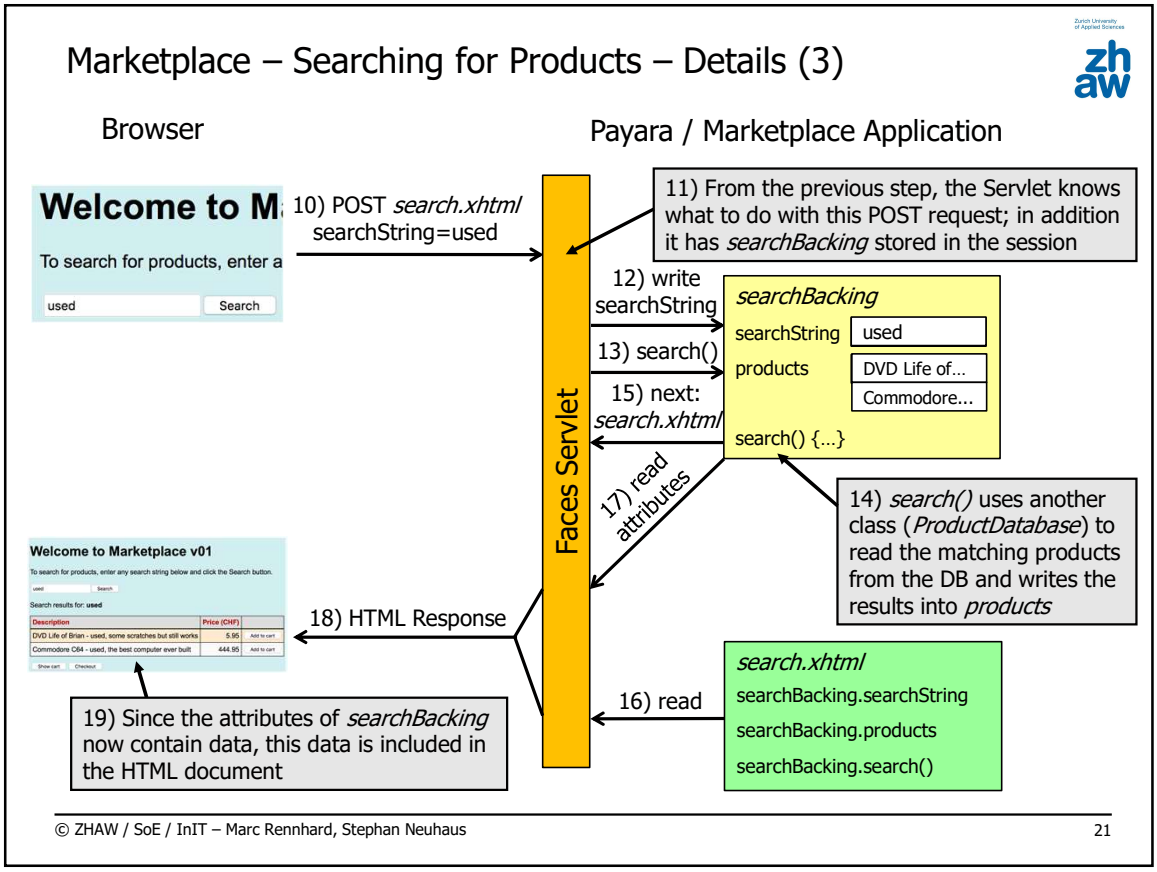
- Step 4: As the request targets *search.xhtml*, the Faces Servlet reads this file (which is a Facelet).
- Steps 4/5: By inspecting *search.xhtml*, the Faces Servlet knows that *search.xhtml* uses a backing bean of type *SearchBacking*, as the Facelet code includes references to the attributes *searchString* and *products* of *SearchBacking* and as it uses its *search* method (this can be seen in the code in a few slides). As this is a new session that does not contain a *SearchBacking* object yet, the Servlet creates a new *SearchBacking* bean and stores it in the session of the user. The two attributes of the *SearchBacking* bean are used to store the search string entered by the user (attribute *searchString*) and to store the products that correspond to the search results (attribute *products*). Currently, these attributes contain the empty string and null as values, as no search has been conducted so far.
- Steps 6/7: When creating the HTML document from *search.xhtml*, the Faces Servlet reads the attributes from *SearchBacking* so they can be included in the HTML document. As they currently contain the empty string and null, the resulting HTML page neither contains a search string nor search results.
- Step 7: The response also contains a Set-Cookie HTTP header to set the session ID in the browser. This means the session ID will be included in subsequent requests by the browser, which allows the web application to associate them with the newly created session.
- Step 9: The Faces Servlet knows this because the search input field of *search.xhtml* is linked to attribute *searchString* in *searchBacking* and because the *Search* button in *search.xhtml* is linked to the *search* method (as can be seen in the code in a few slides).



Searching for Products

After having received the search page (previous slide), this slide shows the first steps that happen when the user enters a search string and submits the search by clicking the *Search* button. Below, additional information to the steps 10-11 illustrated above are given.

- Step 11: The *searchBacking* bean loaded from the session is in the same state as when it was stored before, so *searchString* is empty and products is null.



Searching for Products

This slide shows the remaining steps that happen when the user enters a search string and submits the search by clicking the *Search* button. Below, additional information to the steps 12-19 illustrated above are given.

- Step 12: The search string that was entered by the user and that was included as a parameter in the POST request is written into the *SearchBacking* bean.
- Steps 13/14: The Faces Servlet knows that when receiving this POST request and after the search string has been updated in the *SearchBacking* bean, the *search()* method must be called. As a result of this, attribute *products* now contains the search results (i.e., a list of matching products).
- Step 15: The *search()* method returns a string which defines the Facelet to be delivered to the browser, which in this case is *search.xhtml*.
- Steps 16/17/18: When creating the HTML document from *search.xhtml*, the Faces Servlet reads the attributes from *SearchBacking* so they can be included in the HTML document. As the attributes of the *SearchBacking* bean now contain meaningful data, both the search string and the search results are included in the HTML page.

Marketplace Code – Bean *Product.java* (1)

- To temporarily store data in the application (e.g., a product or a purchase), the Marketplace application uses simple JavaBeans (aka beans)
  - «Simple» to separate them from CDI beans or EJBs, which are managed by the application server
- Beans must follow a well-defined format:
  - Private instance variables to hold the data
  - A public constructor without any arguments to create a bean (standard constructor)
  - Public getter and setter methods to get and set the values of the instance variables
  - The naming must be done according to the following convention:
    - If the instance variable is named *creditCard*…
    - …the methods must be named *getCreditCard* and *setCreditCard*
- The main reason for storing data in beans is that this allows Facelets to easily access this data with, e.g., *.creditCard*
  - I.e., the data can be accessed by using the name of the instance variable

Marketplace Code

The following slides contain parts of the code of the Marketplace application. Basically, these slides include all the code that is involved when searching for products and when putting a product into the shopping cart. The other components of the application are implemented in a similar way, which means that the code described on the following slides is representative for the entire source code of the application.

JavaBeans

JavaBeans must contain the following:

- Instance variable to hold the data (state) of the bean
- A public standard constructor
- Public getter and setter methods for the instance variables
- Serializable – but this is often omitted for simple beans. If a bean is serializable, this allows easily storing beans, e.g., in the database or in a file, or sending beans across the network (by serializing it).

Marketplace Code – Bean *Product.java* (2)

```
public class Product {
    private String code;
    private String description;
    private double price;
    private String username;

    public String getCode() {
        return code;
    }

    public void setCode(String code) {
        this.code = code;
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    public double getPrice() {
        return price;
    }
```

Private instance variables that hold the content of the bean

Public getter and setter methods for all attributes

Note: Standard constructor is not implemented, as the default constructor does exactly what we want

Default Constructor

When not providing any constructor, Java always implicitly includes a default constructor with no arguments that does nothing, e.g., with the *Product* class above:

```
public void Product() {
}
```

Marketplace Code – Bean *Product.java* (3)

```
    public void setPrice(double price) {
        this.price = price;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }
        if (!(obj instanceof Product)) {
            return false;
        }
        Product other = (Product) obj;
        return code.equals(other.code);
    }
}
```

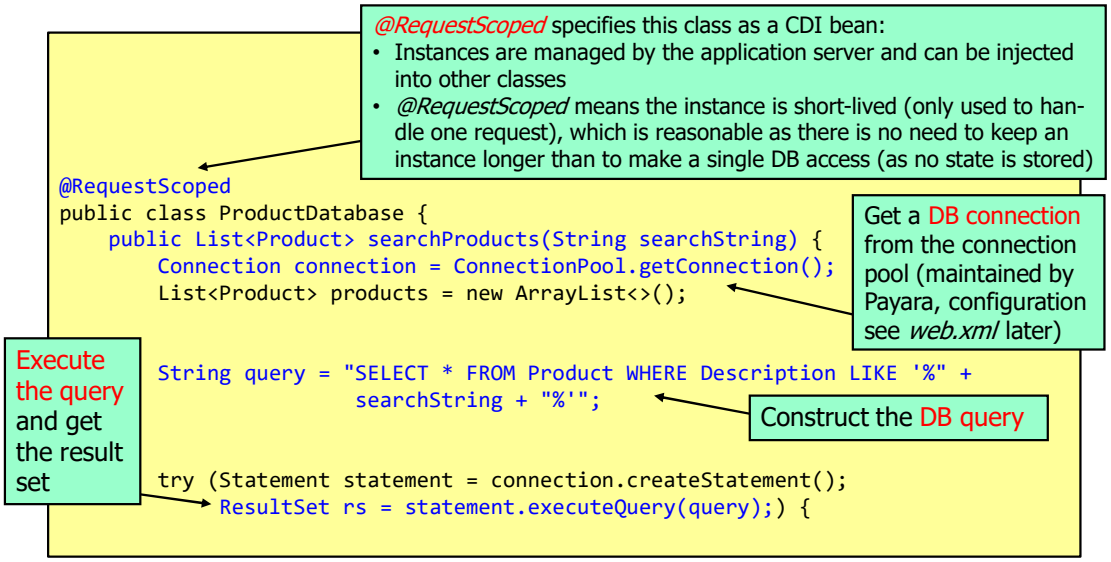
Public getter and setter methods for all attributes

equals() method to compare two products based on their code

- Needed to check whether a product is already in the shopping cart

Marketplace Code – *ProductDatabase.java* (1)

- *ProductDatabase* provides a method to search for products
  - It uses the bean *Product* as a container to return the results



Import Statements

Due to space restrictions, we left out the import statements above:

```
import ch.zhaw.securitylab.marketplace.common.model.Product;
import java.sql.*;
import java.util.ArrayList;
import java.util.List;
import jakarta.enterprise.context.RequestScoped;
```

@RequestScoped

This annotation turns this class into a CDI bean, which means instances of it are maintained by the application server. This is one of the strengths of Jakarta EE and relieves the developer from creating such beans and from maintaining the lifecycle of such beans. Also, such beans can be injected into other classes using *@Inject* (see later).

@RequestScoped means that an instance of the class is short-lived, i.e., it is destroyed after the request (and database access) has completed. This is the reasonable choice in this case as there is no need to keep an instance longer than for a single DB access because it does not store any state that is needed across multiple requests (for this, instance variables would be needed). To keep an instance for a longer time, there's *@SessionScoped* (see later).

ConnectionPool.getConnection()

This uses the JDBC resource that was configured in Payara and returns a connection. For details, see the notes on the next slide.

Connection Pooling

Opening a connection to the database is a time-consuming process that can degrade the performance of an application. It is therefore common practice to create a collection of connection objects and store them in another object, which is usually called a connection pool. The connections in the pool are then shared by all users of the web application. You usually don't «invent your own connection pool», but use one that is provided by the application server.

try-with-resources with JDBC objects

Note that the lines

```
try (Statement statement = connection.createStatement();
     ResultSet rs = statement.executeQuery(query);) {
```

use the so-called "try-with-resources" construct that is available since Java SE 7. It allows you to automatically close an object (in this case the Statement and ResultSet object) when it is no longer needed and allows more compact code because otherwise, they would need be closed in the finally block. For details, refer to *https://blogs.oracle.com/WebLogicServer/entry/using_try_with_resources_with*

Marketplace Code – *ProductDatabase.java* (2)

Go through the result set, create *Product* beans and add them to a List of *Product* beans

```
Product product;
while (rs.next()) {
    product = new Product();
    product.setCode(rs.getString("Code"));
    product.setDescription(rs.getString("Description"));
    product.setPrice(rs.getDouble("Price"));
    products.add(product);
}
    } catch (SQLException e) {
        // Returns an empty list
    } finally {
        ConnectionPool.freeConnection(connection);
    }
    return products;
  }
}
```

Return the connection to the connection pool

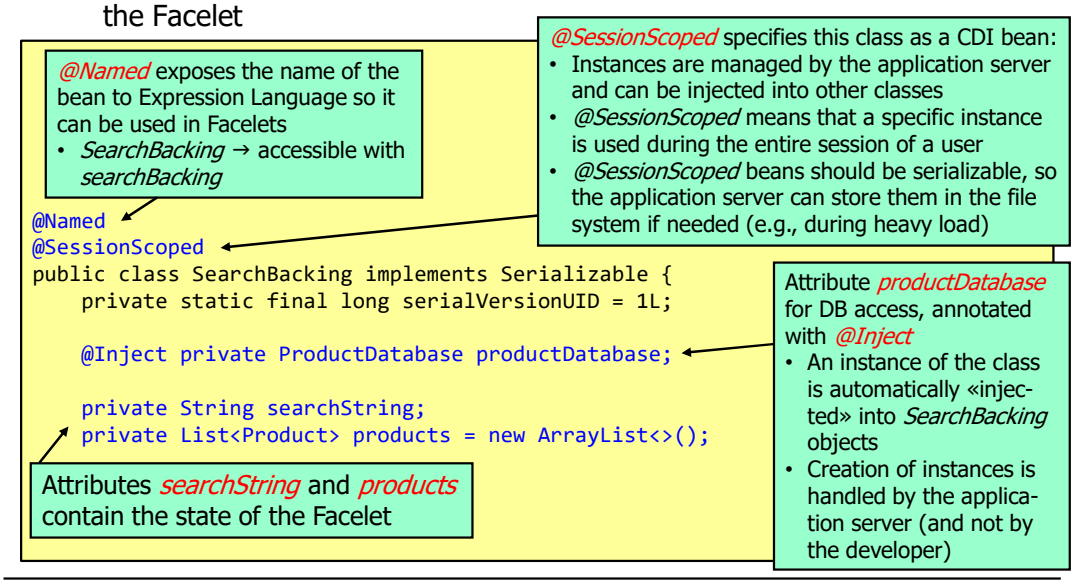
ConnectionPool Class

For completeness, the code of the class from which to get a connection from the pool:

```
import javax.sql.DataSource;
import java.sql.Connection;
import javax.naming.InitialContext;

public class ConnectionPool {
    private static DataSource dataSource = null;
    static {
        try {
            dataSource = (DataSource) new
                         InitialContext().lookup("java:global/marketplace");
        } catch (Exception e) {
            // Do nothing
        }
    }

    public static Connection getConnection() {
        try {
            return dataSource.getConnection();
        } catch (Exception e) {
            return null;
        }
    }

    public static void freeConnection(Connection c) {
        try {
            c.close();
        } catch (Exception e) {
            // Do nothing
        }
    }
}
```

Marketplace Code – *SearchBacking.java* (1)

- *SearchBacking* is the backing bean for the Facelet *search.xhtml* (which corresponds to the web page to search for products)
  - Its task is to store the state of the Facelet and to provide functionality for the Facelet



Import Statements

Due to space restrictions, we left out the import statements above:

```
import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;
import jakarta.inject.Inject;
import jakarta.inject.Named;
import jakarta.enterprise.context.SessionScoped;
import ch.zhaw.securitylab.marketplace.common.database.ProductDatabase;
import ch.zhaw.securitylab.marketplace.common.model.Product;
```

State

The state that is stored in the backing bean are the search string received from the user and the list of products that corresponds to the search results. The attributes *searchString* and *products* are chosen accordingly.

@SessionScoped

Just like *@RequestScoped*, *@SessionScoped* specifies a class as a CDI bean. In contrast to *@RequestScoped*, where an instance is destroyed after a request has been handled, *@SessionScoped* guarantees that once an instance has been created, the instance will be used throughout the entire session. As every user has its own session, this implies of course also that every user has its own instance of a *@SessionScoped* bean (but there's only one instance per user / session)

The main reason for making this backing bean *@SessionScoped* is to make sure that when the user returns to the search page while navigating the application, he will still get the previous search results. With *@SessionScoped*, this is guaranteed as the same instance of the *SearchBacking* bean (including the state that is stored in the backing bean) is used during the entire session of the user. With *@RequestScoped*, this would not be the case as the backing bean would be destroyed after each request.

@Inject

With *@Inject*, an instance of a class (here: a *ProductDatabase* object) is automatically injected and therefore available as if it were created explicitly (using *new*). The object that is injected is either newly created by the application server (which is likely the case here as *ProductDatabase* is *RequestScoped*) or it may already be available in the application server. As discussed before, *ProductDatabase* is *@RequestScoped*, which means an instance is freshly created for each request and therefore for each database access.

Marketplace Code – *SearchBacking.java* (2)

```
    public String getSearchString() {
        return searchString;
    }

    public void setSearchString(String searchString) {
        this.searchString = searchString;
    }

    public List<Product> getProducts() {
        return products;
    }

    public String search() {
        products = productDatabase.searchProducts(searchString);
        return "/view/public/search";
    }

    public int getCount() {
        return products.size();
    }
}
```

Getter and setter methods

search() method searches for products that match the *searchString*
- Will be used by the Facelet *search.xhtml*
- Received products are stored in attribute *products*
- The method returns the name of the Facelet to send back to the browser: */view/public/search* → *search.xhtml*

getCount() method to get the number of products, also used by *search.xhtml*

Marketplace Code – *search.xhtml* (1)

- *search.xhtml* is the Facelet that implements the web page that is used to search for products and to display the search results
  - Basically, a Facelet is an XHTML document that uses both standard HTML tags and JSF tags (which typically start with *h:*, e.g., *<h:head>*)

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

    <h:head>
        <title>Marketplace</title>
        <h:outputStylesheet library="css" name="marketplace.css" />
    </h:head>


    <h:body>
        <div id="header">
            <h1>Welcome to Marketplace v01</h1>
            <h:messages globalOnly="true" layout="table"
                        infoClass="redItalicText"/>
        </div>
```

xmlns attributes declare the namespaces of the JSF tags (≈import statements)

h:messages displays all messages that were generated when the previous request was handled by the web application
- E.g., if the previous request successfully completed a purchase, a message is generated at the end of the purchase, which is included here in the search page because the search page is served to the browser next

<h:head> vs. *<head>*

For the standard HTML tags, one can also use the «normal» HTML tags, e.g., *<head>*. However, one usually uses the JSF versions of the HTML tags in Facelets (if they are available), so we use *<h:head>*.

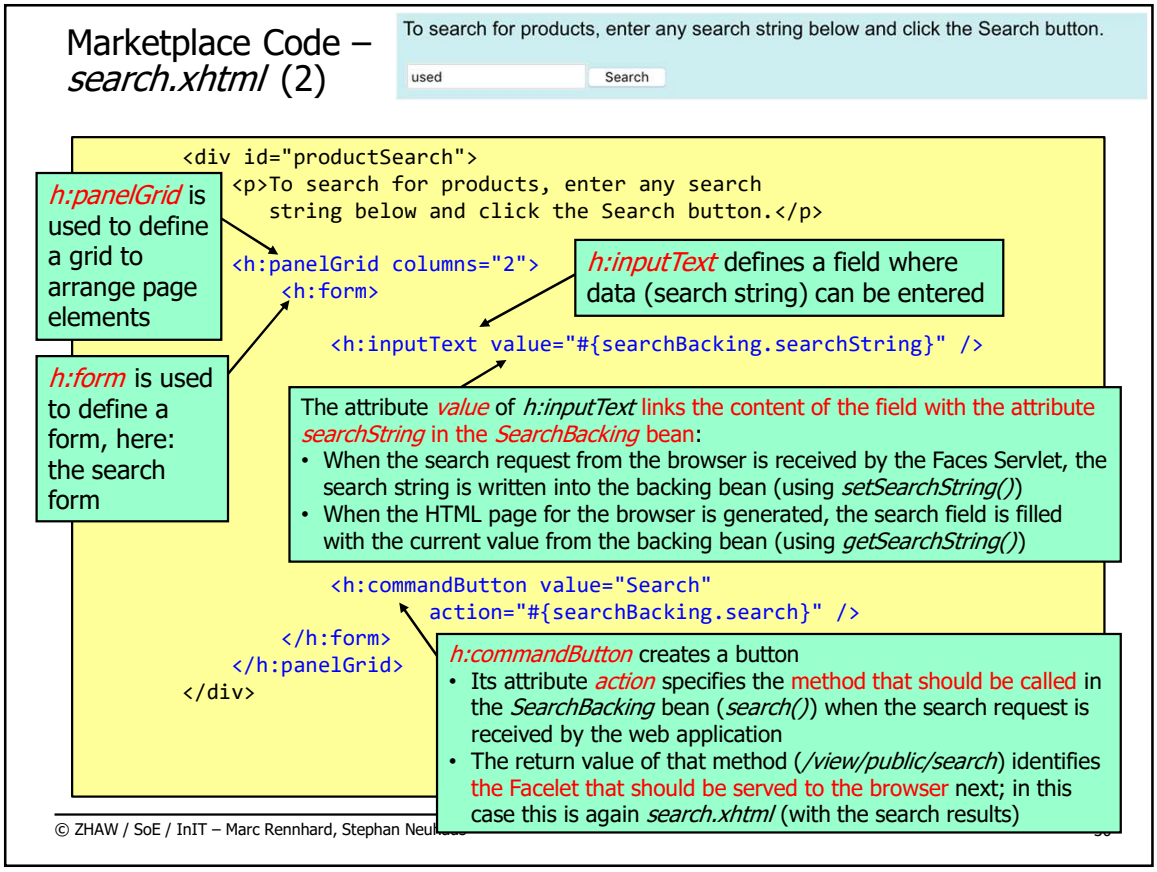
<h:messages>

For instance, when successfully completing the checkout process, the *CheckoutBacking* bean generates a message "*Your purchase has been completed, thank you for shopping with us"*. As the next Facelet that is served to the browser is the search Facelet and as it includes *<h:messages>*, the message is included into the generated search page.

How does Message passing between Components work?

When a component sets a message (e.g., the *CheckoutBacking* bean in our case), the message is stored in the so called faces context. The faces context is a core object that is used whenever a JSF application handles a request. Basically, it's used to store all relevant state information while the request is handled and the response is generated. This means when using *<h:messages>*, in a Facelet, this faces context is accessed to read possibly available messages that have been set before.

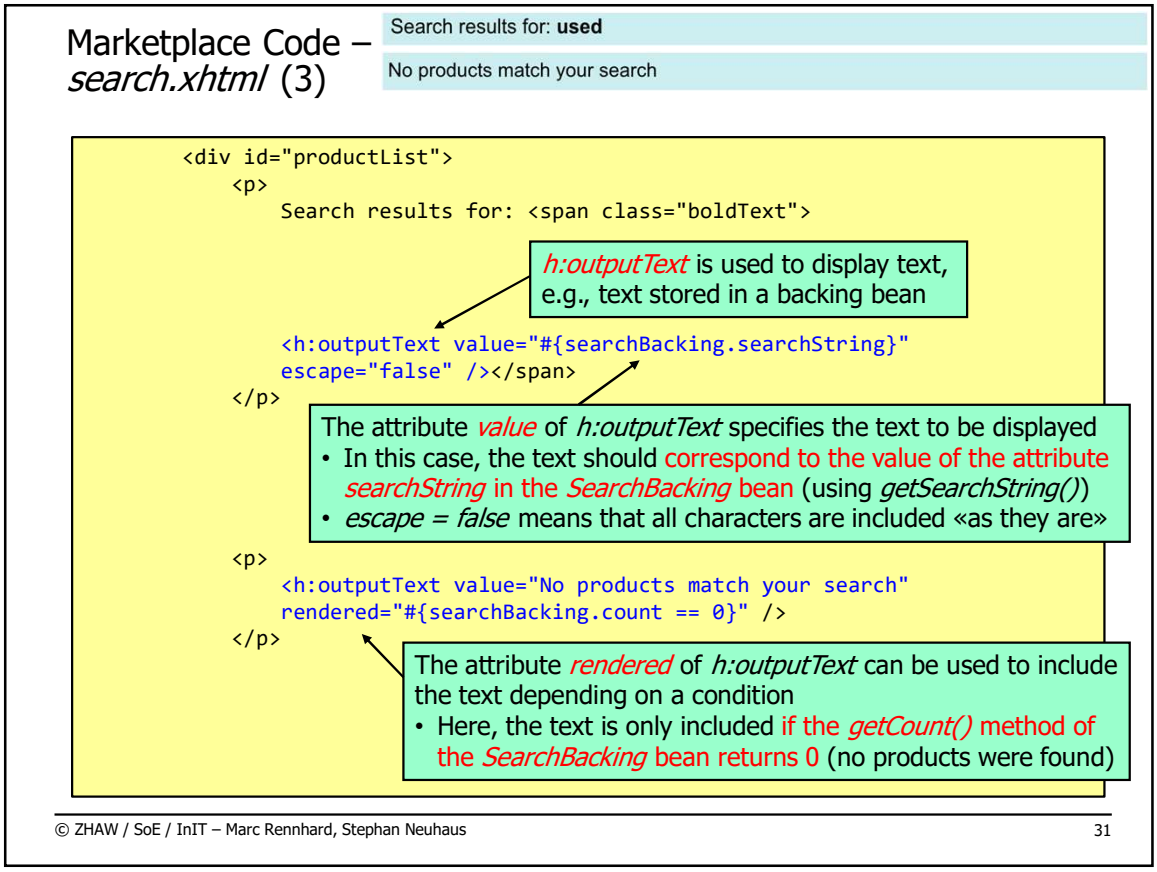
From the Jakarta EE API documentation: *FacesContext contains all of the per-request state information related to the processing of a single Jakarta Server Faces request, and the rendering of the corresponding response. It is passed to, and potentially modified by, each phase of the request processing lifecycle.*



searchBacking.searchString* and *searchBacking.search

searchBacking.searchString corresponds to attribute *searchString* the the *SearchBacking* bean and *searchBacking.search* corresponds to a call of method *search()* in the *SearchBacking* bean.

When the Faces Servlet creates the HTML page from this Facelet, the resulting search form contains the search string as a «normal» form field and the action is a «normal» form action that uses POST to submit the form. In addition, the Faces Servlet includes a hidden field into the form which uniquely identifies the form. Based on the value of this hidden field, the Facelet can unambiguously associate a POST request it receives from the browser with the corresponding form in the Facelet, which allows the Facelet Servlet to perform the right activities. In the case of the search form, this includes first updating attribute *searchString* in the *SearchBacking* bean based on the received POST parameter and then calling method *search()* in the *SearchBacking* bean.



h:outputText

When the Faces Servlet sees such a tag when creating the HTML page from the Facelet, it replaces the tag with text that corresponds to what is specified in attribute *value*. In this case, the attribute is linked with the instance variable (Attribute) *searchString* in the *SearchBacking* bean, which means the tag is replaced with the current value of this instance variable. When accessing the search page for the first time, then this instance variable contains the empty string (as no search has been conducted so far). When accessing it later, it contains the search string that was used previously.

Marketplace Code – *search.xhtml* (4)

Description	Price (CHF)	
DVD Life of Brian - used, some scratches but still works	5.95	Add to cart
Commodore C64 - used, the best computer ever built	444.95	Add to cart

```
<h:dataTable value="#{searchBacking.products}" var="product"
    rendered="#{searchBacking.count > 0}" styleClass="list"
    rowClasses="odd,even" columnClasses="text,price,link">
```

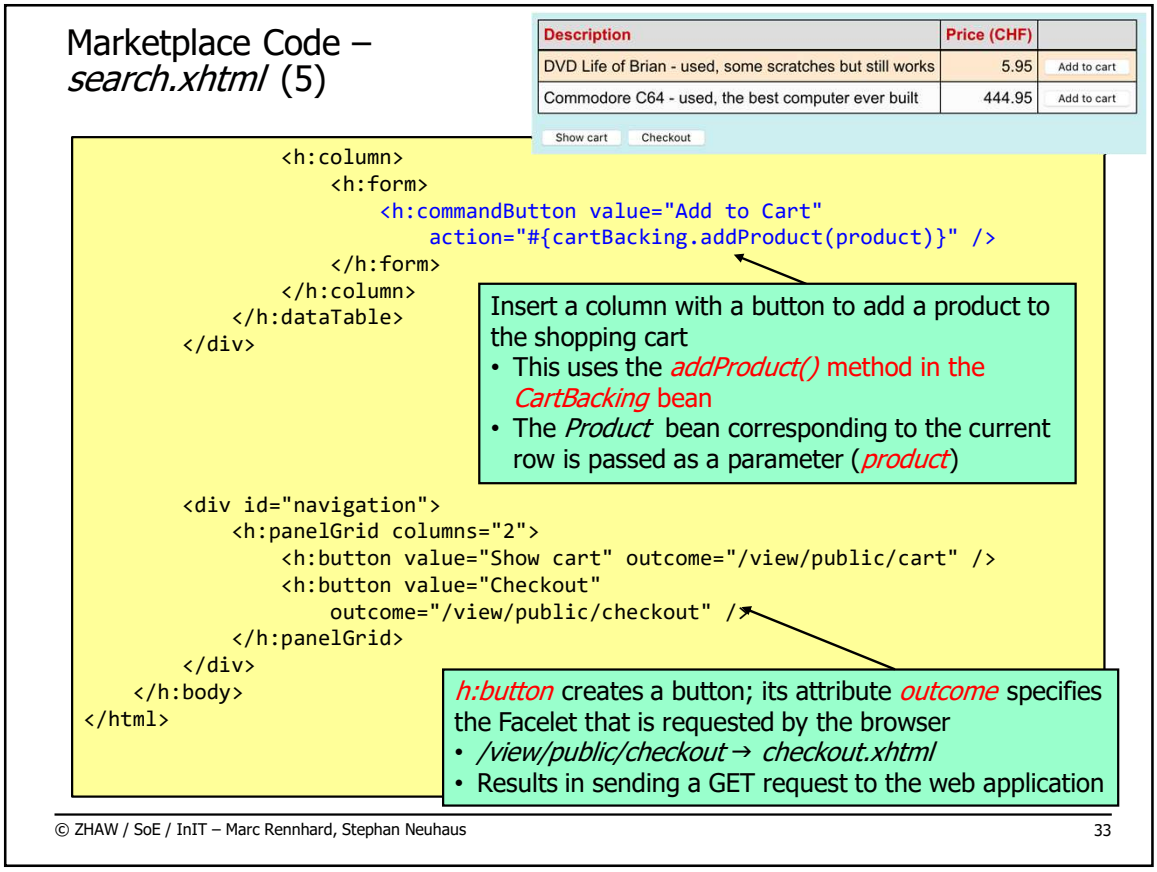
h:dataTable is used to create a table based on data in the *SearchBacking* bean
- *value* specifies to use the data in attribute *products (List<Product>)*
- *var* specifies the name to identify an individual entry in the list (*product*)
- The table is only displayed if the *getCount()* method of *SearchBacking* is > 0
- The Faces Servlet translates this to a standard HTML table

h:column inserts a column into the table
- *f:facet* is used for the column header
- The values for the rows correspond to the description of the individual products in the product list (using *getDescription()* from the *Product* bean)

```
<h:column>
    <f:facet name="header">Description</f:facet>
    #{product.description}
</h:column>

<h:column>
    <f:facet name="header">Price (CHF)</f:facet>
    <h:outputText value="#{product.price}">
        <f:convertNumber pattern="#0.00" />
    </h:outputText>
</h:column>
```

#{...} (called a *value expression*) is the preferred alternative for *h:outputText*, should be used if no specific options (e.g., *rendered*, *escape*) are needed

Insert a column that contains the price (formatted using *f:convertNumber*)

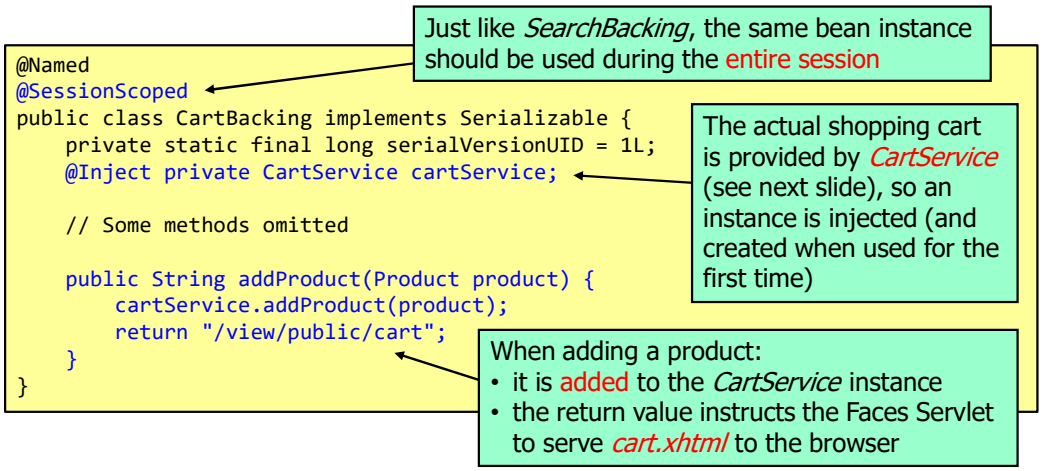


h:commandButton

When generating the HTML page, the Faces Servlet translates a *commandButton* into a form that contains a button (a form field with *input type="submit"*), which creates a POST request. Just like with other forms, a hidden field is included that uniquely identifies the form, which allows the Facelet Servlet to associate the received POST request with the *commandButton* and to call the desired method in the backing bean (here: *addProduct(Product product)* in *CartBacking* bean).

Marketplace Code – *CartBacking.java*

- *search.xhtml* uses not only *searchBacking*, but also *cartBacking* to put a product into the cart (*addProduct()* method)
- Creating *cartBacking* works the same as with *searchBacking*:
  - When it is used for the first time, it is created by the Faces Servlet



Import Statements

Due to space restrictions, we left out the import statements above:

```
import java.io.Serializable;
import java.util.List;
import jakarta.inject.Inject;
import jakarta.inject.Named;
import jakarta.enterprise.context.SessionScoped;
import ch.zhaw.securitylab.marketplace.common.model.Product;
import ch.zhaw.securitylab.marketplace.common.service.CartService;
```

@SessionScoped

There's just one shopping cart per session so it's reasonable to make this bean *@SessionScoped*. However, the bean could also be *@RequestScoped* as it does not contain any state (the state of the shopping cart is stored in *CartService.java*). However, there's also no reason to create a new *CartBacking* bean whenever the cart content is displayed, so *@SessionScoped* is a reasonable choice for performance reasons, especially as only «very little storage space» is needed to keep it in memory.

Marketplace Code – *CartService.java*

CartService implements the shopping cart; obviously, the same shopping cart should be used during the entire session

```
@SessionScoped
public class CartService implements Serializable {
    private static final long serialVersionUID = 1L;
    private List<Product> products = new ArrayList<>();

    public List<Product> getProducts() {...}
    public int getCount() {...}
    public void empty() {...}

    public double getTotalPrice() {
        double total = 0.0;
        for (Product product : products) {
            total += product.getPrice();
        }
        return total;
    }

    public void addProduct(Product product) {
        if ((product != null) && !(products.contains(product))) {
            products.add(product);
        }
    }
}
```

Getter method and methods to count the number of products and to empty the cart

Returns the total price of all products in the shopping cart (used during checkout)

Adds a product to the cart if it's not there yet

Import Statements

Due to space restrictions, we left out the import statements above:

```
import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;
import jakarta.enterprise.context.SessionScoped;
import ch.zhaw.securitylab.marketplace.common.model.Product;
```

Why a separate Class?

The actual shopping cart could also be maintained in *CartBacking*. However, the shopping cart will also be used in the next chapter later when extending Marketplace with web service functionality, so we use a separate class for the shopping class that can be reused when implementing the web service functionality.

Marketplace Code – *web.xml* (1)

- Every Jakarta EE web application requires a file *web.xml* that contains several configurations, including security configurations

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="https://jakarta.ee/xml/ns/jakartaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee
         https://jakarta.ee/xml/ns/jakartaee/web-app_5_0.xsd"
         version="5.0">

    <servlet>
        <servlet-name>Faces Servlet</servlet-name>
        <servlet-class>jakarta.faces.webapp.FacesServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>Faces Servlet</servlet-name>
        <url-pattern>/faces/*</url-pattern>
    </servlet-mapping>

    <context-param>
        <param-name>jakarta.faces.PROJECT_STAGE</param-name>
        <param-value>Production</param-value>
    </context-param>

    <session-config>
        <session-timeout>10</session-timeout>
    </session-config>
```

Standard entries in every JSF application, make sure that all requests are handled by the Faces Servlet

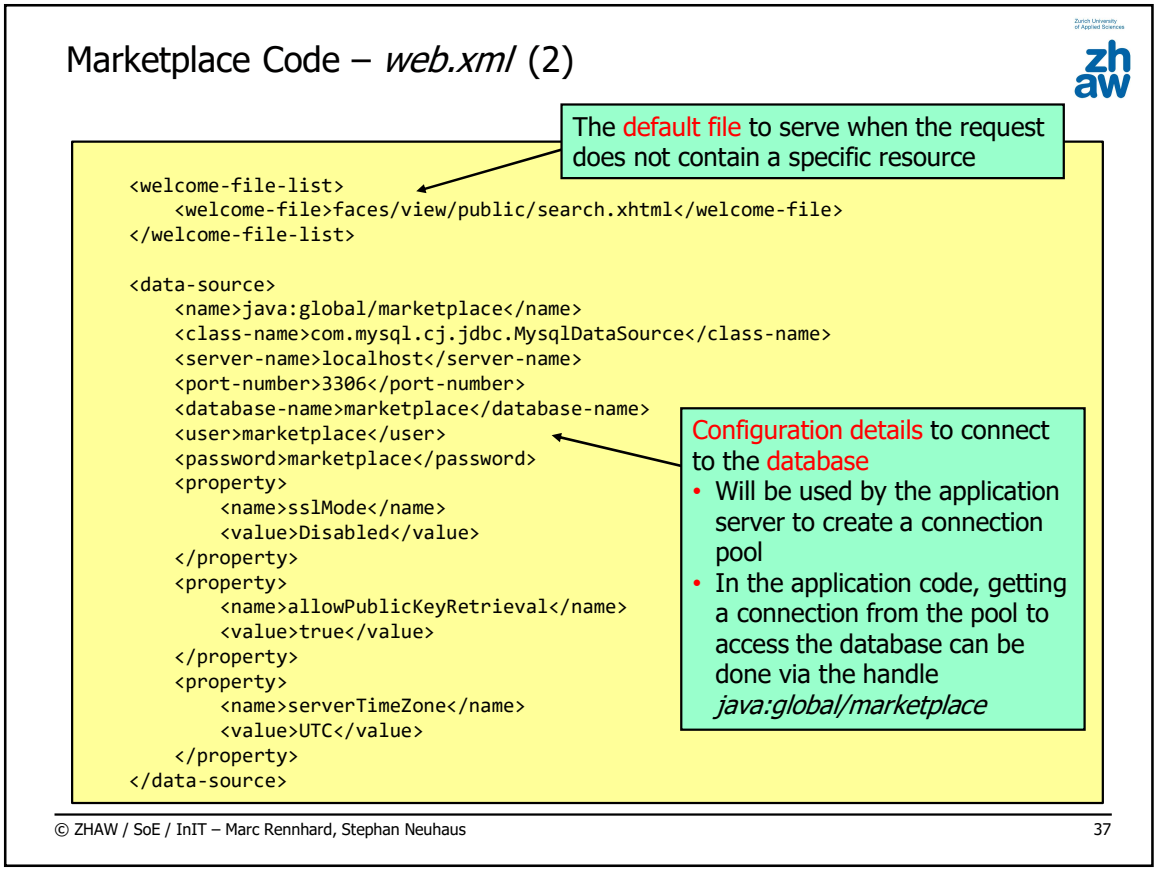
Specifies the details of error messages

The session timeout in minutes (after 10 minutes inactivity, the server terminates the session)

Attributes of *web-app* Tag

You don't really have to understand the attributes of the web-app tag as they are usually created correctly by your IDE. In this example, you can see that the servlet version is 5.0, which is the version that is part of Jakarta EE 9.

Marketplace Code – *web.xml* (2)



Connection Pool

Modern web applications usually use such a connection pool for performance reasons Otherwise, it would be necessary to create / destroy a database connection for every single database access, which implies a performance overhead.

sslMode

The Marketplace application does not use TLS (SSL) to connect from the web application (on Payara) to the database application, as both applications run on the same server. In a productive setting and especially if the database runs on a different host, TLS should be used. Check out the documentations of the application server and database server that is used to configure this correctly.

Marketplace Code – *web.xml* (3)

- The final configuration in *web.xml* enforces that HTTPS is used throughout the entire Marketplace application
  - This should be used for any web application these days
  - The effect of this entry that any request to an HTTP resource (e.g., to the entry page) is redirected to the same location but over HTTPS, which is the standard behavior that should be used in practice
- Payara support HTTPS out-of-the-box (using port 8181 (HTTP: 8080))
  - It uses a self-signed standard certificate (and will trigger a warning in the browser) – so make sure to use a real certificate in a productive setting

```
    <security-constraint>
        <web-resource-collection>
            <web-resource-name>HTTPS everywhere</web-resource-name>
            <url-pattern>/*</url-pattern>
        </web-resource-collection>
        <user-data-constraint>
            <transport-guarantee>CONFIDENTIAL</transport-guarantee>
        </user-data-constraint>
    </security-constraint>
</web-app>
```

The *url-pattern* /* guarantees that any request to HTTP is redirected to HTTPS

<security-constraint>

security-constraint entries in *web.xml* are used to do security configurations. It includes a web-resource-collection element that defines the URLs to be affected by the constraint (*url-pattern*). The *web-resource-name* can be anything, it's merely used to comment the entry to specify what it is used for. The *user-data-constraint* entry is used to define whether HTTPS has to be used for the defined *url-pattern(s)*. Enforcing HTTPS is done by setting *transport-guarantee* to *CONFIDENTIAL*.

security-constraint entries are not only used to enforce HTTPS, but also to enforce access restrictions for users, as will be discussed later.

Marketplace – Exercise

Based on what you have seen so far about the Marketplace application, could you already spot some security issues?