Spring Framework

Devdatta Kulkarni

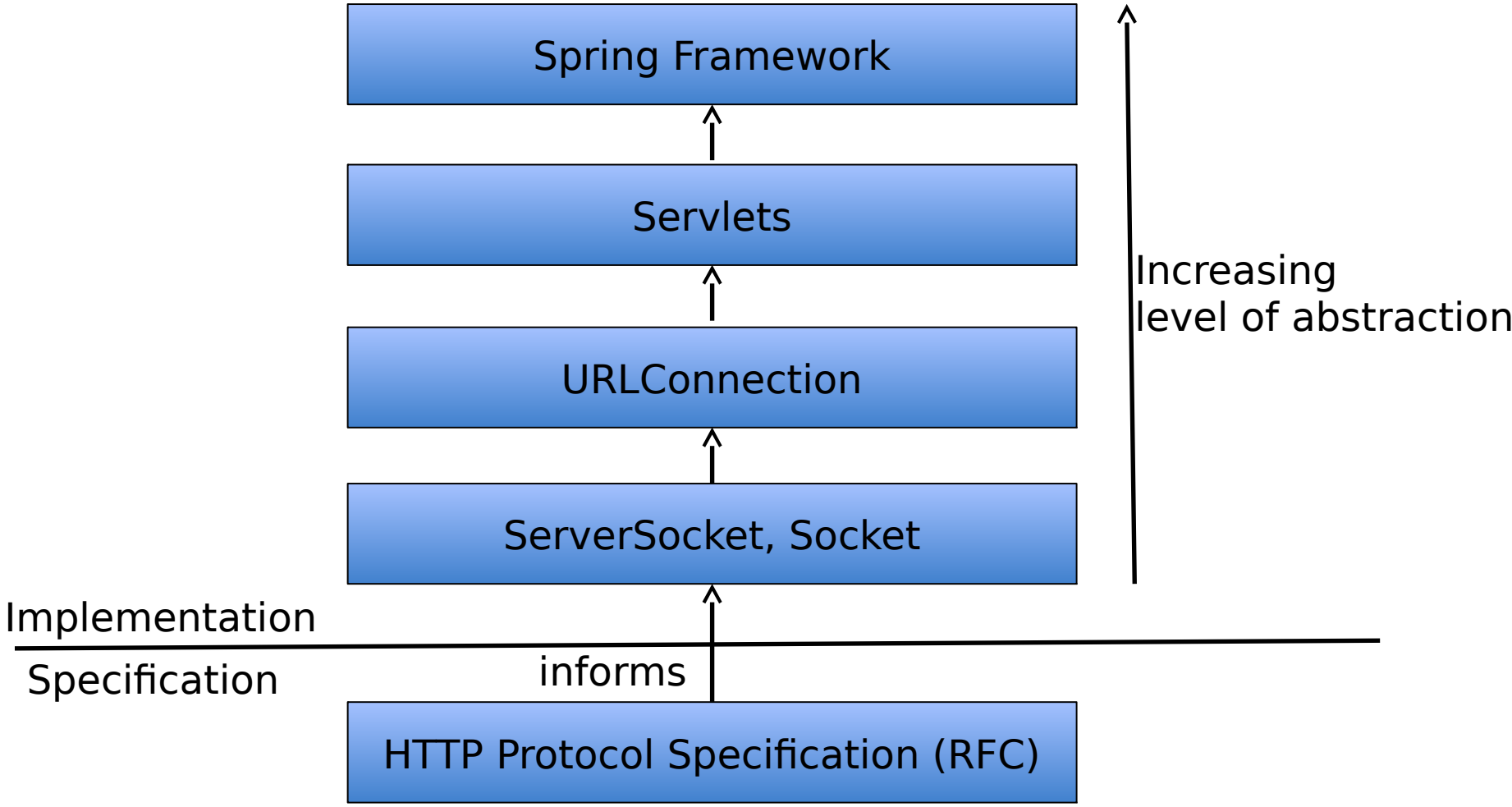
Issue with Servlets

- Interface is oriented towards underlying low-level concepts (e.g.: HTTP), rather than application-level concepts
  - *doGet*, *doPost*, instead, for example, *queryAustinSocialMedia*

Spring Framework:

- What is it?
  - Its an *application container* for Java that provides many useful features, such as
    - *Dependency Injection*
    - Transaction management
    - Support for developing aspect oriented programs
- Why is it useful?
  - One level more abstracted from Servlets
    - Request/Response are abstracted
  - Provides method-level mapping for resources
    - Logical code groupings
      - Map Servlet actions to methods

Levels of Abstraction



Dependency Injection

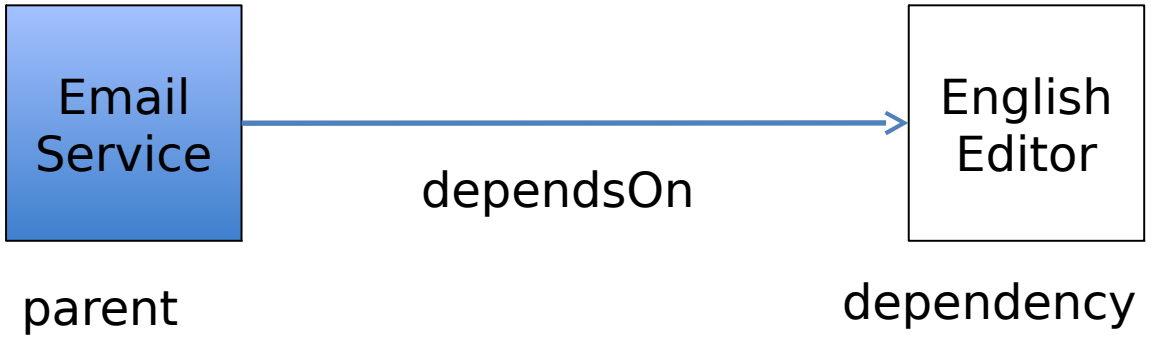
- What is it?
  - Separating creation of dependencies from their consumption
- Why dependency injection matters?
  - Allows flexibility in configuring a class
    - E.g.: Easy to provide Test class as a dependency for testing vs. real dependency
- How to achieve it?
  - Developing against an Interface instead of concrete class
  - Let the application container like Spring to construct the *parent -> dependency* object graph

Need of dependency injection

- To demonstrate this issues lets look at an another example
- Suppose we want to build a *Email Service* that supports composing emails in *English or Spanish*
- Suppose that the ability to compose emails in English is provided by an object of type *EnglishEditor*
- To compose emails in English the Email Service needs an instance of the *EnglishEditor* object

Email Service

- Build a Email Service that supports creation of emails in *English*



Reference: Example referenced from book *"Dependency Injection"* by Dhanji R. Prasanna

Email Service: Initial design

```
public class EmailService {
    private class EnglishEditor editor;

    public EmailService() {
        this.editor = EnglishEditor();
    }

    public void sendEmail() {
        this.editor.compose();
    }
}
```

Create the dependency in the constructor

Problems with this approach?

- Cannot easily modify EmailService to support emails in Spanish

- Cannot easily test EmailService's functionality
  - For instance, we would like to test that:
    - When *sendEmail* method is called
    - Verify that the *compose* method of the editor is called

Addressing concrete binding issue

- Develop against an *Interface* instead of using *concrete* implementation classes
  - Instead of using *EnglishEditor* within EmailService, define and use *Editor* interface
    - Interface vs. implementation distinction example
      - Map is an interface
      - HashMap, Hashtable, TreeMap, LinkedHashMap are implementations of the Map interface
- How is this useful?
  - The Editor class is free from any one particular implementation

Addressing testability issue

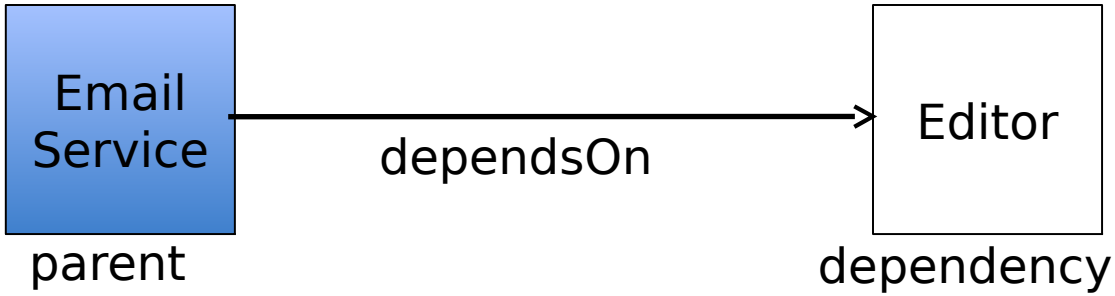
- *Externalize* dependency creation
  - Don’t create the dependency in the constructor, but create it elsewhere and pass it to the parent
  - Verify that required methods were called on the dependency

Testing EmailService

```
public class EmailService {
    private class Editor editor;

public EmailService(Editor editor) {
        this.editor = editor;
 }

    public void sendEmail() {
        this.editor.compose();
    }
}
```

```
public class TestEmailService {
    private EmailService emailService;

  public void testEditorCompose() {
     Editor editor = mock(Editor.class);
     emailService = new
EmailService(editor)
     emailService.sendEmail();
     verify(editor.called(compose));
  }
}
```

Dependency Injection



EmailService *depends* on Editor

Spring Framework performs dependency injection by:

- creating a Editor object and passing it to the EmailService

Dependency Injection is also called *Inversion of Control (IoC)*

Why Inversion of Control?
Because instead of the parent object creating and instantiating the dependency, control of
creating the dependency is given to the container.

Container creates the dependency and injects it into the parent object

Spring Framework Basics

- Bootstrapping and configuration
- Dependency Injection
- Java Beans
- Dispatcher Servlet
- Annotations

Bootstrapping and Configuration

- Bootstrapping refers to starting up of the Spring framework
- Requires an instance of the *DispatcherServlet* given configuration file in the form of a *contextConfigLocation* init parameter and instructed to load on startup
- Example:
  - spring-email-service

contextConfigLocation: servletContext.xml

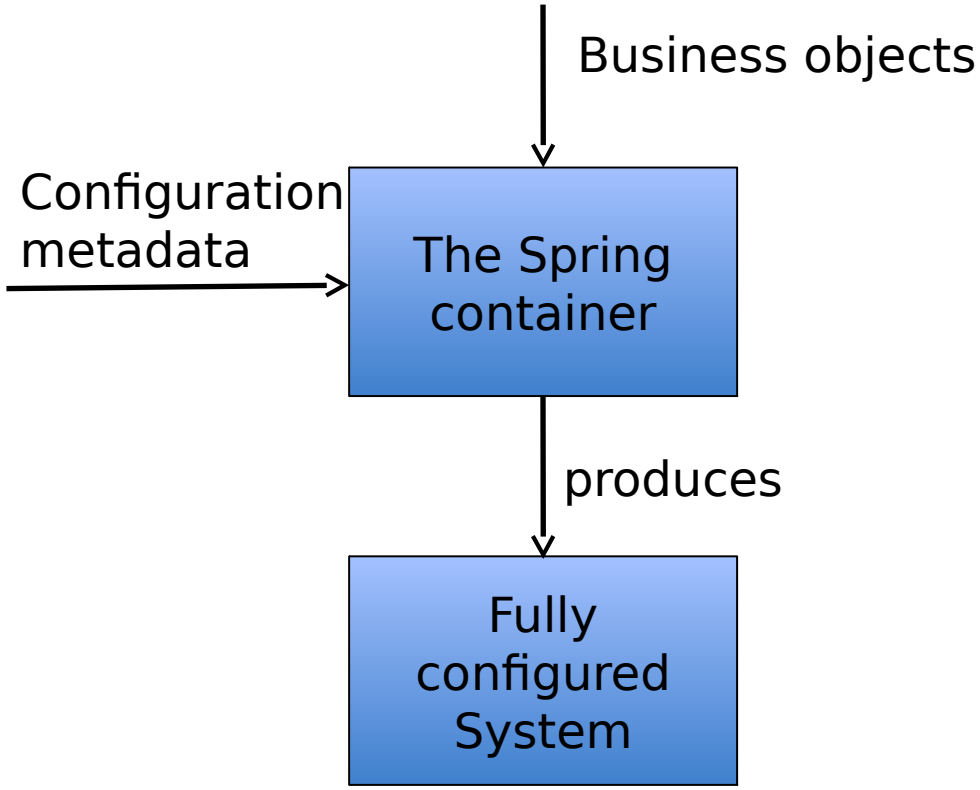
- Defines the configuration metadata about your application in XML
- Configuration metadata allows you to express the objects that compose your application and the rich interdependencies between such objects
- The objects that compose your application are called *beans*

contextConfigLocation: servletContext.xml

- A bean is an object that is instantiated, assembled, and managed by a Spring IoC container
- The container gets its instructions on what objects to instantiate, configure, and assemble by reading this configuration metadata

http://docs.spring.io/spring/docs/4.0.x/spring-framework-reference/html/beans.html#beans-introduction

Spring IoC



http://docs.spring.io/spring/docs/4.0.x/spring-framework-reference/html/beans.html#beans-introduction

Java Beans details

- Java Beans:
  - What is a Java Bean?
    - Java class that is built based on following convention:
      - Properties are accessible through getter/setter methods
        - » get/set + property_name_with_first_letter_capital
  - EmailController
    - Setter method for a property

Dispatcher Servlet

- Central dispatcher for HTTP request handlers/controllers
- Dispatches to registered handlers for processing a web request
- http://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/servlet/DispatcherServlet.html

Spring Example Details

- web.xml
  - Servlet name: springDispatcher
  - Init-param: contextConfigLocation
  - load-on-startup
- servletContext.xml
  - Beans:
    - emailController
    - editorServiceImpl

Dependency Injection

- Setter injection
  - setter methods need to be defined for each dependent variable
    - Example: EmailController
- Constructor injection
  - constructor parameters passed in for each dependent variable

Setter Injection

- In bean configuration file:

```
<bean name="emailController" class="EmailController”>
    <property name="editorService"
ref="editorServiceImpl" />
</bean>
```

- In code:

```
public void setEditorService(EditorService editorService)
{
     this.editorService = editorService;
}
```

Constructor Injection

- In bean configuration file:

```
<bean name="emailController" class="EmailController”>
  <constructor-arg
ref="englishEditorServiceImpl"></constructor-arg>
</bean>
```

Typical Issues in getting Spring working

- Class not found: DispatcherServlet
- Resolution:
  - Add Maven Dependencies to “Deployment Assembly”
    - In Eclipse,
      - Right click project -> Build Path -> Configure Build Path -> Deployment Assembly -> Add -> Java Build Path Entries -> Maven Dependencies
  - Add “build” and “WEB-INF/lib” folders to Java Build Path
    - In Eclipse,
      - Right click project -> Build Path -> Configure Build Path -> Java Build Path -> Source -> Add folder

Reading

- Chapter 12 from Java for Web Applications book

References

http://www.martinfowler.com/articles/injection.html

Application Context

- Represented by org.springframework.context.ApplicationContext interface
  - It represents the Spring container itself and is responsible for instantiating, configuring, and assembling the Java classes that make up your application
- A Spring application always has at least one application context