Spring Framework

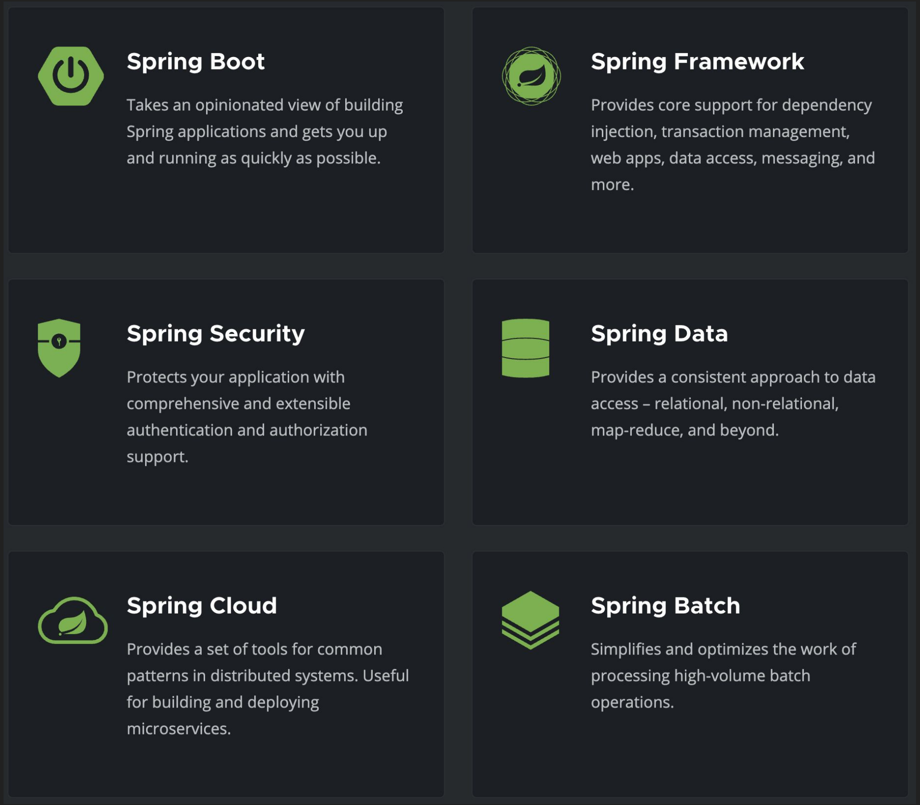
Agenda

Overview

- Spring is the most popular application development framework that provides a comprehensive programming and configuration model for modern Java-based enterprise applications - on any kind of deployment platform.

- First version was released at October 2002
- Latest version is 5.3.3 (as of January 2021)

Spring Platform projects, more at [Spring Projects]



Spring Boot

Takes an opinionated view of building Spring applications and gets you up and running as quickly as possible.

Spring Framework

Provides core support for dependency injection, transaction management, web apps, data access, messaging, and more.

Spring Security

Protects your application with comprehensive and extensible authentication and authorization support.

Spring Data

Provides a consistent approach to data access – relational, non-relational, map-reduce, and beyond.

Spring Cloud

Provides a set of tools for common patterns in distributed systems. Useful for building and deploying microservices.

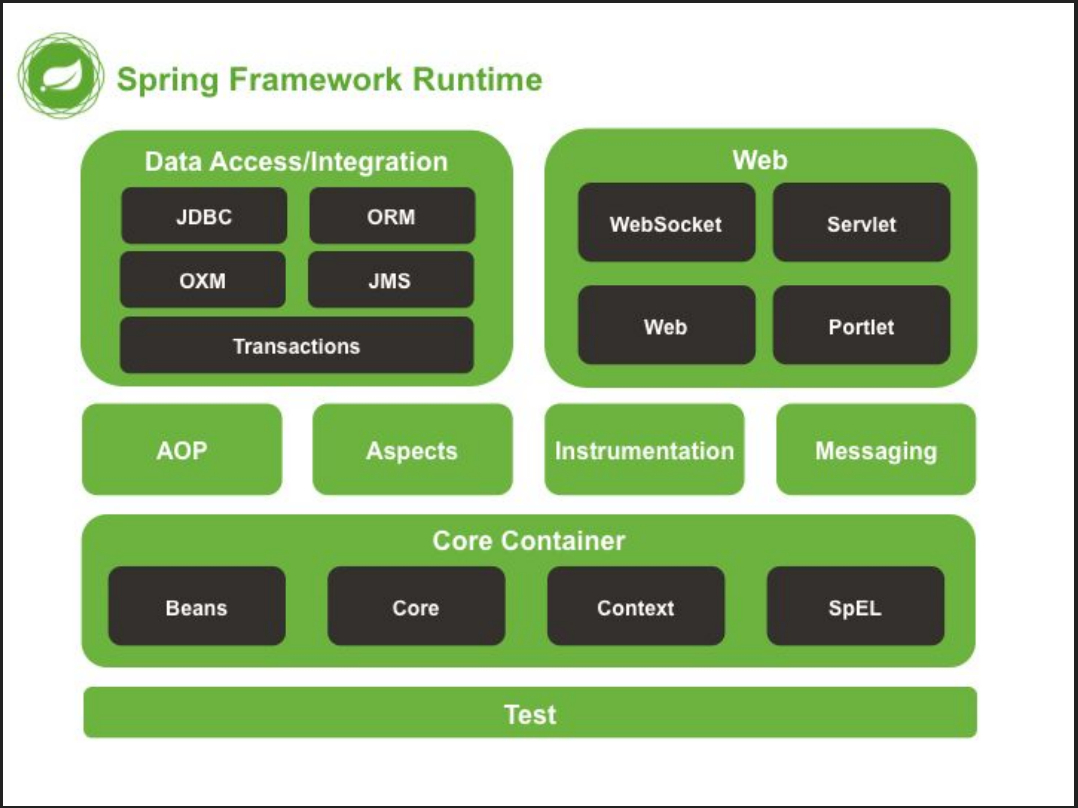
Spring Batch

Simplifies and optimizes the work of processing high-volume batch operations.

Spring Framework Features

- Core technologies: dependency injection, events, resources, i18n, validation, data binding, type conversion, SpEL, AOP.
- Testing: mock objects, TestContext framework, Spring MVC Test, WebTestClient.
- Data Access: transactions, DAO support, JDBC, ORM.
- Spring MVC and Spring WebFlux web frameworks.
- Integration: email, tasks, scheduling, cache.
- Languages: Kotlin, Groovy, dynamic languages.

Spring Framework Runtime



Spring Framework - Core Container

- The Core module provides the fundamental parts of the framework, including the IoC and Dependency Injection features.
- The Bean module provides BeanFactory which is a sophisticated implementation of the factory pattern.
- The Context module builds on the solid base provided by the Core and Beans modules and it is a medium to access any objects defined and configured.
- The Expression Language module provides a powerful expression language for querying and manipulating an object graph at runtime.

Spring Framework - Data Access

- The JDBC module provides a JDBC-abstraction layer that removes the need to do tedious JDBC related coding.
- The ORM module provides integration layers for popular object-relational mapping APIs, including JPA, JDO, Hibernate, and iBatis.
- The Transaction module supports programmatic and declarative transaction management.

Spring Framework - WEB

- Spring's Web MVC (model-view-controller) provides basic web-oriented integration features such as multipart file-upload functionality and the initialization of the IoC container using servlet listeners and a web-oriented application context, also provides a clean separation between domain model code and web forms
- The WebFlux reactive-stack web framework, Spring WebFlux, was added later in version 5.0. It is fully non-blocking, supports Reactive Streams, and runs on such servers as Netty, Undertow, and Servlet 3.1+ containers.

Spring AOP

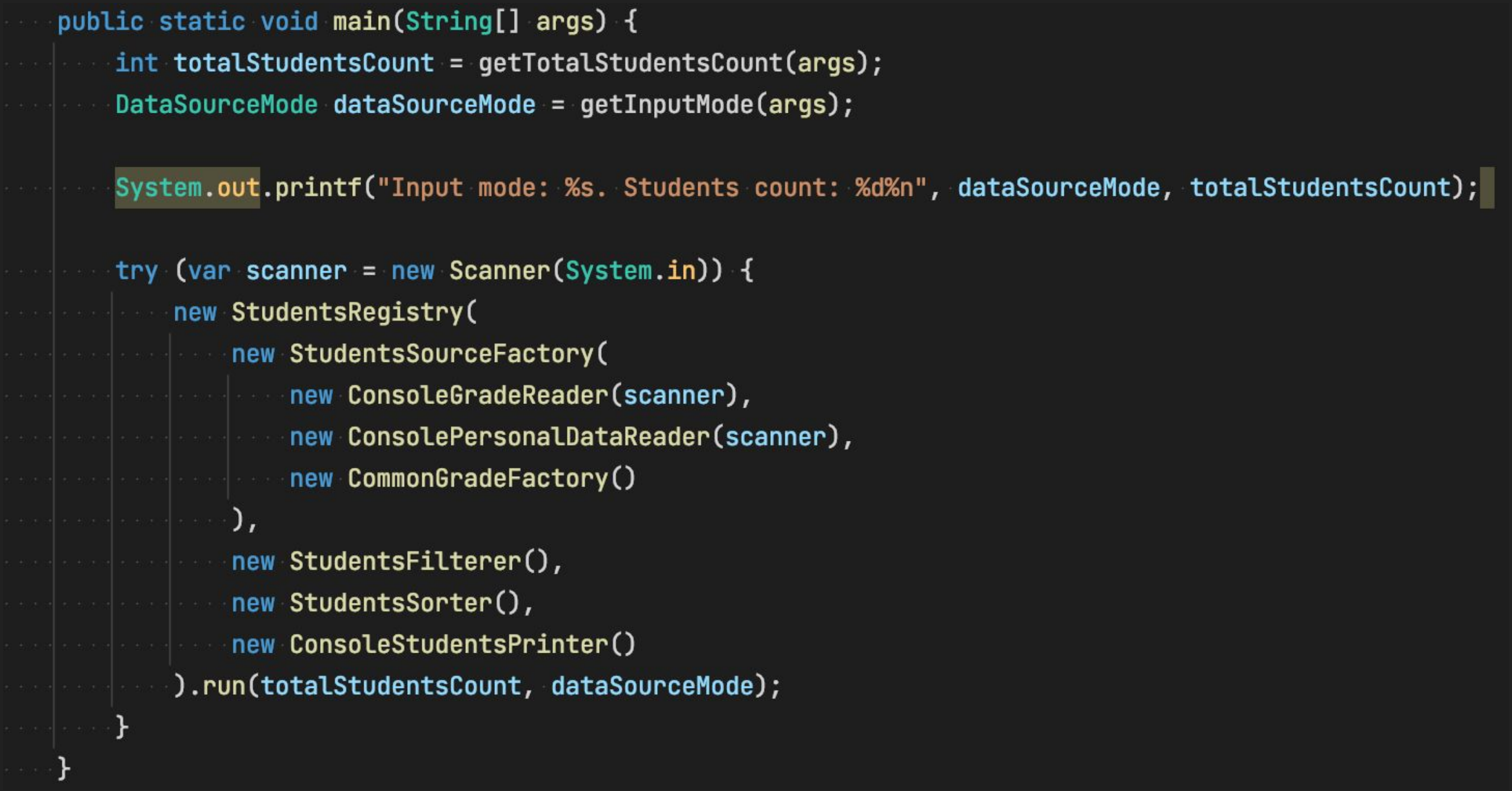
- Spring' AOP module provides an AOP Alliance-compliant aspect-oriented programming implementation allowing you to define, for example, method-interceptors and pointcuts to cleanly decouple code that implements functionality that should be separated.
- The separate Aspects module provides integration with AspectJ.

Example - Hello World

https://github.com/vrudas/spring-framework-examples/tree/main/example-00-hello

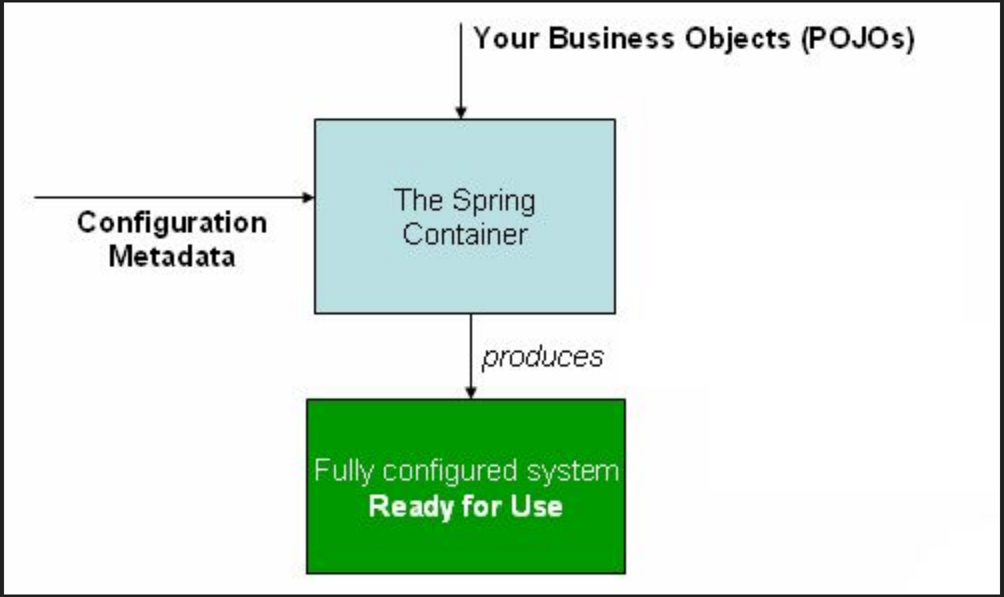
Inversion of Control - The Problem?

Inversion of Control - The Problem?



```java
    public static void main(String[] args) {
        int totalStudentsCount = getTotalStudentsCount(args);
        DataSourceMode dataSourceMode = getInputMode(args);

        System.out.printf("Input mode: %s. Students count: %d%n", dataSourceMode, totalStudentsCount);

        try (var scanner = new Scanner(System.in)) {
            new StudentsRegistry(
                new StudentsSourceFactory(
                    new ConsoleGradeReader(scanner),
                    new ConsolePersonalDataReader(scanner),
                    new CommonGradeFactory()
                ),
                new StudentsFilterer(),
                new StudentsSorter(),
                new ConsoleStudentsPrinter()
            ).run(totalStudentsCount, dataSourceMode);
        }
    }
```

Inversion of Control



Inversion of Control in Spring

1. The Spring container is at the core of the Spring Framework.
2. The Spring container uses dependency injection (DI) to manage the components that make up an application.
3. The container will create the objects, wire them together, configure them, and manage their complete lifecycle from creation till destruction.
4. The container gets its instructions on what objects to instantiate, configure, and assemble by reading configuration metadata provided. The configuration metadata can be represented either by XML, Java annotations, or Java code.

Dependency Injection Containers

- Spring BeanFactory Container - this is the simplest container providing basic support for DI. There are a number of implementations of the BeanFactory interface that come supplied straight out-of-the-box with Spring. The most commonly used BeanFactory implementation is the XmlBeanFactory class.
- Spring ApplicationContext Container - includes all functionality of the BeanFactory, and adds more enterprise-specific functionality such as the ability to resolve textual messages from a properties file and the ability to publish application events to interested event listeners.

Example - Containers

https://github.com/vrudas/spring-framework-examples/tree/main/example-01-bean-factory

What is Bean?

Beans

- The objects that form the backbone of your application and that are managed by the Spring IoC container are called beans.

- A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container. These beans are created with the configuration metadata that you supply to the container, for example, in the form of XML <bean/> definitions which you have already seen in previous chapters.

Beans - Definition

The bean definition contains the information called configuration metadata which is needed for the container to know the followings:

- How to create a bean
- Bean's lifecycle details
- Bean's dependencies

Beans - Definition

Property	Description
class	The bean class to be used to create the bean.
name	The unique bean identifier.
scope	The scope of the objects created from a particular bean definition.
lazy-initialization mode	Tells the IoC container to create a bean instance when it is first requested, rather than at startup.
constructor-args	Used to inject the dependencies into the class through a class constructor
properties	Used to inject the dependencies into the class through setter methods
initialization method	A callback to be called just after all necessary properties on the bean have been set by the container.
destruction method	A callback to be used when the container containing the bean is destroyed.

Example - Bean Definition

https://github.com/vrudas/spring-framework-examples/tree/main/example-02-bean-definition

Beans - Scopes

Property	Description
singleton	This scopes the bean definition to a single instance per Spring IoC container (default).
prototype	This scopes a single bean definition to have any number of object instances.
request*	This scopes a bean definition to an HTTP request.
session*	This scopes a bean definition to an HTTP session.

Example - Bean Scope

https://github.com/vrudas/spring-framework-examples/tree/main/example-03-bean-scope

Beans - Lifecycle

The life cycle of a Spring bean is clear to understand.

When a bean is instantiated, it may be required to perform some initialization to get it into a usable state.

When the bean is no longer required and is removed from the container, some cleanup may be required.

Beans - Lifecycle - Initialization

- The org.springframework.beans.factory.InitializingBean interface specifies a single method:

  ```
  void afterPropertiesSet() throws Exception;
  ```

- In the XML-based configuration metadata, you can use the init-method attribute to specify the name of the method that has a void no-argument signature:

  ```
  <bean id="..." class="..." init-method="init"/>
  ```

- Annotate the method with @PostConstruct:

  ```
  @PostConstruct
  public void init() {
      ...
  }
  ```

Beans - Lifecycle - Destruction

- The org.springframework.beans.factory.DisposableBean interface specifies a single method:

```
void destroy() throws Exception;
```

- In the XML-based configuration metadata, you can use the init-method attribute to specify the name of the method that has a void no-argument signature:

```
<bean id="..." class="..." destroy-method="destroy"/>
```

- Annotate the method with @PreDestroy:

```
@PreDestroy

public void destroy() {

    ...

}
```

Beans - Multiple Lifecycle Mechanisms

Multiple lifecycle mechanisms configured for the same bean are called in the following order:

- Initialization:
  - Methods annotated with @PostConstruct
  - afterPropertiesSet() as defined by the InitializingBean callback interface
  - A custom configured init() method
- Destruction:
  - Methods annotated with @PreDestroy
  - destroy() as defined by the DisposableBean callback interface
  - A custom configured destroy() method

Example - Beans Lifecycle

https://github.com/vrudas/spring-framework-examples/tree/main/example-04-bean-lifecycle

Dependency Injection

When writing a complex Java application, application classes should be as independent as possible of other Java classes to increase the possibility to reuse these classes and to test them independently of other classes while doing unit testing.

Dependency Injection (or sometime called wiring) helps in gluing these classes together and same time keeping them independent.

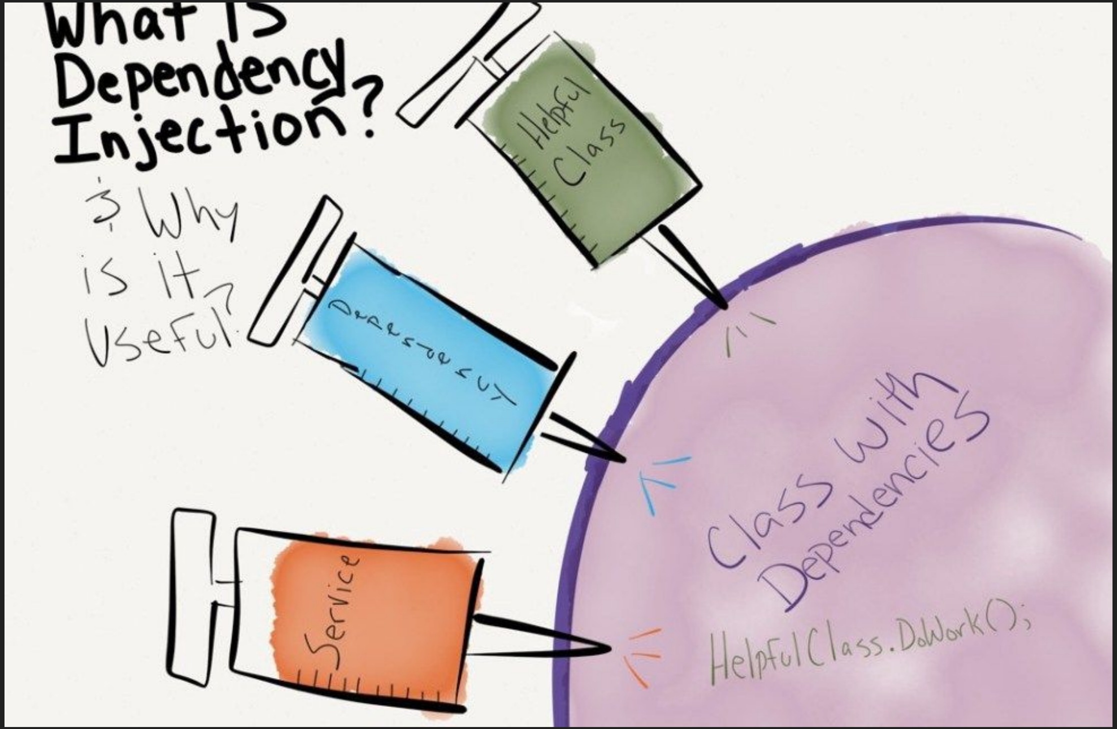
Dependency Injection - The Problem?

Dependency Injection

When writing a complex Java application, application classes should be as independent as possible of other Java classes to increase the possibility to reuse these classes and to test them independently of other classes while doing unit testing.

Dependency Injection helps in combining these classes together and same time keeping them independent.

Dependency Injection



Dependency Injection Types

- Constructor-based DI - is accomplished when the container invokes a class constructor with a number of arguments, each representing a dependency on other class.
- Setter-based DI - is accomplished by the container calling setter methods on your beans after invoking a no-argument constructor or no-argument static factory method to instantiate your bean.

Example - Dependency Injection

https://github.com/vrudas/spring-framework-examples/tree/main/example-05-dependency-injection

Annotation Based Configuration (since Spring 2.5)

@Autowired	Marks a constructor, field, setter method, or config method as to be autowired by Spring's dependency injection facilities.
@Qualifier	used on a field or parameter as a qualifier for candidate beans when autowiring.
@Component	Indicates that an annotated class is a "component". Such classes are considered as candidates for auto-detection when using annotation-based configuration and classpath scanning.
@Service	Indicates that a class is used for code of a "Business Logic". This annotation is a general-purpose stereotype and individual teams may narrow their semantics and use as appropriate.
@Repository	Indicates that an annotated class is a "Repository" - a mechanism for encapsulating storage, retrieval, and search behavior which emulates a collection of objects".
JSR-250 Annotations	Spring supports JSR-250 based annotations which include @Resource, @PostConstruct and @PreDestroy annotations.

Example - Annotation Based Configuration

https://github.com/vrudas/spring-framework-examples/tree/main/example-06-annotation-config

Java Based Configuration

- Framework independent approach without XML usage

Operates with additional annotations:

- @Configuration indicates that the class can be used by the Spring IoC container as a source of bean definitions.
- @Bean annotation tells Spring that a method annotated with @Bean will return an object that should be registered as a bean in the Spring application context.

Example - Java Based Configuration

https://github.com/vrudas/spring-framework-examples/tree/main/example-07-java-config

Properties

Example - Properties

https://github.com/vrudas/spring-framework-examples/tree/main/example-08-properties

Spring MVC

Spring Web MVC Framework

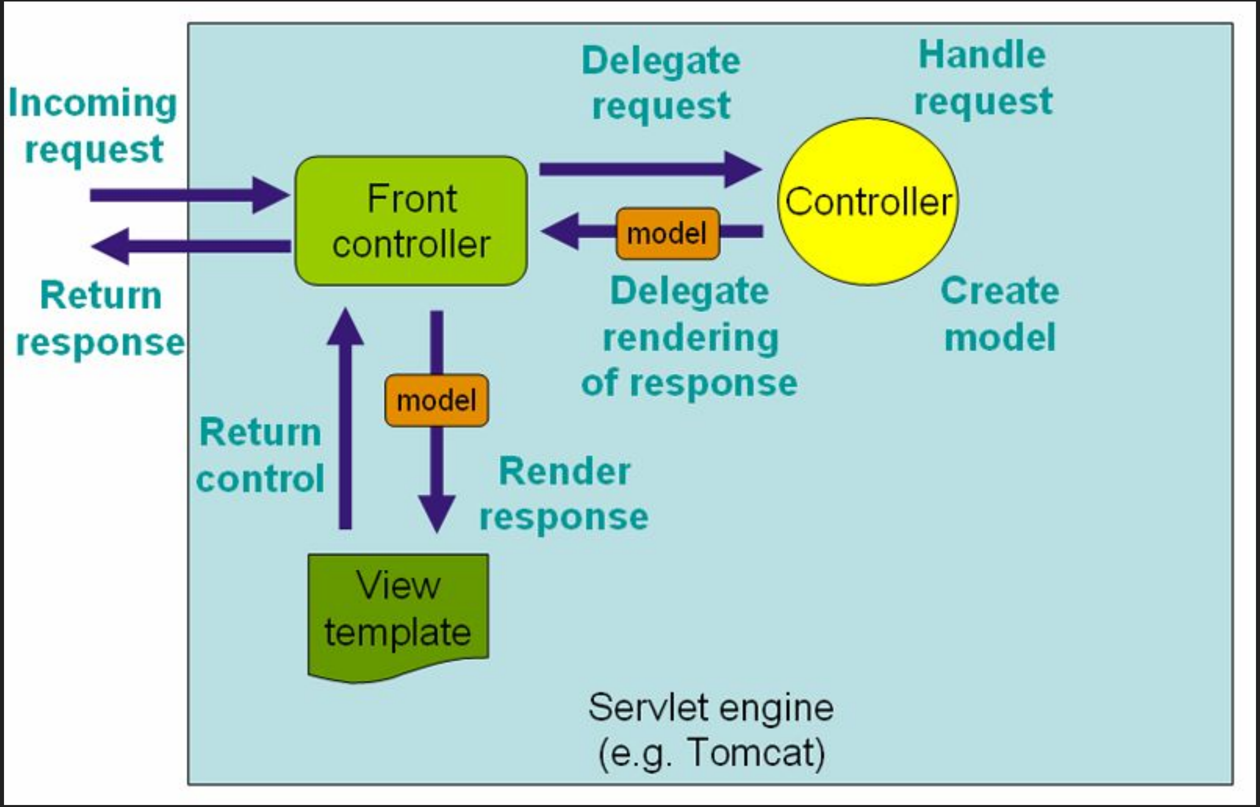
The Spring web MVC framework provides model-view-controller architecture and ready components that can be used to develop flexible and loosely coupled web applications.

The MVC pattern results in separating the different aspects of the application (input logic, business logic, and UI logic), while providing a loose coupling between these elements.

Spring Web MVC Framework

- The Model encapsulates the application data and in general they will consist of POJO.
- The View is responsible for rendering the model data and in general it generates HTML output that the client's browser can interpret.
- The Controller is responsible for processing user requests and building appropriate model and passes it to the view for rendering.

DispatcherServlet (aka Front Controller design pattern)



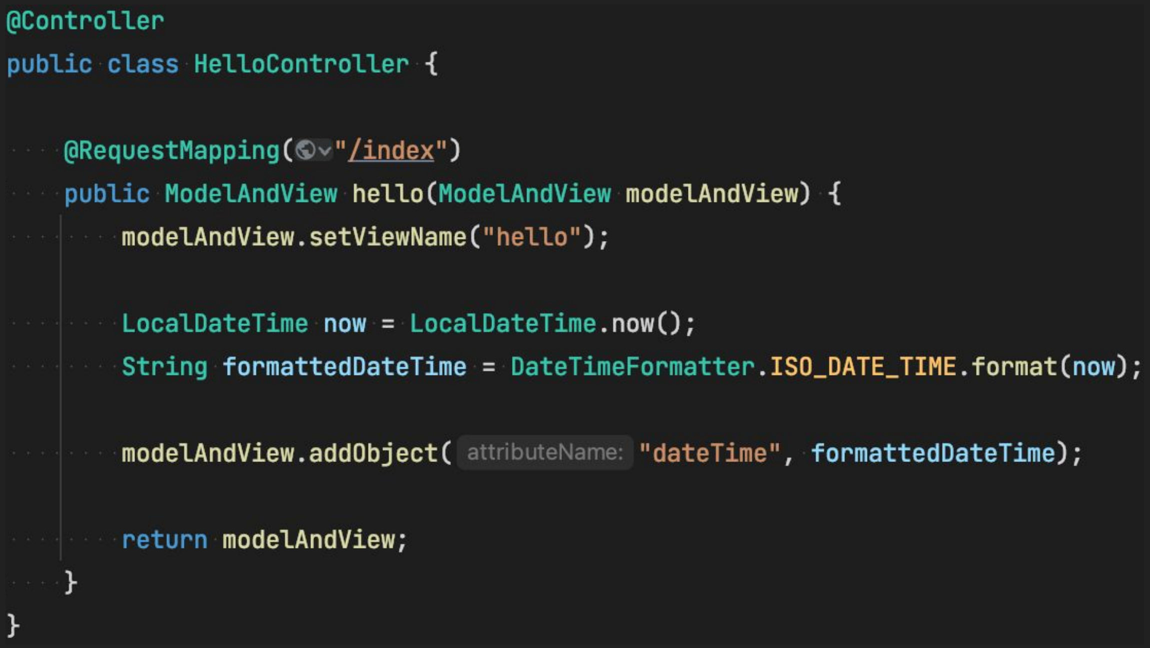
Example - RequestDispatcher example

https://github.com/vrudas/spring-framework-examples/tree/main/example-09-dispatcher-servlet

Spring Web MVC - Controller

- DispatcherServlet delegates the request to the controllers to execute the functionality specific to it.
- The @Controller annotation indicates that a particular class serves the role of a controller.
- The @RequestMapping annotation is used to map a URL to either an entire class or a particular handler method. The class-level usage of @RequestMapping indicates that all handling methods on this controller are relative to his path.

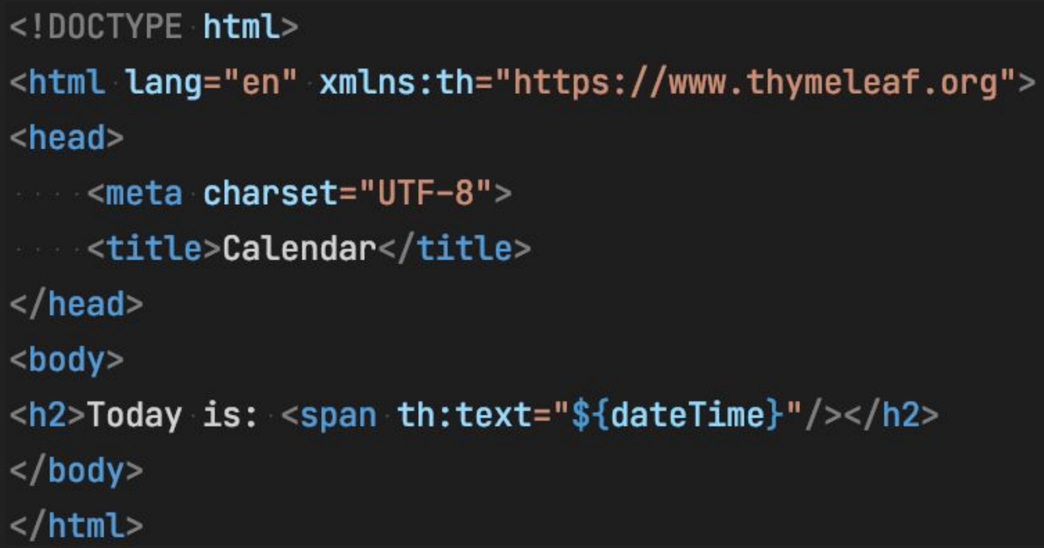
Spring Web MVC - Controller and Model



```java
@Controller
public class HelloController {

    @RequestMapping("/index")
    public ModelAndView hello(ModelAndView modelAndView) {
        modelAndView.setViewName("hello");

        LocalDateTime now = LocalDateTime.now();
        String formattedDateTime = DateTimeFormatter.ISO_DATE_TIME.format(now);

        modelAndView.addObject( attributeName: "dateTime", formattedDateTime);

        return modelAndView;
    }
}
```

Spring Web MVC - View

Spring MVC supports many types of views for different presentation technologies. These include - HTML, XML, JSON etc. Here example of HTML template written with Thymeleaf:



```html
<!DOCTYPE html>
<html lang="en" xmlns:th="https://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Calendar</title>
</head>
<body>
<h2>Today is: <span th:text="${dateTime}"/></h2>
</body>
</html>
```

Thymeleaf

Thymeleaf

- Thymeleaf is a Java library
- It is an XML/HTML template engine able to apply a set of transformations to template files in order to display data and/or text produced by your applications
- Often used as a Back-end rendering technology for web applications
- Has own dialect language to manipulate with data

More info at:

https://www.thymeleaf.org/doc/tutorials/3.0/usingthymeleaf.html

https://www.baeldung.com/thymeleaf-in-spring-mvc

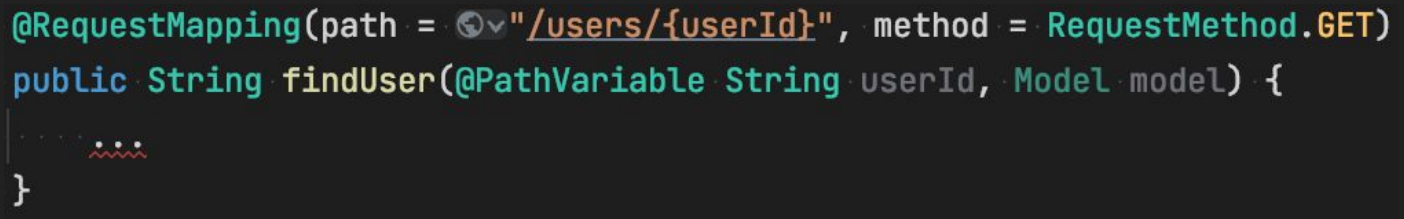
@RequestMapping - Details

- URI templates can be used for convenient access to selected parts of a URL in a @RequestMapping method.

- A URI Template is a URI-like string, containing one or more variable names. When you substitute values for these variables, the template becomes a URI.

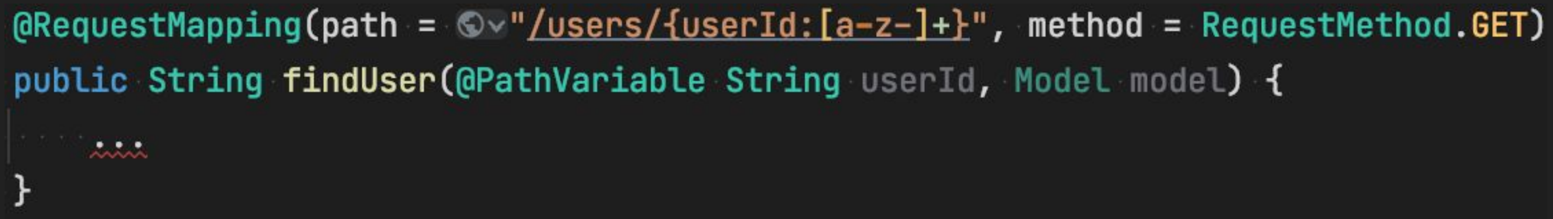
- For example, the URI Template http://localhost:8080/users/{userId} contains the variable userId. Assigning the value 123 to the variable placeholder provide http://localhost:8080/users/123.

@RequestMapping - Details



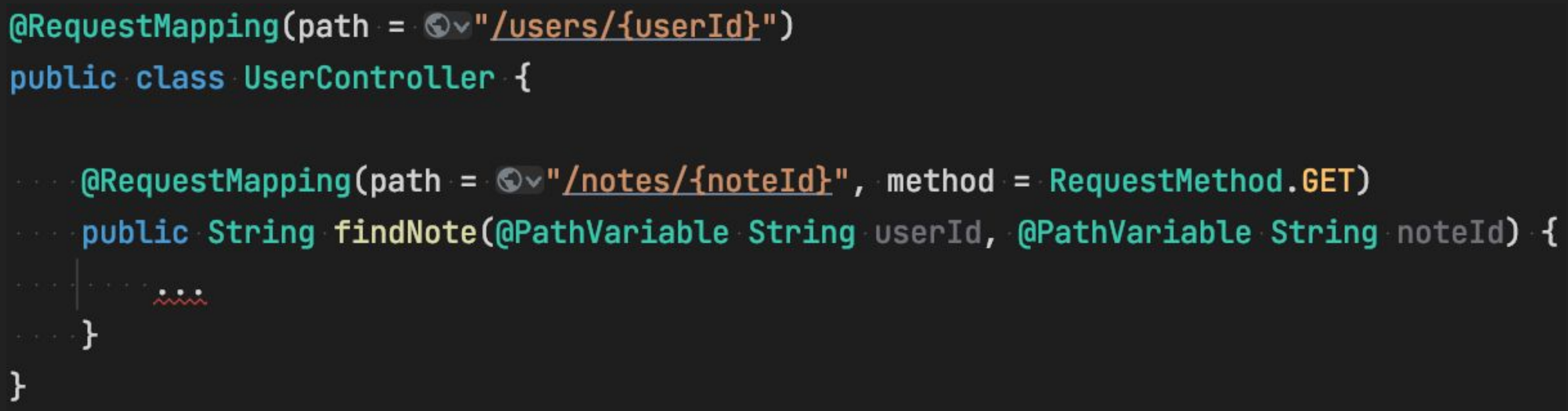
```java
@RequestMapping(path = "/users/{userId}", method = RequestMethod.GET)
public String findUser(@PathVariable String userId, Model model) {
    ...
}
```

@RequestMapping - Details



```
@RequestMapping(path = "/users/{userId:[a-z-]+}", method = RequestMethod.GET)
public String findUser(@PathVariable String userId, Model model) {
    ...
}
```

@RequestMapping - Details



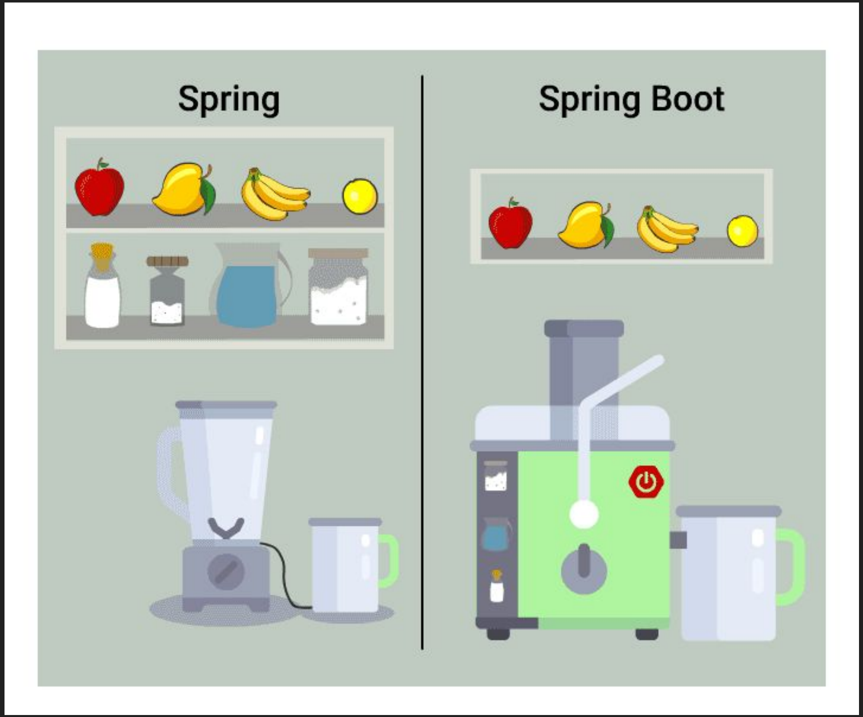
```java
@RequestMapping(path = "/users/{userId}")
public class UserController {

    @RequestMapping(path = "/notes/{noteId}", method = RequestMethod.GET)
    public String findNote(@PathVariable String userId, @PathVariable String noteId) {
            ...
    }
}
```

Example - Spring Web MVC

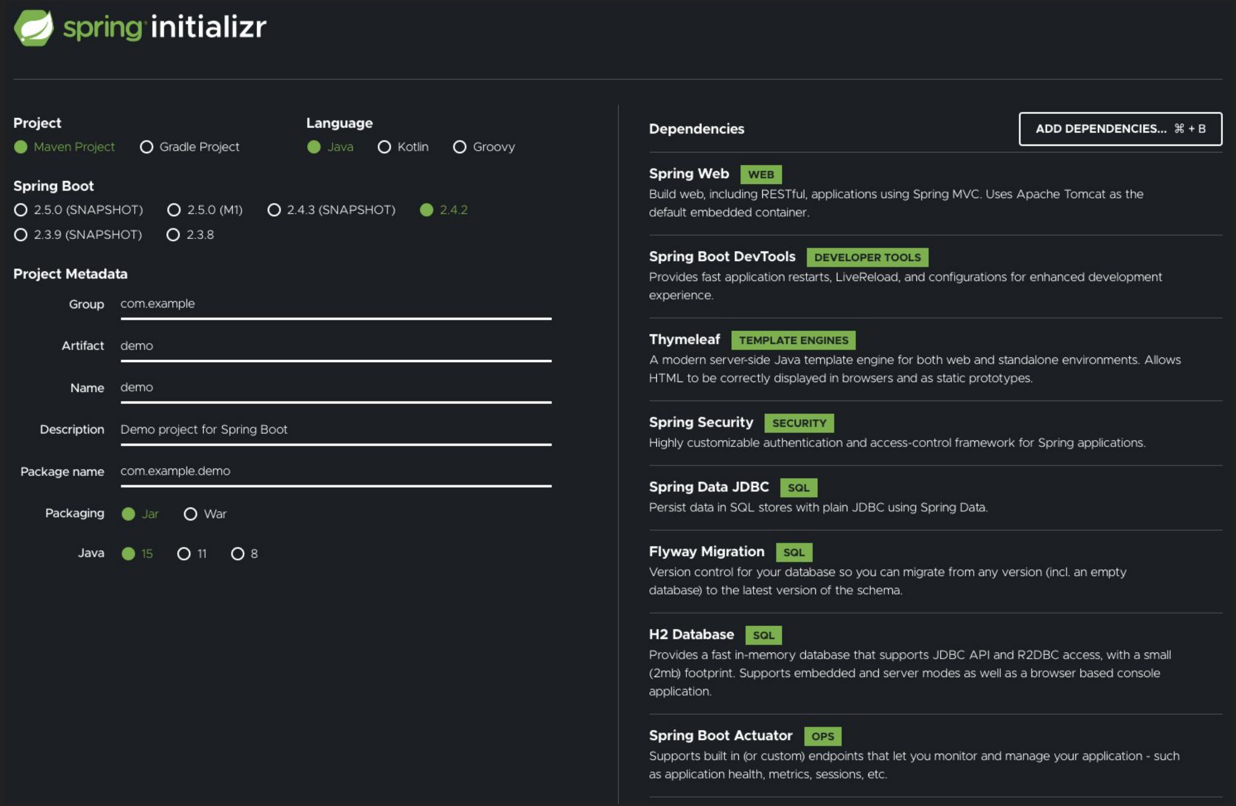
https://github.com/vrudas/spring-framework-examples/tree/main/example-10-spring-mvc

Spring Boot

Spring Boot



Spring Boot Starters (https://start.spring.io/)



spring initializr

Project

[✓ Maven Project] Gradle Project

Language

[✓ Java] Kotlin Groovy

Spring Boot

2.5.0 (SNAPSHOT) 2.5.0 (M1) 2.4.3 (SNAPSHOT) [✓ 2.4.2]
2.3.9 (SNAPSHOT) 2.3.8

Project Metadata

Group	com.example
Artifact	demo
Name	demo
Description	Demo project for Spring Boot
Package name	com.example.demo
Packaging	[✓ Jar] War
Java	[✓ 15] 11 8

Dependencies ADD DEPENDENCIES... ⌘ + B

Spring Web WEB
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Spring Boot DevTools DEVELOPER TOOLS
Provides fast application restarts, LiveReload, and configurations for enhanced development experience.

Thymeleaf TEMPLATE ENGINES
A modern server-side Java template engine for both web and standalone environments. Allows HTML to be correctly displayed in browsers and as static prototypes.

Spring Security SECURITY
Highly customizable authentication and access-control framework for Spring applications.

Spring Data JDBC SQL
Persist data in SQL stores with plain JDBC using Spring Data.

Flyway Migration SQL
Version control for your database so you can migrate from any version (incl. an empty database) to the latest version of the schema.

H2 Database SQL
Provides a fast in-memory database that supports JDBC API and R2DBC access, with a small (2mb) footprint. Supports embedded and server modes as well as a browser based console application.

Spring Boot Actuator OPS
Supports built in (or custom) endpoints that let you monitor and manage your application - such as application health, metrics, sessions, etc.

Example - Spring Boot

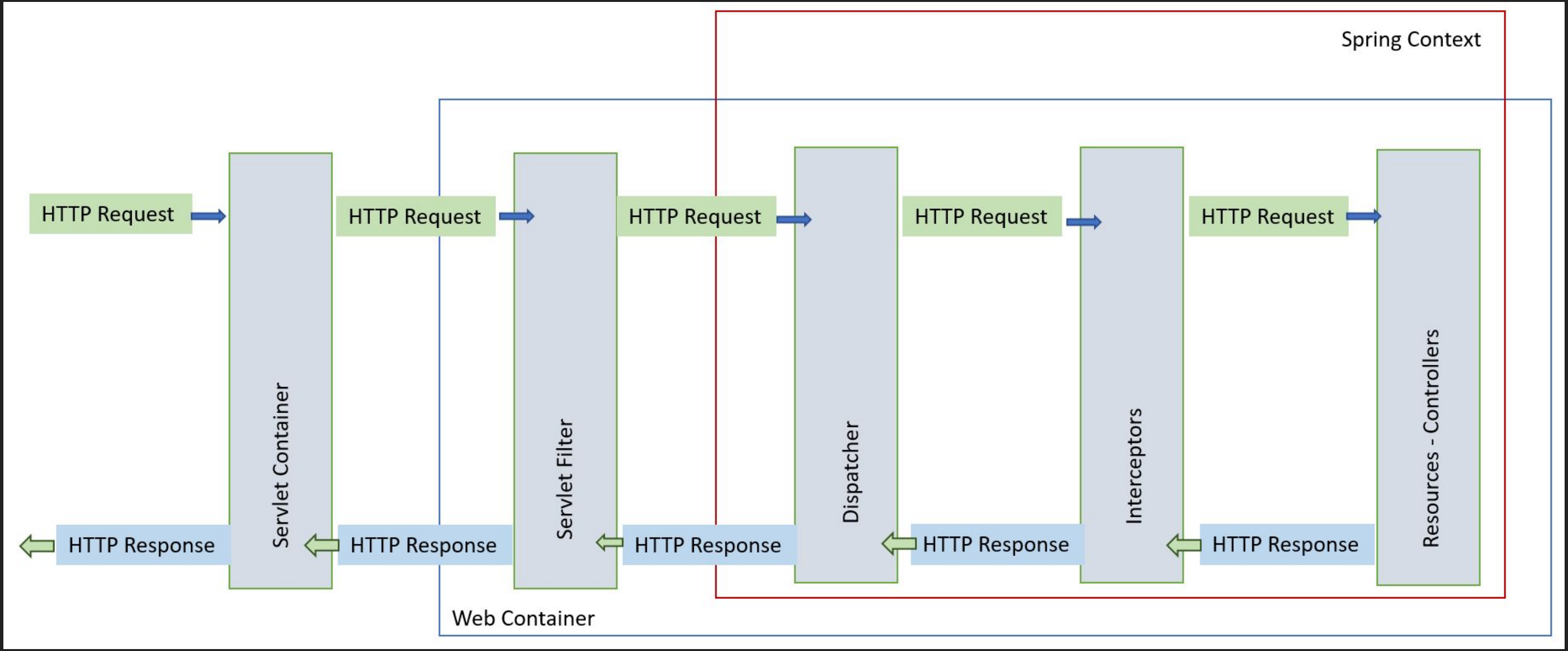
https://github.com/vrudas/spring-framework-examples/tree/main/example-11-spring-boot

Notes Application

Example - Notes Application

https://github.com/vrudas/spring-framework-examples/tree/main/notes-app

Request Interceptor



Example - Interceptor

https://github.com/vrudas/spring-framework-examples/blob/main/notes-app/src/main/java/io/sfe/notesapp/web/common/LoggerInterceptor.java

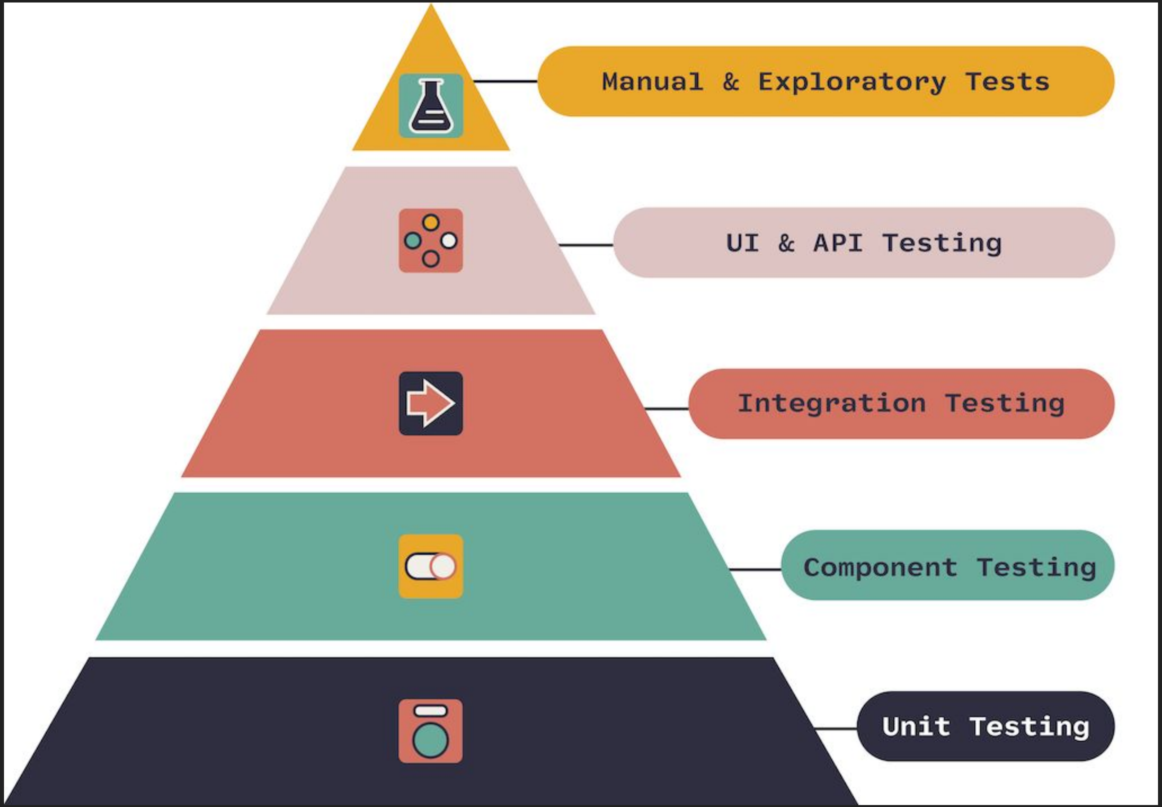
Error Handling

Example - Error Handling

https://github.com/vrudas/spring-framework-examples/blob/main/notes-app/src/main/java/io/sfe/notesapp/web/common/ControllerExceptionHandler.java

Integration Testing

Testing Pyramid



Integration Testing Concept

@TestPropertySources

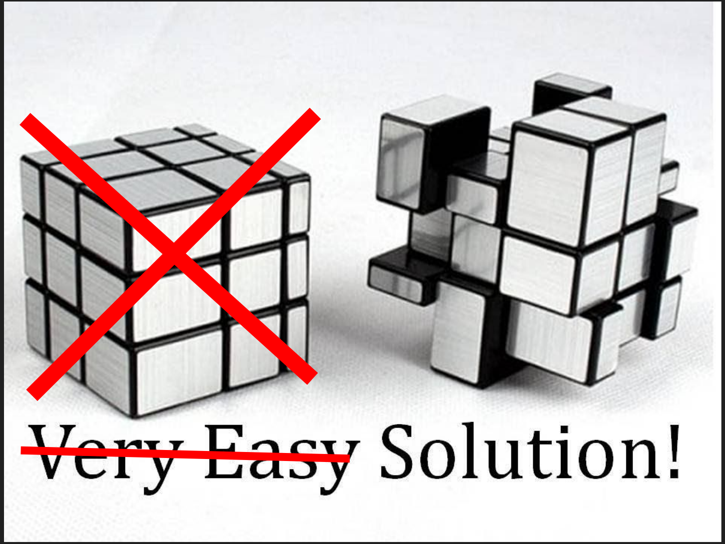
@TestPropertySource - is a class-level annotation that is used to configure the locations of properties files and inlined properties to be added to the Environment's set of PropertySources for an ApplicationContext for integration tests.

Example - @TestPropertySource

https://github.com/vrudas/spring-framework-examples/blob/main/notes-app/src/test/java/io/sfe/notesapp/TestPropertySourcesTest.java

@ContextConfiguration

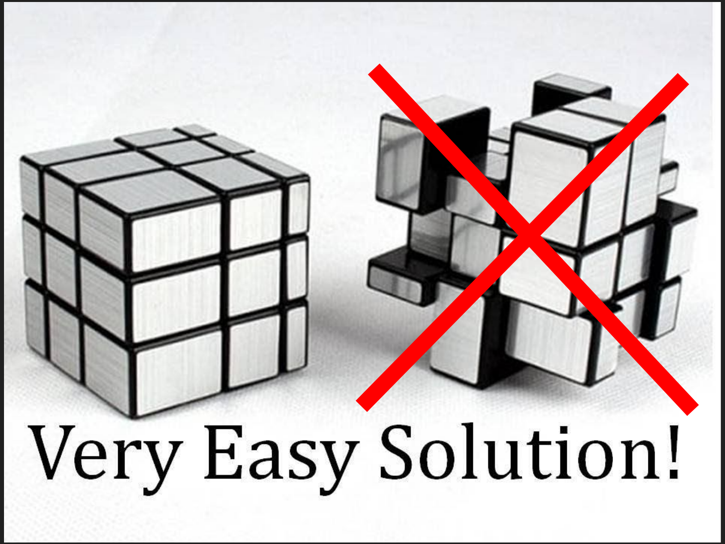
@ContextConfiguration - defines class-level metadata that is used to determine how to load and configure an ApplicationContext for integration tests.



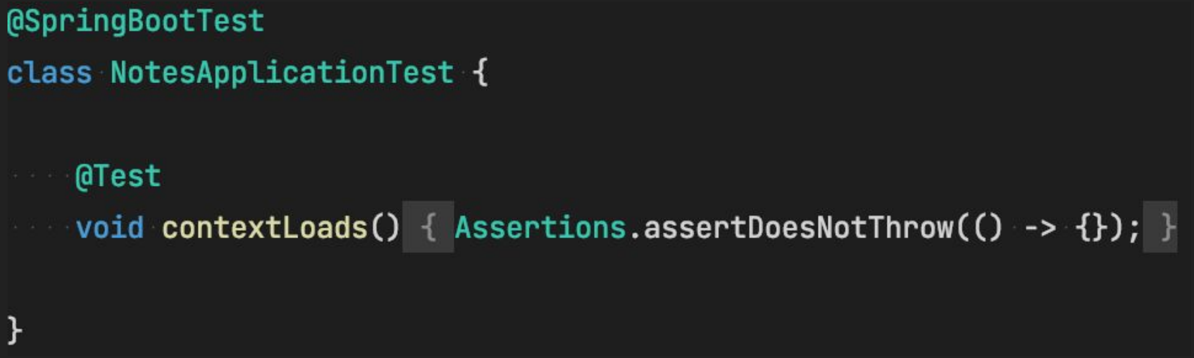
Example - @ContextConfiguration

https://github.com/vrudas/spring-framework-examples/blob/main/notes-app/src/test/java/io/sfe/notesapp/domain/note/NoteServiceContextConfigurationTest.java

@SpringBoot for help

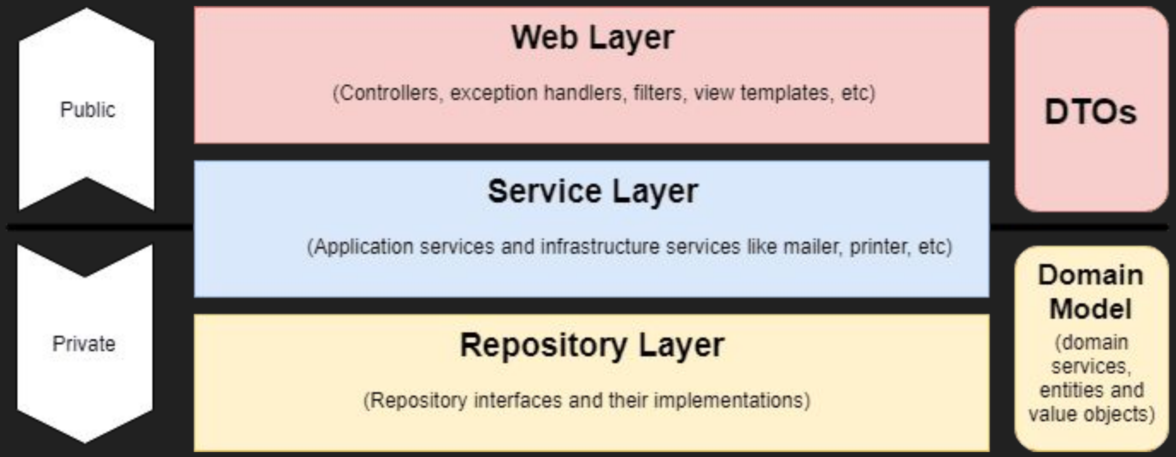


Example - Spring Boot Integration test



```java
@SpringBootTest
class NotesApplicationTest {

    @Test
    void contextLoads() { Assertions.assertDoesNotThrow(() -> {}); }

}
```

Layered Testing Concept


Public
Private
Web Layer
(Controllers, exception handlers, filters, view templates, etc)
Service Layer
(Application services and infrastructure services like mailer, printer, etc)
Repository Layer
(Repository interfaces and their implementations)
DTOs
Domain Model
(domain services, entities and value objects)

Spring Boot Layered Testing

- @JdbcTest - focuses only on JDBC-based components (tests are transactional and roll back at the end of each test. They also use an embedded in-memory database).
- @DataJdbcTest - focuses only on Data JDBC components (tests are transactional and roll back at the end of each test. They also use an embedded in-memory database).
- @WebMvcTest - focuses only on Spring MVC components (auto-configure Spring Security and MockMvc).

- @SpringBootTest - is useful when we need to bootstrap the entire container (aka Integration Test).

Example - @JdbcTest (NoteTableSchemaTest)

https://github.com/vrudas/spring-framework-examples/blob/main/notes-app/src/test/java/io/sfe/notesapp/storage/note/NoteTableSchemaTest.java

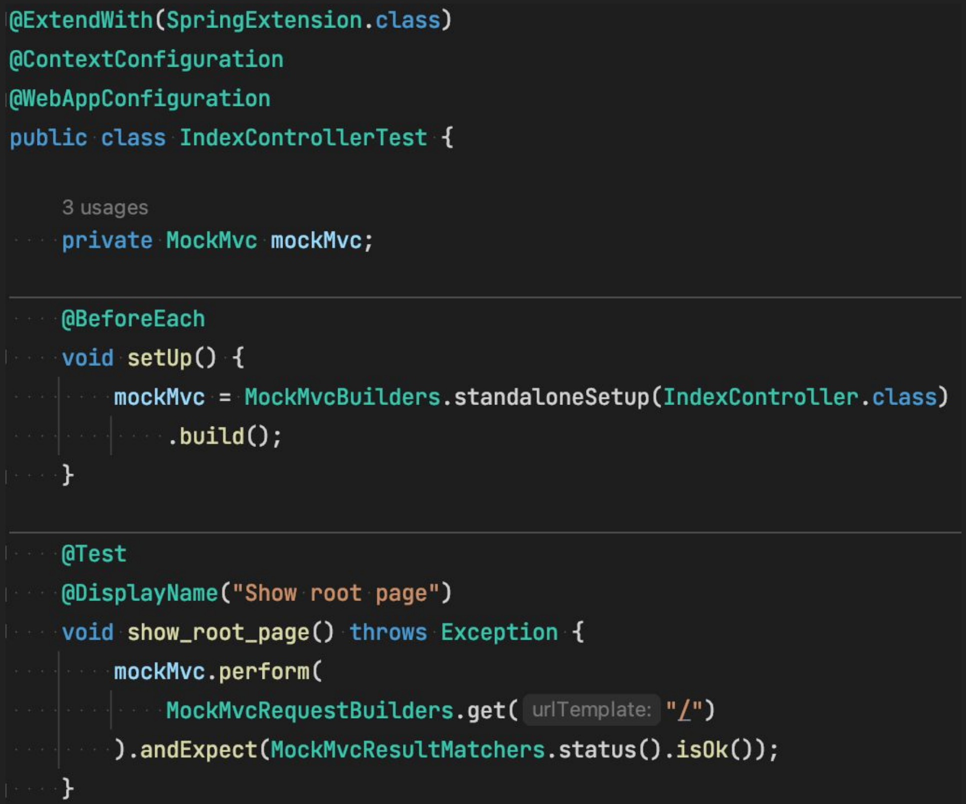
Example - @JdbcTest (NoteJdbcTemplateRepositoryTest)

https://github.com/vrudas/spring-framework-examples/blob/main/notes-app/src/test/java/io/sfe/notesapp/storage/note/NoteJdbcTemplateRepositoryTest.java

Example - @DataJdbcTest (NoteRepositoryTest)

https://github.com/vrudas/spring-framework-examples/blob/main/notes-app/src/test/java/io/sfe/notesapp/storage/note/NoteRepositoryTest.java

Example - Spring Web MVC Manual Config



```
@ExtendWith(SpringExtension.class)
@ContextConfiguration
@WebAppConfiguration
public class IndexControllerTest {

    3 usages
    private MockMvc mockMvc;

    @BeforeEach
    void setUp() {
        mockMvc = MockMvcBuilders.standaloneSetup(IndexController.class)
            .build();
    }

    @Test
    @DisplayName("Show root page")
    void show_root_page() throws Exception {
        mockMvc.perform(
            MockMvcRequestBuilders.get( urlTemplate: "/")
        ).andExpect(MockMvcResultMatchers.status().isOk());
    }
```

Example - MockMvcResultMatchers

https://github.com/vrudas/spring-framework-examples/blob/main/notes-app/src/test/java/io/sfe/notesapp/web/IndexControllerTest.java

Example - @WebMvcTest

https://github.com/vrudas/spring-framework-examples/blob/main/notes-app/src/test/java/io/sfe/notesapp/web/note/NoteControllerTest.java

Example - Business Domain Integration Test

https://github.com/vrudas/spring-framework-examples/blob/main/notes-app/src/test/java/io/sfe/notesapp/domain/note/NoteServiceIntegrationTest.java

Example - Full Control Flow Integration Test

https://github.com/vrudas/spring-framework-examples/blob/main/notes-app/src/test/java/io/sfe/notesapp/web/note/NoteControllerIntegrationTest.java

More Spring Boot Layered Testing

- @DataMongoTest - can be used for a MongoDB test that focuses only on MongoDB components.
- @JooqTest - focuses only on jOOQ-based components.
- @JsonTest - focuses only on JSON serialization.
- @DataJpaTest - focuses only on JPA components.
- @RestClientTest - only on beans that use RestTemplateBuilder (apply only configuration relevant to rest client tests (i.e. Jackson or GSON auto-configuration and @JsonComponent beans, but not regular @Component beans)).
- @WebFluxTest - focuses only on Spring WebFlux components.

Spring Security

Contents

About Spring Security

Spring Security is a framework that provides:

- authentication
- authorization
- protection against common attacks

With first class support for both imperative and reactive applications, it is the de-facto standard for securing Spring-based applications.

Example - Small app does a lot

https://github.com/vrudas/spring-framework-examples/tree/main/example-12-boot-security

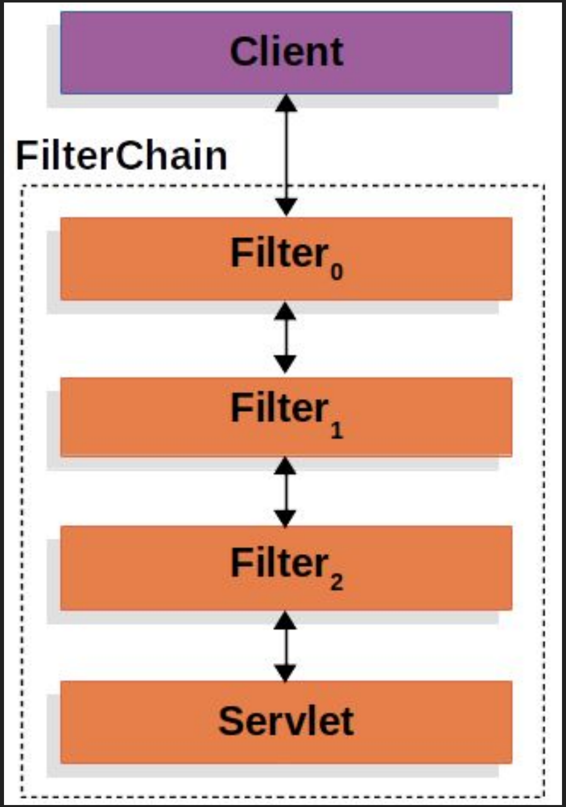
Spring Boot Security Auto Configuration

- Enable security using Filter bean springSecurityFilterChain (responsible for all protection in app)
- Require an authenticated user for any interaction with the application
- Generate a default login form for you
- Creates a UserDetailsService bean with a username of user and a randomly generated password
- Protects the password storage with BCrypt
- Lets the user log out

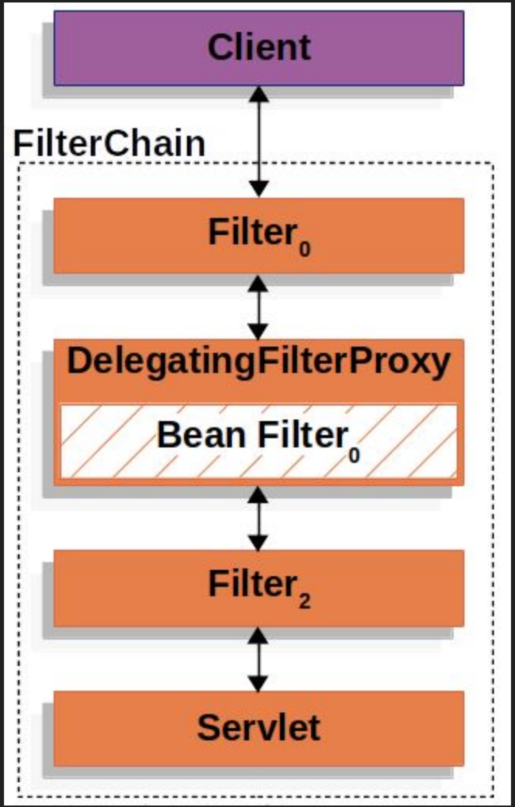
Security Auto Configuration - Protection

- CSRF attack prevention
- Session Fixation protection
- Security Header integration
    - HTTP Strict Transport Security for secure requests
    - X-Content-Type-Options integration
    - Cache Control (can be overridden later by your application to allow caching of your static resources)
    - X-XSS-Protection integration
    - X-Frame-Options integration to help prevent Clickjacking

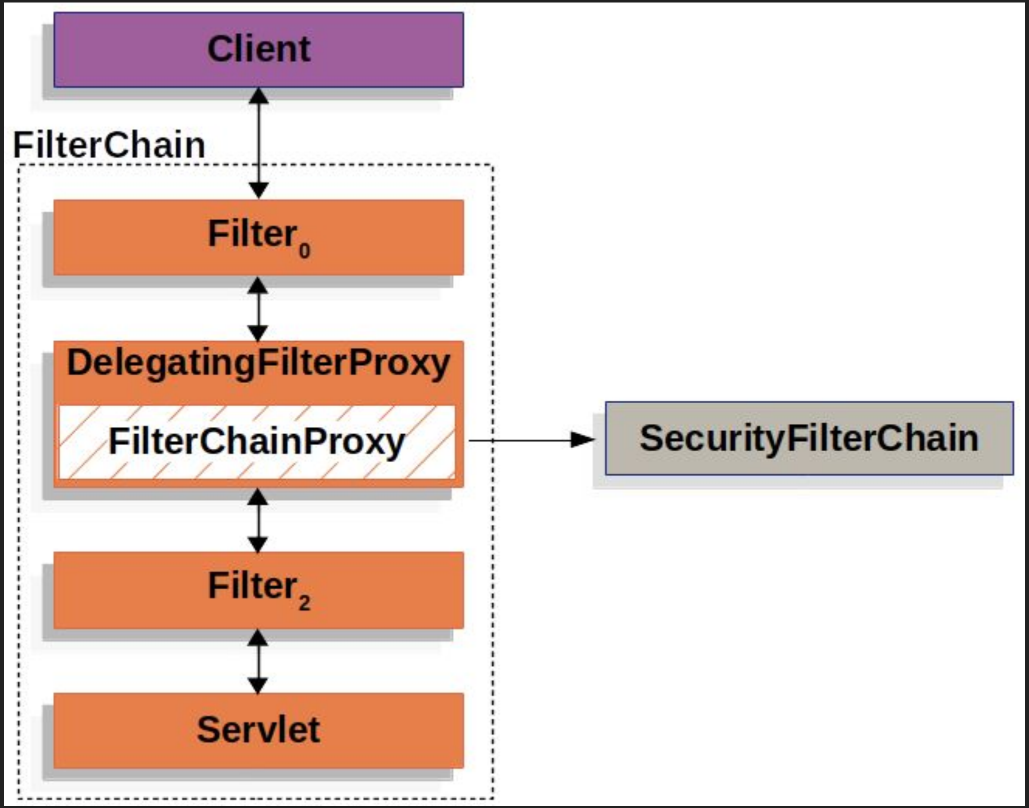
Servlet Security: The Big Picture - A Review of Filters



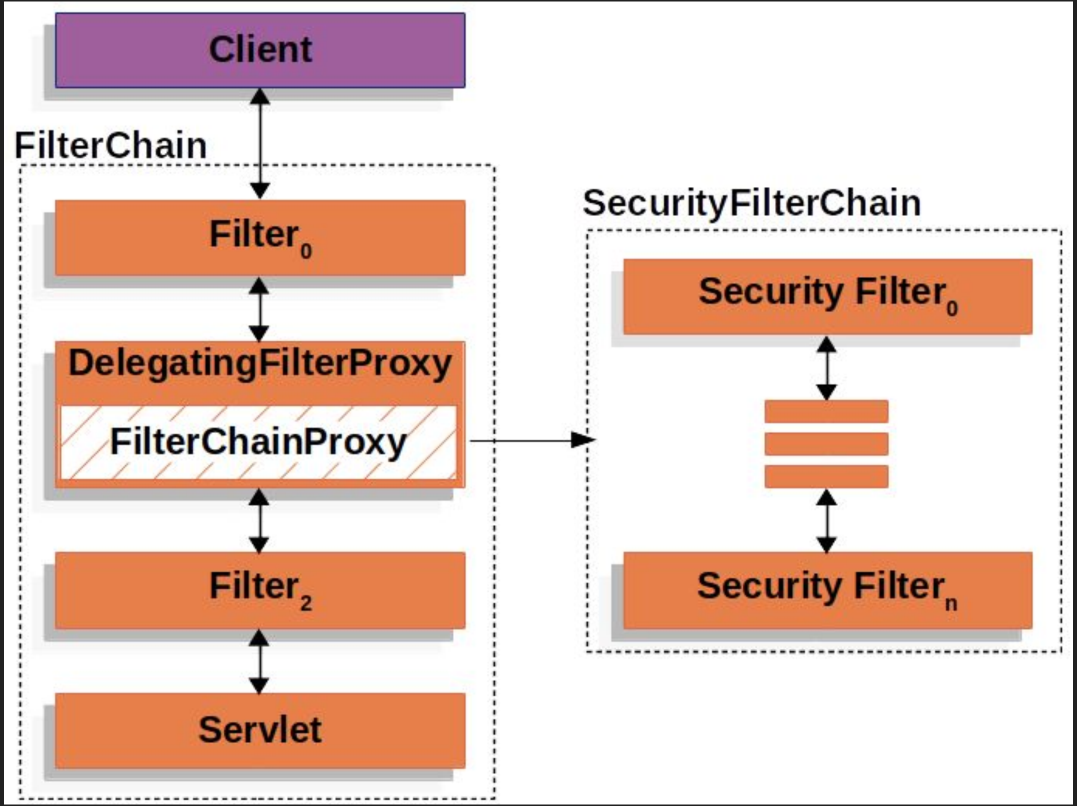
DelegatingFilterProxy



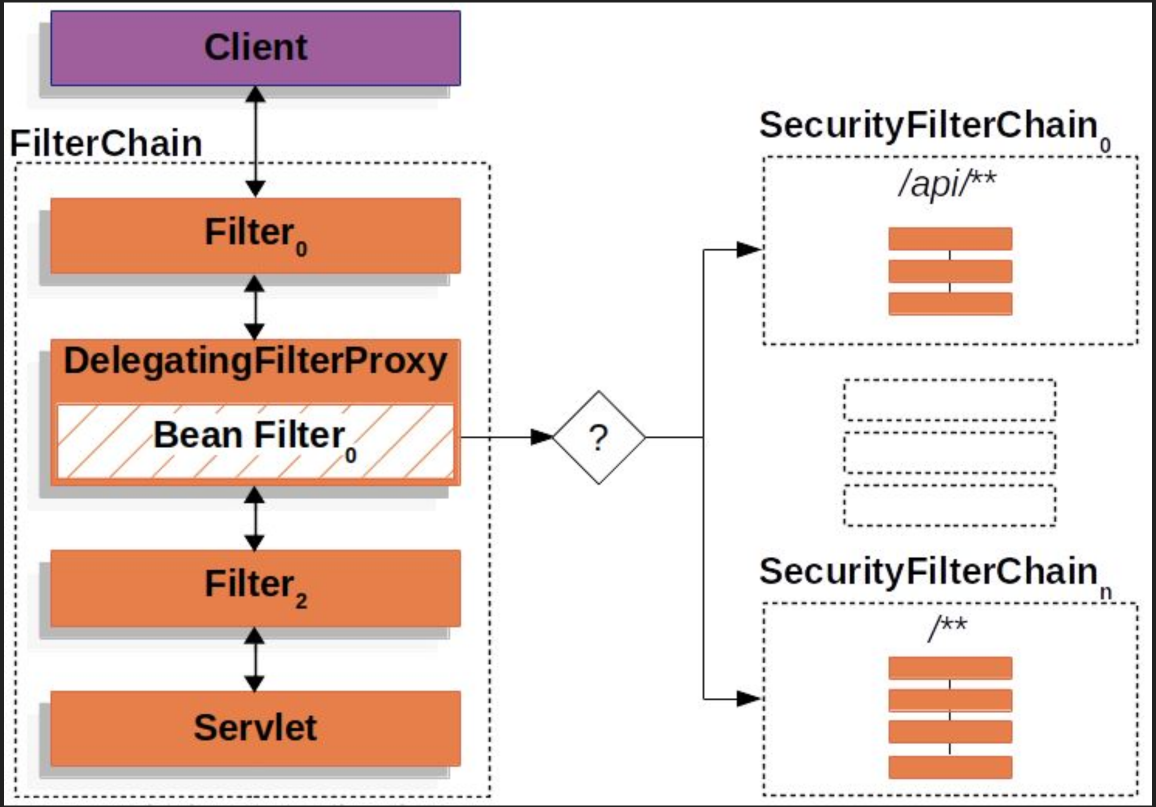
FilterChainProxy



SecurityFilterChain



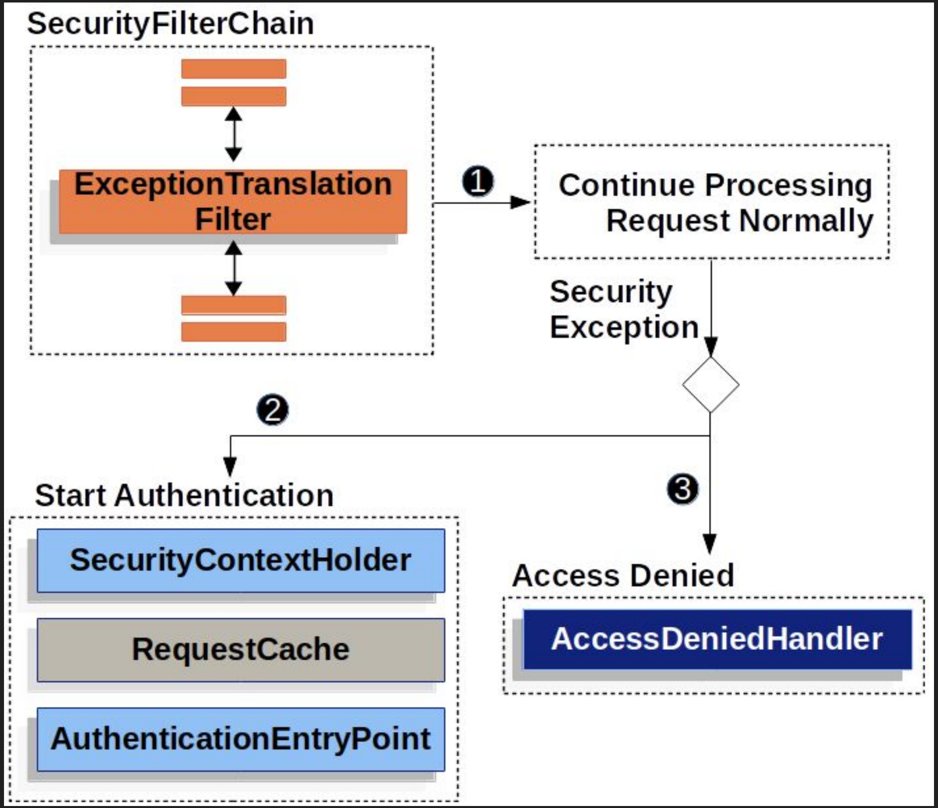
Multiple SecurityFilterChain



Security Filters (more at Filters list)

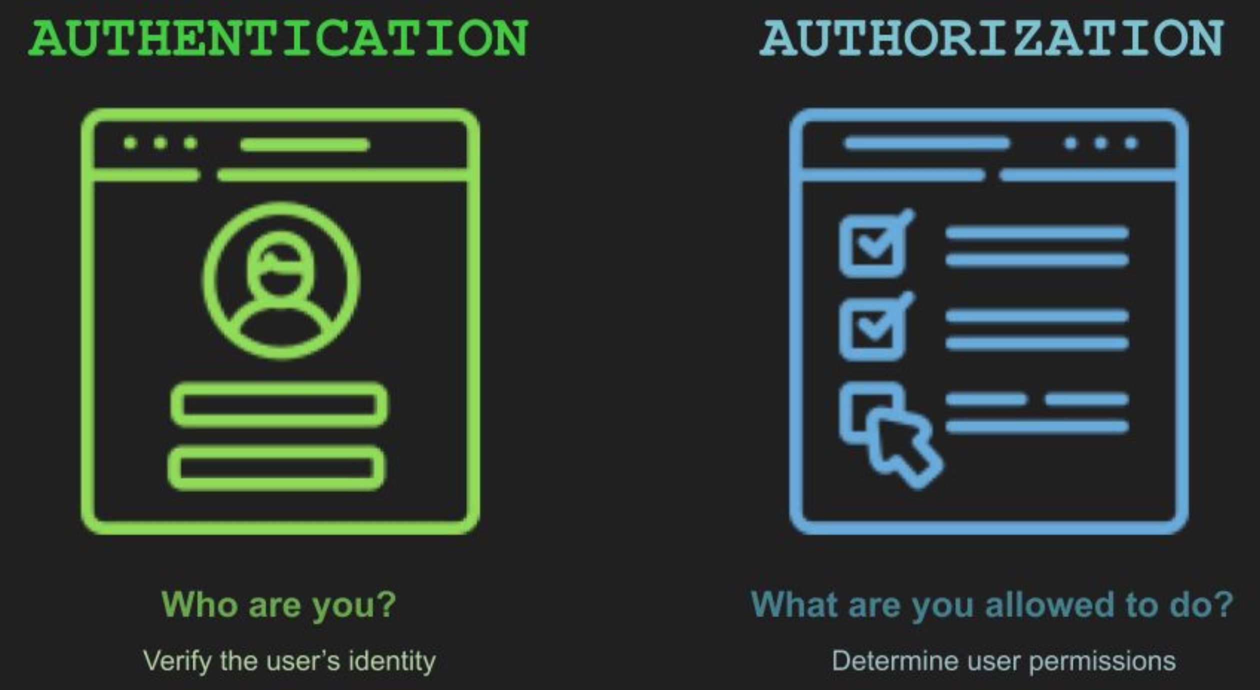
- HeaderWriterFilter
- CorsFilter
- CsrfFilter
- LogoutFilter
- OAuth2LoginAuthenticationFilter
- UsernamePasswordAuthenticationFilter
- DefaultLoginPageGeneratingFilter
- DefaultLogoutPageGeneratingFilter
- BearerTokenAuthenticationFilter
- BasicAuthenticationFilter

Handling Security Exceptions



Authentication vs Authorization - The problem?

Authentication vs Authorization - The problem?


AUTHENTICATION
Who are you?
Verify the user’s identity
AUTHORIZATION
What are you allowed to do?
Determine user permissions

Authentication - Architecture Components

- SecurityContextHolder - The SecurityContextHolder is where Spring Security stores the details of who is authenticated.
- SecurityContext - is obtained from the SecurityContextHolder and contains the Authentication of the currently authenticated user.
- Authentication - Can be the input to AuthenticationManager to provide the credentials a user has provided to authenticate or the current user from the SecurityContext.
- GrantedAuthority - An authority that is granted to the principal on the Authentication (i.e. roles, scopes, etc.)

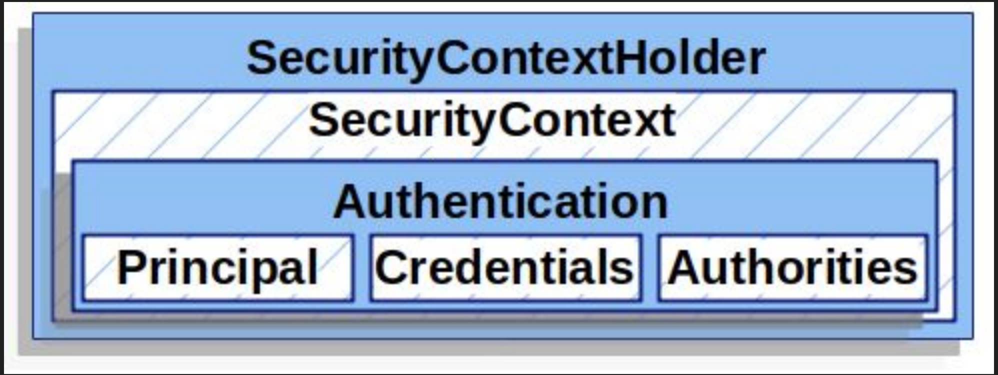
Authentication - Architecture Components

- AuthenticationManager - the API that defines how Spring Security's Filters perform authentication.
- ProviderManager - the most common implementation of AuthenticationManager.
- AuthenticationProvider - used by ProviderManager to perform a specific type of authentication.
- AuthenticationEntryPoint - used for requesting credentials from a client (i.e. redirecting to a login page, sending a WWW-Authenticate response, etc.)
- AbstractAuthenticationProcessingFilter - a base Filter used for authentication. This also gives a good idea of the high level flow of authentication and how pieces work together.

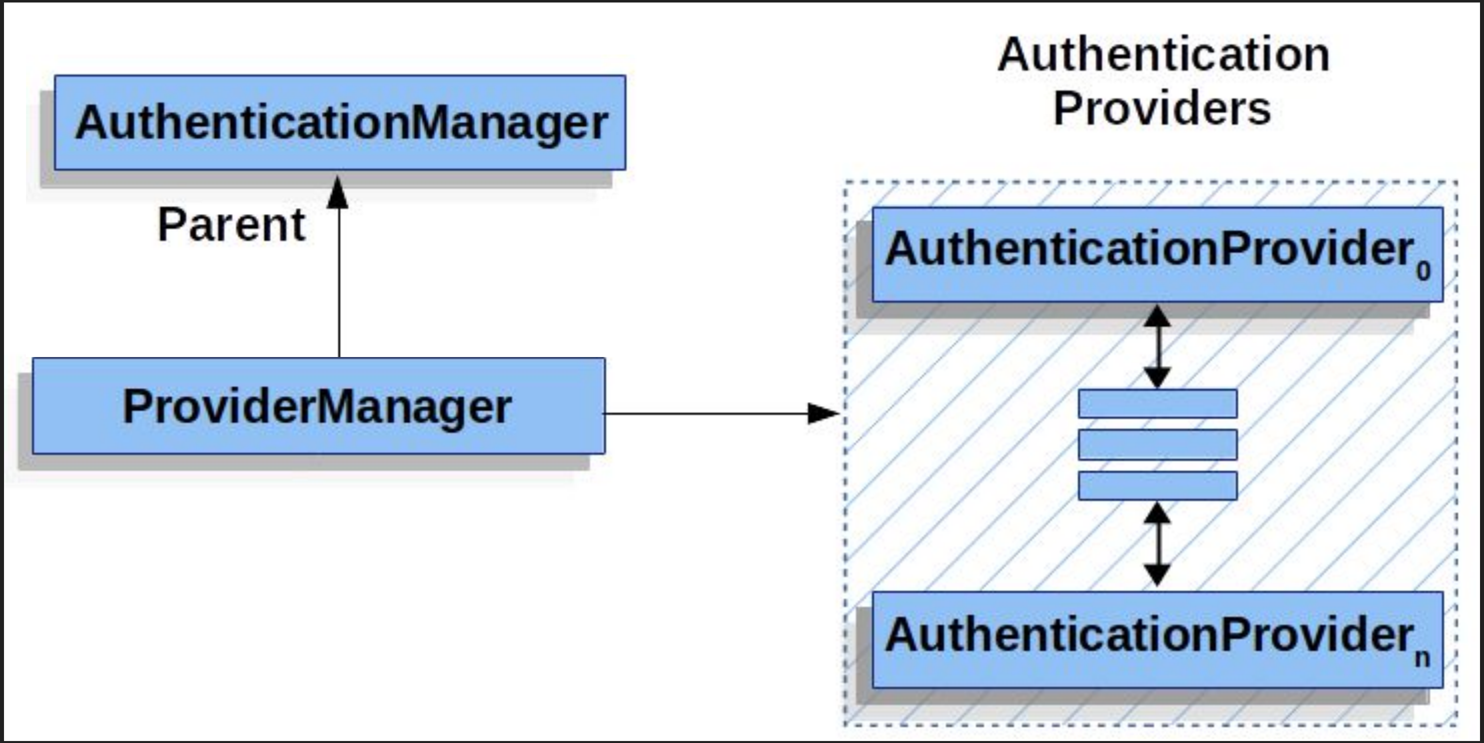
Authentication Mechanisms

- Username and Password - how to authenticate with a username/password
- OAuth 2.0 Login - OAuth 2.0 Log In with OpenID Connect and non-standard OAuth 2.0 Login (i.e. GitHub)
- SAML 2.0 Login - SAML 2.0 Log In
- Remember Me - How to remember a user past session expiration
- OpenID - OpenID Authentication (not to be confused with OpenID Connect)
- …

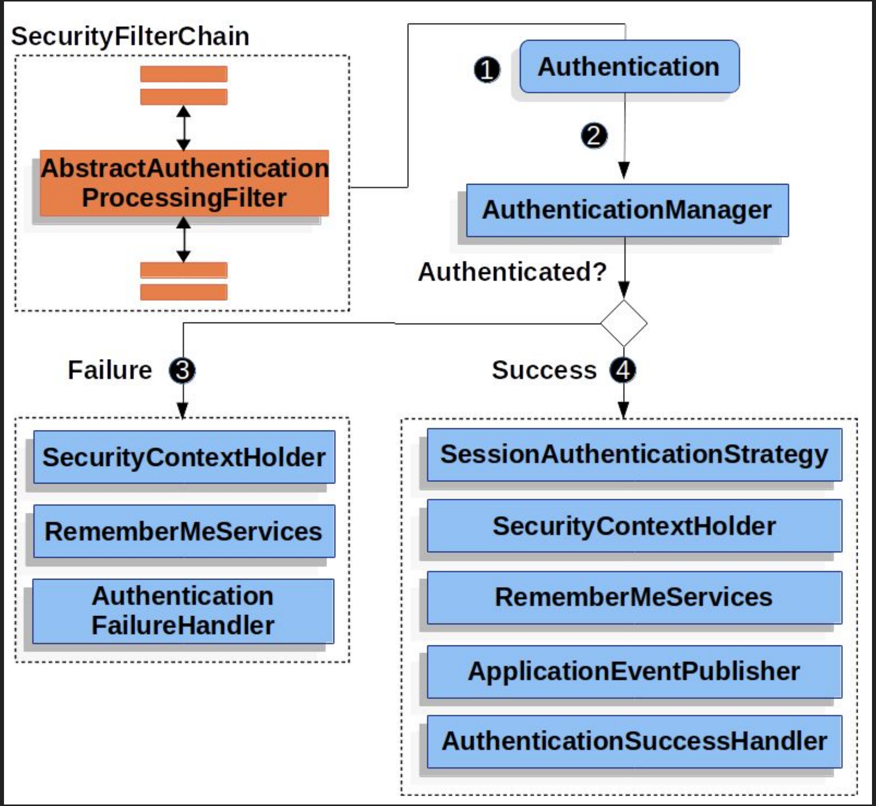
Authentication holding via SecurityContextHolder



AuthenticationManager instance via ProviderManager



AbstractAuthenticationProcessingFilter

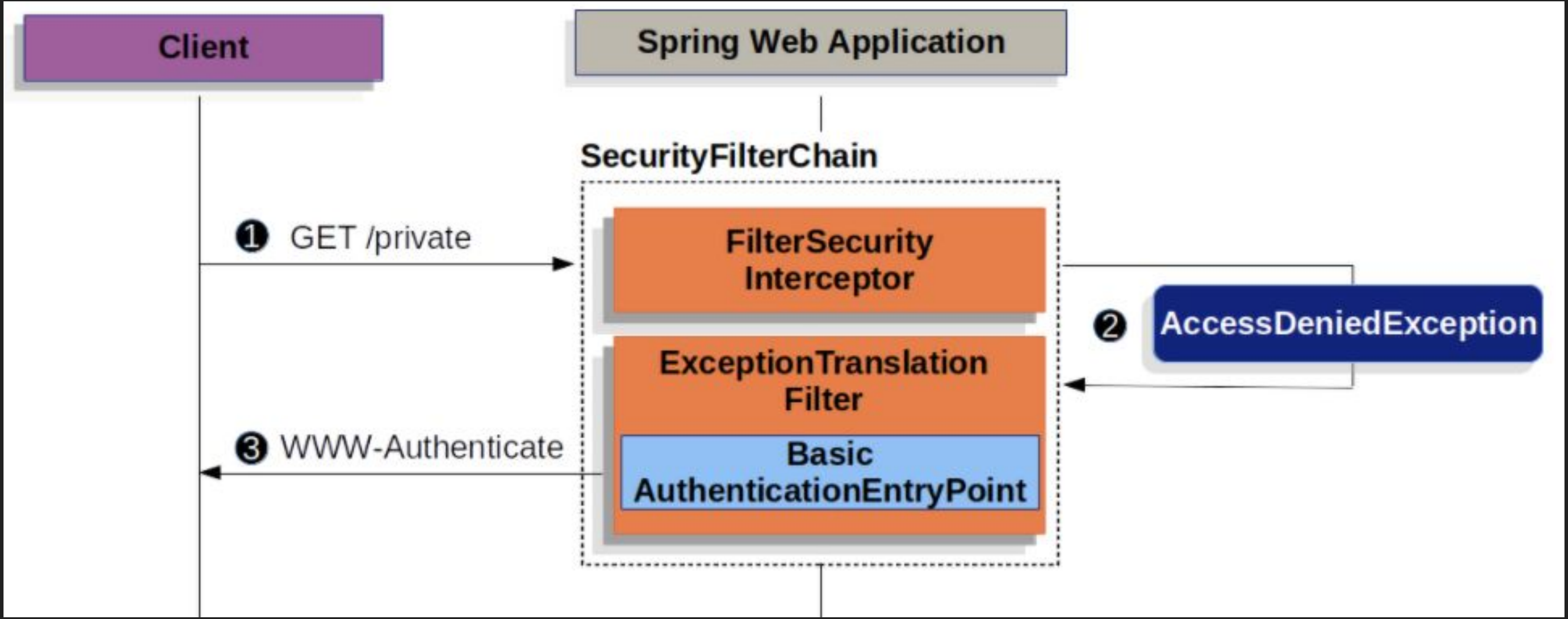


Username/Password Authentication

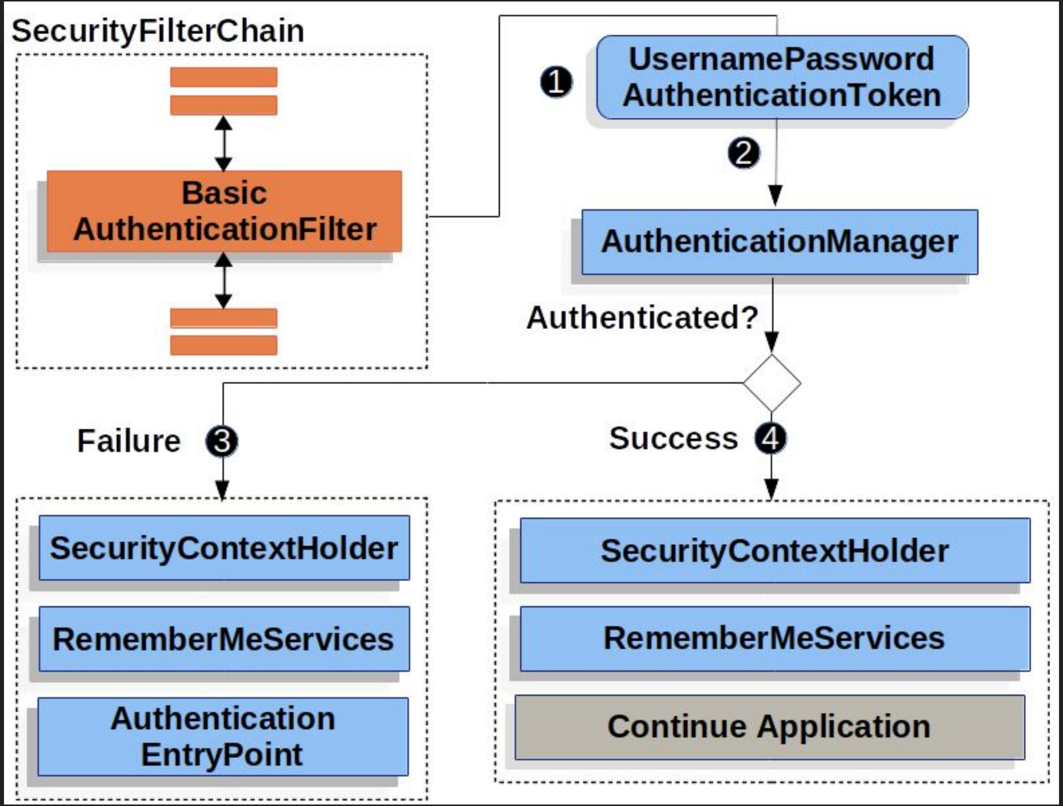
- Reading the Username & Password
  - Basic Authentication (Basic)
  - Form Login

- Storage Mechanisms
  - Simple Storage with In-Memory Authentication
  - Relational Databases with JDBC Authentication
  - Custom data stores with UserDetailsService

Basic Authentication - WWW-Authenticate header



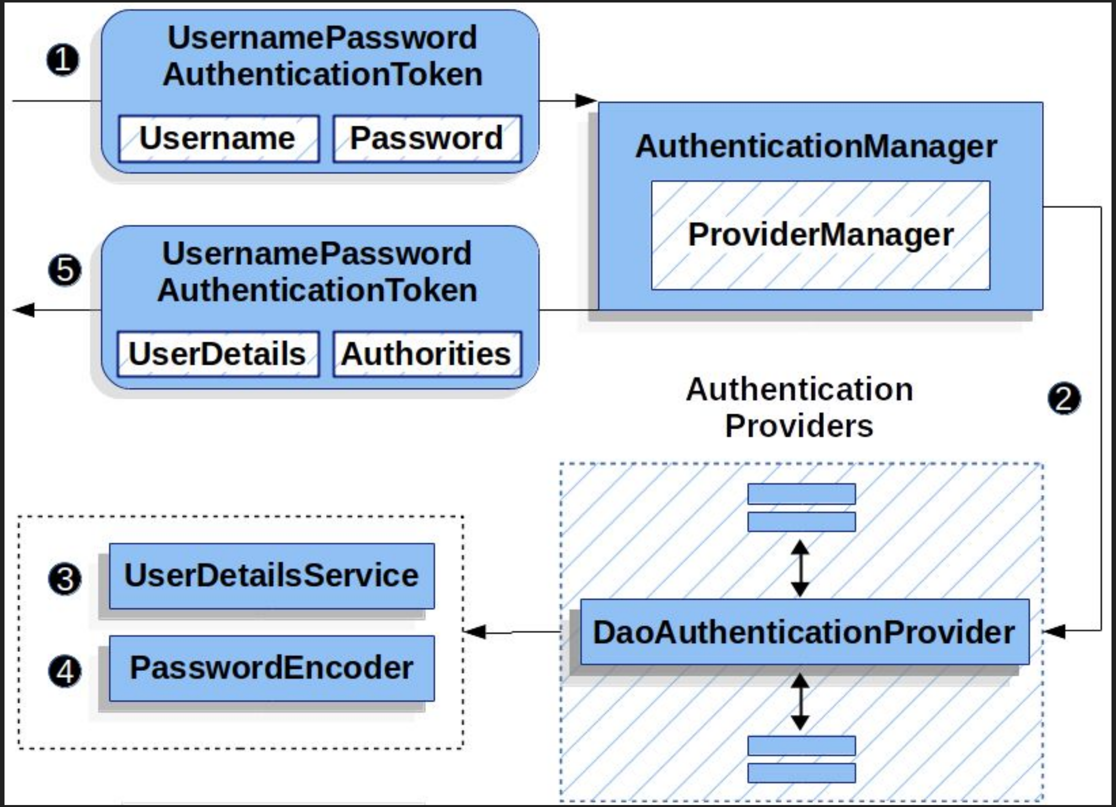
Basic Authentication - Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==



Example - Basic Authentication with In-Memory storage

https://github.com/vrudas/spring-framework-examples/tree/main/example-13-inmemory-security

DaoAuthenticationProvider (aka JDBC storage)



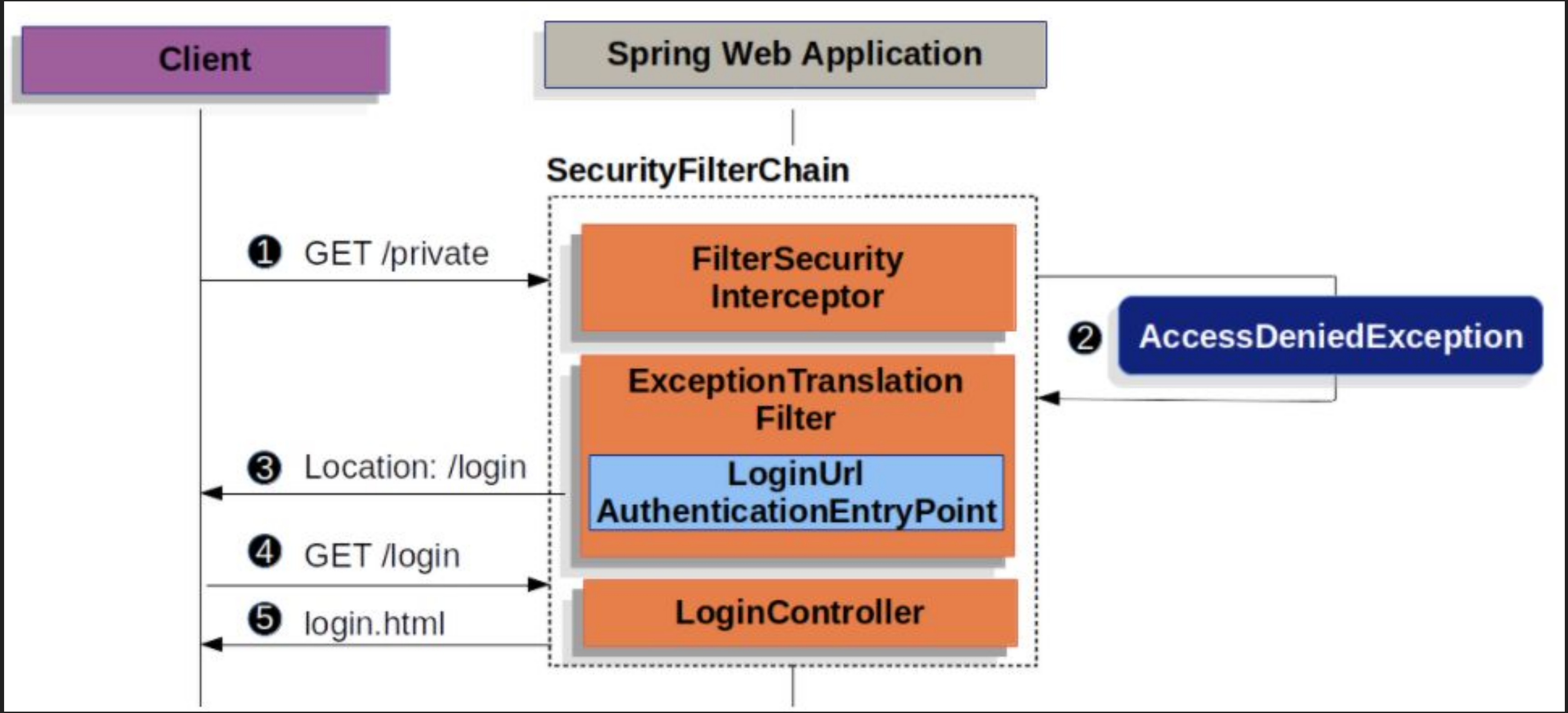
Example - DB Storage

https://github.com/vrudas/spring-framework-examples/tree/main/example-14-jdbc-auth-security

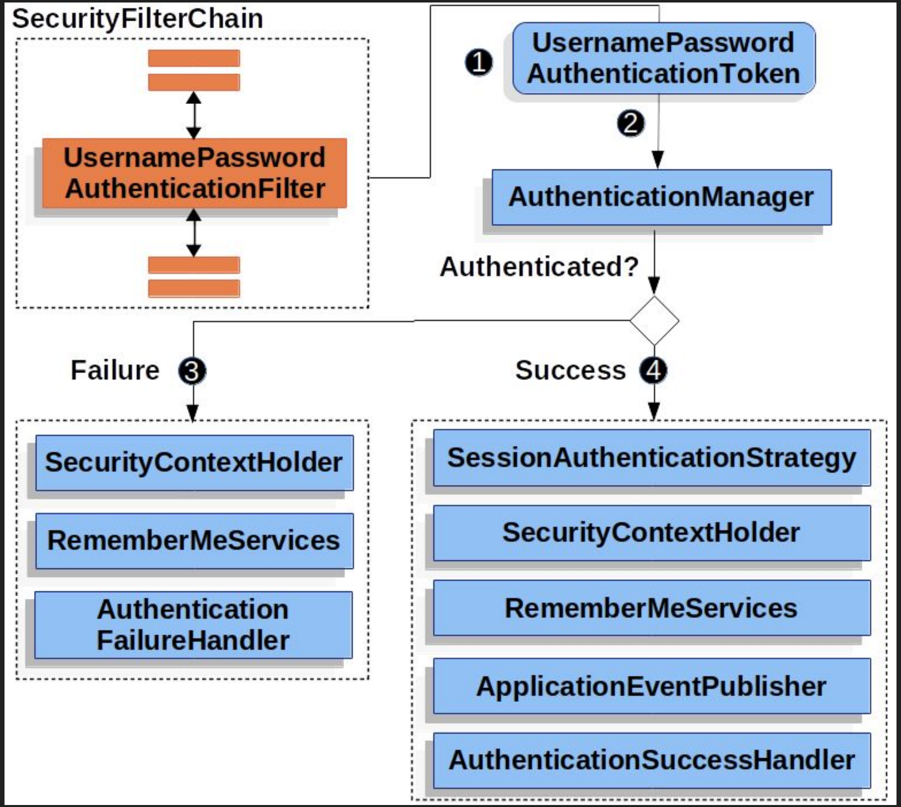
Example - User CRUD flow, Current user Detection

https://github.com/vrudas/spring-framework-examples/tree/main/example-15-user-flow-security

Form Login


Client
Spring Web Application
SecurityFilterChain
❶ GET /private
FilterSecurity Interceptor
❷ AccessDeniedException
ExceptionTranslation Filter
LoginUrl AuthenticationEntryPoint
❸ Location: /login
❹ GET /login
❺ login.html
LoginController

Form Login


SecurityFilterChain
UsernamePassword AuthenticationFilter
1
UsernamePassword AuthenticationToken
2
AuthenticationManager
Authenticated?
Failure 3
Success 4
SecurityContextHolder
RememberMeServices
Authentication FailureHandler
SessionAuthenticationStrategy
SecurityContextHolder
RememberMeServices
ApplicationEventPublisher
AuthenticationSuccessHandler

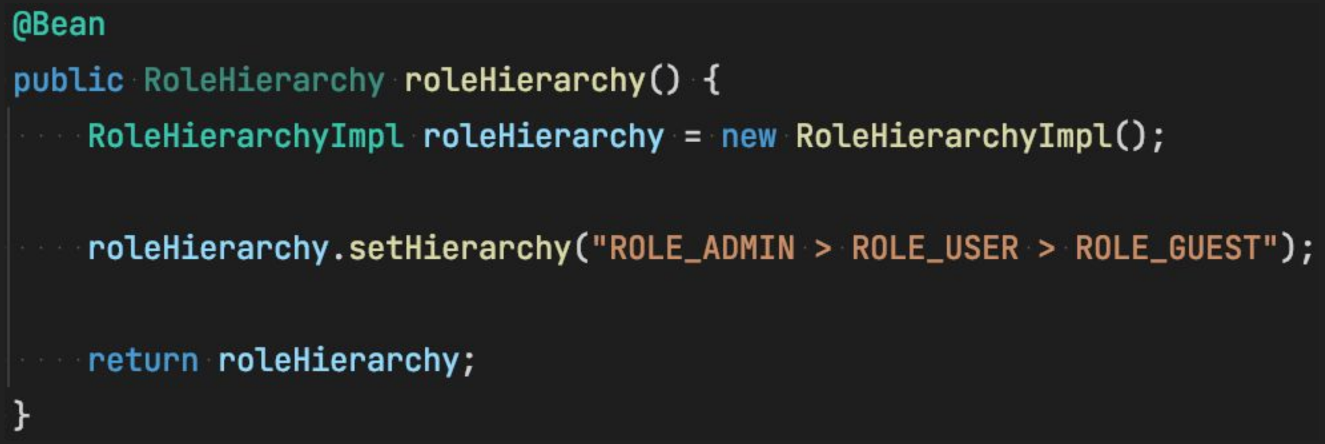
Example - Custom Form Login and AuthenticationProvider

https://github.com/vrudas/spring-framework-examples/tree/main/example-16-custom-auth-provider

Authorization

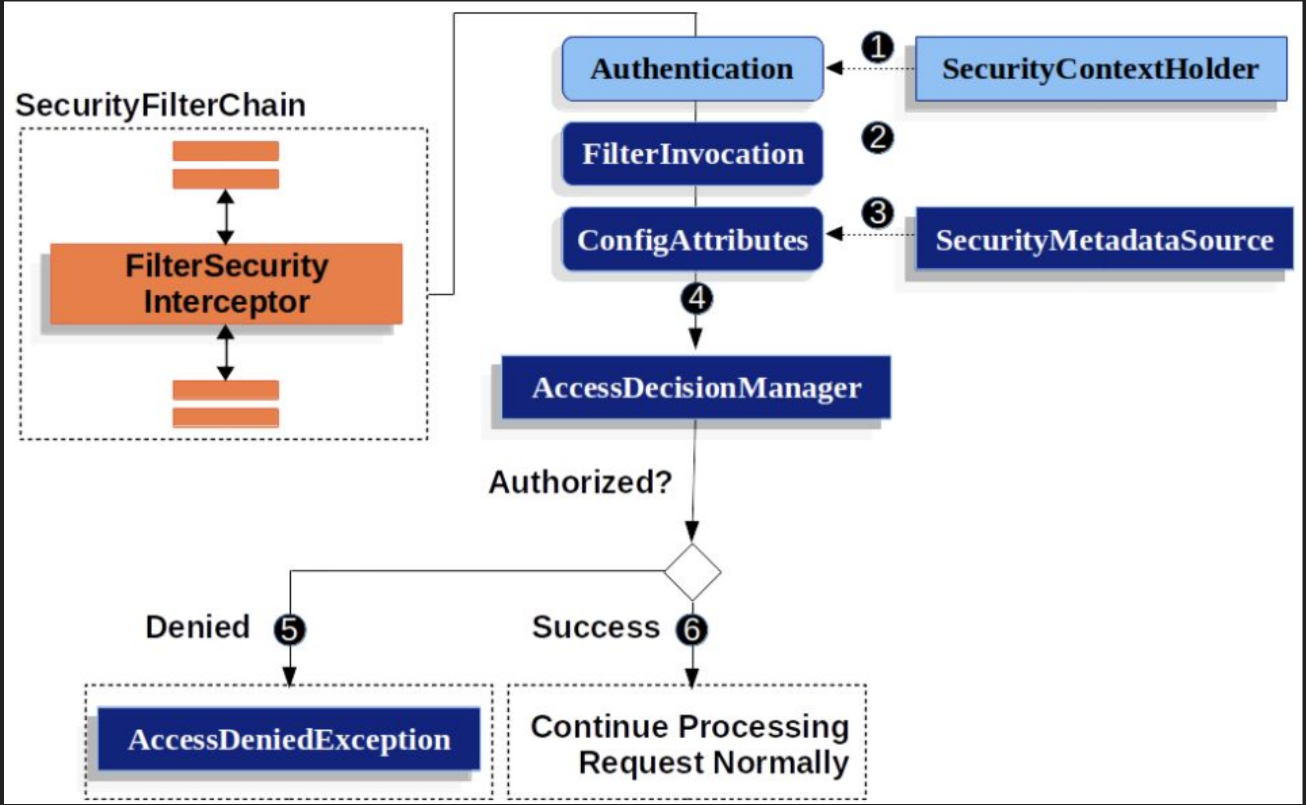
- Authorities - represents actions available for user. Examples:
  - READ_USERS
  - DELETE_USERS
  - …
- Roles - represent user permission in a group-like approach. Examples:
  - ROLE_ANONYMOUS
  - ROLE_GUEST
  - ROLE_USER
  - ROLE_ADMIN
  - ROLE_SUPER_ADMIN
  - …

Role Hierarchy



```java
@Bean
public RoleHierarchy roleHierarchy() {
    RoleHierarchyImpl roleHierarchy = new RoleHierarchyImpl();

    roleHierarchy.setHierarchy("ROLE_ADMIN > ROLE_USER > ROLE_GUEST");

    return roleHierarchy;
}
```

Authorize HttpServletRequest with FilterSecurityInterceptor



Authorization Expression-Based control

- hasRole(String role)
- hasAnyRole(String… roles)
- hasAuthority(String authority)
- hasAnyAuthority(String… authorities)
- principal - Allows direct access to the principal object representing the current user
- authentication - Allows direct access to the current Authentication object obtained from the SecurityContext
- permitAll
- denyAll
- isAnonymous()
- isRememberMe()
- isAuthenticated() - Returns true if the user is not anonymous
- isFullyAuthenticated() - Returns true if the user is not an anonymous or a remember-me user

Example - Authorization

https://github.com/vrudas/spring-framework-examples/tree/main/example-17-authorization

Example - Method security

https://github.com/vrudas/spring-framework-examples/tree/main/example-18-method-security

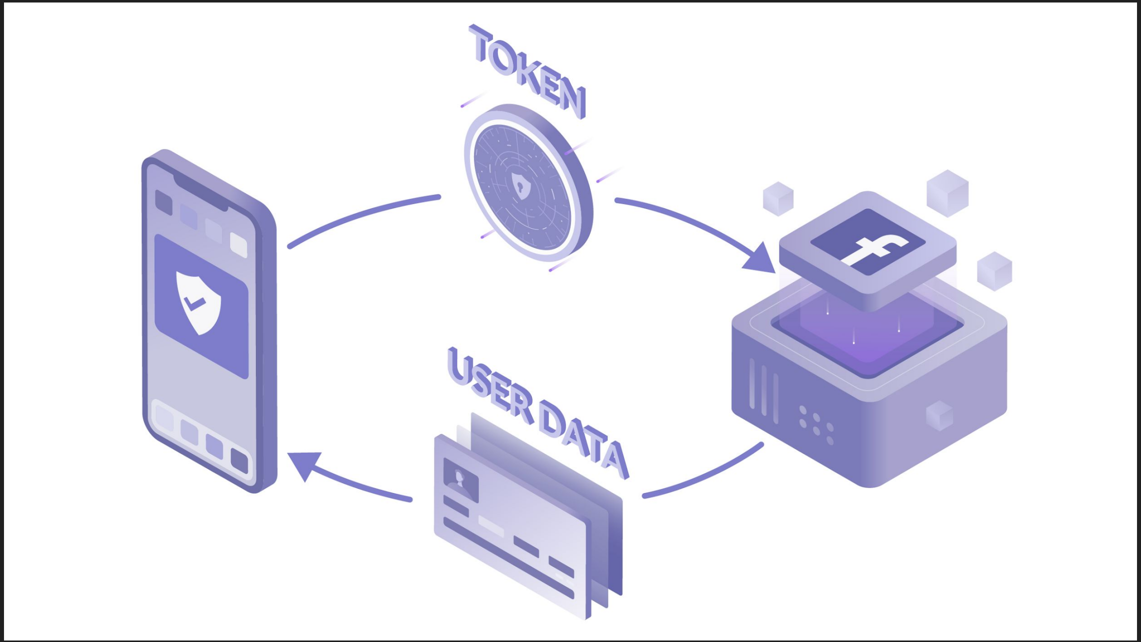
Remember Me - The problem?

REMEMBER
ME
CAPCOM

Example - Remember Me

https://github.com/vrudas/spring-framework-examples/tree/main/example-19-remember-me

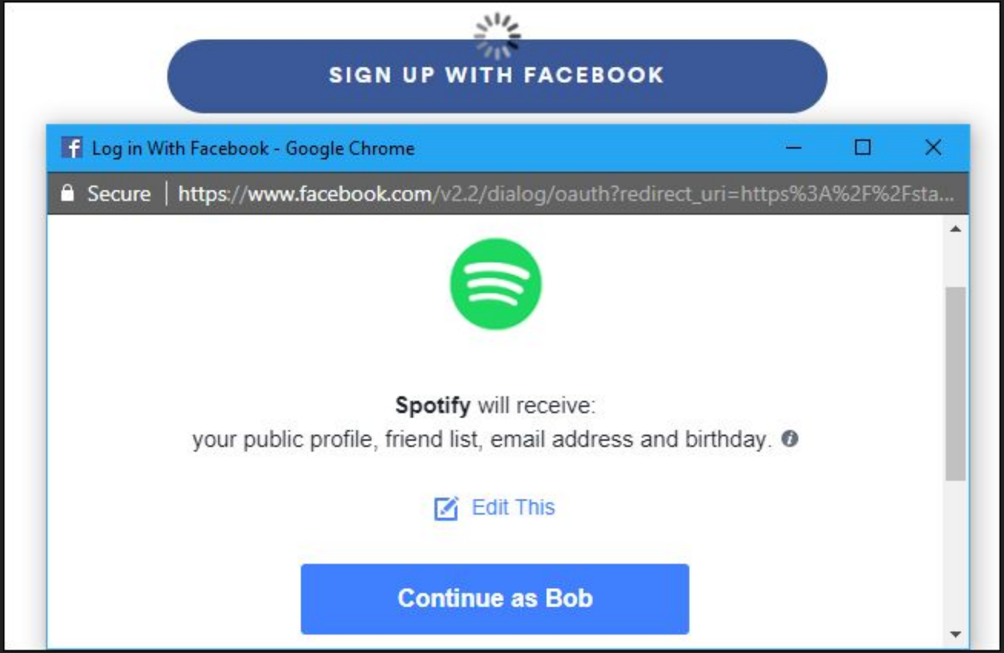
OAuth2 - The Problem?



OAuth Terminology

- Resource Owner: Entity that is capable of granting access to protected resources (Us)
- Resource Server: Server that hosts the protected resources and handles requests for access
- Client: Application that wants to access the Resource Server and perform actions on behalf of the Resource Owner
- Authorization Server: Server that knows the Resource Owner and can authorize the Client to access the Resource Server

What Does OAuth Do?



SIGN UP WITH FACEBOOK

Log in With Facebook - Google Chrome

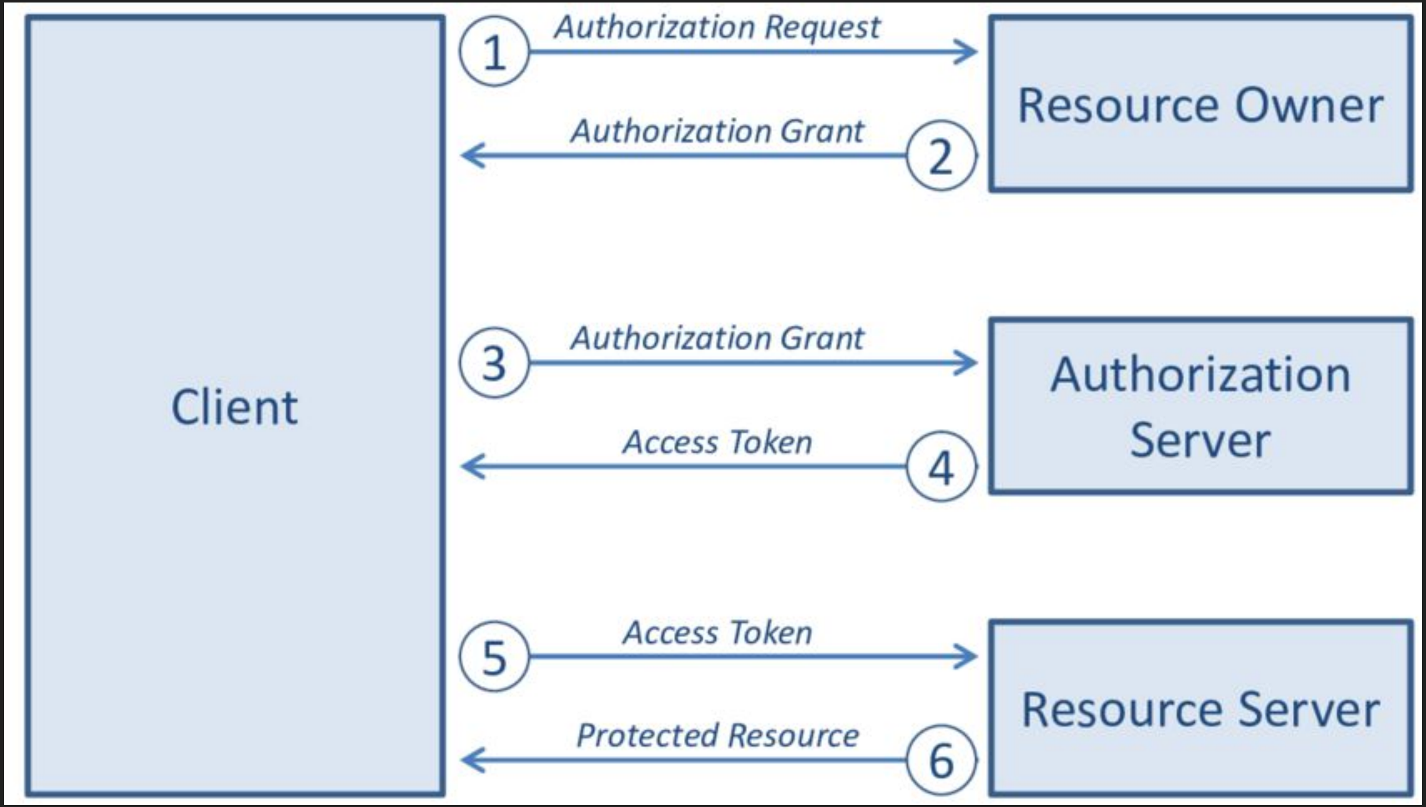
Secure | https://www.facebook.com/v2.2/dialog/oauth?redirect_uri=https%3A%2F%2Fsta...

Spotify will receive:
your public profile, friend list, email address and birthday.

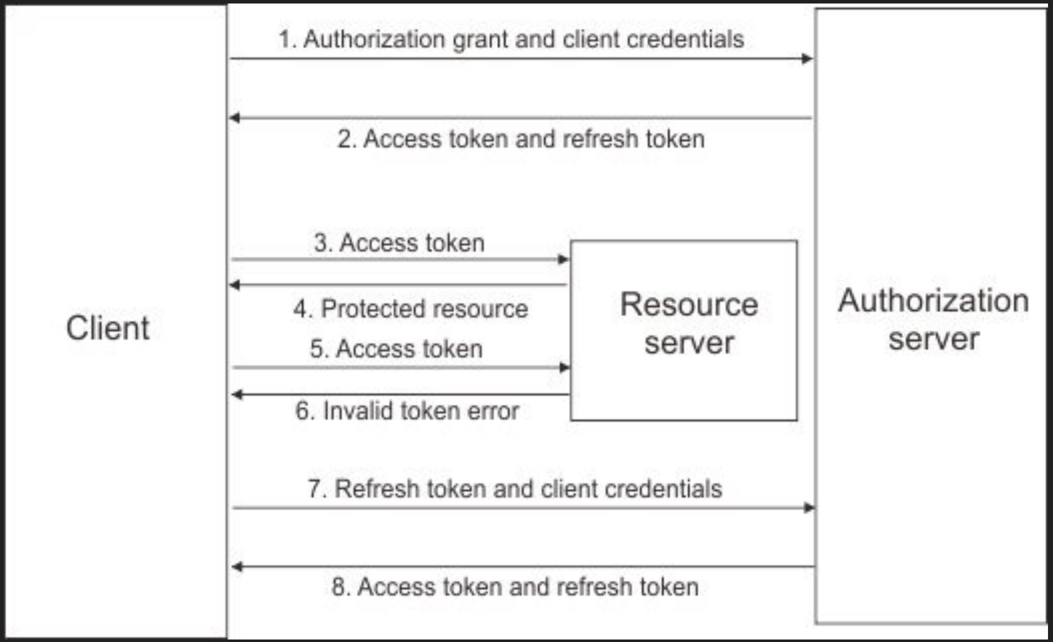
Edit This

Continue as Bob

What Does OAuth Do?



How Does OAuth Work? (JWT under the hood 👀)


1. Authorization grant and client credentials
2. Access token and refresh token
3. Access token
4. Protected resource
5. Access token
6. Invalid token error
7. Refresh token and client credentials
8. Access token and refresh token
Client
Resource server
Authorization server

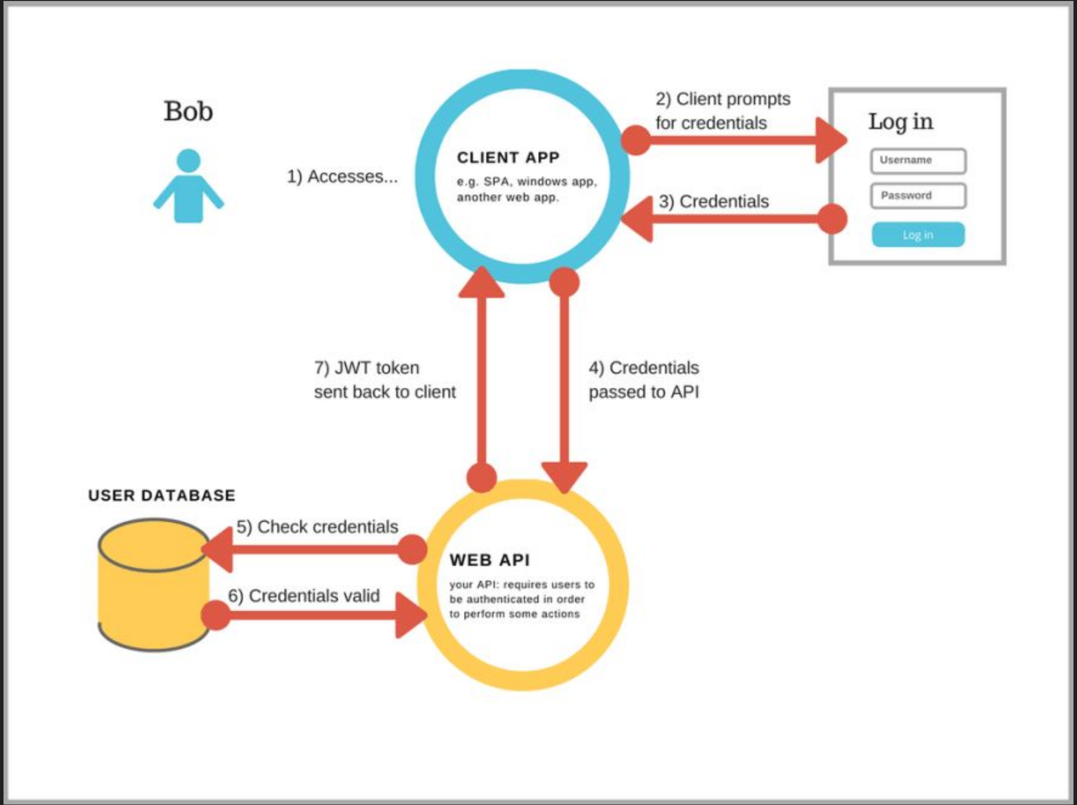
Example - OAuth2.0

https://github.com/vrudas/spring-framework-examples/tree/main/example-20-oauth

JSON Web Token aka JWT

JWT

JWT How it Works?


Bob
1) Accesses...
CLIENT APP
e.g. SPA, windows app, another web app.
2) Client prompts for credentials
Log in
Username
Password
Log in
3) Credentials
4) Credentials passed to API
7) JWT token sent back to client
USER DATABASE
5) Check credentials
6) Credentials valid
WEB API
your API: requires users to be authenticated in order to perform some actions

Example - JWT

https://github.com/vrudas/spring-framework-examples/tree/main/example-21-jwt

Any questions?

More detailed JWT implementation at [Spring Boot JWT]

Congrats!

🎉🎊

💪🎓🎯

💻🏃📈

🌟