**Spring Framework**

**Interview Questions**

**► What is Spring (Framework)?**

Spring is an open source development framework / platform for enterprise Java. It was introduced in **2003** as a response to the complexity of the early J2EE specifications, mostly EJBs. While some consider Java EE and Spring to be in competition, Spring is, in fact, complementary to Java EE. The Spring model does not embrace the Java EE platform specification; rather, it integrates with carefully selected individual specifications from the EE. Spring provides many extensions for building web applications on top of the Java EE platform. Spring promotes programming practice to build applications from "plain old Java objects" (**POJOs**) using **dependency injection**.

**► What are benefits of using Spring?**

Major benefits of using Spring Framework:

* lightweight
* modular
* Inversion of control (IOC)
* beans container and life cycle management
* exception handling
* (declarative) transaction management
* Model View Control (MVC) framework
* Aspect Oriented Programming (AOP)

Spring is lightweight when it comes to size and transparency. The basic version of Spring framework is around 2MB. It allows to use only those modules that you need, without having to bring in the rest. Loose coupling is achieved in Spring using the technique of Inversion of Control (IOC). The objects give their dependencies instead of creating or looking for dependent objects. Spring contains and manages the **life cycle and configuration** of application objects.

Spring provides a consistent **transaction management** interface that can scale down to a local transaction (using a single database, for example) and scale up to global transactions (using JTA, for example). Spring provides a convenient API to translate technology-specific exceptions (thrown by JDBC, Hibernate, or JDO, for example) into consistent, unchecked exceptions.

Spring's **web framework** is a well-designed web **MVC** framework, which provides a great alternative to web frameworks such as Struts or other over engineered or less popular web frameworks. Spring supports **AOP** and enables cohesive development by separating application business logic from system services.

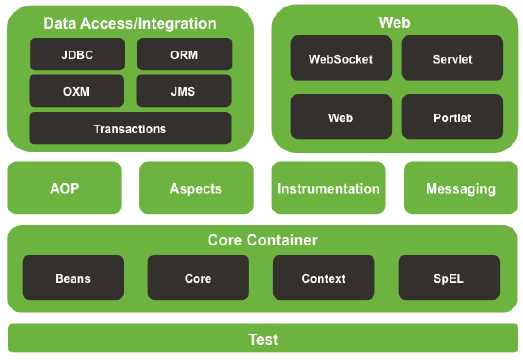
**►► Versions of Spring, which versions you used?**

Releases of the framework - Spring 1 (**2003**), Spring 2 (**2006**), Spring 3 (**2009**), Spring 4 (**2013**) – Java 8 support, WebSocket; Spring 5 (**2017**). **Spring 5.0** requires the **Java 8+** and **Java EE 7** level (e.g. Servlet 3.1+, JPA 2.1+) while also providing integration with newer APIs at the Java EE 8 level (e.g. Servlet 4.0, JSON Binding API) when encountered at runtime. This keeps Spring fully compatible with e.g. Tomcat 8 and 9, WebSphere 9, and JBoss EAP 7.

Used Spring 2; Spring 3.x with Java 7, migrated to Spring 4.1 with Java 8 – XML config, MVC, Hibernate, Java beans config, constructor-based DI, RESTful controllers, scheduling and async methods using @Async annotation, etc.

**► What are the different modules in Spring framework?**

The Spring Framework consists of features organized into about **20** modules.



**Spring 4 modules**

* **Web layer**
  + Web
  + Servlet
  + Portlet
  + Struts (removed in 4.x)
  + WebSockets (added in 4.x)
  + CORS support
* **Authentication and authorization**
* **Messaging (added in 4.x)**
* **Test**

# Inversion of Control / Dependency Injection

**►► Explain Inversion of Control (IoC)?**

Normally, object creates or finds the objects it depends on by itself and then uses them. Inversion of Control (IOC) means that we invert the control of creating objects from within our code to container or framework. Now it is the responsibility of a container to create dependent objects as required. Martin Fowler suggested renaming this principle to **Dependency Injection (DI)** to make it more self-explanatory. With DI an object gets other dependent objects created and injected by container.

**►►► Explain Dependency Injection (DI)?**

Dependency Injection (DI) means that instead of objects creating or locating dependent objects by themselves, dependent objects are created by some container (e.g. Spring IoC container) and injected into the objects that need them. Original concept of IoC was renamed to DI but some say that DI is one concrete example of IoC.

No DI - creation example, bean creates dependencies:

WebController() {

this.config = new Configuration();

}

No DI - location example, bean gets dependencies:

WebController() {

this.config = Configuration.getConfiguration();

}

Constructor DI example, bean has dependency injected:

WebController(Configuration config) {

this.config = config;

}

**►► What are the different types of dependency injection?**

Types of DI in Spring are:

* Constructor based DI
* Setter based DI

**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.

**► Which DI would you suggest: constructor based or setter based?**

Spring recommends constructor DI for mandatory dependencies and setters - for optional dependencies. Note, that the use of a @Required annotation on a setter can be used to make setters required dependencies.

**►► What are the benefits of DI / IoC?**

The main benefits of DI / IoC are:

* Loose coupling is promoted with minimal effort and least intrusive mechanism.
* IoC containers support eager instantiation and lazy loading of services.
* It minimizes the amount of code in your application.
* It makes your application easier to test as it doesn't require any singletons or JNDI lookup in unit test cases.

**► What is Spring IoC container?**

The Spring IoC creates objects called "beans", wires them together, configures them, and manages their complete lifecycle from creation untill destruction. The Spring container uses DI to manage the beans that make up an application.

**► What are types of IoC containers?**

There are two types of IoC containers:

* Bean Factory container
* ApplicationContext container

**► What is the difference between Bean Factory and ApplicationContext?**

The BeanFactory provides the underlying basis for Spring’s IoC functionality but it is only used directly in integration with other 3rd frameworks and is now largely historical in nature for most users of Spring. It could be preferred where the resources are limited like mobile devices or applet based applications. Spring docs says: "Use an ApplicationContext unless you have a good reason for not doing so." ApplicationContext container adds more enterprise-specific functionality:

* Application contexts provide a means for resolving text messages, including support for i18n of those messages.
* Application contexts provide a generic way to load file resources, such as images.
* Application contexts can publish events to beans that are registered as listeners.
* Certain operations on the container or beans in the container, which have to be handled in a programmatic fashion with a bean factory, can be handled declaratively in an application context.
* The application context implements MessageSource, an interface used to obtain localized messages, with the actual implementation being pluggable.

**► Give an example of BeanFactory implementation.**

The most commonly used BeanFactory implementation is the XmlBeanFactory class. This container reads the configuration metadata from an XML file and uses it to create a fully configured system or application.

**► What are the common implementations of the XML-based ApplicationContext?**

The three commonly used implementation of ApplicationContext are:

* FileSystemXmlApplicationContext
* ClassPathXmlApplicationContext
* WebXmlApplicationContext

FileSystemXmlApplicationContext and ClassPathXmlApplicationContext load the definitions of the beans from an XML file. ClassPathXmlApplicationContext need to have CLASSPATH set properly. WebXmlApplicationContext loads the XML file with definitions of all beans from within a web application.

**► How do you provide configuration metadata? What is Spring configuration?**

The container gets its instructions on what objects to instantiate, configure, and assemble by reading configuration metadata. The configuration metadata could be represented in

* XML based configuration file
* Java annotations (starting with Spring 2.5)
* Java code (starting with Spring 3.0).

**► What are Spring 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.

**► What does a bean definition contain?**

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

**► What is bean wiring?**

Bean wiring is when beans are combined together within the Spring container. When wiring beans, the Spring container needs to know what beans are needed and how the container should use dependency injection to tie them together.

**► Explain the Spring bean lifecycle.**

Spring framework is based on IoC/DI, the beans reside and are managed by the IoC container. Spring beans are just POJOs.

* The container will look the bean definition inside configuration file (e.g. bean.xml).
* By using reflection container will create the object and if any property is defined inside the bean definition then it will also be set.
* If the bean implements the BeanNameAware interface, the factory calls setBeanName() passing the bean’s ID.
* If the bean implements the BeanFactoryAware interface, the factory calls setBeanFactory(), passing an instance of itself.
* If there are any BeanPostProcessors associated with the bean, their post-ProcessBeforeInitialization() methods will be called before the properties for the Bean are set.
* If an init() method is specified for the bean, it will be called.
* If the Bean class implements the DisposableBean interface, then the method destroy() will be called when the Application no longer needs the bean reference.
* If the Bean definition in the Configuration file contains a 'destroy-method' attribute, then the corresponding method definition in the Bean class will be called.

**►►► What are bean scopes that Spring support?**

The Spring Framework supports the following scopes (note, that there are **more than 5** that they usually expect in interviews based on Google):

* singleton
* prototype
* thread (available as of 3.0 but is not registered by default)
* Scopes for **web**-aware application contexts:
* request
* session
* global-session (used by portlets)
* application (added in 4.x)
* websocket (added in 4.x)

**► What is singleton scope?**

Only one shared instance of a singleton bean is managed by IoC container, and all requests for beans with an id or ids matching that bean definition result in that one specific bean instance being returned by container. This single instance is stored in a cache of singleton beans, and all subsequent requests and references for that named bean return the cached object.

**►► Is Spring singleton the same as GoF singleton?**

No, Spring’s concept of a singleton bean differs from the Singleton pattern as defined in the GoF patterns book. The GoF Singleton hard-codes the scope of an object such that one and only one instance of a particular class is created per ClassLoader.

The scope of the Spring singleton is best described as "per container and per bean". This means that if you define one bean for a particular class in a single Spring container, then the Spring container creates one and only one instance of the class defined by that bean definition. But we can also create several Spring singleton beans of the same class but with different names and properties.

**► Are singleton beans thread safe?**

Spring framework does not do anything under the hood concerning the multithreaded behavior of a singleton bean. It is the developer's responsibility to deal with concurrency issue and thread safety of the singleton bean. Based on Google searches, they might expect: "No, singleton beans are not thread-safe in Spring framework."

**► What is default scope of Spring bean?**

The default scope of Spring bean is "singleton".

**► What is prototype scope?**

Prototype scope results in the creation of a new bean instance every time a request for that specific bean is made.

**► What are Web application scopes?**

**Request** scopes a bean to a HTTP request, **session** - to HTTP session, **global-session** - to a global HTTP session for portlet-based web applications, **application** – to a ServletContext, **websocket** – to a websocket. All these scopes are only valid in the context of a web-aware Spring ApplicationContext.

**►► What is application scope?**

Application scope is similar to a singleton but differs in two important ways:

* It is a singleton per **ServletContext**, not per ApplicationContext (which may be several in any web app),
* it is exposed and visible as a ServletContext attribute.

**► What are inner beans in Spring?**

A <bean/> element inside the <property/> or <constructor-arg/> elements defines a so-called inner bean. An inner bean definition does not require a defined id or name; the container ignores these values. It also ignores the scope flag. Inner beans are always anonymous and they are always scoped as prototypes.

**► Can you inject null and empty string values in Spring?**

Yes.

**► How can you inject Java Collection in Spring?**

Spring offers four types of collection configuration elements which are as follows:

* <list>: This helps in wiring i.e. injecting a list of values, allowing duplicates.
* <set>: This helps in wiring a set of values but without any duplicates.
* <map>: This can be used to inject a collection of name-value pairs where name and value can be of any type.
* <props>: This can be used to inject a collection of name-value pairs where the name and value are both Strings.

**►► Can two beans have the same bean id in an Application Context / Dispatcher Servlet Context hierarchy?**

Any given Spring context can only have one bean for any given id or name. In the case of the XML id attribute, this is enforced by the schema validation. In the case of the name attribute, this is enforced by Spring's logic. If a context is constructed from two different XML descriptor files, and an id is used by both files, then one will "override" the other. The exact behaviour depends on the ordering of the files when they get loaded by the context.

In the Web MVC framework, each DispatcherServlet has its own WebApplicationContext, which **inherits** all the beans already defined in the root WebApplicationContext. These inherited global scope beans can be **overridden** in the servlet-specific scope by beans defined with the same name.

# Annotations / Java Based Configuration

**►►► What is Spring Java / Annotation based container configuration? Examples of annotations?**

Java annotation-based configuration option enables you to write most of your Spring configuration **without XML** but with the help of Spring **annotations**. The central artifacts in Java configuration are **@Configuration** annotated classes and **@Bean** annotated methods. Instead of using XML to describe a bean wiring, the developer moves the configuration into the component class itself by using annotations on the relevant class, method, or field declaration. Some examples of Spring annotations are:

* **@Configuration**
* **@EnableWebMvc**
* @ComponentScan
* @EnableScheduling
* @Import
* **@Bean**
* **@Autowired**
* @Resource
* @Component
* **@Controller**
* **@RestController**
* **@RequestMapping**
* @Repository

**► Are annotations or Java code better than XML for configuring Spring?**

It depends, each approach has its pros and cons, and usually it is up to the developer to decide which strategy suits them better. XML excels at wiring up components without touching their source code or recompiling them. Some developers prefer having the wiring close to the source while others argue that annotated classes are no longer POJOs and, furthermore, that the configuration becomes decentralized and harder to control. No matter the choice, Spring can accommodate all styles and even mix them together.

**► Can we combine Java and XML configuration? How do you turn on annotation wiring?**

Spring’s @Configuration class support does not aim to be a 100% complete replacement for Spring XML. Some facilities such as Spring XML namespaces remain an ideal way to configure the container. In cases where XML is convenient or necessary, you have a choice:

* XML-centric way
* Java-centric fashion

With **XML-centric** way instantiate the container using for example, ClassPathXmlApplicationContext. Annotation wiring is not turned on in the Spring container by default, enable it in XML configuration using <context:annotation-config/>.

In a **Java-centric** fashion use AnnotationConfigApplicationContext and the @ImportResource annotation to import XML as needed.

When XML and Java configurations are combined, annotation injection is performed before XML injection, thus the XML configuration will override Java beans.

**► Explain @Configuration.**

Annotation @Configuration indicates that the class can be used by the Spring IoC container as a source of bean definitions. E.g.

@Configuration

@EnableWebMvc

@ComponentScan( basePackages="hit.integrad.web.qrs")

@EnableScheduling

@Import( { RcmManagerSpringConfig.class} )

public class WebApplicationConfig {

**► Explain @Bean.**

The @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.

@Bean(name = SYSTEM\_CONFIG\_DEFAULT)

@Scope("singleton")

SystemConfigurationService **defaultSystemConfiguration**() {

SystemConfigurationService defaultSystemConfiguration =

new SystemConfiguration( defaultLogConfig());

}

**► Are the JSR-250 Annotations supported?**

Spring supports several JSR-250 - Common Java EE and Java SE - annotations:

* @PostConstruct: This annotation can be used as an alternate of initialization callback.
* @PreDestroy: This annotation can be used as an alternate of destruction callback.
* @Resource : This annotation can be used on fields or setter methods. The @Resource annotation takes a 'name' attribute which will be interpreted as the bean name to be injected. You can say, it follows by-name autowiring semantics.

**►► What is Stereotype [annotation]?**

Spring has org.springframework.**stereotype** package with annotations denoting the roles of types or methods in the overall architecture (at a **conceptual**, rather than implementation, level). Stereotype annotation provides hints for people reading the code. Always use these annotations over **concrete classes**, not over interfaces.

* @Component - base for others
* @Controller, @RestController
* @Repository
* @Service

**► What is @Component?**

@Component is a generic stereotype for any Spring-managed component. @Repository, @Service, and @Controller are specializations of @Component for more specific use cases, for example, in the persistence, service, and presentation layers, respectively. Therefore, you can annotate your component classes with @Component, but by annotating them with @Repository, @Service, or @Controller instead, your classes are more properly suited for processing by tools or associating with Aspects.

The @Component annotation marks a Java class as a bean so the component-scanning mechanism of Spring can pick it up and pull it into the application context. **@Named** is Java standard equivalent to the @Component.

**► What is @Controller?**

@Controller is specialization of @Component indicating that annotated class is a Spring Web MVC Controller. Beans marked with it are automatically imported into the DI container. When you add the @Controller annotation to a class, you can use annotated handler methods based on the @RequestMapping to map URLs to instance methods of a class.

**► What is @Service?**

@Service is specialization of @Component which indicates that an annotated class is a "Service", e.g. a business service facade, contains business logic. It doesn’t currently provide any additional behavior over the @Component annotation, but it’s a good idea to use @Service over @Component because it specifies intent better. Additionally, tool support and additional behavior might change in the future.

@Service("tradeService")

public class TradeServiceImpl implements TradeService {

**►► What is @Repository?**

The @Repository annotation is a marker for any persistence class that fulfills the role (stereotype) of a repository (also known as Data Access Object or DAO). Among the uses of this marker is importing the DAOs into the DI container and automatic translation of unchecked exceptions (thrown from DAO methods) into Spring DataAccessException.

@Repository ("employeeDao")

public class EmployeeDAOImpl implements EmployeeDAO {

**► What is the difference between @Component, @Repository, @Service? When would you use them?**

@Component is a base / generic stereotype for any Spring-managed component. @Repository, @Service, and @Controller are specializations of @Component for more specific use cases. Therefore, it is preferred to use these specific annotations as properly suited for processing by tools or associating with Aspects.For example, these stereotype annotations make ideal targets for **pointcuts**.

@Service doesn’t currently provide any additional behavior over the @Component annotation, but it’s a good idea to use @Service over @Component because it specifies intent better and tool support and behavior might change in the future.

@Repository is already supported as a marker for automatic exception translation in your **persistence** layer.

# Autowiring

**► What is [bean] autowiring? What is @Autowired annotation?**

Autowiring means automatic injection of an instance of one bean into the desired field in an instance of another bean. Both classes should be beans, i.e. they should be defined to live in the Spring application context. The Spring **@Autowired** annotation can be used to autowire bean on:

* a property (to get rid of the setter methods)
* constructor,
* setter method (just like @Required annotation),
* method with arbitrary names and/or multiple arguments.

Beans can now be ordered when they are autowired into lists and arrays. Both the **@Order** annotation and Ordered interface are supported. Spring treats **generic types** as a form of qualifier, so we can now inject a specific implementation:

@Autowired Repository<Customer> customerRepository;

**► @Autowired vs @Inject?**

[Java 8] Dependency Injection (JSR-330) annotation @javax.inject.**Inject** can be used in place of Spring’s @Autowired annotation.

**► What does @Qualifier annotation mean?**

Because autowiring **by type** may lead to multiple candidates, it is often necessary to have more control over the selection process. One way to accomplish this is with Spring’s **@Qualifier** annotation. You can associate qualifier values with specific arguments, narrowing the set of type matches so that proper bean is chosen for each argument:

@Autowired

@Qualifier("mainCatalog")

private MovieCatalog movieCatalog;

**► What are different modes of autowiring?**

The autowiring functionality has five (?) modes which can be used to instruct Spring container to use autowiring for dependency injection:

* no (default)
* byName
* byType
* constructor
* autodetect (not in Spring 3.2 - 4.1, probably was available before)

"**No**" is default setting which means no autowiring, and you should use explicit bean reference for wiring. To autowire by property name "**byName**" Spring container tries to match and wire its properties with the beans defined by the same name in the configuration file.

To autowire by property datatype "**byType**" Spring tries to match and wire a property if its type matches with exactly one of the beans name in configuration file. If more than one such beans exist, a fatal exception is thrown. Autowire by "**constructor**" is similar to byType, but type applies to constructor arguments. If there is not exactly one bean of the constructor argument type in the container, a fatal error is raised. With "**autodetect**" (not in Spring 3.2 - 4.1) Spring first tries to wire using autowire by constructor, if it does not work, Spring tries to autowire by byType.

**► What does @Required annotation mean?**

The @Required annotation applies to bean property setter methods and indicates that the affected bean property must be populated at configuration time, through an explicit property value in a bean definition or through autowiring. The container throws BeanInitializationException if the affected bean property has not been populated.

**► What are the issues / limitations with autowiring?**

Limitations of autowiring are:

* Overriding possibility: You can still specify dependencies using <constructor-arg> and <property> settings which will always override autowiring.
* Primitive data types: You cannot autowire so-called simple properties such as primitives, Strings, and Classes.
* Confusing nature: Autowiring is less exact than explicit wiring, so if possible prefer using explicit wiring.

# Event Handling

**► How is event handling done in Spring?**

Event handling in the ApplicationContext is provided through the ApplicationEvent class and ApplicationListener interface. So if a bean implements the ApplicationListener, then every time an ApplicationEvent gets published to the ApplicationContext, that bean is notified.

**► Describe some of the standard Spring events.**

Spring provides the following standard events:

* ContextRefreshedEvent: This event is published when the ApplicationContext is either initialized or refreshed. This can also be raised using the refresh() method on the ConfigurableApplicationContext interface.
* ContextStartedEvent: This event is published when the ApplicationContext is started using the start() method on the ConfigurableApplicationContext interface. You can poll your database or you can re/start any stopped application after receiving this event.
* ContextStoppedEvent: This event is published when the ApplicationContext is stopped using the stop() method on the ConfigurableApplicationContext interface. You can do required housekeep work after receiving this event.
* ContextClosedEvent: This event is published when the ApplicationContext is closed using the close() method on the ConfigurableApplicationContext interface. A closed context reaches its end of life; it cannot be refreshed or restarted.
* RequestHandledEvent: This is a web-specific event telling all beans that an HTTP request has been serviced.

# Data Access / Services

**►► Explain the JDBC Abstraction and DAO module.**

The Data Access Object (DAO) support in Spring allows to use different data access technologies in a consistent way, so that we can switch between persistence technologies easily and to code without worrying about catching exceptions that are specific to each technology.

Spring provides a convenient translation from technology-specific exceptions like **SQLException** or **Hibernate** exceptions to its own exception class hierarchy with the **DataAccessException** as the root. The best way to guarantee that your DAOs or repositories provide exception translation is to use the **@Repository** annotation.

The **JDBC module** takes care of all the low-level details that can make JDBC such a tedious API to develop with - open/close connection, prepare and execute statements, handle transactions and exceptions. Spring’s **AOP module** could be used to have **declarative transaction management**.

**►►► What are DB / data access technologies supported by Spring? Which one you used?**

The DAO module in Spring is aimed at making it easy to work with different data access technologies in a consistent way without worrying about catching exceptions that are specific to each technology:

* JDBC,
* Hibernate,
* JPA (Java Persistence API)
* JDO (Java Data Objects)

**► How JDBC can be used in Spring framework? What is JdbcTemplate?**

JDBC can be used with the help of a Spring template class **JdbcTemplate**. This class provides many convenience methods such as converting database data into primitives or objects, executing prepared and callable statements, and providing custom database error handling.

**►► What are the types of the transaction management Spring supports?**

Spring supports two types of transaction management:

* Programmatic
* Declarative

Programmatic transaction management means that you manage the transactions in your code. That gives you extreme flexibility, but it is difficult to maintain. Declarative transaction management means you separate transaction management from the business code. You only use **annotations** or XML based configuration to manage the transactions.

**►► Which of transaction management types is preferable?**

Declarative transaction management is preferable over programmatic transaction management though it is less flexible than programmatic transaction management, which allows you to control transactions through your code.

## Object Relational Mapping (ORM)

**► Explain the Object Relational Mapping (ORM) integration module, benefits?**

Spring supports an Object/Relational Mapping (ORM) over straight JDBC by providing the ORM module.

Spring supports integration with several popular ORM frameworks for resource management, data access object (DAO) implementations, and transaction strategies.

**Benefits** of using the Spring Framework to create your ORM DAOs include:

* Easier testing
* Common data access exceptions
* General resource management
* Integrated transaction management

Spring’s IoC approach makes it easy to swap the implementations and configuration locations of Hibernate SessionFactory instances, JDBC DataSource instances, transaction managers, and mapped object implementations (if needed). This in turn makes it much easier to test each piece of persistence-related code in isolation.

Spring can wrap exceptions from your ORM tool, converting them from proprietary (potentially checked) exceptions to a common runtime DataAccessException hierarchy.

Spring application contexts can handle the location and configuration of Hibernate SessionFactory instances, JPA EntityManagerFactory instances, JDBC DataSource instances, and other related resources.

You can wrap your ORM code with a declarative, aspect-oriented programming (AOP) style method interceptor either through the @Transactional annotation or by explicitly configuring the transaction AOP advice in an XML configuration file. In both cases, transaction semantics and exception handling (rollback, and so on) are handled for you.

**► What are the ORM's Spring supports?**

Spring supports the following ORM's :

* Hibernate
* JPA (Java Persistence API)
* JDO (Java Data Objects)
* iBatis
* TopLink
* OJB

## Hibernate

**►►► What are ways to integrate Hibernate with Spring?**

There are two ways to access Hibernate using Spring:

* DI with a **Hibernate Template** and **Callback**.
* Extending **HibernateDAOSupport** and applying an **AOP Interceptor** node.

Define resources in the Spring container - JDBC DataSource and Hibernate SessionFactory using it.

<bean id="mySessionFactory" class= "org.springframework.orm.hibernate5.LocalSessionFactoryBean">

<property name="**dataSource**" ref="myDataSource"/>

<property name="mappingResources">

<list>

<value>product.hbm.xml</value>

</list>

</property>

<property name="hibernateProperties">

<value>hibernate.dialect=org.hibernate.dialect.HSQLDialect</value>

</property>

</bean>

**►►► What is HibernateTemplate?**

When Spring and Hibernate integration started, Spring ORM provided two helper classes – **HibernateDaoSupport** and **HibernateTemplate** to get the Session from Hibernate and get the benefit of Spring transaction management and exception translation.

However from Hibernate 3.0.1, we can use SessionFactory getCurrentSession() method to get the current session with Spring transaction management benefits. Exception translation can be achieved by using **@Repository** annotation with service classes. So now we **should not use** these classes anymore.

**► How to use plain Hibernate API?**

**Data access (DAO)** could be implemented based on **plain Hibernate API**, using contextual sessions, wherein Hibernate itself manages one current Session per transaction. This is roughly equivalent to Spring’s synchronization of one Hibernate Session per transaction. The main advantage of this DAO style is that it depends on Hibernate API only; no import of any Spring class required, and it is more natural to Hibernate developers.

public class ProductDaoImpl implements ProductDao {

private SessionFactory sessionFactory; //-- use DI

public Collection loadProductsByCategory(String category) {

return this.sessionFactory.getCurrentSession()

.createQuery("from test.Product product where product.category=?")

.setParameter(0, category)

.list();

}

}

The DAO could throw HibernateException (which is unchecked), which means that callers depend / should handle Hibernate’s exception hierarchy.

**►► How to configure transaction manager?**

Spring recommends to use Spring’s declarative transaction support or programmatic transaction demarcation by using wrapper org.**springframework**.orm.**hibernate5**.**HibernateTransactionManager**.Set up in the container as a bean, add "<tx:annotation-driven/>" entry, opting into **@Transactional** processing for annotated methods.

<bean id="transactionManager"

class="**org.springframework.orm.hibernate5.HibernateTransactionManager**">

<property name="sessionFactory" ref="**sessionFactory**"/>

</bean>

**<tx:annotation-driven/>**

<bean id="myProductService" class="product.SimpleProductService">

<property name="productDao" ref="myProductDao"/>

</bean>

Annotate service layer methods with **@Transactional** annotation:

public class ProductServiceImpl implements ProductService {

private ProductDao productDao; //-- use DI to inject

**@Transactional(readOnly = false)**

public void increasePriceOfAllProductsInCategory(final String category) {

List productsToChange = this.productDao.loadProductsByCategory(category);

// ...

}

}

# Web Layer / MVC

**► Explain the web module (layer).**

Spring Web "layer" contains the following modules:

* Web module
* Servlet
* Portlet
* Struts (module removed in 4.x while Struts still supported)
* **WebSockets** (added in 4.x)
* **CORS** support

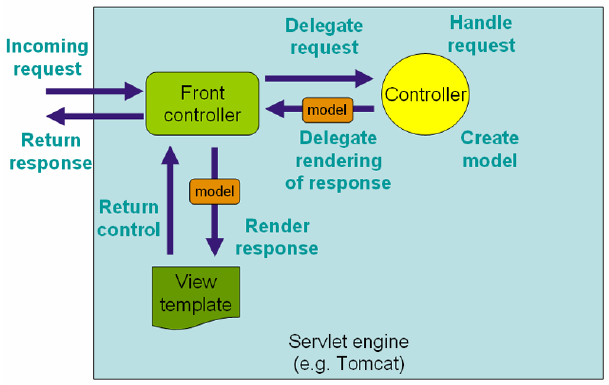
The Spring web module is built on the application context module, providing a context that is appropriate for web-based applications. This module also contains support for several web-oriented tasks such as transparently handling multipart requests for file uploads and programmatic binding of request parameters to your business objects. Spring also provides own Model-View-Controller (MVC) framework.

**►►► What is Spring MVC framework?**

The Spring **Web Model-View-Controller (MVC)** framework provides architecture and components that can be used to develop flexible and loosely coupled web applications. The MVC pattern allows to separate input logic, business logic, and UI logic. It is designed around a DispatcherServlet. With Spring 3.0, the **@Controller** mechanism also allows to create RESTful Web appls, through the @PathVariable annotation and other features.

**►► What is a DispatcherServlet?**

The Spring Web MVC framework is designed around a **DispatcherServlet** (an actual **Servlet,** inherits from the HttpServlet base) that dispatches HTTP requests to handlers (**controllers**), with configurable mappings, view resolution, locale, time zone and theme resolution as well as support for uploading files. The default handler is based on the **@Controller** and **@RequestMapping** annotations.



The request processing workflow of the Spring Web MVC DispatcherServlet implements "**Front Controller**" design pattern. In the Web MVC framework, each DispatcherServlet has its own **WebApplicationContext,**.which inherits and could override the beans already defined in the root WebApplicationContext, including middle-tier services, datasources, etc.

**► What is WebApplicationContext?**

The WebApplicationContext is an extension of the plain ApplicationContext that has some extra features necessary for web applications. It differs from a normal ApplicationContext in that it is capable of resolving themes, and that it knows which servlet it is associated with.

**► Compare Spring MVC with Struts MVC?**

Spring's MVC is very versatile and flexible based on interfaces but Struts forces Actions and Form object into concrete inheritance. Spring provides both interceptors and controllers, thus helps to factor out common behavior to the handling of many requests.

In Spring MVC Controllers can be configured using DI (IoC) that makes its testing and integration easy. Struts force your Controllers to extend a Struts class but Spring doesn't, there are many convenience Controller implementations that you can choose to extend.

Spring can be configured with different view technologies like Freemarker, JSP, Tiles, Velocity, XLST etc. and also you can create your own custom view mechanism by implementing Spring View interface. In Struts, Actions are coupled to the view by defining ActionForwards within a ActionMapping or globally. SpringMVC has HandlerMapping interface to support this functionality.

With Struts, validation is usually performed (implemented) in the validate method of an ActionForm. In SpringMVC, validators are business objects that are NOT dependent on the Servlet API which makes these validators to be reused in your business logic before persisting a domain object to a database.

**► What is @Controller in Spring MVC framework?**

Controllers provide access to the application behavior that you typically define through a service interface. Controllers interpret user input and transform it into a model that is represented to the user by the view. Spring implements a controller in a very abstract way, which enables you to create a wide variety of controllers. The **@Controller** annotation indicates that a particular class serves the role of a controller. Spring does not require you to extend any controller base class or reference the Servlet API.

**►► Explain @RequestMapping annotation.**

@RequestMapping annotation is used to map a URL to either an entire class or a particular handler method. E.g.

**@Controller**

**@RequestMapping("/appointments")**

**public class StudyManagerController {**

**@RequestMapping** (

value = "/\_move/{day}",

method = RequestMethod.POST,

produces = MediaType.APPLICATION\_JSON\_UTF8\_VALUE

)

**public String moveAppointments** (

@PathVariable @DateTimeFormat(iso=ISO.DATE) Date day ) {

. . .

return "moveStatus"; //-- logical View name

}

**► What is ViewResolver?**

ViewResolver resolves / maps logical view name (string) to the actual View type (class).

**►►► How to handle exceptions? Which annotations are used?**

The DefaultHandlerExceptionResolver translates Spring MVC exceptions to specific error status codes. It is registered by default with the MVC. If customization is needed, add exception handling method(s) with **@ExceptionHandler** annotation. (@ResponseBody is no longer needed.) ResponseEntity is HttpEntity that adds HTTP Status code.

**@ExceptionHandler**(IllegalArgumentException.class)

public **ResponseEntity<String>** handleException(IllegalArgumentException iae) {

String jsonOutcome = "{\"error\": 9234}";

HttpStatus httpStatus = HttpStatus.BAD\_REQUEST;

ResponseEntity<String> response = new ResponseEntity<>( jsonOutcome, httpStatus);

return response;

}

A business exception can be annotated with **@ResponseStatus**. When the exception is raised, the ResponseStatusExceptionResolver handles it by setting the status of the response accordingly.

## CORS

**►► What is CORS?**

For security reasons, browsers prohibit AJAX calls to resources residing outside the current origin. E.g., as you’re checking your bank account in one tab, the scripts from another tab should not be able to make AJAX requests to your bank API using your credentials.

**Cross-origin resource sharing (CORS)** is a W3C specification implemented by most browsers that allows you to specify in a flexible way what kind of **cross domain requests are authorized**, instead of using some less secured and less powerful hacks like IFRAME or **JSONP**.

**►► How Spring supports CORS?**

As of Spring 4.2, CORS is supported out of the box. CORS requests (including **preflight** ones with an OPTIONS method) are automatically dispatched to the various registered HandlerMappings. They handle CORS preflight requests and intercept CORS simple and actual requests thanks to a CorsProcessor implementation (DefaultCorsProcessor by default) to add the relevant CORS response headers (like **Access-Control-Allow-Origin**) based on the CORS configuration you have provided.

## REST

**►►► How to create RESTful services in Spring? Which annotations are used?**

One approach to implement REST is to use standard Spring MVC **@Controller** and manually implementing REST API by serving back JSON, XML or custom MediaType content. In this case **@RequestMapping** methods with need to be annotated with **@ResponseBody.**

Instead, we can use **@RestController** annotation that combines @Controller and @ResponseBody. We will also use **@RequestMapping** and **@RequestParam**. E.g.:

**@RestController**

public class QueryRetrieveRestController {

**@RequestMapping (**

**value = "/studies",**

**method = RequestMethod.GET,**

**produces =** MediaType.APPLICATION\_JSON\_UTF8\_VALUE

**)**

public String findStudies (

**@RequestParam** Map<String,String> searchParams

// specific param: @RequestParam(value="patient") String patient

)

**► HTTP REST methods supported by Spring?**

Spring 4 class RequestMethod enumerates:

GET, HEAD, POST, PUT, PATCH, DELETE, OPTIONS, TRACE

**► How to enable CORS for specific REST method?**

You can add an **@CrossOrigin** annotation to @RequestMapping annotated handler method to enable CORS on it. By default, @CrossOrigin allows all origins and the HTTP methods specified in the @RequestMapping annotation:

@RestController

@RequestMapping("/account")

public class AccountController {

**@CrossOrigin**

@RequestMapping("/{id}")

public Account retrieve(@PathVariable Long id) {

// ...

# Integration

**►►► What is included in Spring Integration? Which integrations did you use?**

The Spring Framework’s integration allows to integrate with several Java EE and related technologies:

|  |  |
| --- | --- |
| **Task Execution and Scheduling** | Abstractions for asynchronous execution and scheduling of tasks with the TaskExecutor and TaskScheduler, **thread pools**, scheduling using Timer and the **Quartz** Scheduler, asynchronous execution. |
| **Cache Abstraction** | Allows consistent use of various caching solutions (e.g. **Ehcache**, **JCache** annotations) with minimal impact on the code. |
| **Remoting and web services** | Exposing POJOs as RMI or web services – RMI, HTTP, Hessian, Burlap, JAX-WS, JMS, AMQP. |
| **EJB integration** | Spring EJB wrappers, accessors, interceptors. |
| **Java Message Service (JMS)** | Simplifies the use of the JMS API by using **JMS template** classes. |
| **JMX** | Register any Spring bean as JMX bean. |
| **JCA** | Support for Java Connector Architecture (JCA) |
| **Email** | Utility library for sending email (**MailSender**) that shields the user from the specifics of the underlying mailing system and is responsible for low level resource handling on behalf of the client. |
| **Dynamic language support** | Support for JRuby, Groovy, BeanShell. |

## Task Execution and Scheduling

**►► What are task execution and scheduling options in Spring?**

The Spring provides abstractions for asynchronous execution and scheduling of tasks with the TaskExecutor and TaskScheduler interfaces, respectively. Spring also features implementations of those interfaces that support thread pools or delegation to CommonJ within an application server environment.

Spring also features integration classes for supporting scheduling with the Timer, part of the JDK since 1.3, and the Quartz Scheduler. Both of those schedulers are set up using a FactoryBean with optional references to Timer or Trigger instances, respectively. Furthermore, a convenience class for both the Quartz Scheduler and the Timer is available that allows you to invoke a method of an existing target object (analogous to the normal MethodInvokingFactoryBean operation). Spring provides annotation support for both task scheduling and asynchronous method execution.

**►► Can we use task scheduling or asynchronous method execution out-of-box?**

No, to enable support for @Scheduled and @Async annotations add @EnableScheduling and @EnableAsync to one of your @Configuration classes:

@Configuration

@EnableAsync

@EnableScheduling

public class AppConfig { ...

**►► How to schedule a method or task?**

The @Scheduled annotation can be added to a method along with trigger metadata. For fixed-delay and fixed-rate tasks, an initial delay may be specified indicating the number of milliseconds to wait before the first execution of the method. If simple periodic scheduling is not expressive enough, then a "**cron**" expression may be provided.

@Scheduled(initialDelay=1000, fixedRate=5000)

public void doSomething() {

...

}

Another option – create **ScheduledExecutorTask** and add it to **ScheduledExecutorFactoryBean**:

@Bean

@Scope("singleton")

ScheduledExecutorTask scheduledSessionsPolicyTask() {

SessionsPolicyTimer sessionsPolicyTimer = sessionsPolicyTimer();

ScheduledExecutorTask scheduledTask = new ScheduledExecutorTask();

scheduledTask.setDelay( 30 \* 1000L ); //-- in milliseconds

scheduledTask.setPeriod( sessionServiceConfig().getExpirePeriod() \* 1000L );

scheduledTask.setRunnable( sessionsPolicyTimer);

return scheduledTask;

}

**► How to execute a method asynchronously?**

The @Async annotation can be provided on a method so that invocation of that method will occur asynchronously. In other words, the caller will return immediately upon invocation and the actual execution of the method will occur in a task that has been submitted to a Spring TaskExecutor. Unlike the methods annotated with the @Scheduled annotation, these methods can expect arguments, because they will be invoked in the "normal" way by callers at runtime rather than from a scheduled task being managed by the container. For example,

@Async

public void preloadMyResources( ResourcePool resourcePool) {

...

}

# Aspect-Oriented Programming (AOP)

**► What is AOP? What is Aspect?**

Aspect-Oriented Programming (AOP) is a programming technique that allows programmers to modularize **crosscutting concerns**, or behavior that cuts across the typical divisions of responsibility, such as logging and transaction management.

The core construct of AOP is the **Aspect**, which encapsulates behaviors affecting multiple classes into reusable modules, it is a module which has a set of APIs providing cross-cutting requirements. For example, a logging module would be called AOP aspect for logging. An application can have any number of aspects depending on the requirements.

**► What is AspectJ?**

AspectJ is an AOP extension for the Java created at Xerox in 2001. It is available in Eclipse Foundation open-source projects, both stand-alone and integrated into Eclipse. AspectJ has become a widely used de facto standard for AOP by emphasizing simplicity and usability for end users.

**► How is AOP supported by Spring?**

Spring has 2 modules supporting AOP:

* The "AOP" (**spring-aop**) module, AOP Alliance - compliant,
* The "Aspects" (**spring-aspects**) module, Spring integration with AspectJ.

The Spring AOP module provides an AOP Alliance - compliant AOP implementation to define, for example, method interceptors and pointcuts to cleanly decouple code that implements functionality that should be separated. Using source-level metadata functionality, you can also incorporate behavioral information into your code, in a manner similar to that of .NET attributes. The Spring team decided not to introduce new AOP-related terminology; therefore, in the Spring reference documentation and API, terms such as aspect, join point, advice, etc. have the same meanings as in most other AOP frameworks, particularly AspectJ.

The separate "Aspects" (spring-aspects) module provides Spring integration with **AspectJ** using regular classes annotated with the AspectJ style @Aspect annotation .

**► What is XML Schema based aspect implementation?**

Aspects are implemented using regular classes along with XML based configuration.

**► What is the difference between concern and cross-cutting concern in Spring?**

* **Concern** is behavior which we want to have in a module of an application. Concern may be defined as a functionality we want to implement. Issues in which we are interested define our concerns.
* **Cross-cutting concern** is a concern which is applicable throughout the application and it affects the entire application, e.g. logging, security and data transfer are the concerns which are needed in almost every module of an application, hence are cross-cutting concerns.

**► What is Join point?**

This represents a point in your application where you can plug-in AOP aspect. You can also say, it is the actual place in the application where an action will be taken using Spring AOP framework.

**► What is Advice?**

This is the actual action to be taken either before or after the method execution. This is actual piece of code that is invoked during program execution by Spring AOP framework.

**► What is Pointcut?**

This is a set of one or more joinpoints where an advice should be executed. You can specify pointcuts using expressions or patterns as we will see in our AOP examples.

**► What is Introduction?**

An introduction allows you to add new methods or attributes to existing classes.

**► What is Target object?**

The object being advised by one or more aspects, this object will always be a proxy object. Also referred to as the advised object.

**► What is Weaving?**

Weaving is the process of linking aspects with other application types or objects to create an advised object.

**► What are the different points where weaving can be applied?**

Weaving can be done at compile time, load time, or at runtime.

**► What are the types of advice?**

Spring aspects can work with five kinds of advice mentioned below:

* before: Run advice before the a method execution.
* after: Run advice after the a method execution regardless of its outcome.
* after-returning: Run advice after the a method execution only if method completes successfully.
* after-throwing: Run advice after the a method execution only if method exits by throwing an exception.
* around: Run advice before and after the advised method is invoked.

# Spring Boot

**► What is Spring Boot?**

Spring Boot is Spring project which makes it easy to create stand-alone, production-grade Spring based Applications that you can "just run". Most Spring Boot applications need very little Spring configuration. You can use Spring Boot to create Java applications that can be started using **java -jar** or more traditional **war** deployments. They also provide a command line tool that runs “spring scripts”. Features:

* Create **stand-alone** Spring applications
* Embed **Tomcat**, Jetty or Undertow directly (no need to deploy WAR files)
* Provide opinionated 'starter' POMs to simplify your Maven configuration
* Automatically configure Spring whenever possible
* Provide production-ready features such as metrics, **health checks** and externalized configuration
* Absolutely no code generation and no requirement for XML configuration

Spring Boot’s executable jars are ready-made for most popular **cloud PaaS** providers. These providers tend to require that you “bring your own container”; they manage application processes (not Java applications specifically), so they need some intermediary layer that adapts your application to the cloud’s notion of a running **process**.

Spring Boot 2.0 requires **Java 8** and Spring **Framework 5.0** or above. Explicit build support is provided for Maven 3.2+, and Gradle 2.9+ and 3. The following embedded servlet containers are supported out of the box - **Servlet 3.1**, **Jetty** 9.4, **Tomcat** 8.5. Spring Boot also provides an optional **Maven plugin** spring-boot-maven-plugin to create executable jars.

Spring Boot provides several “Starters” that make easy to add jars to your classpath, e.g. **spring-boot-starter-parent** in the parent section of the Maven POM. This is a special starter that provides useful Maven defaults and a dependency management section so that you can omit **version** tags for “blessed” dependencies.

**► How Spring Boot app starts?**

Use **main** method to delegate to Spring Boot’s SpringApplication class by calling run. SpringApplication will **bootstrap** our application, starting Spring which will in turn start e.g. auto-configured Tomcat web server. To gracefully exit the application hit **ctrl-c**.

public static void main(String[] args) {

**SpringApplication.run( Application.class, args);**

}

Another way:

**SpringApplication** springApplication = new SpringApplication(Application.class);

ApplicationContext ctx = springApplication.run(args);

**► How to create [fat] executable jar?**

Java does not provide any standard way to load **nested jar** files (i.e. jar files that are themselves contained within a jar), [**war** files could be used instead]. This can be problematic if you are looking to distribute a self-contained application. To solve this problem, many developers use “**uber**” jars. An uber jar simply packages all **classes**, from all jars, into a single archive. The problem with this approach is that it becomes hard to see which libraries you actually use. It can also be problematic if the same filename is used (but with different content) in multiple jars.

Spring Boot takes a different approach and allows you to actually **nest jars directly**. **Executable jars** (sometimes called “**fat jars**”) are archives containing your compiled classes along with all the jar dependencies that your code needs to run. To create an executable jar we need to add the **spring-boot-maven-plugin** to our pom.xml and run “**mvn package**”. The fat jar will have /BOOT-INF/lib/\*.jar

**► How to configure app, Java vs XML?**

Spring Boot favors **Java-based** configuration. Although it is possible to call SpringApplication.run() with a Spring **XML** source, they recommend that primary source is a **@Configuration** class. Usually the class that defines the **main** method is also a good candidate as the primary @Configuration.

**► How auto configure works?**

The **@EnableAutoConfiguration** or **@SpringBootApplication** annotation tells Spring Boot to “**guess**” how you will want to configure Spring, based on the **jar dependencies** that were added. Auto-configuration is designed to work well with “**Starters**”. E.g. if spring-boot-starter-web is used, it will add Tomcat and Spring MVC, so the auto configuration will assume that you are developing a **web application** and setup Spring accordingly. You should only ever add **one** @EnableAutoConfiguration annotation, to your primary @Configuration class.

If you find that specific auto-configure classes are being applied that you don’t want, you can use the **exclude** attribute of @EnableAutoConfiguration to disable them:

@Configuration

**@EnableAutoConfiguration(exclude={DataSourceAutoConfiguration.class})**

public class MyConfiguration {

}

The @**SpringBootApplication** annotation is equivalent to using @Configuration, @EnableAutoConfiguration and @ComponentScan with their default attributes.

**► How to handle Application exit?**

Each SpringApplication will register a **shutdown hook** with the JVM to ensure that the ApplicationContext is closed gracefully on exit. All the standard Spring lifecycle callbacks (such as the **DisposableBean** interface, or the @**PreDestroy** annotation) can be used. In addition, beans may implement the org.springframework.boot **ExitCodeGenerator** interface if they wish to return a specific **exit code** when the application ends.

**► How to externalize configuration?**

Spring Boot allows you to externalize your configuration so you can work with the same application code in different environments. Property values can be injected directly into your beans using the **@Value** annotation, accessed via Spring’s **Environment** abstraction or bound to structured objects via @**ConfigurationProperties**. Spring Boot uses a very particular PropertySource order for **17** config sources to allow sensible overriding of values. You can use:

* properties files,
* YAML files,
* environment variables
* command-line arguments

The RandomValuePropertySource is useful for injecting **random** values (e.g. into secrets or test cases). It can produce integers, longs, uuids or strings.

**► Can we use YAML instead of Properties?**

YAML is a superset of JSON, and as such is a very convenient format for specifying **hierarchical** configuration data. The SpringApplication class will automatically support YAML as an alternative to properties whenever you have the **SnakeYAML** library on your classpath. If you use ‘Starters’, SnakeYAML will be automatically provided via spring-boot-starter. Example YAML document:

environments:

dev:

url: http://dev.bar.com

name: Developer Setup

prod:

url: http://foo.bar.com

name: My Cool App

**► How is OAuth supported?**

If you have **spring-security-oauth2** on classpath you can take advantage of some autoconfiguration to make it easy to set up OAuth Authorization or Resource Server.

To create an **Authorization Server** and grant access tokens you need to use **@EnableAuthorizationServer** and provide security.oauth2.client.client-id and security.oauth2.client.client-secret properties. The client will be registered for you in an in-memory repository.

To use the access token you need a **Resource Server** (which can be the same as the Authorization Server). To create a Resource Server add @**EnableResourceServer** and provide some configuration to allow the server to decode access tokens.

**► Explain Actuator. How to enable production ready features?**

An **actuator** is a manufacturing term, referring to a mechanical device for moving or controlling something. Actuators can generate a large amount of motion from a small change.

The **Spring Boot Actuator** located in spring-boot-actuator module provides all of Spring Boot’s **production ready features** to help you monitor and manage your application when it’s pushed to production. You can choose to manage and monitor your application using **HTTP** endpoints or with **JMX**. **Auditing**, **health** and **metrics** gathering can be automatically applied to your application. The simplest way to enable the features is to add a dependency to the spring-boot-starter-actuator ‘Starter’.

Spring Boot’s actuator module includes additional support that is activated when you deploy to a compatible **Cloud Foundry** instance. The extended support allows Cloud Foundry management UIs (such as the web application that you can use to view deployed applications) to be augmented with Spring Boot actuator information. For example, an application status page may include full health information instead of the typical “running” or “stopped” status.

**► How to add Java EE Filter?**

Add FilterRegistrationBean with your class implementing javax.servlet.Filter and URL pattern for the filter to @Configuration class, e.g.

@Bean

public **FilterRegistrationBean** loggingFilter() {

FilterRegistrationBean registrationBean = new FilterRegistrationBean();

registrationBean.setFilter(new **RequestResponseLoggingFilter**());

registrationBean.addUrlPatterns("**/users/\***");

return registrationBean;

}

# Spring Cloud

**►►► What is Spring Cloud?**

Spring Cloud provides tools for developers to quickly build some of the **common patterns in distributed systems** (e.g. configuration management, service discovery, **circuit breakers**, intelligent routing, micro-proxy, control bus, one-time tokens, global locks, leadership election, distributed sessions, cluster state). Coordination of distributed systems leads to boiler plate patterns, and using Spring Cloud developers can quickly stand up services and applications that implement those patterns. They will work well in any distributed environment, including the developer's own laptop, bare metal data centers, and managed platforms such as Cloud Foundry.

<https://projects.spring.io/spring-cloud/>

* It is building blocks for Cloud and Microservices
* It provides microservices infrastructure.
* It provides several other open source projects like Netflix OSS.
* It provides PaaS like Cloud Foundry, AWS and Heroku.
* It uses Spring Boot style starters
* There are many use-cases supported by Spring Cloud like Cloud Integration, Dynamic Reconfiguration, Service Discovery, Security,Client side Load Balancing etc.
* Service Discovery (how do services find each other?)
* Client-side Load Balancing (which service instance to use?)

**► How to monitor microservices, e.g. for health?**

Spring Boot provides **actuator endpoints** to monitor metrics of individual microservices. These endpoints are very helpful for getting information about applications like if they are up, if their components like database etc are working good. But a major drawback or difficulty about using actuator enpoints is that we have to individually hit the enpoints for applications to know their status or health. Imagine microservices involving 50 applications, the admin will have to hit the actuator endpoints of all 50 applications. To help us deal with this situation, we will be using open source project located at https://github.com/codecentric/spring-boot-admin. Built on top of Spring Boot Actuator, it provides a web UI to enable us visualize the metrics of multiple applications.

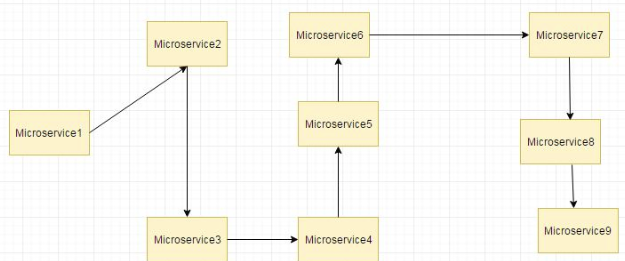
**► How to achieve server side load balancing using Spring Cloud?**

Server side load balancingcan be achieved using **Netflix Zuul**. Zuul is a JVM based router and server side load balancer by Netflix. It provides a single entry to our system, which allows a browser, mobile app, or other user interface to consume services from multiple hosts without managing cross-origin resource sharing (**CORS**) and authentication for each one. We can integrate Zuul with other Netflix projects like Hystrix for fault tolerance and Eureka for service discovery, or use it to manage routing rules, filters, and load balancing across your system.

**► In which business scenario to use Netflix Hystrix?**

Hystrix is a latency and fault tolerance library designed to isolate points of access to remote systems, services and 3rd party libraries, stop cascading failure and enable resilience in complex distributed systems where failure is inevitable.

Usually for systems developed using Microservices architecture, there are many microservices involved. These microservices collaborate with each other. Consider the following microservices:



Suppose the Microservice9 in the above diagram failed, then using the traditional approach we will propagate an exception. But this will still cause the whole system to crash. This problem gets more complex as the number of microservices increase. The number of microservices can be as high as 1000. This is where Hystrix comes into picture, so that we can use two features of Hystrix:

* Fallback method
* Circuit Breaker