Spring Boot vs. Spring MVC vs. Spring: How Do They Compare?

In this article, you will receive overviews of Spring, Spring MVC, and Spring Boot, learn what problems they solve, and where they're best applied. The most important thing that you will learn is that Spring, Spring MVC, and Spring Boot are not competing for the same space. They solve different problems and they solve them very well.

What Is the Core Problem That Spring Framework Solves?

Think long and hard. What's the problem Spring Framework solves?

> The most important feature of Spring Framework is Dependency Injection. At the core of all Spring Modules is Dependency Injection or IOC Inversion of Control.

Why is this important? Because, when DI or IOC is used properly, we can develop loosely coupled applications. And loosely coupled applications can be easily unit tested.

Let's consider a simple example.

Example Without Dependency Injection

Consider the example below: WelcomeController depends on WelcomeService to get the welcome message. What is it doing to get an instance of WelcomeService?

```
WelcomeService service = new WelcomeService();
```

It's creating an instance of it. And that means they are tightly coupled. For example: If I create a mock for WelcomeService in a unit test for WelcomeController, how do I make WelcomeController use the mock? Not easy!

```
@RestController
public class WelcomeController {

    private WelcomeService service = new WelcomeService();

    @RequestMapping("/welcome")
```

```
    public String welcome() {
        return service.retrieveWelcomeMessage();
    }
}
```

Same Example with Dependency Injection

The world looks much simpler with dependency injection. You let the Spring Framework do the hard work. We just use two simple annotations: @Component and @Autowired.

- Using `@Component`, we tell Spring Framework: Hey there, this is a bean that you need to manage.
- Using `@Autowired`, we tell Spring Framework: Hey find the correct match for this specific type and autowire it in.

In the example below, Spring framework would create a bean for WelcomeService and autowire it into WelcomeController.

In a unit test, I can ask the Spring framework to auto-wire the mock of WelcomeService into WelcomeController. (Spring Boot makes things easy to do this with @MockBean. But, that's a different story altogether!)

```
@Component
public class WelcomeService {
    //Bla Bla Bla
}

@RestController
public class WelcomeController {

    @Autowired
    private WelcomeService service;

    @RequestMapping("/welcome")
    public String welcome() {
        return service.retrieveWelcomeMessage();
    }
```

```
}
```

What Else Does Spring Framework Solve?

Problem 1: Duplication/Plumbing Code

Does Spring Framework stop with Dependency Injection? No. It builds on the core concept of Dependency Injection with a number of Spring Modules

- Spring JDBC
- Spring MVC
- Spring AOP
- Spring ORM
- Spring JMS
- Spring Test

Consider Spring JMS and Spring JDBC for a moment.

Do these modules bring in any new functionality? No. We can do all this with J2EE or Java EE. So, what do these bring in? They bring in simple abstractions. The aim of these abstractions is to

- Reduce Boilerplate Code/Reduce Duplication
- Promote Decoupling/Increase Unit Testability

For example, you need much less code to use a JDBCTemplate or a JMSTemplate compared to a traditional JDBC or JMS.

Problem 2: Good Integration With Other Frameworks

The great thing about Spring Framework is that it does not try to solve problems that are already solved. All that it does is to provide a great integration with frameworks which provide great solutions.

- Hibernate for ORM
- iBatis for Object Mapping
- JUnit and Mockito for Unit Testing

What Is the Core Problem That Spring MVC Framework Solves?

> Spring MVC Framework provides decoupled way of developing web applications. With simple concepts like Dispatcher Servlet, ModelAndView and View Resolver, it makes it easy to develop web applications.

Why Do We Need Spring Boot?

Spring based applications have a lot of configuration.

When we use Spring MVC, we need to configure component scan, dispatcher servlet, a view resolver, web jars(for delivering static content) among other things.

```
<bean

class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix">
            <value>/WEB-INF/views/</value>
        </property>
        <property name="suffix">
            <value>.jsp</value>
        </property>
  </bean>

  <mvc:resources mapping="/webjars/**" location="/webjars/"/>
```

The code snippet below shows the typical configuration of a dispatcher servlet in a web application.

```
<servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>
            org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
```

```
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>/WEB-INF/todo-servlet.xml</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
```

When we use Hibernate/JPA, we would need to configure a datasource, an entity manager factory, a transaction manager among a host of other things.

```
<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource"
        destroy-method="close">
        <property name="driverClass" value="${db.driver}" />
        <property name="jdbcUrl" value="${db.url}" />
        <property name="user" value="${db.username}" />
        <property name="password" value="${db.password}" />
    </bean>

    <jdbc:initialize-database data-source="dataSource">
        <jdbc:script location="classpath:config/schema.sql" />
        <jdbc:script location="classpath:config/data.sql" />
    </jdbc:initialize-database>

    <bean

class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean"
        id="entityManagerFactory">
        <property name="persistenceUnitName" value="hsql_pu" />
        <property name="dataSource" ref="dataSource" />
    </bean>

    <bean id="transactionManager"
```

```
class="org.springframework.orm.jpa.JpaTransactionManager">
        <property name="entityManagerFactory" ref="entityManagerFactory" />
        <property name="dataSource" ref="dataSource" />
    </bean>

    <tx:annotation-driven transaction-manager="transactionManager"/>
```

Problem #1: Spring Boot Auto Configuration: Can We Think Different?

Spring Boot brings a new thought process around this.

> Can we bring more intelligence into this? When a spring mvc jar is added into an application, can we auto configure some beans automatically?

- How about auto-configuring a Data Source if Hibernate jar is on the classpath?
- How about auto-configuring a Dispatcher Servlet if Spring MVC jar is on the classpath?

There would be provisions to override the default auto configuration.

> Spring Boot looks at a) Frameworks available on the CLASSPATH b) Existing configuration for the application. Based on these, Spring Boot provides basic configuration needed to configure the application with these frameworks. This is called `Auto Configuration`.

Problem #2: Spring Boot Starter Projects: Built Around Well-Known Patterns

Let's say we want to develop a web application.

First of all, we would need to identify the frameworks we want to use, which versions of frameworks to use and how to connect them together.

All web application have similar needs. Listed below are some of the dependencies we use in our Spring MVC Course. These include Spring MVC, Jackson Databind (for data binding), Hibernate-Validator (for server side validation using Java Validation API) and Log4j (for logging). When creating this course, we had to choose the compatible versions of all these frameworks.

```

<dependency>
```

```
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>4.2.2.RELEASE</version>
</dependency>

<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.5.3</version>
</dependency>

<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>5.0.2.Final</version>
</dependency>

<dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.17</version>
</dependency>
```

Here's what the Spring Boot documentations says about starters.

> Starters are a set of convenient dependency descriptors that you can include in your application. You get a one-stop-shop for all the Spring and related technology that you need, without having to hunt through sample code and copy paste loads of dependency descriptors. For example, if you want to get started using Spring and JPA for database access, just include the spring-boot-starter-data-jpa dependency in your project, and you are good to go.

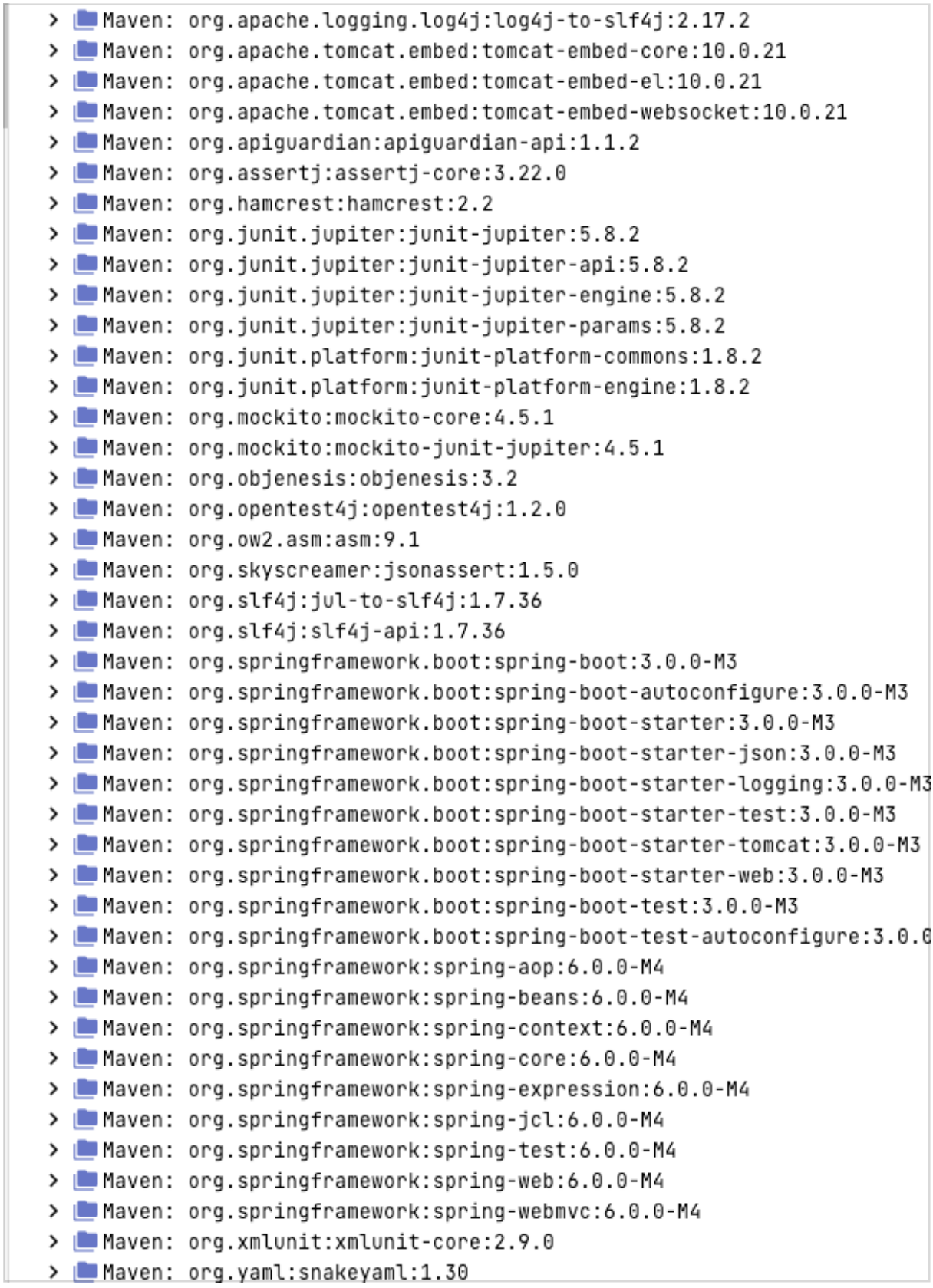
Let's consider an example starter: Spring Boot Starter Web.

If you want to develop a web application or an application to expose restful services, Spring Boot Start Web is the starter to pick. Let's create a quick project with Spring Boot Starter Web using Spring Initializr.

Dependency for Spring Boot Starter Web

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

The following screenshot shows the different dependencies that are added into our application



Dependencies can be classified into:

- Spring: core, beans, context, aop
- Web MVC: (Spring MVC)
- Jackson: for JSON Binding
- Validation: Hibernate Validator, Validation API

- Embedded Servlet Container: Tomcat
- Logging: logback, slf4j

Any typical web application would use all these dependencies. Spring Boot Starter Web comes pre-packaged with these. As a developer, I would not need to worry about either these dependencies or their compatible versions.

Spring Boot Starter Project Options

As we see from Spring Boot Starter Web, starter projects help us in quickly getting started with developing specific types of applications.

- spring-boot-starter-web-services: SOAP Web Services
- spring-boot-starter-web: Web and RESTful applications
- spring-boot-starter-test: Unit testing and Integration Testing
- spring-boot-starter-jdbc: Traditional JDBC
- spring-boot-starter-hateoas: Add HATEOAS features to your services
- spring-boot-starter-security: Authentication and Authorization using Spring Security
- spring-boot-starter-data-jpa: Spring Data JPA with Hibernate
- spring-boot-starter-cache: Enabling Spring Framework's caching support
- spring-boot-starter-data-rest: Expose Simple REST Services using Spring Data REST

Other Goals of Spring Boot

There are a few starters for technical stuff as well

- spring-boot-starter-actuator: To use advanced features like monitoring and tracing to your application out of the box
- spring-boot-starter-undertow, spring-boot-starter-jetty, spring-boot-starter-tomcat: To pick your specific choice of Embedded Servlet Container
- spring-boot-starter-logging: For Logging using logback
- spring-boot-starter-log4j2: Logging using Log4j2

Spring Boot aims to enable production ready applications in quick time.

- Actuator: Enables Advanced Monitoring and Tracing of applications.
- Embedded Server Integrations: Since the server is integrated into the application, I would need to have a separate application server installed on the server.

- Default Error Handling

Spring Boot vs. Spring MVC vs. Spring: How Do They Compare?