Security Lab – Developing Secure Web Applications and REST APIs

Virtual Machine

This lab can be done with the VM that is provided to you. The remainder of this document assumes you are working with this VM.

You can basically also do this lab on your own system, but several software packages have to be installed (*IntelliJ* or another IDE, Java, MySQL,...). All this is already installed on the provided VM, so it's easiest to work with this VM.

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 developing secure web applications and REST APIs by using the security features provided by Spring and by 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 Moodle:
  - *Marketplace_Lab.zip* and *Marketplace_Lab-REST-Test.zip*
  - *Marketplace.sql* and *Marketplace_UpdateEncryptedCreditCards.sql*
- Move the files to an appropriate location (e.g., into a directory *securitylabs* in the home directory */home/hacker*).
- To create the Marketplace database and the technical user used by the Marketplace application to access the database, do the following in a terminal as user *hacker* (this can be repeated at any time to reset the database):
  - Start the MySQL database with *sudo service mysql start* (to stop it, use *sudo service mysql stop*).
  - Us the *mysql* command line tool to access the MySQL database by entering *mysql -u root -p*, followed by the password *root*.
  - Load the script *Marketplace.sql* with *SOURCE Marketplace.sql* (make sure to use the correct path to the file), which creates the Marketplace database and technical database user.
  - Check that the Marketplace database was created with *SHOW DATABASES;* (*marketplace* should be listed among other databases).
  - Make the Marketplace database the default database with *USE marketplace*.
  - To list the tables, use *SHOW TABLES;* (which should list the four tables on the Marketplace database).
  - Use, e.g., *SELECT * FROM Product;* to list the content of a table.
  - To reset the Marketplace application to its initial state, simply execute *SOURCE Marketplace.sql* again.
- Unzip the downloaded file *Marketplace_Lab.zip*, which results in directory *Marketplace_Lab* that contains the Marketplace web application and the Marketplace REST API. This is a Maven project consisting of three modules. *Marketplace-common* includes code that is used by the web application and the REST API. *Marketplace-web* includes everything that is specific for the web application and *Marketplace-rest* everything that is specific for the REST API (see also Section 3).
- Unzip the downloaded file *Marketplace_Lab-REST-Test.zip*. This results in directory *Marketplace_Lab-REST-Test*, which contains a Maven project that is used to test the REST API.

- The following steps assume that you are using *IntelliJ*, which is installed on the VM.
- If you haven't activated *IntelliJ* yet, do so now using the description in *SecLab_Using-VM.pdf.*
- Start *IntelliJ* (*Applications* menu top left => *All Applications* => *IntelliJ IDEA Ultimate Edition*) and open project *Marketplace_Lab*.
- To build and run either the web application or the REST API, select *Marketplace Web* or *Marketplace REST* on the right side of the title bar and click the *Run* button next to it. Once it is running, this is indicated in the title bar with a green *Rerun* button. Clicking the red *Stop* button terminates the application. As the web application and the REST API both use port 8443, only one of them can run at any time. In the beginning, only the web application is used. The REST API will be used later in this lab (task 4, see Section 9).
- The *Console* window at the bottom shows any messages created by the running application. It also SOURCE shows thrown exceptions and may help you to identify problems when starting or running the application. If you use *System.out.println()* in the code, then the content will be printed to this *Console* window.
- To build and run *Marketplace_Lab-REST-Test* (this will be used in task 4, see Section 9), open the project in *IntelliJ* (make sure to open it in a new window) and click the *Run* button on the right side of the title bar. This executes method *main* in class *Test*, which contains all the tests.
- Any changes you make in the code are saved automatically. To reflect code changes in the running web application or REST API, you must rerun it (either by first stopping it and running it again, or simply by clicking the *Rerun* button).
- The web application is reached with *http://localhost:8080* (or *https://localhost:8443*) 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. The REST API is reached with URLs below *https://localhost:8443/rest/*, e.g., *https://localhost:8443/rest/products*.
- The web application includes an "attack page" reachable at *https://localhost:8443/public/attacks.* This can be used to easily do a few tests. To use this, open the page in a separate browser tab and click the *Do Attack* buttons. Before you click such a button, it's recommended to do a reload of the page, especially after having logged in or out. Throughout the remainder of this document, you will be informed when to use such a button to do a specific test.

3 Project Organization

As you have seen above, the *Marketplace_Lab* project consists of three modules. This is mainly done to re-use code that is used by both the web application and the REST API. The three modules contain the following:

- *Marketplace-common* includes everything that is used by both the web application and the REST API (common config classes, model classes, repository classes, service classes, validation classes)
- *Marketplace-web* includes everything that is specific for the web application (web application-specific config and controller classes, *main* class, templates and *application.properties*)
- *Marketplace-rest* includes everything that is REST API specific (REST API-specific config, controller and service classes, *main* class and *application.properties*)

4 General Remarks

Your primary objective is to solve the four tasks in Sections 6 - 9. In the first three tasks, you'll extend the Marketplace web application and in the fourth task, you'll extend the Marketplace REST API.

Beyond solving the tasks correctly, it is also expected that your extensions do not introduce typical vulnerabilities such as Cross-Site Scripting, SQL injection, access control problems and so on. As you know from the lecture, this can be prevented by correctly using the security features that are offered by Spring and Thymeleaf.

Make sure to study the provided skeletons of Thymeleaf templates and Java classes so you understand them before you implement your extensions.

Finally, the lab points section at the end contains the tests your program must 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 Users, Passwords and Roles

The following table lists the available users, their passwords, and the roles they get after successful login. For instance, user *alice* has the password *rabbit* and gets the role *SALES* after successful login. The passwords are shown here in plaintext, but they are stored securely in the database (using bcrypt, as discussed in the lecture).

Username	Password	Role
alice	rabbit	SALES
bob	patrick	BURGERMAN
daisy	yellow	PRODUCTMANAGER
john	wildwest	SALES
luke	force	PRODUCTMANAGER
robin	arrow	MARKETING
snoopy	woodstock	MARKETING

6 Task 1: Extending Admin Area

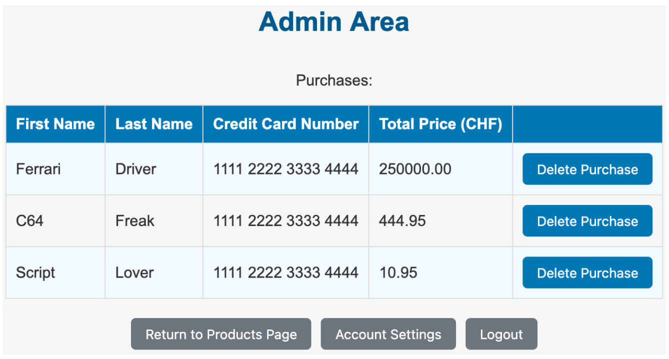
Your first task is to extend the admin area of the web application so that products can be listed, deleted, and added. This is only allowed by users with the role *PRODUCTMANAGER*. In addition, every authenticated user (with any role) should be able to change his password.

6.1 Listing and Deleting Products

In the first step, the admin area should be extended such that the products are listed if a user with role *PRODUCTMANAGER* accesses the admin area. Of course, this 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 admin area should appear as follows:

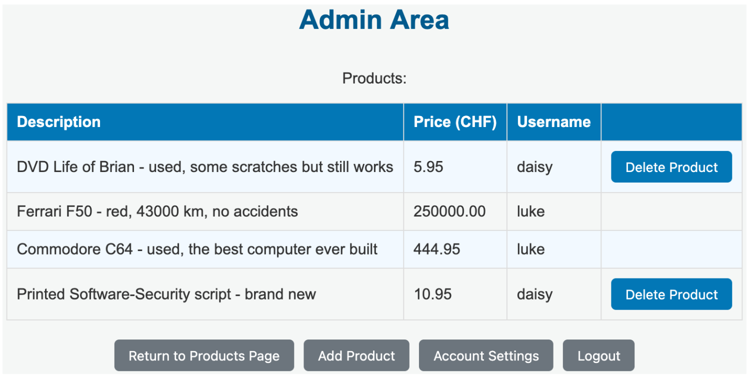
- Role *SALES* (role *MARKETING* is similar, but does not display the *Delete Purchase* buttons):


Admin Area

Purchases:

First Name	Last Name	Credit Card Number	Total Price (CHF)	
Ferrari	Driver	1111 2222 3333 4444	250000.00	Delete Purchase
C64	Freak	1111 2222 3333 4444	444.95	Delete Purchase
Script	Lover	1111 2222 3333 4444	10.95	Delete Purchase

- Role *PRODUCTMANAGER*:

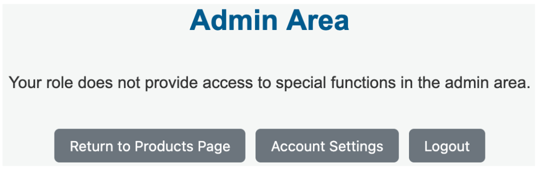


Admin Area

Products:

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

- Role *BURGERMAN* (any other role than *MARKETING* / *SALES* / *PRODUCTMANAGER*):



Admin Area

Your role does not provide access to special functions in the admin area.

Do the following to implement this functionality:

- Extend method *securityFilterChain* in class *SecurityConfig* (in *Marketplace-web*) to grant the two new roles *PRODUCTMANAGER* and *BURGERMAN* access to resources below */admin/*.
- Extend method *adminPage* in class *AdminController* (in *Marketplace-web*) so it sets two model attributes if the role of the current user is *PRODUCTMANAGER*. The first attribute should have name *products* and contains a list of all *products* in the database. To get them, class *ProductService* (in *Marketplace-common*) provides a method *findAll*. The second attribute should have name *username* and contains the username of the current user. To get this name, class *UtilityService* (in *Marketplace-common*) provides a method. Note that *UtilityService* also contains a method to get the role of the current user, which is used at the beginning of method *adminPage* to distinguish between different roles. Also note that objects *productService* of type *ProductService* and *utilityService* of type *UtilityService* are already injected in class *AdminController*.
- Extend *template/admin/admin.html*. The version from the lecture has already been adapted such that for roles *MARKETING*, *SALES* and *BURGERMAN*, the correct content is displayed. You must extend *admin.html* such that it also shows the correct content for role *PRODUCTMANAGER*. The <div> element for roles *MARKETING* and *SALES* should provide you with a good template how this can be done. To delete a product, a POST request is sent to, e.g., */admin/deleteproduct/3*, where 3 is the ID of the product to delete.
- Make sure that a product manager is only allowed to *delete the own products*. For instance, in the example above, *daisy* is logged in and she should only be able to delete the 1st and 4th products in the list, where her name is listed in the *Username* column of the table. Therefore, make sure that in *template/admin/admin.html*, the *Delete Purchase* button is only shown in the rows that can be deleted by the current user, as shown above. You can do this by using *th:if="${product.username == username}"* in the corresponding *<form>* tag (here, the *username* on the right side of == corresponds to the username you set as a model attribute in method *adminPage* in class *AdminController*).
- Complete method *deleteProduct* in class *AdminController*, which handles POST requests to delete a product. You can implement this similar to method *deletePurchase*. However, you must additionally check that users can only delete their own products, as not including the corresponding *Delete Product* buttons in *template/admin/admin.html* is of course not enough as this can easily be circumvented by submitting the desired POST request manually. To get the name of the current user, you can once more use class *UtilityService*. If a user tries to delete a product that is not his

own product, throw an *AccessDeniedException* by using *throw new AccessDeniedException("");*, which creates a response with HTTP status code 403.

- Finally, extend method *securityFilterChain* in class *SecurityConfig* once more to make sure that only users with role *PRODUCTMANAGER* can access the resource to delete a purchase. Be careful to insert this rule at the right position, as the access control rules are evaluated from top to bottom!

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

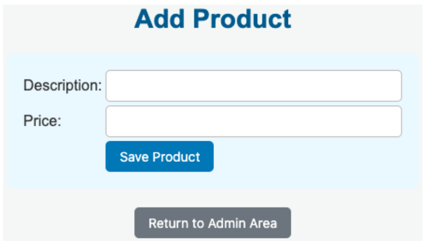
6.2 Adding new Products

Clicking the *Add Product* button in the admin area allows product managers to add new products. The following must be true such that a new product is accepted:

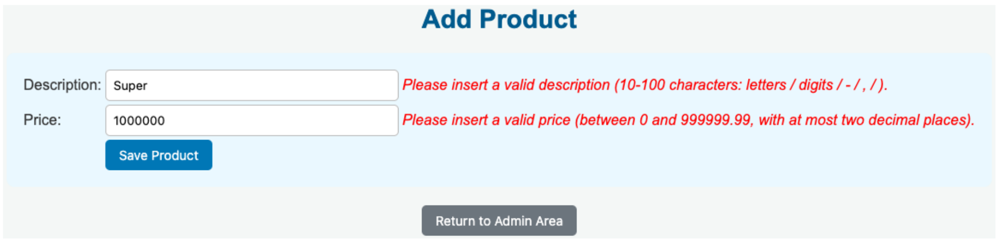
- The description must contain at least 10 and at most 100 characters. Allowed characters are letters (lower- and uppercase), digits, the space character, and the special characters comma (,), quote (') and dash (-).
- The price must be a decimal number in the range 0 – 999’999.99, with at most two digits after the decimal point.

The behavior should be as follows:

- Clicking the *Add Product* button shows *template/admin/addproduct.html*.

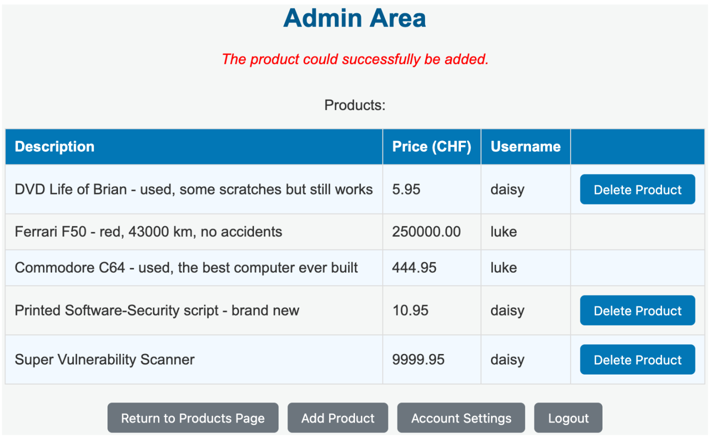


- Entering invalid data for the input fields and clicking *Save Product* results in validation errors being displayed:



Description:	Super	Please insert a valid description (10-100 characters: letters / digits / - / , /).
Price:	1000000	Please insert a valid price (between 0 and 999999.99, with at most two decimal places).

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



The product could successfully be added.

Products:

Description	Price (CHF)	Username	
DVD Life of Brian - used, some scratches but still works	5.95	daisy	Delete Product
Ferrari F50 - red, 43000 km, no accidents	250000.00	luke	
Commodore C64 - used, the best computer ever built	444.95	luke	
Printed Software-Security script - brand new	10.95	daisy	Delete Product
Super Vulnerability Scanner	9999.95	daisy	Delete Product

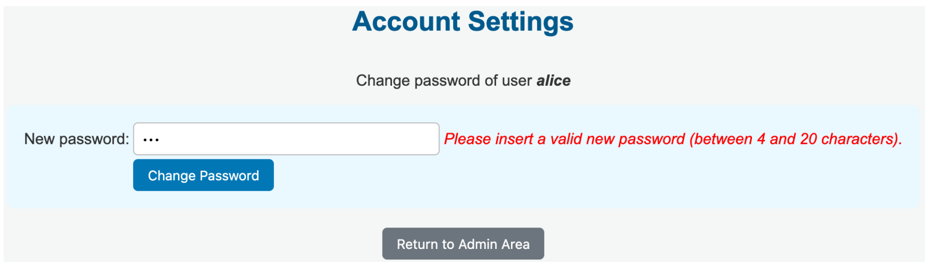
Do the following to implement this functionality:

- To get to the *Add Product* page, a GET request to */admin/addproduct* is used. When clicking the *Save Product* button on the *Add Product* page, a POST request is sent to */admin/saveproduct*. Extend method *securityFilterChain* in class *SecurityConfig* to make sure that only users with role *PRODUCTMANAGER* can use this functionality.
- Method *addProductPage* in class *AdminController* to get to the *Add Product* page is already provided. Inspect it so you know how it works. Next, complete *template/admin/addproduct.html*. This should be done similar as in *template/public/checkout.html*.
- Add Bean Validation annotations[1] to the instance variables *description* and *price* in class *Product* to enforce the restrictions described above and to make sure that only valid products are inserted into the. Using the Bean Validation annotations is done similar as in class *Purchase*. The *description* can be handled with *@NotNull* and *@Pattern*. For the *price*, it's best to combine *@NotNull*, *@PositiveOrZero* and *@Digits* (simply use all of them above the attribute *price*, which enforces them all). Make sure to use the same validation error messages as illustrated in the screenshot.
- As a final step, complete method *saveProduct* in class *AdminController*, which handles POST requests to save the new product in the database. First, make sure that you use the correct annotation above the method to handle the desired requests, Also, an annotation must be added to the method parameters. Before saving the product in the database, you must set its username to the username of the current user. To actually save the product, method *save* in class *ProductService* is already provided. Method *savePurchase* in class *PublicController* may give you some hints how to do this.

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

6.3 Account Settings – Change Password

In the admin area, there's a button *Account Settings*. Clicking this button opens *template/admin/accountsettings.html* as illustrated below:


Account Settings
Change password of user ***alice***
New password: •••
Please insert a valid new password (between 4 and 20 characters).
Change Password
Return to Admin Area

The page allows authenticated users with any role to change their password. Methods *accountSettingsPage* and *changePassword* in class *AdminController* to get to this page and to change the password are already implemented; study them as a basis for the code adaptations you'll do. Note that there are minimal requirements for passwords in the Marketplace application (at least 4 characters), which is of course not reasonable to be used in practice but which we consider good enough for this lab setting. To do the actual password change, method *changePassword* in class *UserService* is used.

Your task is to complete the functionality to change the password. Do the following to implement this functionality:

- Looking at the screenshot above, you should realize that this password change is designed in an insecure way (and with this, we don't mean the too simplistic requirements for the new password, and also not the fact the user doesn't have to insert the new password twice (which would be reasonable to do, but which is omitted here)). Identify the issue and start fixing it by adapting *template/admin/accountsettings.html* and the data class *ChangePassword*. When doing these extensions, make sure to consider Bean Validation for any data entered by the user.

[1]Details about the annotations: *https://jakarta.ee/learn/docs/jakartaee-tutorial/current/beanvalidation/bean-validation/bean-validation.html*

- Next, extend the code in method *changePassword* in class *UserService* so that it allows to change the password in a secure way. If the password change is successful, the method returns true, otherwise false. Take the following into account when extending the method:
  - The method already contains the code to get the correct *User* entity from the database and to save the modified entity in the database (so that the new bcrypt hash corresponding to the new password is saved).
  - To complete the implementation, use object *passwordEncoder* that is injected into *UserService*. This object is of type *BCryptPasswordEncoder*, see class *CommonSecurityConfig* (in *marketplace-common*), and provides methods that you can use to complete this task[2].
- Finally, think about whether you must adapt method *securityFilterChain* in class *SecurityConfig* to make sure that access control for the account settings is configured correctly. If yes, add the required configuration(s). Explain whether you have to add anything or not.

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

7 Task 2: Login Input Validation and Limiting Online Password Guessing Attacks

Your second task is to improve security of the login functionality. This includes input validation during the login process and limiting online password guessing attacks.

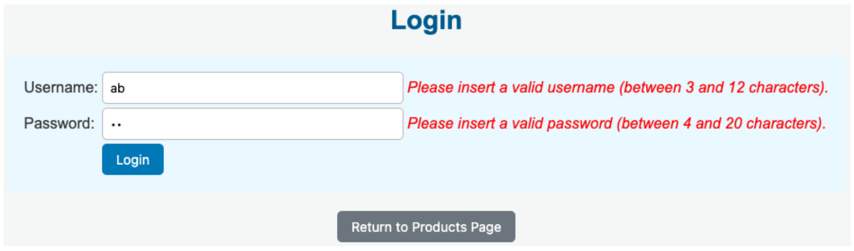
7.1 Login Input Validation

So far, there's no input validation done during login. Security-wise, one should always validate all data that is received to make sure that no malformed data is processed. Currently, login is implemented as follows (for full details, see the lecture slides):

- The login page is accessed with a GET request to */public/login* (configured in class *SecurityConfig* with *.loginPage("/public/login")*).
- On this login page, when entering username and password and clicking *Login*, a POST request to the same URL */public/login* is sent.
- Processing this POST request and checking the entered username and password is done automatically based on method *authenticationManager* in class *CommonSecrityConfig*.

This high degree of automation during the last step is great as it means that less code must be implemented, but it also means that it's not possible to integrate input validation. In the following, the program will be extended so that it supports input validation during login. As a result of this extension, login should behave as follows if the username or the password do not fulfill the criteria with respect to character length:

[2] Details can be found in the Spring Security API documentation: *https://docs.spring.io/spring-security/site/docs/current/api/index.html*

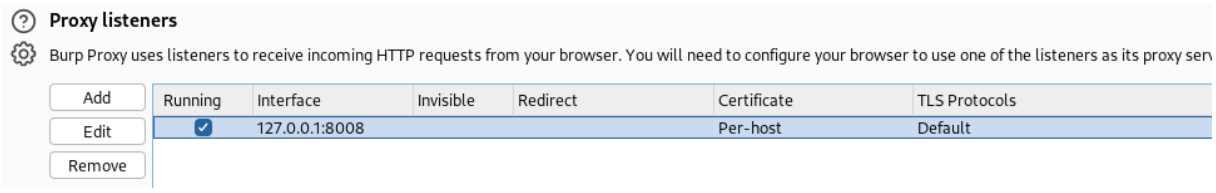

Login
Username: ab
Please insert a valid username (between 3 and 12 characters).
Password: ••
Please insert a valid password (between 4 and 20 characters).
Login
Return to Products Page

Do the following to implement this functionality:

- A data class is required to process the login manually. We can reuse class *Credentials*, which was introduced during the lecture as part of the REST API and which already contains the correct annotations. Briefly inspect the class so you know what it contains.
- Method *loginPage* in class *PublicController* must be extended so that an object of type *Credentials* is added as a model attribute. This is also already implemented, so just inspect this method so that you understand what is being done.
- Modify *template/public/login.html* so that the validation error messages are displayed in case input validation fails. In addition, change the URL of the *th:action* attribute in the login form from */public/login* to */public/login-check*. With this, processing the login request is no longer done automatically (which would be the case with a POST request to */public/login*), but we can handle it ourselves.
- To handle POST requests to */public/login-check*, a method *processLogin* was added to class *PublicController*. The method is already completely implemented, but make sure that you understand what it does. It first enforces input validation of the received username and password. If input validation fails, it returns the login page, which – assuming you implemented input validation correctly during the previous steps – includes the validation error messages. If no validation error happens, the request is forwarded to */public/login*, i.e., to the component that handles the login automatically. So the "only purpose" of using the controller method *processLogin* is to enforce input validation. The actual login is still handled automatically.

Verify that input validation indeed works correctly. However, the current implementation is not secure yet, and it's actually possible for an attacker to circumvent input validation during login. Do you have an idea why this is so? Write your reasoning into the following box.

Let's check if is indeed possible to circumvent input validation. Start an interceptor proxy. On the VM, you can use *Burp Suite* for this. Per default *Burp Suite* listens on port 8080, which is also used by the Marketplace web application. Therefore, after having started *Burp Suite*, select the tab *Proxy* and next *Proxy settings*. Then, select the configured row under *Proxy listeners* and use the *Edit* button to change the port from *8080* to *8008*, and mark the *Running* checkbox. The result should be as in the screenshot below.



Burp Proxy uses listeners to receive incoming HTTP requests from your browser. You will need to configure your browser to use one of the listeners as its proxy serv

Running	Interface	Invisible	Redirect	Certificate	TLS Protocols
✓	127.0.0.1:8008			Per-host	Default

Next, configure the browser (this description assumes that you are using Firefox) to use *Burp Suite* as a proxy. Open the *Settings* in Firefox, select *General*, then scroll down to *Network Settings* and click

Settings. In the opening *Connection Settings* window, select *Manual proxy configuration* and enter *localhost* and port *8008* under *HTTP proxy*, and mark the checkbox *Also use this proxy for HTTPS*. In addition, remove all entries from the *No Proxy for* box in case there are entries listed. As the browser and the proxy (*Burp Suite*) are running on the same system, you need to change another Firefox setting to make sure that requests to *localhost* are actually sent through *Burp Suite*. To do this, enter *about:config* in the address bar and then click on the button *Accept the Risk and Continue*. Then, enter *localhost* in the search field and double-click on the line *network.proxy.allow_hijacking_localhost* to set it to *true*.

Use the browser to navigate to the login page. Next, turn on the interceptor mode in *Burp Suite*, go back to the browser, enter exactly one character for username and password and click *Login*. In *Burp Suite*, modify the URL of the POST request from */public/login-check* to */public/login*. Then, forward the request and check (in the browser) the error message. You should get *Invalid username or password, please try again*. This shows that input validation was indeed circumvented, and that the login was attempted.

The problem is that we have not yet blocked direct access of POST requests to */public/login*. The obvious way to block such requests would be with an additional access control configuration in method *securityFilterChain* in class *SecurityConfig*. However, this does not work as with the configuration *.loginPage("/public/login")*, POST requests to this URL are always permitted.

Therefore, another way is required. The best option is to use a request filter. Such a filter is already provided in class *BlockLoginPostRequestFilter*. If you study it, it can easily be seen that this blocks POST requests to */public/login* and sends a response with HTTP status code 403 to indicate that access is forbidden. All you have to do now is to enforce usage of this filter in method *securityFilterChain* in class *SecurityConfig*. To do this, add the following *addFilterBefore* configuration add the end of method *securityFilterChain*, as illustrated below (*blockLoginPostRequestFilter* is an object of class *BlockLoginPostRequestFilter* and is already injected in class *SecurityConfig*):

```
  .requiresChannel(channel -> channel.anyRequest().requiresSecure())
  .addFilterBefore(blockLoginPostRequestFilter, UsernamePasswordAuthenticationFilter.class);
return http.build();
```

Note that this inserts this filter before the *UsernamePasswordAuthenticationFilter*, which is the filter used by Spring Security to process login requests. This guarantees that POST requests to */public/login* are indeed filtered before the request is processed.

Try to circumvent input validation during login again, in the same way as above. Now you should get a response with HTTP status code 403.

This is a good example which shows that implementing security features can be tricky, and it's not always easy and sometimes cumbersome to truly end up with a secure implementation. It is also a good example to demonstrate the importance of understanding the underlying framework (here: Spring), so one can find ways to eventually achieve the security goals.

Burp Suite won't be used any more in this lab, so it's best to turn off proxy usage in Firefox. To do this, set the *Connection Settings* back to *Use system proxy settings*. In case Firefox refuses to connect to the Marketplace application after this, select menu *History* in Firefox => *Clear Recent History* => *Clear*.

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

7.2 Limiting Online Password Guessing Attacks

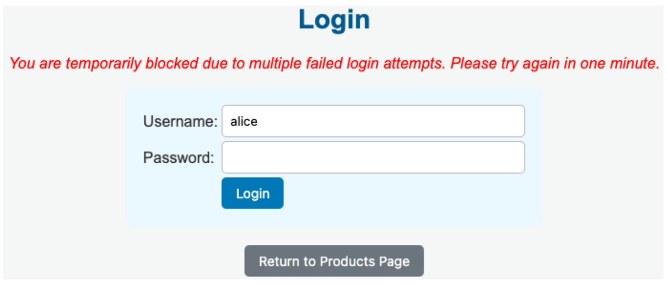
Right now, an attacker can perform as many login attempts as he likes, which allows online password guessing attack. Note that the usage of bcrypt throttles the attacker already to a certain degree, but it is desirable to have a more effective throttling mechanism.

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 making sure that a fresh session is established 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 many different IP addresses. This would typically be done using a botnet.
- After a certain number of failed logins per username, block login with that username 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 username and per time interval.

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

- Whenever a user tries to log in with a wrong password and the used username is currently not blocked, he gets the message *Invalid username or password, please try again.* (this corresponds to the current behavior of the application).
- After three failed login attempts with a particular username, login with that username 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 *You are temporarily blocked due to multiple failed login attempts. Please try again in one minute.* is displayed.


Login

You are temporarily blocked due to multiple failed login attempts. Please try again in one minute.

Username: alice

Password:

- After 60 seconds, a user can login again, but entering the wrong password (which displays the message *Invalid username or password, please try again.*) results in blocking the user once more 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.
- With this mechanism, an attacker is significantly slowed down because even when using many computers or IP addresses in parallel, he can at most do 1’440 password attempts for a given username per day.

To implement the mechanism, you should do the following:

- To keep track of failed logins and blocked users, a service class *LoginThrottlingService* is used, of which a skeleton already exists. 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 that user.
  - *public boolean isBlocked (String username)*: This method returns whether user *username* is currently blocked.
  - Note that class *LoginThrottlingService* is annotated with *@Service*. This means that one instance of this class is created when the application is started and used during the entire ap-

plication lifetime. Consequently, if an object of this class is injected in other classes, it is guaranteed that always the same (and only) instance of class *LoginThrottlingService* is used.

- 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 the functionality internally. 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 minimal resources. In particular, don't use the database to keep track of failed logins but implement everything «in memory» in the class *LoginThrottlingService*.
  - 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 in this case and always display the message *Invalid username or password, please try again.*). Otherwise, the attacker could exhaust your tracking mechanism by submitting large amounts of fictitious usernames.
  - To check the correctness of your approach, you can print the internal State of *LoginThrottlingService* whenever a user logs in (successfully or not) by using *System.out.print()*. The output should be visible in the *Console* window in *IntelliJ*.
- Once class *LoginThrottlingService* is implemented, it must be integrated into the login process so that the instance of *LoginThrottlingService* is correctly informed whenever login has failed or succeeded. Per default, Spring automatically handles failed and successful logins by redirecting the user to the admin area (in case of login success) or to the login page (in case of login failure). However, Spring also allows to implement custom behavior and provides two interfaces for this case: *AuthenticationFailureHandler* and *AuthenticationSuccessHandler*. Two classes that implement these interfaces are already existing: *CustomAuthFailureHandler* and *CustomAuthSuccessHandler*. You must complete these two classes:
  - In class *CustomAuthFailureHandler*, complete method *onAuthenticationFailure*. This method is called whenever login has failed. Use the public methods of class *LoginThrottlingService* to get the intended login throttling behavior. Note that objects of the classes *LoginThrottlingService* and *UserService* are already injected, so you can use them. At the end, the user is redirected to the login page, so we have the same behavior as before (i.e., the same behavior as without using *CustomAuthFailureHandler*).
  - In class *CustomAuthSuccessHandler*, complete method *onAuthenticationSuccess*. This method is called whenever login has succeeded. Use the public methods of class *LoginThrottlingService* to get the intended login throttling behavior. Here, an object of the class *LoginThrottlingService* is already injected. At the end, the user is redirected to the admin area, so we have the same behavior as before (i.e., the same behavior as without using *CustomAuthSuccessHandler*).
  - In addition, both classes must be registered with the login process. This is done by adding the following two configurations to method *securityFilterChain* in class *SecurityConfig* and by removing the current line *.failureUrl("/public/login?error=true")*:

    ```
    .loginPage("/public/login")
    ~~.failureUrl("/public/login?error=true")~~
    .failureHandler(authFailureHandler)
    .successHandler(authSuccessHandler)
    .permitAll())
    ```

    authFailureHandler and *authSuccessHandler* are objects of classes *CustomAuthFailureHandler* and *CustomAuthSuccessHandler*, and are already injected into class SecurityConfig.

- What remains to be done is actually blocking login attempts in case the user is blocked. This can be implemented with a request filter. The corresponding class, *LoginThrottlingFilter* is already partly implemented.
  - First, complete the class *LoginThrottlingFilter*. If the username that is provided during login is currently blocked, redirect the user to URL */public/login?blocked=true* and return from method *doFilterInternal*.
  - Extend *template/public/login.html* so that it displays the message *You are temporarily blocked due to multiple failed login attempts. Please try again in one minute.* in case the request to get the login page included a URL parameter with name *blocked* (see step above). This can be done in the same way as is already done in the template with URL parameter *error*.
  - Enforce usage of this filter in method *securityFilterChain* in class *SecurityConfig*. To do this, add the following *addFilterBefore* configuration as follows (an object *loginThrottlingFilter* of class *LoginThrottlingFilter* is already injected in class *SecurityConfig*):

    ```
      .requiresChannel(channel -> channel.anyRequest().requiresSecure())
      .addFilterBefore(blockLoginPostRequestFilter, UsernamePasswordAuthenticationFilter.class)
      .addFilterBefore(loginThrottlingFilter, UsernamePasswordAuthenticationFilter.class);
    return http.build();
    ```

    This inserts this filter after the *BlockLoginPostRequestFilter* and before the *UsernamePasswordAuthenticationFilter*, which guarantees that POST requests to do a login won't be processed if the user is blocked.

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

8 Task 3: Encrypting Credit Card Numbers

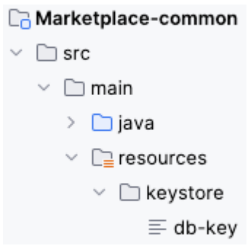
Currently, the credit card numbers are stored in plaintext in the database. This means an attacker can possibly get access to them via SQL injection, by breaking into the database server, by getting physical access to the database hard disk etc. Likewise, the legitimate (and possibly malicious) database administrator could simply read the credit card numbers directly from the database. To reduce such risks, the application should store credit card numbers only in encrypted form. Within the application, the credit card numbers should of course still be displayed in plaintext.

The mechanism you have to implement is specified as follows:

- AES/GCM/NoPadding with a 128-bit key is used to encrypt the credit card numbers. As GCM (Galois/Counter Mode) is used, a credit card number is not only encrypted, but also integrity-protected (for the sake of simplicity, we will simply say «encrypted» credit card numbers here, but this actually means «encrypted and integrity-protected»). The initialization vector (IV) is stored together with the encrypted credit card number (by concatenating IV and ciphertext). Use a fresh IV for each credit card number. The length of the Auth Tag (used by GCM for integrity-protection) is 128 bits.

- 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 before 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 the web application to run with its own user id and make sure only this user (and therefore the running web application) can access the file. The key file (*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, the service class *AESCipherService* is used. 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 there exist skeletons.
- The Jakarta Persistence API (JPA) provides *AttributeConverters*, which can be used to apply operations on data just before they are written to the database and just after they have been read from the database. We can use this to encrypt the credit card number just before writing it to the database and decrypting it just after reading it from the database. For this, the *AttributeConverter* interface must be implemented. This has already been done with class *AESConverter* 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 attribute *creditCardNumber* in entity class *Purchase* to use *AESConverter.class*. This guarantees that the credit card number will be encrypted before it is stored in the database and that it will be decrypted right after reading it from the database. Do this with the annotation *@Convert* as follows:

  ```
  @NotNull(message = "Credit card number is missing.")
  @CreditCardCheck
  @Convert(converter = AESConverter.class)
  private String creditCardNumber;
  ```

- To encrypt the credit card numbers of the already existing purchases in table *Purchase*, use the *mysql* command line tool to execute the script *Marketplace_UpdateEncryptedCreditCards.sql* by entering *SOURCE Marketplace_UpdateEncryptedCreditCards.sql*, in the same way as you did with *Marketplace.sql* at the beginning of this lab. 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.

9 Task 4: REST API

The final task is to extend the Marketplace REST API so that it provides the functionality for product managers. In addition, you are going to make sure that login throttling is also done with the REST API.

9.1 REST API for Product Managers

To do this, you have to complete the REST API controller class *AdminController*. The class contains three new skeleton methods *getProducts*, *postProduct* and *deleteProduct*, which are used to process GET, POST and DELETE requests. The purpose of the three request types is as follows:

- The GET request is used to get all products. For this, a new data transfer object (DTO) class *AdminProduct* is used. This class is already available and it sends JSON objects in the following form to the client:

  [{"productID":1,"description":"DVD Life of Brian - used, some scratches but still works","price":5.95,"username":"daisy"},{"productID":2,"description": "Ferrari F50 - red, 43000 km, no accidents","price":250000.00,"username":"luke"}]

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

- The POST request is used to create a new product. For this, class *Product* is used as DTO (which has already been used as DTO in the lecture to get the results when searching for products). The new product is included as JSON data in the request, using the following form:

  {"description":"Super Vulnerability Scanner","price":9999.95}

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

- The DELETE request is used to delete a product. A product manager can only delete the own products, which must be checked by the REST API.

  The URL to be used by the client is *rest/admin/products/{id}*, where *id* identifies the primary key (*ProductID*) of the product that should be deleted.

To test your REST API, project *Marketplace_Lab-REST-Test* is used. It's an extension of the test class used in the lecture, and it tests the already existing REST API functionality and also the extensions you will do in this section. The tests implement various cases using non-authenticated users, authenticated users, valid input data, invalid input data, and so on. For each test, the result is *PASSED* or *FAILED*, where *PASSED* means that the test produced the expected response from the REST API. In Section 2, it is described how to run the tests. You can run them anytime you want, e.g., whenever you have modified one of the methods to handle REST requests to check whether it works as intended. Make sure to always reset the database before running the tests (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 this functionality was already implemented during the lecture and is therefore included in the version you are using. Note also that you may have to inject additional objects of already existing service classes into class *AdminController* so you can use them when implementing the three methods.

Method *getProducts* (used to process *GET rest/admin/products*):

- Getting the products can be done in a similar way as getting the purchases in method *getPurchases*. However, the list of products retrieved from the database must be converted to a list of *AdminProduct* objects before they can be sent to the client. To facilitate this, class *AdminProduct* provides a constructor to create an *AdminProduct* object from a *Product* object.
- Extend method *securityFilterChain* in class *SecurityConfig* to make sure that only users with role *PRODUCTMANAGER* can use this functionality.
- If a user with a different role than *PRODUCTMANAGER* tries to access the resource, he should get a 403 response with JSON content *{"error":"Access denied."}*. Assuming you have extended class *SecurityConfig* correctly (see previous step), this should work out of the box as 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 should get a 401 response with JSON content *{"error":"Authentication required."}*. Just like above, this is already handled in the version you are using and should work out of the box.

Method *postProduct* (used to process *POST rest/admin/products*):

- Make sure that the received *Product* object is validated before processing it, and make sure to set the username of the current user in the received *Product* before inserting it into the database.
- If saving the product in the database fails, throw a *RuntimeException* (similar as in method *postPurchase* in class *PublicController*). Processing this exception type and generating the correct response was already done in the lecture and is therefore included in the version you are using (in class *GlobalExceptionHandler*).
- If a validation error occurs, a 400 response with JSON data containing the validation error message(s) should be sent to the client, e.g., *{"error":"Please insert a valid price (between 0 and 999'999.99, with at most two decimal places)."}*. Assuming you implemented validation correctly (see first step above), this should work without further adaptations as handling the validation exceptions was also already implemented in the lecture, in class *GlobalExceptionHandler*.
- If needed, extend method *securityFilterChain* in class *SecurityConfig* to make sure that only users with role *PRODUCTMANAGER* can use this functionality.
- 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 requests above and should work out of the box.

Method *deleteProduct* (used to process *DELETE rest/admin/products/{id}*):

- This should be done in a similar way as method *deletePurchase* in class *AdminController*.
- Use validation annotations with method parameter *id* to make sure the *id* is between 1 and 999’999. The message should be *The product ID must be between 1 and 999'999.*
- Make sure that product managers can only delete their own products.
- If an *id* is used that is not in the permitted range, a 400 response with JSON content *{"error":"The product ID must be between 1 and 999'999."}* should be sent to the client. Assuming you used the validation annotations correctly (see second step above), this should work without further adaptations as handling the validation exceptions was already implemented in the lecture, in class *GlobalExceptionHandler*.
- If an *id* is used for which no product exists, a 400 response with JSON content *{"error":"The product with product ID = '999' does not exist"}* should be sent to the client. You can do this by throwing an *InvalidParameterException* with the desired error message. The processing of the exception and the generation of the correct response is also already handled correctly, in class *GlobalExceptionHandler*.
- If a product manager 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 *AccessDeniedException* (use *throw new AccessDeniedException("");* for this) if a product manager tries to delete a product that is not his own product. In addition, you must handle this exception type in class *GlobalExceptionHandler* so that the desired response is created.
- Extend method *securityFilterChain* in class *SecurityConfig* to make sure that only users with role *PRODUCTMANAGER* can use this functionality.
- 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 requests above and should work out of the box.

Before you continue, check the following:

- When running the tests in project *Marketplace_Lab-REST-Test*, all tests should show *PASSED* (except the login throttling tests, as login throttling has not been implemented yet for the REST API).

9.2 Limiting Online Password Guessing Attacks

Finally, make sure that the REST API cannot be abused for online password guessing. As you have implemented class *LoginThrottlingService* before (see task 2), not much remains to be done:

- You must extend method *postAuthenticate* in class *AuthControler*. Object *loginThrottlingController* of type *LoginThrottlingController* is already injected and therefore available.
- The rules are the same as in the web application: A username is blocked for 60 seconds after three wrong passwords and after these 60 seconds, the user gets one attempt to log in successfully before the username is blocked again. Just like in the web application, ignore non-existing usernames in your blocking mechanism.
- Currently, if the user uses a wrong username or password, a 400 response with JSON content *{"error":"Invalid username or password, please try again."}* is sent to the client. This is done by throwing an *InvalidParameterException*, which is handled in *GlobalExceptionHandler*. This behavior should remain the same, but only if the provided username is not blocked.
- If the provided username is blocked, a 400 response with JSON content *{"error":"You are temporarily blocked due to multiple failed login attempts. Please try again in one minute."}* should be sent to the client. You can do this by throwing an *InvalidParameterException* with the desired error message.

Before you continue, check the following:

- When running the tests in project *Marketplace_Lab-REST-Test*, all tests should show *PASSED*.

Lab Points

In this lab, you can get **6 Lab Points**. To get them, you must demonstrate that your extended 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 6).
- You get 1 point for solving *Task 2: Login Input Validation and Limiting Online Password Guessing Attacks* (see Section 7).
- You get 1 point for solving *Task 3: Encrypting Credit Card Numbers* (see Section 8).
- You get 2 points for solving *Task 4: REST API* (see Section 9).

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

Test	**Passed**
Task 1: Extending Admin Area	
If a non-authenticated user clicks the *Admin Area* button, the user is redirected to the login page.	
Logging in with *alice/rabbit* shows the purchases.	
Logging in with *bob/patrick* shows an «empty» *Admin Area*.	
Logging in with *daisy/yellow* shows the products and the *Add product* button.	
The *Delete Product* button is only visible in the rows that correspond to *daisy*'s products.	
Deleting a product as *daisy* that is owned by *luke* results in a 403 response. You can do this	

test with button *Do Attack 1* on the attack page.	
Deleting a purchase as *daisy* results in a 403 response. You can do this test with button *Do Attack 2* on the attack page.	
Deleting a product as *daisy* deletes the product.	
When *daisy* enters a new product, the validation rules work correctly, and appropriate error messages are shown.	
Accessing */admin/addproduct* over HTTP (port 8080) and as *daisy* results in a redirection to HTTPS. You can do this test with button *Do Attack 3* on the attack page.	
Accessing */admin/addproduct* as *alice* results in a 403 response. You can do this with button *Do Attack 4* on the attack page.	
Changing the password of a user works and is designed in a secure way.	
If a non-authenticated user accesses */admin/accountsettings*, the user is redirected to the login page. You can do this test with button *Do Attack 5* on the attack page.	
Task 2: Login Input Validation and Limiting Online Password Guessing Attacks	
Input validation during login works as specified.	
POST requests to */public/login* are not processed and result in a 403 response. You can do this test with button *Do Attack 6* on the attack page, as this performs the same test that you did before by using *Burp Suite*: It sends a POST request to */public/login* using only one character for username and password.	
Log in with the same user four times in a row using a wrong password. Three times, you should get the message *Invalid username or password, please try again.* After the fourth login attempt, you should get the message *User xyz is temporarily blocked, try again in one minute*.	
Log in right again with the same user, but using the correct password. The user should still be blocked and get the message *You are temporarily blocked due to multiple failed login attempts. Please try again in one minute.*	
Wait more than 60 seconds and log in again with the same user, with a wrong password. You should get the message *Invalid username or password, please try again.*	
Log in right again with the same user, but using the correct password. You should get the message *You are temporarily blocked due to multiple failed login attempts. Please try again in one minute.*	
Wait more than 60 seconds and log in again with the same user, with the correct password. The login should be successful	
Log in four times (or more) with a non-existing username. You should get the message *Invalid username or password, please try again.* every time, but never the message that the user is blocked.	
Task 3: Encrypting Credit Card Numbers	

Make two purchases using the credit card number 1111 2222 3333 4444 both times. Then, use the *mysql* command line tool and enter *SELECT * FROM Purchase;* to check that credit card numbers 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):
`fGd10a46VT4C99RTtDJZqG0XRa8/oPElDBUcMh3+Qs4qFHS59+DU1CgLS1dqGZZ61+Yp`
`p84vqX+tI4C9wlvW3a2VqCw0O3EB/+6FjnriOsqiRPbxw7fJ3orQ04au3IFsQXqJFBcB`
`4NeQvkHbNbjUx6AS8bB7ZAHWZLphrY1JzkyPredw4uKd/JbcFqTLLPlBFGjpA4JsLv76`	

Accessing the *Admin Area* as user *alice* (or *robin*) shows the credit card numbers in plaintext.	
Task 4: REST API	
When running the tests in project *Marketplace_Lab-REST-Test*, all tests show *PASSED*.	

Handing in your solution can be done in two different ways:

- Demonstrate that your extended Marketplace application passes all the tests to the instructor in classroom, during the official lab lessons.
- Make a video that shows that your extended Marketplace applications passes all the tests. Send the video by e-mail to the instructor. Use *SecLab - Marketplace - group X - name1 name2* as the e-mail subject, corresponding to your group number and the names of the group members. If the video is too big for an e-mail, place it somewhere (e.g., SWITCHdrive) and include a link to it in the e-mail.