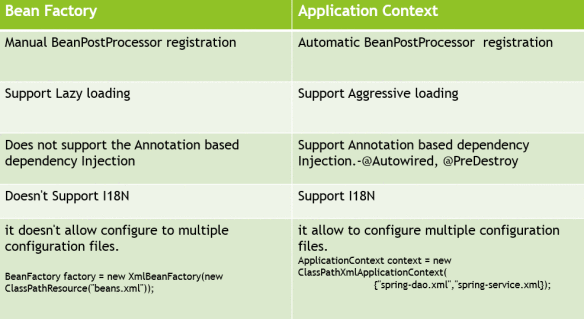
|  |  |  |
| --- | --- | --- |
| **Component** | **Sub component** | **Task** |
| **Spring Container** | **BeanFactory** | The container will create the objects, wire them together, configure them, and manage their complete life cycle from creation till destruction |
| **ApplicationContext** |
| **Bean** |  | A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container. created with the configuration metadata that you supply to the container |



**Configuration metadata** (information contained in bean definition): class, name, scope, constructor-arg, properties, autowiring mode, lazy-initialization mode, initialization method, destruction method

|  |  |  |
| --- | --- | --- |
| Methods to provide configuration metadata to the Spring Container | XML based configuration file. | <beans>  <bean id = "helloWorld" class = "com.tutorialspoint.HelloWorld" /> </beans> |
| Annotation-based configuration | First configure the same using <context:annotation-config/> tag. Once that is done, we can use @Required, Autowired, @Qualifier annotations. |
| **Java-based** configuration | **@Configuration** public class HelloWorldConfig {  **@Bean**   public HelloWorld helloWorld(){  return new HelloWorld();  } } |

|  |  |  |
| --- | --- | --- |
| **Scopes** | **singleton** | This scopes the bean definition to a **single instance per Spring IoC container** (default). **Not threadsafe** |
| **prototype** | This scopes a single bean definition to have **any number of object instances**. |
| request | This scopes a bean definition to an HTTP request. Only valid in the context of a web-aware Spring ApplicationContext. |
| session | This scopes a bean definition to an HTTP session. Only valid in the context of a web-aware Spring ApplicationContext. |
| global-session | This scopes a bean definition to a global HTTP session. Only valid in the context of a web-aware Spring ApplicationContext. |

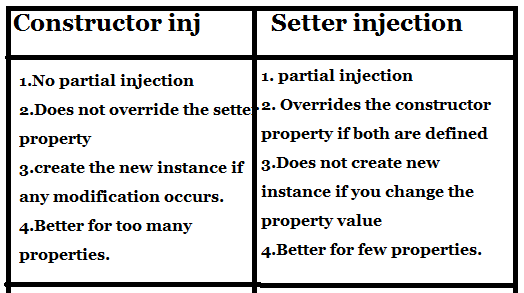
Following is **sequence of a spring** **bean lifecycle** in Spring:

* **Instantiate**: First the spring container finds the bean’s definition from the XML file and instantiates the bean.
* **Populate** **properties**: Using the dependency injection, spring populates all of the properties as specified in the bean definition.
* **Set Bean Name**: If the bean implements BeanNameAware interface, spring passes the bean’s id to setBeanName() method.
* **Set Bean factory**: If Bean implements BeanFactoryAware interface, spring passes the beanfactory to setBeanFactory() method.
* **Pre Initialization**: Also called post process of bean. If there are any bean BeanPostProcessors associated with the bean, Spring calls postProcesserBeforeInitialization() method.
* **Initialize beans**: If the bean implements IntializingBean,its afterPropertySet() method is called. If the bean has init method declaration, the specified initialization method is called.
* **Post Initialization**: – If there are any BeanPostProcessors associated with the bean, their postProcessAfterInitialization() methods will be called.
* **Ready to use**: Now the bean is ready to use by the application
* **Destroy**: If the bean implements DisposableBean , it will call the destroy() method

**Important bean lifecycle methods:**

* The first method is the **setup** method which is called during the loading of the bean into the container.
* The second is when the bean is unloaded from the container, and this method is called the **teardown**.

|  |  |  |
| --- | --- | --- |
| **BeanPostProcessor** | This interface defines callback methods that you can implement to provide your own instantiation logic, dependency-resolution logic, etc. You can also implement some custom logic after the Spring container finishes instantiating, configuring, and initializing a bean by plugging in one or more BeanPostProcessor implementations. | |
| **Dependency injection** | **Constructor based** :: <constructor-arg> tags for constructor-based injection :: use constructor arguments for mandatory dependencies | classes should be as independent as possible of other Java classes to increase the possibility to reuse these classes and to test them independently of other classes while unit testing. Dependency Injection (or sometime called wiring) helps in gluing these classes together and at the same time keeping them independent. |
| **Setter based** :: <property> tags for setter-based injection :: setters for optional dependencies. |



|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Inner beans** | | Beans that are defined within the scope of another bean. Thus, a <bean/> element inside the <property/> or <constructor-arg/> elements is called inner bean | | | | | |
| **Prototype scoped bean inside a Singleton bean.** 2 ways to handle this. | | **Lookup methods**  <bean id="requestProcessor" class="com.pramati.spring.RequestProcessor">  <lookup-method name="getValidator" bean="validator"/> </bean> | | | | Whenever we define a bean with lookup methods, Spring creates a subclass of the bean and overrides those methods which are marked as lookup-methods. And this subclassed bean gets registered into the context. The subclass delegates all the non-lookup methods to the original class. For the lookup methods, it overrides the implementation. In case of look up method injection, **proxy is created for singleton bean** | |
| **Scoped Proxies** <bean id="requestProcessor" class="com.pramati.spring.RequestProcessor">  <property name="validator" ref="validator"/> </bean>  <bean id="validator" scope="prototype" class="com.pramati.spring.RequestValidator">  <!-- This instructs the container to proxy the current bean-->  <aop:scoped-proxy/> </bean> | | | | **proxy is created for prototype bean** and wired into the singleton bean during the process of registering the singleton bean in the context. The proxy thus created understands the scope and returns instances based on the requirements of the scope | |
| Injecting collection | | **<list>** | This helps in wiring ie 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**. | | |
| **Event handling** | | Event handling in the ApplicationContext is provided through the ApplicationEvent class and ApplicationListener interface. | | | | |
| **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 restart 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. | | |
| Aspect oriented programming (**AOP**) | | **Cross cutting concerns** | | 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. In AOP the unit of modularity is the aspect | | |
| **Aspect** | | This 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 requirement. | | |
| **Join point** | | This represents a **point** in your application **where you can plug-in the AOP aspect**. You can also say, it is the actual place in the application where an action will be taken using Spring AOP framework. | | |
| **Advice** | | This is the **actual action to be taken** either before or after the method execution. This is an actual piece of code that is invoked during the program execution by Spring AOP framework. It is **implementation of an aspect.** Types: **Before Advice, After returning advice. After throwing advice, Finally advice, Around advice** | | |
| **Pointcut** | | This is a set of one or more join points where an **advice should be executed**. You can specify pointcuts using expressions or patterns as we will see in our AOP examples. | | |
| **Introduction** | | An introduction allows you to add new methods or attributes to the existing classes. | | |
| **Target object** | | The **object being advised** by one or more aspects. This object will always be a **proxied object**, also referred to as the advised object. | | |
| **Weaving** | | Weaving is the **process of linking aspects with other application** types or objects to create an advised object. This can be done at compile time, load time, or at runtime. Weaving is used to **create new proxy object by applying aspects to target object.** | | |
| Custom Aspect Implementation | | XML Schema based | | Aspects are implemented using the regular classes along with XML based configuration. | | |
| AspectJ based | | @AspectJ refers to a style of declaring aspects as regular Java classes annotated with Java 5 annotations. | | |
| Spring JDBC Framework | | **JdbcTemplate** | | 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. Instances of the JdbcTemplate class are **threadsafe** once configured. So you can configure a single instance of a JdbcTemplate and then safely inject this shared reference into multiple DAOs. | | |
| execute() method in JdbcTemplate | | To execute DDL statements. | | |
| Transaction management | | Atomicity | | A transaction should be treated as a single unit of operation, which means either the entire sequence of operations is successful or unsuccessful. | | |
| Consistency | | This represents the consistency of the referential integrity of the database, unique primary keys in tables, etc. | | |
| Isolation | | There may be many transaction processing with the same data set at the same time. Each transaction should be isolated from others to prevent data corruption. | | |
| Durability | | Once a transaction has completed, the results of this transaction have to be made permanent and cannot be erased from the database due to system failure. | | |
| Spring MVC | | **The Model** | | **encapsulates the application data** and in general they will consist of POJO. | | |
| **The View** | | It is responsible for **rendering the model data** and in general it **generates HTML output** that the client's browser can interpret. | | |
| **The Controller** | | It is **responsible for processing user requests** and building an appropriate model and passes it to the view for rendering. | | |
| HandlerMapping | | After receiving an HTTP request, *DispatcherServlet* consults the *HandlerMapping* to call the appropriate *Controller*. | | |
| 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 *DispatcherServlet*. | | |
| ViewResolver | | The *DispatcherServlet* will take help from *ViewResolver* to pickup the defined view for the request. View Resolver pattern is a J2EE pattern which allows a web application to dynamically choose it's view technology | | |
| View | | Once view is finalized, The *DispatcherServlet* passes the model data to the view which is finally rendered on the browser. | | |



* After receiving an HTTP request, DispatcherServlet consults the HandlerMapping 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 DispatcherServlet.
* The DispatcherServlet will take help from ViewResolver to pickup the defined view for the request.
* Once view is finalized, The DispatcherServlet passes the model data to the view which is finally rendered on the browser.

**Autowiring**

* **no**: no autowire
* **byName** : Autowiring that can be done by property name
* **byType** : property type as autowired
* **constructor**: It is similar to byType and it is property is in constructor
* **autodetect** :  Spring is allowed to select autowiring from byType or constructor

**Transaction management:**

* **Programmatic** transaction management : When only a **small amount of transactional operations** is there.
* **Declarative** transaction management: If there is a **big amount of transactional operations** to be taken care of.

**Autoproxying** is used to create proxy automatically for the spring users

Hibernate can be accessed in the following two ways:

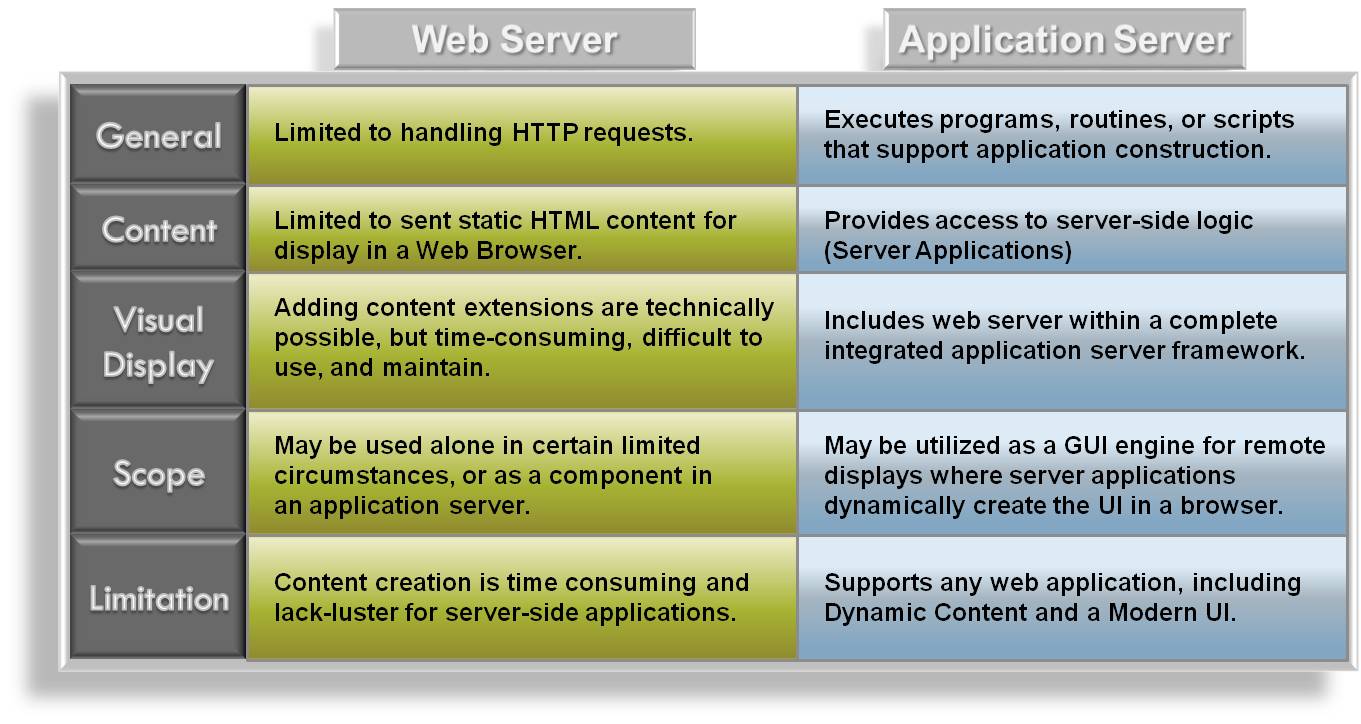
* By IOC with a Callback and HibernateTemplate.
* By applying an AOP Interceptor and broadening the HibernateDaoSupport.

**PreparedStatementCreator** is one of the most commonly used interfaces for writing data to the database. **createPreparedStatement()** is a method that can be used to create and return PreparedStatement from the Connection argument, and exception handling is automatically taken care of

Web.xml file



* **<servlet-mapping>** tag indicates what URLs will be handled by which DispatcherServlet.
* **load-on-startup: Pre initialization of servlet**
  + **If positive value :** element of web-app loads the servlet at the time of deployment or server start
  + **If negative value**: servlet will be loaded at request time, at first request



Examples of Web Servers are: **Apache Tomcat**and **Resin**.

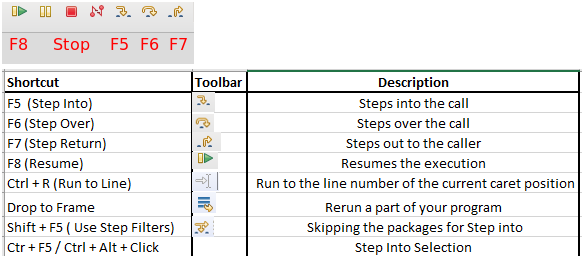
Examples of Application server: **JBoss, Glassfish, Weblogic, Websphere**

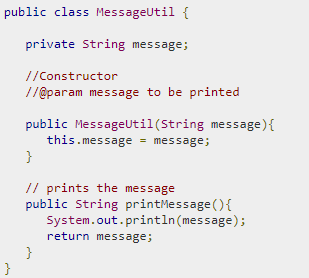
|  |  |
| --- | --- |
| **Spring 2.5** | **Spring 3.0** |
| In Spring 2.5, return ModelAndView is an old style code writing. | In spring 3.0 add later you can return both ModelAndView or nameView either. |
| Spring framework 2.5 is compatible with Java 1.4 and higher versions. | Spring framework 3.0 is compatible with Java 5 and higher versions. |
| In Spring framework 2.5, native expression language exists which is less powerful than spring expression language of spring 3.0 and it has different parsing rules. | Spring framework 3.0 introduces Spring Expression Language which defines bean definitions based XML and Annotation. |
| Spring framework 2.5 does not support type conversions and field formatting. | Spring framework 3.0 has type converting system and field formatting. |
| Spring framework 2.5 does not offer complete support for JSR303 bean validation API. | Spring framework 3.0 fully supports the JSR303 bean validation API. |
| Spring framework 2.5 does not support the embedded database engines. | Spring framework 3.0 offers support for various embedded database engines like HSQL, Derby and H2. |
| Spring framework 2.5 does not support Comprehensive REST. | Comprehensive REST support is available in Spring framework 3.0. |
| This is not available in 2.5. | We provide support for asynchronous method invocations through the use of the new @Async annotation. |
| This is not available in 2.5. | A mvc namespace has been introduced that greatly simplifies Spring MVC configuration. Additional annotations such as @CookieValue and @RequestHeaders have been added. |
|  |  |
| **Spring 3.0** | **Spring 4.0** |
| New features in Spring 3.0 compare to its previous version: | New features in Spring 4.0 compare to Spring 3.0: |
| - Spring MVC Test Framework | - JSR-335 Lambda expressions |
| - Asynchronous MVC processing on Servlet 3.0 | - JSR-310 Date-Time value types for Spring data binding and formatting. |
| - custom @Bean definition annotations in @Configuration classes | - JSR-343 JMS 2.0. |
| - @Autowired and @Value to be used as meta-annotations | - JSR-338 JPA 2.1. |
| - Concurrency refinements across the framework | - JSR-349 Bean Validation 1.1. |
| - Loading WebApplicationContexts in the TestContext framework | - JSR-236 Java EE 7 Enterprise Concurrency support. |
| - JCache 0.5 (JSR-107) | - JSR-356 Spring’s WebSocket endpoint mode. |
|  | - Configuring and implementing Spring style application using Groovy 2. Also they specify that first class support for the Groovy applications. |
|  | - Also spring plans to add the HATEOS (Hypermedia as the Engine of Application State) support for REST APIs. |
|  |  |
| **Java 7** | **Java 8** |
| Features Added: | Code name is Spider. Features Added: |
| -Support for dynamically-typed languages (InvokeDynamic): Extensions to the JVM, the Java language, and the Java SE API to support the implementation of dynamically-typed languages at performance levels near to that of the Java language itself | - JSR 335, JEP 126: Language-level support for lambda expressions. |
| - Strict class-file checking: Class files of version 51 (SE 7) or later must be verified with the typechecking verifier; the VM must not fail over to the old inferencing verifier. | - JSR 223, JEP 174: Project Nashorn, a JavaScript runtime which allows developers to embed JavaScript code within applications. |
| - Small language enhancements (Project Coin): A set of small language changes intended to simplify common, day-to-day programming tasks: Strings in switch statements, try-with-resources statements, improved type inference for generic instance creation (\"diamond\"), simplified varargs method invocation, better integral literals, and improved exception handling (multi-catch) | - JSR 308, JEP 104: Annotation on Java Types. |
| - Upgrade class-loader architecture: A method that frees the underlying resources, such as open files, held by a URLClassLoader | - Unsigned Integer Arithmetic. |
| - Concurrency and collections updates: A lightweight fork/join framework, flexible and reusable synchronization barriers, transfer queues, concurrent linked double-ended queues, and thread-local pseudo-random number generators. | - JSR 337, JEP 120: Repeating annotations. |
| - Internationalization Upgrade: Upgrade on Unicode 6.0, Locale enhancement and Separate user locale and user-interface locale. | - JSR 310, JEP 150: Date and Time API. |
| - More new I/O APIs for the Java platform (NIO.2), NIO.2 filesystem provider for zip/jar archives, SCTP, SDP, TLS 1.2 support. | - JEP 178: Statically-linked JNI libraries. |
| - Security & Cryptography implemented Elliptic-curve cryptography (ECC). | - JEP 153: Launch JavaFX applications (direct launching of JavaFX application JARs). |
| - Upgrade to JDBC 4.1 and Rowset 1.1. | - JEP 122: Remove the permanent generation. |
| - XRender pipeline for Java 2D, Create new platform APIs for 6u10 graphics features, Nimbus look-and-feel for Swing, Swing JLayer component, Gervill sound synthesizer. | - Java 8 is not supported on Windows XP. But as of JDK 8 update 5, it still can run under Windows XP after forced installation by directly unzipping from the installation |
| - Upgrade the components of the XML stack to the most recent stable versions: JAXP 1.4, JAXB 2.2a, and JAX-WS 2.2. |  |
| - Enhanced Managed Beans. |  |

|  |  |  |
| --- | --- | --- |
| **Black-Box Testing** | **Grey-Box Testing** | **White-Box Testing** |
| The internal workings of an application need not be known. | The tester has limited knowledge of the internal workings of the application. | Tester has full knowledge of the internal workings of the application. |
| Also known as closed-box testing, data-driven testing, or functional testing. | Also known as translucent testing, as the tester has limited knowledge of the insides of the application. | Also known as clear-box testing, structural testing, or code-based testing. |
| Performed by end-users and also by testers and developers. | Performed by end-users and also by testers and developers. | Normally done by testers and developers. |
| Testing is based on external expectations - Internal behavior of the application is unknown. | Testing is done on the basis of high-level database diagrams and data flow diagrams. | Internal workings are fully known and the tester can design test data accordingly. |

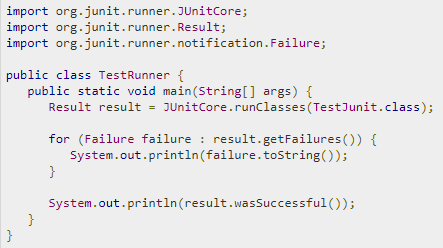
|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Testing** | **Category** | **Type** | **Sub type** | | **Purpose** | **Tool used** | | |
| **Java Code** | **JS Code** | **Angular JS code** |
| Testing | Functional | Unit | | | The goal of unit testing is to isolate each part of the program and show that individual parts are correct in terms of requirements and functionality | Junit, Mockito, Arquillian | JSUnit | Jasmine, Karma |
| Integration | Bottom up | | This testing begins with unit testing, followed by tests of progressively higher-level combinations of units called modules or builds. |
| Top down | | In this testing, the highest-level modules are tested first and progressively, lower-level modules are tested thereafter. |
| System | | | System testing tests the system as a whole. Once all the components are integrated, the application as a whole is tested rigorously to see that it meets the specified Quality Standards. |  |  |  |
| Regression | | | Regression testing is performed to verify that a fixed bug hasn't resulted in another functionality or business rule violation. The intent of regression testing is to ensure that a change, such as a bug fix should not result in another fault being uncovered in the application. | JWebUnit |  |  |
| Acceptance | | | This is arguably the most important type of testing, as it is conducted by the Quality Assurance Team who will gauge whether the application meets the intended specifications and satisfies the client’s requirement. |  |  |  |
| Non functional | Performance | | Load | It is a process of testing the behavior of a software by applying maximum load in terms of software accessing and manipulating large input data. | Apache Jmeter |  |  |
| Stress | The aim of stress testing is to test the software by applying the load to the system and taking over the resources used by the software to identify the breaking point. |  |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Junit | Fixtures |  | | The purpose of a test fixture is to ensure that there is a well-known and fixed environment in which tests are run so that results are repeatable. It includes: setUp() method, which runs before every test invocation & tearDown() method, which runs after every test method. |
| Test suites |  | | A test suite bundles a few unit test cases and runs them together. In JUnit, both @RunWith and @Suite annotation are used to run the suite test. |
| Test runners |  | | Test runner is used for executing the test cases. |
| JUnit classes | Assert | | Contains a set of assert methods. (Class's methods for which testing is done) |
| TestCase | | Contains a test case that defines the fixture to run multiple tests. |
| TestResult | | Contains methods to collect the results of executing a test case. |
|  |  |  | |  |
|  |  | **Junit API in junit.framework** | | |
|  |  | **Class** | **Methods** | |
|  |  | Assert | assertEquals(boolean expected, boolean actual),assertFalse(boolean condition),assertNotNull(Object object),assertNull(Object object),assertTrue(boolean condition),fail() | |
|  |  | TestCase | int countTestCases(),TestResult createResult(),String getName(),TestResult run(),void run(TestResult result)... | |
|  |  | TestResult | void addError(Test test, Throwable t), void addFailure(Test test, AssertionFailedError t), void endTest(Test test), int errorCount()… | |
|  |  | TestSuite | void addTest(Test test),void addTestSuite(Class<? extends TestCase> testClass),int countTestCases()… | |



Class Test case



Test Runner

**Spring Security**

* Where **authentication** is the process of establishing a principal (user) who claim to be.
* Where **authorization** is the process of deciding whether the logged in principal (user) allowed to perform a certain action.

Authentication models like [LDAP](https://javabeat.net/introduction-to-spring-ldap/), OpenID, ESB, JAAS, etc.

**Spring Security modules**: Since [spring security 3.0](https://javabeat.net/spring-security-3-0/), entire code has been divided and modules have been introduced to separate the different functionalities and third party dependencies to different modules

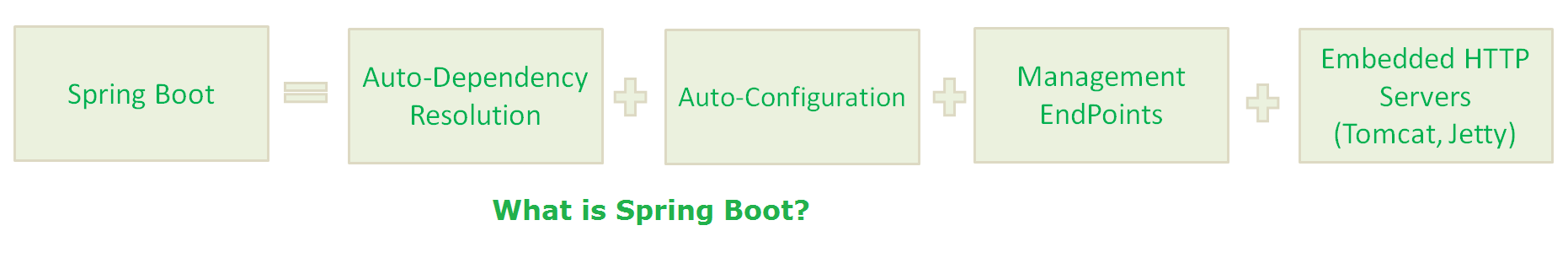
* **Core** – This module contains the APIs for basic authentication and access-control related mechanism. This is mandatory for ant spring security applications.
* **Remoting** – This module provides integration to the Spring Remoting. You don’t need to include this module unless you are writing a remote client applications.
* **Web** – This module contains APIs for servlet filters and any web based authentication like access restriction for URLs. Any web application would require this module.
* **Config** – You need it if you are using the Spring Security XML namespace for configuration. If you are not using XML configurations, you can ignore this module.
* **LDAP** – Required if you need to use LDAP authentication or manage LDAP user entries.
* **ACL** – Specialized domain object ACL implementation.
* **CAS** – Spring Security’s CAS client integration.
* **OpenID** – OpenID web authentication support.

|  |  |
| --- | --- |
| **Core Component** | **Description** |
| **SecurityContext** | As the name implies, this **interface** is the **corner stone of storing all the security related details** for your application. When you enable spring security for your application, a SecurityContext will enable for each application and stores the details of authenticated user, etc. It uses Authentication object for storing the details related to authentications. |
| **SecurityContextHolder** | This class is **important for accessing any value from the SecurityContext**. You would never directly access the security context, you have to use SecurityContextHolder to get the context and then access the details. In simple terms, it is an interface between client and context.By default, this SecurityContextHolder **uses ThreadLocal for storing the details**. |
| **UserDetailsService** | It is a **core interface** in spring security **to load user specific data**. This interface is considered as user DAO and will be implemented by specific DAO implementations. For example, for a basic in memory authentication, there is a **InMemoryUserDetailsManager**. This interface declares only one method **loadUserByUsername(String username)** which simplifies the implementation classes to write other specific methods. |
| **GrantedAuthority** | Apart from authenticating to the application, another important component is to **get the list of granted authorities for the logged in user**. This comes as **part of the authorization process**. This is retrieved by calling the **getAuthorities()**in Authentication object. This returns the list of GrantedAuthority which denotes roles for the users. |
| **DelegatingFilterProxy** | It is a **Proxy for standard Servlet Filter**, delegating to a Spring-managed bean that implements the Filter interface. Its the **starting point in the springSecurityFilterChain which instantiates the Spring Security filters** according to the Spring configuration. **It is delegating proxy to automatically intercept a URL with a particular pattern to apply spring** **security.** |

**Spring Boot**

Advantages:-

* 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.
* **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.



Prerequisites for Spring Boot

* Java 1.8
* Gradle 2.3+ or Maven 3.0+
* Spring Framework 5.0.0.BUILD-SNAPSHOT
* An IDE (Spring Tool Suit) is recommended.

Features:

* **Web Development**: can use the spring-boot- starter-web module to start and running application quickly
* **SpringApplication**: Class which provides the convenient way to bootstrap a spring application which can be started from main method.
  + This annotation (@SpringApplication) is equivalent to declaring these 3 annotations.
    - @Configuration, @EnableAutoConfiguration, @ComponentScan
* **Application events and listeners**
* **Externalized Configuration**: Allows us to externalize our configuration so that we can work with the same application in different environments
* **YAML Support**
* **Type-safe Configuration:** Strong type-safe configuration is provided to govern and validate the configuration of application. Application configuration is always a crucial task which should be type-safe
* **Logging**: Uses Common logging for all internal logging. Logging dependencies are managed by default
* **Security**: it is secure by default with basic authentication on all HTTP endpoints. A rich set of Endpoints are available for develop a secure Spring Boot application.

|  |  |
| --- | --- |
| **Approach to create project** | **Details** |
| **Spring Maven Project** | **Create Maven project Add following dependency** <dependencies>   <dependency>   <groupId>org.springframework.boot</groupId>   <artifactId>spring-boot-starter-web</artifactId>   </dependency>  </dependencies> **Add Java version** <properties>   <java.version>1.8</java.version>  </properties>  **Create Main class and call the main method as shown below** @SpringBootApplication public class Main {  public static void main(String args[]) {  SpringApplication.run(Main.class, args);  } } |
| **Spring Starter Project Wizard** | • Select Spring Starter project in Spring Tool Suite (STS)  • Select dependency as web to create web project • Default pom.xml is created. • This project auto generates a Java file as given below inside the src/main/java which can be run as Java application. |
| **Spring Initializr** | • Spring Boot Initilizr is a Spring Boot tool to bootstrap Spring Boot or Spring Applications very easily. It is a web tool which is provided by Spring on official site. You can create Spring Boot project by providing project details • Select Maven project and dependencies. Fill other details and click on generate project. • After clicking it asked for download project • Save project and extract it. • Now import this project by using import option from the STS (Spring Tool Suite) IDE. • And extract as a maven project. |
| **Spring Boot CLI** | • It is a tool which you can download from the official site of Spring Framework.  • Download the CLI tool from official site. • After downloading, extract the zip file. It contains a bin folder, in which spring setup is stored. We can use it to execute Spring Boot application. • CLI executes groovy files. |

**@RestController**annotation informs to the Spring to render the result back to the caller.

**“Transitively Dependency Resolution Management”** means: If we define a “A” dependency in build scripts, “A” is dependent on “B” and “B” is dependent on “C”, That means “A” is also dependent on “C”. Then Build Tools will download and add all Three Jar files “A”, “B” and “C” to our application classpath.

**@RequestMapping**annotation is used to provide routing information. It tells to the Spring that any HTTP request should map to the corresponding method

**Spring Boot Starters**

* Starters are a **set of convenient dependency descriptors** which we can include in our application.
* Spring Boot provides built-in starters which makes development easier and rapid. For example, if we want to get started using Spring and JPA for database access, just include the **spring-boot-starter-data-jpa** dependency in your project.
* Starter should follow a naming pattern like: **spring-boot-starter**-\*, where \* is a particular type of application. This naming structure is intended to help when you need to find a starter.
* **spring-boot-starter-actuator**
  + It is **used to access current state of running application in production environment**.
  + It is used for Spring Boot’s Actuator which provides production ready features to help you monitor and manage your application.
  + Spring boot actuator provides restful web services end points which you can simply use and check various metrics.  
    For example: **/metrics :**This restful end point will show you metrics such as free memory, processors, uptime and many more properties.
  + Actuator is a tool which has HTTP endpoints.
  + When application is pushed to production, you can choose to manage and monitor your application using HTTP endpoints.
* **Thymeleaf**
  + It is a **server side Java template engine for web application**. It's main goal is to bring elegant natural templates to your web application.
  + It can be integrate with Spring Framework and **ideal for HTML5** Java web applications.

**LDAP**

* Lightweight Directory Access Protocol (LDAP) is the protocol for accessing the preeminent directory services deployed in the world today. LDAP is a global directory service, industry-standard protocol, which is based on client-server model and runs on a layer above the TCP/IP stack.
* The LDAP provides a facility to connect to, access, modify, and search the internet directory.
* Working:
  + The LDAP servers contain information which is organized in the form of a directory tree.
  + The clients ask server to provide information or to perform some operation on a particular information.
  + The server answers the client by providing required information if it has one, or it refers the client to another server for action on required information.
  + The client then acquires the desired information from another server.
  + The **tree structure of directory is maintained same across all the participating servers**.
  + This is a **prominent feature of LDAP** directory service.
  + Hence, irrespective of which server is referred to by the client, the client always gets required information in an error-free manner.
* **Types of LDAP: Transparent and Plugin**
* The action in LDAP takes place around a data structure known as an *entry*.

