

Developing Secure Traditional Web Applications – Part 2/3

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

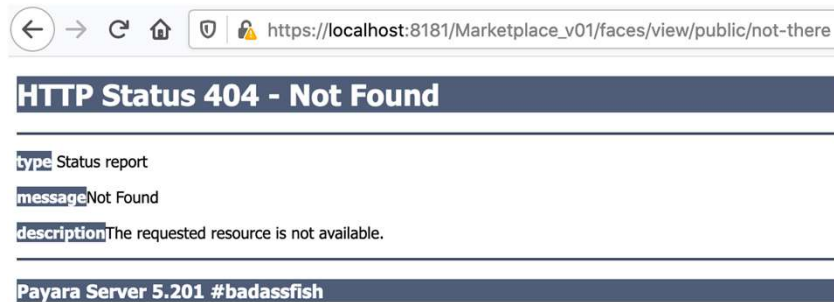
Preventing Information Leakage in Error Messages

Marketplace V02

The extensions of this section are integrated in Marketplace_v02.

Information Leakage in Error Messages (1)

- Per default, an error in a Jakarta EE web applications results in sending an **application server-specific message** to the browser
- If an **exception** is thrown and not handled, this usually results in **including the stack trace in the error message**
 - This is convenient during development and testing
 - But should not be done in a productive system
- Example 1: accessing a **non-existing resource**:



Information Leakage in Error Messages (2)

- Example 2: **unhandled exception** when entering a first name with more than 50 characters during checkout



- Such an exception should never be leaked to the browser, as it provides an attacker with **internal information about the application**
 - Which may help to discover and exploit vulnerabilities (e.g., SQL injection)

Exception

This exception above happened because of the following code:

```
String query = "INSERT INTO Purchase (Firstname, Lastname,
CreditCardNumber, TotalPrice) " +
    "VALUES ('" + purchase.getFirstname() + "', '" +
    purchase.getLastname() + "', '" + purchase.getCreditCardNumber() +
    "', " + purchase.getTotalPrice() + ")";
try (Statement statement = connection.createStatement()) {
    return statement.executeUpdate(query);
} catch (SQLException e) {
    StringWriter sw = new StringWriter();
    e.printStackTrace(new PrintWriter(sw));
    throw new RuntimeException("Offending SQL query: " + query +
        "\n" + sw.toString());
} finally {
    ConnectionPool.freeConnection(connection);
}
```

Apparently, the developer wanted to insert additional information (here: the offending SQL query) into the exception message to detect problems within the application. That's fine, but he did this in a way such that the exception is also sent to the browser, which should never be done. Exceptions should only be logged locally on the application server, but never sent to the browser.

Marketplace – Standard Error Handling (1)

- Jakarta EE makes it very simple to enforce a standard error handling that is used throughout the application
 - This is configured in the deployment descriptor (*web.xml*)

- Handling of specific HTTP error codes:

```
<error-page>
  <error-code>404</error-code>
  <location>/faces/view/public/error.xhtml</location>
</error-page>
```

The HTTP error code

The resource to display, *error.xhtml* is a simple Facelet that contains a generic error message

- Handling of unhandled exceptions:

```
<error-page>
  <exception-type>java.lang.Throwable</exception-type>
  <location>/faces/view/public/error.xhtml</location>
</error-page>
```

The exception to handle
(here: any exception)

- Handling of **all errors** (usually the best option):

```
<error-page>
  <location>/faces/view/public/error.xhtml</location>
</error-page>
```

error.xhtml

We use just a simple Facelet for this:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      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>Error</h1>
      <p>Something went wrong, please try again later.</p>
    </div>
  </h:body>
</html>
```

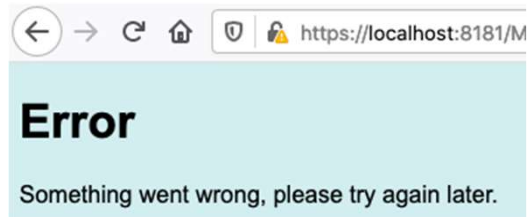
Unhandled Exception and HTTP error 500

An unhandled exception is always sent to the browser in the form of an HTTP 500 message. Therefore, one could also use the following for the second `<error-page>` tag:

```
<error-page>
  <error-code>500</error-code>
  <location>faces/view/public/error.xhtml</location>
</error-page>
```

Marketplace – Standard Error Handling (2)

- Adding the configuration to handle all errors to *web.xml* results in the following in case of a 404 error or an unhandled exception:



- Of course, the code section where the previously observed exception is thrown **should also be fixed!**
 - `<error-page>` elements in *web.xml* to prevent exceptions from leaking to the browser should be considered only as a **second line of defense!**
- **Best practice:**
 - Add standard error handling to *web.xml* early during development
 - Comment the entries during development so errors and exceptions are visible in the browser (so they can be detected and corrected)
 - Don't forget to uncomment them in production!

Data Sanitation

Marketplace V03

The extensions of this section are integrated in Marketplace_v03.

Reflected Cross-Site Scripting (XSS)

- The Marketplace application **reflects the inserted search string**

To search for products, enter any search string below and click the Search button.

Brian Search

Search results for: **Brian**

Description	Price (CHF)	
DVD Life of Brian - used, some scratches but still works	5.95	Add to Cart

- Such data reflection always **bears the risk of a reflected XSS vulnerability**, in particular if control characters are not properly sanitized (encoded) before they are inserted into the HTML document
- Proof-of-concept of an **existing XSS vulnerability**

To search for products, enter any search string below and click the Search button.

<script>alert("XSS");</script> Search

To search for products, enter any search string below and click the Search button.

<script>alert("XSS");</script> Search

Search results for:

XSS

OK

- There are two options to fix this problem: **input validation or data sanitation**
 - **Input validation**: Do not accept search strings that include JavaScript code
 - **Data sanitation**: Encode critical control characters before the search string is included in the web page (e.g., replace `<` with `<`;))
- Question: Should this be fixed with input validation or data sanitation?

Data Sanitation with JSF (1)

- Currently, *search.xhtml* is implemented as follows:

```
<p>
  Search results for: <span class="boldText">
    <h:outputText value="#{searchBacking.searchString}"
      escape="false" /></span>
</p>
```

- The reason why XSS is possible is because the *escape* attribute is set to *false*
 - As a result, the control characters in the search string are not sanitized
 - Therefore, the search string is included in the resulting HTML document «as it is»: `<script>alert("XSS");</script>`
- To *prevent XSS*, the control characters in the JavaScript must be sanitized
 - JSF provides a built-in mechanism to do this

Data Sanitation with JSF (2)

- To enable data sanitation, set the *escape* attribute to true

```
Search results for: <span class="boldText">
<h:outputText value="#{searchBacking.searchString}" escape="true" />
</span>
```

- Or – even better – omit it completely, as it's true per default

```
Search results for: <span class="boldText">
<h:outputText value="#{searchBacking.searchString}" />
</span>
```

- Or – this is the best option – use a value expression instead of *h:outputText* if no specific options are used

- With value expressions, *escape* is always implicitly true

```
Search results for: <span class="boldText">
#{searchBacking.searchString}</span>
```

- This means that unless you are actively deactivating data sanitation by setting *escape* to false, XSS shouldn't be an issue in JSF applications

Data Sanitation with JSF (3)

- Reflected XSS does **no longer work**

To search for products, enter any search string below and click the Search button.

Search

Search results for: **<script>alert('XSS');</script>**

No products match your search

- Inspecting the HTML code confirms that JavaScript control characters (< and >) are **HTML encoded**
 - As a result, the code is not executed but only displayed

```
<div id="productList">
  <p>
    Search results for: <span class="boldText">
      &lt;script&gt;alert('XSS');&lt;/script&gt;</span>
    </p>
    <p>No products match your search
  </p>
</div>
```

More Data Sanitation? (1)

- We have now fixed the case of reflected XSS on the search page
- Question: Are there [other places in the Marketplace application](#) where we should perform data sanitation? If yes, where and why?

More Data Sanitation? (2)

- Example: A malicious product manager / database administrator inserts JavaScript code in a product description:
 - INSERT INTO Product VALUES ('5', '0005', 'XSS: `<script>alert("XSS");</script>`', '1.95', 'luke')
- Search results show the following:

Search results for:

Description	Price (CHF)	
DVD Life of Brian - used, some scratches but still works	5.95	<input type="button" value="Add to Cart"/>
Ferrari F50 - red, 43000 km, no accidents	250000.00	<input type="button" value="Add to Cart"/>
Commodore C64 - used, the best computer ever built	444.95	<input type="button" value="Add to Cart"/>
Printed Software-Security script - brand new	10.95	<input type="button" value="Add to Cart"/>
XSS: <code><script>alert("XSS");</script></code>	1.95	<input type="button" value="Add to Cart"/>

- → No problems so far as the description column uses the value expression `#{product.description}`, which performs data sanitation

Malicious Product Manager

It may also be that product managers get an own tool to manage the products, and security (e.g., input validation) is often neglected in such internal administrator tools. All this justifies that it is very important to sanitize all data sent to the user.

More Data Sanitation? (3)

- Adding the product to the shopping cart results in:



- The reason is that the following code is used:

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

- To prevent such attacks, it's important that you perform data sanitation with all data that is read from external components and that is included in web pages, no matter where the data stems from
 - From the user, the database, a file, a third party system,...
 - This should prevent all XSS (or JavaScript injection) attacks, in particular also stored XSS attacks
 - In JSF, this means that you should never set the *escape* attribute to false, unless you really want to do this and know what you are doing

More Data Sanitation? (4)

- Correct data sanitation in *cart.xhtml*:

```

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

```

- Shopping cart page:

Your Shopping Cart

Description	Price (CHF)
XSS: <script>alert("XSS");</script>	1.95

- Final Remark
 - In JSF, preventing XSS is easy because it **works correctly per default**
 - But with other technologies, it may be required to **actively set an attribute / flag** to enforce data sanitation
 - Therefore, it's very important you understand the technology you are using and especially its security features / options

Secure Database Access

Marketplace V04

The extensions of this section are integrated in Marketplace_v04.

SQL Queries based on String Concatenation (1)

- Currently, the Marketplace application uses **string concatenation to build SQL queries**
 - This is very critical, especially if the data received from the user is included in the string concatenation without proper input validation
- To demonstrate that Marketplace is vulnerable to SQL injection, we abuse the search function to **access all data in the table *UserInfo***
 - This table will be user later, which is why it wasn't introduced yet in the context of the basic Marketplace application

- ***UserInfo* table:**

Username	Pbkdf2Hash
alice	PBKDF2WithHmacSHA512:100000:A6kJSvRyVyADUIKum2UMhk0OxU5...
bob	PBKDF2WithHmacSHA512:100000:UC7b0kdr35kVDsflAMYvZyg8iWWT...
donald	PBKDF2WithHmacSHA512:100000:yeukX8nk0vQJszfihTvXUtep44QV42...
john	PBKDF2WithHmacSHA512:100000:FX6Me8UamCrYiafwoVOa4RY8yQD...
luke	PBKDF2WithHmacSHA512:100000:Qb2r0ep0HI42KmEZ6xa8EvHKWoA...
robin	PBKDF2WithHmacSHA512:100000:vx0EYPm/hLEs5aqsq12jRXEjZmCwj...
snoopy	PBKDF2WithHmacSHA512:100000:GOZNnVcF/czUpuHxAyoc7+DfKKJC...

SQL Queries based on String Concatenation (2)

- In *ProductDatabase.java*, the query is built as follows:
 - String query = "SELECT * FROM Product
WHERE Description LIKE '%" + *searchString* + "%'";
- To also read the contents of table *UserInfo*, we have to inject the following:
 - DVD%' UNION SELECT 1,2,CONCAT_WS(" - ",Username,
Pbkdf2Hash),4,5 FROM UserInfo--
 - As only one varchar column of the predefined SELECT statement is displayed, we use the *CONCAT_WS* MySQL function to concatenate the columns for *Username* and *Pbkdf2Hash*
- Resulting query:
 - String query = "SELECT * FROM Product
WHERE Description LIKE '%DVD%' UNION
SELECT 1,2,CONCAT_WS(" - ",Username,Pbkdf2Hash),4,5
FROM UserInfo-- %'

searchString directly corresponds to the search string received from the user

Note: With MySQL, a space character must follow the comment mark!

UNION Statements

Remember: The SELECT statements combined in a UNION statement must both return the same number of columns and the data types must match.

SQL Queries based on String Concatenation (3)

- Submitting the query as search string results in the following:

Search results for: DVD%' UNION SELECT 1,2,CONCAT_WS("-",Username,Pbkdf2Hash),4,5 FROM UserInfo WHERE User

Description
DVD Life of Brian - used, some scratches but still works
alice - PBKDF2WithHmacSHA512:100000:A6kJSvRyVvADUIKum2UMhk0OxU5kGz5zqMaOkILQrWo2PVD4D9rXNqlUQldoujWnJKKP
bob - PBKDF2WithHmacSHA512:100000:UC7b0kdr35kVDsflAMYyZyg8iWWTkHRXia9ZVvoR9AxMPxR06AJVQBkKENDvSCNY3X0v
donald - PBKDF2WithHmacSHA512:100000:yeukX8nk0vQJsZfihTvXUtep44QV42onNdI8TarDAswnYb1HO4ZMQ2q2Df6WpbTS /3spiJ2eRKh+AmNG+0Kjp+s=
john - PBKDF2WithHmacSHA512:100000:FX6Me8UamCrYiafoVOa4RY8yQDt88IIKUQM2afLc3DBdvUYqwQdL+fb+O8ei9Zch /nZFNi90RiUEejuocy74jvG+lzy1NY=
luke - PBKDF2WithHmacSHA512:100000:Qb2r0ep0HI42KmEZ6xa8EvHKWoA4x+Ye7iyP1f1ykCrGTq/KhRH/JnAjPRVQqgHrdi
robin - PBKDF2WithHmacSHA512:100000:vx0EYPm/hLEs5aqsq12jRXEjZmCwjSFDa4VD1JU0OmK3P1GcZ+juc9mrwO4z6uf0xJhUqw
snoopy - PBKDF2WithHmacSHA512:100000:GOZNnVcF/czUpuHxAyoc7+DfKKJCOnc4cTHZ73j/fKodn/7hyc20gWUC7PFEyyYr /LLEHI7Kkbq9bsuuXxEgD8PE8=

What if the attacker does not know the database schema?

Try to access system tables – also with SQL injection. With MySQL, use INFORMATION_SCHEMA.TABLES and .COLUMNS.

- To get all tables the DB user *marketplace* is allowed to access:
DVD%' UNION SELECT 1,2, TABLE_NAME,4,5 FROM INFORMATION_SCHEMA.TABLES WHERE
TABLE_TYPE = 'BASE TABLE' --
- And from the table *UserInfo*, get the column names:
DVD%' UNION SELECT 1,2,COLUMN_NAME,4,5 FROM INFORMATION_
SCHEMA.COLUMNS WHERE TABLE_NAME = 'UserInfo' --

MySQL and INFORMATION_SCHEMA.TABLES and .COLUMNS

Each MySQL user has the right to access these tables, but can see only the rows in the tables that correspond to objects for which the user has the proper access privileges (e.g., SELECT). As a result, it is usually possible to get schema information with SQL injection when MySQL is used – provided an SQL injection vulnerability exists.

See <http://dev.mysql.com/doc/refman/5.5/en/information-schema.html>

SQL Injection on INSERT queries (1)

- We can also exploit the **INSERT** query that inserts a purchase
- In *PurchaseDatabase.java*, the query is built as follows:
 - `String query = "INSERT INTO Purchase (Firstname, Lastname, CreditCardNumber, TotalPrice) VALUES ('" + purchase.getFirstname() + "', '" + purchase.getLastname() + "', '" + purchase.getCreditCardNumber() + "', " + purchase.getTotalPrice() + ")";`
- In the application, the first three values **directly correspond to the values provided by the user during checkout**; the fourth (total price) is computed internally
- Question: **How could this be exploited in a «beneficial way»?**

First name:	<input type="text"/>
Last name:	<input type="text"/>
Credit card number:	<input type="text"/>
<input type="button" value="Complete purchase"/>	

SQL Injection on INSERT queries (2)

- Exploiting the vulnerability allows to set the **value of the total price to an arbitrary value**, e.g., to 0 or even to a negative value
- To execute the attack, the following is inserted in the credit card number field during checkout:
 - 1111 2222 3333 4444', -1000)--
- Resulting query:
 - INSERT INTO Purchase (Firstname, Lastname, CreditCardNumber, TotalPrice) VALUES ('Mickey', 'Mouse', '1111 2222 3333 4444', -1000)-- ', 5.95)
- Result in DB:

PurchaseID	Firstname	Lastname	CreditCardNumber	TotalPrice
4	Mickey	Mouse	1111 2222 3333 4444	-1000.00

Negative Price

Inserting a negative total price may allow the attacker to receive money via a credit card chargeback.

Prepared Statements

- Again, one can argue that **proper input validation** should prevent this
 - But what if user should be allowed to search for any string?
- The fundamentally right approach to get **protection from SQL injection** in general is therefore to use **Prepared Statements**
- What are prepared statements?
 - Prepared statements are SQL statements that are sent to the DBMS **before they are actually «used and executed»**
 - When the DBMS receives a prepared statement, it is checked for **syntactical correctness and precompiled**
 - They **typically contain parameters** that are specified before the statement is actually executed
 - **Why is this secure? Because after precompiling, the statement «is fixed» and the specified parameters will only be interpreted as simple strings and cannot change the semantics of the query**
 - **If the specified parameters contain SQL control characters, they will be escaped and interpreted as simple characters (e.g., ' → \')**
 - Side note: Prepared statements – if executed repeatedly – **improve performance** as syntax checking and compilation must be done only once

Support for Prepared Statements

Most powerful relational DBMS support prepared statements. Besides SELECT (as used in the following example), prepared statements can also be used with UPDATE, INSERT and DELETE and sometimes for further (e.g., DROP, ALTER etc.) statements. More information about prepared statements in Java can be found here:

<https://docs.oracle.com/javase/tutorial/jdbc/basics/prepared.html>

Why do Prepared Statements prevent SQL Injection?

More information about how prepared statements work and why they prevent SQL injection can be found here:

https://medium.com/@jaredablon_31568/how-to-prevent-sql-injection-vulnerabilities-how-prepared-statements-work-f492c369614f

Marketplace – Prepared Statements (1)

- Modify *searchProducts* method in *ProductDatabase.java*:

```
public List<Product> searchProducts(String searchString) {  
    Connection connection = ConnectionPool.getConnection();  
    List<Product> products = new ArrayList<>();  
  
    // Create the query string using ? to identify parameters  
    String query = "SELECT * FROM Product WHERE Description LIKE ?";
```

Returns an object of
type *PreparedStatement*

The **prepared SQL statement**

- No string concatenation is used
- A ? is used for each parameter that will be specified later (here: for the search string)

```
    try (PreparedStatement ps = connection.prepareStatement(query)) {
```

```
        ps.setString(1, "%" + searchString + "%");
```

Send the prepared
statement **to the DBMS**

```
        try (ResultSet rs = ps.executeQuery()) {  
            Product product;
```

Set the **first parameter**
(a string) to the
specified search string

Execute the query (just like before)

Marketplace – Prepared Statements (2)

- The remainder of the method is unchanged

```
        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;
}
```

- Modify *insert* method in *PurchaseDatabase.java*:

```
public int insert(Purchase purchase) {  
    Connection connection = pool.getConnection();  
  
    String query = "INSERT INTO Purchase (FirstName, LastName, CCNumber,  
        TotalPrice) VALUES (?, ?, ?, ?)";  
  
    try (PreparedStatement ps = connection.prepareStatement(query)) {  
        ps.setString(1, purchase.getFirstname());  
        ps.setString(2, purchase.getLastname());  
        ps.setString(3, purchase.getCreditCardNumber());  
        ps.setDouble(4, purchase.getTotalPrice());  
        return ps.executeUpdate();  
    }  
    ...  
}
```

For modifying queries, use
executeUpdate() instead of
executeQuery()

- Conclusion: Using prepared statements requires very little adaptation and is by no means more difficult than using «normal» statements

- Trying the **SELECT SQL injection attack again** does no longer work:

To search for products, enter any search string below and click the Search button.

DVD%' UNION SELECT 1,2, Search

Search results for: DVD%' UNION SELECT 1,2,CONCAT_WS(" - ",Username,Pbkdf2Hash),4,5 FROM UserInfo--

No products match your search

Show cart Checkout

- Executed query:
 - `SELECT * FROM Product WHERE Description LIKE '%DVD%' UNION SELECT 1,2,CONCAT_WS(" - ",Username,Pbkdf2Hash),4,5 FROM UserInfo\-\- %'`
 - This is **only one** SELECT statement (not two as before) as the submitted control characters were escaped (`'` → `\'`, `-` → `\-`)
 - So the entire injected SQL statement is now interpreted as the search string and the **result set is empty** because there is no product description which matches this string

Marketplace – Prepared Statements (5)

- Trying the **INSERT SQL injection attack again** still seems to work...
(using 1111 2222 3333 4444', -1000)--)

Welcome to Marketplace v04

Your purchase has been completed, thank you for shopping with us

To search for products, enter any search string below and click the Search button.

- Analyzing the DB shows that **SQL injection actually is prevented**
 - Because the entire injected string was accepted as the credit card number
(again because control characters were escaped)

PurchaseID	Firstname	Lastname	CreditCardNumber	TotalPrice
4	Mickey	Mouse	1111 2222 3333 4444', -1000)--	5.95

- But apparently, we have an **input validation issue** here, which will be fixed later

Jakarta Persistence API

Marketplace V05

The extensions of this section are integrated in Marketplace_v05.

- The Jakarta Persistence API (JPA) is the standard **Object-Relational-Mapping (ORM)** framework in Jakarta EE
- Modern applications often use such ORM frameworks for various reasons
 - Provide the **bridge** between **objects** in the code and the database **model**
 - Allows to **read information** from the database directly into objects
 - Allows to **write information** stored in objects directly into the database
 - Not necessary to write low-level and possibly DB-specific **SQL code**
 - If done correctly, **SQL injection** shouldn't be an issue
- Therefore, we adapt the **Marketplace** application such that it uses JPA
 - Just like with JSF, this will also be a basic introduction into the API

- The basis of JPA are **entity classes**
 - Typically, they are associated with a **table** in the database and **each instance corresponds to a row** in the table
- Properties of entity classes
 - Must be annotated with **@Entity** → Jakarta EE recognizes that the classes are used for ORM
 - Contain **instance variables** that correspond to the columns of the table
 - Have a public constructor without any arguments (**standard constructor**) to create instances
 - **Getter and setter** methods for the instance variables (for those that should read / write accessible by other program components)
 - Should be **serializable** so that they can be sent to remote program components over the network if this is needed
 - Not needed in Marketplace, but we do this nevertheless for «best practice»
 - Typically contain **named queries** to specify the SELECT queries
 - Using **@NamedQueries** and **@NamedQuery**

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

- Entity *Product.java* is used for table *Product*
 - It's reasonable to give the entity the name of the table
 - This entity is not only used for database access, but it also replaces the currently used simple JavaBean *Product.java*

```
@Entity
@Table(name = "Product")
@NamedQuery(name = "Product.findByDescription", query = "SELECT p FROM
    Product p WHERE p.description LIKE :description")
public class Product implements Serializable {
    private static final long serialVersionUID = 1L;

    @Id private int productID;
    private String code;
    private String description;
    private BigDecimal price;
    private String username;
```

• **@Entity** marks this class as a JPA entity
• **@Table** specifies the corresponding table

Named query to get all products that match a specified description (JPQL syntax, see later)

@Id specifies the attribute that corresponds to the primary key in the table
• If no further annotations are used, this means the key is automatically generated by the DBMS (auto incremented in our case)

Private instance variables for the table columns
• JPA recommends using *BigDecimal* for columns with DB type *decimal*

Standard Constructor

Just like with the simple JavaBean we used for *Product.java* before, the standard constructor is omitted, as the default constructor does exactly what we want.

Import Statements

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

```
import java.io.Serializable;
import java.math.BigDecimal;
import jakarta.persistence.*;
```

@Table

If the entity has the name of the table, then it's not necessary to use `@Table`. However, it's a good idea to always use `@Table` to make sure that it works on any operating systems (the details are out of scope, but have to do with the fact that on systems with case-sensitive file systems (e.g., Unix Linux), it wouldn't work as per default, JPA uses only uppercase letters for table names, e.g., `PRODUCT` instead of `Product` in the case above).

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

```
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 BigDecimal getPrice() {return price;}
public void setPrice(BigDecimal price) {this.price = price;}
public String getUsername() {return username;}
public void setUsername(String username) {
    this.username = username;}
```

Public
getter and
setter
methods
for some
attributes

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

equals() method to compare two
products based on their *ProductID*
(primary key)
• We already had this in the
simple JavaBean

Automatically creating Entity Classes

Most IDEs allow to automatically generate entities from databases tables. However, this often includes more information (code) than necessary.

- Entity *Purchase.java* does not (yet) contain named queries, as so far, the table *Purchase* is only written but not read

```
@Entity
@Table(name = "Purchase")
public class Purchase implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id private int purchaseID;
    private String firstname;
    private String lastname;
    private String creditCardNumber;
    private BigDecimal totalPrice;

    public String getCreditCardNumber() {...}
    public void setCreditCardNumber(String creditCardNumber) {...}
    public String getFirstname() {...}
    public void setFirstname(String firstname) {...}
    public String getLastname() {...}
    public void setLastname(String lastname) {...}
    public BigDecimal getTotalPrice() {...}
    public void setTotalPrice(BigDecimal totalPrice) {...}
}
```

Jakarta Persistence Query Language (JPQL) (1)

- When using JPA, the **Jakarta Persistence Query Language** is used
 - Very similar to SQL and is translated to native SQL during runtime
 - But JPQL works on objects (entities) instead of tables and columns
- Example 1: **Select all *Product* objects (entities)** from the database:

```
SELECT p FROM Product AS p
```

- JPQL always requires an alias for the entities that are requested (here *p*)
- The AS can be omitted and usually, the query is written as follows:

```
SELECT p FROM Product p
```

- Example 2: **Select all *Product* objects that have a specific description**

```
SELECT p FROM Product p WHERE p.description LIKE '%used%'
```

- This returns only the objects where the description contains *used*

SELECT p FROM Product AS p

The syntax can be understood as follows:

- We have a number of *Product* objects (corresponding to rows in table *Product* in the database) that are identified with *p*
- With *SELECT p* and without using any *WHERE* clause, all of them are selected

- Example 3: Use a **named parameter** for the description

```
SELECT p FROM Product p WHERE p.description LIKE :description
```

- Named parameters – here *description* – are prefixed with a colon (:)
- The value for *description* is specified before executing the query
- Very similar as prepared statements with the advantage that it uses names instead of '?' to identify the parameters
- When working with JPA, one should always use named parameters to avoid SQL injection problems

- Named query in entity *Product.java*

```
@NamedQuery(name = "Product.findByDescription", query = "SELECT p  
FROM Product p WHERE p.description LIKE :description")
```

- This uses JPQL syntax and delivers all *Product* objects with the description specified in the named parameter *description*
- The query is identified with name *Product.findByDescription*
- This name can be used by other classes to access the query

Marketplace Code – Entity Manager and *persistence.xml*

- The core JPA component to interact with the database is the **Entity Manager** (*EntityManager* class, provided by Jakarta EE)
- To configure the entity manager, the file *persistence.xml* is used
 - This file specifies **persistence units**, which are sets of entities that are managed by the same entity manager

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence ... version="3.0">
```

- Specifies a **name** for the persistence unit
- JTA means the creation / handling of the entity manager is done **automatically** by the application server

```
<persistence-unit name="marketplace" transaction-type="JTA">
```

- Specify the **data source** for database access
 - Corresponds to the handle that is configured in *web.xml*

```
<jta-data-source>java:global/marketplace</jta-data-source>
```

```
<shared-cache-mode>NONE</shared-cache-mode>
```

- Turn off **caching** (makes sure to always read current data from the DB)

```
</persistence-unit>
```

```
</persistence>
```

Persistence Units

Usually, one uses one persistence unit per database that is used by the application. The Marketplace application just uses one database, so only one persistence unit is used.

Persistence Version Details

For simplicity, attributes of the persistence element have been omitted in the slide above, as they are standard attributes generated by the IDE:

```
<persistence xmlns="https://jakarta.ee/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="https://jakarta.ee/xml/ns/persistence
  https://jakarta.ee/xml/ns/persistence/persistence_3_0.xsd"
  version="3.0">
```

Facade Classes for Database Access

- To provide easy database usage for the other classes (e.g., for the backing beans), the Marketplace application uses **facade classes**
 - *ProductFacade.java* for accessing table *Product*
 - *PurchaseFacade.java* for accessing table *Purchase*
 - They will be used **instead** of the currently used classes *ProductDatabase.java* and *PurchaseDatabase.java*
- In addition, an **abstract class** *AbstractFacade.java* is used as the superclass for both facade classes
 - The superclass provides methods that are used by all facade classes, e.g., to insert, update or delete rows in the database
- The facade classes are annotated with **@Stateless**
 - This turns them into **stateless EJBs**, managed by the application server
 - Stateless because Marketplace just uses **simple DB transactions** consisting of one method call and a single SELECT or INSERT statement
 - **A stateless EJB automatically handles transactions, so we must not actively begin / commit / rollback transactions**

@Stateless

As we have no complicated DB transactions that require multiple method calls of one of the facade classes (in our case, each DB «transaction» is just a single read or write access to the DB that is done with a single method call), **@Stateless** is the reasonable choice. If more complex transactions were involved, **@Stateful** would be used.

Marketplace Code – *ProductFacade.java*

```
@Stateless
public class ProductFacade extends AbstractFacade<Product> {

    @PersistenceContext(unitName = "marketplace")
    private EntityManager entityManager;

    public ProductFacade() {
        super(Product.class);
    }

    @Override
    protected EntityManager getEntityManager() {
        return entityManager;
    }

    public List<Product> findByDescription(String description) {
        Query query = entityManager.createNamedQuery(
            "Product.findByDescription");
        query.setParameter("description", "%" + description + "%");
        return query.getResultList();
    }
}
```

Instance variable for the entity manager

- Using **@PersistenceContext** in front of it injects the entity manager for the specified persistence unit (*marketplace*)

Returns the products for the specified description

- Uses the entity manager to create a named query object based on the named query defined in the *Product* entity (identified as *Product.findByDescription*)
- Sets the parameter with name *description* in the named query
- Executes the query and returns the list of products

© ZHAW / SoE / InIT – Marc Ren

Import Statements

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

```
import java.util.List;
import java.math.BigDecimal;
import java.util.ArrayList;
import jakarta.ejb.Stateless;
import jakarta.persistence.EntityManager;
import jakarta.persistence.PersistenceContext;
import jakarta.persistence.Query;
import ch.zhaw.securitylab.marketplace.common.model.Product;
```

Serializable

EJBs should implement the `Serializable` interface so instances can be stored in the file system if needed. As the superclass `AbstractFacade` implements this interface, `ProductFacade` is automatically also serializable.

```
@Stateless
public class PurchaseFacade extends AbstractFacade<Purchase> {
    @PersistenceContext(unitName = "marketplace")
    private EntityManager entityManager;

    public PurchaseFacade() { super(Purchase.class); }

    @Override
    protected EntityManager getEntityManager() {
        return entityManager;
    }
}
```

There's no *findBy...* method, as there's currently no read access needed to table *Purchase*

- But inserting a row into table *Purchase* is needed during checkout
 - For this, a method *create()* is provided by the superclass *AbstractFacade*:

```
public abstract class AbstractFacade<T> implements Serializable {
    ...
    public void create(T entity) {
        getEntityManager().persist(entity);
    }
    ...
}
```

The *persist()* method of the entity manager inserts a new row into the database, based on the entity that is passed as an argument

Marketplace Code – Using the Facade Classes

- Finally, we can **use the facade classes** where we used *ProductDatabase.java* and *PurchaseDatabase.java* before
 - Just like CDI beans, EJBs can be **injected**

```
public class SearchBacking implements Serializable {  
    @Inject private ProductFacade productFacade;  
    public String search() {  
        products = productFacade.findByDescription(searchString);  
        return "/view/public/search";  
    }  
}
```

```
public class CheckoutBacking implements Serializable {  
    @Inject private PurchaseFacade purchaseFacade;  
    @Inject private CartService cartService;  
    private Purchase purchase = new Purchase();  
    public String completePurchase() {  
        if (cartService.getCount() > 0) {  
            purchase.setTotalPrice(cartService.getTotalPrice());  
            purchaseFacade.create(purchase);  
            ...  
        }  
    }  
}
```

JPA and SQL Injection (1)

- As mentioned before, one positive side effect of using the JPA is that **SQL injection attacks don't work**



Search results for: `DVD%' UNION SELECT 1,2,CONCAT_WS(" - ",Username,Pbkdf2Hash),4,5 FROM UserInfo--`

No products match your search	PurchaseID	Firstname	Lastname	CreditCardNumber	TotalPrice
	4	Mickey	Mouse	1111 2222 3333 4444', -1000)--	5.95

- Reason: For the **SELECT** statement, we use a **named parameter** to specify the description to search for
 - With named parameters, **JPA escapes control characters** such as ' (quote)
 - Therefore, it's not possible to semantically change the query, which means SQL injection is not possible
- With the **INSERT** statement, JPA automatically creates the query **from the instance variables** of the *Purchase* entity
 - In this case, **control characters are also escaped**, which again prevents SQL injection

JPA and SQL Injection (2)

- But: It is possible to **use JPA «in a wrong way»** such that SQL injection attacks can be carried out

- Example 1: **JPQL query** is built using **string concatenation**

```
public List<Product> findByDescription(String description) {
    Query query = entityManager.createQuery("SELECT p FROM Product p WHERE
        p.description LIKE '%" + description + "%'");
    return query.getResultList();
}
```

- An SQL injection proof of concept can easily be done by searching for ***no-match%' OR '%' = '***
 - All products are returned, which clearly shows that SQL injection took place (no escaping of ')

no-match%' OR '%' = ' Search

Search results for: no-match%' OR '%' = '

Description	Price (CHF)	
DVD Life of Brian - used, some scratches but still works	5.95	Add to Cart
Ferrari F50 - red, 43000 km, no accidents	250000.00	Add to Cart
Commodore C64 - used, the best computer ever built	444.95	Add to Cart
Printed Software-Security script - brand new	10.95	Add to Cart

JPQL and UNION

Note that JPQL does not support UNION in general (although some JPA implementations do), so one often cannot using the typical UNION SELECT attack to access additional data.

- Example 2: A native query with string concatenation is used

```
public List<Product> findByDescription(String description) {
    Query query = entityManager.createNativeQuery("SELECT * FROM Product
        WHERE Description LIKE '%" + description + "%'");
    List<Object[]> results = query.getResultList();
    List<Product> products = new ArrayList<>();
    Product product;
    for (Object[] result : results) { // copy from results to products }
        return products;
    }
}
```

- In this case, SQL injection can be done in the same way as before, e.g.

- *DVD%' UNION SELECT 1,2,CONCAT_WS(' - ',Username,Pbkdf2Hash),4,5 FROM UserInfo--*

Description
DVD Life of Brian - used, some scratches but still works
alice - PBKDF2WithHmacSHA512:100000:ABuSvRyYjADUKum2UMh4DxU5Gz5zqMeOxLQWo2PvD4D9XNqUQd4
bob - PBKDF2WithHmacSHA512:100000:1CT79kds35vQdLMMY2yglWWtH4RXu9ZVvRBAAMPAR6AJVQBKEEds
donald - PBKDF2WithHmacSHA512:100000:yeuX8n4vQ3u9tHtXUJap44Q42onN8t8TerDAswnYs1HO4ZM2Gd
/SspL2eR0h+AmNG+QKp+s=
john - PBKDF2WithHmacSHA512:100000:F08Me8JumCYafawV0d4RYByQD888IKUQM2aLc3D8dvU7qWGL+E
/uZfN88RE5Equney7qyGn4p7NYr
luke - PBKDF2WithHmacSHA512:100000:Qb2d9p9H4ZKneEZba8EvhKWoA4+Ye7yP1HkCgTqK9RHJJAAP
robin - PBKDF2WithHmacSHA512:100000:vdEYPmHLE5seqc12RXEJZmCwJ5Fda4VD1JU0cmk3P1Gz2+ju3lrmwO4z
anony - PBKDF2WithHmacSHA512:100000:GOZhnvVf7czLpu4hAyp07+DRKJCDh4tH2Z73/JKodn7m9p29yW
4LEH7H0q28uuaX4g8PFE5h

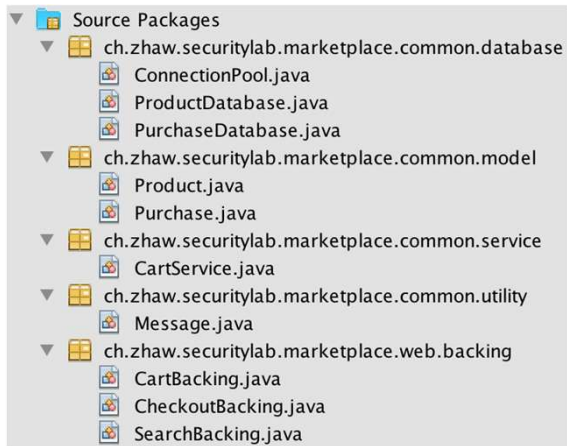
- Conclusions: To prevent SQL injection with JPA, make sure to use named parameters and don't use native queries!

Native Queries

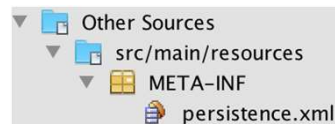
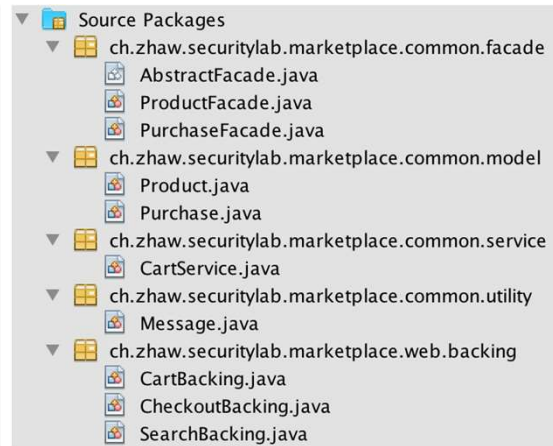
When using native queries, the code gets more complicated, as we are no longer getting entities back from the SELECT statements, but just «rows» that are represented as a list of arrays of *Objects* (`List<Object[]>`). Therefore, this list must additionally be converted to a list of *Product* objects.

Marketplace – Project Organization before / after JPA

Before moving to JPA



After moving to JPA



Model Classes

Just like before, we have *Product.java* and *Purchase.java*, but before, they were simple JavaBeans and now they are JPA Entities.

Access Control and Authentication

Marketplace V06

The extensions of this section are integrated in Marketplace_v06.

Marketplace – Admin Area Extension (1)

- The Marketplace application is extended with an **admin area**
- This area allows **sales and marketing personnel** of the company which operates the Marketplace application to perform administrative tasks:
 - They can **view the purchases** made by customers (i.e., the content of the table *Purchase*)
 - **They can remove a purchase** from the table *Purchase*
- To get to the admin area, a **new button** is added to the search page:

Welcome to Marketplace v06

To search for products, enter any search string below and click the Search button.

Search results for:

No products match your search

Marketplace – Admin Area Extension (2)

- Clicking the *Admin area* button opens the admin page
 - Shows all entries in the table *Purchase*

Admin Area

Purchases:

First Name	Last Name	Credit Card Number	Total Price (CHF)	
Ferrari	Driver	1111 2222 3333 4444	250000.00	Remove purchase
C64	Freak	1234 5678 9012 3456	444.95	Remove purchase
Script	Lover	5555 6666 7777 8888	10.95	Remove purchase

[Return to search page](#) [Logout](#)

- Removing a purchase removes the entry from the database and the list

Admin Area

Purchases:

First Name	Last Name	Credit Card Number	Total Price (CHF)	
C64	Freak	1234 5678 9012 3456	444.95	Remove purchase
Script	Lover	5555 6666 7777 8888	10.95	Remove purchase

[Return to search page](#) [Logout](#)

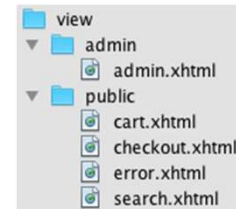
Marketplace – Admin Area Extension (3)

What is needed to extend the application with the new functionality?

- A new Facelet *admin.xhtml* for the admin page, which lists all purchases and that contains buttons to remove a specific purchase:

```
<h:column>
  <h:form>
    <h:commandButton value="Remove purchase"
      action="#{adminPurchaseBacking.removePurchase(purchase)}" />
  </h:form>
</h:column>
```

- This Facelet is placed in the subdirectory */view/admin/*
 - This allows protecting access to the Facelet using declarative security mechanisms

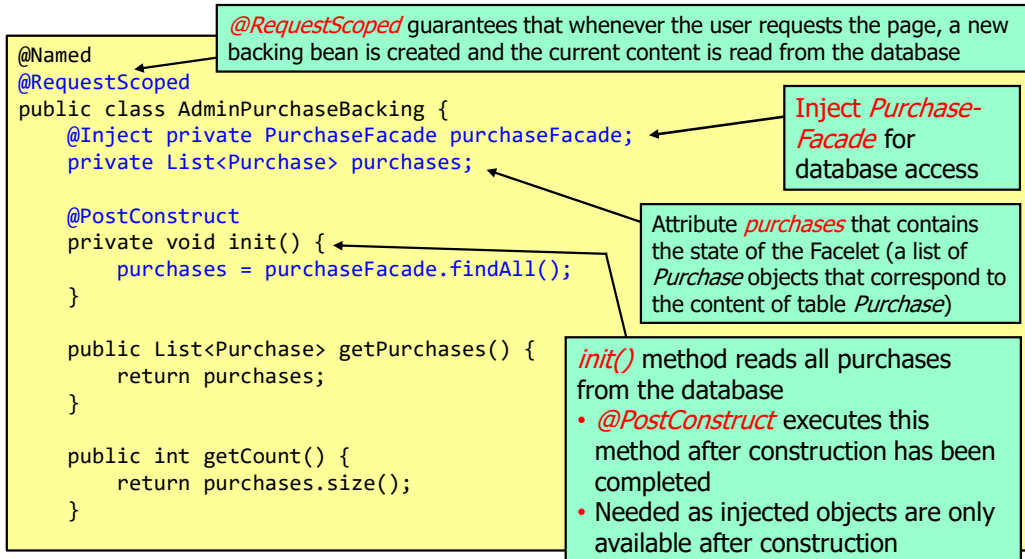


- A new button in the Facelet *search.xhtml* to go to the admin area:

```
<h:button value="Admin area" outcome="/view/admin/admin" />
```

Marketplace – Admin Area Extension (4)

- A new backing bean *AdminPurchaseBacking.java* that contains state information and provides functionality for the new Facelet:



Marketplace – Admin Area Extension (5)

```
public String removePurchase(Purchase purchase) {  
    purchaseFacade.remove(purchase);  
  
    purchases.remove(purchase);  
  
    return "/view/admin/admin";  
}
```

Removes the selected *Purchase* object from the database

Also removes the object from the list

The return value instructs the Faces Servlet to serve *admin.xhtml* to the browser

- The methods to **fetch all purchases** and to **remove a purchase** are not provided by *PurchaseFacade*, but by its superclass *AbstractFacade*:

```
public List<T> findAll() {  
    CriteriaQuery criteriaQuery = getEntityManager()  
        .getCriteriaBuilder().createQuery();  
    criteriaQuery.select(criteriaQuery.from(entityClass));  
    return getEntityManager().createQuery(criteriaQuery).getResultList();  
}
```

Based on any entity class (e.g., *Purchase*), this method builds and executes the SELECT query to fetch all corresponding objects from the database

- E.g., *Purchase*: **SELECT p FROM Purchase p**
- This could also be done with an explicit query in *Purchase*, but this is the generic way

```
public void remove(T entity) {  
    getEntityManager().remove(getEntityManager().merge(entity));  
}
```

Standard JPA way to **remove any object (entity)** from the database

- First call **merge** to make sure the database contains the current state of the object
- Then call **remove** to perform the actual removal of the object

merge before remove

For instance, if a foreign key in the entity was changed and cascading delete is used, it's important to first update the state of the entity in the database (so the new foreign key is used) before deleting it.

See also: <https://stackoverflow.com/questions/16086822/when-using-jpa-entitymanager-why-do-you-have-to-merge-before-you-remove>

Access Control with Declarative Security (1)

- With these extensions, the admin functionality is available as intended, but **unfortunately to every user...**
- We therefore need some **access control mechanisms** to enforce that only sales and marketing personnel can access the admin area
- We could implement this **programmatically**:
 - Define users and their rights and store this in the database
 - Implement a login mechanism to authenticate a user
 - Store the identity of the authenticated user in the session
 - Whenever a user accesses the admin area, check whether he is authenticated and has the necessary rights
- However, the problem with this approach is that it's very easy to make **security-relevant programming mistakes**
 - E.g., forgetting to consistently check the access rights with every request

Access Control with Declarative Security (2)

- To overcome this problem, Jakarta EE (and other technologies) provides a **built-in declarative security mechanism to handle access control**
 - «Declarative» means that the access control mechanism **can mainly be configured** and only a small amount of actual program code is needed
 - Using this is easier and more secure than with programmatic security
- **In Jakarta EE, this mechanism is based on role-based access control:**
 - **Define roles** that exist within the application
 - Specify **which resources can be accessed with which roles**
 - **Define users** that exist within the application and **assign one or more roles** to each user
 - As a result, a user gets access to all resources **that are accessible with his roles**
- Defining the roles and specifying **which roles are allowed to access what resources** is done in *web.xml*

Marketplace – Define Access Control Rules in *web.xml* (1)

1. Define the roles to be used within the application
 - Role *sales* for sales personnel, role *marketing* for marketing personnel
2. Specify that only roles *sales* and *marketing* can access */faces/view/admin/**

```
<security-role>
  <description>Sales Personnel</description>
  <role-name>sales</role-name>
</security-role>
<security-role>
  <description>Marketing Personnel</description>
  <role-name>marketing</role-name>
</security-role>

<security-constraint>
  <web-resource-collection>
    <web-resource-name>Admin Area</web-resource-name>
    <url-pattern>/faces/view/admin/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>sales</role-name>
    <role-name>marketing</role-name>
  </auth-constraint>
  <user-data-constraint>
    <transport-guarantee>CONFIDENTIAL</transport-guarantee>
  </user-data-constraint>
</security-constraint>
```

Define the roles *sales* and *marketing*, *<description>* is purely informative

<security-constraint> is used to specify access restrictions in Jakarta EE applications

Access to resources in */faces/view/admin/* is only allowed for users with role *sales* or *marketing*

This is needed to make sure the *url-pattern* can only be accessed over HTTPS (explanation see notes)

<auth-constraint>

Using the special role name *** in an *auth-constraint* means all roles described in the deployment descriptor.

<security-role>

Usually, the application will also function correctly without the *<security-role>* elements. Nevertheless, you should include them to make sure that it conforms “to the standard” and that no runtime problems occur with any application server.

<user-data-constraint>* in *<security-constraint>

It seems to be surprising that the *<user-data-constraint>* has to be included in this *<security-constraint>*, as we have already configured another *<security-constraint>* in *web.xml* that enforces HTTPS in general:

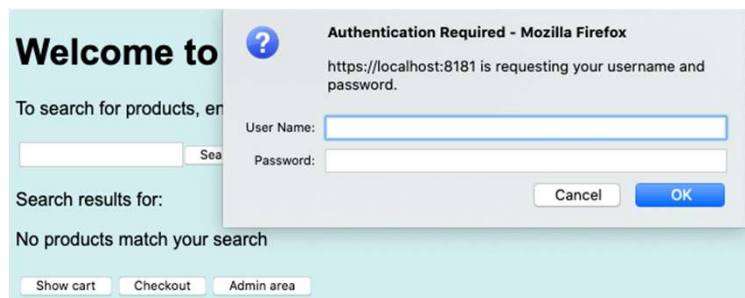
```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>HTTPS everywhere</web-resource-name>
    <url-pattern>/*</url-pattern>
  </web-resource-collection>
  <user-data-constraint>
    <transport-guarantee>CONFIDENTIAL</transport-guarantee>
  </user-data-constraint>
</security-constraint>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

The reason that *<user-data-constraint>* must be included again is because of the way it is decided what *<security-constraint>*s to apply when processing a request. The rule is that only the constraint(s) with the best match according to the *<url-pattern>* are considered. This means that when accessing anything below */faces/view/admin/*, only the constraint illustrated in the slide above is considered and not the general one for HTTPS access, because the *<url-pattern>* */faces/view/admin/** is the better match than */**. Therefore, HTTPS would not be enforced without the additional *<user-data-constraint>*, which is why we added it once more.

The approach with «the best URL match» seems unnecessarily complex. However, there are situations where this is beneficial, especially when you want to use different access control rules for subdirectories (see lab exercise). For example, assume there's a resource */faces/view/admin/product/addproduct.xhtml* which should be available to users with role *productmanager* but not to users with roles *sales* or *marketing*. This can be done by specifying an additional security constraint for resource */faces/view/admin/product/addproduct.xhtml* (or */faces/view/admin/product/**), which is accessible only for role *productmanager*. Now, when accessing */faces/view/admin/product/addproduct.xhtml*, only users with role *productmanager* are granted access because the new security constraint provides the better URL match than the one for */faces/view/admin/**.

Marketplace – Define Access Control Rules in *web.xml* (2)

- Running the application and trying to **access the admin area** results in the following:



- What happened?
 - We have **tried to access a resource** that is only accessible by authenticated users that have a specific role
 - As we are currently not authenticated, the web application **initiates user authentication**, using the default mechanism configured in the application server

Username, Passwords and Roles

- What we need next are **users** that can be authenticated and that get specific **roles** after successful authentication

- In Marketplace, we define the following usernames, passwords and roles:

Username	Password	Role
alice	rabbit	sales
bob	patrick	burgerman
donald	daisy	productmanager
john	wildwest	sales
luke	force	productmanager
robin	arrow	marketing
snoopy	woodstock	productmanager

- This means that if **alice** logs in successfully using password **rabbit**, she gets assigned the role **sales**
 - Which means **alice** should get access to the admin area

Jakarta EE Security API (1)

- Jakarta EE provides two different approaches to implement and configure user authentication
- Up to Java EE 7, this was based on so called **realms**
 - A realm is **storage location** that contains usernames, passwords and roles
 - Storage locations: database, file, LDAP server,...
 - The realms are configured in the underlying **application server**
 - Different server products provide different realms, but the popular realms are usually supported in most products
 - Very easy to use, but requires application server-specific configuration and cannot easily be extended (inflexible)
- Jakarta EE 8 introduced a new approach: **Jakarta EE Security API**
 - Application server-independent and extensible, but also a bit more complicated to use
 - Realms are still supported, but the Jakarta EE Security API will most likely be the approach of choice in the future

Realms

Although the different server products offer different realms, they are often similar. For instance, Tomcat, JBoss and Payara/GlassFish all support the JDBCRealm to store credentials in the database, but the configuration is application server-specific.

- The three main components of the Jakarta EE Security API are the **Identity Store**, the **Authentication Mechanism** and the **Security Context**
- An **identity store** provides access to any «storage system» where credentials are stored
 - Not limited to «classic» credential storages such as a database or LDAP server, but can be «anything», e.g., an in-memory storage
 - To implement a specific identity store, one has to implement the interface *IdentityStore*

```
public interface IdentityStore {  
    CredentialValidationResult validate(Credential credential);  
}
```

- To facilitate usage of some **typical identity stores**, Jakarta EE includes corresponding implementations
 - E.g., *DatabaseIdentityStore*, *LdapIdentityStore*

- The **authentication mechanism** defines the authentication method that is used by the web application
 - To implement a specific authentication mechanism, one has to implement the interface *HTTPAuthenticationMechanism*

```
public interface HttpAuthenticationMechanism {  
    AuthenticationStatus validateRequest(HttpServletRequest request,  
        HttpServletResponse response, HttpContext  
        httpMessageContext) throws AuthenticationException;  
}
```

- To facilitate usage of some **typical authentication mechanisms**, Jakarta EE includes corresponding implementations
 - E.g., *BasicAuthenticationMechanism*, *FormAuthenticationMechanism*, *CustomFormAuthenticationMechanism*
- The **security context** allows developers to do authentication and checks to grant or deny access to application resources programmatically
 - Provided by an object of type *SecurityContext*

Authentication Mechanism

Different mechanism means, for instance, that credentials can be included in a standard request header (such as *Authorization*), that a login FORM is used for authentication which results in an authenticated session, and so on.

Storing Passwords

- In the Marketplace application, we store usernames, passwords and roles in the database
- Passwords should be stored in the database in a secure way
 - Don't store them in plaintext, as attacks such as SQL injection or DB server compromise directly provide access to them
 - Don't store direct hashes of passwords as this may allow cracking the passwords using a precompiled (dictionary) attack
- The best option is to use a function that is specifically optimized for password hashing, e.g., PBKDF2, bcrypt and scrypt
 - Basically, they use salt and several rounds of hashing to make cracking efforts difficult
- Here, we are using PBKDF2 (= Password-Based Key Derivation Function 2)
 - The main reasons for using PBKDF2 is that it is well-established, widely used, and supported by the Jakarta EE Security API
 - It's not only intended to hash passwords, but also to derive a secret key from a password and a salt value

Different Options to store the Password

Option 1: Store the password in plaintext

- That's obviously a bad choice, as an attacker who manages to access the database (e.g., SQL injection, DB server compromise, DB admin) directly gets the passwords

Option 2: Store a hash of the password, e.g., with SHA-512

- Hash = SHA-512(password)
- Advantage: Protects from the attacks above
- Disadvantage: An attacker who gets access to the hashes can try to crack them with a precompiled (dictionary) attack

Option 3: Combine salt & password and hash once, e.g., with SHA-512

- Hash = SHA-512(salt|password)
- Advantage: Protects from precompiled (dictionary) attacks
- Disadvantage: Hash functions such as SHA-512 are optimized for speed, therefore the attacker can still test huge amounts of passwords
- Using a state-of-the-art graphics card allows to test several 100 million passwords per second

Option 4: Use several rounds of hashing, e.g., 5'000

- Hash = SHA-512(SHA-512(SHA-512(salt|password)|salt|password)...))
- That's much better than a single round as it significantly increases the work the attacker has to do to crack the passwords, and this would be a reasonable solution

Option 5: Use a function that is specifically optimized for password hashing => «slow hash functions»

- Several such functions exist, notably PBKDF2, bcrypt and scrypt
- Basically they use salt and several rounds of hashing – so nothing new here
- But in addition, they are designed to make brute force attacks on hashes that use special hardware much more difficult
 - E.g., by consuming relatively large amounts of memory and using operations that have limited performance gain on special hardware such as graphics cards

Utility Class *Pbkdf2.java*

- To generate the PBKDF2-hashed passwords from the plaintext passwords, we use *Pbkdf2.java*:

```
public class Pbkdf2 {
    private static final int SALT_SIZE_BYTES = 64;
    private static final int HASH_SIZE_BYTES = 32;
    private static final int ITERATIONS = 100000;
    private static final String ALGORITHM = "PBKDF2WithHmacSHA512";

    private static byte[] createSalt() {
        byte[] salt = new byte[SALT_SIZE_BYTES];
        SecureRandom random = new SecureRandom();
        random.nextBytes(salt);
        return salt;
    }

    private static byte[] computeHash(String password, byte[] salt) {
        PBEKeySpec keySpec = new PBEKeySpec(password.toCharArray(), salt,
            ITERATIONS, HASH_SIZE_BYTES * 8);
        SecretKeyFactory keyFactory = SecretKeyFactory.getInstance(ALGORITHM);
        return keyFactory.generateSecret(keySpec).getEncoded();
    }

    public static void main(String args[]) {
        byte[] salt = createSalt();
        byte[] hash = computeHash(args[0], salt);
        System.out.println(ALGORITHM + ":" + ITERATIONS + ":" +
            Base64.getEncoder().encodeToString(salt) + ":" +
            Base64.getEncoder().encodeToString(hash));
    }
}
```

PBKDF2 parameters
(typical values that are considered to be secure)

Creates new salt value

Computes PBKDF2 hash from a password string and a salt value

main method: Gets a password as a command line argument, generates a new salt value, computes the corresponding PBKDF2 hash, and returns salt, hash and metadata in a standard format

Import Statements

The class above requires the following import statements:

```
import java.security.SecureRandom;
import javax.crypto.SecretKeyFactory;
import javax.crypto.spec.PBEKeySpec;
import java.util.Base64;
```

PBKDF2

PBKDF2 uses internally a pseudorandom function of two parameters, typically a MAC function. In our case, we are using HMAC based on SHA-512 (i.e., PBKDF2WithHmacSha512).

SecretKeyFactory

It seems a bit strange that *computeHash* uses a *SecretKeyFactory*. But as PBKDF2 is basically a function to create a key from a password, this makes perfectly sense.

Generating PBKDF2 Hashes and Database Tables

- *Pbkdf2.java* can now be used to create the password hashes
 - The output uses the format *algorithm:iterations:salt:hash*

```
$ java PBKDF2 rabbit
PBKDF2WithHmacSHA512:100000:A6kJSvRyVyADUIKum2UMhk00xU5kGz5zqMa0k1LQrWo2
PVD4D9rXNq1UQIdoujWnJKKPhJEfLvYKXFDHk6PC8A==:be0X2N7mcg/m0oxk79SP1EBYJtU
AR+Hfky9NyjILawk=
```

- Usernames, password hashes and roles are stored in **two tables**

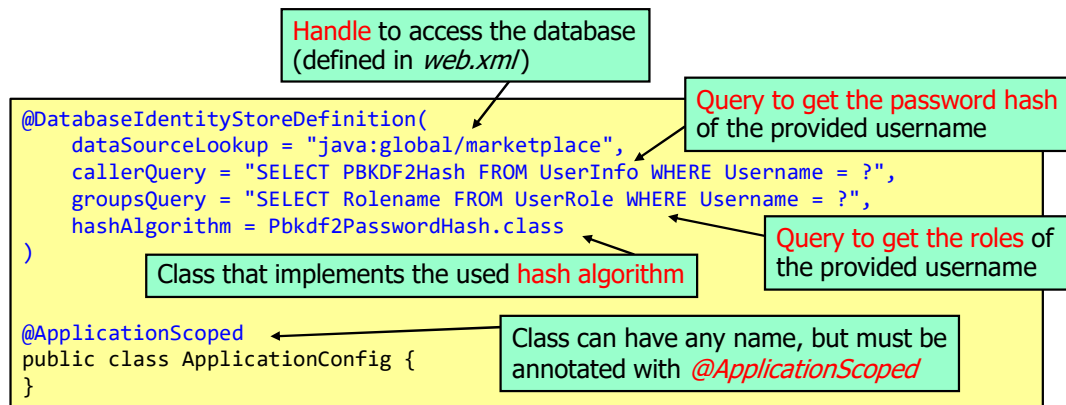
<i>UserInfo</i>		<i>UserRole</i>	
Username	Pbkdf2Hash	Username	Rolename
alice	PBKDF2WithHmacSHA512:100000:A6kJSvRyVyADUIKu...	alice	sales
bob	PBKDF2WithHmacSHA512:100000:UC7b0kdr35kVDsflA...	bob	burgerman
donald	PBKDF2WithHmacSHA512:100000:yeukX8nk0vQJszfihT...	donald	productmanager
john	PBKDF2WithHmacSHA512:100000:FX6Me8UamCrYiafw...	john	sales
luke	PBKDF2WithHmacSHA512:100000:Qb2r0ep0HI42KmEZ...	luke	productmanager
robin	PBKDF2WithHmacSHA512:100000:vx0EYPm/hLEs5aqs...	robin	marketing
snoopy	PBKDF2WithHmacSHA512:100000:GOZNnVcF/czUpuHx...	snoopy	productmanager

UserInfo* and *UserRole

One could also use just one table with *Username*, *PBKDF2Hash* and *UserRole*. However, this would only allow that a user can have one role. With two tables, a user can have multiple roles, so using two tables is the more flexible approach (although we are currently using only one role per user).

Marketplace – Configuring the Identity Store

- As the credentials are stored in the database, we can use the default *DatabaseIdentityStore* provided by Jakarta EE
 - To configure this, the annotation *@DatabaseIdentityStoreDefinition* can be used
 - This annotation must be placed in any class that is annotated with *@ApplicationScoped*



@ApplicationScoped

Using this annotation has the effect that exactly one object of the class is created when the application is started and exists until the application is terminated.

SQL Injection

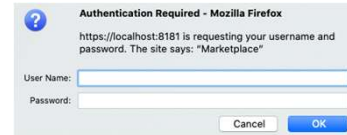
Looking at the syntax of the two queries, one can see that prepared statements are used, so SQL injection is not an issue.

Authentication Mechanism

- In most cases, one of the following authentication mechanisms is used:

- **HTTP BASIC authentication**

- To initiate authentication, a dialog box opens where username and password are entered
- Username and password are sent to the web application in an **HTTP authorization header**
- Every subsequent request also includes the **authorization header**, which grants the user continuous access to the protected area



- **FORM-based authentication combined with sessions**

- To initiate authentication, a login form is used, username and password are entered, and sent to the web application in **GET or POST parameters**
- If authentication is successful, the web application stores information about the current user **in the session of the user**
- Every subsequent request will include the **corresponding session-ID** of the user, which grants the user continuous access to the protected area

Authentication Security

Of course, both all three variants should only be used over HTTPS. While HTTP Digest Authentication avoids easy interception of the password by an attacker if HTTP is used, it still can easily be attacked by, e.g., a man in the middle. To do so, the attacker simply modifies the **HTTP 401 Unauthorized** response from the server and adapts it such that BASIC authentication is used by the browser.

HTTP Digest Authentication

This is also supported: The browser does not send the password but a hash over the password and a challenge from the server.

CLIENT-CERT Authentication

Jakarta EE also supports CLIENT-CERT authentication: Use a certificate for user authentication (always uses HTTPS / HTTP over TLS).

When using certificates for user authentication, a realm (e.g., UserDataRealm or JDBCRealm) is still needed. Since no user password is needed, the password can be set to null and the X.500 name of the subject in the certificate is used as the username.

Marketplace – HTTP BASIC Authentication (1)

- Using **HTTP BASIC authentication** is done with the built in authentication mechanism *BasicAuthenticationMechanism*
 - To configure this, the annotation *@BasicAuthenticationMechanismDefinition* can be used
 - This annotation must also be placed in any class that is annotated with *@ApplicationScoped* (usually, the same as for the identity store is used)

```
@DatabaseIdentityStoreDefinition(...)
```

```
@BasicAuthenticationMechanismDefinition(  
    realmName="Marketplace"  
)
```

```
@ApplicationScoped  
public class ApplicationConfig {  
}  
}
```

realmName can be optionally used to define a string that is displayed in the authentication dialogue box


Marketplace – HTTP BASIC Authentication (2)

The following happens if a user **tries to access the admin area** (*/faces/view/admin/**):

- The application sees that a **protected resource** should be accessed
 - As there's a *<security-constraint>* with an *<auth-constraint>*
- Assuming the request **does not include credentials** (no authorization header), HTTP BASIC authentication is initiated
 - According to *@BasicAuthenticationMechanismDefinition*
- When receiving username and password, the application **verifies** them using the configuration in *@DatabaseIdentityStoreDefinition*
 - It first gets the hashed password from the database using the *callerQuery*...
`callerQuery = "SELECT PBKDF2Hash FROM UserInfo WHERE Username = ?"`
 - ...then it hashes the received password with PBKDF2 (according to *hashAlgorithm*) and compares the hash with the hashed password from the database...
`hashAlgorithm = Pbkdf2PasswordHash.class`
 - ...if this is correct it gets the roles(s) of the user using *groupsQuery*
`groupsQuery = "SELECT Rolename FROM UserRole WHERE Username = ?"`
- If one of these roles is **marketing** or **sales**, the user gets access to the admin area

Marketplace – HTTP BASIC Authentication (3)

- Accessing the admin area now causes the following:
 - The reason why the dialogue box is opened is because the server responded with an HTTP response with status code 401



- Entering credentials sends the same request again, but includes the credentials in a HTTP authorization header (base64-encoded)
 - E.g., *Authorization: Basic YWxpY2U6cmFiYmI0*

- The correct credentials allow access to the admin area, clicking *Cancel* shows the HTTP 401 Unauthorized message



HTTP Basic Authentication

When trying to access the admin area, the server replies with an HTTP 401 “Unauthorized” message, which causes the browser to show the login dialogue box. Entering wrong credentials causes the dialogue box to pop up again and again, clicking *Cancel* displays the HTTP 401 Unauthorized status message.

HTTP BASIC Authentication – Limitations

- While easy to use and supported by every browser, HTTP BASIC authentication has one major drawback:
 - Once the credentials have been entered, the browser includes the corresponding HTTP authorization header in every subsequent request to the admin area
- The reason for this is that HTTP BASIC authentication was designed to provide the user with an «authenticated session» even if the web application does not support sessions
 - The only way to achieve this is by including the HTTP authorization header in each request as otherwise, the user would have to enter the credentials with each single access to the protected area
- The consequence of this is that there is no simple way to log out the user or log in as a different user unless we are telling the browser to forget the credentials – which can only be done by closing the browser

Logging out

Well, you can enforce a new login dialogue box programmatically: write a servlet that sends an *HTTP 401 Unauthorized* status message, but that's not declarative security and therefore more complex. In addition, even if the user wants to log out, he'd get again the login dialogue box, which he'd have to "click away" with "Cancel", which is ugly from a usability point of view.

Marketplace – FORM-based Authentication

- FORM-based authentication combined with sessions is the **preferred authentication method today**
 - Does not have the limitations of HTTP BASIC authentication
 - The credentials are sent only once: during actual authentication
 - Can be integrated into the look-and-feel of the application
- Using **FORM-based authentication** is done with the built in authentication mechanism *FormAuthenticationMechanism*
 - To configure this, the annotation *@FormAuthenticationMechanismDefinition* can be used

```
@FormAuthenticationMechanismDefinition(  
    loginToContinue = @LoginToContinue(  
        loginPage="/faces/view/public/login.xhtml",  
        errorPage="/faces/view/public/login-error.xhtml"  
    )  
)
```

Within the annotation, the attribute *loginToContinue* must be defined

The *login page* and the *login error page* to use

Resources for Login / Login Error Pages

Note that one can use arbitrary resources for the login and error pages, e.g., HTML, XHTML, JSPs, Servlets etc.

- The login page must contain a login form that must use **specific values** for the action and the parameter names for username and password
 - *j_security_check*, *j_username*, *j_password*
 - These values are defined by Jakarta EE and must be used so the generated request is interpreted as a **login request**

```
<p>Please log in to continue.</p>
<form action="j_security_check" method="post">
  <h:panelGrid columns="2">
    <h:outputLabel value="Username:" />
    <input type="text" name="j_username" />
    <h:outputLabel value="Password:" />
    <input type="password" name="j_password" />
    <h:outputText value="" />
    <h:panelGrid columns="2">
      <input type="submit" name="submit" value="Login" />
      <h:button value="Cancel" outcome="/view/public/search" />
    </h:panelGrid>
  </h:panelGrid>
</form>
```

It is mandatory to use these values for the action and the names of the username and password parameters

Request to *j_security_check*

The request can use GET or POST, both works. But passwords should never be transmitted in GET requests.

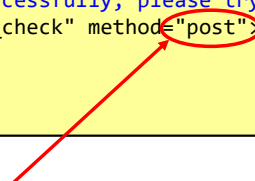
<form> instead of **<h:form>**

Note that a standard form is used here, not a JSF form. That's necessary because the login request must use the standard identifiers as described above and must not be changed by the Faces Servlet in any way.

- The login error page is sent to the browser if login fails, this can basically contain any content
 - E.g., again a login page with an additional message

```
<div id="header">
  <h1>Login Failed</h1>
</div>

<div id="loginForm">
  <p>You did not log in successfully, please try again to continue.</p>
  <form action="j_security_check" method="post">
    ...
  </form>
</div>
```



- The login request uses POST – does this matter?

Marketplace – FORM-based Authentication

- Trying to access the admin area now causes the following:
- Successful login allows access to the admin area, wrong login shows the login error page

Login

Please log in to continue.

Username:

Password: 

Admin Area

Purchases:

First Name	Last Name	Credit Card
Ferrari	Driver	1111 2222 33
C64	Freak	1234 5678 90
Script	Lover	5555 6666 77

Login Failed

You did not log in successfully, please try again to continue.

Username:

Password:

[Return to search page](#) [Logout](#)

What happens in the background?

- Assuming the user is not logged in and tries to access the admin area, the web application sends the login page to the browser
- If the application receives the correct credentials, it stores information about the user and his roles in the session and sends a redirection response (status code 302) to the browser, which points to the originally requested page (admin area)
- The browser accesses the admin area again and the web application allows access (as the session is now an authenticated session)
- Likewise, future accesses to the admin area are also permitted

Logging Out

- The web application stores information about the logged in user and corresponding roles **in the session** of the user
- To log out a user, the method *logout* of the interface *HttpServletRequest* is used
 - An object implementing this interface is always available to servlets (the Faces Servlet in JSF) and contains information about the received request and the current session
 - When calling the method, the currently authenticated user is removed from the session → we have again an anonymous session

- To provide the logout functionality, we implement a new backing bean *AuthenticationBacking.java*
 - This bean will be extended with additional functionality related to authentication later

```

@Named
@SessionScoped
public class AuthenticationBacking implements Serializable {
    private static final long serialVersionUID = 1L;

    public String logout() {
        FacesContext facesContext = FacesContext.getCurrentInstance();
        HttpServletRequest request = (HttpServletRequest)
            facesContext.getExternalContext().getRequest();
        try {
            request.logout();
        } catch (ServletException e) {
            // Do nothing
        }
        Message.setMessage("You have been logged off");
        return "/view/public/search";
    }
}
    
```

Standard way to get the *HttpServletRequest* object of the Faces Servlet (boilerplate code)

Do the *logout*

Set a message and instruct the Faces Servlet to *serve search.xhtml* to the browser (*Message* class see notes below)

Class *Message.java*

```

package ch.zhaw.securitylab.marketplace.util;

import jakarta.faces.application.FacesMessage;
import jakarta.faces.context.FacesContext;

public class Message {
    public static void setMessage(String message) {
        FacesContext facesContext = FacesContext.getCurrentInstance();
        FacesMessage facesMessage = new FacesMessage(message);
        facesContext.addMessage(null, facesMessage);
    }
}
    
```

@SessionScoped

This bean could also be *@RequestScoped* as it (currently) does not contain any state. However, there's also no reason create a new backing bean whenever a Facelet uses this bean, so *@SessionScoped* is a reasonable choice for performance reasons. In addition, the bean will be extended with state (attributes) later.

FacesContext

The faces context is a core object when 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.

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

- We also add a new button *Logout* to the admin page

Admin Area

Purchases:

First Name	Last Name	Credit Card Number	Total Price (CHF)	
Script	Lover	5555 6666 7777 8888	10.95	<input type="button" value="Remove purchase"/>

- Clicking the button calls the *logout* method in the *AuthenticationBacking* bean:

```
<h:commandButton value="Logout"
  action="#{authenticationBacking.logout}" />
```

Marketplace – Logging Out

- Clicking the button results in **logging out the user** and displaying the search page including a message:

Welcome to Marketplace v06

You have been logged off

To search for products, enter any search string below and click the Search button.

- Trying to access the admin area again presents the user the **login page**
 - This is reasonable, as **the application has «forgotten» about the previously logged in user**

Login

Please log in to continue.

Username:

Password:

Access Control and Authentication – Summary

- The basis for access control are *<security-constraint>*s in *web.xml*
 - This allows to specify which roles can access which resources
- **Username**s, **password**s and **roles** can be stored at arbitrary locations
 - Often, the database is used for this, which requires two tables with usernames, passwords and roles
 - To configure how the application can access these credentials in the database, the annotation *@DatabaseIdentityStoreDefinition* can be used
- To specify the authentication method, one can use the annotations *@BasicAuthenticationMechanismDefinition* or *@FormAuthenticationMechanismDefinition*
 - Often, FORM-based authentication combined with sessions is used, the corresponding form must use the correct values for the action and the parameter names
- **Authentication is automatically requested** by the web application when a non-authenticated user accesses a protected resource

Note that most of this was done with declarative security, we have implemented only little code (login form, logout functionality)

Access Control and Authentication with Programmatic Security

Access Control and Authentication could also be implemented using a completely own mechanism enforced directly in the code, but this is much more error-prone and also more complex. Therefore, whenever possible, you should use the declarative mechanisms provided by Jakarta EE.