### **What is a Spring Framework?**

* Spring is a powerful open source, application framework created to reduce the complexity of enterprise application development.

**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.

* **Spring Core Container –**This layer is basically the core of Spring Framework.It contains the following modules :

1. Spring Core
2. Spring Bean
3. SpEL (Spring Expression Language)
4. Spring Context

* **Data Access/Integration –**This layer provides support to interact with the database. It contains the following modules:

1. JDBC (Java Database Connectivity)
2. ORM (Object Relational Mapping)
3. OXM (Object XML Mappers)
4. JMS (Java Messaging Service)
5. Transaction

* **Web –**This layer provides support to create web application. It contains the following modules:

1. Web
2. Web – MVC
3. Web – Socket
4. Web – Portlet

**Spring IOC Container:** Spring IoC container is the program that injects dependencies into an object and make it ready for our use.

Spring IoC container classes are part of org.springframework.beans and org.springframework.context packages. Spring IoC container provides us different ways to decouple the object dependencies.

BeanFactory is the root interface of Spring IoC container. ApplicationContext is the child interface of BeanFactory interface that provide Spring AOP features, i18n etc.

ApplicationContext

* **FileSystemXmlApplicationContext: It is a type of container which loads the definitions of beans from an XML file. For that, you should be able to provide the full path of the XML bean config file to a constructor.**

ApplicationContext context = new **FileSystemXmlApplicationContext**

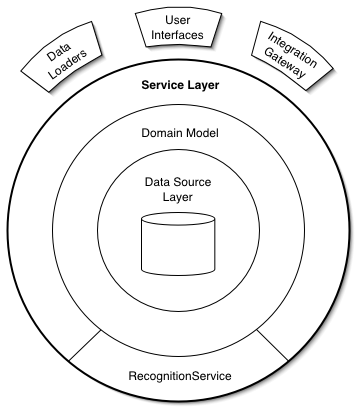
("C:/Users/ADMIN/workspace/HelloSpring/src/Beans.xml");

HelloWorld obj = (HelloWorld) context.**getBean**("helloWorld");

* **ClassPathXmlApplicationContext:** **This type of container loads definitions of the beans from XML file but you don’t need to provide the full path of the XML file. Only the CLASSPATH has to set properly as this container will look like Bean config XML file.**
* **WebXmlApplicationContext:** This type of container loads the XML file with all bean definitions within a web application.

**BeanPostProcessors**: The BeanPostProcessors helps you to do some operations before and after the creation of Spring Bean. The interface BeanPostProcessors operates on the instances of Bean. the methods you studied in Bean lifecycle init() and destroy() are different than the Spring BeanPostProcessors. The BeanPostProcessor in Spring Framework is common to all of the beans.

1. public class InitHelloWorld implements BeanPostProcessor {
2. public Object **postProcessBeforeInitialization**(Object bean, String beanName)
3. throws BeansException {
4. System.out.**println**("BeforeInitialization : " + beanName);
5. return bean; // you can return any other object as well
6. }
7. public Object **postProcessAfterInitialization**(Object bean, String beanName)
8. throws BeansException {
9. System.out.**println**("AfterInitialization : " + beanName);
10. return bean; // you can return any other object as well
11. }
12. }



With the [JDBC abstraction and DAO module](http://examples.javacodegeeks.com/enterprise-java/spring/jdbc/spring-jdbctemplate-example/) we can be sure that we keep up the database code clean and simple, and prevent problems that result from a failure to close database resources. It provides a layer of meaningful exceptions on top of the error messages given by several database servers.

### JdbcTemplate

JdbcTemplate class provides many convenience methods for doing things such as converting database data into primitives or objects, executing prepared and callable statements, and providing custom database error handling.

**Spring BOOT**

### Advantages of Spring Boot:

* It is very easy to develop Spring Based applications with Java or Groovy.
* It reduces lots of development time and increases productivity.
* It avoids writing lots of boilerplate Code, Annotations and XML Configuration.
* It is very easy to integrate Spring Boot Application with its Spring Ecosystem like Spring JDBC, Spring ORM, Spring Data, Spring Security etc.
* It follows “Opinionated Defaults Configuration” Approach to reduce Developer effort
* It provides Embedded HTTP servers like Tomcat, Jetty etc. to develop and test our web applications very easily.
* It provides CLI (Command Line Interface) tool to develop and test Spring Boot(Java or Groovy) Applications from command prompt very easily and quickly.
* It provides lots of plugins to develop and test Spring Boot Applications very easily using Build Tools like Maven and Gradle
* It provides lots of plugins to work with embedded and in-memory Databases very easily.

In Simple terminology:

What Is Spring Boot, Spring Boot Tutorial

**Main Goal of Spring Boot:**

The main goal of Spring Boot Framework is to reduce Development, Unit Test and Integration Test time and to ease the development of Production ready web applications very easily compared to existing Spring Framework, which really takes more time.

To Start Opinionated Approach to create Spring Boot Applications, The Spring Team (The Pivotal Team) has provided the following three approaches.

* Using Spring Boot CLI Tool
* Using Spring STS IDE
* Using Spring Initializer Website

**@NoRepositoryBean**

The annotation is used to avoid creating repository proxies for interfaces that actually match the criteria of a repo interface but are not intended to be one. It's only required once you start going into extending all repositories with functionality. Let me give you an example:

Assume you'd like to add a method foo() to all of your repositories. You would start by adding a repo interface like this

public interface com.foobar.MyBaseInterface<…,…> extends CrudRepository<…,…> {

void foo();

}

You would also add the according implementation class, factory and so on. You concrete repository interfaces would now extend that intermediate interface:

public interface com.foobar.CustomerRepository extends MyBaseInterface<Customer, Long> {

}

Now assume you bootstrap - let's say Spring Data JPA - as follows:

<jpa:repositories base-package="com.foobar" />

You use com.foobar because you have CustomerRepository in the **same package**. The Spring Data infrastructure now has no way to tell that **the MyBaseRepository is not a concrete repository** interface but rather acts as intermediate repo to expose the additional method. So it would try to create a repository proxy instance for it and fail. You can now use @NoRepositoryBean to annotate this intermediate interface to essentially tell Spring Data: don't create a repository proxy bean for this interface.

That scenario is also the reason why CrudRepository and PagingAndSortingRepository carry this annotation as well. If the package scanning picked those up by accident (because you've accidentally configured it this way) the bootstrap would fail.

Long story short: use the annotation to prevent repository interfaces from being picked up as candidates to end up as repository bean instances eventually.

# Global Exception Handling With @ControllerAdvice

@ControllerAdvice is an annotation provided by Spring allowing you to write global code that can be applied to a wide range of controllers — varying from all controllers to a chosen package or even a specific annotation. In this brief tutorial, we will focus on handling exceptions using @ControllerAdvice and @ExceptionHandler (@InitBinder and @ModalAttribute can also be used with @ControllerAdvice).

By default, @ControllerAdvice will apply to all classes that use the @Controller annotation (which extends to classes using @RestController). If you wanted this to be more specific, there are a few properties provided that allow this.

To reduce the applicable classes down by package,

1. you simply need to add the name of the package to the annotation. When a package is chosen, it will be enabled for classes inside that package as well as sub-packages. Multiple packages can also be chosen by following the same process, but using an array instead of a singular string (all properties in @ControllerAdvice can be singular or multiple).

@ControllerAdvice("my.chosen.package")

@ControllerAdvice(value = "my.chosen.package")

@ControllerAdvice(basePackages = "my.chosen.package")

1. Another way to specify a package is via the basePackageClasses property, which will enable @ControllerAdvice to all controllers inside the package that the class (or interface) lives in.

@ControllerAdvice(basePackageClasses = MyClass.class)

1. To apply to specific classes, use assignableTypes.

@ControllerAdvice(assignableTypes = MyController.class)

1. And finally, what if you want to apply it to controllers with certain annotations? The below snippet would only assist controllers annotated with @RestController (which it covers by default) but will not include @Controller annotated classes.

@ControllerAdvice(annotations = RestController.class)

**Important:**

@ExceptionHandler allows you to define a method that, as the name suggests, handles exceptions. If you weren’t using @ControllerAdvice, the code for handling these exceptions would be in the controllers themselves, which could add quite a bit of duplication and clutter to the class and leading to it not being as “clean”. You could move the @ExceptionHandler methods into a base class that the controller extends to separate the code. This method is not perfect and comes with the issue that every controller where you need this global exception handling will now need to extend the base controller. Therefore, when you create a new controller and forget to extend this base class, you are now no longer handling some exceptions and might get bitten in the butt later on. Using @ControllerAdvice along with @ExceptionHandler prevents this by providing global (and more specific) error handling so you don’t need to remember to implement them yourself or extend another class every time.

@ControllerAdvice

@RequestMapping(produces = "application/vnd.error+json")

public class PersonControllerAdvice {

@ExceptionHandler(PersonNotFoundException.class)

public ResponseEntity < VndErrors > notFoundException(final

PersonNotFoundException e) {

return error(e, HttpStatus.NOT\_FOUND, e.getId().toString());

}

private ResponseEntity < VndErrors > error(final Exception exception, final HttpStatus httpStatus, final String logRef) {

final String message = Optional.of(exception.getMessage()).orElse(exception.getClass().getSimpleName());

return new ResponseEntity < > (new VndErrors(logRef, message), httpStatus);

}

@ExceptionHandler(IllegalArgumentException.class)

public ResponseEntity < VndErrors > assertionException(final IllegalArgumentException e) {

return error(e, HttpStatus.NOT\_FOUND, e.getLocalizedMessage());

}

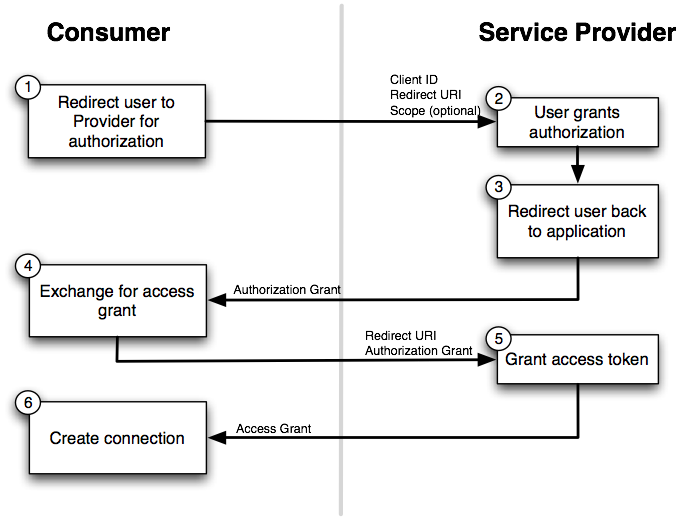
}

If you define more than one @ExceptionHandler for the same exception, you need to be on the lookout. When defined in the same class, Spring is kind enough to throw an exception and fail on startup. But when they appear in different classes, say two @ControllerAdvice classes, both with a handler for the PersonNotFoundException, the application would start — but will use the first handler it finds. This could cause unexpected behavior if you are not aware.

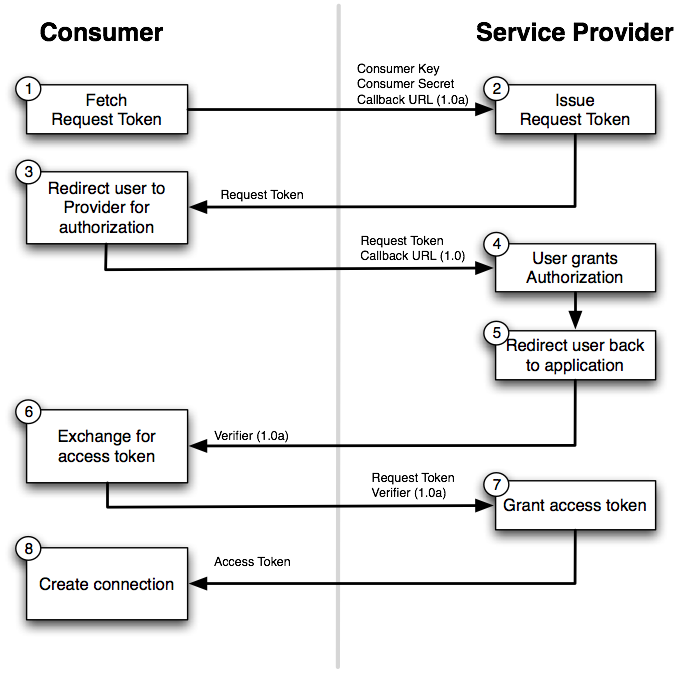
I see great answers up here but what I miss were some diagrams and since I had to work with Spring Framework I came across [their explanation](http://docs.spring.io/spring-social/docs/1.1.0.RELEASE/reference/htmlsingle/#section_oauth2ServiceProviders).

I find the following diagrams very useful. They illustrate the difference in communication between parties with OAuth2 and OAuth1.

## OAuth 2



## OAuth 1



**Transaction management:**

Transaction can be managed in the following ways:

Programmatically manage by writing custom code

Use Spring to manage transaction

Spring supports **two** types of transaction management:

1. **Programmatic transaction management**: This means that you have to manage the transaction with the help of programming. That gives you extreme flexibility, but it is difficult to maintain.
   1. using the TransactionTemplate (Recommended by Spring Team)
   2. using a PlatformTransactionManager implementation directly
2. **Declarative transaction management**: This means you separate transaction management from the business code. You only use annotations or XML based configuration to manage the transactions.

**Choosing between Programmatic and Declarative Transaction Management:**

* Programmatic transaction management is good only if you have a small number of transactional operations. (Most of the times, this is not the case.)
* Transaction name can be explicitly set only using Programmatic transaction management.
* Programmatic transaction management should be used when you want explicit control over managing transactions.
* On the other hand, if your application has numerous transactional operations, declarative transaction management is worthwhile.
* Declarative Transaction management keeps transaction management out of business logic, and is not difficult to configure.

**Declarative Transaction (Usually used almost in all scenarios of any web application)**

@Configuration

@EnableTransactionManagement

**Example of programmatically Transaction management**

public class ServiceImpl implements Service

{

private final TransactionTemplate transactionTemplate;

public ServiceImpl(PlatformTransactionManager transactionManager) {

this.transactionTemplate = new TransactionTemplate(transactionManager);

}

// the transaction settings can be set here explicitly if so desired hence better control. This can also be done in xml file

this.transactionTemplate.setIsolationLevel(TransactionDefinition.ISOLATION\_READ\_UNCOMMITTED);

this.transactionTemplate.setTimeout(30); // 30 seconds

// and so forth...

public Object someServiceMethod()

{

return transactionTemplate.execute(new TransactionCallback()

{

// the code in this method executes in a transactional context

public Object doInTransaction(TransactionStatus status)

{

updateOperation1();

return resultOfUpdateOperation2();

}

});

}}

**Weaving**

Spring AOP does dynamic weaving of aspects by creating proxy of the target objects.

# [What is the difference between DAO and Repository patterns?](https://stackoverflow.com/questions/8550124/what-is-the-difference-between-dao-and-repository-patterns)

DAO is an abstraction of data persistence. Repository is an abstraction of a collection of objects.

DAO would be considered closer to the database, often table-centric. Repository would be considered closer to the Domain, dealing only in Aggregate Roots. A Repository could be implemented using DAO's, but you wouldn't do the opposite.

Also, a Repository is generally a narrower interface. It should be simply a collection of objects, with a Get(id), Find(ISpecification), Add(Entity). A method like Update is appropriate on a DAO, but not a Repository - when using a Repository, changes to entities would usually be tracked by separate UnitOfWork.