Spring Framework

Agenda

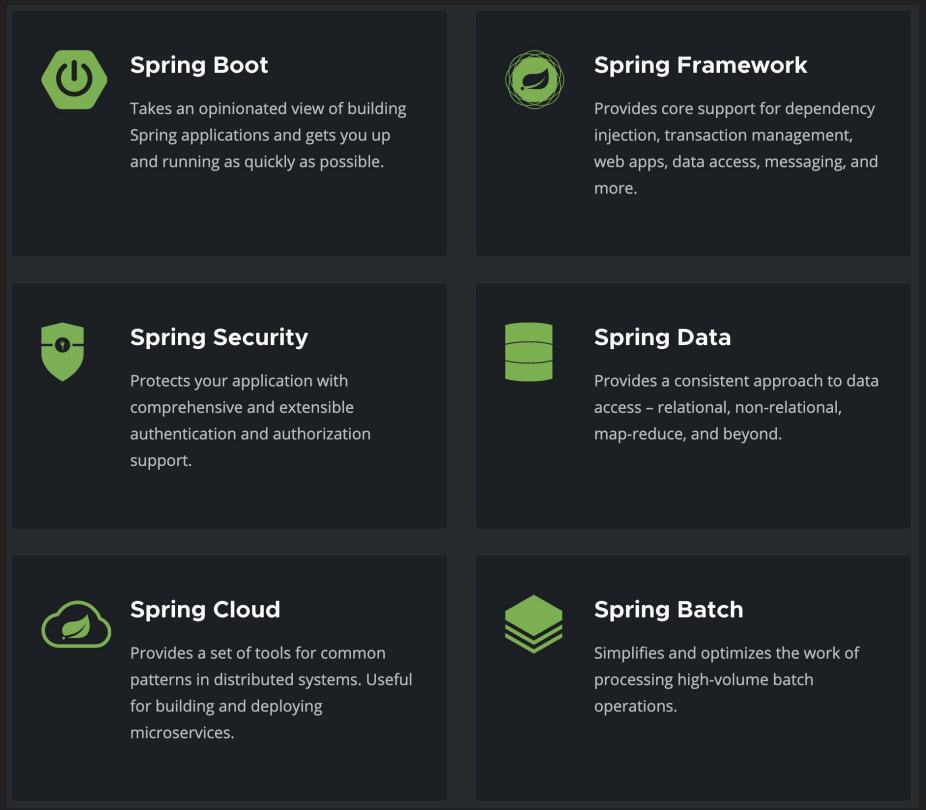
- Overview
- Spring Platform
- Spring Framework
- Inversion of Control
- Dependency Injection

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)

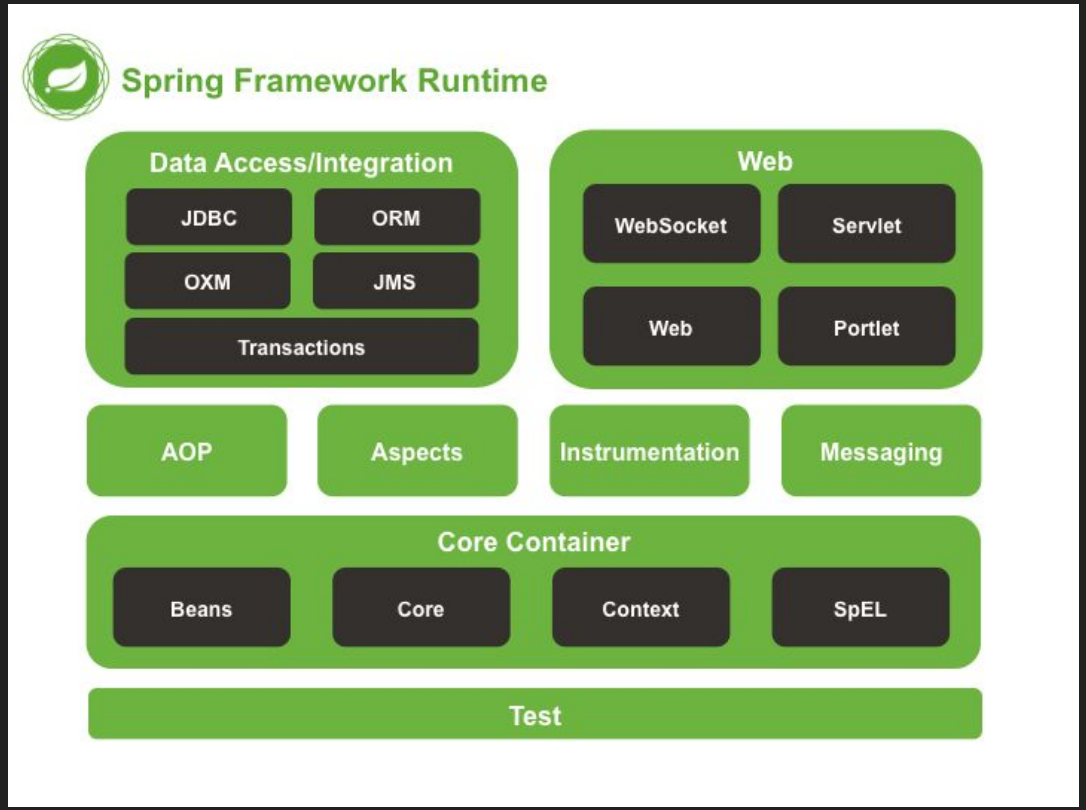
Spring Platform projects, more at Spring Projects



Spring 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.

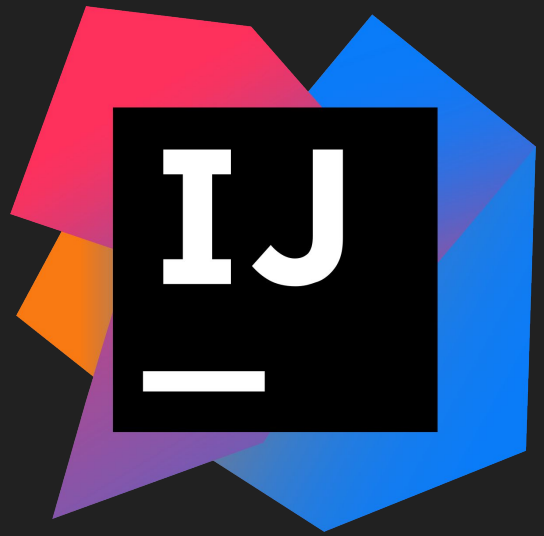
Spring Framework - WEB

- 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



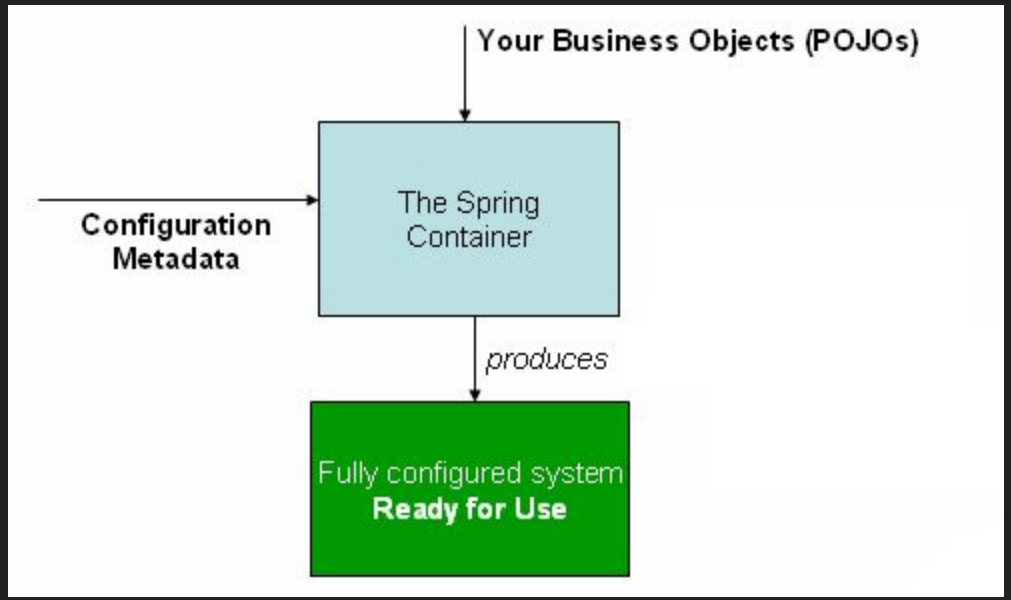
https://github.com/vrudas/spring-framework-examples/tree/main/example-00-hello

Inversion of Control - The Problem?

Inversion of Control - The Problem?

```
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



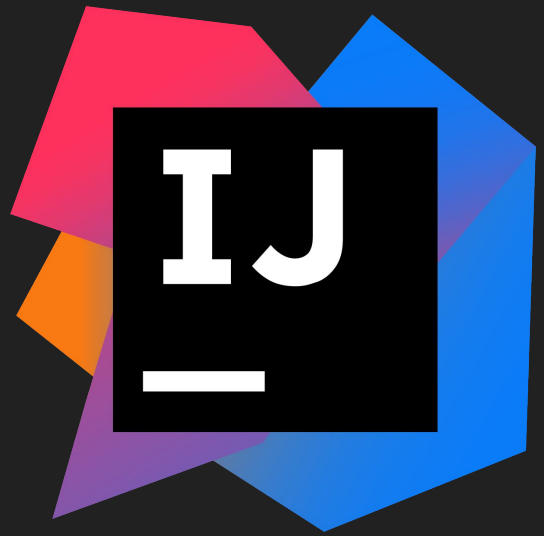
Inversion of Control in Spring

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

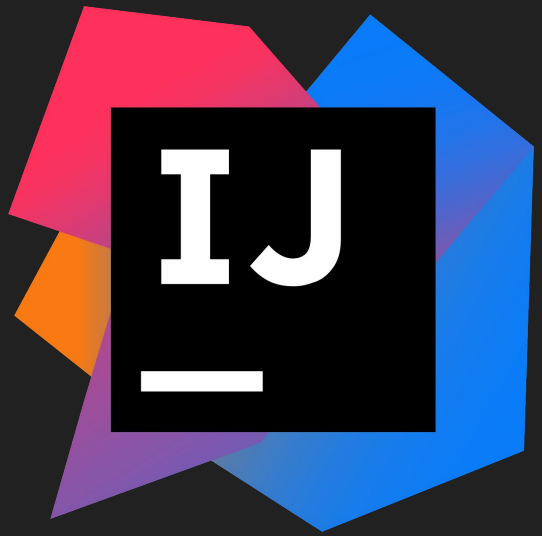
The bean definition contains the information called configuration metadata which is needed for the container to know the followings:

- 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

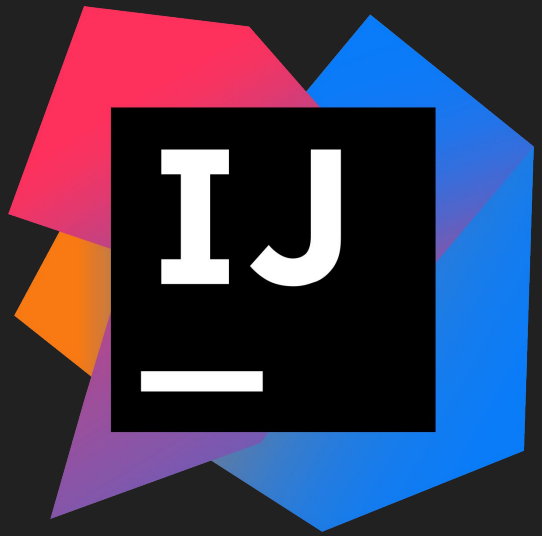


https://github.com/vrudas/spring-framework-examples/tree/main/example-02-bean-definition

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:

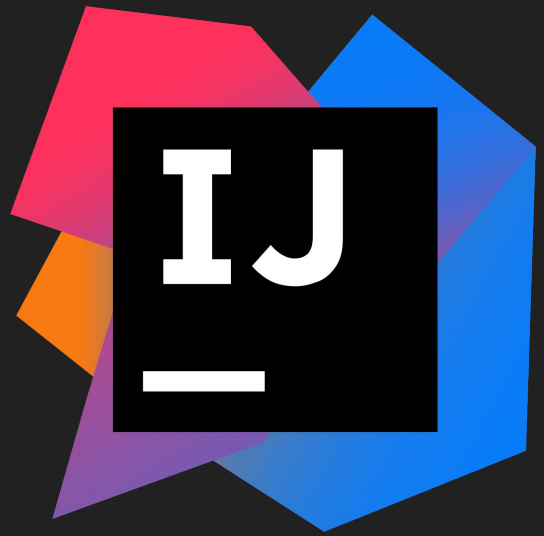
```
@PreDestroy

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.

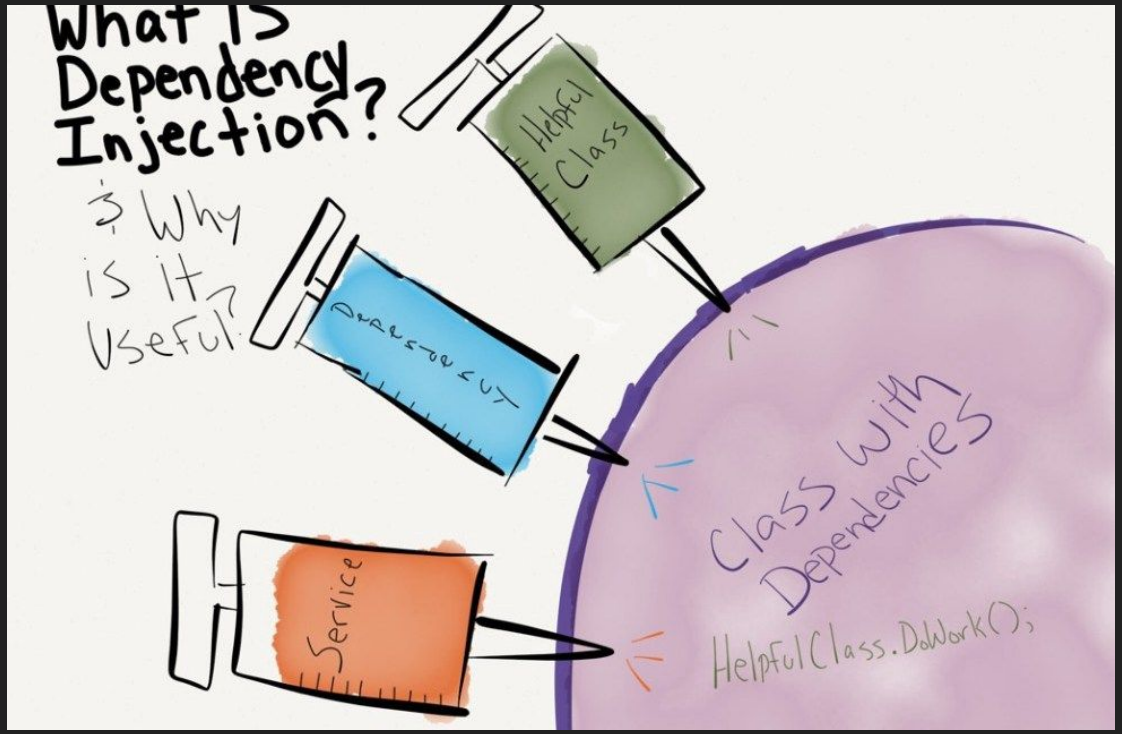
Dependency Injection - The Problem?

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.

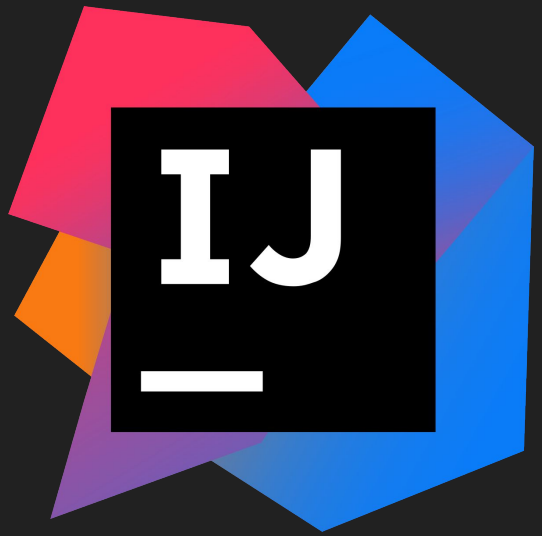
Dependency Injection



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

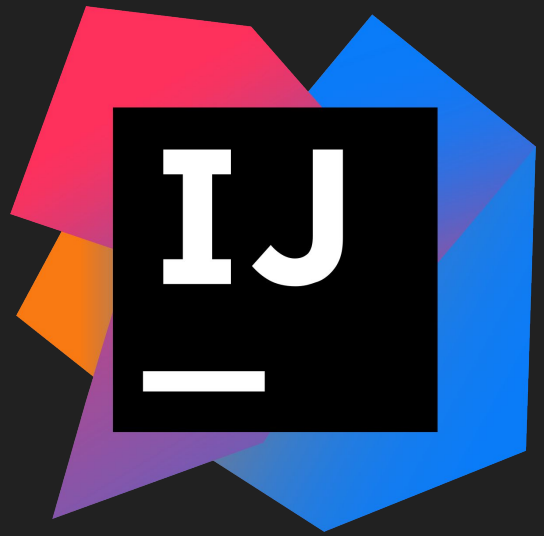


https://github.com/vrudas/spring-framework-examples/tree/main/example-05-dependency-injection

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

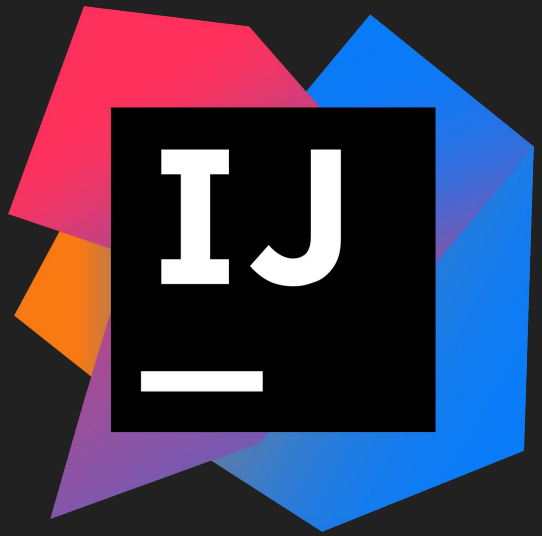
Java Based Configuration

- 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.

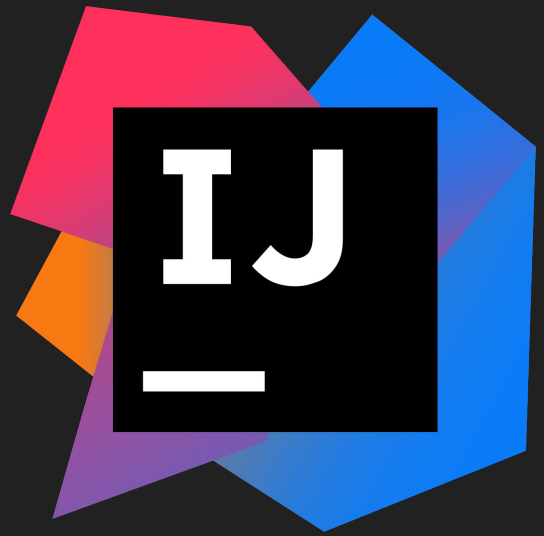
Example - Java Based Configuration



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

Any questions?