

Developing Secure Traditional Web Applications – Part 3/3

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

Session Handling

Marketplace V07

The extensions of this section are integrated in Marketplace_v07.

Session Handling and Cookies (1)

- Cookies were introduced to solve the **session handling** problem in the Web as HTTP itself is stateless
 - Beyond this, cookies can also be used for other purposes, e.g., to **re-recognize users during subsequent visits** or to **track users across multiple web sites** (e.g., for personalized advertising)
 - Here we only look at session handling
- **Session handling with cookies** typically works as follows:
 - If the web application gets a request which is not associated with a known session, it **creates a new session and a corresponding session ID**
 - The session ID is sent to the browser in the HTTP response, using a **Set-Cookie response header**
 - **The browser receives the session ID and stores it**
 - **In all following requests to the web application, the browser includes the session ID in a Cookie header**
 - As a result, the web application **associates these requests with the correct session**

Session Handling and Cookies (2)

- Example cookie set by a web application:

```
Set-Cookie: session-id=28A468D70FF4118B066EA51; expires=Fri, 23-Dec-2025 11:09:37 GMT; Domain=www.myhost.com; Path=/Marketplace; Secure; HttpOnly
```

- Explanation:

- `session-id=28A468D70FF4118B066EA51`: The session ID chosen by the web application to identify the session
 - This session ID will be included by the browser in subsequent requests
- `expires`: Cookies with an expiry date are kept until this date even after closing the browser (persistent cookies), if no expiry date is used, the cookie is deleted when closing the browser (session cookies)
 - When cookies are used for session handling, don't use an expiry date
- `Domain, Path`: Specifies when to send the cookie to the web application, here: Any request to resources below `www.myhost.com/Marketplace/`
- `Secure flag`: Only send the cookie over HTTPS
- `HttpOnly flag`: JavaScript cannot access the cookie (via `document.cookie`)

Exam

Some best practices with respect to cookie usage:

- Use them only if necessary, and think twice before using them beyond pure session tracking purposes. Cookies have a somewhat bad reputation due to significant tracking abuse years ago (before the times of privacy policies).
- Be careful when using persistent cookies for authentication purposes. This may be reasonable from some services such as social media, but should never be used for services where money is involved (e-business). They will remain on the disk and are easily accessible by intruders

Domain

This attribute works as follows (from <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie>):

- Identifies the host to which the cookie will be sent. If omitted, defaults to the host of the current document URL, not including subdomains.
- Contrary to earlier specifications, leading dots in domain names (`.example.com`) are ignored.
- Multiple host/domain values are not allowed, but if a domain is specified, then subdomains are always included.

This means that if the domain attribute is not used, the cookie will only be included in requests to the host that is currently used (e.g., `www.myhost.com`). In practice, many websites use the domain of the host (e.g., `myhost.com`) in the `Set-Cookie` header, as this will include the cookie to all subdomains as well (e.g., in this case, to `myhost.com`, `www.myhost.com`, `www2.myhost.com` etc.). This is reasonable as all these hostnames are typically in control of the same organization.

Session Handling in Jakarta EE / Payara (1)

- **Session handling is critical** in web applications and a lot can go wrong
 - Therefore, we are analyzing in detail **whether Jakarta EE / Payara uses a secure session handling mechanism**
- Restarting the browser and requesting http://localhost:8080/Marketplace_v06/ causes a redirection to HTTPS and sets a cookie as follows:
 - *Set-Cookie: JSESSIONID=ce00603ae279008c541002776386; Path=/Marketplace_v06; Secure; HttpOnly*
 - The session ID has a length of **14 bytes (112 Bits)**
 - First two bytes are time-dependent and the rest can be assumed to be «random» (no vulnerabilities have been reported)
 - This means that the session ID is certainly «**long and random enough**» in the sense that an attacker won't be able to hijack a session of another user by correctly guessing the session ID
 - **HttpOnly flag is used** (used per default in Payara), which is good as this means that in the case of an XSS attack, the JavaScript code used by the attacker cannot access the cookie (via *document.cookie*)
 - **Secure flag is used**, which is good because when using HTTPS and setting a Cookie over HTTPS, the browser should never send the cookie over HTTP, as this allows an attacker to hijack the (possibly sensitive) session

OWASP Top Ten

Vulnerabilities related to session handling are all related to OWASP Top Ten Nr. 2: “Broken Authentication” – so they are rated as highly critical!

Secure Flag

A cookie that is set with the secure flag will only be sent over HTTPS by the browser and never over HTTP.

Domain omitted

As the domain is omitted here, the cookie will only be included in requests to localhost, i.e., to the host from which the initial resource was requested by the browser (see notes on previous slide).

- When logging in, a new cookie is set, and the session ID changes
 - Set-Cookie: *JSESSIONID=ce87c0ab194613dc42d1d743abef; Path=/Marketplace_v06; Secure; HttpOnly*
 - This change of the session ID is paramount, mainly for two reasons:
 - It makes session fixation attacks less effective, because it prevents that an attacker gets access to the authenticated session of the victim
 - If a web application first uses HTTP and only switches to HTTPS before login and providing access to the secure area, this prevents that an attacker who sniffs the session ID over HTTP can use it to hijack the authenticated session
 - For the web application, it's still the same session, but it's now identified with a different session ID
- Important: the session ID is only changed if one uses the built-in Jakarta EE access control mechanisms as discussed before
 - *<security-constraint>*, *<auth-constraint>*, *FormAuthenticationMechanism*,...
 - If one uses an own login approach with own access control handling, Jakarta EE won't recognize logins and won't change the session ID
 - In this case, you have to make sure the session ID is changed programmatically

Session Handling in Jakarta EE / Payara (3)

- When logging out, a new cookie is set, and the session ID changes again
 - The reason is that we use the *logout* method
 - When calling the method, the currently authenticated user is removed from the session and a new session ID is used to identify the session
 - From a security perspective, this change of the session ID would not be necessary (no threat is prevented by this)
- When having an authenticated session over HTTPS and accessing a resource over HTTP, the browser does not include the session ID
 - Because it was set with the Secure flag, so it must not be transmitted over HTTP
 - When getting the HTTP request, the web application redirects the request to HTTPS (configured in *web.xml*) and the HTTPS request then includes the session ID and the session continues normally
- To summarize: When using Jakarta EE together with Payara, session handling is **automatically done in a secure way**
 - When using other technologies, always verify this as done here!
 - If you ever have to implement your own session handling mechanism, then use the approach illustrated here as a blueprint!

<session-config> Element in *web.xml* (1)

- Besides <session-timeout>, the <session-config> element in *web.xml* allows two additional configurations
- <cookie-config> allows specifying the usage of the *HttpOnly* and *Secure* flag
 - In Payara, usage of both flags is enabled per default, so – assuming these settings have not been changed – this configuration is not absolutely required
 - However, enabling the flags in *web.xml* means they will always be used by the web application, independent of the actual settings in the underlying application server → **good idea to do this!**

```
<session-config>
  <session-timeout>10</session-timeout>
  <cookie-config>
    <http-only>true</http-only>
    <secure>true</secure>
  </cookie-config>
</session-config>
```

<secure>

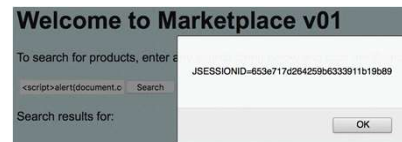
- With <secure>true</secure>, it is enforced that the *Secure* flag is set in every *Set-Cookie* header.
- In this case, the flag is used in every *Set-Cookie* header, even over HTTP, which means sessions no longer work over HTTP.
- It is therefore only a reasonable setting in HTTPS-only applications, as is the case with the Marketplace application.

When using <secure>false</secure> in the Marketplace application, the *Set-Cookie* header still uses the *Secure* flag when cookies are set over HTTPS, so - at least in Payara – it's not possible to deactivate the *Secure* flag when a cookie is set over HTTPS. But this may be different with other application servers.

Effect of the *HTTPOnly* Flag

- In Payara, the *HTTPOnly* flag is always **used per default**
- As a result, the XSS vulnerability in Marketplace v01 cannot be exploited to get the cookie
 - `<script>alert(document.cookie);</script>`
- But when setting the flag to false in *web.xml*, the attack is possible

```
<cookie-config>
  <http-only>false</http-only>
</cookie-config>
```



- The cookie with the session ID is now set as follows:
 - *Set-Cookie: JSESSIONID=ce00603ae279008c541002776386; Path=/Marketplace_v01; Secure*
- **In general: Don't disable it**, there's no point in doing so...

Accessing the Cookies

Accessing the cookies (and therefore also the session ID) with JavaScript is easy, e.g., `<script>alert(document.cookie);</script>`.

<session-config> Element in *web.xml* (2)

- <tracking-mode> allows specifying how session IDs are transported
- So far, we have used the standard settings of Payara, which means:
 - If the browser supports cookies, then cookies are used to transport the session ID
 - If the user disables cookies, the web application includes the session ID in the URLs so that users can still be tracked

 https://localhost:8181/Marketplace_y06/faces/view/public/search.xhtml;sessionId=102b57a15390340585cc

- Question: Do you think this is a good idea?

<session-config> Element in web.xml (3)

- To prevent that the session ID is included in the URL, the *<tracking-mode>* element has to be set to the value *COOKIE*

```
<session-config>
  <session-timeout>10</session-timeout>
  <cookie-config>
    <http-only>true</http-only>
    <secure>true</secure>
  </cookie-config>
  <tracking-mode>COOKIE</tracking-mode>
</session-config>
```
- The effect is as follows:
 - The session ID is only transported within cookies and it is never included in the URL
 - The web application does not accept the session ID in the URL, which effectively prevents session fixation attacks
 - But users that have cookies disabled can no longer use the web application

JSF ViewState – and its Security Implications

ViewState

- Whenever a JSF application serves a web page to the browser, it stores a corresponding ViewState on the server side
 - Other technologies / frameworks use similar concepts, e.g., ASP.NET
 - It stores not only the ViewState of the most recently served page, but also a history of the ViewState of previously served pages
- The main purpose of the ViewState is that the application knows what to do when it receives a subsequent request from the browser
 - I.e., the user clicks a button, which sends a POST request → the ViewState allows to determine the backing bean(s) to update and the method to call
- In addition, the ViewState has security benefits if JSF is used correctly
 - As a security-aware software developer, you should understand these security benefits, so you know what they provide and what you still have to solve yourself
 - This is also a good example to demonstrate why understanding a technology and knowing how to use it correctly is highly important to end up with a secure system

ViewState in Action – Product Search (1)

- Implementation of the search form in [Facelet search.xhtml](#):

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

- This specifies that if the corresponding POST request is received, the search string should be stored in attribute `searchString` in the `SearchBacking` bean, and that the method `search` in the `SearchBacking` bean should be called
- In the generated web page, this form looks as follows:

```
<form id="j_idt10" name="j_idt10" method="post"
  action="/Marketplace_v07/faces/view/public/search.xhtml">
  <input type="hidden" name="j_idt10" value="j_idt10" />
  <input type="text" name="j_idt10:j_idt11" value="" />
  <input type="submit" name="j_idt10:j_idt12" value="Search" />
  <input type="hidden" name="javax.faces.ViewState"
    id="j_id1:javax.faces.ViewState:0"
    value="4636790964174261753:6797988510121825426" />
</form>
```

- The purpose of the ViewState is to link the received POST request to the actions that must be performed in the backing bean

Why is the ViewState needed?

Maybe you are wondering why this ViewState is needed for every single web page that is generated, although – in the case of the Marketplace application – the web pages are basically always the same for a specific Facelet. So just having a per-Facelet mapping of forms and parameters to actions in backing bean(s) should be enough? Well, that's not exactly true. Even in the simple Marketplace application, the web pages generated from the same Facelet can vary. E.g., the number of forms on the search page depends on the search results (as there's one button/form per listed product). So remembering the exact ViewState for this specific web page allows the web application to strictly control that only the listed products can actually be placed in the shopping cart as that any other request (e.g., one that has been manipulated by the user) is rejected to prevent potentially negative side effects from such requests. This shows that the ViewState as used in JSF also has the goal to make it easier for the developer to maintain the integrity of the application and to make it more robust, as invalid requests are strictly blocked from being processed, so the developer does not have to deal with this. Note also that many JSF applications are of course much more complex than the Marketplace application and use advanced features such as adapting Facelets (and therefore also the web pages generated from them) during runtime and Facelets can also be composed of subcomponents that can be dynamically put together during runtime (this goes way beyond what we are discussing here for the purpose to understand security in JSF applications). To support such scenarios, a static per-Facelet mapping would definitely not be enough, which further motivates why a per-web page ViewState information is required.

Simplified ViewState Description

The explanation above and on the next two slides is simplified. For instance, when receiving a request and accessing the stored ViewState information, this also considers the current state of the backing bean(s), which may have changed in the meantime. I.e., if a form would no longer be included in the web page because of the current state of the backing bean(s), then the corresponding form would also not be included any more in the restored ViewState information. You can verify this by putting a product in the shopping cart and go to the checkout page. Then, open a second browser tab and also go to the checkout page. Complete the checkout on the first tab (which will work) and then do the same on the second tab. It won't work, because the shopping cart has been emptied in the meantime (after the first checkout) and with an empty shopping cart, no checkout form would be included on the checkout web page. This is taken into account when accessing and restoring the ViewState information during the second checkout attempt. As a result of this, the checkout form is no longer included in the restored ViewState information, and the purchase won't be processed.

If you want to understand this in detail, refer to the official Jakarta EE documentation (<https://eclipse-ee4j.github.io/jakartaee-tutorial/toc.html>) or a good book. But it's not the purpose of this module to provide a detailed insight into how the ViewState works exactly, which is also not needed to develop a secure Jakarta EE application.

ViewState in Action – Product Search (2)

```
<form id="j_idt10" name="j_idt10" method="post"
  action="/Marketplace_v07/faces/view/public/search.xhtml">

  <input type="hidden" name="j_idt10" value="j_idt10" />

  <input type="text" name="j_idt10:j_idt11" value="" />

  <input type="submit" name="j_idt10:j_idt12" value="Search" />

  <input type="hidden" name="javax.faces.ViewState"
    id="j_id1:javax.faces.ViewState:0"
    value="4636790964174261753:6797988510121825426" />

</form>
```

Hidden field, which
is used to **identify**
the form
(here: *j_idt10*)

Field for the
search string
(currently
empty)

Search button

Hidden field that contains the **ViewState identifier**

- This identifier is used to **identify the stored ViewState of this web page**

- JSF applications use **identifiers such as *j_idt10*** to identify forms and parameters
 - In every generated web page, these identifiers are **unique per form and parameter** → They unambiguously identify both the form and the parameters
- All forms that are included in the generated web page are **stored in the ViewState and the forms and parameters are linked to the actions in the backing beans**
 - Parameter *j_idt10:j_idt11* is linked to the attribute *searchBacking.searchString*
 - Parameter *j_idt10:j_idt12* is linked to the method *searchBacking.search*

ViewState in Action – Product Search (3)

- When submitting the search form, the following request is sent:

```
POST /Marketplace_v07/faces/view/public/search.xhtml HTTP/1.1
...
j_idt10=j_idt10&
j_idt10:j_idt11=DVD&
j_idt10:j_idt12=Search&
javax.faces.ViewState=4636790964174261753%3A6797988510121825426
```

- When receiving the request, the web application does the following:
 - It uses the received **ViewState identifier** to access the stored ViewState of the web page from which the request was triggered
 - **It checks whether the ViewState contains a form that corresponds to the received POST request**, i.e., that contains the form identifier `j_idt10=j_idt10`
 - **If this is not the case, the request is not processed further, and the currently used web page is returned to the browser**
 - **If the ViewState contains such a form, the application completely processes the request using the information from the ViewState:**
 - Store the value of parameter `j_idt10:j_idt11` in attribute `searchBacking.searchString`
 - Call the method that is linked to parameter `j_idt10:j_idt12`, which is `searchBacking.search`

Do all Form Field have to be included in the Request?

No. As an example, if the request above does not include the search string parameter `j_idt10:j_idt11`, the attribute `searchBacking.searchString` won't be updated, but the action will still be executed. Conversely, if the request does include the search string but not the submit parameter `j_idt10:j_idt12`, then the attribute is updated but the action is not called.

But what's important is that if the form is not included in the web page that was sent to the user and therefore also not in the ViewState, then it won't be processed at all. If you want to make sure all parameters are included in the request, this can be enforced by input validation measures (see explanation in the notes of a slide in section *Input Validation*).

Security Implications of the ViewState

- While not intended (primarily) for security, the ViewState is **beneficial for security**
 - It **prevents forced browsing**
 - It **prevents Cross-Site Request Forgery (CSRF) attacks**
- **What is forced browsing?**
 - Normal user behavior: The user sends requests by clicking links and buttons → he only sends requests «**provided to him**» by the application
 - Forced browsing behavior: The user (attacker) sends requests that were **not provided to him** by the application
- Typical forced browsing example
 - Assume an e-shop provides an **area for admins to maintain products**
 - An attacker wants to **change the price** of a product without admin rights
 - Which means he does not have access to the admin area that provides the links and buttons to change the prices
 - If he knows the request to change the price, he can **simply send it to the application**, hoping the application does not check the access rights

Forced Browsing

The goal of forced browsing is often to exploit access control vulnerabilities.

Forced Browsing Prevention with ViewState

- The ViewState prevents forced browsing because it guarantees that only requests that were provided to the user by the web application are accepted
 - All other request – especially also those that are «crafted by the attacker» when doing forced browsing – are not accepted
- The reason why this works is because a JSF application only processes a received request if the corresponding form is included in the ViewState
 - If the request is a forced request, the corresponding form was obviously never provided to the user in a web page...
 - ...which implies the form can also not be included in a stored ViewState of the user...
 - ...which means the forced request will never be processed by the web application

Access Control: ViewState and <security-constraint>

Note that the ViewState does not prevent all kinds of access control vulnerabilities, so it's still required to configure <security-constraint>s correctly. For instance, if access to the admin page is not restricted by a <security-constraint>, then an attacker can easily access the admin page and see all the purchases, and also delete them assuming the buttons (and therefore the forms) are there. So in this case, the ViewState is of no help at all.

But there are cases where the JSF ViewState prevents additional access control attacks even if the <security-constraint>s are configured correctly. A more detailed discussion of this follows later in this chapter and also during the lab exercise.

CSRF Protection (1)

- In a CSRF attack, an attacker forces a victim to **execute unwanted actions** in a web application in which the victim is currently logged in
 - E.g., by sending the victim a link to a web page, which contains an **IMG-tag** and where the image source corresponds to the desired request
- The typical solution to prevent CSRF attacks works as follows:
 - Create a **user-specific CSRF token** after a user has logged in
 - Should be random and long enough such that it cannot be predicted
 - The CSRF token is stored in the session of the user
 - Adapt the web pages sent to the user such **that any subsequent request includes the CSRF token**, in a GET or POST parameter
 - When receiving a request, check whether the **received token has the same value** as the token stored in the session of the user
 - The request is only accepted if the tokens match
- Why does this work?
 - An attacker **would have to guess the CSRF token** to create a valid request, which is considered not feasible if the token is random and long enough

CSRF Protection (2)

- The ViewState identifier provides us with a CSRF token «for free» because it has all properties that we expect of a CSRF token
 - It is random and long enough to be non-predictable
 - It is stored in the user's session
 - It is included as a parameter in every POST request sent to the web application
 - The web application only accepts a request if it includes a valid ViewState identifier
- For an attacker to still perform a CSRF attack, he would have to know the valid ViewState identifier
 - That's not feasible under realistic assumptions, which means that the ViewState identifier effectively protects from CSRF attack

CSRF Prevention with ViewState

- Simulating a CSRF attack with an interceptor proxy:

Admin Area

Purchases:

First Name	Last Name	Credit Card Number	Total Price (CHF)	
Ferrari	Driver	1111 2222 3333 4444	25000.00	Remove purchase
C64	Freak	1234 5678 9012 3456	444.95	Remove purchase
Script	Lover	5555 6666 7777 8888	10.95	

Return to search page Logout

```

POST /Marketplace_v07/faces/view/admin/admin.xhtml HTTP/1.1
Host: localhost:8181
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:74.0) Gecko/20100101 Firefox/74.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded
Content-Length: 155
Origin: https://localhost:8181
DNT: 1
Connection: close
Referer: https://localhost:8181/Marketplace_v07/faces/view/admin/admin.xhtml
Cookie: JSESSIONID=14e1a98df2718dddef7628a8786
Upgrade-Insecure-Requests: 1

j_idt12%3A0%3Aj_idt26%3A0%3Aj_idt26%3Aj_idt12%3A0%3Aj_idt26%3Aj_idt27=Remove+purchase&
javax.faces.ViewState=6348997752027192336%3A6064149822293623717

```

HTTP Status 500 - Internal Server Error

type Exception report

message Internal Server Error

description The server encountered an internal error that prevented it from fulfilling this request.

exception

javax.servlet.ServletException: viewId:/view/admin/admin.xhtml - View /view/admin/admin.xhtml could not be restored.

root cause

javax.faces.application.ViewExpiredException: viewId:/view/admin/admin.xhtml

note The full stack traces of the exception and its root causes are available in the Payara Server 4.1.1.161.1 #backtrace logs.

View /view/admin/admin.xhtml could not be restored.

Modify or remove the ViewState identifier

Does the ViewState Identifier prevent Reflected XSS?

In many cases, CSRF tokens prevent reflected XSS. The reason is that to execute a reflected XSS attack, an attacker has to place a link «somewhere» (e.g., in an e-mail message or on a web page), which contains the malicious JavaScript code, and then the victim must click the link to trigger the attack. If CSRF tokens are used, the resulting request when the link is clicked will only be processed by the web application if the request includes a valid CSRF token of the victim, and getting or guessing such a valid token is usually not feasible for the attacker. As a result of this, reflected XSS is often not possible if CSRF tokens are used.

However, this only works if the request is completely blocked by the web application and the response does only include a generic error message without including the received JavaScript code. If the response page does include the JavaScript code in non sanitized form and only blocks the resulting action of the request, then reflected XSS may still be possible.

As the ViewState identifier basically corresponds to a CSRF token, the ViewState identifier also helps to prevent reflected XSS. And as the response contains only a generic error message and no JavaScript code (see slide above), it works in any case – assuming the ViewState identifier cannot be predicted by the attacker.

Nevertheless, you shouldn't consider the ViewState identifier (or CSRF tokens in general) as a method to prevent reflected XSS. One reason is that the attacker may still somehow get access to a valid ViewState identifier, which would allow him to execute a reflected XSS attack. The more important reason is that the ViewState identifier (and CSRF tokens in general) can only help against *reflected* XSS, but not against stored XSS. This means the ViewState identifier won't solve all XSS problems and as a result of this, if you rely on the ViewState identifier «too much» to prevent XSS, it may easily happen that you'll overlook something so attacks may still be possible. Therefore, to be on the safe side, always make sure to use strict data sanitation to prevent XSS in any case in JSF applications (and also when you are using any other underlying technology).

Explicit CSRF Token with JSF

- JSF also supports **explicit CSRF tokens**, which can be configured in *faces-config.xml*:

```
<protected-views>
  <url-pattern>/view/public/cart.xhtml</url-pattern>
</protected-views>
```

- Corresponding requests then **include the token as a GET parameter**

https://localhost:8181/Marketplace_v07/faces/view/public/cart.xhtml `javax.faces.Token=qjv5w4rsGXgzhHzuMFeZses8g8ht30jP61AnNs4%3D`

- If the web application receives a request with an invalid token, the **request is not processed** and the application responds with an error

HTTP Status 500 - Internal Server Error

2025 Exception report

Message: Internal Server Error

javax.faces.application.ProtectedViewException

javax.servlet.ServletException

Root cause:

javax.faces.application.ProtectedViewException

Note: The full stack traces of the exception and its root causes are available in the JSP error page 4.1

- When should this be used?
 - As JSF usually performs **all critical operations using forms / POST requests**, the ViewState identifier will be adequate to prevent CSRF in most cases
 - If you want to additionally want to protect **GET requests**, use this token

Programmatic Security Features in Jakarta EE

Marketplace V08

The extensions of this section are integrated in Marketplace_v08.

Programmatic Security Features in Jakarta EE (1)

- Up to now, we mainly used **declarative security features** of Jakarta EE that can be configured in *web.xml* or with annotations
 - *<security-role>*, *<security-constraint>*, *@DatabaseIdentityStoreDefinition*, *@FormAuthenticationMechanismDefinition*,...
- In addition, Jakarta EE also offers some **programmatic security features**
 - They are **extensions of the declarative security mechanisms**, so they can only be used if one uses declarative security features as a basis
- The main purposes of these features are the following:
 - Get information about **username, roles and access permissions** of the current user to use this to make further decisions in the program code
 - Perform **programmatic authentication and logout** (logout is already used in Marketplace)

Programmatic Security Features in Jakarta EE (2)

- Up to **Java EE 7**, the programmatic security features are provided by methods of the interface *HttpServletRequest*:
 - *String* **getRemoteUser()** – returns the user name of the current user
 - *boolean* **isUserInRole(String role)** – returns whether the current user is logged in and has the specified role
 - *Principal* **getUserPrincipal()** – returns an object that contains name and roles of the current user
 - *void* **logout()** – performs a logout, the authenticated user is forgotten by the web application (already used in Marketplace)
- With **Jakarta EE 8** and the new Security API, additional features are provided by the interface *SecurityContext*:
 - *boolean* **isCallerInRole(String role)** – same as *isUserInRole()* above
 - *Principal* **getCallerPrincipal()** – same as *getUserPrincipal()* above
 - *boolean* **hasAccessToWebResource(String resource, String... methods)** – checks if caller has access to the web resource via one of the methods provided
 - *AuthenticationStatus* **authenticate(HttpServletRequest req, HttpServletResponse res, AuthenticationParameters param)** – performs programmatic authentication using the configured identity store

getRemoteUser()

If no user is currently logged in, *getRemoteUser()* returns null.

Marketplace – Make Decisions based on the Role of the Current User (1)

- To demonstrate how security-relevant decisions can be done based on the role of the current users, we extend the Marketplace application:
 - Current behavior: Users with the role *marketing* or *sales* can access the admin area to **view and remove purchases**
 - New behavior: Both roles still can access the admin area to view the purchases, but **only users with role *sales* can remove purchases**
- To support this, the application should **display the *Remove purchase* buttons** only if the current user has the role *sales*
- Implementing this can be done with ***rendered*** in *admin.xhtml*:



request provides access to the *HttpServletRequest* object of the Faces Servlet

```
<h:column rendered="#{request.isUserInRole('sales')}">
  <h:form>
    <h:commandButton value="Remove purchase"
      action="#{adminPurchaseBacking.remove(purchase)}" />
  </h:form>
</h:column>
```

Accessing the *HttpServletRequest* Object

As mentioned before, an object implementing the *HttpServletRequest* interface is available to the Faces Servlet. From a Facelet, this object can be accessed with *request.*, which is a so-called implicit object of the JSF Expression language.

Why not using *isCallerInRole* of *SecurityContext*?

The main reason is that there is no (at least not yet) implicit object of *SecurityContext* that can be used in Facelets.

Marketplace – Make Decisions based on the Role of the Current User (2)

- Login as *alice (sales)*

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](#)

- Login as *robin (marketing)*

Admin Area

Purchases:

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

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

- Question: We have prevented that marketing users are getting the buttons – but they probably can still remove a purchase by **manually submitting the corresponding request**
 - Is this possible or not?

Marketplace – Make Decisions based on the Role of the Current User (3)

- No – this is **not possible** – because the ViewState of JSF applications provides protection from forced browsing
 - The admin page that is provided to a user with role *marketing* does **not include the buttons (forms)** to remove a purchase...
 - ...therefore, the corresponding forms are also **not included in the stored ViewState**...
 - ...which means that if a user with role *marketing* sends the request to remove a purchase, **the request won't be processed** because requests are only processed if the corresponding form is stored in the ViewState
- **Important:** When using JSF, you «get this for free», but with **other technologies / frameworks**, you typically must **solve this on your own**
 - This is done by **checking the rights of the user again** when he submits the request to remove a purchase

Admin Area

Purchases:

First Name	Last Name	Credit Card Number	Total Price (CHF)
Ferrari	Driver	1111 2222 3333 4444	250000.00
CB4	Freak	1234 5678 9012 3456	444.95
Script	Lover	5555 6666 7777 8888	10.95

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

rendered Attribute and Forced Browsing

In some cases, the *rendered* attribute behaves a bit in a strange way, which can have an impact on security. Specifically, when the *rendered* attribute is used in a `<h:form>` tag, the request is sometimes accepted by the web application although the form is not included in the web page (such a request should never be accepted according to the behavior if the ViewState). For instance, looking at the Facelet *checkout.xhtml* in the Marketplace application, we use the *rendered* attribute in the `<h:panelGrid>` tag instead of the `<h:form>` tag. This way, the request to complete a purchase cannot be submitted by forced browsing in case the checkout form has not been included in the checkout web page (because the shopping cart is empty as this is checked in this case with the *rendered* attribute). However, if exactly the same *rendered* attribute is used in the `<h:form>` tag that is surrounding the `<h:panelGrid>` tag, then doing a purchase using forced browsing is possible, although the form is again «correctly» not included in the web page if the shopping cart is empty. The reason for this behavior is unclear and the web does not provide concluding answers. The general recommendation to prevent such issues in general is either not using the *rendered* attribute in the `<h:form>` tag (as this seems to be the only problematic case) or – in addition – use the same *rendered* attribute also in the `<h:commandButton>` tag, as this always «strictly» prevents forced browsing, no matter in what other tags *rendered* is used. In the current example (although it's not needed as it works securely as it currently is implemented), this would mean extending the code as follows:

```
<h:column rendered="#{request.isUserInRole('sales')}}">
  <h:form>
    <h:commandButton value="Remove purchase"
      rendered="#{request.isUserInRole('sales')}}"
      action="#{adminPurchaseBacking.remove(purchase)}" />
  </h:form>
</h:column>
```

All code examples in this chapter (and in the corresponding lab) work securely in the way they are discussed here, without the need for the additional *rendered* attribute in the `<h:commandButton>` tag.

Sidenote: It seems using *rendered* in `<h:form>` is mainly a possible problem if a form is included in the web page just once (as in the case of *checkout.xhtml* discussed above). If `<h:form>` is used (typically multiple times) within `<h:dataTable>`, then no problems could be observed so far.

Marketplace – Make Decisions based on the Role of the Current User (4)

- Hmm... isn't this a bit risky? This is only secure if one makes sure that web pages **only include the buttons (forms) that correspond to actions that should be accessible** by the current user – right?
 - That's true – if one **accidentally includes a form** that shouldn't be accessible, then there will be an access control vulnerability
 - Therefore, to rely on this, you must make sure to **not include any navigation elements** that provide access to unauthorized actions
 - E.g., in the current example, this means you must take care to **use the *rendered* attribute correctly!**
- Of course, you can also use a **server-side check to enforce access control** instead of relying on navigation elements and the ViewState
 - You can also use this **as an additional 2nd access control check** (to be on the «very secure side»), but technically it's not needed if the navigation elements are used correctly

```
// Extend AdminPurchaseBacking.java
@Inject private SecurityContext securityContext;
public String removePurchase(Purchase purchase) {
    if (securityContext.isCallerInRole("sales")) {
        purchaseFacade.remove(purchase);
        purchases.remove(purchase);
    }
    return "/view/admin/admin";
}
```

Isn't this a bit Risky?

In the end, using the *rendered* attribute correctly is not really more difficult than implementing such checks correctly in the backend. It's just a different way to do this and once you get used to it, it shouldn't be a problem in practice. It's also the recommended way to do in JSF applications.

Feel free to do an additional check in the backend, it certainly won't hurt, but it also means that you are (somewhat) repeating code and functionality.

When discussing the Marketplace REST API later, then we cannot rely on such an approach as provided by JSF and there, we'll have to implement the checks in the backend.

- Currently, Marketplace uses the standard *FormAuthenticationMechanism* provided by Jakarta EE, which works basically well and is quite easy to use, but it has its limitations
 - Requires *two Facelets / HTML pages* (login / error) that are often very similar
 - *Not well integrated* with JSF, as a standard form must be used (*<form>* instead of *<h:form>*), which implies no backing bean can be used
 - Not possible to *programmatically influence* the login process, e.g., to slow down the user after a few failed login attempts
- To overcome this, the Jakarta EE Security API provides a more flexible *CustomFormAuthenticationMechanism*
 - Configuration is still very easy using annotation *@CustomFormAuthenticationMechanismDefinition*
 - But performing the actual authentication within the application must now be done *programmatically* and does not «happen automatically»
 - For this, method *authenticate* of the interface *SecurityContext* must be used

Marketplace – Custom FORM-based Authentication (1) – *@CustomFormAuthenticationMechanismDefinition*

- Configuring *CustomFormAuthenticationMechanism* works very similar to *FormAuthenticationMechanism*
 - Using annotation *@CustomFormAuthenticationMechanismDefinition*

```
@DatabaseIdentityStoreDefinition(...)

@CustomFormAuthenticationMechanismDefinition(
    loginToContinue = @LoginToContinue(
        loginPage="/faces/view/public/secure/login.xhtml",
        errorPage=""
    )
)

@ApplicationScoped
public class ApplicationConfig {
}
```

The login page is now a «real»
Facelet that uses a backing bean

Setting *errorPage* to the empty string means
that the login page is used as error page

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.

Marketplace – Custom FORM-based Authentication (2) – Login Facelet

```
<div id="header">
  <h1>Login</h1>
  <h:messages globalOnly="true" layout="table" infoClass="redItalicText"/>
</div>

<div id="loginForm">
  <p>Please log in to continue.</p>
  <h:form>
    <h:panelGrid columns="2">
      <h:outputLabel value="Username:" />
      <h:inputText value="#{authenticationBacking.username}" />
      <h:outputLabel value="Password:" />
      <h:inputSecret value="#{authenticationBacking.password}" />
      <h:outputText value="" />
    <h:panelGrid columns="2">
      <h:commandButton value="Login"
        action="#{authenticationBacking.login}" />
      <h:button value="Cancel" outcome="/view/public/search" />
    </h:panelGrid>
  </h:form>
</div>
```

Displays an **error message** if the login has failed
• Set by the backing bean when trying to do the login

The **backing bean** *AuthenticationBacking* is used to store the state (username & password) and to provide the login functionality

Login Facelet

- Before, the login page had to use the standard identifiers *j_security_check*, *j_username*, and *j_password*.
- As a result, the web application automatically did the login when receiving the request.
- As we want to integrate the login process better into the JSF Marketplace application, this is no longer used and instead, we turn the login page into a Facelet that uses a backing bean.

<h:inputSecret>

Note that for a password field, JSF provides the tag **<h:inputSecret>**.

Marketplace – Custom FORM-based Authentication (3) – *AuthenticationBacking.java* (1)

- *AuthenticationBacking.java* is already available, but so far it has only been used for logout
 - It is extended to hold the *state of the login Facelet* and to provide the *login method*

```
public class AuthenticationBacking implements Serializable {
    private static final long serialVersionUID = 1L;
    @Inject private SecurityContext securityContext;
    private String username;
    private String password;

    public String getUsername() {
        return username;
    }
    public void setUsername(String username) {
        this.username = username;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
}
```

SecurityContext provides programmatic access to authentication and authorization operations

Private instance variables to hold the state of the Facelet

Public getter and setter methods for the attributes

SecurityContext

This object allows developers to do authentication and checks to grant or deny access to application resources programmatically. It is provided by the Jakarta EE Security API. The object is maintained by the underlying application server and the developer doesn't have to care about its creation. To use it, it can be injected as illustrated in the code above.

Marketplace – Custom FORM-based Authentication (4) – *AuthenticationBacking.java* (2)

```

public void login() {
    FacesContext facesContext = FacesContext.getCurrentInstance();
    Credential credential =
        new UsernamePasswordCredential(
            username, new Password(password));
    AuthenticationStatus status =
        securityContext.authenticate(
            getRequest(facesContext),
            getResponse(facesContext),
            withParams().credential(credential));
    if (status.equals(SEND_CONTINUE)) {
        facesContext.responseComplete();
    } else if (status.equals(SEND_FAILURE)) {
        Message.setMessage(
            "Username or password wrong");
    }
}

private static HttpServletRequest getRequest(FacesContext context) {
    return (HttpServletRequest) context.getExternalContext().getRequest();
}

private static HttpServletResponse getResponse(FacesContext context) {
    return (HttpServletResponse) context.getExternalContext().getResponse();
}

public String logout() { // Unchanged }
}

```

Credential object is created from the received username and password

authenticate performs the authentication using the configured identity store (*DatabaseIdentityStore*)

- Successful authentication stores information about the user and roles in the session

If authentication is **successful**, use *facesContext.responseComplete()* to redirect the browser to the originally requested page (302 response to the admin page)

- As the user has been authenticated, access will be granted

If authentication is **not successful**, set an error message in the faces context and return from the method, which serves the current resource (*login.xhtml*) again to the client

Standard way to get the *HttpServletRequest* and *HttpServletResponse* objects of the Faces Servlet (boilerplate code)

© ZHAW / SoE / IHT – Marc Reinhard, Stephan Neudaus

FacesContext

The faces context is a core object that is used whenever a JSF application handles a request. Basically, it's used to store all relevant state information while the request is handled and the response is generated.

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

withParams()

This is a static helper method provided by the class *AuthenticationParameters* to create an object of type *AuthenticationParameters* from the credentials.

Import Statements

The class above requires the following import statements:

```

import java.io.Serializable;
import jakarta.inject.Named;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.enterprise.context.SessionScoped;
import jakarta.faces.context.FacesContext;
import jakarta.inject.Inject;
import jakarta.security.enterprise.AuthenticationStatus;
import jakarta.security.enterprise.SecurityContext;
import jakarta.security.enterprise.credential.Credential;
import jakarta.security.enterprise.credential.Password;
import jakarta.security.enterprise.credential.UsernamePasswordCredential;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServletResponse;
import static jakarta.security.enterprise.AuthenticationStatus.SEND_CONTINUE;
import static jakarta.security.enterprise.AuthenticationStatus.SEND_FAILURE;
import static jakarta.security.enterprise.authentication.mechanism.http.
    AuthenticationParameters.withParams;
import ch.zhaw.securitylab.marketplace.common.utility.Message;

```

Behavior compared to *FormAuthenticationMechanismDefinition*

With the exception that the actual authentication is now done programmatically, the overall behavior is the same as with *FormAuthenticationMechanismDefinition* before: Successful authentication stores user / role(s) in the session and after successful authentication, the web application sends the browser a redirection response that points to the originally requested page (admin area).

Marketplace – Custom FORM-based Authentication (5)

- Testing with user *alice* works as expected
 - First with wrong password and then with correct password:

Welcome to Marketplace v08

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

Search results for:

No products match your search

Login

Please log in to continue.

Username:

Password:

Admin Area

Purchases:

First Name	Last Name	Credit Card Number	Total Price (CHF)	
Ferrari	Driver	1111 2222 3333 4444	250000.00	<input type="button" value="Remove purchase"/>
C64	Freak	1234 5678 9012 3456	444.95	<input type="button" value="Remove purchase"/>
Script	Lover	5555 6666 7777 8888	10.95	<input type="button" value="Remove purchase"/>

Login

Username or password wrong

Please log in to continue.

Username:

Password:

Input Validation

Marketplace V09 and V10

The extensions of this section are integrated in Marketplace_v09 and v10.

Bean Validation Framework (1)

- Input validation means that data received by the web application is **first validated (or filtered)** before it is processed further
 - This is important with respect to security as it can **protect from several attacks** (injection attacks, parameter tampering,...)
 - But it's also important for **functional aspects** to make sure, e.g., that users only submit correctly formatted credit card numbers or e-mail addresses
- Jakarta EE provides a powerful whitelisting-based input validation approach: the **Bean Validation framework**
 - It's a generic input validation framework in the sense that it can be used in the context of different Jakarta EE applications, e.g., with JSF applications (see here) and with RESTful web services based on JAX-RS (see later)
- If the Bean Validation framework is used in a JSF application, the **input validation rules (named validation constraints)** must be specified for the attributes that are **bound to the JSF input elements**
 - Typically, the constraints are used in backing beans or entities

Bean Validation Framework

Validation constraints can also be applied to method and constructor parameters and return values. But we are not going to use this here.

Whitelisting or Blacklisting?

With input validation, one can use two different approaches: whitelisting or blacklisting.

Whitelisting means that one defines what is legitimate for the received data, e.g.:

- The data must contain at least 5 and at most 100 characters.
- The allowed characters include letters, digits, dots (.) and slashes (/).

The opposite is blacklisting, i.e., defining what is forbidden.

In general, whitelisting has several advantages:

- It is more natural because when asking yourself what input data is legitimate, you usually think about what characters are allowed.
 - A name of a person has at most 30 characters and consists of letters...
- It's usually more secure as with blacklisting, it's easy to forget something, which may have a negative security impact.

Therefore, for security reasons, always use whitelisting. The Bean Validation framework provides a whitelisting approach, i.e., it allows to define what data is legitimate (by using annotations), so it follows this recommendation.

Java Input Validation Libraries beyond what is included in Jakarta EE

Apache Commons Validator: <https://commons.apache.org/validator/>

OWASP ESAPI: https://www.owasp.org/index.php/Category:OWASP_Enterprise_Security_API

Bean Validation Framework (2)

- Example 1: **search string input field** in *search.xhtml*

```
<h:inputText value="#{searchBacking.searchString}" />
```

- → Validation constraints should be added to attribute *searchString* in the backing bean *SearchBacking.java*

- Example 2: **first name input field** in *checkout.xhtml*


```
<h:inputText value="#{checkoutBacking.purchase.firstname}" />
```

- → Validation constraints should be added to attribute *firstName* in entity *Purchase.java*

- **What happens within the application if Bean Validation is used:**

- When receiving a POST request, the Faces Servlet **tries to write the received parameter values** into the attributes of the backing bean(s)
- If any of these attributes has a **validation constraint**, the corresponding **parameter value is first validated**
- If any validation **fails**, none of the **attributes in the backing bean(s) are updated** and the **action of the request is not invoked**
- In this case, the application returns the currently used Facelet to the browser, so the user can «try again»
 - Of course, one should display a message to tell the user what went wrong

- We first add input validation to the **search field**
 - We restrict the search string to **50 characters at most**
- 1st step: Add a **Bean Validation annotation** in front of attribute ***searchString*** in the backing bean

```
public class SearchBacking implements Serializable {  
     @Size defines the maximum number of characters  
    and a message to be used if validation fails  
    @Size(max=50, message = "The search string must not be longer than 50  
        characters")  
    private String searchString;  
}
```

- Besides **@Size**, the Bean Validation framework provides **many other annotations**, e.g.:
 - **@NotNull, @Min, @Max, @Pattern, @Past, @Future,...**

Bean Validation Annotations

Jakarta EE provides several of them, for a full list refer to
<https://eclipse-ee4j.github.io/jakartaee-tutorial/bean-validation002.html>

Bean Validation Annotations follow a Whitelisting approach

This can nicely be seen with the **@Size** annotation: It defines what is legitimate for the received data so that it is written into the attribute, which is exactly how whitelisting works (blacklisting would be defining what is not legitimate for the received data so that it is not written into the attribute). In this case, it defines that legitimate data contains at most 50 characters. Otherwise, input validation fails, the data is not written into the attribute, and the request is not processed further. The other annotations work in the same way: They always define what is legitimate for the corresponding attribute.

- 2nd step: Add an *h:message* element to *search.xhtml* to display the validation message in case validation fails

The grid requires 3 columns as we also want to display error messages

```
<h:panelGrid columns="3">
  <h:form>

    <h:inputText id="searchString"
      value="#{searchBacking.searchString}" />
    <h:commandButton value="Search" action="#{searchBacking.search}" />
    <h:message for="searchString" errorClass="redItalicText" />
  </h:form>
</h:panelGrid>
```

h:inputText requires an *id*, so validation errors occurring with this field are associated with this id

h:message displays the validation message in case validation fails

- The attribute *for* specifies the message to be displayed
- As it uses the same value as the attribute *id* above (*searchString*), only the validation messages of the search field are displayed

Marketplace – Bean Validation (3)

- Entering a **too long** search string:

Welcome to Marketplace v09

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

The search string must not be longer than 50 characters

Search results for:

- Entering a **valid** string:

Welcome to Marketplace v09

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

Search results for: DVD

Description	Price (CHF)	
DVD Life of Brian - used, some scratches but still works	5,95	<input type="button" value="Add to cart"/>

The search string is only included in the page if validation is successful
 → This shows the value of the attribute (and therefore the model) is not changed if validation fails

- Next, we add input validation to the checkout process, using the following rules:
 - **First name and last name:**
 - Minimum 2, maximum 32 characters
 - Allowed are lower- and uppercase letters and the single quote ('), as O'Neill and O'Connor want to shop as well
 - The **credit card number** must have a valid credit card number format
- The first rule can be covered with a **regular expression** `^[a-zA-Z']{2,32}$`
- For the second rule, we have to write a **custom Bean Validation annotation**

Regular Expressions and Whitelisting

Regular expressions are a natural and very well-suited choice to define whitelisting-based rules, as they allow to define – in a very flexible way – to which pattern the string to be checked must correspond.

- First, we add **Bean Validation annotations** in the *Purchase* entity

```
public class Purchase implements Serializable {
```

```
    @Id private int purchaseID;
```

```
    @Pattern(regexp = "[a-zA-Z']{2,32}", message = "Please insert a  
        valid first name (between 2 and 32 characters)")  
    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;
```

```
    @CreditCardCheck  
    private String creditCardNumber;
```

@Pattern defines a regex to use for validation and a message to be used if validation fails

@CreditCardCheck is a custom Bean Validation annotation we implement ourselves

- To **implement a custom Bean Validation annotation**, we must do the following:
 - Implement the **annotation** itself
 - Write a class that implements the interface *ConstraintValidator* and that contains the actual code that performs the validation

Custom Validator Annotation

For a good overall explanation, see: <https://softwarecave.org/2014/03/27/custom-bean-validation-constraints/>

Marketplace – Bean Validation (6) – Annotation *CreditCardCheck.java*

```
@Target({FIELD, METHOD})  
@Retention(RUNTIME)  
@Documented
```

Standard annotations when implementing an annotation,
@Target specifies the annotation can be applied to attributes
(fields) of classes and parameters and return values of methods

@Constraint marks this annotation as a Bean Validation
annotation and specifies the class used for validation

```
@Constraint(validatedBy = CreditCardValidator.class)
```

In Java, *@interface* is used to
implement an annotation type

```
public @interface CreditCardCheck {
```

```
    String message() default "Please insert a valid credit  
                                card number (16 digits)";
```

```
    Class<?>[] groups() default {};
```

```
    Class<? extends Payload>[] payload() default {};
```

```
}
```

Three attributes which must be implemented in Bean Validation annotations (boilerplate
code), we set a **default value for the message** to be used if validation fails

@Constraint

For more information, refer to the API documentation:

<https://jakarta.ee/specifications/platform/9/apidocs/jakarta/validation/constraint>

@Target

In this chapter, we will only apply this annotation to attributes and not to methods, but in general, bean validation annotations should be applicable to both attributes and method parameters and return values, so we configure the annotation accordingly.

Import Statements

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

```
import java.lang.annotation.Documented;  
import java.lang.annotation.Retention;  
import java.lang.annotation.Target;  
import jakarta.validation.Constraint;  
import jakarta.validation.Payload;  
import static java.lang.annotation.ElementType.*;  
import static java.lang.annotation.RetentionPolicy.*;
```

Marketplace – Bean Validation (7) – *CreditCardValidator.java* (1)

The validator class must implement interface *ConstraintValidator*, which requires two type parameters

- The first identifies the annotation for which this class is used
- The second identifies the type of the value to be validated

```
public class CreditCardValidator implements  
    ConstraintValidator<CreditCardCheck, String> {
```

Can be used for initialization,
implemented empty in our case

```
@Override  
public void initialize(CreditCardCheck constraintAnnotation) {}
```

isValid performs the actual
validation, returning true or false

```
@Override  
public boolean isValid(String value, ConstraintValidatorContext context) {  
    if (value == null) { return false; }  
}
```

Extract the digits from the received card number (\D matches any character except digits)

```
String digits = value.replaceAll("\\D", "");  
return checkRawFormat(value) && luhnCheck(digits);  
}
```

Use two private methods to perform the **actual validation**,
isValid only returns true if both these methods return true

Import Statements

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

```
import jakarta.validation.ConstraintValidator;  
import jakarta.validation.ConstraintValidatorContext;
```

initialize method

This is usually used to get values of attributes used by the annotations. For instance, in the case of the bean validation annotation *@size* that uses the *max* attribute, *initialize* would access the value of the attribute using *constraintAnnotation.max()* and store it in an instance variable of the object. Then, this value can be used in *isValid* to check if the length of the received data is OK.

Check for *null*

At the beginning of *isValid*, there's a check that *value* is not null (if *value* is null, there would be a *NullPointerException* when calling *value.replaceAll()*). The way we are currently using the annotation in the Marketplace annotation, however, the value cannot be null as JSF guarantees that the request to do a checkout contains a corresponding parameter. However, in other cases where the annotation may be used, this is not guaranteed (e.g., in the case of the REST API that follows in the next chapter), and therefore, and in general to make sure the code is robust, this check is reasonable.

Check the **raw format**: 4 groups with 4 digits, separated by 0 or 1 space characters, e.g., 1111222233334444 or 1111 2222 3333 4444

```
private boolean checkRawFormat(String cardNumber) {
    return cardNumber.matches("^\\d{4}[ ]?\\d{4}[ ]?\\d{4}[ ]?\\d{4}$");
}
```

Verify the **checksum over the credit card number** with the *Luhn algorithm* (details out of scope)

```
private boolean luhnCheck(String cardDigits) {
    int sum = 0;
    for (int i = cardDigits.length() - 1; i >= 0; i -= 2) {
        sum += Integer.parseInt(cardDigits.substring(i, i + 1));
        if (i > 0) {
            int d = 2 * Integer.parseInt(cardDigits.
                substring(i - 1, i));

            if (d > 9) {
                d -= 9;
            }
            sum += d;
        }
    }
    return sum % 10 == 0;
}
```

- Just like in *search.xhtml*, we now only have to use *h:message* to display the validation messages in case validation fails


```
<h:panelGrid columns="3" rendered="#{cartBacking.count != 0}">
  <h:outputLabel value="First name:" />
  <h:inputText id="firstname"
    value="#{checkoutBacking.purchase.firstname}" />
  <h:message for="firstname" errorClass="redItalicText" />
  <h:outputLabel value="Last name:" />
  <h:inputText id="lastname"
    value="#{checkoutBacking.purchase.lastname}" />
  <h:message for="lastname" errorClass="redItalicText" />
  <h:outputLabel value="Credit card number:" />
  <h:inputText id="creditCardNumber"
    value="#{checkoutBacking.purchase.creditCardNumber}" />
  <h:message for="creditCardNumber" errorClass="redItalicText" />
  <h:outputText value="" />
  <h:commandButton value="Complete Purchase"
    action="#{checkoutBacking.completePurchase}" />
  <h:outputText value="" />
</h:panelGrid>
```

Marketplace – Bean Validation (10)

- Testing shows that validation works as expected:

First name:	<input type="text" value="Jo hn"/>	Please insert a valid first name (between 2 and 32 characters)
Last name:	<input type="text" value="o"/>	Please insert a valid last name (between 2 and 32 characters)
Credit card number:	<input type="text" value="1111 2222 3333 4445"/>	Please insert a valid credit card number (16 digits)
<input type="button" value="Complete purchase"/>		

First name:	<input type="text" value="John"/>	
Last name:	<input type="text" value="O'Connor"/>	
Credit card number:	<input type="text" value="1111 2222 333 34444"/>	Please insert a valid credit card number (16 digits)
<input type="button" value="Complete purchase"/>		

First name:	<input type="text" value="John"/>		Welcome to Marketplace v09 Your purchase has been completed, thank you for shopping with us
Last name:	<input type="text" value="O'Connor"/>		
Credit card number:	<input type="text" value="1111 2222 3333 4444"/>		
<input type="button" value="Complete purchase"/>			

Making sure all required Parameters are included in the Request

With the current implementation, it is not enforced that the request to do a purchase contains all request parameters. For instance, if the firstname parameter (or any other parameters) is removed from the request (e.g., by an attacker who wants to check «what happens»), then the corresponding attribute won't be written in the backing bean and there will be an exception when trying to insert the new row into the database. This is not security critical in this case and also, no information leakage happens if errors are handled in a standard way (with `<error-page>` in `web.xml`), so we don't discuss this in detail. But to increase robustness of an application, it usually makes sense to handle this correctly. This can be done by using the `required` attribute together with the `requiredMessage` attribute in the `<h:inputText>` tag in the Facelet.

The following shows how this can be done with the firstname field:

```
<h:inputText id="firstname" required="true" requiredMessage="Please insert  
a valid first name (between 2 and 32 characters)"  
value="#{checkoutBacking.purchase.firstname}" />
```

With this, it is enforced (server-side) that the user enters a first name, i.e., if no first name is entered, then the request is not processed and the message specified in `requiredMessage` is shown as the input validation error message, just like with the standard Bean validation mechanism.

However, this only makes sure that using no value for the firstname parameter is not accepted, but it won't have any effect if the firstname parameter is completely removed from the request. To enforce that the `required` attribute is also considered if the parameter is not included at all in the request, one must add the following configuration to `web.xml`:

```
<context-param>  
  <param-name>jakarta.faces.ALWAYS_PERFORM_VALIDATION_WHEN_REQUIRED_IS_TRUE</param-name>  
  <param-value>true</param-value>  
</context-param>
```

Using this approach consistently with all parameters, one can enforce that requests must always include all required parameters and any tampering of an attacker that remove parameters completely from requests should therefore be detected. In general, it's recommended to do this for robustness reasons and to prevent negative side-effect (e.g., `NullPointerException`s due to backing bean attributes that contain unexpected null values).

Input Validation and Encoding (1)

- Assume that we **relax the input validation rules** for the first name a little bit (we also allow **digits** and **%**, and up to **50 characters**)

```
public class Purchase implements Serializable {  
    @Pattern(regexp = "[a-zA-Z'%0-9]{2,50}", message = "Please insert a  
        valid first name (between 2 and 50 characters)")  
    private String firstname;  
}
```

- In addition, the column to display the first name in the Facelet *admin.xhtml* has the **escape attribute set to false**

```
<h:column>  
    <f:facet name="header">First Name</f:facet>  
    <h:outputText value="#{purchase.firstname}" escape="false" />  
</h:column>
```

- Question:
 - The goal of the attacker is to insert `<script>alert("XSS");</script>` in the first name field during checkout such that it is executed when a user with role *sales* or *marketing* views the purchases (stored XSS attack)
 - Can this attack be carried out and if yes, how?

Input Validation and Encoding (2)

- Answer: The attack most likely does not work (yet), but at least **we can get the script past the input validation** filter
- The trick is to **encode the non-permitted characters** in the script [`<` `>` `"` `(` `)` `/` `;`] with an encoding scheme that uses only **legitimate characters**
- As **letters, digits and % are allowed, we can use URL encoding**
 - **URL encoding uses the % character followed by the ASCII code as hex value**
- Therefore, `<script>alert("XSS");</script>` can be encoded to **`%3Cscript%3Ealert%28%22XSS%22%29%3B%3C%2Fscript%3E`**
 - This will be accepted, as only legitimate characters are used
- But: This is not enough for the attack to be successful as URL encoded scripts are **usually not executed in the browser**

URL Encoding

The ASCII code of `<` is 60 or 3C (hex), so its URL encoding is `%3C`.

Input Validation and Encoding (3)

- We need a little bit more **help from the developer**
 - Basically, the attack will only work if the first name is **URL decoded** before the script is sent to the browser
- Assume the developer has made exactly this mistake by **decoding the first name before writing it into the database**
 - Maybe he wants to make sure that only decoded data is stored in the database

```
String decodedFirstname;
decodedFirstname = URLDecoder.decode(firstname);
```

- In this case, the attack will be successful:

URL Decoding before writing into the Database

With JPA, this can easiest be implemented with an *AttributeConverter*. This makes sure that data is transformed just before writing it into the database or just after reading it from the database. To URL decode data just before writing it into the database, this converter would look as follows:

```
@Converter
public class URLConverter implements AttributeConverter<String, String> {
    @Override
    public String convertToDatabaseColumn(String data) {
        try {
            return URLDecoder.decode(data, "UTF-8");
        } catch (UnsupportedEncodingException e) {}
        return "";
    }

    @Override
    public String convertToEntityAttribute(String data) {
        return data;
    }
}
```

To apply the converter to the *firstname* attribute in entity *Purchase*, the following annotation must be used (in addition to the already used Bean Validation annotation):

```
@Pattern(regexp = "[a-zA-Z'%0-9]{2,50}$", message = "Please insert a
    valid first name (between 2 and 50 characters)")
@Convert(converter = URLConverter.class)
private String firstname;
```

Input Validation and Encoding (4)

- One can argue that **many things had to come together** for the attack to be successful:
 - An **encoding scheme** must be supported by the permitted characters
 - The application has to **decode** the encoded script
 - **Data sanitation** must be turned **off**
- Nevertheless, **such attacks have happened**, and this examples shows that simply filtering some special characters (e.g., <, >, ' etc.) using **input validation may not always be enough to prevent attacks** such as XSS (as illustrated here), SQL injection, and others
- Best practices to **prevent such encoding attacks**:
 - **If possible, don't decode any data within your application**
 - If decoding has to be done for any reason, **think hard** about its possible impact on security (and do decoding first and input validation afterwards)
 - If there are better defenses than input validation, make sure to use them, e.g., **data sanitation to prevent XSS** and **prepared statements to prevent SQL injection** – this should work even if input validation is done incorrectly

Execution of Encoded Scripts in Browsers

There have been several cases in the past where browsers contained vulnerabilities such that they executed URL encoded scripts. In this case, the decoding step is not necessary for attacks to be possible.

Summary (1)

- When developing a web application **many security-relevant details** must be considered
- **Jakarta EE provides many security features** that are helpful to secure web applications, e.g.:
 - **JSF** protects from XSS, CSRF and forced browsing
 - **JPA** protects from SQL injection
 - **Standard error pages** to prevent information leakage
 - **Bean Validation framework** to implement input validation
 - **Security constraints** for access control
 - **Jakarta EE Security API** for authentication
- As a Jakarta EE developer, it's very important that you **really understand the Jakarta EE technology** to make the most out of these features and to use them correctly

Summary (2)

- When using **other technologies / frameworks** than Jakarta EE, the technical details how security should be implemented / configured will be different...
 - ...but the **conceptual approaches** to secure web applications are always the same → the approaches learned here should be considered as a template to evaluate security features offered by other technologies and if you have to implement security features on your own
 - ...and no matter what technology you are using, it's always **paramount you really understand it** to make the right decisions with respect to security
- So is the Marketplace application **secure**?
 - It certainly has a **decent level of security** and most likely does not contain any major vulnerabilities, but there's room for **additional security measures**, e.g.:
 - There's no limit in login attempts an attacker can perform, so he may perform an **online password guessing** attack
 - The **credit cards are stored in the database in plaintext**, which exposes them to attacks