- ❑ Specification

- ❑ Framework

- ❑ Pattern

Specification

Provides API , standards, recommended practices, codes
And technical publications, reports and studies.

JCP - Java Community Process
JSR - Java Specification Request

JSRs directly relate to one or more of the Java platforms.

There are 3 collections of standards that comprise the three Java editions:

Standard, Enterprise and Micro

Java EE (47 JSRs)
The Java Enterprise Edition offers APIs and tools for developing multitier enterprise applications.

Java SE (48 JSRs)
The Java Standard Edition offers APIs and tools for developing desktop and server-side enterprise applications.

Java ME (85 JSRs)
Java ME technology, Java Micro Edition, designed for embedded systems (mobile devices)

JSR 168,286,301 - Portlet Applications

JSR 127,254,314 - JSF

JSR 220 - Ejb3.0 & Jpa   JSR 318 – EJB 3.1

JSR 340: Java Servlet 3.1 Specification

JSR 250 - Common Annotations for java

JSR 303 - Java Bean Validations

JSR 224- Jax-ws

JSR 311,370 - Jax-RS

JSR 299 - Context & DI

JSR 330 – DI

JSR-303 - Bean Validations

JSR208,312 – JBI (Java Business Integration)

A **framework** is a body of pre-written code that acts as a template or skeleton, which a developer can then use to create an application by filling in their own code as needed to get the app to work as they intend it to.

A framework is created to be used over and over so that developers can program their application without the manual overhead of creating every line of code from scratch.

Java frameworks are bodies of prewritten code used by developers to create apps using the Java programming language.

A Java framework is a type of framework specific to the Java programming language, used as a platform for developing software applications and Java programs.

Ex: Spring, Hibernate, Spring Boot, apache cxf/ eclipse jersey etc.,
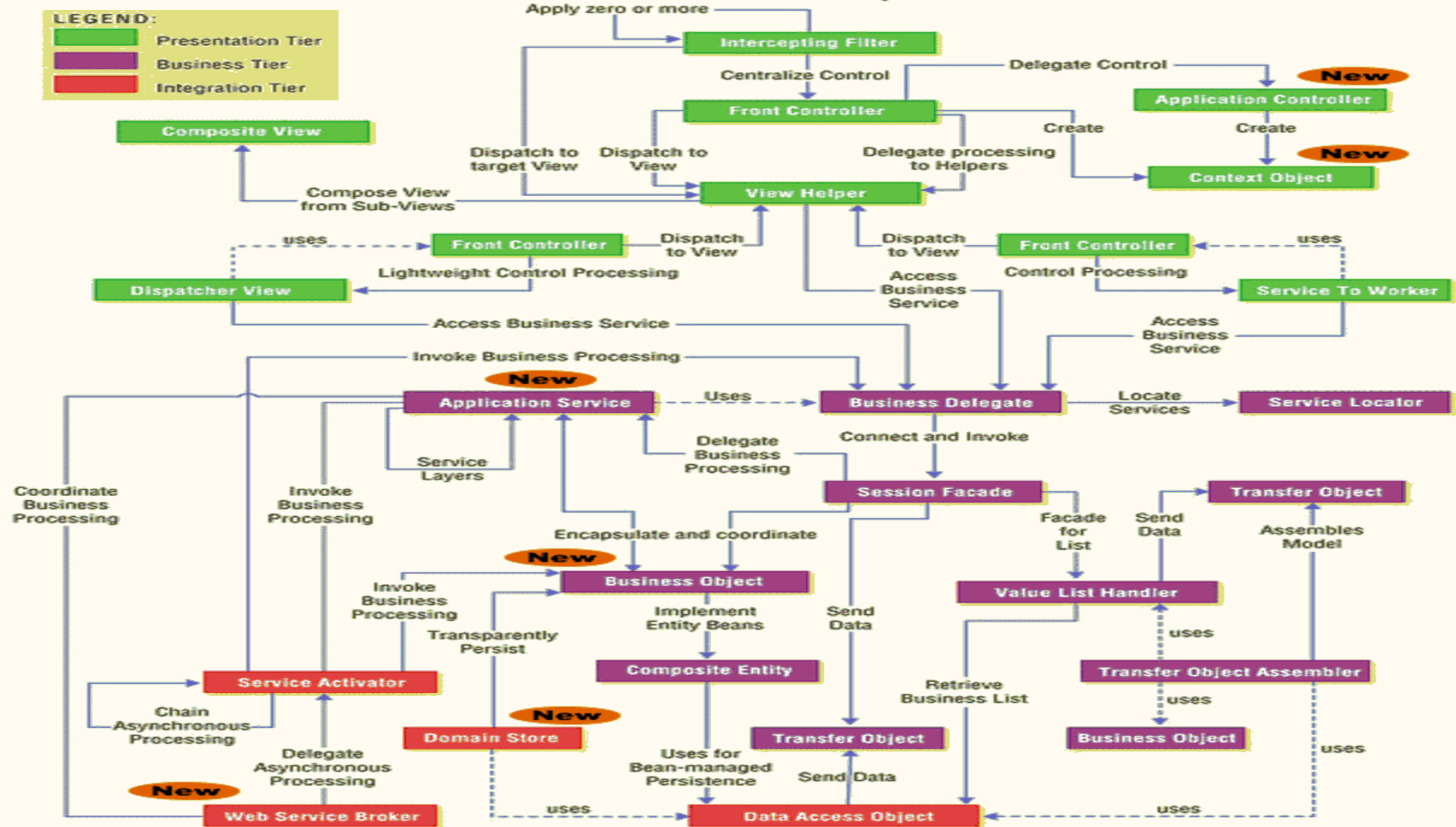
Design Pattern

In software engineering, a software design pattern is a general, reusable solution to a commonly occurring problem within a given context in software design

GOF Design Patterns

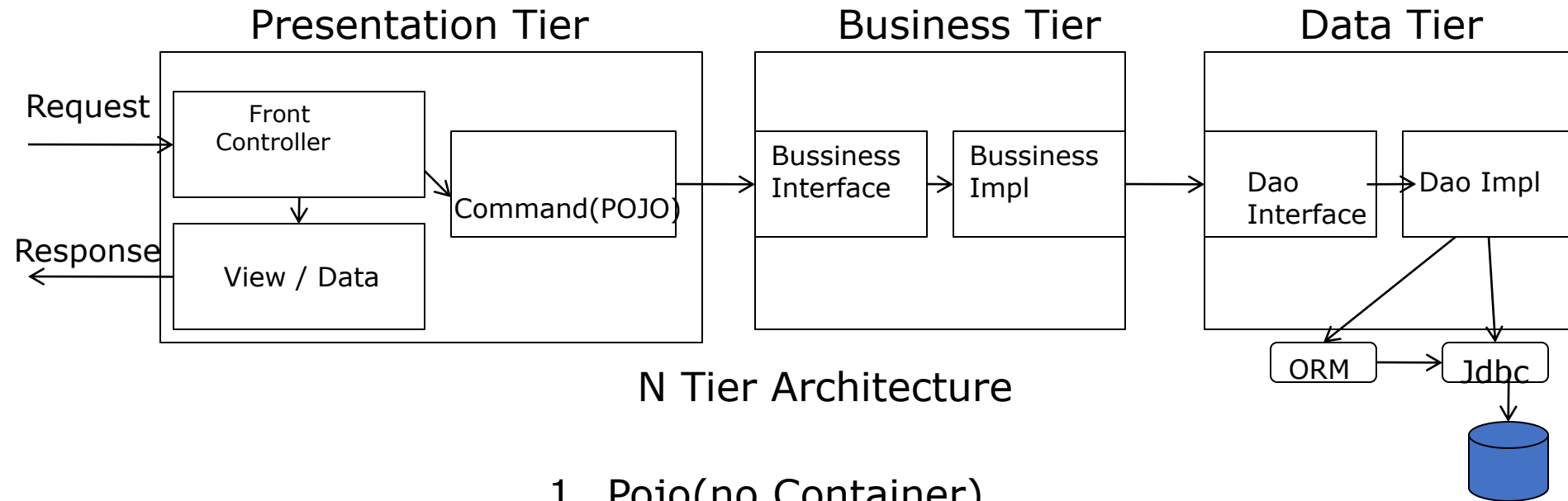|		Purpose		
		Creational	**Structural**	**Behavioral**
S C O P E	**Class**	Factory Method	Class Adapter	Interpreter Template Method
	O B J E C T	Abstract Factory Builder Prototype Singleton	Bridge Composite Decorator Façade Flyweight Object Adapter Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Core J2EE Patterns, 2nd Edition

LEGEND:
- Presentation Tier
- Business Tier
- Integration Tier

Apply zero or more → Intercepting Filter

Centralize Control

Delegate Control

New Application Controller

Create

New Context Object

Create

Front Controller

Composite View

Dispatch to target View

Dispatch to View

Delegate processing to Helpers

Compose View from Sub-Views

View Helper

Front Controller — uses

Dispatch to View

Dispatch to View

Front Controller — uses

Lightweight Control Processing

Control Processing

Dispatcher View

Access Business Service

Service To Worker

Access Business Service

Access Business Service

Invoke Business Processing

New Application Service — Uses → Business Delegate — Locate Services → Service Locator

Service Layers

Delegate Business Processing

Connect and Invoke

Coordinate Business Processing

Invoke Business Processing

Encapsulate and coordinate

Session Facade

Facade for List

Send Data

Transfer Object

Assembles Model

New Business Object

Implement Entity Beans

Value List Handler

uses

Invoke Business Processing

Transparently Persist

Composite Entity

Send Data

Transfer Object Assembler

uses

Service Activator

Chain Asynchronous Processing

New Domain Store

Uses for Bean-managed Persistence

Retrieve Business List

uses

Business Object

Delegate Asynchronous Processing

Transfer Object

Send Data

uses

New Web Service Broker

uses

Data Access Object

uses

RESTful API Patterns

- ❑ Statelessness
- ❑ Content Negotiation
- ❑ URI Templates
- ❑ Pagination
- ❑ Versioning
- ❑ Authorization
- ❑ API facade
- ❑ Discoverability
- ❑ Idempotent
- ❑ Circuit breaker

Microservice Patterns:

- ❑ API gateway
- ❑ Service registry
- ❑ Circuit breaker
- ❑ Messaging
- ❑ Database per Service
- ❑ Access Token
- ❑ Saga
- ❑ Event Sourcing & CQRS

Presentation Tier

Front Controller

Request →

Response ←

View / Data

Command(POJO)

Business Tier

Bussiness Interface	Bussiness Impl

Data Tier

Dao Interface	Dao Impl

ORM → Jdbc

N Tier Architecture

1. Servlet/jsp
2. MVC
   Struts
   JSF
   Flex
   Gwt
   Spring MVC
   …

1. Pojo(no Container)
2. Ejb 2.x(HW Container)
   -Session Bean
   -Mdb
3. Pojo + LW Container
   - Spring
   - Microcontainer
   - Xwork
4. Ejb3.0

1. Jdbc(pojo)
2. Ejb 2.x – Entity Bean
3. Jdo
4. ORM
   - Hibernate
   - Kodo
   - Toplink
   - MyBatis
5. JPA

   + Spring Templates

Web Application:

It is an end-to-end solution for a user.

Which means, User can:

❑ Open it using a browser Interact with it.

❑ User can click on something and after some processing, its result will be reflected in the browser screen. Human-System interaction.

Web API / Web service

With Web APIs alone, a user can not interact with it, because it only returns data, not views.

❑ It is a system which interacts with another system

❑ It does not return views, it returns data

❑ It has an endpoint set, which can be hit by other systems to get data which it provides.

Microservice Architecture
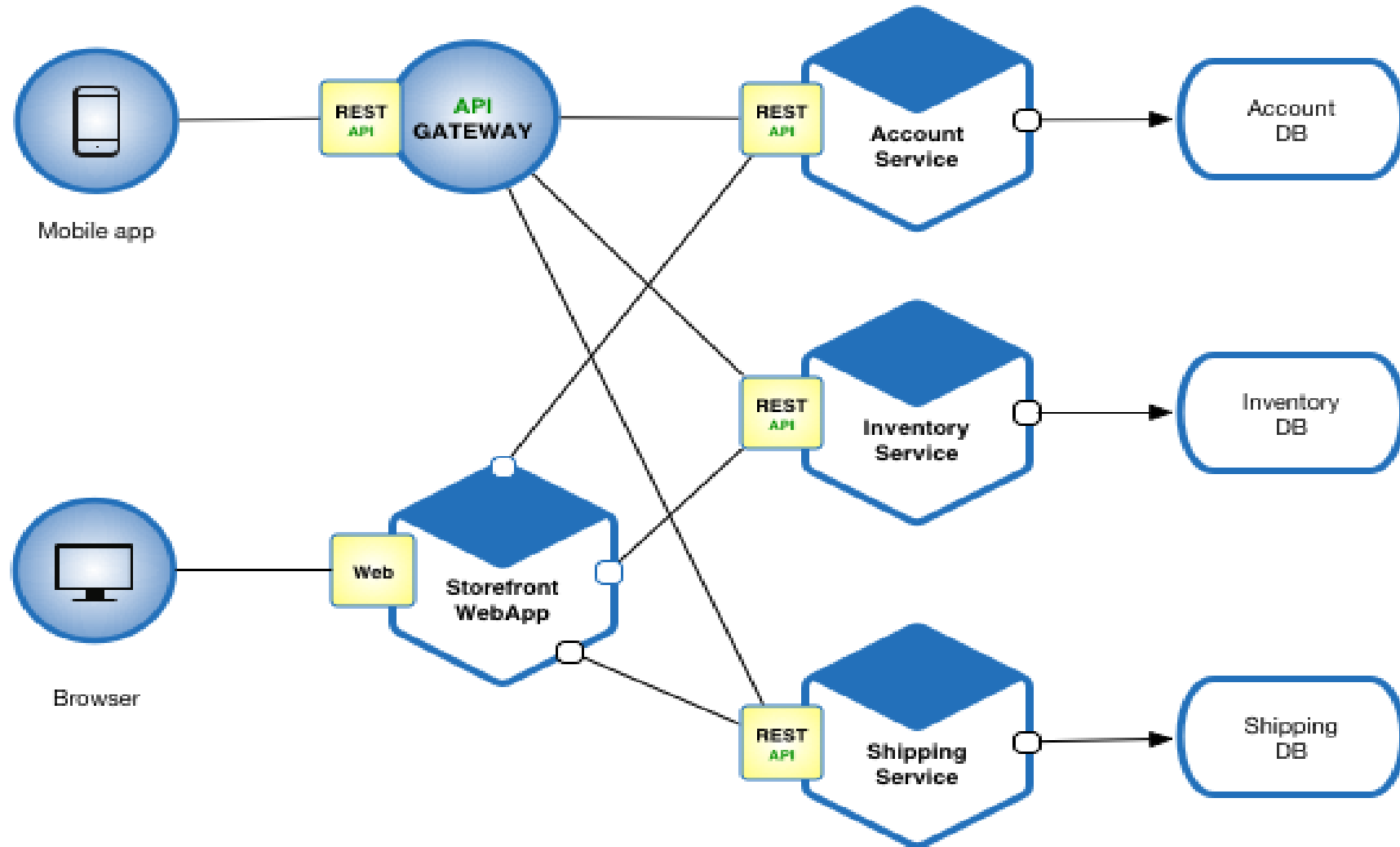
Spring Framework Features:

1. Core Container - IOC, DI & AOP
   Application Context (Object creation – Singleton/Prototype)
   Dependency Injection (Object Graph Creation)

2. Data Access Layer (Template pattern)

3. Spring MVC & REST

4. Other features like Security, Messaging, Spring Integration,
    Spring Batch, Spring Cloud etc.,

IOC is used to decouple common task from implementation.

Six basic techniques to implement Inversion of Control.

These are:

1.using a factory pattern
2.using a service locator pattern
3.using a constructor injection
4.using a setter injection
5.using an interface injection
6.using a contextualized lookup

Constructor, setter, and interface injection are all aspects of Dependency injection.

What is a container?

✓ The container will create the objects,
✓ wire them together,
✓ configure them,
✓ and manage their complete life cycle from creation till destruction.

Java POJO classes

Metadata → The Spring container

Final Result

Ready to use application
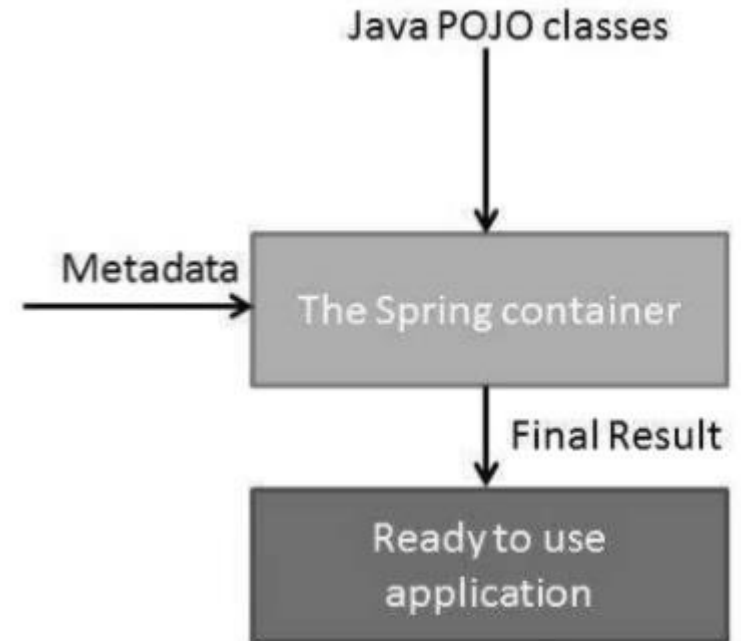
The container gets its instructions on

Java POJO classes

✓ what objects to instantiate,
✓ configure,
✓ and assemble by reading the configuration metadata provided.
✓ The configuration metadata can be represented either by XML or Annotation.

Metadata → The Spring container

Final Result

Ready to use application

❑ Apache Tomcat is a Servlet Container.
❑ Weblogic/Websphere/JBoss provides EJB Container
❑ Spring is a POJO container.
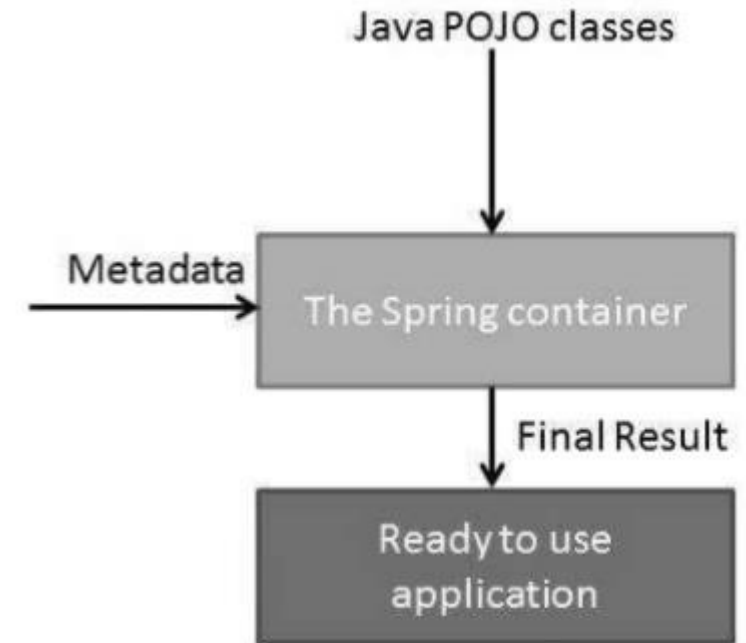
Spring XML Configuration

```
package demo.web;

Class AccountDaoJdbc implements AccountDao
{

DataSource  dataSource;

Public voic setDatasource(DataSource datasource)
{
dataSource = datasource;
}
..
}
```

```xml
<bean id = "accountDao"   class="demo.web.AccountDaoJdbc>
<property name="datasource">
<ref bean="dataSource"/>
</property>
</bean>
```

Spring Annotation Configuration

```
package demo.web;

@Repository("accountDao")
Class AccountDaoJdbc implements AccountDao
{

@Autowired
DataSource  dataSource;

Public voic setDatasource(DataSource datasource)
{
dataSource = datasource;
}
..
}
```
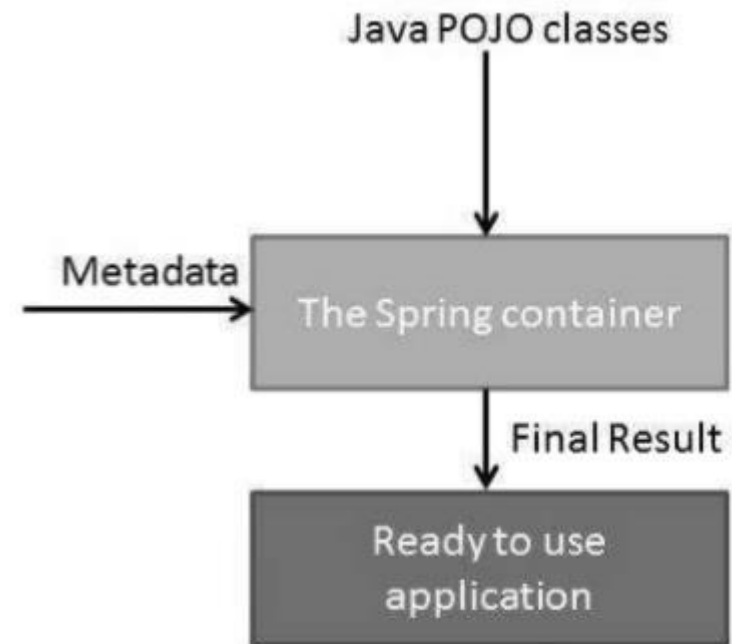
```
<bean id = "accountDao"   class="demo.web.AccountDaoJdbc>
<property name="datasource">
<ref bean="dataSource"/>
</property>
</bean>
```

Lifecycle Management

Bean Managed Life cycle

Container Managed Life cycle

AccountBean acc = new AccountBean();
acc.setAmt(5000);
….

acc=null;

Java POJO classes

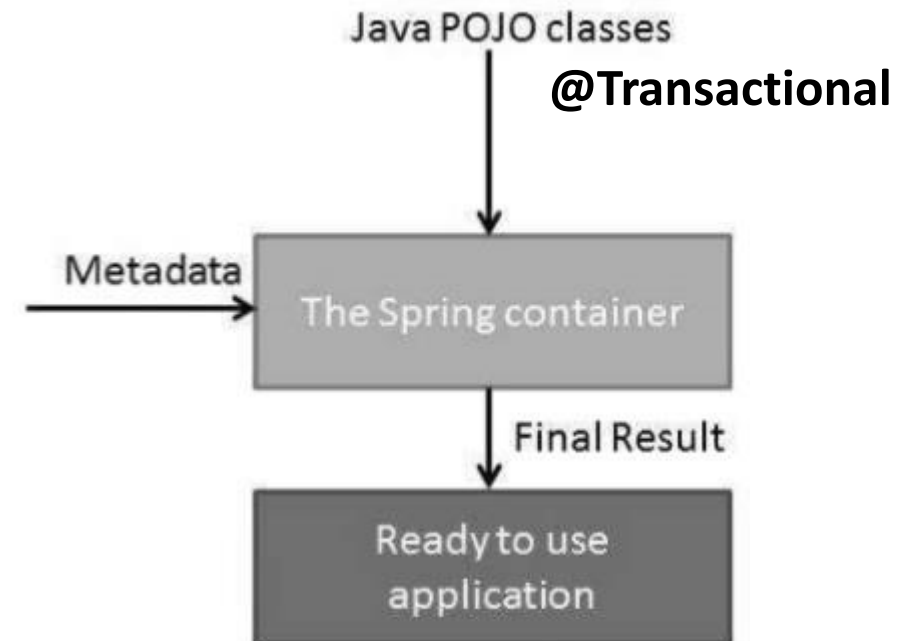Metadata

The Spring container

Final Result

Ready to use
application

ApplicationContext ctx = …..
AccountBean acc = ctx.getBean("account");

Container Managed Services

Bean managed transactions
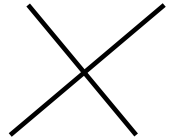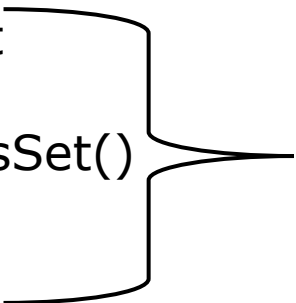
Container managed transactions

Transaction tx= ….

tx.begin;

database operations;

tx.commit() / tx.rollback();

Java POJO classes

@Transactional

Metadata → The Spring container

Final Result

Ready to use application

Life cycle of a bean within Spring application context

1. Instantiate

2. Populate properties(DI)

3. Aware interfaces [BeanNameAware's setBeanName()
   ApplicationContextAware's setApplicationContext ()

4. **Pre-initialization Bean Post processors**

5. Any method with @PostConstruct

6. InitializingBean's afterPropertiesSet() —— (use @PostConstruct)

7. Call custom init-method

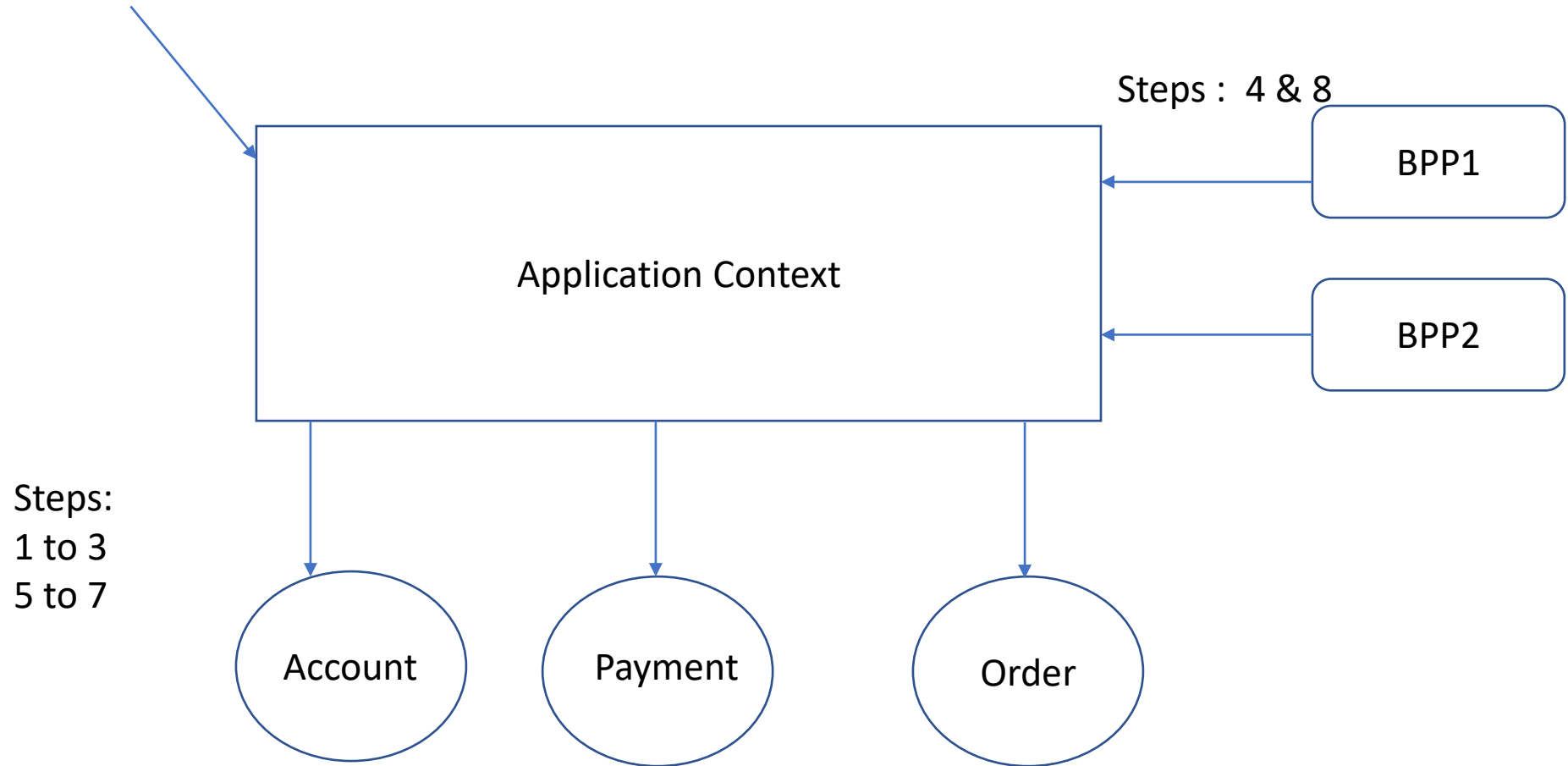8. **Post-initialization BeanPostProcessors**

9. Now Bean is ready to use

Container is shutdown

1. Any method with @PreDestroy
2. DisposableBean's destroy
3. Call custom destroy-method
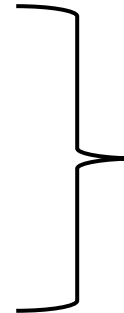
Use @PreDestroy

ApplicationContext context = new ……
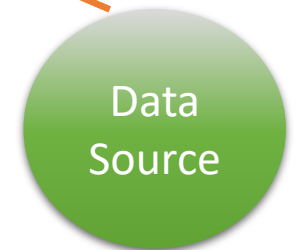Account account = context.getBean("account")

Steps : 4 & 8

BPP1

Application Context

BPP2

Steps:
1 to 3
5 to 7

Account

Payment

Order

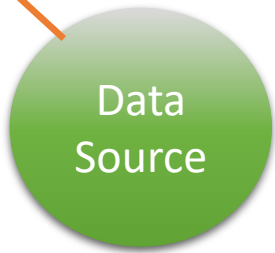# Presentation Tier

Business Tier

Data Tier

@RestController
Class AccountController
{
@Autowired
AccountService accService;
.....
}

@Service
Class AccountServiceBean
        implements AccountService
{
@Autowired
AccountDao accountDao;
.....

}

@Repository
Class AccountDaoJdbc
        implements AccountDao
{
@Autowired
DataSource dataSource;
.....
}
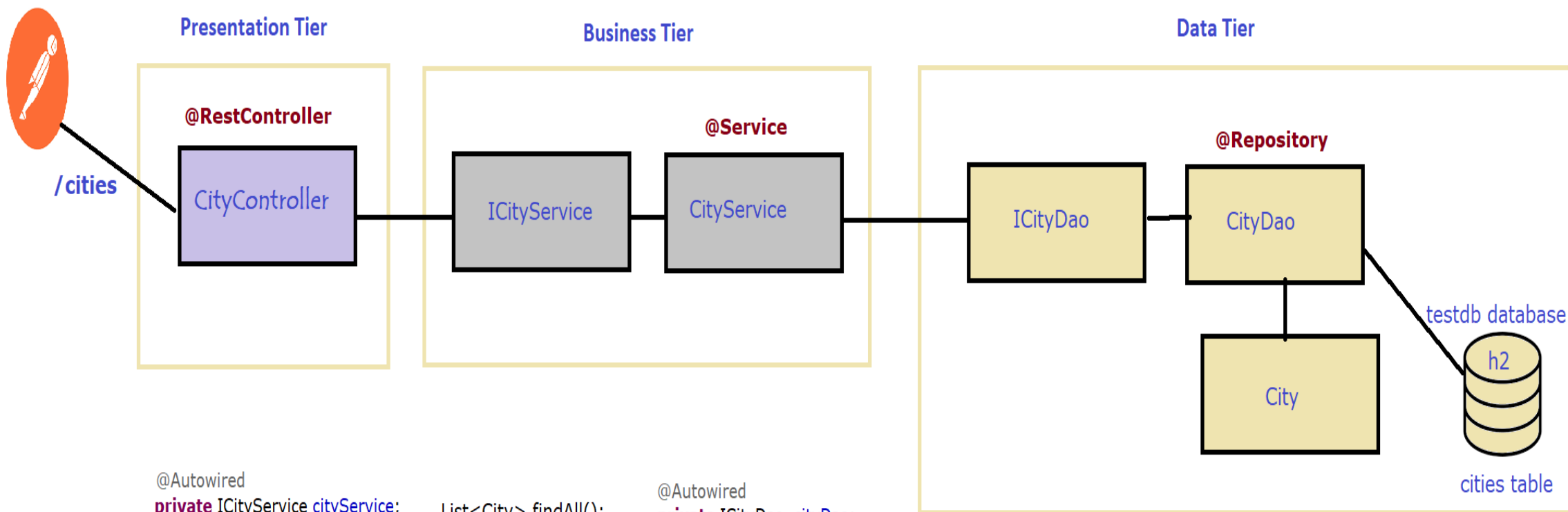
Spring AOP

Spring AOP

Spring AOP enables Aspect-Oriented Programming in spring applications.

In AOP, aspects enable the modularization of concerns such as :

- ❑ transaction management,
- ❑ Logging,
- ❑ Exception Handling
- ❑ Security etc.,

(often termed crosscutting concerns)

Cross Cutting Concerns

Account Bean with business logic

Client Application ⟷ AccountBean implements IAccount

IAccount acc = applicationContext.getBean("account"));
acc.deposit();

deposit()

withdraw()

balance()

Business Logic

Account Bean with business & Cross-Cutting Concerns (programmatic)

Client Application ⟷ @Service
AccountBean
implements
IAccount

IAccount acc = applicationContext.getBean("account"));
acc.deposit();

deposit()

withdraw()

balance()

Business Logic

+

Cross-cutting Concerns (Security, Logging, Transaction, Exception handling Etc.,)
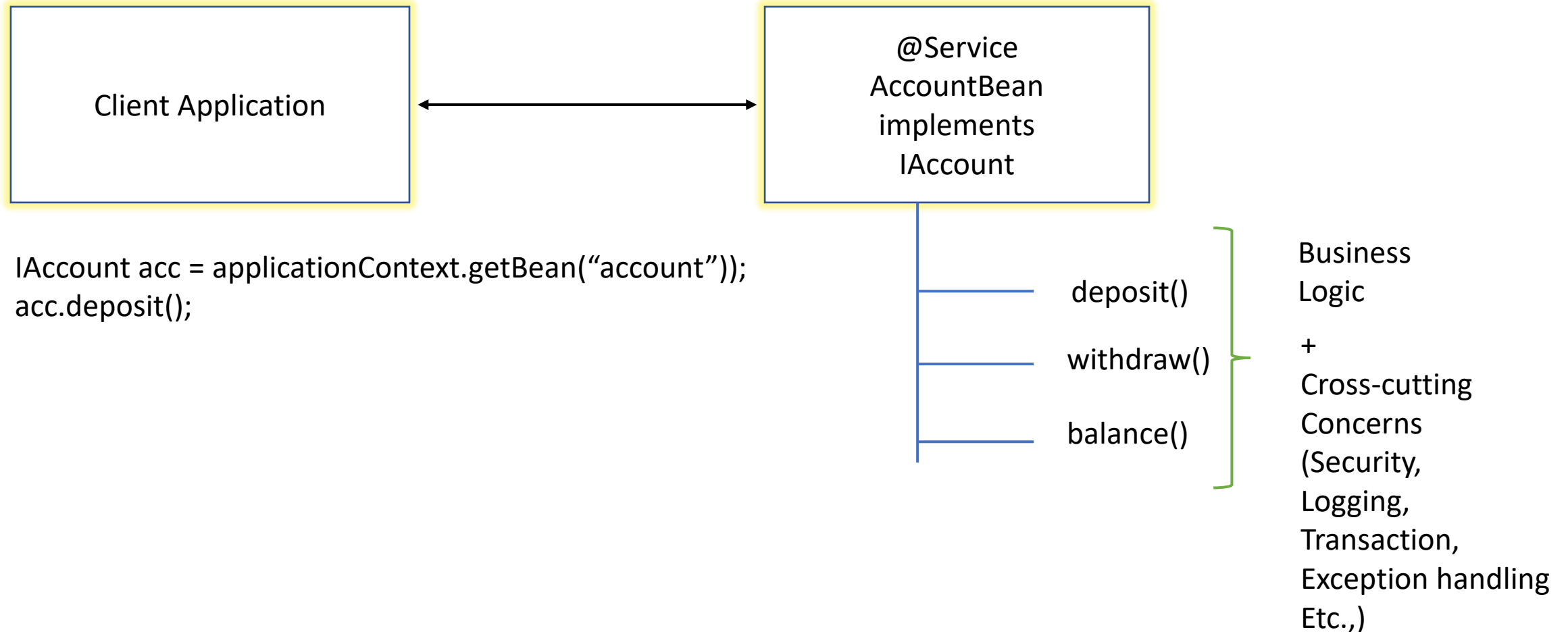
Account Bean with business & Cross-Cutting Concerns (Declarative – using proxy)
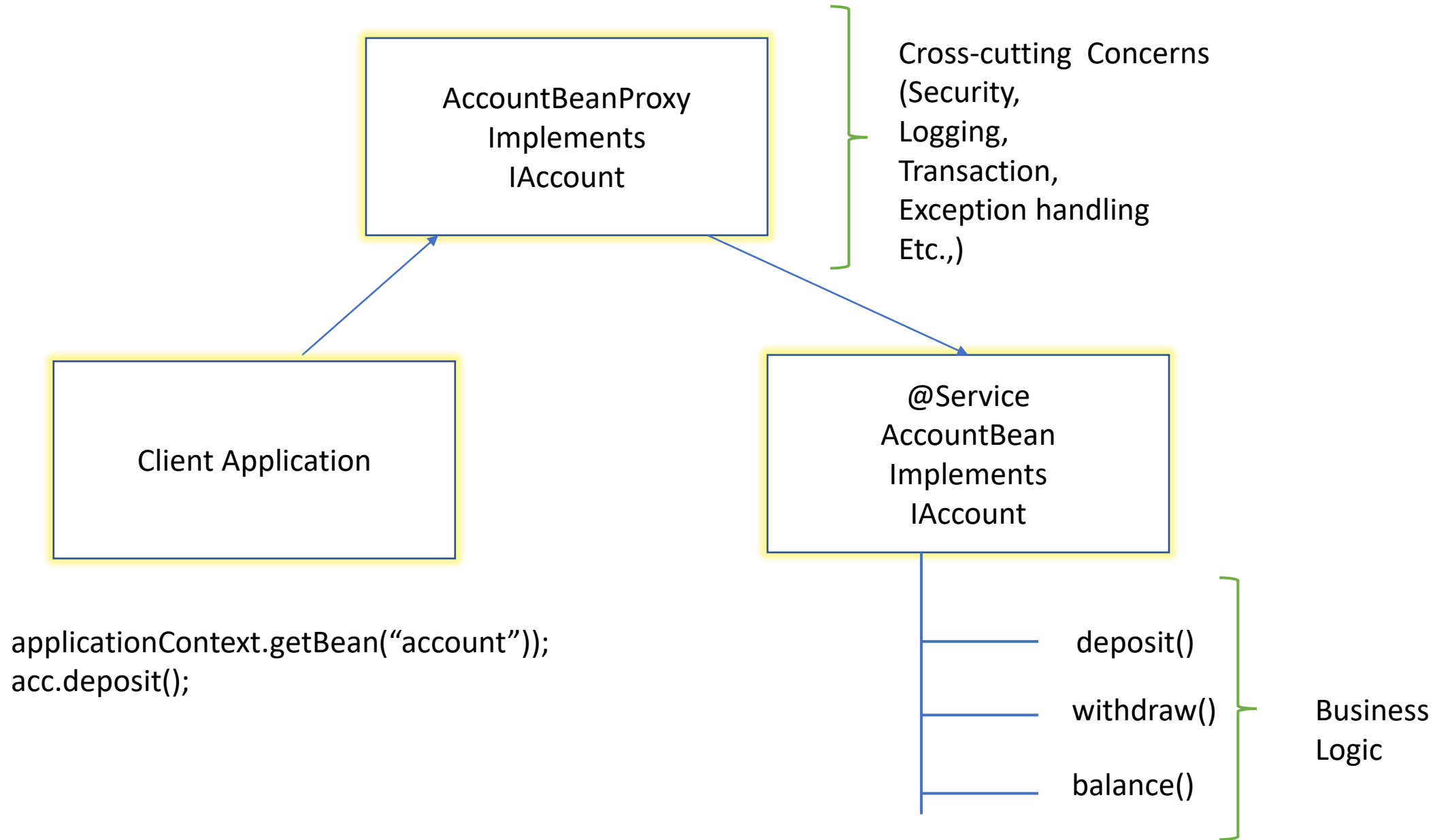
AOP Terminology

- Aspect – a modularization of a Crosscutting concern.

- Target object - object being advised by one or more aspects.

- AOP proxy – Will manage the way of Applying Aspects at particular Pointcuts.
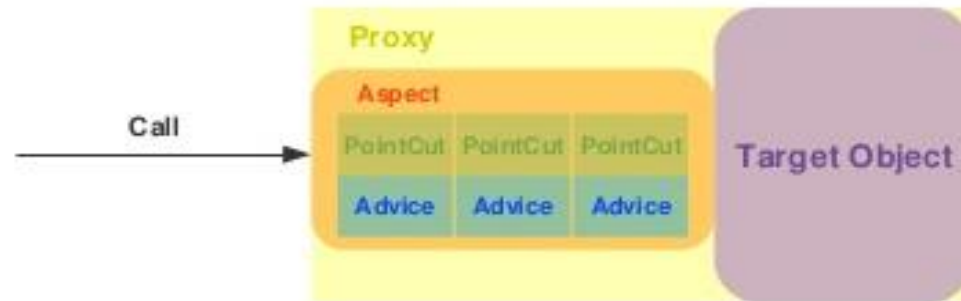
AOP Terminology

- Joinpoint – Defines A Point During the Execution of a program. We can insert Additional logics at Joinpoints
- Advice – Action taken(Functionality) at a Joinpoint.
- Pointcut – Combination of Joinpoints where the Advice need to be Applied.
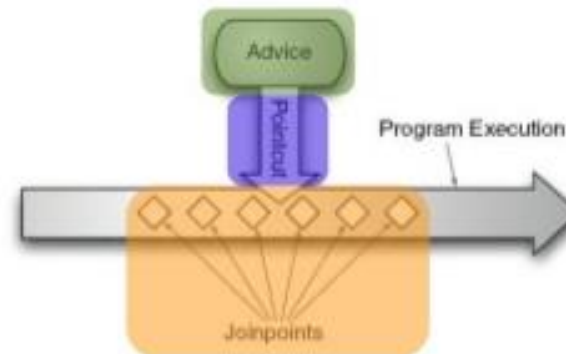
Advice Types

- **Before Advice**
- **After returning Advice**
- **Around Advice**
- **Throws Advice**
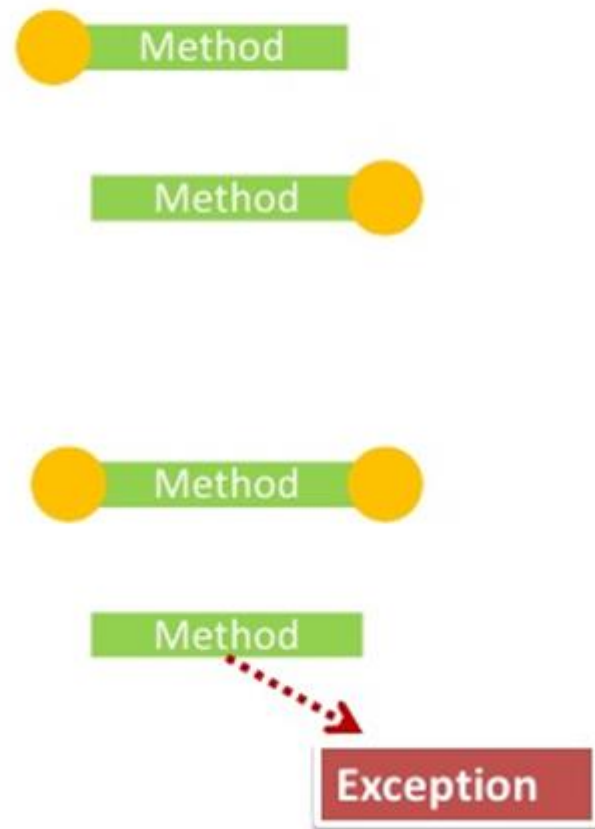
Spring Boot Introduction

Spring Boot is a project that is built on the top of the **Spring** Framework. It provides an easier and faster way to set up, configure, and run both simple and web-based applications.

Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that we can "just run".

Spring Framework + Embedded HTTP Servers (Tomcat, Jetty) − XML \<bean> Configuration or @Configuration = Spring Boot
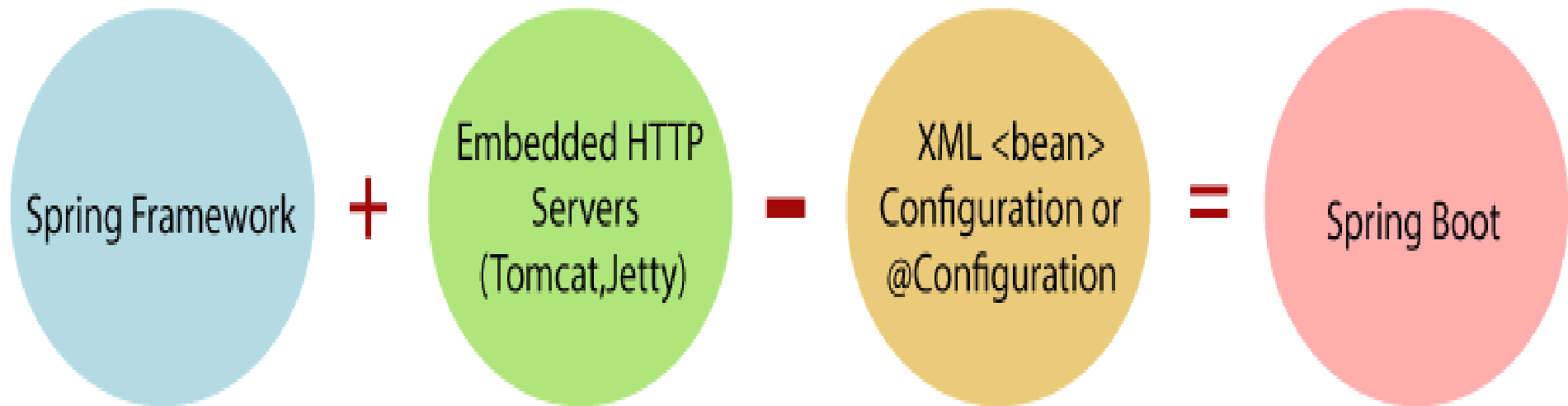
Twelve-Factor Applications

1. CODEBASE
One codebase tracked
in SCM, many deploy

2. DEPENDENCIES
Explicitly declare
isolate dependencies

3. CONFIGURATION
Store config in
the environment

4. BACKING SERVICES
Treat backing services as
attached resources

5. BUILD, RELEASE, RUN
Strictly separate build
and run stages

6. PROCESSES
Execute app as
stateless processes

7. PORT BINDING
Export services via
port binding

8. CONCURRENCY
Scale out via the
process model

9. DISPOSABILITY
Maximize robustness &
graceful shutdown

10. DEV/ PROD PARITY
Keep dev, staging, prod as
similar as possible
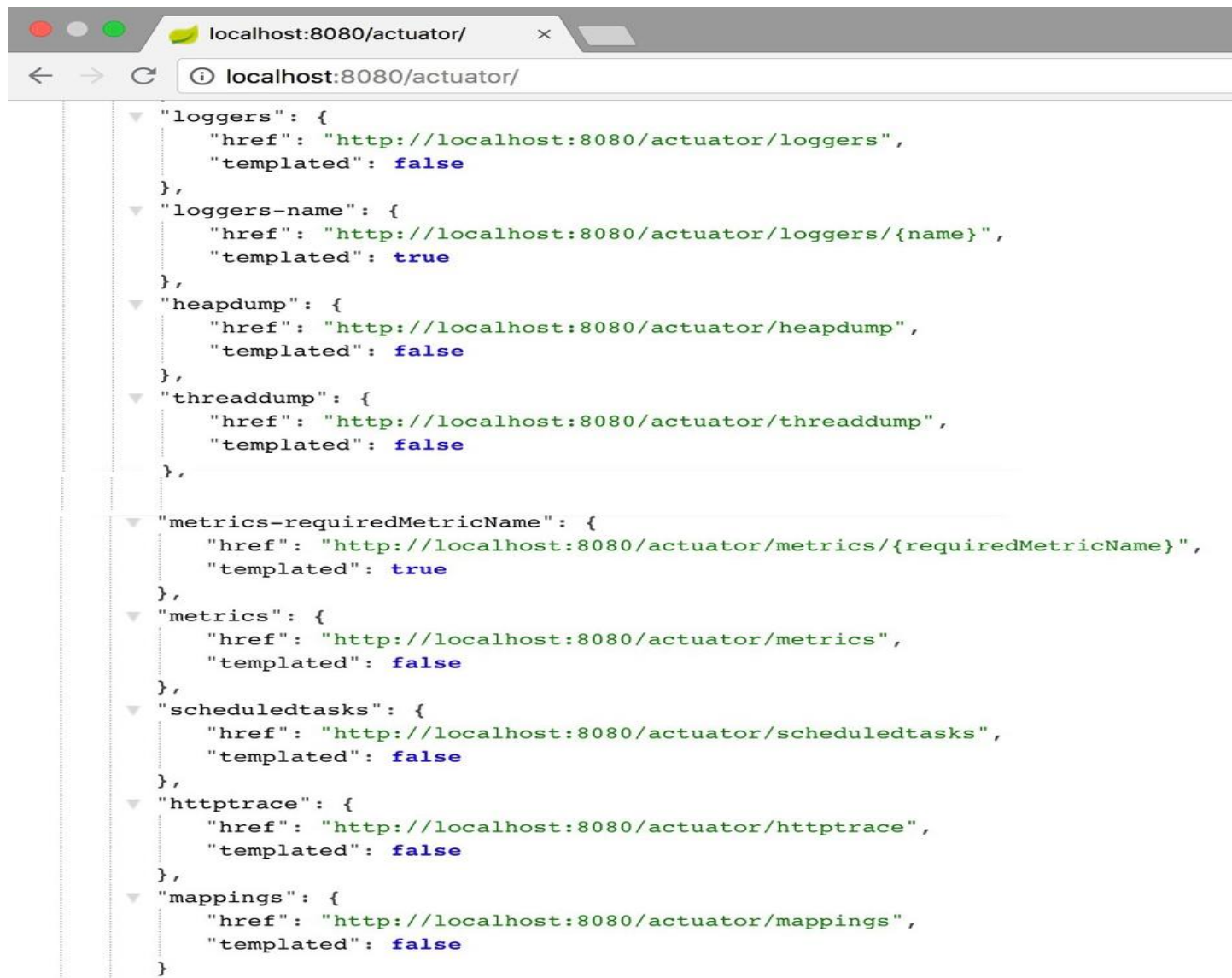
11. LOGS
Treat logs as
event stream

12. ADMIN PROCESSES
Run admin / mgmt tasks
as one-off processes

Spring Boot is an

- ➢ Opinionated,
- ➢ Convention over configuration,
- ➢ Get started with minimum effort,
- ➢ Create stand-alone,
- ➢ Production-grade applications.

```json
    "loggers": {
        "href": "http://localhost:8080/actuator/loggers",
        "templated": false
    },
    "loggers-name": {
        "href": "http://localhost:8080/actuator/loggers/{name}",
        "templated": true
    },
    "heapdump": {
        "href": "http://localhost:8080/actuator/heapdump",
        "templated": false
    },
    "threaddump": {
        "href": "http://localhost:8080/actuator/threaddump",
        "templated": false
    },


    "metrics-requiredMetricName": {
        "href": "http://localhost:8080/actuator/metrics/{requiredMetricName}",
        "templated": true
    },
    "metrics": {
        "href": "http://localhost:8080/actuator/metrics",
        "templated": false
    },
    "scheduledtasks": {
        "href": "http://localhost:8080/actuator/scheduledtasks",
        "templated": false
    },
    "httptrace": {
        "href": "http://localhost:8080/actuator/httptrace",
        "templated": false
    },
    "mappings": {
        "href": "http://localhost:8080/actuator/mappings",
        "templated": false
    }
```

We can use Spring Boot to create Java applications that can be started using java -jar or more traditional war deployments. We also provide a command line tool that runs "spring scripts".

Spring Boot primary goals are:

❑ Provide a radically faster and widely accessible getting started experience for all Spring development.

❑ Be **opinionated** out of the box, but get out of the way quickly as requirements start to diverge from the defaults.

❑ Provide a range of non-functional features that are common to large classes of projects (e.g. embedded servers, security, metrics, health checks, externalized configuration).

❑ Absolutely no code generation and no requirement for XML configuration.

Spring Boot follows "Opinionated Defaults Configuration" Approach to reduce Developer effort. Due to opinionated view of spring boot, what is required to get started but also we can get out if not suitable for application.

- Spring Boot uses sensible defaults, "opinions", mostly based on the classpath contents.

- For example

– Sets up a JPA Entity Manager Factory if a JPA implementation is on the classpath.

– Creates a default Spring MVC setup, if Spring MVC is on the classpath.

- Everything can be overridden easily

– But most of the time not needed

Spring Boot Auto-configuration

• Auto-configuration works by analysing the class path

– If we forget a dependency, Spring Boot can't configure it
– A dependency management tool is recommended
– Spring Boot parent and starters make it much easier

• Spring Boot works with Maven, Gradle, Ant

Adding classpath dependencies

Spring Boot provides a number of "Starters" that make easy to add jars to your classpath.

The **spring-boot-starter-parent** is a special starter that provides useful Maven defaults. It also provides a dependency-management section so that we can omit version tags.

Other "Starters" simply provide dependencies that we are likely to need when developing a specific type of application. Since we are developing a web application, we will add a **spring-boot-starter-web** dependency — but before that, let's look at what we currently have.

The **mvn dependency:tree** command prints a tree representation of your project dependencies.

Writing the code

```java
import org.springframework.boot.*;
import org.springframework.boot.autoconfigure.*;
import org.springframework.stereotype.*;
import org.springframework.web.bind.annotation.*;

@RestController
@EnableAutoConfiguration
public class Example {

    @RequestMapping("/")
    String home() {
        return "Hello World!";
    }

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

}
```

The @RestController and @RequestMapping annotations are Spring MVC annotations

@EnableAutoConfiguration is used to configure Spring, based on the jar dependencies that we have added. Since spring-boot-starter-web added Tomcat and Spring MVC, the auto-configuration will assume that we are developing a web application and setup Spring accordingly.

@EnableAutoConfiguration(exclude={DataSourceAutoConfiguration.class})

When using SpringBootApplication, the auto-configuration of the context is automatically enabled and adding this annotation has therefore no additional effect.

@SpringBootApplication

Indicates a configuration class that declares one or more @Bean methods and also triggers auto-configuration and component scanning.

This is a convenience annotation that is equivalent to declaring @Configuration, @EnableAutoConfiguration and @ComponentScan.

```
@Configuration
@EnableAutoConfiguration
@ComponentScan
public class Example {

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

}
```

(OR)

```
@SpringBootApplication
public class Example {

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

}
```

@ComponentScan

it will scan for stereotype annotations like

@Controller/@RestController – Presentation Tier
@Service – Business Tier
@Repository – Data Tier
@Component - Generic (AOP)

@EnableAutoConfiguration [spring boot]

web - @EnableWebMVC
security - @EnableWebSecurity
jms - @EnableJMS

@Configuration

It will scan for beans annotated with @Bean (these are configuration
beans like RestTemplate, JdbcTemplate, Docket etc.,)

Understanding auto-configured beans

Under the hood, auto-configuration is implemented with standard @Configuration classes.

Additional @Conditional annotations are used to constrain when the auto-configuration should apply. Usually auto-configuration classes use @ConditionalOnClass and @ConditionalOnMissingBean annotations.

This ensures that auto-configuration only applies when relevant classes are found and when we have not declared your own @Configuration.

We can browse the source code of spring-boot-autoconfigure to see the @Configuration classes that we provide (see the META-INF/spring.factories file).

Package Explorer ⌗

- ⊟ 🫙 spring-boot-1.5.7.RELEASE.jar - D:\MST\software\m2\re|
- ⌄ 🫙 spring-boot-autoconfigure-1.5.7.RELEASE.jar - D:\MST\s
  - › ⊞ org.springframework.boot.autoconfigure
  - › ⊞ org.springframework.boot.autoconfigure.admin
  - › ⊞ org.springframework.boot.autoconfigure.amqp
  - › ⊞ org.springframework.boot.autoconfigure.aop
  - › ⊞ org.springframework.boot.autoconfigure.batch
  - › ⊞ org.springframework.boot.autoconfigure.cache
  - › ⊞ org.springframework.boot.autoconfigure.cassandra
  - › ⊞ org.springframework.boot.autoconfigure.cloud
  - › ⊞ org.springframework.boot.autoconfigure.condition
  - › ⊞ org.springframework.boot.autoconfigure.context
  - › ⊞ org.springframework.boot.autoconfigure.couchbase
  - › ⊞ org.springframework.boot.autoconfigure.dao
  - › ⊞ org.springframework.boot.autoconfigure.data
  - › ⊞ org.springframework.boot.autoconfigure.data.cassand
  - › ⊞ org.springframework.boot.autoconfigure.data.couchb
  - › ⊞ org.springframework.boot.autoconfigure.data.elastics
  - › ⊞ org.springframework.boot.autoconfigure.data.jpa
  - › ⊞ org.springframework.boot.autoconfigure.data.ldap
  - › ⊞ org.springframework.boot.autoconfigure.data.mongo
  - › ⊞ org.springframework.boot.autoconfigure.websocket
  - ⌄ 🗁 META-INF
    - › 🗁 maven
    - {} additional-spring-configuration-metadata.json
    - 📄 MANIFEST.MF
    - 📄 spring-autoconfigure-metadata.properties
    - {} spring-configuration-metadata.json
    - 📄 spring.factories

spring.factories ⌗

```
# Initializers
org.springframework.context.ApplicationContextInitializer=\
org.springframework.boot.autoconfigure.SharedMetadataReaderFactoryContextInitializer,\
org.springframework.boot.autoconfigure.logging.AutoConfigurationReportLoggingInitializer

# Application Listeners
org.springframework.context.ApplicationListener=\
org.springframework.boot.autoconfigure.BackgroundPreinitializer

# Auto Configuration Import Listeners
org.springframework.boot.autoconfigure.AutoConfigurationImportListener=\
org.springframework.boot.autoconfigure.condition.ConditionEvaluationReportAutoConfigurationImportLis

# Auto Configuration Import Filters
org.springframework.boot.autoconfigure.AutoConfigurationImportFilter=\
org.springframework.boot.autoconfigure.condition.OnClassCondition

# Auto Configure
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
org.springframework.boot.autoconfigure.admin.SpringApplicationAdminJmxAutoConfiguration,\
org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,\
org.springframework.boot.autoconfigure.amqp.RabbitAutoConfiguration,\
org.springframework.boot.autoconfigure.batch.BatchAutoConfiguration,\
org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration,\
org.springframework.boot.autoconfigure.cassandra.CassandraAutoConfiguration,\
org.springframework.boot.autoconfigure.cloud.CloudAutoConfiguration,\
org.springframework.boot.autoconfigure.context.ConfigurationPropertiesAutoConfiguration,\
org.springframework.boot.autoconfigure.context.MessageSourceAutoConfiguration,\
org.springframework.boot.autoconfigure.context.PropertyPlaceholderAutoConfiguration,\
org.springframework.boot.autoconfigure.couchbase.CouchbaseAutoConfiguration,\
org.springframework.boot.autoconfigure.dao.PersistenceExceptionTranslationAutoConfiguration,\
org.springframework.boot.autoconfigure.data.cassandra.CassandraDataAutoConfiguration,\
org.springframework.boot.autoconfigure.data.cassandra.CassandraRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.couchbase.CouchbaseDataAutoConfiguration,\
```

The "main" method

The final part of the application is the main method. This is just a standard method that follows the Java convention for an application entry point.

The main method delegates to Spring Boot's **SpringApplication** class by calling **run**. Spring Application will bootstrap our application, starting Spring which will in turn start the auto-configured Tomcat web server.

We need to pass Example.class as an argument to the run method to tell SpringApplication which is the primary Spring component. The args array is also passed through to expose any command-line arguments.

Type **mvn spring-boot:run** from the root project directory to start the application.

Spring Aware interfaces are similar to servlet listeners with callback methods and implementing observer design pattern.

Some of the important Aware interfaces are:

ApplicationContextAware – to inject ApplicationContext object, example usage is to get the array of bean definition names.

BeanNameAware – to know the bean name defined in the configuration file.

ResourceLoaderAware – to inject ResourceLoader object, example usage is to get the input stream for a file in the classpath.

ServletContextAware – to inject ServletContext object in MVC application, example usage is to read context parameters and attributes.

ServletConfigAware – to inject ServletConfig object in MVC application, example usage is to get servlet config parameters.

Spring Initializer

Spring Initializer

It is a web tool which is provided by Spring on official site. We can create Spring Boot project by providing project details. Select Maven project and dependencies , Fill other details and click on generate project.

SPRING INITIALIZR bootstrap your application now

Generate a [Maven Project ∨] with [Java ∨] and Spring Boot [2.0.0 (SNAPSHOT) ∨]

Project Metadata

Artifact coordinates

Group

com.example

Artifact

spring-boot-demo

Dependencies

Add Spring Boot Starters and dependencies to your application

Search for dependencies

Web, Security, JPA, Actuator, Devtools...

Selected Dependencies

Web ✕

Generate Project   alt + ⏎

Select Generate Project

SPRING INITIALIZR bootstrap your application now

Generate a [Maven Project ∨] with [Java ∨] and Spring Boot [2.0.0 (SNAPSHOT)]

Project Metadata

Artifact coordinates

Group

com.example

Artifact

spring-boot-demo

Dependencies

Add Spring Boot Starters and dependencies to your application

Search for dependencies

Web, Security, JPA, Actuator, Devtools...

Selected Dependencies

Web ✕

Don't know what to look for? Want more options?

spring-boot-demo.zip - WinRAR (evaluation copy)

File Commands Tools Favorites Options Help

Add   Extract To   Test   View   Delete   Find   Wizard   Info   VirusScan   Comment   SFX

⬆ spring-boot-demo.zip - ZIP archive, unpacked size 62,465 bytes

Name	Size	Packed	Type	Modified	CRC32
..			Local Disk		
spring-boot-demo			File folder	17-10-2017 13:...	
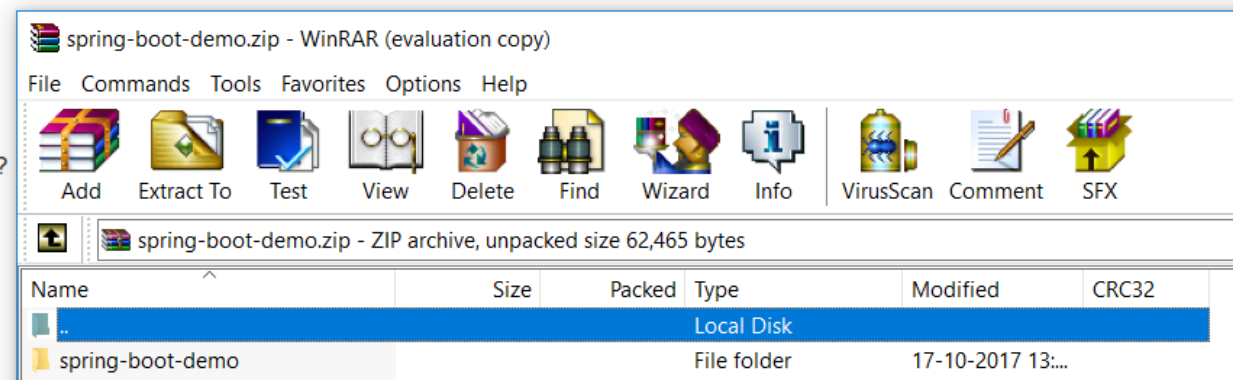
Save project and extract it.

Now import this project by using import option from the STS (Spring Tool Suite) IDE.
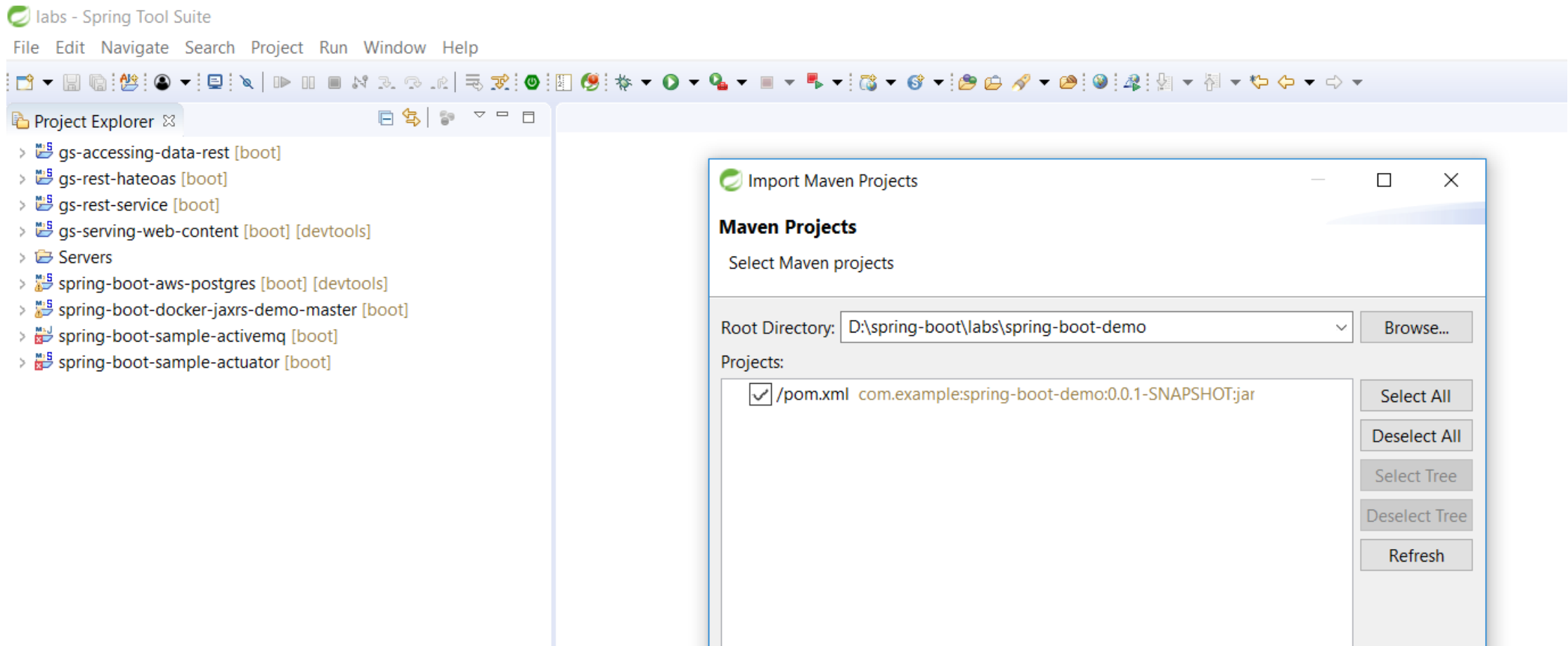
Steps to execute the demo lab:

Refer to pom.xml file -> It contains all the dependencies and configuration for the project.

Refer to generated Java file in the src/main/java directory.

Run this project by selecting Java Application from run options

Note : If the below error comes

Exception in thread "main" java.lang.NoClassDefFoundError:
org/springframework/core/env/EnvironmentCapable
            at java.lang.ClassLoader.defineClass1(Native Method)
            at java.lang.ClassLoader.defineClass(ClassLoader.java:763)

```
<parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.5.8.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
</parent>
```

Change to

```
<parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.5.7.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
</parent>
```

Observe the server started log

```
  _   _   _ | | _ \/ _   | \ \ \ \
 .)| .)| | | | |( | |  ) ) ) )
 _| ._|_| |_|_| |_\_, | / / / /
==|_|===========|___/=/_/_/_/
 ng Boot ::       (v1.5.7.RELEASE)

17 20:22:15.611  INFO 23508 --- [           main] c.e.s.SpringBootDemoApplication        : Starting SpringBootDemoApplication on DESKTOP-P5DJBAL with PID 23508 (D:\spring-boot
17 20:22:15.618  INFO 23508 --- [           main] c.e.s.SpringBootDemoApplication        : No active profile set, falling back to default profiles: default
17 20:22:15.780  INFO 23508 --- [           main] ationConfigEmbeddedWebApplicationContext : Refreshing org.springframework.boot.context.embedded.AnnotationConfigEmbeddedWebAppl:
17 20:22:19.845  INFO 23508 --- [           main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat initialized with port(s): 8080 (http)
17 20:22:19.871  INFO 23508 --- [           main] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
17 20:22:19.873  INFO 23508 --- [           main] org.apache.catalina.core.StandardEngine  : Starting Servlet Engine: Apache Tomcat/8.5.20
17 20:22:20.081  INFO 23508 --- [ost-startStop-1] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
17 20:22:20.081  INFO 23508 --- [ost-startStop-1] o.s.web.context.ContextLoader            : Root WebApplicationContext: initialization completed in 4307 ms
17 20:22:20.430  INFO 23508 --- [ost-startStop-1] o.s.b.w.servlet.ServletRegistrationBean  : Mapping servlet: 'dispatcherServlet' to [/]
17 20:22:20.446  INFO 23508 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter: 'characterEncodingFilter' to: [/*]
17 20:22:20.447  INFO 23508 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter: 'hiddenHttpMethodFilter' to: [/*]
17 20:22:20.448  INFO 23508 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter: 'httpPutFormContentFilter' to: [/*]
17 20:22:20.448  INFO 23508 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter: 'requestContextFilter' to: [/*]
17 20:22:20.988  INFO 23508 --- [           main] s.w.s.m.m.a.RequestMappingHandlerAdapter : Looking for @ControllerAdvice: org.springframework.boot.context.embedded.AnnotationCo
17 20:22:21.129  INFO 23508 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/error]}" onto public org.springframework.http.ResponseEntity<java.util.Ma
17 20:22:21.148  INFO 23508 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/error],produces=[text/html]}" onto public org.springframework.web.servlet.
17 20:22:21.191  INFO 23508 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMapping  : Mapped URL path [/webjars/**] onto handler of type [class org.springframework.web.ser
17 20:22:21.191  INFO 23508 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMapping  : Mapped URL path [/**] onto handler of type [class org.springframework.web.servlet.res
17 20:22:21.256  INFO 23508 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMapping  : Mapped URL path [/**/favicon.ico] onto handler of type [class org.springframework.wel
17 20:22:21.292  INFO 23508 --- [           main] oConfiguration$WelcomePageHandlerMapping : Adding welcome page: class path resource [static/index.html]
17 20:22:21.448  INFO 23508 --- [           main] o.s.j.e.a.AnnotationMBeanExporter        : Registering beans for JMX exposure on startup
17 20:22:21.533  INFO 23508 --- [           main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http)
17 20:22:21.540  INFO 23508 --- [           main] c.e.s.SpringBootDemoApplication          : Started SpringBootDemoApplication in 7.062 seconds (JVM running for 8.807)
17 20:22:22.051  INFO 23508 --- [nio-8080-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring FrameworkServlet 'dispatcherServlet'
17 20:22:22.051  INFO 23508 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet        : FrameworkServlet 'dispatcherServlet': initialization started
17 20:22:22.083  INFO 23508 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet        : FrameworkServlet 'dispatcherServlet': initialization completed in 32 ms
```

From browser access the url http://localhost:8080
HTTP 404 Not Found Error-> add static/index.html file

The **mvn dependency:tree** command prints a tree representation of your project dependencies.

What is Spring Boot Starter Parent?

All Spring Boot projects typically use spring-boot-starter-parent as the parent in pom.xml.

```
<parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.5.7.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
</parent>
```

Parent-pom allow us to manage the following things for multiple child projects and modules:

Configuration - Java Version and Other Properties
Depedency Management - Version of dependencies
Default Plugin Configuration

What is inside Spring Boot Starter Parent?

First of all - Spring Boot Starter Parent defines spring-boot-dependencies as the parent pom. It inherits dependency management from spring-boot-dependencies.

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

Default java version is 1.6. A project can override this by specifying a property <java.version>1.8</java.version> in the project pom.

A few other settings related to encoding and source, target version are also set in the parent pom.

```
<java.version>1.6</java.version>
<resource.delimiter>@</resource.delimiter> <!-- delimiter that doesn't
clash with Spring ${} placeholders -->
<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
<project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
<maven.compiler.source>${java.version}</maven.compiler.source>
<maven.compiler.target>${java.version}</maven.compiler.target>
```

Spring Boot Starter Parent specifies the default configuration for a host of plugins including maven-failsafe-plugin, maven-jar-plugin and maven-surefire-plugin.

What does Spring Boot Starter Parent inherit from spring-boot-dependencies?

Spring Boot Dependencies defines the default dependency management for all Spring Boot projects. If we would want to use a new version of a specific dependency, we can override the version by specifying a new property in the project pom.

```
<properties>
        <activemq.version>5.13.4</activemq.version>
        ...
        <ehcache.version>2.10.2.2.21</ehcache.version>
        <ehcache3.version>3.1.1</ehcache3.version>
        ...
        <h2.version>1.4.192</h2.version>
```

```
[INFO] +- org.springframework.boot:spring-boot-starter-web:jar:1.5.7.RELEASE:compile
[INFO] |  +- org.springframework.boot:spring-boot-starter:jar:1.5.7.RELEASE:compile
[INFO] |  |  +- org.springframework.boot:spring-boot:jar:1.5.7.RELEASE:compile
[INFO] |  |  +- org.springframework.boot:spring-boot-autoconfigure:jar:1.5.7.RELEASE:compile
[INFO] |  |  +- org.springframework.boot:spring-boot-starter-logging:jar:1.5.7.RELEASE:compile
[INFO] |  |  |  +- ch.qos.logback:logback-classic:jar:1.1.11:compile
[INFO] |  |  |  |  \- ch.qos.logback:logback-core:jar:1.1.11:compile
[INFO] |  |  |  +- org.slf4j:jcl-over-slf4j:jar:1.7.25:compile
[INFO] |  |  |  +- org.slf4j:jul-to-slf4j:jar:1.7.25:compile
[INFO] |  |  |  \- org.slf4j:log4j-over-slf4j:jar:1.7.25:compile
[INFO] |  |  \- org.yaml:snakeyaml:jar:1.17:runtime
[INFO] |  +- org.springframework.boot:spring-boot-starter-tomcat:jar:1.5.7.RELEASE:compile
[INFO] |  |  +- org.apache.tomcat.embed:tomcat-embed-core:jar:8.5.20:compile
[INFO] |  |  +- org.apache.tomcat.embed:tomcat-embed-el:jar:8.5.20:compile
[INFO] |  |  \- org.apache.tomcat.embed:tomcat-embed-websocket:jar:8.5.20:compile
[INFO] |  +- org.hibernate:hibernate-validator:jar:5.3.5.Final:compile
[INFO] |  |  +- javax.validation:validation-api:jar:1.1.0.Final:compile
[INFO] |  |  +- org.jboss.logging:jboss-logging:jar:3.3.1.Final:compile
[INFO] |  |  \- com.fasterxml:classmate:jar:1.3.4:compile
[INFO] |  +- com.fasterxml.jackson.core:jackson-databind:jar:2.8.10:compile
[INFO] |  |  +- com.fasterxml.jackson.core:jackson-annotations:jar:2.8.0:compile
[INFO] |  |  \- com.fasterxml.jackson.core:jackson-core:jar:2.8.10:compile
[INFO] |  +- org.springframework:spring-web:jar:4.3.11.RELEASE:compile
[INFO] |  |  +- org.springframework:spring-aop:jar:4.3.11.RELEASE:compile
[INFO] |  |  +- org.springframework:spring-beans:jar:4.3.11.RELEASE:compile
[INFO] |  |  \- org.springframework:spring-context:jar:4.3.11.RELEASE:compile
[INFO] |  \- org.springframework:spring-webmvc:jar:4.3.11.RELEASE:compile
[INFO] |     \- org.springframework:spring-expression:jar:4.3.11.RELEASE:compile
```

The Spring Boot Dashboard

Starting small and incrementally, the Spring Boot Dashboard comes as an additional view in STS/Eclipse.

We can open it by pressing the Spring Boot button in your main toolbar. It opens up a simple view that is in sync with your workspace projects, showing those projects of your workspace that are Spring Boot projects.
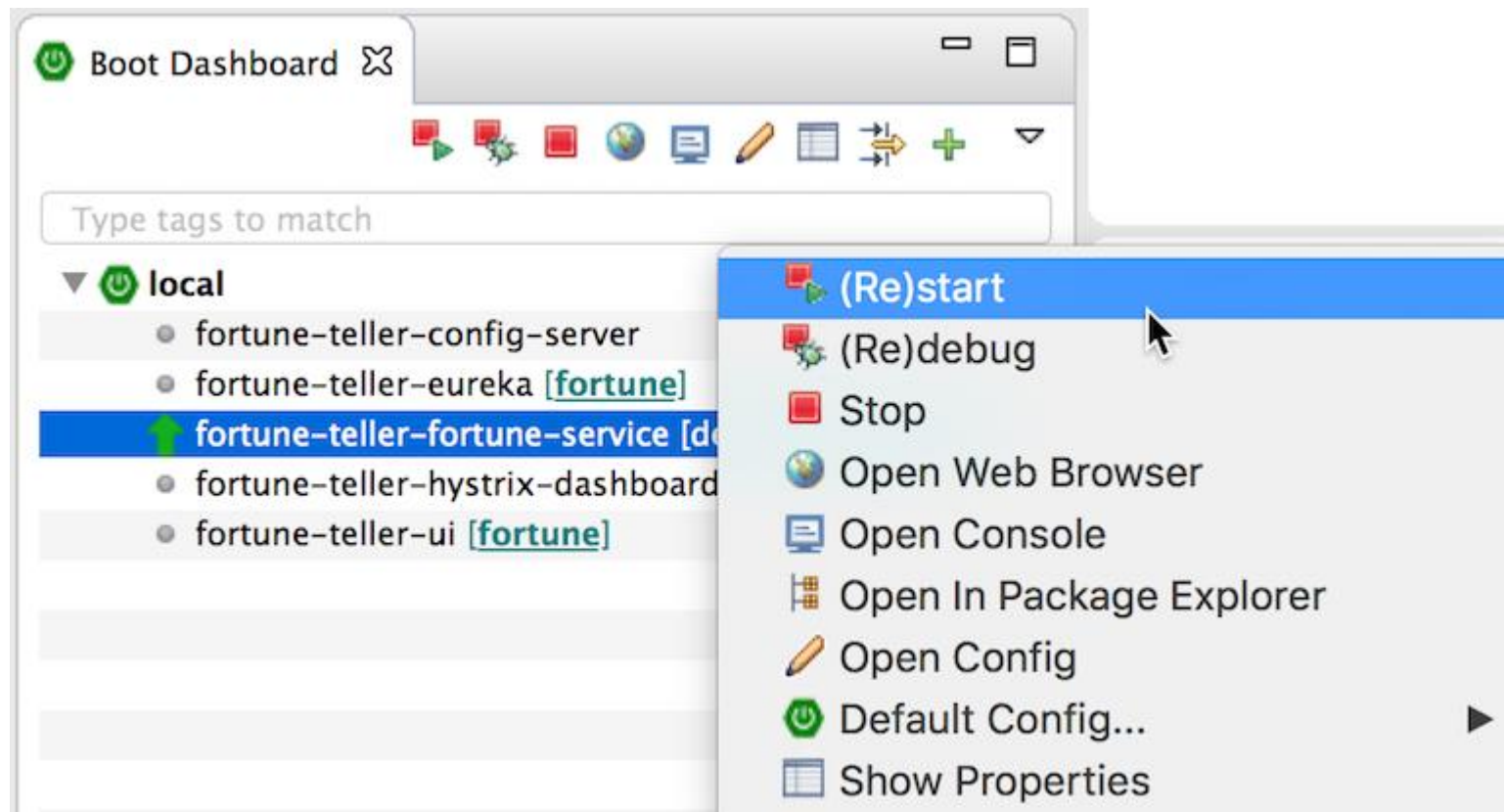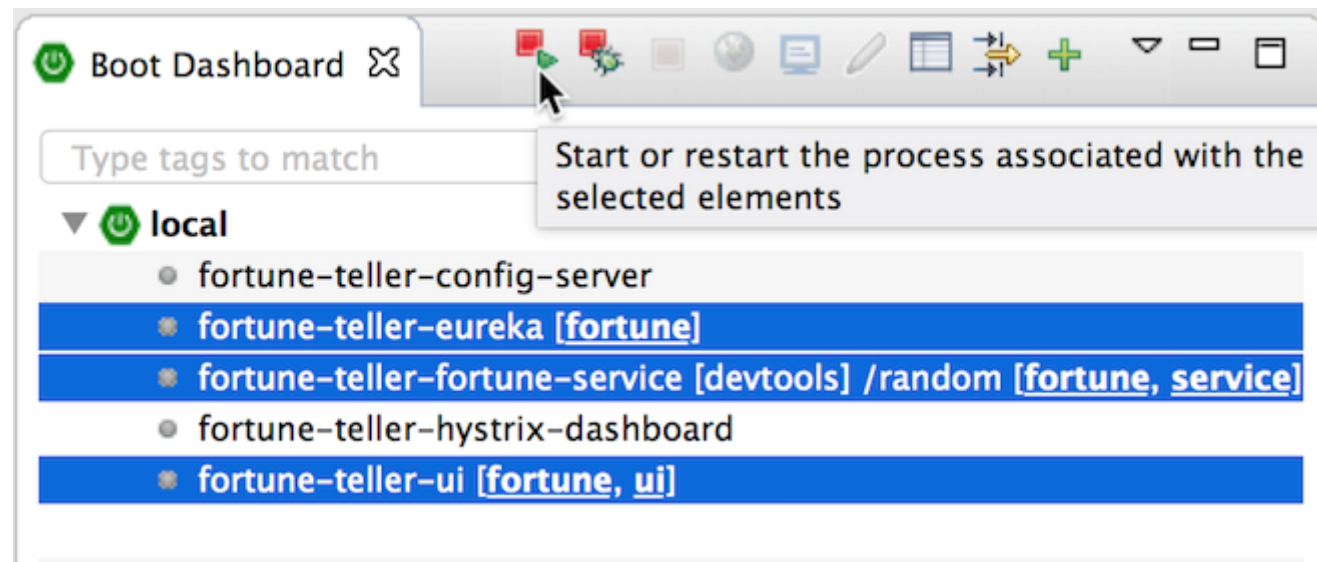
Boot Dashboard ⋊

Type tags to match

▼ ⏻ **local**
- fortune-teller-config-server
- fortune-teller-eureka [fortune]
- fortune-teller-fortune-service [devtools] /random [fortune, service]
- fortune-teller-hystrix-dashboard
- fortune-teller-ui [fortune, ui]

The main purpose of the Boot Dashboard is to give us quick access to the things that we do every day, every hour, or even more frequently.
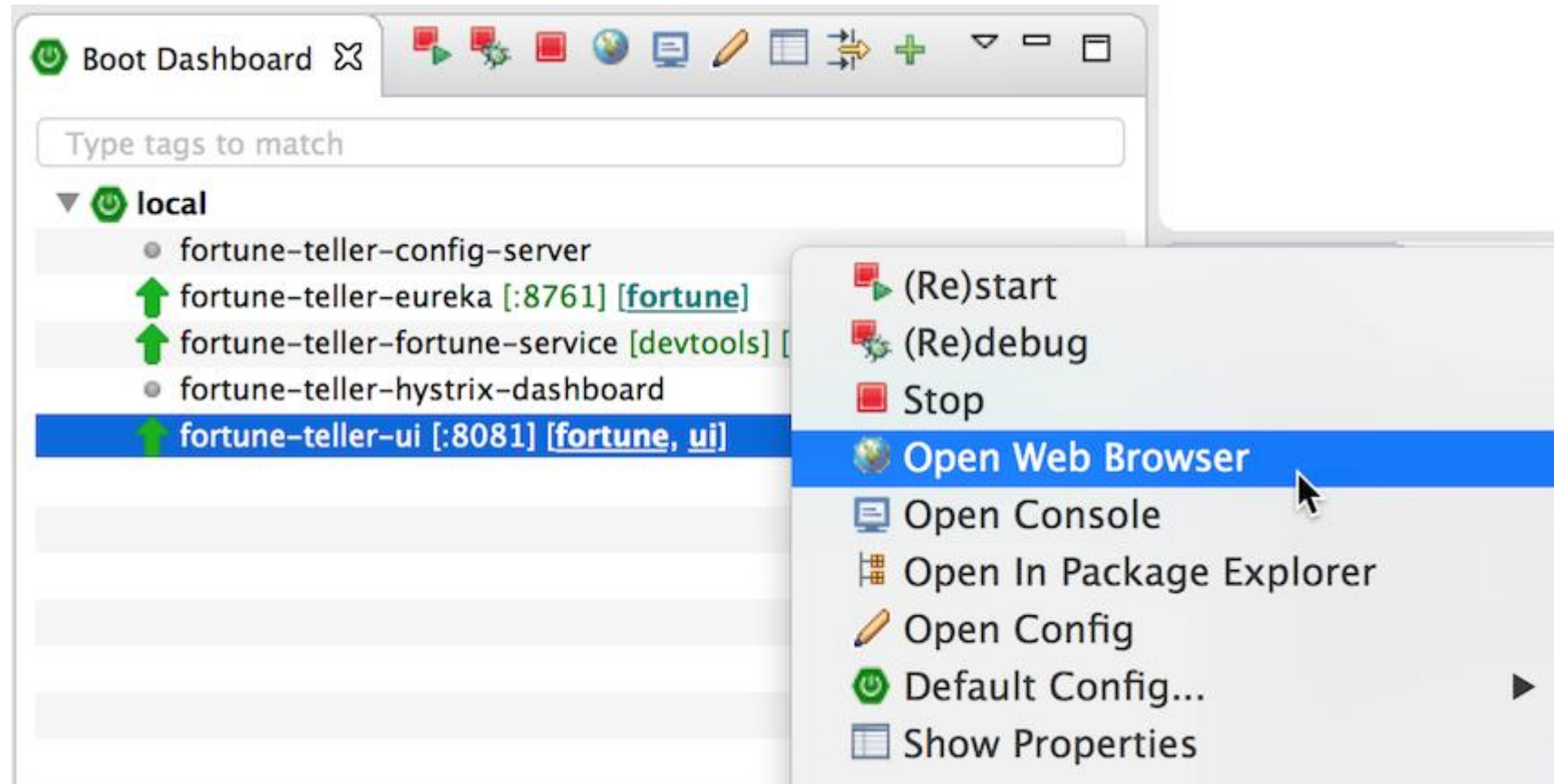
We can quickly start your boot apps (in run or debug mode) by selecting them in the boot dashboard and pressing the "run" or "debug" action in the toolbar

If We want to start or stop a bunch of boot apps at the same time, for example to startup a set of collaborating services, just select them all in the dashboard and press "run".
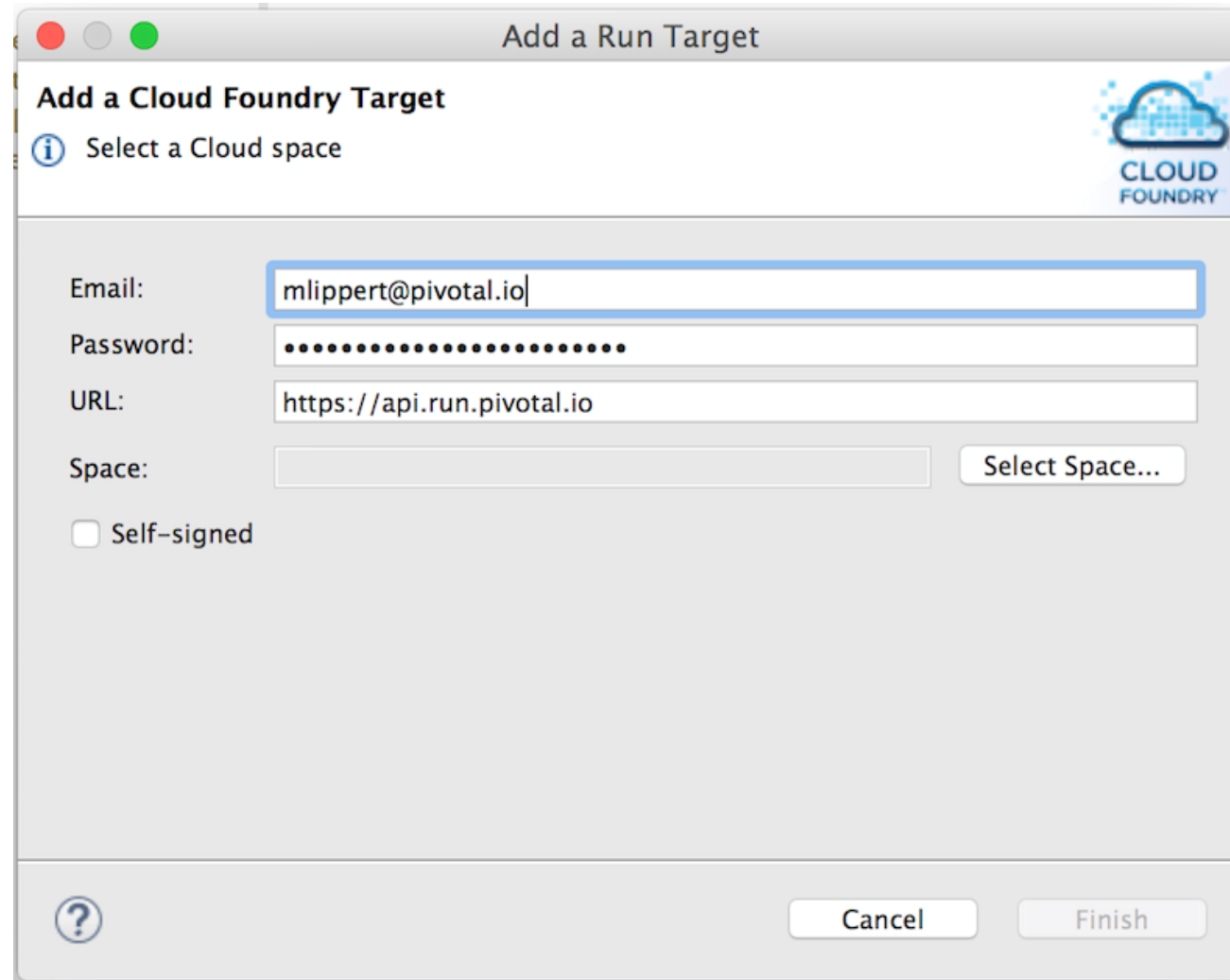
Open a browser tab for the app

Cloud runtime support

We can add a Cloud Foundry section to the boot dashboard using the big plus icon in the toolbar.

The route is invalid: host format pivotal

Spring Project name must be small

*** The route is invalid*: *host* must be no more than 63 characters

Droplet

An archive within Elastic Runtime that contains the application ready to run on Diego. A droplet is the result of the application staging process.

Diego

Cloud Foundry uses the Diego architecture to manage application containers. Diego components assume application scheduling and management responsibility from the Cloud Controller.

We can select multiple entries in the boot dashboard across those target sections.
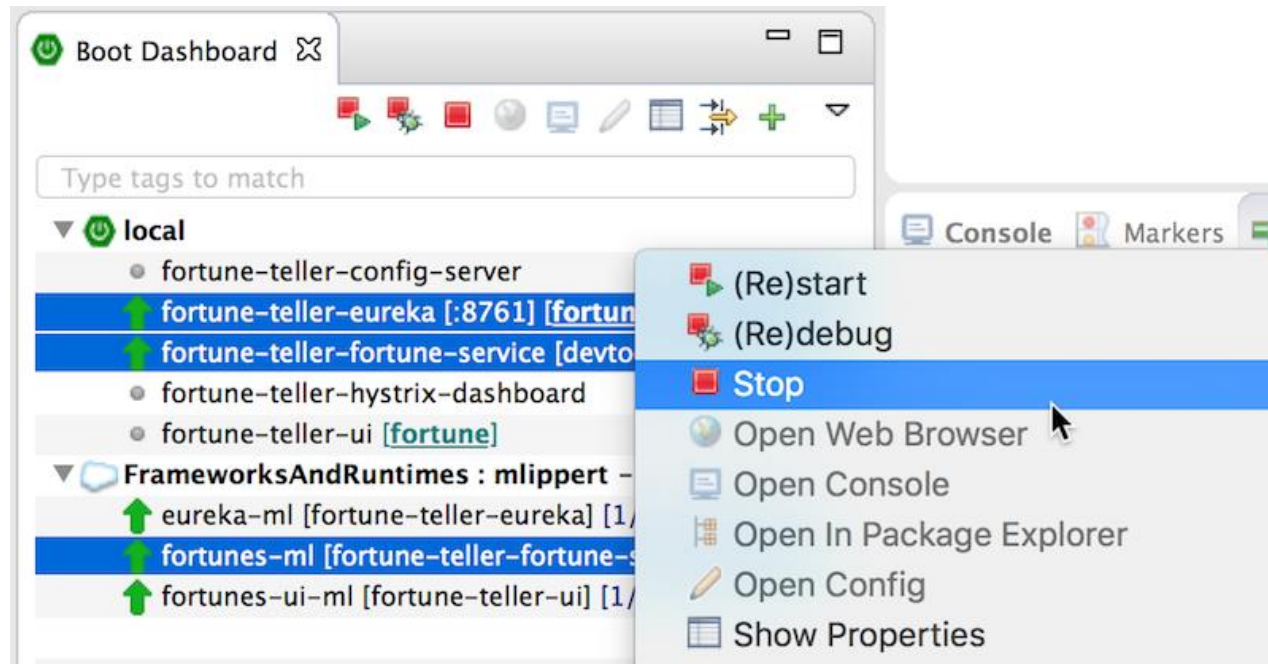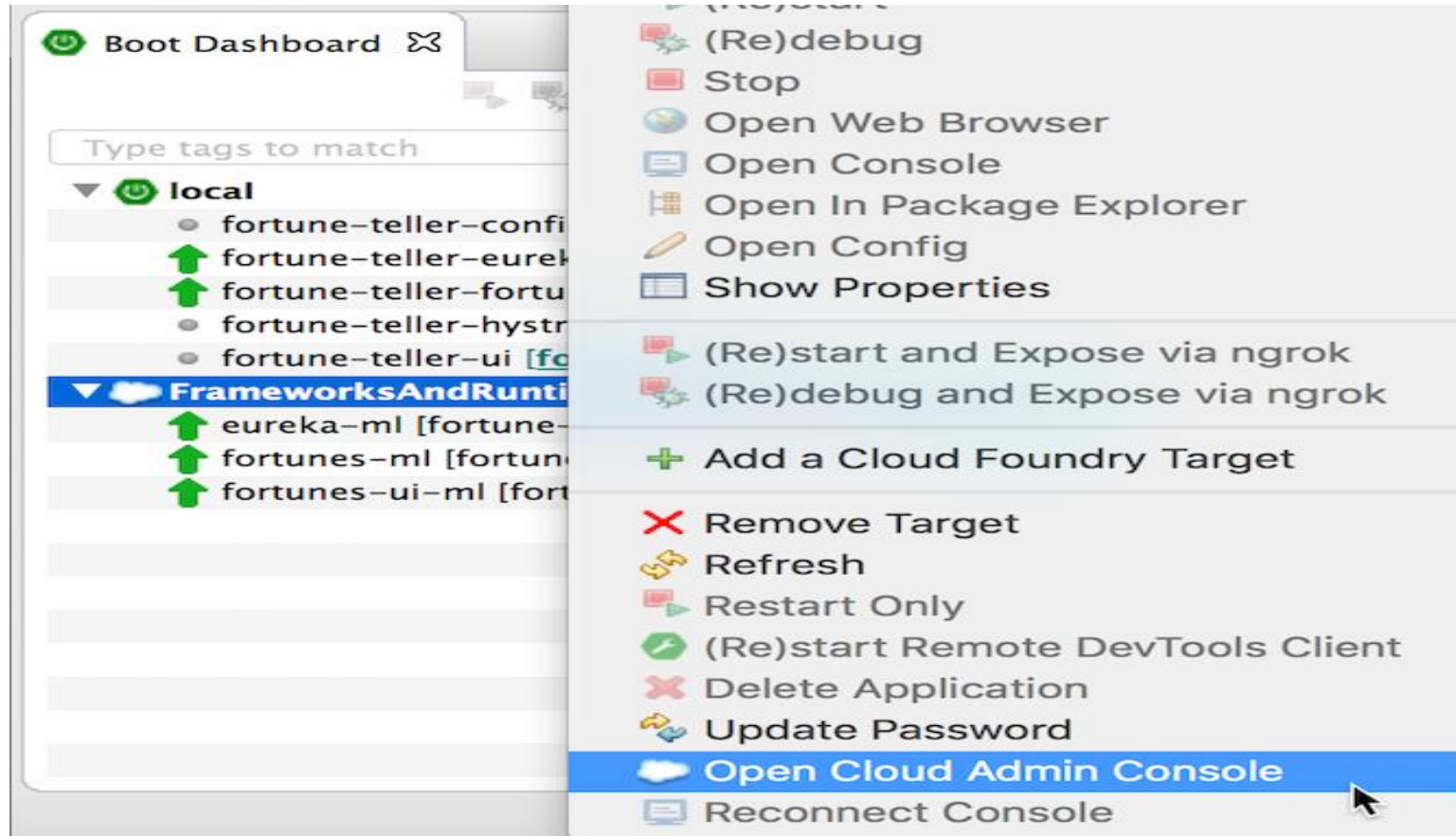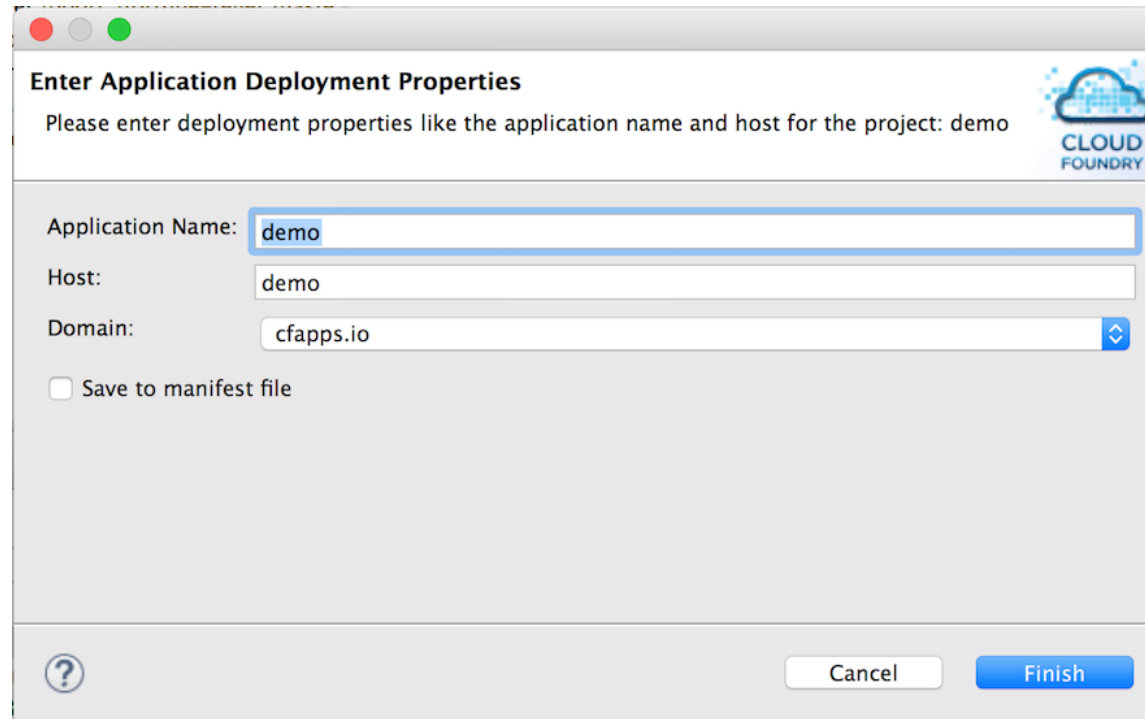
Deploying to Cloud Foundry

If application contains a manifest.yml file, this will be used to configure the application for Cloud Foundry. This typically contains the name of the app, the domain, memory settings, number of instances, and potentially a lot more.

If we don't have a manifest.yml file in our project, the deploy action will prompt you in a dialog for the basic information it needs to deploy the app

Quickly deploy code changes

For local apps, using the Spring Boot Devtools is extremely easy and straightforward.

As soon as we add the Spring Boot Devtools to our project as a dependency (Project Explorer->select Project->Spring Tools->Add Spring Project Nature->then select Spring Tools->Add Boot Devtools) and start your app, it will listen for local changes to configuration and class files and kick a restart of the app for you automatically.

We don't even need to restart the app yourself, the Spring Boot Devtools will do that for us automatically.

Project Explorer->select Project->Spring Tools->Add Spring Project Nature ->then select Spring Tools->Add Boot Devtools
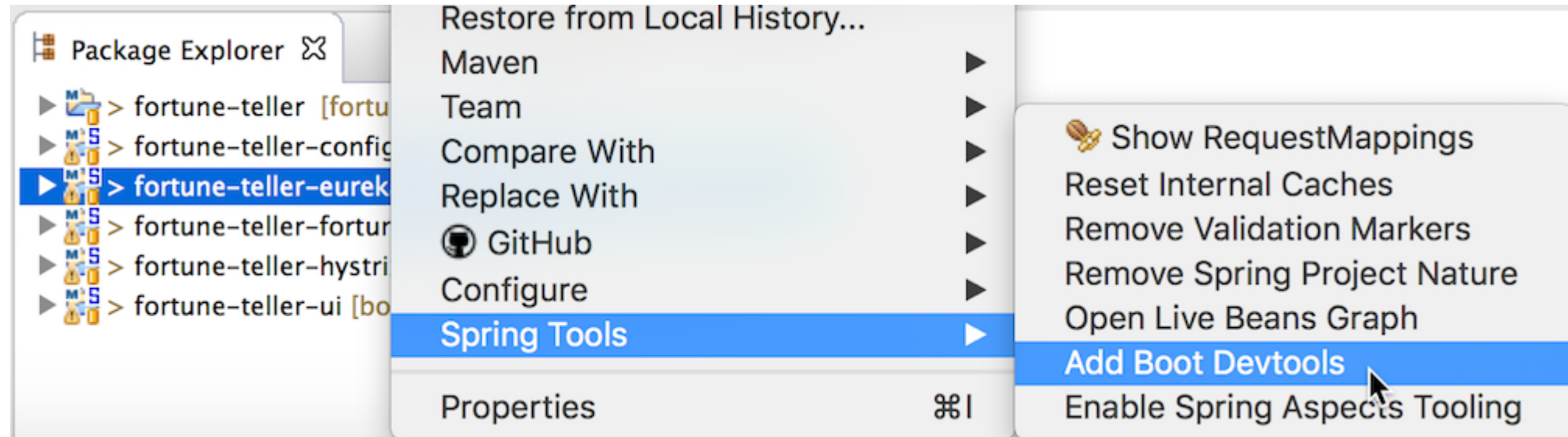
(Re)start Remote DevTools Client:

Once the devtools-enabled boot app is running on Cloud Foundry, you can easily start the boot devtools remote client application for it.

This client application runs locally on your machine and connects to the remote application on Cloud Foundry.

It watches for file changes, uploads them to the app on Cloud Foundry and triggers a restart of the app on Cloud Foundry.
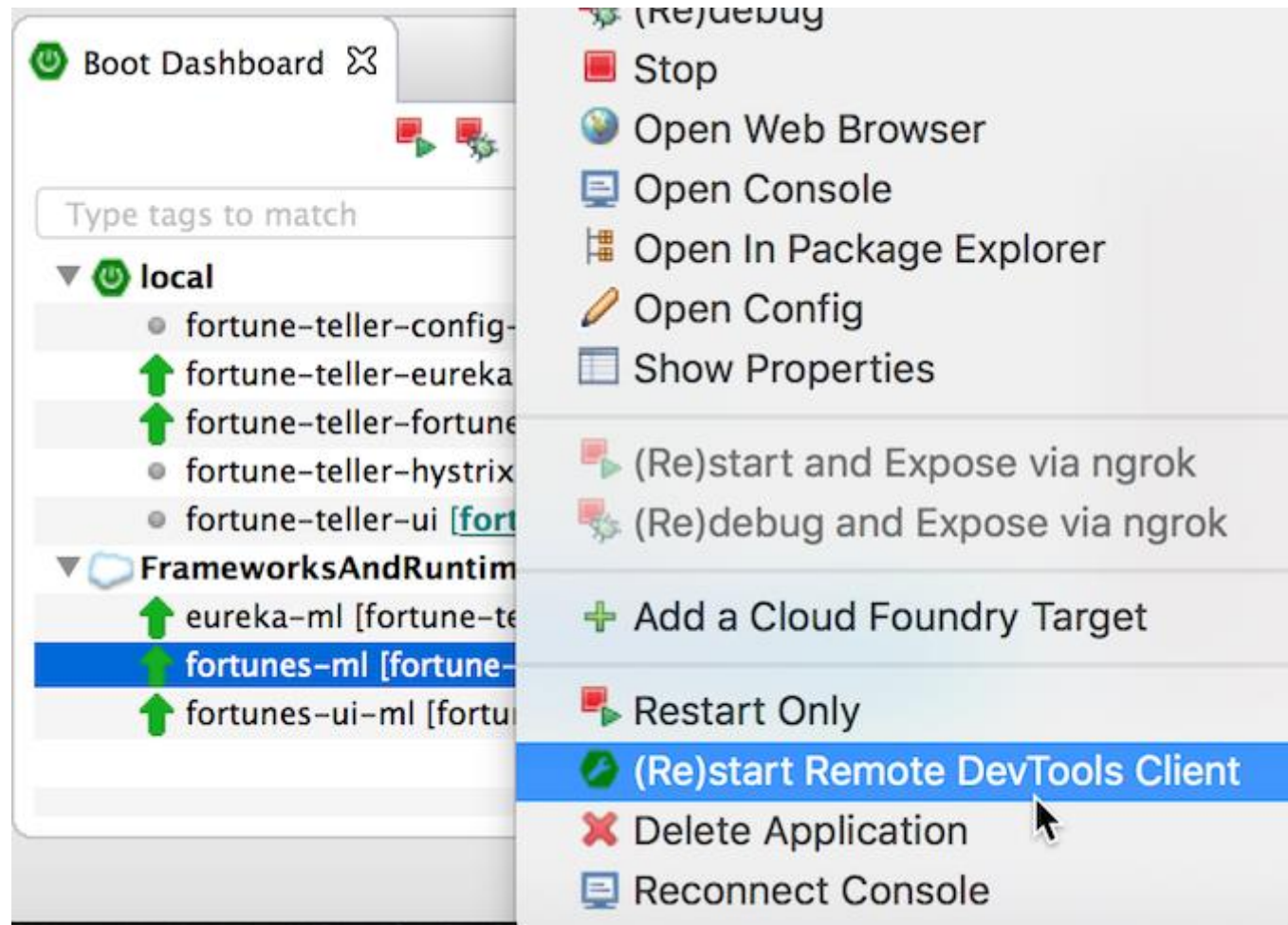
Choose a manifest file when deploying an app(manifest.yml)

Spring Boot Dashboard supports Cloud Foundry manifest files as a first-class citizen.

This starts to show up when you deploy an application from within the Boot Dashboard to a Cloud Foundry target for the first time.

A dialog comes up and asks you which manifest file should be used for this deployment operation.

Select Deployment Manifest for 'gs-rest-service-complete'

ⓘ Choose an existing deployment manifest YAML file from the local file system.

Manifest Type

● File   ○ Manual

Workspace File

▶ fortune-teller-ui
▼ 🗂⁵ gs-rest-service-complete
    📄 manifest-multiple-apps.yml
    📄 manifest-pws-integration.yml
    📄 **manifest-pws.yml**
    📄 manifest-with-inherit.yml
    ☁ manifest.yml
    📄 parent-manifest.yml
▶ 🗂⁵ gs-rest-service-cors-complete

⟳ Refresh

Manifest YAML files – *manifest*.yml ▲▼

YAML Content

/gs-rest-service-complete/manifest-pws.yml

```
---
applications:
- name: rest-service-ml-pws-dev
  host: rest-service-ml-pws-dev
  memory: 1024M
  instances: 1
  disk_quota: 512M
  domain: cfapps.io
```

ⓘ?   Cancel   OK

Updated manifest.yml file (Redelpoy)

We can change our mind and decide to choose a different manifest file.

An action in the context menu allows us to open the same "choose the manifest file" dialog and make a different choice.

The next redeploy of the app will take that change into account.
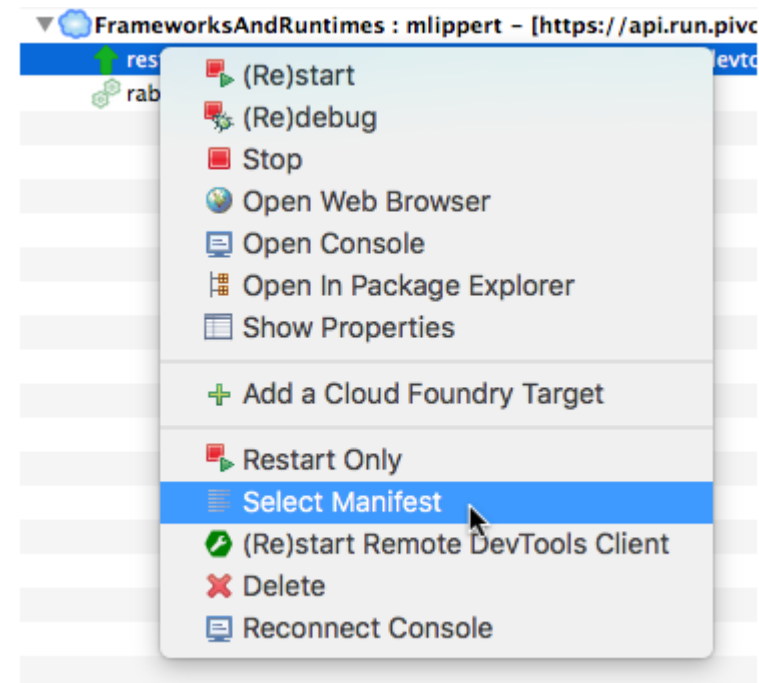
Deploying without a manifest file
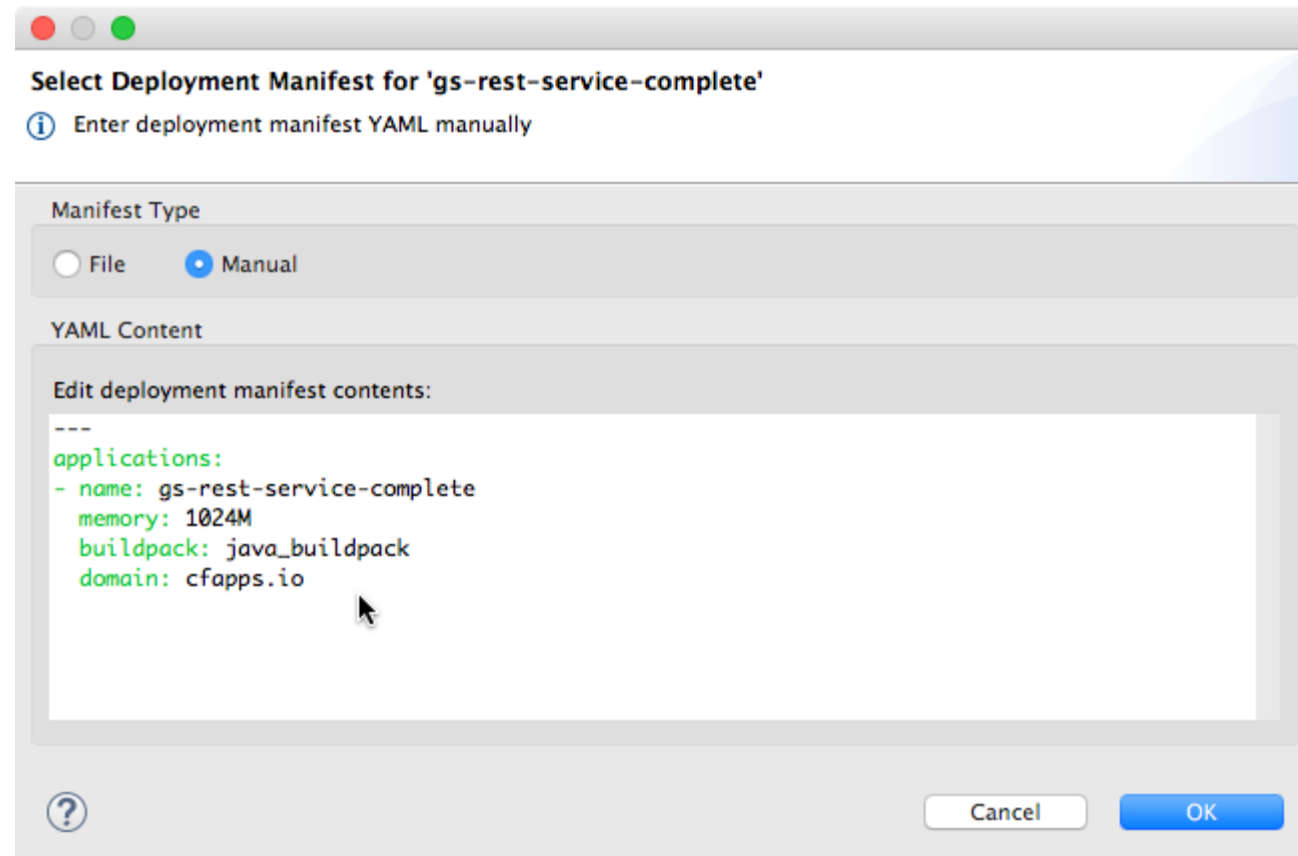
Spring Boot application starters

Name	Description
spring-boot-starter	Core starter, including auto-configuration support, logging and YAML
spring-boot-starter-activemq	Starter for JMS messaging using Apache ActiveMQ
spring-boot-starter-amqp	Starter for using Spring AMQP and Rabbit MQ
spring-boot-starter-aop	Starter for aspect-oriented programming with Spring AOP and AspectJ

Spring Boot production starters

Name	Description
spring-boot-starter-actuator	Starter for using Spring Boot's Actuator which provides production ready features to help you monitor and manage your application
spring-boot-starter-remote-shell	Starter for using the CRaSH remote shell to monitor and manage your application over SSH. Deprecated since 1.5

Spring Boot technical starters

Name	Description
spring-boot-starter-jetty	Starter for using Jetty as the embedded servlet container. An alternative to spring-boot-starter-tomcat
spring-boot-starter-log4j2	Starter for using Log4j2 for logging. An alternative to spring-boot-starter-logging
spring-boot-starter-logging	Starter for logging using Logback. Default logging starter
spring-boot-starter-tomcat	Starter for using Tomcat as the embedded servlet container. Default servlet container starter used by spring-boot-starter-web
spring-boot-starter-undertow	Starter for using Undertow as the embedded servlet container. An alternative to spring-boot-starter-tomcat

Structuring your code

Spring Boot does not require any specific code layout to work, however, there are some best practices that help.

Using the "default" package

When a class doesn't include a package declaration it is considered to be in the "default package".

The use of the "default package" is generally discouraged, and should be avoided.

It can cause particular problems for Spring Boot applications that use @ComponentScan, @EntityScan or @SpringBootApplication annotations, since every class from every jar, will be read.

Locating the main application class

It is recommend that the main application class is in a root package above other classes.

The @EnableAutoConfiguration annotation is often placed on the main class, and it implicitly defines a base "search package" for certain items. For example, if you are writing a JPA application, the package of the @EnableAutoConfiguration annotated class will be used to search for @Entity items.

Using a root package also allows the @ComponentScan annotation to be used without needing to specify a basePackage attribute.

We can also use the @SpringBootApplication annotation if the main class is in the root package.

Here is a typical layout:

```
com
 +- example
     +- myproject
         +- Application.java
         |
         +- domain
         |   +- Customer.java
         |   +- CustomerRepository.java
         |
         +- service
         |   +- CustomerService.java
         |
         +- web
             +- CustomerController.java
```

SpringApplication

The SpringApplication class provides a convenient way to bootstrap a Spring application that will be started from a main() method.

In many situations we can just delegate to the static SpringApplication.run method:

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

By default INFO logging messages will be shown, including some relevant startup details such as the user that launched the application.

Startup failure

If the application fails to start, registered FailureAnalyzers get a chance to provide a dedicated error message and a concrete action to fix the problem. For instance if you start a web application on port 8080 and that port is already in use, we should see something similar to the following:

```
***************************
APPLICATION FAILED TO START
***************************


Description:

Embedded servlet container failed to start. Port 8080 was already in use.

Action:

Identify and stop the process that's listening on port 8080 or configure this application to listen on another port.
```

FailureAnalyzer

FailureAnalyzer is a great way to intercept an exception on startup and turn it into a human-readable message, wrapped into a FailureAnalysis. Spring Boot provides such analyzer for application context related exceptions, JSR-303 validations and more.

AbstractFailureAnalyzer is a convenient extension of FailureAnalyzer that checks the presence of a specified exception type in the exception to handle. We can extend that to handle the exception only when it is actually present.

FailureAnalyzer implementations are to be registered in a META-INF/spring.factories: the following registers ProjectConstraintViolationFailureAnalyzer:

org.springframework.boot.diagnostics.FailureAnalyzer=\
com.example.ProjectConstraintViolationFailureAnalyzer

Customize the Environment or ApplicationContext before it starts

A SpringApplication has ApplicationListeners and ApplicationContextInitializers that are used to apply customizations to the context or environment. Spring Boot loads a number of such customizations for use internally from META-INF/spring.factories.

There is more than one way to register additional ones:

Programmatically per application by calling the addListeners and addInitializers methods on SpringApplication before you run it.

Declaratively per application by setting context.initializer.classes or context.listener.classes.

Declaratively for all applications by adding a META-INF/spring.factories and packaging a jar file that the applications all use as a library.

The SpringApplication sends some special ApplicationEvents to the listeners,and then registers the listeners for events published by the ApplicationContext as well.

An ApplicationStartingEvent is sent at the start of a run, but before any processing except the registration of listeners and initializers.

An ApplicationEnvironmentPreparedEvent is sent when the Environment to be used in the context is known, but before the context is created.

An ApplicationPreparedEvent is sent just before the refresh is started, but after bean definitions have been loaded.

An ApplicationReadyEvent is sent after the refresh and any related callbacks have been processed to indicate the application is ready to service requests.

An ApplicationFailedEvent is sent if there is an exception on startup.

Externalize the configuration of SpringApplication

SpringApplication properties can be set using Java API or externalize the configuration using properties

application.properties :

spring.main.web-environment=false
spring.main.banner-mode=off

Using Java API:

```
new SpringApplicationBuilder()
    .bannerMode(Banner.Mode.OFF)
    .sources(demo.MyApp.class)
    .run(args);
```

Using 'short' command line arguments

We can define --port=9000 instead of --server.port=9000 to set configuration properties on the command line.

We can easily enable this by using placeholders in application.properties

e.g.,
server.port=${port:8080}

Use YAML for external properties

YAML is a superset of JSON and as such is a very convenient syntax for storing external properties in a hierarchical format. E.g.

```
spring:
    application:
        name: cruncher
    datasource:
        driverClassName: com.mysql.jdbc.Driver
        url: jdbc:mysql://localhost/test
server:
    port: 9000
```

The example YAML above corresponds to an application.properties file:

```
spring.application.name=cruncher
spring.datasource.driverClassName=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost/test
server.port=9000
```

Profiles

Spring Profiles provide a way to segregate parts of the application configuration and make it only available in certain environments. Any @Component or @Configuration can be marked with @Profile to limit when it is loaded:

```
@Configuration
@Profile("production")
public class ProductionConfiguration {
    // ...   }
```

In the normal Spring way, we can use a spring.profiles.active Environment property to specify which profiles are active. We can specify the property in any of the usual ways, for example you could include it in your application.properties:

spring.profiles.active=dev,hsqldb

Change configuration depending on the environment

A YAML file is actually a sequence of documents separated by --- lines, and each document is parsed separately to a flattened map.

If a YAML document contains a spring.profiles key, then the profiles value (comma-separated list of profiles) is fed into the Spring Environment.acceptsProfiles() and if any of those profiles is active that document is included in the final merge (otherwise not).
For Example:

```
server:
    name : Test Application
---
spring:
    profiles: development
server:
    port: 9001
---
spring:
    profiles: production
server:
    port: 8001
```

"spring-boot-autoconfigure" module

The following auto-configuration classes are from the spring-boot-autoconfigure module:

ActiveMQAutoConfiguration
AopAutoConfiguration
CacheAutoConfiguration
CassandraAutoConfiguration
WebMvcAutoConfiguration
HibernateJpaAutoConfiguration
ValidationAutoConfiguration
etc.,

Note : We can autoconfigure everything except ActiveMQ, you would use
@EnableAutoConfiguration(exclude=ActiveMQAutoConfiguration.class)

@EnableAutoConfiguration -> what has been configured?

The main feature of Spring Boot is its ability to automatically configuration the Spring application based on its included jar files. So the first thing you may want to look at is what have been configured for you. This can be done by running the application JVM argument **"-Ddebug**". We will then see the "Auto Configuration Report" displayed in your console like below:

```
============================
AUTO-CONFIGURATION REPORT
============================
Positive matches:
——————–
AopAutoConfiguration
– @ConditionalOnClass classes found:
```

Spring Framework: @RestController vs @Controller

The traditional MVC controller relies on the View technology, the RESTful web service controller simply returns the object and the object data is written directly to the HTTP response as JSON/XML.

Spring MVC traditional workflow
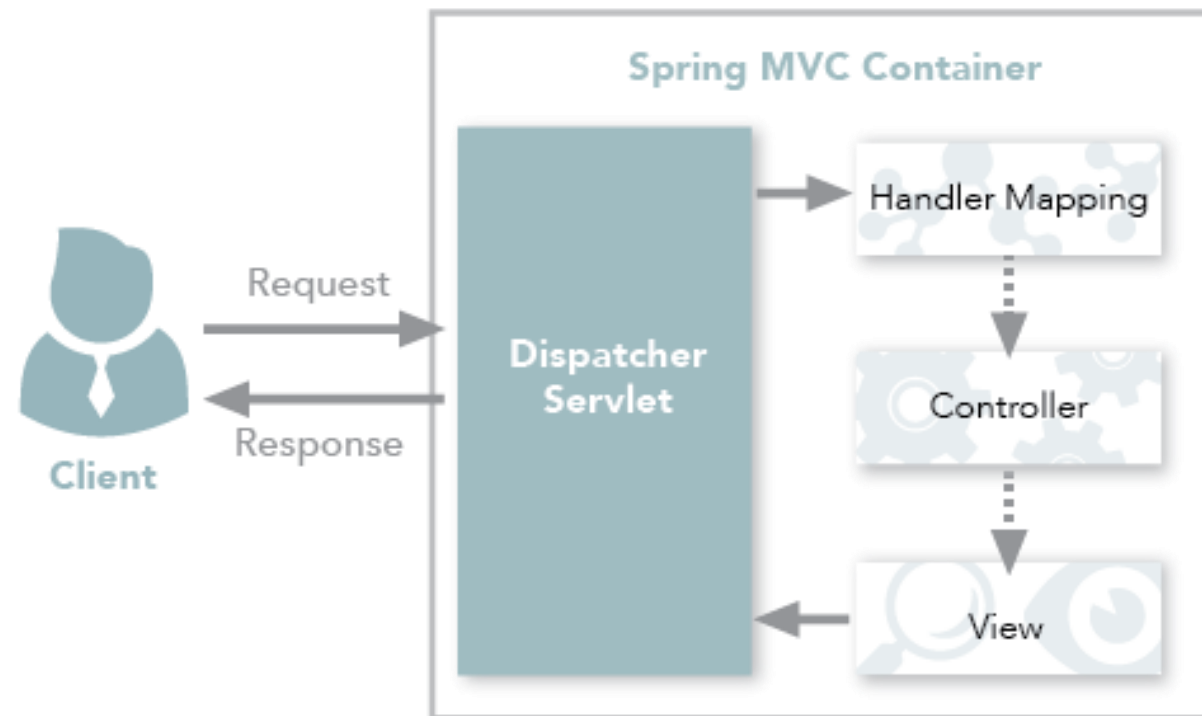
Spring 3.x MVC RESTful web services workflow

Using the @ResponseBody Annotation

When we use the @ResponseBody annotation on a method, Spring converts the return value and writes it to the http response automatically.

Each method in the Controller class must be annotated with @ResponseBody.
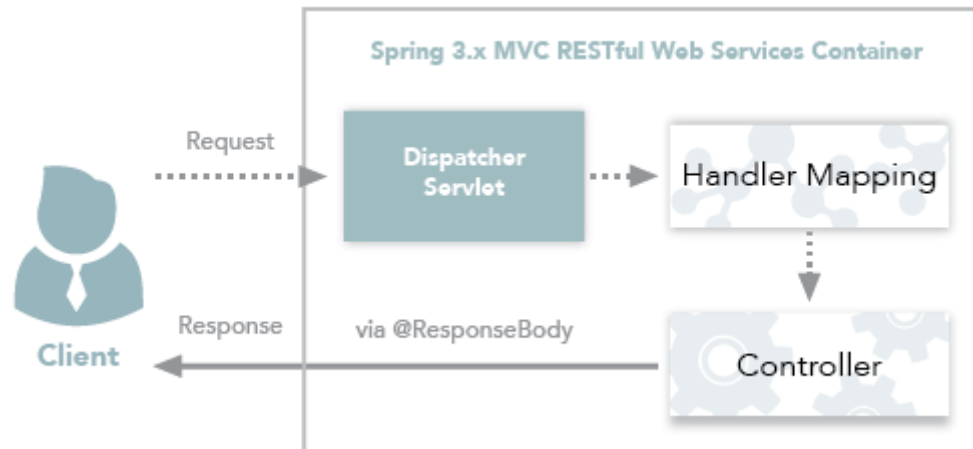
Spring 4.x MVC RESTful Web Services Workflow

Using the @RestController Annotation

Spring 4.0 introduced @RestController, a specialized version of the controller which is a convenience annotation that does nothing more than add the @Controller and @ResponseBody annotations.

By annotating the controller class with @RestController annotation, we no longer need to add @ResponseBody to all the request mapping methods.

The @ResponseBody annotation is active by default.
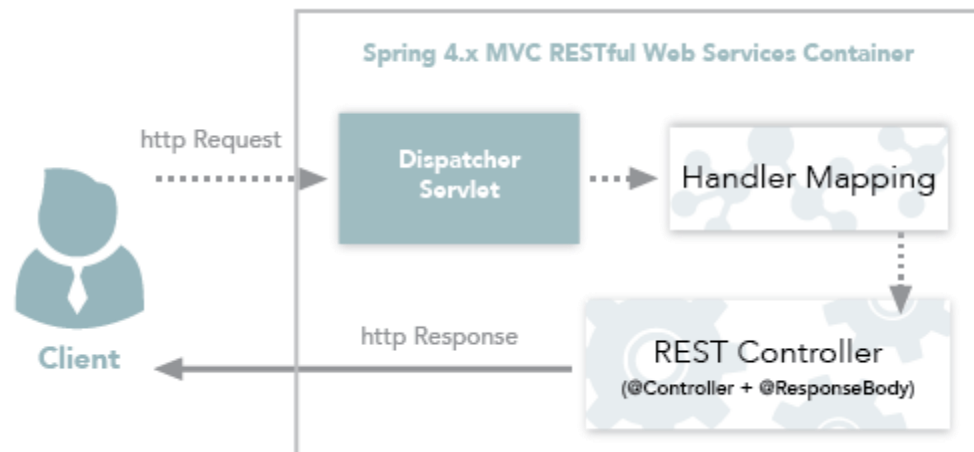
Spring 4.3 onwards:

@GetMapping is a composed annotation that acts as a shortcut for
@RequestMapping(method = RequestMethod.GET)

@PostMapping is a composed annotation that acts as a shortcut for
@RequestMapping(method = RequestMethod.POST)

@PutMapping is a composed annotation that acts as a shortcut for
@RequestMapping(method = RequestMethod.PUT)

@DeleteMapping is a composed annotation that acts as a shortcut for
@RequestMapping(method = RequestMethod.DELETE)

Hypermedia & HATEOAS

Hypermedia is an extension to what is known as hypertext, or the ability to open new Web pages by clicking text links on a Web browser.

Hypermedia extends upon this by allowing the user to click images, movies, graphics and other media apart from text to create a nonlinear network of information

HATEOAS (Hypermedia as the Engine of Application State) is a constraint of the REST application architecture.

A hypermedia-driven site provides information to navigate the site's REST interfaces dynamically by including hypermedia links with the responses.

Spring HATEOAS provides some APIs to ease creating REST representations that follow the HATEOAS principle when working with Spring and especially Spring MVC.

```
<dependency>
<groupId>org.springframework.hateoas</groupId>
<artifactId>spring-hateoas</artifactId>
<version>0.23.0.RELEASE</version> </dependency>
```

A HATEOAS-based response would look like this:

```
{
   "name": "Alice",
   "links": [ {
      "rel": "self",
      "href": "http://localhost:8080/customer/1"
   } ]
}
```

This response not only has the person's name, but includes the self-linking URL where that person is located.

rel means relationship. In this case, it's a self-referencing hyperlink.

More complex systems might include other relationships. For example, an order might have a "rel":"customer" relationship, linking the order to its customer (/customer/order/...)

href is a complete URL that uniquely defines the resource.

Spring Data – REST

Data Access Layer

JPA / Hibernate / JDBC

JPA

CRUD operations
EntityManager
EntityManagerFactory
Persistence

Hibernate

CRUD operations
Session
SessionFactory
Configuration

JDBC

CRUD operations
Connection
DataSouce

Persistence.xml

Javax.persistence.provide = Hibernate

+

Hibernate-jpa.jar
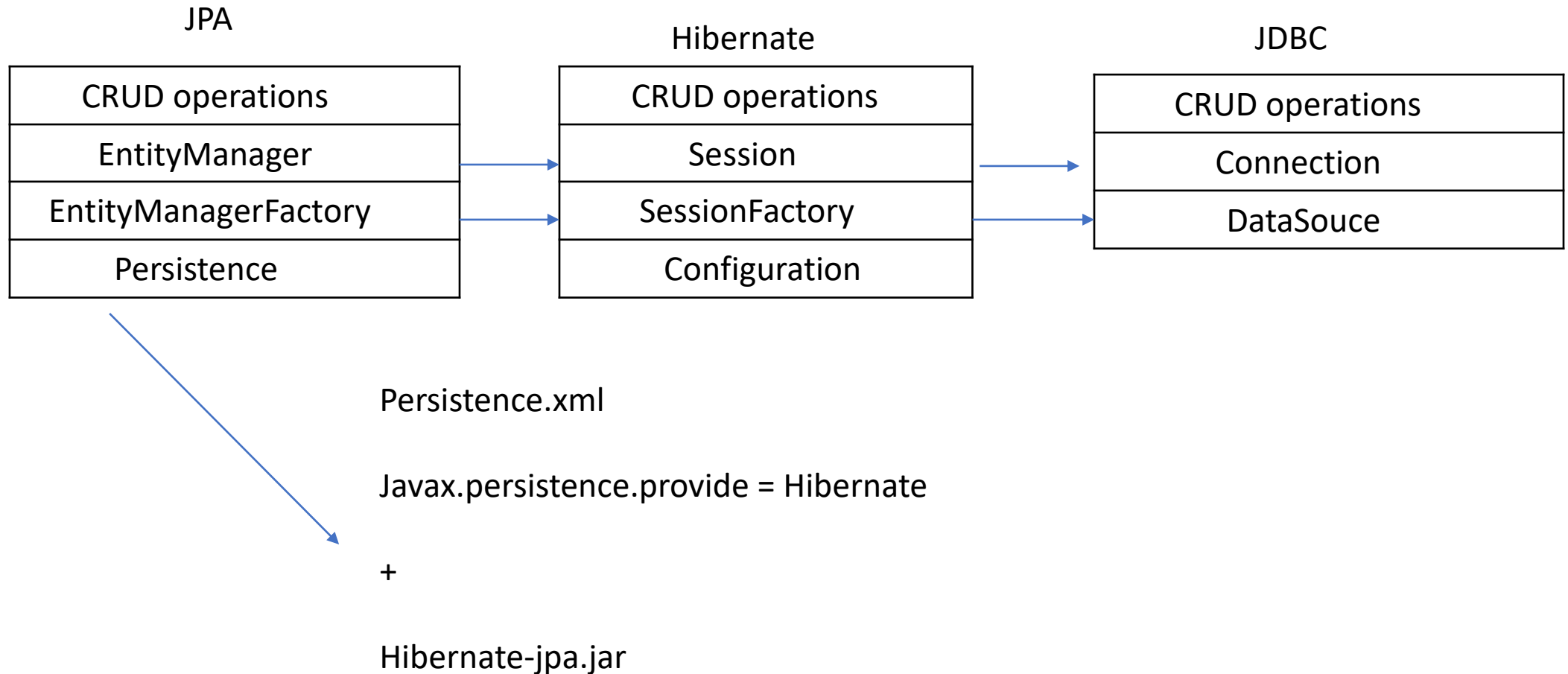
```java
@Entity
Class User
{
String name;
....
getName(){}

setName(){}
}
```

```java
interface UserDao
{

List<User> getUsers();

User getUser(long id);

void updateUser(User user);

void deleteUser(User user);

}
```

```java
@Repository
Class UserDaoJpa  implements UserDao
{

@Autowired
EntityManager  manager;


List<User> getUsers()
{  return manager.createQuery("from User"); }

User getUser(long id)
{ return  manager.find(User.class, id); }

void updateUser(User user) { ... }

void deleteUser(User user)  { ... }

}
```

Spring Data

Spring Data's goal is to provide a familiar and consistent, Spring-based programming model for data access while still retaining the special traits of the underlying data store.

It makes it easy to use data access technologies, relational and non-relational databases, map-reduce frameworks, and cloud-based data services

domain-driven design patterns

Features

❑ Powerful repository and custom object-mapping abstractions

❑ Dynamic query derivation from repository method names

❑ Support for transparent auditing (created, last changed)

❑ Possibility to integrate custom repository code

❑ Easy Spring integration via JavaConfig and custom XML namespaces

❑ Advanced integration with Spring MVC controllers

❑ Experimental support for cross-store persistence

Main modules

Spring Data Commons - Core Spring concepts underpinning every Spring Data project.

Spring Data JPA - Makes it easy to implement JPA-based repositories.

Spring Data KeyValue - Map-based repositories and SPIs to easily build a Spring Data module for key-value stores.

Spring Data LDAP - Provides Spring Data repository support for Spring LDAP.

Spring Data MongoDB - Spring based, object-document support and repositories for MongoDB.

Spring Data REST - Exports Spring Data repositories as hypermedia-driven RESTful resources.

Spring Data for Apache Cassandra - Spring Data module for Apache Cassandra.

org.springframework.data.repository

Interface Repository<T,ID>

Type Parameters:
T - the domain type the repository manages
ID - the type of the id of the entity the repository manages

All Known Subinterfaces:
CrudRepository<T,ID>, PagingAndSortingRepository<T,ID>,
ReactiveCrudRepository<T,ID>, ReactiveSortingRepository<T,ID>,
RevisionRepository<T,ID,N>, RxJava2CrudRepository<T,ID>,
RxJava2SortingRepository<T,ID>

CrudRepository Methods:

count()
Returns the number of entities available.

delete(T entity)
Deletes a given entity.

deleteAll(Iterable<? extends T> entities)
Deletes the given entities.

deleteById(ID id)
Deletes the entity with the given id.

save(S entity)
Saves a given entity.

saveAll(Iterable<S> entities)
Saves all given entities.

org.springframework.data.repository
Interface **PagingAndSortingRepository**<T,ID>

public interface PagingAndSortingRepository<T,ID>
extends CrudRepository<T,ID>

Extension of CrudRepository to provide additional methods to retrieve
entities using the pagination and sorting abstraction.

PagingAndSortingRepository methods are:

findAll(Pageable pageable)
Returns a Page of entities meeting the paging restriction
provided in the Pageable object.

findAll(Sort sort)
Returns all entities sorted by the given options.

JpaRepository - which extends PagingAndSortingRepository and, in turn, the CrudRepository.

It provides JPA related methods such as flushing the persistence context and delete records in a batch.

org.springframework.data.jpa.repository
Interface JpaRepository<T,ID>

All Superinterfaces:
CrudRepository<T,ID>, PagingAndSortingRepository<T,ID>,
QueryByExampleExecutor<T>, Repository<T,ID>

All Known Implementing Classes:
QuerydslJpaRepository, SimpleJpaRepository

Methods are :

deleteAllInBatch()
Deletes all entities in a batch call.

deleteInBatch(Iterable<T> entities)
Deletes the given entities in a batch which means it will create a single Query.

flush()
Flushes all pending changes to the database.

....

Methods inherited from interface
org.springframework.data.repository.PagingAndSortingRepository

❑ findAll

Methods inherited from interface org.springframework.data.repository.CrudRepository

❑ count, delete, deleteAll, deleteAll, deleteById, existsById, findById, save

Methods inherited from interface
org.springframework.data.repository.query.QueryByExampleExecutor

❑ count, exists, findAll, findOne

public @interface **RepositoryRestResource**
Annotate a Repository with this to customize export mapping and rels.

Simplifies building hypermedia-driven REST web services on top of Spring Data repositories.

Examples

The following code represents a Customer object.

```
class Customer
{
    String name;
}
```

A simple JSON presentation is traditionally rendered as:

```
{
    "name" : "Great"
}
```

The customer data is there, but the data contains nothing about its relevant links.

A HATEOAS-based response would look like this:
```
{
   "name": "Great",
   "links": [ {
      "rel": "self",
      "href": "http://localhost:8080/customer/1"
   } ]
}
```

This response not only has the person's name, but includes the self-linking URL where that person is located.

rel means relationship. In this case, it's a self-referencing hyperlink.

More complex systems might include other relationships. For example, an order might have a "rel":"customer" relationship, linking the order to its customer.
href is a complete URL that uniquely defines the resource.

Note : XML is also accepted as a standard response format

Spring Boot Actuator

Spring Boot ships with a module called actuator which enables things like metrics and statistics about the application.

For example, we can collect logs, view metrics, perform thread dumps, show environment variables, understand garbage collection, and show what beans are configured in the BeanFactory.

We can expose this information via HTTP, JMX, or we can even log in directly to the process via SSH.

Enable actuator

To start using the existing actuators in Boot – we'll just need to add the spring-boot-actuator dependency to the pom:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
  <version>1.4.2.RELEASE</version>
</dependency>
```

Customizing the management server port

Exposing management endpoints using the default HTTP port is a sensible choice for cloud based deployments.

However, we can expose a separate management HTTP port.

The management.port property can be used to change the HTTP port.

management.port=8081

Security

Most of the times, the details exposed via the endpoints are sensitive and requires authorized user to view the details. We can add spring security to secure your endpoints.

Just add the dependency, when the spring security files are available in the classpath, it will be automatically configured.

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

application.properties

```
management.security.enabled=true
security.basic.enabled=true
security.user.name=admin
security.user.password=admin
```

Actuator Endpoint

Provides a hypermedia-based "discovery page" for the other endpoints. Requires Spring HATEOAS to be on the classpath.

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

Actuator Endpoints

Here are some of the most common endpoints Boot provides out of the box:

/health – Shows application health information (a simple 'status' when accessed over an unauthenticated connection or full message details when authenticated). It is not sensitive by default.

/info – Displays arbitrary application info. Not sensitive by default.

/metrics – Shows 'metrics' information for the current application. It is also sensitive by default.

/trace – Displays trace information (by default the last few HTTP requests). We can find the full list of existing endpoints over on the official docs.

Metrics Endpoint

The metrics endpoint is one of the more important endpoints as it gathers and publishes information about OS, JVM and Application level metrics; out of the box, we get things like:

memory, heap, processors, threads, classes loaded, classes unloaded, thread pools along with some HTTP metrics as well.

Exposing Endpoints

Since Endpoints may contain sensitive information, careful consideration should be given about when to expose them.

Property	Default
management.endpoints.jmx.exposure.exclude	
management.endpoints.jmx.exposure.include	*
management.endpoints.web.exposure.exclude	
management.endpoints.web.exposure.include	info, health

For example, to stop exposing all endpoints over JMX and only expose the health and info endpoints, use the following property:

management.endpoints.jmx.exposure.include=health,info

* can be used to select all endpoints. For example, to expose everything over HTTP except the env and beans endpoints, use the following properties:

management.endpoints.web.exposure.include=*
management.endpoints.web.exposure.exclude=env,beans

* has a special meaning in YAML, so be sure to add quotes if you want to include (or exclude) all endpoints, as shown in the following example:

```
management:
        endpoints:
                web:
                        exposure:
                                include: "*"
```

Spring Boot Graceful shutdown (/shutdown -> POST request)

1.Maven pom.xml:
```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

2.application.properties:
```
#No auth  protected
endpoints.shutdown.sensitive=false

#Enable shutdown endpoint
endpoints.shutdown.enabled=true
```

All endpoints are listed here:
3.Send a post method to shutdown the app:
```
curl -X POST localhost:port/shutdown
```

Spring Boot Graceful shutdown with security

```
endpoints.shutdown.sensitive=true
endpoints.shutdown.enabled=true

management.security.enabled=true
security.basic.enabled=true
security.user.name=admin
security.user.password=admin
```

maven dependencies to pom.xml to use default login feature of spring boot during unauthorised access.

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

application. Properties

```
management.security.enabled=true
security.user.name=admin
security.user.password=admin
```

Customizing the Management Server Port

We can set the management.port property to access the actuator endpoints

management.port=8081

Configuring Management-specific SSL

server.port=8443
server.ssl.enabled=true
server.ssl.key-store=classpath:store.jks
server.ssl.key-password=secret
management.server.port=8080
management.server.ssl.enabled=false

Both the **main server and the management** server can use SSL but with different key stores, as follows:

```
server.port=8443
server.ssl.enabled=true
server.ssl.key-store=classpath:main.jks
server.ssl.key-password=secret
management.server.port=8080
management.server.ssl.enabled=true
management.server.ssl.key-store=classpath:management.jks
management.server.ssl.key-password=secret
```

Customizing the Management Server Address

```
management.server.address=127.0.0.1
```

Gradle combines good parts of both maven and ant, and builds on top of them with **DSL** and other improvements.

Gradle does not use XML. Instead, it had its own DSL based on Groovy (one of JVM languages). As a result, Gradle build scripts tend to be much shorter and clearer than those written for Ant or Maven.