Western Norway
University of
Applied Sciences

Web Frameworks: Spring

Created by Lasse Jenssen

Home

Agenda: The Spring Framwork

- History
- The IoC Container
- IoC - Inversion of Controll
- Dependency Injection
- Spring Boot
- Spring Initializer

Syllabus for this lecture

DZone: Dependency Injection in Spring

Link: https://dzone.com/articles/dependency-injection-in-spring

The Spring Framework

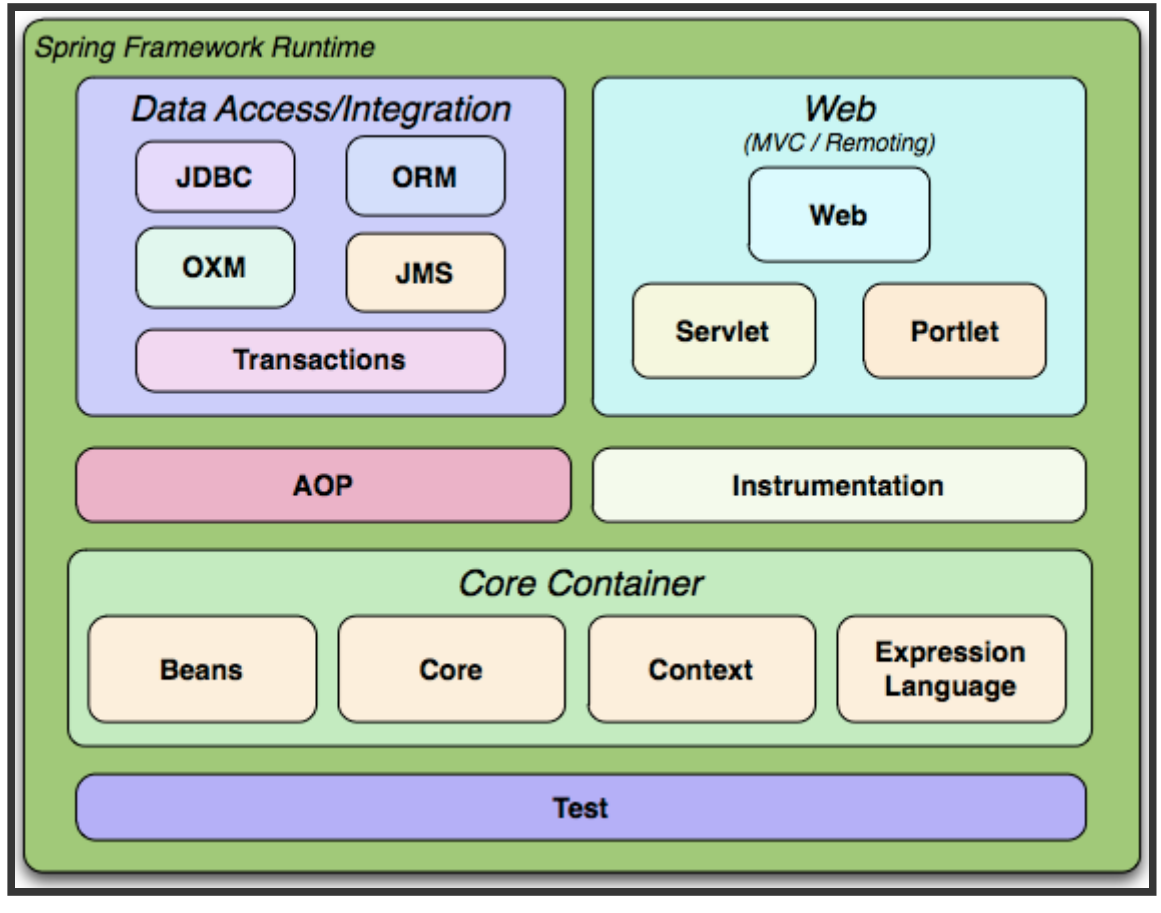
History

- 2002: Book "Expert One-on-One J2EE Design and Development", Rod Johnsen (first version)
- 2003: 0.9 version of the Framwork released (under Apache 2.0 license)
- 2004 March: First production release (1.0)
- Latest version (Per 5th of May 2022): 5.3

The Spring Framework

Introduction

- A response to the complexity of the early J2EE specifications.
- A complementary to Java EE.
- Integrates with carefully selected individual specifications from the EE umbrella: Servlet API, WebSocket API, JSON Binding API, JPA etc.
- Very much integrated with Maven (or Gradle).


Spring Framework Runtime
Data Access/Integration
JDBC
ORM
OXM
JMS
Transactions
Web
(MVC / Remoting)
Web
Servlet
Portlet
AOP
Instrumentation
Core Container
Beans
Core
Context
Expression Language
Test

The Spring Framework

Design Philosophy

- Provide choice at every level.
- Accommodate diverse perspectives

  (not opinionated about how things should be done).
- Maintain strong backward compatibility.
- Care about API design.
- Set high standards for code quality.

http://spring.io/projects

The Spring Framework

The IoC Container

- The core of Spring Framework.
- Responsible to instantiate, configure and assemble the objects.
- Components are called "Beans" (can be any POJOs).
- Two types of IoC Containers: BeanFactory and **ApplicationContext** (superset of BeanFactory)
- Tasks:
  - to instantiate the application class.
  - to configure the objects (Spring Beans).
  - to assemble the dependencies between the objects.
  - to resolve messages / internationalization

```
<dependencies>
   <dependency>
      <groupid>org.springframework</groupid>
      <artifactid>spring-core</artifactid>
      <version>${spring.version}</version>
   </dependency>

   <dependency>
      <groupid>org.springframework</groupid>
      <artifactid>spring-context</artifactid>
      <version>${spring.version}</version>
   </dependency>
   ...
</dependencies>
```

The IoC Container

Interface: ApplicationContext

- Implementations:
  - FileSystemXmlApplicationContext
  - ClassPathXmlApplicationContext
  - AnnotationConfigWebApplication (implements WebApplicationContext)
  - XmlWebConfigWebApplication (implements WebApplicationContext)

```
public class App {

  public static void main( String[] args ) {

    ApplicationContext context =
              new ClassPathXmlApplicationContext("spring-config.xml");

    ...

  }
}
```

The IoC Container

What is a Spring Bean

- In Spring, **a bean** is an object that the Spring container instantiates, assembles, and manages.
- So should we configure all of the objects of our application as Spring beans?

  Well, as a best practice, we shouldn't.
- In general, we should define beans for ...
  - service layer objects
  - data access objects (DAOs)
  - presentation objects
  - infrastructure objects such as Hibernate SessionFactories
  - JMS Queues, and so forth

```java
@Repository
public class ItemDaoImpl implements ItemDao {

   @Autowired
   DataSource ds;

   ...
}
```

```
# application.properties
spring.datasource.driver-class-name=oracle.jdbc.OracleDriver
spring.datasource.url=jdbc:oracle:thin:@my-host:1522/my-service
spring.datasource.username=lasse
spring.datasource.password=secret

spring.datasource.hikari.poolName=HikariPoolBooks
spring.datasource.hikari.minimumIdle=5
spring.datasource.hikari.maximumPoolSize=5
```

```java
@Service
public class ItemService {
   private final Logger log = LoggerFactory.getLogger(ItemService.class);

   @Autowired
   ItemDao itemDao;

   ...
}
```

Dependency Injection

Core Spring

- Demo: demo-spring-framework.zip.
- Just some simple demo code to show technics used in Spring.

First some trivial code

```
1  public class OldCar {
2
3     private String color;
4     private Engine engine;
5
6     public OldCar(String color) {
7        System.out.println("Got a new " + color + " car");
8        this.color = color;
9        this.engine = new DieselEngine();
10    }
11
12    public void start() {
13          engine.turnOn();
14    }
15 }
```

First some trivial code

```
1  public class OldCar {
2
3     private String color;
4     private Engine engine;
5
6     public OldCar(String color) {
7        System.out.println("Got a new " + color + " car");
8        this.color = color;
9        this.engine = new DieselEngine();          # Dependency
10    }
11
12    public void start() {
13          engine.turnOn();
14    }
15 }
```

```java
public interface Engine {
   void turnOn();
}
```

```java
public class DieselEngine implements Engine {

   @Override
   public void turnOn() {
      System.out.println("Started diesel engine");
   }
}
```

```
public class App {

  public static void main( String[] args ) {

    ApplicationContext context =
              new ClassPathXmlApplicationContext("spring-config.xml");

      OldCar car1 = new OldCar("Blue");
      car1.start();
  }
}
```

Output:

```
Got a new blue car.
Started diesel engine
```

Let's remove the dependency

```
public class Car {

   private String color;
   private Engine engine;

   public Car(String color, Engine engine) {
      System.out.println("Got a new " + color + " car");
      this.color = color;
      this.engine = engine;
   }

   public void start() {
         engine.turnOn();
   }
}
```

Let's remove the dependency

```
public class Car {

   private String color;
   private Engine engine;

   public Car() {}

   public Car(String color, Engine engine) {
      System.out.println("Got a new " + color + " car");
      this.color = color;
      this.engine = engine;
   }

   public void start() {
         engine.turnOn();
   }

   // Getters and Setters
   ...
}
```

Let's invert the controll of initiating objects (IoC)

```
 1 public class App {
 2
 3    public static void main( String[] args ) {
 4       ApplicationContext context =
 5                     new ClassPathXmlApplicationContext("spring-config.xml");
 6
 7       OldCar car1 = new OldCar("Blue");
 8       car1.start();
 9
10       Car car2 = (Car) context.getBean("redFerrari");
11       car2.start();
12    }
13 }
```

Bean initialization **by constructor**

File: src/main/resources/spring-config.xml

```
<beans>

   <bean id="myEngine" class="no.hvl.dat152.entity.parts.DieselEngine"></bean>

   <bean name="redFerrari" class="no.hvl.dat152.entity.Car">
      <constructor-arg value="red"></constructor-arg>
      <constructor-arg><ref bean="myEngine"></ref></constructor-arg>
   </bean>

</beans>
```

Let's invert the controll of initiating objects (IoC)

```
 1 public class App {
 2
 3    public static void main( String[] args ) {
 4       ApplicationContext context =
 5                     new ClassPathXmlApplicationContext("spring-config.xml");
 6
 7       OldCar car1 = new OldCar("Blue");
 8       car1.start();
 9
10       Car car2 = (Car) context.getBean("redFerrari");
11       car2.start();
12    }
13 }
```

Output: (Why do we get this order?)

```
Got a new red car.
Got a new blue car.
Started diesel engine
Started diesel engine
```

Bean initialization **by propery**

File: src/main/resources/spring-config.xml

```
1  <beans>
2
3     <bean id="myEngine" class="no.hvl.dat152.entity.parts.DieselEngine"></bean>
4
5     <bean name="redFerrari" class="no.hvl.dat152.entity.Car">
6        <property name="color" value="red"></property>
7        <property name="engine" ref="myEngine"></property>
8     </bean>
9
10 </beans>
```

Output: (What happend to the output about the red car?)

```
Got a new blue car.
Started diesel engine
Started diesel engine
```

Let's introduce a new type of engine: Electrical

```
1 public class ElectricalEngine implements Engine {
2
3    @Override
4    public void turnOn() {
5          System.out.println("Started electric engine");
6    }
7
8 }
```

Bean initialization **by propery**

File: src/main/resources/spring-config.xml

```
<beans>

   <bean id="myEngine" class="no.hvl.dat152.entity.parts.ElectricalEngine">
   </bean>

   <bean name="redFerrari" class="no.hvl.dat152.entity.Car">
      <property name="color" value="red"></property>
      <property name="engine" ref="myEngine"></property>
   </bean>

</beans>
```

Output: (What happend to the output about the red car?)

```
Got a new blue car.
Started diesel engine
Started electric engine
```

Bean initialization **by Code**

Later: Configure beans with code

```
1  @Configuration
2  public class InternationalizationConfig implements WebMvcConfigurer {
3        @Bean
4        public LocaleResolver localeResolver() {
5           SessionLocaleResolver localeResolver = new SessionLocaleResolver();
6           localeResolver.setDefaultLocale(Locale.UK);
7           return localeResolver;
8        }
9
10       @Bean
11       public LocaleChangeInterceptor localeChangeInterceptor() {
12          LocaleChangeInterceptor localeChangeInterceptor = new LocaleChangeInt
13          localeChangeInterceptor.setParamName("lang");
14          return localeChangeInterceptor;
15       }
16
17       @Override
18       public void addInterceptors(InterceptorRegistry registry) {
19          registry.addInterceptor(localeChangeInterceptor());
20       }
21 }
```

The heart of the Spring Framework

Dependency Injection

- Programming technique .
  - Making a class independent of its dependencies.
- By decoupling the usage of an object from its creation.
- Dependent of 4 roles:
  - An **interface** that is used by the client and implemented by the service.
  - The **service/instance** you want to use.
  - The **client** that uses the service/instance.
  - The **injector** which creates a service instance and injects it into the client.

Sources

Dependency Injection

https://www.programmergirl.com/spring-dependency-injection/

https://www.baeldung.com/constructor-injection-in-spring

Spring Boot

An easier approach to the Spring Framework

An introduction to

Spring Boot

- Spring Boot is built on the top of the Spring framework (an extension).
- Features:
  - Standalone (Embedded server: No need for a Tomcat Web Server).
  - Opinionated (Prebuild configuration of dependencies).
  - Autoconfiguration.

Spring Boot

Provides a number of starter dependencies for different Spring modules.

- spring-boot-starter-web
- spring-boot-starter-data-jpa
- spring-boot-starter-security
- spring-boot-starter-test
- And several more

Maven Dependencies

The parent dependency

```
<parent>
    <groupid>org.springframework.boot</groupid>
    <artifactid>spring-boot-starter-parent</artifactid>
    <version>2.7.0</version>
    <relativepath></relativepath>  <!-- lookup parent from repository -->
</parent>
```

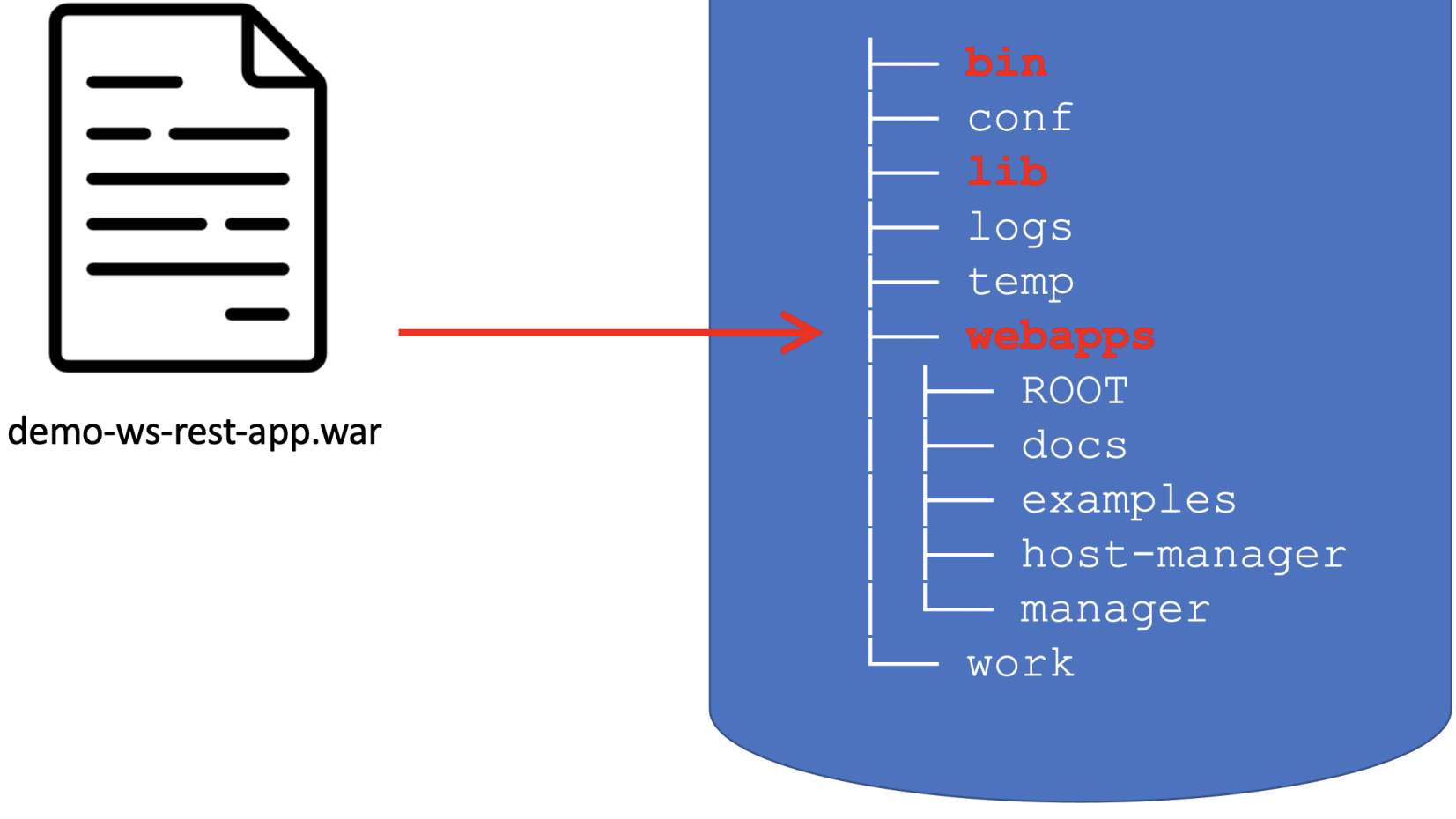
Maven Dependencies

Starter Dependencies

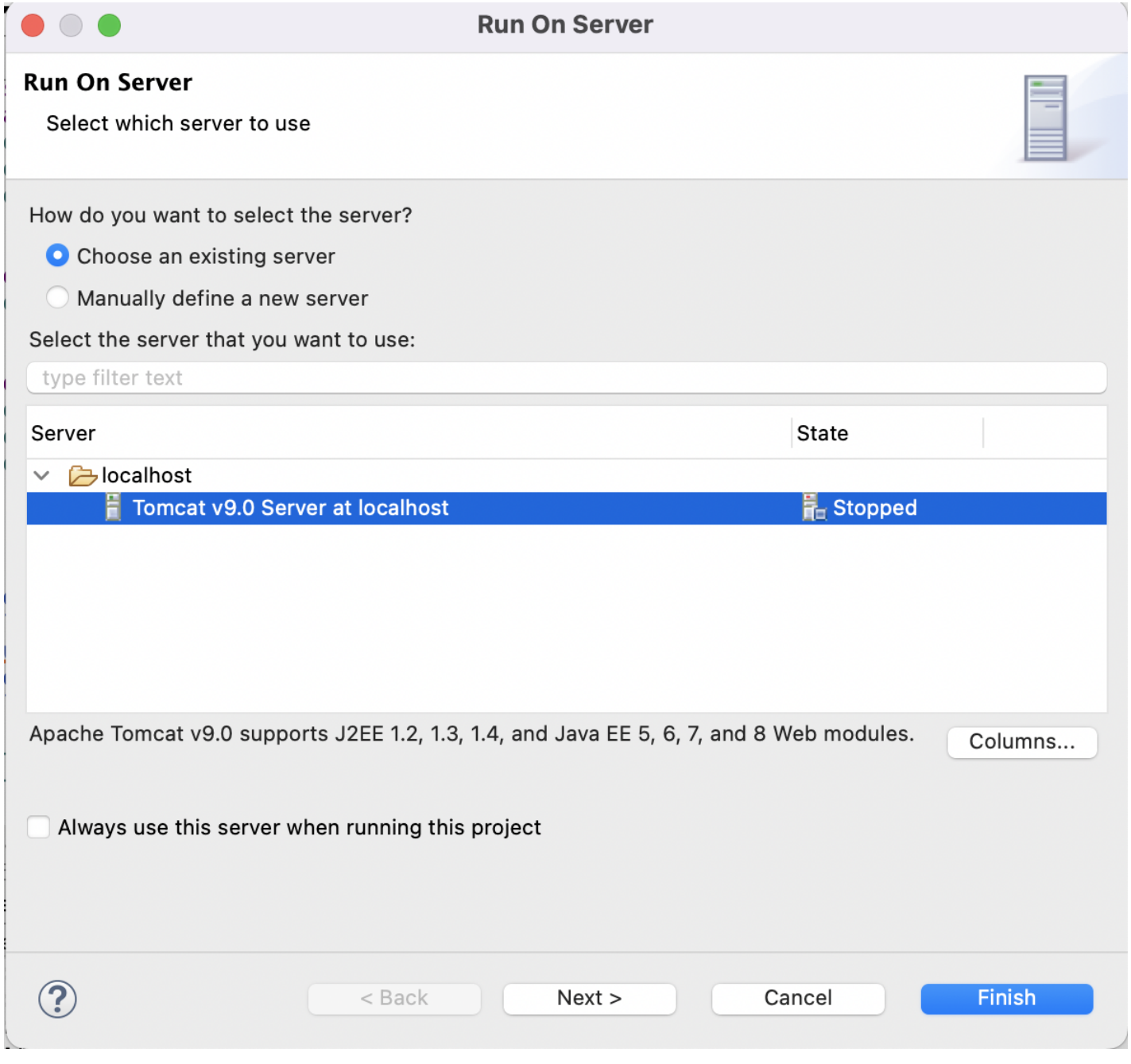
```
<dependency>
    <groupid>org.springframework.boot</groupid>
    <artifactid>spring-boot-starter-web</artifactid>
    <version>2.4.4</version>
</dependency>
```

Core Spring vs Spring Boot

External vs Embedded Server


demo-ws-rest-app.war
Tomcat EE
bin
conf
lib
logs
temp
webapps
ROOT
docs
examples
host-manager
manager
work

Applications					
Path	**Version**	**Display Name**	**Running**	**Sessions**	**Commands**
/	*None specified*	Welcome to Tomcat	true	0	Start Stop Reload Undeploy Expire sessions with idle ≥ 30 minutes
/demo-01-several-controllers	*None specified*	Demo 01 - Serveral Controllers	true	0	Start Stop Reload Undeploy Expire sessions with idle ≥ 30 minutes
/docs	*None specified*	Tomcat Documentation	true	0	Start Stop Reload Undeploy Expire sessions with idle ≥ 30 minutes
/examples	*None specified*	Servlet and JSP Examples	true	0	Start Stop Reload Undeploy Expire sessions with idle ≥ 30 minutes
/host-manager	*None specified*	Tomcat Host Manager Application	true	0	Start Stop Reload Undeploy Expire sessions with idle ≥ 30 minutes
/manager	*None specified*	Tomcat Manager Application	true	1	Start Stop Reload Undeploy Expire sessions with idle ≥ 30 minutes


Run On Server
Run On Server
Select which server to use
How do you want to select the server?
Choose an existing server
Manually define a new server
Select the server that you want to use:
type filter text
Server
State
localhost
Tomcat v9.0 Server at localhost
Stopped
Apache Tomcat v9.0 supports J2EE 1.2, 1.3, 1.4, and Java EE 5, 6, 7, and 8 Web modules.
Columns...
Always use this server when running this project
< Back
Next >
Cancel
Finish

External Web Servers

Pros

- Potentially more flexible application architecture.
- Really easy to switch servers later.
- Application errors can't harm the server.
- Easy to deploy app updates without restarting the server.

External Web Servers

Cons

- Extra performance overhead: there could be anything from an extra layer of method abstraction up to CGI-level overhead for your app and the server to communicate.
- Deployment complexity: you have to maintain the web server and the application, deploy them individually, ad hoc version testing, etc.
- Trickier development environment.

Spring Boot Default

Embedded Web Server

```
1 > mvn spring-boot:run
```

```
1 > mvn package
2 > java -cp target/demo-01-several-controllers-0.0.1-SNAPSHOT.jar /
3         no.hvl.dat152.Demo01JettyWebApp
```

Embedded Web Servers

Pros

- More self-contained applications. This helps a lot during development.
- As a dependency of your application, you can test against server versions just like any other dependency.
- More control over how the web server behaves (custom filters, headers, caching).
- Single object to be deployed.
- Easy to integrate with Docker, Kubernetes, OpenShift etc.

Embedded Web Servers

Cons

- Your application has to be designed around the API of whatever server you are using, making it harder to change servers later.
  (Java doesn't really have this problem, as you can still use the servlet API when embedding)

- Dependency bloat, as you have to include all the dependencies of the web server.
- More effort to deploy hotfixes to security exploits in the server.
- You can't group multiple applications behind one server without a proxy.
  (Not really an issue if deploying to virtual platform)

- A single uncaught exception is enough to take down the entire application server.

Next

Web Development: Using Spring Web MVC

Home