Spring Documentation Part 2

Duy NTC

Version	**0.1**
Status:	**Draft**
Date:	**08.11.2021**
Prepared by:	**OpenWay**
Author:	**Duyntc**
Approved By:	**Duyntc**

1. Table of Contents

2. Restful web service

2.1. Restful fundamentals

REST fundamentals

REST works on top of the HTTP protocol. Each URI works as an API resource. Therefore, we should use nouns as endpoints instead of verbs. RPC-style endpoints use verbs, for example, `api/v1/getPersons`. In comparison, in REST, this endpoint could be simply written as `api/v1/persons`. You must be wondering, then, how can we differentiate between the different actions performed on a REST resource? This is where HTTP methods help us. We can make our HTTP methods act as a verb, for example, GET, DELETE, POST (for creating), PUT (for modifying), and PATCH (for partial updating). We'll discuss this in more detail later. For now, the `getPerson` RPC-style endpoint is translated into `GET api/v1/persons` in REST.

> **Note**
>
> The REST endpoint is a unique URI that represents a REST resource. For example, `https://demo.app/api/v1/persons` is a REST endpoint. Additionally, `/api/v1/persons` is the endpoint path and `persons` is the REST resource.

Here, there is client and server communication. Therefore, REST is based on the **Client-Server** concept. The client calls the REST API and the server responds with a response. REST allows a client (that is, a program, web service, or UI app) to talk to a remotely (or locally) running server (or web service) using HTTP requests and responses. The client sends to the web service with an API command wrapped in an HTTP request to the web. This HTTP request may contain a payload (or input) in the form of query parameters, headers, or request bodies. The called web service responds with a success/failure indicator and the response data wrapped inside the HTTP response. The HTTP status code normally denotes the status, and the response body contains the response data. For example, an HTTP status code of *200 OK* normally represents success.

From a REST perspective, an HTTP request is self-descriptive and has enough context for the server to process it. Therefore, REST calls are **stateless**. States are either managed on the client side or on the server side. A REST API does not maintain its state. It only transfers states from the server to the client or vice versa. Therefore, it is called REpresentation State Transfer, or REST for short.

It also makes use of HTTP cache control, which makes REST APIs **cacheable**. Therefore, the client can also cache the representation (that is, the HTTP response) because every representation is self-descriptive.

POST

The HTTP POST method is normally what you want to associate with creating resource operations. However, there are certain exceptions when you might want to use the POST method for read operations. However, it should be put into practice after a well-thought-out process. One such exception is the search operation where filter criteria have too many parameters that might cross the GET call's length limit.

A GET query string has a limit of 256 characters. Additionally, the GET HTTP method is limited to a maximum of 2,048 characters minus the number of characters in the actual path. On the other hand, the POST method is not limited by the size of the URL for submitting name and value pairs.

You may also want to use the POST method with HTTPS for a read call if the submitted input parameters contain any private or secure information.

For successful create operations, you can respond with the *201 Created* status, and for successful search or read operations, you should use the *200 OK* or *204 No Content* status codes, although the call is made using the POST HTTP method.

For failed operations, REST responses may have different error status codes based on the error type, which we will look at later in this section.

GET

The HTTP GET method is what you usually want to associate with read resource operations. Similarly, you must have observed the GitHub GET `/licenses` call that returns the available licenses in the GitHub system. Additionally, successful GET operations should be associated with the *200 OK* status code if the response contains data, or *204 No Content* if the response contains no data.

DELETE

The HTTP DELETE method is what you want to associate with delete resource operations. GitHub does not provide the DELETE operation on the licenses resource. However, if you assume it exists, it will certainly look very similar to `DELETE /licenses/agpl-3.0`. A successful delete call should delete the resource associate with the `agpl-3.0` key. Additionally, successful DELETE operations should be associated with the *204 No Content* status code.

PATCH

The HTTP PATCH method is what you want to associate with partial update resource operations. Additionally, successful PATCH operations should be associated with a *200 OK* status code. PATCH is relatively new as compared to other HTTP operations. In fact, a few years ago, Spring did not have state-of-the-art support for this method for REST implementation due to the old Java HTTP library. However, currently, Spring provides built-in support for the PATCH method in REST implementation.

HTTP status codes

There are five categories of HTTP status codes, as follows:

- Informational responses (100–199)
- Successful responses (200–299)
- Redirects (300–399)

- Client errors (400–499)
- Server errors (500–599)

Hateoas:

- ***Restful API best practice:***

1. Use nouns and not verbs when naming a resource in the endpoint path

We previously discussed HTTP methods. HTTP methods use verbs. Therefore, it would be redundant to use verbs yourself, and it would make your call look like an RPC endpoint, for example, `GET /getlicenses`. In REST, we should always use the resource name because, according to REST, you transfer the states and not the instructions.

2. Use the plural form for naming the collection resource in the endpoint path

If you observe the GitHub license API, you might find that a resource name is given in the plural form. It is a good practice to use the plural form if the resource represents a collection. Therefore, we can use `/licenses` instead of `/license`. A `GET` call returns the collection of licenses. A `style` call creates a new license in the existing license collection. For `delete` and `patch` calls, a license key is used to identify the specific license.

3. Use hypermedia (HATEOAS)

Hypermedia (that is, links to other resources) makes the REST client's job easier. There are two advantages if you provide explicit URL links in a response. First, the REST client is not required to construct the REST URLs on their own. Second, any upgrade in the endpoint path will be taken care of automatically and this, therefore, makes upgrades easier for clients and developers.

4. Always version your APIs

The versioning of APIs is key for future upgrades. Over time, APIs keep changing, and you may have customers who are still using an older version. Therefore, you need to support multiple versions of APIs.

- **Using headers**: The GitHub API uses this approach. You can add an `Accept` header that tells you which API version should serve the request; for example, consider the following:

```
Accept: application/vnd.github.v3+json
```

  This approach gives you the advantage of setting the default version. If there is no `Accept` header, it should lead to the default version. However, if a REST client that uses a versioning header is not changed after a recent upgrade of APIs, it may lead to a functionality break. Therefore, it is recommended that you use a versioning header.

- **Using an endpoint path**: In this approach, you add a version in the endpoint path itself; for example, `https://demo.app/api/v1/persons`. Here, `v1` denotes that version 1 is being added to the path itself.

  You cannot set default versioning out of the box. However, you can overcome this limitation by using other methods, such as request forwarding. Clients always use the intended versions of the APIs in this approach.

5. Nested resources

`GET /customers/1/addresses`: This returns the collection of addresses for customer 1.

`GET /customers/1/addresses/2`: This returns the second address of customer 1.

`POST /customers/1/addresses`: This adds a new address to customer 1's addresses.

6. Secure APIs

Securing your API is another expectation that requires diligent attention. Here are some recommendations:

- Always use HTTPS for encrypted communication.
- Always look for OWASP's top API security threats and vulnerabilities. These can be found on their website (`https://owasp.org/www-project-api-security/`) or the GitHub repository (`https://github.com/OWASP/API-Security`).
- Secure REST APIs should have authentication in place. REST APIs are stateless; therefore, REST APIs should not use cookies or sessions. Instead, these should be secure using JWT or OAuth 2.0-based tokens.

7. Documentation

Documentation should be easily accessible and up to date with the latest implementation with their respective versioning. It is always good to provide sample code and examples. It makes the developer's integration job easier.

A change log or a release log should list all of the impacted libraries, and if some APIs are deprecated, then replacement APIs or workarounds should be elaborated on inside the documentation.

8. Status codes

You might have already learned about status code in the *Exploring HTTP methods and status codes* section. Please follow the same guidelines discussed there.

9. Caching

HTTP already provides the caching mechanism. You just have to provide additional headers in the REST API response. Then, the REST client makes use of the validation to make sure whether to make a call or use the cached response. There are two ways to do it:

ETag: ETag is a special header value that contains the hash or checksum value of the resource representation (that is, the response object). This value must change with respect to the response representation. It will remain the same if the resource response doesn't change.

> Now, the client can send a request with another header field, called `If-None-Match`, which contains the ETag value. When the server receives this request and finds that the hash or checksum value of the resource representation value is different from `If-None-Match`, only then should it return the response with a new representation and this hash value in the ETag header. If it finds them to be equal, then the server should simply respond with a `304 (Not Modified)` status code.

Last-Modified: This approach is identical to the ETag way. Instead of using the hash or checksum, it uses the timestamp value in RFC-1123 format (*Last-Modified: Wed, 21 Oct 2015 07:28:00 GMT*). It is less accurate than ETag and should only be used for the falling mechanism.

Here, the client sends the `If-Modified-Since` header with the value received in the `Last-Modified` response header. The server compares the resource-modified timestamp value with the `If-Modified-Since` header value and sends a `304` status if there is a match; otherwise, it sends the response with a new `Last-Modified` header.

10. Rate limit

This is important if you want to prevent the overuse of APIs. The HTTP status code `429 Too Many Requests` is used when the rate limit goes over. Currently, there is no standard to communicate any warning to the client before the rate limit goes over. However, there is a popular way to communicate about it using response headers; these include the following:

- `X-Ratelimit-Limit`: The number of allowed requests in the current period
- `X-Ratelimit-Remaining`: The number of remaining requests in the current period
- `X-Ratelimit-Reset`: The number of seconds left in the current period
- `X-Ratelimit-Used`: The number of requests used in the current period

2.2. Introduction to Spring

Spring is a framework and is written in the Java language. It provides lots of modules, such as Spring Data, Spring Security, Spring Cloud, Spring Web, and so on. It is popular for building enterprise applications. Initially, it was looked at as a Java **Enterprise Edition (EE)** alternative. However, over the years, it has been preferred over Java EE. Spring supports **Dependency Injection** (**Inversion of Control**) and **Aspect-Oriented Programming** out of the box as its core. Apart from Java, Spring also supports other JVM languages such as Groovy and Kotlin.

The Dependency Injection pattern

Let's say you are writing a program that needs some data from a database. The program needs a database connection. You could use the JDBC database connection object. You could instantiate and assign the database connection object instantly in the program. Or, you could simply take the connection object as a constructor or a setter/factory method parameter. Then, the framework creates the connection object as per the configuration and assigns that object to your program at runtime. Here, the framework actually injects the connection object at runtime. This is called DI. Spring supports DI for class compositions.

> **Note:**
> The Spring Framework throws an error at runtime if any dependency is not available, or the proper object name is not marked when more than one type of object is available. In contrast to that, there are some frameworks that also check these dependencies at compile time, for example, Dagger.

DI is a type of IoC. IoC containers construct and maintain implementation objects. These types of objects (objects required by other objects – a form of dependency) are injected into objects that need them in a constructor, setter, or interface. This decouples the instantiation and allows dependency injection at runtime. Dependency injection can also be achieved using the Service Locator pattern. However, we'll stick to the IoC pattern approach.

TL;DR:

Depenpendency Injection is simply created object at **runtime** depend on configuration.

The Aspect-Oriented Programming paradigm

We have talked about procedural programming (for IoC) and object-oriented programming (for IoC and DI). Then comes AOP, another programming paradigm. AOP works in tandem with OOP. It's a good practice in OOP to handle only a single responsibility in a particular class – this principle is called the **Single Responsibility Principle** (applicable for modules/classes/methods). For example, if you are writing a `Gear` class in an automotive domain application, then the `Gear` class should only allow functions related to the gear object, or it should not be allowed to perform other functions such as braking. However, in programming models, often you need a feature/function that scatters across more than one class. In fact, sometimes, most classes use features such as logging or metrics.

AOP allows you to do the following:

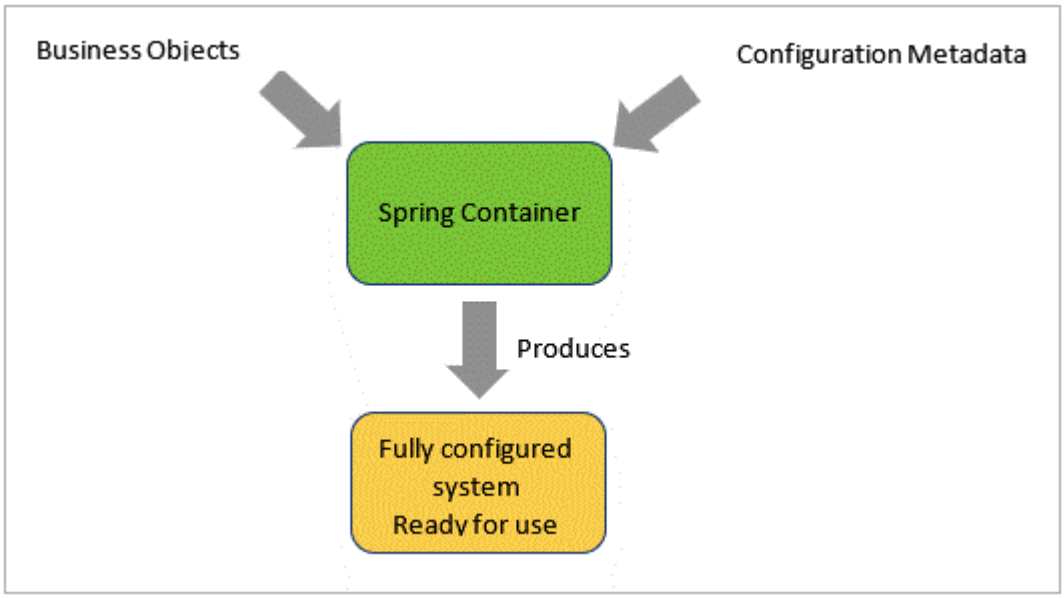
- To abstract and encapsulate cross-cutting concerns.
- To add aspects behavior around your code.
- To make code modular for cross-cutting concerns to easily maintain and extend it.
- To focus on your business logic inside code. This makes code clean. Cross-cutting concerns are encapsulated and maintained separately.

2.2.1. Basic of Spring frameworks

- ***Application context:***

Application is IoC Container manage the **beans**. Bean get instantiate, assemble and configure by:

- XML configuration.
- Java annotation.
- Java code.



- ***Defining beans***

Beans are Java objects that are managed by **IoC containers**. The developer supplies the configuration metadata to the IoC container, which is then used by the container to construct, assemble, and manage the beans.

Beans should have a unique identifier inside a container. A bean may have more than one identity using an **alias.**

```java
@Slf4j
class SampleBean {
    public void init(){
      log.info("Initialize Sample Bean");
    }

    public void destroy(){
        log.info("Destroy Sample Bean");
    }

}

@Configuration
public class BeanConfiguration {

    @Bean(initMethod = "init", destroyMethod = "destroy", name = {"Sample Bean"})
    @Description("Sample Bean LMAO")
    public SampleBean getSampleBean(){
        return new SampleBean();
    }

}
```

- ***Bean scope***

The default scope is singleton, that is, only one instance would be created per IoC container, and the same instance would be injected. If you want to create a new instance each time it is requested, you can define the prototype scope for the bean.

Scope	Usage
singleton	Creates a new instance per IoC container. Default scope.
prototype	Creates a new instance for each injection (for collaborated beans).
request	Only for web-aware context. A single bean instance would be created for each HTTP request and valid throughout the HTTP request life cycle.
session	Only for web-aware context. A single bean instance would be created for each HTTP session and valid throughout the HTTP session life cycle.
application	Only for web-aware context. A single instance would be created for application scope, that is, valid throughout the life cycle of the servlet-context.
websocket	Only for web-aware context. A single instance would be created for each WebSocket session.

- ***Depend on:***

Depend on can be used to configure bean initalize order.

```java
@Slf4j
class SampleBean {
    public void init(){
        log.info("Initialize Sample Bean");
    }
}

@Slf4j
class PreBean {
    public void init(){
        log.info("Initialize This Bean first before Sample Bean");
    }
}
```

```java
@Configuration
public class BeanConfiguration {

    @Bean(initMethod = "init")
    public PreBean getPreBean(){
        return new PreBean();
    }

    @Bean(initMethod = "init", name = {"Sample Bean"})
    @DependsOn({"getPreBean"})
    @Description("Sample Bean LMAO")
    public SampleBean getSampleBean(){
        return new SampleBean();
    }
}
```

```
2021-10-26 08:24:42.815  INFO 5144 --- [  restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port(s): 8080 (http)
2021-10-26 08:24:42.815  INFO 5144 --- [  restartedMain] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2021-10-26 08:24:42.815  INFO 5144 --- [  restartedMain] org.apache.catalina.core.StandardEngine  : Starting Servlet engine: [Apache Tomcat/9.0.54]
2021-10-26 08:24:42.859  INFO 5144 --- [  restartedMain] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
2021-10-26 08:24:42.859  INFO 5144 --- [  restartedMain] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 684 ms
2021-10-26 08:24:42.891  INFO 5144 --- [  restartedMain] com.dinj.restful.config.PreBean          : Initialize This Bean first before Sample Bean
2021-10-26 08:24:42.891  INFO 5144 --- [  restartedMain] com.dinj.restful.config.SampleBean       : Initialize Sample Bean
2021-10-26 08:24:43.060  INFO 5144 --- [  restartedMain] o.s.b.d.a.OptionalLiveReloadServer       : LiveReload server is running on port 35729
2021-10-26 08:24:43.091  INFO 5144 --- [  restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8080 (http) with context path ''
2021-10-26 08:24:43.091  INFO 5144 --- [  restartedMain] com.dinj.restful.RestfulApplication      : Started RestfulApplication in 1.248 seconds (JVM running for 1.78)
```

- ***Dependency Injection***

	Type
Constructor base injection	
Setter injection	
Field Injection	 Field injection is not recommend because: Field-based dependency injection won't work on fields that are declared final/immutable as this fields must be instantiated at class instantiation.

- ***Qualifier annotation***

```
@Controller
public class CartController {    @Autowired
    @Qualifier("cartService1")
    private CartService service1;

    @Autowired
    @Qualifier("cartService2")
    private CartService service2;
}
```

Match by name

Let's define a service using the `@Service` annotation, which is a type of `@Component`. Let's assume we have a component scan in place:

```
@Service(value="cartServc")
public class CartService {
    // code
}

@Controller
public class CartController {    @Autowired
    private CartService cartServc;
}
```

What is the purpose of @Primary?

In the previous subsection, we saw that `@Qualifier` helps you to resolve which type should be used when multiple beans are available for injection. The `@Primary` annotation allows you to set one of the type's beans as the default. Bean annotation with `@Primary` will be injected into autowired fields:

```
@Configuration
public class AppConfig {    @Bean
    @Primary
    public CartService cartService1() {
        return new CartServiceImpl1();
```

When we can use @Value?

Spring supports the use of external property files: `<xyz>.properties` or `<xyz>.yml`. Now, you want to use the value of any property in your code. You can achieve this using the `@Value` annotation. Let's have a look at the sample code:

```
sample.bean.init=Initialize Sample bean
```

```
@Slf4j
class SampleBean {

    @Value("${sample.bean.init}")
    String initString;

    public void init(){
      log.info(initString);
    }
}
```

```
6420 --- [  restartedMain] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in

6420 --- [  restartedMain] com.dinj.restful.config.PreBean          : Initialize This Bean first before Sample Bean
6420 --- [  restartedMain] com.dinj.restful.config.SampleBean       : Initialize Sample bean
6420 --- [  restartedMain] o.s.b.d.a.OptionalLiveReloadServer       : LiveReload server is running on port 35729
6420 --- [  restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8080 (http) with context path
```

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-configuration-processor</artifactId>
    <optional>true</optional>
</dependency>
```

2.2.2. Aspect Oriented Programming

Writing code for AOP

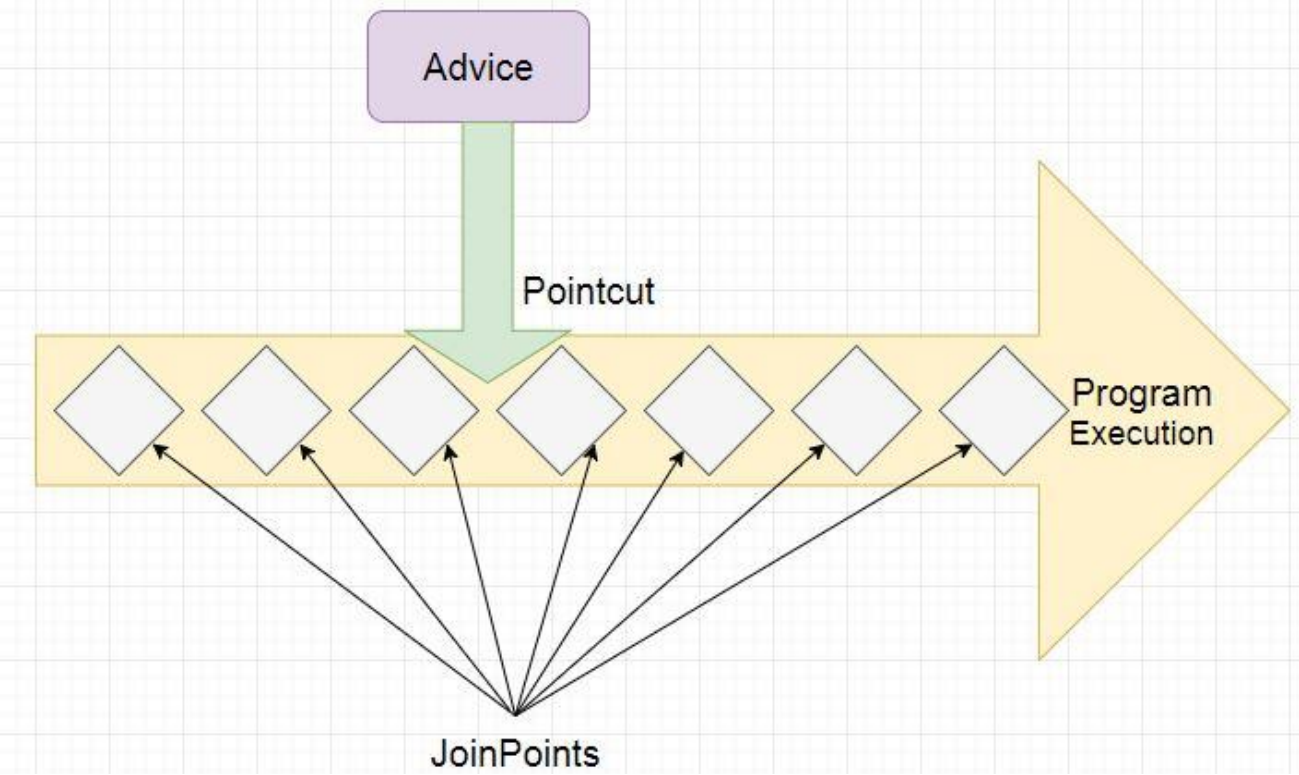
We discussed AOP previously, in the *Introduction to Spring* section. In simple terms, it is a programming paradigm that solves cross-cutting concerns such as logging, transactions, security, and so on. These cross-cutting concerns are known as aspects in AOP. It allows you to modularize your code and place cross-cutting concerns in a central place.

This code captures the time taken by itself. If you have hundreds of methods in your application, then you need to add a time-capturing piece of code in each one of them to monitor time. Moreover, what if you want to modify the code? It means you have to modify the code in all those places. You don't want to do that. This is where AOP helps you. It makes your cross-cutting code modular.

During software development, functions that span multiple points of an application are called **cross-cutting concerns**. These cross-cutting concerns differ from the applicatio main business logic.

Hence, separating these cross-cutting concerns from the business logic is where aspect-oriented programming (AOP) comes into picture.

Aspects have responsibility they are meant to **discharge**. This responsibility of an aspect is called **advice**.



Aspect Oriented Programming Core Concepts

Aspect: An aspect is a class that implements enterprise application concerns that cut across multiple classes, such as transaction management.

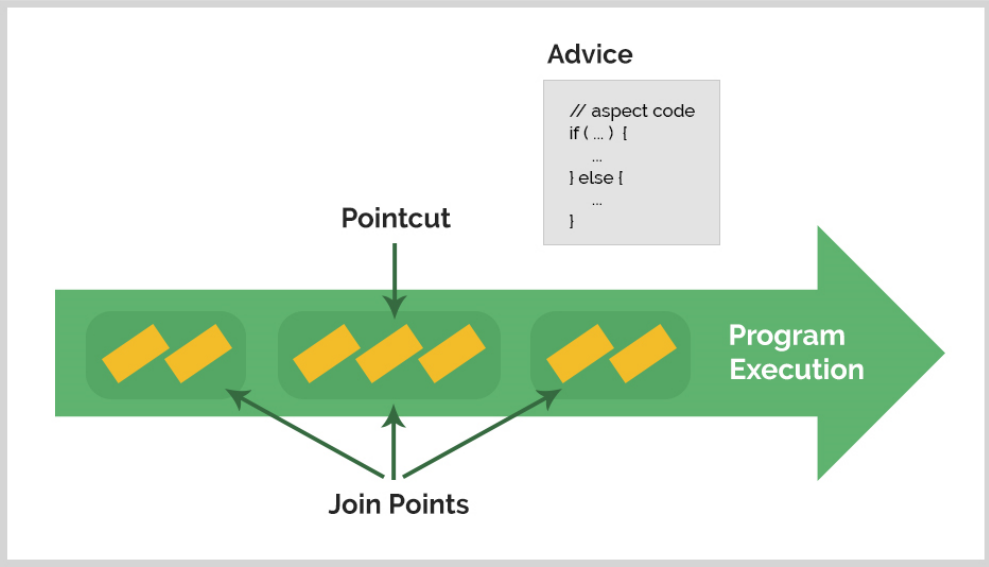
Join Point: A join point is **a specific point** in the application such as **method execution, exception handling, changing object variable values**, etc. In Spring AOP a join point is always the **execution of a method**.

Advice: Advice are **actions** taken for a particular join point. In terms of programming, they are methods that get executed when a certain join point with matching pointcut is reached in the application.

Pointcut: Pointcut is expressions that are matched with join points to determine whether advice needs to be executed or not. Pointcut uses different **kinds of expressions** that are matched with the **join points** and **Spring framework** uses the **AspectJ pointcut expression language**.

Target Object: They are the **object on which advice are applied**. Spring AOP is implemented using runtime proxies, so this object is always a proxied object.

Weaving: It is the process of linking aspects with other objects to create the advised proxy objects



AOP Advice Types

Based on the execution strategy of advice, they are of the following types.

1. **Before Advice**: These advices runs before the execution of join point methods. We can use `@Before` annotation to mark an advice type as Before advice.
2. **After (finally) Advice**: An advice that gets executed after the join point method finishes executing, whether normally or by throwing an exception. We can create after advice using `@After` annotation.
3. **After Returning Advice**: Sometimes we want advice methods to execute only if the join point method executes normally. We can use `@AfterReturning` annotation to mark a method as after returning advice.
4. **After Throwing Advice**: This advice gets executed only when join point method throws exception, we can use it to rollback the transaction declaratively. We use `@AfterThrowing` annotation for this type of advice.
5. **Around Advice**: This is the most important and powerful advice. This advice surrounds the join point method and we can also choose whether to execute the join point method or not. We can write advice code that gets executed before and after the execution of the join point method. It is the responsibility of around advice to invoke the join point method and return values if the method is returning something. We use `@Around` annotation to create around advice methods.

▶ *Sample code for AOP*

```java
@Component
public class EmployeeManager
{
    public String getEmployeeById(Integer employeeId) {
        System.out.println("Method getEmployeeById() called");
        return "getEmployeeById [" + employeeId + "]";
    }
}
```

1. Target Class

```java
@SpringBootApplication
public class RestfulApplication {

    public static void main(String[] args) {
        ConfigurableApplicationContext context = SpringApplication.run(RestfulApplication.class, args);
        EmployeeManager manager = (EmployeeManager) context.getBean( s: "employeeManager");
        manager.getEmployeeById( employeeId: 1);
    }
```

2. Bean called

```java
@Before("execution(* com.dinj.restful.model.EmployeeManager.getEmployeeById(..))")        //point-cut expression
public void logBeforeV1(JoinPoint joinPoint)
{
    System.out.println("Before EmployeeCRUDAspect.logBeforeV1() : " + joinPoint.getSignature().getName());
}

@After("execution(* com.dinj.restful.model.EmployeeManager.getEmployeeById(..))")        //point-cut expression
public void logAfterV1(JoinPoint joinPoint)
{
    System.out.println("After EmployeeCRUDAspect.logBeforeV1() : " + joinPoint.getSignature().getName());
}
```

3. Simple join point expression

```java
@Pointcut("execution(* com.dinj.restful.model.EmployeeManager.getEmployeeById(..))")
private void getEmployee() {};

@Before("getEmployee()")
public void doBeforeTask(JoinPoint joinPoint){
    System.out.println("@Pointcut EmployeeCRUDAspect.doBeforeTask() : " + joinPoint.getSignature().getName());
}
```

4. Point Cut

```java
@Around("getEmployee()")
public Object aroundAdvice(ProceedingJoinPoint joinPoint) throws Throwable{
    System.out.println("Start around advice");
    Object[] args = joinPoint.getArgs();
    if(args.length>0){
        System.out.print("Arguments passed: " );
        for (int i = 0; i < args.length; i++) {
            System.out.print("arg "+(i+1)+": "+args[i]);
        }
        System.out.println();
    }
    Object result = joinPoint.proceed(args);
    System.out.println("Returning " + result);
    return result;
}
```

5. Around consume the method and return afterwards

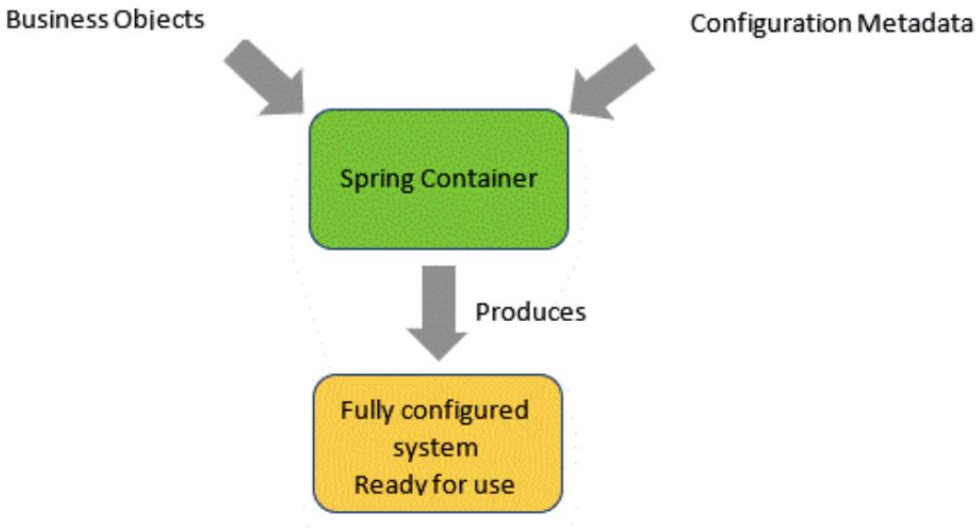
2.2.3. The IoC container

IoC container is the backbone of spring:

- Bean Factory
- Application Context

ApplicationContext

Now, you know that the `ApplicationContext` interface represents the IoC container and manages the beans, you must be wondering how it gets to know about what beans to instantiate, assemble, and configure. From where does it get its instruction? The answer is configuration metadata. Configuration metadata allows you to express your application objects and interdependencies among those objects. Configuration metadata can be represented in three ways: through XML configuration, Java annotations, and Java Code. You write the business objects and provide the configuration metadata, and the Spring container generates a ready-to-use fully configured system as shown:



Defining beans

Beans are Java objects that are managed by IoC containers. The developer supplies the configuration metadata to the IoC container, which is then used by the container to construct, assemble, and manage the beans. Beans should have a unique identifier inside a container. A bean may have more than one identity using an **alias**.

```java
@Slf4j
class SampleBean {

    @Value("Initialize Sample bean")
    String initString;

    public void init() { log.info(initString); }
}

@Slf4j
class PreBean {
    public void init() { log.info("Initialize This Bean first before Sample Bean"); }
}

@Configuration
public class BeanConfiguration {

    @Bean(initMethod = "init")
    public PreBean getPreBean() { return new PreBean(); }

    @Bean(initMethod = "init", name = {"Sample Bean"})
    @DependsOn({"getPreBean"})
    @Description("Sample Bean LMAO")
    public SampleBean getSampleBean() { return new SampleBean(); }
}
```

The bean's scope

The Spring container is responsible for creating the bean's instances. How instances will be created by the Spring container are defined by `scope`. The default `scope` is `singleton`, that is, only one instance would be created per IoC container and the same instance would be injected. If you want to create a new instance each time it is requested, you can define the `prototype` scope for the bean.

Scope	Description
`singleton` (default)	Single bean object instance per spring IoC container
`prototype`	Opposite to singleton, it produces a new instance each and every time a bean is requested.
`request`	A single instance will be created and available during complete lifecycle of an HTTP request. Only valid in web-aware Spring `ApplicationContext`.
`session`	A single instance will be created and available during complete lifecycle of an HTTP Session. Only valid in web-aware Spring `ApplicationContext`.
`application`	A single instance will be created and available during complete lifecycle of `ServletContext`. Only valid in web-aware Spring `ApplicationContext`.
`websocket`	A single instance will be created and available during complete lifecycle of `WebSocket`. Only valid in web-aware Spring `ApplicationContext`.

Configuring beans using Java

Before Spring 3, you could only define beans using Java. Spring 3 introduced the `@Configuration`, `@Bean`, `@import`, and `@DependsOn` annotations to configure and define Spring beans using Java.

You have already learned about the `@Configuration` and `@Bean` annotation in the *Defining beans* section. Now, you will explore how to use the `@import` and `@DependsOn` annotations.

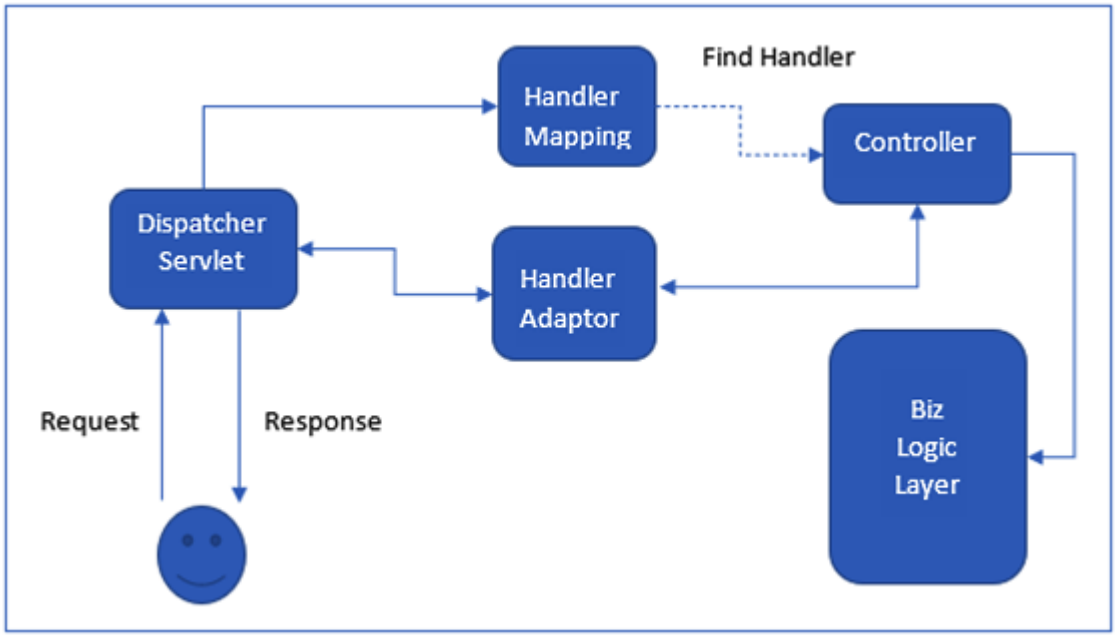
The `@Import` annotation is more useful when you develop an application without using autoconfiguration.

Purpose of servlet dispatcher

In the previous chapter, you learned that RESTful web services are developed on top of the HTTP protocol. Java has a Servlets feature to work with HTTP protocol. Servlets allow you to have path mapping that can work at REST endpoints and provides the HTTP method for identification. It also allows you to form different types of response objects, including JSON and XML. However, it is a crude way of implementing REST endpoints. You have to handle the request URI, parse the parameters, convert JSON/XML, and handle the responses.

Spring MVC comes to your rescue. Spring MVC is based on the **Model-View-Controller (MVC)** pattern and has been part of the Spring Framework since its first release. MVC is a well-known design pattern:

- **Model**: Models are Java objects (POJOs) that contain the application data. They also represent the state of the application.
- **View**: The view is a presentation layer that consists of HTML/JSP/template files. The view renders the data from models and generates the HTML output.
- **Controller**: The controller processes the user requests and builds the model.



3. API Specifications and Implementation

- ***Design API with OAS***

Designing APIs with OAS

You could directly start coding the API; however, this approach leads to many issues, such as frequent modifications, difficulty in API management, and difficulty in reviews specifically lead by non-technical domain teams. Therefore, you should use the **design-first** approach.

Understanding the basic structure of OAS

The OpenAPI definition structure can be divided into the following sections (all are keyword- and case-sensitive):

- `openapi` (version)
- `info`
- `externalDocs`
- `servers`
- `tags`
- `paths`
- `components`

- ***Coding***

4. Writing business logic for APIs

Overview of the service design

We are going to implement a multilayered architecture that comprises four layers – the presentation layer, application layer, domain layer, and infrastructure layer. Multilayered architecture is a fundamental building block in the architecture style known as **Domain-Driven Design** (**DDD**). Let's have a brief look at each of these layers:

- **Presentation layer**: This layer represents the **User Interface** (**UI**). In the upcoming *Chapter 7*, *Designing a User Interface*, we'll develop the UI for our e-commerce app.
- **Application layer**: The application layer contains the application logic and maintains and coordinates the overall flow of the application. Just to remind you, it only contains the application logic and **not** the business logic. RESTful web services, async APIs, gRPC APIs, and GraphQL APIs are a part of this layer.

  We already covered the REST API interfaces and controllers (implementing REST API interfaces) in *Chapter 3*, *API Specifications and Implementation*, which are part of the application layer. We implemented the controllers for demonstration purposes in the previous chapter. In this chapter, we'll implement a controller extensively to serve real data.
- **Domain layer**: This layer contains the business logic and domain information. It contains the state of the business objects such as Order, Product, and so on. It is responsible for reading/persisting these objects to the infrastructure layer. The domain layer consists of services and repositories too. We'll also be covering these in this chapter.

- **Infrastructure layer**: The infrastructure layer provides support to all other layers. It is responsible for communication such as interaction with the database, message brokers, filesystems, and so on. Spring Boot works as an infrastructure layer and provides support for communication and interaction with both external and internal systems such as databases, message brokers, and so on.

@Repository annotation

Repository components are Java classes marked with the `@Repository` annotation. This is a special Spring component that is used for interacting with databases.

`@Repository` is a general-purpose stereotype that represents both DDD's Repository and the Java **Enterprise Edition** (**EE**) pattern, the **Data Access Object** (**DAO**). Developers and teams should handle Repository objects based on the underlying approach. In DDD, a Repository is a central object that carries references to all the objects and should return the reference of a requested object. We need to have all the required dependencies and configurations in place before we start writing classes marked with `@Repository`.

We'll use the following libraries as database dependencies:

- **H2 database for persisting data**: We are going to use H2's memory instance, however, you can also use a file-based instance.
- **Hibernate Object Relational Mapping** (**ORM**): For database object mapping.
- **Flyway for database migration**: This helps maintain the database and maintains a database changes history that allows rollbacks, version upgrades, and so on.

Adding repositories

All the repository have been added to `https://github.com/PacktPublishing/Modern-API-Development-with-Spring-and-Spring-Boot/blob/main/Chapter04/src/main/java/com/packt/modern/api/repository`.

Repositories are simplest to add for CRUD operations, thanks to Spring Data JPA. You just have to extend the interfaces with default implementations, such as `CrudRepository`, which provides all the CRUD operation implementation such as `save`, `saveAll`, `findById`, `findAll`, `findAllById`, `delete`, and `deleteById`. The `Save(Entity e)` method is used for both create and update entity operations.

Let's create `CartRepository`:

```
public interface CartRepository extends
      CrudRepository<CartEntity, UUID> {

  @Query("select c from CartEntity c join c.user u where u.id =
        :customerId")
  public Optional<CartEntity> findByCustomerId(@
      Param("customerId") UUID customerId);

}
```

https://github.com/PacktPublishing/Modern-API-Development-with-Spring-and-Spring-Boot/blob/main/Chapter04/src/main/java/com/packt/modern/api/repository/CartRepository.java

The `CartRepository` interface extends the `CrudRepository` part of the `org.springframework.data.repository` package. You can also add methods supported by the JPA Query Language marked with the `@Query` annotation (part of the `org.springframework.data.jpa.repository` package). The query inside the `@Query` annotation is written in **Java Persistence Query Language** (**JPQL**). JPQL is very similar to SQL, however, here you used the Java class name mapped to a database table instead of using the actual table name. Therefore, we have used `CartEntity` as the table name instead of `Cart`.

It is also marked as `@Transactional`, which is a special annotation that means that transactions performed by methods in this class will be managed by Spring. It removes all the manual work of adding commits and rollbacks. You can also add this annotation to a specific method inside the class.

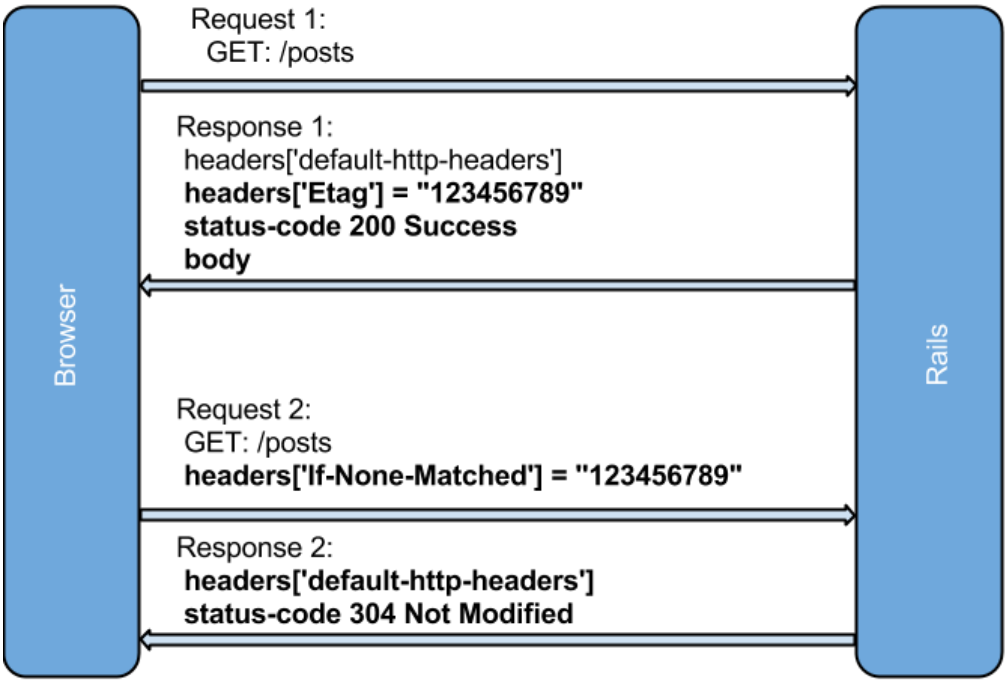
Adding a Service component

The `Service` component is an interface that works between controllers and repositories and is where we'll add the business logic. Though you can directly call repositories from controllers, it is not a good practice as repositories should only be part of the data retrieval and persistence functionalities. Service components also help in sourcing data from various sources, such as databases and other external applications.

Service components are marked with the `@Service` annotation, which is a specialized Spring `@Component` that allows implemented classes to be auto-detected using class-path scanning. Service classes are used for adding business logic. Like Repository, the `Service` object also represents both DDD's Service and Java EE's Business Service Façade pattern. Like Repository, it is also a general-purpose stereotype and can be used according to the underlying approach.

The methods added to `CartService` are directly mapped to serve each of the APIs defined in the `CartController` class. Now, we can implement each of the methods in the `CartServiceImpl` class, which is an implementation of the `CartService` interface. Each method in `CartServiceImpl` makes use of a specific Repository object to carry out the operation:

- ***Etags***



5. Asynchronous API design

5.1. Understanding Reactive Streams

Normal Java code achieves asynchronicity by using **thread pools**. Your web server uses a thread pool for serving requests – it assigns a thread to each incoming request. The application uses the thread pool for database connections.

Each database call **uses a separate thread** and waits for the result. Therefore, each web request and database call use its own thread. However, there is a wait associated with this and therefore, these are blocking calls. The thread waits and utilizes the resources until a response is received back from the database or a response object is written. This is kind of a limitation when you scale as you can only use the resources available to JVM. You overcome this limitation by using a load balancer with other instances of the service, which is a type of horizontal scaling.

In asynchronous calls, threads become free as soon as a call is done and use a callback utility such as JavaScript. When data is available at the source, it pushes the data. Reactive Streams uses the **publisher-subscriber model**, where the source of data, the publisher, pushes the data to the subscriber.

You might be aware that, on the other hand, Node.js uses a single thread to make use of most resources. It is based on an **asynchronous non-blocking** design, known as an event loop.

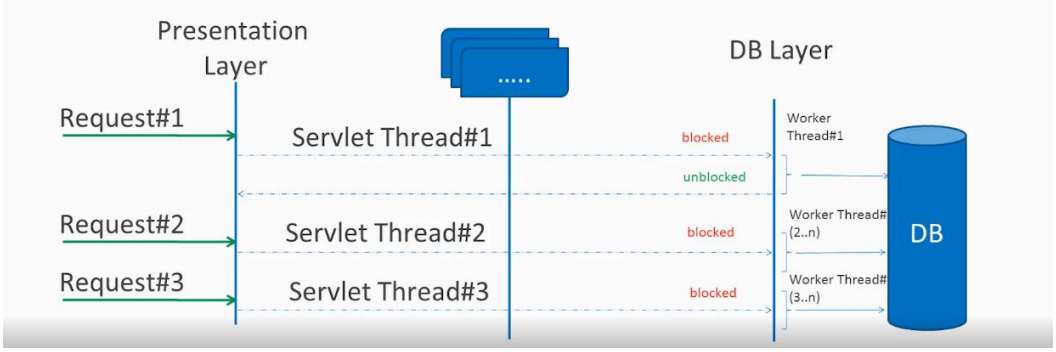
In **Reactive Streams**, streams of data are asynchronous and non-blocking and support **back-pressure**.

- Subcriber
- Publisher
- Subcription
- Processor

► ***Blocking vs non-blocking request processing.***

Blocking request processing:

In traditional MVC applications, when a request come to server, a servlet thread is created. It delegates the request to worker threads for I/O operations such as database access etc. During the time worker threads are busy, servlet thread (request thread) remains in waiting status and thus it is blocked. It is also called synchronous request processing.

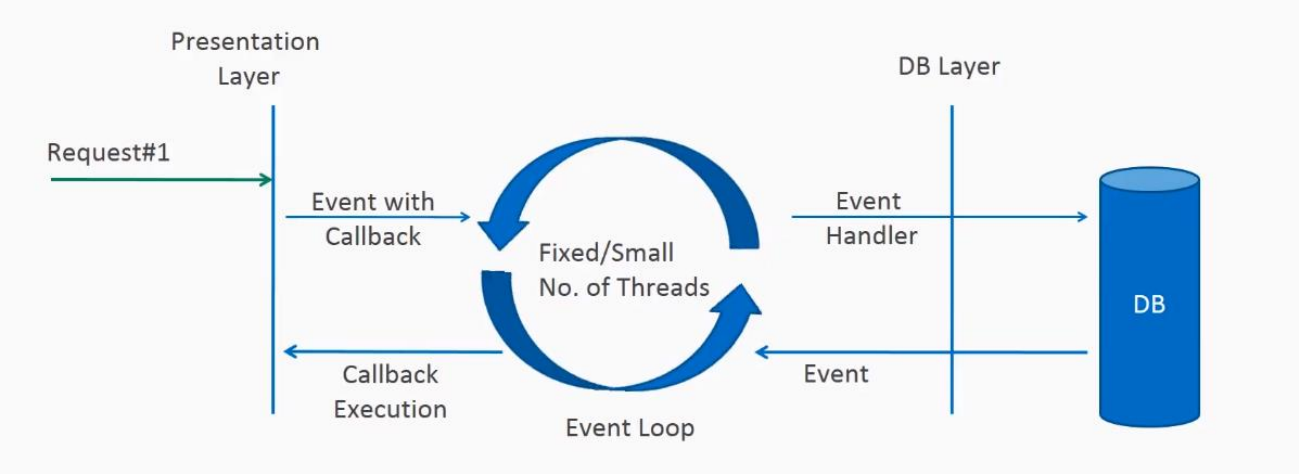


As server can have some **finite number** of request threads, it limits the server capability to process that number of requests at maximum server load. It may **hamper the performance and limit the full utilization of server capability**.

- ***Non blocking request processing***

In non-blocking or asynchronous request processing, no thread is in waiting state. There is generally only one request thread receiving the request.

All incoming requests come with an event handler and call back information. Request thread delegates the incoming requests to a thread pool (generally small number of threads) which delegate the request to its handler function and immediately start processing other incoming requests from request thread.



- ***Reactive stream component***

Publisher: Emits a sequence of events to subscribers according to the demand received from its subscribers. A publisher can serve multiple subscribers.

It has a single method:

```
Publisher.java

public interface Publisher<T>
{
    public void subscribe(Subscriber<? super T> s);
}
```

Subcriber: Receives and processes events emitted by a Publisher. Please note that no notifications will be received until Subscription#request(long) is called to signal the demand.

```
Subscriber.java

public interface Subscriber<T>
{
    public void onSubscribe(Subscription s);
    public void onNext(T t);
    public void onError(Throwable t);
    public void onComplete();
}
```

Subscription: Defines a one-to-one relationship between a Publisher and a Subscriber. It can only be used once by a single Subscriber. It is used to both signal desire for data and cancel demand (and allow resource cleanup).

Subscription.java

```
public interface Subscription<T>
{
    public void request(long n);
    public void cancel();
}
```

Processor: Represents a processing stage consisting of both a Subscriber and a Publisher and obeys the contracts of both.

Processor.java

```
public interface Processor<T, R> extends Subscriber<T>, Publisher<R>
{
}
```

- ***Spring webflux***

Spring WebFlux is parallel version of Spring MVC and supports fully non-blocking reactive streams. It support the back pressure concept and uses **Netty** as inbuilt server to run reactive applications. If you are familiar with Spring MVC programming style, you can easily work on webflux also.

Spring webflux uses project reactor as reactive library. Reactor is a Reactive Streams library and, therefore, all of its operators support non-blocking back pressure. It is developed in close collaboration with Spring.

- **Mono**: Returns 0 or 1 element.

```
Mono<String> mono = Mono.just("Alex");
Mono<String> mono = Mono.empty();
```

- **Flux**: Returns 0…N elements. A Flux can be endless, meaning that it can keep emitting elements forever. Also it can return a sequence of elements and then send a completion notification when it has returned all of its elements.

```
Flux<String> flux = Flux.just("A", "B", "C");
Flux<String> flux = Flux.fromArray(new String[]{"A", "B", "C"});
Flux<String> flux = Flux.fromIterable(Arrays.asList("A", "B", "C"));

//To subscribe call method

flux.subscribe();
```

- ***Server sent events***

  - **Create event source**:

```
const evtSource = new EventSource("ssedemo.php");
```

If the event generator script is hosted on a different origin, a new `EventSource` object should be created with both the URL and an options dictionary. For example, assuming the client script is on `example.com`:

```
const evtSource = new EventSource("//api.example.com/ssedemo.php", { withCredentials: true } );
```

Once you've instantiated your event source, you can begin listening for messages from the server by attaching a handler for the `message` event:

```
evtSource.onmessage = function(event) {
  const newElement = document.createElement("li");
  const eventList = document.getElementById("list");

  newElement.textContent = "message: " + event.data;
  eventList.appendChild(newElement);
}
```

  - **Send events from server**:

5.2. Spring reacrive details

6. dSpring security

You have learned about `DispatcherServlet` in *Chapter 2, Spring Concepts and REST APIs*. This is an interface between a client request and the REST controller. Therefore, if you want to place a logic for token-based authentication and authorization, you would have to do this before the request reaches `DispatcherServlet`. Spring Security libraries provide the servlet pre-filters (as a part of filter chain) that are processed before the request reaches `DispatcherServlet`. A pre-filter is a servlet filter that is processed before it reaches the actual servlet, which in Spring Security's case is `DispatcherServlet`. Similarly, post filters get processed after a request has been processed by the servlet/controller.

There are two ways you can implement token-based (JWT) authentication: by using either `spring-boot-starter-security` or `spring-boot-starter-oauth2-resource-server`. **We are going to use the latter.**

The former contains the following libraries:

- `spring-security-core`
- `spring-security-config`
- `spring-security-web`

`spring-boot-starter-oauth2-resource-server` provides the following, along with all three previously mentioned **Java ARchive** files (**JARs**):

- `spring-security-oauth2-core`
- `spring-security-oauth2-jose`

- `spring-security-oauth2-resource-server`

After apply dependency, **DefaultSecurityFilterChain** is auto configured:

1. `WebAsyncManagerIntegrationFilter`
2. `SecurityContextPersistenceFilter`
3. `HeaderWriterFilter`
4. `CorsFilter`
5. `CsrfFilter`
6. `LogoutFilter`
7. `BearerTokenAuthenticationFilter`
8. `RequestCacheAwareFilter`
9. `SecurityContextHolderAwareRequestFilter`
10. `AnonymousAuthenticationFilter`
11. `SessionManagementFilter`
12. `ExceptionTranslationFilter`
13. `FilterSecurityInterceptor`
14. Finally reaches the controller

Username/password authentication flow using filters

Authentication with a username/password works as shown in the following diagram. If a user submits a valid username/password combination, the call succeeds and the user gets a token with a `200 OK` status code (a successful response). If a call fails due to an invalid username/password combination, you'll receive a response with a `401 Unauthorized` status code:

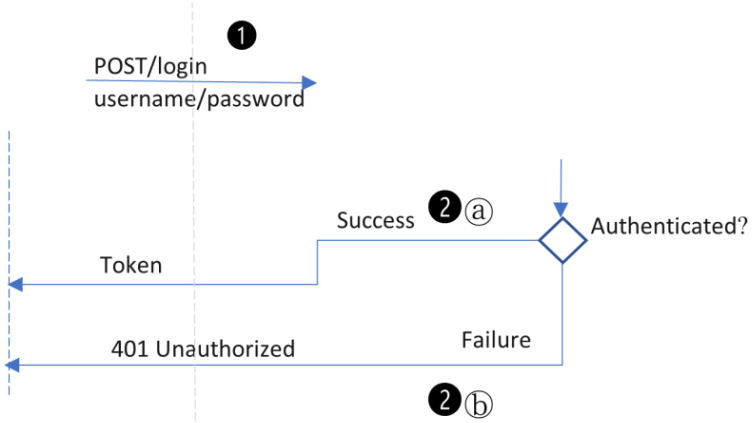


Figure 6.1 – Username/password authentication flow

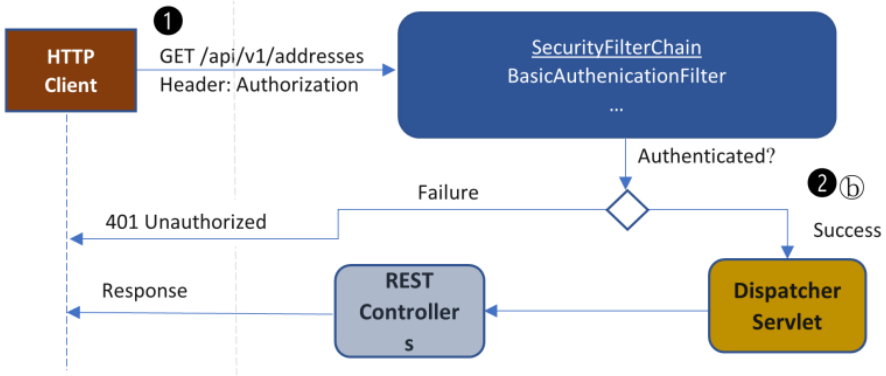


Figure 6.2 – Token authorization flow using BasicAuthenticationFilter

Adding the required Gradle dependencies

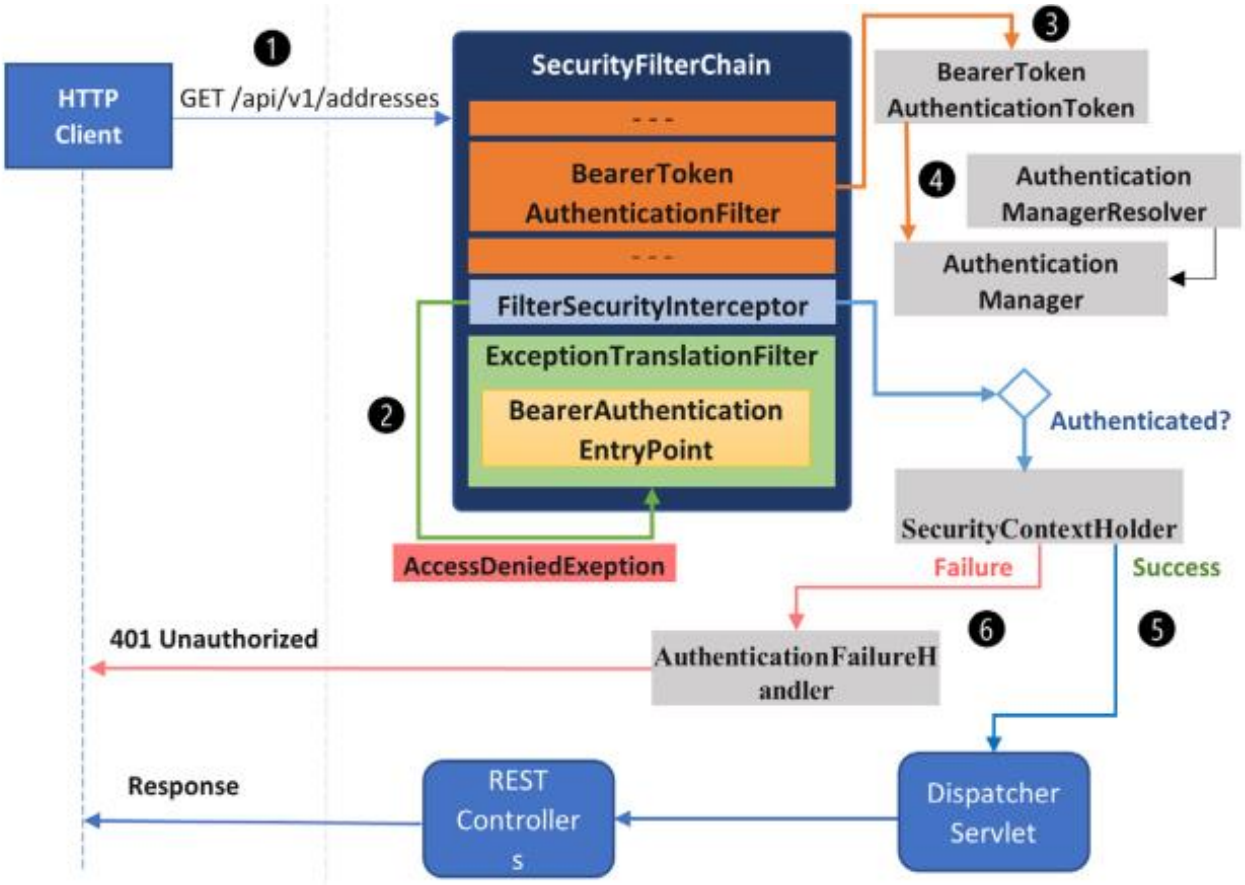
Let's add the following dependencies into the `build.gradle` file, as shown next:

```
implementation 'org.springframework.boot:spring-boot-starter-oauth2-resource-server'
implementation 'com.auth0:java-jwt:3.12.0'
```

https://github.com/PacktPublishing/Modern-API-Development-with-Spring-and-Spring-Boot/blob/main/Chapter06/build.gradle

The Spring Boot Starter OAuth 2.0 resource server dependency would add the following JARs:

- `spring-security-core`
- `spring-security-config`
- `spring-security-web`
- `spring-security-oauth2-core`
- `spring-security-oauth2-jose`
- `spring-security-oauth2-resource-server`



1. The client sends a `GET` HTTP request to `/api/v1/addresses`.
2. `BearerTokenAuthenciationFilter` comes into action. If the request doesn't contain the `Authorization` header then `BearerTokenAuthenticationFilter` does not authenticate the request since it did not find the bearer token. It passes the call to `FilterSecurityInterceptor`, which does the authorization. It throws an `AccessDeniedException` exception (marked as **2** in *Figure 6.3*). `ExceptionTranslationFilter` springs into action. Control is moved to `BearerTokenAuthenticationEntryPoint`, which responds with a `401 Unauthorized` status and a `WWW-Authenticate` header with a `Bearer` value. If the client receives a `WWW-Authenticate` header with a `Bearer` value in response, it means it has to retry with the `Authorization` header that holds the valid bearer token. At this stage, the request cache is `NullRequestCache` (that is, empty) due to security reasons because the client can replay the request.

3. Let's assume the HTTP request contains an `Authorization` header. It extracts the `Authorization` header from the HTTP request and, apparently, the token from the `Authorization` header. It creates an instance of `BearerTokenAuthenticationToken` using the token value. `BearerTokenAuthenticationToken` is a type of `AbstractAuthenticationToken` class that implements an `Authentication` interface representing the token/principal for the authenticated request.
4. The HTTP request is passed to `AuthenticationManagerResolver`, which provides the `AuthenticationManager` based on the configuration. `AuthenticationManager` verifies the `BearerTokenAuthenticationToken` token.

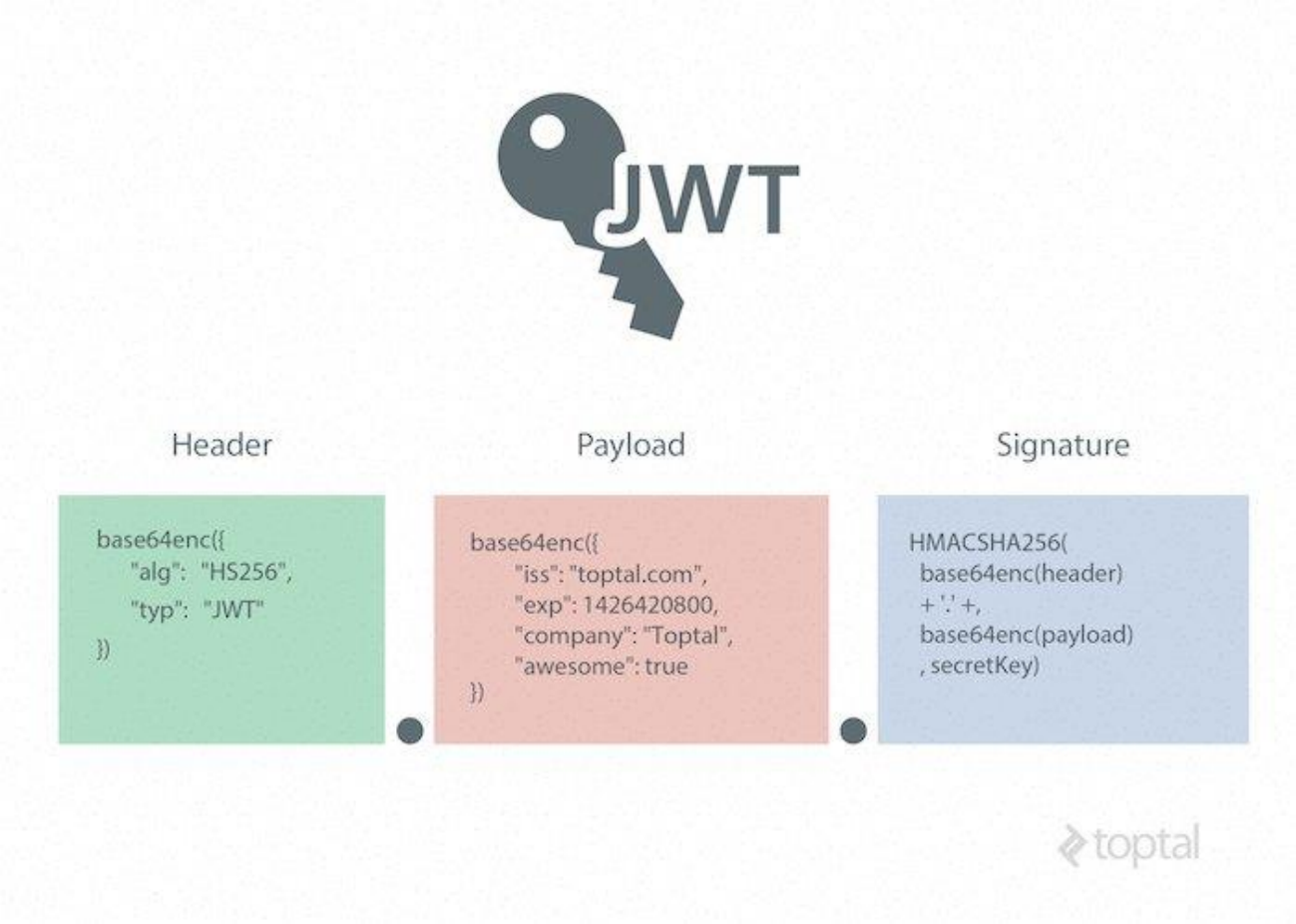
5. If authentication is successful, then `Authentication` is set on the `SecurityContext` instance. This instance is then passed to `SecurityContextHolder.setContext()`. The request is passed to the remaining filters for processing and then routes to `DispatcherServlet` and then, finally, to `AddressController`.

6. If authentication fails, then `SecurityContextHolder.clearContext()` is called to clear the context value. `ExceptionTranslationFilter` springs into action. Control is moved to `BearerTokenAuthenticationEntryPoint`, which responds with a `401 Unauthorized` status and a `WWW-Authenticate` header with a value that contains the appropriate error message, such as `Bearer error="invalid_token", error_description="An error occurred while attempting to decode the Jwt: Jwt expired at 2020-12-14T17:23:30Z", error_uri="https://tools.ietf.org/html/rfc6750#section-3.1"`.

JWT structure

A JWT is an encoded string such as `aaa.bbb.ccc`, consisting of the following three parts separated by dots (`.`):

- Header
- Payload
- Signature



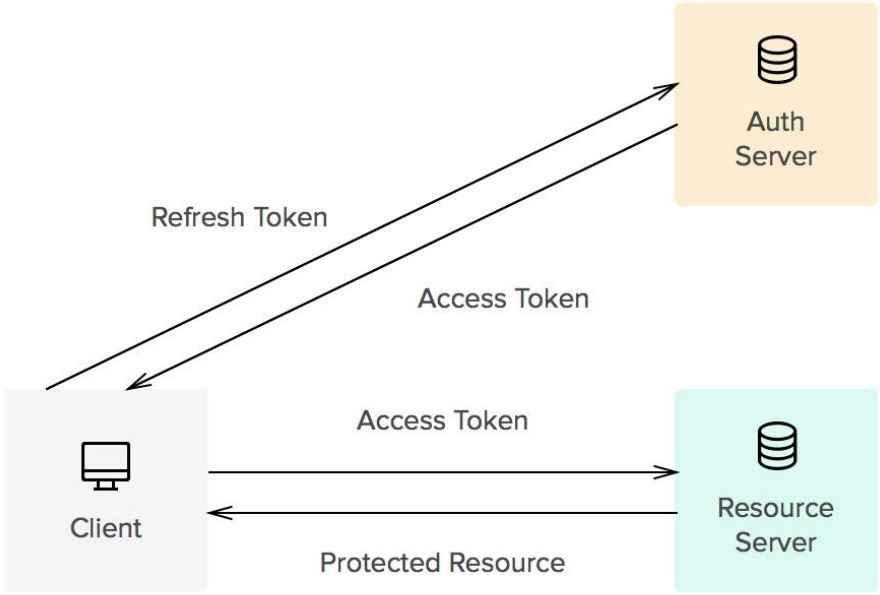
Securing REST APIs with JWT

In this section, you'll secure the REST endpoints exposed in *Chapter 4, Writing Business Logic for APIs*. Therefore, we'll use the code from *Chapter 4, Writing Business Logic for APIs* and enhance it to secure the APIs.

The REST APIs should be protected with the following features:

- No secure API should be accessed without JWT.
- A JWT can be generated using sign-in/sign-up or a refresh token.
- A JWT and a refresh token should only be provided for a valid user's username/password combination or a valid user sign-up.
- The password should be stored in encoded format using a `bcrypt` strong hashing function.
- The JWT should be signed with **RSA** (for **Rivest, Shamir, Adleman**) keys with a strong algorithm.
- Claims in the payload should not store sensitive or secured information. If they do, then these should be encrypted.
- You should be able to authorize API access for certain roles.

- ***Refresh token***



Spring Security Step by Step:

1. **Adding Gradle Dependency**

Adding the required Gradle dependencies

Let's add the following dependencies into the `build.gradle` file, as shown next:

```
implementation 'org.springframework.boot:spring-boot-starter-oauth2-resource-server'
implementation 'com.auth0:java-jwt:3.12.0'
```

2. **Add UserDetailsService**
3. **Extend WebSecurityConfiguerAdapter**
4. **Configure (http) overide**
5. **Adding end point for API:**
    a. **Signup end point:** api/v1/users: request user info (**User** schema), return 201 SignInUser object, save user to database,

```xml
<?xml version="1.0" encoding="UTF-8"?>
<SignedInUser>
        <refreshToken>string</refreshToken>
        <accessToken>string</accessToken>
        <username>string</username>
        <userId>string</userId>
</SignedInUser>
```

   b. **Signin end point:** api/v1/auth/token: request (**SignInReq** schema), return SignedInUser
   c. **Sign out end point:** api/v1/auth/token delete, (input refresh token)
   d. **Refresh token end point**: /api/v1/auth/token/refresh, input refresh token, return **SignInUser.**

6. Create Utils Constant Class
7. Create bean **JWT manager**:
   a. Input RSA private key, RSA public key
   b. Provide **create** function.
8. Gen public private key: `keytool -genkey -alias "jwt-sign-key" -keyalg RSA -keystore jwt-keystore.jks -keysize 4096`
9. Load properties of keystore to project

```
app.security.jwt.keystore-location=jwt-keystore.jks
app.security.jwt.keystore-password=password
app.security.jwt.key-alias=jwt-sign-key
app.security.jwt.private-key-passphrase=password
```

10. Configure keystore:

```java
@Value("jwt-keystore.jks")
private String keyStorePath;

@Value("ows1234a")
private String keyStorePassword;

@Value("jwt-sign-key")
private String keyAlias;

@Value("password")
private String privateKeyPassphrase;
```

11. Create keyStore, , rsaPrivateKey.
12. Create **JwtDecoder** bean rsaPublicKey.
13. Add new **USER_TOKEN** entity for store user token.
14. Add new **UserTokenRepository**: **findByRefreshToken**, **deleteByUserId**
15. Enchange **UserService** class:

   a. **FindUserByUsername**
   b. **createUser (after signup) createUserWithRefreshToken, createSignedUser.**
   c. **getSignedInUser: return user info with refresh token, access token.**
   d. **getAccessToken**
   e. **removeRefreshToken.**

16. Enhance **UserRepository** class add new method **findByUsername** (String username).

17. Add bean password Encoder.

18. Add authController:

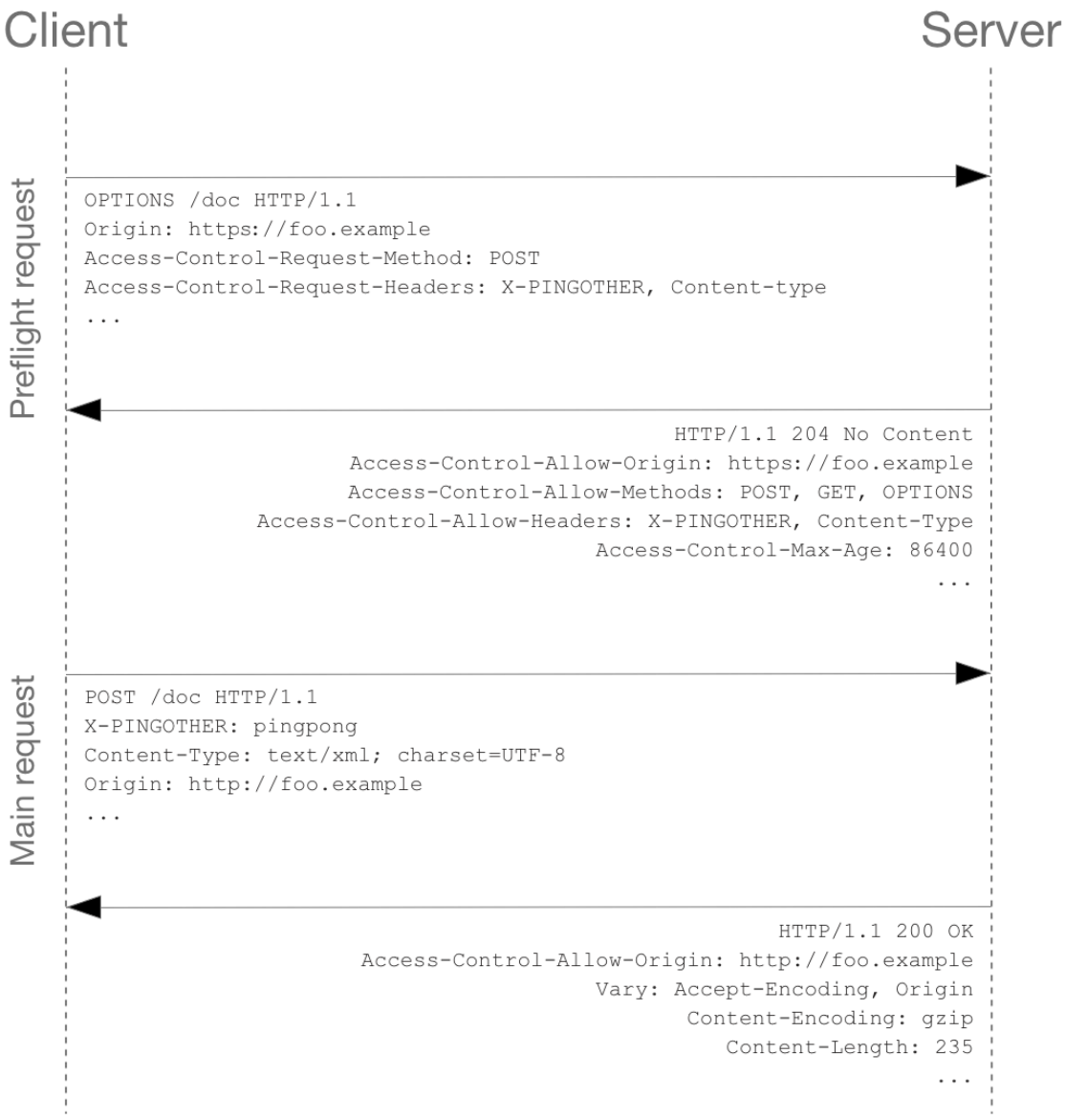
   a. Get access token, return Token
   b. Sign in: return SigninUser
   c. Sign out: delete refresh token
   d. Sign up: return SignedUser

19. Configure WebSecurityAdapter:

   a. httpBasic().disable()
   b. formLogin().disable()
   c. Enable JWT support Oauth 2.0 (**oauth2ResourceServer(OAuth2ResourceServerConfigurer::jwt**).
   d. Enalble stateless policy.
   e. Enable **X-frame-option = SAMEORIGIN** for allow, other page render it inside.

20. Configure **CORS**:

Before make a request, browser intialize a pre-flight request and and check **Access-Control-Allow-Origin**: if server response with the * or the domain of the request, client can be proceed to make next request.

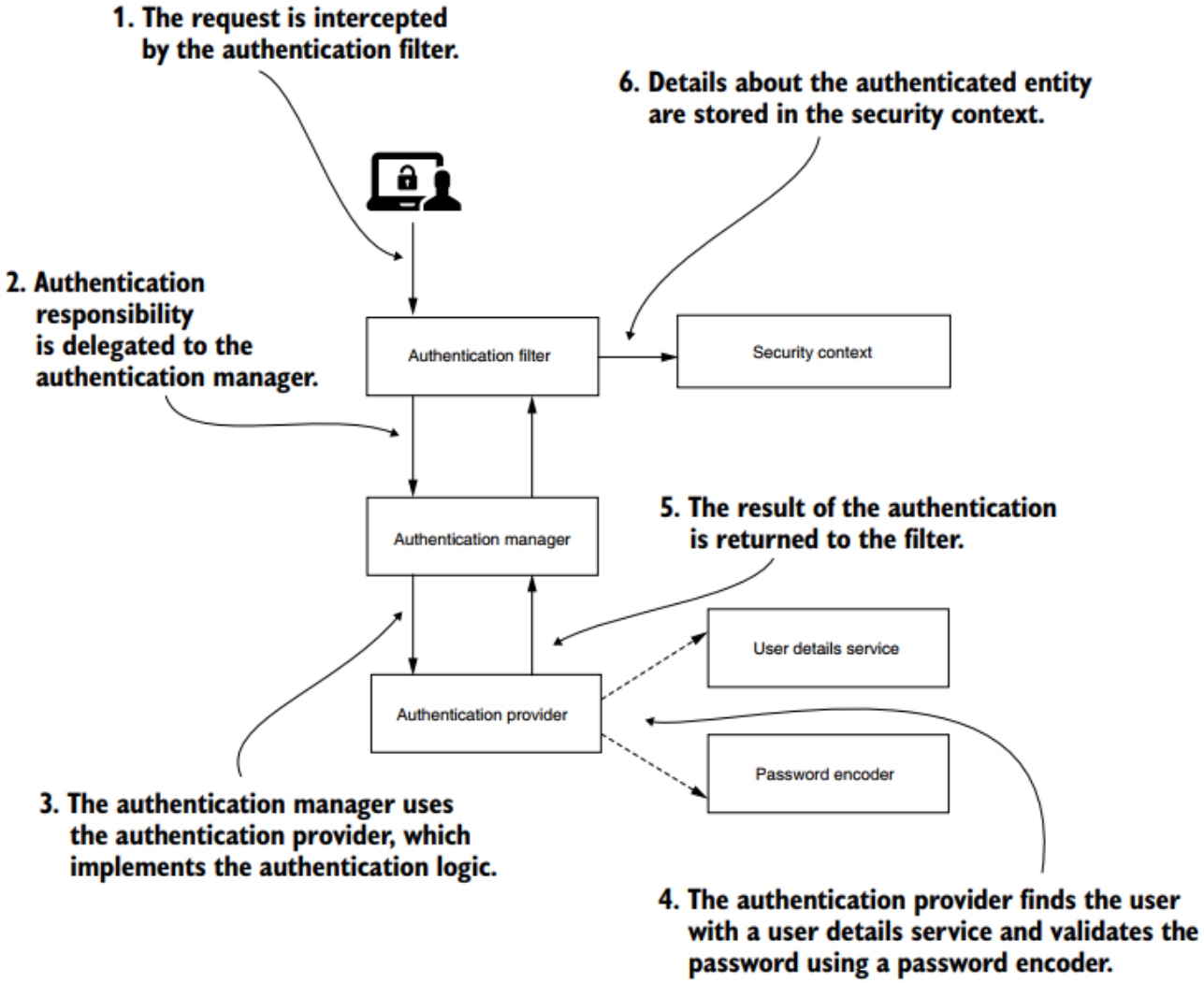


21. CSRF

22. Create Enum ROLE implement GrantedAuthority:

- **a. Customer**
- **b. Admin**
- **c. Support**

6.1. Spring security fundamentals



Custom **PasswordEncoder** and **UserDetailsService**:

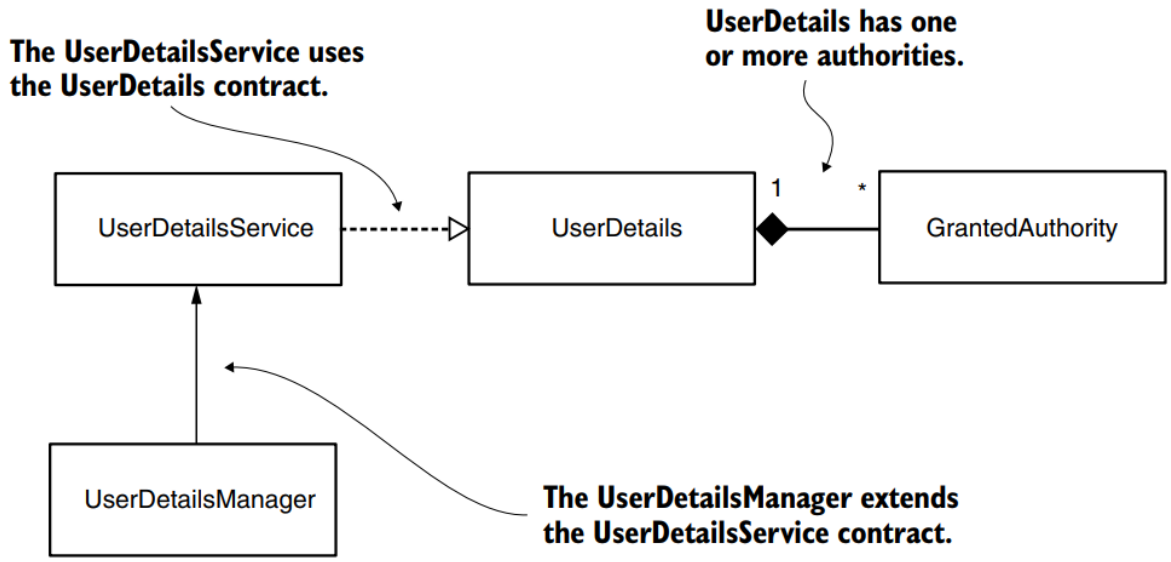
```java
@Configuration
public class UserManagementConfig {

    @Bean
    UserDetailsService userDetailsService(){
        var authenticationManager = new InMemoryUserDetailsManager();
        var user :UserDetails = User.withUsername("user")
                                    .password("user")
                                    .authorities("READ")
                                    .build();
        authenticationManager.createUser(user);
        return authenticationManager;
    }

    @Bean
    PasswordEncoder passwordEncoder(){
        return NoOpPasswordEncoder.getInstance();
```

Custome **AuthenticationProvider**:

```java
@Component
public class CustomAuthenticationProvider implements AuthenticationProvider {

    @Override
    public Authentication authenticate(Authentication authentication) throws AuthenticationException {
        String user = authentication.getName();
        String password = String.valueOf(authentication.getCredentials());

        if (user.equals("user") && password.equals("user")){
            return new UsernamePasswordAuthenticationToken(user, password, Collections.emptyList());
        } else {
            throw new AuthenticationCredentialsNotFoundException("Error in authentication");
        }
    }

    @Override
    public boolean supports(Class<?> authenticationType) {
        return UsernamePasswordAuthenticationToken.class
                .isAssignableFrom(authenticationType);
    }
```

6.1.1. Manage User

- **UserDetails**, which describes the user for Spring Security.
- **GrantedAuthority**, which allows us to define actions that the user can execute.
- **UserDetailsManager**, which extends the **UserDetailsService** contract. Beyond the inherited behavior, it also describes actions like **creating a user** and **modifying** or **deleting a user's password**.



UserDetails contract:

```
public interface UserDetails extends Serializable {
  String getUsername();
  String getPassword();
  Collection<? extends GrantedAuthority>
  ➥getAuthorities();
  boolean isAccountNonExpired();
  boolean isAccountNonLocked();
  boolean isCredentialsNonExpired();
  boolean isEnabled();
}
```

These methods return the user credentials.

Returns the actions that the app allows the user to do as a collection of GrantedAuthority instances

These four methods enable or disable the account for different reasons.

GrantAuthority Contract:

```
public interface GrantedAuthority extends Serializable {
    String getAuthority();
}
```

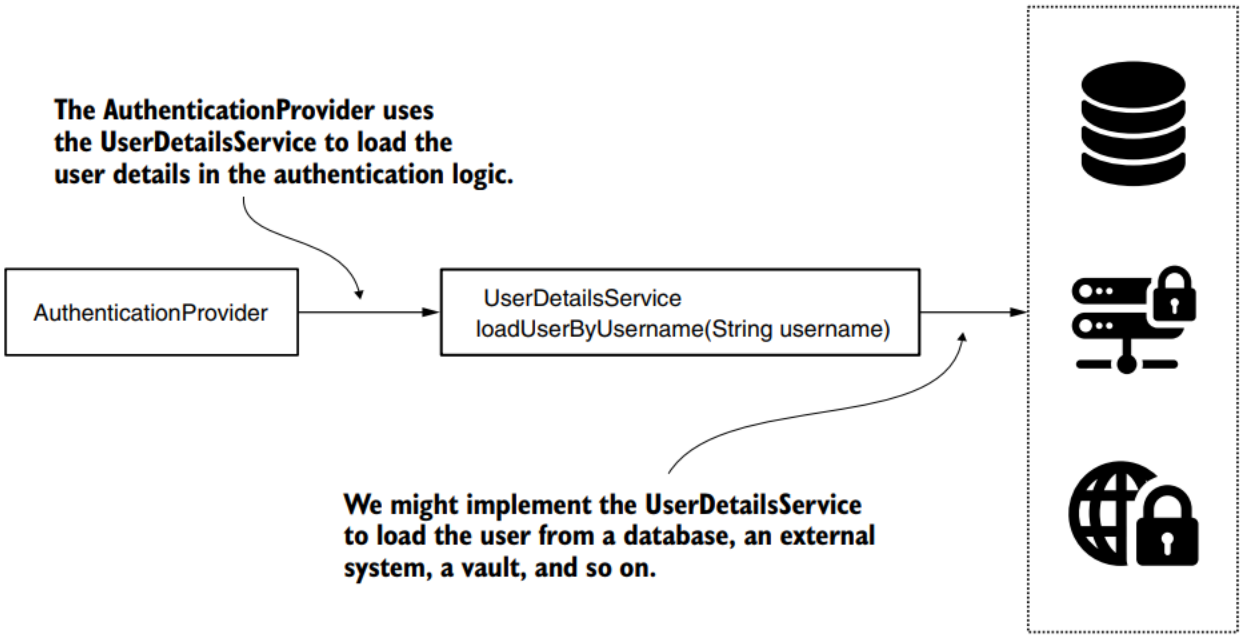
```
GrantedAuthority g1 = () -> "READ";
GrantedAuthority g2 = new SimpleGrantedAuthority("READ");
```

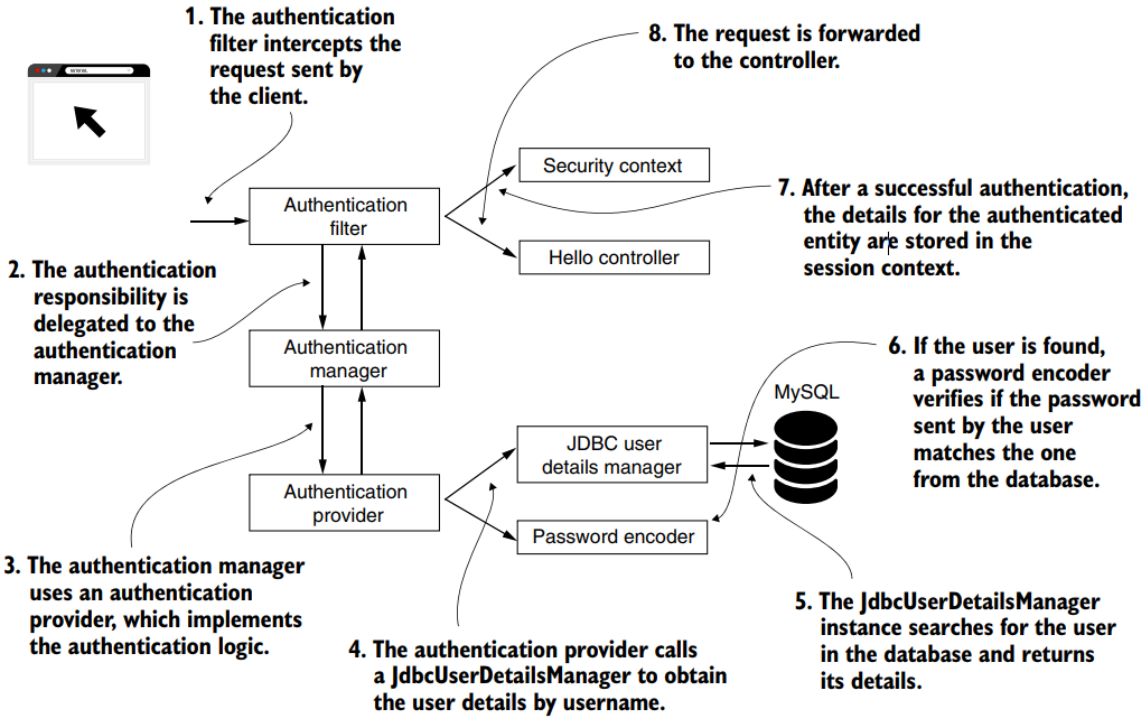
Create a decorate class over **User** Entity:

```
public class SecurityUser implements UserDetails {

  private final User user;

  public SecurityUser(User user) {
    this.user = user;
  }

  @Override
  public String getUsername() {
    return user.getUsername();
  }

  @Override
  public String getPassword() {
    return user.getPassword();
  }

  @Override
  public Collection<? extends GrantedAuthority> getAuthorities() {
    return List.of(() -> user.getAuthority());
  }

  // Omitted code

}
```

UserDetailsService Contract:

```
public interface UserDetailsService {

  UserDetails loadUserByUsername(String username)
      throws UsernameNotFoundException;
}
```





Use JdbcUsersDetailsManager as UserDetailsService:

```java
@Configuration
public class UserManagementConfig {

    @Bean
    public UserDetailsService userDetailsService(DataSource dataSource){
        return new JdbcUserDetailsManager(dataSource);
    }
```

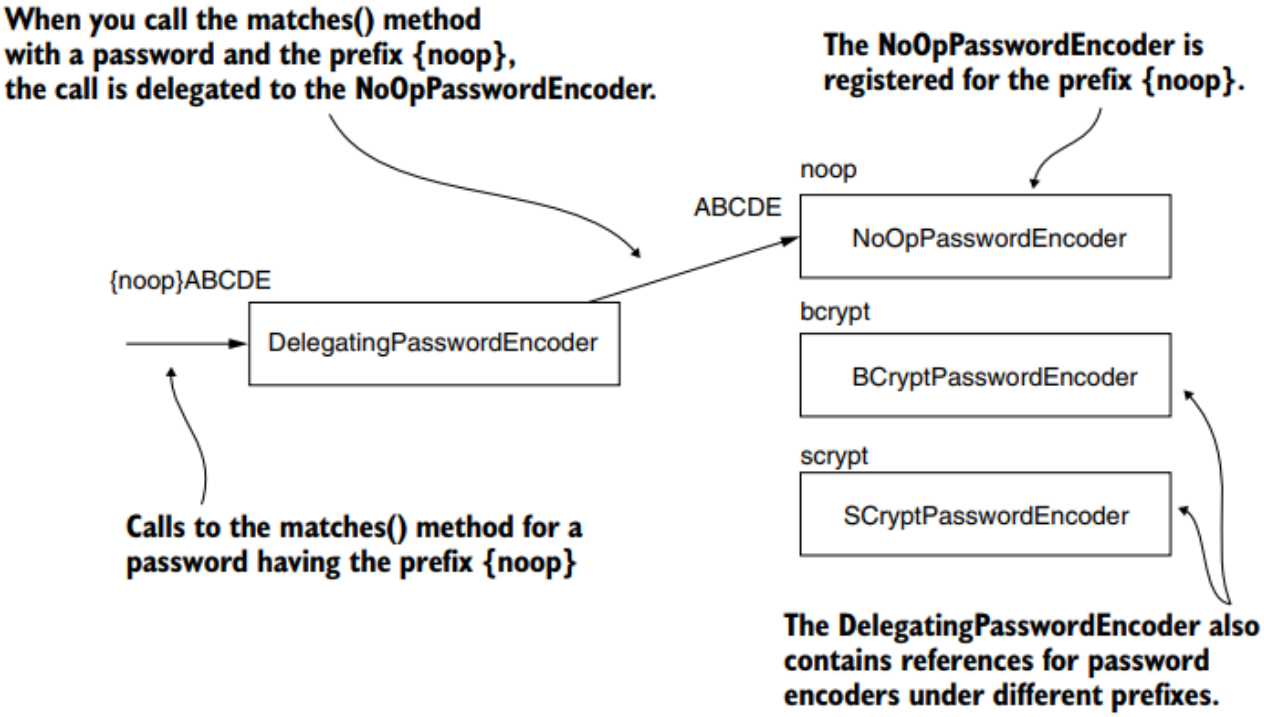
```
spring.h2.console.enabled=true
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=password
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
```

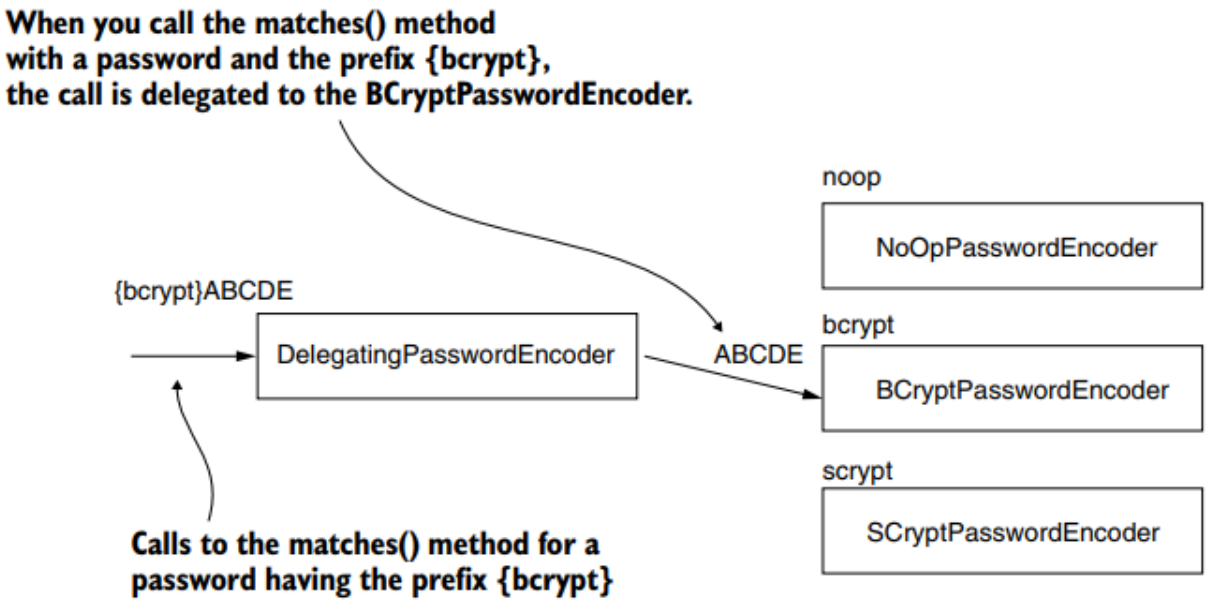
```
@Bean
public UserDetailsService userDetailsService(DataSource dataSource) {

    String usersByUsernameQuery = "select username, password, enabled from users
                                   where username = ?";

    String authsByUserQuery = "select username, authority from spring.authorities
                               where username = ?";

    var userDetailsManager = new JdbcUserDetailsManager(dataSource);

    userDetailsManager.setUsersByUsernameQuery(usersByUsernameQuery);

    userDetailsManager.setAuthoritiesByUsernameQuery(authsByUserQuery);

    return userDetailsManager;
}
```

Password encoder:

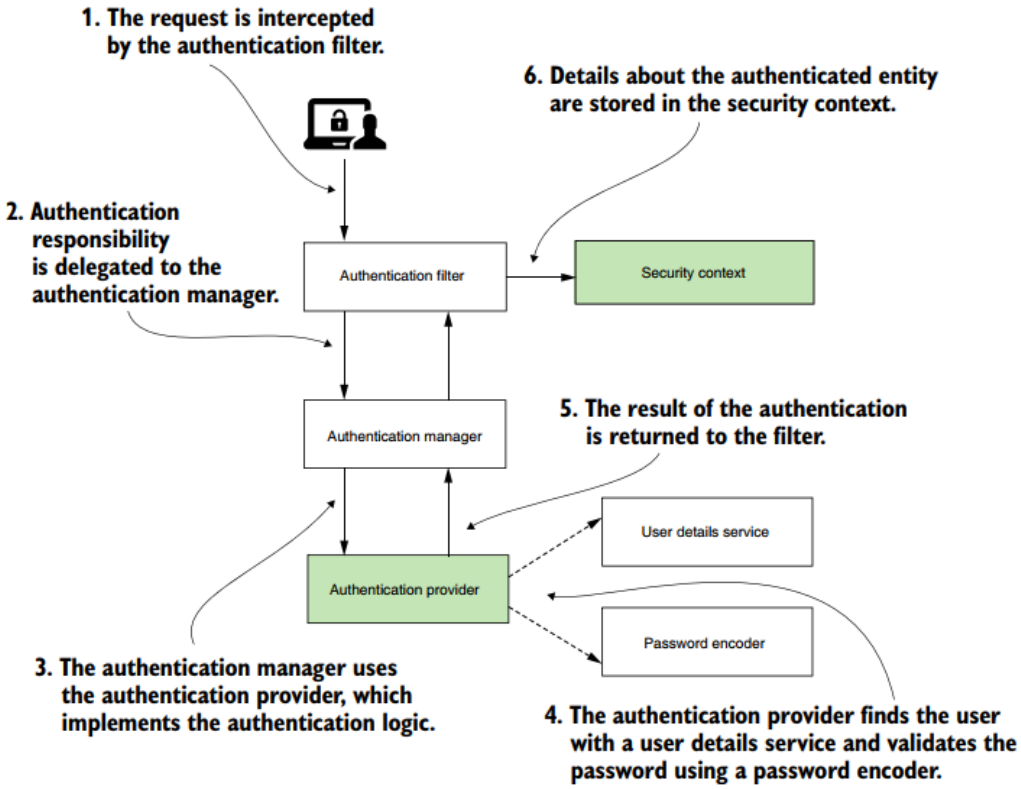
- **NoOpPasswordEncoder**—Doesn't encode the password but keeps it in cleartext. We use this implementation only for examples. Because it doesn't hash the password, you should never use it in a real-world scenario.
- **StandardPasswordEncoder**—Uses SHA-256 to hash the password. This implementation is now deprecated, and you shouldn't use it for your new implementations. The reason why it's deprecated is that it uses a hashing algorithm that we don't consider strong enough anymore, but you might still find this implementation used in existing applications.
- **Pbkdf2PasswordEncoder**—Uses the password-based key derivation function (PBKDF2).
- **BCryptPasswordEncoder**—Uses a bcrypt strong hashing function to encode the password.
- **SCryptPasswordEncoder**—Uses a scrypt hashing function to encode the password

Multiple encoding strategies with DelegatingPasswordEncoder:

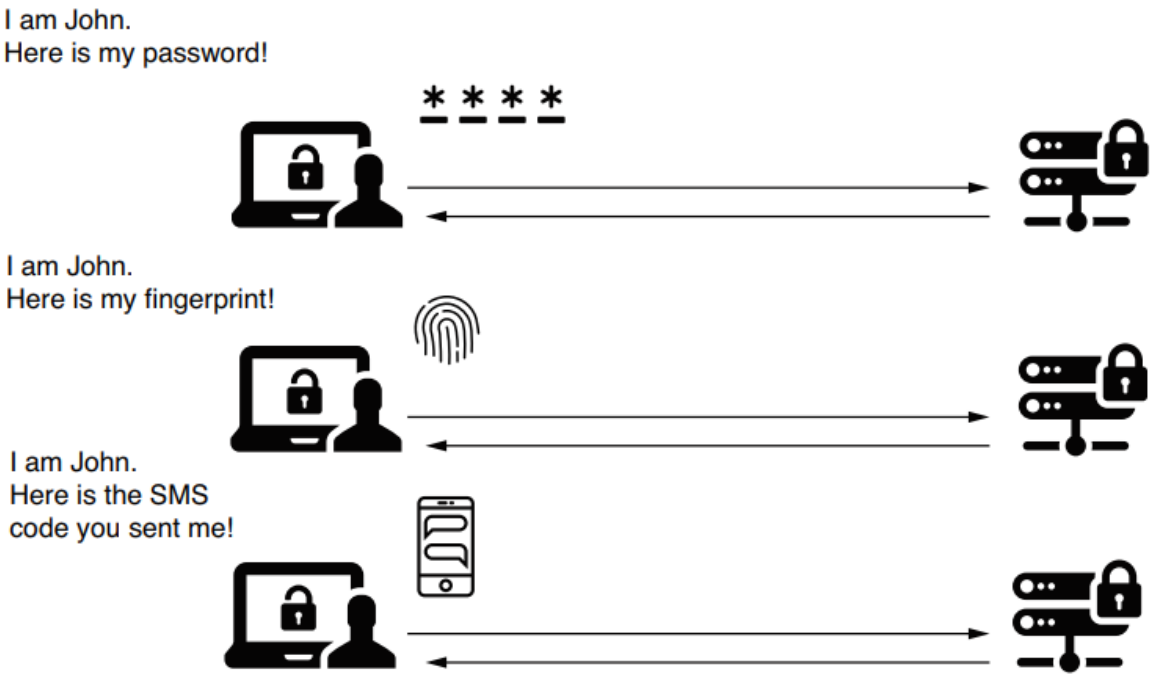

When you call the matches() method with a password and the prefix {noop}, the call is delegated to the NoOpPasswordEncoder.
The NoOpPasswordEncoder is registered for the prefix {noop}.
noop
ABCDE
NoOpPasswordEncoder
{noop}ABCDE
DelegatingPasswordEncoder
bcrypt
BCryptPasswordEncoder
scrypt
SCryptPasswordEncoder
Calls to the matches() method for a password having the prefix {noop}
The DelegatingPasswordEncoder also contains references for password encoders under different prefixes.

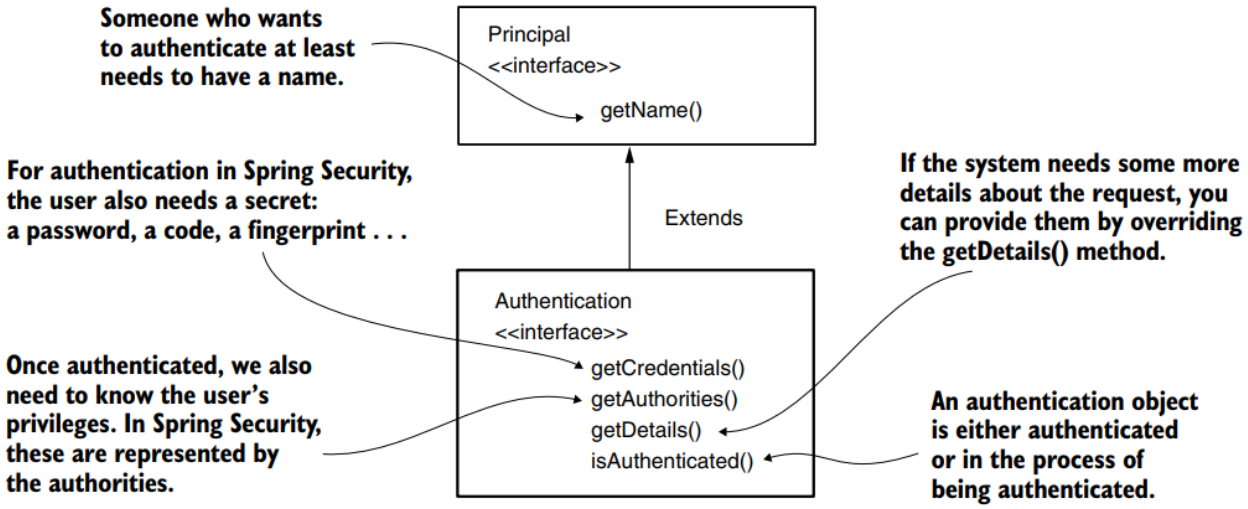

When you call the matches() method with a password and the prefix {bcrypt}, the call is delegated to the BCryptPasswordEncoder.
noop
NoOpPasswordEncoder
{bcrypt}ABCDE
DelegatingPasswordEncoder
ABCDE
bcrypt
BCryptPasswordEncoder
scrypt
SCryptPasswordEncoder
Calls to the matches() method for a password having the prefix {bcrypt}

Contract	Description
`UserDetails`	Represents the user as seen by Spring Security.
`GrantedAuthority`	Defines an action within the purpose of the application that is allowable to the user (for example, read, write, delete, etc.).
`UserDetailsService`	Represents the object used to retrieve user details by username.
`UserDetailsManager`	A more particular contract for `UserDetailsService`. Besides retrieving the user by username, it can also be used to mutate a collection of users or a specific user.
`PasswordEncoder`	Specifies how the password is encrypted or hashed and how to check if a given encoded string matches a plaintext password.




```java
public interface AuthenticationManager {
    Authentication authenticate(Authentication var1)
    throws AuthenticationException;
}

public interface AuthenticationProvider {
    Authentication authenticate(Authentication var1)
    throws AuthenticationException;

    boolean supports(Class<?> var1);
}
```

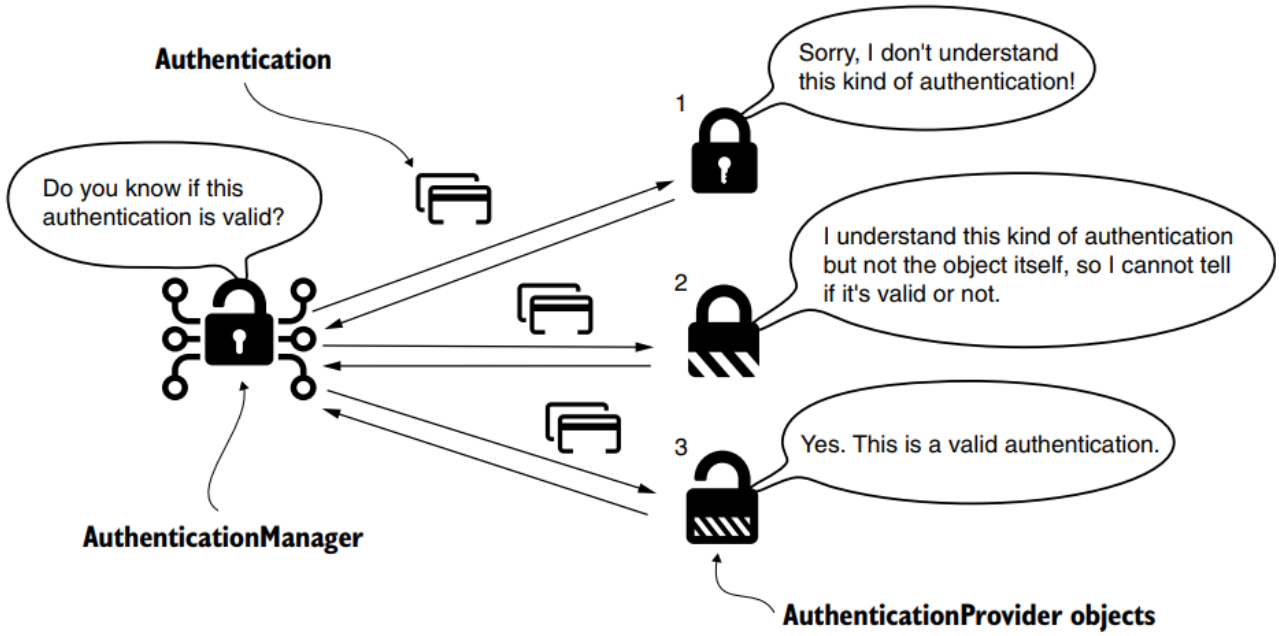


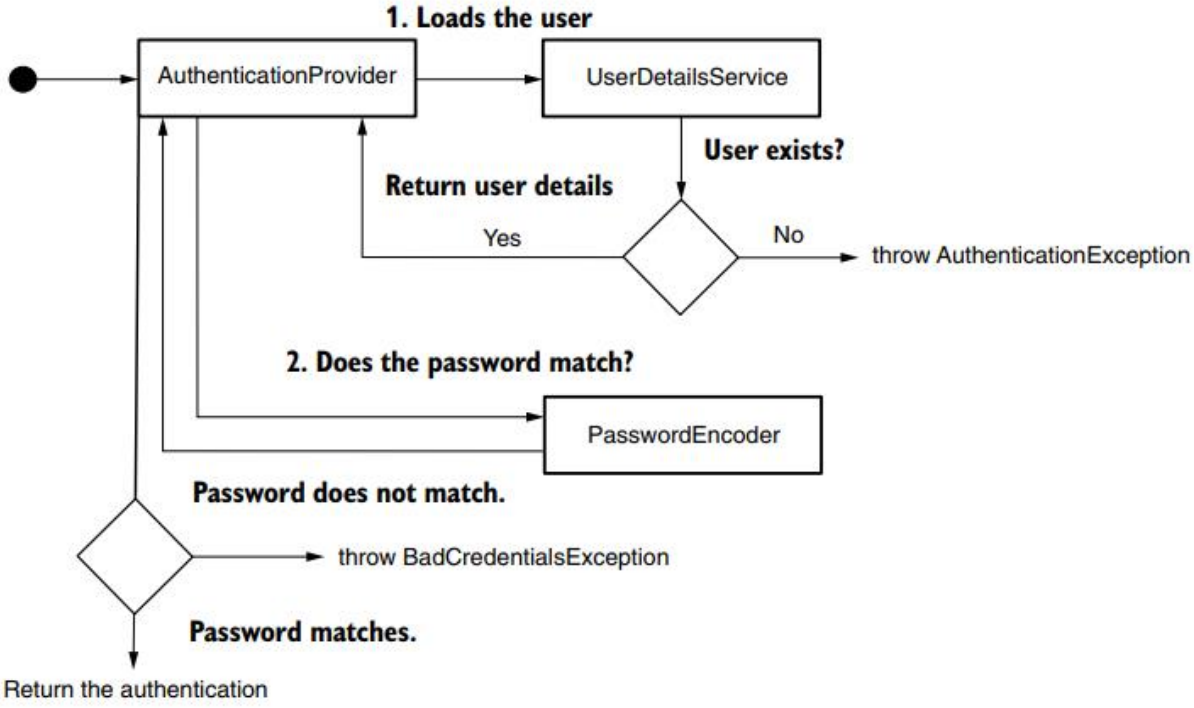


The **AuthenticationProvider** in **Spring Security** takes care of the authentication logic. The default implementation of the **AuthenticationProvider** interface delegates the responsibility of finding the system's user to a **UserDetailsService**.

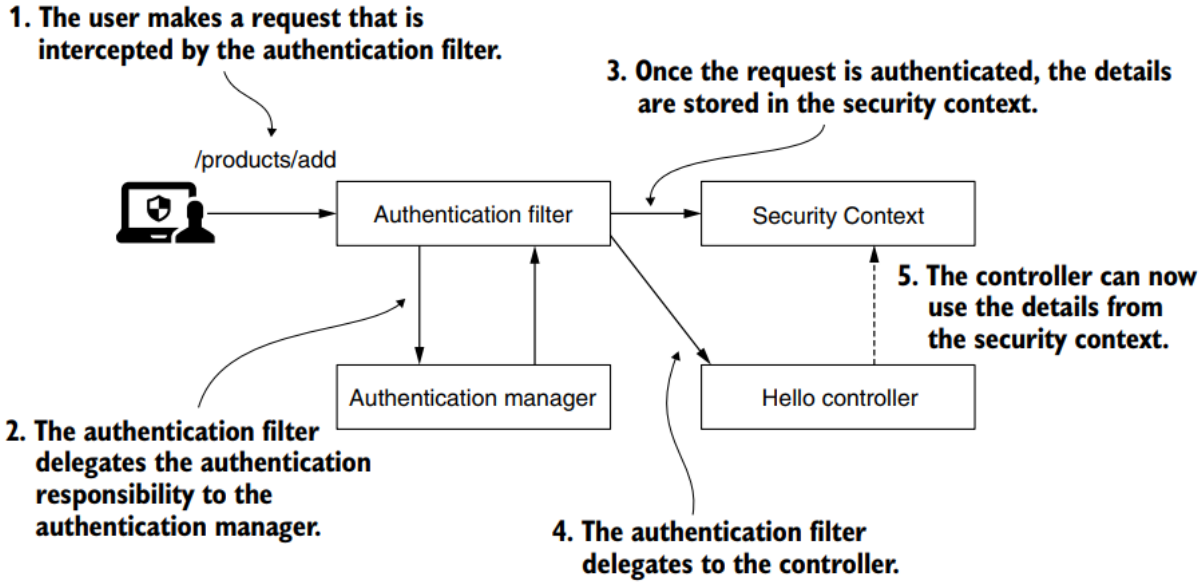
It uses the **PasswordEncoder** as well for password management in the process of authentication.

An analogy of how the authentication manager and authentication provider work together to validate or invalidate an authentication request is having a more complex lock for your door. The lock itself is the **authentication manager** it delegated the responsibility to **authentication provider**.





Once the **AuthenticationManager** completes the authentication process successfully, it stores the **Authentication** instance for the rest of the request. The instance storing the **Authentication** object is called the **security context**.



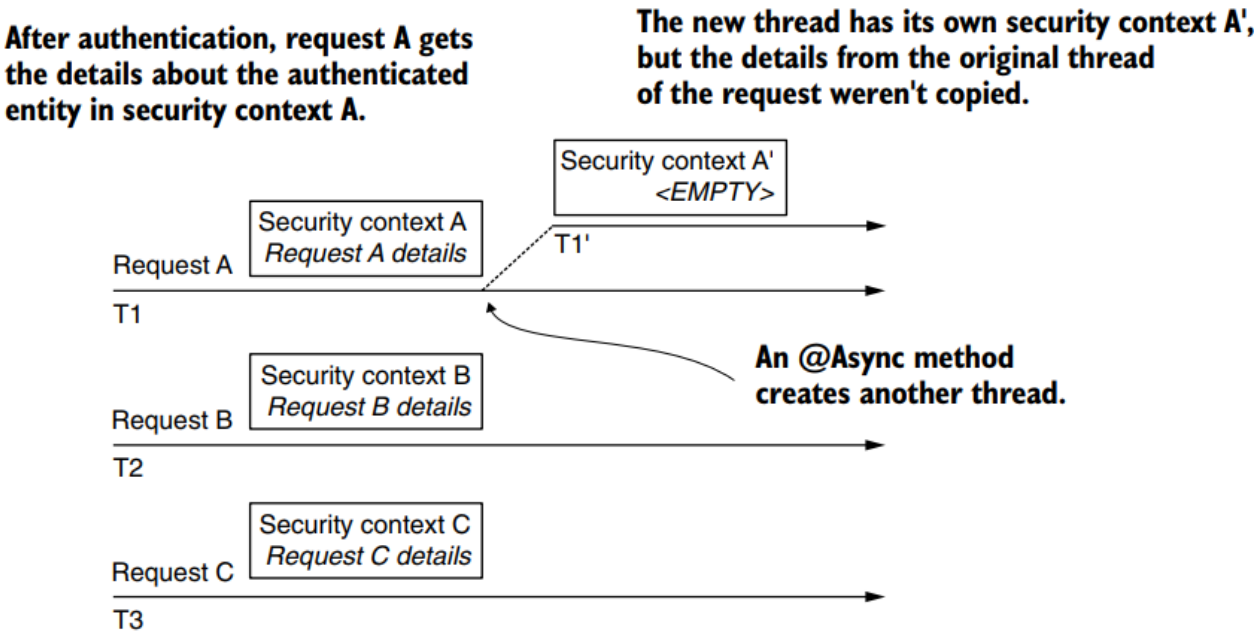
SecurityContext use **SecurityContextHolder** for storing authentication object:

Listing 5.6 The `SecurityContext` interface

```
public interface SecurityContext extends Serializable {

  Authentication getAuthentication();
  void setAuthentication(Authentication authentication);
}
```

```
public class SecurityContextHolder {
    public static final String MODE_THREADLOCAL = "MODE_THREADLOCAL";
    public static final String MODE_INHERITABLETHREADLOCAL = "MODE_INHERITABLETHREADLOCAL";
    public static final String MODE_GLOBAL = "MODE_GLOBAL";
    public static final String SYSTEM_PROPERTY = "spring.security.strategy";
```

- **THREAD_LOCAL**: thread per request application.
- **INHERITABLETHREADLOCAL:** for async call, allow other thread copy context.
- **MODE_GLOBAL:** make all threads see context each other.

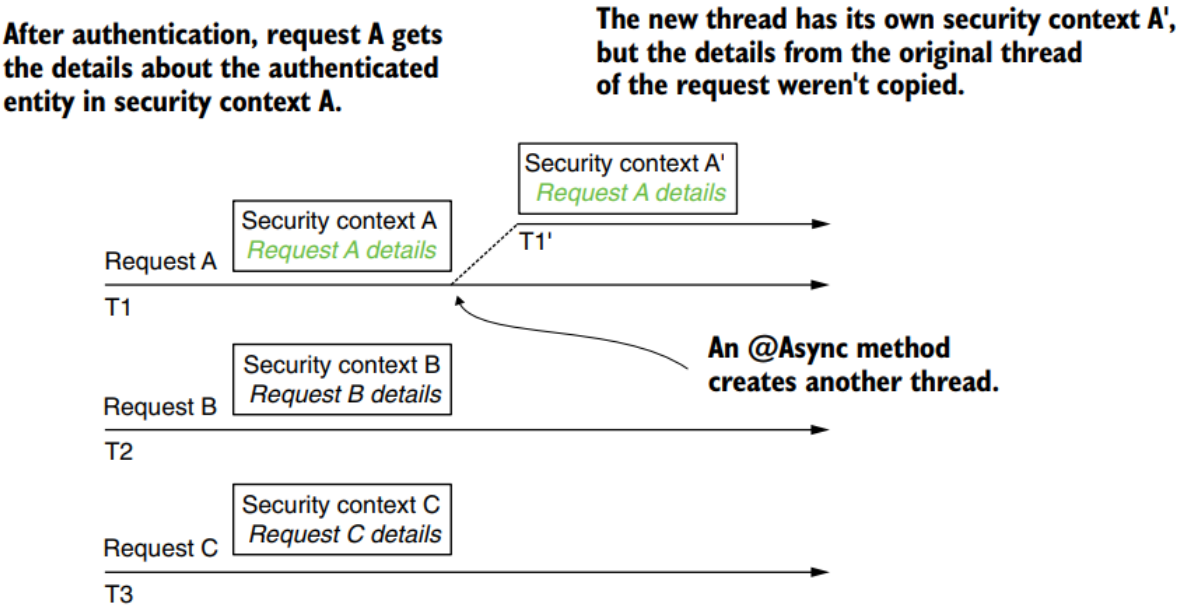



```
@GetMapping("/hello")
public String hello() {
  SecurityContext context = SecurityContextHolder.getContext();
  Authentication a = context.getAuthentication();

  return "Hello, " + a.getName() + "!";
}
```

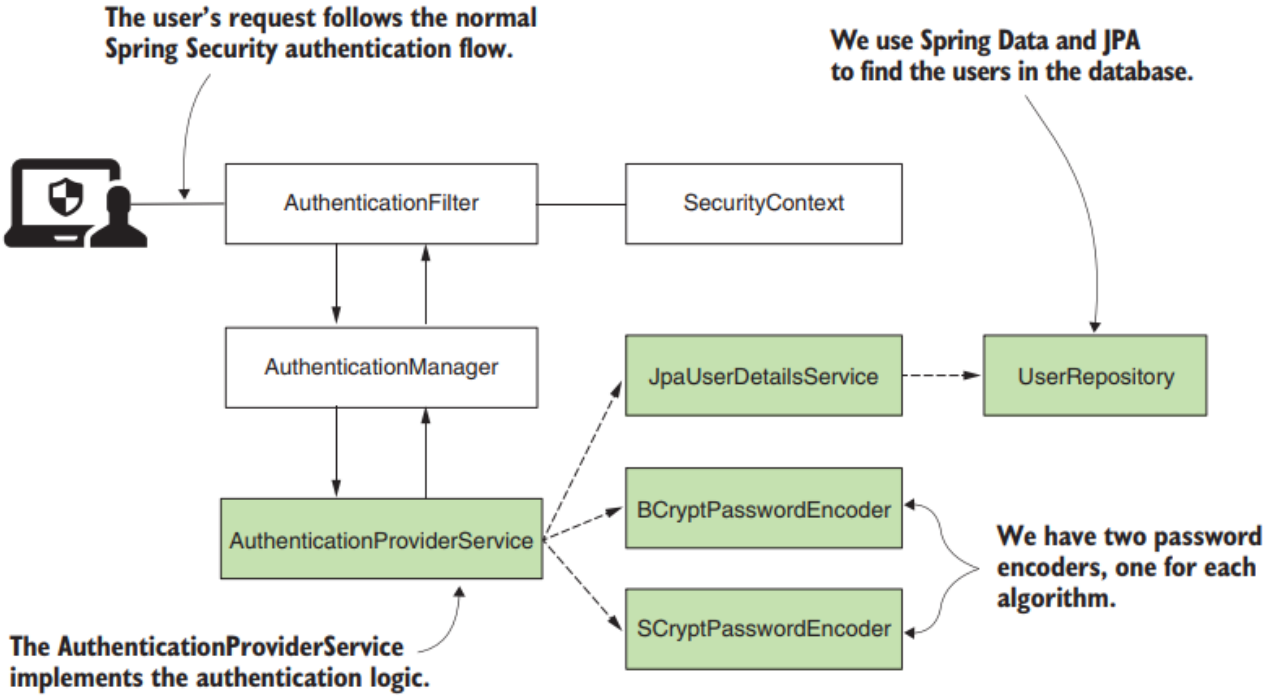
Using a holding strategy for asynchronous calls:

```
@GetMapping("/bye")
@Async
public void bye() {
    Authentication authentication = SecurityContextHolder.getContext().getAuthentication();
    log.debug("Authentication name {} ", authentication.getName());

}
}
```

```java
@Configuration
@EnableAsync
public class SecurityWebConfig extends WebSecurityConfigurerAdapter {

    @Bean
    InitializingBean initializingBean(){
        return () -> SecurityContextHolder.setStrategyName(SecurityContextHolder.MODE_INHERITABLETHREADLOCAL);
    }
```

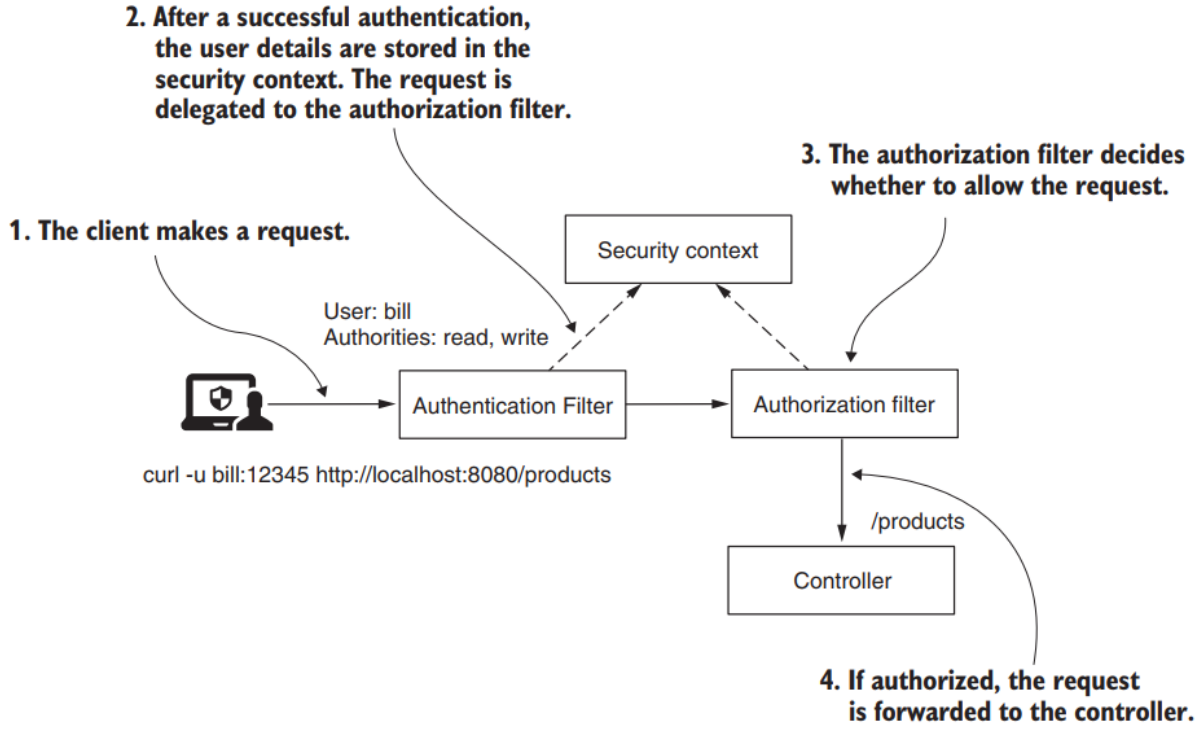


- ***Upgrade project***



The main steps we take to implement this project are as follows:

1. Set up the database
2. Define user management
3. Implement the authentication logic
4. Implement the main page
5. Run and test the application



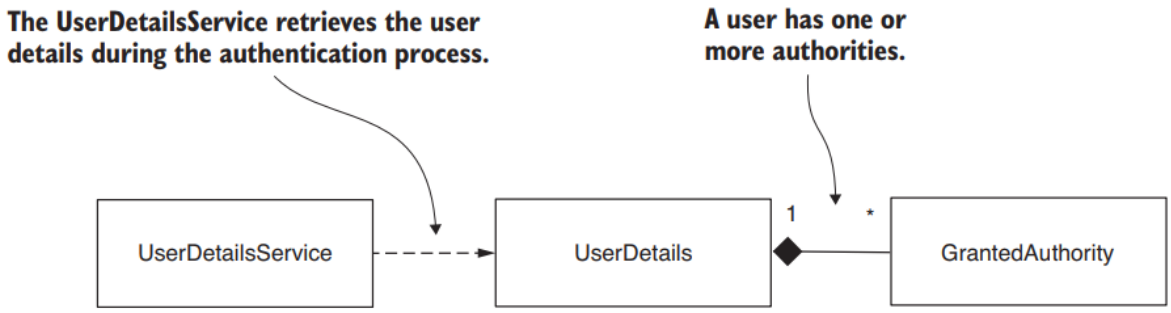


Figure 7.3 A user has one or more authorities (actions that a user can do). During the authentication process, the `UserDetailsService` obtains all the details about the user, including the authorities. The application uses the authorities as represented by the `GrantedAuthority` interface for authorization after it successfully authenticates the user.

Role and authorities implement in the same **GrantAuthority**, but ROLE have prefix **ROLE_**,

Apply role:

```
@Bean
public UserDetailsService userDetailsService() {
  var manager = new InMemoryUserDetailsManager();

  var user1 = User.withUsername("john")
          .password("12345")
          .roles("ADMIN")
          .build();

  var user2 = User.withUsername("jane")
          .password("12345")
          .roles("MANAGER")
          .build();

  manager.createUser(user1);
  manager.createUser(user2);

  return manager;
}

@Bean
public PasswordEncoder passwordEncoder() {
  return NoOpPasswordEncoder.getInstance();
}

@Override
protected void configure(HttpSecurity http) throws Exception {
  http.httpBasic();

  http.authorizeRequests()
        .mvcMatchers("/hello").hasRole("ADMIN")
        .mvcMatchers("/ciao").hasRole("MANAGER");
}

}
```

Only calls the path /hello if the user has the ADMIN role

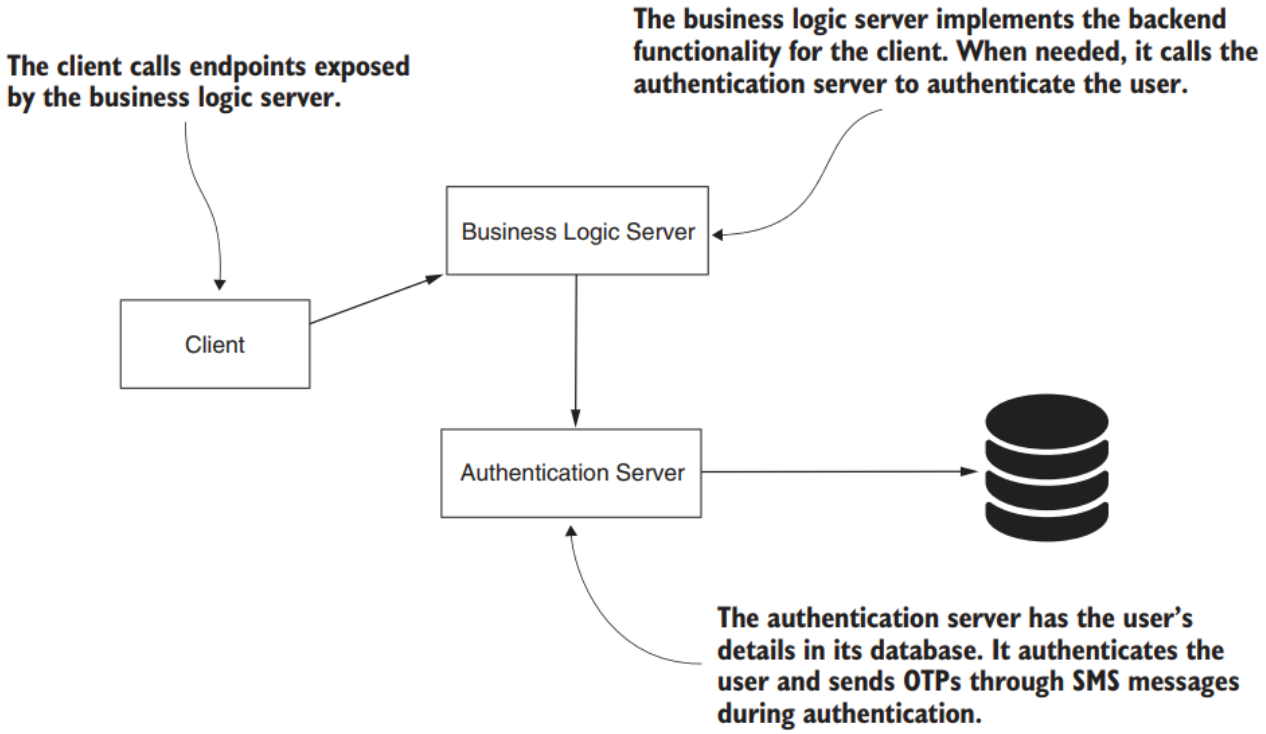
Only calls the path /ciao if the user has the Manager role

MVC matcher:

Table 8.1 Common expressions used for path matching with MVC matchers

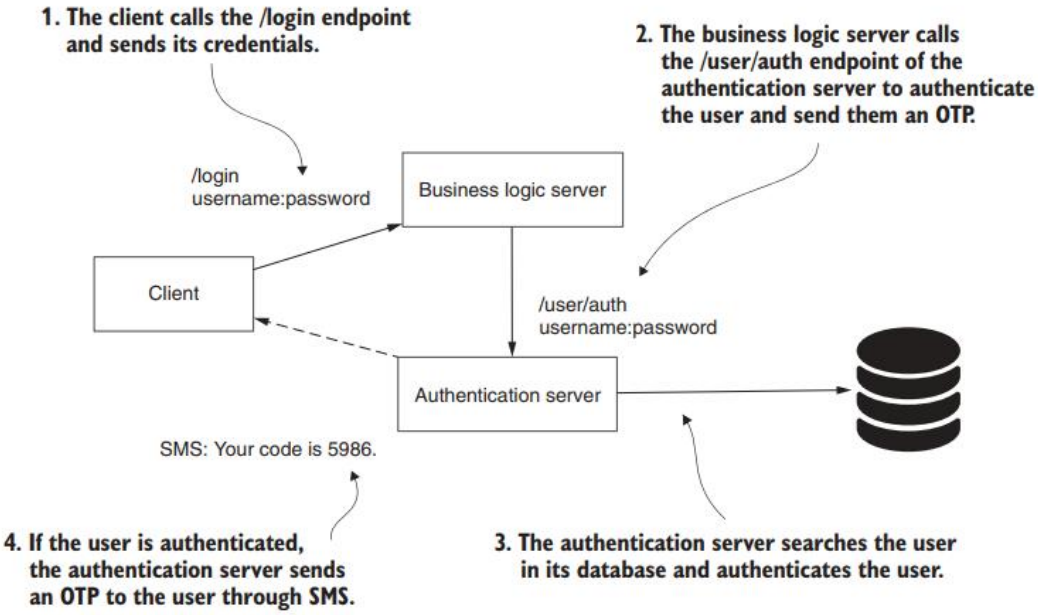
Expression	Description
`/a`	Only path /a.
`/a/*`	The * operator replaces one pathname. In this case, it matches /a/b or /a/c, but not /a/b/c.
`/a/**`	The ** operator replaces multiple pathnames. In this case, /a as well as /a/b and /a/b/c are a match for this expression.
`/a/{param}`	This expression applies to the path /a with a given path parameter.
`/a/{param:regex}`	This expression applies to the path /a with a given path parameter only when the value of the parameter matches the given regular expression.

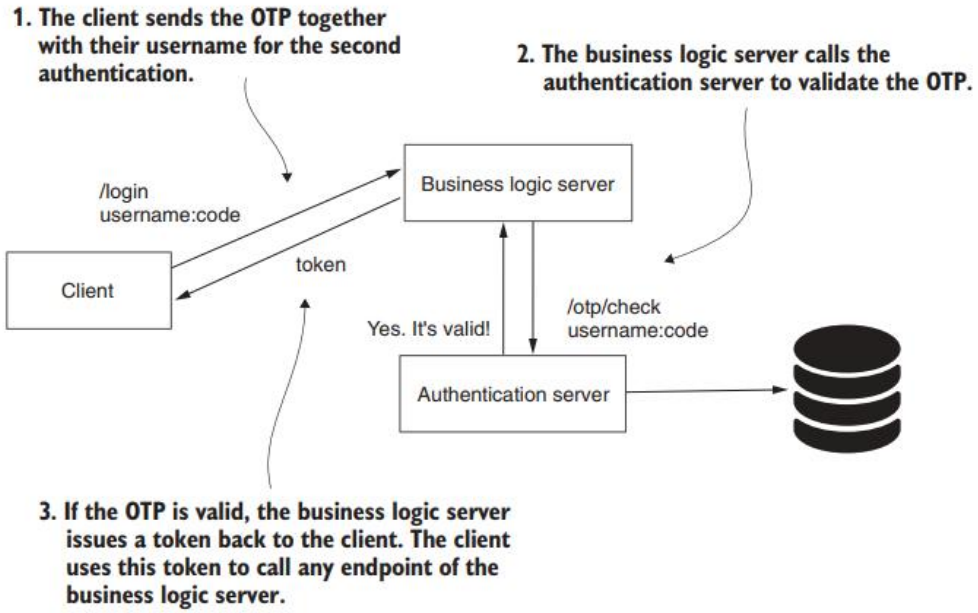
- ***Handon implement business authentication***



To call any endpoint on the business logic server, the client has to follow three steps:

1. Authenticate the username and password by calling the /login endpoint on the business logic server to obtain a randomly generated OTP.
2. Call the /login endpoint with the username and OTP.
3. Call any endpoint by adding the token received in step 2 to the `Authorization` header of the HTTP request.





- ***Token used case:***

- Tokens help you avoid sharing credentials in all requests.
- You can define tokens with a short lifetime.
- You can invalidate tokens without invalidating the credentials.
- Tokens can also store details like user authorities that the client needs to send in the request.
- Tokens help you delegate the authentication responsibility to another component in the system.

A JWT has three parts, each part separated from the others by a dot (a period). You find an example in this code snippet:

```
eyJhbGciOiJIUzI1NiJ9.eyJ1c2VybmFtZSI6ImRhbmllbGxlIn0.wg6LFProg7s_KvFxvnYGiZF-
    Mj4rr-0nJA1tVGZNn8U
```

The first two parts are the header and the body. The header (from the beginning of the token to the first dot) and the body (between the first and the second dot) are formatted as JSON and then are Base64 encoded. We use the header and the body to store details in the token. The next code snippet shows what the header and the body look like before these are Base64 encoded:

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

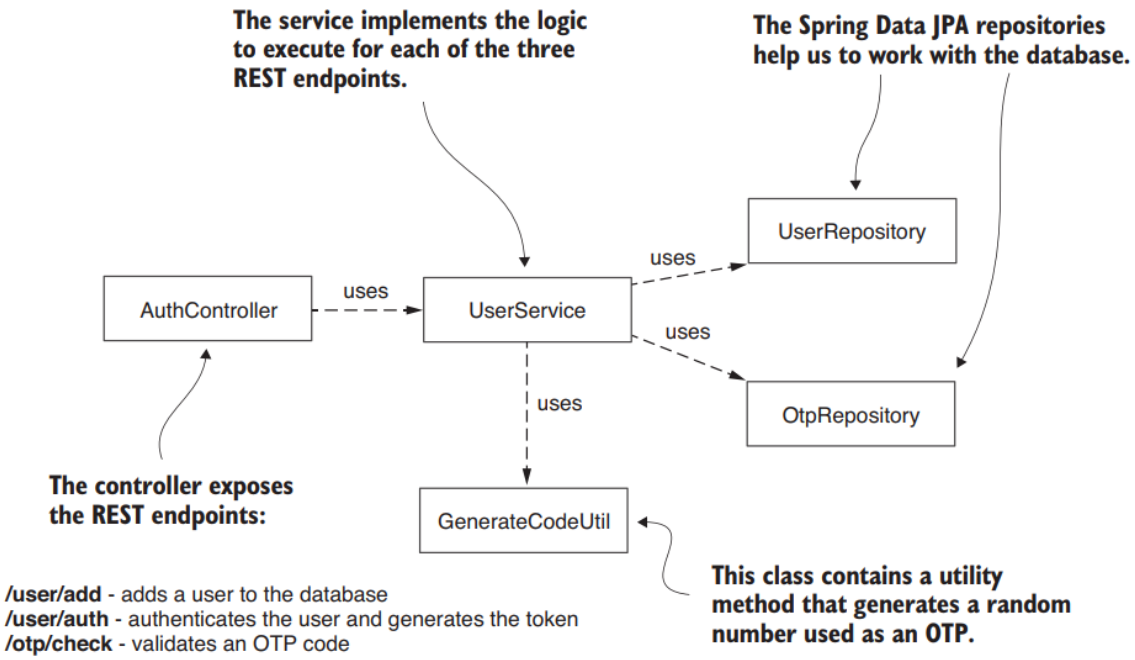
The Base64 encoded header

```
{
  "username": "danielle"
}
```

The Base64 encoded body

In our scenario, the authentication server connects to a database where it stores the user credentials and the OTPs generated during request authentication events. We need this application to expose three endpoints (figure 11.9):

- */user/add*—Adds a user that we use later for testing our implementation.
- */user/auth*—Authenticates a user by their credentials and sends an SMS with an OTP. We take out the part that sends the SMS, but you can do this as an exercise.
- */otp/check*—Verifies that an OTP value is the one that the authentication server generated earlier for a specific user.



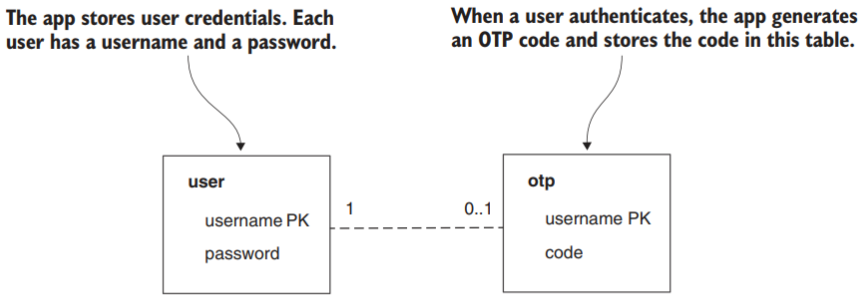


Figure 11.10 The app database has two tables. In one of the tables, the app stores user credentials, while in the second one, the app stores the generated OTP codes.

▶ *Implement business logic*

1. Create an endpoint that represents the resource we want to secure.
2. Implement the first authentication step in which the client sends the user credentials (username and password) to the business logic server to log in.
3. Implement the second authentication step in which the client sends the OTP the user receives from the authentication server to the business logic server. Once authenticated by the OTP, the client gets back a JWT, which it can use to access a user's resources.
4. Implement authorization based on the JWT. The business logic server validates the JWT received from a client and, if valid, allows the client to access the resource.

Technically, to achieve these four high-level points, we need to

1. Create the business logic server project. I name it ssia-ch11-ex1-s2 .
2. Implement the `Authentication` objects that have the role of representing the two authentication steps.
3. Implement a proxy to establish communication between the authentication server and the business logic server.
4. Define the `AuthenticationProvider` objects that implement the authentication logic for the two authentication steps using the `Authentication` objects defined in step 2.
5. Define the custom filter objects that intercept the HTTP request and apply the authentication logic implemented by the `AuthenticationProvider` objects.
6. Write the authorization configurations.

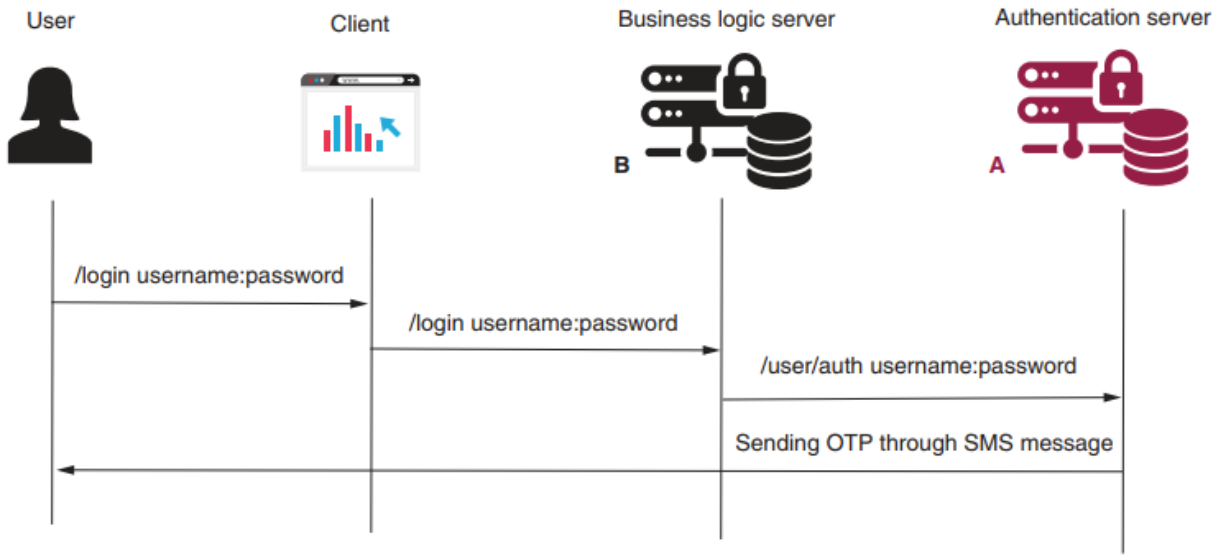


Figure 11.11 The first authentication step. The user sends their credentials for authentication. The authentication server authenticates the user and sends an SMS message containing the OTP code.

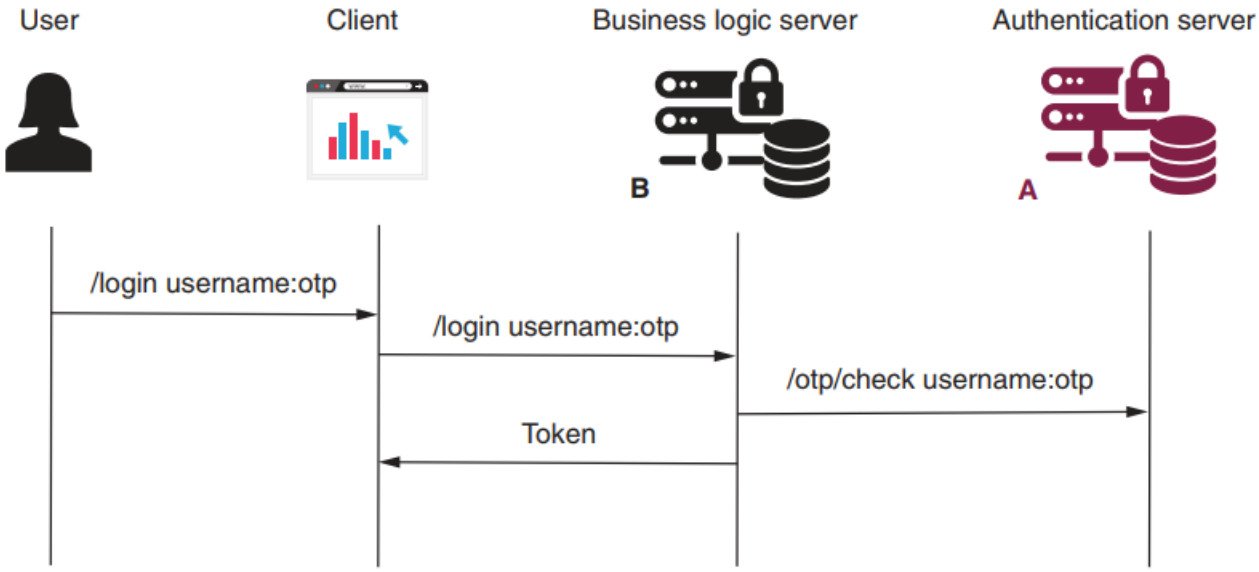


Figure 11.12 The second authentication step. The user sends the OTP code they received as a result of the first authentication step. The authentication server validates the OTP code and sends back a token to the client. The client uses the token to access the user's resources.

- Authentication with the token to access the endpoint (figure 11.13).

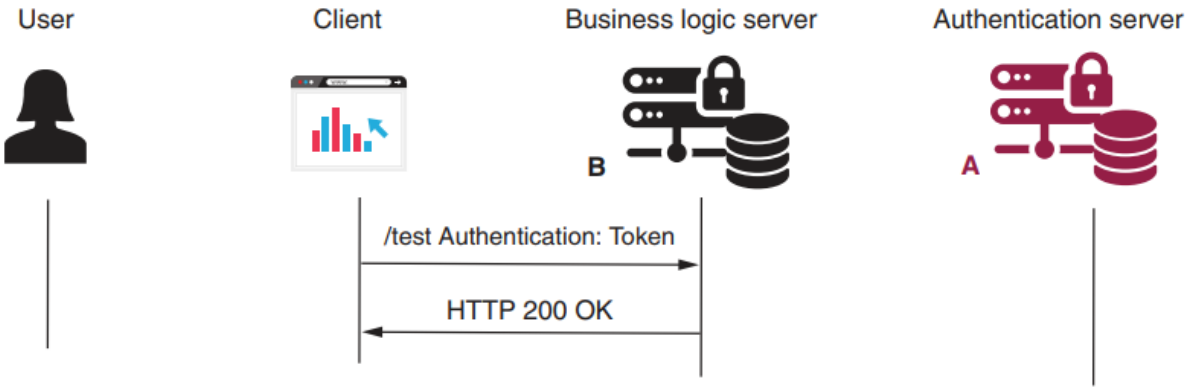
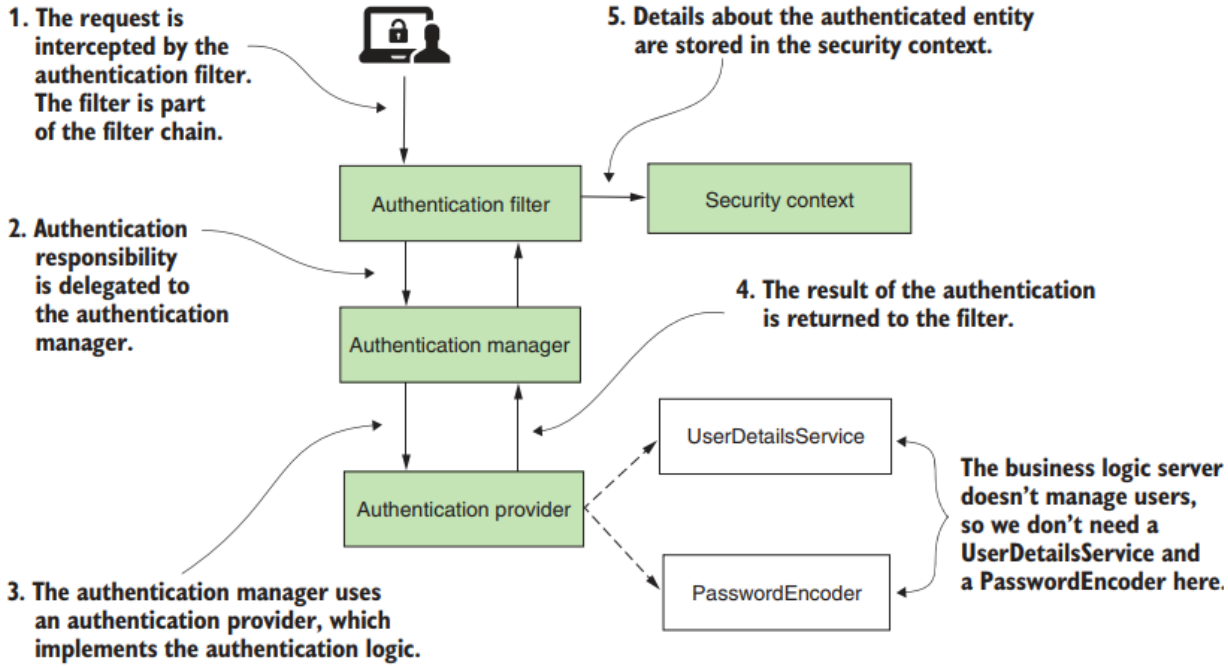
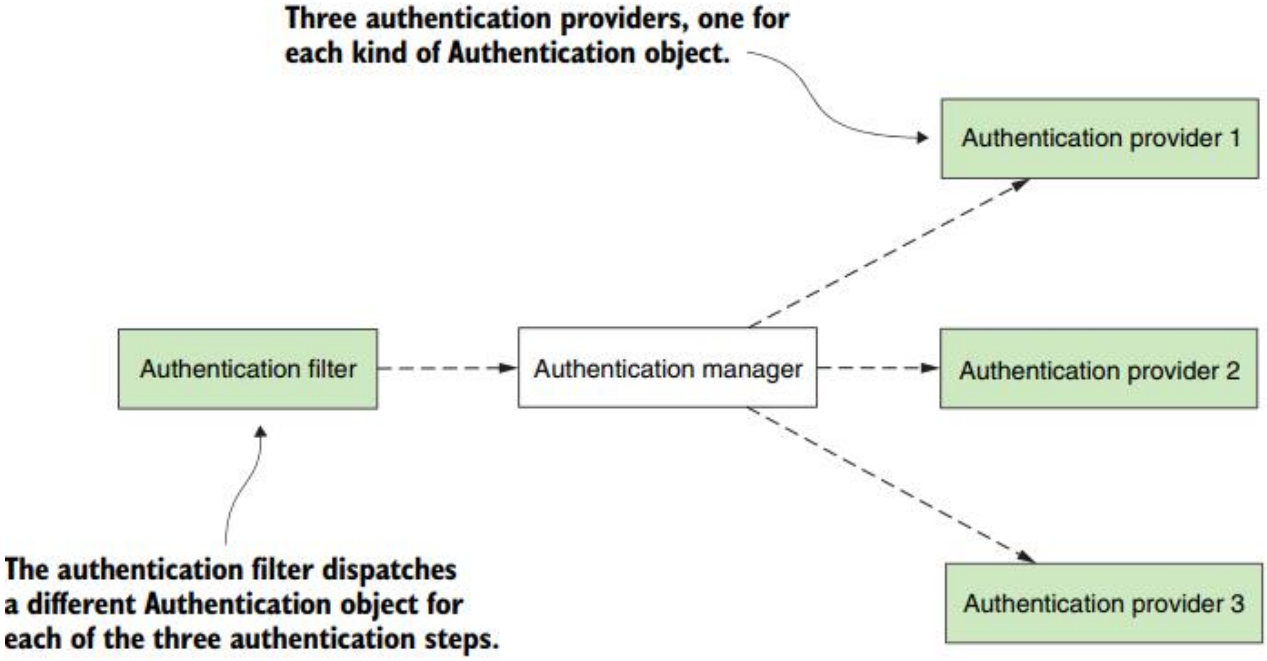
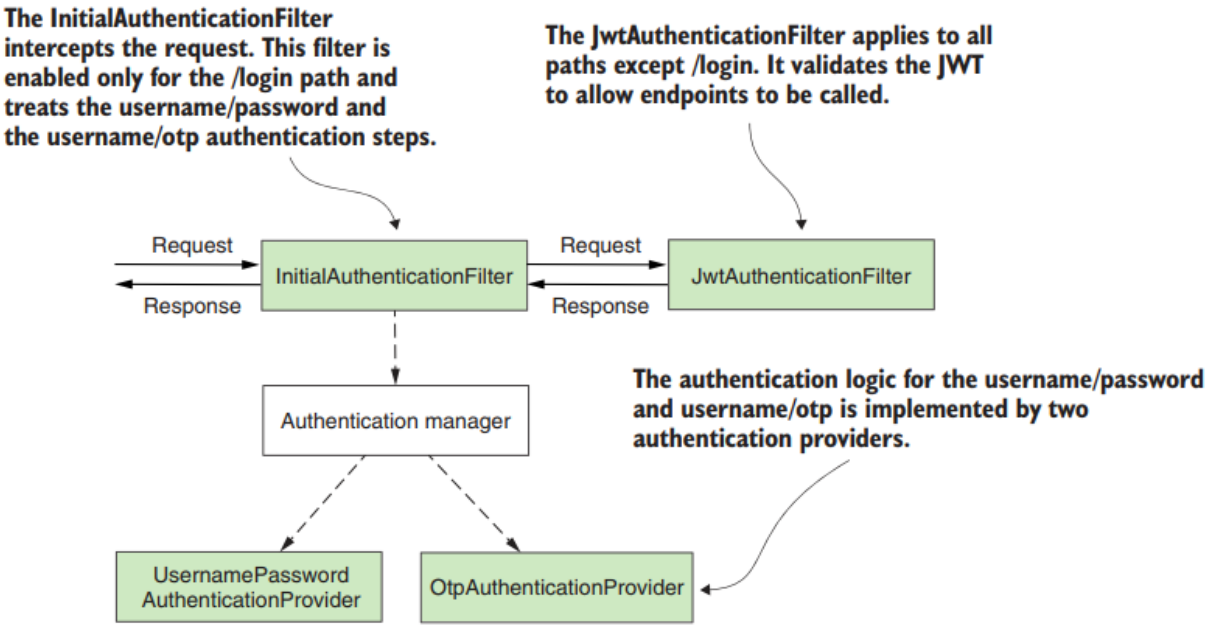
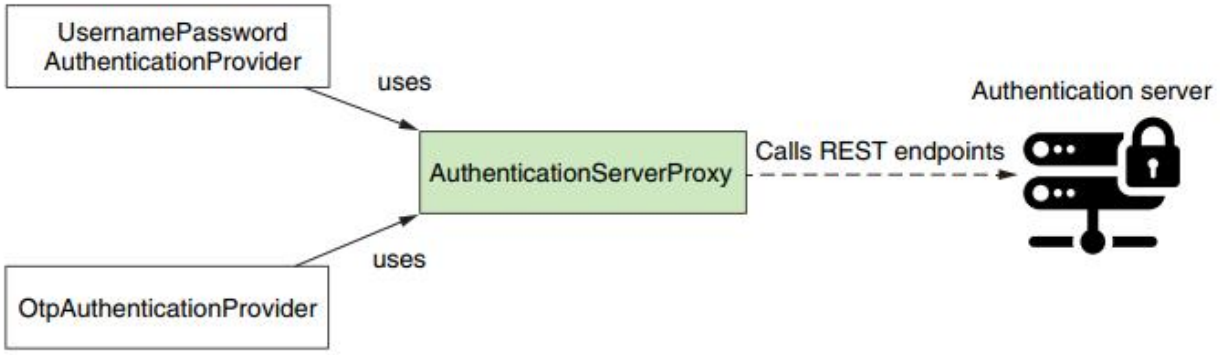


Figure 11.13 The last authentication step. The client uses the token obtained in step 2 to access resources exposed by the business logic server.




Three authentication providers, one for each kind of Authentication object.
Authentication provider 1
Authentication filter
Authentication manager
Authentication provider 2
Authentication provider 3
The authentication filter dispatches a different Authentication object for each of the three authentication steps.


The InitialAuthenticationFilter intercepts the request. This filter is enabled only for the /login path and treats the username/password and the username/otp authentication steps.
The JwtAuthenticationFilter applies to all paths except /login. It validates the JWT to allow endpoints to be called.
Request
Response
InitialAuthenticationFilter
Request
Response
JwtAuthenticationFilter
Authentication manager
The authentication logic for the username/password and username/otp is implemented by two authentication providers.
UsernamePassword AuthenticationProvider
OtpAuthenticationProvider



For this implementation, we need to

1. Define a model class `User`, which we use to call the REST services exposed by the authentication server
2. Declare a bean of type `RestTemplate`, which we use to call the REST endpoints exposed by the authentication server
3. Implement the proxy class, which defines two methods: one for username/password authentication and the other for username/otp authentication

Testing Api:

```
curl -H "username:danielle" -H "password:12345" http://localhost:9090/login
```

```
Username: danielle
Code: 6271
```

```
curl -v -H "username:danielle" -H "code:6271" http://localhost:9090/login
```

```
. . .
< HTTP/1.1 200
< Authorization:
     eyJhbGciOiJIUzI1NiJ9.eyJ1c2VybmFtZSI6ImRhbmllbGxlIn0.wg6LFProg7s_KvFxvnY
     GiZF-Mj4rr-0nJA1tVGZNn8U
```

```
curl -H "Authorization:eyJhbGciOiJIUzI1NiJ9.eyJ1c2VybmFtZSI6ImRhbmllbGxlIn0
     .wg6LFProg7s_KvFxvnYGiZF-Mj4rr-0nJA1tVGZNn8U"
     http://localhost:9090/test
```

The response body is

```
Test
```

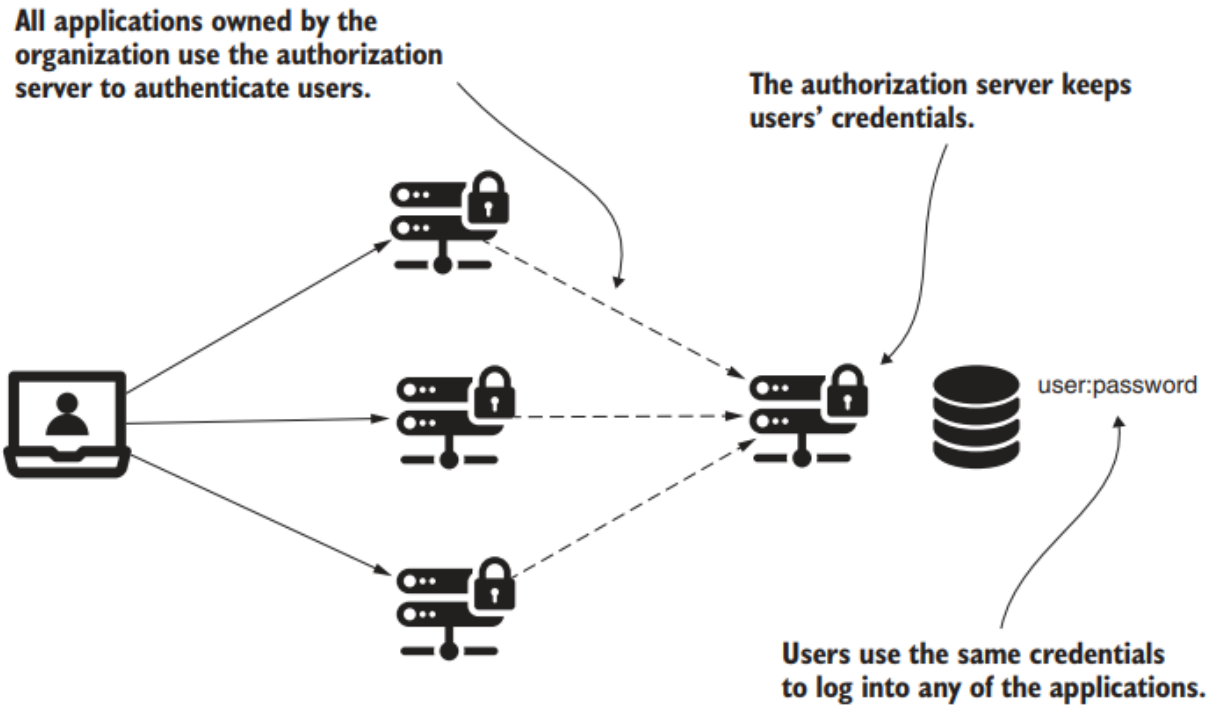
6.2. JWT Oauth implementation

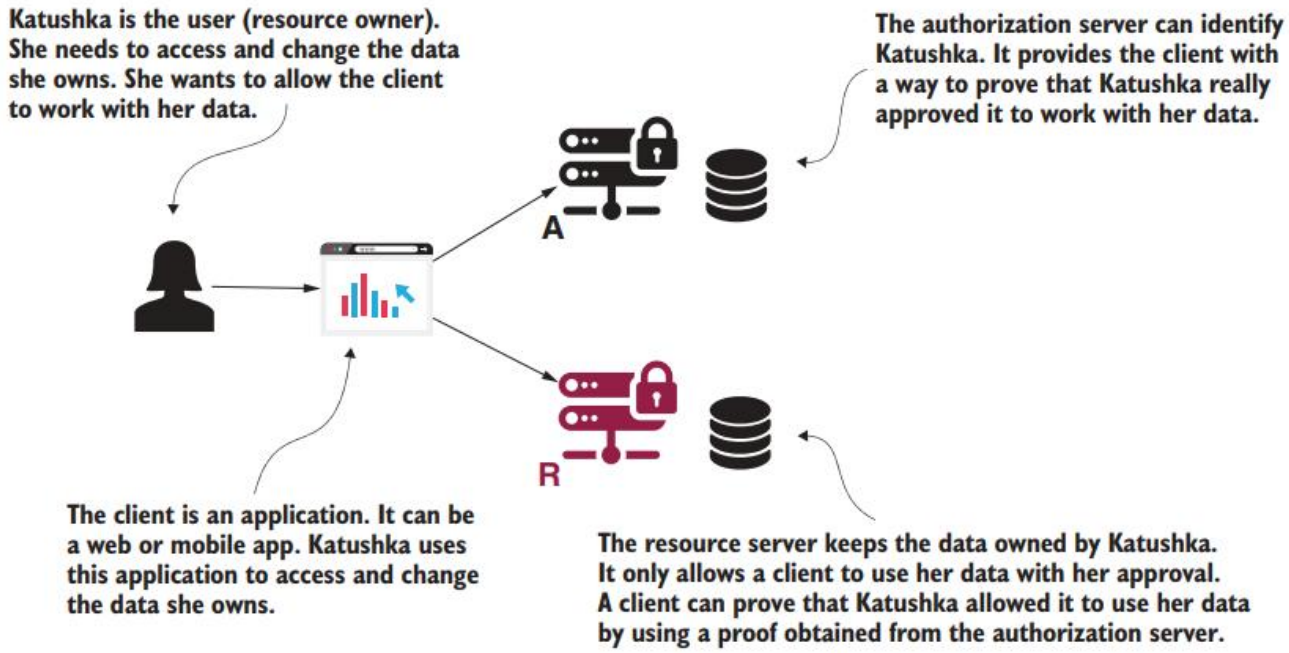
6.2.1. How does Oauth 2 works?

Oath2 is an authentication framework include:

- Resource server
- Authentication serer
- Client app
- Resource owner

Issue with traditional application:

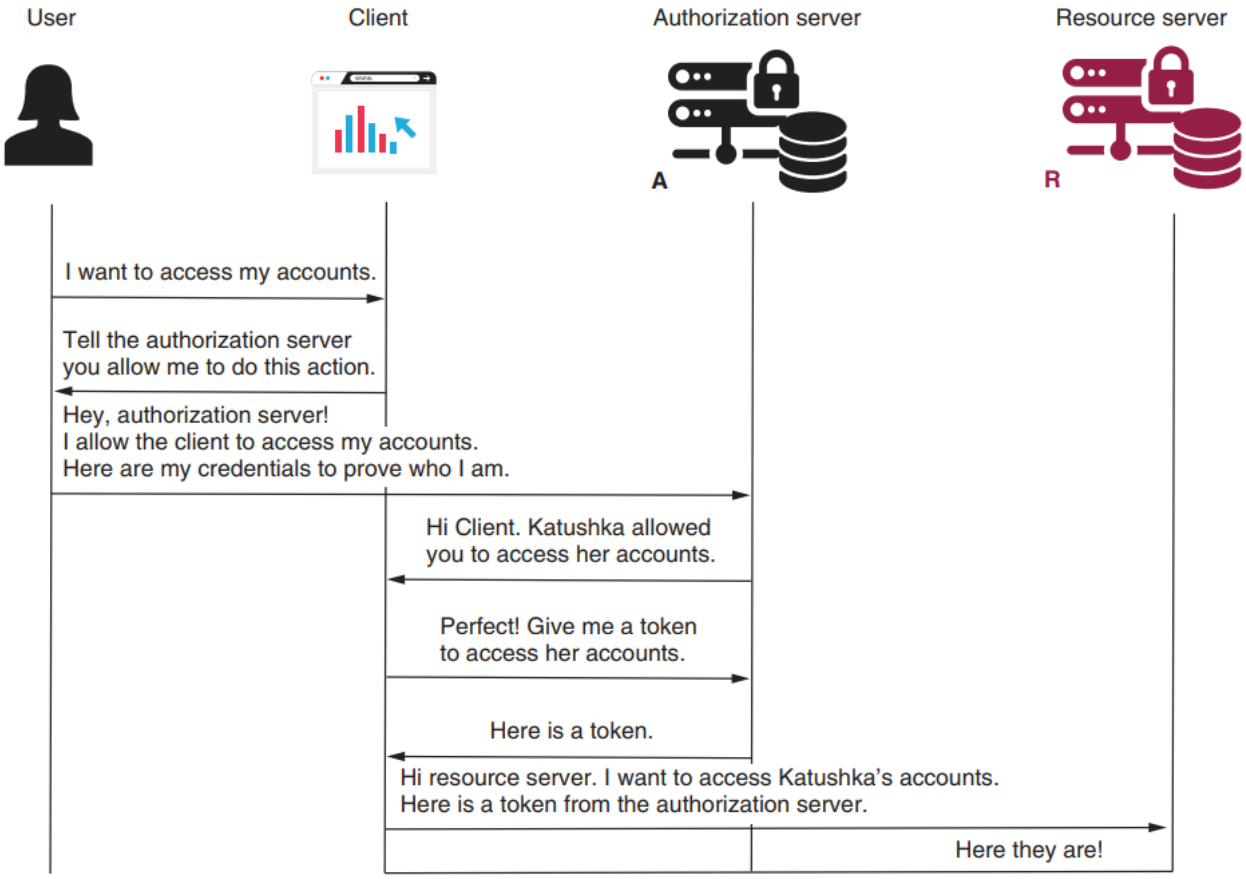




Oauth 2 have mutiple scheme to obtain token:

- Authorization code
- Password
- Client credentials
- Refresh token

- ***Authorization code scheme:***



Step 1: MAKING THE AUTHENTICATION REQUEST WITH THE AUTHORIZATION CODE GRANT TYPE.

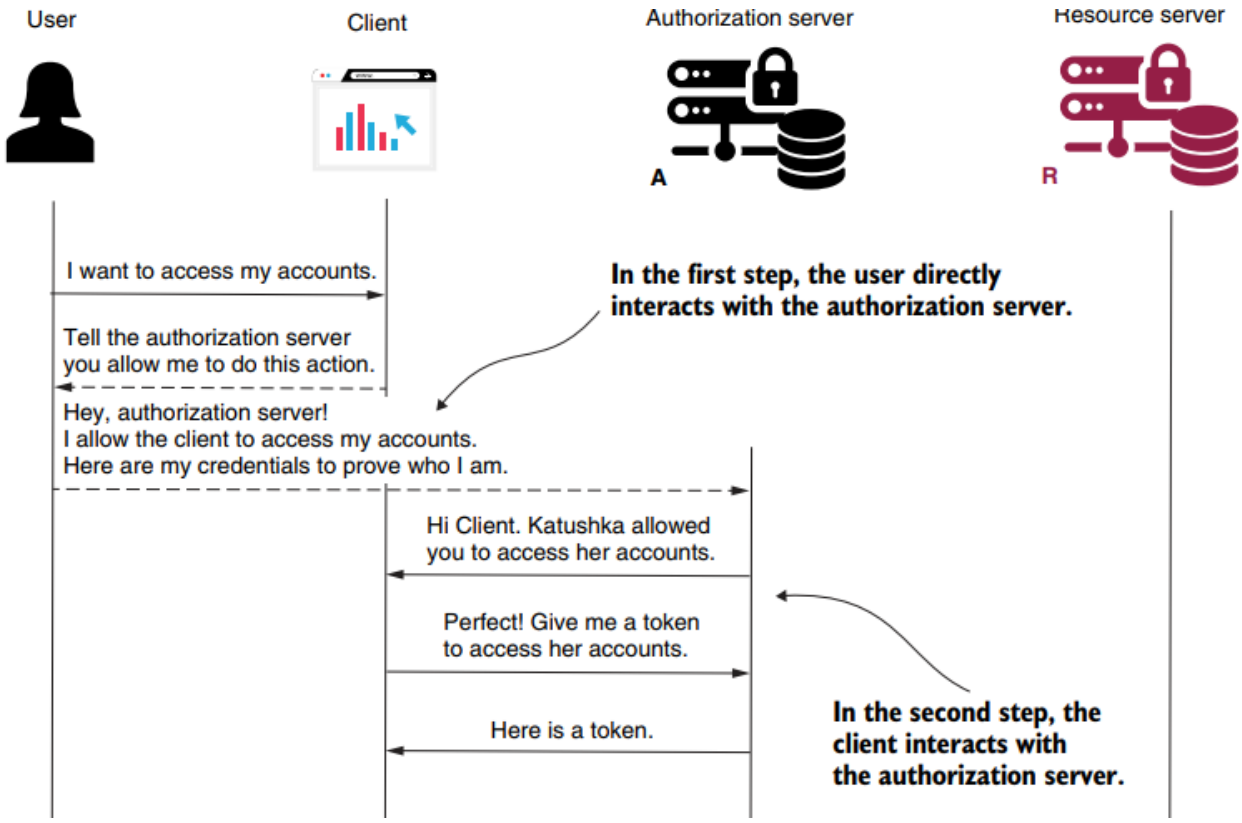
The client calls the authorization endpoint with the following details in the request query:

- `response_type` with the value `code`, which tells the authorization server that the client expects a code. The client needs the code to obtain an access token, as you'll see in the second step.
- `client_id` with the value of the client ID, which identifies the application itself.
- `redirect_uri`, which tells the authorization server where to redirect the user after successful authentication. Sometimes the authorization server already knows a default redirect URI for each client. For this reason, the client doesn't need to send the redirect URI.

- `scope`, which is similar to the granted authorities we discussed in chapter 5.
- `state`, which defines a cross-site request forgery (CSRF) token used for the CSRF protection we discussed in chapter 10.

After successful authentication, the authorization server calls back the client on the redirect URI and provides a **code** and the **state(csrf)** value.

STEP 2: OBTAINING AN ACCESS TOKEN WITH THE AUTHORIZATION CODE GRANT TYPE



People are generally puzzled about why the flow needs two calls to the authorization server and two different tokens—the **authorization code** and an **access token**. Take a moment to understand this:

- The authorization server generates the first code as proof that the user directly interacted with it. The client receives this code and has to authenticate again using it and its credentials to obtain an access token.
- The client uses the second token to access resources on the resource server.

The client makes a final call to get an access token and sends

- The **authorization code**, which proves the user authorized them
- Their **credentials**, which proves they really are the same **client** and not someone else who **intercepted** the authorization codes

To return to step 2, technically, the client now makes a request to the authorization server. This request contains the following details:

- `code`, which is the authorization code received in step 1. This proves that the user authenticated.
- `client_id` and `client_secret`, the client's credentials.
- `redirect_uri`, which is the same one used in step 1 for validation.
- `grant_type` with the value `authorization_code`, which identifies the kind of flow used. A server might support multiple flows, so it's essential always to specify which is the current executed authentication flow.

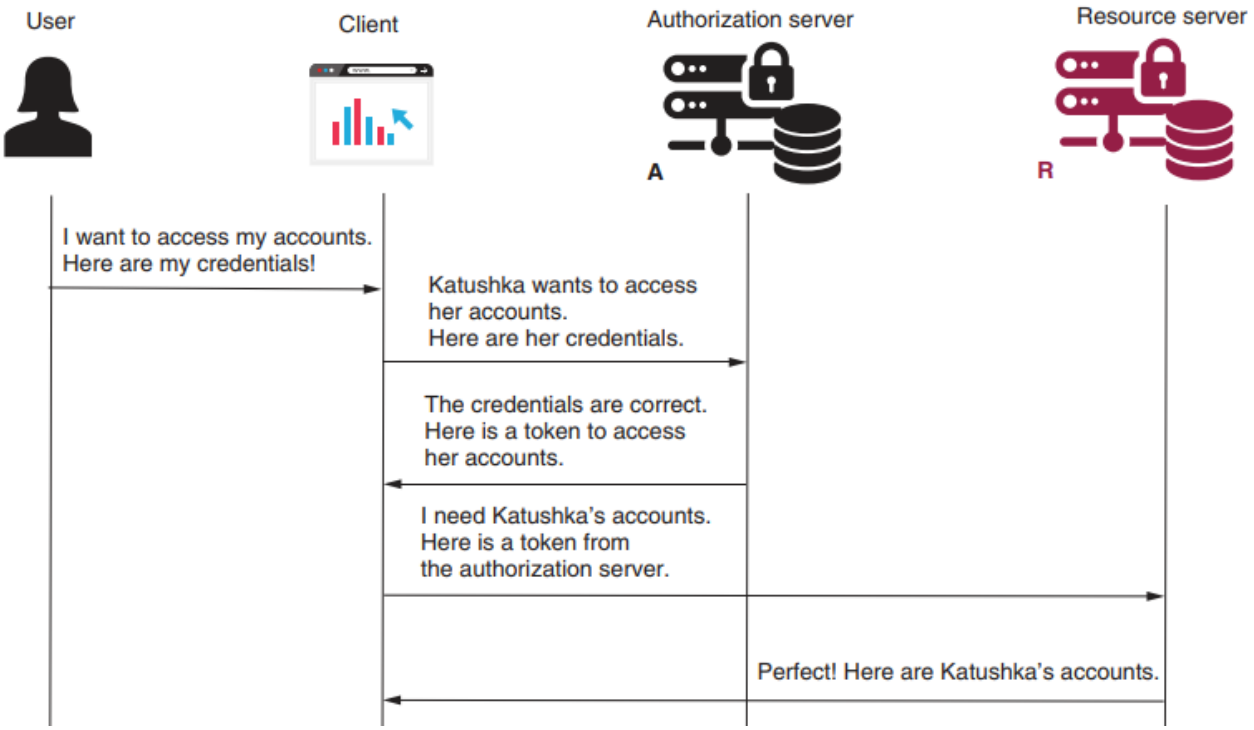
As a response, the server sends back an `access_token`. This token is a value that the client can use to call resources exposed by the resource server.

STEP 3: CALLING THE PROTECTED RESOURCE WITH THE AUTHORIZATION CODE GRANT TYPE

After successfully obtaining the access token from the authorization server, the client can now call for the protected resource. The client uses an access token in the authorization request header when calling an endpoint of the resource server.

► ***Implementing the password grant type***

In this section, we discuss the password grant type (figure 12.7). This grant type is also known as the resource owner credentials grant type. Applications using this flow assume that the client collects the user credentials and uses these to authenticate and obtain an access token from the authorization server



STEP 1: REQUESTING AN ACCESS TOKEN WHEN USING THE PASSWORD GRANT TYPE

The flow is much simpler with the password grant type. The client collects the user credentials and calls the authorization server to obtain an access token. When requesting the access token, the client also sends the following details in the request:

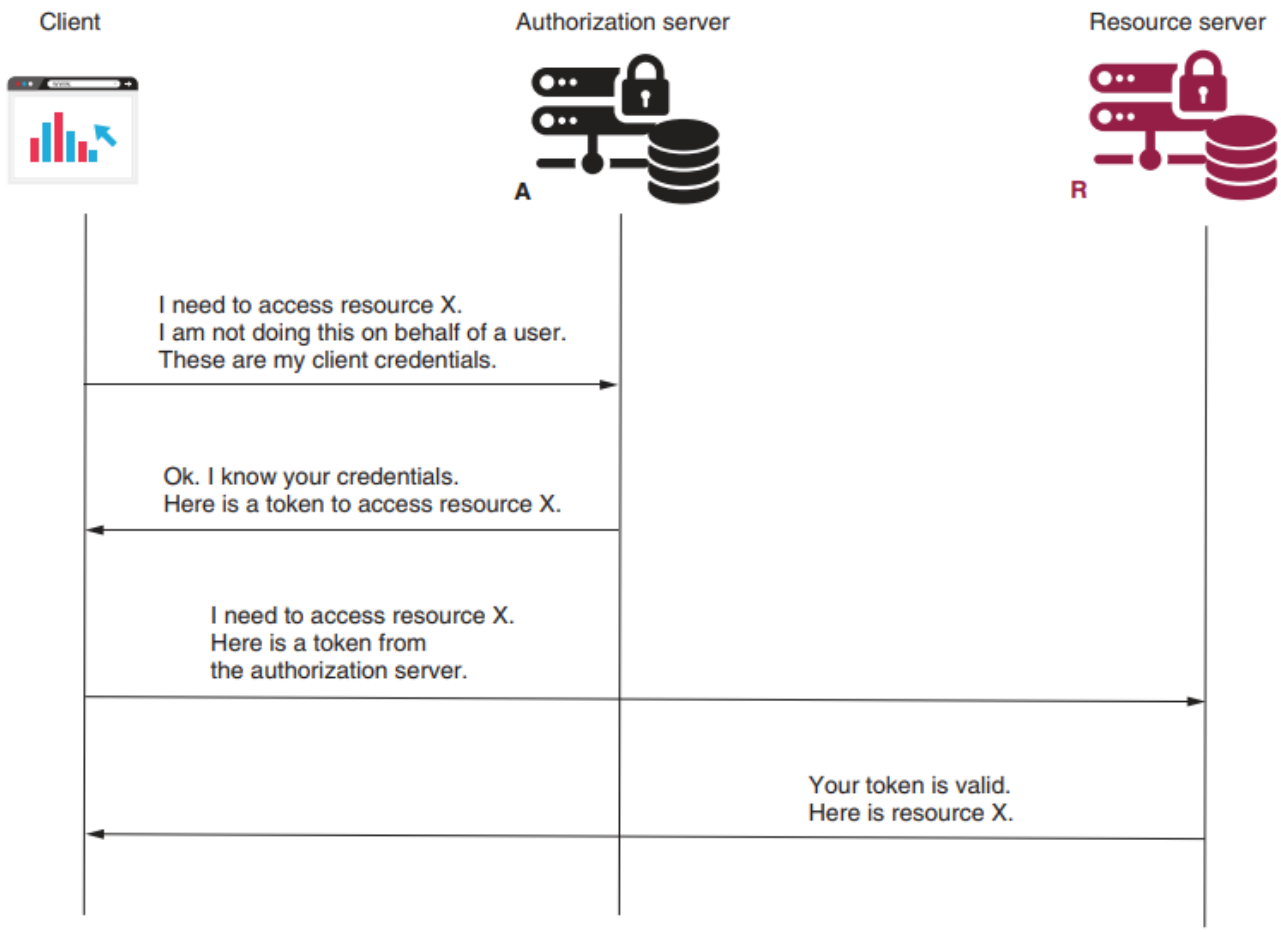
- `grant_type` with the value `password`.
- `client_id` and `client_secret`, which are the credentials used by the client to authenticate itself.
- `scope`, which you can understand as the granted authorities.
- `username` and `password`, which are the user credentials. These are sent in plain text as values of the request header.

The client receives back an access token in the response. The client can now use the access token to call the endpoints of the resource server.

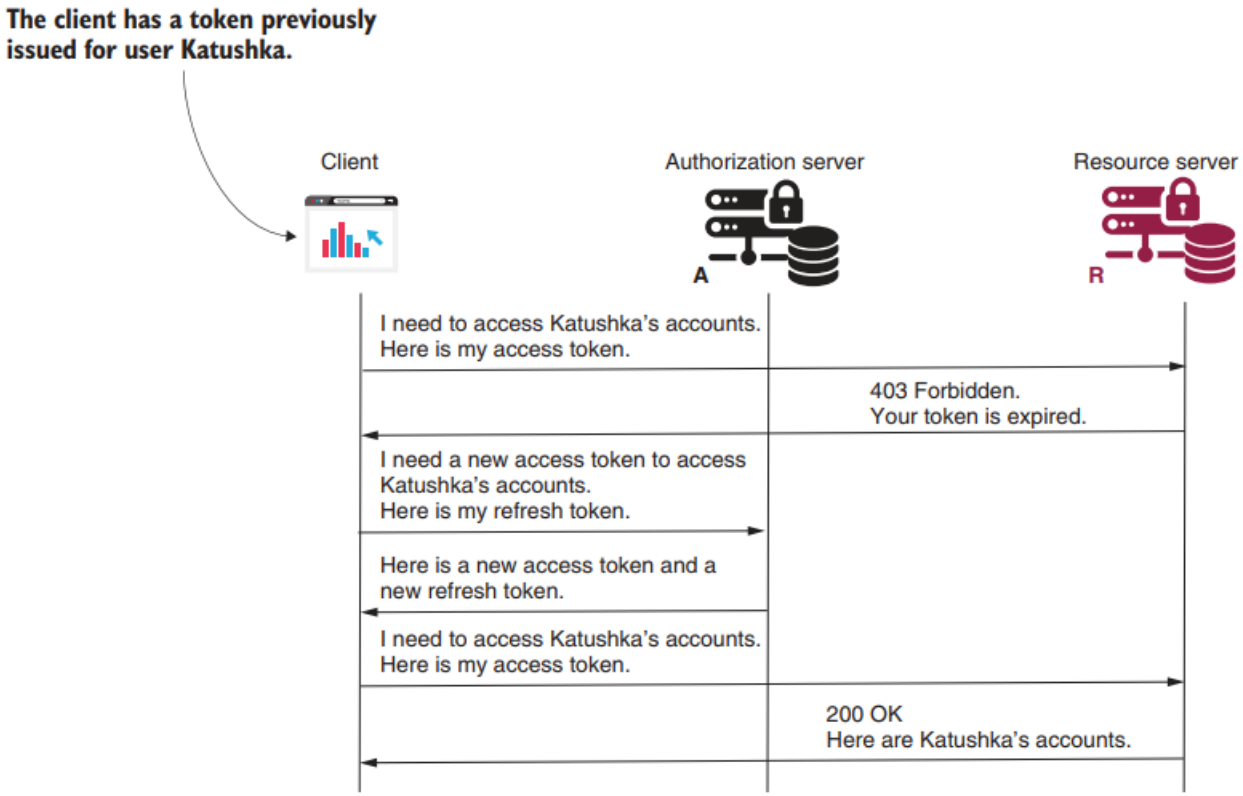
STEP 2: USING AN ACCESS TOKEN TO CALL RESOURCES WHEN USING THE PASSWORD GRANT TYPE

Once the client has an access token, it uses the token to call the endpoints on the resource server, which is exactly like the authorization code grant type. The client adds the access token to the requests in the authorization request header.

- ***Implementing the client credentials grant type***



- ***Using refresh tokens to obtain new access tokens***



Note on Oauth 2:

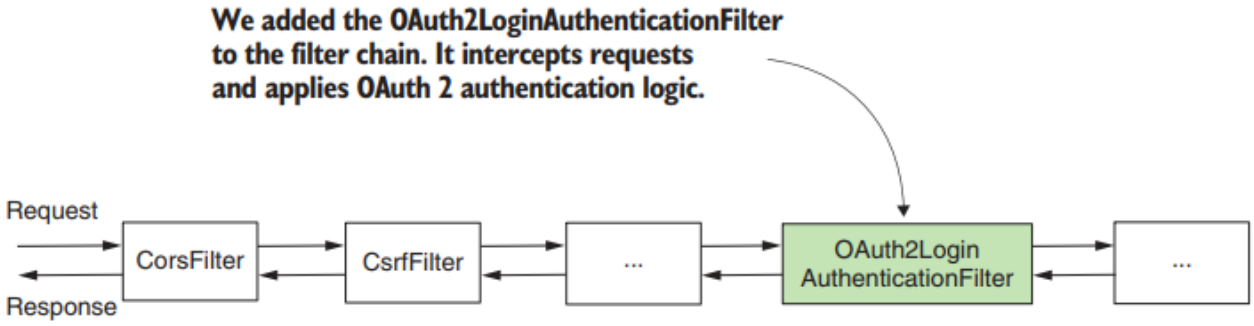
Configuration class:

Listing 12.2 The configuration class

```
@Configuration
public class ProjectConfig
  extends WebSecurityConfigurerAdapter {

  @Override
  protected void configure(HttpSecurity http) throws Exception {
    http.oauth2Login();

    http.authorizeRequests()
          .anyRequest()
            .authenticated();
  }
}
```

Sets the authentication method

Specifies that a user needs to be authenticated to make a request



Implementing ClientRegistration:

The `ClientRegistration` interface represents the client in the OAuth 2 architecture. For the client, you need to define all its needed details, among which we have

- The client ID and secret
- The grant type used for authentication
- The redirect URI
- The scopes

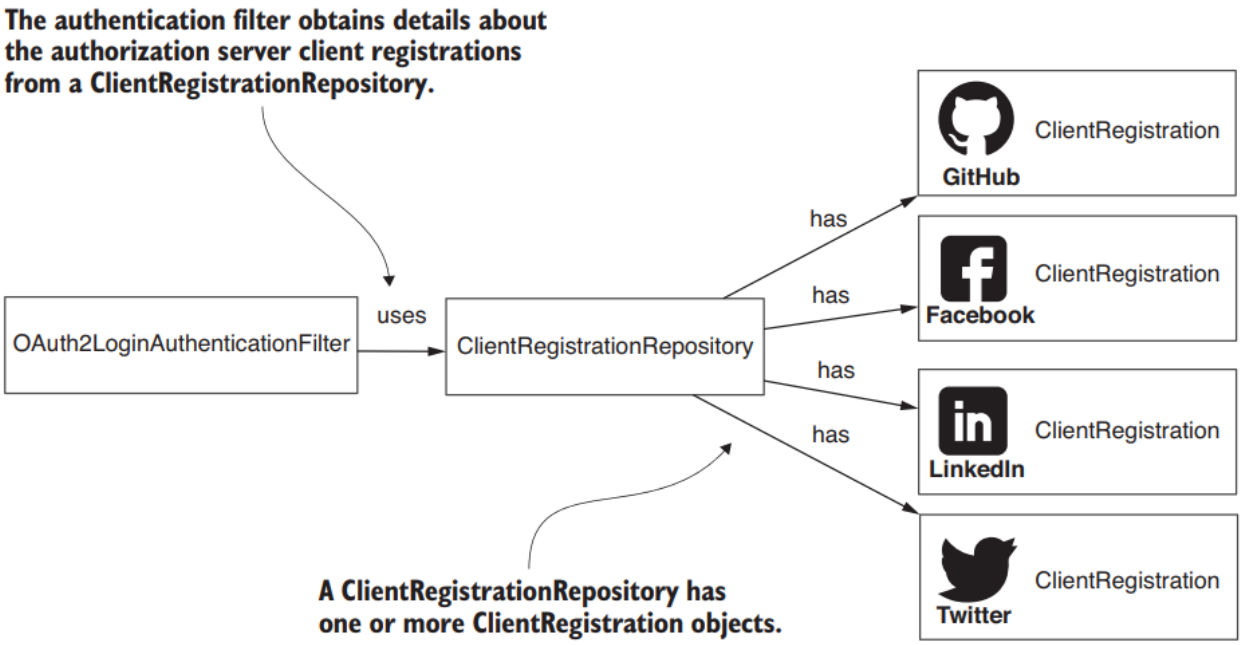
Listing 12.3 Creating a `ClientRegistration` instance

```
ClientRegistration cr =
   ClientRegistration.withRegistrationId("github")
     .clientId("a7553955a0c534ec5e6b")
     .clientSecret("1795b30b425ebb79e424afa51913f1c724da0dbb")
     .scope(new String[]{"read:user"})
     .authorizationUri(
        "https://github.com/login/oauth/authorize")
     .tokenUri("https://github.com/login/oauth/access_token")
     .userInfoUri("https://api.github.com/user")
     .userNameAttributeName("id")
     .clientName("GitHub")
     .authorizationGrantType(AuthorizationGrantType.AUTHORIZATION_CODE)
     .redirectUriTemplate("{baseUrl}/{action}/oauth2/code/{registrationId}")
     .build();
```

Oh! Where did all those details come from? I know listing 12.3 might look scary at first glance, but it's nothing more than setting up the client ID and secret. Also, in listing 12.3, I define the scopes (granted authorities), a client name, and a registration ID of my choice. Besides these details, I had to provide the URLs of the authorization server:

- *Authorization URI*—The URI to which the client redirects the user for authentication
- *Token URI*—The URI that the client calls to obtain an access token and a refresh token, as discussed in section 12.3
- *User info URI*—The URI that the client can call after obtaining an access token to get more details about the user

Implementing ClientRegistrationRepository:



The **ClientRegistrationRepository** interface is similar to the **UserDetailsService** interface, which you learned about in chapter 2. In the same way that a **UserDetailsService** object finds **UserDetails** by its **username**, a **ClientRegistrationRepository** object finds **ClientRegistration** by its registration ID.

```
private ClientRegistrationRepository clientRepository() {
    var c : ClientRegistration = clientRegistration();
    return new InMemoryClientRegistrationRepository(c);
}

private ClientRegistration clientRegistration(){
    return CommonOAuth2Provider.GITHUB
                               .getBuilder( s: "github")
                               .clientId("71dd41f366744deb5cec")
                               .clientSecret("3900b5d59c05b7a946fd9d0ad7ab5663b0e00b0f")
                               .build();
}
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.oauth2Login();
    http.authorizeRequests()
        .anyRequest()
        .authenticated();
}
```

Request to Github:

https://github.com/login/oauth/authorize?client_id=71dd41f366744deb5cec&**redirect_uri=http://localhost:8080/login/oauth2/code/github**&**response_type=**code&**scope=**read:user&**state=6SjrJrnyG9TmPJGM2AJGbqLcNKYqEQQ-HXBVnWKX-WQ=**

Response from Git:

http://localhost:8080/login/oauth2/code/github?code=e3979c04e29afc805845&state=**6SjrJrnyG9TmPJGM2AJGbqLcNKYqEQQ-HXBVnWKX-WQ=**

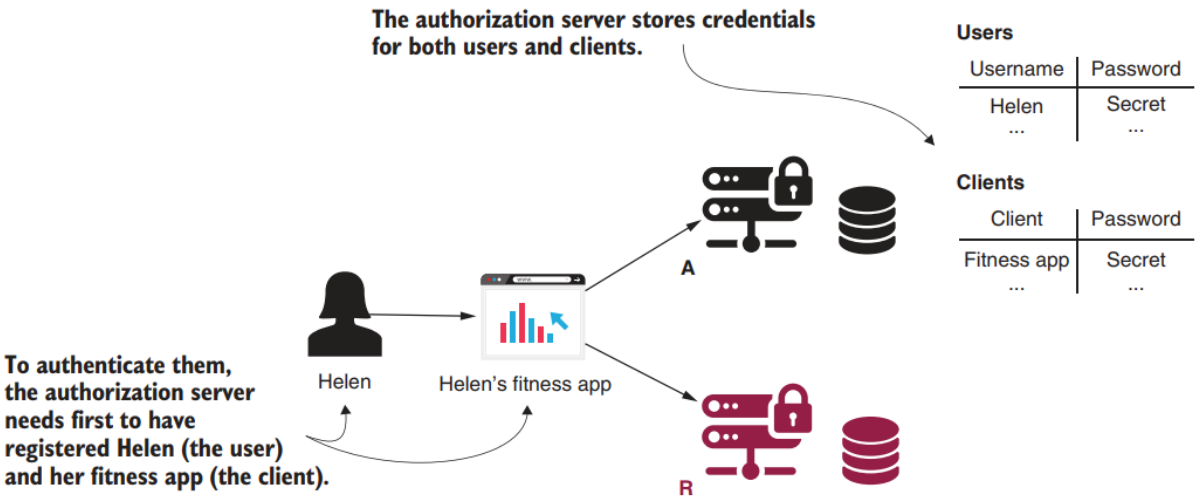
6.2.2. Implement Authorization Server

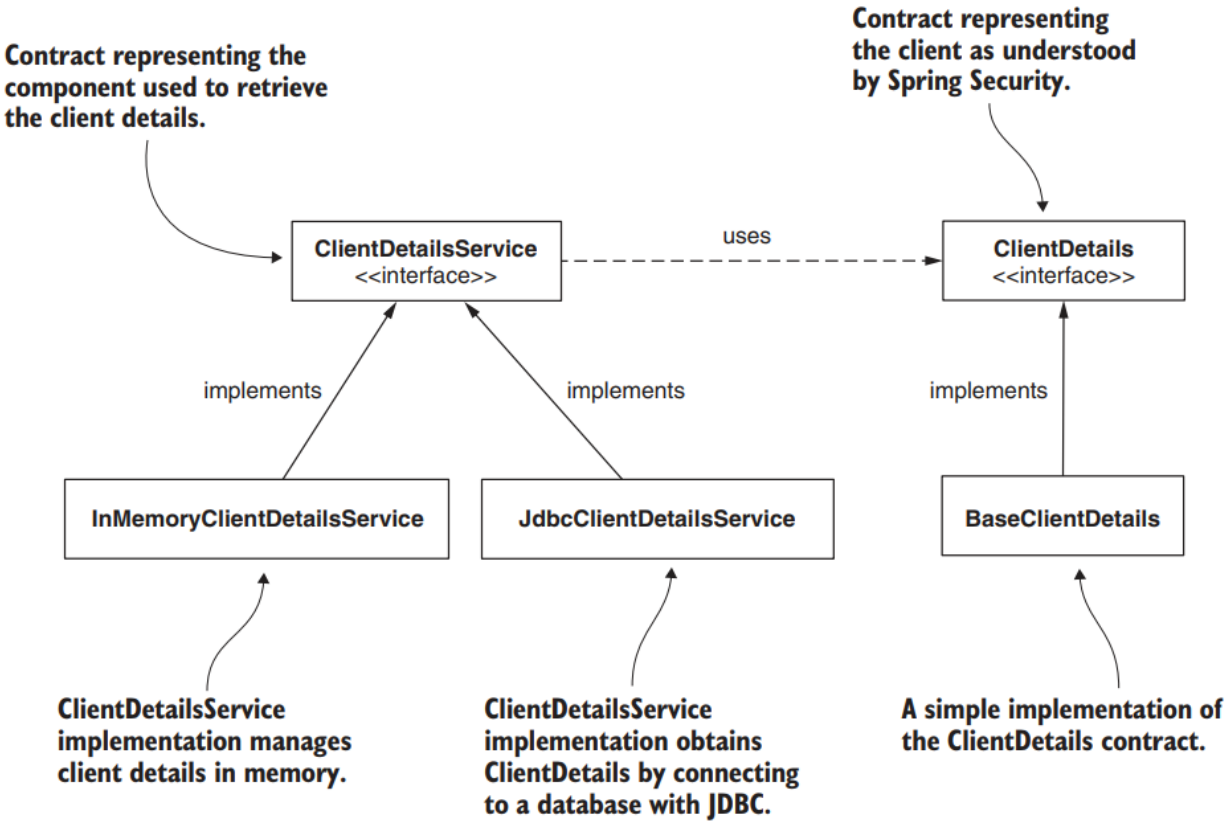
- **Config User management:**

```
@Configuration
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
    @Bean
    public UserDetailsService uds() {
        var uds = new InMemoryUserDetailsManager();
        var u : UserDetails = User.withUsername("john")
                    .password("12345")
                    .authorities("read")
                    .build();
        uds.createUser(u);
        return uds;
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return NoOpPasswordEncoder.getInstance();
    }

    @Override
    @Bean
    public AuthenticationManager authenticationManagerBean() throws Exception {
        return super.authenticationManagerBean();
    }
}
```

```java
@Configuration
@EnableAuthorizationServer
public class AuthServerConfig extends AuthorizationServerConfigurerAdapter {

    private AuthenticationManager manager;

    @Autowired
    public void setAuthenticationManager(AuthenticationManager manager){
        this.manager = manager;
    }

    @Override
    public void configure(AuthorizationServerEndpointsConfigurer endpoints) throws Exception {
        endpoints.authenticationManager(this.manager);
    }
}
```

- ***Register Client with authorization Server***





We can sum up these similarities in a few points that you can easily remember:

- `ClientDetails` is for the client what `UserDetails` is for the user.
- `ClientDetailsService` is for the client what `UserDetailsService` is for the user.
- `InMemoryClientDetailsService` is for the client what `InMemoryUserDetailsManager` is for the user.
- `JdbcClientDetailsService` is for the client what `JdbcUserDetailsManager` is for the user.

```
@Configuration
@EnableAuthorizationServer
public class AuthServerConfig
   extends AuthorizationServerConfigurerAdapter {

   // Omitted code

  @Override
  public void configure(
    ClientDetailsServiceConfigurer clients)
    throws Exception {

    var service = new InMemoryClientDetailsService();

    var cd = new BaseClientDetails();
    cd.setClientId("client");
    cd.setClientSecret("secret");
    cd.setScope(List.of("read"));
    cd.setAuthorizedGrantTypes(List.of("password"));

    service.setClientDetailsStore(
                Map.of("client", cd));

    clients.withClientDetails(service);
  }
}
```

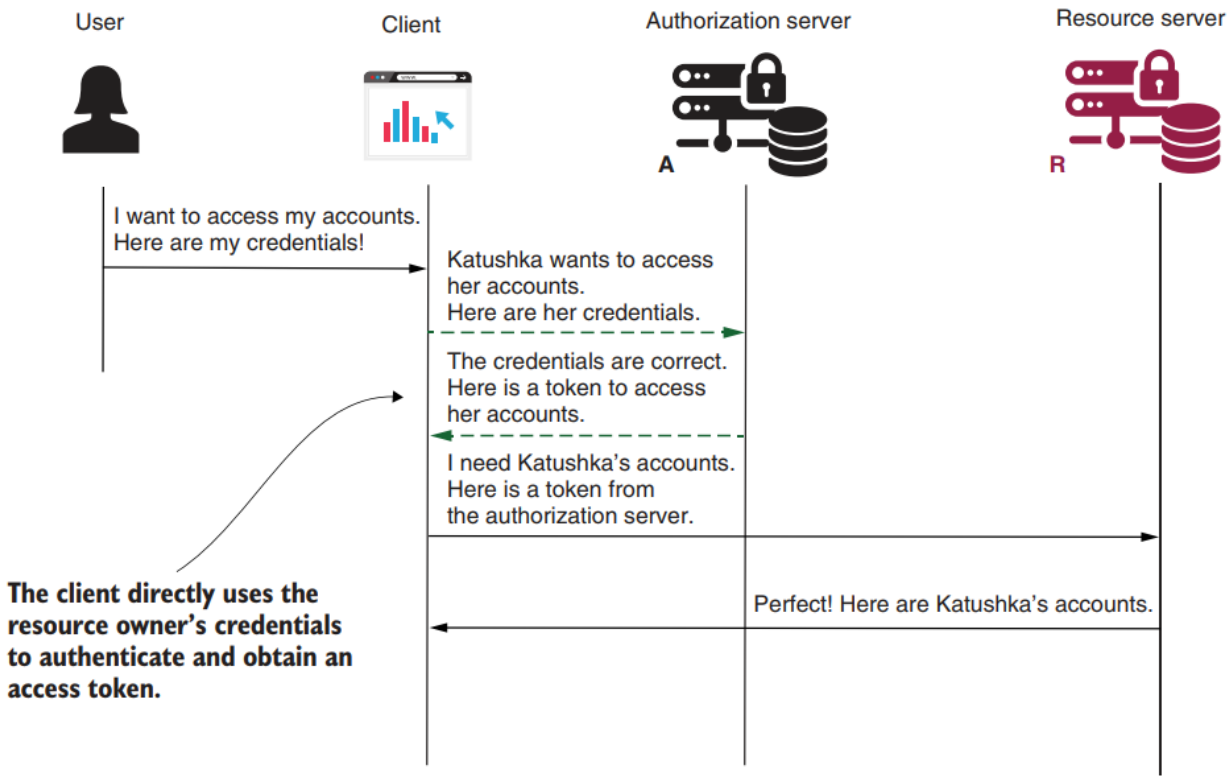
Overrides the configure() method to set up the ClientDetailsService instance

Creates an instance using the ClientDetailsService implementation

Creates an instance of ClientDetails and sets the needed details about the client

Adds the ClientDetails instance to InMemoryClientDetailsService

Configures ClientDetailsService for use by our authorization server



7. Testing API

7.1. Testing Overview

- **Unit testing**: Unit testing is performed by the developers to test the smallest unit (such as a class method) of code.
- **Integration testing**: Integration testing is performed by the developers to test the integration of different layers of components.
- **Contract testing**: Contract testing is performed by the developers to make sure any changes that are made to the API won't break the consumer code. The consumer code should always comply with the producer's contract (API). It is primarily required in microservices-based development.
- **End-to-End (E2E) testing**: E2E testing is performed by the **Quality Assurance** (**QA**) team to test end-to-end scenarios, such as from the UI (consumer) to the backend.
- **User Acceptance Testing (UAT)**: This is performed by the business users from a business perspective, and may overlap with E2E testing.

This extra assurance for high-quality deliverables need more time and effort. Therefore, software development cycles used to be huge in comparison to today's. **Time to market** (**TTM**) is a huge factor in today's competitive software industry. Today, you need faster release cycles. Moreover, quality checks, also known as testing, is an important and major part of release cycles.

> You already know that unit tests test the smallest testable code unit. But how can we write a unit test for controller methods? Controller runs on web servers and have the Spring web application context. If you write a test that uses `WebApplicationContext` and is running on top of a web server, then you can call it an integration test rather than a unit test.
>
> Unit tests should be lightweight and must be executed quickly. Therefore, you must use `MockMvc`, a special class provided by the Spring test library, to test the controllers. You can use the standalone setup for `MockMvc` for unit testing. You can also use `MockitoExtension` to run the unit test on the JUnit Platform (JUnit 5 provides an extension for runners) that supports object mocking and method stubbing. You will also use the Mockito library to mock the required dependencies. These tests are really fast and help developers build faster.

7.2. Testing Implementation

- ***Initialize Mock Test***

```
@ExtendWith(MockitoExtension.class)
public class MockitoBasic {
```

- ***Verify method have been called***

```
@Test
public void whenNotUseMock_thenCorrect(){
    List<String> mockList = Mockito.mock(ArrayList.class);
    mockList.add("one");
    Mockito.verify(mockList).add("one");
    assertEquals( expected: 0, mockList.size());
}
```

- ***Conditionals verify***

  - **times(number)**
  - **never()**
  - **atLeastOnce()**
  - **atLeast(number)**
  - **atMost(number)**
  - **verifyNoMoreInteractions(mockedObjects)**
  - **verifyZeroInteractions**

- ***@Psy***

```java
@Test
public void whenSpyingOnList_thenCorrect() {
    List<String> list = new ArrayList<String>();
    List<String> spyList = Mockito.spy(list);

    spyList.add("one");
    spyList.add("two");

    Mockito.verify(spyList).add("one");
    Mockito.verify(spyList).add("two");

    assertEquals( expected: 2, spyList.size());
    /*Note how the real method add() is actually called, and how the size of spyList becomes 2.*/
}
```

- ***Mock vs Spy in Mockito***

When Mockito creates a **mock**, it does so from the Class of a Type, not from an actual instance. The mock simply creates a bare-bones shell instance of the Class, entirely instrumented to track interactions with it.

On the other hand, the **spy** will wrap an existing instance. It will still behave in the same way as the normal instance, the only difference is that it will also be instrumented to track all the interactions with it.

- ***Mocking Exception***

```java
class MyDictionary {
    private Map<String, String> wordMap = new HashMap<>();
    public void add(String word, String meaning) {
        wordMap.put(word, meaning);
    }
    public String getMeaning(String word) {
        return wordMap.get(word);
    }
}

@Test
public void whenConfigNonVoidReturnMethodToThrowEx_thenExIsThrown() {
    MyDictionary dictMock = mock(MyDictionary.class);
    when(dictMock.getMeaning(anyString())).thenThrow(NullPointerException.class);
    assertThrows(NullPointerException.class, () -> {
        dictMock.getMeaning( word: "word");
    });
}
```

8. gRPC

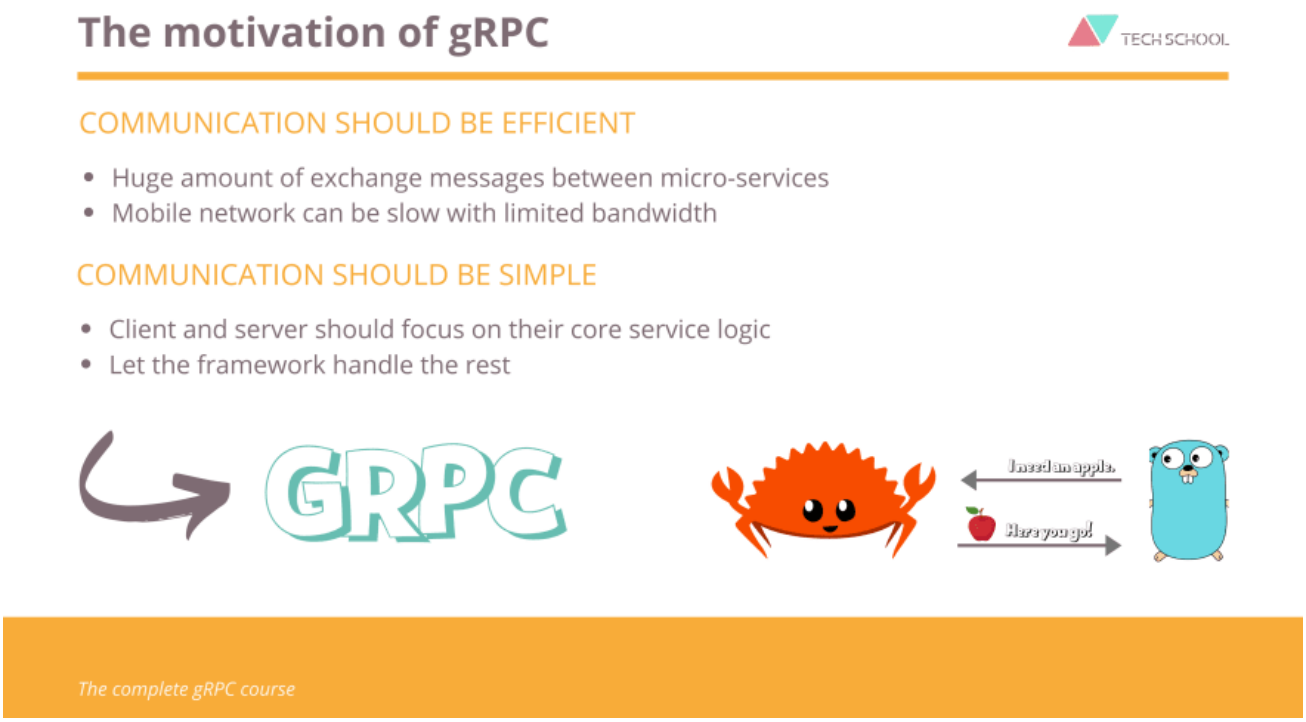
8.1. gRPC fundamentals

gRPC is an open source framework for general-purpose RPC across a network. gRPC supports full-duplex streaming and is also mostly aligned with HTTP/2 semantics. It supports different media formats, such as Protobuf (default), JSON, XML, and Thrift. The use of **Protocol Buffer** (**Protobuf**) aces the others because of higher performance.

- Scalability
- Load balacing and fail over
- Support cascade cacellation
- Offer wide communication from mobile app to server…

gRPC vs REST:

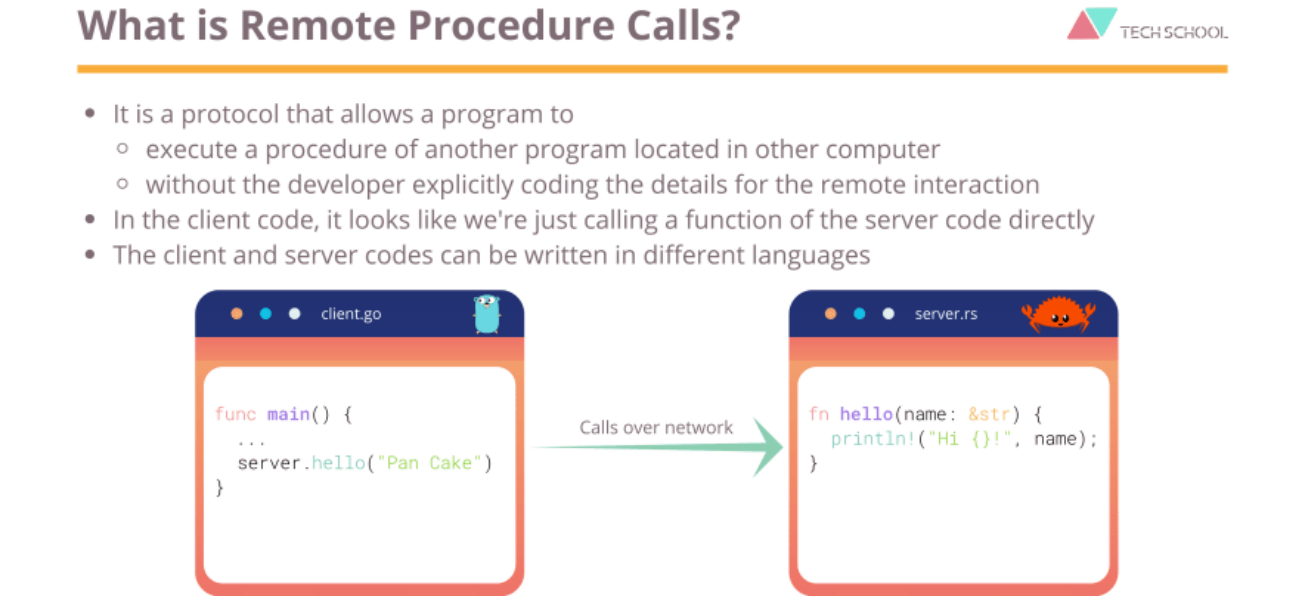
FEATURE	GRPC	REST
Protocol	HTTP/2 (fast)	HTTP/1.1 (slow)
Payload	Protobuf (binary, small)	JSON (text, large)
API contract	Strict, required (.proto)	Loose, optional (OpenAPI)
Code generation	Built-in (protoc)	Third-party tools (Swagger)
Security	TLS/SSL	TLS/SSL
Streaming	Bidirectional streaming	Client → server request only
Browser support	Limited (require gRPC-web)	Yes



How about RPC? RPC stands for Remote Procedure Calls. It is a protocol that allows a program to execute a procedure of another program located in other computer.

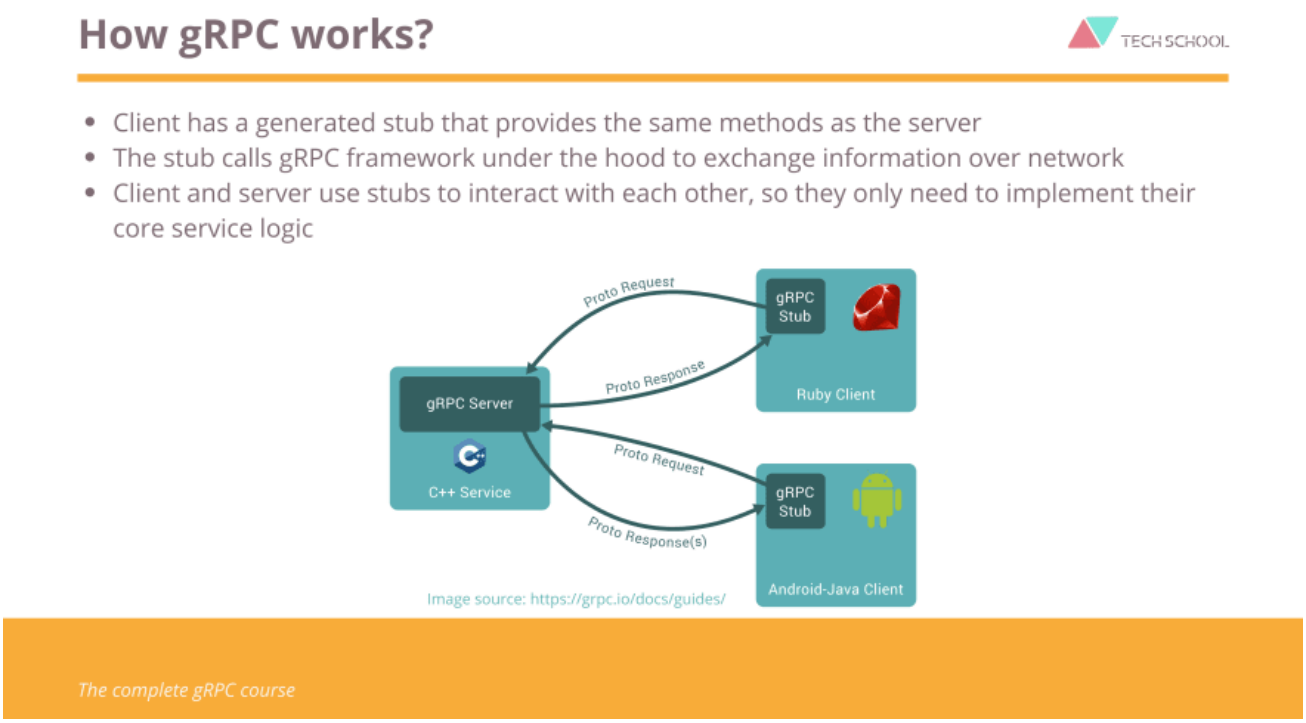
And what's awesome about it is that developers don't have to explicitly coding the details of the network interaction. It's automatically handled by the underlying framework.

So, in the client code, it looks like we're just **calling a function of the server code directly**. And it works even if the codes on client and server are written in different programming languages. Like in this example, the client code is written in Go, and the server code is written in Rust.



So how can gRPC do that? Basically, the client has a **stub** that provides the same method (or function) as the server. That stub is automatically generated for you by gRPC.

The **stub** will call gRPC framework under the hood to exchange information with the server over the network.



From this proto file, the server and client stub codes are generated by the **protocol buffer compiler (or protoc)**. Depending on the programming language, we will have to tell the compiler to use the correct gRPC plugin for it.

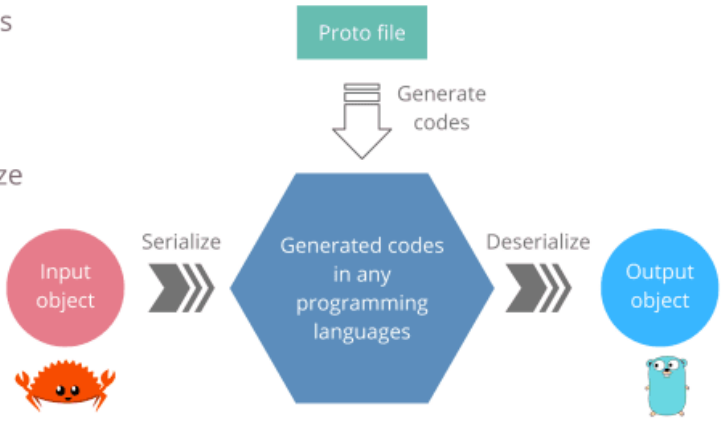
Why protobuf:

- Readable
- Automatic code gen support
- Smaller size then JSON

Why gRPC uses Protocol Buffer?

TECH SCHOOL

- Human-readable Interface Definition Language (IDL)
- Programming languages interoperable:
  - Code generators for many languages
- Binary data representation:
  - Smaller size
  - Faster to transport
  - More efficient to serialize / deserialize
- Strongly typed contract
- Conventions for API evolution
  - Backward & forward compatibility
- Alternative options
  - Google flatbuffers
  - Microsoft bond

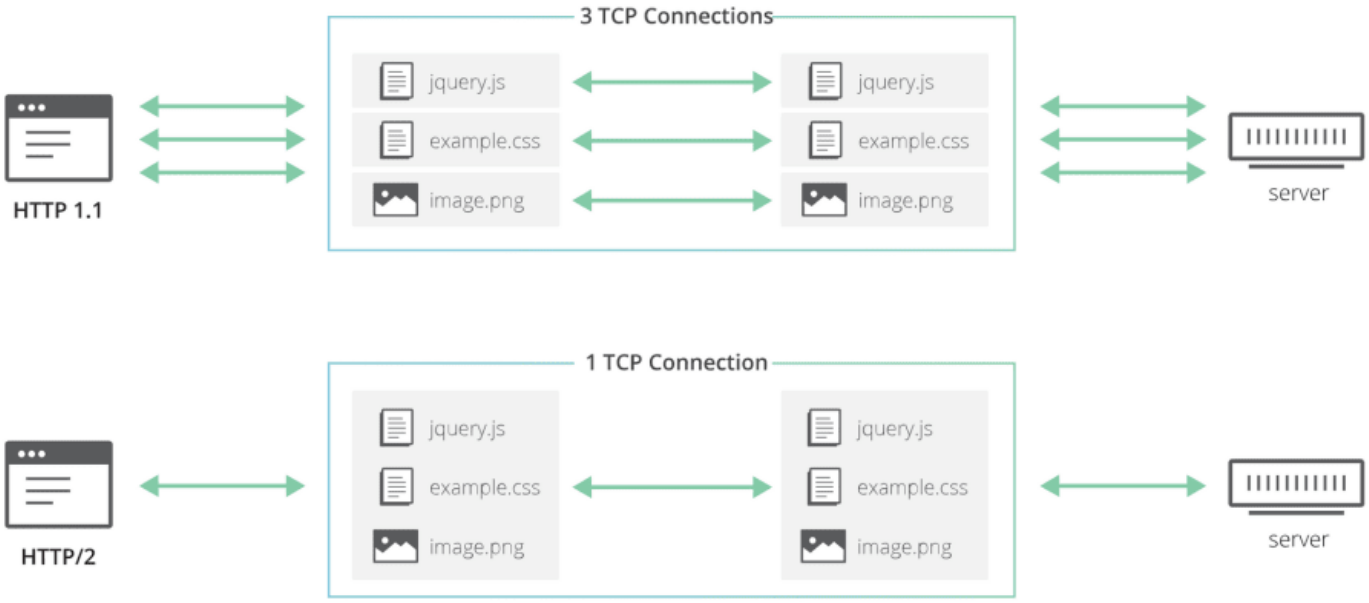


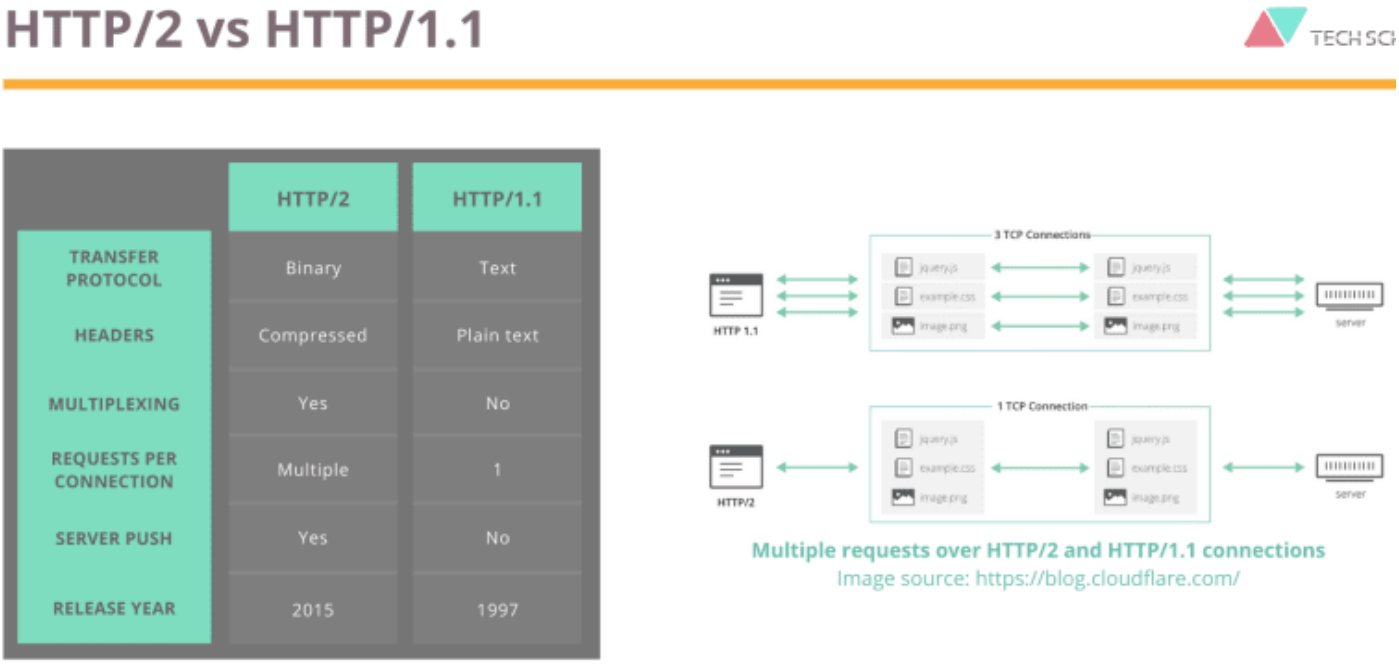
Summary
Motivation for gRPC: • Stricter API contract • Transefer data between microservices • HTTP 2 utilize • Low latency • Suport fail over, load balancing Remote procedural call: • Client or server call to **stub** as normal function the underline framework handle the jobs • Stub is define using procto file generate by protoc

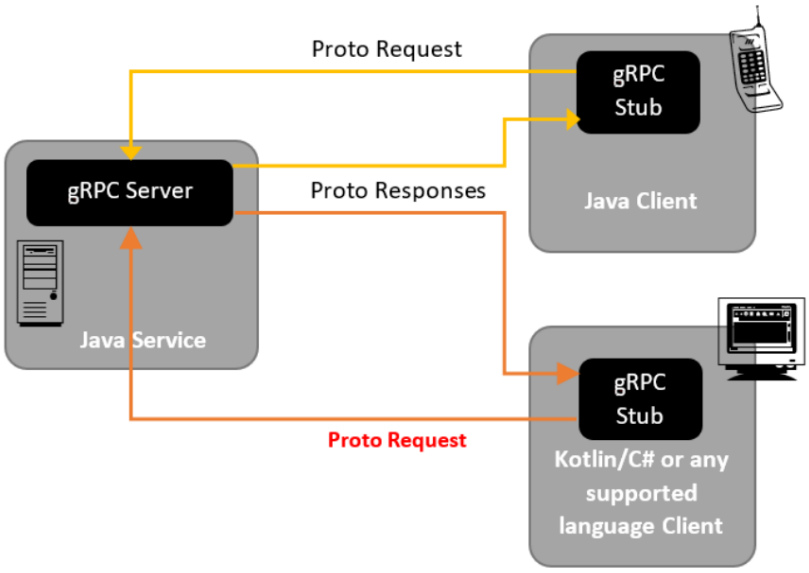
▶ *HTTP/2*

HTTP/2 is better than HTTP/1

- Support Server push
- Header compress with HPACK
- Multiplex, multiple request response under 1 TCP connection
- Binary framing

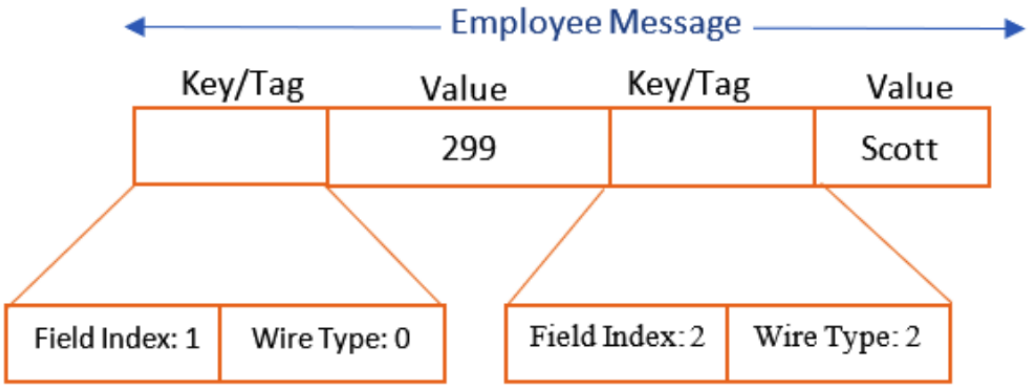






Layer or gRPC
Stub: define by .proto file
Channel: The stub uses Application Binary Interfaces (**ABIs**) for communicating with the server.
Transport: This is the lowest layer and uses HTTP/2 as its protocol.

- ***Protocol buf***



9. Cocurrency

9.1. Multi Threading theory

Multithreading

WHAT IS THE MOTIVATION BEHIND MULTITHREADING?

By default programming languages are **sequential**: they execute the statements and commands one by one (line by line)

```
public static void main(String[] args) {

    initializeArrays();
    downloadStocks();
    initalizeTimeSeriesModels();
    makePredictions();

}
```

In single threaded application these methods will be called one by one: we have to wait for them to finish one by one ...

Not the best solution possible: time consuming operations may freeze the application and the users may not know what's happening !!!

Multithreading

WHAT IS THE MOTIVATION BEHIND MULTITHREADING?

downloading images from the web — THREAD #1

IO operations: copying files and parse the content — THREAD #2

doin heavy calculations: simulations or numerical methods (differential equations) — THREAD #3

Multithreading

*„Multithreading is the ability of the **CPU** to execute multiple processes or threads concurrently"*

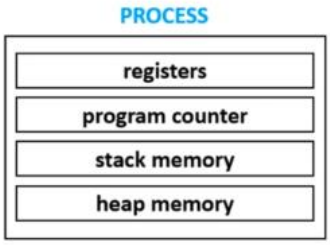
Both **processes** and **threads** are independent sequences of execution

PROCESS: a process is an instance of program execution

→ when you open a software or a web browser: these are distinct processes

→ the **OS** assigns distinct registers, stack memory and heap memory to every single process

in **Java** we can create processes with the help of **ProcessBuilder** class



Multithreading

*„Multithreading is the ability of the **CPU** to execute multiple processes or threads concurrently"*

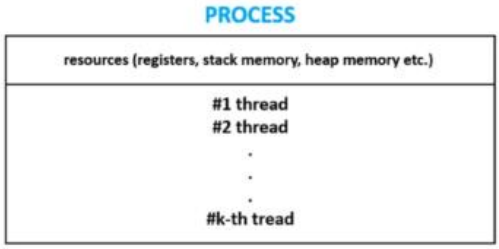
Both **processes** and **threads** are independent sequences of execution

THREADS: a thread is a „light-weight" process

→ It is a unit of execution within a given process (a process may have several threads)

→ each thread in a process shares the memory and resources and this is why programmers have to deal with concurrent programming and multithreading

creating a new thread requires fewer resources than creating a new process



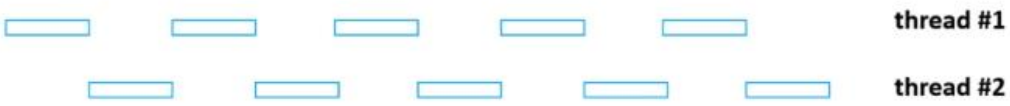
Multithreading

MULTITHREADING AND TIME-SLICING ALGORITHM

Assume we have **k** threads (so more than **1** thread in our application)

→ somehow the single processor has to deal with all of the **k** threads
~ one approach is to use time-slicing algorithm

„Processing time for a single core is shared among processes and threads. This is called time-slicing"



„multithreaded execution" (with time-slicing)

In this case the **CPU** will run **thread #1** for a small amount of time then **thread #2** then again **thread #1** and then **thread #2** and so on ...

Benefit	Drawbacks

Multithreading

WHAT IS THE MOTIVATION BEHIND MULTITHREADING?

downloading images from the web — THREAD #1

IO operations: copying files and parse the content — THREAD #2

doin heavy calculations: simulations or numerical methods (differential equations) — THREAD #3

→ we can design more responsive applications: we can do several operations concurrently

→ we can achieve better resource utilization (**CPU** utilization for example)
By default every **Java** program is single threaded: there may be several processor cores which we do not utilize without using multiple threads

→ overall we can improve performance

// performance improvement has something to do with multiple cores and parallel computing so we'll discuss it in that chapter

DISADVANTAGES

→ of course there are some costs involved in multithreading

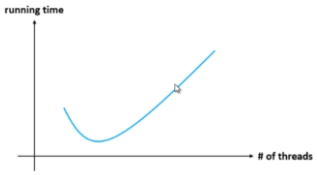
MULTITHREADING IS NOT ALWAYS BETTER !!!

→ threads manipulate data located on the same memory area because they belong to the same process and we have to deal with this fact (**synchronization**)

→ it's not easy to design multithreaded applications (easy to make bugs / hard to detect bugs)
~ with modern frameworks (fork-join) it's getting easier

→ EXPENSIVE OPERATION: switching between threads is expensive
CPU has to save local data, application pointer etc. of the current thread and has to load the data of the other thread as well

DISADVANTAGES



It's expensive to switch between threads: this is why using threads is not always the fastest way possible (for example sorting algorithms)

RULE OF THUMB: for small problems it is unncecessary to use multithreading

- ***Threading state***

Multithreading

THREAD STATES

1.) *NEW* when we instantiate a thread
It is in this state until we start it
~ **start()** method

2.) *RUNNABLE* after we have started the thread
The thread is executing its task in this state

3.) *WAITING* when a thread is waiting: for example for another thread to finish its task
When other thread signals then this thread goes back to the **runnable** state
~ **wait()** and **sleep()** methods

4.) *DEAD* after the thread finishes its task

In the previous lectures we've seen that we can create threads either with **Runnable interfaces** or with **Thread classes**. You may pose the question: what approach to prefer?

Usually using the Runnable interface approach is preferred.

- if we extends Thread then we can't extend any other class (usually a huge diasvantage) because in Java a given class can extends one class exclusively
- a class may implement more interfaces as well - so implementing the Runnable interface can do no harm in the software logic (now or in the future)

Memory Management

→ there are **processes** and **threads** (light-weight processes)

„The typical difference is that threads (of the same process) run in a shared memory space, while processes run in separate memory spaces"

STACK MEMORY	⟷	HEAP MEMORY
the local variables and method arguments as well as method calls are stored on the stack		objects are stored on the heap memory and live as long as it is referenced from somewhere in the application.

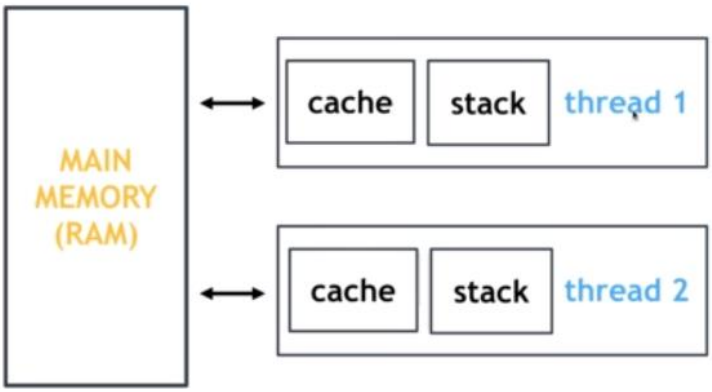
EVERY THREAD HAS ITS OWN STACK MEMORY BUT ALL THREADS SHARE THE HEAP MEMORY (SHARED MEMORY SPACE)

→ the main purpose of **synchronization** is the sharing of resources without interference using mutual exclusion.

Stack is individual for each thread, but heap can be shared.

→ there are **processes** and **threads** (light-weight processes)

„The typical difference is that threads (of the same process) run in a shared memory space, while processes run in separate memory spaces"



Intrinsic Lock (Monitor)

```
public synchronized void increment() {
    counter++;
}
```

→ every object in Java has a so-called intrinsic lock

"A thread that needs exclusive and consistent access to an object's fields has to acquire the object's intrinsic lock before accessing them, and then release the intrinsic lock when it's done with them"

→ because of the monitor lock: no **2** threads can execute the same **synchronized** method at the same time

→ a thread owns the intrinsic lock between the time it has acquired the lock and released the lock.

→ as long as a thread owns an intrinsic lock no other thread can acquire the same lock

→ the other thread will block when it attempts to acquire the lock.

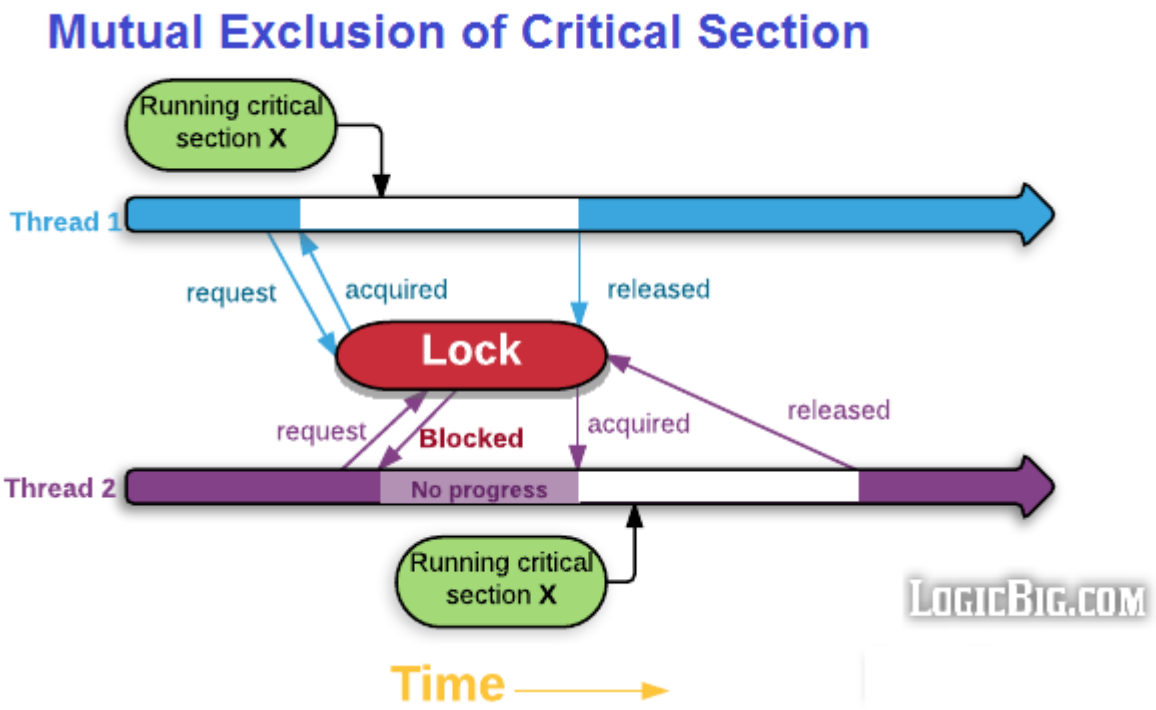
THE PROBLEM IS THAT EVERY OBJECT HAS A SINGLE MONITOR LOCK

→ if we have **2** independent **synchronized** methods than the threads have to wait for each other to release the lock

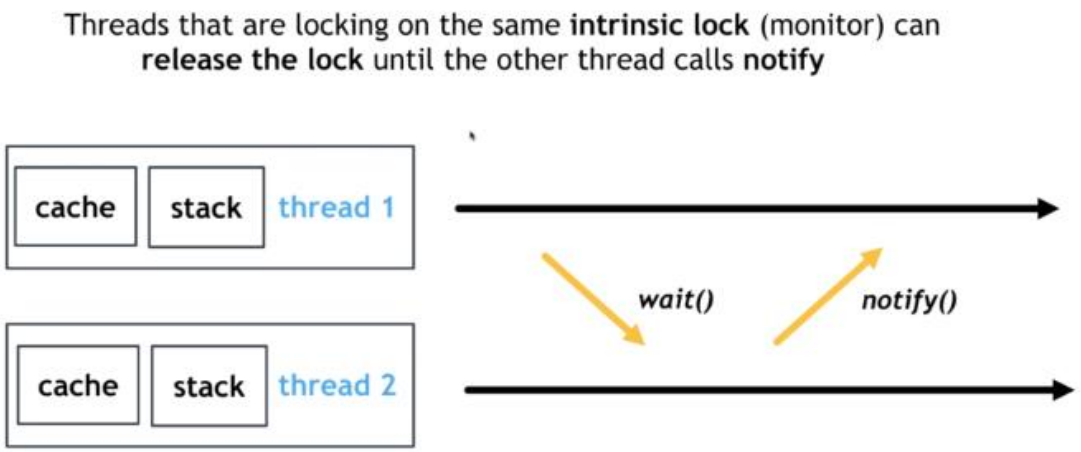
An **intrinsic lock (aka monitor lock) is an implicit internal entity** associated with **each instance of objects**.

The intrinsic lock enforces exclusive access to an object's state. Here 'access to an object' refers to an instance method call.

When a **synchronized** method is called from a thread it needs to acquire the intrinsic lock. The lock is release when the thread is done executing the method or an exception is thrown in the method which is not handled/caught.



- ***Wait and notify:***




```java
public class WaitNotify {

    public void produce() throws InterruptedException{
        synchronized (this){
            System.out.println("Running the produce method");
            wait();
            System.out.println("Return the produce method");
        }
    };
    public void consume() throws InterruptedException{
        Thread.sleep(500);
        synchronized (this){
            System.out.println("Running the consume method");
            notify();
        }
    };
    public static void main(String[] args) {
        WaitNotify waitNotify = new WaitNotify();

        Thread producer = new Thread(() -> {
            try {
                waitNotify.produce();
            } catch (InterruptedException exception) {
                exception.printStackTrace();
            }
        });
        Thread consume = new Thread(() -> {
            try {
                waitNotify.consume();
            } catch (InterruptedException exception) {
                exception.printStackTrace();
            }
        });
```

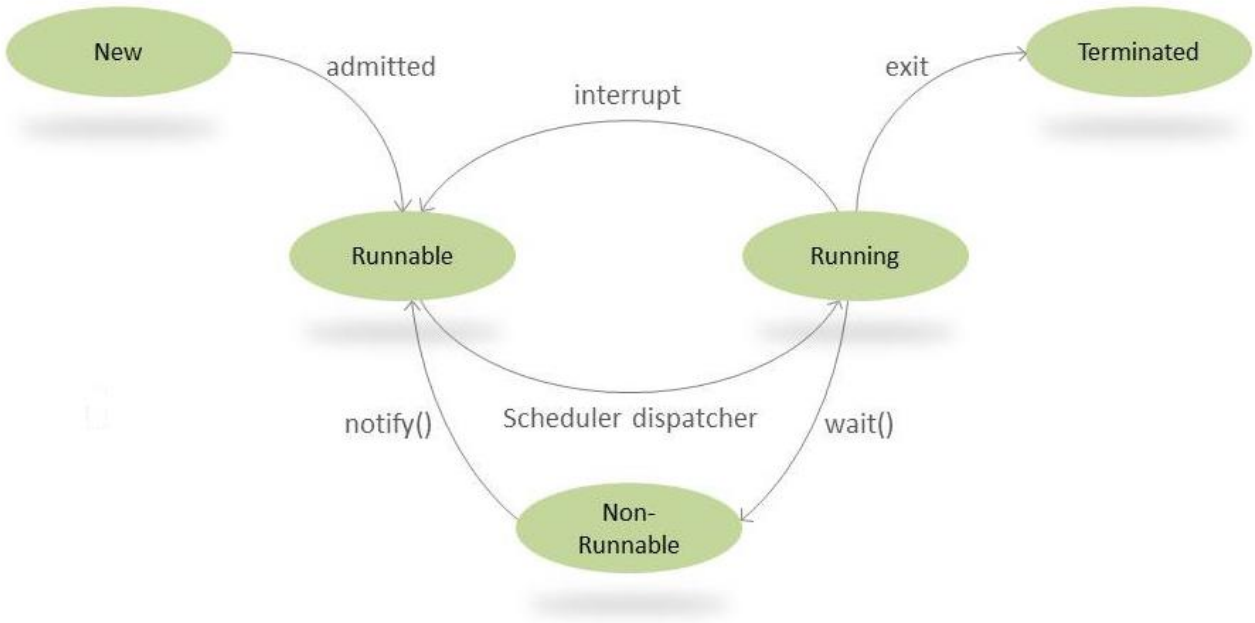
```
        producer.start();
        consume.start();
    }
}
```

One tool we can use to coordinate actions of multiple threads in Java is guarded blocks. Such blocks keep a check for a particular condition before resuming the execution.

With that in mind, we'll make use of the following:

- **Object.wait() to suspend a thread**
- **Object.notify() to wake a thread up**

We can better understand this from the following diagram depicting the life cycle of a Thread:




```
public class Processor {

    private List<Integer> list = new ArrayList<>();
    private final static int UPPER_LIMIT = 5;
    private final static int LOWER_LIMIT = 0;
    private static int counter;

    private final Object lock = new Object();

    public void producer() throws InterruptedException{
        synchronized (lock){
           while(true){
               if (list.size() == UPPER_LIMIT){
                   Thread.sleep(500);
                   System.out.println("Waiting for remove items");
```

```
                    lock.wait();
                } else {
                    Thread.sleep(500);
                    counter += 1;
                    list.add(counter);
                    System.out.printf("Adding %d to list \n", counter);
                    lock.notify(); // Keep waking up
                }
            }
        }
    };

    public void consumer() throws InterruptedException{
        synchronized (lock){
            while(true){
                if (list.size() == LOWER_LIMIT){
                    Thread.sleep(500);
                    counter = 0;
                    System.out.println("Waiting for adding items");
                    lock.wait();
                } else {
                    Thread.sleep(500);
                    System.out.printf("Remove %d from list \n",
                    list.remove(list.size()-1));
                    lock.notify(); // Keep waking up
                };
            }
        }
    };

    public static void main(String[] args) throws InterruptedException {
        Processor processor = new Processor();
        Thread producer = new Thread(() -> {
            try {
                processor.producer();
            } catch (InterruptedException exception) {
                exception.printStackTrace();
            }
        });
        Thread consume = new Thread(() -> {
            try {
                processor.consumer();
            } catch (InterruptedException exception) {
                exception.printStackTrace();
            }
        });
        producer.start();
        consume.start();
    }
}
```

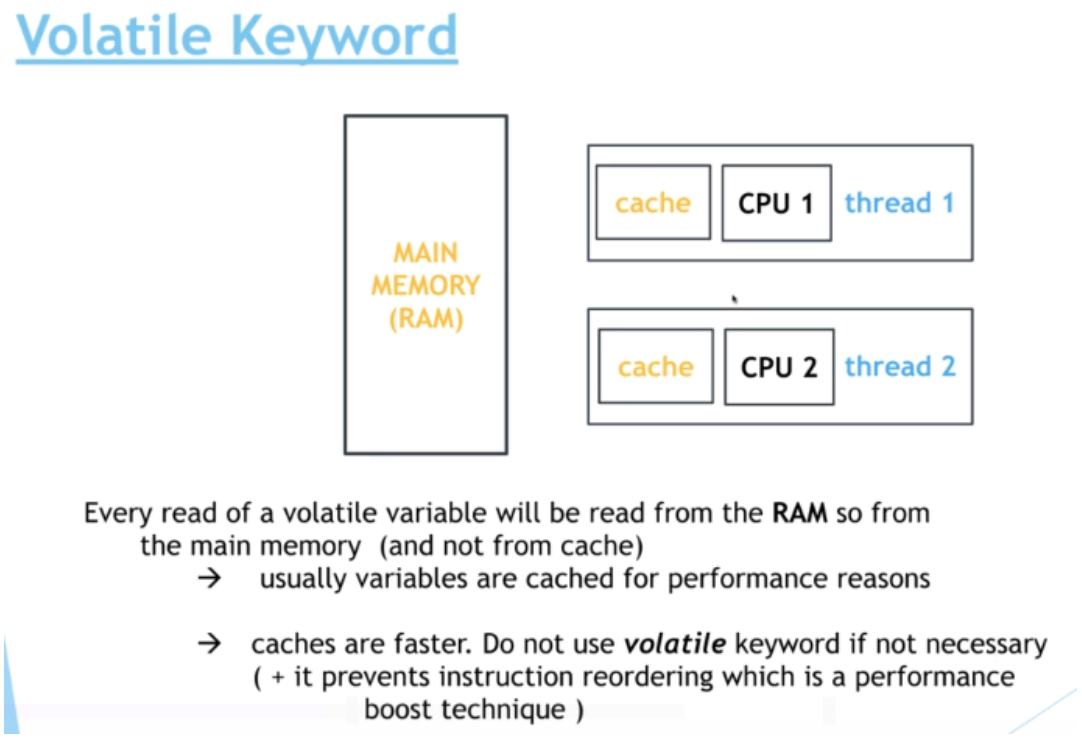
- ***Reentrant lock***

Behave the same as sync lock, but with enhance feature.

```
public class ReentrantLockTest {

    private final ReentrantLock lock = new ReentrantLock();
    private final Condition condition = lock.newCondition();

    private final List<Integer> list = new ArrayList<>();
    private final static int UPPER = 5;
    private final static int LOWER = 0;
    private final static int DELAY = 300;
    private int counter;

    public void producer() throws InterruptedException{
        while (true){
            lock.lock();
            if (list.size() == UPPER){
                Thread.sleep(DELAY);
                System.out.println("Array is full awaiting to remove");
                condition.await();
            } else {
                Thread.sleep(DELAY);
                list.add(counter);
                System.out.printf("Adding value %d to array \n",
                counter);
                counter++;
                condition.signal();
            }
            lock.unlock();
        }
    };

    public void consumer() throws InterruptedException{
        while (true){
            lock.lock();
            if (list.size() == LOWER){
                counter = 0;
                Thread.sleep(DELAY);
                System.out.println("Array is empty awaiting to add");
                condition.await();
            } else {
                Thread.sleep(DELAY);
                System.out.printf("Remove value %d to array \n",
                list.remove(list.size()-1));
                condition.signal();
            }
            lock.unlock();
        }
    };

    public static void main(String[] args) throws InterruptedException {
        ReentrantLockTest processor = new ReentrantLockTest();
        Thread producerLocks = new Thread(() -> {
```

```
            try {
                processor.producer();
            } catch (InterruptedException exception) {
                exception.printStackTrace();
            }
        });
        Thread consumeLocks = new Thread(() -> {
            try {
                processor.consumer();
            } catch (InterruptedException exception) {
                exception.printStackTrace();
            }
        });
        producerLocks.start();
        consumeLocks.start();
    }
}
```

- ***Votalite Keyword***



Volative fetch object from RAM not from cache.

► *Deadlock and livelock*

Deadlock

"Deadlock occurs when two or more threads wait forever for a lock or resource held by another of the threads"

→ **deadlock** is a situation in which two or more competing actions are each waiting for the other to finish, and thus neither ever does

1.) DEADLOCK IN DATABASES:

Deadlock happens when two processes each within its own transaction updates two rows of information but in the opposite order.

For example: process **A** updates row **1** then row **2** in the exact timeframe that process **B** updates row **2** then row **1**

Database deadlock:

```
CREATE OR REPLACE PROCEDURE doUpd ( id1 IN NUMBER, id2 IN NUMBER ) IS
BEGIN
   UPDATE tableA set colA = 'upd1' where id = id1;
   dbms_lock.sleep (20);
   UPDATE tableA set colA = 'upd2' where id = id2;
END;
/
```

Then run in session 1:

```
execute doUpd( 21, 12 );
```

Immediate in session 2:

```
execute doUpd( 12, 21 );
```

Session 1: `UPDATE tableA set colA = 'upd1' where id = 21;`

Session 1: `sleep 20`

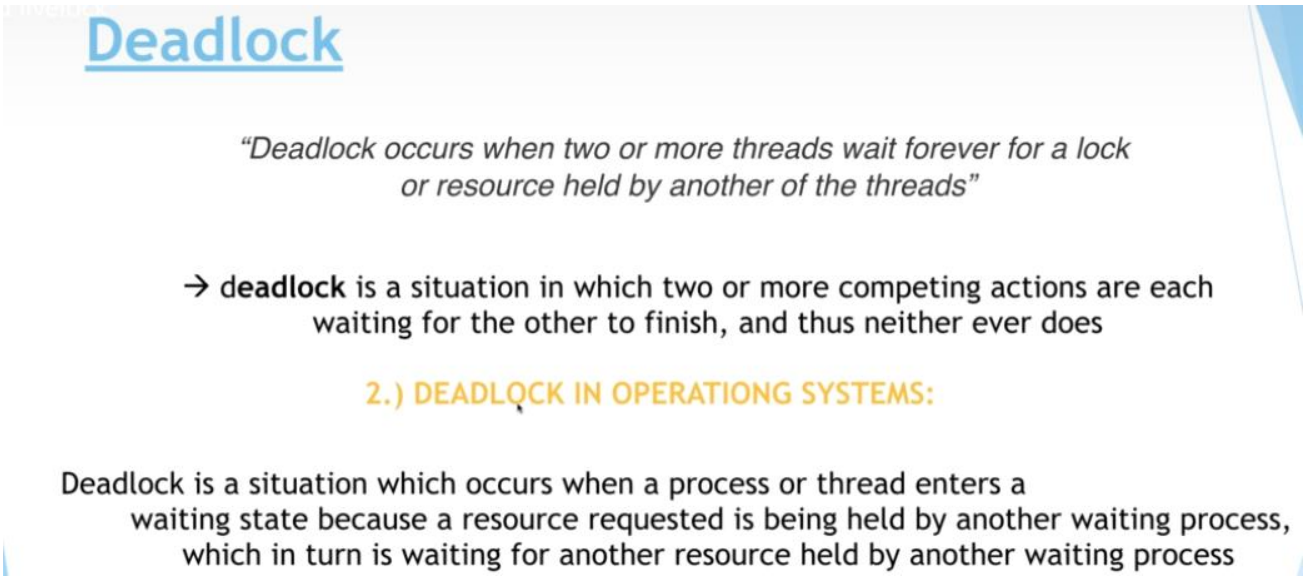
Session 2: `UPDATE tableA set colA = 'upd1' where id = 12;`

Session 2: `sleep 20`

Session 1: `UPDATE tableA set colA = 'upd2' where id = 12;` -- blocked until session 2 commit/rollback

Session 2: `UPDATE tableA set colA = 'upd2' where id = 21;` -- blocked until session 1 commit/rollback

Session 1 and 2 are now deadlocked.

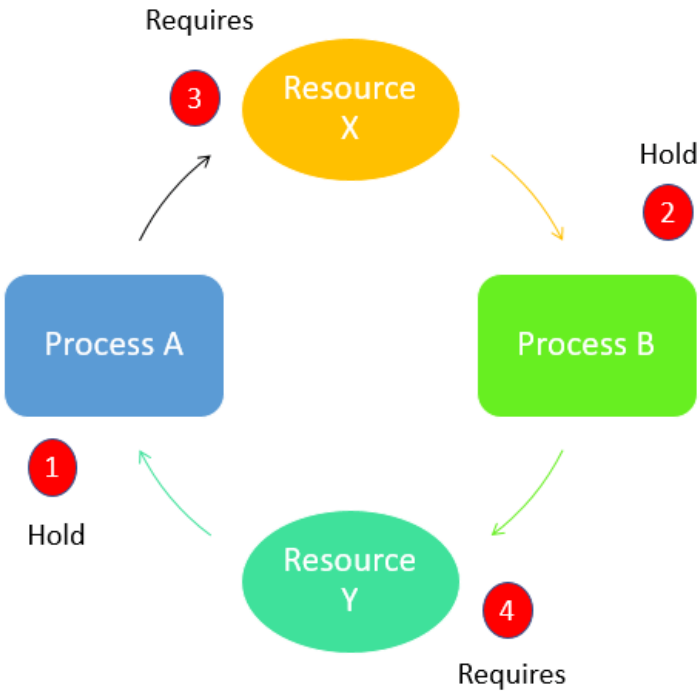


A **Livelock** is a situation where a request for an exclusive lock is denied repeatedly, as many overlapping shared locks keep on interfering each other. The processes keep on changing their status, which further prevents them from completing the task. This further prevents them from completing the task.

Example 1:

An easiest example of Livelock would be two people who meet face-to-face in a corridor, and both of them move aside to let the other pass. They end up moving from side to side without making any progress as they move the same way at the time. Here, they never cross each other.

Example 2:




```
public class DeadLock {
    Lock firstLock = new ReentrantLock();
    Lock secondLock = new ReentrantLock();

    private void firstProcess(){
        firstLock.lock();
        System.out.println("1st process acquire 1st lock");

        try {
            Thread.sleep(300);
        } catch (InterruptedException exception) {
            exception.printStackTrace();
        }
        secondLock.lock();
        System.out.println("1st process acquire 2nd lock");
        firstLock.unlock();
        secondLock.unlock();
    }

    private void secondProcess() {
        secondLock.lock();
        System.out.println("2nd process acquire 2nd lock");

        try {
            Thread.sleep(300);
        } catch (InterruptedException exception) {
            exception.printStackTrace();
        }
        firstLock.lock();
```

```
        System.out.println("2nd process acquire 1st lock");

        firstLock.unlock();
        secondLock.unlock();
    }

    public static void main(String[] args) {
        DeadLock deadLock = new DeadLock();
        new Thread(deadLock::firstProcess).start();
        new Thread(deadLock::secondProcess).start();
    }
}
```

- ***Atomic variable***

```
class SynchronizedCounter {
    private int c = 0;

    public synchronized void increment() {
        c++;
    }
    public synchronized void decrement() {
        c--;
    }
    public synchronized int value() {
        return c;
    }
}
```

Using Atomic variable.

```
class AtomicCounter {
    private AtomicInteger c = new AtomicInteger(0);

    public void increment() {
        c.incrementAndGet();
    }

    public void decrement() {
        c.decrementAndGet();
    }

    public int value() {
        return c.get();
    }

}
```

- ***Semaphores***

Semaphores

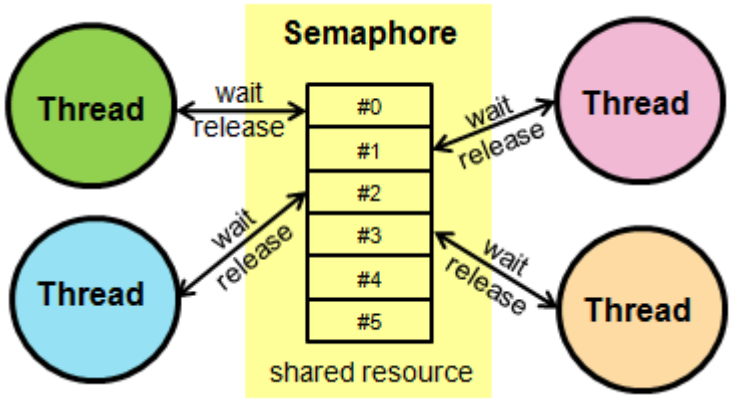
- invented by **Dijkstra** back in **1962**
- semaphores are **simple variables** (or abstract data types) that are used for controlling access to a common resource
- it is an important concept in operating systems as well

„It is a record of how many units of a particular resource are available. We have to wait until a unit of the resource becomes available again."

COUNTING SEMAPHORES: allows an arbitrary resource count

BINARY SEMAPHORES: semaphores that are restricted to the values **0** and **1**

Semaphores are used to manage and protect access to shared resources. Semaphores are very similar to Mutexes. Whereas a Mutex permits just one thread to access a shared resource at a time, a semaphore can be used to permit a fixed number of threads to access a pool of shared resources. Using semaphores, access to a group of identical peripherals can be managed (for example multiple DMA channels).



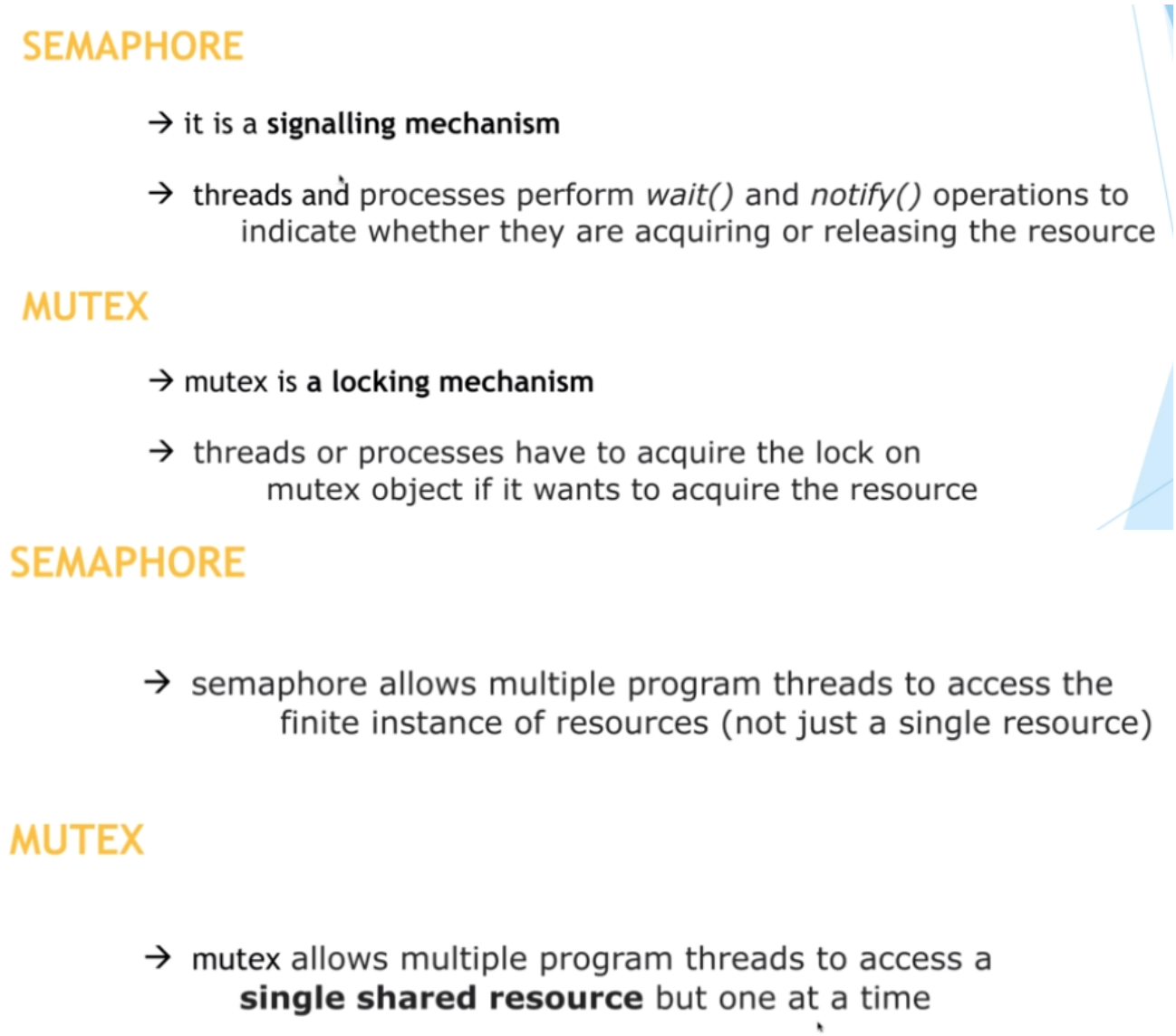
1.) semaphores tracks only **how many resources are free** - it does not keep track of which of the resources are free

2.) the semaphore count may serve as a useful **trigger** for a number of different actions (web servers)

3.) **producer-comsumer problem** can be solved and implemented with the help of semaphores (Dijkstra's approach)

- ***Mutex***

A mutex (or mutual exclusion) is the simplest type of synchronizer – it ensures that only one thread can execute the critical section of a computer program at a time.



- ***Callable interface and Runnable***

Runnable and **Callable** both run on a different threads than the calling thread but **Callable** can return a value and **Runnable** can not

RUNNABLE: a so-called *run-and-forget* action. We execute a given operation in the **run()** method without a return value

CALLABLE<T>: we use **Callable** interface's **call()** method if we want to return a given value from the given thread

→ **Callable** interface will not return the value: this is why we need **Future<T>** object

→ calling thread will be blocked till the **call()** method is executed and **Future<T>** returns with the results

The **ExecutorService** can handle both of the interfaces (**Runnable** and **Callable** interfaces)

executorService.execute()

This method executes a **Runnable** interface so it means there is no return value (void **run()** method)

executorService.submit()

This method can handle **Runnable** interfaces as well as **Callable** interfaces

→ it can handle a **Future<T>** return value and we can get the **T** value with **get()** on the future object

To summarize few notable Difference are

- A Runnable object does not return a result whereas a Callable object returns a result.
- A Runnable object cannot throw a checked exception wheras a Callable object can throw an exception.
- The Runnable interface has been around since Java 1.0 whereas Callable was only introduced in Java 1.5.

Few similarities include:

- Instances of the classes that implement Runnable or Callable interfaces are potentially executed by another thread.
- Instance of both Callable and Runnable interfaces can be executed by ExecutorService via submit() method.
- Both are functional interfaces and can be used in Lambda expressions since Java8.

► *Synchornized Collection*

```
public class ArraySync {
    private static final
    List<Integer> nums = Collections.synchronizedList(
                                            new ArrayList<>());


    public static void main(String[] args) throws InterruptedException {
        Thread firstThread = new Thread(() -> {
            for (int i = 0; i < 100; i++) {
                nums.add(i);
            }
        });
        Thread secondThread = new Thread(() -> {
            for (int i = 0; i < 100; i++) {
                nums.add(i);
            }
        });

        firstThread.start();
        secondThread.start();

        firstThread.join();
        secondThread.join();

        System.out.println("Final Array size :" + nums.size());
    }
}
```

Synchronize list is not very efficient.

► *Blocking queue*

A **Queue** that additionally supports operations that wait for the queue to become non-empty when retrieving an element, and wait for space to become available in the queue when storing an element.

BlockingQueue methods come in four forms, with different ways of handling operations that cannot be satisfied immediately, but may be satisfied at some point in the future:

- one throws an exception,

- the second returns a special value (either null or false, depending on the operation),
- the third blocks the current thread indefinitely until the operation can succeed,
- the fourth blocks for only a given maximum time limit before giving up.

These methods are summarized in the following table:

	Throws exception	*Special value*	*Blocks*	*Times out*
Insert	`add(e)`	`offer(e)`	`put(e)`	`offer(e, time, unit)`
Remove	`remove()`	`poll()`	`take()`	`poll(time, unit)`
Examine	`element()`	`peek()`	*not applicable*	*not applicable*

```
public class BlockingQueueTest implements Runnable{

    private final BlockingQueue<Integer> queue = new ArrayBlockingQueue<>(10);
    private boolean isFull = false;
    private int counter;

    public static void main(String[] args) throws InterruptedException {
        BlockingQueueTest test = new BlockingQueueTest();
        new Thread(test).start();
    }

    @Override
    public void run() {
        while (true){
            try {
                Thread.sleep(300);
            } catch (InterruptedException exception) {
                exception.printStackTrace();
            }
            if(!isFull){
                try {
                    if (queue.size() == 10){
                        isFull = true;
                    } else {
                        queue.put(counter);
                        System.out.println("Putting " + counter + " into queue");
                        counter++;
                    }
                } catch (InterruptedException exception) {
                    exception.printStackTrace();
                }
            } else {
                try {
                    if (queue.isEmpty()){
                        isFull = false;
                    } else {
                        System.out.println("Removing " + queue.take()
                        + " from queue");
                        counter--;
                    }
                } catch (InterruptedException exception) {
```

```
                    exception.printStackTrace();
                }
            }
        }
    }
}
```

▶ *Priority queue*

```
@Builder
@Data
class Item implements Comparable<Item> {
    private int size;

    public Item(int size) {
        this.size = size;
    }

    @Override
    public int compareTo(Item o) {
        return Integer.compare(size, o.size);
    }
}

public class PriorityBlockingQueueTest implements Runnable{
    private static final PriorityBlockingQueue<Item> queue =
                    new PriorityBlockingQueue<>(10);

    private boolean isFull = false;
    private int counter;

    public static void main(String[] args) throws InterruptedException {
        PriorityBlockingQueueTest test = new PriorityBlockingQueueTest();
        new Thread(test).start();
    }


    @Override
    public void run() {
        while (true){
            try {
                Thread.sleep(300);
            } catch (InterruptedException exception) {
                exception.printStackTrace();
            }
            if(!isFull){
                if (queue.size() == 10){
                    isFull = true;
                } else {
                    queue.put(Item.builder().size(counter).build());
                    System.out.println("Putting Item size [" + counter
                    + "] into queue");
                    counter++;
                }
            } else {
                try {
                    if (queue.isEmpty()){
```

```
                    isFull = false;
                } else {
                    System.out.println("Removing " + queue.take()
                    + " from queue");
                    counter--;
                }
            } catch (InterruptedException exception) {
                exception.printStackTrace();
            }
        }
    }
}
```

- ***Copy on Write array***

```
List<T> list = new CopyOnWriteArray<>();
```

*this is the efficient implementation of the synchronized **ArrayList***

ANY THREAD CAN READ FROM THE ACTUAL VERSION OF THE LIST !!!

*there is **no need for locking** when reading data from the list*

*this is the efficient implementation of the synchronized **ArrayList***

THREADS THAT CHANGE THE VALUE IN THE LIST MAKE COPY OF THE LIST – O(N) !!!

*this is how the update (remove or set operations) will be **atomic** – threads must wait for each other to update the list*