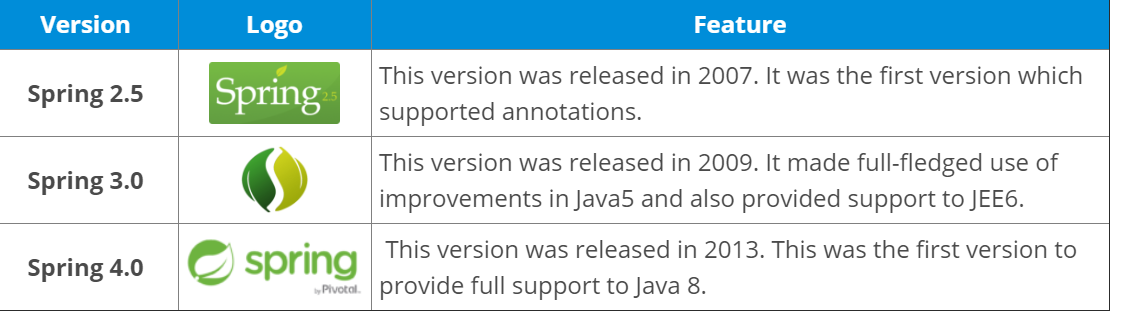
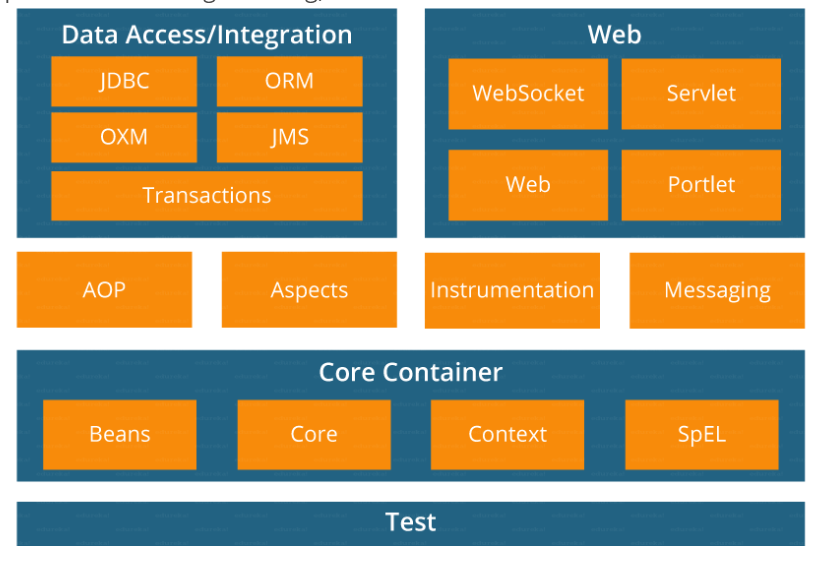
**Spring Questions**

**What are the major features in different versions of Spring Framework?**



**How many modules are there in Spring Framework and what are they?**

There are around 20 modules which are generalized into **Core Container, Data Access/Integration, Web, AOP (Aspect Oriented Programming), Instrumentation and Test**.



**What is Inversion of Control (IoC)?**

**Inversion of Control** is a principle by which control of objects or program is transferred to a container or framework.

In traditional programming our custom code makes call to a library, IoC enables framework to take control of program and call our custom code.

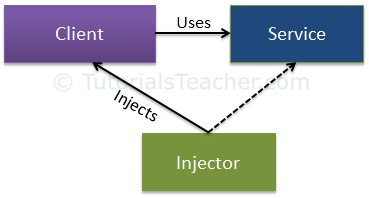
Inversion of Control can be achieved through various mechanisms such as: **Strategy design pattern, Service Locator pattern, Factory pattern, and Dependency Injection (DI).**

**What is Dependency Injection?**

**Dependency Injection** is a pattern through which objects are injected into other objects by an **assembler** rather than by the objects themselves.

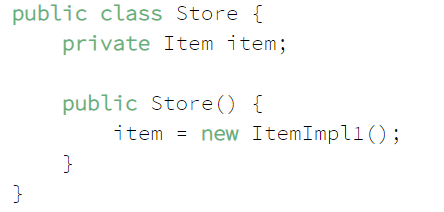
The client delegates the responsibility of providing its dependencies to external code (the injector). The client is not allowed to call the injector code; it is the injecting code that constructs the services and calls the client to inject them.

Dependency Injection means passing a dependent object as a parameter to a method, rather than having the method create the dependent object.

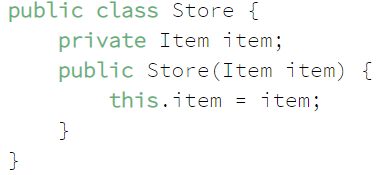


Dependency Injection is one of the patterns to implement IoC, where the control being inverted is setting the object dependencies.

**Traditional programming**



**Dependency Injection**



What are the advantages of Dependency Injection?

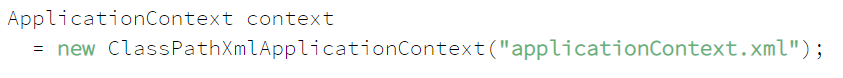
1. DI allows a client the flexibility of being configurable. Only client's behavior is fixed.
2. Unit testing can be performed using mock objects.
3. Helps to enable loose coupling architecture.
4. DI allows a client to remove all knowledge of a concrete implementation that needs to use. It is more reusable, more testable, more readable code.
5. DI does not require any changes in code behavior it can be applied to legacy code as refactoring.
6. Boiler plate code is reduced, as initializing of dependencies is done by the injector component.
7. DI allows concurrent or independent development.

**Spring IoC Container**

IoC container is much more than object creation: In the Spring framework, the IoC container is represented by the interface **ApplicationContext**. The Spring container is **responsible for instantiating, configuring and assembling objects known as beans, as well as managing their lifecycle**.

The Spring framework provides several implementations of the **ApplicationContext** interface — **ClassPathXmlApplicationContext** and **FileSystemXmlApplicationContext** for standalone applications, and **WebApplicationContext** for web applications.

Although programmers relinquish their control on the objects, In order to assemble beans, the container uses configuration metadata, which can be in the form of **XML configuration or Annotations** and must be provided by programmer.



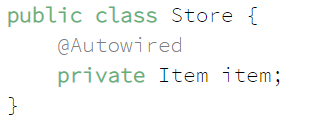
**What are the different types of Dependency Injections are possible in spring?**

* Constructor based Dependency Injection
* Setter based Dependency Injection
* Field based Dependency Injection

The **Spring documentation** recommends using **constructor**-based injection for **mandatory** dependencies, and **setter**-based injection for **optional** ones.

**What are the drawbacks of Field based Injection?**

In case of Field-Based DI, we can inject the dependencies by marking them with an @Autowired annotation:



**Drawbacks**

* This method **uses reflection to inject the dependencies**, which is costlier than constructor-based or setter-based injection
* It's **easy to keep adding multiple dependencies** using this approach. If you were using **constructor injection having multiple arguments** would have **made us think** that the class does more than one thing which can **violate the Single Responsibility Principle**.

**Which Is the Best Way of Injecting Beans and Why?**

The recommended approach is to use **constructor arguments for mandatory dependencies** and setters for optional ones. Constructor injection allows injecting values to immutable fields and **makes testing easier.**

**What are the different types of Autowiring is possible in spring?**

Wiring allows the Spring container to automatically resolve dependencies between collaborating beans by inspecting the beans that have been defined.

There are four modes of autowiring a bean using an XML configuration:

* **no**: the default value – this means no autowiring is used for the bean and we have to explicitly name the dependencies
* **byName**: autowiring is done based on the name of the property, therefore Spring will look for a bean with the same name as the property that needs to be set
* **byType**: similar to the byName autowiring, only based on the type of the property. This means Spring will look for a bean with the same type of the property to set. If there's more than one bean of that type, the framework throws an exception.
* **constructor**: autowiring is done based on constructor arguments, meaning Spring will look for beans with the same type as the constructor arguments.

**What is the difference between BeanFactory and ApplicationContext?**

**BeanFactory** is an interface representing a container that provides and manages bean instances. The default implementation **instantiates beans lazily** when getBean() is called.

**ApplicationContext** is an interface representing a container holding all information, metadata, and beans in the application. It also extends the BeanFactory interface but the default implementation **instantiates beans eagerly** **when the application starts**. This behavior can be overridden for individual beans. Application Context creates all the beans at the time of instantiation.

**What are the different bean scope?**

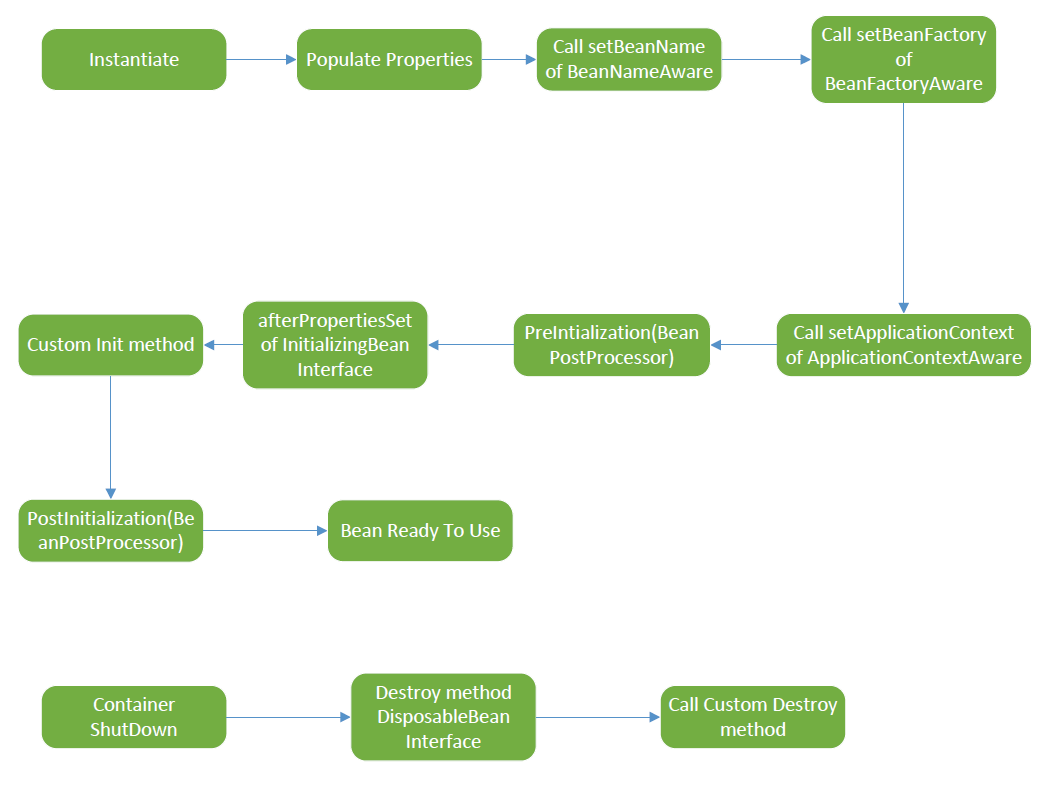
To set Spring Bean's scope, we can use **@Scope** annotation or “**scope**” attribute in XML configuration files. There are five supported scopes:

* singleton (default)
* prototype (**As a rule, use the prototype scope for all stateful beans and the singleton scope for stateless beans.)**
* request – Only Valid in context of web-aware spring application context.
* session– Only Valid in context of web-aware spring application context.
* global-session– A single instance per global session. Used with portlets. Only Valid in context of web-aware spring application context.
* application – bean is scoped to lifecycle of a ServletContext
* Websocket

**Are Singleton beans thread-safe?**

No, **singleton beans are not thread-safe**, as thread safety is about execution, whereas the singleton is a design pattern focusing on creation. Thread safety depends only on the bean implementation itself.

**What is Spring Bean Life Cycle?**



**What are the callback methods?**

InitializingBean.afterPropertiesSet()

DisposableBean.destroy()

Lifecycle annotations

@PostConstruct

@PreDestroy

**What is BeanPostProcessor?**

It gives a mean to tap into the Spring context life cycle and interact with beans as they are processed.

* PostProcessBeforeInitialization()
* PostProcessAfterInitialization()

What is @Qualifier and @Primary annotation?

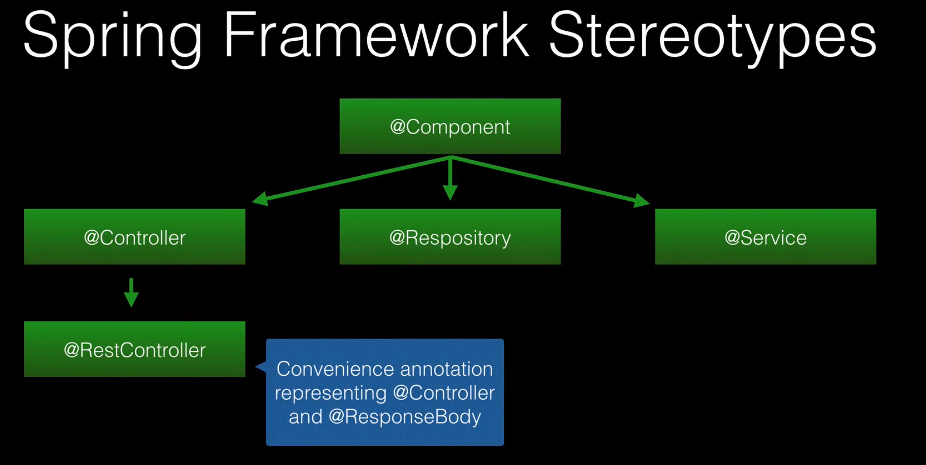
**Spring Configuration**

What are the Spring Configuration options available?

* **XML Based Configuration** was introduced in Spring Framework 2.0
* **Annotation Based Configuration** was introduced in Spring Framework 3.0 – Component Scans, @Controller, @Service, @Repository, @Component etc.
* **Java Based Configuration** was introduced in Spring Framework 3.0 - **@Configuration, @Bean**
* **Groovy Bean Definition** DSL Configuration was introduced in Spring Framework 4.0

**What are Stereotypes?**

It is used to define spring bean in spring context using **@Controller, @Service, @Repository, @Component**



**What is special about @Repository?**

Spring will detect the platform specific persistence exception and re-throw them as Spring exception.

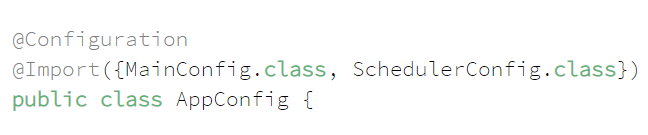
**What is @ComponentScan?**

It scans packages and subpackages for Spring Bean. Configuring @ComponentScan override the scanning done by @SpringBootApplication

**Can We Have Multiple Spring Configuration Files in One Project?**

Yes, in large projects, having multiple Spring configurations is recommended to increase maintainability and modularity.

You can load multiple Java-based configuration files:



**Spring Boot Auto Configuration**

**What is Spring Boot Starter?**

It is a POM which declares the common set of dependencies. Spring boot starter is available for most of the java projects.

**What are the different annotation included in @SpringBootApplication Annotation?**

@**SpringBootApplication**

- @**Configuration** – Declares the class as Spring Configuration

- @**EnableAutoConfiguration** – Enables auto configuration

- @ **ComponentScan** – Scans for the component in current packages and sub-packages

**How to configure an external class coming as jar in spring project as bean?**

Configuring it using @Configuration and @Bean

**Spring External Properties**

**What are the ways to load the external properties in Spring?**

1. **@Value** annotation
2. **@ConfigurationProperties** annotation
3. **PropertySourcePlaceholderConfigurer**

**@PropertySource** is used to define the path of properties file.

**Spring Data JPA**

**What are the different Types of Relationship?**

* @OneToOne
* @OneToMany
* @ManyToOne (inverse of OneToMany)
* @ManyToMany (A join table used to define the relationship)

**Difference between Unidirectional vs Bidirectional?**

Bidirectional is preferred since you can navigate the object graph in either direction.

**What are the different Fetch Type?**

* Lazy Fetch Type
* Eager Fetch Type

**What are the different Cascade Types?**

* PERSIST
* MERGE
* REFRESH
* REMOVE
* DETACH
* ALL

By default, no operations are cascaded.

**What are the different types of Inheritance?**

**MappedSuperclass** – Entities inherit from a super class. A database table is not created for the super class.

**Single Table** – Hibernate Default – **One table is used for all the subclasses**.

**Joined Table** – Base class and subclasses have their own table. Fetching subclass entities require a join to the parent table.

**Table Per Class** – Each subclass has its own table.

**How can you Create and Update Timestamps for entity?**

@PrePersist @PreUpdate -- JPA

@CreationTimestamp @UpdateTimestamp -- Hibernate

**Spring Exception Handling**

**@ResponseStatus** – Allows to **annotate custom exception classes** to indicate to the framework the **HTTP status** you want returned when that exception is thrown. Global to the application.

**@ExceptionHandler** – Works at the **controller level**. Allows to define custom exception handling. Can be used with **@ResponseStatus**. Can be used to return view. Also can take total control of Model and View.

HandlerExceptionResolver – It is an interface you can implement for custom exception handling. It is used internally by Spring MVC.

- ExceptionHandlerExceptionResolver

- ResponseStatusExceptionResolver

- DefaultHandlerExceptionResolver

SimpleMappingExceptionResolver – A spring bean you can define to map exceptions to specific views.

**@ControllerAdvice** – It is used to globally handle the exception.

**Spring Data Validation**

JSR-303 Bean Validation 2.0

Hibernate Validator (Implementation of Bean Validation)

**javax.validation.constraints**

**org.hibernate.validator**

@NotNull @Size etc.

BindingResults can be used to show the low level errors.

**BindingResult** is an interface from **org.springframework.validation** package that represents binding results. We can use it to detect and report errors in the submitted form.

**Spring Internationalization**

**LocaleResolver**

* + AcceptHeaderLocaleResolver (Spring Boot Default)
  + CookieLocaleResolver
  + SessionLocaleResolver

**What is Thymeleaf?**

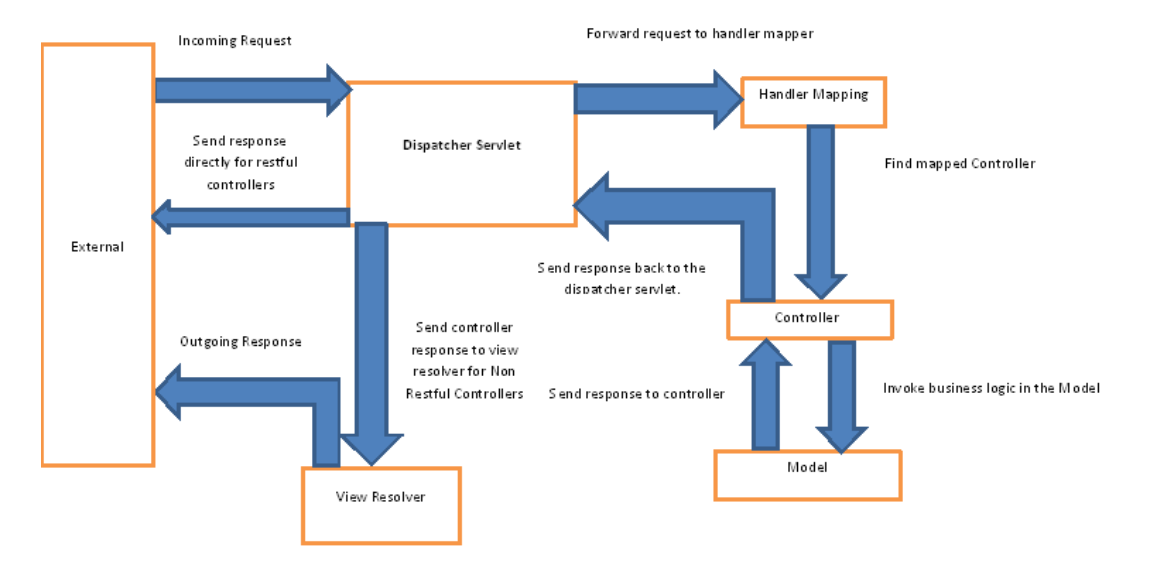
Thymeleaf is a java template engine producing XML, XHTML and HTML5. It is a replacement of JSPs. It is not a web framework and not tied to web environment. It is valid HTML document that can be viewed in browser, it helps to speed up the development.

**Spring Web MVC**

Explain **FrontController** in Spring MVC Architecture?

At a very high level, here are the main responsibilities we're looking at:

* Intercepts incoming requests
* Converts the payload of the request to the internal structure of the data
* Sends the data to Model for further processing
* Gets processed data from the Model and advances that data to the View for rendering



**How Does the @RequestMapping Annotation Work?**

The @RequestMapping annotation is used to map web requests to Spring Controller methods. In addition to simple use cases, we can use it for mapping of HTTP headers, binding parts of the URI with @PathVariable, and working with URI parameters and the @RequestParam annotation.

**What is @ModelAttribute**?

It binds a method parameter, or a method return value to a named model attribute and then exposes it to a web view.

**Explain Model, ModelMap and ModelAndView?**

The **Model** interface defines a holder for model attributes. The **ModelMap** has a similar purpose, with the ability to pass a collection of values. It then treats those values as if they were within a Map. We should note that in Model (ModelMap) we can only store data. We put data in and return a view name.

On the other hand, with the **ModelAndView**, we return the object itself. We set all the required information, like the data and the view name, in the object we're returning.

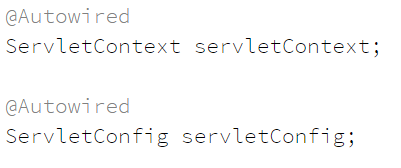
**What is ViewResolver in Spring?**

The **ViewResolver** enables an application to render models in the browser – without tying the implementation to a specific view technology – by mapping view names to actual views.

**How to Get ServletContext and ServletConfig Objects in a Spring Bean?**

You can do either by:

* Implementing Spring-aware interfaces. The complete list is available here.
* Using @Autowired annotation on those beans:



**What are DispatcherServlet and ContextLoaderListener?**

Simply put, in the **Front Controller design pattern**, a single controller is responsible for directing incoming Http requests to all of the application’s other controllers and handlers. Spring’s **DispatcherServlet** implements this pattern and is, therefore, responsible for correctly coordinating the Http requests to the right handlers.

On the other hand, **ContextLoaderListener** starts up and shuts down Spring’s root **WebApplicationContext**. It ties the lifecycle of **ApplicationContext** to the lifecycle of the **ServletContext**. We can use it to define shared beans working across different Spring contexts.

**What is Spring MVC Interceptor and How to Use It?**

**Spring MVC Interceptors** allow us to intercept a client request and process it at three places – before handling, after handling, or after completion (when the view is rendered) of a request. The interceptor can be used for cross-cutting concerns and to avoid repetitive handler code like **logging, changing globally used parameters in Spring model**, etc. **HandlerInterceptor** **interface preHandle(), postHandle(), afterCompletion().**

**What is @ControllerAdvice?**

The **@ControllerAdvice** annotation allows us to write global code applicable to a wide range of controllers.

**What is @ExceptionHandler?**

The **@ExceptionHandler** annotation allows us to define a method that will handle the exceptions. We may use the annotation independently, but it's a far better option to use it together with the **@ControllerAdvice**.

**Explain exception handling in Spring MVC?**

We have three options for exceptions handling in Spring MVC:

* per exception
* per controller
* globally

1. If an unhandled exception is thrown during web request processing, the server will return an HTTP 500 response. To prevent this, we should annotate any of our custom exceptions with the **@ResponseStatus** annotation. This kind of exceptions is resolved by **HandlerExceptionResolver**. This will cause the server to return an appropriate HTTP response with the specified status code when a controller method throws our exception. We should keep in mind that we shouldn't handle our exception somewhere else for this approach to work.
2. Another way to handle the exceptions is by using the **@ExceptionHandler** annotation. We add @ExceptionHandler methods to any controller and use them to handle the exceptions thrown from inside that controller. These methods can handle exceptions without the **@ResponseStatus** annotation, redirect the user to a dedicated error view, or build a totally custom error response. We can also pass in the servlet-related objects (HttpServletRequest, HttpServletResponse, HttpSession, and Principal) as the parameters of the handler methods. But, we should remember that we can't put the Model object as the parameter directly.
3. The third option for handling errors is by **@ControllerAdvice** classes. It'll allow us to apply the same techniques, only this time at the application level and not only to the particular controller. To enable this, we need to use the @ControllerAdvice and the **@ExceptionHandler** together. This way exception handlers will handle exceptions thrown by any controller.

**Spring Data Access**

**What Is Spring Jdbctemplate Class and How to Use It?**

The **Spring JDBC template** is the primary API through which we can access database operations logic that we’re interested in:

* creation and closing of connections
* executing statements and stored procedure calls
* iterating over the ResultSet and returning results

To use it, we'll need to define the simple configuration of **DataSource**:



**How Would You Enable Transactions in Spring and What Are Their Benefits?**

There are two distinct ways to configure Transactions – with **annotations** or by using **Aspect Oriented Programming (AOP)** – each with their advantages.

The benefits of using **Spring Transactions** are:

* Provides a consistent programming model across different transaction APIs such as JTA, JDBC, Hibernate, JPA, and JDO
* Support declarative transaction management
* Provide a simpler API for programmatic transaction management than some complex transaction APIs such as JTA
* Integrate very well with Spring's various data access abstractions

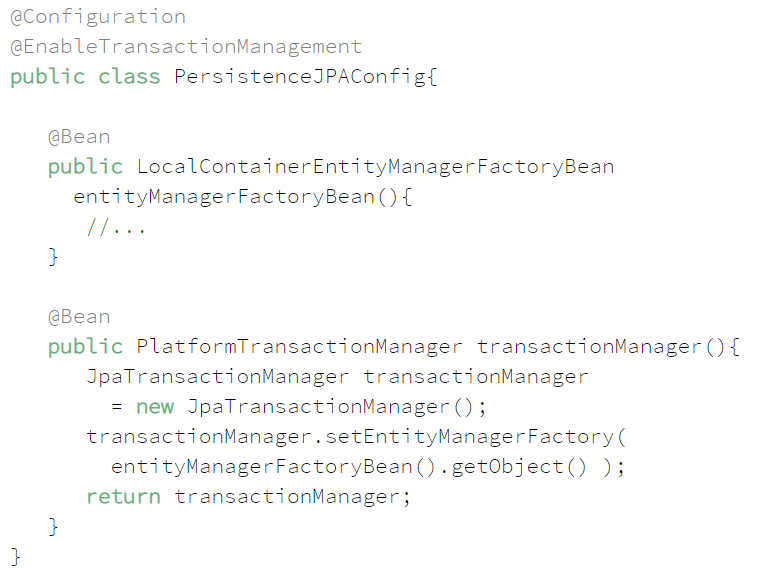
**What Is Spring Dao?**

**Spring Data Access Object** is Spring's support provided to work with data access technologies like JDBC, Hibernate, and JPA in a consistent and easy way.

**How can you configure Transaction in Spring?**

Basically, there are two distinct ways to configure Transactions – **Annotations** and **AOP**

**Spring 3.1** introduces the **@EnableTransactionManagement** annotation that we can use in a **@Configuration** class and configure **TransactionManager**:



However, if we're using a **Spring Boot project**, and have a spring-data-\* or spring-tx dependencies on the classpath, then **transaction management** will be **enabled** by **default**.

**How does @Transactional works in Spring?**

At a high level, Spring creates **proxies** for all the classes annotated with @Transactional – either on the class or on any of the methods. The proxy allows the framework to inject transactional logic before and after the running method – mainly for starting and committing the transaction.

There are two things we need to take care of:

* Any self-invocation calls will not start any transaction.
* only public methods should be annotated with @Transactional.

**Spring Aspect-Oriented Programming (AOP)**

What Is **Aspect-Oriented Programming**?

**Aspects** enable the modularization of cross-cutting concerns such as transaction management that span multiple types and objects by adding extra behavior to already existing code without modifying affected classes.

What is **Aspect**, **Advice**, **Pointcut**, and **Joinpoint** in Aop?

* **Aspect**: a class that implements cross-cutting concerns, such as transaction management
* **Advice**: the methods that get executed when a specific JoinPoint with matching Pointcut is reached in the application
* **Pointcut**: a set of regular expressions that are matched with JoinPoint to determine whether Advice needs to be executed or not
* **JoinPoint**: a point during the execution of a program, such as the execution of a method or the handling of an exception

What are the different types of **Advices**?

* **Before**: These types of advices execute before the joinpoint methods and are configured using @Before annotation mark.
* **After returning**: These types of advices execute after the joinpoint methods completes executing normally and are configured using @AfterReturning annotation mark.
* **After throwing**: These types of advices execute only if joinpoint method exits by throwing an exception and are configured using @AfterThrowing annotation mark.
* **After (finally)**: These types of advices execute after a joinpoint method, regardless of the method’s exit whether normally or exceptional return and are configured using @After annotation mark.
* **Around**: These types of advices execute before and after a joinpoint and are configured using @Around annotation mark.

What Is **Weaving**?

**Weaving** is a process that **links** **aspects** with other application types or **objects to create an advised object**. This can be done at compile time, load time, or at runtime. Spring AOP, like other pure Java AOP frameworks, performs weaving at runtime.

**What are the new features in Spring 5?**

**What are inner beans in Spring?**

A bean can be declared as an inner bean only when it is used as a property of another bean. For defining a bean, the Spring’s XML based configuration metadata provides the use of <bean> element inside the <property> or <constructor-arg>. Inner beans are always anonymous and they are always scoped as prototypes.

References

<https://www.baeldung.com/spring-interview-questions>

<https://www.baeldung.com/inversion-control-and-dependency-injection-in-spring>

<https://dzone.com/articles/ioc-vs-di>

<https://www.youtube.com/watch?v=EPv9-cHEmQw>

**Content Negotiation**

We can provide the content type we want in “Accept” header and based on the Accept header value API/Resource returns the response in corresponding format.

Internationalization

- LocalResolver

Default Locale – Locale.US

- ResourceBundleMessageSource – It is used to read the message properties file.

Spring Actuator

Spring HATEOAS

**Versioning**

1. URI Versioning – Twitter
2. Request Parameter Versioning - Amazon
3. Custom Headers Versioning - Microsoft
4. Media type versioning (a.k.a. "content negotiation" or "accept header") – GitHub

**Factors to decide Versioning**

* URI Pollution
* Misuse of HTTP Headers
* Caching
* Can we execute request on the browser?
* API Documentation

There is no perfect solution for versioning.

**Spring Security**

**What is Spring Security?**

It’s an Application Security Framework which provides following features.

* Login and Logout functionality
* Allow/Block access to URLs to logged in users
* Allow/Block access to URLs to logged in users and with certain Roles
* Provides uniform way of handling security which flexible and customizable
* Handles common vulnerabilities

**What Spring Security can do?**

* Username/password authentication
* SSO/Okta/LDAP
* App Level Authorization
* Intra App Authorization like OAuth
* Microservice Security (using tokens, JWT)
* Method Level Security

**What are the five core concepts of Spring Security?**

**Authentication** – Who are you? Show your id that your authentication.

There are other ways of authentication.

* Knowledge based authentication
* Possession based authentication
* Multifactor Authentication

**Authorization** – What do you want? Are you authorize to access the resource you’re requesting for?

**Principal** – Currently logged in user. Application remembers the Principal once the user has been authenticated.

**Granted Authority** – Provides list of authorities the logged in user have.

**Role** – Group of authorities

**How to add Spring Security to Spring Boot project?**

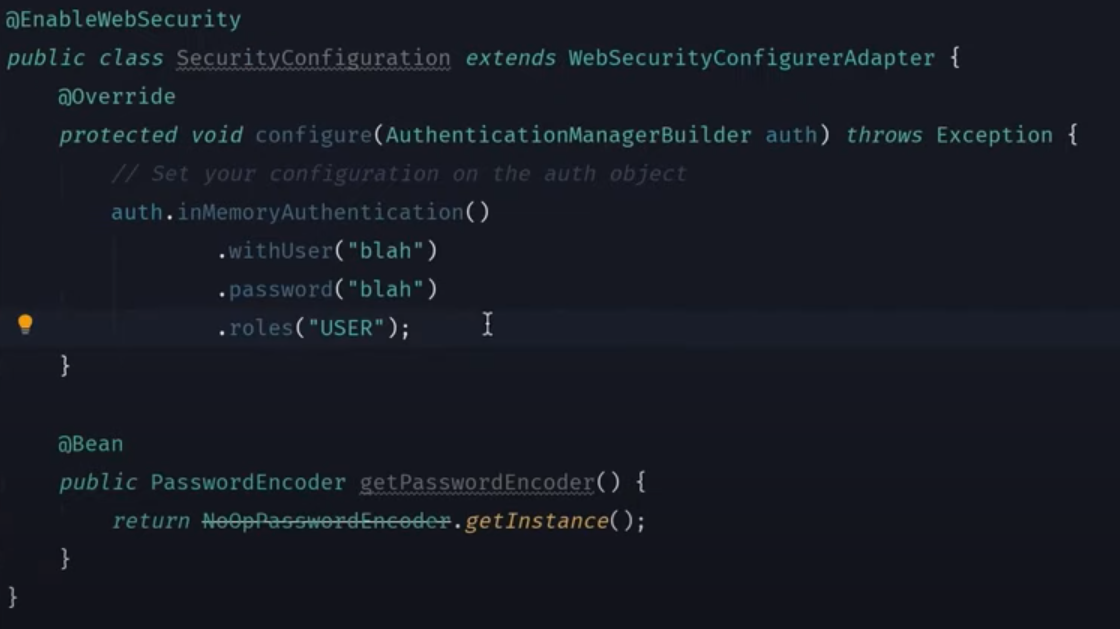
As soon as you use dependency spring-boot-starter-security, Spring Security by default

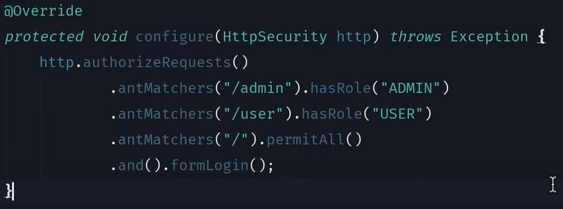
* Adds the mandatory authentication for URLs
* Adds login form
* Handles login error
* Creates a user and sets a default password (if user and password has not been provided in configuration)

How to configure Authentication and Authorization?

AuthenticationManagerBuilder – Authentication

HttpSecurity - Authorization





Different Types of Authentication using Spring Security?

* In Memory Authentication
* JDBC Authentication
* JPA Authentication
* LDAP Authentication
* JWT Authorization
* OAuth Authentication with Facebook, Github etc.

**What is JWT**?

It stands for the **JSON Web Token**. Its popular way of doing **user authorization** in web app and microservices today. **JWT is specific to Authorization**.

Authorization Strategies:

* **Session Token** – When you authenticate server creates a session and keeps a track of it. Server creates Session Id for that session and return it to the client. Subsequently, Client passes this Session Id in every request. Server used to verify the Session Id. Session Id can be stored in cookie.

**Problem with this approach**

* + It assumes there is always one monolithic server.
  + Modern web apps share the load behind the load balancer. Load balancer cannot route the Session Id to the same server for every request.
  + To solve this problem, we can introduce Shared Cache (Redis) for session management but then we introduce the single point of failure.
  + Some implementation follows Sticky Session pattern but that is not scalable. Moreover, it doesn’t work in microservices world.
* **JWT Token** – When you authenticate server returns the signed token back to the client. Client send this token with every request to the server and server can validate authenticity based on signature

JWT has three parts to it.

* + Header
  + Payload
  + Signature

Anybody can encrypt/decrypt the JWT token except the Signature. Server validates signature using the Secret Key which was used to create the token.

Do not put any sensitive information in the payload.

**What if someone steals my JWT and uses it themselves?**

It should be used with HTTPS and in conjunction with other Authorization and Authentication mechanism such as OAuth.

What is OAuth?

OAuth was originally designed for Authorization between services.

The OAuth 2.0 authorization framework is a protocol that allows a user to grant a third-party web site or application **“secure delegated access”** to the user's protected resources, without necessarily revealing their long-term credentials or even their identity.

OAuth introduces an authorization layer and separates the role of the client from that of the resource owner. In OAuth, the client requests access to resources controlled by the resource owner and hosted by the resource server and is issued a different set of credentials than those of the resource owner. Instead of using the resource owner's credentials to access protected resources, the client obtains an access token--a string denoting a specific scope, lifetime, and other access attributes. Access tokens are issued to third-party clients by an authorization server with the approval of the resource owner. Then the client uses the access token to access the protected resources hosted by the resource server.

Auth0 generates access tokens for API authorization scenarios, in JSON web token (JWT) format. The permissions represented by the access token, in OAuth terms, are known as scopes. When an application authenticates with Auth0, it specifies the scopes it wants. If those scopes are authorized by the user, then the access token will represent these authorized scopes.

**Roles**

An OAuth 2.0 flow has the following roles:

**Resource Owner**: Entity that can grant access to a protected resource. Typically, this is the end-user.

**Resource Server**: Server hosting the protected resources. This is the API you want to access.

**Client**: Application requesting access to a protected resource on behalf of the Resource Owner.

**Authorization Server**: Server that authenticates the Resource Owner and issues access tokens after getting proper authorization. In this case, Auth0.

**Grant types**

**OAuth 2.0 defines four flows to get an access token**. These flows are called grant types. Deciding which one is suited for your case depends mostly on your application type.

**Authorization Code Flow**: used by Web Apps executing on a server. This is also used by mobile apps, using the Proof Key for Code Exchange (PKCE) technique.

**Implicit Flow with Form Post**: used by JavaScript-centric apps (Single-Page Applications) executing on the user's browser.

**Resource Owner Password Flow**: used by highly trusted apps.

**Client Credentials Flow**: used for machine-to-machine communication. Authorization between microservices.

The specification also provides an extensibility mechanism for defining additional grant types. To learn more about how each grant type works and when it should be used, see Authentication and Authorization Flows.