SPRING BOOT: ADVANCED ORM RELATIONS & SECURITY

DAY 2

COURSE OVERVIEW

TODAY'S TOPICS

- ORM relations
  - OneToOne and ManyToOne
  - ManyToMany
  - Embedded and Embeddable
  - Inheritance strategies
  - Advanced mapping: MappedSuperclass and SecondaryTable(s)
- Security
  - Architecture
  - Github Oauth2
  - Keycloak

ONE-TO-ONE RELATIONSHIP

A record in a table is associated with one (and only one) record in another table.

@OneToOne: This annotation is used to create a one-to-one relationship between two entities.

@OneToOne can be bidirectional or unidirectional.

@JoinColumn: This annotation is usually used to specify the foreign key column.

EXERCISE

- Create a one-to-one relationship between two entities: Person and Passport.
- Set up your Spring Boot project with dependencies for Spring Web, Spring Data JPA, and your chosen database driver.
- Create the Person entity: Add fields for id (as the primary key) and name.
- Create the Passport entity: Add fields for id (primary key), passportNumber, and a reference to the Person entity. Annotate the reference to Person with @OneToOne to establish the relationship.
- Configure the relationship: In the Passport entity, use @JoinColumn to specify the foreign key column.
- Create repositories for both entities (PersonRepository and PassportRepository) extending JpaRepository.
- Implement a REST controller to handle CRUD operations for Person. Include endpoints to create a person and assign a passport to them.
- Test the relationship by creating a person and a passport, then retrieving the person's details along with their passport information.

EXAMPLE

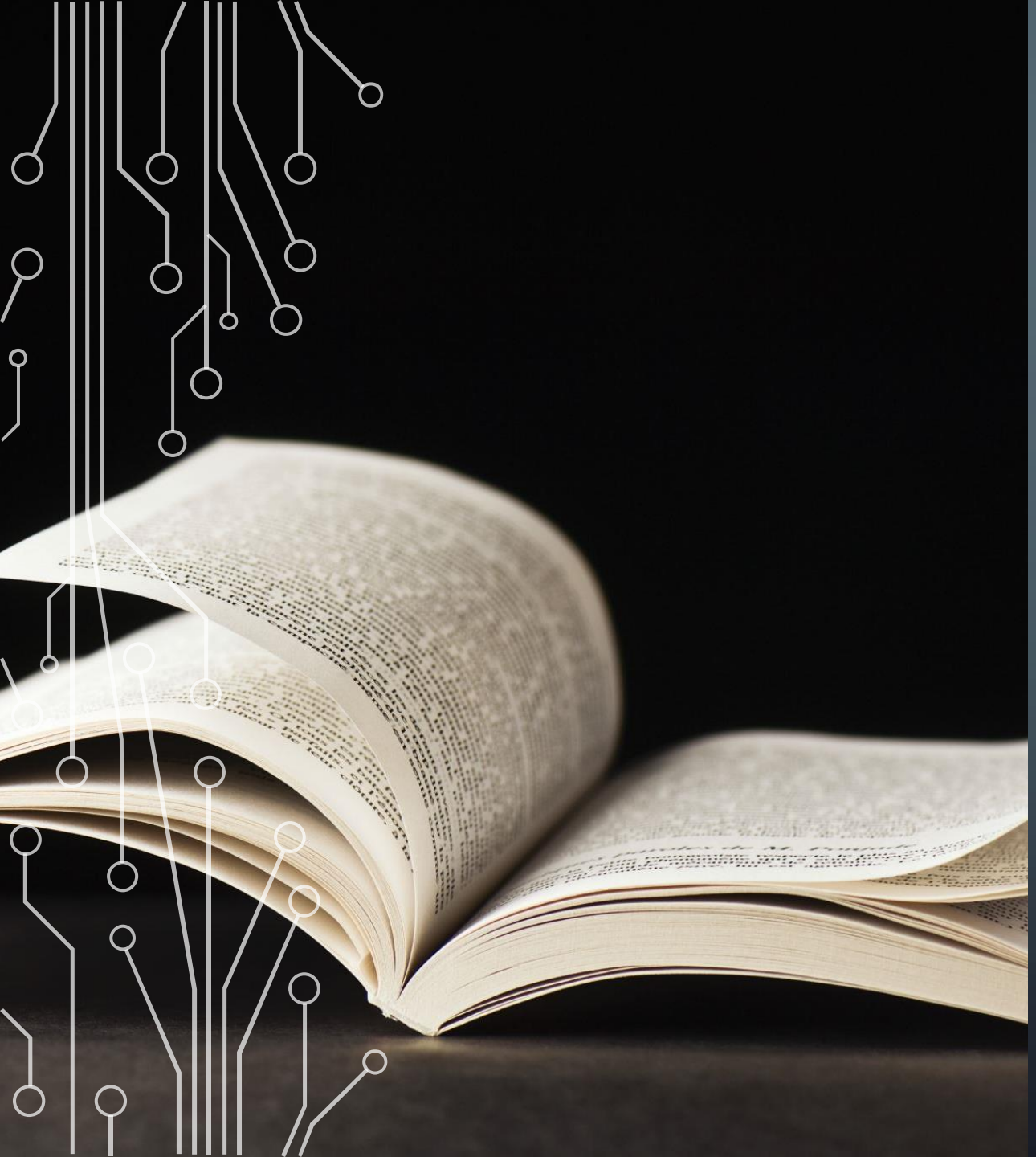
- A User entity and a UserProfile entity where each user has one user profile and each profile is associated with one user.

ONE-TO-MANY / MANY-TO-ONE RELATIONSHIP

When a record in one table can be associated with one or more records in another table.

@OneToMany: Used to establish a one-to-many relationship between two entities (with the reverse side being @ManyToOne).

@ManyToOne: Indicates that many entities on one side are related to one entity on the other side.



EXAMPLE

- An Author entity and a Book entity where one author can write many books, but each book has one author.

EXERCISE

- Implement a one-to-many relationship between Owner and Pet, where one owner can have many pets.
- Initialize a new Spring Boot project with necessary dependencies.
- Design the Owner entity:
  - Include an id (primary key) and a name.
  - Use an appropriate annotation to mark it as an Entity.
- Design the Pet entity:
  - Add fields for id (primary key), title, and a reference back to the Owner.
  - Establish the many-to-one relationship with the Owner entity using annotations.
- Map the relationship: On the Owner side, use @OneToMany to map the collection of pets. On the Pet side, use @ManyToOne to reference the owner.
- Create JPA repositories for both entities.
- Create a REST controller for Pet entities, allowing the creation of pets and assignment to an owner.
- Test your implementation by creating an owner, several pets, and associating those pets with the owner.

MANY-TO-MANY RELATIONSHIP

- When multiple records in a table are associated with multiple records in another table.
- **@ManyToMany**: This annotation is used to define a many-to-many relationship between two entities.
- **@JoinTable**: Often used in conjunction with @ManyToMany to define the join table (and join columns) that holds the foreign keys that reference the entities.

EXAMPLE

- A Student entity and a Course entity where a student can enroll in many courses and a course can include many students.

LAZY VS EAGER FETCHING

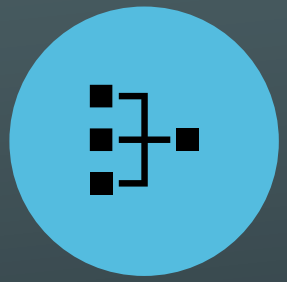
You can specify how the related entities are fetched in relation to the main entity.

This is controlled by the fetch attribute of relationship annotations (@OneToOne, @OneToMany, @ManyToOne, and @ManyToMany).

FetchType.EAGER: related entities are loaded immediately with the main entity, even if they are not needed.

FetchType.LAZY: related entities are not loaded when the main entity is fetched; instead, they are loaded on-demand when accessed for the first time.

CASCADE TYPES



Cascading operations are JPA operations that propagate from parent entities to child entities.

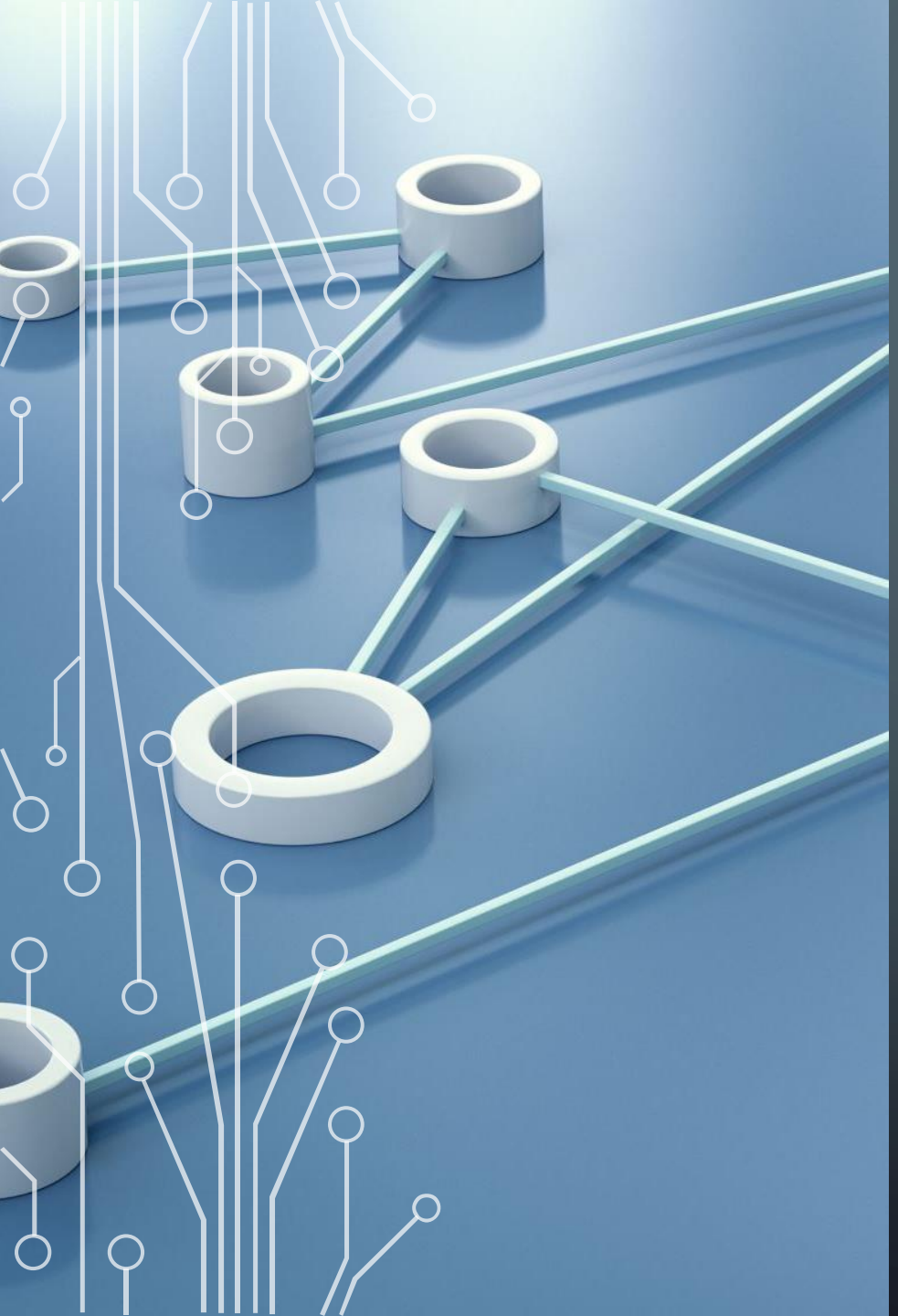
The cascade attribute of relationship annotations defines which operations (like persist, remove, merge, etc.) should be cascaded.

Without proper cascading, you might end up with an inconsistent database state, where you save a parent entity without saving the new or modified child entities.

On the other hand, overly aggressive cascading can cause unintended deletions or updates to propagate, potentially leading to data loss.

EXERCISE

- Create a many-to-many relationship between Artist and Album entities, demonstrating how artists can collaborate on multiple albums and albums can feature multiple artists.



EMBEDDED OBJECTS

- A part of an entity is reusable across different entities.
- For example, an Address entity that can be used by both User and Company entities as an embedded object.

@EMBEDDABLE

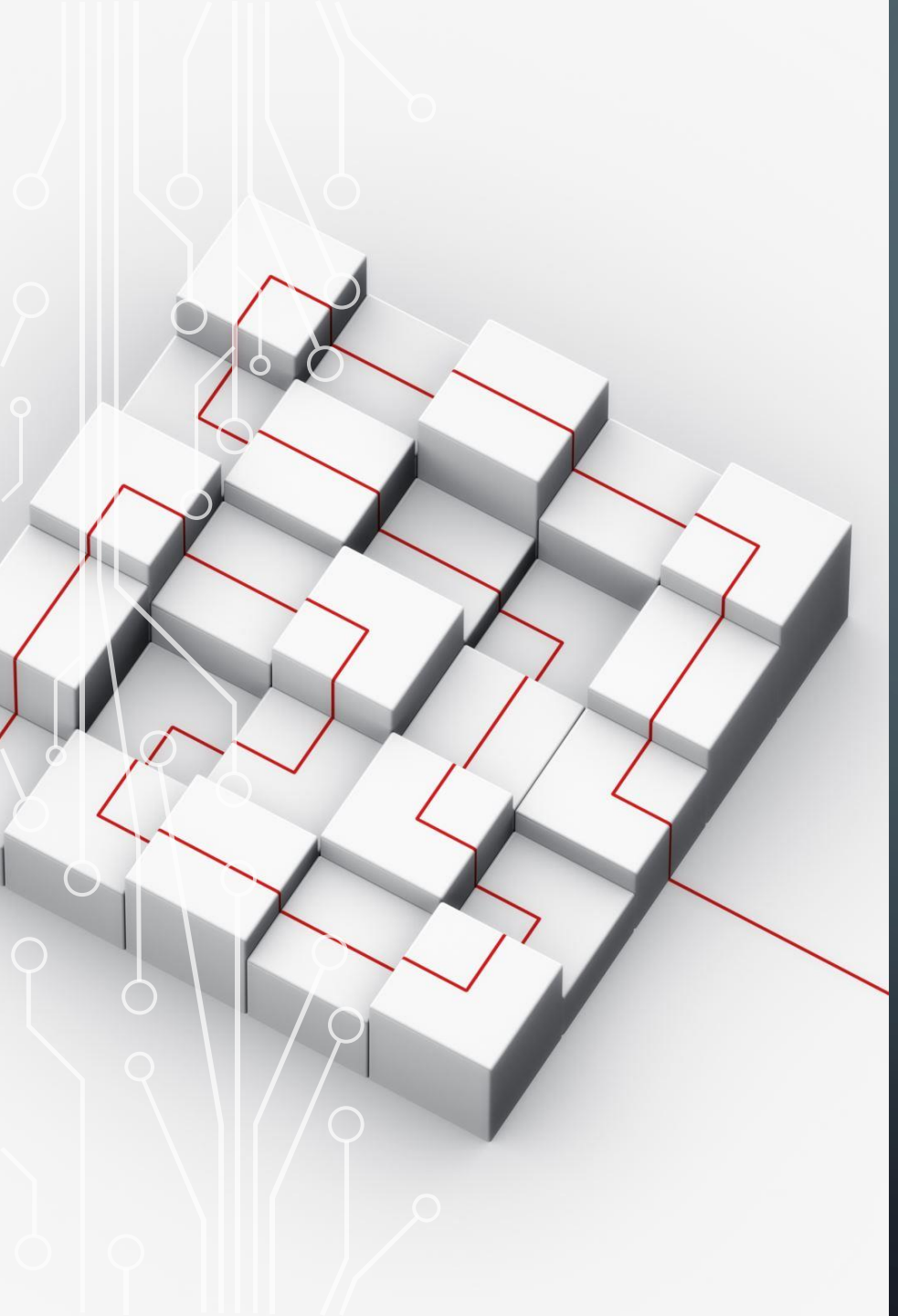
- This is a way to embed a "component" inside an entity.
- @Embeddable is used to mark a class as embeddable.
- This class is not an entity itself and does not have its own database table.
- Instances of an @Embeddable class are stored as part of an entity that owns it, within the owning entity's table.

@EMBEDDED

- @Embedded is used within an entity to include an instance of an @Embeddable class as if it were a part of the entity itself.
- The fields of the embeddable class are mapped to columns in the entity’s table.
- You use @Embedded when you want to include an instance of an @Embeddable class in an entity and when the fields of the embedded instance should appear as columns in the entity's database table.

EMBEDDABLE VS EMBEDDED

- @Embeddable is used to declare a class that can be embedded, while @Embedded is used to embed that class into an entity.
- An @Embeddable class is not an entity and does not have its own lifecycle, whereas an @Embedded object shares the lifecycle of the owning entity.
- The @Embeddable class does not require an identifier since it depends on the owning entity for identity.



@ATTRIBUTEOVERRIDE(S)

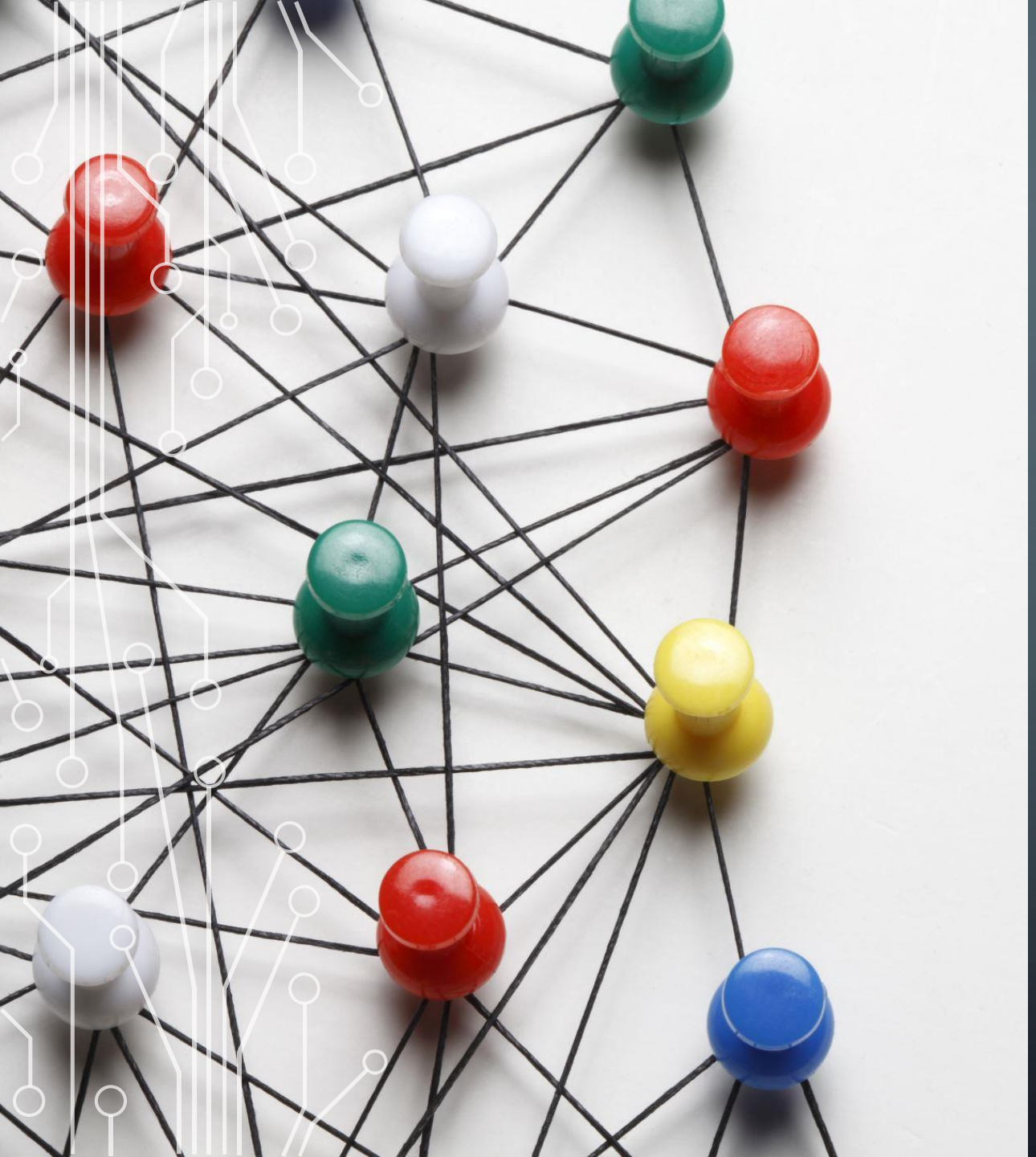
- When you embed an embeddable class into an entity, it might be necessary to change the column mappings of the embedded fields.
- (e.g. naming conflicts with other fields in the entity or if you want to customize the column definitions for a particular use of an embeddable)
- This can be done with @AttributeOverride or @AttributeOverrides

@ATTRIBUTEOVERRIDE AND @ATTRIBUTEOVERRIDES

- @AttributeOverride: override the column mappings of a single field of an embedded object.
- Used when the entity's table column names need to differ from the column names defined in the @Embeddable class or when you're embedding the same @Embeddable class more than once in the same entity and you need to distinguish between the columns of each embedded instance.
- @AttributeOverrides: group multiple @AttributeOverride annotations when you need to override multiple fields of an embedded object
- Used when you have multiple column mappings to override within a single embedded instance or when you're embedding multiple instances of the same @Embeddable class in an entity and overriding the mappings for each instance.

EXERCISE

- Let's embed a Dimensions object in a Product entity to manage product dimensions as part of the product details.
- Create an @Embeddable class Dimensions with properties like height, width, depth, and weight.
- Design the Product entity with id, name, and an instance of Dimensions embedded.
- Use @Embedded on the Dimensions field in the Product entity to indicate it's an embedded object.
- Generate a repository for the Product entity.
- Create a REST controller for managing Product entities, ensuring you can create and retrieve products along with their dimensions.
- Perform tests by creating products with dimensions and fetching them to see the embedded dimensions in action.



INHERITANCE STRATEGIES

- Different options for defining how Java class hierarchies map to database table structures.
- @Inheritance is used at the class level to define which inheritance strategy to use for a class hierarchy.

STRATEGY: SINGLE TABLE

- Maps all entities of the inheritance structure to a single table.
- This table contains a column for every field defined in any class in the hierarchy, with extra discriminator columns to determine which class the data belongs to.
- @DiscriminatorColumn: This is used to specify the name and type of the column used to distinguish between entity types. If not specified, the name defaults to DTYPE.
- @DiscriminatorValue: Used at the class level to specify the value that indicates the type of the entity when stored in the table.

EXERCISE

- Use the SINGLE_TABLE inheritance strategy to manage different types of payments in a single table.
- Create base class Payment with common properties like id, amount, and a discriminator column to differentiate between payment types.
- Extend Payment with specific payment method classes like CreditCardPayment, PaypalPayment, and BankTransferPayment, adding method-specific properties to each subclass.
- Annotate the base class with @Inheritance(strategy = InheritanceType.SINGLE_TABLE) and use @DiscriminatorColumn to define the discriminator column.
- Setup repositories and REST controllers for handling and testing the various payment types.
- Test creating and retrieving different payment types, observing how they are stored and fetched from a single table.

STRATEGY: TABLE PER CLASS

Each concrete class in the hierarchy is mapped to its own table. All fields, including inherited ones, are mapped to the table of the class.

Great when classes have distinct fields, and sharing a table would lead to many nullable columns.

Only use when polymorphic operations are not frequent since this strategy can lead to complex and less performant queries, as it might require UNION operations to gather data from multiple tables.

STRATEGY: JOINED

- The base class in the hierarchy has its own table, which includes the primary key and persistent attributes. Each subclass has a separate table that includes only the fields specific to it, along with a foreign key that joins to the primary key of the base class table.
- All the tables are related through foreign key relationships. To construct a complete subclass entity, a join operation is performed between the subclass table and the base class table.
- Great when database normalization is a priority.
- Also great when subclasses add many fields that are not present in the superclass, thus avoiding null values in the table.

@MAPPEDSUPERCLASS

Indicates that it's a superclass, and its properties should be included in the table mapping of any entity that inherits from it.

The class annotated with @MappedSuperclass is not an entity itself and won't be mapped to a database table.

Its properties are either mapped to the tables of the entities that extend this class or used in a SINGLE_TABLE strategy where a single table is used to map the entire entity hierarchy.

@MappedSuperclass strategy is used when you have common properties or fields that you want to share across multiple entities.

@SECONDARYTABLE(S)

Sometimes, a single entity has data spanning multiple tables.

To map such an entity to more than one table, you can use the @SecondaryTable and @SecondaryTables annotations.

@SecondaryTable on an entity class specifies a secondary table for the additional fields.

Used when you want to decompose a single entity across two tables, usually to segregate fields to a related table which might be updated or accessed less frequently.

When an entity spans more than two tables, you can use the @SecondaryTables annotation to specify multiple secondary tables.

EXERCISE

- Model a company's organizational structure using the JOINED strategy to store common fields in an Employee table and specific fields in subclass tables.
- Define a base Employee class with shared attributes like id, name, and department.
- Create subclasses Manager, Developer, and SalesRepresentative, each with role-specific attributes.
- Annotate the Employee class with @Inheritance(strategy = InheritanceType.JOINED) to indicate the inheritance strategy.
- Implement repositories and REST controllers for creating and managing employees of different types.
- Conduct tests to create and retrieve various employee types, noting the separate tables for each subclass and the joined queries used to fetch them.

SECURITY

AUTHENTICATION VS. AUTHORIZATION

- Authentication is the process of verifying who a user is.
- Authorization is the process of verifying what they have access to.
- Spring Security handles both aspects and can be highly customized to meet various security requirements.

GETTING STARTED WITH SECURITY

- Using Spring Security with Spring Boot works out of the box
- You add the dependency:
- ```
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
  </dependency>
  ```
- And get default behavior for Spring Security
  ```

DEFAULT CONFIGURATION

- When you run the Spring Boot app, you'll see something new in the log output

```
Using generated security password: af42f46e-327b-4e66-b63b-72467f47d0d5

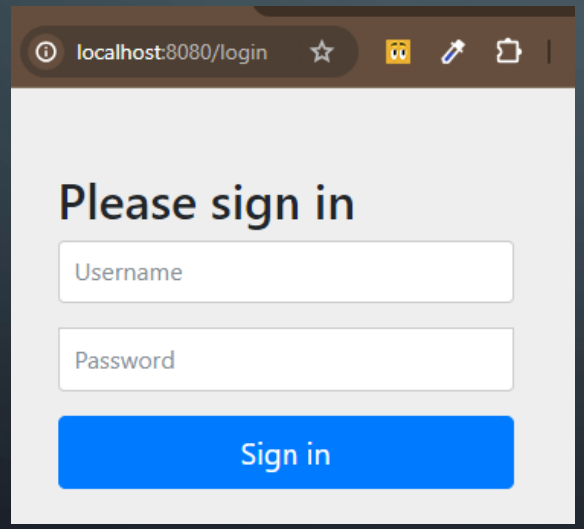
This generated password is for development use only. Your security
 configuration must be updated before running your application in production.
```

- Trying to access any endpoint will now give: 401 Unauthorized

DEFAULT LOGIN

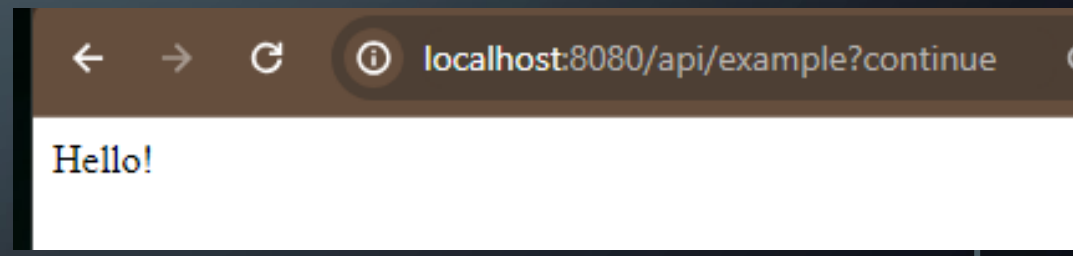
- Trying to access the get endpoints in the browser now redirects:
  Going to: localhost:8080/api/example

Brings us to:



DEFAULT LOGIN

- We can login with the generated password in the output and default username “user”



MORE DEFAULT BEHAVIOR

- Protects password storage with BCrypt as well as others
- Provides form-based login and logout flows
- Authenticates form-based login as well as HTTP Basic
- Provides content negotiation; for web requests, redirects to the login page; for service requests, returns a 401 Unauthorized
- Mitigates CSRF attacks
- Mitigates Session Fixation attacks
- Writes Strict-Transport-Security to ensure HTTPS
- Writes X-Content-Type-Options to mitigate sniffing attacks
- Writes Cache Control headers that protect authenticated resources
- Writes X-Frame-Options to mitigate Clickjacking
- Integrates with HttpServletRequest's authentication methods
- Publishes authentication success and failure events

https://docs.spring.io/spring-security/reference/servlet/getting-started.html#servlet-hello-auto-configuration

JAVA CONFIGURATION

- Spring Security can be configured with Java configuration
- We'll have to create a config class for Spring Security
- This creates a servlet filter (springSecurityFilterChain) in the application

JAVA CONFIGURATION EXAMPLE

- Necessary authentication for all URLs
- Create a login form
- Allow the user with to authenticate via the form with user / password
- Allow the user to logout
- Prevent CSRF attacks
- Security headers are added
- Default password is no longer generated, because there is one specified now

```
@Configuration
@EnableWebSecurity
public class SimpleWebSecurityConfig {

    no usages
    @Bean
    public UserDetailsService userDetailsService() {
        InMemoryUserDetailsManager manager = new
         InMemoryUserDetailsManager();
        manager.createUser(User.withDefaultPasswordEncoder().username
         ("user").password("password").roles("USER").build());
        return manager;
    }
}
```

JAVA CONFIGURATION - HTTPSECURITY

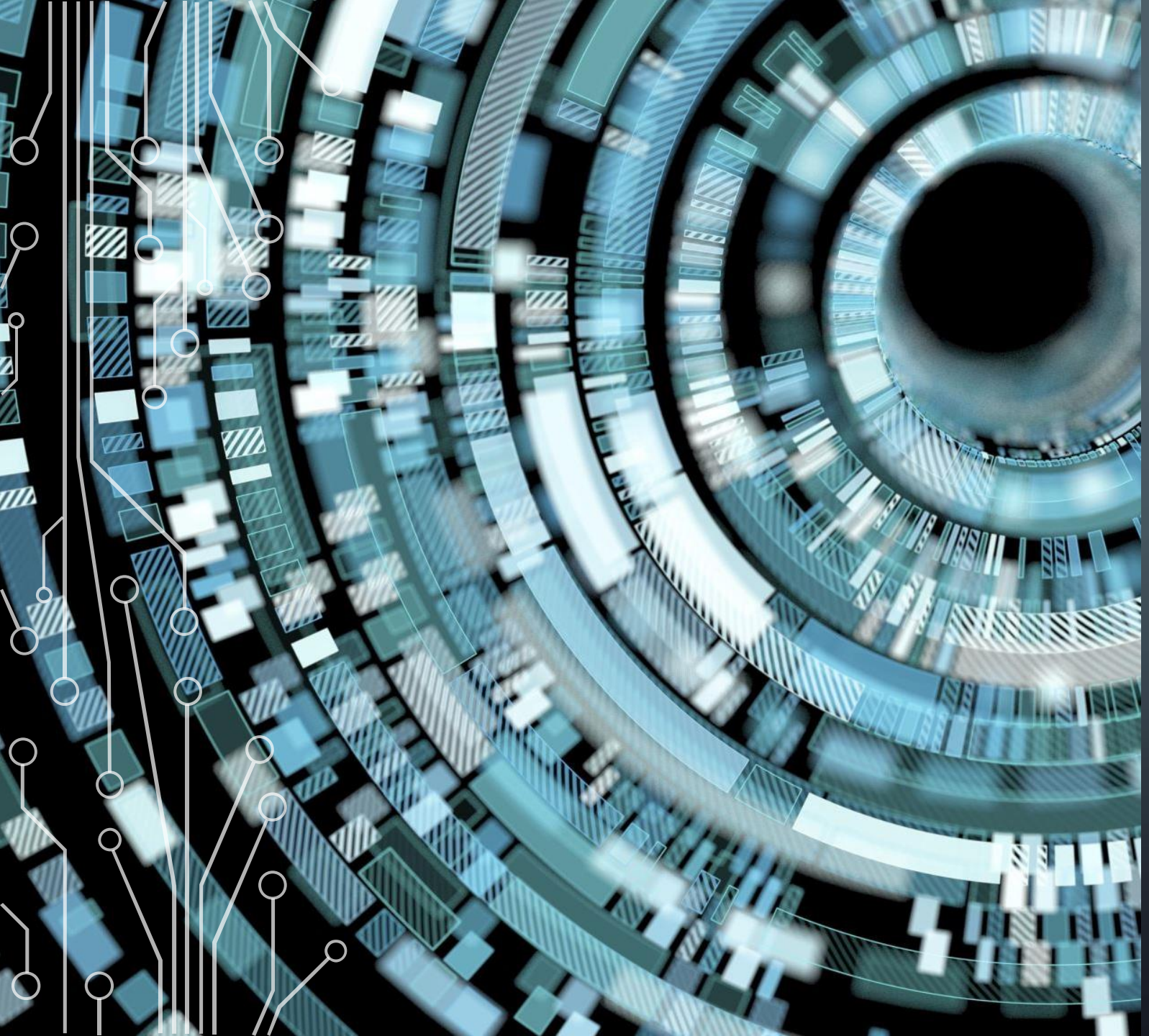
- We now only have how to authenticate.
- Default, it's specified that all users have to be authenticated for all pages (default looks like on the right here)
- This can be altered (see next example)

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http)
 throws Exception {
    http
            .authorizeHttpRequests(authorize -> authorize
                    .anyRequest().authenticated()
            )
            .formLogin(withDefaults())
            .httpBasic(withDefaults());
    return http.build();
}
```

CUSTOM HTTPSECURITY

- Quite some updates compared to Spring Framework 5 in this area
- This SecurityFilterChain has order 1 and will be considered before the previous one
- If the endpoint starts with /api these rules apply
- If it doesn't the previous one is used (because it's still in the class, not because that's default)

```
@Bean
@Order(1)
public SecurityFilterChain apiFilterChain(HttpSecurity http)
 throws Exception {
    http
            .securityMatcher( ...patterns: "/api/**")
            .authorizeHttpRequests(authorize -> authorize
                    .anyRequest().hasRole("ADMIN")
            )
            .httpBasic(withDefaults());
    return http.build();
}
```



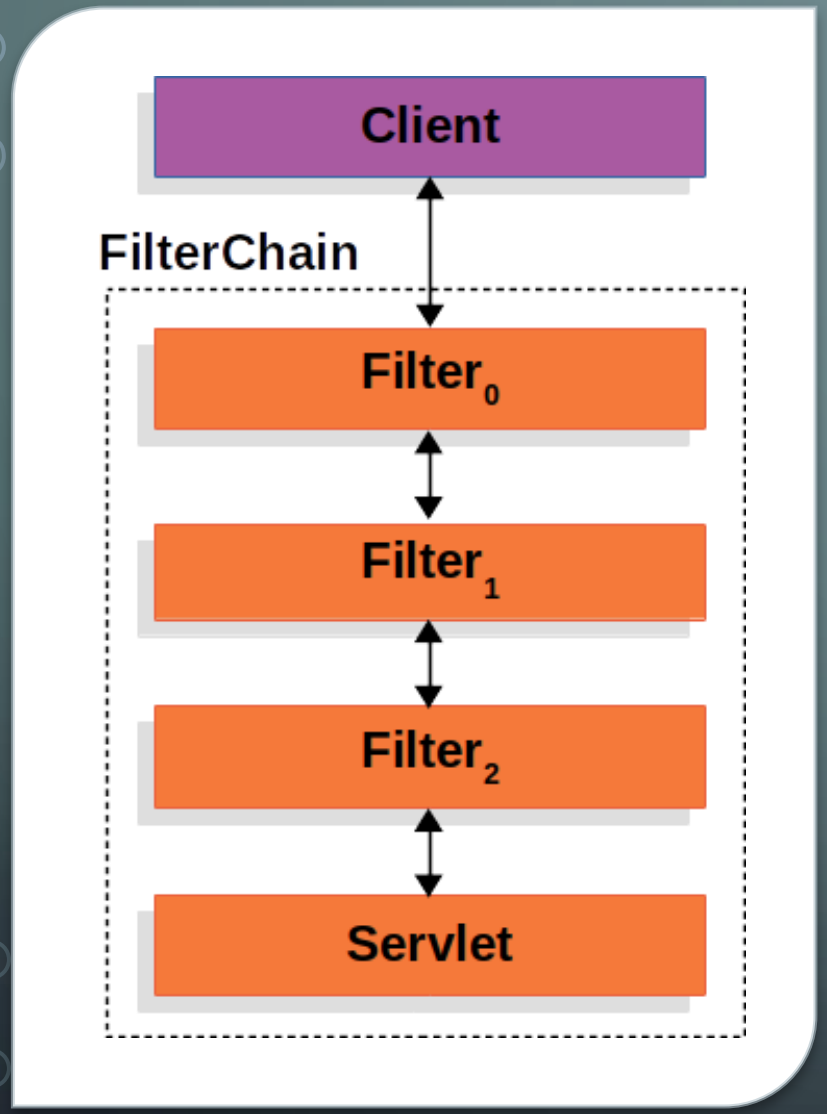
DEMO BASIC AUTHENTICATION AND AUTHORIZATION

EXERCISE

- Create controller that has the following endpoints:
  - Get: api/greet/{name} >> everybody, also not logged in users
  - Get: api/greetUser >> greets the user with the use of the username, only for logged in users
  - Get: api/users >> shows a list of all the users, only for logged in admins

SECURITY ARCHITECTURE

- Spring Security's Servlet leans on the use of Servlet Filters
- (A Servlet is a class that deals with requests and replies with a response)
- A client sends a request to the application which is then filtered through the FilterChain
- At the end of the FilterChain is the Servlet that will process the HttpServletRequest

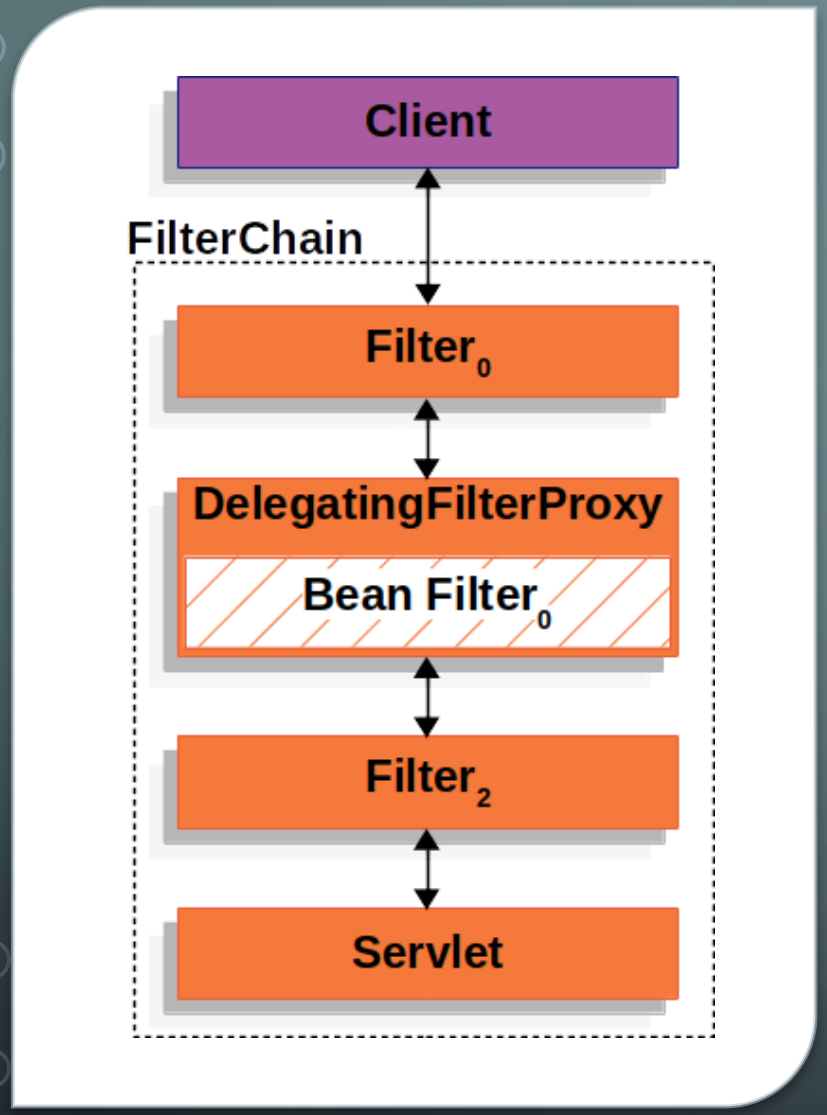


ARCHITECTURE - FILTERCHAIN

- FilterChain contains instances of the filter and the servlet
- When a request doesn't make it past a filter, the Filter and Servlet downstream will not be invoked and the Filter sends back the HttpServletResponse
- https://docs.spring.io/spring-security/reference/servlet/architecture.html

DELEGATINGFILTERPROXY

- Special Filter implementation provided by Spring
- The Servlet cannot see any of the Spring Beans
- DelegatingFilterProxy connects the Servlet container's lifecycle to Spring's ApplicationContext
- DelegatingFilterProxy can be registered via the Servlet container mechanisms, but all the work is delegated to the Spring Bean that implements Filter. Hence the name: DelegatingFilterProxy



DELEGATINGFILTERPROXY

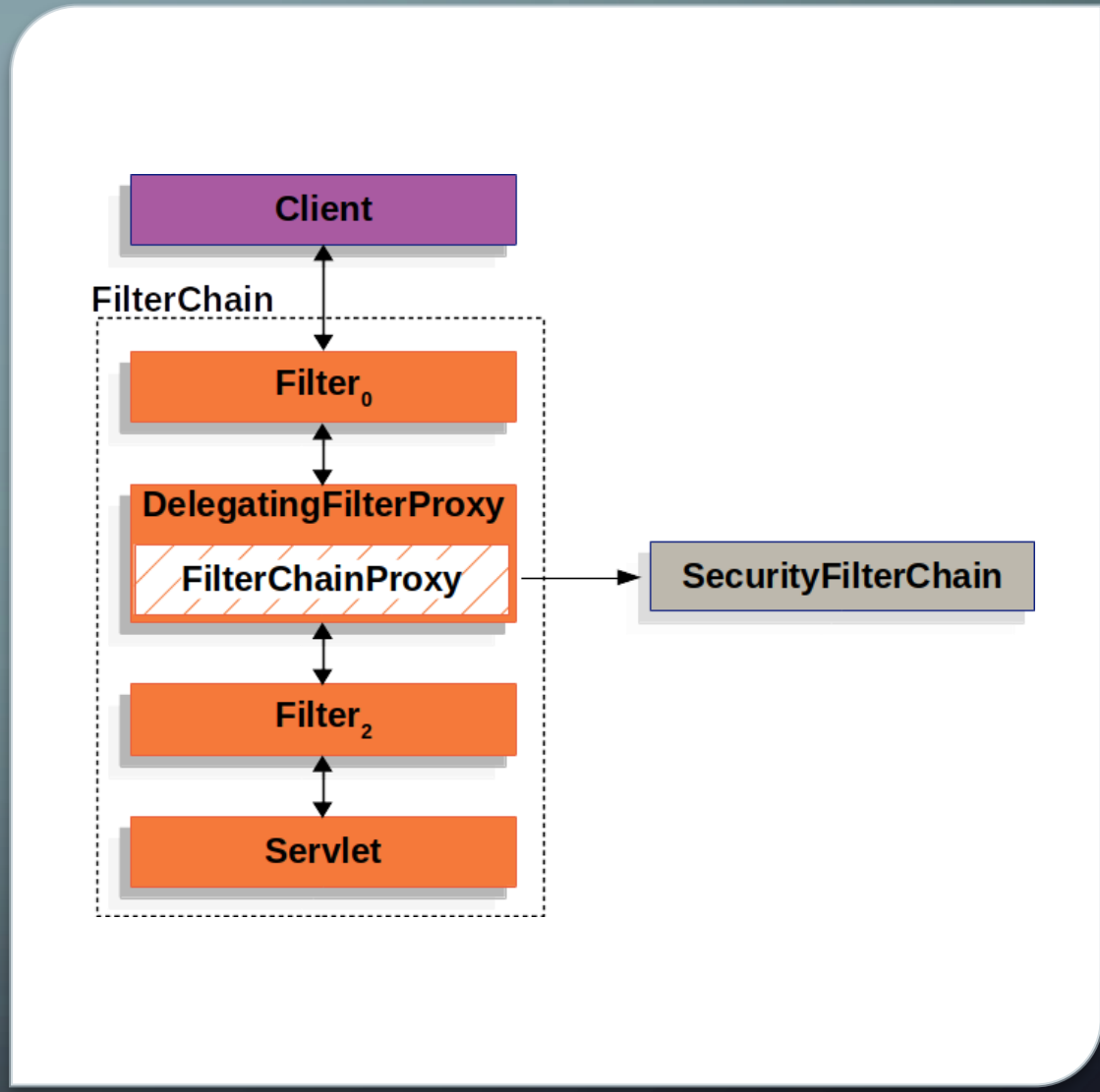
- The DelegatingFilterProxy uses the application context to get the Bean Filter
- The Bean Filter does the filtering
- https://docs.spring.io/spring-security/reference/servlet/architecture.html

FILTERCHAINPROXY

- DelegatingFilterProxy can also delegate to FilterChainProxy
- FilterChainProxy is a special Filter that allows delegation to multiple Filter instances through SecurityFilterChain
- FilterChainProxy is a bean as well

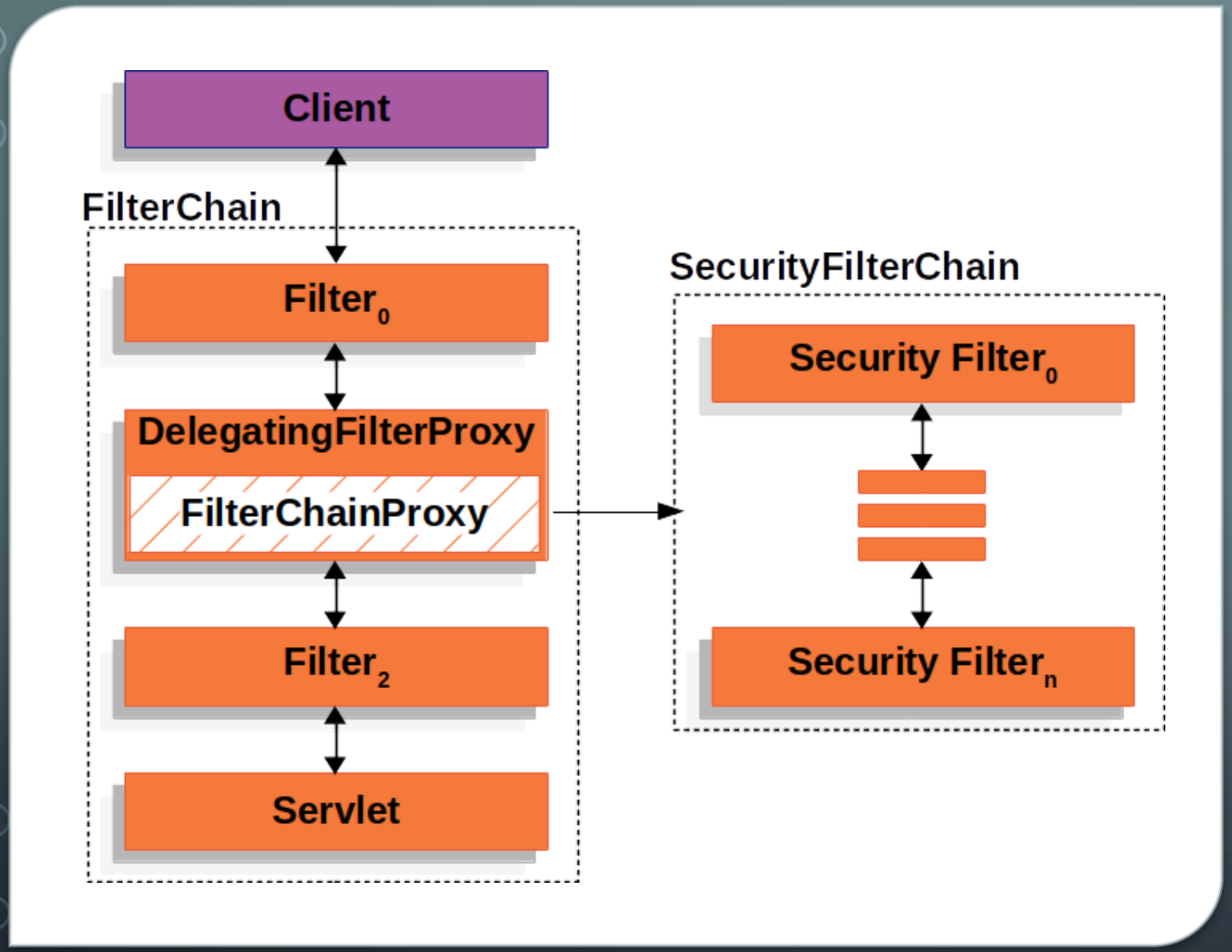
FILTERCHAINPROXY

HTTPS://DOCS.SPRING.IO/SPRING-SECURITY/REFERENCE/SERVLET/ARCHITECTURE.HTML

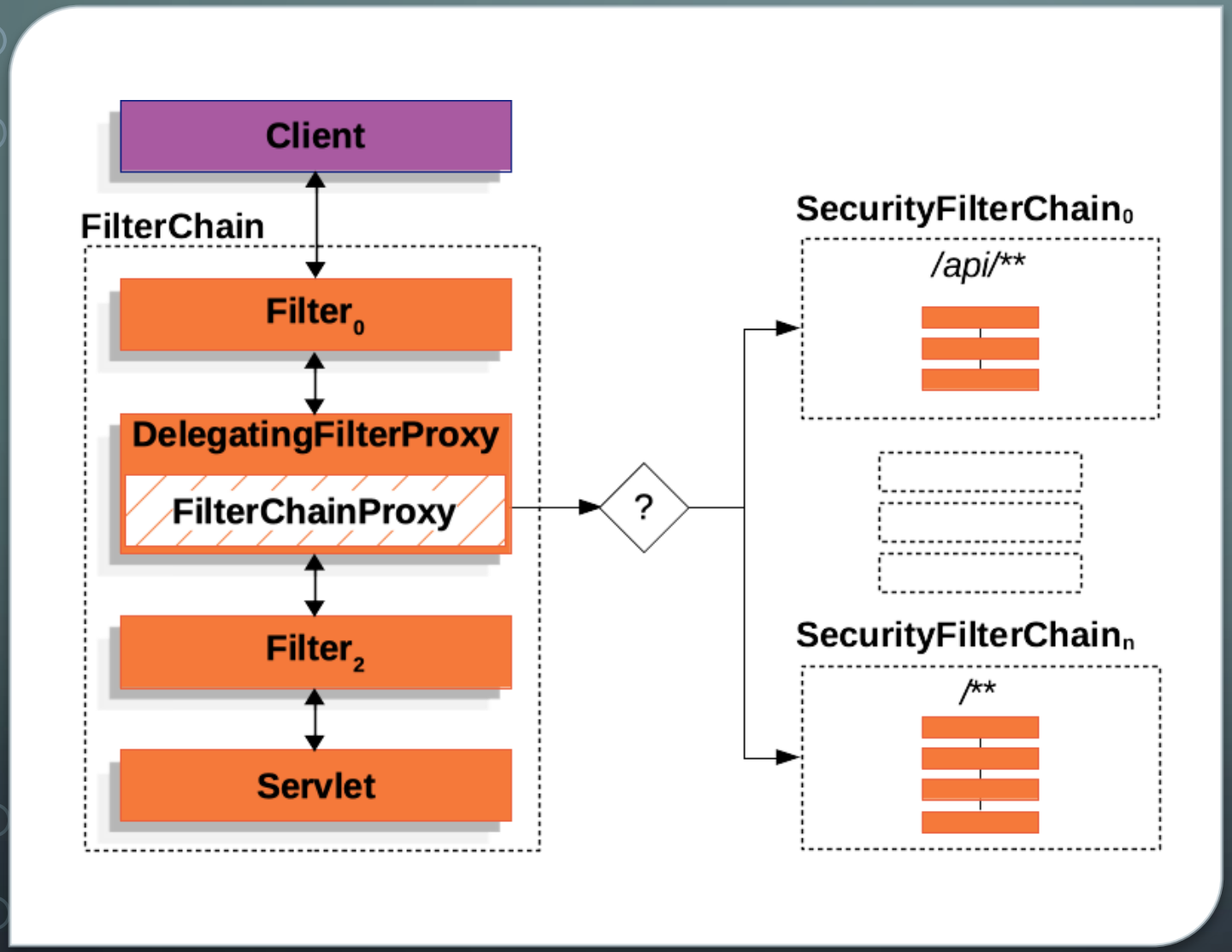


SECURITYFILTERCHAIN

- SecurityFilterChain is used by FilterChainProxy to determine which Spring Security Filter instances need to be called for the request
- SecurityFilterChain holds Security Filters (beans), these are registered with FilterChainProxy instead of DelegatingFilterProxy, because:
  - Spring Security's Servlet support bundled in FilterChainProxy
  - FilterChainProxy is used for Spring Security and operates in a safe way (e.g. clearing SecurityContext to prevent memory leaks)
  - FilterChainProxy can determine whether Filter should be invoked based on properties of the HttpServletRequest (rather than just the URL)



HTTPS://DOCS.SPRING.IO/SPRING-SECURITY/REFERENCE/SERVLET/ARCHITECTURE.HTML



- https://docs.spring.io/spring-security/reference/servlet/architecture.html

SECURITY FILTERS

- Security Filters are added to the FilterChainProxy as part of the SecurityFilterChain
- Filters can be used for authentication, authorization and exploit protection
- Filters are executed in a specific order, every filter gets a FilterOrderRegistration code

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http)
 throws Exception {
    http
            .authorizeHttpRequests(authorize -> authorize
                    .anyRequest().authenticated()
            )
            .formLogin(withDefaults())
            .httpBasic(withDefaults());
    return http.build();
}
```

```java
@Bean
public SecurityFilterChain filterChain(HttpSecurity http)
 throws Exception {
    http
            .authorizeHttpRequests(authorize -> authorize
                    .anyRequest().authenticated()
            )
            .formLogin(withDefaults())
            .httpBasic(withDefaults());
    return http.build();
}
```

Filter	Added by
UsernamePasswordAuthenticationFilter	
AuthorizationFilter	
basicAuthenticationFilter	

WHEN A FILTER THROWS AN EXCEPTION

- In the FilterChainProxy, one of the security filters is the ExceptionTranslationFilter
- When a Filter throws an AccessDeniedException or AuthenticationException, the ExceptionTranslationFilter makes the following happens:
  - When the user isn't authenticated or the Exception is of type AuthenticationException, the authentication is started: SecurityContextHolder cleared, HttpServletRequest is saved (in the RequestCache) so that it can go back there after authentication, AuthenticationEntryPoint is activated (e.g. login page redirect)
  - If it's another exception, the AccessDeniedHandler is invoked
- Other exceptions don't trigger the ExceptionTranslationFilter

CUSTOM FILTER

- Custom filters can be added to the chain
- Create a class (e.g. SpecialFilter) that implements the Filter interface
- Override the doFilter method, this should just complete successfully if the Filter succeeds, throws an AccessDeniedException or AuthenticationException if the Filter should fail

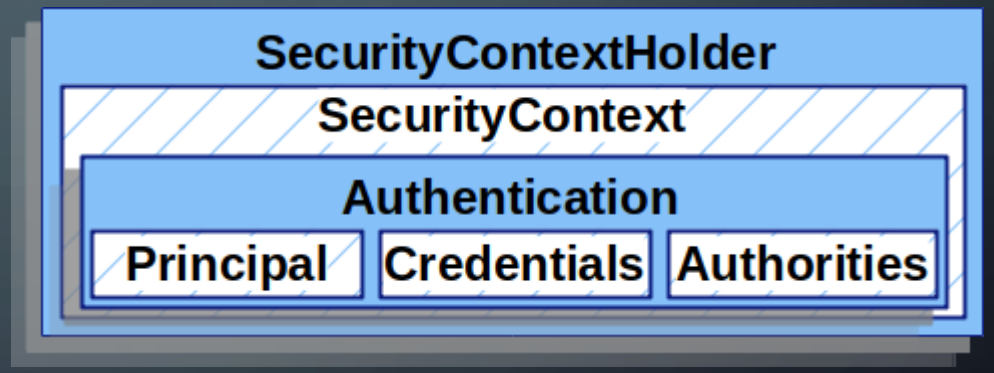
- Could be added to the chain like this:
  .addFilterBefore(new SpecialFilter(), AuthorizationFilter.class); //second argument is the filter it should go before

SERVLET AUTHENTICATION ARCHITECTURE

- https://docs.spring.io/spring-security/reference/servlet/authentication/architecture.html

SECURITYCONTEXTHOLDER

- Contains the SecurityContext, containing the details of who is authenticated
- SecurityContext can be fetched with the static getContext() method
- The Authentication object can be fetched from the securityContext with the getAuthentication() method



GRANTEDAUTHORITY

- High level permissions for the user
- Can be obtained from the Authentication object with the getAuthorities() method
- This returns a collection of GrantedAuthority objects
- E.g. ROLE_USER, ROLE_ADMIN
- Used for general permissions (due to memory and performance limitations)

AUTHENTICATION IN SPRING SECURITY

- Authentication in Spring Security is defined by the AuthenticationManager interface
- It specifies how Spring Security's filters perform the authentication
- It returns the Authentication object which is set on the SecurityContextHolder by the filter instance that invoked the AuthenticationManager
- The most commonly used implementation is ProviderManager, which delegates to a chain of AuthenticationProvider instances.

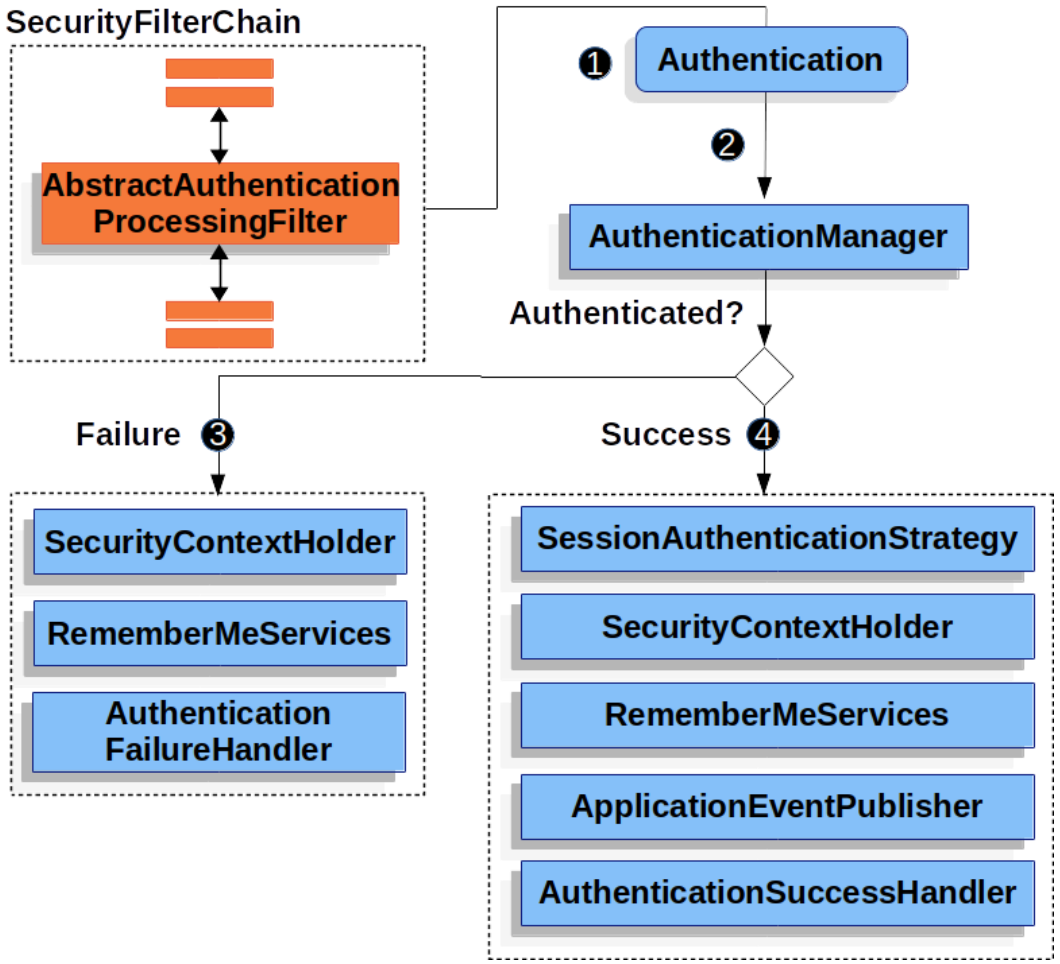
AUTHENTICATIONPROVIDER

- Multiple AuthenticationProviders can be injected in the ProviderManager
- These different instances can perform different types of authentication, for example:
  - DaoAuthenticationProvider >> username/password
  - JwtAuthenticationProvider >> JWT token
- Each AuthenticationProvider tries to authenticate, if none of them manage to authenticate, it fails with a ProviderNotFoundException

AUTHENTICATIONENTRYPOINT

- AuthenticatoinEntryPoint sends the HTTP response requesting client credentials (for example by redirecting to a login page)
- When the credentials are included in the request, this step is skipped
- The credentials that are sent as a result are verified by the AbstractAuthenticationProcessingFilter

ABSTRACTAUTHENTICATION-PROCESSINGFILTER

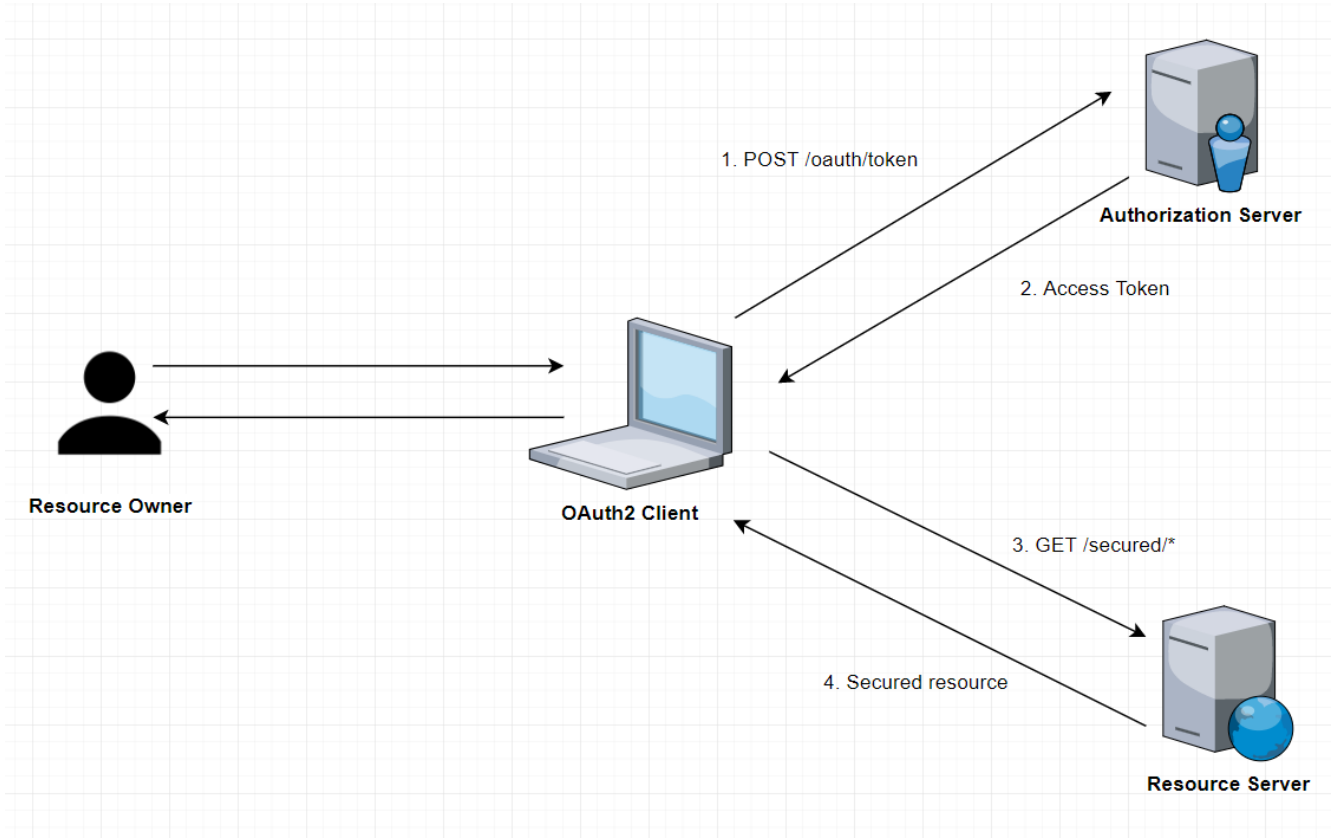


AUTHORIZATION IN SPRING SECURITY

- Authorization is managed using access control lists (ACLs), role-based access control (RBAC), or more complex attribute-based access control (ABAC).
- In Spring Security, this is typically handled using one of the below:
  - @PreAuthorize on your controller methods
  - @Secured on your controller methods
  - By configuring URL-based security in your security configuration

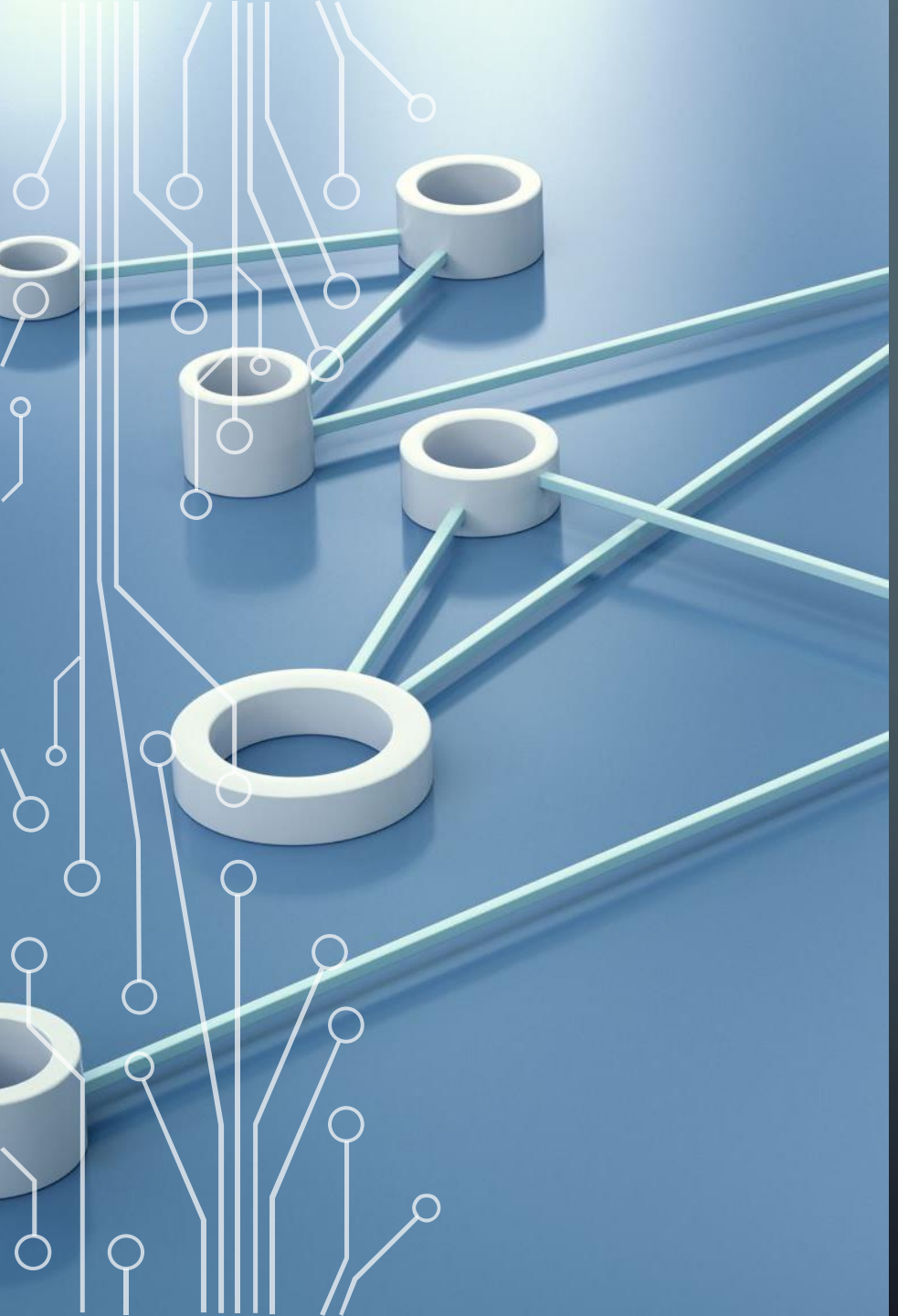
OAUTH 2.0

- OAuth 2.0 is an authorization framework that enables applications to obtain limited access to user accounts on an HTTP service.
- It works by delegating user authentication to the service that hosts the user account and authorizing third-party applications to access the user account.
- It provides several "grant types" for different use cases, such as authorization code, implicit, resource owner password credentials, and client credentials.
- In Spring Security, OAuth 2.0 can be configured using the spring-security-oauth2-client and spring-security-oauth2-resource-server dependencies.



HTTPS://DZONE.COM/ARTICLES/SECURE-SPRING-REST-WITH-SPRING-SECURITY-AND-OAUTH2

- Four relevant roles:
  - Resource owner: the user that uses the application
  - Client: the application that the user is in that needs access to data/services on the resource server
  - Resource server: holds data and services which returns this to authenticated clients
  - Authorization server: authenticates user's identity and gives out the auth token (this is used by the resource server to validate user)



OPENID CONNECT

- OpenID Connect is an identity layer on top of the OAuth 2.0 protocol. It allows clients to verify the identity of the end-user based on the authentication performed by an authorization server, as well as to obtain basic profile information about the end-user.
- It introduces the concept of an id_token, which is a JWT that contains information about the user's authentication event.
- In Spring Security, OpenID Connect is supported out of the box as part of the OAuth 2.0 Login feature.
- You would configure an OAuth 2.0 client to use the authorization_code grant type along with the necessary configurations to validate the id_token.

OAUTH2

- Spring Security has 2 main parts of OAuth 2.0 support in the Oauth 2.0 authorization framwork:
  - Resource server
  - Client (including OAuth2 login)
- Separate project built on Spring security:
  - Spring authorization server (can also be third party)

OAUTH2 RESOURCE SERVER

- Contains services whose access are protected using Oauth2 access token, two types of Bearer tokens are supported by Spring Security:
  - JWT (JwtDecoder bean for validation and decoding)
  - Opaque token (OpaqueTokenIntrospector bean)
- spring-security-oauth2-resource-server dependency needed for creating a resource server

CUSTOM JWT

- Frontends often connect with the API using JWTs
- OAuth2 resource server supports custom JWT with the use of a JwtDecoder bean
- Spring Security will pick up this bean for protection in the SecurityFilterChain

OAUTH2 CLIENT

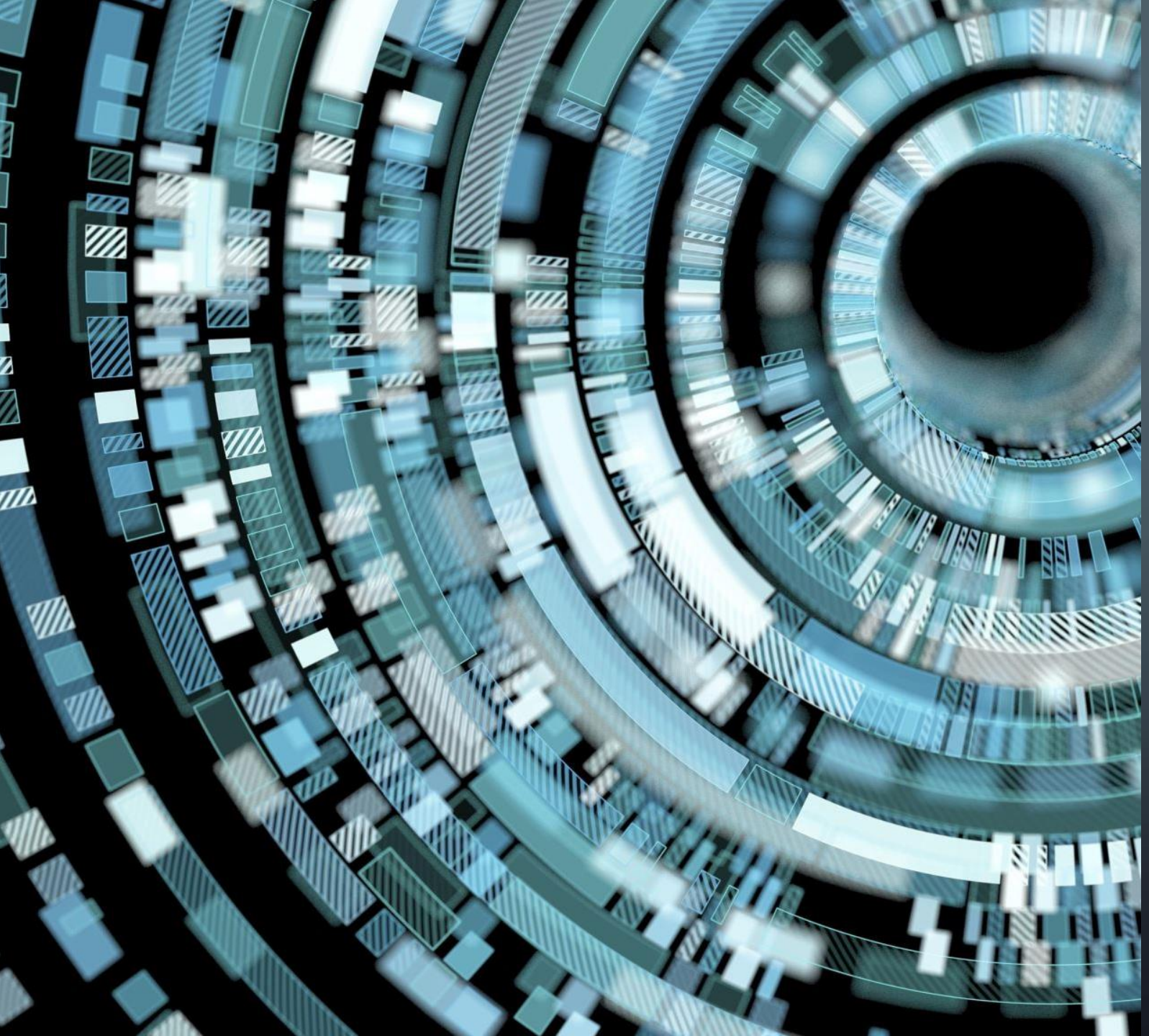
- OAuth2 client will enable uses to verify their identity and be logged in
- Start by adding spring-security-oauth2-client to your project

- We'll discuss two use cases:
  - Logging in using OAuth 2.0 / OpenID Connect 1.0
  - Obtaining a JWT access token

- You can have apps acting as both client and resource at the same time

LOGGING IN USING OAUTH 2.0 / OPENID CONNECT 1.0

- OpenID provides the id_token, this is an OAuth2 Client that can verify identity and log users in
- OAuth2 can be used directly to log users in (e.g. GitHub, Google, Facebook)
- The SecurityFilterChain will have to specify .oauth2Login in order for the application to act as an OAuth2 Client that can log users in via OAuth2 or OpenID Connect



DEMO OAUTH2

EXERCISE GH AUTH EXERCISE

- Set up the following:
  - Authorization server >> Use Github (or set up Spring Authorization Server) to allow access to the application.
  - Client >> Create a basic application that manages a catalog of books, accessible only with a valid JWT. (Can be Controller + template engine or RestController)
- Use case:
  - Users must log in to access the system via Github
  - Once authenticated, they can view a list of books and add books to the list

ACCESSING PROTECTED RESOURCES

- Third party APIs can be protected by OAuth2
- The client puts a Bearer token in the Authorization header
- Current users can also access protected resources when they are logged in via OAuth2 or OpenID connect, they will get an access token from the authorization server that they can use directly

SPRING AUTHORIZATION SERVER

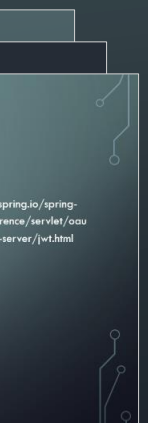
- Fully customizable
- Light weight
- Own authorization server, so no license cost
- In real life, alternatives such as Okta and Keycloak are used
- It’s great for learning and testing and POCs

KEYCLOAK

- Open source identity and access management implementation
- SSO
- Allows to create users database with custom roles and groups
- Maintained by Redhat

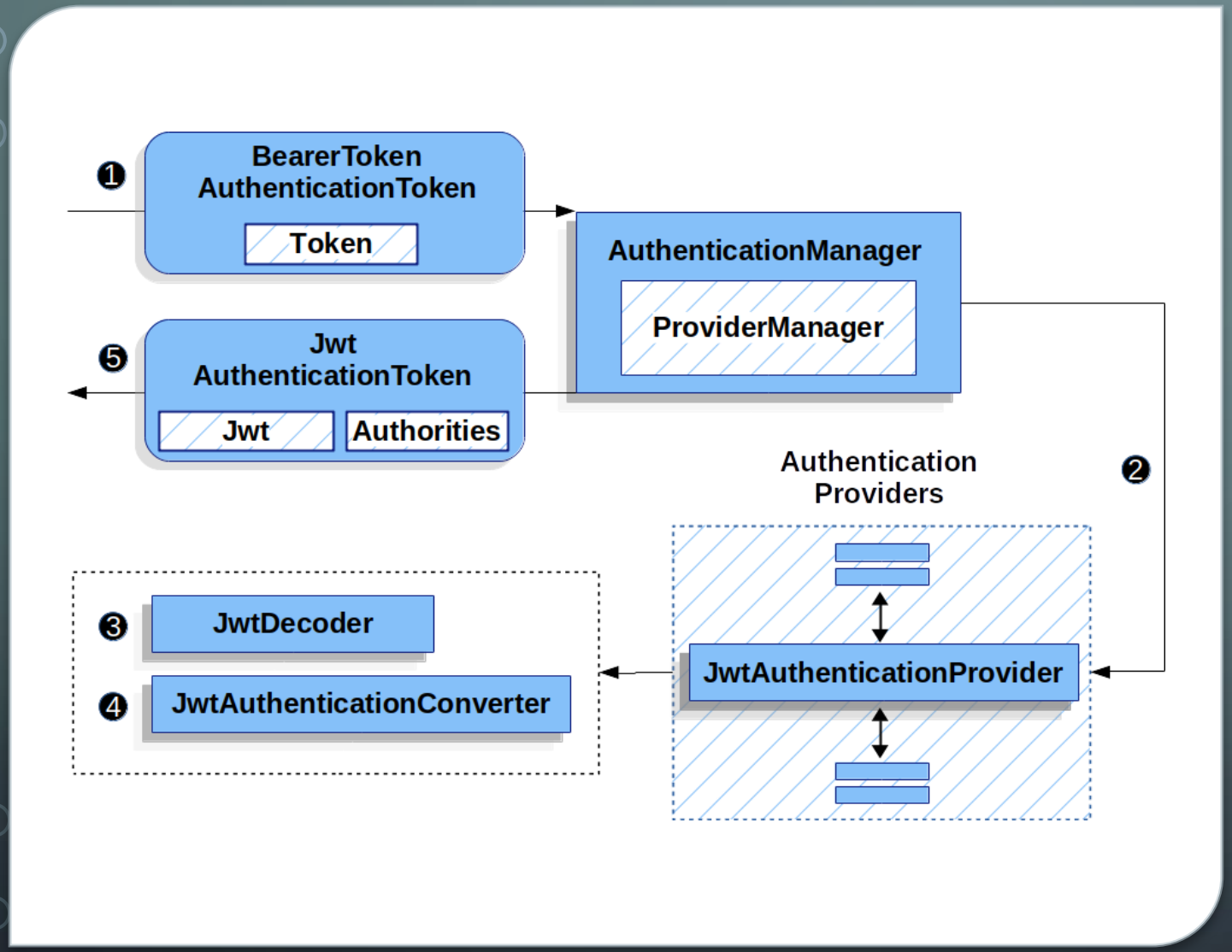
JWT (JSON WEB TOKEN)

- JWT is a compact, URL-safe means of representing claims between two parties.
- It consists of three parts: the header, the payload, and the signature.
- The header typically consists of two parts: the type of the token, which is JWT, and the signing algorithm being used, such as HMAC SHA256 or RSA.
- The payload contains the claims. Claims are statements about an entity (typically, the user) and additional metadata.
- The signature is used to verify that the sender of the JWT is who it says it is and to ensure that the message wasn't changed along the way.



STATELESS AUTHENTICATION

- Stateless authentication means the server does not need to keep additional information (session state) about the user for the duration of the request.
- Authentication is performed on each request by decoding and verifying the JWT, which includes all necessary data about the user's session.
- In Spring Security, you can implement stateless JWT authentication by integrating an authentication filter that checks for the presence of a JWT in the Authorization header of each request.



JWT

- https://docs.spring.io/spring-security/reference/servlet/oauth2/resource-server/jwt.html

KEYCLOAK

Guides Docs Downloads Community Blog

Downloads 24.0.1

For a list of community maintained extensions check out the Extensions page.

Server

Download and unzip

Keycloak	Distribution powered by Quarkus	ZIP (sha1) TAR.GZ (sha1)
Container image	For Docker, Podman, Kubernetes and OpenShift	Quay
Operator	For Kubernetes and OpenShift	OperatorHub

Quickstarts

Quickstarts distribution	GitHub	ZIP

Client Adapters

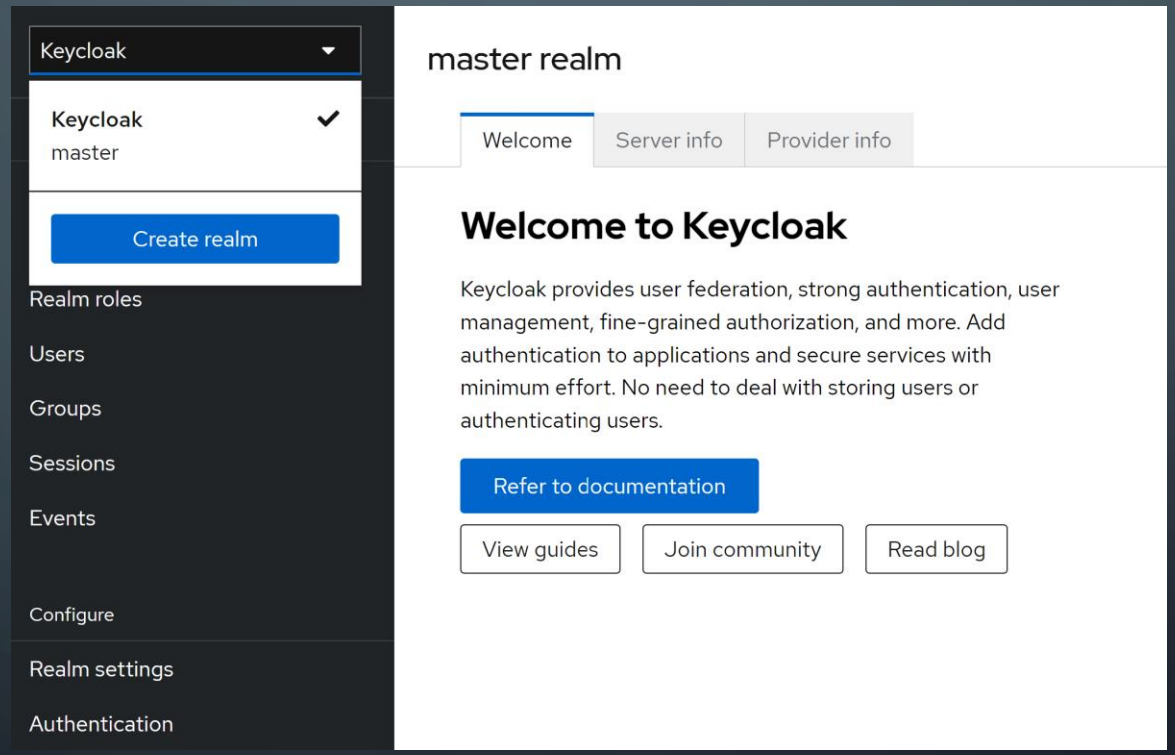
OpenID Connect SAML 2.0

JavaScript	NPM	ZIP (sha1)	TAR.GZ (sha1)

GETTING STARTED

- Many different guides for getting started
- On windows run: bin\kc.bat start-dev
  On Mac/Linux: bin/kc.sh start-dev
- Go to: http://localhost:8080/ and create an admin user
- Open administration console and log in

CREATE REALM



Create realm

A realm manages a set of users, credentials, roles, and groups. A user belongs to and logs into a realm. Realms are isolated from one another and can only manage and authenticate the users that they control.

Resource file

Drag a file here or browse to upload | Browse... | Clear

1

Upload a JSON file

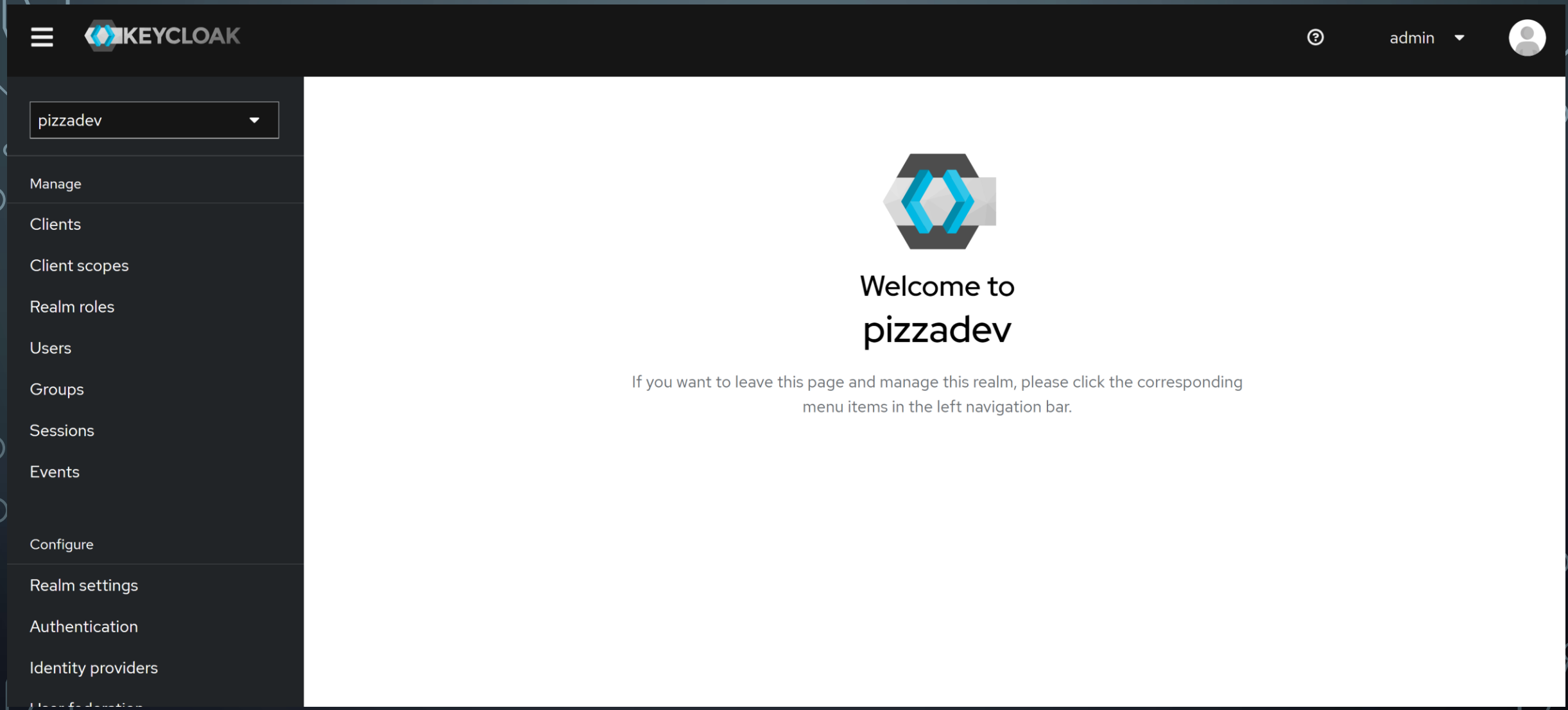
Realm name *

pizzadev

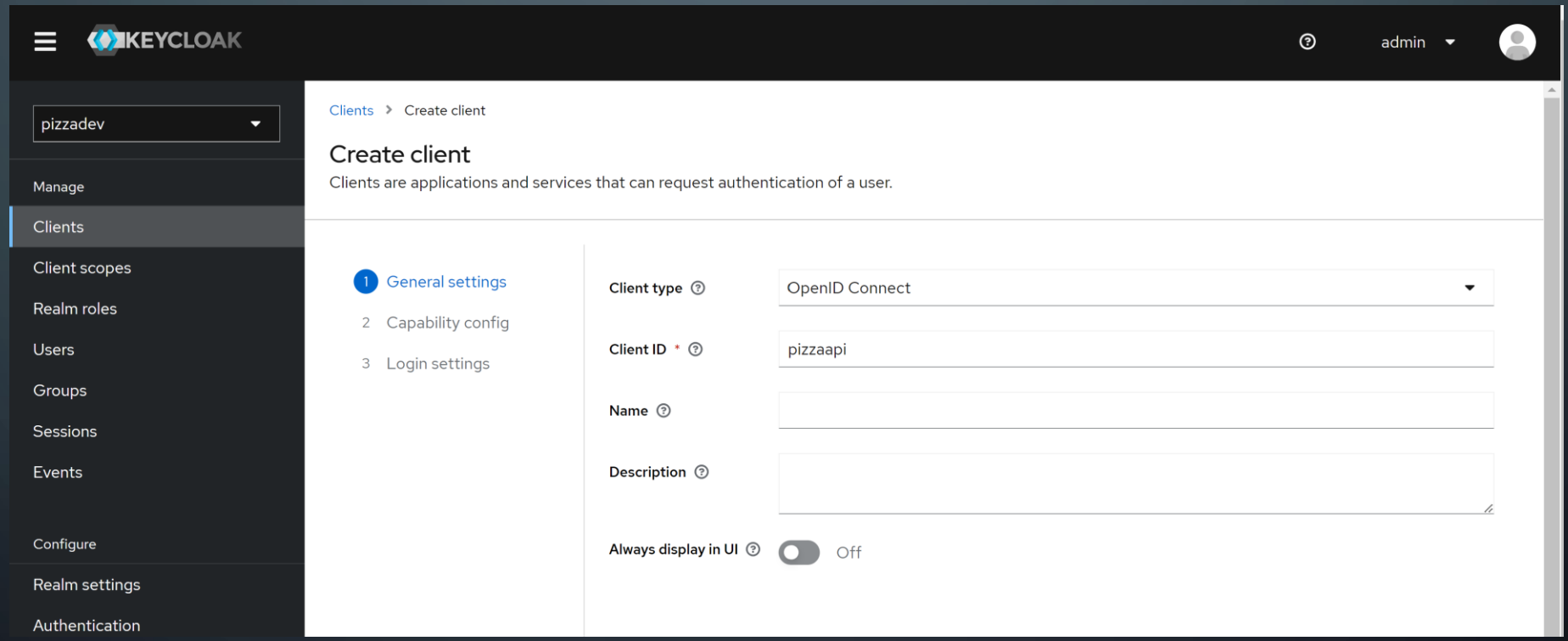
Enabled

On

Create | Cancel


KEYCLOAK
admin
pizzadev
Manage
Clients
Client scopes
Realm roles
Users
Groups
Sessions
Events
Configure
Realm settings
Authentication
Identity providers
Welcome to
pizzadev
If you want to leave this page and manage this realm, please click the corresponding menu items in the left navigation bar.

CREATE CLIENT



Client created successfully

Clients > Client details

pizzaapi

OpenID Connect

Enabled

Action

Clients are applications and services that can request authentication of a user.

Settings | Keys | Credentials | Roles | Client scopes | Service accounts roles | Sessions | Advanced

General settings

Client ID * pizzaapi

Name

Description

Always display in UI Off

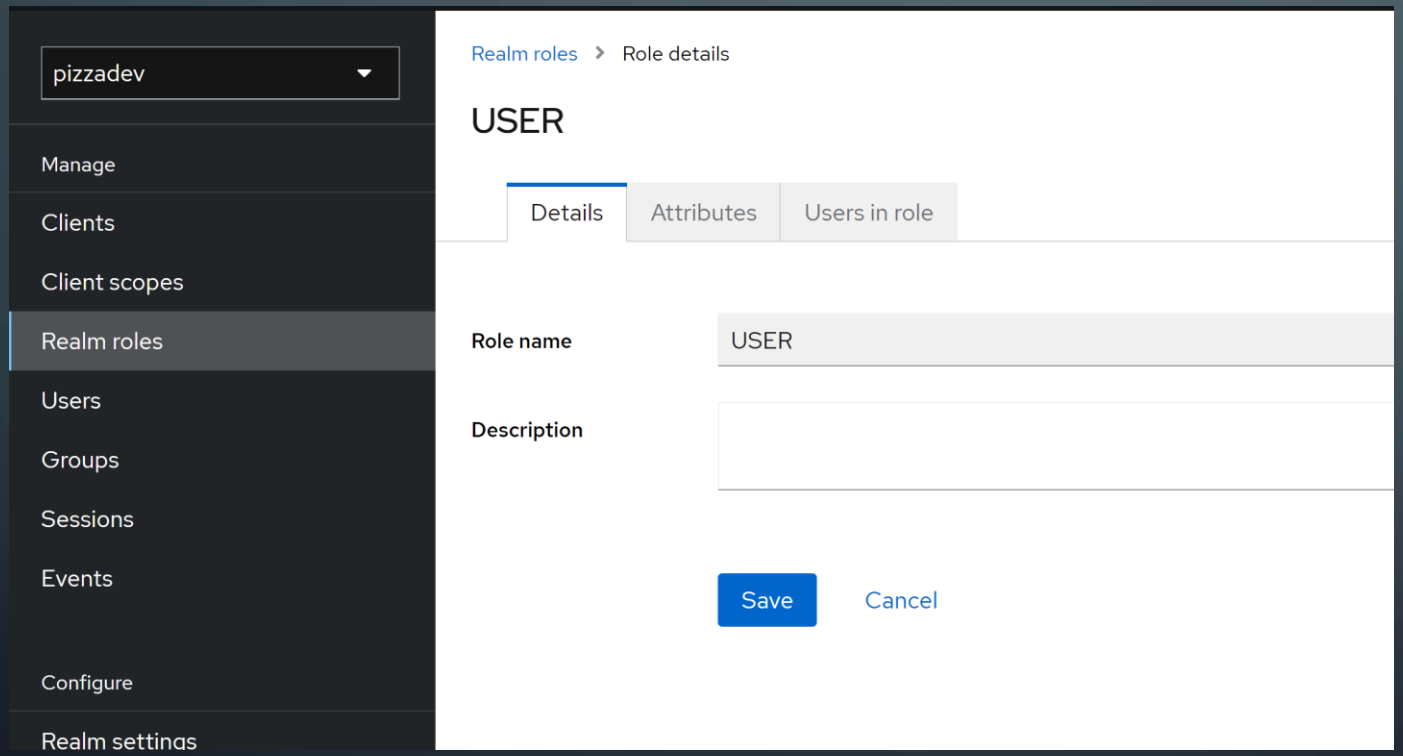
Access settings

Jump to section

- General settings
- Access settings
- Capability config
- Login settings
- Logout settings

Save Revert

CREATE ROLES USER AND ADMIN

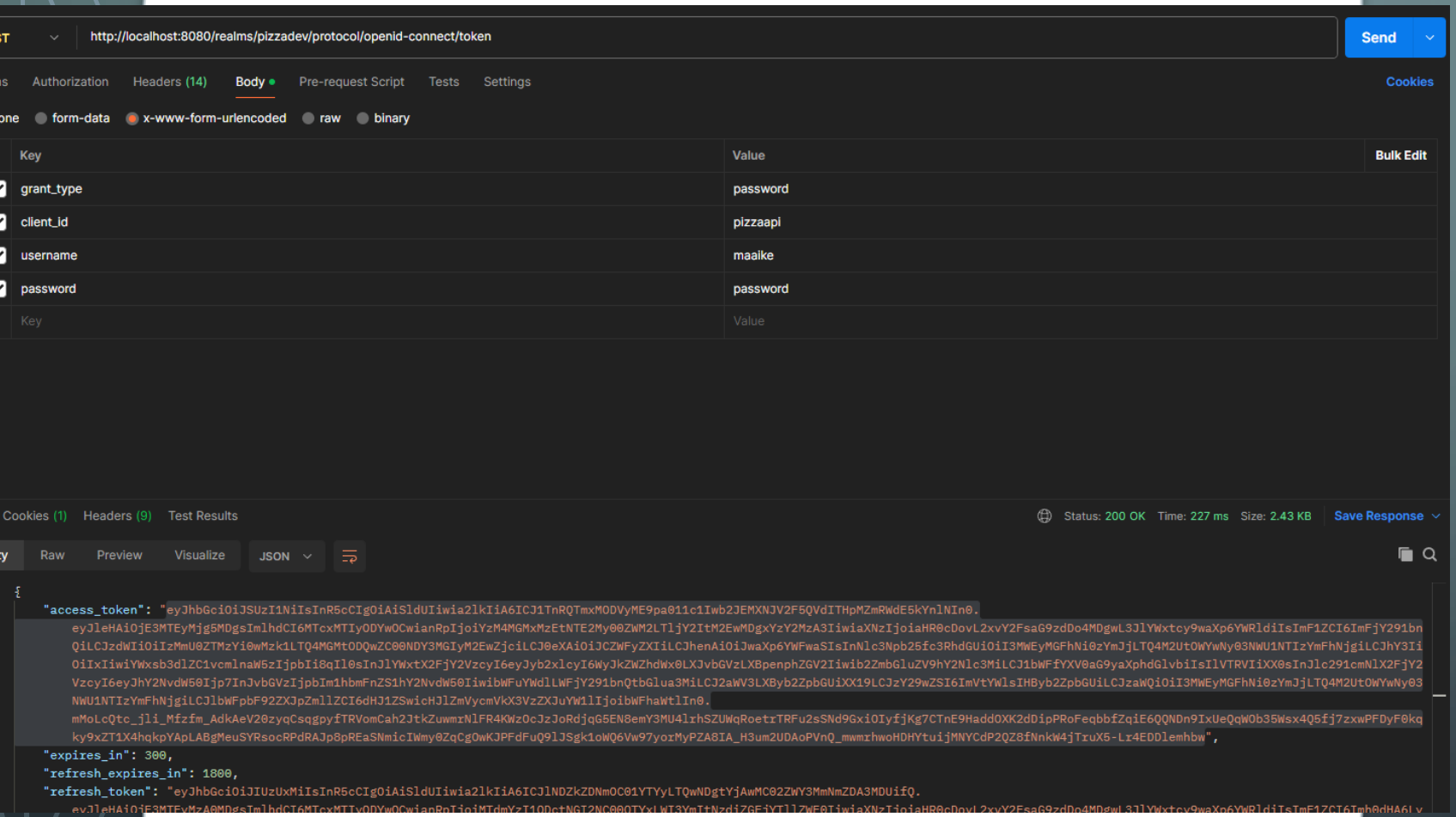


NEXT STEPS

- Create users >> give them credentials (make sure to not have temp password)
- Assign roles to users
- Go to client scopes and set *microprofile-jwt* to *default*

GENERATE TOKENS WITH POSTMAN

- Add the Content-Type application/x-www-form-urlencoded to your request
- You can find the URL in keycloak



CREATE A RESOURCE SERVER

- In this case it's a simple Pizza API with customers, orders and pizza endpoints
- Pizza is open to everybody
- Customers only to admin
- Orders are open to users

SECURITY CONFIG

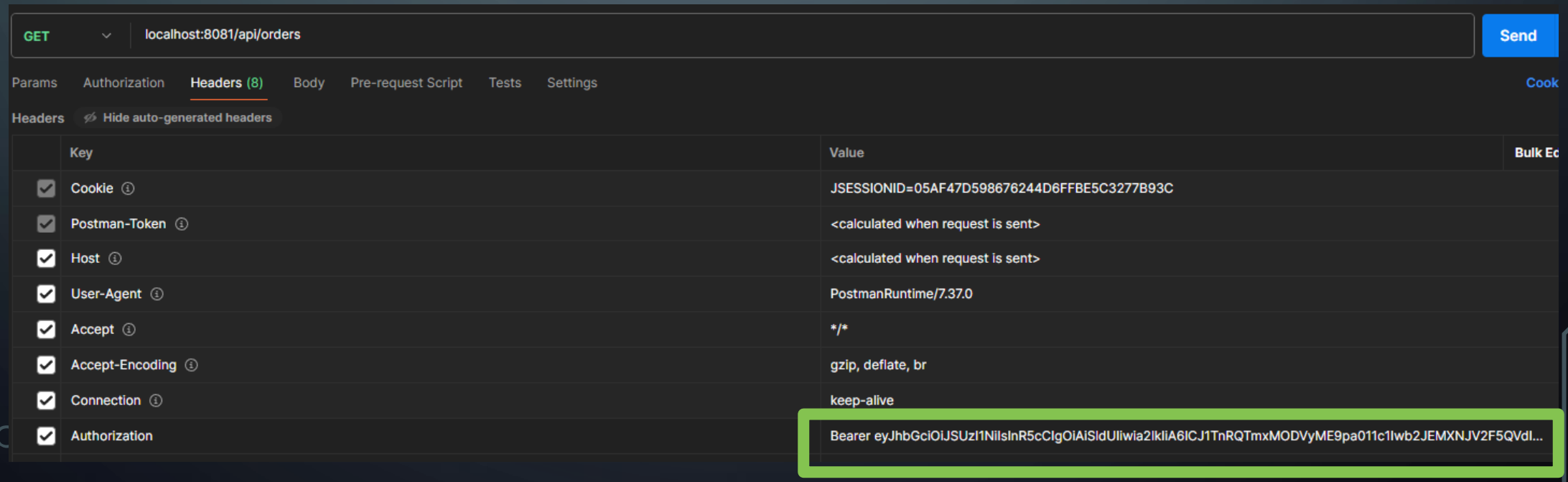
```java
@Configuration
@EnableWebSecurity
public class SecurityConfig {
    no usages  new *
    @Bean
    public SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {
        CsrfTokenRequestAttributeHandler requestHandler = new CsrfTokenRequestAttributeHandler();
        requestHandler.setCsrfRequestAttributeName("_csrf");
        JwtAuthenticationConverter jwtAuthenticationConverter = new JwtAuthenticationConverter();
        jwtAuthenticationConverter.setJwtGrantedAuthoritiesConverter(new KeycloakRoleTranslator());

        http.sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
                .csrf((csrf) -> csrf.csrfTokenRequestHandler(requestHandler).ignoringRequestMatchers( ...patterns: "/contact","/register")
                        .csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse()))
                .authorizeHttpRequests((requests)->requests
                        .requestMatchers(new AntPathRequestMatcher( pattern: "/api/pizzas/**")).permitAll() // Public access to view pizzas
                        .requestMatchers(new AntPathRequestMatcher( pattern: "/api/customers/**")).hasRole("ADMIN")
                        .requestMatchers(new AntPathRequestMatcher( pattern: "/api/orders/**")).hasRole("USER")
                        .anyRequest().authenticated())
                .oauth2ResourceServer(oauth2ResourceServerCustomizer ->
                        oauth2ResourceServerCustomizer.jwt(jwtCustomizer -> jwtCustomizer.jwtAuthenticationConverter(jwtAuthenticationConverter)));
        return http.build();
    }

}
```

CREATE A TRANSLATOR FOR THE ROLES

```java
public class KeycloakRoleTranslator implements Converter<Jwt, Collection<GrantedAuthority>> {

    new *
    @Override
    public Collection<GrantedAuthority> convert(Jwt jwt) {
        Map<String, Object> realmAccess = (Map<String, Object>) jwt.getClaims().get("realm_access");

        if (realmAccess == null || realmAccess.isEmpty()) {
            return new ArrayList<>();
        }

        Collection<GrantedAuthority> returnValue = ((List<String>) realmAccess.get("roles"))
                .stream().map(roleName -> "ROLE_" + roleName) Stream<String>
                .map(SimpleGrantedAuthority::new) Stream<SimpleGrantedAuthority>
                .collect(Collectors.toList());

        return returnValue;
    }

}
```

CALLING THE ENDPOINTS

- You need to obtain the token with the correct role to be able to access the endpoint, prefix with Bearer



EXERCISE

- Set up the following:
  - Use Keycloak or set up Spring Authorization Server to issue JWTs.
  - Create an application (similar to the previous) that allows seeing the list of books for all members of the library. But adding, editing and removing the catalog of books is accessible only for staff.
- Use case:
  - Library staff must be able to edit, delete and add books.
  - Members can see books.
  - Everybody can go to the contact endpoint to get the info about the library.
  - Challenge (not discussed): Set up an Angular/React/etc client app

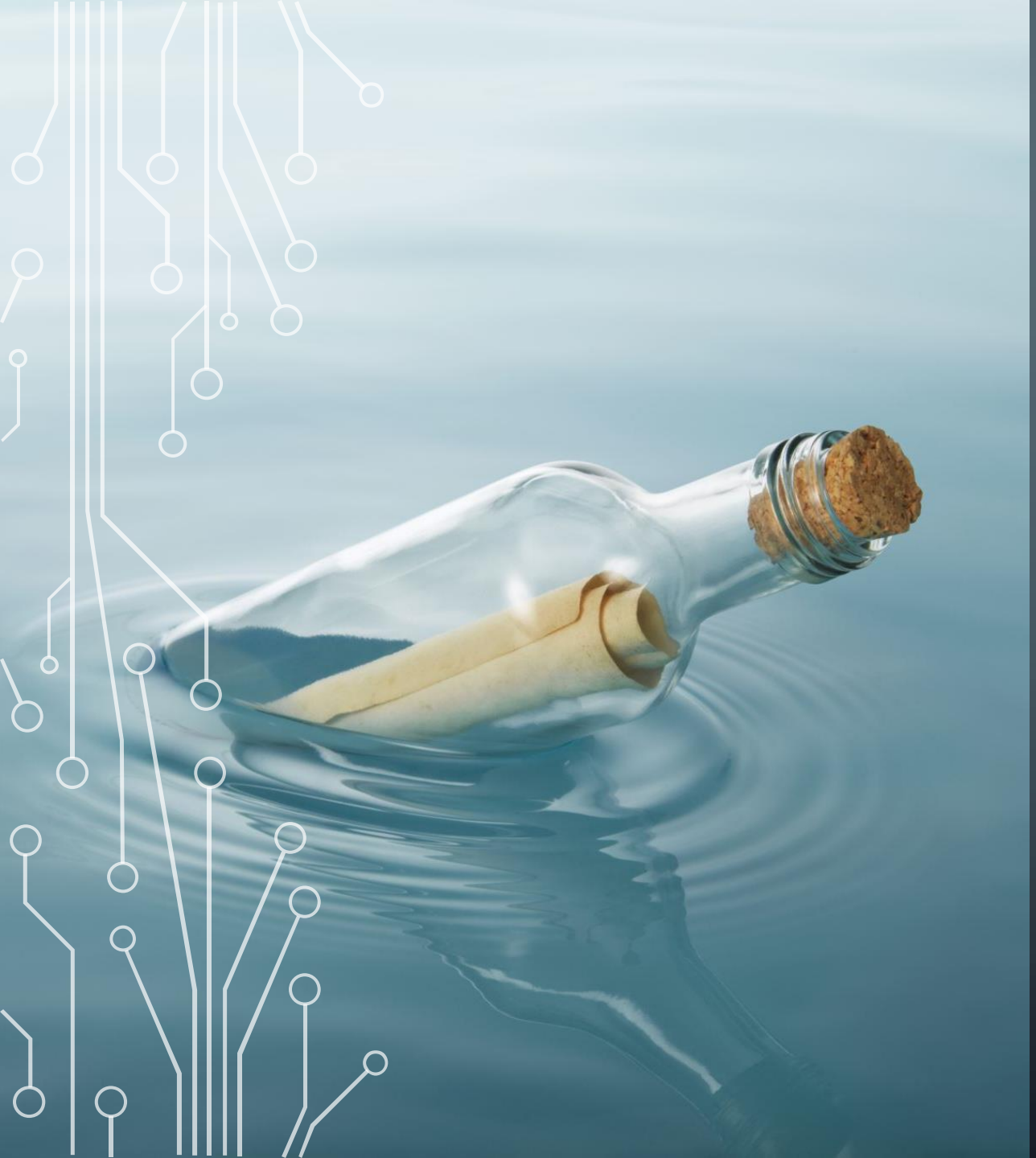
SECURITY EXPRESSIONS WITHIN @QUERY

- https://docs.spring.io/spring-security/reference/servlet/integrations/data.html (for configuration)

```
@Repository public interface MessageRepository extends
PagingAndSortingRepository<Message,Long> {
  @Query("select m from Message m where m.to.id = ?#{
      principal?.id }")
  Page<Message> findInbox(Pageable pageable);
}
```

NEXT UP

- Testing
- Webflux



QUESTIONS?

- Maaike.vanputten@brightboost.nl
- Or Whatsapp: +31683982426
- Don’t hesitate to contact me, I love to help!