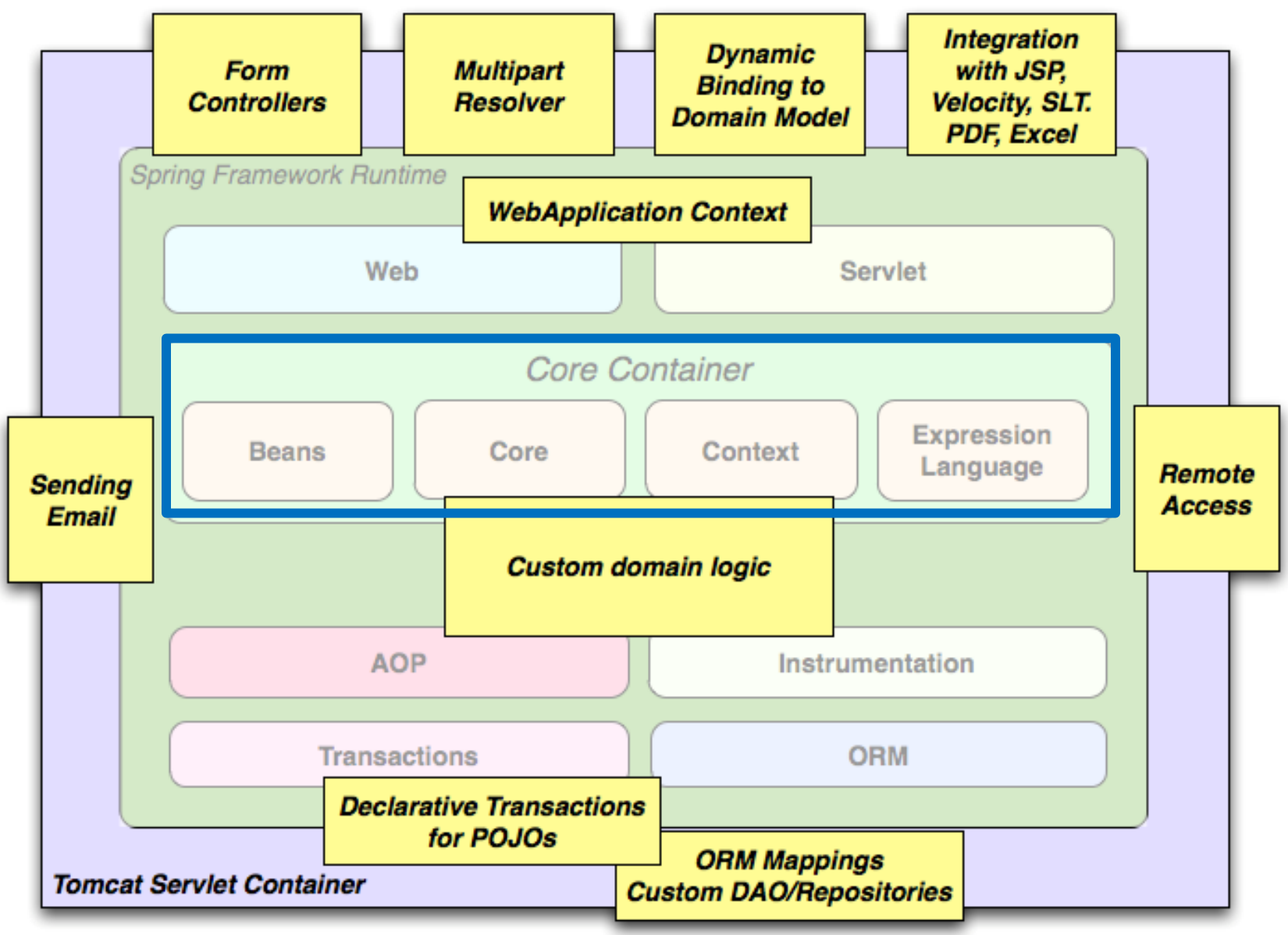
CORE SPRING FRAMEWORK

Water the Root

Spring Core


Form Controllers
Multipart Resolver
Dynamic Binding to Domain Model
Integration with JSP, Velocity, SLT. PDF, Excel
Spring Framework Runtime
WebApplication Context
Web
Servlet
Core Container
Beans
Core
Context
Expression Language
Sending Email
Remote Access
Custom domain logic
AOP
Instrumentation
Transactions
ORM
Declarative Transactions for POJOs
ORM Mappings Custom DAO/Repositories
Tomcat Servlet Container

Spring Core Technologies

- **IoC *****

  Inversion of Control Container

- **AOP *****

  Aspect-Oriented Programming

- **Validation*****

  Data Validation W/pluggable interface

- **SpEL ****

  Spring Expression Language that supports querying and manipulating an object graph at runtime.

- **Resource ****

  Common API that abstracts the type of underlying resource such as a URL, file or class path resource.

Core Technologies

Technologies absolutely integral to the Spring Framework.

Foremost is the Inversion of Control (IoC) container.

Inversion of Control [IoC]

A Programming Principle

"Hollywood Principle: Don't call us, we'll call you".

…the flow of application is inverted…

Advantages:

Decouples execution of a task from its implementation

Easier to switch between implementations

Greater modularity of a program

Allows components to communicate through contracts

Easier to isolate & mock dependencies for testing

Inversion of Control [IoC]

Inversion of Control is a common characteristic of a ***framework***

Any callable extension point defined by a framework is a form of IoC

The ***framework*** calls the developer, rather than the developer calling the framework.

```
public class MyServlet extends  HttpServlet {

  protected void doPost(…) {
     // developer code…
  }
  protected void doGet(…)  {
     // developer code
  }
```

The HttpServlet has program control.
The "developer" doGet() and doPost() are automatically called by the Servlet Framework

Inversion of Control [IoC] & Dependency Injection [DI]

The terms IoC and DI are often used interchangeably

HOWEVER

Dependency Injection ***is but one type*** of IoC

DEPENCENCY INJECTION means

Objects do not create other objects that they depend on.

IN OTHER WORDS

“Injecting” a dependency into a client, rather than a client actively creating the dependency, is the fundamental purpose of DI

Dependency Injection

Whenever we create object using

new()

we violate the

principle of programming to an interface rather than implementation

programming to implementation eventually results in code that is inflexible and difficult to maintain.

The CORE of Spring Core

The **HEART** of the Spring Framework is the

Spring Inversion Of Control [IOC]

Container

The Spring IoC container uses **DI** to Manage & Configure

Plain Old Java Objects [POJO]

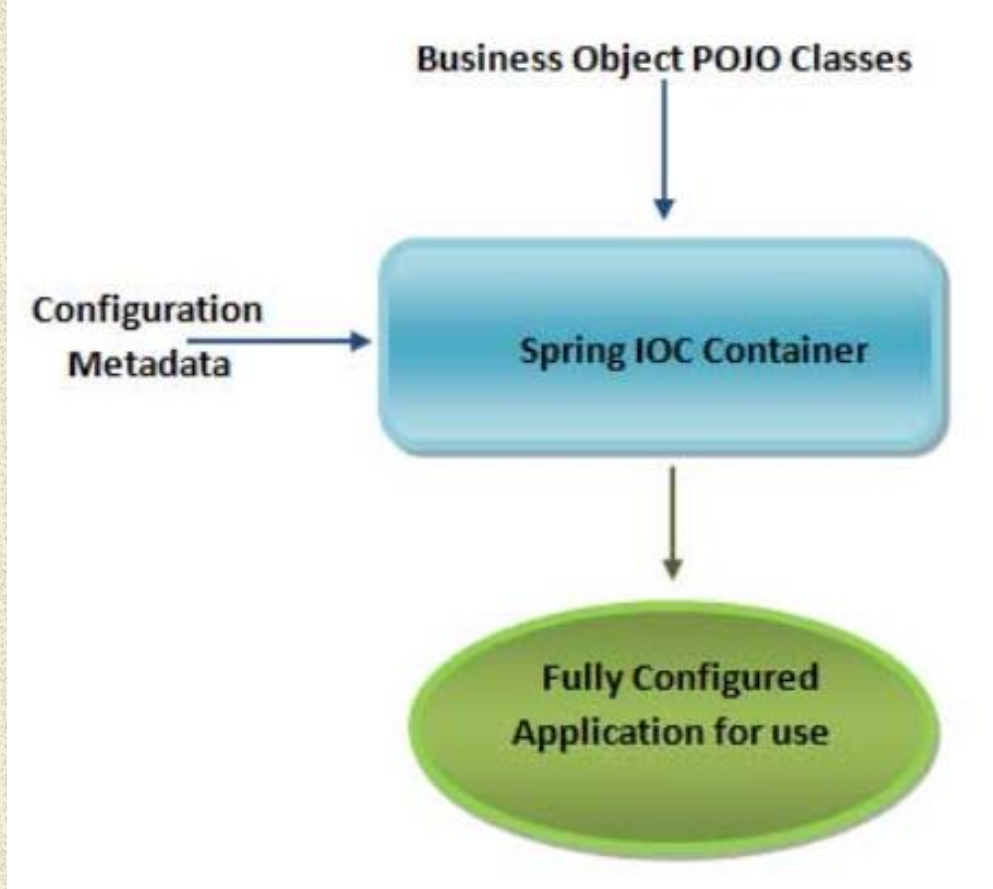
Through

Interfaces [usually] **

**** A Class can be Managed & Configured. However you lose some of the advantages[Testing, etc.]**

Spring Core – IoC Container

The Essence of a Spring Application



JavaBeans .vs. POJO .vs. Spring Bean

- JavaBean

  Adhere to Sun's JavaBeans specification

  In

  pu

  Reusable Java classes for visual application composition

POJO

  'Fancy' way to describe ordinary Java Objects

  D

  D ıment

  Simpler, lightweight compared to 'heavyweight' EJBs

Spring Bean

  Spring managed - configured, instantiated and injected

A Java object can be a JavaBean, a POJO and a Spring bean all at the same time.

Spring Documentation:
"Component" is used interchangeably with POJO class
Both mean a Java class from which an object instance is created
.

Spring Documentation:
"Bean" is used interchangeably with POJO instance
Both mean object instance created from a Java class.

Dependency Injection [DI]

Reduces "glue" code

Less setup of dependency in component

Simplifies Configuration

Declaratively re-configure to change implementations

Improves Testability

Substitute "mock" implementations

Fosters good design

Design to Interfaces….

Main Point

The Inversion of Control Container manages the lifecycle for the objects required by our application, allowing us to focus on the functionality of our logic and giving flexibility for future implementations.

Science of Consciousness: *Through the holistic field of life, the full range of life, the pure nature of creative intelligence, we can enrich all aspects of life.*

Hello World

Who has NOT seen it?

```
package edu.mum;

public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

See Demo HelloWorld

BUT the Message is HARDCODED!

As is the Display Mechanism!

Let's IMPROVE the Design

"Externalize" Message & Display Mechanisms

```
public class ConfigInMemory {
private static final ConfigInMemory instance = new ConfigInMemory();
 // HashMap containing the "managed" beans
 Map<String, Object> beans = new HashMap<String, Object>();

 ConfigInMemory() {
    // HARD CODE the "managed" beans
    StandardOutMessageDisplay standardOutMessageDisplay =
                                        new StandardOutMessageDisplay();
    HelloWorldMessageSource helloWorldMessageSource =
                                          new HelloWorldMessageSource();
    // MANUALLY DI
    StandardOutMessageDisplay.setMessageSource(helloWorldMessageSource);
     // Register as "Managed Beans"
    beans.put("MessageDisplay", standardOutMessageDisplay);
    beans.put("MessageSource", helloWorldMessageSource);
```

Modularized "bean" configuration

HelloWorldRedesigned

Here's the "NEW" Look

```
import edu.mum.component.MessageDisplay;
import edu.mum.configuration.ConfigInMemory;

public class HelloWorldReDesigned {
    public static void main(String[] args) {

    // "Configure" application - set up "managed beans"
    ConfigInMemory configInMemory = ConfigInMemory.getInstance();

    // Lookup the MessageDisplay bean
    MessageDisplay messageDisplay =
            (MessageDisplay) configInMemory.getBean("MessageDisplay");
    messageDisplay.display();
    }
```

Nice Improvement – SoC, Modularization, Resource Management

BUT WAIT! – the "Managed Beans" are STILL HARDCODED in Config!!!

AND – the Dependency Injection is STILL Manual!!

Externalize Configuration of Resources

```
public class MessageConfiguration {
private static final MessageConfiguration instance= new MessageConfiguration();

    public static MessageConfiguration getInstance() { return instance; }

    private MessageConfiguration() {
        properties = new Properties();
        String fileName = "HelloWorld.properties";
        InputStream input=
                getClass().getClassLoader().getResourceAsStream(fileName);
        properties.load(input);

        Enumeration  enumeration = properties.keys();
        while (enumeration.hasMoreElements()) {
            String key = (String) enumeration.nextElement();
            bean = ObjectFactory.getInstance((String)properties.get(key));
            beans.put(key, bean);
       // look thru beans for @Autowired annotation – Do dependency Injection!!!
        ProcessAnnotations.handleAnnotations(beans);
```

Get components “declaratively” from properties file

Build map of component instances [beans]

@Autowired Annotation

```
@Documented
@Retention(java.lang.annotation.RetentionPolicy.RUNTIME)
@Target({java.lang.annotation.ElementType.FIELD})
public @interface AutoWired {}
```

Usage in StandardOutMessageDisplay.java

```
public class StandardOutMessageDisplay implements MessageDisplay
  @AutoWired
  private MessageSource messageSource;
    ...
```

We are using @Autowired Annotation to implement Dependency Injection

Add External Configuration [Cont.]

Here's the "NEW" Look

```
import edu.mum.component.MessageDisplay;
import edu.mum.component.MessageSource;
import edu.mum.configuration.MessageConfiguration;

public class HelloWorldReDesignedWithConfiguration {
    public static void main(String[] args) {

      MessageConfiguration messageConfiguration=
                                    MessageConfiguration.getInstance();
     MessageDisplay messageDisplay =
             (MessageDisplay) messageConfiguration.getBean("MessageDisplay");
     messageDisplay.display();
    }
}
```

Lookin' Pretty GOOD –
BUT STILL lots of CUSTOM "glue" code…
Reading configuration file…

Spring Solution

```
package edu.mum;

public class HelloWorld {
 public static void main(String[] args) {
  ApplicationContext context=
   new ClassPathXmlApplicationContext("spring/applicationContext.xml");
```

IOC – Dependency Lookup

```
MessageDisplay messageDisplay= context.getBean("display",MessageDisplay.class);
   messageDisplay.display();
  }
}
```

SEE Demo HelloWorldSpringX

Spring Solution
XML based Configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">
```

Spring manages components

```
   <bean id="source" class="edu.mum.component.impl.HelloWorldMessageSource"/>

   <bean id="display" class="edu.mum.component.impl.StandardOutMessageDisplay"
        p:messageSource-ref="source"/>
</beans>
```

Spring “injects” source dependency [DI]

Spring Configuration Metadata

XML based

- Wire components without touching their source code or recompiling them.
- **CLAIM:** Annotated classes are no longer POJOs ****
- Configuration centralized and easier to control.

Annotation [Version 2.5]

- Component wiring close to the source
- Shorter and more concise configuration.

JavaConfig [Version 3.0]

- Define beans external to your application classes by using Java rather than XML files

Annotation injection is performed *before* XML injection. Therefore XML injection takes precedence over Annotation injection. It is the "last word"

You can Mix and Match 'em

Annotation Based JavaConfig Configuration

SOURCE: JavaConfiguration.java

```
@Configuration
// search the edu.mum.component package for @Component classes
@ComponentScan("edu.mum")
public class JavaConfiguration { }
```

_____SOURCE: StandardOutMessageDisplay .java_____

Spring manages Annotated @Components

```
@Component
public class StandardOutMessageDisplay implements MessageDisplay {

@Autowired
private MessageSource messageSource;
```

Spring Injects @Autowired dependencies

SEE Demo HelloWorldSpringAJ

Annotation Based JavaConfig [Cont.]

```
@Component
public class HelloWorld {

  @Autowired
  MessageDisplay messageDisplay;

    public static void main(String[] args) {
    ApplicationContext applicationContext = new
                AnnotationConfigApplicationContext( JavaConfiguration.class );
    applicationContext.getBean(HelloWorld.class)
                                        .mainInternal(applicationContext);
 }

   private void mainInternal(ApplicationContext applicationContext) {
        messageDisplay.display();
    }
}
```

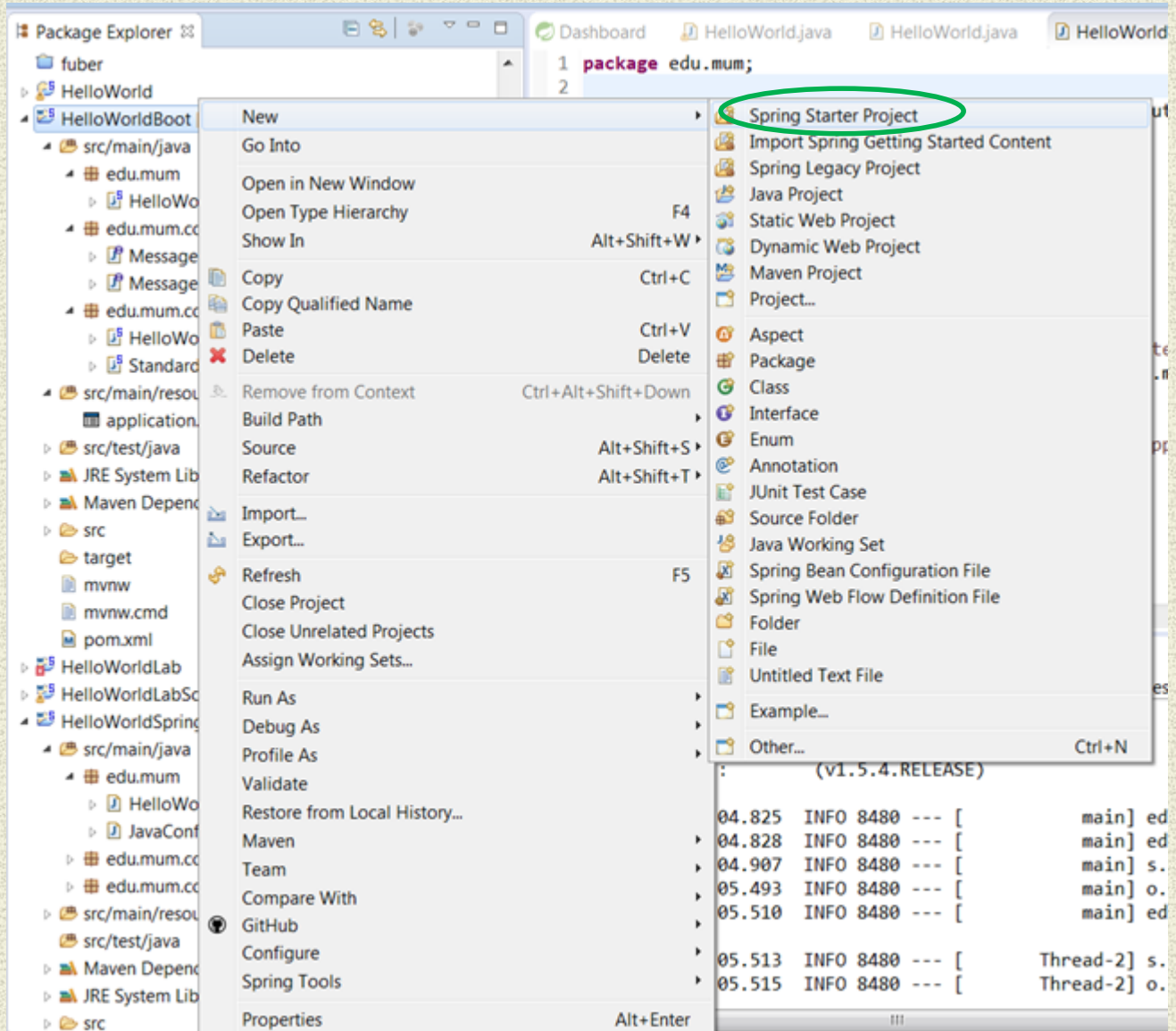
SEE Demo HelloWorldSpringAJ

Annotations with XML Config – How to Scan

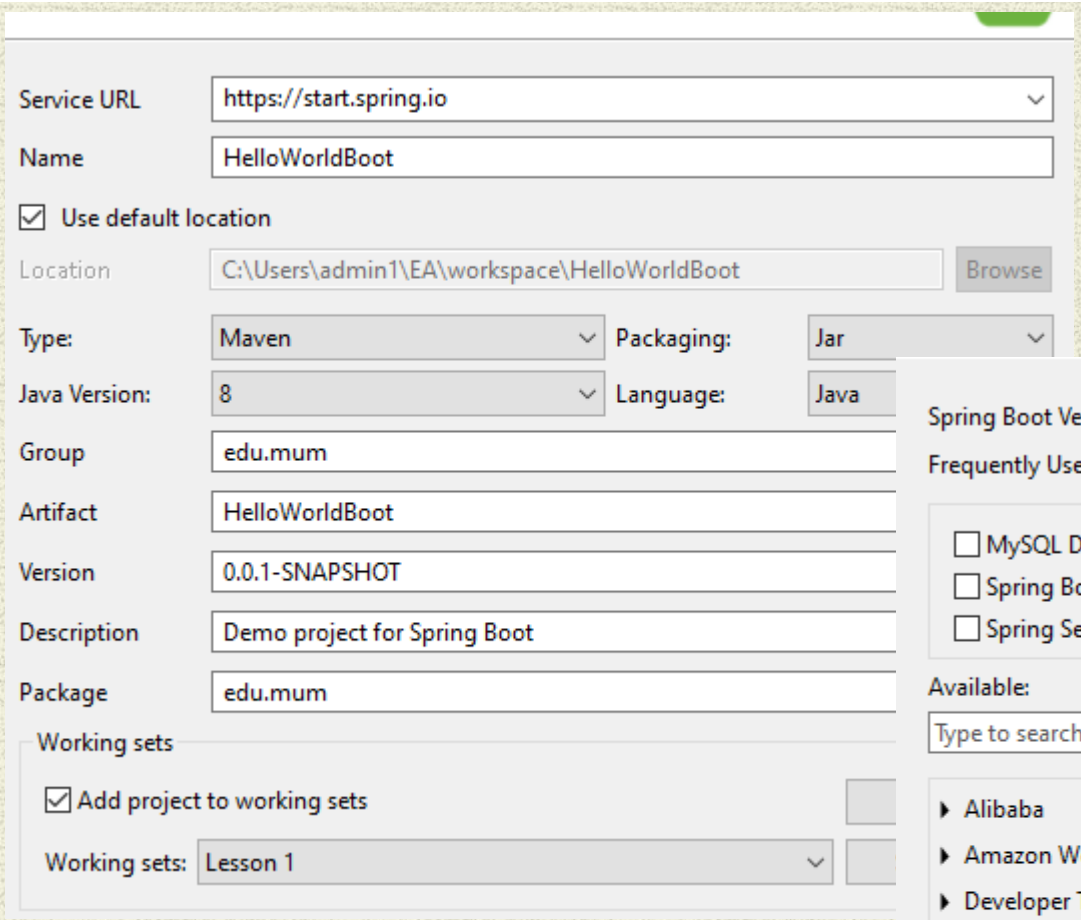
```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context.xsd">

     <context:component-scan base-package= "edu.mum" />

   </beans>
```

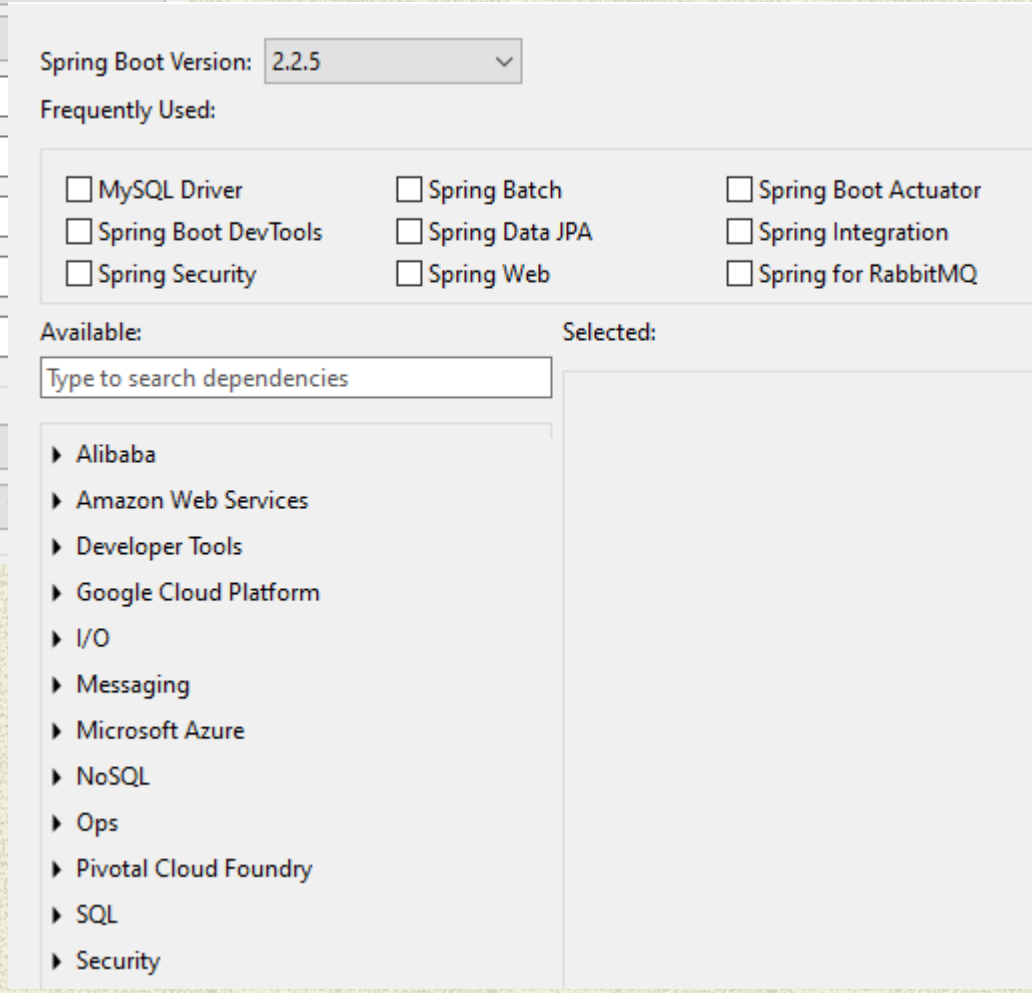
SEE Demo HelloWorldSpringAX

Spring Boot Project



New Spring Boot Project



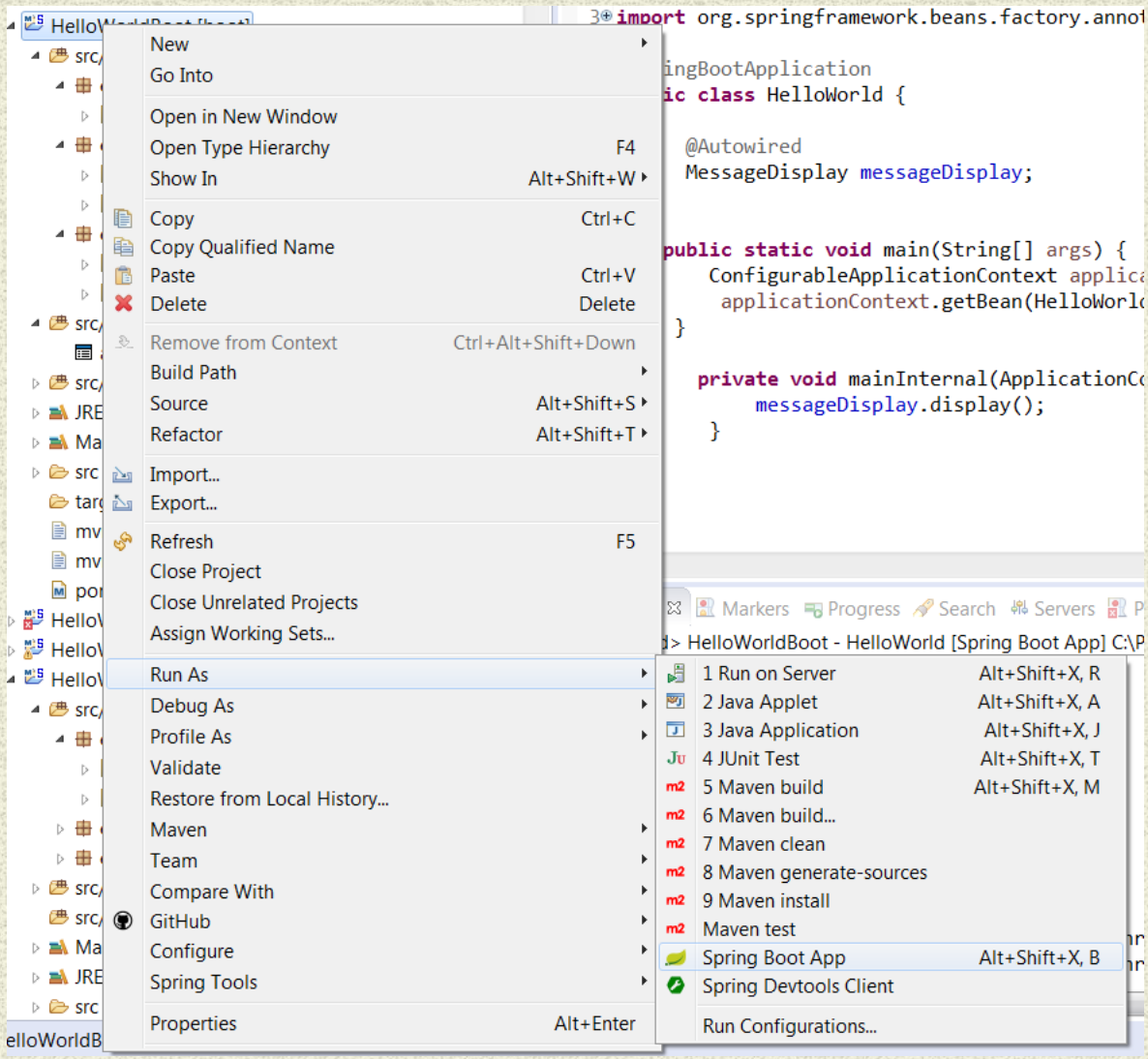


Spring Boot Example

```
@SpringBootApplication
public class HelloWorld {

  @Autowired
  MessageDisplay messageDisplay;


public static void main(String[] args) {
// SpringApplication.run() -- startup/configure framework -- using THIS class !!!
     ConfigurableApplicationContext applicationContext =
                                  SpringApplication.run(HelloWorld.class, args);
   applicationContext.getBean(HelloWorld.class).mainInternal(applicationContext);
 }

   private void mainInternal(ApplicationContext applicationContext) {
        messageDisplay.display();
    }
```

Looks Similar to Java Config

Run As Boot Application



Dependency Injection Annotations

Annotation	Package	Source
@Resource	javax.annotation	JSR 250
@Inject	javax.inject	JSR 330
@Autowired	org.springframework.bean.factory	Spring
@Qualifier **	javax.inject	JSR 330

Name a component example –

```
@Autowired
@Qualifier("production")
private MemberService MemberService;
```

```
@Component("production")
public class MemberServiceImpl implements MemberService
```

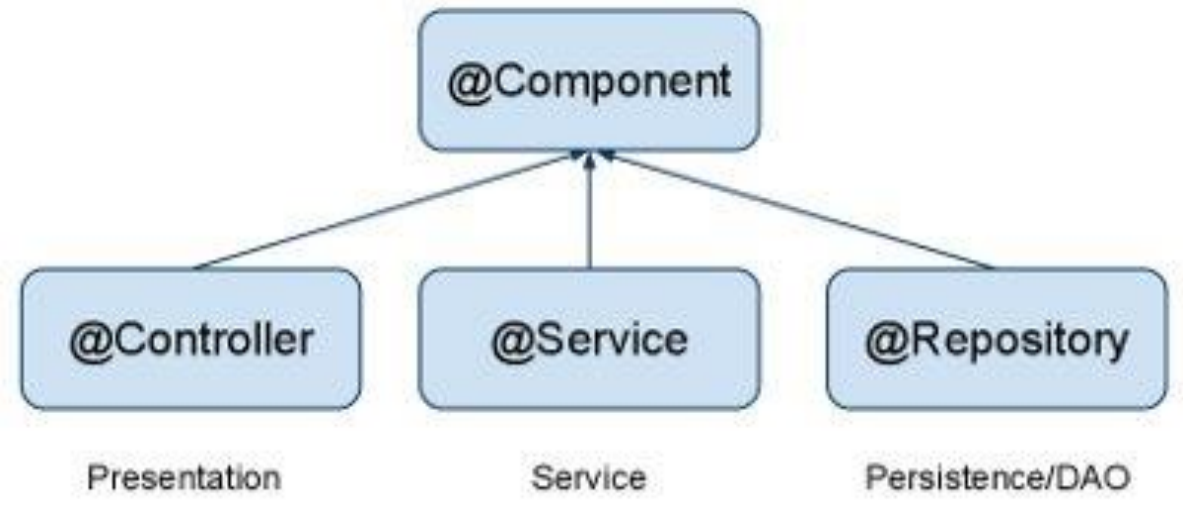
Named Component Comparison Examples

```
@Autowired
@Qualifier(“production")
private MemberService memberService;

@Inject
@Qualifier(“production")
private MemberService memberService;

@Resource(name="production")
private MemberService memberService;
```

Referenced Component:

```
@Component(“production”)
 public class MemberServiceImpl implements MemberService
```

Spring Component Annotations



@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.

Dependency Injection [DI] placement

DI exists in three major variants

Dependencies defined through:

- Property-based dependency injection.
- Setter-based dependency injection.
- Constructor-based dependency injection

Container *injects* dependencies when it creates the bean.

Dependency Injection examples

Property based[byType]:

```
@Autowired
ProductService productService;
```

Setter based[byName]:

```
ProductService productService;
@Autowired
public void setProductService(ProductService productService){
        this.productService = productService;
```

Constructor based:

```
ProductService productService;
@Autowired
public ProductController(ProductService productService) {
        this.productService = productService;
```

When do we use DI?

MAINLY when referencing components **BETWEEN** layers

Also

When an object references another object whose implementation might change

- You want to plug-in another implementation

When an object references a plumbing object

- An object that sends an email
- A DAO object

When an object references a resource

- For example a database connection

Main Point

Dependency Injection allows us to support better separation of concerns and creates more malleable applications with minimal effort.

Science of Consciousness: *The experience of transcending gives us a better understanding of the order and flexibility in life in an effortless way.*