

SPRING BOOT and MICROSERVICES

Ameya J. Joshi

Email : ameya.joshi_official@outlook.com

Cell No : +91 9850676160

1

About You

- * Name
- * Current Role and Technology
- * Java/JEE/Spring prior Experience
- * Hobbies.

2

AGENDA

- * Spring framework Concept and Limitations
- * Introduction To Spring Boot
- * Spring Boot features
- * Setup Spring Boot
- * Writing First App

3

AGENDA

- * Various ways to create Spring Boot App
- * Spring Boot Actuators
- * Spring Boot Spring Security
- * Microservices
- * Microservices Using Spring Boot

4

AGENDA

- * Microservices Design Patterns
- * Microservices Dos' and Don'ts'
- * Spring Cloud
- * Spring Cloud Config Server
- * Netflix Eureka with Spring Boot
- * Netflix ZUUL with Spring Boot

5

AGENDA

- * Netflix Hystrix with Spring Boot
- * ZipKin with Spring Boot
- * Netflix Ribbon with Spring Boot
- * Spring Boot Reactive Programming with webflux
- * Spring Boot –Testing with Mockito and RESTAssured

6

SPRING Framework

7

What is spring?

- * Spring framework is an open source Java platform and it was initially written by Rod Johnson and was first released under the Apache 2.0 license in June 2003.
- * Spring is lightweight when it comes to size and transparency.
- * Spring framework helps to simplify the development of Java based Enterprise Applications.

8

Spring features

- * Lightweight
- * Inversion of control (IOC)
- * Aspect oriented (AOP)
- * Container
- * MVC Framework
- * uses Plain Old Java Objects(POJO)

9

Benefits of Using Spring Framework

- * Spring enables developers to develop enterprise-class applications using POJOs.
- * Spring's web framework is a well-designed web MVC framework
- * Lightweight IoC containers

10

Beans

- * The objects that form the backbone of application and that are managed by the Spring IoC container are called beans.
- * A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container.

11

Different Forms of Dependency Injection

- Setter Injections
- Constructor Injections
- Method Injections

12

Spring Framework modules

- * Spring IOC container(core container)

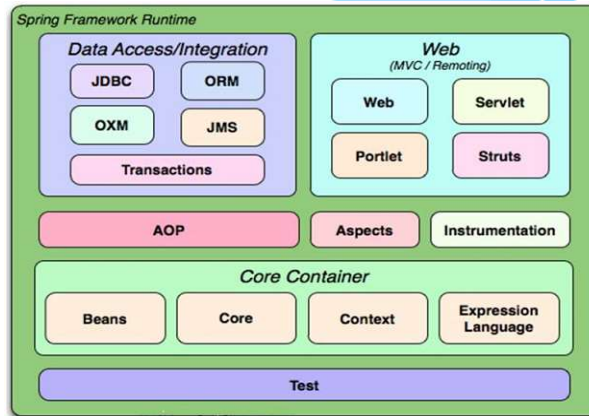
- * Spring AOP

- * Spring DAO

- * Spring ORM

- * JEE Module

- * Spring web MVC



13

13

Spring Core Container

- * The Spring core container provides an implementation for IOC (Inversion of control) supporting dependency injection.
- * IOC is an architectural pattern that describes to have an external entity to perform Dependency injection (DI) at creation time, that is, object creation time.
- * Dependency Injection is a process of injecting/pushing the dependencies into an object.

14

14

Spring AOP

- * Spring AOP module provides an implementation of AOP (Aspect Oriented Programming).
- * Spring is a proxy-based framework implemented in Java.
- * AOP integrates the concerns dynamically into the system.
- * Concerns: A part of the system divided based on the functionality (transaction, security, logging, caching).

15

15

DAO

- * Data Access Object(DAO) is a design pattern the describes to separate the persistence logic from the business logic.
- * The Spring DAO includes a support for the common infrastructures required in creating the DAO.
- * Spring includes DAO support classes to take this responsibility such as:
 - * Jdbc DAO support
 - * Hibernate DAO support
 - * JPA DAO support

16

16

ORM

- * The Object/Relation Mapping(ORM) module of Spring framework provides a high level abstraction for well accepted object-relational mapping API's such as Hibernate, JPA, OJB and iBatis.
- * The Spring ORM module is not a replacement or a competition for any of the existing ORM's, instead it is designed to reduce the complexity by avoiding the boilerplate code from the application in using ORMs.

17

17

JEE

- * The JEE module of Spring framework is build on the solid base provided by the core package.
- * This provides a support for using the remoting services in a simplified manner.
- * This supports to build POJOs and expose them as remote objects without worrying about the specific remoting technology given rules.

18

18

Web

- * This part of Spring framework implements the infrastructure that is required for creating web based MVC application in java.
- * The Spring Web MVC infrastructure is built on top of the Servlet API, so that it can be integrated into any Java Web Application server.
- * This uses the Spring IOC container to access the various framework and application objects.

19

19

Spring IOC



20

20

What is IOC ?

- * IOC is also known as dependency injection (DI).
- * Dependency Injection is the act of injecting dependencies into an Object.
- * Inversion of Control is the general style of using Dependency injection to wire together application layers.
- * Hence Spring is an Inversion of Control container. That is, it is a container that handles Dependency Injection for you.

21

21

Configuration Metadata

- * XML-based configuration metadata.
- * Annotation-based configuration
- * Java-based configuration

22

22

XML-based configuration

```
<beans>
  <bean id="msgBean" class="com.ameya.Message">
    <property name="message"
      value=" Hello World ! " />
  </bean>
</beans>
```

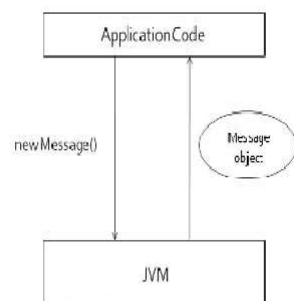
* The string " Hello World " is injected to the bean msgBean

23

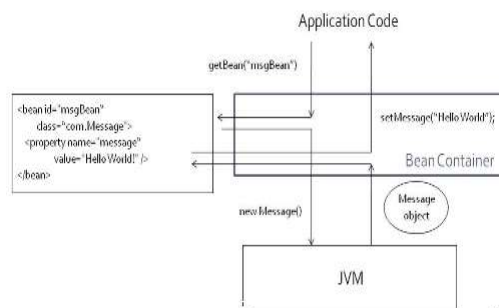
23

Understanding Bean Container

Without a bean container



With a bean container



24

24

Bootstrapping the IOC container

- * To start an app using IOC :
 - * Create an ApplicationContext object and tell it where applicationContext.xml is.
 - * `ApplicationContext appContext = new ClassPathXmlApplicationContext("applicationContext.xml");`
 - * This just has to be done once on startup, and can be done in the main method or whatever code bootstraps the application.

25


25

Dependency Injection Types

- * Setter Injection
- * Constructor Injection

26

26




Create the Account.java:

```
package com.ameya.models;
public class Account {
    private String firstName, lastName;
    private double balance;
    private Address address;
    public Account(){
        address = new Address();
    }
    public Account(String firstName, String lastName, double balance) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.balance = balance;
        this.address = new Address();
    }
    public Account(String firstName, String lastName, double balance, Address address)
    {this.firstName=firstName;
    this.lastName=lastName;
    this.balance = balance;
    this.address = address;
    }
    //Setter & Getter
}
```

27

27



Create the Address.java:

```
package com.ameya.models;
public class Address {
    private String building, street, city, state, country;
    public Address() {}
    public Address(String building, String street, String city, String state,
        String country) {this.building = building;
        this.street = street;
        this.city = city;
        this.state = state;
        this.country = country;
    }
    //Setter & Getter
}
```

28

28

Constructor-based dependency Injection

```
<beans....>
  <bean id="account"
    class="com.ameya.models.Account">
    <constructor-arg value="Ameya"/>
    <constructor-arg value="10000"/>
    <constructor-arg value="Joshi"/>
  </bean>
</beans>
```


29

29

```
<beans ...>
  <bean id="account3" class=
    "com.ameya.models.Account">
    <constructor-arg type="java.lang.String"
      value="Ameya"/>
    <constructor-arg type="double"
      value="10000"/>
    <constructor-arg type="java.lang.String"
      value="Joshi"/>
  </bean>
```

30


30



```
<beans...>
  <bean id="account4"
    class="com.ameya.models.Account">
    <constructor-arg value="Ameya"
      index="0"/>
    <constructor-arg value="Joshi"
      index="1"/>
    <constructor-arg value="10000"
      index="2"/>
    </bean>
</beans>
```

31

31



```
<beans...>
  <bean id="account5"
    class="com.ameya.models.Account">
    <constructor-arg name="firstName"
      value="Ameya"/>
    <constructor-arg name="lastName" value=
      "Joshi"/>
    <constructor-arg name="balance" value=
      "10000"/>
    </bean>
</beans>
```

32

32

Setter-based dependency Injection

```
<bean id="account6" class="com.ameya.models.Account">
  <property name="firstName" value="Ameya"/>
  <property name="lastName" value="Joshi"/>
  <property name="balance" value="10000"/>
</bean>
<bean id="address" class="com.ameya.models.Address">
  <property name="building" value="Champions"/>
  <property name="street" value="JM Road">
  <property name="city" value="Pune"/>
  <property name="country" value="India"/>
  <property name="state" value="MH"/>
</bean>
```

33

33

```
<bean id="account7"
class="com.ameya.models.Account">
  <property name="firstName" value="Ameya"/>
  <property name="lastName" value="Joshi"/>
  <property name="balance" value="10000"/>
  <property name="address" ref="address"/>
</bean>
```

34

34

Factory Methods

```
package com.ameya.services;  
public class AccountService {  
    private static AccountService service = new AccountService();  
    private AccountService() {}  
    public static AccountService getInstance(){return service;  
    }  
    public static AccountService getInstance(String serviceName){  
        System.out.println("Service Name :"+serviceName);  
        return service;  
    }  
}
```

35

35

```
<bean id="accountService"  
class="com.ameya.services.AccountService"  
factory-method="getInstance"/>  
  
<bean id="accountService2"  
class="com.ameya.services.AccountService"  
factory-method="getInstance">  
    <constructor-arg name="serviceName"  
        value="Account"/>  
</bean>
```

36

36

Assigning Values To Properties

- * In Spring, there are multiple ways to inject values into bean properties.
- * Normal way
- * shortcut
- * "p" namespace (Spring 2.0 and later)
- * "c" namespace (Spring 2.0 and later)

37

37

* Normal way :

```
<bean id="account" class="com.ameya.models.Account">
  <property name="firstName">
    <value>Ameya</value>
  </property>
  <property name="lastName">
    <value>Joshi</value>
  </property>
  <property name="balance">
    <value>10000</value>
  </property>
</bean>
```

38

38

* **Shortcut Way :**

```
<bean id="account" class="com.ameya.models.Account">
    <property name="firstName" value="Ameya"/>
    <property name="lastName" value="Joshi"/>
    <property name="balance" value="10000"/>
</bean>
```

39

39

* **"p" namespace :**

```
<bean id="account" class="com.ameya.models.Account"
    p:firstName="Ameya"
    p:lastName="Joshi"
    p:balance="10000">
</bean>
```

40

40

- * **"c" namespace :**

```
<bean id="account" class="com.ameya.models.Account"  
      c:firstName="Ameya"  
      c:lastName="Joshi"  
      c:balance="10000">  
</bean>
```

41

41

Autowiring

- * The Spring container can autowire relationships between collaborating beans.
- * You can allow Spring to resolve collaborators (other beans) automatically for your bean by inspecting the contents of the ApplicationContext.

42

42

Autowiring

- * The autowiring functionality has four modes:
- * **no** (Default) : No autowiring. Bean references must be defined via ref element.
- * **byName** : Autowiring by property name. Spring looks for a bean with the same name as the property that needs to be autowired.
- * **byType** : Allows a property to be autowired if exactly one bean of the property type exists in the container. If more than one exists, a fatal exception is thrown, which indicates that you may not use byType autowiring for that bean. If there are no matching beans, nothing happens; the property is not set.
- * **constructor** : Analogous to byType, but applies to constructor arguments. If there is not exactly one bean of the constructor argument type in the container, a fatal error is raised.

43

43

@Autowired

- * The @Autowired annotation provides more fine-grained control over where and how autowiring should be accomplished.
- * The @Autowired annotation can be used to auto wire bean on the setter method just like @Required annotation, constructor, a property or methods with arbitrary names and/or multiple arguments.

44

44

@Autowired on Setter Methods

- * You can use @Autowired annotation on setter methods to get rid of the <property> element in XML configuration file.
- * When Spring finds an @Autowired annotation used with setter methods, it tries to perform byType autowiring on the method.

```
import org.springframework.beans.factory.annotation.Autowired;
public class TextEditor {
    private SpellChecker spellChecker;
    @Autowired
    public void setSpellChecker( SpellChecker spellChecker ){
        this.spellChecker = spellChecker;
    }
    ....
}
```

45

45

@Autowired on Properties

- * You can use @Autowired annotation on properties to get rid of the setter methods.
- * When you will pass values of auto wired properties using @Autowired Spring will automatically assign those properties with the passed values or references.

```
import org.springframework.beans.factory.annotation.Autowired;
public class TextEditor {
    @Autowired
    private SpellChecker spellChecker;
    ....
}
```

46

46

@Autowired on Constructor

- * You can apply @Autowired to constructors as well.
- * A constructor @Autowired annotation indicates that the constructor should be autowired when creating the bean, even if no <constructor-arg> elements are used while configuring the bean in XML file.

```
import org.springframework.beans.factory.annotation.Autowired;
public class TextEditor {
    private SpellChecker spellChecker;
    @Autowired
    public TextEditor(SpellChecker spellChecker){
        System.out.println("Inside TextEditor constructor.");
        this.spellChecker = spellChecker;
    }
    ....
}
```

47

47

Bean Scopes

- * When you create a bean definition, you create a recipe for creating actual instances of the class defined by that bean definition.
- * The idea that a bean definition is a recipe is important, because it means that, as with a class, you can create many object instances from a single recipe.

48

48

Bean Scopes

- * Singleton
- * Prototype
- * Request
- * Session
- * GlobalSession

49

49

Annotation-based container configuration

- * Starting from Spring 2.5 it became possible to configure the dependency injection using annotations. So instead of using XML to describe a bean wiring, you can move the bean configuration into the component class itself by using annotations on the relevant class, method, or field declaration.
- * Annotation injection is performed before XML injection, thus the later configuration will override the former for properties wired through both approaches.

50

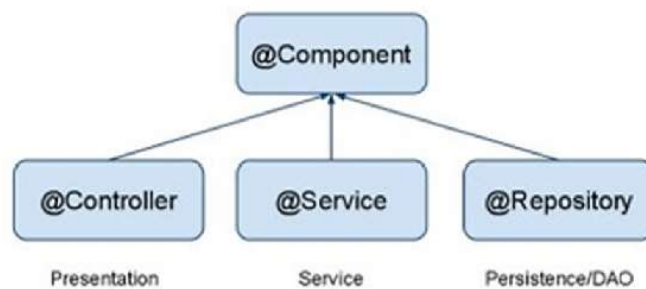
50

- * Annotation wiring is not turned on in the Spring container by default. So, before we can use annotation-based wiring, we will need to enable it in our Spring configuration file.
- * To do this You need to add the context schema and the below tag to xml file
 - * **<context: annotation-config />**

51

51

StereoType Annotations



52

52

@Component

- * Annotate your other components (for example REST resource classes) with @Component.

```
@Component
public class ContactResource {
    .....
}
```

@Component is a generic stereotype for any Spring-managed component.

@Repository, @Service, and @Controller are specializations of @Component for more specific use cases, for example, in the persistence, service, and presentation layers, respectively.

53

53

@Service

- * Annotate all your service classes with @Service.
- * All your business logic should be in Service classes.

```
@Service
public class CompanyServiceImpl implements
CompanyService {
    .....
}
```

54

54

@Repository

- * Annotate all your DAO classes with @Repository.
- * All your database access logic should be in DAO classes.

@Repository

```
public class CompanyDAOImpl implements CompanyDAO {  
    ....  
}
```

55

55

Java Based Configuration

- * Java based configuration option enables you to write most of your Spring configuration without XML but with the help of few Java-based annotations.
- * @Configuration
- * @Bean
- * @Import
- * @Scope

56

56

@Configuration & @Bean

- * Annotating a class with the @Configuration indicates that the class can be used by the Spring IOC container as a source for bean definitions.
- * The @Bean annotation tells Spring that a method annotated with @Bean will return an object that should be registered as a bean in the Spring application context.

57

57

```
import org.springframework.context.annotation.*;
@Configuration
public class HelloWorldConfig {
    @Bean
    public HelloWorld helloWorld(){
        return new HelloWorld();
    }
}
```

- * Above code will be equivalent to the following XML configuration:

```
<beans>
    <bean id="helloWorld" class="com.ameya.Helloworld"/>
</beans>
```

58

58

@Scope

- * The default scope is singleton, but you can override this with the @Scope annotation as follows:

```
@Configuration
public class AppConfig {
    @Bean
    @Scope("prototype")
    public Employee employee {
        return new Employee();
    }
}
```

59

59

Spring JDBC And Transactions

60

60

Spring DAO Templates

- * Built in code templates that support JDBC, Hibernate, JDO, and iBatis SQL Maps.
- * Simplifies data access coding by reducing redundant code and helps avoid common errors.
- * Alleviates opening and closing connections in your DAO code.
- * No more ThreadLocal or passing Connection/Session objects.
- * Transaction management is handled by a wired bean.
- * You are dropped into the template with the resources you need for data access – Session, PreparedStatement, etc..
- * Optional separate JDBC framework.

61

61

Consistent Exception Handling

- * Spring has it's own exception handling hierarchy for DAO logic.
- * No more copy and pasting redundant exception logic!
- * Exceptions from JDBC, or a supported ORM, are wrapped up into an appropriate, and consistent, `DataAccessException` and thrown.
- * This allows you to decouple exceptions in your business logic.
- * These exceptions are treated as unchecked exceptions that you can handle in your business tier if needed. No need to try/catch in your DAO.
- * Define your own exception translation by subclassing classes.

62

62

Transactions

- * A transaction comprises a unit of work performed within a database management system (or similar system) against a database.
- * Transactions provide “all-or-nothing” proposition. Each unit of work performed should be entirely committed or rolled back.

63

63

Transactions - ACID

- * Atomic: Atomicity ensures that all operations in the transaction happen or none of them happen.
- * Consistent: The system should be left in a consistent state even if the transaction succeeds or fails.
- * Isolated: Should allow multiple users to work. Transactions should be isolated from each others work. This prevents concurrent reads and writes.
- * Durable: After successful transaction, the system should store the details permanently.

64

64

Before Spring transactions

- * EJB was a powerful API available offering Bean Managed(BMT) and Container Managed Transactions(CMT).
- * It offers both programmatic and declarative based transaction management.
- * Problems:
 - * Needs an application server to run the CMT.
 - * EJB relies on JTA for transactions.
 - * Using EJB itself is a heavy choice.

65

65

Spring Transactions Advantages

- * Offers both programmatic and Declarative transactions.
- * Declarative transactions are equivalent of the Container managed transactions and there is no need for an application server.
- * No need for using JTA.
- * Wide range of Transactional Managers are defined in spring transactions API.

66

66

- * Declarative transactions on the other hands are less intrusive and are defined in a Configuration file.
- * Developed based on the AOP concepts. This gives an advantage of keeping the cross-cutting concerns like transactions out of our DAO layer code.

67

67

Spring TransactionManagers

```
<bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
  <property name="dataSource" ref="dataSource"/>
</bean>

<bean id="transactionManager"
class="org.springframework.orm.hibernate3.HibernateTransactionManager"> <property
name="sessionFactory" ref="sessionFactory"/>
</bean>

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

<bean id="transactionManager"
class="org.springframework.transaction.jta.JtaTransactionManager"> <property
name="transactionManagerName" value="java:/TransactionManager"/>
</bean>
```

68

68

With Annotations

```
* @Bean(name="transactionManager")
public PlatformTransactionManager transactionManager() {
    DataSourceTransactionManager transactionManager = new
        DataSourceTransactionManager();//XXXTransactionManager();
    transactionManager.setDataSource(dataSource());
    return transactionManager;
}
```

**** Note DataSource must be injected in TransactionManager**

69

ISOLATION Levels

- * Dirty reads occur when transaction reads an uncommitted data.
- * Non-repeatable Reads occurs when a transaction reads the same data multiple times but gets different results each time.
- * Phantom Reads occur when two transactions work on a same row where one updates and other reads. The reading transaction gets new data.

70

70

- * **ISOLATION_DEFAULT**: default isolation specific to the data source.
- * **ISOLATION_READ_UNCOMMITTED**: Read changes that are uncommitted. Leads to dirty reads, Phantom reads and non repeatable reads.
- * **ISOLATION_READ_COMMITTED**: Reads only committed data. Dirty read is prevented but repeatable and non repeatable reads are possible.
- * **ISOLATION_REPEATABLE_READ**: Multiple reads of same field yield same results unless modified by same transaction. Dirty and non repeatable reads are prevented but phantom reads are possible as other transactions may edit the fields.
- * **ISOLATION_SERIALIZABLE**: Dirty, phantom and non repeatable reads are prevented. But hampers the performance of application.

71

71

Read-Only

- * A read-only transaction does not modify any data.
- * Read-only transactions can be a useful optimization in some cases

72

72

Scope of Beans

- * When defining a <bean> in Spring, a scope for that bean needs to be specified.
- * There are five scopes for a bean in Spring
 - * Singleton
 - * Prototype
 - * Request
 - * Session
 - * Global-session

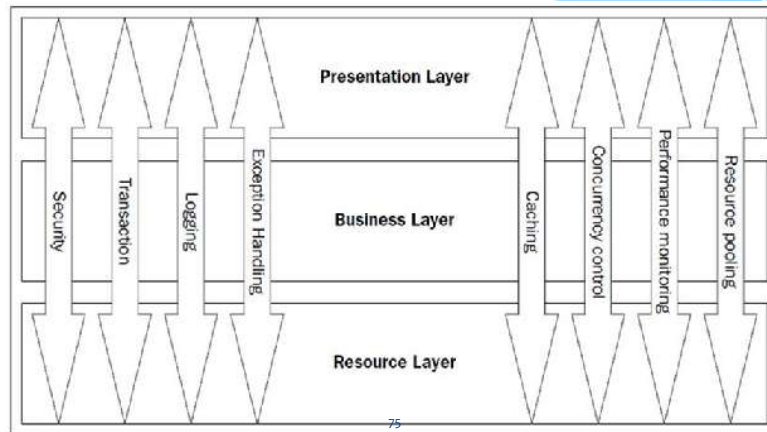
73

Spring AOP

74

74

Object Oriented Programming-Limits



75

Aspect-oriented programming

- * Aspect-oriented programming supports object-oriented programming by de-coupling modules that implement cross-cutting concerns.
- * Its purpose is the separation of concerns.
- * In object-oriented programming the basic unit is the Class, whereas in aspect-oriented programming it's the Aspect.

76

76

AOP terms: (Aspect)

- * A modularization of a concern that cuts across multiple classes.
- * Transaction management is a good example of a cross-cutting concern in enterprise Java applications.

77

77

AOP terms: (Join point)

- * A point during the execution of a program, such as the execution of a method or the handling of an exception.
- * In Spring AOP a join point always represents a method execution.

78

78

AOP terms: (Advice)

- * This is action taken by an aspect at a particular join point.
- * Different types of advice include "around", "before" and "after" advice.
- * Many AOP frameworks, including Spring, model an advice as an interceptor, maintaining a chain of interceptors around the join point.

79

79

AOP terms: (Pointcut)

- * A predicate that matches join points.
- * Advice is associated with a pointcut expression and runs at any join point matched by the pointcut (for example, the execution of a method with a certain name).

80

80

Types Of Advice

Type	Interface
Around	org.aopalliance.intercept.MethodInterceptor
Before	org.springframework.aop.BeforeAdvice
After	org.springframework.aop.AfterReturningAdvice
Throws	org.springframework.aop.ThrowsAdvice

81

81

AspectJ Advices

- * @Before
- * @After
- * @AfterReturning
- * @AfterThrowing
- * @Around

82

82

Spring MVC

83

83

Spring MVC

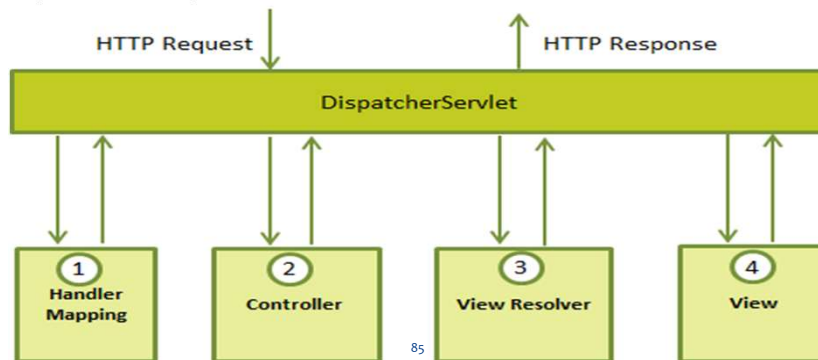
- * The Spring web MVC framework provides model-view-controller architecture and ready components that can be used to develop flexible and loosely coupled web applications.
- * The Model encapsulates the application data and in general they will consist of POJO.
- * The View is responsible for rendering the model data and in general it generates HTML output that the client's browser can interpret.
- * The Controller is responsible for processing user requests and building appropriate model and passes it to the view for rendering.
- * Thus MVC pattern results in separating the different aspects of the application (input logic, business logic, and UI logic), while providing a loose coupling between these elements.

84

84

The DispatcherServlet

- * The Spring Web model-view-controller (MVC) framework is designed around a DispatcherServlet that handles all the HTTP requests and responses.



85

85

Controller & View

* HomeController.java

```
@Controller
public class HomeController {
    @RequestMapping(value = { "/", "/home" },
        method = RequestMethod.GET)
    public String showHomePage(ModelMap
        model) {
        model.addAttribute("message", "Hello
        Spring MVC Framework!");
        return "home";
    }
}
```

* home.jsp

```
<html>
<head>
<title>Here is home page</title>
</head>
<body>
<h2>${message}</h2>
</body>
</html>
```

86

86

Request Mappings

- * RequestMappings are really flexible
- * You can define a @RequestMapping on a class and all methods
- * @RequestMapping will be relative to it.
- * There are a number of ways to define them:
 - * URI Patterns
 - * HTTP Methods (GET, POST, etc)
 - * Request Parameters
 - * Header values

87

87

RequestMapping - Class Level

```
package com.ameya.controllers;  
  
@RequestMapping ("/portfolio")  
@Controller  
public class PortfolioController {  
    @RequestMapping ("/create")  
    public String create() {  
        return "create";  
    }  
}
```

- * The URL for this (relative to your context root) would be:
/portfolio/create

88

88

RequestMapping - HTTP Methods

```
package com.ameya.controllers;

@RequestMapping("/portfolio")
@Controller
public class PortfolioController {
    @RequestMapping(value="/create",method=RequestMethod.POST)
    public String save() {
        return "view";
    }
}
```

* Same URL as the previous example, but responds to POSTs

89

89

RequestMapping - Request Params

```
package com.ameya.controllers;

@RequestMapping("/portfolio")
@Controller
public class PortfolioController {
    @RequestMapping(value="/view",params="details=all")
    public String viewAll() {
        return "viewAll";
    }
}
```

* This will respond to
/portfolio/view?details=all

90

90

RequestMapping - URI Templates

```
package com.ameya.controllers;

@RequestMapping("/portfolio")
@Controller
public class PortfolioController {
    @RequestMapping("/viewProject/{projectId}")
    public String viewProject() {
        return "viewProject";
    }
}

* The URL for this (relative to your context root) would be:
  /portfolio/viewProject/10
```

91

91

Controller Method Arguments

- * Sometimes you need access to the request, session, request body, or other items
- * If you add them as arguments to your controller method, Spring will pass them in.

```
@RequestMapping(value="/")
public String getProject(HttpServletRequest request,
                        HttpSession session,
                        @RequestParam("projectId") Long projectId,
                        @RequestHeader("content-type") String contentType)
{
    return "index";
}
```

92

92

Spring REST

93

93

HTTP Message

* What does an HTTP message look like?

```
GET /view/1 HTTP/1.1 ← Request Line
User-Agent: Chrome    ← Headers
Accept: application/json
[CRLF]

POST /save HTTP/1.1 ← Request Line
User-Agent: IE
Content-Type: application/x-www-form-urlencoded ← Headers
[CRLF]
name=x&id=2 ← Request Body
```

94

94

HTTP Message - Responses

```
HTTP/1.1 200 OK ← Status Line
Content-Type: text/html ← Headers
Content-Length: 1337
[CRLF]
<html>
Some HTML Content. ← Response Body
</html>
```

```
HTTP/1.1 500 Internal Server Error ← Status Line
```

```
HTTP/1.1 201 Created ← Status Line
Location: /view/7 ← Headers
[CRLF]
Some message goes here. ← Response Body
```

95

95

RequestBody

- * Annotating a handler method parameter with `@RequestBody` will bind that parameter to the request body

```
@RequestMapping("/echo/string")
public void writeString(@RequestBody String input) {}
```

```
@RequestMapping("/echo/json")
public void writeJson(@RequestBody SomeObject input) {}
```

96

96

ResponseBody

- * Annotating a return type with `@ResponseBody` tells Spring MVC that the object returned should be treated as the response body

```
@RequestMapping("/echo/string")  
public @ResponseBody String readString() { }
```

```
@RequestMapping("/echo/json")  
public @ResponseBody SomeObject readJson() { }
```

97

97

HttpMessageConverters

- * How does Spring MVC know how to turn a JSON string into `SomeObject`, or vice-versa?
- * `HttpMessageConverters`
- * These are responsible for converting a request body to a certain type, or a certain type into a response body
- * Spring MVC figures out which converter to use based on Accept and Content-Type headers, and the Java type
- * Your Accept and Content-Type headers DON'T have to match. For example, you can send in JSON and ask for XML back

98

98

StringHttpMessageConverter

- * Reads and writes Strings.
- * Reads text/*
- * Writes text/plain

99

99

StringHttpMessageConverter

```
@RequestMapping("/echo/string")  
public @ResponseBody String echoString(@RequestBody String input) {  
    return "Your Text Was: " + input;  
}
```

```
POST /echo/string HTTP/1.1  
Accept: text/plain  
Content-Type: text/plain  
  
Hello!
```

REQUEST

```
HTTP/1.1 200 OK  
Content-Type: text/plain  
Content-Length: 17  
  
Your Text Was: Hello!
```

RESPONSE

100

100

MappingJacksonHttpMessageConverter

- * Maps to/from JSON objects using the Jackson library
- * Reads application/json
- * Writes application/json

101

101

MappingJacksonHttpMessageConverter

```
public Person {  
    // String name, int age;  
}  
  
@RequestMapping("/echo/json")  
public @ResponseBody Person echoJson(@RequestBody Person person) {  
    // Upper case name, square age  
    return person;  
}
```

```
POST /echo/string HTTP/1.1  
Accept: application/json  
Content-Type: application/json  
  
{ "name" : "Spencer", "age" : 5 }
```

REQUEST

```
HTTP/1.1 201 Created  
Content-Type: application/json  
  
{ "name" : "SPENCER", "age" : 25 }
```

RESPONSE

102

102

Jaxb2RootElementHttpMessageConverter

- * Maps to/from XML objects
- * Must have your object at least annotated with `@XmlElement`
- * Reads text/xml, application/xml
- * Writes text/xml, application/xml

103

103

Jaxb2RootElementHttpMessageConverter

```
@XmlElement
public Person { // String name, int age; }

@RequestMapping("/echo/xml")
public @ResponseBody Person echoXml(@RequestBody Person person) {
    // Upper case name, square age
    return person;
}
```

```
POST /echo/string HTTP/1.1
Accept: application/xml
Content-Type: application/xml
```

REQUEST

```
<thing><name>Spencer</name><age>5</age></thing>
```

```
HTTP/1.1 201 Created
Content-Type: application/xml
```

RESPONSE

```
<thing><name>SPENCER</name><age>25</age></thing>
```

104

Other parts of the HttpResponseMessage

- * What if you need to get/set headers?
- * Or set the status code?

```
@RequestMapping("/echo/string")
public String echoString(
    @RequestBody String input,
    HttpServletRequest request,
    HttpServletResponse response) {
    String requestType = request.getHeader("Content-Type");
    response.setHeader("Content-Type", "text/plain");
    response.setStatus(200);
    return input
}
```

105

105

@ResponseStatus

- * There is a convenient way to set what the default status for a particular handler should be

```
@RequestMapping("/create")
@ResponseStatus(HttpStatus.CREATED) // CREATED = 201
public void echoString(String input) {
}
```

106

106

HttpEntity

- * Convenience class for dealing with bodies, headers, and status
- * Converts messages with `HttpMessageConverters`

```
@RequestMapping("/image/upload")
public ResponseEntity<String> upload(HttpEntity<byte[]> rEntity) {
    String t = rEntity.getHeaders().getFirst("Content-Type");
    byte[] data = rEntity.getBody();// Save the file
    HttpHeaders responseHeader = new HttpHeaders();
    responseHeader.set("Location", "/image/1");
    return new ResponseEntity<String>
        ("Created",responseHeader, HttpStatus.CREATED);
}
```

107

107

Spring REST Template

112

112

RestTemplate

- * RestTemplate communicates with HTTP server using RESTful principals.
- * RestTemplate provides different methods to communicate via HTTP methods.
- * This class provides the functionality for consuming the REST Services in a easy and graceful manner.
- * When using the said class the user has to only provide the URL, the parameters(if any) and extract the results received.
- * The RestTemplate manages the HTTP connections.

113

113

RestTemplate Methods

HTTP method	RestTemplate methods
DELETE	<code>delete(java.lang.String, java.lang.Object...)</code>
GET	<code>getForObject(java.lang.String, java.lang.Class<T>, java.lang.Object...)</code> <code>getForEntity(java.lang.String, java.lang.Class<T>, java.lang.Object...)</code>
POST	<code>postForLocation(java.lang.String, java.lang.Object, java.lang.Object...)</code> <code>postForObject(java.lang.String, java.lang.Object, java.lang.Class<T>, java.lang.Object...)</code>
PUT	<code>put(java.lang.String, java.lang.Object, java.lang.Object...)</code>

114

HTTP GET Using RestTemplate

```
RestTemplate restTemplate = new RestTemplateO;  
  
String url ="http://localhost:8080/demo/rest/employees/{id};  
  
Map<String, String> map = new HashMap<String,String>();  
map.put("id", "101");  
  
ResponseEntity<Employee> entity =restTemplate.getForEntity(url,  
Employee.class, map);  
  
System.out.println(entity.getBody());
```

115

HTTP POST Using RestTemplate

```
RestTemplate restTemplate = new RestTemplate();  
  
String url ="http://localhost:8080/demo/rest/employees";  
  
Employee employee =restTemplate.postForObject(url,  
newEmployee,Employee.class);  
  
System.out.println(employee);
```

116

116

SPRING BOOT

117

Spring Framework Limitations

- * Huge framework
- * Multiple setup steps
- * Multiple configuration steps
- * Multiple Build and Deploy steps
- * **Can We abstract these all steps?**

118

SPRING BOOT

119

What is Spring Boot?

- * Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that you can “just run”.

120

What is Spring Boot?

- * Opinionated (It makes certain assumptions)
- * Convention Over Configuration
- * Stand alone
- * Production ready
- * Spring module which provides RAD (Rapid Application Development) feature to Spring framework.

121

Features

- * Create stand-alone Spring applications
- * Embed Tomcat, Jetty or Undertow directly (no need to deploy WAR files)
- * Provide opinionated 'starter' POMs to simplify your Maven configuration
- * Automatically configure Spring whenever possible
- * Provide production-ready features such as metrics, health checks and externalized configuration
- * Absolutely **no code generation** and **no requirement for XML** configuration

122

Setup Spring Boot

- * Pre-requisites
 - * Hardware
 - * Core i5 machine
 - * 4 gb ram
 - * Software
 - * 64 bit Windows 7/10
 - * Java 1.8
 - * Spring Tools Suite (We use sts 3.9.2)

123

Setup Spring Boot

- * Install Maven
 - * Set MAVEN_HOME
 - * Add it to PATH Environment Variable
 - * Run `mvn -version` from command prompt to ensure maven is installed

124

Setup Spring Boot

- * Start STS
- * Create new Maven Project
 - * In the STS UI
 - * Check Create a simple project
 - * Click on next
 - * Enter Group Id (com.ameya)
 - * Enter Artifact Id (course-api)
 - * Enter Version (Keep default)
 - * Enter Name (Ameya Joshi Course Api)

125

Setup Spring Boot

- * Add following in pom.xml ,
 - * save the file and update the Maven Project
- ```
<parent>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-parent</artifactId>
 <version>2.1.2.RELEASE</version>
</parent>
<dependencies>
 <dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-web</artifactId>
 </dependency>
</dependencies>
<properties>
 <java.version>1.8</java.version>
</properties>
```

126

## Few more dependencies.

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

<dependency>
 <groupId>org.apache.derby</groupId>
 <artifactId>derby</artifactId>
 <scope>runtime</scope>
</dependency>
```

127

## Few more dependencies.

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

<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-eureka</artifactId>
</dependency>
```

128

## Writing First App

\* Type following code and run as java application

```
package com.ameya;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class CourseApiApp {

 public static void main(String[] args) {
 SpringApplication.run(CourseApiApp.class, args);
 }

}
```

129

## Behind the Scenes

```
SpringApplication.run(CourseApiApp.class, args);
```

This runs the CourseApiApp class

This class is annotated with @SpringBootApplication

The @SpringBootApplication annotation is equivalent to using @Configuration, @EnableAutoConfiguration, and @ComponentScan

130

## Behind the Scenes

- \* As a result Spring Boot :
- \* Sets up the default configuration
- \* Starts Spring application context
- \* Performs classpath scan
- \* Starts tomcat server

131

## Few Other ways to start Spring Boot Application

- \* Spring Initializr
- \* Spring Boot CLI
- \* STS IDE

132



## Spring Initializr

- \* Browse to :
  - \* <http://start.spring.io/>
- \* Fill in the necessary and required fields.
- \* Add the required dependencies
- \* Click on the Generate Project.
- \* Extract the downloaded zip file/
- \* Go through the pom.xml

133

## Spring Boot CLI(Command Line Interface)

- \* It is a command line tool that can be used if you want to quickly prototype with spring. It allows you to run Groovy scripts, which means that you have familiar Java-like syntax, without much boilerplate code.
- \* You can download The Spring CLI distribution from the spring software repository
- \* Once downloaded the zip file, follow the instructions from the install.txt from unpacked archive.
- \* You could also use SDKMAN(Windows)/Homebrew(Osx) for installation.

134

## STS IDE

- \* Create new Spring starter Project
- \* In the Guided UI fill in the appropriate values
- \* Select the required dependencies