8. Java Web Application Security – Appendix

Prof. Dr. Marc Rennhard
Institut für angewandte Informationstechnologie InIT
ZHAW School of Engineering
rema@zhaw.ch

Appendix ESAPI
(for personal education)

Information about Validation Errors (1)

- To get more information why validation fails, ESAPI offers variations of the `getValidXY()` methods that take an additional parameter
  - The parameter has the type `ValidationErrorList`
  - The passed object gets filled with the errors that happen during validation

- To demonstrate this, we use a second variant of `validateCCNumber()` in Validation.java:

```
public static String validateCCNumber(String input,
    ValidationErrorList errors) {

    Validator validator = ESAPI.validator();
    String clean = validator.getValidCreditCard("CreditCard",
                   input.trim(), false, errors);
    if (errors.isEmpty()) {
        return clean;
    } else {
        return null;
    }
}
```

Errors that occur during validation are inserted into a ValidationErrorList. An empty list indicates successful validation (no exception is thrown).

Information about Validation Errors (2)

- We modify PurchaseServlet.java such that the new method is used
  - In addition to validating the credit card, any errors that occur during validation are written to the Tomcat log
    - This helps, e.g., to detect erroneous validation rules

```
// Validate the received data
ValidationErrorList errors = new ValidationErrorList();
if ((firstName = Validation.validatePersonName(firstName)) == null) {
    ...
} else if ((ccNumber = Validation.validateCCNumber(ccNumber, errors)) ==
           null) {
    message = "Please insert a valid credit card number";
    url = "/checkout";
    for (Object vo : errors.errors()) {
        log("Error (validateCCNumber): " +
            ((ValidationException) vo).getMessage());
    }
} else {
    // Store purchase in DB
```

Write the error messages to the Tomcat log

Information about Validation Errors (3)

- Entering an invalid credit card number…

First name:	John
Last name:	Wayne
Credit card number:	1111 2222 3333 444

- …results in the following log entry:

```
INFO: PurchaseServlet: Error (validateCCNumber): CreditCard:
Invalid input. Please conform to regex ^(\d{4}[- ]?){3}\d{4}$
with a maximum length of 19
```

  - This also shows the regex used by the built-in method to check credit card number

Appendix JSF
(for personal education)

JavaServer Faces

- JavaServer Faces is *the* Java EE component for developing user interfaces in web applications (view layer)
  - First version 1.0: 2004 (based on JSP)
  - Current version 2.2: 2013 (independent of JSP)

- JSF consists of:
  - Several prefabricated UI components
  - An event-driven programming model

- This appendix provides the following:
  - A (very) brief introduction to JSF using a simple application
  - How to develop secure Java EE web applications with JSF
    → This is done by re-implementing the Marketplace application (final servlet/JSP-based version) with JSF

A simple JSF Application (1)

- A user enters his cedentials and clicks the Login button
- A welcome screen with the username is displayed

A simple JSF Application (2)

Ingredients of this application:

- Two pages for the logon and welcome screens: `index.xhtml` and `welcome.xhtml`
  - Since JSF 2.0, they are also called Facelets
  - JSF pages must follow the xhtml format (basically html that is propper XML)
- A bean `UserBean.java` that manages the data (a so called managed or named bean)
- A deployment descriptor `web.xml`

During JSF development, the developer primarily implements
- facelets (xhtml files) for the view and
- models (managed beans) for business logic and data management

Interaction between the facelets and the models is handled by JSF

A simple JSF Application – `index.xhtml`

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">
  <h:head>
    <title>Welcome</title>
  </h:head>
  <h:body>
    <h:form>
      <h3>Please enter your name and password.</h3>
      <table>
        <tr>
          <td>Name:</td>
          <td><h:inputText value="#{user.name}"/></td>
        </tr>

        <tr>
          <td>Password:</td>
          <td><h:inputSecret value="#{user.password}"/></td>
        </tr>
      </table>
      <p><h:commandButton value="Login" action="welcome"/></p>
    </h:form>
  </h:body>
</html>
```

Import JSF tags (accessed with prefix `h:`)

Using a JSF tag, here inputText for a text field to input data

Input data is linked to a bean property → stored in the bean when the user submits the form

Next facelet to display when submitting the form (welcome.xhtml)

A simple JSF Application – `UserBean.java`

```
import java.io.Serializable;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;

@ManagedBean(name="user")
@SessionScoped
@SuppressWarnings("serial")
public class UserBean implements Serializable {
  private String name;
  private String password;

  public String getName() { return name; }
  public void setName(String newValue) { name = newValue; }

  public String getPassword() { return password; }
  public void setPassword(String newValue) { password = newValue; }
}
```

Identify as managed bean with name user

The bean lives during the entire session (other options RequestScoped, Viewscoped etc.)

Properties of the bean (corresponds to the entered displayed data in index.xhtml)

Getter and setter methods for the properties

A simple JSF Application – `welcome.xhtml`

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">
  <h:head>
    <title>Welcome</title>
  </h:head>
  <h:body>
    <h3>Welcome to JavaServer Faces, #{user.name}!</h3>
  </h:body>
</html>
```

Output data is linked to a bean property → data stored in the bean is displayed

A simple JSF Application – `web.xhtml`

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
  http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
  id="WebApp_ID" version="3.0">


  <servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>/faces/*</url-pattern>
  </servlet-mapping>
  <welcome-file-list>
    <welcome-file>faces/index.xhtml</welcome-file>
  </welcome-file-list>
  <context-param>
    <param-name>javax.faces.PROJECT_STAGE</param-name>
    <param-value>Development</param-value>
  </context-param>
</web-app>
```

A JSF application consists of a single Faces servlet, which is provided by JSF

Welcome file, all facelets are accessed below /faces/ (per default)

Useful for more detailed error messages during development, can be changed to Production

A simple JSF Application (3)

What happens in details (assume a fresh session):

- The user accesses `index.xhtml`
  - The server (the Faces servlet) creates the page for the user and also creates a UserBean object (as the bean `user` is used in `index.xhtml`)
- The user enters name and password and clicks Login
  - The browser sends a POST request (a commandButton always sends a POST request in JSF)
  - The server stores the received data in the UserBean object (as this is specified with #{user.name} and #{user.password})
  - The server creates the next page (`welcome.xhtml`, this is specified in the action attribute in `index.html` and included in the POST request)
  - The server includes the data from the UserBean in the page (as this is specified with #{user.name})
- If the user reloads `index.xhtml` subsequently, the input fields in the created page include the previously entered data
  - Because the bean still exists (SessionScoped)...
  - ...and data stored in beans is always automatically included in the fields

Marketplace with JSF

Zurich University of Applied Sciences

Just like with the servlet/JSP-based Marketplace application, we look at the various security-relevant details that must be addressed:

- Suppressing detailed error messages
- Data sanitation
- Secure database access
- Authentication
- Access Control
- Secure communication
- Session handling
- Input validation
- Cross-Site Request Forgery prevention

Marketplace with JSF – Libraries

- The JSF version of the Marketplace applications uses two additional Java EE components that are not supported by Tomcat out of the box
  - Remember that Tomcat is basically just a servlet runner
  - A fully compliant Java EE application such as GlassFish includes these libraries out of the box

- 1st required component: JSF reference implementation
  - http://javaserverfaces.java.net

- 2nd required component: Bean Validation Framework (see slides about input validation)
  - http://validator.hibernate.org

Marketplace with JSF – Project Organization (1)

- jsf_marketplace_v07
  - JAX-WS Web Services
  - Deployment Descriptor: jsf_ma
  - Java Resources
    - src
      - beans
      - data
      - managedbeans
        - CartBean.java
        - CheckoutBean.java
        - LoginBean.java
        - PurchasesBean.java
        - SearchBean.java
      - util
      - validation
        - CreditCardValidator.java
    - Libraries
  - JavaScript Resources

beans (non-managed) and data packages from servlet/JSP-based Marketplace, unchanged

New managed beans used by JSF

Utility classes from servlet/JSP-based, unchanged

Class that performs credit card number validation

Marketplace with JSF – Project Organization (2)

- WebContent
  - admin
    - login.xhtml
    - purchases.xhtml
  - META-INF
    - context.xml
    - MANIFEST.MF
  - resources
    - css
      - styles.css
  - WEB-INF
    - lib
    - faces-config.xml
    - web.xml
  - cart.xhtml
  - checkout.xhtml
  - error_401.html
  - error_403.html
  - error_404.html
  - index.xhtml
  - products.xhtml

New xhtml pages used by JSF (admin area)

Tomcat configuration, unchanged

Style sheet, added for HTML5-compliance

JSF-specific configurations

Deployment descriptor, adapted from servlet/JSP Marketplace

New xhtml pages used by JSF and "normal" html pages for standard error handling

Standard Error Handling

- JSF allows to specify the detail-level of error messages in web.xml
  - Value `Development` delivers more information than `Production`

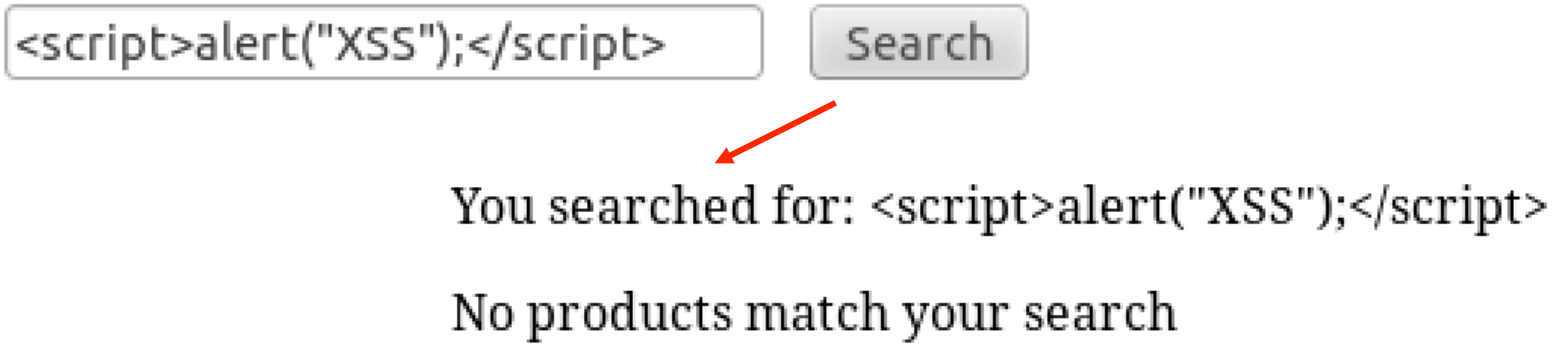
```
<context-param>
  <param-name>javax.faces.PROJECT_STAGE</param-name>
  <param-value>Development</param-value>
</context-param>
```

- However, `Production` simply outputs the container-specific message
  - Therefore, they should be suppressed using Java EE standard error handling, as done in the servlet/JSP-based Marketplace application

```
<error-page>
  <error-code>404</error-code>
  <location>/error_404.html</location>
</error-page>
...
```

Data Sanitation (1)

- Out of the box, JSF encodes (sanitises) critical characters in a similar way as when using the `out` tag in JSP files
  - <, >, and " are encoded to <, >, and &quot
  - As a result, JSF provides resistance against XSS / HTML injection per default



- Page source:

```
<p>You searched for: &lt;script&gt;alert(&quot;XSS&quot;);&lt;/script&gt;</p>
```

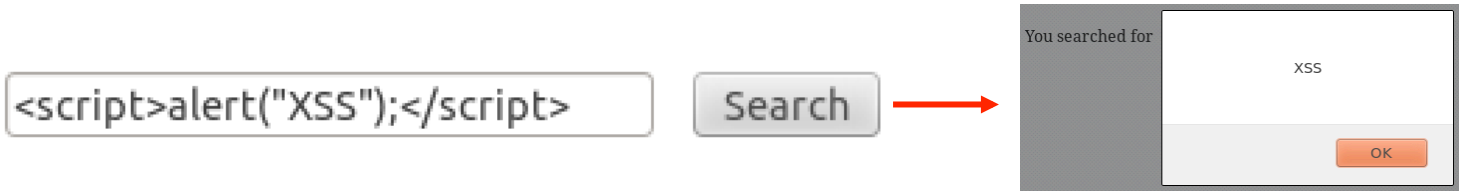
Data Sanitation (2)

- Source code in `products.xhtml`:

```
<p>You searched for: #{search.searchString}</p>
```

- Data sanitation can be deactivated by using the `outputText` tag and setting the `escape` attribute to false
  - But of course, this enables script/HTML injection attacks!

```
<p><h:outputText value="You searched for #{search.searchString}"
                 escape="false"/></p>
```



- Rule: Never set the `escape` attribute to false unless you have clear reasons to do so!
  - Per default, the `escape` attribute is true

Secure Database Access

- JSF is completely unrelated to database access

- Secure database access should be done in exactly the same way as with the servlet/JSP-based Marketplace application
  - Using Prepared Statements prevents SQL injection attacks, whether you use the statements directly or a ORM framework such as hibernate

- The difference is only where database access is implemented:
  - Servlet/JSP: In the servlet, which in our case used the classes in package data
  - JSF: In the managed bean, which uses the same classes in package data

Secure Database Access – Example

```
<h:form><table><tr>
  <td><h:inputText value="#{search.searchString}"/></td>
  <td><h:commandButton value="Search"
                       action="#{search.search}"/></td>
</tr></table></h:form>
```

When submitting the search form in index.xhtml, the search method in managed bean search is called

```
@ManagedBean(name="search") @SessionScoped
public class SearchBean implements Serializable {
  private String searchString;
  private List<Product> products;
  private String message;

  // Getter and setter for properties searchString etc. and
  // further methods omitted

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

The search method uses ProductDB to perform the search

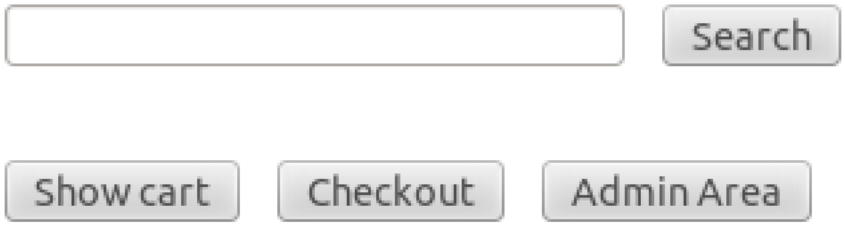
The returned string is used to determine the next page (action attribute in index.html) to display (here: products.xhtml)

Access Control and Authentication

- JSF is completely unrelated to authentication and access control
- Options to handle authentication and access control
  - Using declarative (container-based) security offered by Java EE
  - Using additional programmatic security features offered by Java EE
  - Using a 3rd party security framework for Java EE (e.g. Spring Security)
  - Using a homegrown approach
- Here: We use the declarative and programmatic features offered by Java EE
  - In the same way as we have done in the final version of the servlet/JSP-based Marketplace application
  - One main difference: Access Control Annotations can only be used with servlets, which means we cannot use them with JSF
    - We therefore use security-constraints in `web.xml`
- The process and parts of the code are illustrated on the following slides

Access Control and Authentication – Login Button

- The Admin Area button on `index.xhtml` calls the `adminArea` method in the `login` managed bean




```
<h:form>
  <h:commandButton value="Admin Area" action="#{login.adminArea}"/>
</h:form>
```

Access Control and Authentication – `adminArea` method

- If the user has role sales or marketing, he is forwarded to `admin/purchases.xhtml`
  - Just like with the servlet/JSP application, administrative resources are placed below `/admin/` to easily define security-constraints to restrict access (see `web.xml` later)
- Otherwise, he is forwarded to the login page (`admin/login.xhtml`)

```
@ManagedBean(name="login") @SessionScoped
public class LoginBean implements Serializable {
  @ManagedProperty(value="#{search}")
  private SearchBean search;
  private String username;
  private String password;

  public String adminArea() {
    FacesContext context = FacesContext.getCurrentInstance();
    HttpServletRequest request = (HttpServletRequest)
                                 context.getExternalContext().getRequest();
    if (request.isUserInRole("sales") || request.isUserInRole("marketing")) {
      return "/admin/purchases";
    } else {
      return "/admin/login?faces-redirect=true";
    }
  }

  // Other methods ommitted
}
```

Access Control and Authentication – `admin/login.xhtml`

- This page presents the user a form to enter the credentials
- Clicking the Login button calls the `login` method in the `login` managed bean

Please enter your username and password to continue.

Username

Password

Login

```
<h:form>
  <table>
    <tr>
      <td align="right">Username</td>
      <td><h:inputText value="#{login.username}"/></td>
    </tr>
    <tr>
      <td align="right">Password</td>
      <td><h:inputSecret value="#{login.password}"/></td>
    </tr>
    <tr>
      <td><h:commandButton value="Login" action="#{login.login}"/></td>
    </tr>
  </table>
</h:form>
```

Access Control and Authentication – `login` method

- The `login` method performs the programmatic login (including salt)
- If successful, the user is forwarded to `admin/purchases.xhtml`, otherwise to `index.xhtml`

```
@ManagedBean(name="login") @SessionScoped
public class LoginBean implements Serializable {
  @ManagedProperty(value="#{search}")
  private SearchBean search;
  private String username; private String password;

  public String login() {
    String salt = LoginDB.getSalt(username);
    try {
      String digest = Crypto.printHex(Crypto.computeSHA1(password + salt));
      FacesContext context = FacesContext.getCurrentInstance();
      HttpServletRequest request = (HttpServletRequest)
                                   context.getExternalContext().getRequest();
      request.login(username, digest);
      return "/admin/purchases";
    } catch (Exception e) {
      search.setMessage("Login failed.");
      return "/index";
    }
  }
  // Other methods ommitted
}
```

Access Control and Authentication – admin/purchases.xhtml

- This page lists the purchases, the Complete Purchase link is only available for sales persons; the page uses the managed bean purchases

```
<h:dataTable value="#{purchases.purchases}" var="purchase"
             rendered="#{purchases.numberOfPurchases != 0}">
  <h:column>
    <f:facet name="header">First Name</f:facet>
    #{purchase.firstName}
  </h:column>
  <h:column>
    <f:facet name="header">Last Name</f:facet>
    #{purchase.lastName}
  </h:column>
  ...
</h:column>
  <h:column rendered="#{request.isUserInRole('sales')}">
  <h:form>
    <h:commandLink value="Complete Purchase"
                   action="#{purchases.completePurchase(purchase.id)}"/>
    </h:form>
  </h:column>
</h:dataTable>
```

purchases method of purchases bean returns a List of purchase objects that contain the table data

The rendered attribute is used to display a column or not

Call completePurchase in the purchases managed bean to complete a purchase

Access Control and Authentication – `purchases` bean (1)

- This managed bean `purchases` handles the data of the purchases and interacts with the database

```
@ManagedBean(name="purchases") @ViewScoped
public class PurchasesBean implements Serializable {
  private List<Purchase> purchases; private String message;

  public PurchasesBean() {
    purchases();
  }

  public List<Purchase> getPurchases() {
    return purchases;
  }

  public String getMessage() {
    String returnMessage = message;
    message = "";
    return returnMessage;
  }

  ...
```

Constructor, fetch the purchases from the database

Access Control and Authentication – `purchases` bean (2)

```
  ...

  public int getNumberOfPurchases() {
    if (purchases == null || purchases.size() == 0) {
      return 0;
    } else {
      return purchases.size();
    }
  }

  public String completePurchase(int purchaseId) {

    // Get the purchaseId to delete and delete it
    if (PurchaseDB.delete(purchaseId) != 1) {
      message = "A problem occurred when completing the purchase";
    } else {
      message = "Purchase completed";
    }
    return "/admin/purchases";
  }

  private void purchases() {
    purchases = PurchaseDB.getPurchases();
  }
}
```

Complete a purchase and display again /admin/purchases.xhtml

Fetch the purchases from the database

Access Control and Authentication – Logout Button on `admin/purchases.xhtml`

- The Logout button on admin/purchases.xhtml calls the `logout` method in the `login` managed bean, which logs out the user and shows `index.xhtml`

Return to search page | Logout and return to search page

```
<h:form>
  <h:commandButton value="Logout and return to search page"
                   action="#{login.logout}" />
</h:form>
```

```
public String logout() {
  FacesContext context = FacesContext.getCurrentInstance();
  HttpServletRequest request = (HttpServletRequest)
                               context.getExternalContext().getRequest();
  try {
    request.logout();
  } catch (Exception e) {}
  search.setMessage("You have been logged off.");
  return "/index";
}
```

Access Control and Secure Communication – `web.xml` (1)

- Finally, security-constraints in `web.xml` are needed
  - In addition, secure communication is enforced for selected areas

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>Admin Area</web-resource-name>
    <url-pattern>/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>
...
```

Allow access to resources in /admin/* (login.xhtml and purchases.xhtml) only to roles sales and marketing and only over TLS

Access Control and Secure Communication – `web.xml` (2)

```
...
<security-constraint>
  <web-resource-collection>
    <web-resource-name>Purchase</web-resource-name>
    <url-pattern>/checkout.jsf</url-pattern>
  </web-resource-collection>
    <user-data-constraint>
      <transport-guarantee>CONFIDENTIAL</transport-guarantee>
    </user-data-constraint>
  </security-constraint>
</security-constraint>
<security-constraint>
  <web-resource-collection>
    <web-resource-name>Login page</web-resource-name>
    <url-pattern>/admin/login.jsf</url-pattern>
  </web-resource-collection>
  <user-data-constraint>
    <transport-guarantee>CONFIDENTIAL</transport-guarantee>
  </user-data-constraint>
</security-constraint>
...
```

Allow access to checkout.jsf only over TLS

Allow access /admin/login.jsf to everyone, but only over TLS

Access Control and Secure Communication – `web.xml` (3)

```
...
<security-constraint>
  <web-resource-collection>
    <web-resource-name>restricted methods</web-resource-name>
    <url-pattern>/*</url-pattern>
    <url-pattern>/admin/*</url-pattern>
    <url-pattern>/checkout.jsf</url-pattern>
    <http-method-omission>GET</http-method>
    <http-method-omission>POST</http-method>
  </web-resource-collection>
  <auth-constraint/>
</security-constraint>
```

Allow only GET and POST

Access Control, Authentication and Secure Communication – Final Remarks

- Session (ID) handling is controlled by Java EE/Tomcat and works exactly the same as analyzed with the servlet/JSP-based application
  - But remember; this only works if the declarative / programmatic security features of Java EE – as we have done here – are used

- Security-constraints are used with servlets/JSPs and with JSF to restrict access to resources
  - With servlets/JSPs, they are used to specify which servlets can be access be whom and how
  - With JSF, they are used to specify which xhtml files can be accessed

- With the servlet/JSP-base application, we had to move most JSP files below /WEB-INF to make sure they cannot be accessed directly
  - With JSF, this is not necessary (at least in this case), as the user is allowed to access all xhtml files directly

Input Validation

- JSF provides two ways to perform input validation:
  - Using JSF validators within JSF pages
  - By using the Bean Validation Framework (which is part of Java EE)

- Both variants are illustrated using the checkout process
  - The validation rules are the same as with the servlet/JSP application

Please insert the following information to complete your purchase:

First name:

Last name:

Credit card number:

Purchase

Standard JSF Validation – First Name

- First name must match the regex `^[a-zA-Z']{2,32}$`
- This can directly be integrated into `checkout.xhtml`

```
<h:form rendered="#{cart.count != 0}">
  <h:panelGrid columns="3">
    First name:
      <h:inputText id="firstname" value="#{checkout.firstName}"
          label="First name" validatorMessage="Please insert a
          valid first name (between 2 and 32 characters)">
        <f:validateRegex pattern="^[a-zA-Z']{2,32}$"/>
      </h:inputText>
      <h:message for="firstname" errorClass="errors"/>
    ...

  </h:panelGrid>
  <h:commandButton value="Purchase" action="#{checkout.purchase}"/>
</h:form>
```

validatorMeassge: Message to display if validation fails

validateRegex: The regex used for validation

Prints the message if validation fails

Custom JSF Validation – Credit Card Number (1)

- Credit card number must be valid
- Can not be expressed in a regex → write a custom JSF validator
  - This slide shows usage of the custom validator in the JSF page, the next one shows the implementation of the validator itself

```
<h:form rendered="#{cart.count != 0}">
  <h:panelGrid columns="3">
    ...

    Credit card number:
    <h:inputText id="ccnumber" value="#{checkout.ccNumber}"
        label="Credit card number">
      <f:validator validatorId="validation.CreditCardValidator"/>
    </h:inputText>
    <h:message for="ccnumber" errorClass="errors"/>

  </h:panelGrid>
  <h:commandButton value="Purchase" action="#{checkout.purchase}"/>
</h:form>
```

Use the custom Validator with name validation.CreditCardValidator

Prints the message (which is defined in the custom validator implementation) if validation fails

Custom JSF Validation – Credit Card Number (2)

A custom validator implements the Validator interface and implements the validate method

```
@FacesValidator("validation.CreditCardValidator")
public class CreditCardValidator implements Validator {


  public void validate(FacesContext context, UIComponent component,
                       Object value) {
    String rawValue = (String) value;
    String numberValue = rawValue.replaceAll("\\D", ""); // remove non-digits
    if(!checkRawFormat(rawValue) || !luhnCheck(numberValue) ) {
      FacesMessage message = new FacesMessage("Please insert a valid credit
                                              card number (16 digits)");
      message.setSeverity(FacesMessage.SEVERITY_ERROR);
      throw new ValidatorException(message);
    }
  }

  private static boolean checkRawFormat(String number) {
    return number.matches("^\\d{4}[ ]?\\d{4}[ ]?\\d{4}[ ]?\\d{4}$");
  }
  private static boolean luhnCheck(String cardNumber) {
    // returns whether the card number checksum is correct (see below)
  }
}
```

- If validation succeeds, the method should returns normally
- If validation fails, throw a ValidatorException, which can include a message, which is included in the JSF page using the h:message tag

Bean Validation – Last Name (1)

- Bean validation is not part of JSF, but of Java EE (since Java EE 6)
- Bean validation can be well-integrated with JSF
  - Advantage: if a managed bean is manipulated via multiple JSF pages, validation must only be implemented in one place (the bean)
- Last name must match the regex `^[a-zA-Z']{2,32}$`

```
@ManagedBean(name="checkout") @SessionScoped
public class CheckoutBean implements Serializable {
  ...
  private String firstName;
  @Pattern(regexp="^[a-zA-Z']{2,32}$",
           message="Please insert a valid last name (between 2 and
                    32 characters)")
  private String lastName;
  private String ccNumber;
  private String message;

  ...
}
```

Use @Pattern to match a regex, applies to the next property (lastName)

Use this error message if validation fails

Bean Validation – Last Name (2)

- In `checkout.xhtml`, no validation checks are included for the last name, only a h:message tag to print the error message if validation fails

```
<h:form rendered="#{cart.count != 0}">
  <h:panelGrid columns="3">
    ...

    Last name:
    <h:inputText id="lastname" value="#{checkout.lastName}"
        label="Last name"/>
    <h:message for="lastname" errorClass="errors"/>

    ...
  </h:panelGrid>
  <h:commandButton value="Purchase" action="#{checkout.purchase}"/>
</h:form>
```

Prints the message if bean validation fails

Effect of JSF Validation and Bean Validation – and Limitations

https://localhost:8443/marketplace/checkout.jsf

Checkout

Please insert the following information to complete your purchase:

First name:	a	*Please insert a valid first name (between 2 and 32 characters)*
Last name:	ABCDefgh/	*Please insert a valid last name (between 2 and 32 characters)*
Credit card number:	1111 2222 3333 4445	*Please insert a valid credit card number (16 digits)*

Purchase

Return to search page | Show cart

- JSF and bean validation both have a limitation: They only validate, but do not perform any normalization
  - That's OK if you do not consider encoded content a problem (or if the allowed characters do not allow any meaningful encoding)
  - If you need normalization, you can use ESAPI, e.g. by writing custom JSF validators that themselves use ESAPI

Cross Site Request Forgery Protection (1)

- For POST requests, JSF offers CSRF protection since 2.0
  - This is done with a hidden field named javax.faces.ViewState
    - This viewstate is basically computed over the page displayed to the user
  - Since JSF 2.2, the value is encrypted per default and is considered to be secure enough for CSRF protection
  - This is activated per default, so you don't have to configure anything

- In addition, JSF offers a CSRF token since 2.2
  - This allows also to protect GET request
  - Can also be used for POSTs, but this is not necessary due to viewstate
  - It is necessary to configure the resources where the CSRF token should be used in `faces-config.xml`

Cross Site Request Forgery Protection (2)

- As an example, we configure JSF to use use the CSRF token when accessing the cart (`cart.xhtml`)
- Necessary configuration in `faces-config.xml`:
  - For multiple resources, use multiple url-pattern tags

```
<?xml version='1.0' encoding='UTF-8'?>
<faces-config version="2.2" xmlns="http://java.sun.com/xml/ns/
  javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
  http://xmlns.jcp.org/xml/ns/javaee/web-facesconfig_2_2.xsd">
  <protected-views>
    <url-pattern>/cart.xhtml</url-pattern>
  </protected-views>
</faces-config>
```

Cross Site Request Forgery Protection (3)

Search

Show cart | Checkout | Admin Area

- Resulting page `index.xhtml`:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"><head id="j_idt2">
  <title>Marketplace</title><link type="text/css" rel="stylesheet" href="/marketplace/javax.faces.resource/styles.css.jsf?ln=css" /></head><bc

<h1>Welcome to the Marketplace</h1>

<p><font color="red"></font></p>

<p>To search for products, enter any search string below and click the Search button</p>
<form id="j_idt7" name="j_idt7" method="post" action="/marketplace/index.jsf" enctype="application/x-www-form-urlencoded">
<input type="hidden" name="j_idt7" value="j_idt7" />

  <table>
    <tr>
      <td><input type="text" name="j_idt7:j_idt9" /></td>
      <td><input type="submit" name="j_idt7:j_idt11" value="Search" /></td>
    </tr>
  </table><input type="hidden" name="javax.faces.ViewState" id="j_id1:javax.faces.ViewState:0" value="1321604271034931438:3704877409165595279"
</form>

<br />

<table>
  <tr>
    <td><input type="button" onclick="window.location.href='/marketplace/cart.jsf?javax.faces.Token=1392551797297'; return false;" value="Show
    </td>
    <td><input type="button" onclick="window.location.href='/marketplace/checkout.jsf'; return false;" value="Checkout" />
    </td>
    <td>
<form id="j_idt18" name="j_idt18" method="post" action="/marketplace/index.jsf" enctype="application/x-www-form-urlencoded">
<input type="hidden" name="j_idt18" value="j_idt18" />
<input type="submit" name="j_idt18:j_idt19" value="Admin Area" /><input type="hidden" name="javax.faces.ViewState" id="j_id1:javax.faces.View
</form>
    </td>
  </tr>
</table></body>
</html>
```

Hidden viewstate field with POSTs

CSRF token in GET request for cart

Security Features Servlet/JSP vs. JSF

	Servlet/JSP	JSF
Standard error handling	Configure standard error pages in `web.xml`	
Data sanitation	Use `c:out` tag in JSP file	Works per default
Secure database access	Use prepared statements or a ORM framework that itself uses prepared statements	
Authentication, Access Control, Secure Communication	Use declarative/programmatic security offered by Java EE (or 3rd party security framework or homegrown approach)	
Session Handling	Correctly done by Tomcat/Java EE if declarative/ programmatic security offered by Java EE is used	
Input Validation	Use ESAPI (or another 3rd party library)	Use JSF Validation or Bean Validation (or ESAPI if normalization is necessary)
CSRF Protection	Use a 3rd party library or implement an own token-based approach	Use JSF built-in CSRF protection