**Spring:**

The Spring container is at the core of the Spring Framework. The container will create the objects, wire them together, configure them, and manage their complete life cycle from creation till destruction. The Spring container uses DI to manage the components that make up an application. These objects are called Spring Beans.

The container gets its instructions on what objects to instantiate, configure, and assemble by reading the configuration metadata provided. The configuration metadata can be represented either by XML, Java annotations, or Java code. The following diagram represents a high-level view of how Spring works. The Spring IoC container makes use of Java POJO classes and configuration metadata to produce a fully configured and executable system or application.



[**Spring Bean Factory Container**](https://www.tutorialspoint.com/spring/spring_beanfactory_container.htm)

This is the simplest container providing the basic support for DI and is defined by the org.springframework.beans.factory.BeanFactory interface.

[**Spring ApplicationContext Container**](https://www.tutorialspoint.com/spring/spring_applicationcontext_container.htm)

This container adds more enterprise-specific functionality such as the ability to resolve textual messages from a properties file and the ability to publish application events to interested event listeners

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.

The Spring Framework supports the following five scopes, three of which are available only if you use a web-aware Application Context.

**Singleton:** This scope the bean definition to a single instance per Spring IoC container (default).  
**Prototype:** This scope a single bean definition to have any number of object instances.

**Request:** This scope a bean definition to an HTTP request. Only valid in the context of a web-aware Spring Application Context.

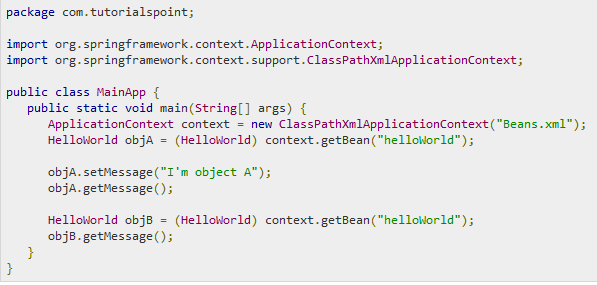
**Session:** This scope a bean definition to an HTTP session. Only valid in the context of a web-aware Spring Application Context.

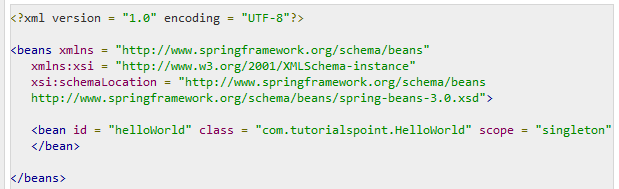
**global-session:** This scope a bean definition to a global HTTP session. Only valid in the context of a web-aware Spring Application Context.

**The singleton scopes**

If a scope is set to singleton, the Spring IoC container creates exactly one instance of the object defined by that bean definition. This single instance is stored in a cache of such singleton beans, and all subsequent requests and references for that named bean return the cached object.







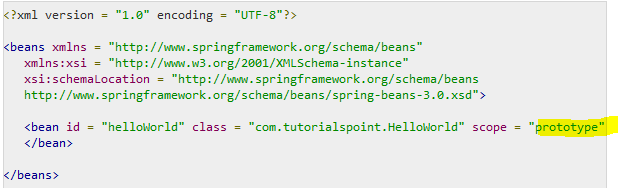
Once you are done creating the source and bean configuration files, let us run the application. If everything is fine with your application, it will print the following message −

Your Message: I'm object A

Your Message: I'm object A

## The prototype scopes

If the scope is set to prototype, the Spring IoC container creates a new bean instance of the object every time a request for that specific bean is made. As a rule, use the prototype scope for all state-full beans and the singleton scope for stateless beans.

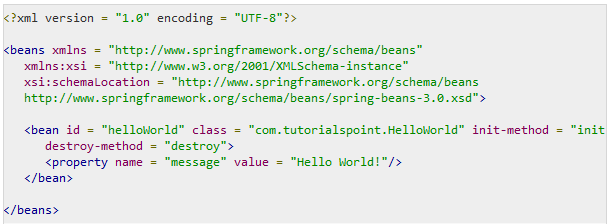


Once you are done creating the source and bean configuration files, let us run the application. If everything is fine with your application, it will print the following message −

Your Message: I'm object A

Your Message: null

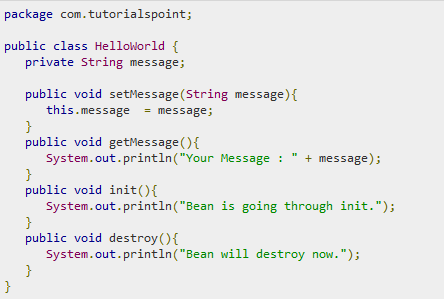
To define setup and teardown for a bean, we simply declare the <bean> with **init-method** and/or **destroy-method** parameters. The init-method attribute specifies a method that is to be called on the bean immediately upon instantiation. Similarly, destroy method specifies a method that is called just before a bean is removed from the container.

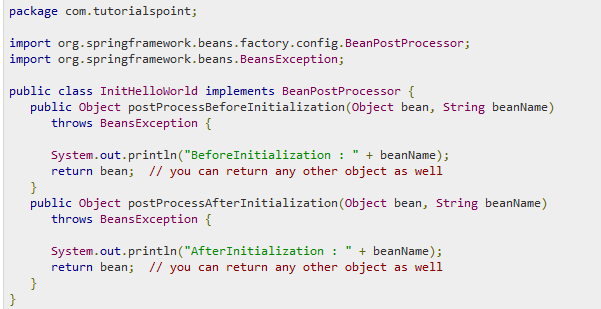


**BeanPostProcessors**

The BeanPostProcessors operate on bean (or object) instances, which means that the Spring IoC container instantiates a bean instance and then BeanPostProcessor interfaces do their work.

An **ApplicationContext** automatically detects any beans that are defined with the implementation of the **BeanPostProcessor** interface and registers these beans as postprocessors, to be then called appropriately by the container upon bean creation.







BeforeInitialization: helloWorld

Bean is going through init.

AfterInitialization: helloWorld

Your Message: Hello World!

Bean will destroy now.

**Dependency Injection:**

Consider you have an application which has a text editor component and you want to provide a spell check. Your standard code would look something like this −

public class TextEditor {

private SpellChecker spellChecker;

public TextEditor() {

spellChecker = new SpellChecker();

}

}

**In an inversion of control scenario, we would instead do something like this −**

public class TextEditor {

private SpellChecker spellChecker;

public TextEditor(SpellChecker spellChecker) {

this.spellChecker = spellChecker;

}

}

Here, the TextEditor should not worry about SpellChecker implementation. The SpellChecker will be implemented independently and will be provided to the TextEditor at the time of TextEditor instantiation. This entire procedure is controlled by the Spring Framework.

The code is cleaner with the DI principle and decoupling is more effective when objects are provided with their dependencies. The object does not look up its dependencies and does not know the location or class of the dependencies, rather everything is taken care by the Spring Framework.

**Spring - Beans Auto-Wiring**

The Spring container can auto wire relationships between collaborating beans without using <constructor-arg> and <property> elements, which helps cut down on the amount of XML configuration you write for a big Spring-based application.

**Spring - Annotation Based Configuration**

Once <context:annotation-config/> is configured, you can start annotating your code to indicate that Spring should automatically wire values into properties, methods, and constructors.

**@Required** annotation applies to bean property setter methods and it indicates that the affected bean property must be populated in XML configuration file at configuration time. Otherwise, the container throws a BeanInitializationException exception.

**@Autowired** annotation provides more fine-grained control over where and how auto wiring should be accomplished. The @Autowired annotation can be used to auto wire bean on the setter method just like @Required annotation, constructor, a property or methods with arbitrary names and/or multiple arguments

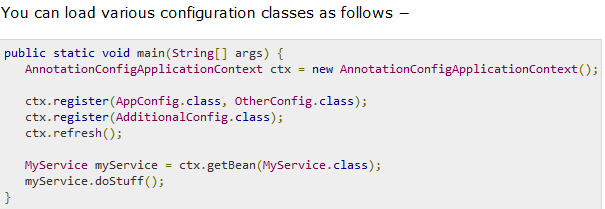
There may be a situation when you create more than one bean of the same type and want to wire only one of them with a property. In such cases, you can use the **@Qualifier** annotation along with **@Autowired** to remove the confusion by specifying which exact bean will be wired.

## @Configuration & @Bean Annotations

Annotating a class with the **@Configuration** indicates that the class can be used by the Spring IoC container as a source of bean definitions. 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

Ex:





## Lifecycle Callbacks

The @Bean annotation supports specifying arbitrary initialization and destruction callback methods, much like Spring XML's init-method and destroy-method attributes on the bean element −

public class Foo {

public void init () {

// initialization logic

}

public void cleanup () {

// destruction logic

}

}

@Configuration

public class AppConfig {

@Bean (initMethod = "init", destroyMethod = "cleanup")

public Foo foo () {

return new Foo ();

}

}

# **AOP with Spring Framework**

The functions that span multiple points of an application are called cross-cutting concerns and these cross-cutting concerns are conceptually separate from the application's business logic. There are various common good examples of aspects like logging, auditing, declarative transactions, security, caching, etc.

Dependency Injection helps you decouple your application objects from each other and AOP helps you decouple cross-cutting concerns from the objects that they affect.

# **Spring - JDBC Framework Overview**

Spring JDBC Framework takes care of all the low-level details starting from opening the connection, prepare and execute the SQL statement, process exceptions, handle transactions and finally close the connection.

## JdbcTemplate Class

The JDBC Template class executes SQL queries, updates statements, stores procedure calls, performs iteration over ResultSets, and extracts returned parameter values. It also catches JDBC exceptions and translates them to the generic, more informative, exception hierarchy defined in the org.springframework.dao package.

A common practice when using the JDBC Template class is to configure a *DataSource* in your Spring configuration file, and then dependency-inject that shared DataSource bean into your DAO classes, and the JdbcTemplate is created in the setter for the DataSource.

## Data Access Object (DAO)

DAO stands for Data Access Object, which is commonly used for database interaction. DAOs exist to provide a means to read and write data to the database and they should expose this functionality through an interface by which the rest of the application will access them.

The DAO support in Spring makes it easy to work with data access technologies like JDBC, Hibernate, JPA, or JDO in a consistent way.

**Querying for an integer/Long**

String SQL = "select count (\*) from Student";

int rowCount = jdbcTemplateObject.queryForInt(SQL);

long rowCount = jdbcTemplateObject.queryForLong(SQL);

**A simple query using a bind variable**

String SQL = "select age from Student where id =?";

int age = jdbcTemplateObject.queryForInt(SQL, new Object [] {10});

**Querying for a String**

String SQL = "select name from Student where id =?";

String name = jdbcTemplateObject.queryForObject(SQL, new Object [] {10}, String.class);

**Querying and returning an object**

String SQL = "select \* from Student where id =?";

Student student = jdbcTemplateObject.queryForObject(

SQL, new Object [] {10}, new StudentMapper());

public class StudentMapper implements RowMapper<Student> {

public Student mapRow(ResultSet rs, int rowNum) throws SQLException {

Student student = new Student ();

student.setID(rs.getInt("id"));

student.setName(rs.getString("name"));

student.setAge(rs.getInt("age"));

return student;

}

}

**Querying and returning multiple objects**

String SQL = "select \* from Student";

List<Student> students = jdbcTemplateObject.query(

SQL, new StudentMapper());

public class StudentMapper implements RowMapper<Student> {

public Student mapRow(ResultSet rs, int rowNum) throws SQLException {

Student student = new Student ();

student.setID(rs.getInt("id"));

student.setName(rs.getString("name"));

student.setAge(rs.getInt("age"));

return student;

}

}

**Inserting a row into the table**

String SQL = "insert into Student (name, age) values (?,?)";

jdbcTemplateObject.update(SQL, new Object [] {"Zara", 11});

**Updating a row into the table**

String SQL = "update Student set name =? where id =?";

jdbcTemplateObject.update(SQL, new Object [] {"Zara", 10});

**Deleting a row from the table**

String SQL = "delete Student where id =?";

jdbcTemplateObject.update(SQL, new Object [] {20} );

## Programmatic vs. Declarative

[Programmatic transaction management](https://www.tutorialspoint.com/spring/programmatic_management.htm) − 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.

[Declarative transaction management](https://www.tutorialspoint.com/spring/declarative_management.htm) − This means you separate transaction management from the business code. You only use annotations or XML-based configuration to manage the transactions.

Spring supports declarative transaction management through the Spring AOP framework.

## The Dispatcher Servlet

The Spring Web model-view-controller (MVC) framework is designed around a Dispatcher Servlet that handles all the HTTP requests and responses. The request processing workflow of the Spring Web MVC Dispatcher Servlet is illustrated in the following diagram –

* After receiving an HTTP request, Dispatcher Servlet consults the Handler Mapping to call the appropriate Controller.
* The Controller takes the request and calls the appropriate service methods based on used GET or POST method. The service method will set model data based on defined business logic and returns view name to the Dispatcher Servlet.
* The Dispatcher Servlet will take help from View Resolver to pick up the defined view for the request.
* Once view is finalized, The Dispatcher Servlet passes the model data to the view which is finally rendered on the browser.

