## Spring Boot intro

* A Spring module which provides RAD (Rapid Application Development) feature to Spring framework.
* Create stand alone spring based application that you can just run because it needs very little spring configuration.

=> Convention over configuration => decrease efforts of Developer.

Spring Boot offers a fast way to build applications. It looks at your classpath and at beans you have configured, makes reasonable assumptions about what you’re missing, and adds it. With Spring Boot you can focus more on business features and less on infrastructure.

For example:

Got Spring MVC? There are several specific beans you almost always need, and Spring Boot adds them automatically. A Spring MVC app also needs a servlet container, so Spring Boot automatically configures embedded Tomcat.

Got Jetty? If so, you probably do NOT want Tomcat, but instead embedded Jetty. Spring Boot handles that for you.

* Create stand-alone Spring applications that can be started using **java -jar**.
* Embed Tomcat, Jetty or Undertow directly. You don't need to deploy WAR files.
* It provides opinionated 'starter' POMs to simplify your Maven configuration.
* It automatically configure Spring whenever possible.
* It provides production-ready features such as metrics, health checks and externalized configuration.
* Absolutely no code generation and no requirement for XML configuration.

Spring Boot doesn’t generate code or make edits to your files. Instead, when you start up your application, Spring Boot dynamically wires up beans and settings and applies them to your application context.

Add parent (**spring-boot-starter-parent**) to maven project --> To make a maven project **Spring-boot project.**

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>**spring-boot-starter-parent**</artifactId>

<version>1.4.2.RELEASE</version>

</parent>

**Spring Boot web app:**

@RestController

public class HelloController {

@RequestMapping("/")

public String index() {

**@RestController** = @Controller + @ResposeBody

* ready for use by Spring MVC to handle web requests.
* @RequestMapping maps / to the index() method
* web requests returning data rather than a view.

**@SpringBootApplication** => convenience annotation that adds all of the following:

* **@Configuration** tags the class as a source of bean definitions for the application context.
* **@EnableAutoConfiguration** tells Spring Boot to start adding beans based on classpath settings, other beans, and various property settings.
* Normally you would add **@EnableWebMvc** for a Spring MVC app, but Spring Boot adds it automatically when it sees **spring-webmvc** on the classpath. This **flags the application as a web application** and activates key behaviors such as setting up a **DispatcherServlet**.
* **@ComponentScan** tells Spring to look for other components, configurations, and services in the **package** of the class, allowing it to find the controllers.

The **main()** method uses Spring Boot’s **SpringApplication.run()** method to launch an application. There wasn’t a single line of XML? **No web.xml** file either. This web application is **100% pure Java**.

**Adding Unit Tests to Spring Boot:**

1. Add below dependecy to Maven project:

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>**spring-boot-starter-test**</artifactId>

<scope>test</scope>

</dependency>

1. Write a simple unit test that mocks the servlet request and response through your endpoint:

@RunWith(SpringRunner.class)

@SpringBootTest

@AutoConfigureMockMvc

public class HelloControllerTest {

@Autowired

private MockMvc mvc;

@Test

public void getHello() throws Exception {

mvc.perform(MockMvcRequestBuilders.get("/")

.accept(MediaType.APPLICATION\_JSON))

.andExpect(status().isOk())

.andExpect(content().string(equalTo("Greetings from Spring Boot!")));

}

}

**MockMvc** comes from Spring Test (**@AutoConfigureMockMvc** + **@SpringBootTest** inject it)and allows you, via a set of convenient builder classes, to send HTTP requests into the DispatcherServlet and make assertions about the result.

**@SpringBootTest** => whole app context to be created. Alternative is **@WebMvcTest** which asks Spring Boot to create only the web layers of the context.

Spring Boot automatically tries to locate the main application class of your application.

**Adding full-stack integration to Spring Boot:**

@RunWith(SpringRunner.class)

@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM\_PORT)

public class HelloControllerIT {

@LocalServerPort

private int port;

private URL base;

@Autowired

private TestRestTemplate template;

@Before

public void setUp() throws Exception {

this.base = new URL("http://localhost:" + port + "/");

}

@Test

public void getHello() throws Exception {

ResponseEntity<String> response = template.getForEntity(base.toString(), String.class);

assertThat(response.getBody(), equalTo("Greetings from Spring Boot!"));

}

}

The embedded server is started up on a random port by virtue of the **webEnvironment = SpringBootTest.WebEnvironment.RANDOM\_PORT** and the actual port is discovered at runtime with the **@LocalServerPort**.

## Core Java

**Java 7 Features:**

<https://way2java.com/java-versions-2/jdk-1-7-features/>

<http://javarevisited.blogspot.in/2014/04/10-jdk-7-features-to-revisit-before-you.html>

Java 6 was nothing on feature; it was all about JVM changes and performance.

|  |  |  |
| --- | --- | --- |
| String in Switch | * Earlier to **JDK 1.**7, **switch** expression takes**int** values or convertible to **int**. * From JDK 1.7, switch accepts string objects also as expression. | * equals() and hashcode() method from java.lang.String is used in comparison, which is case-sensitive. * Java compiler can generate more efficient code than using nested if-then-else statement. |
| Automatic Type Inference in Generic object instantiation  (Diamond operator, <> , in collection classes) | * Empty angle brackets (known as **diamond operator**), **<>**, can be used in specifying generic type instead of writing the exact one. | * Specify types on now only needed on left hand side:   Ex.  Map<String, List<String>> employeeRecords = new HashMap<>();  List<Integer> primes = new ArrayList<>(); |
| Automatic Resource Management | * Before JDK 7, we need to use a finally block, to ensure that a resource is closed regardless of whether the try statement completes normally or abruptly. * JDK 7 introduces a try-with-resources statement, which ensures that each of the resources in try(resources) is closed at the end of the statement by calling close() method of AutoClosable. * Any resource (class) that implements interface "java.lang.AutoCloseable" is eligible as a resource statement to write in try block. | * Now in Java 7, you can use try-with-resource feature to automatically close resources, which implements AutoClosable and Closeable interface e.g. Streams, Files, Socket handles, database connections etc. * Since Java is taking care of closing opened resources including files and streams, may be no more leaking of file descriptors and probably an end to file descriptor error. Even JDBC 4.1 is retrofitted as AutoClosable too. * The close() method of AutoCloseable is called implicitly to close the handles. close() method java.lang.Closeable interface is very different from this. |
| Handling multiple exceptions in a single catch block | * A single catch block can handle more than one exception types **separated by pipe ( | ) symbol**. * Super class exception must be caught separately (it is a constraint) i.e . Alternatives in a multi-catch statement cannot be related by sub classing. | * **try** { ...... } **catch**(**ClassNotFoundException**|**SQLException** ex) { ex.printStackTrace(); } * } catch (FileNotFoundException | IOException ex) {   OR  } catch(ArithmeticException | RuntimeException e) {  will throw compile time error. |
| Fork Join Framework | * The fork/join framework is an implementation of the **ExecutorService interface** that allows you to take advantage of multiple processors available in modern servers. | * It is designed for work that can be broken into smaller pieces recursively. * The goal is to use all the available processing power to enhance the performance of your application. |
| Underscore in Numeric literals | * Can insert underscore(s) '\_' in between the digits in an **numeric literals (integral and floating-point literals)** to improve readability. | * long creditCardNumber = 1234\_4567\_8901\_2345L; //16 digit   long ssn = 777\_99\_8888L;  double pi = 3.1415\_9265;  float pif = 3.14\_15\_92\_65f; |
| Integral Types as Binary Literals | * The **integer whole numbers** like byte, short, int and long can be expressed in binary format also with a prefix of 0b or 0B. | * Earlier, we have **0 prefix for octal** and **0x prefix for hexa** and no prefix for binary. |
| Static Blocks | * Earlier to JDK 1.7, to print **static blocks**, no main() method is required. | * From JDK 1.7, if no main() exists, static blocks will not be executed. |
| G1 Garbage Collector | * Garbage first ( G1) performs clean-up where there is most garbage. * It's said that G1 is quite predictable and provides greater **throughput for memory intensive applications**. | * To achieve this it split Java heap memory into multiple regions as opposed to 3 regions in the prior to Java 7 version (new, old and permgen space). |
| More Precise Rethrowing of Exception | * The Java SE 7 compiler performs more precise analysis of re-thrown exceptions than earlier releases of Java SE. This enables you to specify more specific exception types in the **throws clause of a method declaration.** | * Before JDK 7, re-throwing an exception was treated as throwing the type of the catch parameter. * public void obscure() throws Exception{    try {  new FileInputStream("abc.txt").read();  new SimpleDateFormat("ddMMyyyy").parse("12-03-2014"); } catch (Exception ex) {  throw ex;  } } |

**Java 8 Features:**

Oracle released a new version of Java as Java 8 in March 18, 2014. It was a revolutionary release of the Java for software development platform. It includes **various upgrades to the Java programming, JVM, Tools and libraries.**

|  |  |  |
| --- | --- | --- |
| Programming Language Enhancements | * Lambda expressions, * Method references, * Functional interfaces, * Optional class, * ForEach() method * Stream API, * Default methods, * Base64 Encode Decode, * Static methods in interface, * Collectors class * Parallel array sorting, * Nashorn JavaScript Engine, * Parallel Array Sorting, * Type and Repating Annotations, * IO Enhancements, * Concurrency Enhancements, * JDBC Enhancements etc. | * **Lambda expression** helps us to write our code in functional style. It provides a clear and concise way to implement SAM interface(Single Abstract Method) by using an expression. It is very useful in collection library in which it helps to iterate, filter and extract data. * **Method reference** is a compact and easy form of lambda expression. Each time when you are using lambda expression to just referring a method, you can replace your lambda expression with method reference. * An Interface that contains only one abstract method is known as **functional interface**. Also known as Single Abstract Method Interfaces (SAM Interfaces). * **java.util.Optional** - public final class which is used to deal with **NullPointerException** in Java application. It provides methods to check the presence of value for particular variable. * **forEach() method in java.lang.Iterable interface.** |
|  |  |  |