Developing Secure Modern Web Applications

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

Content and Goals

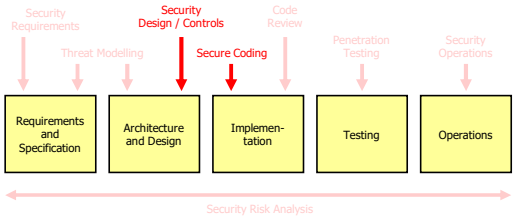
Content

- Introduction to modern web applications
- Development of the Marketplace REST API – as an example how to develop the server-side of a modern web application in a secure way
- Discussion of the security mechanisms that must be considered at the client-side, using a Marketplace SPA based on Angular as an example

Goals

- You understand the security mechanisms that must be considered in REST APIs and you can develop secure REST APIs using the Jakarta EE technology
- You understand what security mechanisms must be taken into account at the client-side of a modern web application

- Security activities covered in this chapter:



Introduction

Developing Secure Modern Web Applications (1)

- The previous chapter was about developing secure traditional web applications
  - «Traditional» web applications: Monolithic web applications where most of the code runs server-side and where the server mainly serves HTML documents to the client
- This chapter is about developing secure modern web applications
  - The term «modern» web application is of course not clearly defined, but here we mean single page web applications (SPA) that use REST based microservices and lots of JavaScript in the browser
- Security-wise, traditional and modern web applications have a lot in common
  - Most security mechanisms that should be used in traditional web applications must be considered as well
  - But there are also several differences that you should understand, e.g., authentication (tokens instead of sessions), cross-origin requests, client-side security, etc.

Developing Secure Modern Web Applications (2)

- Just like in the previous chapter, we are using the Marketplace application as the example scenario
  - Just like with traditional web applications, there are many technologies that can be used to develop SPAs, client- and server-side
- For the RESTful web service API, we are using Jakarta EE
  - Jakarta EE is not only well suited to develop traditional web applications, but also to develop REST based APIs that are used in the context of modern single page applications
  - This allows us to re-use a significant portion of the code that we have already developed
- For the client-side, we are using Angular
  - Angular is a TypeScript-based open-source web application framework
  - We are not doing an introduction into Angular and won't look at a lot of code; instead we will focus on discussing client-side security aspects in general and will use Angular mainly for demonstration purposes

Example Technologies

Just like in the previous chapter, Jakarta EE and Angular is «just the example technologies» we are using here, and the primary goal of the chapter is to learn in general what must be considered when designing secure modern (SPA) web applications. This means that once you have learned to secure a Jakarta EE / Angular based modern web application, you should be able to transfer what you've learned to other technologies and programming languages

Marketplace REST API

Marketplace V11

The extensions of this section are integrated in Marketplace_v11.

Marketplace RESTful Web Service API

- First, the Marketplace RESTful Web Service API will be developed
  - This API will then be used by the client-side part of the Marketplace SPA
  - Likewise, the API could also be used by a mobile Marketplace app
- Basic properties of RESTful web services
  - They use HTTP and the 5 methods GET, POST, PUT/PATCH and DELETE to read, create, update and delete resources within the service
  - Stateless, there are no sessions, every request must include all information such that the server can process the request
  - Resources are identified via meaningful URLs, e.g., */customers* to identify all customers and */customers/1234* to identify customer with ID 1234
  - To transport data, JSON format is typically used
- The REST API should provide the same functionality and the same security level as the traditional web application
  - E.g., access control, preventing SQL injection, input validation,...

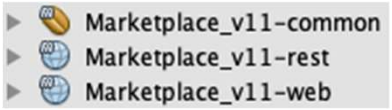
REST HTTP Methods

See, e.g., *https://restfulapi.net/http-methods/*

Both PUT and PATCH are used for updates. PUT is used to replace a resource (the data associated with the resource), while PATCH is used to partly update an existing resource. Also, if the resource that should be updated with PUT is not existing, then – depending on how PUT is implemented - PUT sometimes creates the resource, similar to POST.

Project Reorganization

- To Marketplace REST API is implemented as a separate application, but it re-uses the database and many classes already implemented in the traditional Marketplace web application
- To this, the project is reorganized:
  - *Marketplace_v11-common* includes everything that is used by both the traditional web application and the REST API (model classes, facade classes, service classes, utility classes, validation classes)
  - *Marketplace_v11-web* includes everything that is specific for the traditonal web application (Facelets, backing beans, web application-specific security classes and *web.xml*)
  - *Marketplace_v11-rest* includes everything that is REST API specific (REST classes, REST API-specific security classes and *web.xml*)



- Both *Marketplace_v11-web* and *-rest* include *Marketplace_v11-common* as a dependency, which includes the common components during compile time

Running the REST API

- To run the REST API, the same Payara application server that was already used for the traditional web application is used
  - This is actually a bit «overkill» as Payara is a full-fledged Jakarta EE application server that supports a lot of stuff we don't need in the Marketplace REST API
  - We are nevertheless going to this for simplicity and as we are not going to split the REST API into different microservices that run independently

- But if required, Jakarta EE and Payara can easily be used for a true microservice architecture
  - Use multiple Jakarta EE projects where each provides one microservice
  - Run each microservice individually on the same or on different hosts and run it as many times as needed (to scale to your requirements)
  - To run each microservice, use Payara Micro, which is a heavily reduced lightweight version of Payara server optimized for running microservices

Payara Micro

https://www.payara.fish/products/payara-micro/

REST API in Marketplace (1)

- The following two examples demonstrate how the Marketplace REST API works in general
- Example 1: Read all products: `GET /rest/products`
  - This results in an HTTP response with status code 200 (OK)
  - JSON response by Marketplace:

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

  - To get products that have a specific description, the parameter *filter* is used: `GET /rest/products?filter=DVD`

URLs

In general, we are using the prefix *rest/* for all web service URLs in the Marketplace application.

REST API in Marketplace (2)

- Example 2: Complete a purchase: `POST /rest/purchases`
  - The request includes JSON data to specify the purchase

```
{
    "firstname":"John",
    "lastname":"Doe",
    "creditCardNumber":"1111 2222 3333 4444",
    "productCodes":["0001", "0002", "0004"]
}
```

  - If everything works correctly (e.g., no validation errors, at least one of the products exists), this results in an HTTP response with status code 204
    - Status code 204 means «OK, but no content sent back»

- In general, if an error happens, the server responds with status code 4xx (often 400 = Bad Request) and includes JSON error data, e.g.

```
{
    "error":"Please insert a valid credit card number (16 digits)"
}
```

Marketplace REST – Application Config

- To implement RESTful web services, Jakarta EE provides the Jakarta API for RESTful Web Services (JAX-RS) and every JAX-RS application requires a class that extends *jakarta.ws.rs.core.Application*
  - Defines the core classes of a JAX-RS application
  - It's boilerplate code and often (partly) generated by the IDE

@ApplicationPath defines the base URL for all REST URLs:
- In this case: *Marketplace_v11-rest/rest/...*

```
@ApplicationPath("rest")
public class ApplicationConfig extends Application {

    @Override public Set<Class<?>> getClasses() {
        Set<Class<?>> resources = new HashSet<>();
        addRestResourceClasses(resources);
        return resources;
    }

    private void addRestResourceClasses(Set<Class<?>> resources) {
       resources.add(ch.zhaw.securitylab.marketplace.rest.core.AdminPurchaseRest.class);
       resources.add(ch.zhaw.securitylab.marketplace.rest.core.AuthenticationRest.class);
       resources.add(ch.zhaw.securitylab.marketplace.rest.core.ProductRest.class);
       resources.add(ch.zhaw.securitylab.marketplace.rest.core.PurchaseRest.class);
       resources.add(ch.zhaw.securitylab.marketplace.rest.core.
                                       ConstraintViolationExceptionMapper.class);
       resources.add(ch.zhaw.securitylab.marketplace.rest.core.
                                       InvalidParameterExceptionMapper.class);
       resources.add(ch.zhaw.securitylab.marketplace.rest.core.CORSFilter.class);
    }
}
```

Here, the core REST classes must be added

Marketplace REST – *web.xml*

- Just like Jakarta EE web applications, JAX-RS applications also require a file *web.xml* that contains some configurations

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app ...>
    <data-source>
        <name>java:global/marketplace</name>
        <class-name>com.mysql.cj.jdbc.MysqlDataSource</class-name>
        ...
    </data-source>
    <security-role>
        <description>Sales Personnel</description>
        <role-name>sales</role-name>
    </security-role>
    <security-role>
        <description>Marketing Personnel</description>
        <role-name>marketing</role-name>
    </security-role>
</web-app>
```

Configuration details to connect to the database (same as in the traditional web application)

Define the roles *sales* and *marketing*

- In addition, make sure to disable HTTP in the application server so that the REST API can only be reached over HTTPS
  - Using a *<security-constraint>* that redirects from HTTP to HTTPS does not make sense as the REST calls are not issued by users, but by the client-side SPA

<data-source>

It's exactly the same configuration that was used in *web.xml* of the traditional web application. The full configuration is given below:

```
<data-source>
    <name>java:global/marketplace</name>
    <class-name>com.mysql.cj.jdbc.MysqlDataSource</class-name>
    <server-name>localhost</server-name>
    <port-number>3306</port-number>
    <database-name>marketplace</database-name>
    <user>marketplace</user>
    <password>marketplace</password>
    <property>
        <name>sslMode</name>
        <value>Disabled</value>
    </property>
    <property>
        <name>allowPublicKeyRetrieval</name>
        <value>true</value>
    </property>
    <property>
      <name>serverTimeZone</name>
      <value>UTC</value>
    </property>
</data-source>
```

Disable HTTP

With the REST API, completely disabling HTTP in the application server is reasonable, as there's no real need (in contrast to a web application) to redirect HTTP requests to HTTPS. With the REST API, usage of HTTPS can be configured in the client application (which is the SPA in our case, but which could also be a mobile app) to make sure that the client always uses HTTPS.

Marketplace REST – *CORSFilter.java*

- This class makes sure the client-side part of the Marketplace SPA can call the Marketplace REST API
  - Basically, this includes the required response headers needed for Cross-Origin Resource Sharing (CORS) – a detailed explanation follows later...

```
import jakarta.ws.rs.container.ContainerRequestContext;
import jakarta.ws.rs.container.ContainerResponseContext;
import jakarta.ws.rs.container.ContainerResponseFilter;
import jakarta.ws.rs.ext.Provider;

@Provider
public class CORSFilter implements ContainerResponseFilter {

    @Override
    public void filter(ContainerRequestContext requestContext,
                       ContainerResponseContext response) {
        response.getHeaders().putSingle("Access-Control-Allow-Origin", "*");
        response.getHeaders().putSingle("Access-Control-Allow-Methods",
                                        "OPTIONS, GET, POST, PUT, DELETE");
        response.getHeaders().putSingle("Access-Control-Allow-Headers", "*");
    }
}
```

Marketplace REST – Data Transfer Objects (DTOs)

- A nice feature of JAX-RS is that Java objects can directly be serialized to JSON strings (and vice versa)
  - → The developer always works with objects instead of JSON strings
  - These object are often identified as Data Transfer Objects (DTOs)
  - DTOs are simple Java beans (attributes, getters/setters, standard constructor)
- If there's already a JPA entity available, one can usually reuse it
  - To omit attributes of the entity, they can be annotated with *@JsonbTransient*
- Example: Adaptations to *Product.java* so it can also be used as a DTO

```
public class Product implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id private int productID;
    private String code;
    private String description;
    private BigDecimal price;
    @JsonbTransient
    private String username;


    public String getCode() { return code; }
    public void setCode(String code) { this.code = code; }
    ...
}
```

Is serialized to / deserialized from the following JSON string (attributes that are static, have no getter (*productID* here), or are annotated with *@JsonbTransient* are ignored):

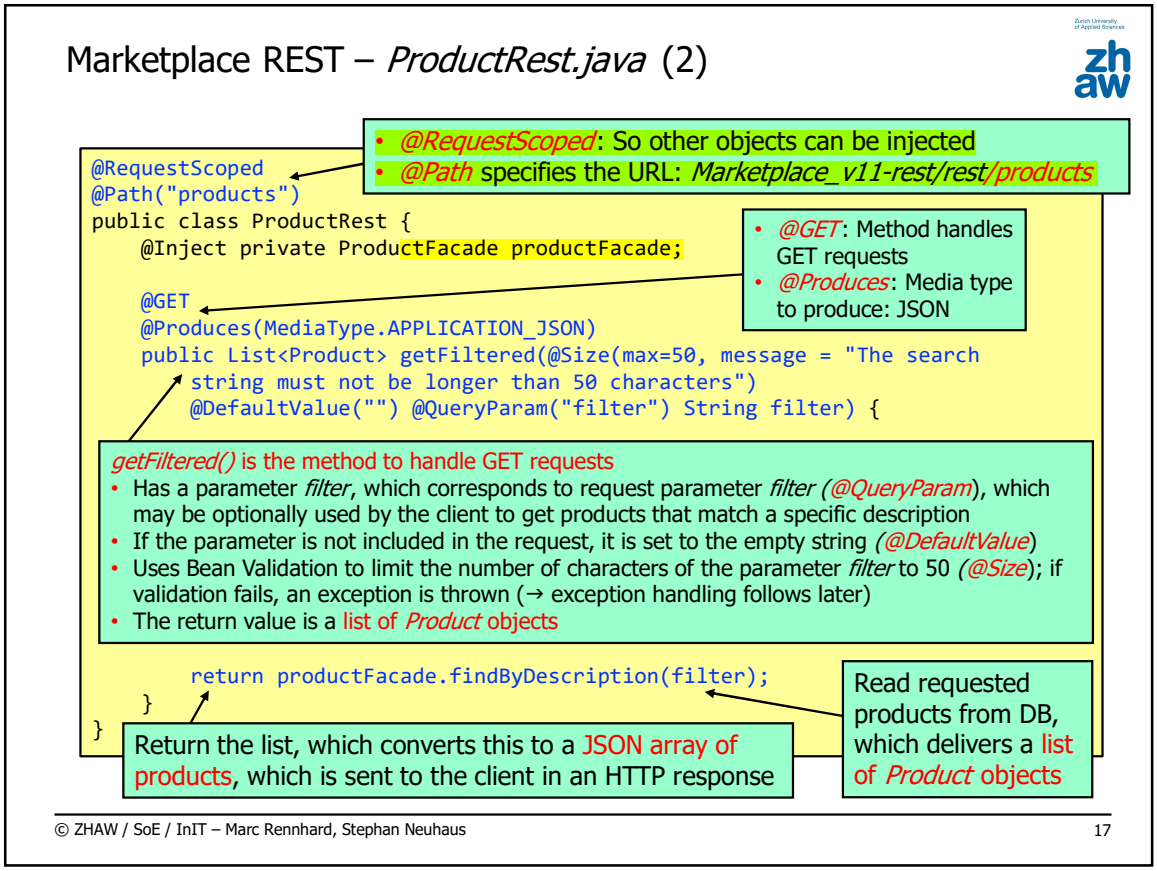
```
{
  "code":"0001",
  "description":"DVD Life of Brian ... ",
  "price":5.95
}
```


DTOs

A DTO is basically a simple Java bean and requires private instance variables, a standard constructor, and getter and setter methods. But they can also contain additional methods or constructors.

Marketplace REST – *ProductRest.java* (1)

- *ProductRest.java* provides the REST interface associated with products
- To implement a REST interface, one method must be implemented for each request type that is supported
  - Currently, the only request type that is supported is GET to read all products (optionally filtered)
  - Therefore, only one method has to be provided
- If a request should return data to the client, the corresponding method uses a DTO as return value (or a list of DTOs in the case of multiple DTOs)
  - Generating the HTTP response and serializing the returned DTO to JSON happens transparently for the developer
  - If no data should be included in the response to the client, return void



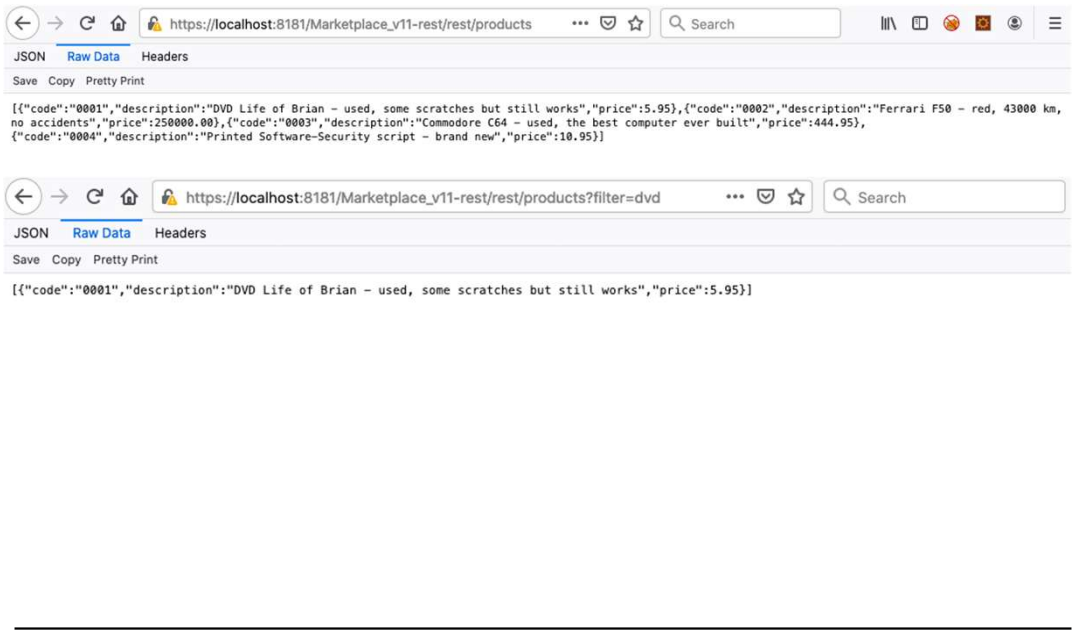
Import Statements

The class above requires the following import statements:

```
import java.util.List;
import jakarta.inject.Inject;
import jakarta.ws.rs.DefaultValue;
import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.Produces;
import jakarta.ws.rs.QueryParam;
import jakarta.ws.rs.core.MediaType;
import jakarta.enterprise.context.RequestScoped;
import jakarta.validation.constraints.Size;
import ch.zhaw.securitylab.marketplace.common.facade.ProductFacade;
import ch.zhaw.securitylab.marketplace.common.model.Product;
```

Marketplace REST – GET Products Test

- Testing this with the browser works as expected:



URL

https://localhost:8181/Marketplace_v11-rest/rest/products

Marketplace REST – *PurchaseRest.java* (1)

- *PurchasesRest.java* handles the checkout process
  - During checkout, a purchase is created → provide a method that handles POST requests

```
@RequestScoped
@Path("purchases")
public class PurchaseRest {
    @Inject private PurchaseFacade purchaseFacade;
    @Inject private ProductFacade productFacade;

    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    public void post(@Valid CheckoutDto checkoutDto) {
```

@Path specifies the URL: *Marketplace_v11-rest/rest/purchases*

- *@POST*: Method handles POST requests
- *@Consumes*: Media type expected in request: JSON

The method has a parameter of type *CheckoutDto*, which corresponds to the JSON data received from the client

- The object is automatically created by JAX-RS from the received JSON data
- *@Valid* validates the object using Bean Validation before it is processed; if validation fails, an exception is thrown (→ exception handling follows later)

Marketplace REST – *CheckoutDto.java*

- *CheckoutDto.java* includes the Bean Validation annotations for *firstName*, *lastName*, *creditCardNumber* and *productCodes* → these attributes are validated when using *@Valid*

```
public class CheckoutDto {
    @NotNull(message = "Element with key 'firstname' is missing")
    @Pattern(regexp = "^[a-zA-Z']{2,32}$", message = "Please insert a
             valid first name (between 2 and 32 characters)")
    private String firstname;
    @NotNull(message = "Element with key 'lastname' is missing")
    @Pattern(regexp = "^[a-zA-Z']{2,32}$", message = "Please insert a
             valid last name (between 2 and 32 characters)")
    private String lastname;
    @NotNull(message = "Element with key 'creditCardNumber' is missing")
    @CreditCardCheck
    private String creditCardNumber;
    @NotNull(message = "Element with key 'productCodes' is missing")
    private List<String> productCodes;

    public String getFirstname() { return firstname; }

    public void setFirstname(String firstname) {
        this.firstname = firstname;
    }

    // Other getter and setter methods omitted
}
```

We re-use the Bean Validation annotations from the «traditional» Marketplace web application and also use *@NotNull* annotations to make sure the received JSON data contains all elements that we are expecting

@NotNull

If the JSON data does not include, e.g., the *firstname* element, then the corresponding attribute won't be set in the *CheckoutDto* object, i.e., it has the value null. As a consequence, there would be an exception when trying to store the purchase in the database. This is no real security problem as database access is implemented in a secure way and in addition, possible information leakage can easily be prevented using *<error-page>* in *web.xml* (see later). Nevertheless, for robustness reasons and to prevent potentially negative side effects, it's always a good idea to block requests that do not contain all required values and to enforce this, we make sure that the *CheckoutDto* object is only processed is all required elements are included in the JSON data from the client by using the *@NotNull* annotation in front of all attributes.

Note that in contrast to «traditional» web applications such as the Marketplace JSF application (as discussed before) that provide both the client- and server-side of the application in an integrated way, REST APIs typically only provide the server-side, and the client-side (which may be, e.g., an Angular application or a mobile app) is often developed using another technology and often also by another team. Therefore, it can easily happen that the client-side does not follow the REST API specification which may result, e.g., in missing JSON elements in requests. Therefore, using @NotNull definitely makes sense with REST APIs to detect and prevent corresponding problems.

Note also that the *@Pattern* annotation always returns true if the attribute is null, so the *@NotNull* annotation is explicitly needed in addition to *@Pattern*.

Invalid JSON data that cannot be converted to a DTO

If the received JSON data is not valid, e.g., because it is malformed in general (e.g., due to a missing a double quote character or a missing bracket) or because it uses a wrong data type (e.g., 0001 instead of "0001" for a product code), it cannot be converted to a DTO (*CheckoutDTO* in the case above). This will throw an exception, the method to handle the request won't be called, and a response with status code 500 is sent back. This could be handled with an exception handler (see later). Here, we are omitting this for simplicity. However, the error-handling servlet that will be introduced later in this chapter creates a generic error message in this case (*{"error":"Unknown error"}*), which we consider to be «good enough» as it prevents that any information can be leaked.

Marketplace REST – *PurchaseRest.java* (2)

```
        BigDecimal totalPrice = new BigDecimal(0.0);
        Product product;
        int validProductCodes = 0;


        for (String productCode : checkoutDto.getProductCodes()) {
            product = productFacade.findByCode(productCode);
            if (product != null) {
                totalPrice = totalPrice.add(product.getPrice());
                validProductCodes++;
            }
        }
        if (validProductCodes > 0) {
            Purchase purchase = new Purchase();
            purchase.setFirstname(checkoutDto.getFirstname());
            purchase.setLastname(checkoutDto.getLastname());
            purchase.setCreditCardNumber(checkoutDto.getCreditCardNumber());
            purchase.setTotalPrice(totalPrice);
            purchaseFacade.create(purchase);
        } else {
            throw new InvalidParameterException("The purchase could not be
                completed because no valid product codes were submitted");
        }
    }
}
```

Get the products that correspond to the received product codes from the DB and sum up the total price

ProductFacade is extended with method *findByCode* (see notes below)

If at least one valid product code was received store the purchase in the DB

If no valid product code was received, throw an exception (→ exception handling follows later)

findByCode in *ProductFacade.java* und Named Query in *Product.java*

```
public Product findByCode(String code) {
    Query query = entityManager.createNamedQuery("Product.findByCode");
    query.setParameter("code", code);
    return getSingleResultOrNull(query);
}
@NamedQueries({
    @NamedQuery(name = "Product.findByDescription", query = "SELECT p
         FROM Product p WHERE p.description LIKE :description"),
    @NamedQuery(name = "Product.findByCode", query = "SELECT p FROM
         Product p WHERE p.code = :code")})
```

Marketplace REST – *AdminPurchaseRest.java* (1)

- *AdminPurchasesRest.java* handles requests by *marketing* and *sales* users to read and delete purchases
  - Provide methods to handle GET and DELETE requests

@Path specifies the URL: *Marketplace_v11-rest/rest/admin/purchases*
- The URL-part *admin* is used to clearly identify this as admin functionality

```
@RequestScoped
@Path("admin/purchases")
public class AdminPurchaseRest {
    @Inject private PurchaseFacade purchaseFacade;

    @GET
    @RolesAllowed({"marketing", "sales"})
    @Produces(MediaType.APPLICATION_JSON)
    public List<Purchase> get() {
        return purchaseFacade.findAll();
    }
```

@GET: Method to handle GET requests, returns all purchases

@RolesAllowed: Specifies what roles are allowed to access the method
- Here: Only authenticated users with role *marketing* or *sales* get access
- If that's not the case, the request won't be processed and a response with HTTP status code 401 or 403 is created (see later)

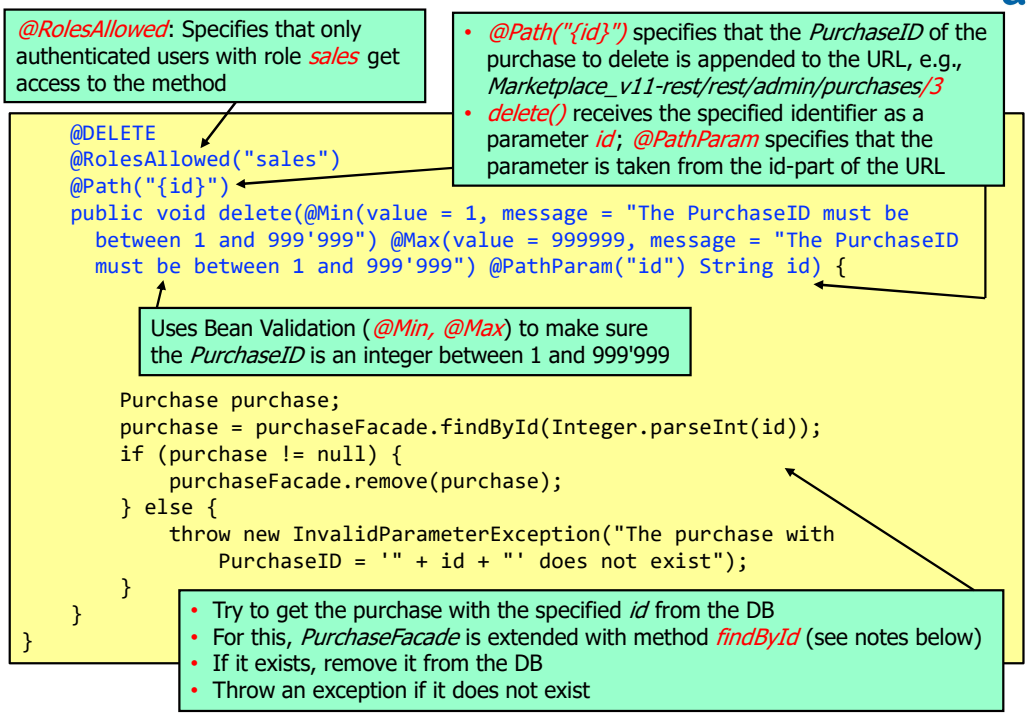
- Note: One can also use *<security-constraint>*s in *web.xml* to configure access restrictions, but using *@RolesAllowed* is the preferred approach

Purchase Entity as DTO

As can be seen from the code above, the *Purchase* entity already developed in the context of the traditional Marketplace web application can be reused here as a DTO as its attributes mostly correspond to the information that should be included in the JSON string. There's one exception, though: The following method was added to *Purchase* to make sure that the *purchaseID* is also included in the JSON data. This purchase ID is needed by the client to delete the corresponding purchase if needed (using the method on the next slide).

```
public int getPurchaseID() {
    return purchaseID;
}
```

Marketplace REST – *AdminPurchaseRest.java* (2)



findById in *PurchaseFacade.java* und Named Query in *Purchase.java*

```
public Purchase findById(int id) {
    Query query = entityManager.createNamedQuery("Purchase.findById");
    query.setParameter("purchaseID", id);
    return getSingleResultOrNull(query);
}
```

```
@NamedQuery(name = "Purchase.findById", query = "SELECT p FROM Purchase p WHERE p.purchaseID = :purchaseID")
```

Marketplace REST – Authentication (1)

- With this, the Marketplace REST API provides all required functionality and access control is configured as well...
  - ...but what's missing is a way to authenticate users and a way to grant authenticated users with roles *marketing* and *sales* access to protected resources
- A truly RESTful web service is stateless and should not use sessions, which means that every request to protected resources must include the complete information to authenticate the user
- With REST, this is usually done with the HTTP Authorization header
  - Includes «some sort of credentials», which are verified by the service to grant or deny access

Marketplace REST – Authentication (2)

- An obvious choice is to directly include the username and password of a user in every request to protected resources
  - However, this is a bad idea, mainly for performance reasons
  - Requires to compute an expensive PBKDF2 hash with every request → lots of computational overhead

+ Better option: The user authenticates himself once and gets an authentication token that can be used in the subsequent requests

- Such a token should have the following properties:
  - It cannot be forged or guessed by an attacker
  - It should be usable during a configurable time and expire after this time
  - Checking the token by the REST service should only require a small computational effort
  - Should ideally be stateless: The service doesn't have to store the tokens (neither in a database nor in memory)

JSON Web Token (1)

- JSON Web Token (JWT) is a standard for such authentication tokens
- A token consists of three parts (shown here as used in Marketplace)
  - Header (JSON), containing the used MAC algorithm (here: HMAC-SHA256)

```
{
  "alg":"HS256"
}
```

  - Payload (JSON), containing the issuer, the subject, and the expiry date (UNIX timestamp)

```
{
  "iss":"Marketplace",
  "sub":"alice",
  "exp":"1519643485"
}
```

  - MAC (binary) over the header and the payload, based on a key known to the REST service

```
HMAC-SHA256(header +
"." + payload, key)
```

- The resulting token is built as follows:
  - Base64(header).Base64(payload).Base64(MAC), e.g.
  - eyJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJtYXJrZXRwbGFjZSIsInN1YiI6ImFsaWNlIiwiZXhwIjoxNTE5NjQzNDg1fQ.mhS3WeZPNyjGyxTzfq-xos_jCW9IS3pSgzfb774AyeY

JSON Web Token

For details, see *https://jwt.io* or RFC 7519.

Payload of JWT is Extensible

Note that the payload can also contain different information, which means it is extensible. For instance, one could insert the roles if a user.

Cryptographic Protection

Besides HMAC, RSA is also supported. In this case, the token is signed. This is especially appealing in situations where multiple systems verify a token, as these systems then only require the public key (and not the private key, which is only required by the component / system which issues the tokens).

JSON Web Token (2)

- A JWT fulfills all our requirements:
  - It cannot be forged / guessed assuming the HMAC key remains secret
  - It automatically expires once the expiry date is reached
  - Checking the token only requires verifying a HMAC, which is a small computing overhead
  - It's stateless as all information to verify the token is included in the token (self-contained), so the server doesn't have to store issued tokens
  - Note: Another advantage is that the token is compact and URL-safe, so it can easily be included in GET / POST parameters and HTTP headers
- Usage of JWT in Marketplace:
  - To get a token, the user must first authenticate at the Marketplace REST service using username and password
  - If correct, the service creates a token that includes the username of the user and sends it back to the client
  - To access a protected resource, the client includes the token in the request
  - The REST service verifies the token and – if the token is valid – extracts the username from it → knows the identity of the user who sent the request

Getting a JSON Web Token

We are using here the simplest approach, where we get the JWT directly from the service itself. Beyond this, other approaches could be used. For instance, a separate authentication server could be used where a client can observe a valid JWT.

Another option is using the OAuth 2.0 Authorization Framework to perform the authentication and obtain a token. OAuth2 not only support direct authentication of the user (or client), but also allows a third-party application to obtain an authentication token to access a service in the name of the user. This is for instance done in the context of social logins, based on, e.g., Facebook or Google accounts. In this case, a user can use the third-party login to get access to an application, so there's no need to have dedicated credentials for the application. To do this, the user provides the application (during registration and login) with a token of, e.g., Facebook. Based on this token, the application can then access Facebook to get some basic information of the user that allows to identify the user (e.g., e-mail address and more).

Marketplace REST – *JWT.java* (1)

- To create and verify JWTs, we are using a 3rd party library as a basis: Java JWT (JJWT)

```
public class JWT {
    private static final String ISSUER = "Marketplace";
    private static final long VALIDITY = 3600; // 1 hour
    private static final SignatureAlgorithm ALGORITHM =
                         SignatureAlgorithm.HS256;
    private static final byte[] KEY = "marketplace".getBytes();


    public static String createJWT(String username) {
        String jwt = Jwts.builder()
                    .setIssuer(ISSUER)
                    .setSubject(username)
                    .setExpiration(Date.from(Instant.now()
                                   .plusSeconds(VALIDITY)))
                    .signWith(ALGORITHM, KEY)
                    .compact();
        return jwt;
    }
```

Constant values to be used when creating / verifying a token

To create a token, the Java JWT library is used, *compact* creates the encoded token

JWT Libraries

There are several libraries available. Here, we are using Java JWT (JJWT): *https://github.com/jwtk/jjwt*

Key

For simplicity, we store the key in the code (and the key "marketplace" is obviously very insecure). This is not a very smart idea, especially as changing the key requires changing the code. Alternatively, one could store it in the file system(with minimal file access permissions) or in a hardware security module (HSM).

Marketplace REST – *JWT.java* (2)

```
    public static String validateJWTandGetUsername(String jwt) {
        String username;
        try {
            return Jwts.parser().setSigningKey(KEY).parseClaimsJws(jwt).
                   getBody().getSubject();
        } catch (MalformedJwtException | SignatureException |
                 ExpiredJwtException e) {
        }
        return null;
    }
}
```

If anything is wrong with the token, an exception is thrown and null is returned

This checks that the token has a valid format, that the signature is valid and that the token has not yet expired – if everything is OK, the username is extracted from the token and returned

Marketplace REST – *AuthenticationRest.java* (1)

- *AuthenticationRest.java* provides the REST interface to authenticate users and to create JWTs
  - To authenticate, a POST request with username / password is used
  - If the credentials are correct, a JWT is created and returned to the client

```
@RequestScoped
@Path("authenticate")
public class AuthenticationRest {

    @Inject
    private UserInfoFacade userInfoFacade;

    @Inject
    private UserRoleFacade userRoleFacade;

    @Inject
    private Pbkdf2PasswordHash pbkdf2PasswordHash;
```

@Path specifies the URL: *Marketplace_v11-rest/rest/authenticate*

Objects of types *UserInfoFacade* and *UserRoleFacade* are injected, which use entities *UserInfo* and *UserRole* and which provide DB access to the tables *UserInfo* and *UserRole* (see notes)

An object of type *Pbkdf2PasswordHash* is injected, which is part of Jakarta EE and which provides functionality to verify PBKDF2 hashes

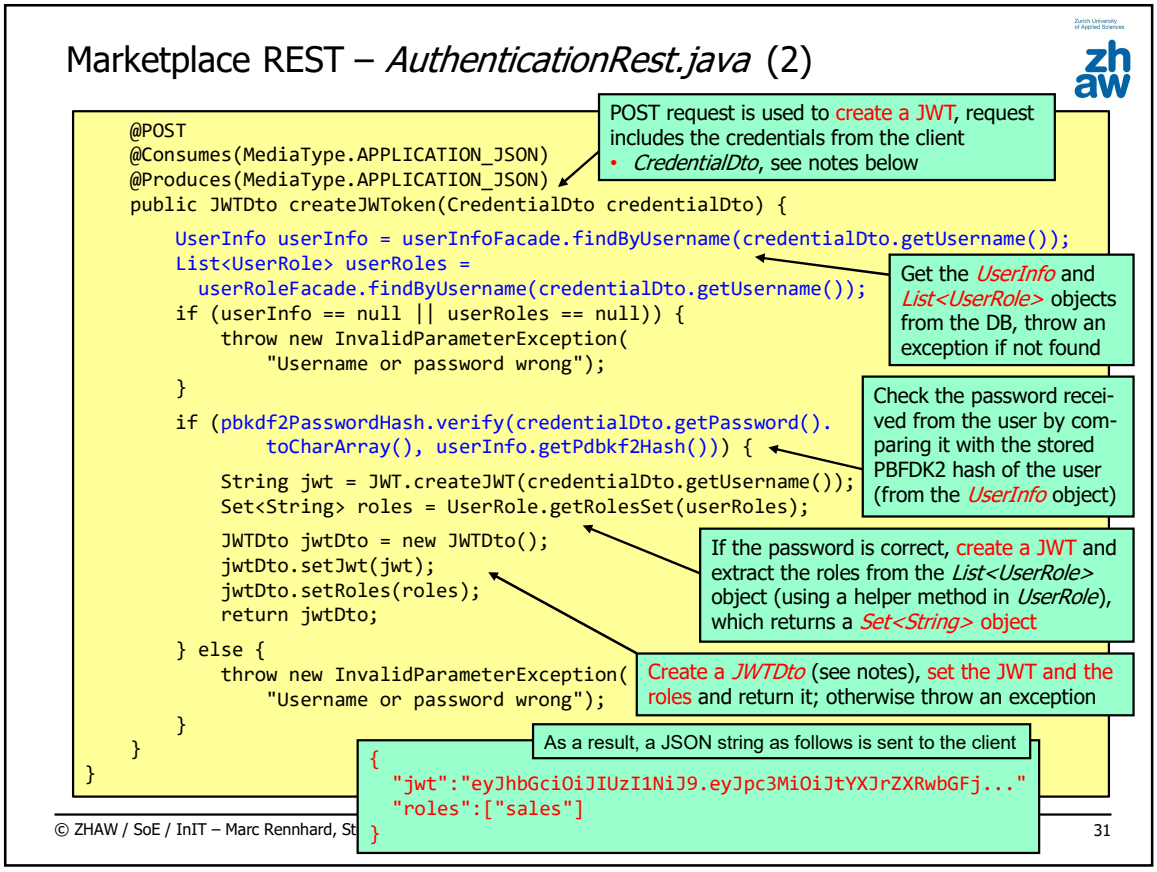
UserInfoFacade.java

```
@Stateless
public class UserInfoFacade extends AbstractFacade<UserInfo> {
    private static final long serialVersionUID = 1L;
    @PersistenceContext(unitName = "marketplace")
    private EntityManager entityManager;

    public UserInfoFacade() {
        super(UserInfo.class);
    }

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

    public UserInfo findByUsername(String username) {
        Query query = entityManager.createNamedQuery("UserInfo.findByUsername");
        query.setParameter("username", username);
        return getSingleResultOrNull(query);
    }
}
```

UserInfo.java

```
@Entity@Table(name = "UserInfo")
@NamedQuery(name = "UserInfo.findByUsername", query = "SELECT u FROM UserInfo u WHERE u.username = :username")
public class UserInfo implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id private String username;
    private String pbkdf2Hash;

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPdbkf2Hash() {
        return pbkdf2Hash;
    }

    public void setPdbkf2Hash(String pbkdf2Hash) {
        this.pbkdf2Hash = pbkdf2Hash;
    }
}
```

Note that *UserRoleFacade.java* and *UserRole.java* are implemented likewise (details see notes of one of the following slides).



Including the Roles

The roles are only included as an information to the client, so it can, e.g., adapt the view depending on the role. For instance, as only *sales* (but not *marketing*) users are allowed to delete purchases, the information about the role can be used to display the corresponding button or not. But for actual authentication and authorization when sending a request, only the JWT is relevant.

CredentialDTO.java

```
public class CredentialDto {
    @NotNull(message = "Element with key 'username' is missing")
    private String username;
    @NotNull(message = "Element with key password' is missing")
    private String password;

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }
}
```

JWTDto.java

```
public class JWTDto {
    private String jwt;
    private Set<String> roles;

    public String getJwt() {
        return jwt;
    }

    public void setJwt(String jwt) {
        this.jwt = jwt;
    }

    public Set<String> getRoles() {
        return roles;
    }

    public void setRoles(Set<String> roles) {
        this.roles = roles;
    }
}
```

Marketplace REST – Authenticated Requests

- Once the client has a valid JWT, the token can be included in the HTTP requests to access protected resources
  - To include authentication tokens, one typically uses the HTTP Authorization header using the BEARER authentication scheme
  - *Authorization: Bearer <token>*
- In contrast to FORM-based authentication as used in the traditional Marketplace web application, JWT-based authentication is not supported by the Jakarta EE Security API «out of the box»
  - Therefore, this must be provided on our own by implementing a class that extends the interface *HttpAuthenticationMechanism*
- This class should work as follows:
  - Must implement the method *validateRequest* (defined by the interface), which is called for every request that is received
  - If this request targets a resource that is not protected, do nothing
  - Otherwise, check the JWT and – if it is valid – inform the underlying servlet (that handles the REST request) that the request should be treated as an authenticated request
    - This authentication information is only valid for the current request – which is exactly what we want with a RESTful web service

HttpAuthenticationMechanism

This interface is provided by the Jakarta EE Security API with the goal to support a wide variety of authentication mechanism. Several implementations of the interface are included in Jakarta EE «out of the box» (e.g., CustomFormAuthenticationMechanism, which we are using in the Marketplace web application), but there is none to authenticate JWTs, which is why we have to implement this on our own.

Marketplace REST – *JWTAuthenticationMechanism.java* (1)

A class that implements *HttpAuthenticationMechanism* must be *@ApplicationScoped*

Constant values to identify protected resources and the scheme of the Authorization header

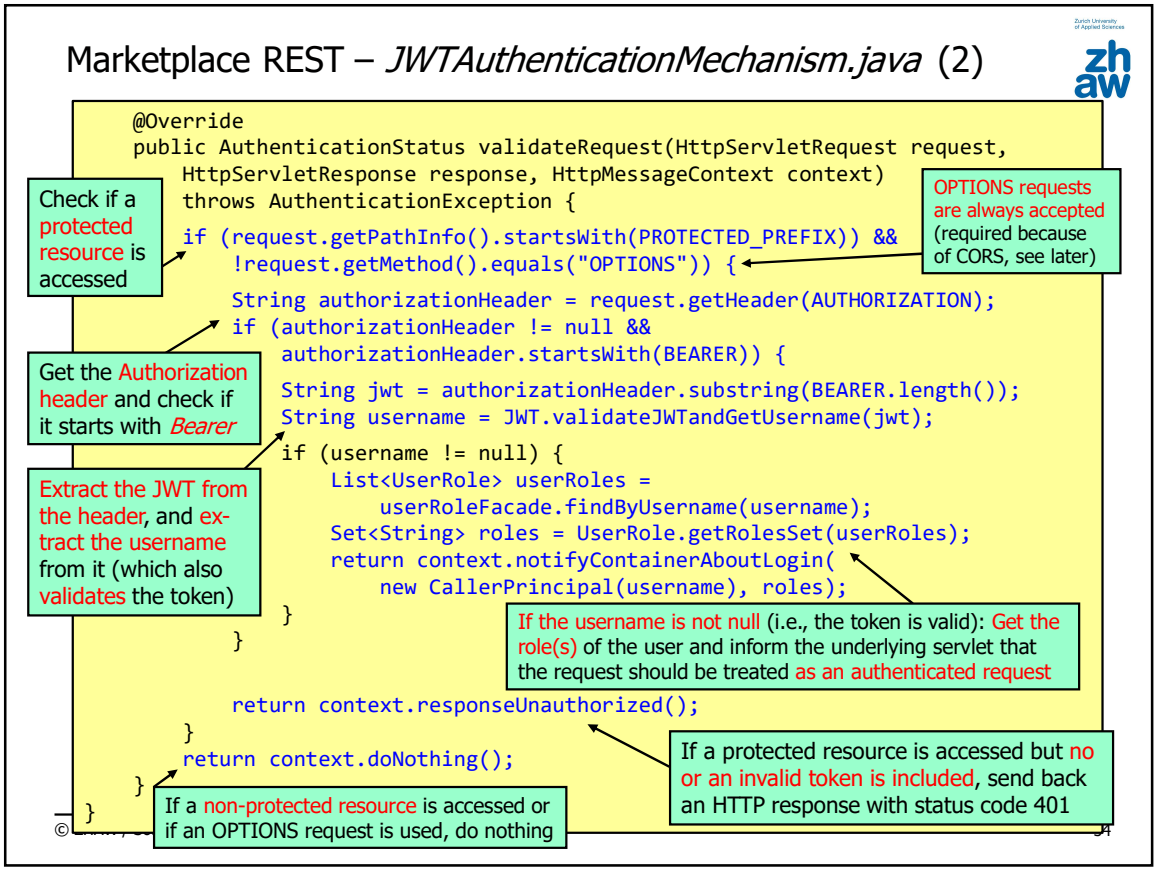
```
@ApplicationScoped
public class JWTAuthenticationMechanism implements
    HttpAuthenticationMechanism {

    private static final String PROTECTED_PREFIX = "/admin";
    private static final String BEARER = "Bearer ";

    @Inject
    private UserRoleFacade userRoleFacade;
```


UserRoleFacade.java

```
@Statelesspublic class UserRoleFacade extends AbstractFacade<UserInfo> {
    private static final long serialVersionUID = 1L;
    @PersistenceContext(unitName = "marketplace")
    private EntityManager entityManager;

    public UserRoleFacade() {
        super(UserInfo.class);
    }

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

    public List<UserRole> findByUsername(String username) {
        Query query = entityManager.createNamedQuery("UserRole.findByUsername");
        query.setParameter("username", username);
        return query.getResultList();
    }
}
```

UserRole.java

```
@Entity
@Table(name = "UserRole")
@NamedQuery(name = "UserRole.findByUsername", query = "SELECT u FROM UserRole u WHERE u.username = :username")
public class UserRole implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id private String username;
    private String rolename;

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getRolename() {
        return rolename;
    }

    public void setRolename(String rolename) {
        this.rolename = rolename;
    }

    public static Set<String> getRolesSet(List<UserRole> userRoles) {
        Set<String> roles = new HashSet();
        for (UserRole userRole : userRoles) {
            roles.add(userRole.getRolename());
        }
        return roles;
    }
}
```



Getting the Role

It would also be possible to include the roles in the JWT when the token is issued. This has the advantage that getting the roles from the DB wouldn't be required in the method *validateRequest*. However, this also means that it wouldn't be possible to easily remove a role from a user at any time, because the already issued JWTs would still be valid (and remember that the validity period could be chosen much longer than one hour). So the advantage of our approach is that if a role of a user is changed at any time or if all roles are removed from the user, the user cannot access corresponding resources anymore using an already issued JWT.

Return Value of *validateRequest*

The *validateRequest* method is invoked early by the servlet that handles the request (e.g., the Faces Servlet in JSF or the servlet that handles a REST request when using JAX-RS). This servlet also gets the return value of *validateRequest*. If *context.doNothing* is called, this return value is *NOT_DONE*, in the case of *context.responseUnauthorized* it's *SEND_FAILURE*, and in the case of *context.notifyContainerAboutLogin* it's *SUCCESS*.

HttpMessageContext

Basically, *HttpMessageContext* , which is passed to the method *validateRequest* as a parameter, provides access to the underlying servlet to inform it about logins. The Jakarta EE API documentation explains it as follows: *HttpMessageContext* contains all of the per-request state information and encapsulates the client request, server response, container handler for authentication callbacks, and the subject representing the caller.

Marketplace REST – Accessing Protected Resources Summary

To summarize, the following happens when accessing a protected resource over REST (e.g., `GET /rest/admin/purchases`):

- When the web service receives the request, method *validateRequest* of *JWTAuthenticationMechanism* is invoked
- If the request contains a valid token, the underlying servlet that handles the REST request is informed that the request is authenticated and in addition, the servlet gets the username and role(s)
  - Otherwise, an HTTP response with status code 401 is sent back
- Before access to the resource is granted, the *@RolesAllowed* annotation of the method that handles the request is evaluated
- Access is only granted if the established role(s) of the user allow access to the resource according to *@RolesAllowed*
  - Otherwise, an HTTP response with status code 403 is sent back
- As no sessions are used over REST, the established user and role(s) are only valid for the current request

Marketplace REST – Almost there... (1)

- With this, the Marketplace REST API is almost completed
  - All required functionality is supported
  - Security has been addressed correctly (SQL injection (JPA), input validation (Bean Validation), authentication (JWT), authorization (*@SecurityRoles*), HTTPS,...)
  - We haven't addressed security issues such as XSS and CSRF, but will come back to this later when discussing the client-side

- However, there's one problem remaining we have to address: Making sure that the correct JSON error messages are used in all cases (currently, this uses HTML messages), in particular:
  - If a Bean Validation rule is violated (*ConstraintViolationExceptions*)
  - If an *InvalidParameterException* is thrown (e.g., wrong password)
  - If *context.responseUnauthorized()* is called in *JWTAuthenticationMechanism.java*
  - If a user tries to access a resource protected with *@RolesAllowed* without having one of the required roles

Marketplace REST – Almost there... (2)

- As an example, this is the response currently generated if a wrong username or password is entered (shortened):

```
HTTP/1.1 500 Internal Server Error
Content-Type: text/html
Content-Length: 1520

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"... java.security.
InvalidParameterException: Username or password wrong</pre></p><p><b>root
cause</b>... <h3>Payara Server #badassfish</h3></body></html>
```

  - This has the wrong status code, the wrong content-type, and the wrong response body

- Instead, we are expecting the following JSON response:

```
HTTP/1.1 400 Bad Request
Content-Type: application/json
Content-Length: 38

{"error":"Username or password wrong"}
```

- This is all not security-relevant, but should be addressed nevertheless

Marketplace REST – Exception Mapper and *RestErrorDto.java*

- To handle exception and send back a custom message to the client, JAX-RS provides the interface *ExceptionMapper*
  - To handle specific exceptions, one has to create a class which implements this interface
- In the Marketplace application, we write two exception mappers, one to handle Bean Validation exceptions and the other to handle *InvalidParameterExceptions*
- Errors should be presented to the client in the following form:

```
{"error":"Please insert a valid credit card number (16 digits)"}
```

- For this, we use a DTO *RestErrorDto.java*:

```
public class RestErrorDto {
    private String error;

    public void setError(String error) { this.error = error; }

    public String getError() { return error; }
}
```

Marketplace REST – *ConstraintViolationExceptionMapper.java*

- This exception mapper handles *ConstraintViolationExceptions*, which are thrown if Bean Validation fails
  - It basically reads the *ConstraintViolations* from the Exception, concatenates the messages of the *ConstraintViolations*, creates a *RestErrotDto* object and sets the concatenated message, and sends the object to the client (as JSON) in a response with status code 400

```
@Provider
public class ConstraintViolationExceptionMapper implements
             ExceptionMapper<ConstraintViolationException> {
    @Override
    public Response toResponse(ConstraintViolationException exception) {
        Set<ConstraintViolation<?>> constraintViolations =
                                    exception.getConstraintViolations();
        StringBuilder errorMessage = new StringBuilder();
        for (ConstraintViolation<?> constraintViolation : constraintViolations) {
            if (errorMessage.length() > 0) {
                errorMessage.append(", ");
            }
            errorMessage.append(constraintViolation.getMessage());
        }
        RestErrorDto error = new RestErrorDto();
        error.setError(errorMessage.toString());
        return Response.status(Response.Status.BAD_REQUEST).entity(error).build();
    }
}
```


@Provider

ExceptionMappers must use the *@Provider* annotation, so they are treated as ExceptionMappers for JAX-RS.

Marketplace REST – *InvalidParameterExceptionMapper.java*

- This exception mapper handles *InvalidParameterExceptions*
  - Thrown, e.g., when authentication fails or when only invalid product codes are used
  - It simply uses the message from the exception and inserts it into a *RestErrotDto* object, and sends the object to the client (JSON / 400)

```
@Provider
public class InvalidParameterExceptionMapper implements
    ExceptionMapper<InvalidParameterException> {
    @Override
    public Response toResponse(InvalidParameterException exception) {
        RestErrorDto error = new RestErrorDto();
        error.setError(exception.getMessage());
        return Response.status(Status.BAD_REQUEST).entity(error).build();
    }
}
```

- In addition, both classes must be registered in *ApplicationConfig.java*:

```
private void addRestResourceClasses(Set<Class<?>> resources) {
  ...
  resources.add(ch.zhaw.securitylab.marketplace.rest.
                                    ConstraintViolationExceptionMapper.class);
  resources.add(ch.zhaw.securitylab.marketplace.rest.InvalidParameterExceptionMapper.class);
}
```

Marketplace REST – 401 and 403 Responses

- As a final step, we want to provide JSON responses in case a 401 or 403 response is used:

```
{"error":"Authentication required"}
```

```
{"error":"Access denied"}
```

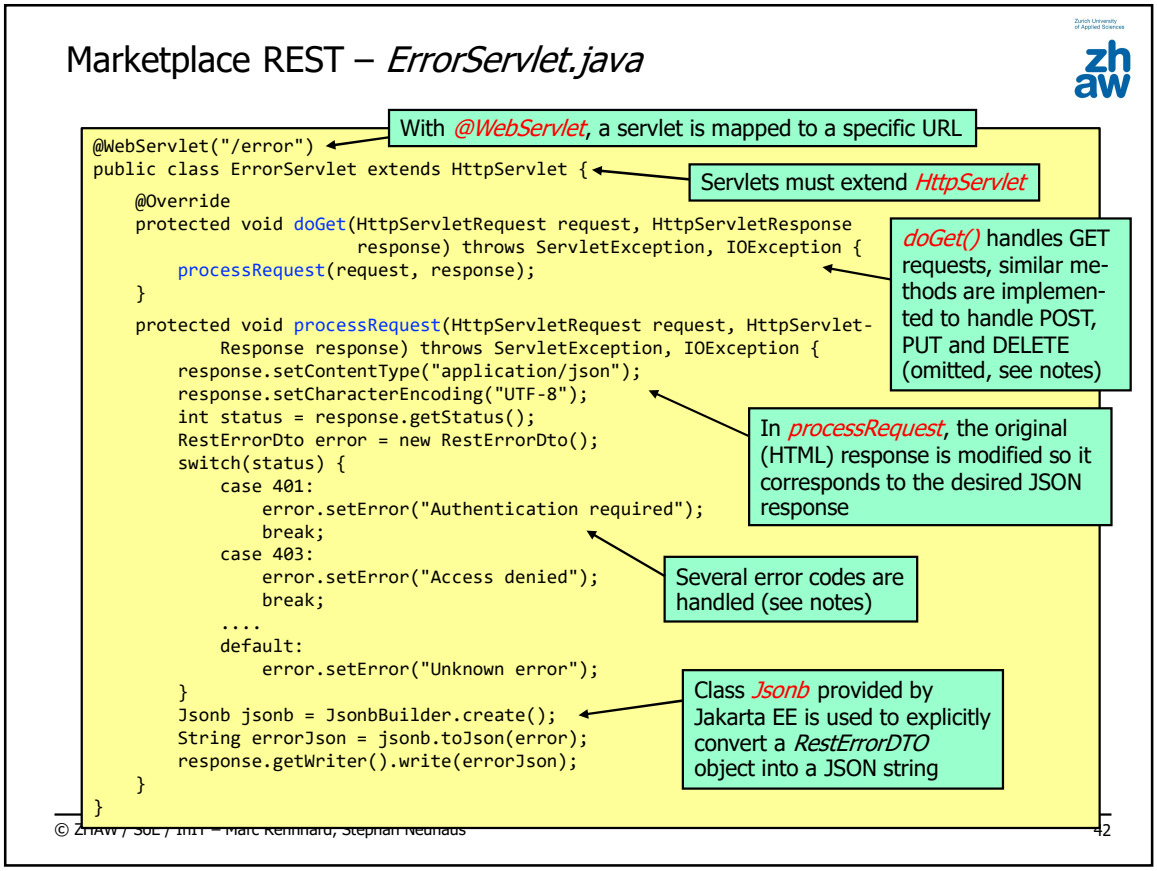
- The reason for the current HTML responses is because of the call of *context.responseUnauthorized()* in *JWTAuthenticationMechanism.java* and because of *@RolesAllowed* annotations
  - This behavior cannot be «overwritten» within the REST application
  - But the responses can be modified using an *<error-page>* element in *web.xml*, which points to a servlet
- Requires the following extension in *web.xml*:

```
<error-page>
    <location>/error</location>
</error-page>
```

Effect:
- If any error response (4xx/5xx) should be sent back...
- ...the resource with URL */error* is called...
- ...which is a servlet that creates the desired response

Precedence of ExceptionMappers

Note that this *<error-page>* element does not affect the ExceptionMappers. So, if an ExceptionMapper is handling an exception, then this has precedence over this *<error-page>* element, i.e., the response is not adapted again after it has been adapted by the ExceptionMapper.



Servlet Methods

Besides *doGet*, there's *doPost*, *doPut*, and *doDelete*.

Complete Class

```
@WebServlet("/error")
public class ErrorServlet extends HttpServlet {
  @Override
  protected void doGet(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
    processRequest(request, response);
  }
  @Override
  protected void doPost(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
    processRequest(request, response);
  }
  @Override
protected void doDelete(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
    processRequest(request, response);
  }
  @Override
  protected void doPut(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
    processRequest(request, response);
  }

  protected void processRequest(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
    response.setContentType("application/json");
    response.setCharacterEncoding("UTF-8");
    int status = response.getStatus();
    RestErrorDto error = new RestErrorDto();
    switch(status) {
      case 401:
        error.setError("Authentication required");
        break;
      case 403:
        error.setError("Access denied");
        break;
      case 404:
        error.setError("Not found");
        break;
      case 405:
        error.setError("Method not allowed")
        break;
      default:
        error.setError("Unknown error")
    }
    Jsonb jsonb = JsonbBuilder.create();
    String errorJson = jsonb.toJson(error);
    response.getWriter().write(errorJson);
  }
}
```

Marketplace REST – Tests (1)

- GET products → works

```
GET products, filter = 'DVD', auth = false, credentials = ''
Status code: 200
Media type: application/json
Body: [{"code":"0001","description":"DVD Life of Brian - used, some scratches but still wo
rks","price":5.95}]
=> PASSED
```

- POST purchase → works

```
POST purchases Max|Meier|1111 2222 3333 4444|0001,0003, auth = false, credentials = ''
Status code: 204
Media type:
Body:
=> PASSED
```

- Authentication with correct credentials → works

```
POST authenticate alice|rabbit
Status code: 200
Media type: application/json
Body: {"jwt":"eyJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJNYXJrZXRwbGFjZSIsInN1YiI6ImFsaWNlIiwiZXhwIjo
xNTg1ODE0NDk1fQ.m-CV5D8hdq8gj-1ZwA5VqfmnS1mpXH5zBQYGKi7TkGg","roles":["sales"]}
=> PASSED
```

Marketplace REST – Tests (2)

- Authentication with wrong credentials → works

```
POST authenticate alice|wrongpassword
Status code: 400
Media type: application/json
Body: {"error":"Username or password wrong"}
```

- GET admin/purchases with valid JWT → works

```
GET admin/purchases, auth = true, credentials = 'eyJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJNYXJrZXRw
bGFjZSIsInN1YiI6ImFsaWNlIiwiZXhwIjoxNTg1ODE0NDk1fQ.m-CV5D8hdq8gj-1ZwA5VqfmnS1mpXH5zBQYGKi7
TkGg'
Status code: 200
Media type: application/json
Body: [{"creditCardNumber":"1111 2222 3333 4444","firstname":"Ferrari","lastname":"Driver"
```

- GET admin/purchases with invalid JWT → works

```
GET admin/purchases, auth = true, credentials = 'eyJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJtYXJrZXR
wbGFjZSIsInN1YiI6ImFsaWNlIiwiZXhwIjoxNTEzNjkxOTkyfQ.XvuZRhhSreAdg6HVg8n7OAeaLzQ789eZQDnDJ
hVTY1J'
Status code: 401
Media type: application/json;charset=UTF-8
Body: {"error":"Authentication required"}
```

Marketplace REST – Tests (3)

- DELETE purchase with wrong role (*marketing*) → works

```
DELETE admin/purchases, id = 1, auth = true, credentials = 'eyJhbGciOiJIUzI1NiJ9.eyJpc3Mi
OiJNYXJrZXRwbGFjZSIsInN1YiI6InJvYmluIiwiZXhwIjoxNTE5NzQwNDA3fQ.eSVpkM_J3lSIR_wEvCU2PoR3Hd
LY0qUc-YqiHT2T9QE'
Status code: 403
Media type: application/json;charset=UTF-8
Body: {"error":"Access denied"}
```

- GET products with too long search string → works

```
GET products, filter = 'looooooooooooooooooooooooooooooooooooooooooooooooooooo', auth = fal
se, credentials = ''
Status code: 400
Media type: application/json
Body: {"error":"The search string must not be longer than 50 characters"}
```

- POST purchase with invalid credit number → works

```
POST purchases Max|Meier|1111 2222 3333 4445|0001,0003,9999, auth = false, credentials = '
'
Status code: 400
Media type: application/json
Body: {"error":"Please insert a valid credit card number (16 digits)"}
=> PASSED
```

Client-Side Security of the Marketplace SPA

Client-Side Security in Single Page Applications (1)

Preliminary remark: We are considering here «pure» SPAs, which means the following:

- The web application itself consists of «one minimalistic HTML page» (usually something such as *index.html*) and several JavaScript files
- Initially, the browser fetches this page and the JavaScript files from the server
- User actions do not result in directly sending an HTTP request to the server; instead, they are handled by JavaScript, which manipulates the DOM to show the results of the user action
- If data has to be read from/written to the server, JavaScript code sends the corresponding request to a stateless REST API provided by the server-side, and also receives and processes the responses
- Many «traditional web application» elements are missing, i.e., no sessions, no «full-fledged» HTML pages that are served to the browser, no direct submission of forms to the server, etc.
  - If such elements are used, then the corresponding security safeguards must be considered (see previous chapter)!

Client-Side Security in Single Page Applications (2)

- The prerequisite for a secure SPA is a secure REST API
  - Protection from injection attacks, authentication, authorization, input validation, HTTPS, etc.
  - With a secure REST API, the server-side is «basically secure» and as a result of this, there are not so many realistic attack vectors remaining in an SPA

- However, things that can go wrong at the client-side with respect to security and it's important that you aware of them
  - When saying «at the client-side», we mean attacks that take advantage of general browser features or that exploit vulnerabilities in the code that is running in the browser of a victim (mainly CSRF and XSS)
    - In contrast to attacks where the attacker directly targets the REST API
  - Here, we are looking at these issues in general
  - In addition, we are illustrating the issues using a Marketplace SPA implemented with Angular (that uses the Marketplace REST API)

Cross-Origin Resource Sharing (1)

- A very important concept that must be taken into account in SPAs is Cross-Origin Resource Sharing (CORS)
- An origin is the combination of protocol, host and port
  - E.g., *http://www.domain1.com*, *https://www.domain2.com:8181*
  - Two origins are only equal if all three parts are the same!
- Traditional web applications work as follows
  - Whenever the user clicks a link (*<a>*), submits a form (*<form>*), an image is embedded in an HTML page (*<img>*), an iframe is embedded (*<iframe>*), JavaScript submits a form, etc. ...
  - ...then the browser will always execute the corresponding GET or POST request, even if it is a cross-origin request
- E.g., if the HTML page is received from *https://www.secure.com*, the request will always be issued, even if it goes to *http://very.evil.org:85*
  - This behavior is intentional, as it is an important prerequisite to enable web applications that combine content from various origins...
  - ...but it's also the reason why, e.g., cookies can be stolen with XSS attacks and why CSRF attacks work unless there are countermeasures

CORS

More detailed information can be found here:
https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS

Cross-Origin Resource Sharing (2)

- On the other hand, if requests are issued by JavaScript using the *XMLHttpRequest* object or the *Fetch API* (aka «AJAX requests»), only a subset of all requests are issued cross-origin per default
- These request are called «simple requests» and have the following properties (simplified):
  - The request must be a GET or a POST request
  - It can be an authenticated request that includes the cookie stored in the browser, but it cannot include an *Authorization* header (e.g., with a JWT)
  - The request can only use a limited set of *Content-Type*s, e.g., *application/x-www-form-urlencoded*, but not *application/json*
  - The response is not made available to the JavaScript code in the browser, i.e., the JavaScript code can issue a request, but it does not get access to the response
- This basically allows the same cross-origin requests that are also possible in traditional web applications when using, e.g., links, forms and *img* tags, so it's reasonable to permit them as no new risks are introduced
- All other «non-simple requests» cannot be issued by JavaScript per default
  - This includes all PUT/PATCH/DELETE requests, all non-simple GET/POST requests, and all requests including JSON data

No New Risks are Introduced

As an example, considering a traditional web application, if one has a form in a web page that submits the form using a POST request to another origin, then this is allowed in every web application. And it doesn't matter whether the form is submitted by the user (clicking a button) or automatically by JavaScript code. But the result of the request (the HTTP response and the enclosed HTML document) is then simply shown in the browser and the response is not made available to the web page that issued the request (as the old web page is no longer active).

Therefore, it is allowed to issue the same POST request also by using *XMLHttpRequest*. And to make sure the we have the «same overall risk» as with traditional web applications, the JavaScript code in the web page that issued the request dues not get access to the response.

This also means that when only looking at simple requests, *XMLHttpRequest* allows the same CSRF attacks as what can be done with attack techniques that use «traditional» web application features. And if CSRF protection is implemented, then this will also block attacks that use *XMLHttpRequest* to trigger the malicious request.

Cross-Origin Resource Sharing (3)

- Example: Assume you are authenticated in a traditional e-banking web application that uses cookies to identify a session
- In a second browser tab, you are accessing a malicious web page (e.g., by clicking a link in an e-mail you received), which contains JavaScript code that uses *XMLHttpRequest* to do a CSRF attack
- Assume this code sends a POST request to do a payment
  - Is the payment done? Yes – the request is sent (with the cookie), which is not good but the same can also be done with a «classic CSRF attack» that uses a form and that submits it by JavaScript code (so no new security problems are introduced and CSRF safeguards will prevent this)
- But not we have a problem: In modern web applications, virtually all requests issued by JavaScript code are non-simple and the used REST API(s) often use different origins than the web application itself
  - E.g., the web application is served from *frontend.example.com*, while the REST API is offered at *api.example.com*

Scenario with GET

As another scenario, assume the code sends a GET request to get the account balance. Does the attacker get the account balance? No – the request is sent (with the cookie) and the response is received by the browser, but the response is not made available to the JavaScript code, i.e., it is not accessible by the attacker. This is again the same behavior as in a «classic CSRF attack» that uses, e.g., a link or a FORM that generates a GET request: In those cases, the request is also sent and processed, but the result is only made available to the user of the web application (the victim) by displaying it in the browser. The attacker won't get access to it.

Using one Origin

If the web application and the REST API are provided by the same origin, e.g., both by *https://www.example.com*, then there are no cross-origin requests, and everything will work out of the box. But if the origins are different, there will be issues with cross-origin requests. In the remainder, we therefore assume the origins are different, but all the discussed security measures should be considered in any case, no matter how many origins are used.

Cross-Origin Resource Sharing (4)

- To fix this, CORS was defined as a W3C standard
- Basic idea: Non-simple cross-origin requests can be sent by the browser, but only if this allowed by the target REST API
  - The REST API communicates this to the browser using response headers
- To allow such requests, we implemented the class *CORSFilter.java* in the REST API, which includes the required response headers

```
response.getHeaders().putSingle("Access-Control-Allow-Origin", "*");
response.getHeaders().putSingle("Access-Control-Allow-Methods",
                                "OPTIONS, GET, POST, PUT, DELETE");
response.getHeaders().putSingle("Access-Control-Allow-Headers", "*");
```

  - This informs the browser that the REST API can be accessed cross-origin...
    - ... from any origin – so no matter where a page that is processed in the browser comes from, cross-origin requests will be allowed from this page
    - ... using any of the methods OPTIONS/GET/POST/PUT/DELETE
    - ... using any header (including, e.g., an *Authorization* header that includes a JWT or a *Content-Type* header with *application/json*)

Cross-Origin Resource Sharing (5)

- One problem remains: How can the browser find out whether a specific cross-origin request is allowed?
  - Naïve idea: Send the request and then learn from the response whether it was allowed or not? Bad idea, as the request has already been processed
- Solution: Preflight requests – the browser first asks the REST API whether a specific request is permitted – this uses OPTIONS requests
- Example: Check whether a purchase may be deleted in Marketplace

First, the browser sends this request to inform the REST API it would like to send a DELETE request to the specified URL using an Authorization header

```
OPTIONS /Marketplace_v11-rest/rest/admin/purchases/3 HTTP/1.1
Access-Control-Request-Method: DELETE
Access-Control-Request-Headers: authorization
```

As a result, the REST API responds with the configured response headers, which tells the browser cross-origin DELETE requests are allowed from any origin using any header → the actual DELETE request will be sent!

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: *
Access-Control-Allow-Methods: OPTIONS, GET, POST, PUT, DELETE
Access-Control-Allow-Headers: *
```

- Remarks:
  - The request headers are often not evaluated at all by the REST API and it simply always sends back the configured response headers (as done here)
  - Make sure that OPTIONS requests are always allowed, even to the protected area (which is what was done in *JWTAuthenticationMechanism.java*)
  - Beyond this, preflight requests are handled automatically by JAX-RS: If an OPTIONS request is sent to an existing resource, the correct response is sent back

DELETE Request

Here, the preflight request includes the special header the subsequent DELETE request wants to use: The *Authorization* header with the JWT, as such a header is not allowed in requests per default. If the response does not allow usage of this header (e.g., by returning *Access-Control-Request-Headers: content-type*), then the DELETE request would not be issued by the browser, as apparently, the *Authorization* header is not permitted.

Another Example: POST Request during Checkout

To complete a purchase, the following preflight request is sent:

```
OPTIONS /Marketplace_v11-rest/rest/purchases HTTP/1.1
Access-Control-Request-Method: POST
Access-Control-Request-Headers: content-type
```

Here, the preflight request asks the REST API whether the header *Content-Type* may be used, i.e., whether all kinds of content types are allowed, which includes application/json. The preflight request does not ask whether the *Authorization* header is permitted, as the POST request is not an authenticated request.

Evaluating the Request Headers sent by the Client

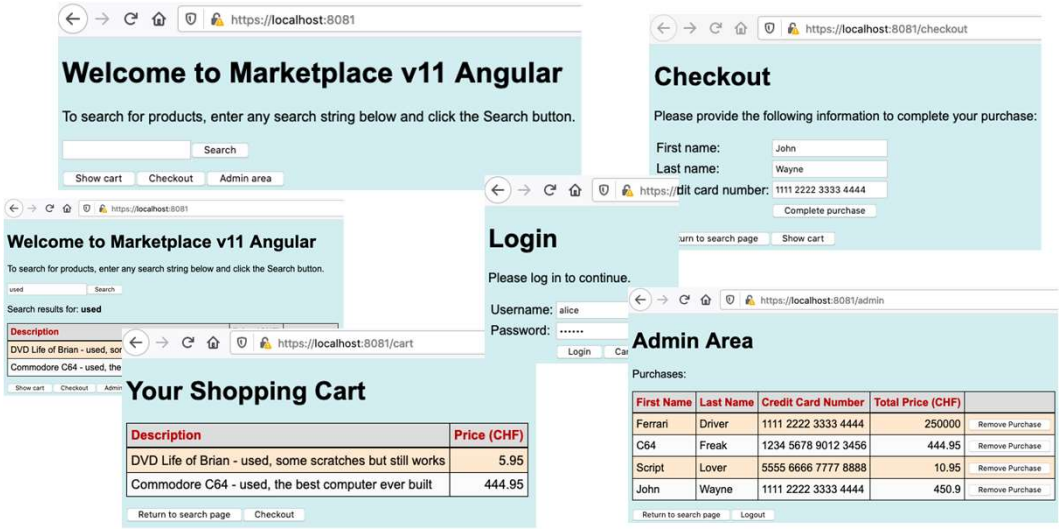
In the case of the relatively simple Marketplace REST API, we simply send back always the same three *Access-Control-Allow* headers and correspondingly, this has been hardcoded in *CORSFilter.java*. For instance, an API may allow GET requests that don't use an *Authorization* header from anywhere and respond accordingly with *Access-Control-Allow-Origin: **. In addition, it allows GET requests that use an *Authorization* header and that target URLs below */application/secure-area-1/* only from the specific origin *https://www.only-from-here.com*, and in this case, the response header could be *Access-Control-Allow-Origin: https://www.only-from-here.com*. And furthermore, it allows GET requests that use an *Authorization* header and that target URLs below */application/secure-area-2/* only from the specific origin *https://www.only-from-there.com*, and in this case, the response header could be *Access-Control-Allow-Origin: https://www.only-from-there.com*. So, as you can see, the REST API can decide what content of the request should be evaluated and what the response should be. But in the case of the simple Marketplace REST API, responding always with the same Access-Control-Allow header is good and secure enough.

JAX-RS and Preflight Requests

JAX-RS checks that the resource specified in the OPTIONS request is existing. If that's not the case, it responds with a status code 404 and no follow-up request (e.g., the actual DELETE request) will be executed by the browser. This may be different with other REST API frameworks.

Cross-Origin Resource Sharing (6)

- With this CORS configuration, the Marketplace SPA works as intended
  - The Marketplace SPA is available at *https://localhost:8081*, the Marketplace REST API at *https://localhost:8181* (two different origins)



Running the Marketplace SPA

The application (basically the *index.html* page and all the JavaScript code files) can be made available using basically any web server. For simplicity, we are using *live-server*, which is a *node.js* package, see *https://www.npmjs.com/package/live-server*.

To build the application and to run it using live server (using HTTPS), do the following steps in a terminal:

```
Marketplace_Angular % ng build --prod
Marketplace_Angular % cd dist/marketplace
Marketplace_Angular % live-server --https=/usr/local/lib/node_modules/live-server-https --port=8081 --entry-file=index.html
```

Introduction to Angular

An excellent introduction to Angular was provided under the link *https://javabrains.io/courses/angular_basics*, but unfortunately, the course is no longer available there. The course videos are still available at YouTube, though. The first video can be found here: *https://www.youtube.com/watch?v=9RG3MiEBEIw*

CSRF in the Marketplace SPA

- With CORS, cross-origin requests work – but at the same time, arbitrary CSRF attacks are now possible:
  - An attacker sends to a Marketplace user an e-mail with a link that points to a malicious web page
  - The web page contains JavaScript code that sends any of the supported REST requests to the Marketplace REST API
    - Due to the current CORS configuration, such requests are permitted, including requests that use an *Authorization* header and that use JSON data
    - In addition, the malicious JavaScript code can access the response
  - This will work because the REST API does not use an anti-CSRF mechanism!

- This brings us to the next question: How can we mitigate this?

Mitigating CSRF in the Marketplace SPA (1)

- The first mitigation is very easy: Simply make sure that cross-origin requests cannot be done from anywhere, but only from the origins from where such requests are required
- With the Marketplace SPA, this can be done by allowing cross-origin request only from the origin where the web application is provided
  - I.e., only from *https://localhost:8081*, which we are using here for the server that serves the client-side of the Marketplace SPA
- To do this, *CORSFilter.java* is adapted as follows:

```
~~response.getHeaders().putSingle("Access-Control-Allow-Origin", "*");~~
response.getHeaders().putSingle("Access-Control-Allow-Origin", "https://localhost:8081");
```

- And as a result, a preflight response contains the following:

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: https://localhost:8081
Access-Control-Allow-Methods: OPTIONS, GET, POST, PUT, DELETE
Access-Control-Allow-Headers: *
```

- This renders CSRF almost impossible, because an attacker can no longer place the malicious web page anywhere, but would have to embed it on the server reachable at *https://localhost:8081*

Mitigating CSRF in the Marketplace SPA (2)

- But let's assume this cannot be done, because the REST API has to be usable from any origin – in this case, CSRF must be considered again
- What about REST requests to resources that do not require authentication – is CSRF an issue?
  - In the Marketplace REST API, this includes the GET request to read the products and the POST request to do a purchase
  - Are they relevant in the context of CSRF? No – because there's no benefit for the attacker by having them executed in the browser of a victim, as the attacker can simply send the requests directly himself
  - Therefore, CSRF attacks are not an issue in the context of requests to resources that don't require authentication
- But what about REST requests to resources that require authentication – is CSRF an issue there?
  - This requires a more detailed analysis

Mitigating CSRF in the Marketplace SPA (3)

- Let's assume the user has authenticated himself and the browser has received an authentication token (e.g., a JWT or any other token)
  - This is always a prerequisite for CSRF attacks to protected resources
- 1st case: A cookie is used store the authentication token in the browser
  - The REST API sets it using a *Set-Cookie* header, and the browser includes it in subsequent requests in a *Cookie* header (this can be done even if no server-side sessions are used)
  - If a CSRF attack is done in this case, the attacker's JavaScript code can easily make sure the cookie is included in the request

```
var xhr = new XMLHttpRequest();
xhr.open('GET', 'http://url');
xhr.withCredentials = true;
```

  - Does using the *HttpOnly* flag when setting the cookie help?
    - No – this prevents only that such a cookie can be stolen in an XSS attack, but it still can be included in a request using *xhr.withCredentials = true*
- Conclusion 1: With REST APIs, don't use cookies for authentication tokens!

Including the Cookie

Note that *xhr.withCredentials = true* will always include the cookie. But JavaScript code in the browser will only get access to the response if the preflight request returns a specific response header: *Access-Control-Allow-Credentials: true*. So, to «enable» a CSRF attack where JavaScript code gets access to the response, this header is required. But of course, if legitimate JavaScript code wants to access the response of CORS requests – which is typically needed – this also requires this response header. Therefore, when using cookies for the authentication token, this response header must virtually always be used, which means CSRF attacks can also access the response of CORS requests.

So, if the Marketplace SPA were using cookies for the authentication token (which, as stated above in the slide, is not a good idea), the following line would have to be added to *CORSFilter.java* to make this work:

```
response.getHeaders().putSingle("Access-Control-Allow-Credentials", "true");
```

What about the *Set-Cookie* Attribute *samesite*?

In general, when requests are issued with JavaScript code, cookies behave in exactly the same way as in «classic» web applications. This means that a cookie that uses *samesite=Lax* will only be sent cross-origin if the request is a GET request and cookies with *samesite=Strict* will never be sent cross-origin. In «classic» web applications, this attribute can provide some protection from CSRF attacks, so does it also help in REST APIs when cookies are used for authentication? Obviously, *samesite=Strict* does not make sense as in this way, no authenticated cross-origin requests would be possible. But would *samesite=Lax* help in REST APIs that only support GET requests? The answer is no. The reason is that in contrast to «classic» web applications where authenticated GET requests are typically not harmful in the context of CSRF attacks as the attacker cannot get access to the response (so using *samesite=Lax* makes sense), authenticated GET requests in REST APIs are often security-critical in the context of CRSF attacks, as the attacker can get access to the response (e.g., the account balance of the user in an e-banking application). Therefore, using *samesite=Lax* does not help at all against CSRF attacks, not even with REST APIs that only support GET. To summarize, the *samesite* attribute has no value in the context of REST APIs and – in case you really want to use cookies for authentication – it's best to set it to *samesite=None* to make sure that the API works correctly.

Mitigating CSRF in the Marketplace SPA (4)

- 2nd case: The authentication token is included in a header
  - Typically *Authorization: Bearer eyJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJNYXJr...*, as it is currently implemented in the Marketplace REST API
  - If a CSRF attack is done in this case, the attacker's JavaScript code can also easily include the required header into the request – provided it can get access to to the token!

```
var xhr = new XMLHttpRequest();
open('GET', 'http://url');
xhr.setRequestHeader('Authori-
zation', 'Bearer ' + token);
```

- This brings us to the next question: Once the user has authenticated at the REST API, where should the client-side application store the received authentication token...
  - ...so it can easily be accessed by legitimate requests...
  - ...but not by malicious requests by an attacker in the context of a CSRF attack

Storing the Auth Token in the Marketplace SPA (1)

- The obvious choice to store the authentication token is to use the Web Storage API, which is supported in every browser
- There are two different types of web storage:
  - Session storage: Stored data is private per browser window/tab and is deleted when the browser is closed
  - Local storage: Stored data is shared across browser windows/tabs that use the same origin (same protocol, host and port) and is not deleted when the browser is closed
  - To store authentication tokens, session storage should be used for security reasons, to make sure the token is removed when closing the browser
    - Gives the same behavior as with session cookies in traditional web applications
    - In some cases, local storage may be used (convenience vs. security)
- Is the token stored in web storage accessible during a CSRF attack?
  - No, because the JavaScript code of the attacker comes from a different origin than the Marketplace SPA (so even when local storage is used, access is not possible)

Session Storage

Another advantage of session storage vs. local storage is that with session storage, some XSS attacks (e.g., reflected XSS) won't give access to the stored authentication token. This will be explained in the notes on a later slide.

Storing the Auth Token in the Marketplace SPA (2)

- With this, legitimate requests can access the stored token and CSRF attacks won't work, but what about Cross-Site Scripting (XSS) attacks?
  - They obviously work, because the JavaScript code injected by an attacker can also access the Web Storage API
- So, is there something better than storing the token in session storage?
  - In a variable in memory? Not better, as the injected JavaScript code can access this as well and the token will be removed during a page reload
  - Use a cookie with the HttpOnly flag set for the token instead of an *Authorization* header? This would prevent this attack, but...
    - ...it would re-introduce the possibility for CSRF attacks
    - ...XSS attacks would still be devastating, as the injected JavaScript by the attacker could send authenticated requests at will, which is almost as good as stealing the token
- Overall, storing the authentication token in session storage is the best option, but this brings us to...
- Conclusion 2: With SPAs, make sure that XSS is prevented!

Cookies vs. Web Storage vs. Local Variable

The web is full of discussions about what is the best strategy of storing the authentication token locally and exchanging it with the REST API. There are different opinions and there's obviously not a single best strategy. Basically, when using cookies, there's the problem with CSRF attacks and when storing the token in web storage (or in memory), there's the problem with XSS which allows to steal the JWT.

Overall, the approach we are proposing here seems to be the best compromise, for two reasons: First of all, we won't have to deal with CSRF by using a CSRF token-like approach, as CSRF attacks to authenticated areas won't work. And second, we have to make sure XSS is prevented anyway with all approaches. So overall, using an authorization header is the easier approach, as one gets CRSF protection for free. Also, cookies have problems that logging out requires a request to the REST API so the cookie is overwritten with an invalid value (as local deletion does not work because of the HttpOnly flag). And if the authentication server is on a different domain, cookies also don't work because a cookie that is set in the browser is bound to the domain that sent it.

This is quite an interesting video that explains that XSS is a problem in any case:
https://www.youtube.com/watch?v=M6N7gEZ-IUQ

CSRF Safeguards

Using the approach discussed here, CSRF should not be a problem in practice, However, one can of course still use an additional explicit CSRF protection mechanism.

Storing the Auth Token in the Marketplace SPA (3)

- Implementing this in Angular (and other frameworks) is easy
  - Technical detail: In Angular, this is implemented using a service, so the functionality can easily be used throughout the application

```
setAccessTokenRoles(accessToken: string, roles: string[]) {
    sessionStorage.setItem('accesstoken', accessToken);
    sessionStorage.setItem('roles', roles.join(","));
}

getAccessToken() {
    let accessToken;
    accessToken = sessionStorage.getItem('accesstoken');
    if (accessToken == null) {
        accessToken = "";
    }
    return accessToken;
}

removeAccessTokenRoles() {
    sessionStorage.removeItem('accesstoken');
    sessionStorage.removeItem('roles');
}
```

Setting the token and the roles of the user after successful login

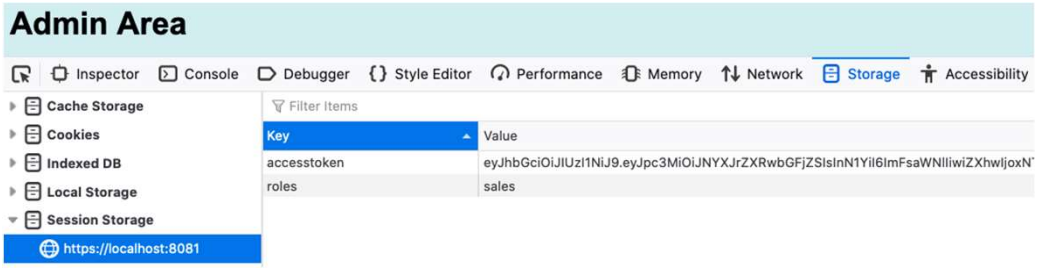
Getting the stored token, e.g., to include it in an authenticated request to the REST API

Removing the stored token and the roles when logging out

Storing the Auth Token in the Marketplace SPA (4)

- Session storage before authentication:

- Session storage after authentication:



XSS in the Marketplace SPA (1)

- In traditional web applications, XSS is typically addressed as follows:
  - Reflected / Stored Server XSS: When the HTML response is generated by the server, sanitize data before inserting it into the generated HTML page
  - DOM-based XSS: When JavaScript code in the browser processes DOM elements that can be influenced by the user (attacker), validate / sanitize the data first before processing it further
- In single page applications XSS works differently and has to be addressed differently:
  - Reflected / Stored Server XSS is no issue (assuming we have a «pure» SPA where no HTML pages are dynamically generated at the server-side)
  - DOM-based XSS is an issue – typically much more than in traditional web applications as a lot of data processing happens at the client-side only
  - Reflected / Stored Client XSS is an issue, because the REST API usually does not sanitize data returned to the client and simply includes the original data (and therefore possibly also executable JavaScript code)
    - It is the task of the client-side JavaScript code to make sure that JavaScript code received from the REST API won't be executed in the browser

XSS in the Marketplace SPA (2)

- Reflected Client XSS example in the Marketplace SPA:
  - Could, e.g., happen if a user enters a search string in a search filed, which is sent to the REST API, and the REST API returns also the search string which is then inserted into the web page

```
{"filter":"used",
 "results":[{"code":"0001","description":"DVD Life of Brian – used...","price":5.95},
            {"code":"0003","description":"Commodore C64 - used,...","price":444.95}]}
```

  - If the user inserts JavaScript code in the search field, it would be reflected in the response and executed in the browser if the code is directly inserted into the web page

```
{"filter":"<script>alert('XSS');</script>",
 "results":[]}
```

- The problem for the attacker is that in a SPA, such a reflected XSS vulnerability cannot easily be exploited
  - It's not possible to prepare a link for a victim, which triggers the attack
    - One could prepare a link to a web page, which contains JavaScript code, that sends the required request to the REST API – but we cannot get this response «into» the actual Marketplace SPA that is used by the user in the browser
  - Only option for the attacker: Convince the victim to copy the attack string into the search field himself... that's possible, but not very realistic...

XSS in the Marketplace SPA (3)

- Stored Client XSS example in the Marketplace SPA:
  - Assume a malicious product manager / database administrator inserts JavaScript code in a product description
    - `INSERT INTO Product VALUES ('6', '0006', 'XSS: <img src onerror= "alert(sessionStorage.getItem(\'accesstoken\'))">', '1.95', 'luke')`
  - When inserting this product into the shopping cart, the code is executed and accesses the stored JWT
  - Of course, in a real attack, the JavaScript code would send the JWT to the attacker

- Remark about the used JavaScript code
  - Angular strictly prevents that any content that uses *<script>* tags is inserted into the DOM (into the HTML page)
    - So, using `<script>alert(sessionStorage...)</script>` wouldn't work, as it would be replaced by Angular with the empty string when inserted into the DOM
    - But there are several other ways to still bring in JavaScript, e.g., by using the *<img>* tag

JavaScript Code in <img> tag

With this trick, JavaScript code can be executed without using *<script>* tags. Basically, the *src* attribute does not define the URL from which to get the image, which means the attribute is malformed. Therefore, when rendering this *<img>* tag in the browser, the *onerror* event is fired, which executes the JavaScript code defined in the *onerror* attribute.

XSS in the Marketplace SPA (4)

- DOM-based XSS example in the Marketplace SPA:
  - When entering `<img src onerror='alert("XSS")'>` in the search field...
  - ...this Javascript code is inserted into the HTML page at *Search results for: ___* ...
  - ...and as a result of this, the code is executed in the browser

- This looks very similar to the reflected XSS attack described before, but it's DOM-based, as it purely happens at the client-side
  - The entered string is copied from the input field (from the DOM) into a JavaScript variable and – after the search has been done – the value of this variable is inserted into the HTML page, so the code is executed
- Using # at the beginning prevents that the code is sent to the REST API
  - `dvd#<img src onerror='alert("XSS")'>`
  - Because # is used to identify a fragment on the page and fragments are never included in requests by the browser

DOM-based XSS

The JavaScript code entered in the search field is not reflected back by the REST API. Instead, when clicking the *Search* button, it's stored locally in a JavaScript variable and this variable is inserted into the HTML page (the DOM). Therefore, this is not a reflected XSS attack, but a DOM-based attack.

Using # at the beginning

As a result of this, the JavaScript code (including the leading #) is still stored in the JavaScript variable and inserted in the HTML document, but the search string won't be sent in the request to the REST API (because it ignores everything starting with the # character). Explanation: The # character is used to identify a fragment on the page and browsers don't include fragments in requests. When issuing requests using JavaScript code (as is done here in the Marketplace SPA), fragments are also not included in the request, so the web application never sees the JavaScript code!

So instead of this request,

https://localhost:8181/Marketplace_v11-rest/rest/products?filter=dvd**#**%3Cimg%20src%20onerror=%27alert(%22XSS%22)%27%3E

the following request is sent, which looks perfectly fine for the REST API:

https://localhost:8181/Marketplace_v11-rest/rest/products?filter=dvd

XSS in the Marketplace SPA (5)

- Just like with the reflected XSS attack example before, this cannot easily be exploited by sending the victim a link that triggers the attack
- But let's assume there's an additional feature in the Marketplace SPA:
  - *https://localhost:8081/search* is the route to get to the search page
  - In addition, the Marketplace SPA allows this to be called with an additional route element that can be used to specify a search string
    - *https://localhost:8081/search/dvd* directly searches for the string *dvd*
  - This is quite a «reasonable» feature as it allows to send customers e-mails with links that directly lead to products they may be interested in
- In this case, the attack can be exploited by sending a victim an e-mail with the following link:
  - *https://localhost:8081/search/dvd%23%3Cimg%20src%20onerror%3D'alert%28%22XSS%22%29'%3E*
  - With such a feature, the (hypothetical) reflected XSS attack described before could be exploited as well

Angular Routes

These routes are used by angular to navigate between «pages». They are used internally (i.e., within the browser by the Angular code), but can also directly be entered in the address bar.

URL Encoding

Some characters in the link must be URL-encoded to make sure that the additional route element is handled correctly by Angular so that the attack works. The decoded URL is as follows:

https://localhost:8081/search/dvd#<img src onerror='alert("XSS")'>

Stealing the Access Token?

Note that if session storage is used, then the attack above cannot steal the access token, because the browser will open a new tab when the user clicks the malicious link in an e-mail. As session storage is individual for each browser window / tab, getting the token is not possible. With local storage, it would be possible! But this does not mean that using session storage makes it impossible to use an XSS attack to steal the token, as this is, e.g., possible with the stored XSS attack described before.

Preventing XSS in the Marketplace SPA

- The good news is: If don't make «big mistakes» when developing an Angular SPA, you'll most likely not have problems with XSS
  - Reason: per default, Angular performs strict sanitation with all data that is inserted into the HTML page, no matter where the data comes from
- Example: Include the search string into the search page (assume the string is stored in a JavaScript variable *searchStringOut*)

```
Search results for: <span class="boldText">
{{ searchStringOut }}</span>
```

«Standard» approach used in Angular: simply include the value that is stored in variable *searchStringOut* → does strict sanitation, XSS not possible!

```
Search results for: <span class="boldText"
[innerHTML]="searchStringOut | safe: 'html'">
</span>
```

Insecure approach used for the previous demos, this disables sanitation by using an Angular pipe (see notes) → XSS possible (don't do this, unless you really know what you are doing!)

Important remark: This is how it works in Angular – if you use another framework, inform yourself how to prevent XSS there!

Angular Security

For details, refer to *https://angular.io/guide/security*

SafePipe

In addition to the code as illustrated above, the following Angular pipe has to be implemented. This is implemented in a flexible way and can not only disable sanitation for HTML code, but also for other components. This is boilerplate code that can be found in various online resources. By «piping» a value into this SafePipe, it basically it is transformed into a value where the normally used sanitation will *not* be applied. We can only repeat here that usually, you shouldn't do this, as it greatly increases the risk of XSS vulnerabilities.

```
import { Pipe, PipeTransform } from '@angular/core';
import { DomSanitizer, SafeHtml, SafeStyle, SafeScript, SafeUrl, SafeResourceUrl } from
'@angular/platform-browser';

@Pipe({
  name: 'safe'
})
export class SafePipe implements PipeTransform {

  constructor(protected sanitizer: DomSanitizer) {}

  public transform(value: any, type: string): SafeHtml | SafeStyle | SafeScript
                  | SafeUrl | SafeResourceUrl {
    switch (type) {
      case 'html': return this.sanitizer.bypassSecurityTrustHtml(value);
      case 'style': return this.sanitizer.bypassSecurityTrustStyle(value);
      case 'script': return this.sanitizer.bypassSecurityTrustScript(value);
      case 'url': return this.sanitizer.bypassSecurityTrustUrl(value);
      case 'resourceUrl': return
           this.sanitizer.bypassSecurityTrustResourceUrl(value);
      default: throw new Error(`Invalid safe type specified: ${type}`);
    }
  }
}
```

Client-Side Security in SPAs – Summary

With respect to client-side security, make sure the following security measures are taken into account:

- Configure CORS only as open as needed, in particular, use the *Access-Control-Allow-Origin* response header to only allow access from legitimate origins – this significantly reduces the potential for CSRF attacks
  - E.g.: *Access-Control-Allow-Origin: https://localhost:8081*
- Use the Authorization header to send the authentication token to the REST API and don't use cookies for this
  - This prevents authenticated requests during a CSRF attack
- In the browser, store the authentication token in session storage – this makes it vulnerable to XSS attacks, but there's no better option
- Prevent XSS vulnerabilities in the SPA, as this may allow an attacker to steal the authentication token and get control over the content displayed in the browser

Summary

- Modern web application and traditional web applications have a lot in common with respect to security measures that must be considered
- At the server-side, the REST API must be secured, which includes prevention from injection attacks, authentication, authorization, input validation, HTTPS, etc.
  - One main difference to traditional web applications is that authorizing access is usually not based on session IDs and cookies, but on authentication tokens (often JWT)
  - Jakarta EE provides many security features that are helpful to secure REST APIs
- With respect to client-side security, make sure to correctly configure CORS, store authentication tokens in session storage, use the Authorization header to send the tokens, and prevent XSS vulnerabilities
  - Angular is a well-suited technology to develop SPAs and provides good protection from XSS per default