Ranga Rao Karanam
Mastering
Spring 5.0
A comprehensive guide to becoming an expert in the Spring Framework
This book is based on Spring Version 5.0 RC1
Packt

Contents

Chapter 1. Dependency Injection

Any Java class we write depends on other classes. The other classes a class depends on are its dependencies. If a class creates instances of other objects or dependencies, a tight coupling is established between the creating class and the class whose instance is being created. With Spring, the responsibility of creating and wiring objects is taken over by a new component called the IOC Container. Classes define dependencies and the Spring IOC (Inversion of Control) container creates objects and wires the dependencies together. This revolutionary concept where the control of creating and wiring dependencies is taken over by the container is famously called Inversion of Control or Dependency Injection.

In this chapter, we start with exploring the need for Dependency Injection. We use a simple example to illustrate how proper use of Dependency Injection reduces coupling and makes code testable. We would explore the Dependency Injection options in Spring. We will end the chapter looking at the standard Dependency Injection specification for Java - CDI (Contexts and Dependency Injection) and how Spring supports it.

This chapter would answer the following questions:

- What is Dependency Injection?
- How does proper use of Dependency Injection make applications testable?
- How does Spring implement Dependency Injection with Annotations?
- What is Component Scan?
- What is the difference between Java and XML application contexts?
- How do you create unit tests for Spring Contexts?
- How does mocking make unit testing simpler?
- What are the different bean scopes?
- What is CDI (Contexts and Dependency Injection) and how does Spring support CDI?

Dependency Injection Example

We will look at an example to understand Dependency Injection. We will write a simple business service talking to a data service. We will make the code testable and see how proper use of Dependency Injection makes the code testable.

Following is the sequence of steps we will follow:

1. Write a simple example of business service talking to a data service. When a business service directly creates an instance of a data service, they are tightly coupled to one another. Unit testing will be difficult
2. Make code loosely coupled by moving the responsibility of creating the data service outside the business service
3. Bring in Spring IOC Container to instantiate the beans and wire them together
4. Explore XML and Java configuration options Spring provides
5. Explore Spring unit testing options
6. Write real unit tests using mocking

Dependencies

We will start with writing a simple example: a business service talking to another data service. Most Java classes depend on other classes. Other classes that a class depends on are called **Dependencies** of that class. Consider an example class **BusinessServiceImpl**below:

```
public class BusinessServiceImpl {
  public long calculateSum(User user) {
    DataServiceImpl dataService = new DataServiceImpl();
    long sum = 0;
    for (Data data : dataService.retrieveData(user)) {
      sum += data.getValue();
    }
    return sum;
  }
}
```

Note

Typically, all well designed applications have multiple layers. Every layer has a well-defined responsibility. Business layer contains business logic. Data layer talks to the external interfaces and/or databases to get the data. In the above example `DataServiceImpl` gets some data related to the user from database. `BusinessServiceImpl`is a typical business service, talking to the data service `DataServiceImpl`for data and adds business logic on top of it (In this example, the business logic is very simple - calculate the sum of data returned by data service).

`BusinessServiceImpl`depends on `DataServiceImpl`. So `DataServiceImpl` is a dependency of `BusinessServiceImpl`.

Focus on how `BusinessServiceImpl`is creating an instance of `DataServiceImpl`.

```
    DataServiceImpl dataService = new DataServiceImpl();
```

`BusinessServiceImpl`creates an instance by itself. This is tight coupling.

Think for a moment about unit testing: How do you unit test class `BusinessServiceImpl` without involving (or instantiating) class `DataServiceImpl`? It's very difficult. One might need to do complicated things like reflection to write a unit test. So, above code is not testable.

Note

A piece of code (method, group of methods or a class) is testable when you can easily write a simple unit test for it without worrying about dependencies. One of the approaches used in unit testing is to mock the dependencies.(We will discuss a little more about mocking later)

Question to think about: How do we make above code testable? How do we reduce tight coupling between `BusinessServiceImpl`and`DataServiceImpl`?

First thing we can do is to create an interface for `DataServiceImpl`. Instead of using the direct class we can use the newly created interface of `DataServiceImpl` in `BusinessServiceImpl`.

Code below shows how to create an interface:

```
public interface DataService
{
  List<Data> retrieveData(User user);
}
```

Lets now update the code in `BusinessServiceImpl` to use the interface.

```
    DataService dataService = new DataServiceImpl();
```

Note

Using interfaces helps in creating loosely coupled code. We can replace the wire any implementation of an interface into a well-defined dependency. For example, consider a business service that needs some sorting.First option is to use the sorting algorithm directly in the code. For example: bubble sort. Second option is to create an interface for sorting algorithm and use the interface. Specific algorithm can be wired in later. In the first option, when we need to change the algorithm, we would need to change code. In the second option, all that we need to change is the wiring.

We are now using the interface `DataService` but `BusinessServiceImpl`is still tightly coupled as it is creating an instance of `DataServiceImpl`. How can we solve that?

How about `BusinessServiceImpl` not creating an instance of `DataServiceImpl` by itself? Can we create an instance of `DataServiceImpl` elsewhere (we will discuss who will create the instance later) and give it to

`BusinessServiceImpl`?

To enable this, we will update the code in `BusinessServiceImpl` to have a setter for `DataService`. Method `calculateSum` is also updated to use this reference. Updated code below:

```
public class BusinessServiceImpl
  {
     private DataService dataService;
     public long calculateSum(User user)
     {
       long sum = 0;
       for (Data data : dataService.retrieveData(user))
       {
         sum += data.getValue();
       }
       return sum;
     }

public void setDataService(DataService dataService)
   {
     this.dataService = dataService;
   }
  }
```

Note

Instead of creating a setter for DataService, we could have also created a `BusinessServiceImpl`constructor accepting DataService as an argument. This is called **Constructor Injection**.

You can see that `BusinessServiceImpl` can now work with any implementation of `DataService`. It is not tightly coupled with a specific implementation - `DataServiceImpl`.

To make the code even more loosely coupled (as we start writing the tests), lets create an interface for `BusinessService`and have the `BusinessServiceImpl`updated to implement the interface.

```
public interface BusinessService
{
  long calculateSum(User user);
}

public class BusinessServiceImpl implements BusinessService
{
  //.... Rest of code..
}
```

Now that we reduced coupling, one question still remains: Who takes the responsibility for creating instance of class `DataServiceImpl` and wiring it to class `BusinessServiceImpl`?

That's exactly where Spring IOC (Inversion of Control) Container comes into the picture.

Spring IOC Container

Spring IOC container creates the beans and wires them together.

Following questions need to be answered:

- **Question 1:** How does Spring IOC Container know which beans to create? Specifically, how does Spring IOC Container know to create beans for classes `BusinessServiceImpl` and `DataServiceImpl`?
- **Question 2:** How does Spring IOC Container know how to wire beans together? Specifically, how does Spring IOC Container know to inject the instance of class `DataServiceImpl` into class `BusinessServiceImpl`?
- **Question 3:** How does Spring IOC Container know where to search for beans? It is not efficient to search all packages in CLASSPATH.

Before we can focus on creating a container, lets focus on Questions 1 & 2: How to define what beans need to be created and how to wire them together?

Defining Beans and Wiring

- Lets address the first question: How does Spring IOC Container know which beans to create?

- We need to tell the Spring IOC Container which beans to create. This can be done using `@Repository` or `@Component` or `@Service` annotations on the classes for which beans have to be created. All these annotations tell the Spring framework to create beans for the specific classes where these annotations are defined.

Note

@Component is the most generic way of defining a Spring bean. Other annotations have more specific context associated with them. **@Service** is used in business service components. **@Repository** is used in DAO (Data Access Object) components.

- We use `@Repository` on the `DataServiceImpl` because it is related to getting data from database. We use `@Service` on the `BusinessServiceImpl` since it is a business service.

```
@Repository
public class DataServiceImpl implements DataService

@Service
public class BusinessServiceImpl implements BusinessServi
```

- Lets now shift our attention to Question 2: How does Spring IOC Container know how to wire beans together? Bean of class `DataServiceImpl` needs to be injected into that of class `BusinessServiceImpl`.
- We can do that by specifying an annotation `@Autowired` on the instance variable of interface `DataService` in class `BusinessServiceImpl`.

```
public class BusinessServiceImpl {
  @Autowired
  private DataService dataService;
```

- Now that we have defined the beans and their wiring, to test this we need an implementation of `DataService` . We will create a simple hard-coded implementation. `DataServiceImpl` returns a couple of pieces of data.

```
@Repository
public class DataServiceImpl implements DataService
{
  public List<Data> retrieveData(User user)
  {
    return Arrays.asList(new Data(10), new Data(20));
  }
}
```

- Now that we have our beans and dependencies defined, lets focus on how to create and run a Spring IOC Container.

Creating Spring IOC Container

- There are two ways to create a Spring IOC Container:
- Bean Factory
- Application Context

Note

Bean Factory is the basis for all Spring IOC (Inversion of Control) functionality - bean lifecycle and wiring. Application Context is basically a superset of Bean Factory with additional functionality typically needed in an enterprise context. Spring recommends using Application Context in all scenarios, except when the additional few Kilobytes of memory that Application Context consumes is critical.

- Lets use an Application Context to create a Spring IOC Container. We can either have a Java Configuration or XML Configuration for an Application Context. Lets start with using a Java Application Configuration.

Java Configuration for Application Context

Example below shows how to create a simple Java Context Configuration:

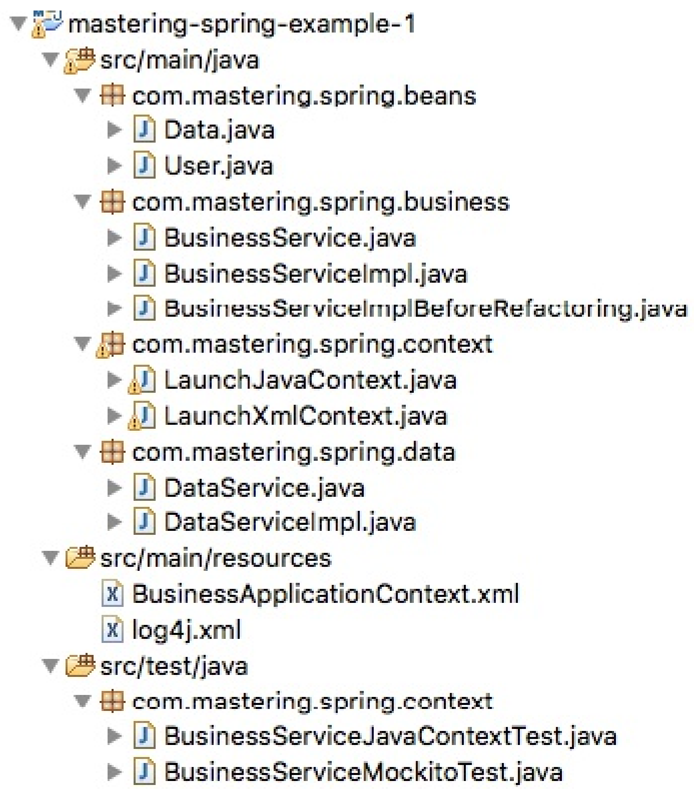
```
@Configuration
class SpringContext {
}
```

- Key is the annotation `@Configuration`. This is what defines this as a Spring configuration.
- One question remains: *How does Spring IOC Container know where to search for beans?*

We need to tell the Spring IOC Container the packages to search for by defining a component scan. Lets add a component scan to our earlier Java Configuration definition:

```
@Configuration
@ComponentScan(basePackages = { "com.mastering.spring" })
class SpringContext {
}
```

- We have defined a component scan for package **"com.mastering.spring"**. Below image shows how all the classes we discussed until now are organized. All the classes we have defined until now are present in this package.


```
mastering-spring-example-1
  src/main/java
    com.mastering.spring.beans
      Data.java
      User.java
    com.mastering.spring.business
      BusinessService.java
      BusinessServiceImpl.java
      BusinessServiceImplBeforeRefactoring.java
    com.mastering.spring.context
      LaunchJavaContext.java
      LaunchXmlContext.java
    com.mastering.spring.data
      DataService.java
      DataServiceImpl.java
  src/main/resources
    BusinessApplicationContext.xml
    log4j.xml
  src/test/java
    com.mastering.spring.context
      BusinessServiceJavaContextTest.java
      BusinessServiceMockitoTest.java
```

Quick Review

Lets take a moment and review all the things we have done until now to get this example working:

- We have defined a Spring Configuration class `SpringContext` with annotation `@Configuration` having a component scan for package `"com.mastering.spring"`.
- We have a couple of files (in the above package)
  - `BusinessServiceImpl` with annotation **@Service**
  - `DataServiceImpl` with annotation **@Repository**
- `BusinessServiceImpl` has `@Autowired` annotation on the instance of `DataService`.

When we launch up a Spring Context following things would happen:

- It would scan the package `"com.mastering.spring"`and find two beans `BusinessServiceImpl` and`DataServiceImpl`.
- `DataServiceImpl`does not have any dependency. So, the bean for

`DataServiceImpl` is created.

- `BusinessServiceImpl` has a dependency on `DataService`. `DataServiceImpl` is an implementation of the interface `DataService`. So, it matches the auto-wiring criteria. So, a bean for `BusinessServiceImpl` is created and the bean created for `DataServiceImpl` is auto wired to it through the setter.

Launch Application Context with Java Configuration

Below program shows how to launch up a Java Context: We use the main method to launch the application context using `AnnotationConfigApplicationContext`.

```
public class LaunchJavaContext
{

  private static final User DUMMY_USER = new User("dummy");

  public static Logger logger =
    Logger.getLogger(LaunchJavaContext.class);

  public static void main(String[] args)
  {
    ApplicationContext context = new
      AnnotationConfigApplicationContext(
      SpringContext.class);

    BusinessService service =
      context.getBean(BusinessService.class);
      logger.debug(service.calculateSum(DUMMY_USER));
  }
}
Following lines of code create the application context. We want t
    ApplicationContext context = new
              AnnotationConfigApplicationContext(
                  SpringContext.class);
```

Once the context is launched, we would need to get the business service bean. We use the `getBean` method passing the type of the bean (`BusinessService.class`) as an argument.

```
BusinessService service = context.getBean(BusinessService.class )
```

- We are now all set to launch the application context by running the program `LaunchJavaContext`.

Console Log

Below are some of the important statements from the log once the context is launched using `LaunchJavaContext`. Lets quickly review the log to get a deeper insight into what Spring is doing:

First lines show the component scan in action:

```
Looking for matching resources in directory tree [/target/classes
```

Spring now starts to create the beans. It starts with `businessServiceImpl` but it has an auto-wired dependency.

```
Creating instance of bean 'businessServiceImpl' Registered inject
```

Spring moves on to `dataServiceImpl` and creates an instance for it.

```
Creating instance of bean 'dataServiceImpl' Finished creating ins
```

Spring auto wires `dataServiceImpl` into `businessServiceImpl`.

```
Autowiring by type from bean name 'businessServiceImpl' to bean n
```

XML Configuration for Application Context

In the previous example, we used a Spring Java Configuration to launch an Application Context. Spring also supports XML Configuration. Below example shows how to launch an application context with XML Configuration. This would have two steps

- Defining XML Spring configuration
- Launch Application Context with XML configuration

Defining XML Spring Configuration

Example below shows a typical XML Spring configuration. This configuration file is created in the `src/main/resources` directory with name `BusinessApplicationContext.xml`.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<beans>  <!-Namespace definitions removed-->
  <context:component-scan base-package ="com.mastering.spring"/>
</beans>
```

- Component scan is defined using `context:component-scan`.

Launch Application Context with XML Configuration

```
Below program shows how to launch up an Application Context using
public class LaunchXmlContext {

  private static final User DUMMY_USER = new User("dummy");

  public static Logger logger = Logger.getLogger(LaunchJavaContex

  public static void main(String[] args) {
    ApplicationContext context = new ClassPathXmlApplicationConte
        "BusinessApplicationContext.xml");

    BusinessService service = context.getBean(BusinessService.cla
    logger.debug(service.calculateSum(DUMMY_USER));
  }
}
```

Following lines of code create the application context. We want to create an application context based on XML Configuration. So, we use a `ClassPathXmlApplicationContext` to create an application context. `AnnotationConfigApplicationContext`.

```
    ApplicationContext context = new
              ClassPathXmlApplicationContext (
                  SpringContext.class);
```

Once the context is launched, we would need to get a reference to the business service bean. This is very similar to how we did with Java Configuration. We use the `getBean` method passing the type of the bean (`BusinessService.class`) as an argument.

We can go ahead and run the class `LaunchXmlContext`. You will notice that we get very similar output to that we get when ran the Context with Java Configuration.

Writing JUnit using Spring Context

In the previous section(s), we look at how to launch up a Spring Context from main method. Now lets shift our attention to launching a Spring Context from a unit test.

We can use `SpringJUnit4ClassRunner.class` as a runner to launch up a spring context.

```
@RunWith(SpringJUnit4ClassRunner.class)
```

We would need to provide the location of Context Configuration. We will use the XML Configuration that we created earlier. Here's how you can declare that

```
@ContextConfiguration(locations = {
               "/BusinessApplicationContext.xml" })
```

We can auto wire a bean from the context into the test by using `@Autowired` annotation. BusinessService is auto wired by type.

```
  @Autowired
  private BusinessService service;
```

As of now the DataServiceImpl, which is wired in, returns `Arrays.asList(new Data(10), new Data(20))`. `BusinessServiceImpl`calculates the sum 10+20 and returns 30. We will assert for 30 in the test method using `assertEquals`.

```
    long sum = service.calculateSum(DUMMY_USER);
    assertEquals(30, sum);
```

Note

Why do we introduce unit testing so early in the book? Actually,

> we believe we are already late. Ideally, we would have loved to use Test Driven Development. Write tests before code. In my experience, doing TDD leads to simple, maintainable and testable code.Unit testing has a number of advantages: 1. Safety net against future defects 2. Defects are caught early 3. Following TDD leads to better Design 4. Well-written tests act as documentation of code and functionality - especially those written using BDD Given When Then style. The first test we write is not really a unit test. We would load up all the beans in this test. The next test, written using mocking, would be a real unit test - where the functionality being unit tested is the specific unit of code being written.

Complete list of the test is below: It has one test method.

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = { "/BusinessApplicationContext.
public class BusinessServiceJavaContextTest {
  private static final User DUMMY_USER = new User("dummy");

  @Autowired
  private BusinessService service;

  @Test
  public void testCalculateSum() {
    long sum = service.calculateSum(DUMMY_USER);
    assertEquals(30, sum);
  }

}
```

There is one problem with the JUnit that we wrote. It is not a true unit test. This test is using the real (almost) implementation of `DataServiceImpl` for the JUnit test. So, we are actually testing the functionality of both `BusinessServiceImpl` and `DataServiceImpl`. That's not unit testing.

Question now is: How to unit test `BusinessServiceImpl` without using a real implementation of `DataService`?

There are two options:

- Create a stub implementation of DataService providing some dummy data in the src\test\java folder. Use a separate test context configuration to auto-wire the stub implementation instead of the real DataServiceImpl.
- Create a mock of `DataService` and auto-wire the mock into `BusinessServiceImpl`.

Creating a stub implementation would mean creation of an additional class and an additional context. Stubs become more difficult to maintain, as we need more variations in data for the unit test.

In the next section, we will explore the second option of using a mock for unit testing. With the advances in mocking frameworks (especially Mockito) in the last few years, you would see that we would not even need to launch up a Spring Context to execute the unit test.

Unit Testing with Mocks

Lets start with understand what Mocking is: **Mocking** is creating objects that simulate the behavior of real objects. In the previous example, in the unit test, we would want to simulate the behavior of `DataService`.

Unlike stubs, mocks can be dynamically created at runtime. We will use the most popular mocking framework Mockito.

We would want to create a mock for `DataService`. There are multiple approaches to creating mocks with Mockito. Let's use the simplest among them - annotations. We use `@Mock` annotation to create a mock for `DataService`.

```
@Mock
private DataService dataService;
```

Once we create the mock, we would need to inject it into the class under test, `BusinessServiceImpl`. We do that using the `@InjectMocks`annotation.

```
@InjectMocks
private BusinessService service =
                              new BusinessServiceImpl();
```

In the test method, we would need to stub the mock service to provide the data that we want it to provide. There are multiple approaches. We will use the BDD style methods provided by Mockito to mock the `retrieveData` method.

```
    BDDMockito.given(
      dataService.retrieveData(
      Matchers.any(User.class)))
      .willReturn(
       Arrays.asList(new Data(10),
       new Data(15), new Data(25)));
```

What we are defining above is called stubbing. As with anything with Mockito, this is extremely readable. When `retrieveData` method is called on the `dataService` mock with any object of type `User`, return a list of 3 items with values specified.

```
When we use Mockito annotations, we would need use a specific JUn
@RunWith(MockitoJUnitRunner.class)
```

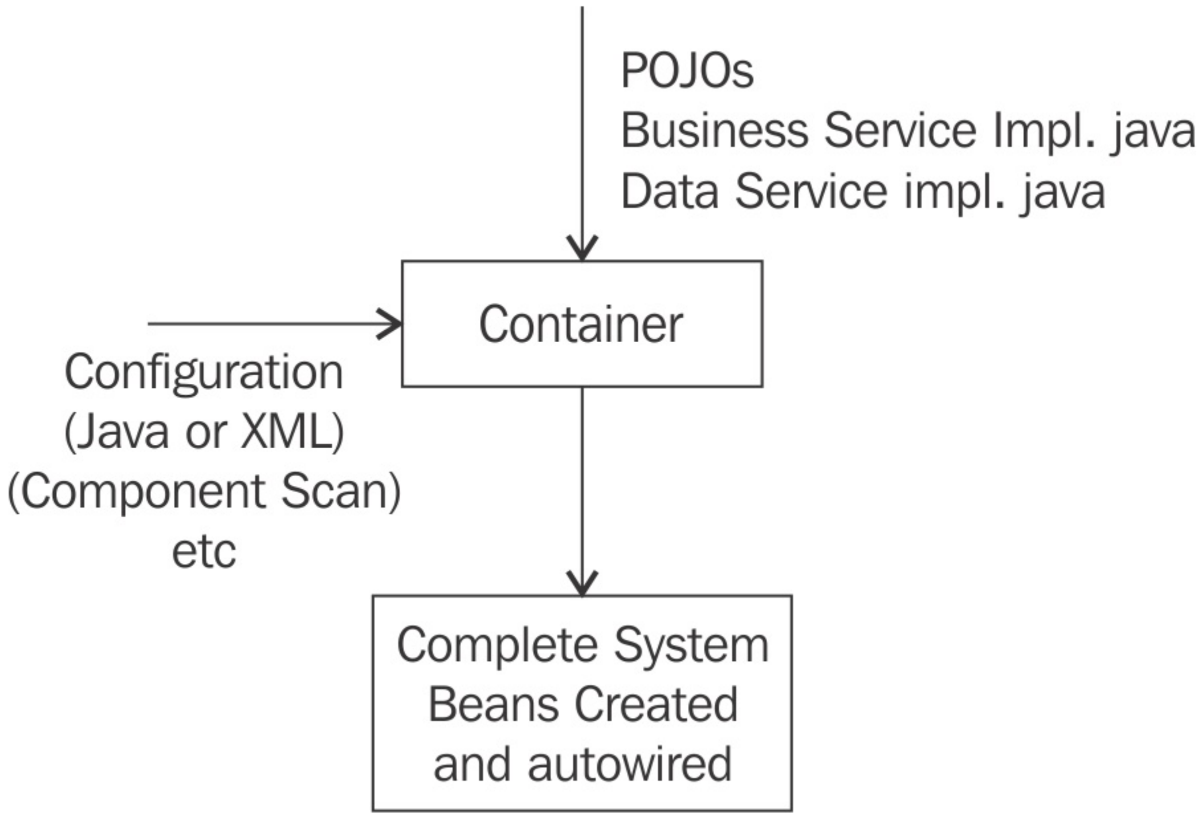
Complete list of the test is below: It has one test method.

```
 @RunWith(MockitoJUnitRunner.class)
 public class BusinessServiceMockitoTest {
   private static final User DUMMY_USER = new User("dummy");

   @Mock
   private DataService dataService;

   @InjectMocks
   private BusinessService service = new BusinessServiceImpl();

   @Test
   public void testCalculateSum()
   {
     BDDMockito.given(dataService.retrieveData(
       Matchers.any(User.class)))
       .willReturn(
        Arrays.asList(new Data(10),
        new Data(15), new Data(25)));

     long sum = service.calculateSum(DUMMY_USER);
```

```
        assertEquals(10 + 15 + 25, sum);
    }
}
```

Container Managed Beans

Instead of a class creating its own dependencies, in the earlier example, we looked at how Spring IOC Container can take over the responsibility of managing beans and their dependencies. The beans that are managed by Container are called Container Managed Beans.



Delegating creation and management of beans to container has many advantages. Few of them are listed below:

- Since classes are not responsible for creating dependencies, they are loosely coupled and testable. Good Design and lesser defects.
- Since container manages the beans, a few hooks around the beans can be introduced in a more generic way. Cross cutting concerns like Logging,

Caching, Transaction Management and Exception Handling can be woven around these beans using Aspect Oriented Programming. This leads to more maintainable code.

Dependency Injection Types

In the previous example, we used a setter method to wire in the dependency. There are two types of dependency injections frequently used:

- Setter Injection
- Constructor Injection

Setter Injection

Setter Injection is used to inject the dependencies through setter methods. In the example below, instance of `DataService` is using setter injection.

```
public class BusinessServiceImpl
{
  private DataService dataService;

  @Autowired
  public void setDataService(DataService dataService) {
    this.dataService = dataService;
  }
}
```

Actually, to use setter injection, you do not even need to declare a setter method. If you specify `@Autowired` on the variable, Spring automatically uses Setter Injection. So, the code below is all that you need to do setter injection for **DataService**.

```
public class BusinessServiceImpl {
@Autowired
  private DataService dataService;
}
```

Constructor Injection

Constructor Injection, on the other hand, uses a constructor for injecting dependencies. Below code shows using a constructor for injecting in the `DataService`.

```
public class BusinessServiceImpl {
  private DataService dataService;

  @Autowired
  public BusinessServiceImpl(DataService dataService) {
    super();
    this.dataService = dataService;
  }
}
```

When you run code with above implementation of BusinessServiceImpl, you would see this statement in the log asserting that auto-wiring took place using the constructor.

```
Autowiring by type from bean name 'businessServiceImpl' via const
```

Constructor vs. Setter Injection

Originally, in XML based application contexts, we use Constructor injection with mandatory dependencies and Setter injection with non-mandatory dependencies. However, an important thing to note is that when we use `@Autowired` on a field or a method, the dependency is by default required. If no candidates are available for an `@Autowired` field, auto-wiring fails and throws an exception. So, the choice is not so clear anymore with Java Application Contexts.

Using setter injection results in the state of the object changing during the creation. For fans of immutable objects, Constructor injection might be the way to go. Using setter injection might sometimes hide the fact that a class has a lot of dependencies. Using Constructor Injection makes it obvious - since the size of Constructor increases.

Spring Bean Scopes

Spring beans can be created with multiple scopes. The default scope is singleton.

Scope can be provided with the annotation @Scope on any spring bean.

```
@Service
@Scope("singleton")
public class BusinessServiceImpl implements BusinessService
```

Table below shows the different types of scopes available for beans:

Scope	Use
singleton	By default, all beans are of scope singleton. Only one instance of such beans is used per instance of Spring IOC Container. Even if there are multiple references to a bean, it is created only once per container. The single instance is cached and used for all subsequent requests using this bean. It is important to differentiate that Spring singleton scope is one object per one Spring Container. If you have multiple spring containers in a single JVM, then there can be multiple instances of same bean. So, Spring singleton scope is a little different from the typical definition of a singleton.
prototype	A new instance is created every time a bean is requested from the Spring Container. If a bean contains state, it is recommended to use prototype scope for it.
request	Available only in Spring web contexts. A new instance of bean is created for every HTTP request. Bean is discarded as soon as the request processing is done. Ideal for beans, which hold data specific to a single request.
session	Available only in Spring web contexts. A new instance of bean is created for every HTTP session. Ideal for data specific to a single

	user - like user permissions - in a web application.
application	Available only in Spring web contexts. One instance of bean per web application. Ideal for things like application configuration for a specific environment.

Java vs. XML Configuration

With the advent of annotations in Java 5, there is widespread use of Java Configuration for Spring based applications. What is the right choice to make if you have to choose between a Java based configuration as opposed to XML based configuration?

Spring provides equally good support for Java and XML based configuration. So, it's left to the programmer and his team to make the choice. Whichever choice is made, it is important to be having consistency across teams and projects. Here are some things you might need to consider when making a choice:

- Annotations lead to shorter and simpler bean definitions
- Annotations are more closer to the code they are applicable on than the XML based configuration
- Classes using Annotations are no longer simple POJOs because they are using framework specific annotations
- Auto-wiring problems when using Annotations might be difficult to solve because the wiring is no longer centralized and is not explicitly declared
- There might be advantages of more flexible wiring using spring context xml if it is packaged outside the application packaging - war or ear.

@Autowired annotation in depth

When @Autowired is used on a dependency, Application Context searches for a matching dependency. By default, all dependencies that are auto-wired are required.

Possible results are:

- One match is found: No problem that's the dependency you are looking for.
- More than one match is found: Auto-wiring fails
- No match is found: Auto-wiring fails

Cases where more than one candidate is found can be resolved in two ways:

- Use @Primary annotation to mark one of the candidates as the one to be used.
- Use @Qualifier to further qualify auto-wiring

@Primary annotation

When @Primary annotation is used on a bean it becomes the primary one to be used when there are more than one candidate is available for auto-wiring a specific dependency.

In the case of the example below, there are two sorting algorithms available: `QuickSort` and `MergeSort`. If the component scan finds both of them, `QuickSort` is used to wire any dependencies on `SortingAlgorithm` because of the `@Primary` annotation.

```
interface SortingAlgorithm {
}

@Component
class MergeSort implements SortingAlgorithm {
  // Class code here
}

@Component
@Primary
class QuickSort implements SortingAlgorithm {
  // Class code here
}
```

@Qualifier annotation

@Qualifier annotation can be used to give a reference to a spring bean. The reference can be used to qualify the dependency that needs to be auto-wired.

In the case of the example below, there are two sorting algorithms available : `QuickSort` and `MergeSort`. But `since @Qualifier("mergesort")` is used in the `SomeService` class, `MergeSort` which also has a qualifier `"mergesort"` defined on it becomes the candidate dependency selected for auto wiring.

```
@Component
@Qualifier("mergesort")
class MergeSort implements SortingAlgorithm {
  // Class code here
}

@Component
class QuickSort implements SortingAlgorithm {
  // Class code here
}

@Component
class SomeService {
  @Autowired
  @Qualifier("mergesort")
  SortingAlgorithm algorithm;
}
```

Other important Spring Annotations

Spring provides great deal of flexibility in defining beans and managing life cycle of a bean. There are few other important Spring annotations that we would discuss in the table below:

Annotations	Use
@ScopedProxy	Sometimes, we would need to inject a request or session scoped bean into a singleton-scoped bean. In such situations @ScopedProxy annotation provides a smart proxy to be

	injected into singleton scoped beans.
@Component, @Service, @Controller, @Repository	@Component is the most generic way of defining a Spring bean. Other annotations have more specific contexts associated with them. • @Service is used in business service layer • @Repository in data access (DAO) layer • @Controller in presentation components
@PostConstruct	On any spring bean, a post construct method can be provided by using the @PostConstruct annotation. This method is called once the bean is fully initialized with dependencies. This will be invoked only once during a bean lifecycle.
@PreDestroy	On any spring bean, a pre destroy method can be provided by using the @PreDestroy annotation. This method is called just before a bean is removed from the container. This can be used to release any resources that are held by the bean.

CDI (Contexts and Dependency Injection)

CDI is Java EE's attempt to bring Dependency Injection into Java EE. While not as full fledged as Spring, CDI aims to standardized basics of how Dependency Injection is done. Spring supports the standard annotations defined in JSR-330. For the most part, these annotations are treated the same way as Spring annotations.

Before we can use CDI, we would need to ensure that we have dependencies for CDI jars included. Here's the code snippet:

```
<dependency>
    <groupId>javax.inject</groupId>
    <artifactId>javax.inject</artifactId>
    <version>1</version>
</dependency>
```

In this table, lets compare the CDI annotations with the annotations provided by Spring framework. It should be noted that @Value, @Required, @Lazy Spring annotations have no equivalent CDI annotations.

CDI Annotation	Comparison with Spring Annotations
@Inject	Similar to @Autowired. One insignificant difference is the absence of required attribute on @Inject.
@Named	@Named is similar to @Component. Identifies named components. In addition, @Named can be used to qualify the bean with a name - similar to @Qualifier spring annotation. This is useful in situations when multiple candidates are available for auto-wiring one dependency.
@Singleton	Is similar to Spring annotation @Scope("singleton").
@Qualifier	Similar to similarly named annotation in Spring - @Qualifier

CDI Example

When we use CDI, this is how the annotations on the different classes would look like. There is no change in how we create and launch Spring Application Context.

CDI has no differentiation between `@Repository`, `@Controller`, `@Service` and `@Component`. We use `@Named` instead of all the above annotations.

In the example, we use `@Named` for `DataServiceImpl` and `BusinessServiceImpl`. We use `@Inject` to inject the `dataService` into `BusinessServiceImpl` (instead of `@Autowired`).

```
@Named //Instead of @Repository
public class DataServiceImpl implements DataService

@Named //Instead of @Service
  public class BusinessServiceImpl {
    @Inject //Instead of @Autowired
    private DataService dataService;
```

Summary

Dependency Injection (or Inversion of Control) is the key feature of Spring. It makes code loosely coupled and testable. Understanding Dependency Injection is the key to making best use of Spring Framework.

In this chapter, we took a deep look at Dependency Injection and the options Spring framework provides. We also looked at examples of writing testable code and wrote a couple of unit tests.

In the next chapter, we will shift our attention towards Spring MVC - the most popular Java web MVC framework. We will understand how Spring MVC makes developing web applications easier.

Chapter 2. Building Web Application with Spring MVC

Spring MVC is the most popular web framework for developing Java web applications. Beauty of Spring MVC lies in its clean, loosely coupled architecture. With clean definition of roles for Controllers, Handler Mappings, View Resolvers and POJO (Plain Old Java Object) command beans; Spring MVC makes it simple to create web applications. With its support of multiple view technologies, it is extensible too.

While Spring MVC can be used to create REST Services, we would discuss that in subsequent chapter dealing with Micro-services. In this chapter, we would focus on reviewing the basics of Spring MVC with simple examples.

This chapter will cover the following topics:

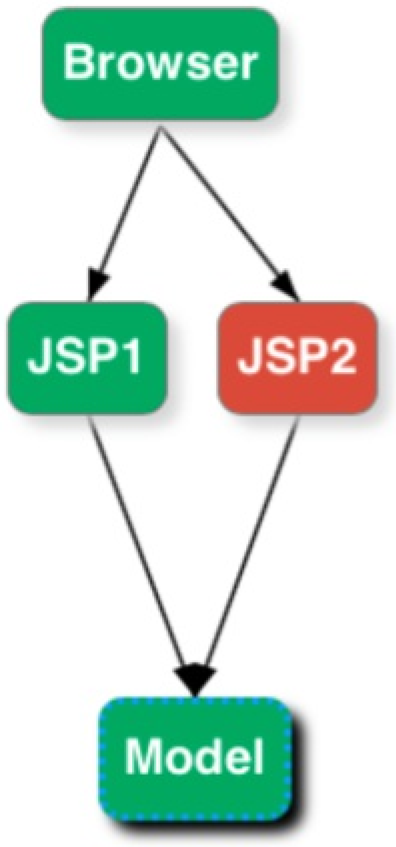
- Spring MVC Architecture
- Role played by Dispatcher Servlet, View Resolvers, Handler Mappings and Controllers.
- Model Attributes and Session Attributes
- Form Binding and Validation
- Integration with Bootstrap
- Basics of Spring Security
- Writing Simple Unit Tests for Controllers.

Java Web Application Architecture

How we develop java web applications has evolved during the last couple of decades. We will discuss the different architectural approaches to developing Java Web Applications and see where Spring MVC fits in:

- Model 1 Architecture
- Model 2 or MVC Architecture
- Model 2 with Front Controller

Model 1 Architecture



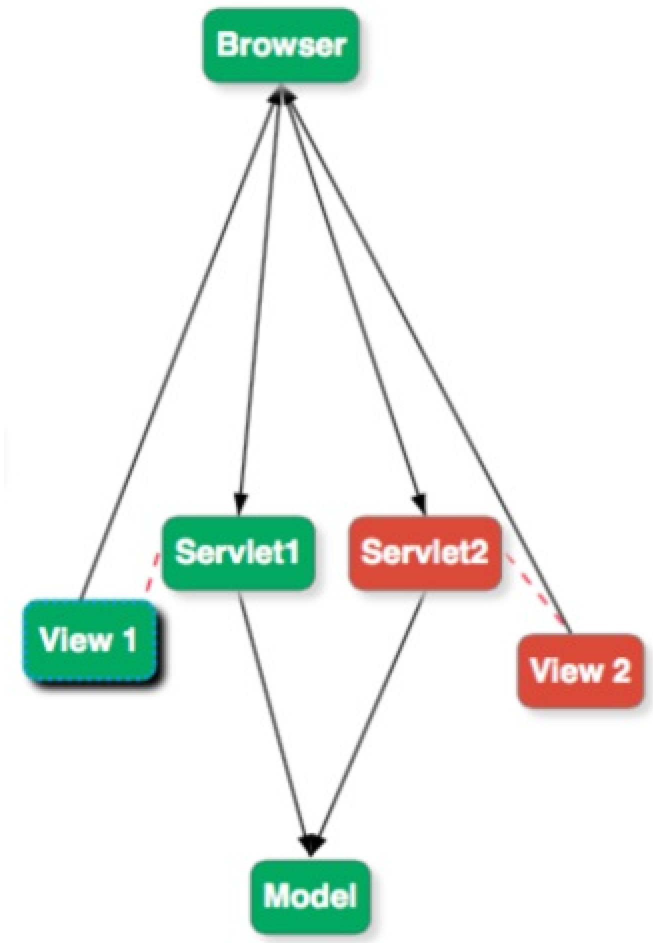
Model 1 architecture is one of initial architecture styles used to develop Java based web applications. A few important details:

- JSP pages directly handled the requests from browser.
- JSP pages made use of model containing simple java beans.
- In some applications of this architecture style, JSPs even performed queries to the database.
- JSPs also handled the flow logic - which page to show next

There are a lot of disadvantages in this approach leading to quick shelving and evolution of other architectures. A few important ones are listed below:

- Hardly any separation of concerns: JSPs were responsible for retrieving data, displaying data, deciding which pages to show next (flow) and some times even business logic as well.
- Complex JSPs: Because JSPs handled a lot of logic, they were huge and difficult to maintain.

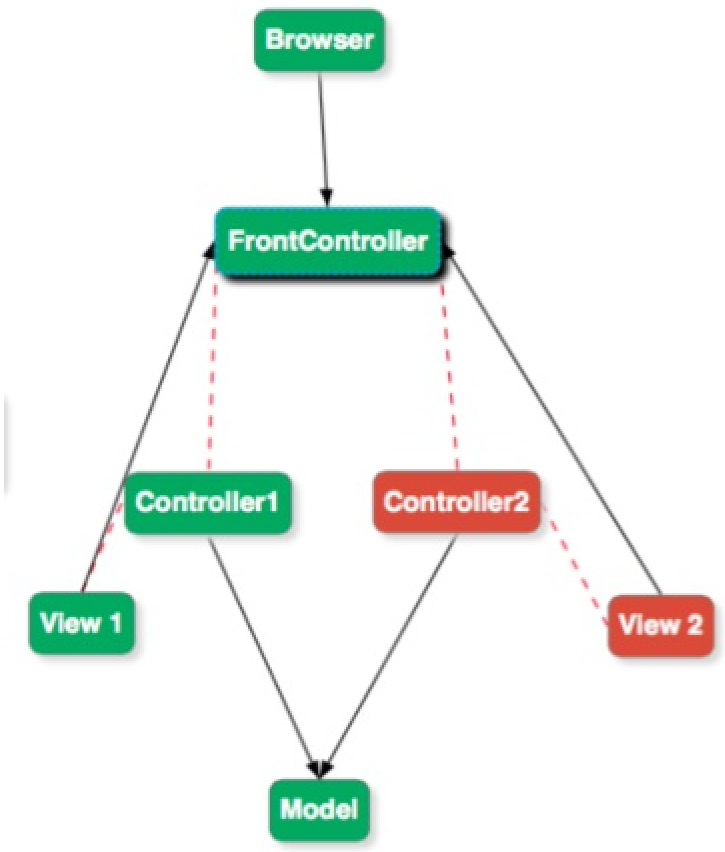
Model 2 Architecture



Model 2 architecture came in to solve the complexity involved with complex JSPs having multiple responsibilities. This forms the base for MVC architecture style. Model 2 architecture has clear separation of roles between Model, View and Controller. This leads to more maintainable applications. A few important details:

- **Model**: Represents the data to be used to generate a View.
- **View**: View uses the Model to render the screen.
- **Controller**: Controls the flow. Gets the request from browser, populates the model and redirects to the View. Examples: Servlet1, Servlet2 in the picture above.

Model 2 Front Controller Architecture



In the basic version of Model 2 architecture, the requests from browser are handler directly by different servlets (or controllers). In a number of business scenarios, one would want to do a few common things in servlets before we handle the request. An example would be to ensure that the logged in user has right authorization to execute the request. This is common functionality that you would not want to be implemented in every servlet.

In Model 2 Front Controller Architecture, all requests flow into a single controller called the Front Controller.

Following are some of the responsibilities of a typical Front controller

- Decides which controller executes the request
- Decides which view to render
- Provides provisions to add more common functionality
- Spring MVC uses a MVC pattern with Front Controller. The Front Controller is called DispatcherServlet. We will discuss about Dispatcher

Servlet a little later.

Basic Flows

Spring MVC uses a modified version of the Model 2 Front Controller architecture. Before we go into details about how Spring MVC works, we will focus on creating a few simple web flows using Spring MVC. In this section, we would create 6 typical web application flows using Spring MVC. The flows are listed below:

- Flow 1: Controller without a view - Serving content on its own
- Flow 2: Controller with a view (a JSP)
- Flow 3: Controller with a view and using ModelMap
- Flow 4: Controller with a view and using ModelAndView
- Flow 5: Controller for a Simple Form
- Flow 6: Controller for a Simple Form with Validation

At the end of every flow we would discuss how to unit test the controller.

Flow 0: Setup

Before we start with the first flow, we would need to get the application setup to use Spring MVC. In the next section, we would start with understanding how to setup Spring MVC in a web application.

We are using Maven to manage our dependencies. Following steps are involved in setting up a simple web application:

1. Add a dependency for Spring MVC
2. Add DispatcherServlet to web.xml
3. Create a Spring Application Context

Dependency For Spring MVC

Lets start with adding the Spring MVC dependency to our pom.xml. Below code shows the dependency to be added in. Since we are using Spring BOM we do not need to specify the artifact version.

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

Dispatcher Servlet is an implementation of Front Controller pattern. Any request to Spring MVC will be handled by the front controller i.e. Dispatcher Servlet.

Adding Dispatcher Servlet to web.xml

To enable this, we would need to add Dispatcher Servlet to web.xml. Lets see how to do it below:

```
<servlet>
   <servlet-name>spring-mvc-dispatcher-servlet</servlet-name>
   <servlet-class>
   org.springframework.web.servlet.DispatcherServlet
   </servlet-class>
   <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>/WEB-INF/user-web-context.xml</param-value>
   </init-param>
   <load-on-startup>1</load-on-startup>
</servlet>

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

First part is to define a servlet. We are also defining a context configuration location`/WEB-INF/user-web-context.xml`. This is where we would define a spring context in the next step. In the second part, we are defining a servlet mapping. We are mapping a url `/`to the dispatcher servlet. So, all requests would be handled by the dispatcher servlet.

Creating Spring Context

Now that we have dispatcher servlet defined in web.xml, we can go ahead and create our Spring Context. Initially we would create a very simple context without really defining anything concrete.

```
<beans > <!-Schema Definition removed -->

  <context:component-scan
   base-package="com.mastering.spring.springmvc"  />

  <mvc:annotation-driven />

</beans>
```

We are defining a component scan for the package `com.mastering.spring.springmvc` so that all the beans and controllers in this package are created and auto-wired.

Using `<mvc:annotation-driven/>`initializes support for a number of features that Spring MVC supports

- Request Mapping
- Exception Handling
- Data Binding and Validation
- Automatic conversion (for example of JSON) when @RequestBody annotation is used

That's all the setup we need to be able to setup a Spring MVC application. We are ready to get started with first flow.

Flow 1: Simple Controller flow without view

Lets start with a simple flow. Showing some simple text that is output from a Spring MVC Controller on the screen.

Creating a Spring MVC Controller

Lets create a simple Spring MVC Controller.

```
@Controller
```

```
public class BasicController
{
   @RequestMapping(value = "/welcome")
   @ResponseBody
   public String welcome()
   {
     return "Welcome to Spring MVC";
   }
}
```

A few important things to note:

- `@Controller` annotation: It defines a Spring MVC Controller that can contain Request Mappings - Mapping URLs to Controller methods.
- `@RequestMapping(value = "/welcome")` annotation: Defines a mapping of url `/welcome` to the `welcome` method. When browser sends a request to `/welcome` Spring MVC does the magic and executes the `welcome` method.
- `@ResponseBody` annotation: In this specific context, the text returned by `welcome` method is sent out to the browser as the response content. `@ResponseBody`does a lot of magic - especially in the context of REST Services. We will discuss that later in the chapter on REST Services.

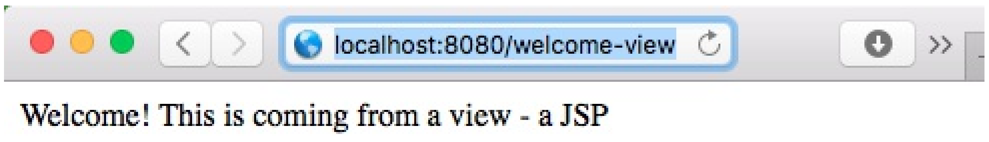
Running the web application

We are using Maven and Tomcat 7 to run this web application. Refer to the appendix to understand how this is setup.

Tomcat 7 server by default launches up on port 8080.

We can run the server by invoking the command `mvn tomcat7:run`.

Here is a screenshot of how this would look on the screen when the URL `http://localhost:8080/welcome` is hit on the browser:



Welcome! This is coming from a view - a JSP

Unit Testing

Unit testing is very important part of developing maintainable applications. We will be use Spring MVC Mock framework to unit test the controllers that we would write during this chapter. We will add in a dependency on spring test framework to use Spring MVC Mock framework.

```
<dependency>
   <groupId>org.springframework</groupId>
   <artifactId>spring-test</artifactId>
   <scope>test</scope>
</dependency>
```

Approach we would be taking would involve

1. Setting up the Controller to test
2. Writing the test method

Setting up the Controller to test

The controller we would want to test is `BasicController`. The convention for creating unit test is class name with a suffix Test. We will create a test class named `BasicControllerTest`.

The basic setup is shown below:

```
public class BasicControllerTest
{
  private MockMvc mockMvc;
  @Before
  public void setup()
    {
      this.mockMvc = MockMvcBuilders.standaloneSetup(
        new BasicController())
        .build();
    }
}
```

A few important things to note:

- `mockMvc` instance variable: This variable can be used across different tests. So, we define an instance variable of class **MockMvc**.
- `@Before setup` method: This method is run before every test to initialize mockMvc.

`MockMvcBuilders.standaloneSetup(new BasicController()).build():` This line of code builds a MockMvc instance. This initializes DispatcherServlet to serve requests to the configured controller(s), **BasicController** in this instance.

Writing Test Method

Complete test method is shown below:

```
@Test
public void basicTest() throws Exception
{
  this.mockMvc
    .perform(
    get("/welcome")
    .accept(MediaType.parseMediaType
    ("application/html;charset=UTF-8")))
    .andExpect(status().isOk())
    .andExpect( content().contentType
    ("application/html;charset=UTF-8"))
    .andExpect(content().
  string("Welcome to Spring MVC"));
}
```

A few important things to note:

- `mockMvc.perform` method: Executes the request and returns an instance of ResultActions that allows chaining calls. In this example, we are chaining the **andExpect** calls to check expectations.
- `get("/welcome").accept(MediaType.parseMediaType("application/8"))`: Create a HTTP get request accepting a response with media type "application/html"
- `andExpect` method : Used to check expectations. This method would fail the test if the expectation is not met
- `status().isOk()` : Result Matcher to check if the response status is that of a successful request - 200
- `content().contentType("application/html;charset=UTF-8"))` : Result Matcher to check if the content type of the response is as specified
- `content().string("Welcome to Spring MVC")`: Result Matcher to check if the response content contains the specified string.

Flow 2: Simple Controller flow with a view

In the previous flow, the text to show on the browser was hardcoded in the Controller. That is not a good practice. Content to be shown on the browser is typically generated from a view. Most frequently used option is a JSP.

In this flow, lets redirect from the Controller to a view.

Spring MVC Controller

Similar to the previous example lets create a simple Controller. Consider the example controller below:

```
  @Controller
public class BasicViewController {
@RequestMapping(value = "/welcome-view")
public String welcome() {
return "welcome";
}
}
```

A few important things to note:

- `@RequestMapping(value = "/welcome-view")`: **We are mapping a url /welcome-view.**
- `public String welcome()`: **There is no @RequestBody annotation on this method. So, Spring MVC tries to match the string that is returned "welcome" to a view.**

Let's create a View - a JSP

Lets create a `welcome.jsp` in the folder "src/main/webapp/WEB-INF/views/welcome.jsp" with following content.

```
<html>
<head>
<title>Welcome</title>
</head>
<body>
<p>Welcome! This is coming from a view - a JSP</p>
</body>
</html>
```

This is simple html with head, body and some text in the body.

Spring MVC has to map the string returned from welcome method - `"welcome"`- to the real JSP at "`/WEB-INF/views/welcome.jsp`". How does this magic happen?

View Resolver

A View Resolver resolves a view name to the actual JSP page.

The view name in this example is `welcome.jsp` and we would want it to resolve to `/WEB-INF/views/welcome.jsp`.

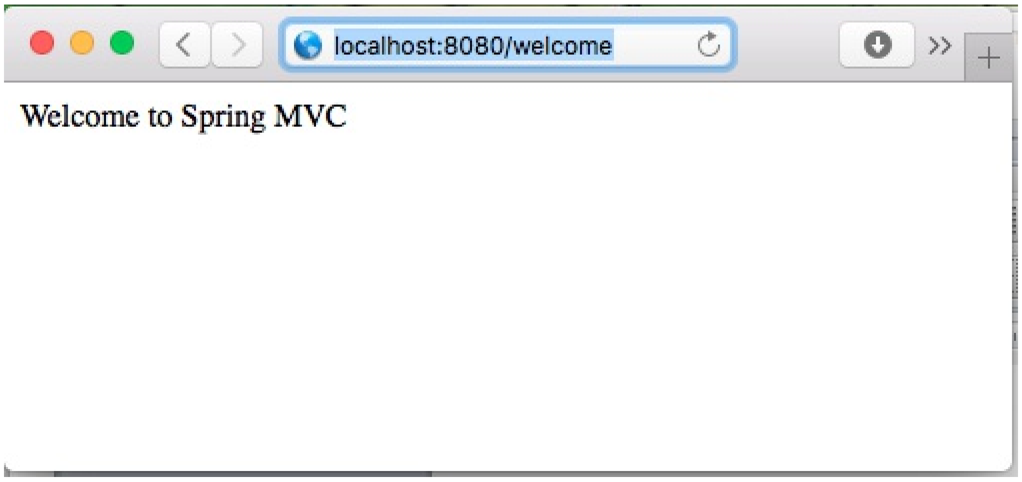
A view resolver can be configured in the spring context `/WEB-INF/user-web-context.xml`. Here's the snippet:

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

```
</property>
<property name="suffix">
<value>.jsp</value>
</property>
</bean>
```

- `org.springframework.web.servlet.view.InternalResourceViewReso` View Resolver supporting JSPs. JstlView is typically used. Also supports tiles with a TilesView.
- `<property name="prefix"> <value>/WEB-INF/views/</value> </property><property name="suffix"> <value>.jsp</value> </property>`: Maps the prefix and suffix to be used by view resolver. View resolver takes the string from controller method and resolves to the view : prefix + viewname + suffix. So, view name welcome is resolved to /WEB-INF/views/welcome.jsp

Here is a screenshot of how this would look on the screen when the URL is hit:



Welcome to Spring MVC

Unit Testing

Stand-alone setup of Mock Mvc Framework creates the bare minimum infrastructure required by the DispatcherServlet. If provided with a View Resolver, it can execute view resolution. However, it would not execute the view. So, during a unit test with Stand-alone setup, we cannot verify the content of the view. However, we can check if the correct view is being delivered.

In this unit test, we want to setup BasicViewController, execute a get request to "/welcome-view" and check if the view name returned is "welcome". In a future section, we will discuss how to execute the integration test, including the rendering of view. As far as this test is concerned, we restrict our purview to verifying the view name.

Setting up the Controller to test

This step is very similar to the previous flow. We would want to test BasicViewController. We instantiate MockMVC using BasicViewController. We also would configure a simple view resolver.

```
public class BasicViewControllerTest
{
  private MockMvc mockMvc;

  @Before
  public void setup()
  {
    this.mockMvc = MockMvcBuilders.standaloneSetup
      (new BasicViewController())
      .setViewResolvers(viewResolver()).build();
  }

  private ViewResolver viewResolver()
  {
    InternalResourceViewResolver viewResolver =
      new InternalResourceViewResolver();

    viewResolver.setViewClass(JstlView.class);
    viewResolver.setPrefix("/WEB-INF/jsp/");
    viewResolver.setSuffix(".jsp");

    return viewResolver;
  }
}
```

Writing Test Method

Complete test method is shown below:

```
@Test
```

```
public void testWelcomeView() throws Exception {
  this.mockMvc
  .perform(get("/welcome-view")
  .accept(MediaType.parseMediaType(
  "application/html;charset=UTF-8")))
  .andExpect(view().name("welcome"));
}
```

A few important things to note:

- `get("/welcome-model-view")`: Execute get request to specified URL.
- `view().name("welcome")` : Result matcher to check if view name returned is as specified.

Flow 3: Controller redirecting to a View with Model

Typically to generate the view, we would need to pass some data to it. In Spring MVC, data can be passed to the view using a model. In this flow, we would set up a model with a simple attribute and use the attribute in the view.

Spring MVC Controller

Lets create a simple Controller. Consider the example controller below:

```
@Controller
public class BasicModelMapController {
  @RequestMapping(value = "/welcome-model-map")
  public String welcome(ModelMap model) {
    model.put("name", "XYZ");
    return "welcome-model-map";
  }
}
```

A few important things to note:

- `@RequestMapping(value = "/welcome-model-map")`: URI mapped is /welcome-model-map
- `public String welcome(ModelMap model)`: The new parameter added in is "ModelMap model". Spring MVC would instantiate a model and make it available for this method. Attributes put into the model will be

available to use in the view
- `model.put("name", "XYZ")`: Adding an attribute with name "name" and value "XYZ" to the model

Creating a View

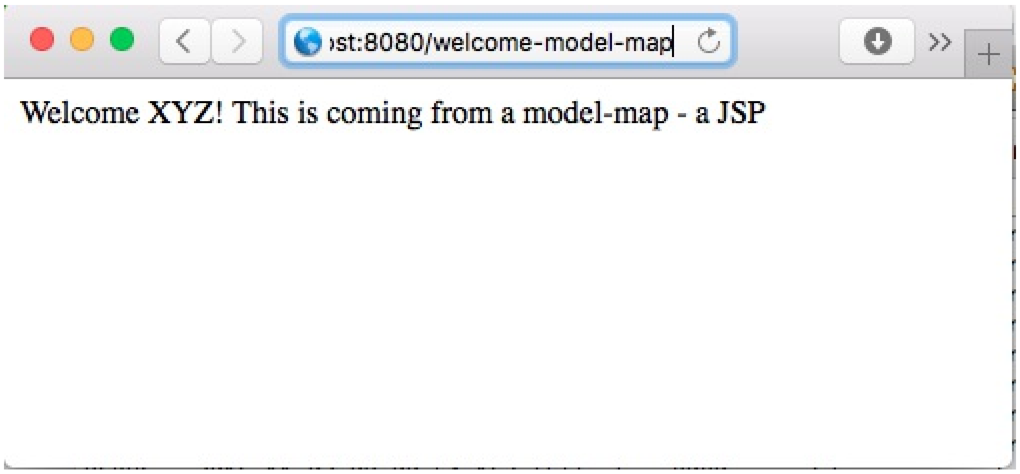
Lets create view using the model attribute "name" that was set into the model in the controller.: Let's create a simple jsp in the path `WEB-INF/views/welcome-model-map.jsp`.

```
Welcome ${name}! This is coming from a model-map - a JSP
```

Note

One thing to note: `${name}`: This uses EL (Expression Language) syntax to access the attribute from the model.

Here is a screenshot of how this would look on the screen when the url is hit:



Welcome XYZ! This is coming from a model-map - a JSP

Unit Testing

In this unit test, we want to setup `BasicModelMapController`, execute a get request to "`/welcome-model-map`" and check if the model has the expected attribute and expected view name is returned.

Setting up the Controller to test

This step is very similar to the previous flow. We instantiate Mock MVC with `BasicModelMapController`.

```
this.mockMvc = MockMvcBuilders.standaloneSetup(new BasicModelMapC
```

Writing Test Method

Complete test method is shown below:

```
@Test
public void basicTest() throws Exception {
this.mockMvc
  .perform(get("/welcome-model-map")
  .accept(MediaType.parseMediaType("application/html;charset=UTF-
  .andExpect(model().attribute("name", "XYZ"))
  .andExpect(view().name("welcome-model-map"));
}
```

A few important things to note:

- `get("/welcome-model-map")`: Execute get request to specified URL.
- `model().attribute("name", "XYZ")`: Result matcher to check if model contains specified attribute **"name"** with specified value **"XYZ"**.
- `view().name("welcome-model-map")`: Result matcher to check if view name returned is as specified.

Flow 4: Controller redirecting to a View with ModelAndView

In the previous flow, we returned a view name and populated the model with attributes to be used in the view. Spring MVC provides an alternate approach using ModelAndView. The controller method can return a ModelAndView object with view name and appropriate attributes in the Model. In this flow, we will explore this alternate approach.

Spring MVC Controller

Consider the controller below:

```
@Controller
public class BasicModelViewController {
   @RequestMapping(value = "/welcome-model-view")
   public ModelAndView welcome(ModelMap model)
   {
     model.put("name", "XYZ");
     return new ModelAndView("welcome-model-view", model);
   }
}
```

A few important things to note:

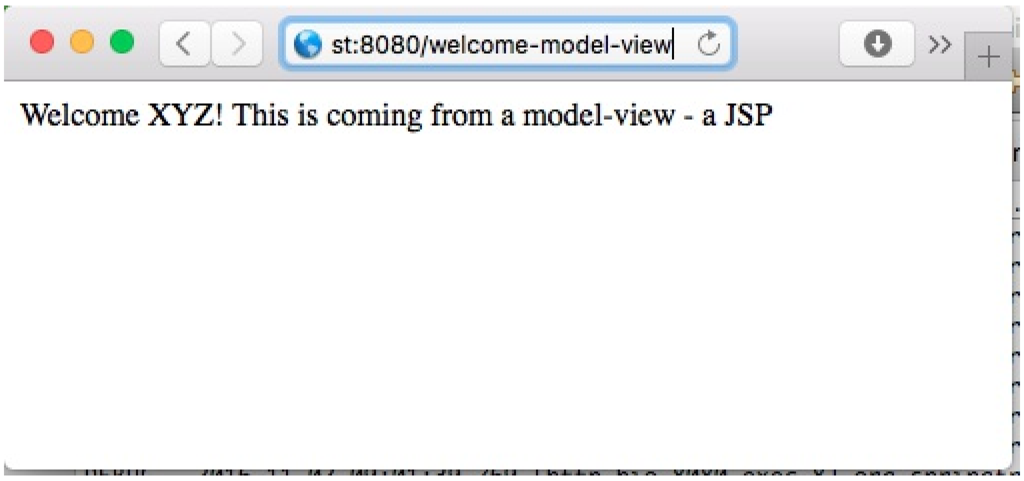
- `@RequestMapping(value = "/welcome-model-view")`: URI mapped is /welcome-model-view
- `public ModelAndView welcome(ModelMap model)`: Notice that the return value is no longer a String. It is a ModelAndView.
- `return new ModelAndView("welcome-model-view", model)`: Create a ModelAndView object with appropriate view name and the model.

Creating a View

Lets create view using the model attribute "name" that was set into the model in the controller.: Let's create a simple jsp in the path `/WEB-INF/views/welcome-model-view.jsp`.

```
Welcome ${name}! This is coming from a model-view - a JSP
```

Here is a screenshot of how this would look on the screen when the url is hit:



Welcome XYZ! This is coming from a model-view - a JSP

Unit Testing

Unit testing for this flow is skipped, as it is very similar to the previous flow.

Flow 5: Controller redirecting to a View with a Form

Lets now shift our attention to creating a simple form to capture input from user.

Following steps will be needed.

- Create a Simple POJO. We want to create a user. We will create a POJO User.
- Create couple of Controller methods: One to display the form. Other to capture the details entered in the form.
- Create a simple view with the form.

Command or Form Backing Object: A simple POJO

We will create a simple POJO to act as a command object. Important parts of the class are listed below:

```
public class User {
   private String guid;
   private String name;
   private String userId;
   private String password;
   private String password2;

   //Constructor
   //Getters and Setters
   //toString
}
```

A few important things to note:

- This class does not have any annotations or Spring related mappings. Any bean can act as a form-backing object.
- We are going to capture name, user id and password in the form. We have a password confirmation field password2 and unique identifier field guid.
- Constructor, getters, setters and toString methods are not shown for brevity

Controller Get Method to show the form

Lets start with creating a simple controller with a logger:

```
@Controller
public class UserController {
   private Log logger = LogFactory.getLog
(UserController.class);
}
```

Lets add the method below to controller:

```
@RequestMapping(value = "/create-user",
method = RequestMethod.GET)
public String showCreateUserPage(ModelMap model) {
  model.addAttribute("user", new User());
  return "user";
}
```

Important things to note:

- `@RequestMapping(value = "/create-user", method = RequestMethod.GET)`: We are mapping a URI "/create-user". For the first time, we are specifying a request method using the method attribute. This method will be invoked only for HTTP Get Requests. HTTP Get Requests are typically used to show the form. This will not be invoked for other types of HTTP requests like Post.

- `public String showCreateUserPage(ModelMap model)`: This is a typical control method.
- `model.addAttribute("user", new User())`: Setting up the model with an empty form backing object.

Important things to note:

Creating the view with form

Lets start creating the file `/WEB-INF/views/user.jsp`**.**

First up lets add in the reference to the tag libraries to use:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt"%>
<%@ taglib uri="http://www.springframework.org/tags/form" prefix=
<%@ taglib uri="http://www.springframework.org/tags" prefix="spri
```

First 2 entries are for JSTL core and formatting tag libraries. We will use the Spring form tags extensively. We provide a prefix to act as a shortcut to refer to tags.

Lets first create a form with one field:

```
<form:form method="post" modelAttribute="user">
   <fieldset>
         <form:label path="name">Name</form:label>
         <form:input path="name"
type="text" required="required" />
   </fieldset>
</form:form>
```

Important things to note:

- `<form:form method="post" modelAttribute="user">`**: This is form tag from Spring form tag library. Two attributes are specified. Data in the form is sent using post method. The second attribute**

"modelAttribute" specifies the attribute from model that acts as the form backing object. In the model, we added an attribute with name "user". We use that attribute as the modelAttribute

- `<fieldset>`**: HTML element to group a set of related controls - labels, form fields and validation messages.**
- `<form:label path="name">Name</form:label>`**: Spring form tag to show a label. path attribute specifies the field name (from bean) this label applies to.**
- `<form:input path="name" type="text" required="required" />`**: Spring form tag to create a text input field. pathattribute specifies the field name in the bean this input field has to be mapped to. requiredattribute indicates that this is a required field.**

When we use the spring form tags, the value from the form backing object (`modelAttribute="user"`**) are bound automatically to the form and (on form submit) the values from the form are automatically bound to the form backing object.**

A more complete list of the form including name and user id fields is listed below:

```
<form:form method="post" modelAttribute="user">
   <form:hidden path="guid" />

   <fieldset>
         <form:label path="name">Name</form:label>
         <form:input path="name"
         type="text" required="required" />
   </fieldset>

   <fieldset>
         <form:label path="userId">User Id</form:label>
         <form:input path="userId"
         type="text" required="required" />
   </fieldset>

   <!-password and password2 fields not shown for brewity-->
   <input class="btn btn-success" type="submit" value="Submit" />
</form:form>
```

Controller Get Method to handle Form Submit

When user submits the form, browser would send a HTTP POST request. Lets now create a method to handle this. To keep things simple, we will log the content of the form object. Full listing of the method below:

```
@RequestMapping(value = "/create-user", method = RequestMethod.PO
public String addTodo(User user) {
  logger.info("user details " + user);
  return "redirect:list-users";
}
```

A few important details:

- `@RequestMapping(value = "/create-user", method = RequestMethod.POST)`**: Since we want to handle the form submit, we use the method RequestMethod.POST**
- `public String addTodo(User user)`**: We are using the form backing object as the parameter. Spring MVC would automatically bind the values from the form to the form backing object.**
- `logger.info("user details " + user)`**: Log the details of the user.**
- `return "redirect:list-users"`**: Typically, on submit of a form, we save the details a database and redirect the user to a different page. Here we are redirecting the user to /list-users. When we use "redirect:", Spring MVC sends a HTTP Response with status 302 i.e. REDIRECT to the new url. Browser, on processing the 302 response, would redirect user to the new url. While POST/REDIRECT/GET pattern is not a perfect fix for the duplicate form submission problem, it does reduce the occurrences, especially those that occur after the view is rendered.**

Code for list users is pretty straightforward. Code is listed below:

```
@RequestMapping(value = "/list-users",
method = RequestMethod.GET)
public String showAllUsers() {
   return "list-users";
}
```

Unit Testing

We will discuss about Unit Testing when we add validations in the next flow.

Flow 6: Adding Validation to previous flow

In the previous flow, we added in a form. However, we did not validate the values in the form. While we can write JavaScript to validate the form content, it is always secure to do validation on the server. In this flow, lets add validation to the form we created earlier on the server side using Spring MVC.

Spring MVC provides great integration with Bean Validation API. The JSR 303 and JSR 349 define specification for the Bean Validation API (version 1.0 and 1.1, respectively), and Hibernate Validator is the reference implementation.

Hibernate Validator Dependency

Lets start with adding Hibernate Validator to our project pom.xml:

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

Simple Validations on the Bean

Bean Validation API specifies a number of validations that can be specified on attributes on the beans. Consider the listing below:

```
@Size(min = 6, message = "Enter at least 6 characters")
private String name;

@Size(min = 6, message = "Enter at least 6 characters")
private String userId;

@Size(min = 8, message = "Enter at least 8 characters")
private String password;

@Size(min = 8, message = "Enter at least 8 characters")
private String password2;
```

A few important things to note:

- `@Size(min = 6, message = "Enter at least 6 characters")` : Specifies that the field should at least have 6 characters. If the validation does not pass, the text from message attribute is used as validation error message.
- Other validations that can be performed using Bean Validation
  - @NotNull: Should not be null
  - @Size(min =5, max = 50)
  - @Past: Should be a date in the past
  - @Future: Should be a future date
  - @Pattern: Should match the provided regular expression
  - @Max: Maximum value for the field
  - @Min: Minimum value for the field

Lets now focus on getting the controller method to validate the form on submits. Complete method listing below:

```
@RequestMapping(value = "/create-user-with-validation",
method = RequestMethod.POST)
public String addTodo(@Valid User user, BindingResult result) {
  if (result.hasErrors()) {
     return "user";
   }

   logger.info("user details " + user);
   return "redirect:list-users";
}
```

Few important things:

- `public String addTodo(@Valid User user, BindingResult result)` : When @Valid annotation is used, Spring MVC validates the bean. The result of validation is made available in the BindingResult instance result.
- `if (result.hasErrors())`: Checking if there are any validation errors
- `return "user"`: If there are validation errors, we send the user back to the user page.

We need to now enhance the user.jsp to show the validation messages in case of validation errors. Complete list for one of the fields is shown below. Other fields have to be similarly updated.

```
<fieldset>
   <form:label path="name">Name</form:label>
   <form:input path="name" type="text" required="required" />
   <form:errors path="name" cssClass="text-warning"/>
</fieldset>
```

- `<form:errors path="name" cssClass="text-warning"/>` : Spring form tag to display the errors related to the field name specified in path. We can also assign the css class used to display the validation error.

Custom Validations

More complex custom validations can be implemented using **@AssertTrue** annotation. Listing below shows an example method added to User class:

```
@AssertTrue(message = "Password fields don't match")
private boolean isValid() {
  return this.password.equals(this.password2);
}
```

`@AssertTrue(message = "Password fields don't match")` : Message to be shown if the validation fails.

Any complex validation logic with multiple fields can be implemented in these methods.

Unit Testing

Unit testing for this part is focused on checking for validation errors. We will write a test for an empty form, which triggers 4 validation errors.

Controller Setup

Controller setup is very simple:

```
this.mockMvc = MockMvcBuilders.standaloneSetup(
                          new UserValidationController()).build();
```

Test Method

Complete test method is listed below:

```
@Test
public void basicTest_WithAllValidationErrors() throws Exception
  this.mockMvc
    .perform(
       post("/create-user-with-validation")
      .accept(MediaType.parseMediaType(
        "application/html;charset=UTF-8")))
      .andExpect(status().isOk())
      .andExpect(model().errorCount(4))
      .andExpect(model().attributeHasFieldErrorCode
   ("user", "name", "Size"));
}
```

Points to note:

- `post("/create-user-with-validation")` : Creates a HTTP POST request to the specified URI. Since we are not passing any request parameters, all attributes are null. This would trigger validation errors.
- `model().errorCount(4)` : Check that there are 4 validation errors on the model
- `model().attributeHasFieldErrorCode("user", "name", "Size")`: Check that attribute "user" has a field "name" with validation error named "Size"

An overview of Spring MVC

Now that we looked at a few basic flows with Spring MVC, we will switch our attention to understanding how these flows work. How does the magic happen with Spring MVC?

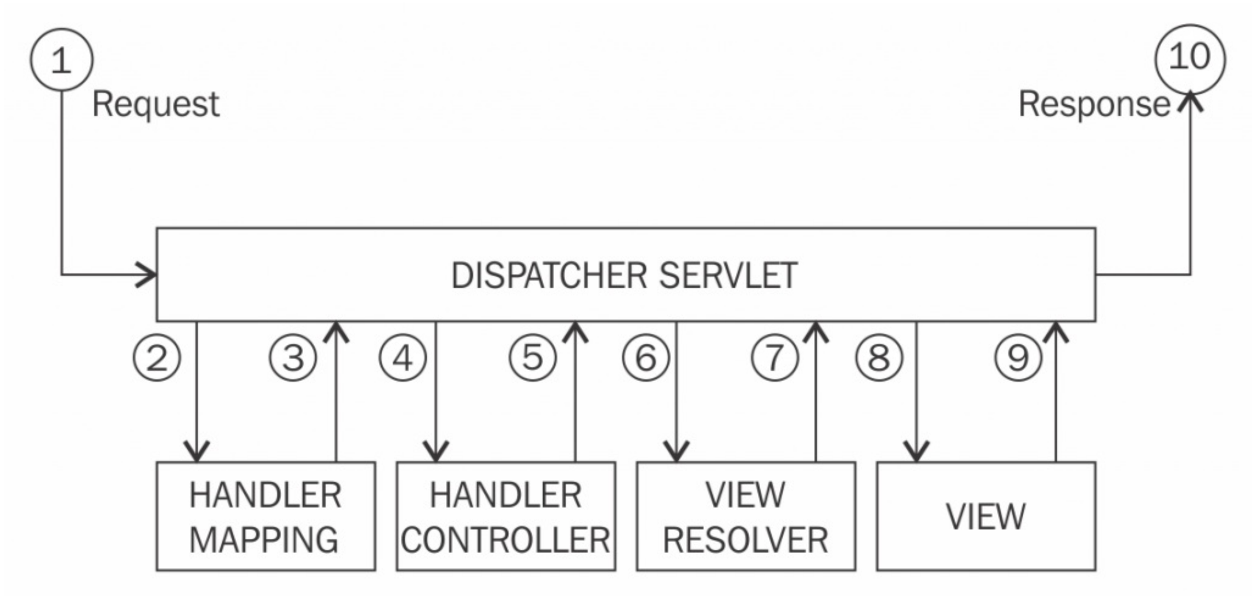
Important Features

While we worked with the different flows, we looked at some of the important features of Spring MVC framework

- Loosely couple architecture with well defined, independent roles for each of the objects
- Highly flexible Controller method definitions. Controller methods can have a varied range of parameters and return values. This gives the flexibility to the programmer to choose the definition that meets his needs.
- Allows reuse of domain objects as form backing objects. Reduces the need to have separate form objects.
- Built in tag libraries (spring, spring-form) with localization support.
- Model uses a HashMap with key value pairs. Allows integration with multiple view technologies.
- Flexible binding. Type mismatches while binding can be handled as validation errors instead of runtime errors.
- Mock MVC Framework to unit test Controllers

How it works?

Key components in the Spring MVC architecture are shown in the picture below:



Lets look at an example flow and understand the different steps involved in executing the flow. We will take Flow 4 returning ModelAndView as the specific example. URL of flow 4 is http://localhost:8080/welcome-model-view. Different steps are detailed below:

1. Browser issues a request to a specific URL. Dispatcher Servlet is the front controller, handling all requests. So, it receives the request.
2. Dispatcher Servlet looks at the URI (in the example, **/welcome-model-view**) and needs to identify the right controller to handle it. To help find the right controller, it talks to the Handler Mapping.
3. Handler Mapping returns the specific Handler method (in the example, welcome method in BasicModelViewController) that handlers the request.
4. Dispatcher Servlet invokes the specific handler method (`public ModelAndView welcome(ModelMap model)`)
5. Step 5: Handler method returns model and view. In this example, ModelAndView object is returned.
6. Dispatcher Servlet has the logical view name (from ModelAndView. In this example "welcome-model-view"). It needs to figure how to determine the physical view name. It checks if there are any View Resolvers available. It finds the view resolver that was configured (org.springframework.web.servlet.view.InternalResourceViewResolver). It calls the view resolver giving it the logical view name (in this example

"welcome-model-view") as input.

7. View resolver does the logic to map the logical view name to the physical view name. In this example, "welcome-model-view" is translated to "/WEB-INF/views/welcome-model-view.jsp"
8. Dispatcher Servlet executes the View. It also makes the Model available to the View.
9. View returns the content to be sent back to Dispatcher Servlet.
10. Dispatcher Servlet sends the response back to the browser.

Important concepts behind Spring MVC

Request Mapping

As we discussed in earlier examples, a Request Mapping is used to map a URI to a Controller or a Controller method. It can be done at class and/or method levels. An optional method parameter allows us to map the method to a specify request method (GET, POST, etc.).

Examples on request mapping

A few examples illustrate the variations below:

Example 1

In the example below, there is only one RequestMapping: On the method showPage. The method showPage will be mapped to GET, POST and any other request types for URI /show-page.

```
@Controller
public class UserController {
  @RequestMapping(value = "/show-page")
  public String showPage() {
    /* Some code */
  }
}
```

Example 2

In the example below, there is a method defined on the RequestMapping: `RequestMethod.GET`. The method `showPage` will be mapped only to GET request for URI /show-page. All other request types would through method not supported exception.

```
@Controller
public class UserController {
   @RequestMapping(value = "/show-page" , method = RequestMethod.
   public String showPage() {
     /* Some code */
   }
}
```

Example 3

In the example below, there are two Request Mappings: One on the class and the other on the method. A combination of both Request Mappings is used to determine the URI. The method `showPage` will be mapped only to GET request for URI /user/show-page.

```
@Controller
@RequestMapping("/user")
public class UserController {
  @RequestMapping(value = "/show-page" , method = RequestMethod.G
  public String showPage() {
    /* Some code */
  }
}
```

Request Mapping Methods: Supported Method Arguments

Following are some of the types of arguments that are supported on Controller methods with Request Mapping.

Argument Type/Annotation	Use
java.util.Map / org.springframework.ui.Model / org.springframework.ui.ModelMap	Acts as model (MVC) that will be the container for values that are exposed to the view.
	Used to bind request

Command or form objects	parameters to beans. Support for validation as well.
org.springframework.validation.Errors / org.springframework.validation.BindingResult	Result of validating the command or form object(form object should be the immediately preceding method argument).
@PreDestroy	On any spring bean, a pre destroy method can be provided by using the @PreDestroy annotation. This method is called just before a bean is removed from the container. This can be used to release any resources that are held by the bean.
@RequestParam	Annotation to access specific HTTP request parameters.
@RequestHeader	Annotation to access specific HTTP request header.
@SessionAttribute	Annotation to access attributes from HTTP Session.
@RequestAttribute	Annotation to access specific HTTP request attributes.

@PathVariable	Annotation allows access to variables from URI template. /owner/{ownerId}. We will look at this in depth when we discuss about micro services.

Request Mapping Methods: Supported Return Types

Request Mapping methods support a varied range of return types. Thinking conceptually, a request mapping method should answer two questions:

- What's the View?
- What's the Model the View needs?

However, with Spring MVC the view and model need not be explicitly declared at all times.

- If a view is not explicitly defined as part of the return type, then it is implicitly defined.
- Similar any model object is always enriched.

Spring MVC uses simple rules to determine the exact view and model. A couple of important rules are listed below:

- Implicit enriching of the Model: If a model is part of the return type, it is enriched with command objects (including results from validation of the command objects). In addition, the results of methods with @ModelAttribute annotations are also added to the model.
- Implicit determination of the View: If a view name is not present in the return type, it is determined using the DefaultRequestToViewNameTranslator. By default, the DefaultRequestToViewNameTranslator removes the leading and trailing slashes as well as file extension from the URI. For example display.html becomes display.

Following are some of the return types that are supported on Controller methods with Request Mapping:

Return Type	What happens?
ModelAndView	Object includes a reference to the model and the view name
Model	Only Model is returned. View name determined using DefaultRequestToViewNameTranslator.
Map	Simple Map to expose a model
View	A view with model implicitly defined.
String	Reference to a view name.

View Resolution

Spring MVC provides very flexible view resolution. It provides multiple view options:

- Integration with JSP, Freemarker
- Multiple view resolution strategies. A few of them are listed below:
    - XmlViewResolver: View resolution based on an external xml configuration
    - ResourceBundleViewResolver: View resolution based on a property file
    - UrlBasedViewResolver: Direct mapping of logical view name to an URL
    - ContentNegotiatingViewResolver: Delegates to other view resolvers based on Accept request header.
        - Support for chaining of view resolvers with explicitly defined order of preference.
        - Directly generation of XML, JSON, Atom using Content

Negotiation.

Configuring JSP View Resolver

Below example shows the common used approach to configure a JSP View Resolver using InternalResourceViewResolver. Physical view name is determined using the configured prefix and suffix for the logical view name using JstlView.

```
<bean id="jspViewResolver" class= "org.springframework.web.servle
    <property name="viewClass"
    value="org.springframework.web.servlet.view.JstlView"/>
    <property name="prefix" value="/WEB-INF/jsp/"/>
    <property name="suffix" value=".jsp"/>
</bean>
```

There are other approaches using property and xml files for mapping.

Configuring Freemarker

Following example shows the typical approach used to configure a Freemarker view resolver.

First bean `freemarkerConfig` is used to load the Freemarker templates.

```
<bean id="freemarkerConfig" class="org.springframework.web.servle
<property name="templateLoaderPath" value="/WEB-INF/freemarker/"/
</bean>
```

The bean definition below shows how to configure a Freemarker view resolver.

```
<bean id="freemarkerViewResolver" class="org.springframework.web.
<property name="cache" value="true"/>
<property name="prefix" value=""/>
<property name="suffix" value=".ftl"/>
</bean>
```

As with JSPs, the view resolution can be defined using properties or an XML file.

Handler Mappings and Interceptors

In version before Spring 2.5 (before there was support for Annotations), the mapping between a URL and a Controller (also called a Handler) was expressed using something called a Handler Mapping. It is almost a historical fact today. The use of annotations eliminated the need for an explicit Handler Mapping.

Handler Interceptors can be used to intercept requests to Handlers (or **Controllers**). Sometimes, you would want to do some processing before and after a request. You might want to log the content of request and response or you might want to find out how much time a specific request took.

There are two steps in creating a Handler Interceptor

1. Defining the Handler Interceptor
2. Mapping the Handler Interceptor to the specific Handlers to intercept

Defining a Handler Interceptor

Following are the methods you can override in **HandlerInterceptorAdapter**.

- `public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)` : Invoked before the Handler method is invoked.
- `public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView)` : Invoked after the Handler method is invoked.
- `public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex)` : Invoked after the request processing is complete

Following example implementation shows how to create a Handler Interceptor: Lets start with creating a new class extending `HandlerInterceptorAdapter`.

```
public class HandlerTimeLoggingInterceptor extends HandlerInterce
```

The preHandle method is invoked before the Handler is called. Lets put an attribute on the request indicating the start time of handler invocation.

```
@Override
public boolean preHandle(HttpServletRequest request,
               HttpServletResponse response, Object handler) thro
  request.setAttribute(
    "startTime", System.currentTimeMillis());
  return true;
}
```

The postHandle method is invoked after the Handler is called. Lets put an attribute on the request indicating the end time of handler invocation.

```
@Override
public void postHandle(HttpServletRequest request,
HttpServletResponse response, Object handler,
ModelAndView modelAndView) throws Exception
{
  request.setAttribute(
    "endTime", System.currentTimeMillis());
}
```

The afterCompletion method is invoked after the request processing is complete. We will identify the time spent in the handler using the attributes that we set into request earlier.

```
@Override
public void afterCompletion(HttpServletRequest request,
  HttpServletResponse response, Object handler, Exception ex)
  throws Exception {
    long startTime = (Long) request.getAttribute("startTime");
    long endTime = (Long) request.getAttribute("endTime");
    logger.info("Time Spent in Handler in ms : " + (endTime -
    startTime));
  }
```

Mapping Handler Interceptor to Handlers

Handler Interceptors can be mapped to specific urls you would want to intercept. Below example shows an example xml context configuration. By default the interceptor would intercept all Handler's (**Controllers**).

```
<mvc:interceptors>
<beanclass="com.mastering.spring.springmvc.controller.interceptor
</mvc:interceptors>
```

We can configure precise URIs to intercept. Below example all Handler except those with URI mapping starting with `/secure/`are intercepted.

```
<mvc:interceptors>
<mapping path="/**"/>
<exclude-mapping path="/secure/**"/>
<beanclass="com.mastering.spring.springmvc.controller.interceptor
</mvc:interceptors>
```

Model Attributes

Common web forms contain a number of dropdown values - list of States, list of Countries etc. These list of values need to be available in the model so that view can display the list. Such common things are typically populated into the model using methods that are marked with **@ModelAttribute** annotations.

There are two variations possible. In the example below, the method returns the object that needs to put into the model.

```
@ModelAttribute
public List<State> populateStateList() {
  return stateService.findStates();
}
```

The approach in the example below is used to add multiple attributes to the model.

```
@ModelAttribute
public void populateStateAndCountryList() {
  model.addAttribute(stateService.findStates());
  model.addAttribute(countryService.findCountries());
}
```

An important thing to note is that there is no limitation on the number of methods that can be marked with `@ModelAttribute` annotation.

Model attributes can be made common across multiple Controllers using Controller Advice. We will discuss Controller Advice later in this section.

Session Attributes

All the attributes and values that we discussed until now are used within a single request. However, there might be values like a specific web user configuration that might not change across requests. These kinds of values would be typically stored in an http session. Spring MVC provides a simple type level (class level) annotation **@SessionAttributes** to specify the attributes that would be stored in session.

Consider the example below:

```
@Controller
@SessionAttributes("exampleSessionAttribute")
public class LoginController {
```

Putting an attribute in Session

Once we define an attribute in @SessionAttributes annotation, it is automatically added into session if the same attribute is added into model.

In the above example, if we put an attribute with name exampleSessionAttribute into the model, it would be automatically stored into session conversation state.

```
model.put("exampleSessionAttribute", sessionValue);
```

Reading an attribute from Session

This value can be accessed in other controllers by first specifying the @SessionAttributes annotation at a type level.

```
@Controller
@SessionAttributes("exampleSessionAttribute")
public class SomeOtherController {
```

Value of the session attribute will be directly made available to all model objects. So, it can be accessed from the model.

```
Value sessionValue =(Value)model.get("exampleSessionAttribute");
```

Removing an attribute from Session

It is important to remove values from session when they are no longer needed. There are two ways we can remove values from session conversational state. The first way is demonstrated in the snippet below. It uses the removeAttribute method available in WebRequest class.

```
@RequestMapping(value="/some-method",method = RequestMethod.GET)
public String someMethod(/*Other Parameters*/
WebRequest request, SessionStatus status) {
  status.setComplete();
  request.removeAttribute("exampleSessionAttribute",
    WebRequest.SCOPE_SESSION);
    //Other Logic
}
```

Below example shows the second approach using the cleanUpAttribute method in SessionAttributeStore.

```
@RequestMapping(value = "/some-other-method",
method = RequestMethod.GET)
public String someOtherMethod(/*Other Parameters*/
 SessionAttributeStore store, SessionStatus status) {
   status.setComplete();
   store.cleanupAttribute(request, "exampleSessionAttribute");
   //Other Logic
}
```

Init Binders

Typical web forms have dates, currencies and amounts. The values in the forms need to be binded to the form backing objects. Customization of how binding happens can be introduced using **@InitBinder** annotation.

Customization can be done in a specific controller or a set of controllers using Handler Advice. Below example shows how to set the default date format to use for form binding.

```
@InitBinder
protected void initBinder(WebDataBinder binder) {
  SimpleDateFormat dateFormat = new SimpleDateFormat("dd/MM/yyyy"
  binder.registerCustomEditor(Date.class, new CustomDateEditor(
    dateFormat, false));
}
```

Controller Advice

Some of functionality we have defined at controller level can be common across the application. For example: we might want to use the same date format across the application. So, the @InitBinder that we defined earlier can be applicable across the application. How do we achieve that? Controller Advice helps us make functionality common across all Request Mappings by default.

For example, consider the Controller advice example listed below: We use an annotation **@ControllerAdvice** on the class and define the method with @InitBinder in this class. By default, the binding defined in this method is applicable to all request mappings.

```
@ControllerAdvice
public class DateBindingControllerAdvice {

  @InitBinder
  protected void initBinder(WebDataBinder binder) {
    SimpleDateFormat dateFormat = new
    SimpleDateFormat("dd/MM/yyyy");
    binder.registerCustomEditor(Date.class,
      new CustomDateEditor(
        dateFormat, false));
   }
}
```

Controller advice can also be used to define common **Model Attributes (@ModelAttribute)** and common **Exception Handling (@ExceptionHandler)**. All that you would need to do is to create methods marked with appropriate annotations. We will discuss Exception Handling in the next section.

Spring MVC - Advanced Features

Exception Handling

Exception Handling is one of the critical parts of any application. It is very important to have a consistent Exception Handling strategy used across the application. One of the popular misconceptions is that only bad applications need Exception Handling. Nothing can be further from the truth. Even well designed, well-written applications need good Exception Handling.

Before the emergence of Spring framework, Exception Handling code was needed across application code due to wide use of Checked Exceptions. For example, most of the JDBC methods threw Checked Exceptions, needing a try catch to handle the exception in every method (unless you would want to declare that the method throws a JDBC Exception) . With Spring framework, most of the exceptions were made into Unchecked Exceptions. This made sure that unless specific exception handling is needed, exception handling could be handled generically across the application.

In this section, we will look at couple of example implementations of Exception Handling.

- Common Exception Handling across all Controllers
- Specific Exception Handling for a Controller

Common Exception Handling across Controllers

Controller advice can also be used to implement common Exception Handling across Controllers.

Consider the code below:

```
@ControllerAdvice
public class ExceptionController {
  private Log logger =
    LogFactory.getLog(ExceptionController.class);

  @ExceptionHandler(value = Exception.class)
  public ModelAndView handleException
  (HttpServletRequest request, Exception ex) {
    logger.error("Request " + request.getRequestURL()
      + " Threw an Exception", ex);
    ModelAndView mav = new ModelAndView();
    mav.addObject("exception", ex);
    mav.addObject("url", request.getRequestURL());
    mav.setViewName("common/spring-mvc-error");
    return mav;
   }

}
```

Few things to Note:

- `@ControllerAdvice`: Controller Advice, by default, is applicable to all controllers.
- `@ExceptionHandler(value = Exception.class)`: Any method with this annotation will be called when an exception of type or the sub type of the class specified(Exception.class) is thrown in the Controllers
- `public ModelAndView handleException (HttpServletRequest request, Exception ex)`: The exception which is thrown is injected into the Exception variable. Method is declared with a ModelAndView return type to be able to return a model with the exception details and an exception view
- `mav.addObject("exception", ex)`: Adding the exception to the model so that the exception details can be shown in the view
- `mav.setViewName("common/spring-mvc-error")`: Exception view

The Error View

Whenever an exception happens, the `ExceptionController` redirects user to the spring-mvc-error view after populating the model with exception details. Following snippet show the complete jsp - /`WEB-INF/views/common/spring-mvc-error.jsp`:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
```

```
<%@page isErrorPage="true"%>
<h1>Error Page</h1>
URL: ${url}
<BR />
Exception: ${exception.message}
<c:forEach items="${exception.stackTrace}" var="exceptionStackTra
${exceptionStackTrace}
</c:forEach>
```

Important things to note:

- `URL: ${url}`: Shows the url from model
- `Exception: ${exception.message}`: Displays the exception message. exception is populated into model from ExceptionController
- `forEach around ${exceptionStackTrace}`: Displays the stack trace from exceptionController Specific Exception Handling

Specific Exception Handling in a Controller

In some situations, there is a need for specific exception handling in a Controller. This situation can easily be handled by implementing a method annotated with `@ExceptionHandler(value = Exception.class).`

In case specific exception handling is needed only for a Specific Exception, the specific exception class can be provided as the value for the value attribute of the annotation.

Internationalization

When we develop applications, we would want them to be usable in multiple locales. You would want the text that is shown to be user to be customized based on the location and language of the user. This is called Internationalization. Internationalization, i18n, is also called Localization.

Internationalization can be implemented using two approaches:

- Session Locale Resolver
- Cookie Locale Resolver

In the case of Session Locale Resolver, the locale chosen by user is stored in the user session and therefore, is valid for the user session only. However, in the case of a Cookie Locale Resolver, the locale chosen is stored as a cookie.

Message Bundle Setup

Lets first setup a message bundler. Code snippet from the spring context below:

```
<bean id="messageSource"  class=
"org.springframework.context.support.ReloadableResourceBundleMess
  <property name="basename" value="classpath:messages" />
  <property name="defaultEncoding" value="UTF-8" />
</bean>
```

Important points to note:

- `class="org.springframework.context.support.ReloadableResource` We are configuring a reloadable resource bundle. Support reloading properties through cacheSeconds setting.
- `<property name="basename" value="classpath:messages" />`: Configure loading of properties from file messages.properties and messages_{locale}.properties. We will discuss about locale soon.

Lets configure a couple of property files and make them available in the src/main/resources folder:

```
message_en.properties
welcome.caption=Welcome in English
message_fr.properties
welcome.caption=Bienvenue - Welcome in French
```

We can display the message from message bundle in a view using the spring:message tag

```
<spring:message code="welcome.caption" />
```

Configuring a Session Locale Resolver

There are two parts. First one is to configure a locale resolver. The second one is to configure an interceptor to handle change in locale.

```
<bean id="springMVCLocaleResolver"
   class="org.springframework.web.servlet.i18n.SessionLocaleResol
   <property name="defaultLocale" value="en" />
</bean>

<mvc:interceptors>
 <bean id="springMVCLocaleChangeInterceptor"
   class="org.springframework.web.servlet.i18n.
   LocaleChangeInterceptor">
   <property name="paramName" value="language" />
 </bean>
</mvc:interceptors>
```

Important things to note:

- `<property name="defaultLocale" value="en" />`: By default, en locale is used
- `<mvc:interceptors>`: LocaleChangeInterceptor is configured as a Handler Interceptor. It would intercept all the Handler requests and check for the locale.
- `<property name="paramName" value="language" />`: LocaleChangeInterceptor is configured to use a request param name called language to indicate the locale. So any URL of format http://server/uri?language={locale} would trigger a change in locale.
- If you append language=en to any URL, you would be using en locale for the duration of the session. If you append language=fr to any URL, then you would be using a French locale

Configuring a Cookie Locale Resolver

We use a Cookie Locale Resolver in the example below:

```
<bean id="localeResolver"
   class="org.springframework.web.servlet.i18n.CookieLocaleResolv
   <property name="defaultLocale" value="en" />
   <property name="cookieName" value="userLocaleCookie"/>
   <property name="cookieMaxAge" value="7200"/>
</bean>
```

Important things to note:

- `<property name="cookieName" value="userLocaleCookie"/>`: Name of the cookie stored in browser will be userLocaleCookie.
- `<property name="cookieMaxAge" value="7200"/>`: Lifetime of the cookie is 2 hours (7200 seconds).
- Since we are using LocaleChangeInterceptor from previous example: If you append language=en to any URL, you would be using en locale for a duration of 2 hours (or until locale is changed). If you append language=fr to any url, then you would be using a French locale for 2 hours(or until locale is changed).

Integration Testing Spring Controllers

In the flows we discussed we looked at using real unit tests - one's which only load up the specific controllers that are being testing.

Another possibility is to load up the entire Spring Context. However, this would be more of an Integration Test as we would load up the entire context. Code below shows how to do a complete launch of a Spring Context launching up all controllers.

```
@RunWith(SpringRunner.class)
@WebAppConfiguration
@ContextConfiguration("file:src/main/webapp/WEB-INF/user-web-cont
public class BasicControllerSpringConfigurationIT {

   private MockMvc mockMvc;

   @Autowired
   private WebApplicationContext wac;

   @Before
   public void setup() {
     this.mockMvc = MockMvcBuilders.webAppContextSetup
       (this.wac).build();
   }

   @Test
   public void basicTest() throws Exception {
     this.mockMvc.perform(get("/welcome")
       .accept(MediaType.parseMediaType
     ("application/html;charset=UTF-8")))
       .andExpect(status().isOk())
       .andExpect(content().string
     ("Welcome to Spring MVC"));
   }
}
```

Few things to note:

- **@RunWith(SpringRunner.class)**: SpringRunner helps us to launch a Spring Context.
- **@WebAppConfiguration**: To launch a web app context with Spring MVC
- **@ContextConfiguration("file:src/main/webapp/WEB-INF/user-web-context.xml")**: Specifies the location of the spring context xml.
- **this.mockMvc = MockMvcBuilders.webAppContextSetup(this.wac).build()**: In the earlier examples, we used a standalone setup. However, in this example we want to launch the entire web app. So, we use a webAppContextSetup.
- Execution of the test is very similar to how we did it in earlier tests.

Serving Static Resources

Most teams today have separate teams delivering front end and back end content. Front end is developed with modern JavaScript frameworks like AngularJs, Backbone etc. Backend is build through web applications or REST services based on frameworks like Spring MVC.

With this evolution in front-end frameworks, it is very important to find the right solutions to version and deliver front-end static content.

Following are some of the important features provided by Spring MVC framework:

- Expose static content - from folders in web application root
- Enable Caching
- Enable GZip Compression of Static Content

Exposing Static Content

Web applications typically have a lot of static content. Spring MVC provides options to expose static content from folders on web application root as well locations on class path. Following snippet shows content within the war can be exposed as static content:

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

Things to note:

- `location="/static-resources/"` : The location specifies the folders inside the war or class path that you would want to expose as static content. In this example, we want to expose all content in folder **static-resources** inside the root of war as static content. We can specify multiple comma-separated values to expose multiple folders under same external facing URI.
- `mapping="/resources/**"` : mapping specifies the external facing URI path. So, a CSS file named `app.css` inside folder static-resources can be accessed using the URI `/resources/app.css`

The complete Java config for the same configuration is shown below:

```
@Configuration
@EnableWebMvc
public class WebConfig extends WebMvcConfigurerAdapter {
  @Override
  public void addResourceHandlers
  (ResourceHandlerRegistry registry) {
    registry
      .addResourceHandler("/static-resources/**")
      .addResourceLocations("/static-resources/");
  }
}
```

Caching Static Content

Caching for static resources can be enabled for improved performance. The browser would cache the resources served for the specified time period. **cache-period** attribute or `setCachePeriod` method can be used to specify the caching interval (in seconds) based on the type of config used. Snippets below show the details:

Java Config

```
registry
.addResourceHandler("/resources/**")
.addResourceLocations("/static-resources/")
.setCachePeriod(365 * 24 * 60 * 60);
```

XML Config

```
<mvc:resources
mapping="/resources/**"
location="/static-resources/"
cache-period="365 * 24 * 60 * 60"/>
```

Response header "Cache-Control: max-age={specified-max-age}" will be sent to the browser.

Enabling GZip Compression of Static Content

Compressing response is a simple way to make web applications faster. All modern browsers support GZip compression. Instead of sending the full static content file, a compressed file can be sent as response. Browser would decompress and use the static content.

Browser can specify that it can accept compressed content with a request header. If the server supports it, it can deliver compressed content - again marked with a response header.

Request Header sent from browser

```
Accept-Encoding: gzip, deflate
```

Response Header (from web application)

```
Content-Encoding: gzip
```

Snippets below show how to add a Gzip resolver to deliver compressed static content:

```
registry
      .addResourceHandler("/resources/**")
      .addResourceLocations("/static-resources/")
      .setCachePeriod(365 * 24 * 60 * 60)
      .resourceChain(true)
      .addResolver(new GzipResourceResolver())
      .addResolver(new PathResourceResolver());
```

Things to note:

- `resourceChain(true)`: We would want to enable Gzip compression but would want to fallback to the delivering full file, if full file were requested. So, we use resource chaining (Chaining of resource resolvers).
- `addResolver(new PathResourceResolver())`: `PathResourceResolver` is the default resolver. It resolves based on the Resource Handlers and Locations configured.
- `addResolver(new GzipResourceResolver())`: `GzipResourceResolver` enables Gzip compression, when requested.

Integrating Spring MVC with Bootstrap

- One of the approaches to use Bootstrap in a web application is to download the JavaScript and CSS files and make the available in the respective folders. However, this would mean that every time there is a new version of Bootstrap, we would need to download and make it available as part of the source code. Question is: Is there a way that we can introduce Bootstrap or any other static (js or css) libraries using dependency management like Maven?
- Answer is WebJars. WebJars are client-side js or css libraries packaged into JAR files. We can use Java build tools (Maven or Gradle) to download and make them available to the application. Biggest advantage is that WebJars are resolve transitive dependencies.
- Lets now use Bootstrap WebJar and include it into our web application. The steps involved are

- Add Bootstrap WebJars as a maven dependency
- Configure Spring MVC Resource Handler to deliver static content from WebJar.
- Use Bootstrap resources (css and js) in the JSP

Bootstrap WebJar as Maven Dependency

- Lets add this to pom.xml

```
<dependency>
  <groupId>org.webjars</groupId>
  <artifactId>bootstrap</artifactId>
  <version>3.3.6</version>
</dependency>
```

Configure Spring MVC Resource Handler to deliver WebJar static content

This is very simple. We need to add following mapping to the spring context.

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

- With above configuration, ResourceHttpRequestHandler makes the content from webjars available as static content.
- As discussed in the section on static content, we can specific cache period if we want to cache the content.

Use Bootstrap resources in JSP

We can add the bootstrap resources just like other static resources in the JSP.

```
<script src=
  "webjars/bootstrap/3.3.6/js/bootstrap.min.js">
</script>
<link
href="webjars/bootstrap/3.3.6/css/bootstrap.min.css"
rel="stylesheet">
```

Spring Security

Critical part of web applications is Authentication and Authorization. Authentication is the process of establishing a users identity - verifying that the user is who he claims to be. Authorization is to check if the user has access perform the specific action. Authorization specifies the access a user has. Can the user view a page? Can the user edit a page? Can the user delete a page?

Best Practice is to enforce authentication and authorization on every page in the application. User credentials and authorization should be verified before executing any request to a web application.

Spring Security provides a comprehensive security solution for Java EE enterprise applications. While providing great support to Spring (and Spring MVC) based applications, it can be integrated with other frameworks as well.

The list below highlights some of vast range of authentication mechanisms that Spring Security supports.

- Form-based authentication: Simple integration for basic applications.
- LDAP: Typically used in most Enterprise applications
- Java Authentication and Authorization Service (JAAS): Authentication and Authorization Standard - Part of Java EE standard specification
- Container Managed Authentication
- Custom Authentication Systems

Lets consider a simple example for enabling Spring Security on simple web application. We will use an in memory configuration.

Steps involved are:

1. Adding Spring Security Dependency
2. Configure interception of all requests
3. Configure Spring Security

4. Add Logout functionality

Adding Spring Security Dependency

We will start with adding the Spring Security dependencies to pom.xml

```
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-web</artifactId>
  <version>4.0.1.RELEASE</version>
</dependency>

<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-config</artifactId>
  <version>4.0.1.RELEASE</version>
</dependency>
```

Configuring a Filter to intercept all requests

We would want to configure Spring Security to intercept all requests to web application. We will use a filter, DelegatingFilterProxy, which delegates to a spring managed bean FilterChainProxy.

```
<filter>
<filter-name>springSecurityFilterChain</filter-name>
<filter-class>
  org.springframework.web.filter.DelegatingFilterProxy
</filter-class>
</filter>
<filter-mapping>
<filter-name>springSecurityFilterChain</filter-name>
<url-pattern>/*</url-pattern>
</filter-mapping>
```

Now, all requests to our web application will go through the filter. However, we have not configured any thing related to security yet. Lets use a simple Java configuration example:

```
@Configuration
@EnableWebSecurity
public class SecurityConfiguration extends
```

```
  WebSecurityConfigurerAdapter {
    @Autowired
    public void configureGlobalSecurity
    (AuthenticationManagerBuilder auth) throws Exception {
      auth
      .inMemoryAuthentication()
      .withUser("firstuser").password("password1")
      .roles("USER", "ADMIN");
    }

   @Override
   protected void configure(HttpSecurity http) throws Exception {
     http
       .authorizeRequests()
       .antMatchers("/login").permitAll()
       .antMatchers("/*secure*/**")
       .access("hasRole('USER')")
       .and().formLogin();
   }
}
```

Things to note:

- `@EnableWebSecurity` : This annotation enables any Configuration class to contain the definition of Spring Configuration. Also provides Spring MVC Integration. In this specific instance we override a couple of methods to provide our specific Spring MVC Configuration.
- `WebSecurityConfigurerAdapter`: This class provides a base class for creating a Spring Configuration(WebSecurityConfigurer).
- `protected void configure(HttpSecurity http)`: This method provides the security needs for different URLs.
- `antMatchers("/*secure*/**").access("hasRole('USER')")`: You would need a role of USER for accessing any URL containing secure.
- `antMatchers("/login").permitAll()`: Permit access of login page to all users
- `public void configureGlobalSecurity(AuthenticationManagerBuilder auth)`: In this example, we are using in memory authentication. This can be used to connect to a database (auth.jdbcAuthentication()) or an LDAP(auth.ldapAuthentication()) or a custom authentication provider (created extending AuthenticationProvider)
- `withUser("firstuser").password("password1")`: Configuring an in

memory valid userid and password combination
- `.roles("USER", "ADMIN")`: Assigning roles to the user.When we try to access any secure URLs you will be redirected a logic page. Spring Security provides ways of customizing the logic page as well as the redirection. Only authenticated users with the right roles will be allowed to access the secured application pages.

Logout

Spring Security provides features to enable a user to logout and be redirect to a specified page.

```
@Controller
public class LogoutController
{

  @RequestMapping(value = "/secure/logout",
    method = RequestMethod.GET)
    public String logout(HttpServletRequest request,
    HttpServletResponse response) {
      Authentication auth =
        SecurityContextHolder.getContext()
          .getAuthentication();

  if (auth != null) {
    new SecurityContextLogoutHandler()
    .logout(request, response, auth);
     request.getSession().invalidate();
  }
     return "redirect:/secure/welcome";
   }
}
```

Things to note:

- `if (auth != null)`: If there is a valid authentication, then end the session
- `new SecurityContextLogoutHandler().logout(request, response, auth)`: SecurityContextLogoutHandler performs a logout by removing the authentication information from the SecurityContextHolder
- `return "redirect:/secure/welcome"`: Redirect to the secure welcome

page

Summary

In this chapter, we learnt how to develop web applications with Spring MVC. Spring MVC can also be used to build REST services. We would discuss that and more about REST services in the subsequent chapters.

In the next chapter, we will shift our attention towards Micro services. We will understand why the world is looking keenly at Micro services. We will understand the importance of applications being cloud native.

We will send you the next chapters when they are ready………!

Hope you like the content presented.

Chapter 3. Evolution towards Microservices and Cloud Native Applications

In the last decade, Spring framework has evolved into the most popular framework to develop **Enterprise Java** applications. Spring framework made it easy to develop loosely coupled, testable applications. Spring framework simplified the implementation of cross cutting concerns.

The world today, however, is very different from a decade back. Over a period of time, applications grew into monoliths, which became difficult to manage. And, because of those problems, new architectures are evolving. The buzzwords in the recent past have been Restful Services, Microservices and **Cloud Native** applications.

In this chapter, we will start with reviewing the problems Spring framework solved in the last decade. We will look at the problems with monolith applications and get introduced into the world of smaller, independently deployable components.

We will understand why world is moving towards Microservices and Cloud Native applications. And we will end the chapter looking at how Spring Framework and Spring Projects are evolving to solve today's problems.

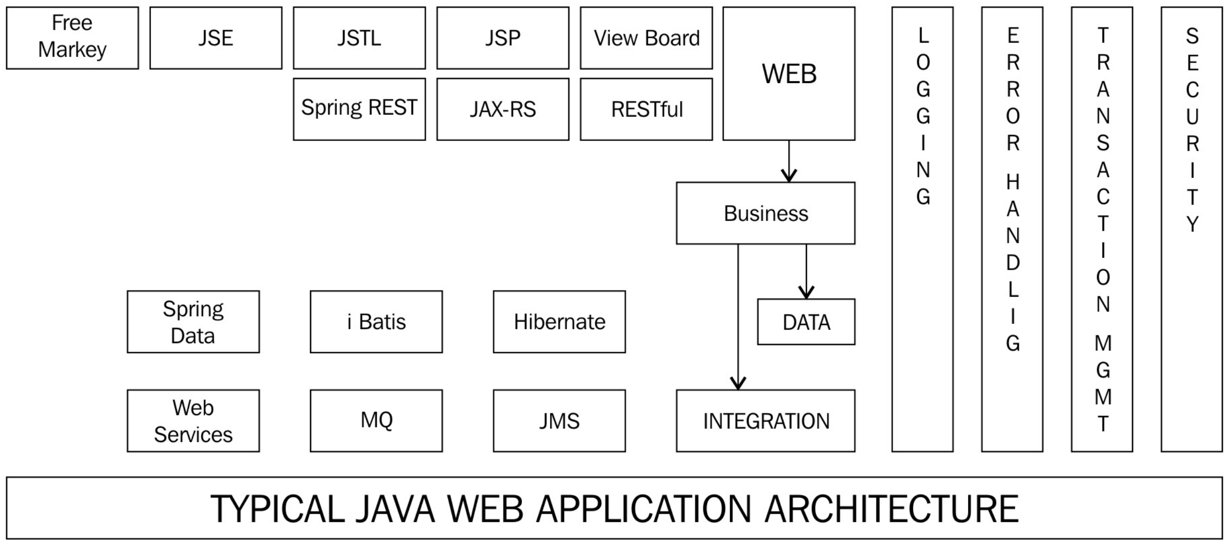
This chapter will cover the following topics:

- Architecture of a Typical Spring Based Application
- Problems solved by Spring framework in the last decade.
- What are our goals when we develop applications?
- What are the challenges with Monolith applications?
- What are Microservices?
- What are the advantages of Microservices?
- What are the challenges with Microservices?
- What are the good practices that help in deploying Microservices to Cloud?
- What are the Spring projects that help us develop Microservices and

Cloud Native applications?

Typical web application architecture with Spring

Spring is the framework of choice to wire Java enterprise applications during the last decade and half. Applications used a layered architecture with all crosscutting concerns being managed using **Aspect Oriented Programming** (**AOP**). The proceeding diagram shows typical architecture for a web application developed with Spring:



Typical layers in such an application are listed below. We will list cross cutting concerns as a separate layer, though in reality, they are applicable across all layers.

- **Web Layer**: Web Layer is typically responsible for the controlling the web application flow (Controller and/or Front Controller) and rendering the view.
- **Business Layer**: This is where all your business logic is written. Most applications have transaction management starting from the business layer.
- **Data Layer**: This is response for talking to the data. Mapping or

translating your Java objects to the database and vice-versa.
- **Integration Layer**: Applications talk to other applications - either over queues or invoking web services. The integration layer establishes such connections with other applications.
- **Cross Cutting Concerns**: These are concerns across different layers - logging, security, transaction management etc. Since Spring IOC Container manages the beans, it can weave these concerns around the beans through AOP.

Let us discuss each of the layers and the frameworks used in more detail.

Web layer

Web layer is dependent on how you would want to expose the business logic to the end user. *Is it a web application?* o*r* *Are you exposing Restful web services?*

Web application - Rendering a HTML View

These web applications use a web **Model-View-Controller** (**MVC**) framework like Spring MVC or **Struts**. View can be rendered using JSP, JSF or template based frameworks like Freemarker.

Restful services

There are two typical approaches used to develop Restful web services:

- **JAX-RS**: Java API for REST Services. This is a standard from Java EE Specification. Jersey is the reference implementation.
- Spring MVC or Spring REST: Restful services can also be developed with Spring MVC.

Spring MVC does not implement JAX-RS. So, the choice is tricky. JAX-RS is a Java EE Standard. But, Spring MVC typically is more innovative and more likely to build new features faster.

Business layer

Business layer typically has all the business logic in an application. Spring framework is typically used in this layer to wire beans together.

This is also the layer where the boundary of transaction management typically begins. Transaction management can be implemented using Spring AOP or **AspectJ**. A decade back, EJBs were the most popular approach to implement your Business layer. With its lightweight nature, Spring is now the framework of choice for Business layer.

EJB 3 is much simpler than EJB 2. However, it has not been able to recover the lost ground.

Data layer

Most applications talk to a database. Data layer is responsible for storing data from your Java objects to your database and vice versa. Following are the most popular approaches to build your data layer:

- **Java Persistence API** (**JPA**): JPA's help you to map **Plain Old Java Objects** (**POJOs**) to your database tables. Hibernate is the most popular implementation for JPA. JPA is typically preferred for all transactional applications. However, the choice for batch and reporting applications is not very clear.
- **MyBatis**: MyBatis (previously **iBatis**) is a simple data-mapping framework. As its website (http://www.mybatis.org/mybatis-3/) says "*MyBatis is a first class persistence framework with support for custom SQL, stored procedures and advanced mappings. MyBatis eliminates almost all of the JDBC code and manual setting of parameters and retrieval of results. MyBatis can use simple XML or Annotations for configuration and map primitives, Map interfaces and Java POJOs (Plain Old Java Objects) to database records*". MyBatis can be considered for batch and reporting applications where SQLs and Stored Procedures are more typically used.
- Spring JDBC: JDBC and Spring JDBC are not so commonly used

anymore.

Integration layer

Integration layer is typically where we talk to other applications. There might be other applications exposing SOAP or Restful services over HTTP (Web) or **Messaging Queue** (**MQ**).

- Spring JMS is typically used to send or receive messages on Queues or Service Buses.
- Spring MVC RestTemplate can be used to invoke Restful services.
- Spring WS can be used to invoke SOAP based web services.

Cross cutting concerns

Cross Cutting Concerns are concerns, which are typically common to multiple layers of an application - logging, security, and transaction management among others. Let's quickly discuss some of these:

- **Logging**: Audit Logging at multiple layers can be implemented using AOP (Spring AOP or AspectJ).
- **Security**: Security is typically implemented using Spring security framework. As discussed in previous chapter, Spring Security makes implementing security very simple.
- **Transaction Management**: Spring framework provides a consistent abstraction for transaction management. More importantly, Spring framework provides great support for declarative transaction management. Following are some of the transaction APIs that Spring framework supports
    - **Java Transaction API (JTA)**: Standard for transaction management. Part of Java EE Specifications.
    - JDBC
    - JPA (including Hibernate)
- **Error Handling**: Most abstractions provided by Spring use unchecked exceptions. So, unless required by business logic, it is sufficient to implement Error Handling in the layer that is exposed to the client (user

or other application). Spring MVC provides Controller Advice to implement consistent error handling across the application.

Spring framework plays a major role in application architecture. Spring IOC is used to wire beans from different layers together. Spring AOP is used to weave cross cutting concerns around the beans. Adding to these is the fact that Spring provides great integration with frameworks in different layers.

In the next section, lets quickly review some of the important problems Spring solved in the last decade or so.

Problems spring solved in the last decade (or so)

Spring is the framework of choice to wire Enterprise Java applications. It solved a number of problems that Enterprise Java applications faced since the complexity associated with EJB 2. A few of them are listed below:

- Loose Coupling and Testability
- Plumbing Code
- Lightweight Architecture
- Architectural Flexibility
- Simplified Implementation of cross-cutting concerns
- Best Design Patterns for Free

Loose coupling and testability

Through **Dependency Injection**, Spring brings loose coupling between classes. While loose coupling is beneficial to application maintainability in the long run, the first benefits are realized with the testability that it brings in.

Testability was not a forte of Java EE (or J2EE as it was called then), before Spring. The only way to test EJB 2 applications was to run them in the container. Unit testing them was incredibly difficult.

That's exactly the problem Spring framework set out to solve. As we saw in the earlier chapters - if objects are wired using Spring, writing unit tests becomes easier. We can easily stub or mock dependencies and wire them into objects.

Plumbing code

Developer of the late 1990's and early to mid 2000's will be familiar with the amount of plumbing code that had to be written to execute a simple query though JDBC and populate the result into a Java object. You had to do the JNDI look up, get a connection and populate the results. This resulted in duplicate code. Typical problem was repeated exception handling code in every method. And this problem is not limited to JDBC.

One of the problems Spring framework solved was to eliminate all plumbing code. With Spring JDBC, Spring JMS and other abstractions, the developers could focus on writing business logic. Spring framework took care of the nitty-gritty details.

Lightweight architecture

Use of EJBs made the applications complex. And, not all applications needed that complexity. Spring provide a simplified, lightweight way of developing applications. If distribution was needed, it could be added in later.

Architecture flexibility

Spring framework is used to wire objects across application, in different layers. In spite of its ever-looming presence, Spring framework did not restrict the flexibility or choice of frameworks that application architects and developers had. A couple of examples are listed as follows:

- Spring framework provided great flexibility in the web layer. If you wanted to use Struts or Struts 2 instead of Spring MVC, it's configurable. You had the choice of integrating with a wider range of view and template frameworks.
- Another good example is the data layer where you had possibilities to connect with JPA, JDBC, and mapping frameworks like MyBatis.

Simplified implementation of cross-cutting concerns

When Spring framework is used to manage beans, the Spring IOC Container manages the lifecycle - creation, use, auto-wiring, and destruction - of the beans. It makes it easier to weave additional functionality like the cross cutting concerns around the beans.

Design patterns for free

Spring framework encourages the use of a number of design patterns by default. A few examples:

- **Dependency Injection or Inversion of Controller**: This is the fundamental design pattern Spring framework is built to enable. Enables loose coupling and testability.
- **Singleton**: All Spring beans are singletons by default.
- **Factory Pattern**: Using the Bean Factory to instantiate beans is a good example of the Factory Pattern.
- **Front Controller**: Spring MVC uses **Dispatcher Servlet** as the Front Controller. So, we are using the Front Controller pattern when we develop applications with Spring MVC.
- **Template Method**: Helps us avoid boilerplate code. Many Spring based classes JdbcTemplate, **JmsTemplate** are implementations of this pattern.

Application development goals

Before we move into the concepts of REST Services, Microservices, and Cloud Native applications, lets take some time to understand the common goals when we develop applications. Understanding these goals will help us understand why applications are moving towards microservices architecture.

First of all, we should remember that the software industry is still a relatively young industry. One thing that has been constant in my decade and half experience with developing, designing and architecturing software is that things change. Requirements of today are not the requirements of tomorrow. Technology today is not the technology tomorrow. While we can try predicting what happens in future, we are more often wrong.

One of the things we did during the initial decades of software development is to build software systems for future. The design and architecture were made complex in preparation for future requirements.

During the last decade, with **Agile** and **Extreme programming**, the focus has shifted to being lean and building good enough systems, adhering to basic principles of design. focus shifted to evolutionary design. The thought process is - "*If a system has good design for today's needs, and is continuously evolving and has good tests, it can easily be refactored to meet tomorrow's needs*".

While we do not know where we are heading, we do know that a big chunk of our goals, when developing applications, have not changed.

The key goals of software development, for a big chunk of applications, can be described with the statement "*Speed and Safety at Scale*".

We will discuss each of these in detail in the next section.

Speed

Speed of delivering new requirements and innovations is increasingly becoming a key differentiator. It is not sufficient to develop (code and test) fast. It is important to deliver (to production) fast. It is now common knowledge that the best software organizations in the world deliver software multiple times to production every day.

Technology and business landscape is in a constant flux and is constantly evolving. How fast can a application adapt to these changes is the key question today.

- New Programming Languages
    - Go
    - Scala
    - Closure
- New Programming Paradigms
    - Functional Programming
    - Reactive Programming
- New Frameworks
- New Tools
    - Development
    - Code Quality
    - Automation Testing
    - Deployment
- Containerizations
- New Processes and Practices
    - Agile
    - Test Driven Development
    - Behavior Driven Development
    - Continuous Integration
    - Continuous Delivery
    - DevOps
- New Devices and Opportunities
    - Mobile
    - Cloud

Safety

What is the use of speed without safety?Who would want to go in a car that can go 300 miles an hour without proper safety features built-in?

Let us consider a few characteristics of a safe application:

Reliability

Reliability is a measure of how accurately the system functions.

Key questions to ask are:

- *Is the system meeting its functional requirements?*
- *How many defects are leaked during different release phases?*

Availability

Most external client facing applications are expected to be available round the clock. Availability is a measure of how much percentage of time, your application is available for your end user.

Security

Security of applications and data is critical to the success of organizations. There should clear procedures for Authentication (*Are you who you claim to be?*), Authorization (*What access does a user have?*), and Data protection (*Is the data received or sent accurate? Is the data safe - not intercepted by unintended users?*)

Performance

If a web application does not respond within a couple of seconds, there is a very high chance that the user of your application would be disappointed. Performance usually refers to the ability of a system to provide agreed upon response times for a defined number of users.

High resilience

As applications become distributed, the probability of failures increases. *How does the application react in case of localized failures or disruptions?Can it provide basic operation without completely crashing?*

This behavior of an application to provide bare minimum service levels in case of unexpected failures is called **Resilience**.

As more and more applications move towards the cloud, Resilience of applications becomes important.

Scalability

Scalability is a measure of how an application would react when resources at its disposal are scaled up. *If an application supports 10,000 users with given infrastructure, can it support at least 20,000 users with double the infrastructure?*

If a web application does not respond within a couple of seconds, there is a very high chance that the user of your application would be disappointed. Performance usually refers to the ability of a system to provide agreed upon response times for a defined number of users.

In the world of cloud, Scalability of applications becomes even more important. It's difficult to guess how successful a startup might be. Twitter or facebook might not have expected such success when they were incubated. Their success, for large measure, depends on how they were able to adapt to multi-fold increase in their user base without affecting performance.

Challenges with monolith applications

Over the last few years, in parallel to working with multiple small applications, I had the opportunity to work on four different monolith applications in varied domains - Insurance, Banking and Health care. All these applications had very similar challenges. In this section, we will start with looking at the characteristics of monoliths and then look at the challenges they bring in.

First of all: *What is Monolith?An application with a lot of code - may be greater than 100K lines of code?* Yeah.

For me, monoliths are those applications for which getting a release out to production is a big challenge. Applications that fall into this category have a number of user requirements that are immediately needed but these applications are able to do new feature releases once every few months. Some of these applications even do feature releases once a quarter or sometimes even as worse as twice a year.

Typically all monolith applications have these characteristics:

- Large size: Most of these monolith applications have more than 100K lines of code. Some have code bases with more than a Million lines of code.
- Large Teams: Team size could vary from 20 to 300.
- Multiple ways of doing same thing: Since team is huge, there is a communication gap. Results in multiple solutions for the same problem in different parts of the application.
- Lack of Automation Testing: Most of these applications have very few unit tests and complete lack of integration tests. These applications have great dependence on Manual Testing.

Because of these characteristics, there are a number of challenges faced by these Monolith applications.

Long release cycles

Making a code change in one part of the monolith may impact some other part of the monolith. Most code changes will need full regression cycle. This results in long release cycles.

Because there is lack of automation testing, these applications depend on manual testing to find defects. Taking functionality live is a major challenge.

Difficult to scale

Typically most monolith applications are not Cloud Native - meaning they are not easy to deploy on the cloud. They depend on manual installation and manual configuration. There is typically a lot of work put in by the operations team before a new application instance is added to the cluster. This makes scaling up and down a big challenge.

The other important challenge is large databases. Typically, monolith applications have databases running into **Terabytes** (**TB's**). The database becomes the bottleneck when scaling up.

Adapting new technologies

Most of the monolith applications use old technologies. Adding new technology to the monolith only makes it more complex to maintain. Architects and developers are reluctant to bring in any new technologies.

Adapting new methodologies

New methodologies like Agile need small (4-7 team members), independent teams. The big questions with monolith are: *How do we prevent teams from stepping on each other's toes?How to create islands that enable teams to work independently?* This is a difficult challenge to solve.

Adapting modern development practices

Modern development practices like **Test Driven Development**, **Behavior Driven Development** need loosely coupled, testable architecture. If the monolith application has tightly coupled layers and frameworks, it is difficult to unit test. It makes adapting modern development practices challenging.

Microservices

The challenges with monolith applications lead to organizations searching for the silver bullet. *How will we be able to take more features live more often?*.

Many organizations tried different architectures and practices to find a solution.

In the last few years, a common pattern emerged among all the organizations that were successful at doing this. From this emerged an architectural style that was called **Microservices Architecture**.

As Sam Newman says in the book *Building Microservices*:

"Many organizations found that embracing fine-grained, microservice architectures enable delivering software faster and adapt to new technologies".

What is Microservice?

One of the principles I love in software is to keep it small. This principles is applicable irrespective of what you are talking about - scope of a variable, size of a method, class, package or a component. You would want all of these to be as small as they possibly could be.

Microservices is a simple extension of this principle. It's an architectural style focused on building small capability-based independently deployable services.

There is no single accepted definition of a microservice. We will look at some of the popular definitions:

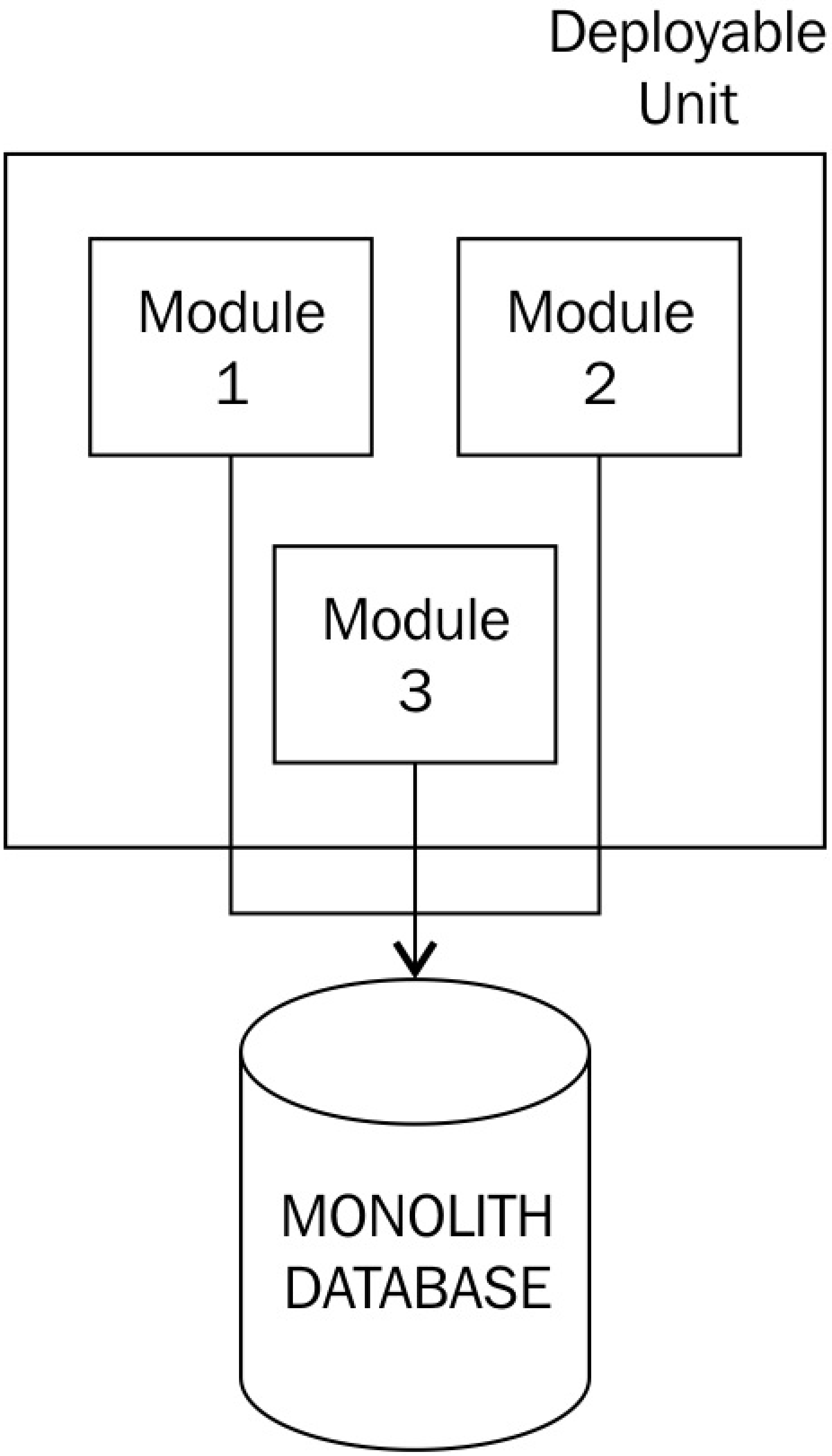
> *"Microservices are small, autonomous services that work together" - Sam Newman, Thoughtworks. "Loosely coupled service-oriented*

architecture with bounded contexts" - Adrian Cockcroft, Battery Ventures "A microservice is an independently deployable component of bounded scope that supports interoperability through message-based communication. Microservice architecture is a style of engineering highly automated, evolvable software systems made up of capability-aligned microservices" in the book Microservice Architecture - Irakli Nadareishvili, Ronnie Mitra, Matt McLarty

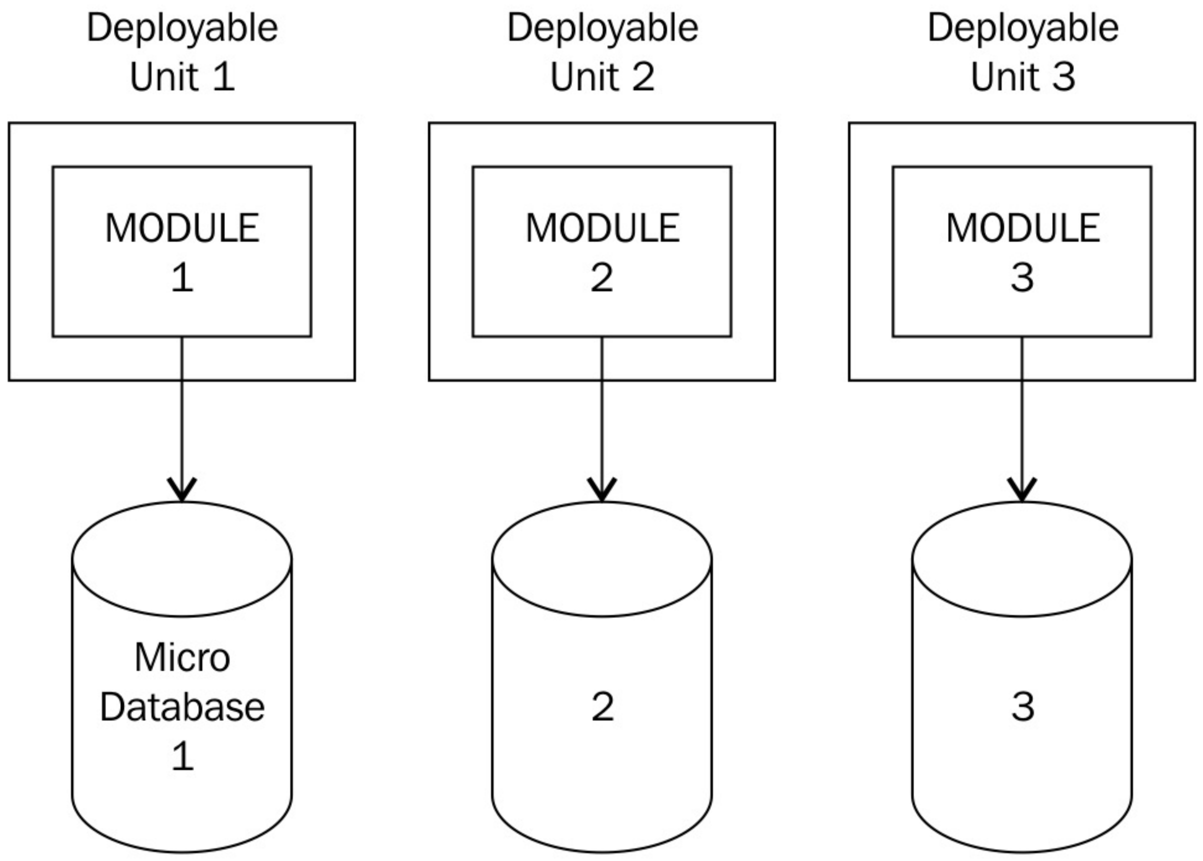
While there is no accepted definition, there are a few characteristics that are commonly featured in all definitions of microservice. Before we look at the characteristics of microservices, we will try and understand the big picture - we will look at how architecture with microservices compares with an architecture using microservices.

Microservice architecture

Monolith applications - even those that are modularized - have a single deployable unit. Below picture shows an example of a monolith application with three modules - **Module 1**, **Module 2**, & **Module 3**. These modules can be some business capability that is part of the monolith application. In a shopping application, one of the modules might be product recommendation:


Deployable Unit
Module 1
Module 2
Module 3
MONOLITH DATABASE

The proceeding screenshot shows what monolith applications looks like when developed using Microservice architecture:

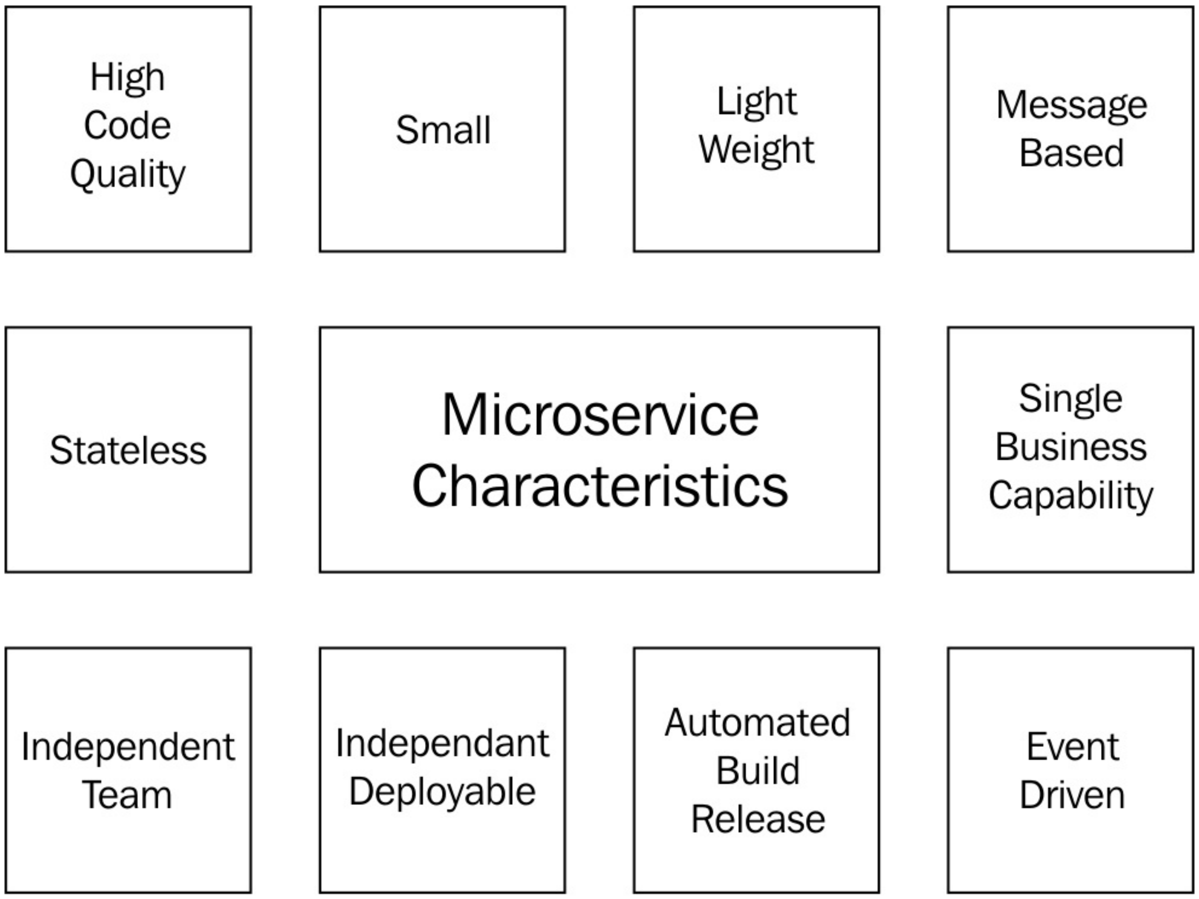


A few important things to note:

- Modules are identified based on business capabilities. *What functionality is the module providing?*
- Each module is independently deployable. In the example below, **Module 1**, **Module 2**, and **Module 3** are separate deployable units. If there is a change in business functionality of Module 3, we can individually build and deploy **Module 3**.

Microservice characteristics

In the previous section, we looked at an example of microservice architecture. An evaluation of experiences at organizations successful at adapting microservices architecture style reveals that there are a few characteristics that are shared by teams and architectures. Lets look at some of them:



It is Small, lightweight

A good microservice provides one business capability. Ideally microservices should follow single responsibility principle. Because of this, Microservices are generally small in size. Typically, a rule of thumb I use is that it should be possible to build and deploy a microservice with-in 5 minutes. If the build and deploy takes any longer, it is likely that you are building a larger than recommended microservice.

Some small and lightweight examples of good microservice are:

- Product Recommendation Service

- Email Notification Service
- Shopping Cart Service.

Interoperable with message based communication

Key focus of microservices is on Interoperability - communication between systems using diverse technologies. Best way to achieve interoperability is using message-based communication.

Capability aligned microservices

It is essential that microservices have clear boundary. Typically every microservice has a single identified business capability that it delivers well. Teams have found success adapting the "*Bounded Context*" concept proposed in the book *Domain-driven design* by Eric J. Evans.

Essentially, for large systems, it is very difficult to create one domain model. Evans talks about splitting the system into different bounded contexts. Identifying the right bounded contexts is the key to success with microservice architecture.

Independent deployable unit

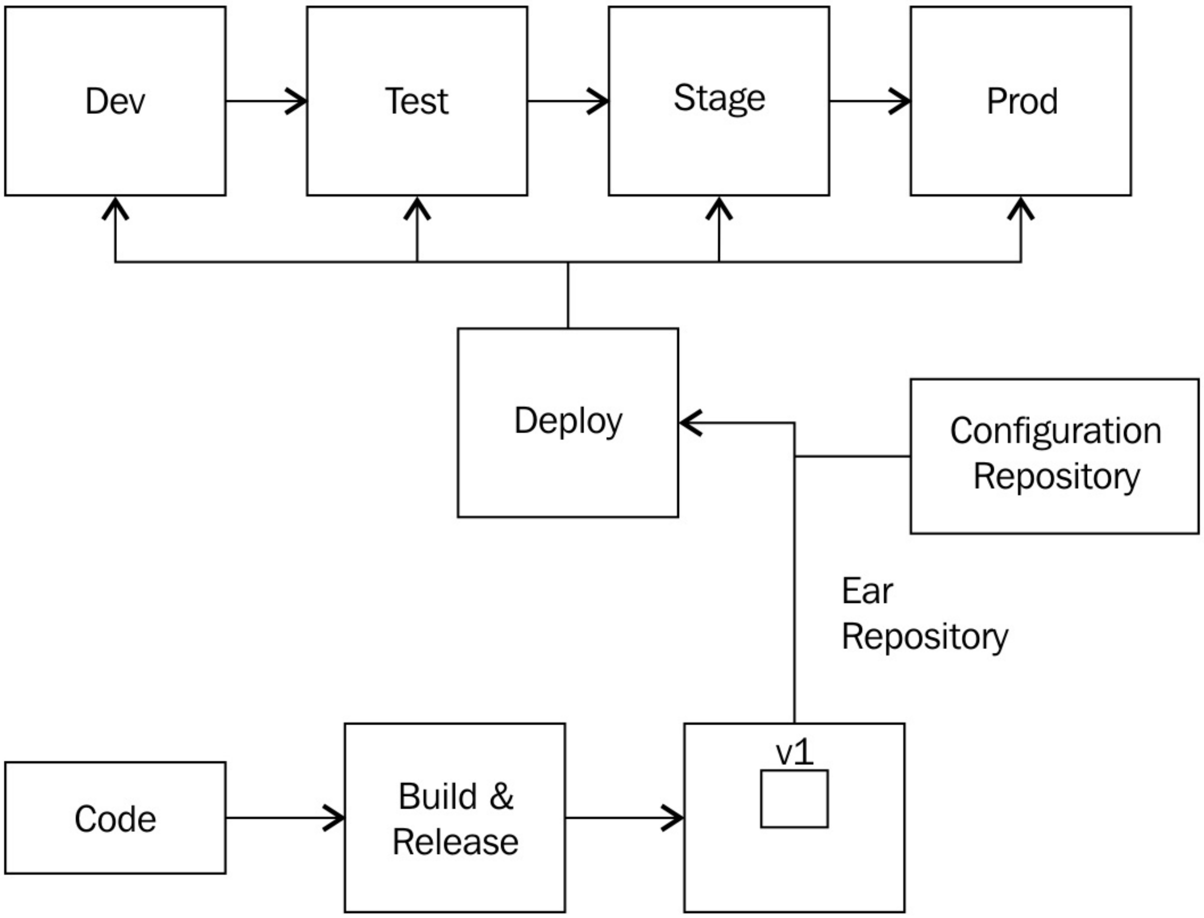
Each microservice can be individually built and deployed. In the example discussed before: **Module 1**, **Module 2**, and **Module 3** can each be independently built and deployed.

Stateless

An ideal microservice does not have state. It does not store any information between requests. All the information needed to create a response is present in the request.

Automated build and release process

Microservices have automated build and release processes. Consider the figure shown. It shows a simple build and release process for a microservice:



When a microservice is built and released, a version of microservice is stored in the repository. The deploy tool has the capability to pick the right version of microservice from the repository, match it with the configuration needed for the specific environment (from the configuration repository) and deploy the microservice to a specific environment.

Some teams take it a step further and combine the microservice package with the underlying infrastructure needed to run the microservice. The deploy tool will replicate this image and match it with environment specific configuration to create an environment.

Event driven

Microservices are typically built with event driven architecture. Lets consider a simple example: Whenever a new customer is registered, there are three things that need to be performed

- Store Customer information to database
- Mail a welcome kit
- Send an email notification

Lets look at two different approaches to design this:

Approach 1: Sequential Approach

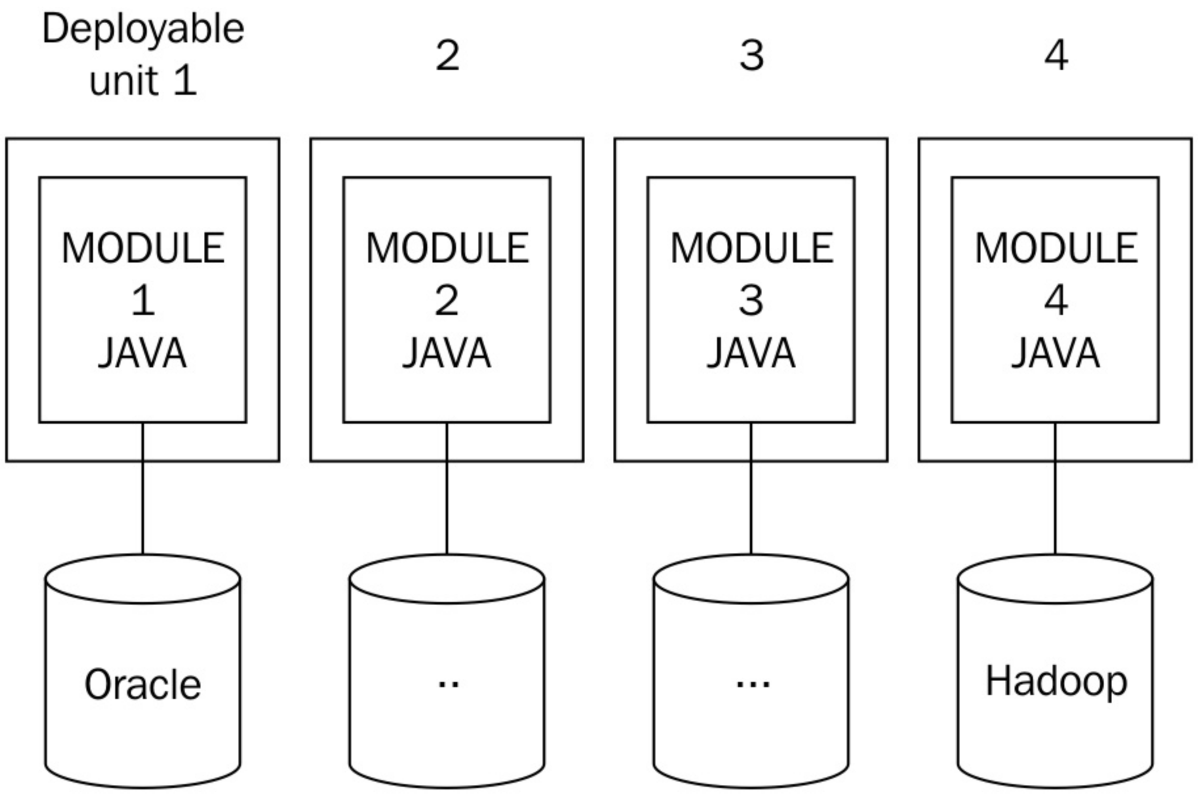
Lets consider three services Customer Information service, Mail service and Email service that can provide the capabilities listed above. We can create a New Customer service with following steps

- Call Customer Information service to save customer information to database
- Call Mail service to mail the welcome kit
- Call Email service to send the email notification

The New Customer service becomes the central place for all business logic. Imagine if we have to do more things when a new customer is created. All that logic would start accumulating and bloating up the New Customer service.

Approach 2: Event Driven Approach

In this approach we use a message broker. New Customer service would create a new event and post it to the message broker. Below picture shows a high level representation:



The three services Customer Information service, Mail service and Email service would be listening on the message broker for new events. When they see the new customer event, they process it and execute the functionality of that specific service.

The key advantage of event driven approach is that there is no centralized magnet for all business logic. Adding new functionality is easier. We can create a new service listening for the event on the **Message Broker**. Another important thing to note is that we don't need to make changes to any of the existing services.

Independent teams

Team developing a microservice is typically independent. It contains all the skills needed to develop, test and deploy a microservice. Typically the team is also responsible for supporting the microservice in productions.

Microservice advantages

The advantages of microservices are a follows:

Faster time to market

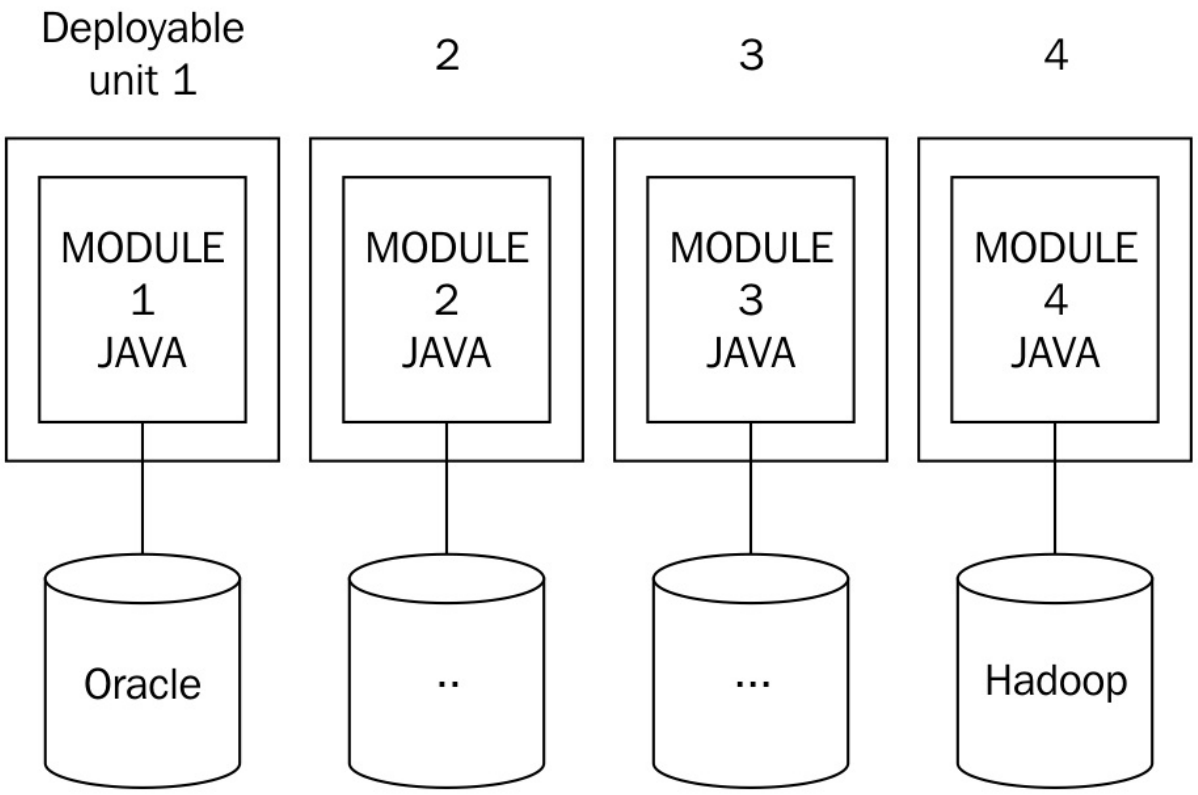
Faster time to market is one of the key differentiators and might determine the success of an organization.

Microservices architecture involves creating small, independently deployable components. Microservice enhancements are easier and less brittle because each microservice focuses on single business capability. All the steps in the process - Build, Release, Deployment, Testing, Configuration management, and Monitoring - are typically automated. Since the responsibility of a microservice is bounded, it is possible to write great automation unit and integration tests.

All above factors result in applications being able to react faster to customer needs.

Technology evolution

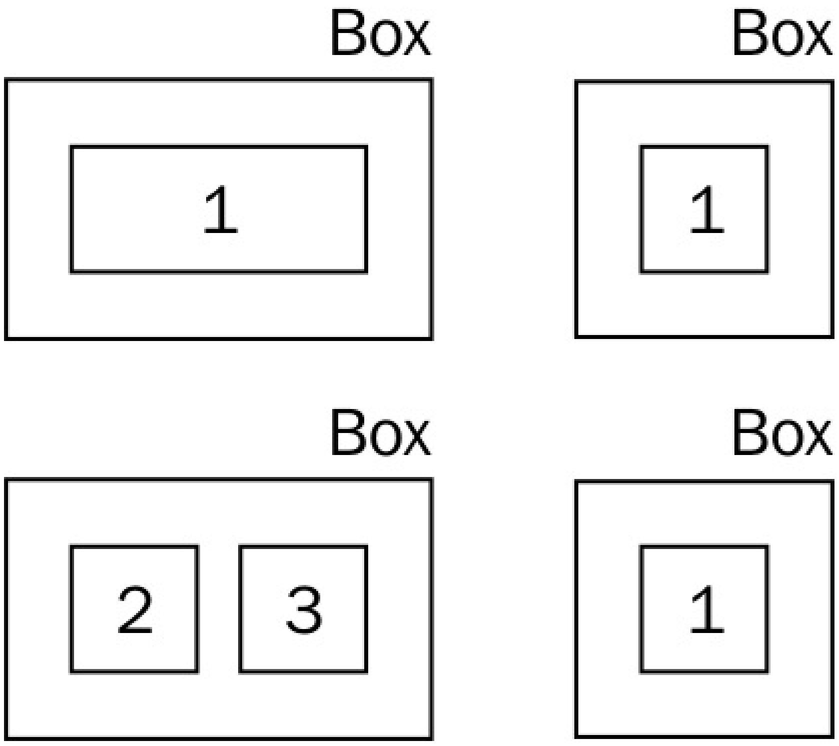
There are new languages, frameworks, practices and automation possibilities emerging every day. It is important that the application architectures allow flexibility to adapt to emerging possibilities. The following diagram shows how different services are developed in different technologies:



Microservice architecture involves creating small services. Within some boundaries, most organizations give the individual teams flexibility to make some of the technology decisions. This allows teams to experiment with new technologies and innovate faster. This helps applications to adapt and stay in tune with the technology evolution.

Availability and scaling

Load on different parts of application are typically very different. For example, in the case of a **Flight Booking** application, a customer typically searches multiple times before making a decision on whether to book a flight. The load on the **Search** module would typically be multiple times more than the load on the **Booking** module. Microservices architecture provides the flexibility of setting up multiple instances of Search service with lesser instances of the Booking service. The following figure shows how we can scale up specific microservices based on the load:



Microservices 2 and **Microservices 3** share a single box (deployment environment). **Microservice 1**, which has more load, is deployed into multiple boxes.

Another point in case is the need of startups. When a startup begins its operations, they are typically unaware of the extent to which they might grow. *What happens if the demand for applications grows very fast?*. If they adapt microservice architecture, it enables them to scale better when the need arises.

Team dynamics

Development methodologies like agile advocate small, independent teams. Since microservices are small, it is possible to build small teams around them. Typically teams are cross-functional with end-to-end ownership of specific microservices.

Microservice architecture fit in very well with agile and other modern development methodologies.

Microservice challenges

Microservice architecture has significant advantages. However, there are significant challenges too. Deciding the boundaries of microservices is a challenging but important decision. Since microservices are small and there would typically be hundreds of microservices in a large enterprise, having great automation and visibility is critical.

Increased need for automation

With microservice architecture, you are splitting up a large application into multiple microservices. So, the number of builds, releases and deployments increase multifold times. It would be very inefficient to have manual processes for these steps.

Test automation is critical to enable faster time to market. Teams should be focused on identifying automation possibilities as they emerge.

Defining the boundaries of subsystems

Microservices should be intelligent. Microservices are not dumb CRUD services. They should model a business capability of the system. They own all business logic in a bounded context. Having said this microservices should not be large. Deciding the boundaries of microservices is a challenge. Finding the right boundaries might be difficult at first go. It is important that as a team gains more knowledge about the business context, the knowledge flows into architecture and new boundaries are determined. Generally, find the right boundaries for microservices is an evolutionary process.

Couple of important points to note:

- **Loose coupling and high cohesion**: This is the fundamental to any programming and architectural decisions. When a system is loosely coupled, changes in one part should not require a change in other parts

- Bounded Contexts represent autonomous business modules representing specific business capabilities.

As Sam Newman says in the book Building Microservices - "Specific responsibility enforced by explicit boundaries". Always think, "What capabilities are we providing to the rest of the domain?"

Visibility and monitoring

With microservices, one application is split into several microservices. To conquer complexity associated with multiple microservices and asynchronous event based collaboration, it is important to have great visibility.

Ensuring high availability means each microservice should be monitored. Automated health management of microservice becomes important.

Debugging problems needs insights into what's happening behind multiple microservices. Centralized logging with aggregation of logs and metrics from different microservices is typically used. Mechanisms like **Correlation IDs** need to be used to isolate and debug issues.

Fault tolerance

Lets say we are building a shopping application. *What happens if the recommendations microservice is down?How does the application react? Does it completely crash? Or will it let the customer shop?* These kinds of situations happen more often as we adapt microservices architecture.

As we make the services small, the chance that some service is down increases. How the application reacts to these situations becomes an important question. In the previous example, a fault tolerant application would show some default recommendations while letting the customer to shop.

As we move into microservices architecture, applications should be more fault tolerant. Applications should be able to provide a toned down behavior when services are down.

Eventual consistency

It is important to have a degree of consistency between microservices in an organization. Consistency between microservices enables similar development, testing, release, deployment and operational processes across the organization. This enables different developers and testers to be productive when they move across teams. It is important to be not very rigid and have a degree of flexibility within limits so as to not stifle innovation.

Lets look at a few capabilities we believe have to be standardized at an enterprise level.

Shared Capabilities (Enterprise Level)

- Hardware: *What hardware do we use? Do we use cloud?*
- Code management: *What version control system do we use? What are our practices in branching and committing code?*
- Build and deployment: *How do we build?What tools do we use to automate deployment?*
- Datastore: *What kind of data stores do we use?*
- Service orchestration: *How do we orchestrate services? What kind of message broker do we use?*
- Security and Identity: *How do we authenticate and authorize users and services?*
- System visibility and monitoring: *How do we monitor our services? How do we provide fault isolation across the system?*

Increased need for Operations Team

As we move into a microservice world, there is a distinct shift in responsibilities of operation teams. The responsibilities shifts from manual things like executing release and deployments to identifying opportunities for automation.

With multiple microservices and increase in communications across different parts of the system, operations team becomes critical. It is important to involve operations as part of the team from the initial stages to enable them identifies solutions to make operations easier.

Cloud Native applications

Cloud is disrupting the world. A number of possibilities emerge that were never possible before. Organizations are able to provision computing, network and storage devices on demand. This has high potential to reduce costs in a number of industries.

Consider the retail industry where there is high demand in pockets (Black Friday, Holiday Season and so on.). *Why should they pay for hardware round the year when they could provision it on demand?*

While we would like to be benefit from the possibilities of the cloud, these possibilities are limited by architecture and the nature of applications.

How do we build applications that can be easily deployed on the cloud? That's where Cloud Native applications come into picture.

Cloud native applications are those that can easily be deployed on the cloud. These applications share a few common characteristics. We will begin with looking at Twelve- Factor app - A combination of common patterns among Cloud Native applications.

Twelve-factor app

Twelve-factor app evolved from experiences of engineers at Heroku. This is a list of patterns that are typically used in Cloud Native application architectures.

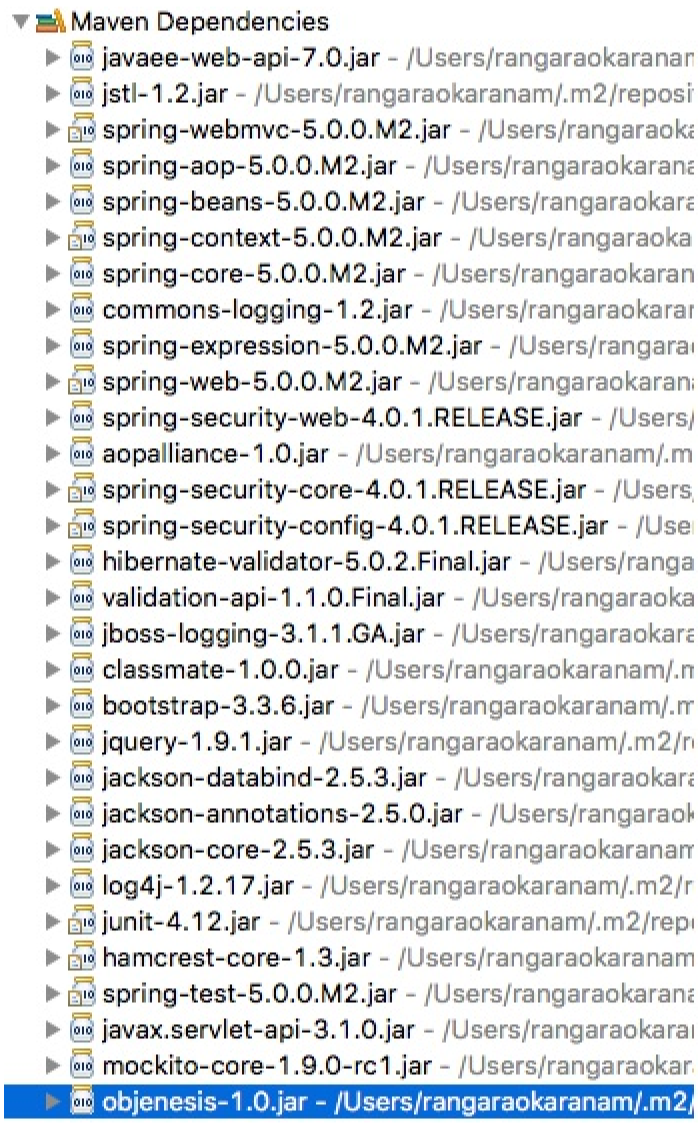
It is important to note that an App here refers to a single deployable unit. Essentially every microservice is an App (because each microservice is independently deployable).

One codebase

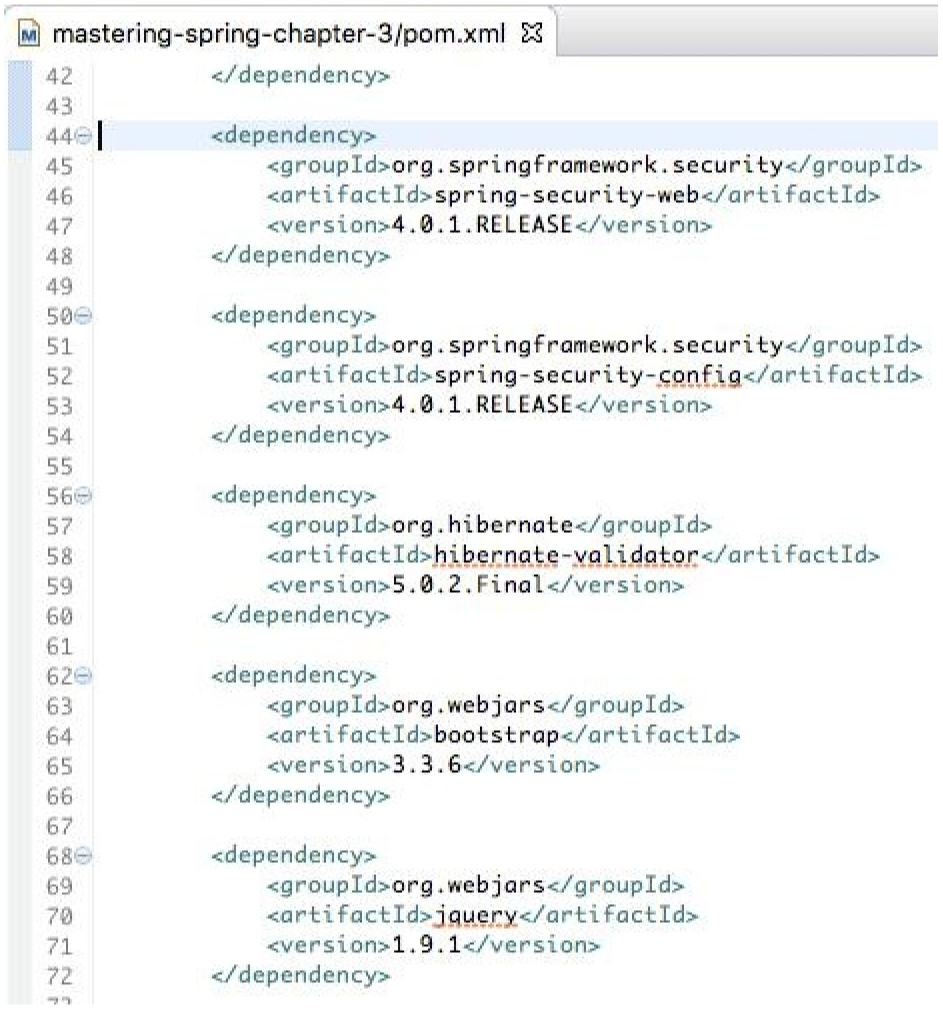
Each App has one codebase in revision control. There can be multiple environments where the App can be deployed. However, all these environments use code from a single codebase. An example anti-pattern is building a deployable from multiple codebases.

Dependencies

Explicitly declare and isolate dependencies. Typical Java applications use build management tools like **Maven** and **Gradle** to isolate and track dependencies. The following screenshot shows the typical Java applications managing dependencies using Maven:



```
Maven Dependencies
  javaee-web-api-7.0.jar - /Users/rangaraokaranar
  jstl-1.2.jar - /Users/rangaraokaranam/.m2/reposi
  spring-webmvc-5.0.0.M2.jar - /Users/rangaraoka
  spring-aop-5.0.0.M2.jar - /Users/rangaraokarana
  spring-beans-5.0.0.M2.jar - /Users/rangaraokara
  spring-context-5.0.0.M2.jar - /Users/rangaraoka
  spring-core-5.0.0.M2.jar - /Users/rangaraokaran
  commons-logging-1.2.jar - /Users/rangaraokarar
  spring-expression-5.0.0.M2.jar - /Users/rangara
  spring-web-5.0.0.M2.jar - /Users/rangaraokarana
  spring-security-web-4.0.1.RELEASE.jar - /Users/
  aopalliance-1.0.jar - /Users/rangaraokaranam/.m
  spring-security-core-4.0.1.RELEASE.jar - /Users
  spring-security-config-4.0.1.RELEASE.jar - /Use
  hibernate-validator-5.0.2.Final.jar - /Users/ranga
  validation-api-1.1.0.Final.jar - /Users/rangaraoka
  jboss-logging-3.1.1.GA.jar - /Users/rangaraokara
  classmate-1.0.0.jar - /Users/rangaraokaranam/.n
  bootstrap-3.3.6.jar - /Users/rangaraokaranam/.m
  jquery-1.9.1.jar - /Users/rangaraokaranam/.m2/r
  jackson-databind-2.5.3.jar - /Users/rangaraokara
  jackson-annotations-2.5.0.jar - /Users/rangaraok
  jackson-core-2.5.3.jar - /Users/rangaraokaranam
  log4j-1.2.17.jar - /Users/rangaraokaranam/.m2/r
  junit-4.12.jar - /Users/rangaraokaranam/.m2/rep
  hamcrest-core-1.3.jar - /Users/rangaraokaranam
  spring-test-5.0.0.M2.jar - /Users/rangaraokarana
  javax.servlet-api-3.1.0.jar - /Users/rangaraokara
  mockito-core-1.9.0-rc1.jar - /Users/rangaraokar
  objenesis-1.0.jar - /Users/rangaraokaranam/.m2/
```



```xml
        </dependency>

        <dependency>
            <groupId>org.springframework.security</groupId>
            <artifactId>spring-security-web</artifactId>
            <version>4.0.1.RELEASE</version>
        </dependency>

        <dependency>
            <groupId>org.springframework.security</groupId>
            <artifactId>spring-security-config</artifactId>
            <version>4.0.1.RELEASE</version>
        </dependency>

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

        <dependency>
            <groupId>org.webjars</groupId>
            <artifactId>bootstrap</artifactId>
            <version>3.3.6</version>
        </dependency>

        <dependency>
            <groupId>org.webjars</groupId>
            <artifactId>jquery</artifactId>
            <version>1.9.1</version>
        </dependency>
```

Config

All applications have configuration that varies from one environment to another environment. Configuration is typically littered at multiple locations - Application code, property files, databases, environment variables, **Java Naming and Directory Interface** (**JNDI**) and system variables are a few examples.

A Twelve-Factor app should store config in the environment. While environment variables are recommended to manage configuration in a Twelve factor app, other alternatives like having a centralized repository for application configuration should be considered for more complex systems.

Note

`Irrespective of mechanism used, we recommended to:`
Manage configuration outside application code (independent of the application deployable unit). Use one standardized way of configuration

Backing services

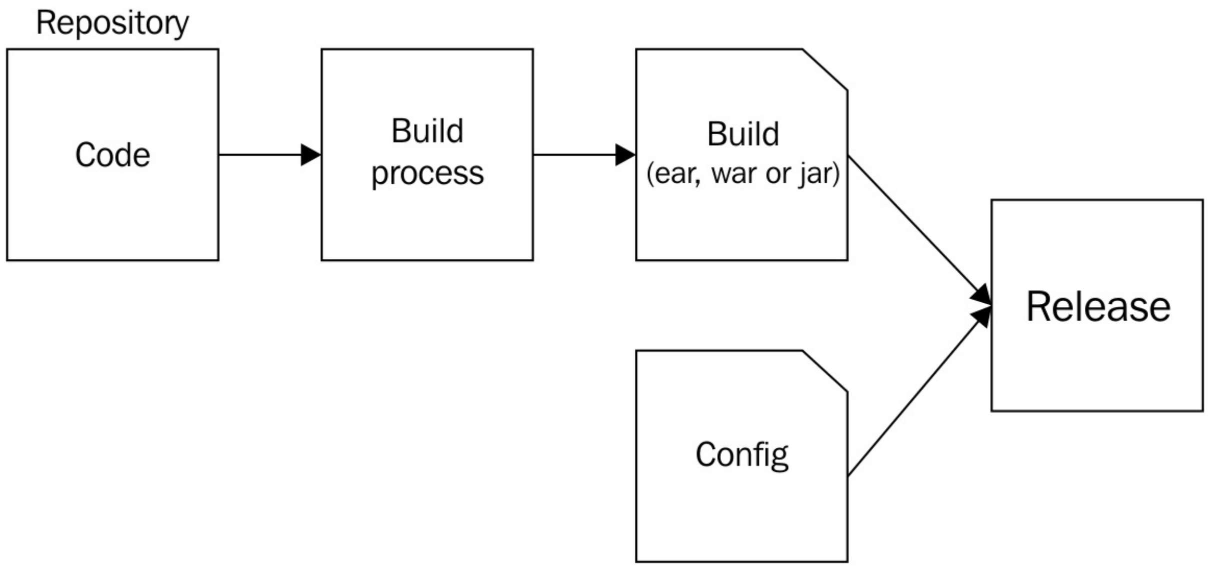
Typically applications depend on other services being available - data-stores, external services among others. Twelve factor app treats backing services as attached resources. A backing service is typically declared via an external configuration.

Loose coupling to a backing service has many advantages including ability to gracefully handle an outage of a backing service.

Build, release, run

Strictly separate build and run stages.

- **Build**: Creates an executable bundle (ear, war or jar) from code and dependencies that can be deployed to multiple environments.
- **Release**: Combine the executable bundle with specific environment configuration to deploy in an environment.
- **Run**: Run the app in an execution environment using a specific release



An anti-pattern is to build separate executable bundles specific for each environment.

Stateless

A Twelve-factor app does not have state. All data that it needs is stored in a persistent store. An anti-pattern is a sticky session.

Port binding

A Twelve factor app exposes all services using port binding. While it is possible to have other mechanisms to expose services, these mechanisms are implementation dependent. Port binding gives full control of receiving and handling messages irrespective of where an App is deployed.

Concurrency

A Twelve factor app is able to achieve more concurrency by scaling out horizontally. Scaling vertically has its limits. Scaling out horizontally provides opportunities to expand without limits.

Disposability

A Twelve factor app should promote elastic scaling. Hence, they should be disposable. They can be started and stopped when needed.

A Twelve factor app should

- Have minimum start up time. Long start up times means long delay before an application can take requests.
- Shutdown gracefully.
- Handle hardware failures gracefully.

Environment parity

All the environments - development, test, staging, and production - should be similar. They should use same processes and tools. With continuous deployment, they should have similar code very frequently. This makes finding and fixing problems easier.

Logs as event streams

Visibility is critical to a Twelve factor app. Since applications are deployed on the cloud and are automatically scaled, it is important to have a centralized visibility into what's happening across different instances of the applications.

Treating all logs as stream enables routing of the log stream to different destinations for viewing and archival. This stream can be used to debug issues, perform analytics and create alerting systems based on error patterns.

No distinction of admin processes

Twelve factor apps treat administrative tasks (migrations, scripts) similar to normal application processes.

Spring projects

As the world moves towards Cloud Native applications and microservices, Spring projects are not staying behind. There are a number of new Spring projects - Spring Boot, Spring Cloud among others - to solve the problems of the emerging world.

Spring Boot

In the era of monoliths, we had the luxury of taking time to set the frameworks up for an application. However, in the era of microservices, we would want to create individual components faster. Spring Boot project aims to solve this problem.

Note

As the official website highlights - Spring Boot makes it easy to create stand-alone, production-grade Spring based applications that you can "just run". We take an opinionated view of the Spring platform and third-party libraries so you can get started with minimum fuss.

Spring Boot aims to take an opinionated view - basically making a lot of decisions for us - to developing Spring based projects.

In the next couple of chapters, we will look at Spring Boot and the different features that enable creating production ready applications faster.

Spring Cloud

Spring Cloud aims to provide solutions to some commonly encountered patterns when building systems on the cloud.

- **Configuration Management**: As we discussed in the *Twelve factor app* section, managing configuration is an important part of developing Cloud Native applications. Spring Cloud provides a centralized configuration management solution for Microservices called **Spring Cloud Config**.
- **Service Discovery**: Service Discovery promotes loose coupling between services. Spring Cloud provides integration with popular service discovery options like Eureka, ZooKeeper and **Consul.**
- **Circuit Breakers**: Cloud Native Applications must be fault tolerant. They should be able to handle failure of backing services gracefully. Circuit breakers play a key role in providing default minimal service in case of failures. Spring Cloud provides integration with **Netflix Hystrix** fault tolerance library.
- **API Gateway**: An API gateway provides centralized aggregation, routing and caching services. Spring Cloud provides integration with the API Gateway library **Netflix Zuul.**

Summary

In this chapter, we looked at how the world evolved towards microservices and Cloud Native applications. We understood how Spring framework and projects are evolving to meeting the needs of today world with projects like Spring Boot, Spring Cloud and Spring Data.

In the next chapter, we will start focusing on Spring Boot. We will look at how Spring Boot makes developing microservices easy.

Chapter 4. Building Microservices with Spring Boot

As we discussed in the last chapter, we are moving towards architectures with smaller, independently deployable microservices. This would mean that there would be a huge number of smaller microservices developed.

An important consequence is that we would need to be able to quickly get off the ground and get running with new components.

Spring Boot aims to solve the problem of getting off fast with a new component. In this chapter, we will start with understanding the capabilities Spring Boot brings to the table. We will answer the following questions:

- Why Spring Boot?
- What are the features that Spring Boot provides?
- What is auto-configuration?
- What Spring Boot is not?
- What happens in the background when you use Spring Boot?
- How do you use Spring Initializr to create new Spring Boot projects?
- How to create basic RESTful services with Spring Boot?

What is Spring Boot?

First of all, let's start with clearing out a few misconceptions about Spring Boot:

- Spring Boot is not a code generation framework. It does not generate any code.
- Spring Boot is neither an application server nor is it a web server. It does provide good integration with different ranges of application and web servers.
- Spring Boot does not implement any specific frameworks or specifications.

The questions still remain:

- What is Spring Boot?
- Why has it become so popular in the last couple of years?

To answer these questions, let's build a quick example. Let's consider an example application that you want to quickly prototype.

Building a quick prototype for a microservice

Let's say we would want to build a microservice using Spring MVC with JPA, implemented with Hibernate, to connect with backend.

Let's consider the different steps in setting up such an application:

1. Decide which versions of Spring MVC, JPA and Hibernate to use.
2. Setup a Spring context to wire all the different layers together.
3. Setup web layer with Spring MVC (including Spring MVC configuration):
    - Configure beans for Dispatcher Servlet, Handler Resolvers, View Resolvers and so on

4. Setup Hibernate in data layer:
   - Configure beans for SessionFactory, data source and so on
5. Decide and implement how to store your application configuration--which varies between different environments.
6. Decide how you would want to do your unit testing.
7. Decide and implement your transaction management strategy.
8. Decide and implement how to implement security.
9. Setup your logging framework.
10. Decide and implement how you want to monitor your application in production.
11. Decide and implement a metrics management system to provide statistics about the application.
12. Decide and implement how to deploy your application to a web or an application server.

At least a few of the steps mentioned have to be completed before we can start with building our business logic. And this might take a few weeks at the least.

When we are building microservices, we would want to make a quick start. All the preceding steps will not make it easy to develop a microservice. And that's the problem Spring Boot aims to solve.

The following quote is an extract from Spring Boot website (http://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/htmlsingle/#boot-documentation)

> *"Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that you can "just run". We take an opinionated view of the Spring platform and third-party libraries so you can get started with minimum fuss. Most Spring Boot applications need very little Spring configuration".*

Spring Boot enables developers to focus on business logic behind their microservice. It aims to take care of all the nitty-gritty technical details involved in developing microservices.

Primary goals

Primary goals of Spring Boot are:

- Enable quickly getting off the ground with Spring based projects.
- Be opinionated. Make default assumptions based on common usage. Provide configuration options to handle deviations from defaults.
- Provide a wide range of non-functional features out of the box.
- Do not use code generation and avoid using a lot of XML Configuration.

Non functional features

Few of the non functional features provided by Spring Boot are:

- Default handling of versioning and configuration of wide range of frameworks, servers, and specifications
- Default options for application security
- Default application metrics with possibilities to extend
- Basic application monitoring using health checks
- Multiple options for externalized configuration

Spring Boot Hello World

We will start with building our first Spring Boot application in this chapter. We will use Maven to manage dependencies.

Following steps are involved in starting up with a Spring Boot application:

1. Configure `spring-boot-starter-parent` in your `pom.xml` file.
2. Configure `pom.xml` file with the required starter projects.
3. Configure `spring-boot-maven-plugin` to be able to run the application.
4. Create your first Spring Boot launch class.

Let's start with step 1: Configuring the starter projects.

Configure spring-boot-starter-parent

Let's start with a simple `pom.xml` file with the spring-boot-starter-parent.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mastering.spring</groupId>
  <artifactId>springboot-example</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>First Spring Boot Example</name>
  <packaging>war</packaging>
 <parent>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot- starter-parent</artifactId>
   <version>1.4.0.RELEASE</version>
 </parent>
 <properties>
   <java.version>1.8</java.version>
 </properties>
```

```
</project>
```

First question: Why do we need `spring-boot-starter-parent`?

`spring-boot-starter-parent` contains the default versions of Java to use, default versions of dependencies that Spring Boot uses and also the default configuration of the Maven plugins.

> **Note**
>
> spring-boot-starter-parent is the parent pom providing dependency and plugin management for Spring Boot based applications.

Let's look at some of the code inside the `spring-boot-starter-parent` to get a deeper understanding about it.

spring-boot-starter-parent

`spring-boot-starter-parent` inherits from Spring Boot dependencies which is defined at the top of the pom. Following code snippet shows an extract from spring-boot-starter-parent:

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-dependencies</artifactId>
  <version>1.4.0.RELEASE</version>
  <relativePath>../../spring-boot-dependencies</relativePath>
</parent>
```

Spring Boot dependencies provides default dependency management for all the dependencies that Spring Boot uses. The following code shows the different versions of various dependencies that are configured in Spring Boot dependencies:

```
<activemq.version>5.13.4</activemq.version>
<aspectj.version>1.8.9</aspectj.version>
<ehcache.version>2.10.2.2.21</ehcache.version>
<elasticsearch.version>2.3.4</elasticsearch.version>
<gson.version>2.7</gson.version>
```

```
<h2.version>1.4.192</h2.version>
<hazelcast.version>3.6.4</hazelcast.version>
<hibernate.version>5.0.9.Final</hibernate.version>
<hibernate-validator.version>5.2.4.Final</hibernate
  validator.version>
<hsqldb.version>2.3.3</hsqldb.version>
<htmlunit.version>2.21</htmlunit.version>
<jackson.version>2.8.1</jackson.version>
<jersey.version>2.23.1</jersey.version>
<jetty.version>9.3.11.v20160721</jetty.version>
<junit.version>4.12</junit.version>
<mockito.version>1.10.19</mockito.version>
<selenium.version>2.53.1</selenium.version>
<servlet-api.version>3.1.0</servlet-api.version>
<spring.version>4.3.2.RELEASE</spring.version>
<spring-amqp.version>1.6.1.RELEASE</spring-amqp.version>
<spring-batch.version>3.0.7.RELEASE</spring-batch.version>
<spring-data-releasetrain.version>Hopper-SR2</spring-
  data-releasetrain.version>
<spring-hateoas.version>0.20.0.RELEASE</spring-hateoas.versio
<spring-restdocs.version>1.1.1.RELEASE</spring-restdocs.versi
<spring-security.version>4.1.1.RELEASE</spring-security.versi
<spring-session.version>1.2.1.RELEASE</spring-session.version
<spring-ws.version>2.3.0.RELEASE</spring-ws.version>
<thymeleaf.version>2.1.5.RELEASE</thymeleaf.version>
<tomcat.version>8.5.4</tomcat.version>
<xml-apis.version>1.4.01</xml-apis.version>
```

If we want to override a specific version of a dependency, we can do that by providing a property with the right name in `pom.xml` file of our application. The code snippet shows an example of configuring our application to use version 1.10.20 of Mockito.

```
<properties>
  <mockito.version>1.10.20</mockito.version>
</properties>
```

Following are some of the other things defined in the spring-boot-starter-parent:

- Default Java version `<java.version>1.6</java.version>`
- Default configuration for maven plugins:
  - maven-failsafe-plugin
  - maven-surefire-plugin

- git-commit-id-plugin

Compatibility between different versions of frameworks is one of the major problems faced by developers. How do I find the latest Spring Session version that is compatible with a specific version of Spring? The usual answer would be to read the documentation. However, if we use Spring Boot, this is made simple by the spring boot starter parent. If we want to upgrade to a newer Spring version, all that we need to do is to find the Spring Boot Starter Parent for that Spring version. Once we upgrade our application to use that specific version of Spring Boot Starter Parent, we would have all the other dependencies upgraded to the versions compatible with new Spring version. One less problem for developers to handle! Always make me happy!

Configure pom.xml with the required starter projects

Whenever we would want to build an application in Spring Boot, we would need to start looking for starter projects. Let's focus on understanding what a starter project is.

Understanding starter projects

Starters are simplified dependency descriptors customized for different purposes. For example, `spring-boot-starter-web` is the starter for building web, including RESTful, applications using Spring MVC. It uses Tomcat as the default embedded container. If I want to develop a web application using Spring MVC, all that I would need to is include `spring-boot-starter-web` in my dependencies, and I get following automatically pre-configured:

- Spring MVC
- Compatible versions of jackson-databind (for binding), hibernate-validator (for form validation)
- `spring-boot-starter-tomcat` (starter project for Tomcat)

The following code snippet shows some of the dependencies configured in `spring-boot-starter-web`:

```
<dependencies>
  <dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId> <artifactId>s
      boot-starter-tomcat</artifactId> </dependency> <depende
    <groupId>org.hibernate</groupId> <artifactId>hibernate-
      validator</artifactId> </dependency> <dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
      <artifactId>jackson-databind</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId> <artifactId>spring
      web</artifactId> </dependency> <dependency>
    <groupId>org.springframework</groupId> <artifactId>spring
      webmvc</artifactId>
  </dependency>
</dependencies>
```

As we can see in the preceding snippet, when we use `spring-boot-starter-web`, we get a lot of frameworks auto-configured.

For the web application we would like to build, we would also want to do some good unit testing and we would want to deploy it on Tomcat. The following snippet shows the different starter dependencies that we would need. We would need to add this to our `pom.xml` file:

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

  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>

  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-tomcat</artifactId>
    <scope>provided</scope>
  </dependency>
```

```
</dependencies>
```

We are adding in three starter projects:

- We already discussed about `spring-boot-starter-web`. It provides us the frameworks need to build a web application with Spring MVC
- `spring-boot-starter-test` provides the following test frameworks needed for unit testing:
  - **JUnit**: Basic Unit Test Framework
  - **Mockito**: For mocking
  - **Hamcrest**, **AssertJ**: For readable asserts
  - **Spring Test**: Unit testing framework for spring context based applications
- `spring-boot-starter-tomcat` is the default for running web applications. We include it for clarity. `spring-boot-starter-tomcat` is the Starter for using Tomcat as the embedded servlet container

We now have our `pom.xml` file configured with starter parent and the required starter projects. Let's now add in the `spring-boot-maven-plugin` which would enable us to run Spring Boot applications.

Configure spring-boot-maven-plugin

When we build applications using Spring Boot, there are a couple of situations that are possible:

- We would want to run the applications in place without building a jar or a war
- Second one is to build a jar and a war for later deployment

`spring-boot-maven-plugin` provides capabilities for both of preceding situations. Following snippet shows how we can configure `spring-boot-maven-plugin` in an application:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
```

```
        </plugin>
      </plugins>
    </build>
```

`spring-boot-maven-plugin` provides several goals for Spring Boot application. The most popular goal is run (executed as `spring-boot:run`), which we will soon use to run the application we are building.

Create your first Spring Boot launch class

Following class how to create a simple Spring Boot launch class. It uses the static run method from `SpringApplication` class as shown in the following code snippet:

```
package com.mastering.spring.springboot;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplicati
import org.springframework.context.ApplicationContext;      @Spri
```

The preceding code is a simple Java main method executing the static run method on `SpringApplication` class.

SpringApplication

`SpringApplication` class can be used to bootstrap and launch a Spring application from a Java main method.

Following are the steps that are typically performed when a Spring Boot application is bootstrapped:

1. Create an instance of Spring `ApplicationContext`
2. Enable functionality for accepting command line arguments and exposing them as Spring properties
3. Load all the Spring beans, as per configuration

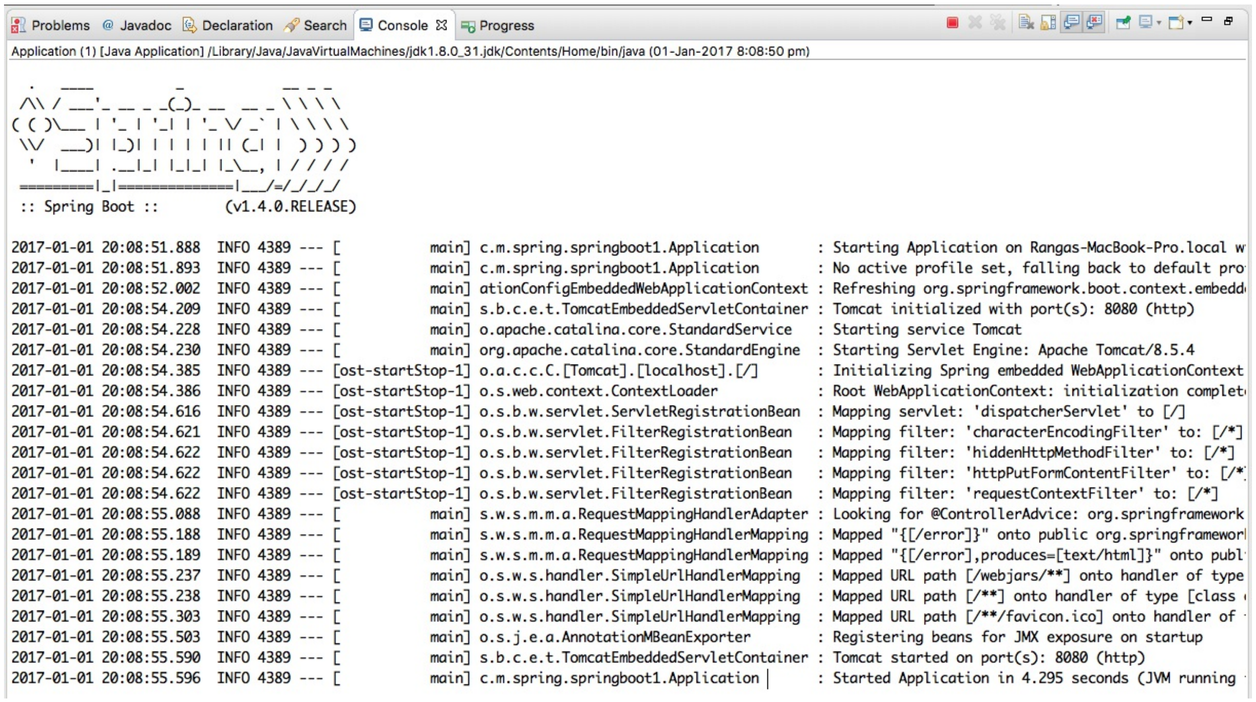
@SpringBootApplication annotation

`@SpringBootApplication` annotation is a shortcut for three annotations

- `@Configuration`: Indicates that this a Spring application context configuration file
- `@EnableAutoConfiguration`: Enables Auto Configuration - an important feature of Spring Boot. We will discuss about auto-configuration later is a separate section.
- `@ComponentScan`: Enables scanning for spring beans in the package of this class and all its sub-packages.

Running our Hello World application

We can run the `Hello World` application in multiple ways. Let's start running it with the simplest option : Running as a Java Application. In your IDE, right click on the Application class and run as Java Application. The following screenshot shows some of the log from running our `Hello World` application:



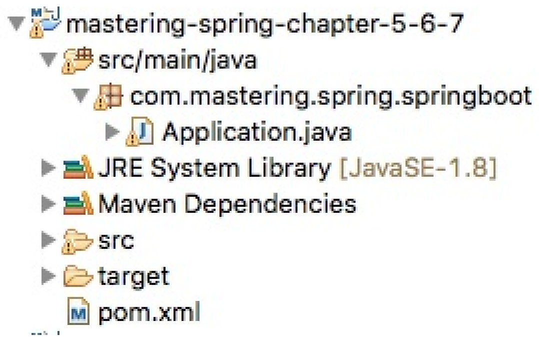
```
Application (1) [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_31.jdk/Contents/Home/bin/java (01-Jan-2017 8:08:50 pm)

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v1.4.0.RELEASE)

2017-01-01 20:08:51.888  INFO 4389 --- [           main] c.m.spring.springboot1.Application       : Starting Application on Rangas-MacBook-Pro.local w
2017-01-01 20:08:51.893  INFO 4389 --- [           main] c.m.spring.springboot1.Application       : No active profile set, falling back to default pro
2017-01-01 20:08:52.002  INFO 4389 --- [           main] ationConfigEmbeddedWebApplicationContext : Refreshing org.springframework.boot.context.embedd
2017-01-01 20:08:54.209  INFO 4389 --- [           main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat initialized with port(s): 8080 (http)
2017-01-01 20:08:54.228  INFO 4389 --- [           main] o.apache.catalina.core.StandardService   : Starting service Tomcat
2017-01-01 20:08:54.230  INFO 4389 --- [           main] org.apache.catalina.core.StandardEngine  : Starting Servlet Engine: Apache Tomcat/8.5.4
2017-01-01 20:08:54.385  INFO 4389 --- [ost-startStop-1] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
2017-01-01 20:08:54.386  INFO 4389 --- [ost-startStop-1] o.s.web.context.ContextLoader            : Root WebApplicationContext: initialization complet
2017-01-01 20:08:54.616  INFO 4389 --- [ost-startStop-1] o.s.b.w.servlet.ServletRegistrationBean  : Mapping servlet: 'dispatcherServlet' to [/]
2017-01-01 20:08:54.621  INFO 4389 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter: 'characterEncodingFilter' to: [/*]
2017-01-01 20:08:54.622  INFO 4389 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter: 'hiddenHttpMethodFilter' to: [/*]
2017-01-01 20:08:54.622  INFO 4389 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter: 'httpPutFormContentFilter' to: [/*
2017-01-01 20:08:54.622  INFO 4389 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter: 'requestContextFilter' to: [/*]
2017-01-01 20:08:55.088  INFO 4389 --- [           main] s.w.s.m.m.a.RequestMappingHandlerAdapter : Looking for @ControllerAdvice: org.springframework
2017-01-01 20:08:55.188  INFO 4389 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/error]}" onto public org.springframewor
2017-01-01 20:08:55.189  INFO 4389 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/error],produces=[text/html]}" onto publ
2017-01-01 20:08:55.237  INFO 4389 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMapping  : Mapped URL path [/webjars/**] onto handler of type
2017-01-01 20:08:55.238  INFO 4389 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMapping  : Mapped URL path [/**] onto handler of type [class
2017-01-01 20:08:55.303  INFO 4389 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMapping  : Mapped URL path [/**/favicon.ico] onto handler of
2017-01-01 20:08:55.503  INFO 4389 --- [           main] o.s.j.e.a.AnnotationMBeanExporter        : Registering beans for JMX exposure on startup
2017-01-01 20:08:55.590  INFO 4389 --- [           main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http)
2017-01-01 20:08:55.596  INFO 4389 --- [           main] c.m.spring.springboot1.Application       : Started Application in 4.295 seconds (JVM running
```

Following are the key things to note:

- Tomcat server is launched up on port 8080. `Tomcat started on port(s): 8080 (http)`

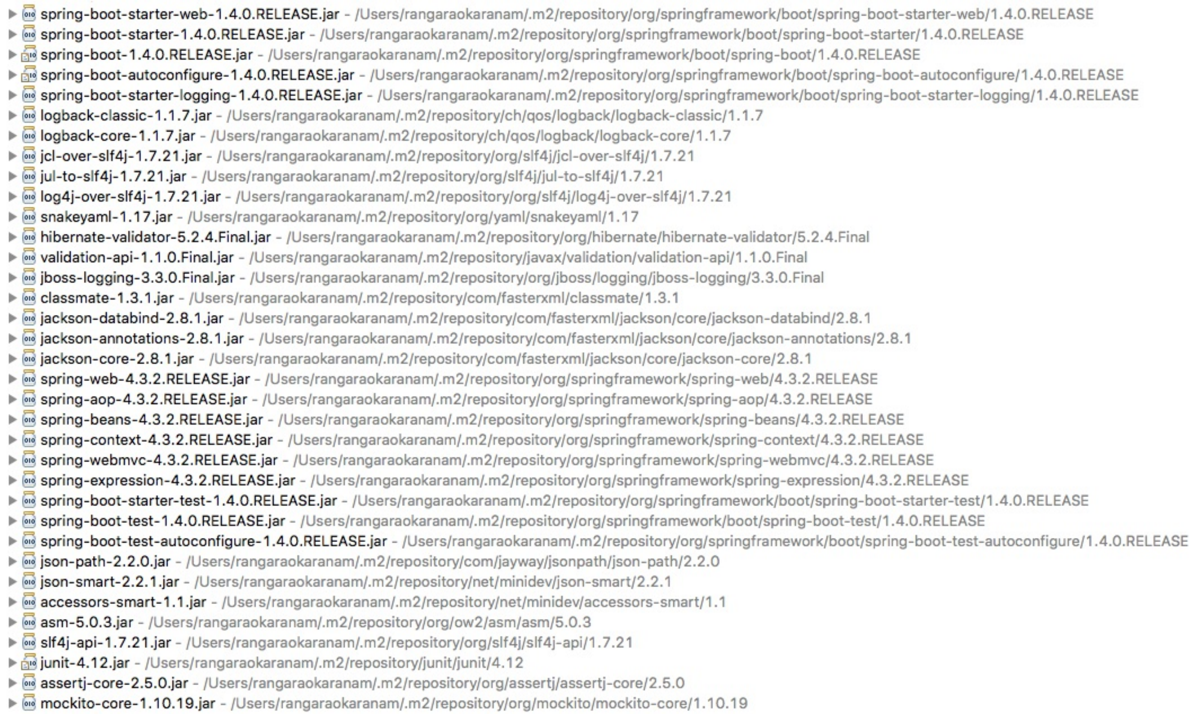
- Dispatcher Servlet is configured. This means Spring MVC framework is ready to accept requests. `Mapping servlet: 'dispatcherServlet' to [/]`
- Four filters - `characterEncodingFilter`, `hiddenHttpMethodFilter`, `httpPutFormContentFilter` and `requestContextFilter`--are enabled by default
- Default error page is configured. `Mapped "{[/error]}" onto public org.springframework.http.ResponseEntity<java.util.Map<java.la java.lang.Object>> org.springframework.boot.autoconfigure.web.BasicErrorControll`
- Web jars are auto-configured. As we discussed in Chapter 3, *Evolution towards Microservices and Cloud Native Applications*, WebJars enable dependency management for Static dependencies like bootstrap and query. `Mapped URL path [/webjars/**] onto handler of type [class org.springframework.web.servlet.resource.ResourceHttpRequestH`

The preceding image shows the application layout as of now. We have just two files - `pom.xml` and `Application.java`



```
mastering-spring-chapter-5-6-7
  src/main/java
    com.mastering.spring.springboot
      Application.java
  JRE System Library [JavaSE-1.8]
  Maven Dependencies
  src
  target
  pom.xml
```

With a simple `pom.xml` file and one java class, we were able to get to launch up Spring MVC application, with all the preceding functionality described. The most important thing about Spring Boot is to understand what happens in the background. Understanding the preceding start up log shown is the first. Let's look at the maven dependencies to get a deeper picture. The following screenshot shows some of the dependencies that are configured with the basic configuration in `pom.xml` file that we created.


```
spring-boot-starter-web-1.4.0.RELEASE.jar - /Users/rangaraokaranam/.m2/repository/org/springframework/boot/spring-boot-starter-web/1.4.0.RELEASE
spring-boot-starter-1.4.0.RELEASE.jar - /Users/rangaraokaranam/.m2/repository/org/springframework/boot/spring-boot-starter/1.4.0.RELEASE
spring-boot-1.4.0.RELEASE.jar - /Users/rangaraokaranam/.m2/repository/org/springframework/boot/spring-boot/1.4.0.RELEASE
spring-boot-autoconfigure-1.4.0.RELEASE.jar - /Users/rangaraokaranam/.m2/repository/org/springframework/boot/spring-boot-autoconfigure/1.4.0.RELEASE
spring-boot-starter-logging-1.4.0.RELEASE.jar - /Users/rangaraokaranam/.m2/repository/org/springframework/boot/spring-boot-starter-logging/1.4.0.RELEASE
logback-classic-1.1.7.jar - /Users/rangaraokaranam/.m2/repository/ch/qos/logback/logback-classic/1.1.7
logback-core-1.1.7.jar - /Users/rangaraokaranam/.m2/repository/ch/qos/logback/logback-core/1.1.7
jcl-over-slf4j-1.7.21.jar - /Users/rangaraokaranam/.m2/repository/org/slf4j/jcl-over-slf4j/1.7.21
jul-to-slf4j-1.7.21.jar - /Users/rangaraokaranam/.m2/repository/org/slf4j/jul-to-slf4j/1.7.21
log4j-over-slf4j-1.7.21.jar - /Users/rangaraokaranam/.m2/repository/org/slf4j/log4j-over-slf4j/1.7.21
snakeyaml-1.17.jar - /Users/rangaraokaranam/.m2/repository/org/yaml/snakeyaml/1.17
hibernate-validator-5.2.4.Final.jar - /Users/rangaraokaranam/.m2/repository/org/hibernate/hibernate-validator/5.2.4.Final
validation-api-1.1.0.Final.jar - /Users/rangaraokaranam/.m2/repository/javax/validation/validation-api/1.1.0.Final
jboss-logging-3.3.0.Final.jar - /Users/rangaraokaranam/.m2/repository/org/jboss/logging/jboss-logging/3.3.0.Final
classmate-1.3.1.jar - /Users/rangaraokaranam/.m2/repository/com/fasterxml/classmate/1.3.1
jackson-databind-2.8.1.jar - /Users/rangaraokaranam/.m2/repository/com/fasterxml/jackson/core/jackson-databind/2.8.1
jackson-annotations-2.8.1.jar - /Users/rangaraokaranam/.m2/repository/com/fasterxml/jackson/core/jackson-annotations/2.8.1
jackson-core-2.8.1.jar - /Users/rangaraokaranam/.m2/repository/com/fasterxml/jackson/core/jackson-core/2.8.1
spring-web-4.3.2.RELEASE.jar - /Users/rangaraokaranam/.m2/repository/org/springframework/spring-web/4.3.2.RELEASE
spring-aop-4.3.2.RELEASE.jar - /Users/rangaraokaranam/.m2/repository/org/springframework/spring-aop/4.3.2.RELEASE
spring-beans-4.3.2.RELEASE.jar - /Users/rangaraokaranam/.m2/repository/org/springframework/spring-beans/4.3.2.RELEASE
spring-context-4.3.2.RELEASE.jar - /Users/rangaraokaranam/.m2/repository/org/springframework/spring-context/4.3.2.RELEASE
spring-webmvc-4.3.2.RELEASE.jar - /Users/rangaraokaranam/.m2/repository/org/springframework/spring-webmvc/4.3.2.RELEASE
spring-expression-4.3.2.RELEASE.jar - /Users/rangaraokaranam/.m2/repository/org/springframework/spring-expression/4.3.2.RELEASE
spring-boot-starter-test-1.4.0.RELEASE.jar - /Users/rangaraokaranam/.m2/repository/org/springframework/boot/spring-boot-starter-test/1.4.0.RELEASE
spring-boot-test-1.4.0.RELEASE.jar - /Users/rangaraokaranam/.m2/repository/org/springframework/boot/spring-boot-test/1.4.0.RELEASE
spring-boot-test-autoconfigure-1.4.0.RELEASE.jar - /Users/rangaraokaranam/.m2/repository/org/springframework/boot/spring-boot-test-autoconfigure/1.4.0.RELEASE
json-path-2.2.0.jar - /Users/rangaraokaranam/.m2/repository/com/jayway/jsonpath/json-path/2.2.0
json-smart-2.2.1.jar - /Users/rangaraokaranam/.m2/repository/net/minidev/json-smart/2.2.1
accessors-smart-1.1.jar - /Users/rangaraokaranam/.m2/repository/net/minidev/accessors-smart/1.1
asm-5.0.3.jar - /Users/rangaraokaranam/.m2/repository/org/ow2/asm/asm/5.0.3
slf4j-api-1.7.21.jar - /Users/rangaraokaranam/.m2/repository/org/slf4j/slf4j-api/1.7.21
junit-4.12.jar - /Users/rangaraokaranam/.m2/repository/junit/junit/4.12
assertj-core-2.5.0.jar - /Users/rangaraokaranam/.m2/repository/org/assertj/assertj-core/2.5.0
mockito-core-1.10.19.jar - /Users/rangaraokaranam/.m2/repository/org/mockito/mockito-core/1.10.19
```

Spring Boot does a lot of magic. Once you have the application configured and running, I recommend playing around with it to gain deeper understanding which would be useful when you are debugging problems.

As Spider Man says

"With great power comes great responsibility"

. It is absolutely true in case of Spring Boot. In time to come, the best developers with Spring Boot would be the ones who understand what happens in the background--dependencies and auto-configuration.

Auto-configuration

To enable us to understand auto-configuration further, let's expand our application class to include a few more lines of code:

```
ApplicationContext ctx = SpringApplication.run(Application.cl
  args);
String[] beanNames = ctx.getBeanDefinitionNames();
Arrays.sort(beanNames);
```

```
    for (String beanName : beanNames) {
      System.out.println(beanName);
    }
```

We are getting all the beans that are defined in the Spring application context and printing their names. When `Application.java` is run as a Java program, it prints the list of beans as shown in the following output :

```
application
basicErrorController
beanNameHandlerMapping
beanNameViewResolver
characterEncodingFilter
conventionErrorViewResolver
defaultServletHandlerMapping
defaultViewResolver
dispatcherServlet
dispatcherServletRegistration
duplicateServerPropertiesDetector
embeddedServletContainerCustomizerBeanPostProcessor
error
errorAttributes
errorPageCustomizer
errorPageRegistrarBeanPostProcessor
faviconHandlerMapping
faviconRequestHandler
handlerExceptionResolver
hiddenHttpMethodFilter
httpPutFormContentFilter
httpRequestHandlerAdapter
jacksonObjectMapper
jacksonObjectMapperBuilder
jsonComponentModule
localeCharsetMappingsCustomizer
mappingJackson2HttpMessageConverter
mbeanExporter
mbeanServer
messageConverters
multipartConfigElement
multipartResolver
mvcContentNegotiationManager
mvcConversionService
mvcPathMatcher
mvcResourceUrlProvider
mvcUriComponentsContributor
mvcUrlPathHelper
```

```
mvcValidator
mvcViewResolver
objectNamingStrategy
autoconfigure.AutoConfigurationPackages
autoconfigure.PropertyPlaceholderAutoConfiguration
autoconfigure.condition.BeanTypeRegistry
autoconfigure.context.ConfigurationPropertiesAutoConfiguration
autoconfigure.info.ProjectInfoAutoConfiguration
autoconfigure.internalCachingMetadataReaderFactory
autoconfigure.jackson.JacksonAutoConfiguration
autoconfigure.jackson.JacksonAutoConfiguration$Jackson2ObjectMapp
autoconfigure.jackson.JacksonAutoConfiguration$JacksonObjectMappe
autoconfigure.jackson.JacksonAutoConfiguration$JacksonObjectMappe
autoconfigure.jmx.JmxAutoConfiguration
autoconfigure.web.DispatcherServletAutoConfiguration
autoconfigure.web.DispatcherServletAutoConfiguration$DispatcherSe
autoconfigure.web.DispatcherServletAutoConfiguration$DispatcherSe
autoconfigure.web.EmbeddedServletContainerAutoConfiguration
autoconfigure.web.EmbeddedServletContainerAutoConfiguration$Embed
autoconfigure.web.ErrorMvcAutoConfiguration
autoconfigure.web.ErrorMvcAutoConfiguration$WhitelabelErrorViewCo
autoconfigure.web.HttpEncodingAutoConfiguration
autoconfigure.web.HttpMessageConvertersAutoConfiguration
autoconfigure.web.HttpMessageConvertersAutoConfiguration$StringHt
autoconfigure.web.JacksonHttpMessageConvertersConfiguration
autoconfigure.web.JacksonHttpMessageConvertersConfiguration$Mappi
autoconfigure.web.MultipartAutoConfiguration
autoconfigure.web.ServerPropertiesAutoConfiguration
autoconfigure.web.WebClientAutoConfiguration
autoconfigure.web.WebClientAutoConfiguration$RestTemplateConfigur
autoconfigure.web.WebMvcAutoConfiguration
autoconfigure.web.WebMvcAutoConfiguration$EnableWebMvcConfigurati
autoconfigure.web.WebMvcAutoConfiguration$WebMvcAutoConfiguration
autoconfigure.web.WebMvcAutoConfiguration$WebMvcAutoConfiguration
autoconfigure.websocket.WebSocketAutoConfiguration
autoconfigure.websocket.WebSocketAutoConfiguration$TomcatWebSocke
context.properties.ConfigurationPropertiesBindingPostProcessor
context.properties.ConfigurationPropertiesBindingPostProcessor.st
annotation.ConfigurationClassPostProcessor.enhancedConfigurationP
annotation.ConfigurationClassPostProcessor.importAwareProcessor
annotation.internalAutowiredAnnotationProcessor
annotation.internalCommonAnnotationProcessor
annotation.internalConfigurationAnnotationProcessor
annotation.internalRequiredAnnotationProcessor
event.internalEventListenerFactory
event.internalEventListenerProcessor
preserveErrorControllerTargetClassPostProcessor
propertySourcesPlaceholderConfigurer
```

```
requestContextFilter
requestMappingHandlerAdapter
requestMappingHandlerMapping
resourceHandlerMapping
restTemplateBuilder
serverProperties
simpleControllerHandlerAdapter
spring.http.encoding-autoconfigure.web.HttpEncodingProperties
spring.http.multipart-autoconfigure.web.MultipartProperties
spring.info-autoconfigure.info.ProjectInfoProperties
spring.jackson-autoconfigure.jackson.JacksonProperties
spring.mvc-autoconfigure.web.WebMvcProperties
spring.resources-autoconfigure.web.ResourceProperties
standardJacksonObjectMapperBuilderCustomizer
stringHttpMessageConverter
tomcatEmbeddedServletContainerFactory
viewControllerHandlerMapping
viewResolver
websocketContainerCustomizer
```

Important things to think about are:

- Where are these beans defined?
- How are these beans being created?

That's the magic of Spring auto-configuration.

Whenever we add a new dependency to a Spring Boot project, Spring Boot Auto Configuration automatically tries to configure the beans based on the dependency.

For example when we add a dependency on `spring-boot-starter-web`, following beans are auto-configured:

- `basicErrorController`, `handlerExceptionResolver`: Basic exception handling. Shows a default error page when an exception occurs.
- `beanNameHandlerMapping`: Used for resolving paths to a handler (Controller).
- `characterEncodingFilter`: Provides Default character encoding UTF-8.
- `dispatcherServlet`: Dispatcher Servlet is the front controller in Spring MVC Applications.

- `jacksonObjectMapper`: Translating object to JSON and JSON to objects in REST services.
- `messageConverters`: Default message converters to convert from object to XML or JSON and vice versa.
- `multipartResolver`: Provides support for uploading files in web applications.
- `mvcValidator`: Support validation of HTTP Requests.
- `viewResolver`: Resolves logical view name to a physical view.
- `propertySourcesPlaceholderConfigurer`: Supports externalisation of application configuration.
- `requestContextFilter`: Defaults filter for requests.
- `restTemplateBuilder`: Used to make calls to REST services.
- `tomcatEmbeddedServletContainerFactory`: Tomcat is the default embedded servlet container for Spring Boot based web applications.

In the next section, let's look at some of the starter projects and the auto-configuration they provide.

Starter Projects

The following table shows some of the important starter projects provided by Spring Boot:

Starter	Description
spring-boot-starter-web-services	Starter project for developing XML based web services
spring-boot starter-	Starter project for building Spring MVC based web applications or RESTful applications. This uses Tomcat as the default

web	embedded servlet container.
spring-boot starter-activemq	Supports message based communication using JMS on ActiveMQ.
spring-boot starter-integration	Supports Spring Integration - Framework that provides implementations for Enterprise Integration Patterns.
spring-boot starter-test	Provides support for various unit testing frameworks - JUnit, Mockito and Hamcrest matchers.
spring-boot starter-jdbc	Provides support for using Spring JDBC. Configures a Tomcat JDBC connection pool by default.
spring-boot starter-validation	Provides support for Java Bean Validation API. Default implementation is Hibernate Validator.
spring-boot starter-hateoas	HATEOAS stands for Hypermedia as the Engine of Application State. RESTful services that use HATEOAS return links to additional resources that are related to the current context in addition to data.

spring-boot starter-jersey	JAX-RS is the Java EE standard for developing REST APIs. Jersey is the default implementation. This starter project provides support for building JAX-RS based REST APIs.
spring-boot starter-websocket	HTTP is stateless. Web sockets allow maintaining connection between server and browser. This starter project provides support for Spring WebSockets.
spring-boot starter-aop	Provides support for Aspect Oriented Programming. Also provides support for AspectJ for advanced Aspect Oriented Programming.
spring-boot starter-amqp	With default as RabbitMQ, this starter projects provides message passing with AMQP.
spring-boot starter-security	This starter project enables auto-configuration for Spring Security.
spring-boot starter-data-jpa	Provides support for Spring Data JPA. Default implementation is Hibernate.
spring-boot-starter	Base Starter for Spring Boot Applications. Provides support for Auto-configuration and logging.

spring-boot starter-batch	Provides support for developing batch applications using Spring Batch.
spring-boot starter-cache	Basic support for Caching using Spring Framework.
spring-boot starter-data rest	Support for exposing REST services using Spring Data REST.

Until now, we have setup a basic web application and understood some of the the important concepts related to Spring Boot:

- Auto Configuration
- Starter Projects
- Spring Boot maven plugin
- Spring Boot starter Parent and
- Annotation `@SpringBootApplication`

Now lets shift our focus to understanding what is REST and building a REST Service.

What is REST?

Representational State Transfer (**REST**) is basically an architectural style for the web. REST specifies a set of constraints. These constraints ensure that clients (service consumers and browsers) can interact with servers in flexible ways.

Let's first understand some common terminology:

- **Server**: Service Provider. Exposes services which can be consumed by clients.
- **Client**: Service Consumer. Could be a browser or another system.
- **Resource**: Any information can be a resource. A Person, an image, a video or a product you want to sell.
- **Representation**: Representation is a specific way a resource can be represented. For example, the Product resource can be represented using JSON, XML or HTML. Different clients might request different representations of the resource.

Some of the important REST Constraints are listed below:

- **Client - Server** : There should be a server (service provider) and a client (service consumer). This enables loose coupling and independent evolution of server and client as new technologies emerge.
- **Stateless** : Each service should be stateless. Subsequent requests should not depend on some data from a previous request being temporarily stored. Messages should be self descriptive. Uniform Interface: Each Resource has a resource identifier. In the case of web services, we use URI. ex: /users/UserName/todos/1
- **Cacheable**: The service response should be Cacheable. Each response should indicate if it is cacheable.
- **Layered system**: Consumer of the service should not assume direct connection to the Service Provider. Since requests can be cached, the client might be getting the cached response from a middle layer.

- **Manipulation of resources through representations**: A resource can have multiple representations. It should be possible to modify the resource through a message with any of these representations.
- **Hypermedia as the engine of application state** (**HATEOAS**): Consumer of a RESTful application should know about only one fixed service URL. All subsequent resources should be discoverable from the links included in the resource representations.

The initial services we develop will not be adhering to all these constraints. As we move into the next chapters, we introduce you to the details of these constraints and add them to the services to make them more RESTful.

First REST Service

Let's start with creating a simple REST Service return a welcome message. We will create a simple POJO `WelcomeBean` with a member field called message and one argument constructor as shown in the following code snippet:

```
package com.mastering.spring.springboot.bean;

public class WelcomeBean {
  private String message;

public WelcomeBean(String message) {
  super();
  this.message = message;
}

public String getMessage() {
  return message;
}
}
```

Simple Method returning String

Let's start with creating a Simple REST Controller method returning a String:

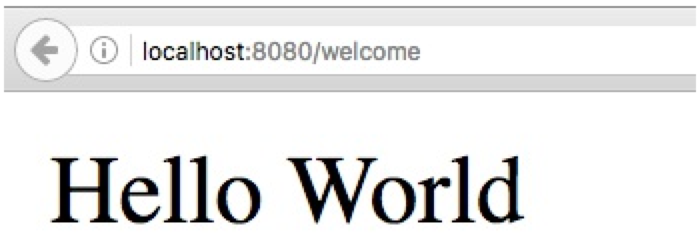
```
@RestController
public class BasicController {
  @GetMapping("/welcome")
  public String welcome() {
    return "Hello World";
  }
}
```

Few important things to note:

- `@RestController`: `@RestController` annotation provides a combination of `@ResponseBody` and `@Controller` annotations. This is typically used to create REST Controllers.

- `@GetMapping("welcome")`: `@GetMapping` is a shortcut for `@RequestMapping(method = RequestMethod.GET)`. This annotation is a readable alternative. The method with this annotation would handle a Get request to URI "`welcome`".

If we run the `Application.java` as a Java application, it would start up the embedded Tomcat container. We can launch up the URL in the browser as shown in the following screenshot:


localhost:8080/welcome

Hello World

Unit Testing

Let's quickly write a unit test to test the above controller method:

```
@RunWith(SpringRunner.class)
@WebMvcTest(BasicController.class)
public class BasicControllerTest {

  @Autowired
  private MockMvc mvc;

  @Test
  public void welcome() throws Exception {
    mvc.perform(
      MockMvcRequestBuilders.get("/welcome")
        .accept(MediaType.APPLICATION_JSON))
        .andExpect(status().isOk())
        .andExpect(content().string(
        equalTo("Hello World")));
  }
}
```

In the above unit test, we would launch up a Mock MVC instance with

`BasicController`. Few quick things to note:

- `@RunWith(SpringRunner.class)`: SpringRunner is a short cut to annotation SpringJUnit4ClassRunner. This launches up a simple spring context for unit testing.
- `@WebMvcTest(BasicController.class)`: This annotation can be used along with SpringRunner for writing simple tests for Spring MVC Controllers. This will only load the beans annotated with Spring MVC Related annotations. In this examples, we are launching up a Web Mvc Test context with the class under test BasicController.
- `@Autowired private MockMvc mvc`: Autowires the MockMvc bean which can be used to make requests.
- `mvc.perform(MockMvcRequestBuilders.get("/welcome").accept(Med` Perform a request to "/welcome" with 'Accept' header value 'application/json'.
- `andExpect(status().isOk())`: Expect that the status of the response is 200 (Success).
- `andExpect(content().string(equalTo("Hello World")))`: Expect that the content of the response is equal to "Hello World".

Integration Testing

When we do integration testing, we would want to launch up the embedded server with all controllers and beans that are configured. Below code shows how we can create a simple integration test as shown in the following code snippet:

```
@RunWith(SpringRunner.class)
@SpringBootTest(classes = Application.class,
  webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class BasicControllerIT {

  private static final String LOCAL_HOST = "http://localhost:

  @LocalServerPort
  private int port;

  private TestRestTemplate template = new TestRestTemplate();

  @Test
  public void welcome() throws Exception {
    ResponseEntity<String> response = template
     .getForEntity(createURL("/welcome"), String.class);
    assertThat(response.getBody(), equalTo("Hello World"));
  }

  private String createURL(String uri) {
    return LOCAL_HOST + port + uri;
  }
}
```

A few important things to note:

- `@SpringBootTest(classes = Application.class, webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)`: Provides additional functionality on top of the Spring Test Context. Provides support for configuring the port for fully running container and TestRestTemplate (to execute requests).
- `@LocalServerPort private int port`: `SpringBootTest` would ensure that the port on which the container is running is auto wired into the port variable.
- `private String createURL(String uri)`: Method to append the local host url and port to the URI to create a full URL.
- `private TestRestTemplate template = new TestRestTemplate()`: `TestRestTemplate` is typically used in integration tests. It provides additional functionality on top of the RestTemplate - especially useful in integration test context. It does not follow redirects so that we can assert response location.
- `template.getForEntity(createURL("/welcome"), String.class)`: Execute a get request for the given URI.
- `assertThat(response.getBody(), equalTo("Hello World"))`: Assert that response body content is "Hello World".

Simple REST method returning an object

In the previous method, we returned a String. Let's now create a method which returns a proper JSON response. Consider the method below:

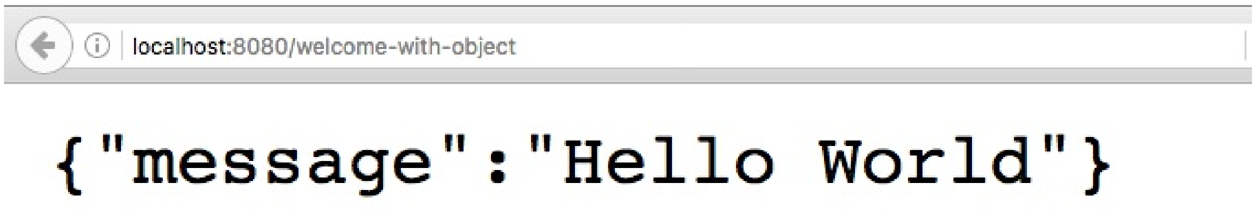
```
@GetMapping("/welcome-with-object")
```

```
    public WelcomeBean welcomeWithObject() {
      return new WelcomeBean("Hello World");
    }
```

Above method is returning a simple WelcomBean initialised with a message `"Hello World"`

Execute a Request

Let's send a test request and see what response we would get. The following screenshot shows the output:



```
{"message":"Hello World"}
```

Response for url `http://localhost:8080/welcome-with-object` is shown below:

```
    {"message":"Hello World"}
```

Question to answer is:

How does the WelcomeBean object that we returned get converted to JSON?

Again its the magic of Spring Boot auto configuration. If Jackson is on the class path of an application, an instance of the default object to JSON (and vice versa) converters are auto configured by Spring Boot.

Unit Testing

Let's quickly write a unit test checking for the JSON Response. Let's add the test to `BasicControllerTest`:

```
    @Test
    public void welcomeWithObject() throws Exception {
      mvc.perform(
        MockMvcRequestBuilders.get("/welcome-with-object")
          .accept(MediaType.APPLICATION_JSON))
        .andExpect(status().isOk())
        .andExpect(content().string(containsString("Hello World")
    }
```

This test is very similar to the earlier unit test except that we are using `containsString` to check if the content contains a sub string `"Hello World"`. We will learn how to write proper JSON tests a little later.

Integration Testing

Let's now shift our focus to writing an integration test. Let's add a method to `BasicControllerIT` as shown in the following code snippet:

```
    @Test
    public void welcomeWithObject() throws Exception {
      ResponseEntity<String> response =
        template.getForEntity(createURL("/welcome-with-object"),
        String.class);
      assertThat(response.getBody(),
      containsString("Hello World"));
    }
```

This method is similar to the earlier integration test except that we are asserting for a sub string using containsString method.

Get Method with Path Variable

Let’s now shift our attention to Path Variables. Path variables are used to bind values from the URI to a variable on the controller method. In the example below, we want to parameterise the name so that we can customize the welcome message with a name:

```
    private static final String helloWorldTemplate = "Hello World
      %s!";

    @GetMapping("/welcome-with-parameter/name/{name}")
    public WelcomeBean welcomeWithParameter(@PathVariable String
    {
```

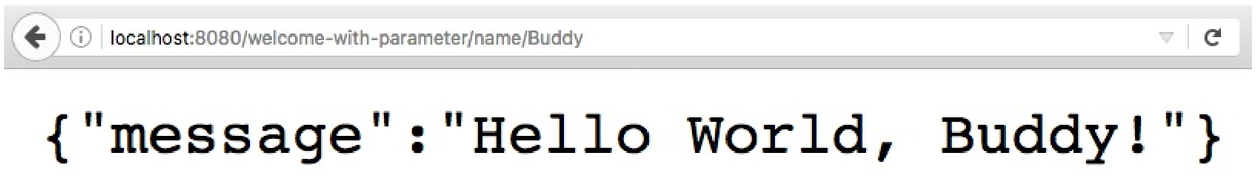
```
    return new WelcomeBean(String.format(helloWorldTemplate, na
  }
```

A few important things to note:

- `@GetMapping("/welcome-with-parameter/name/{name}”)`: `{name}` indicates that this value will be variable. We can have multiple variable templates in an URI.
- `welcomeWithParameter(@PathVariable String name)`: `@PathVariable` ensures that the variable value from the URI is bound to the variable name.
- `String.format(helloWorldTemplate, name)`: Simple String format to replace the `%s` in template with name

Execute a Request

Let's send a test request and see what response we would get. The following screenshot shows the response:


localhost:8080/welcome-with-parameter/name/Buddy

```
{"message":"Hello World, Buddy!"}
```

Response for url `http://localhost:8080/welcome-with-parameter/name/Buddy` is as shown :

```
{"message":"Hello World, Buddy!"}
```

As expected, the name in the URI is used to form the message in the response.

Unit Testing

Let's quickly write a unit test for the above method. We would want to pass a name as part of the URI and check if the response contains the name. Th following code shows how we can do it:

```
@Test
public void welcomeWithParameter() throws Exception {
  mvc.perform(
  MockMvcRequestBuilders.get("/welcome-with-parameter/name/Bu
               .accept(MediaType.APPLICATION_JSON))
  .andExpect(status().isOk())
  .andExpect(
    content().string(containsString("Hello World, Buddy")));
}
```

A few important things to note:

- `MockMvcRequestBuilders.get("/welcome-with-parameter/name/Buddy")`: This matches against the variable template in the URI. We are passing in the name as Buddy.
- `.andExpect(content().string(containsString("Hello World, Buddy”)))`: We expect the response to contain the message with the name.

Integration Testing

Integration test for above method is very simple. Consider the test method below:

```
@Test
public void welcomeWithParameter() throws Exception {
  ResponseEntity<String> response =
    template.getForEntity(
  createURL("/welcome-with-parameter/name/Buddy"), String.cla
  assertThat(response.getBody(),
  containsString("Hello World, Buddy"));
}
```

A few important things to note:

- `createURL("/welcome-with-parameter/name/Buddy")`: This matches against the variable template in the URI. We are passing in the name as Buddy.

- `assertThat(response.getBody(), containsString("Hello World, Buddy”))`: We expect the response to contain the message with the name.

In this section, we looked at the basics of creating a simple REST service with Spring Boot. We also ensured that we have good unit tests and integration tests. While these are really basic, these lay the foundation for more complex REST services we will build in the next section.

Unit tests and integration tests we implemented can have better asserts using JSON comparison instead of a simple sub string comparison. We will focus on it in the tests we write for REST services we create in next sections.

Creating a Todo Resource

We will focus on creating REST services for a basic todo management system. We will create services for

- Retrieving a list of todo's for a given user
- Retrieving details for specific todo
- Creating a todo for a user

Request Methods, Operations and URIs

One of the best practices with REST services is to use appropriate HTTP Request method based on the action we perform. In the services we exposed until now we used GET method as we focused on services which read data.

Below table shows the appropriate HTTP Request method based on the operation that we perform:

HTTP Request Method	Operation
`GET`	Read - Retrieve details for a resource
`POST`	Create - Create a new item or resource
`PUT`	Update/Replace
`PATCH`	Update/Modify a part of the resource
`DELETE`	Delete

Let's now quickly map the services that we want to create to the appropriate request methods:

- Retrieving a list of todo's for a given user: This is `READ`. We will use a `GET`. We will use a URI `/users/{name}/todos`. One more good practice is to use plurals for static things in the URI - users, todos etc. This results in more readable URIs.
- Retrieving details for specific todo: Again a `GET`. We will user URI `/users/{name}/todos/{id}`. You can see that this is consistent with earlier URI that we decided for list of todo's.
- Creating a todo for a user: For create operation, the suggested HTTP Request Method is `POST`. To create a new todo we will post to a `URI /users/{name}/todos.`

Beans and Services

To be able to retrieve and store details of todo, we need a Todo bean and a service to retrieve and store the details.

Let's create a Todo Bean:

```
public class Todo {
  private int id;
  private String user;

  private String desc;

  private Date targetDate;
  private boolean isDone;

  public Todo() {}

  public Todo(int id, String user, String desc,
    Date targetDate, boolean isDone) {
    super();
    this.id = id;
    this.user = user;
    this.desc = desc;
    this.targetDate = targetDate;
```

```
        this.isDone = isDone;
      }

      //ALL Getters
    }
```

We have a created a simple Todo bean with id, name of user, description of todo, todo target date and an indicator for completion status. We added a constructor and getters for all fields.

Let's now add in the `TodoService`:

```
    @Service
    public class TodoService {
      private static List<Todo> todos = new ArrayList<Todo>();
      private static int todoCount = 3;

      static {
        todos.add(new Todo(1, "Jack", "Learn Spring MVC",
        new Date(), false));
        todos.add(new Todo(2, "Jack", "Learn Struts", new Date(),
          false));
        todos.add(new Todo(3, "Jill", "Learn Hibernate", new Date
          false));
      }

      public List<Todo> retrieveTodos(String user) {
        List<Todo> filteredTodos = new ArrayList<Todo>();
        for (Todo todo : todos) {
          if (todo.getUser().equals(user))
          filteredTodos.add(todo);
        }
        return filteredTodos;
      }

      public Todo addTodo(String name, String desc,
      Date targetDate, boolean isDone) {
        Todo todo = new Todo(++todoCount, name, desc, targetDate,
        isDone);
        todos.add(todo);
        return todo;
      }

      public Todo retrieveTodo(int id) {
        for (Todo todo : todos) {
        if (todo.getId() == id)
          return todo;
        }
        return null;
      }

    }
```

Quick things to note:

- To keep things simple, this service does not talk to the database. It maintains an in-memory array list of todo's. This list is initialised using a static initialiser.
- We are exposing couple of simple retrieve methods and a method to add a todo.

Now that we have the service and bean ready, we can create our first service to retrieve a list of todo's for a user.

Retrieve todo list

We will create a new `RestController` called `TodoController`. Code for the retrieve todos method is shown below:

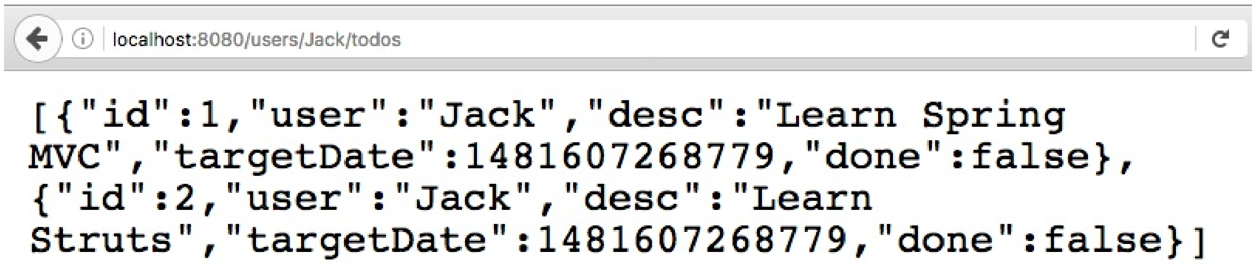
```
@RestController
public class TodoController {
  @Autowired
  private TodoService todoService;

  @GetMapping("/users/{name}/todos")
  public List<Todo> retrieveTodos(@PathVariable String name)
    return todoService.retrieveTodos(name);
  }
}
```

Couple of things to note:

- We are auto wiring the todo service using `@Autowired` annotation
- We use `@GetMapping` annotation to map Get request for URI `"/users/{name}/todos"` to the method `retrieveTodos`

Executing the Service

Let's send a test request and see what response we would get. The following screenshot shows the output:



localhost:8080/users/Jack/todos

```
[{"id":1,"user":"Jack","desc":"Learn Spring
MVC","targetDate":1481607268779,"done":false},
{"id":2,"user":"Jack","desc":"Learn
Struts","targetDate":1481607268779,"done":false}]
```

Response for url `http://localhost:8080/users/Jack/todos` is as shown :

```
[
  {"id":1,"user":"Jack","desc":"Learn Spring
    MVC","targetDate":1481607268779,"done":false},
  {"id":2,"user":"Jack","desc":"Learn
    Struts","targetDate":1481607268779, "done":false}
]
```

Unit Testing

Code to unit test the `TodoController` class is shown in the follwoing screenshot:

```
@RunWith(SpringRunner.class)
@WebMvcTest(TodoController.class)
public class TodoControllerTest {

  @Autowired
  private MockMvc mvc;

  @MockBean
  private TodoService service;

  @Test
  public void retrieveTodos() throws Exception {
    List<Todo> mockList = Arrays.asList(new Todo(1, "Jack",
    "Learn Spring MVC", new Date(), false), new Todo(2, "Jack
    "Learn Struts", new Date(), false));

    when(service.retrieveTodos(anyString())).thenReturn(mockL
    MvcResult result = mvc
      .perform(MockMvcRequestBuilders.get("/users
      /Jack/todos").accept(MediaType.APPLICATION_JSON))
      .andExpect(status().isOk()).andReturn();

   String expected = "["
     + "{id:1,user:Jack,desc:Learn Spring MVC,done:false}" +
     + "{id:2,user:Jack,desc:Learn Struts,done:false}" + "]";

   JSONAssert.assertEquals(expected, result.getResponse()
     .getContentAsString(), false);
  }
}
```

Few important things to note:

- We are writing a unit test. So, we want to test only the logic present in `TodoController` class. So, we initialise a Mock MVC framework with only the `TodoController` class using `@WebMvcTest(TodoController.class)`.
- `@MockBean private TodoService service`: We are mocking out the TodoService using `@MockBean` annotation. In test classes that are run with SpringRunner, the beans defined with `@MockBean` will be replaced by a mock, created by using Mockito framework.
- `when(service.retrieveTodos(anyString())).thenReturn(mockList)` mocking the service method retrieveTodos to return the mock list.
- `MvcResult result = ..` : We are accepting the result of the request into a MvcResult variable to enable us to perform assertions on the response.
- `JSONAssert.assertEquals(expected, result.getResponse().getContentAsString(), false)`: JSONAssert is a very useful framework to perform asserts on JSON. It compares the response text with the expected value. JSONAssert is intelligent enough to ignore values that are not specified. Another advantage is a clear failure message in case of assertion failures. The last parameter false indicates using non strict mode. If it is change to true, then the expected should exactly match the result.

Integration Testing

Code to do integration testing on the `TodoController` class is shown in the following code snippet. It launches up the entire spring context with all controllers and beans defined:

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringBootTest(classes = Application.class, webEnvironment =
  SpringBootTest.WebEnvironment.RANDOM_PORT)
public class TodoControllerIT {

  @LocalServerPort
  private int port;

  private TestRestTemplate template = new TestRestTemplate();

  @Test
  public void retrieveTodos() throws Exception {
    String expected = "["
     + "{id:1,user:Jack,desc:Learn Spring MVC,done:false}" +
     + "{id:2,user:Jack,desc:Learn Struts,done:false}" + "]";

    String uri = "/users/Jack/todos";

    ResponseEntity<String> response =
      template.getForEntity(createUrl(uri), String.class);

    JSONAssert.assertEquals(expected, response.getBody(), fal
 }

 private String createUrl(String uri) {
   return "http://localhost:" + port + uri;
 }
}
```

This test is very similar to the integration test for `BasicController` except that we are using JSONAssert to assert the response.

Retrieving details for specific todo

We will now add the method to retrieve details for a specific todo:

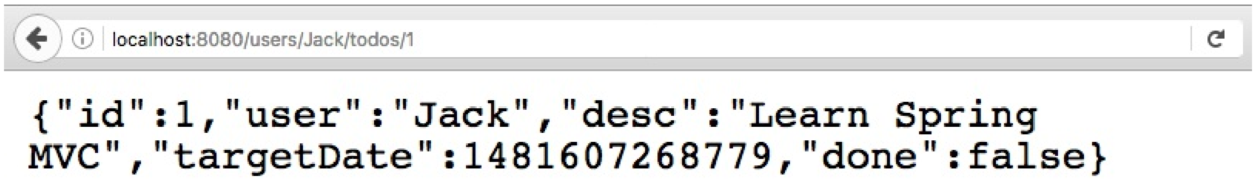
```
@GetMapping(path = "/users/{name}/todos/{id}")
public Todo retrieveTodo(@PathVariable String name, @PathVari
  int id) {
    return todoService.retrieveTodo(id);
    }
```

Couple of things to note:

- URI Mapped is `/users/{name}/todos/{id}`.
- We have two Path Variable's defined for name and id.

Executing the Service

Let's send a test request and see what response we would get is shown in the following screenshot:


localhost:8080/users/Jack/todos/1

```
{"id":1,"user":"Jack","desc":"Learn Spring
MVC","targetDate":1481607268779,"done":false}
```

Response for url `http://localhost:8080/users/Jack/todos/1` is shown below:

```
{"id":1,"user":"Jack","desc":"Learn Spring MVC",
  "targetDate":1481607268779,"done":false}
```

Unit Testing

Code to unit test the retrieveTodo is as shown:

```
@Test
public void retrieveTodo() throws Exception {
  Todo mockTodo = new Todo(1, "Jack", "Learn Spring MVC",
  new Date(), false);

  when(service.retrieveTodo(anyInt())).thenReturn(mockTodo);

  MvcResult result = mvc.perform(
    MockMvcRequestBuilders.get("/users/Jack/todos/1")
    .accept(MediaType.APPLICATION_JSON))
    .andExpect(status().isOk()).andReturn();

  String expected = "{id:1,user:Jack,desc:Learn Spring
```

```
    MVC,done:false}";

  JSONAssert.assertEquals(expected,
    result.getResponse().getContentAsString(), false);

}
```

Few important things to note:

- `when(service.retrieveTodo(anyInt())).thenReturn(mockTodo)`: mocking the service method retrieveTodo to return the mock todo.
- `MvcResult result = ..` : We are accepting the result of the request into a MvcResult variable to enable us to perform assertions on the response.
- `JSONAssert.assertEquals(expected, result.getResponse().getContentAsString(), false)`: Assert if the result is as expected.

Integration Testing

Code to do integration testing on the `retrieveTodos` in `TodoController` is shown in the following code snippet. This would be added to the class `TodoControllerIT`:

```
 @Test
 public void retrieveTodo() throws Exception {
   String expected = "{id:1,user:Jack,desc:Learn Spring
   MVC,done:false}";
   ResponseEntity<String> response = template.getForEntity(
     createUrl("/users/Jack/todos/1"), String.class);
   JSONAssert.assertEquals(expected, response.getBody(), false
}
```

Add a todo

We will now add the method to create a new todo. For create, the HTTP Method to use is Post. We will post to a URI `"/users/{name}/todos"`.

```
 @PostMapping("/users/{name}/todos")
 ResponseEntity<?> add(@PathVariable String name,
 @RequestBody Todo todo)
 {
   Todo createdTodo = todoService.addTodo(name, todo.getDesc()
   todo.getTargetDate(), todo.isDone());
   if (createdTodo == null) {
     return ResponseEntity.noContent().build();
   }

   URI location = ServletUriComponentsBuilder.fromCurrentReque

     .path("/{id}").buildAndExpand(createdTodo.getId()).toUri(
   return ResponseEntity.created(location).build();
 }
```

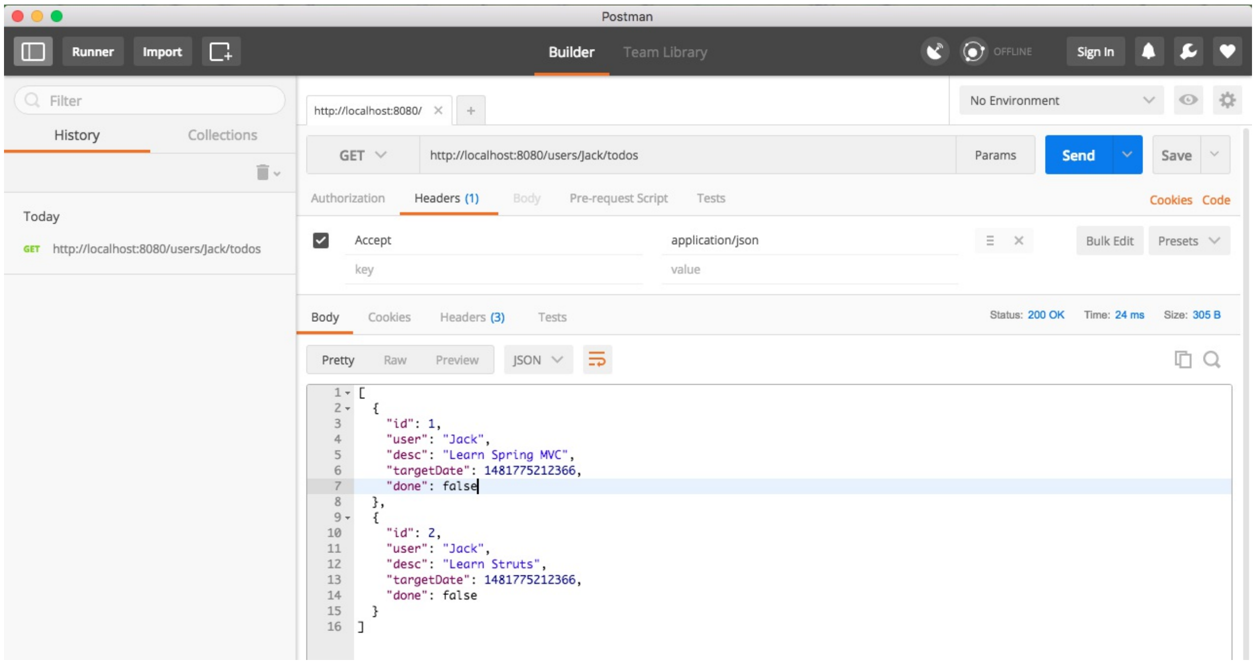
A few things to note:

- `@PostMapping("/users/{name}/todos”)`: `@PostMapping` annotations maps the method to a HTTP Request with POST method
- `ResponseEntity<?> add(@PathVariable String name, @RequestBody Todo todo)`: A HTTP post request should ideally return the URI to the created resources. We use ResourceEntity to do this. `@RequestBody` binds the body of the request directly to the bean.
- `ResponseEntity.noContent().build()`: Used to return that the creation of resource failed.
- `ServletUriComponentsBuilder.fromCurrentRequest().path("/{id}"` Form the URI for the created resource which can be returned in the response.
- `ResponseEntity.created(location).build()`: Returns a status of `201(CREATED)` with a link to the resource created.

PostMan

We will use Postman App to interact with the REST Services. You can install it from their website https://www.getpostman.com/. It is available on Windows and Mac. A Google Chrome plugin is also available.

Let's send a test request and see what response we would get. The following screenshot shows Below picture shows the response:



GET http://localhost:8080/users/Jack/todos

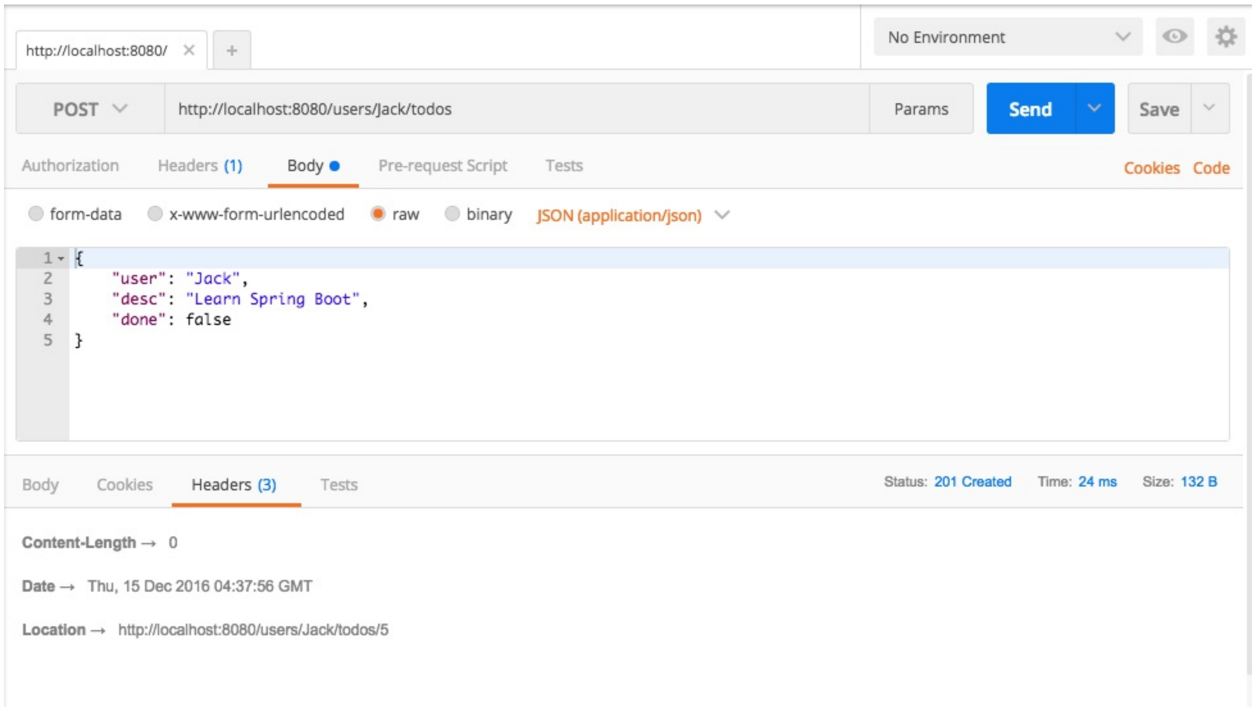
Accept	application/json
key	value

Status: 200 OK Time: 24 ms Size: 305 B

```
[
  {
    "id": 1,
    "user": "Jack",
    "desc": "Learn Spring MVC",
    "targetDate": 1481775212366,
    "done": false
  },
  {
    "id": 2,
    "user": "Jack",
    "desc": "Learn Struts",
    "targetDate": 1481775212366,
    "done": false
  }
]
```

Executing the Post Service

To create a new todo using `POST`, we would need include the JSON for the todo in the body of the request. The screenshot below shows how we can use postman to create the request and the response after executing the request:



POST http://localhost:8080/users/Jack/todos

```
{
    "user": "Jack",
    "desc": "Learn Spring Boot",
    "done": false
}
```

Status: 201 Created Time: 24 ms Size: 132 B

```
Content-Length → 0
Date → Thu, 15 Dec 2016 04:37:56 GMT
Location → http://localhost:8080/users/Jack/todos/5
```

Few important things to note:

- We are sending a post request. So, we choose the POST from the left top dropdown.
- For sending the todo JSON as part of body of request, we select the raw option in Body tab (highlighted with a blue dot). We choose content type as JSON (application/json).
- Once the request is successfully executed, you can see the status of the request in the bar in the middle of the screen. Status: 201 Created.
- Location `http://localhost:8080/users/Jack/todos/5`: This is the URI of the newly created todo that is received in the response.

Full details of the request to `http://localhost:8080/users/Jack/todos` are shown in the block below:

```
Header
  Content-Type:application/json

Body
{
  "user": "Jack",
  "desc": "Learn Spring Boot",
  "done": false
}
```

Unit Testing

Code to unit test the create todo is shown below:

```
@Test
public void createTodo() throws Exception {
  Todo mockTodo = new Todo(CREATED_TODO_ID, "Jack",
  "Learn Spring MVC", new Date(), false);
  String todo = "{"user":"Jack","desc":"Learn Spring MVC",
  "done":false}";

  when(service.addTodo(anyString(), anyString(),
  any(Date.class),anyBoolean()))
  .thenReturn(mockTodo);

  mvc
  .perform(MockMvcRequestBuilders.post("/users/Jack/todos")
        .content(todo)
        .contentType(MediaType.APPLICATION_JSON)
        )
      .andExpect(status().isCreated())
      .andExpect(
          header().string("location",containsString("/users/Jack/t
                                                     + CREATED_TODO
 }
```

Few important things to note:

- `String todo = "{"user":"Jack","desc":"Learn Spring MVC","done":false}"`: Todo content to post to the create todo service.
- `when(service.addTodo(anyString(), anyString(), any(Date.class),anyBoolean())).thenReturn(mockTodo)`: Mocking the service to return a dummy todo.
- `MockMvcRequestBuilders.post("/users/Jack/todos").content(todo` Create a POST to given URI with given content type.
- `andExpect(status().isCreated())`: Expect that the status is created.
- `andExpect(header().string("location",containsString("/users/J + CREATED_TODO_ID)))`: Expect that the header contains "location" with the uri of created resource.

Integration Testing

Code to do integration testing on the create todo in `TodoController` is shown in the following screenshot. This would be added to the class `TodoControllerIT`:

```
    @Test
    public void addTodo() throws Exception {
      Todo todo = new Todo(-1, "Jill", "Learn Hibernate", new Dat
      false);
      URI location = template
        .postForLocation(createUrl("/users/Jill/todos"),todo);
      assertThat(location.getPath(),
      containsString("/users/Jill/todos/4"));
    }
```

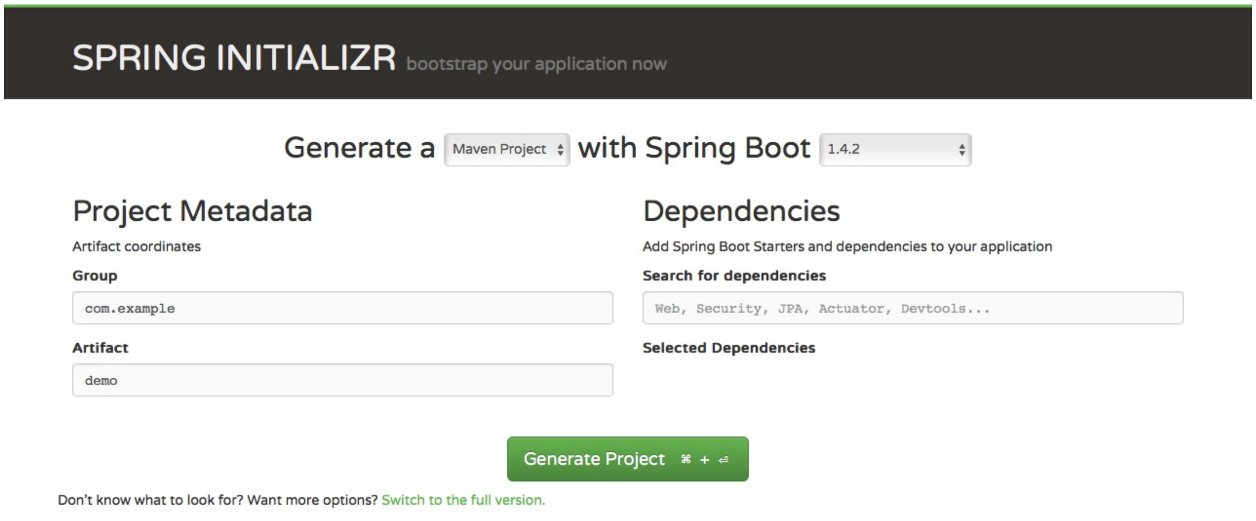
Few important things to note:

- `URI location = template.postForLocation(createUrl("/users/Jill/todos"), todo)`: `postForLocation` is a utility method especially useful in tests for creating new resources. We are posting the todo to the given URI and getting the location from header.
- `assertThat(location.getPath(), containsString("/users/Jill/todos/4”))`: Asserting that the location contains the path to the newly created resource.

Spring Initializr

Do you want to auto-generate Spring Boot projects? Do you want to quickly get started with developing your application? Spring Initializr is the answer.

Spring Initializr is hosted at http://start.spring.io. The screen shot below shows how the website looks.

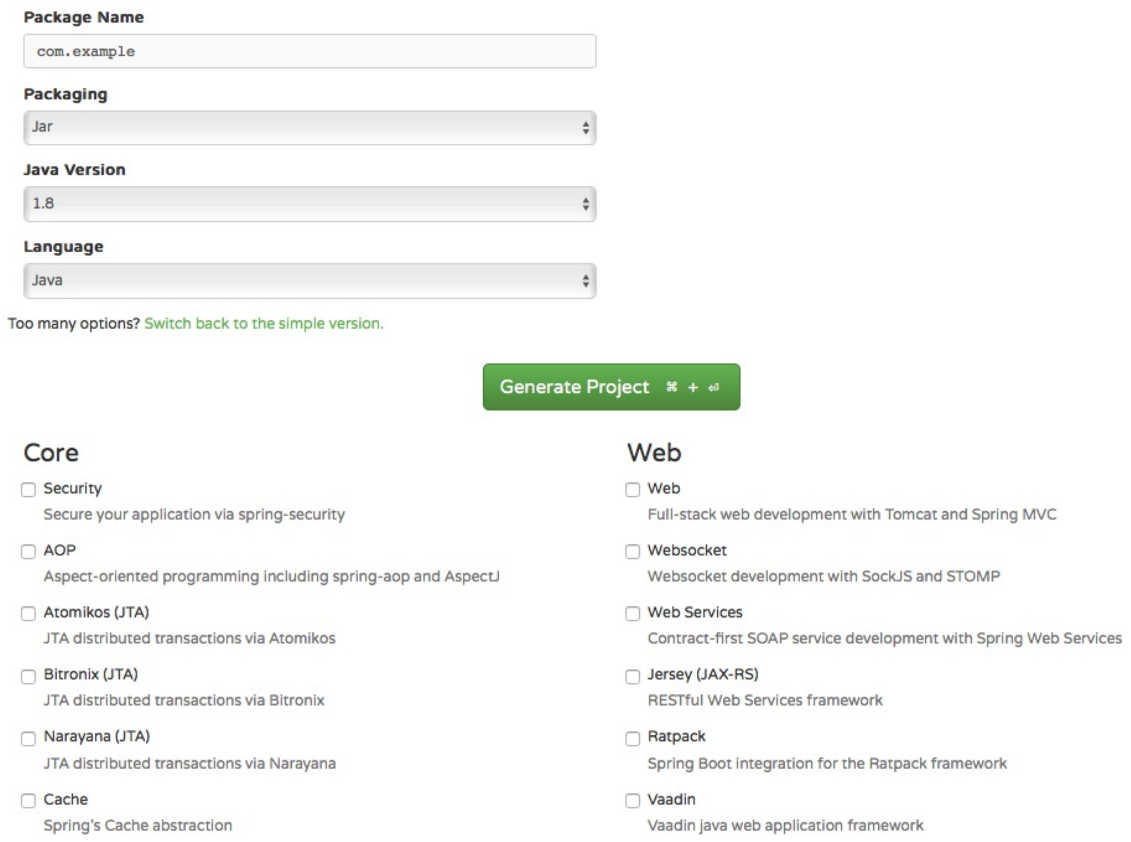

SPRING INITIALIZR bootstrap your application now
Generate a Maven Project with Spring Boot 1.4.2
Project Metadata
Artifact coordinates
Dependencies
Add Spring Boot Starters and dependencies to your application
Search for dependencies
Web, Security, JPA, Actuator, Devtools...
Selected Dependencies
Generate Project ⌘ + ⏎
Don't know what to look for? Want more options? Switch to the full version.

Group: com.example
Artifact: demo

Spring Initializr provides a lot of flexibility in creating projects. You have options to:

- Choose your build tool: Maven or Gradle
- Choose Spring Boot version you want to use
- Configure a Group id and Artifact id for your component
- Choose the starters (Dependencies) that you would want for your project. You can click the link at the bottom of the screen - Switch to the full version. - to see all the starter projects you can choose from.
- Choose how to package your component - Jar or War
- Choose Java version you want to use
- Choose the JVM language you want to use

The following screenshot shows some of the options Spring Initializr provides when you expand (click the link) to the full version:



Package Name

com.example

Packaging

Jar

Java Version

1.8

Language

Java

Too many options? Switch back to the simple version.

Core

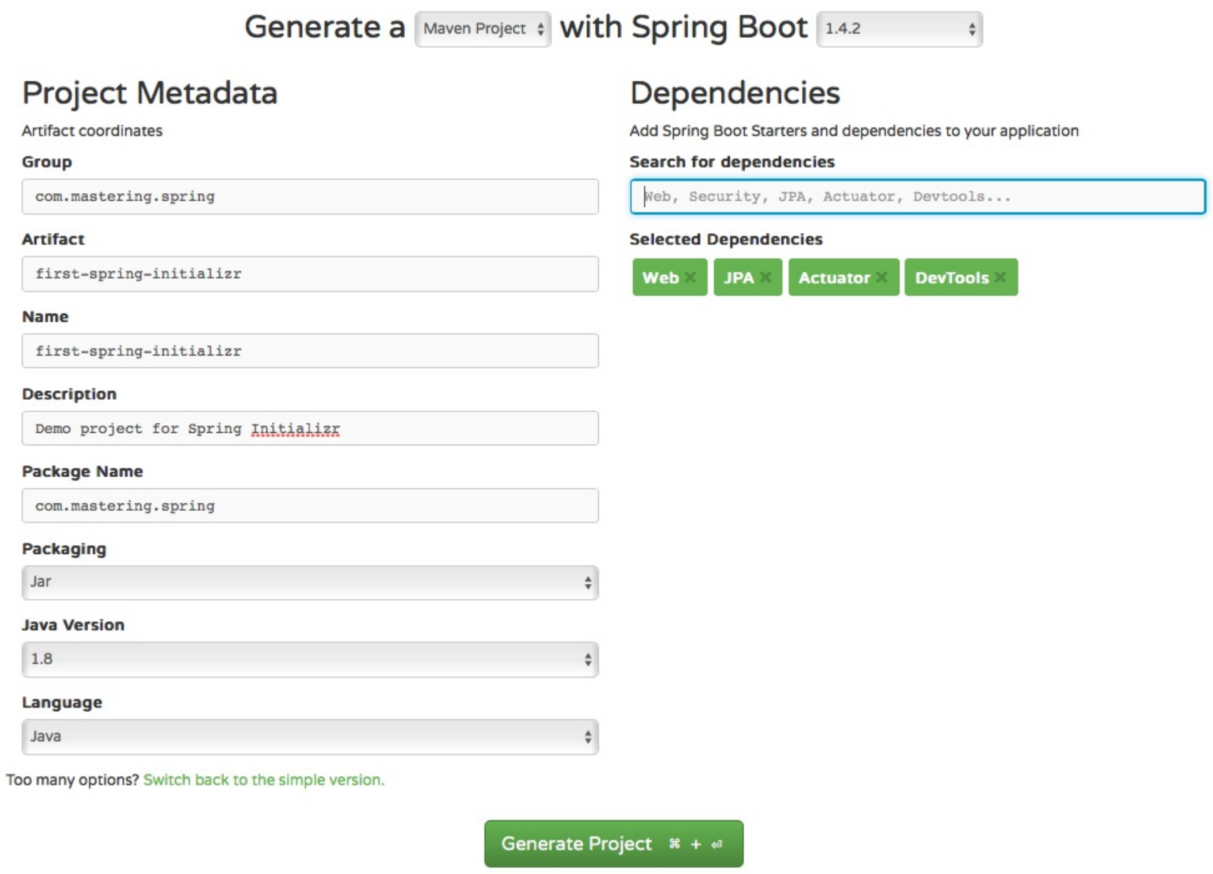
- [ ] Security
  Secure your application via spring-security
- [ ] AOP
  Aspect-oriented programming including spring-aop and AspectJ
- [ ] Atomikos (JTA)
  JTA distributed transactions via Atomikos
- [ ] Bitronix (JTA)
  JTA distributed transactions via Bitronix
- [ ] Narayana (JTA)
  JTA distributed transactions via Narayana
- [ ] Cache
  Spring's Cache abstraction

Web

- [ ] Web
  Full-stack web development with Tomcat and Spring MVC
- [ ] Websocket
  Websocket development with SockJS and STOMP
- [ ] Web Services
  Contract-first SOAP service development with Spring Web Services
- [ ] Jersey (JAX-RS)
  RESTful Web Services framework
- [ ] Ratpack
  Spring Boot integration for the Ratpack framework
- [ ] Vaadin
  Vaadin java web application framework

Creating your first Spring Initializr project

We will use the full version and enter the values as shown in the screenshot below:


Generate a Maven Project with Spring Boot 1.4.2

Project Metadata

Artifact coordinates

Group: com.mastering.spring

Artifact: first-spring-initializr

Name: first-spring-initializr

Description: Demo project for Spring Initializr

Package Name: com.mastering.spring

Packaging: Jar

Java Version: 1.8

Language: Java

Too many options? Switch back to the simple version.

Dependencies

Add Spring Boot Starters and dependencies to your application

Search for dependencies: Web, Security, JPA, Actuator, Devtools...

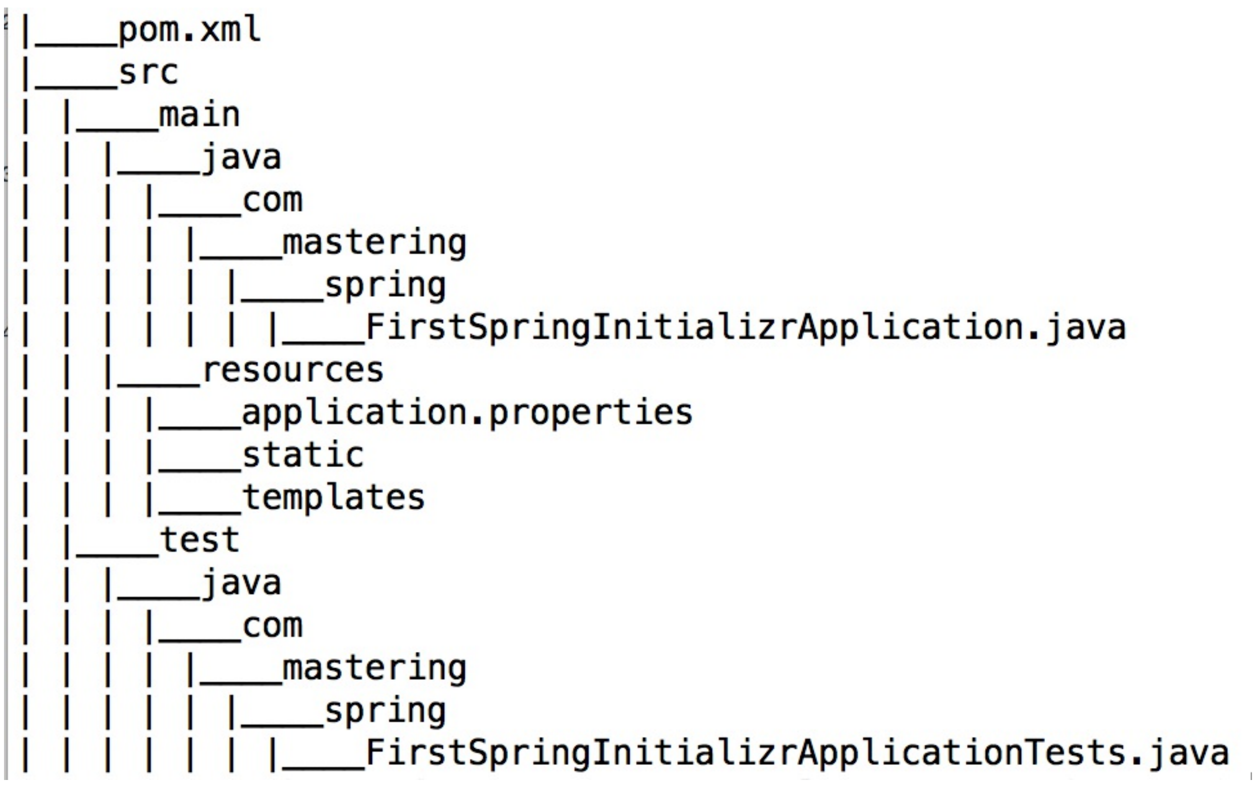
Selected Dependencies: Web, JPA, Actuator, DevTools

Things to note:

- Build tool: Maven
- Spring Boot version: Choose the latest available
- Group: com.mastering.spring
- Artifact: first-spring-initializr
- Selected Dependencies: Choose Web, JPA, Actuator and Dev Tools. Type in each one of these in the text box and press enter to choose them. We will learn more about Actuator and Dev Tools in the next section
- We choose Java version 1.8

Go ahead and click the `Generate Project` button. This will create a zip file and you can download this to your computer.

The screenshot below shows the structure of the project created:

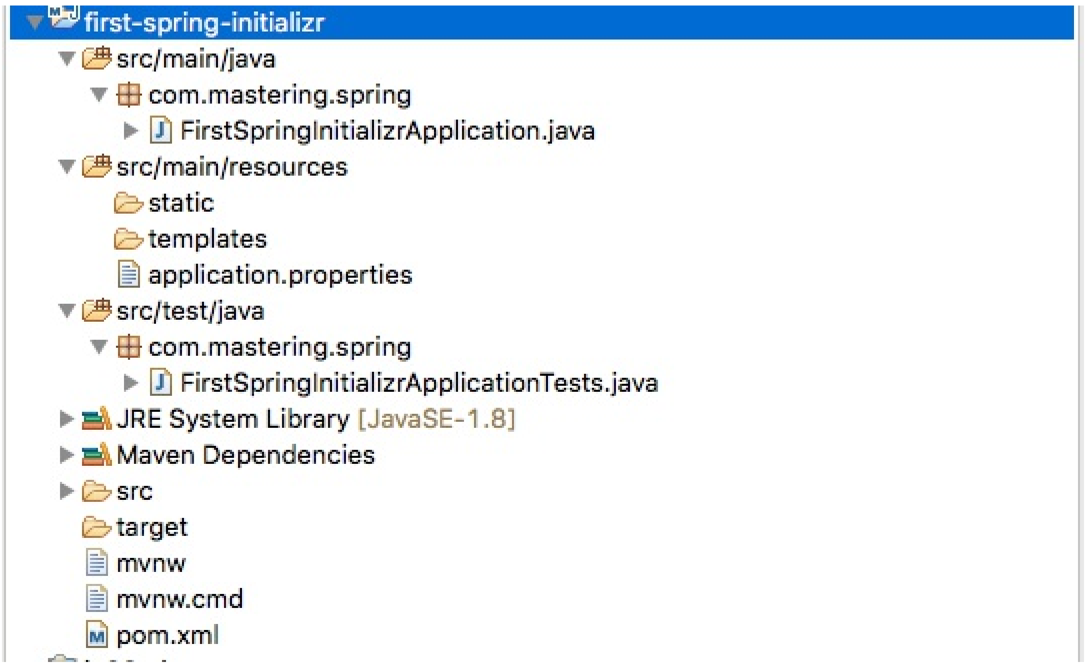


```
|____pom.xml
|____src
| |____main
| | |____java
| | | |____com
| | | | |____mastering
| | | | | |____spring
| | | | | | |____FirstSpringInitializrApplication.java
| | |____resources
| | | |____application.properties
| | | |____static
| | | |____templates
| |____test
| | |____java
| | | |____com
| | | | |____mastering
| | | | | |____spring
| | | | | | |____FirstSpringInitializrApplicationTests.java
```

We will now import this project into Your IDE. In eclipse, you can use following steps:

1. Launch Eclipse.
2. Select **`File > Import`**.
3. Choose Existing Maven Projects.
4. Browse and select the folder that is the root of the Maven project (one containing the file `pom.xml`).
5. Proceed with defaults and click **`Finish`**.

This would import the project into Eclipse. The following screenshot shows the structure of project in Eclipse:



Let's now look at the some of the important files from the generated project.

pom.xml

Below snippet shows the dependencies that are declared:

```
 <dependencies> <dependency> <groupId>org.springframework.boot</g
```

A few other important observations:

- Packaging for this component is jar
- `org.springframework.boot:spring-boot-starter-parent` is declared as the parent pom
- `<java.version>1.8</java.version>`: Java version is 1.8
- Spring Boot Maven Plugin (`org.springframework.boot:spring-boot-maven-plugin`) is configured as a plugin

FirstSpringInitializrApplication

`FirstSpringInitializrApplication.java` is the launcher for Spring Boot:

```
    package com.mastering.spring;
```

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure
  .SpringBootApplication;

@SpringBootApplication
public class FirstSpringInitializrApplication {
  public static void main(String[] args) {
    SpringApplication.run(FirstSpringInitializrApplication.cl
    args);
  }
}
```

FirstSpringInitializrApplicationTests

`FirstSpringInitializrApplicationTests` contains the basic context that can be used to start writing the tests as we start developing the application:

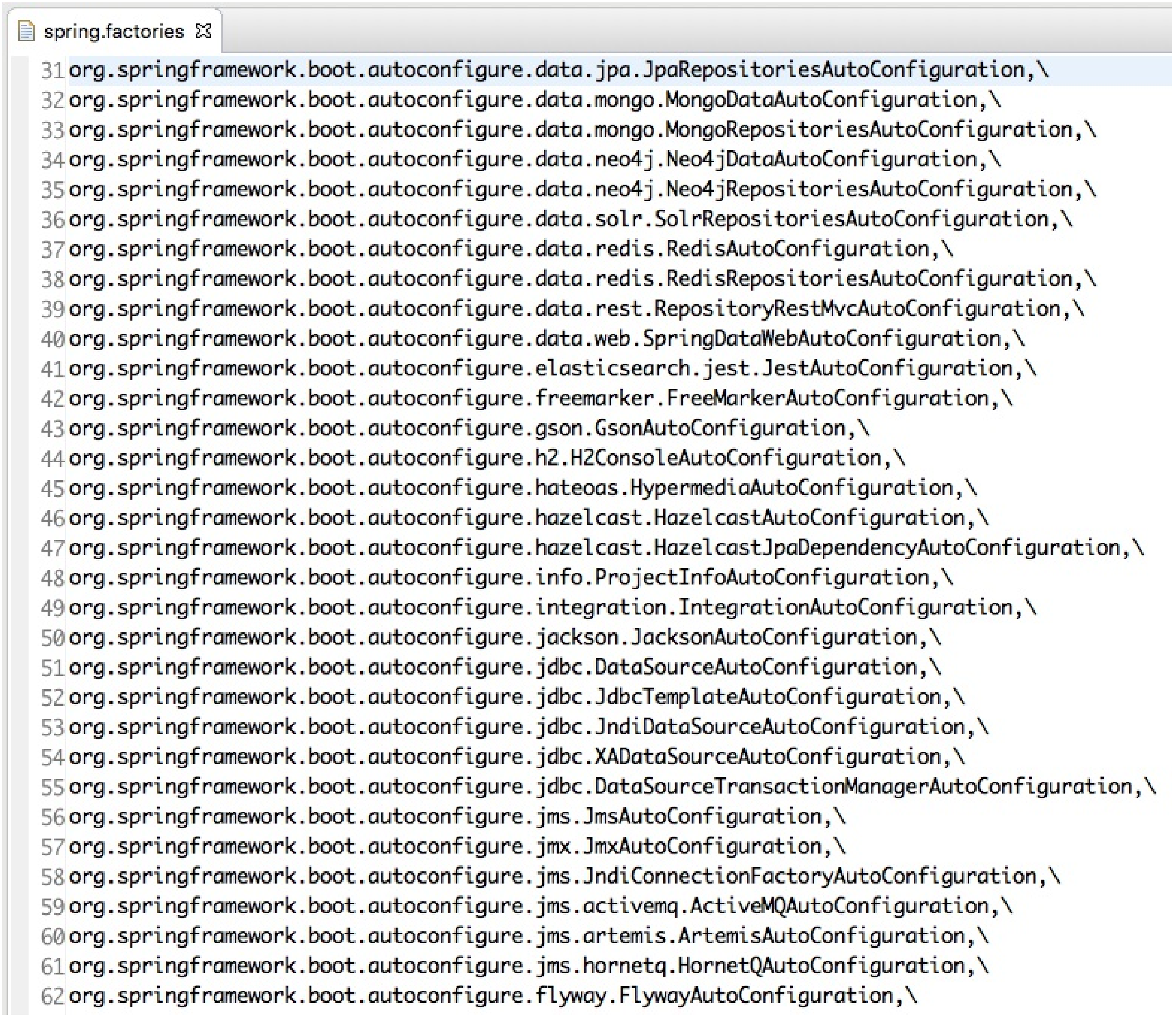
```
package com.mastering.spring;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@SpringBootTest
public class FirstSpringInitializrApplicationTests {

  @Test
  public void contextLoads() {
  }
}
```

A Quick Peek behind Auto Configuration

Auto configuration is one of the most important features of Spring Boot. In this section, we will take a quick peek behind the scenes to understand how Spring Boot Auto Configuration works.

Most of the Spring Boot Auto Configuration magic comes from `spring-boot-autoconfigure-{version}.jar`. When we start any Spring Boot applications, a number of beans get auto configured. How does this happen?

The following screenshot shows an extract from `spring.factories` from `spring-boot-autoconfigure-{version}.jar`. We have filtered out some of the configuration in interest of space.


spring.factories

```
org.springframework.boot.autoconfigure.data.jpa.JpaRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.mongo.MongoDataAutoConfiguration,\
org.springframework.boot.autoconfigure.data.mongo.MongoRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.neo4j.Neo4jDataAutoConfiguration,\
org.springframework.boot.autoconfigure.data.neo4j.Neo4jRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.solr.SolrRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.redis.RedisAutoConfiguration,\
org.springframework.boot.autoconfigure.data.redis.RedisRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.rest.RepositoryRestMvcAutoConfiguration,\
org.springframework.boot.autoconfigure.data.web.SpringDataWebAutoConfiguration,\
org.springframework.boot.autoconfigure.elasticsearch.jest.JestAutoConfiguration,\
org.springframework.boot.autoconfigure.freemarker.FreeMarkerAutoConfiguration,\
org.springframework.boot.autoconfigure.gson.GsonAutoConfiguration,\
org.springframework.boot.autoconfigure.h2.H2ConsoleAutoConfiguration,\
org.springframework.boot.autoconfigure.hateoas.HypermediaAutoConfiguration,\
org.springframework.boot.autoconfigure.hazelcast.HazelcastAutoConfiguration,\
org.springframework.boot.autoconfigure.hazelcast.HazelcastJpaDependencyAutoConfiguration,\
org.springframework.boot.autoconfigure.info.ProjectInfoAutoConfiguration,\
org.springframework.boot.autoconfigure.integration.IntegrationAutoConfiguration,\
org.springframework.boot.autoconfigure.jackson.JacksonAutoConfiguration,\
org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration,\
org.springframework.boot.autoconfigure.jdbc.JdbcTemplateAutoConfiguration,\
org.springframework.boot.autoconfigure.jdbc.JndiDataSourceAutoConfiguration,\
org.springframework.boot.autoconfigure.jdbc.XADataSourceAutoConfiguration,\
org.springframework.boot.autoconfigure.jdbc.DataSourceTransactionManagerAutoConfiguration,\
org.springframework.boot.autoconfigure.jms.JmsAutoConfiguration,\
org.springframework.boot.autoconfigure.jmx.JmxAutoConfiguration,\
org.springframework.boot.autoconfigure.jms.JndiConnectionFactoryAutoConfiguration,\
org.springframework.boot.autoconfigure.jms.activemq.ActiveMQAutoConfiguration,\
org.springframework.boot.autoconfigure.jms.artemis.ArtemisAutoConfiguration,\
org.springframework.boot.autoconfigure.jms.hornetq.HornetQAutoConfiguration,\
org.springframework.boot.autoconfigure.flyway.FlywayAutoConfiguration,\
```

Above list of AutoConfiguration classes are run whenever a Spring Boot Application is launched. Let's take a quick look at one of them:

```
org.springframework.boot.autoconfigure.web.WebMvcAutoConfiguratio
```

Here's a small snippet:

```
    @Configuration
    @ConditionalOnWebApplication
    @ConditionalOnClass({ Servlet.class, DispatcherServlet.class,
    WebMvcConfigurerAdapter.class })
    @ConditionalOnMissingBean(WebMvcConfigurationSupport.class)
    @AutoConfigureOrder(Ordered.HIGHEST_PRECEDENCE + 10)
    @AutoConfigureAfter(DispatcherServletAutoConfiguration.class)
    public class WebMvcAutoConfiguration {
```

Some of the important points to note:

- `@ConditionalOnClass({ Servlet.class, DispatcherServlet.class, WebMvcConfigurerAdapter.class })`: This AutoConfiguration is enabled if any of the mentioned classes are in the classpath. When we add web starter project, we bring in dependencies with all the above classes. Hence, this AutoConfiguration will be enabled.
- `@ConditionalOnMissingBean(WebMvcConfigurationSupport.class)`: This AutoConfiguration is enabled only if the application does not explicitly declare a bean of class WebMvcConfigurationSupport.class.
- `@AutoConfigureOrder(Ordered.HIGHEST_PRECEDENCE + 10)`: Specifies the precedence of this specific auto configuration.

Let's look at another small snippet showing one of the methods from the same class:

```
@Bean
@ConditionalOnBean(ViewResolver.class)
@ConditionalOnMissingBean(name = "viewResolver",
  value = ContentNegotiatingViewResolver.class)
public ContentNegotiatingViewResolver
viewResolver(BeanFactory beanFactory) {
  ContentNegotiatingViewResolver resolver = new
    ContentNegotiatingViewResolver();
  resolver.setContentNegotiationManager
    (beanFactory.getBean(ContentNegotiationManager.class));
  resolver.setOrder(Ordered.HIGHEST_PRECEDENCE);
  return resolver;
}
```

View Resolvers are one of the beans configured by `WebMvcAutoConfiguration class`. Above snippet ensures that if a view resolver is not provided by the application, then Spring Boot auto configures a default view resolver. Here are a few important points to note:

- `@ConditionalOnBean(ViewResolver.class)`: Create this bean if the ViewResolver.class is on class path.
- `@ConditionalOnMissingBean(name = "viewResolver", value = ContentNegotiatingViewResolver.class)`: Create this bean if there are no explicitly declared beans of name `viewResolver` and of type

`ContentNegotiatingViewResolver.class`

- Rest of the method configured the view resolver.

To summarise, all the auto configuration logic is executed at start of a Spring Boot Application. If a specific class (from specific dependency or starter project) is available on the class path, then the auto configuration classes are executed. These auto configuration classes look at what beans are already configured. Based on the existing beans, they enable creation of default beans.

Summary

Spring Boot makes developing Spring based applications easy. It enables us to create production ready applications from day one of a project.

In this chapter, we understood the basics of Spring Boot and REST Services. We discussed different features of Spring Boot and created a few REST Services with great tests. We understood what happens in the background with an in-depth look at auto configuration.

In the next chapter, we will shift our attention towards adding more features to the REST services.