

# Security Lab – Java Web Application and Web Service Security: Extending Marketplace

## VMware

- This lab can be solved with the **Ubuntu image**, which you should start in networking-mode **Nat**. The remainder of this document assumes you are working with the Ubuntu image.
- You can basically solve this lab also on your own system, but several software packages have to be installed (IDE, Java, Payara, Mysql,...). All this is already installed on the Ubuntu image, so it's easiest to work with the image.

## 1 Introduction

In this lab, you are extending the Marketplace application from the lecture. The goal of the lab is that you get more experienced with using security features provided by Java EE and designing and implementing your own security functions. The lab is based on the final version from the lecture and it is assumed that you are familiar quite well with that version.

## 2 Basis for this Lab

- Download the following files from OLAT:
  - *Marketplace\_Lab.zip* and *Marketplace\_Lab-REST-Client.zip*
  - *Marketplace.sql* and *Marketplace\_UpdateEncryptedCreditCards.sql*
- Move the files to an appropriate location (e.g. in a directory *securitylabs* in the home directory */home/user*).
- To create the database schema and the technical user used by the Marketplace application to access the database, do the following (this can be repeated at any time to reset the database):
  - Open *MySQL Workbench*.
  - Click on the left on *Local Instance 3306* and enter *root* as password.
  - Choose *Open SQL Script...* in the menu *File* and select the downloaded file *Marketplace.sql*.
  - Click the *Execute* icon.
- Unzip the downloaded zip files. This results in two directories, *Marketplace\_Lab* and *Marketplace\_Lab-REST-Client*, which contain a Java EE and a Java project based on Maven. They should be importable in any modern IDE. The remainder assumes you are using NetBeans, which is installed on the Ubuntu image.
- Start *NetBeans* and open both projects:
  - *Marketplace\_Lab-REST-Client* can simply be opened in the standard way.
  - With *Marketplace\_Lab*, you first have to open *Marketplace\_Lab*, and then in addition all of its four sub-projects *Marketplace\_Lab-xyz* located within the directory *Marketplace\_Lab*.
- To build either one of the projects, right-click *Marketplace\_Lab* or *Marketplace\_Lab-REST-Client* and select *Clean and Build*.
  - Whenever you do some changes, the project is rebuilt automatically, so it's usually not necessary to use *Clean and Build* again. However, if «something doesn't work as it should», do again a *Clean and Build*.
  - Note that *Marketplace\_Lab* will be used from the beginning of the lab, *Marketplace\_Lab-REST-Client* will only be used in task 4 (see Section 8).
- To run either one of the projects, do the following:

- To run *Marketplace\_Lab*, right-click on *Marketplace\_Lab-ear* and select *Run*. If Payara is not yet running, it will be started, which takes some time.
- To run *Marketplace\_Lab-REST-Client*, right-click on *Test.java* and select *Run File*.
- Under the *Service* tab and expanding *Servers*, the Payara server is listed (it is listed as *GlassFish Server*). Sometimes, it may be necessary to restart Payara, do this with a right-click and *Restart*. Under *Applications*, you can also undeploy applications if necessary.
- The web application is reached with [http://localhost:8080/Marketplace\\_Lab-web](http://localhost:8080/Marketplace_Lab-web).
- The REST interface is reached with URLs below [https://localhost:8181/Marketplace\\_Lab-rest/rest/](https://localhost:8181/Marketplace_Lab-rest/rest/), e.g. [https://localhost:8181/Marketplace\\_Lab-rest/rest/products](https://localhost:8181/Marketplace_Lab-rest/rest/products).
- The application also contains a secure area (HTTPS). Ignore the certificate warning you'll get when accessing it, as this is only a testing environment. You can also accept the certificate permanently to get rid of the warning.

### 3 *Marketplace\_Lab* Project Organisation

As you have seen above, *Marketplace\_Lab* consists of multiple sub projects. The main reason for this is that we want to use different *web.xml* configuration files for the web application and the web service of the Marketplace, and this is only possible by using multiple sub projects. The four sub projects are organised as follows:

- *Marketplace\_Lab-ear*: This does not contain any code, but «holds the other sub projects together». You only use this sub project to run the application (see above).
- *Marketplace\_Lab-common*: Contains the common classes that are used both by the web application and the web service. This includes the entity classes, the database facades, service classes, validation classes, utility classes, and data transfer object (DTO) classes.
- *Marketplace\_Lab-web*: Contains everything specific to the JSF web application. This includes the Facelets, *web.xml*, and the backing bean classes.
- *Marketplace\_Lab-rest*: Contains everything specific to the RESTful interface. This includes *web.xml* and the classes to handle RESTful communication.

### 4 General Remarks

Your primary task is to solve the tasks below. But beyond implementing the tasks correctly from a security point of view, it is also expected that your extensions do not introduce typical web application vulnerabilities such as Cross-Site Scripting, SQL injection and so on. As you know from the lecture, this can easily be prevented using JSF and JPA correctly.

Also, make sure to study the provided skeletons of Facelets and Java classes so you understand them before you implement your extension.

The lower part of *NetBeans* provides an *Output* window, which contains a tab *Glassfish Server*. This provides messages of the Payara server including exceptions that are thrown within the application, which may be helpful during debugging of your program extensions.

Finally, the lab points section at the end contains the tests your program has to pass to get the lab points. When you have solved a part, it's strongly recommended you check whether the corresponding tests work as expected. If anything doesn't work, you should first fix the problem before continuing.

### 5 Task 1: Extending Admin Area

Your task is to extend the admin area with the following functions:

- Products can be listed, removed, and added. This is only allowed for users with the role *product-manager*. There are three users with this role:

- *donald* with password *daisy*
- *luke* with password *force*
- *snoopy* with password *woodstock*
- Authenticated users can change their password. This can be done with all roles.
- Besides the roles *marketing*, *sales*, and *productmanager*, there is a fourth role *burgerman*, which is assigned to one user:
  - *bob* with password *patrick*

## 5.1 Listing the Products

In the first step, *view/admin/admin.xhtml* should be extended such that the products are listed if a user with role *productmanager* clicks the *Admin area* button and accesses the view. Of course, clicking the button requires user authentication if the user is not logged in, but this functionality was already implemented during the lecture and is therefore included in the version you are using.

Depending on the role of the user, the view should appear as follows:

- Role *sales* (role *marketing* is similar, but does not display the *Remove purchase* buttons):

Admin Area				
Purchases:				
First Name	Last Name	Credit Card Number	Total Price (CHF)	
Ferrari	Driver	1111 2222 3333 4444	250000.00	<a href="#">Remove purchase</a>
C64	Freak	1234 5678 9012 3456	444.95	<a href="#">Remove purchase</a>
Script	Lover	5555 6666 7777 8888	10.95	<a href="#">Remove purchase</a>
<a href="#">Return to search page</a> <a href="#">Account settings</a> <a href="#">Logout</a>				

- Role *productmanager*:

Admin Area				
Marketplace products:				
Code	Description	Price (CHF)	Username	
0001	DVD Life of Brian - used, some scratches but still works	5.95	donald	<a href="#">Remove product</a>
0002	Ferrari F50 - red, 43000 km, no accidents	250000.00	luke	
0003	Commodore C64 - used, the best computer ever built	444.95	luke	
0004	Printed Software-Security script - brand new	10.95	donald	<a href="#">Remove product</a>
<a href="#">Return to search page</a> <a href="#">Add product</a> <a href="#">Account settings</a> <a href="#">Logout</a>				

- Role *burgerman* (any other role than *marketing* / *sales* / *productmanager*):

Admin Area	
Your role does not provide access to special functions in the admin area.	
<a href="#">Return to search page</a> <a href="#">Account settings</a> <a href="#">Logout</a>	

Perform the following steps to implement this functionality:

- For the two new roles *productmanager* and *burgerman* (in *Marketplace\_Lab-web*):
  - Add the role-name to group-name mappings in *glassfish-web.xml*

- Add *security-role* elements to *web.xml*
- Adapt *web.xml* such that the two new roles also have access to resources below *view/admin/*
- Extend *view/admin/admin.xhtml*. The version from the lecture has already been adapted such that for roles *sales*, *marketing*, and *burgerman*, the correct content is displayed. You must extend *admin.xhtml* such that it also shows the correct content for role *productmanager*. The element `<h:panelGroup>` for the roles *sales* and *marketing* should provide you with a good template how this can be done. As the backing bean, you can use *AdminProductBacking.java*, which is already completely implemented.
- Note that a product manager is only allowed to *delete the own products*. For instance, in the example above, *donald* should only be able to delete the products with codes 0001 and 0004 (where his name is listed in the *Username* table column / database column). The best way to do this with JSF is to present (using the *rendered* attribute) the *Remove product* button only in the rows that can be deleted by the current user, as shown above (within the Facelet, you can the name of the current user with *request.getRemoteUser()*). Why is this a secure solution?

Before you continue, check whether the applicable tests in the lab points section work as expected.

## 5.2 Adding new Products

Clicking the Add product allows product managers to add new products. The following must be true such that a new product is accepted:

- The code must contain exactly 4 characters (lowercase letters, uppercase letters, or digits).
- The description must contain at least 10 and at most 100 characters. Allowed characters are letters (lower- and uppercase), digits, the space character, and the special characters comma (,), quote (') and dash (-).
- The price must be a decimal number in the range 0 – 999'999.99, with at most two digits after the decimal point.
- In addition, the code must be different from all codes of already existing products.

The behavior should be as follows:

- Clicking the *Add product* button opens the view *view/admin/product/addproduct.xhtml*.

**Add Product**

Please provide the following information to add a product:

Code:

Description

Price:

- Entering invalid data for the input fields and clicking *Add product* result in validation errors being displayed:

## Add Product

Please provide the following information to add a product:

Code:  *Please insert a valid code (4 letters / digits)*

Description:  *Please insert a valid description (10-100 characters: letters / digits / - / , / ')*

Price:  *Please insert a price smaller than 999'999.99*

- Entering an already existing code also display an error:

## Add Product

*The product could not be added as a product with the same code already exists*

Please provide the following information to add a product:

Code:

Description:

Price:

- Entering valid data and clicking *Add product* adds the product and returns to the admin area:

## Admin Area

*The product could successfully be added*

Marketplace products:

Code	Description	Price (CHF)	Username	
0001	DVD Life of Brian - used, some scratches but still works	5.95	donald	<input type="button" value="Remove product"/>
0002	Ferrari F50 - red, 43000 km, no accidents	250000.00	luke	
0003	Commodore C64 - used, the best computer ever built	444.95	luke	
0004	Printed Software-Security script - brand new	10.95	donald	<input type="button" value="Remove product"/>
0005	Super Vulnerability Scanner	9999.95	donald	<input type="button" value="Remove product"/>

Perform the following steps to implement this functionality:

- Adapt *web.xml* such that only role *productmanager* can access resources below *view/admin/product/* and only over HTTPS. Make sure to insert the same URL pattern also in the security-constraint identified as *Restricted Methods*.
- Extend *view/admin/products/addproduct.xhtml*. This should be done similar as in *view/public/secure/checkout.xhtml*. As the backing bean, you can use *AddProductBacking.java*, which is already completely implemented. By inspecting this bean, you can see in method *addProduct* how the name of the current user is determined and set in the *Product* entity before the entity is inserted into the database.
- The backing bean uses the method *insertProduct* in *AdminProductService.java*. You must implement this method. The method has to check whether the product code of the new product already exists. If this is the case, return false and don't insert the product into the database. Otherwise, in-

sert it and return true. The reason why putting this functionality in a separate service class is because it will also be used by the web service later.

- Add Bean Validation annotations<sup>1</sup> to the instance variables *code*, *description*, and *price* in *Product.java* to enforce the restrictions described above (except the uniqueness of the code, which has already been handled in *AdminProductService.java*). The *code* and *description* can be handled with `@Pattern`. For the *price*, it's best to use `@DecimalMin`, `@DecimalMax`, `@Digits`, and `@NotNull`.

Before you continue, check whether the applicable tests in the lab points section work as expected.

### 5.3 Account Settings – Change Password (optional)

This subtask is optional for those who want to implement an entire functionality including backing bean and web.xml configuration completely on their own. It should behave as follows:

- Clicking the *Account setting* button in the Admin area opens a view *view/admin/account/accountsettings.xhtml*. The view already exists but does not provide any functionality so far. The view should provide the user with functionality to change the own password.
- Accessing the view should be available to all roles (also *burgerman*), but not to non logged-in users.
- If you are briefly leaving your computer and are working in the Admin area, then it should not easily be possible for another person to quickly use your computer to change your password. Think about a way to solve this.
- If you want, you can require a minimal password strength for the new password. Use Bean Validation for this.
- Place the actual functionality to change the password into a service class, so it could also be used by the web service later.

In case you are implementing this subtask: Before you continue, test the following:

- Only logged-in users can access *view/admin/account/accountsettings.xhtml*. Non logged-in users should be redirected to the login page.
- If you are using a minimal password strength, check whether it works.

## 6 Task 2: Limit Online Password Guessing Attacks

Right now, an attacker can perform as many login attempts as he likes, which allows online password guessing attack. Note that the usage of script throttles the attacker already to a certain degree, but it is desirable to have a more effective throttling mechanism.

Various throttling mechanisms can be imagined:

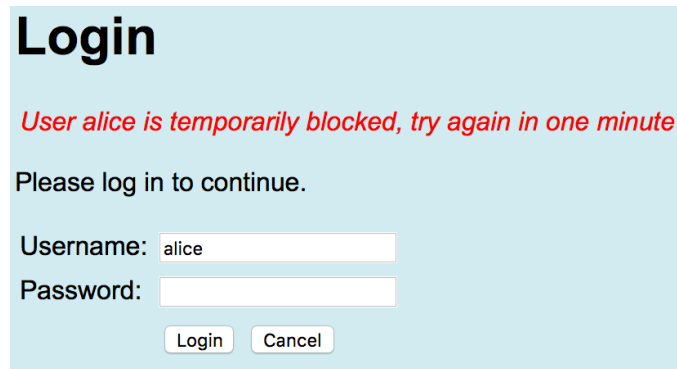
- Throttle the attacker after a certain number of failed logins with the same session ID. This defense is not effective because the attacker can easily circumvent this by using a fresh session for every attempt.
- Throttle the attacker after a certain number of failed logins from the same IP address. This is effective against weak attackers, but determined attackers can still try large amounts of passwords by performing the logins from a large number of different IP addresses. This is typically done using a botnet.
- After a certain number of failed logins per user name, block login with that user name for a certain time. This defense is very effective even against powerful attackers as there is a clear upper bound on the number of login attempts per user name and per time interval.

---

<sup>1</sup>Details about the annotations: <https://docs.oracle.com/javaee/7/tutorial/bean-validation001.htm>

The throttling mechanism you have to implement is specified as follows:

- After three failed attempts for a particular user name, login with that user name is completely blocked for 60 seconds. This is acceptable for legitimate users if they accidentally enter a wrong password three times (which rarely happens) and it does not block users permanently, which could be abused for DoS attacks.
- If a blocked user tries to log in, the message “User xyz is temporarily blocked, try again in one minute” is displayed.



The screenshot shows a light blue login form titled "Login". Below the title, a red message states: "User alice is temporarily blocked, try again in one minute". Below this, it says "Please log in to continue." There are two input fields: "Username:" with the value "alice" and "Password:". At the bottom, there are two buttons: "Login" and "Cancel".

- After 60 seconds, a user can login again, but entering the wrong password results in blocking the user again for 60 seconds (so when a user is un-blocked after 60 seconds, he gets only one attempt).
- If a user logs in successfully, the application forgets previously false login attempts for this user, i.e. the user gets again three login attempts without being blocked.

To implement the mechanism, you should do the following:

- To keep track of failed logins and blocked users, a service class *LoginThrottlingService.java* is used, of which a skeleton already exists. This class is an EJB with the *@Singleton* annotation, which guarantees there's only one instance in the entire application. In addition, *@Singleton* guarantees thread-safe usage, which means you don't have to deal with synchronized methods. The class provides skeletons of three public methods:
  - *public void loginFailed(String username)*: This method is called to inform that login with username has failed. This should increment the login failed counter for that user.
  - *public void loginSuccessful (String username)*: This method is called to inform that login with username has succeeded. This should remove all stored information about failed logins of the user.
  - *public boolean isBlocked (String username)*: This method returns whether user username is currently blocked.
- First you should complete the implementation of this class. The three methods above are just the public interface of the class; it's up to you how to implement it internally.
- Once the class is implemented, you can inject it in *AuthenticationBacking.java* (which handles the login) and use the three public methods described above in the *login* method to limit the number of login attempts per username.

Note that with this mechanism, an attacker is significantly slowed down because even using as many threads, computers, or IP addresses, he can at most perform 1'440 password attempts for a given user name per day.

Take the following implementation hints into account:

- It's quite easy to introduce DoS opportunities with such throttling functionality. Therefore, it is not recommended to wait (actively or passively with *sleep()* or *wait()*) during the blocking time as

this may exhaust all available threads. Instead, the web application should return the response immediately to the client in case a login attempt is blocked.

- Implement the mechanism as lightweight as possible so it consumes only little resources. In particular, don't use the database to keep track of failed logins but implement everything "in memory" in the class *LoginThrottlingService.java*.
- Do only handle existing usernames. If a user logs in with a non-existing username, ignore this in your blocking mechanism (don't keep track of failed logins). Otherwise, the attacker could exhaust your tracking mechanism by submitting large amount of fictitious usernames.
- To check the correctness of your approach, you can print the internal State of *LoginThrottlingService.java* whenever a user logs in (successfully or not) by using *System.out.print()*. The output should be visible in the Payara Server output window in NetBeans.

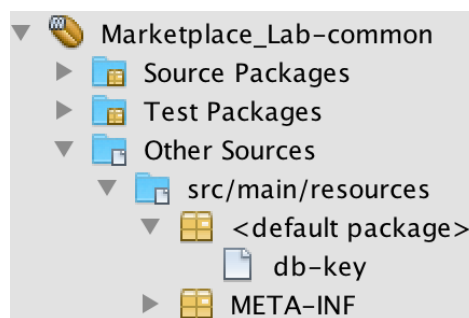
Before you continue, check whether the applicable tests in the lab points section work as expected.

## 7 Task 3: Encrypt Credit Card Numbers

Currently, the credit card numbers are stored in plaintext in the database. This means an attacker can possibly get access to them via SQL injection, by breaking into the database server, by getting physical access to the database disk etc. To reduce this risk, the application should store credit card number only in encrypted form. Within the application, the credit card number should of course still be displayed in plaintext.

The mechanism you have to implement is specified as follows:

- AES-CBC with a 128-bit key is used to encrypt the credit card numbers. The initialization vector is stored together with the encrypted credit card numbers (by concatenating IV and ciphertext). Use a fresh IV for each credit card number.
- Do not modify the database schema. The current column *CreditCardNumber* in table *Purchase* is a *varchar(100)* field, which is big enough to store the encrypted credit card number. The binary ciphertext (including the IV in front of it) is stored in base64-encoded format.
- The key is stored in a file, in hex-format (e.g. 8E60AF3641D394A104FF356C90AC25B4). This is not perfect but is often done this way in practice. On a productive system, one would configure Payara to run with its own user id and make sure only the Payara user can access that file. The key (*db-key*, don't change it) is already available in a location where it can easily be accessed from within the code:



- To provide the actual encryption and decryption, a service class *AESCipherService.java*<sup>2</sup> is used; a skeleton already exists. A static initializer block is already implemented which reads the key from the file system. For encryption and decryption, the methods *encrypt* and *decrypt* are used, for which skeletons also exists.

<sup>2</sup> As it is not possible to inject CDI beans or EJBs into *AttributeConverters*, we use a "normal" class here. <http://stackoverflow.com/questions/31549783/inject-not-working-in-attributeconverter>.



- JPA provides *AttributeConverters*, which can be used to apply operations on data just before they are written to the database and just after they are read from the database. This can ideally be used to encrypt the credit card just before writing it and decrypting it just after reading it. For this, the *AttributeConverter* interface must be implemented. This has already been done with class *AESConverter.java* (in package *ch.zhaw.securitylab.marketplace.model*) and you shouldn't adapt this class. Inspecting the two methods in the class, you can see the following:
  - Method *convertToDatabaseColumn* encrypts the data using the *encrypt* method of *AESCipherService* and returns the Base64-encoded encrypted string (which includes the IV and the encrypted credit card).
  - Method *convertToEntityAttribute* decodes the Base64-encoded encrypted data (consisting of IV and encrypted credit card) and decrypts it using the *decrypt* method of *AESCipherService*.

To complete the implementation, you should do the following:

- Implement the methods *encrypt* and *decrypt* in *AESCipherService*. The Javadoc comments of the methods explain the purpose of the parameters and return values and what the methods should do.
- In addition, you must instruct the field *creditCardNumber* in *Purchase.java* to use *AESConverter.class*. Do this with the annotation *@Convert* as follows:

```
@CreditCardCheck
@Convert(converter = AESConverter.class)
private String creditCardNumber;
```

- To encrypt the credit card numbers of the already existing purchases in table *Purchase*, execute *Marketplace\_UpdateEncryptedCreditCards.sql* in *MySQL Workbench*. This script simply replaces the plaintext credit card numbers with AES-encrypted credit card numbers corresponding to 1111 2222 3333 4444 (same value for all rows).

Before you continue, check whether the applicable tests in the lab points section work as expected.

## 8 Task 4: RESTful API

The final task is to provide a RESTful API for product managers and to make sure that login throttling is also done when the REST interface is used.

### 8.1 RESTful API for Product Managers

To do this, you complete the REST class *AdminProductRest.java*, which is available as a skeleton. You have to implement three methods:

- GET to get all products. For this, a new data transfer object (DTO) *AdminProductDto.java* is already available, which sends JSON objects in the following form to the client:

```
[{"code":"0001","description":"DVD Life of Brian - used, some scratches but still works","price":5.95,"username":"donald"}, {"code":"0002","description":"Ferrari F50 - red, 43000 km, no accidents","price":250000.00,"username":"luke"}]
```

The URL to be used is *rest/admin/products*.

- POST to create a new product. For this, *ProductDto.java* is used (this has already been used before by normal users to get search results) and a new product is sent to the application as follows:

```
{"code":"0005","description":"Super Vulnerability Scanner","price":9999.95}
```

The URL to be used is *rest/admin/products*.

- DELETE to delete a product. Of course, a product manager can only delete the own products, which must be checked by the application. The product to delete is identified by its primary key

ProductID.

The URL to be used is *rest/admin/products/{id}*.

To test your REST interface, there's a test class available in *Marketplace\_Lab-REST-Client*. It is an extension of the test class used in the lecture and tests the already existing interface and also your extensions. It uses various combinations (non-authenticated users, authenticated users, valid / invalid input data etc.). For each test, the result is PASSED or FAILED; Passed means that the test produced the expected response from the application. To run the tests, simply run *Test.java* in Project *Marketplace\_Lab-REST-Client* (as described in Section 2). Of course you can run this test whenever you have added or modified one of the REST methods to check whether it works as intended. It's usually a good idea to reset the database schema and content before running the test (using both SQL scripts).

In the following, additional details of the three methods and the steps to implement them are given. Note that you don't have to deal with authentication, as functionality was already implemented during the lecture and is therefore included in the version you are using. Also, the role *productmanager* has already been added to *glassfish-web.xml* and *web.xml*, so you don't have to do this.

*GET rest/admin/products:*

- This should be done in a similar way as the GET method in *AdminPurchaseRest.java*.
- Make sure to adapt *web.xml* such that users with role *productmanager* are allowed GET access.
- In addition, adapt *web.xml* to make sure that the role *productmanager* can delete the authentication token. To this, add the role to the security constraint identified as «Marketplace REST authenticate DELETE».
- If a user with a different role tries to access the resource, he gets a 403 response with JSON content `{"error": "Access denied"}`. This was already implemented in general during the lecture and is therefore included in the version you are using.
- If a non-authenticated user or a user with an invalid authentication token tries to access the resource, he gets a 401 response with JSON content `{"error": "Authentication required"}`. Just like above, this is already handled in the version you are using.

*POST rest/admin/products:*

- The POST method in *PurchaseRest.java* should provide some hints how this can be implemented.
- The received *ProductDto* should be validated before processing it. To do so, copy the validation annotations from the entity *Product.java* (see task 1) into *ProductDto.java* and use the correct annotation with the parameter of the POST method.
- Make sure to set the username of the current user in the entity *Product* before inserting it into the database. Method *removeAuthenticationToken* in *AuthenticationRest.java* provides hints how to get the username.
- To actually insert the entity *Product*, use the *insertProduct* method in *AdminProductService.java* you implemented above (see task 1).
- Make sure to adapt *web.xml* such that users with role *productmanager* are allowed POST access.
- If a non-authenticated user or a user with a role other than *productmanager* tries to access the resource, the behavior should be the same as with GET above and this should work out of the box.
- If the *insertProduct* method of *AdminProductService.java* returns false, a 400 response with JSON content `{"error": "The product could not be added as a product with the same code already exists"}` should be sent to the client. You can do this by throwing an *InvalidParameterException* in the POST method with the desired error message. The processing of the exception and the generation of the correct response was also done in the lecture and is therefore available in your version (in *CustomExceptionMapper.java*).

- If a validation error occurs, a 400 response with JSON data containing the validation messages should be sent to the client, e.g. `{"error":"Please insert a price with at most two decimal places, Please insert a price smaller than 999'999.99"}`. Assuming you inserted the validation annotations correctly in *ProductDto.java* (see above), this should be handled correctly in the version from the lecture (in *ConstraintViolationExceptionMapper.java*).

*DELETE rest/admin/products/{id}*:

- This should be done in a similar way as the DELETE method in *AdminPurchaseRest.java*.
- Use a validation annotation with the method parameter *id* to make sure the *id* is between 1 and 999'999 (just like in *AdminPurchaseRest.java*).
- Make sure that users can only delete their own products.
- To delete the product, you'll have to extend *Product.java* and *ProductFacade.java* to get a Product object with a specific ProductID.
- Make sure to adapt *web.xml* such that users with role *productmanager* are allowed DELETE access.
- If a non-authenticated user or a user with a role other than *productmanager* tries to access the resource, the behavior should be the same as with GET above and this should work out of the box.
- If an *id* is used that is not in the permitted range, a 400 response with JSON content `{"error":"The ProductID must be between 1 and 999'999"}` should be sent to the client. Assuming you used the validation annotations correctly with parameter *id* (see above), this should work without further adaptations as it was already implemented in the lecture (in *ConstraintViolationExceptionMapper.java*).
- If an *id* is used for which no product exists, a 400 response with JSON content `{"error":"The product with ProductID = '999' does not exist"}` should be sent to the client. You can do this by throwing an *InvalidParameterException* in the DELETE method with the desired error message. The processing of the exception and the generation of the correct response is already handled correctly (in *CustomExceptionMapper.java*).
- If a user with role *productmanager* tries to delete a product that does not belong to him, a 403 response with JSON content `{"error":"Access denied, only the own products can be deleted"}` should be sent to the client. You can do this by throwing an *AccessControlException* in the DELETE method with the desired error message. To process this exception, you have to extend *CustomExceptionMapper.java*. If the received Exception is of type *AccessControlException* (use *instance of*), then use HTTP status 403 (*Status.FORBIDDEN*), otherwise use 400 (*Status.BAD\_REQUEST*, which is currently always used).

Before you continue, test the following:

- When using *Test.java*, all tests should show PASSED (except the login throttling tests, as login throttling has not been implemented yet for the RESTful API).

## 8.2 Limit Online Password Guessing Attacks

Finally, make sure that the RESTful API cannot be abused for online password guessing. As you have implemented *LoginThrottlingService.java* above (see task 2), not much remains to do:

- In the POST method in *AuthenticationRest.java*, use *LoginThrottlingService.java* in a similar way as you have done in the *login* method in *AuthenticationBacking.java*.
- If a user is blocked, a 400 response with JSON content `{"error":"User xyz is temporarily blocked, try again in one minute"}` should be sent to the client. You can do this by throwing an *InvalidParameterException* in the POST method with the desired error message. The processing of the exception and the generation of the correct response is already handled correctly (in *CustomExceptionMapper.java*).

- If the user uses a wrong username or password when the username is not blocked, a 400 response with JSON content `{"error": "Username or password wrong"}` should be sent to the client. The corresponding *InvalidParameterExceptions* are already thrown in the POST method in *AuthenticationRest.java* and are already handled correctly in your version (in *CustomExceptionHandler.java*).

Before you continue, test the following:

- When using *Test.java*, all tests should show PASSED.

## Lab Points

For **6 Lab Points**, you must demonstrate to the instructor that your Marketplace application runs according to the specifications and passes the tests listed in the table below:

- You get 2 points for solving *Task 1: Extending Admin Area* (see Section 5). Note that the *Change Password* functionality is optional and not required to get the 2 points.
- You get 1 point for solving *Task 2: Limit Online Password Guessing Attacks* (see section 6).
- You get 1 point for solving *Task 3: Encrypt Credit Card Numbers* (see section 7).
- You get 2 points for solving Task 4: RESTful API (see section 8).

In addition, perform a static code analysis of your final version using Fortify SCA and show the results to the instructor. As Fortify requires quite a lot of memory, it's best to close all other applications before running the analysis as otherwise, Fortify may complain about a lack of memory. Perform the analysis as follows:

- Start Fortify SCA in a terminal by entering the following as *user*:  

```
cd /opt/HP_Fortify_SCA_and_Apps_16.10/bin
./auditworkbench
```
- When the window opens, click *Advanced Scan...*. Select the source code directory of sub project Marketplace\_Lab-common, which is *Marketplace\_Lab-common/src* (make sure to include the *src* directory in the path selection by double-clicking it the path selection dialogue).
- In the following *Advanced Static Analysis* window, use *Add Directory...* three times to add the *src* directories of the other three sub projects: *Marketplace\_Lab-ear/src*, *Marketplace\_Lab-rest/src*, and *Marketplace\_Lab-web/src*.
- Select *JDK 1.8* in the *Java Version* field and click *Scan*.

The analysis will take a few minutes. As soon as the analysis has been completed, the results are shown in the *Audit Workbench* window. To make sure all findings are listed, select *Security Auditor View* at the *Filter Set* setting at the top left of the window. Then, select the *green* tab in the window at the top left, which shows all findings, and then *OWASP Top 10 2013* in the *Group By* drop list, which arranges the findings according to OWASP Top 10.

The listed OWASP Top 10 vulnerabilities must not include any critical ones (e.g. injection, XSS, passwords sent in plaintext,...). Acceptable are issues with respect to *web.xml* (these are mostly false positives) and CSRF (also false positives). If critical issues, are listed, 2 points will be deducted.

To get the points, your version must pass the following tests:

Test	Passed
<b><i>Task 1: Extending Admin Area</i></b>	
If a non-authenticated user accesses <i>view/admin/admin.xhtml</i> , the user is redirected to the	

login page.	
Login in with <i>alice/rabbit</i> shows the purchases.	
Accessing <i>view/admin/product/addproduct.xhtml</i> as user <i>alice</i> results in a 403 error.	
Login in with <i>bob/patrick</i> shows an «empty» <i>Admin area</i> .	
Login in with <i>donald/daisy</i> shows the products and the <i>Add product</i> button.	
The <i>Remove product</i> button is only visible in the rows that correspond to <i>donald</i> 's products.	
Removing a product as user <i>donald</i> removes the product.	
When <i>donald</i> enters a new product, the validation rules work correctly and appropriate error messages are shown.	
Creating a new product as <i>donald</i> and entering an already existing code does not work and shows an error message.	
Accessing <i>/view/admin/product/addproduct.xhtml</i> over HTTP and as user <i>donald</i> results in a redirection to HTTPS.	
<b>Task 2: Limit Online Password Guessing Attacks</b>	
Log in with the same user four times in a row using a wrong password. Three times, you should get the message about wrong username or password. After the fourth login attempt, you should get the message that the user is blocked.	
Log in again with the same user, but using the correct password. The user should still be blocked.	
Wait more than 60 seconds and log in again, with a wrong password. You should get the message about wrong username or password.	
Log in again with the same user, but using the correct password. You should get the message that the user is blocked.	
Wait more than 60 seconds and log in again, with the correct password. Login should be successful	
<b>Task 3: Encrypt Credit Card Numbers</b>	
Make two purchases using the credit card number 1111 2222 3333 4444 both times. Then, check with MySQL Workbench that credit cards are stored in encrypted form in the database. The entries should look similar as below (the individual entries should all be different due to different IVs):	
<p>CreditCardNumber</p> <pre>jE9cBwBynmsQLw1N5avp2xW1Vg0W1vOi4V+W6h2xsm10VrpjrPY8OukN/rdl0MhH /nEcHkZhCF79Psi4eMBLXCvoiNaHQ4YnCcQlR3CM8hfH70r610Ize5rziQ73vMx8/ JVFzWMoivgSfFQqFsijPhJQl2U3iqUbZ97Zu6X3bbM+Jh1xaUtlLaDlprlA6WPre</pre>	
Check that within the application, the credit card numbers are shown in plaintext.	
<b>Task 4: RESTful API</b>	
When running <i>Test.java</i> in project <i>Marketplace_Lab-REST-Client</i> , all tests should show PASSED.	