

Lecture 10: The Spring Framework

Olivier Liechti
AMT

heig-vd

Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

The Spring Framework

- **Spring**
 - From a book, to an open source project, to a company
 - Core framework + ecosystem
- **Core concepts**
 - Inversion of Control container (IoC) & Dependency Injection (DI)
 - Aspect Oriented Programming (AOP)



Introduction

The Spring Framework

- **When was it developed?**

- The Spring Framework was released in 2003.
- It was developed by Rod Johnson and presented in the book “Expert One-on-One J2EE Design and Development”.
- The framework has quickly become very popular and has expanded a lot since its inception (also through “acquisitions” of open source projects)

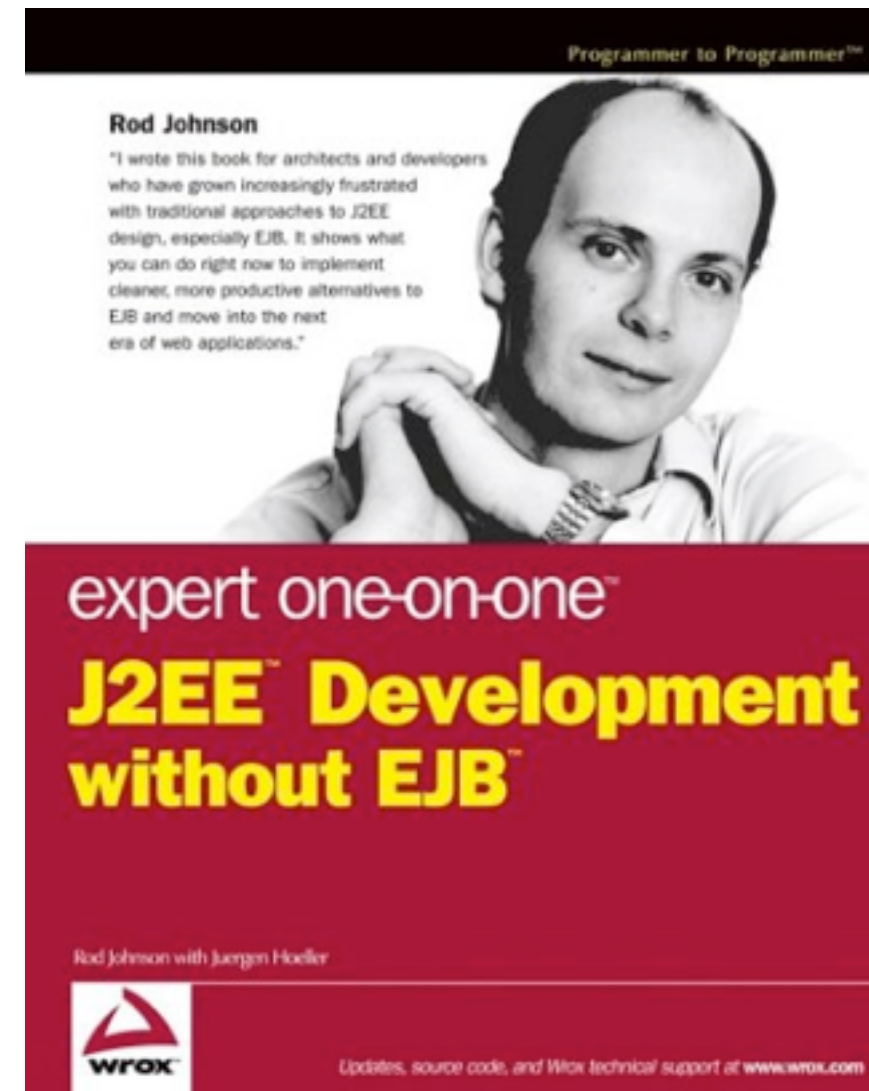
- **Why was it developed?**

- The Spring Framework was developed at the time of J2EE and EJB 2.
- At the time, using Enterprise Java Beans was rather “painful”.
- The Spring Framework proposed a lightweight approach, which was appropriate in many situations (for which J2EE was overkill).

Rod Johnson

heig-vd

Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

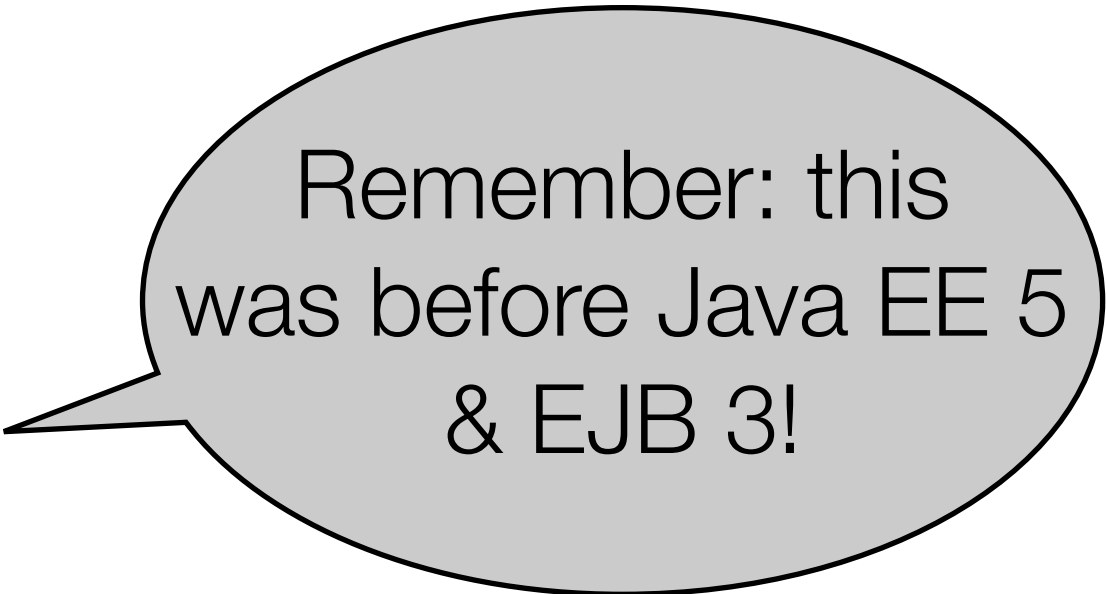


“What Do We Really Want from EJB”

- **Declarative Transaction Management**
- Remoting (RMI)
- Clustering
- Thread Management
- EJB Instance Pooling
- Resource Pooling
- Security
- Business Object Management

“What Don’t We Want from EJB?”

- **The Monolithic, Distinct Container Problem**
- Inelegance and the Proliferation of Classes
- Deployment Descriptor Hell
- Class Loader Hell
- **Testing**
- EJB Overdose
- **Complex Programming Model**
- Simple Things Can Be Hard
- Is the Goal of Enabling Developers to Ignore the Complexity of Enterprise Applications Event Desirable?
- Loss of Productivity
- **Portability Problems**



Remember: this
was before Java EE 5
& EJB 3!

Spring: Core Framework vs. Ecosystem

- **The core Spring framework**

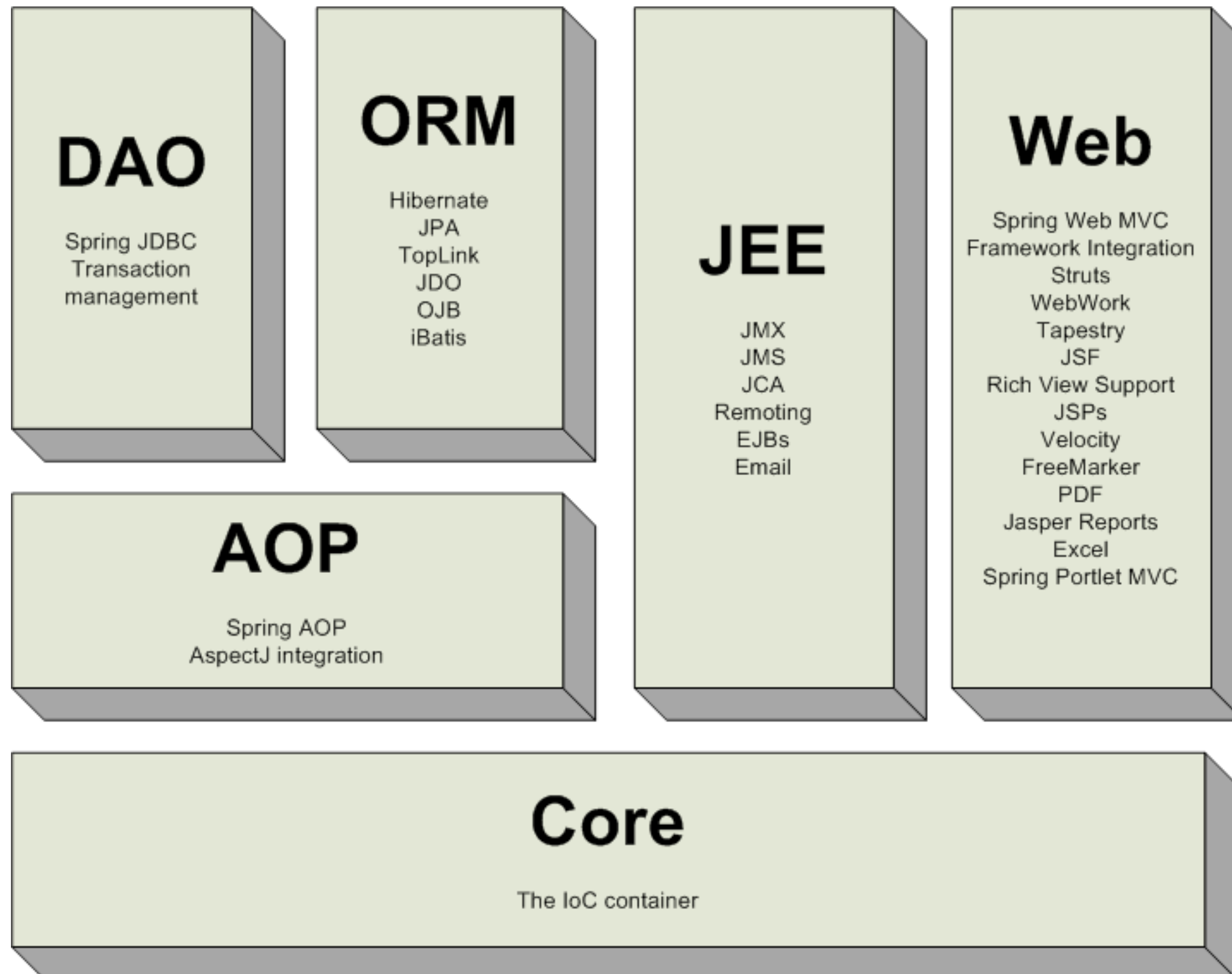
- provides a solution for building elegant object-oriented systems;
- supports inversion of control and aspect-oriented programming as key underlying mechanisms.

- **The Spring ecosystem (now Spring.io)**

- is a set of modules and frameworks built on top of the core framework;
- provides solution in many different domains: data access, web tier, messaging, security, etc.

So... then I can use only the core framework (which is lightweight)

Spring Framework

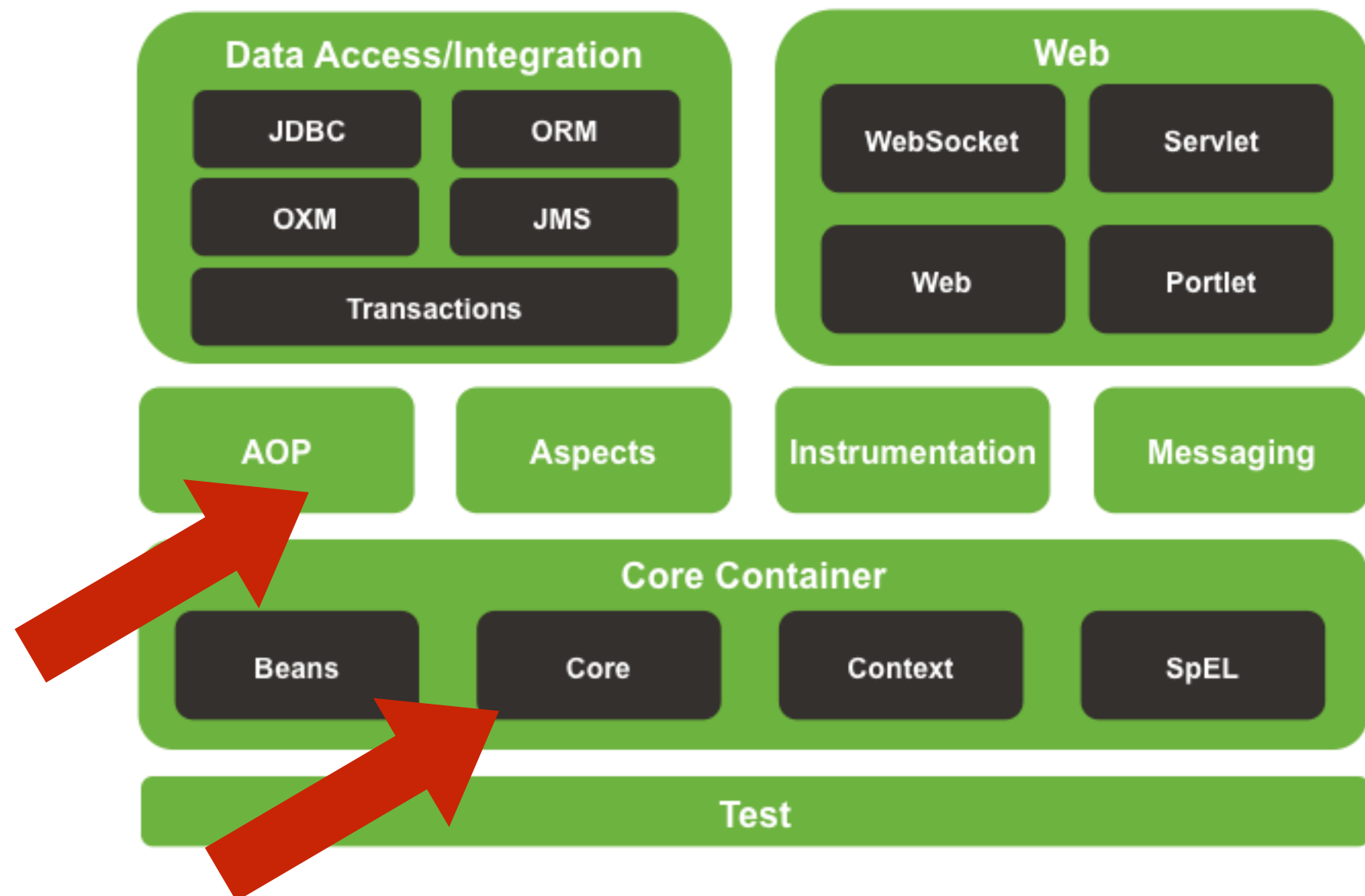


Spring Framework

- Spring enables you to build applications from POJOs and to apply enterprise services non-invasively.
- This capability applies to the Java SE programming model and to full and partial Java EE.

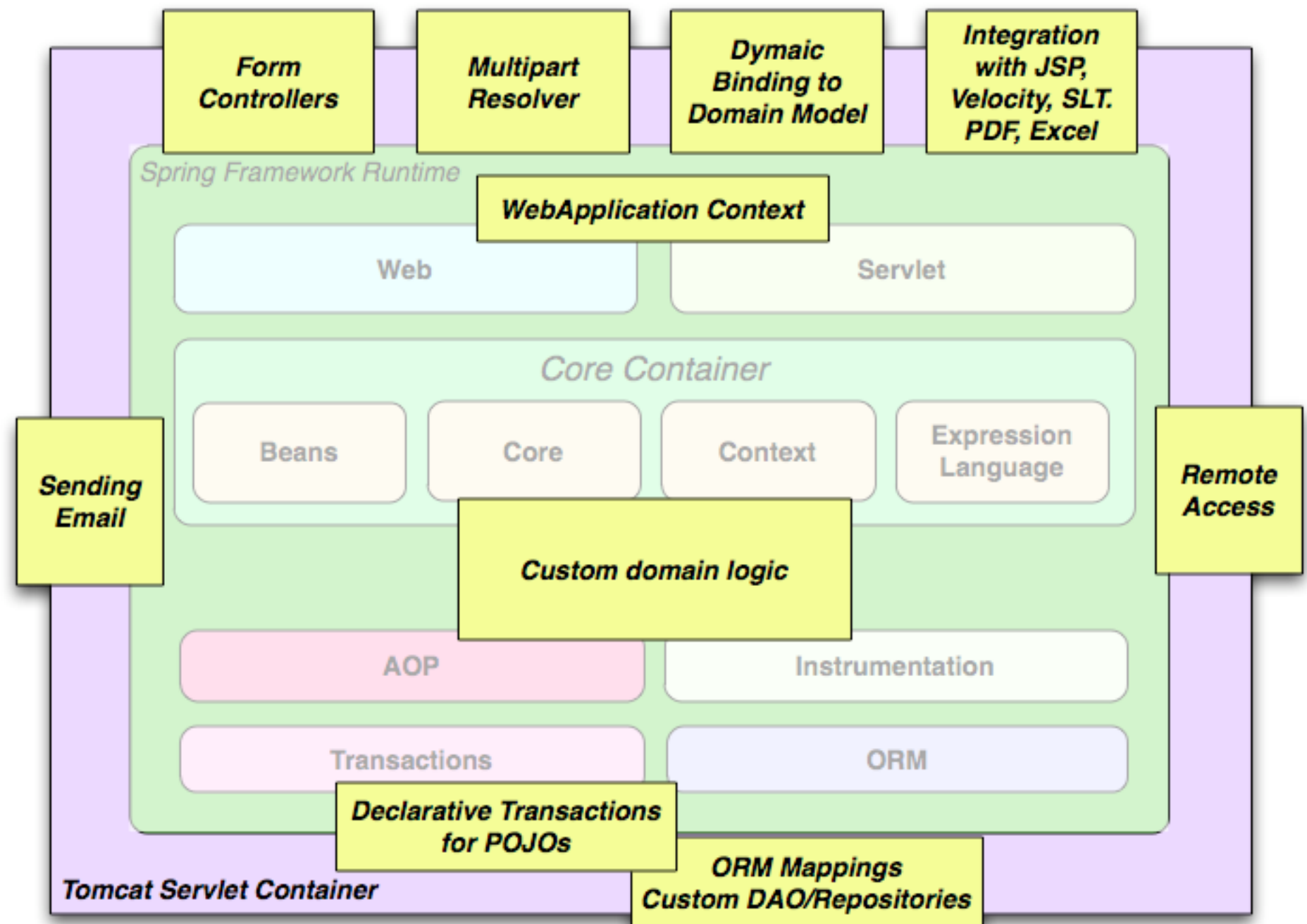


Spring Framework Runtime



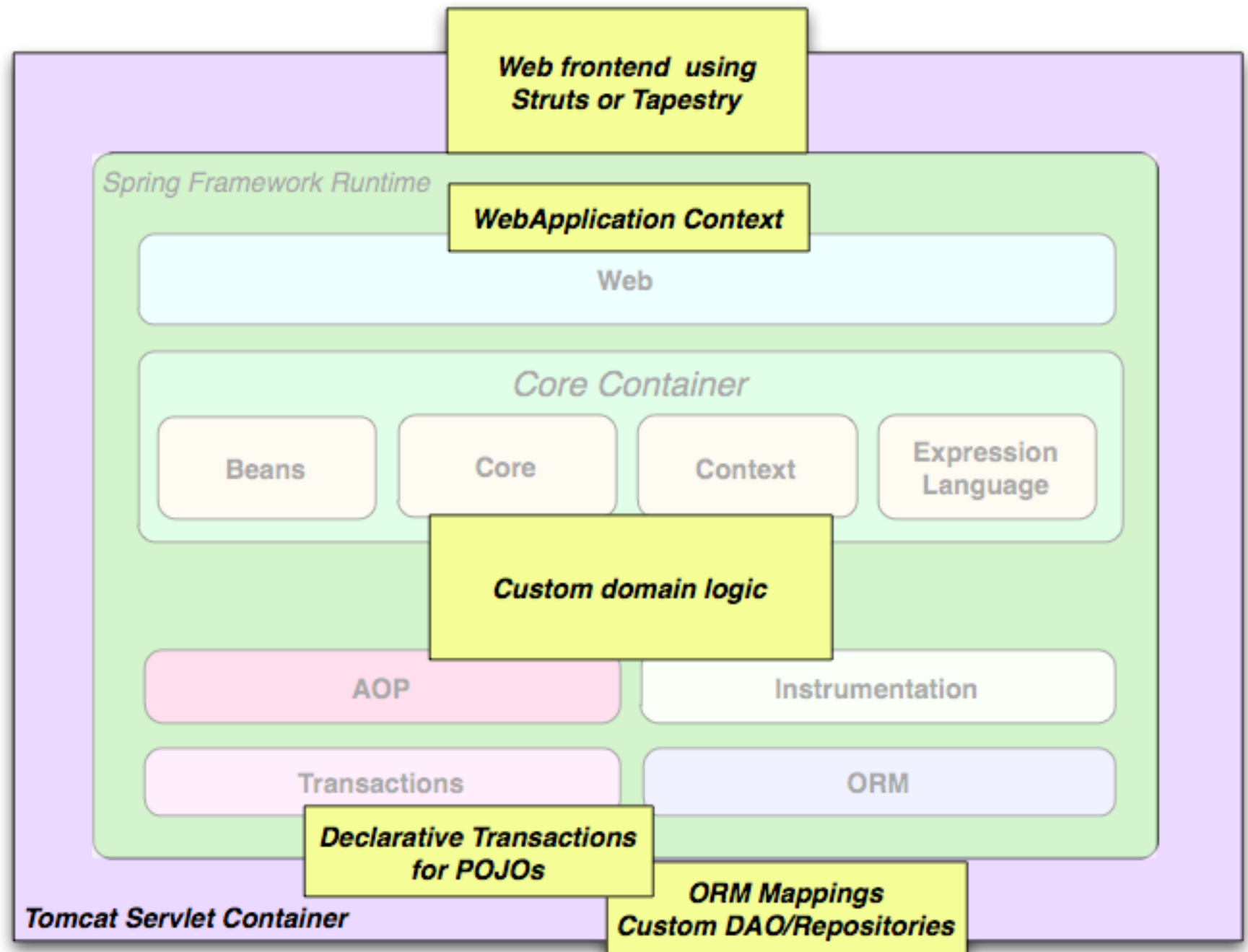
Spring Framework

- Using the full framework, also in the web tier.



Spring Framework

- Integrating with third-party web frameworks.



Spring.io Projects



Spring.io Projects



SPRING INTEGRATION

Supports the well-known
Enterprise Integration Patterns
via lightweight messaging and
declarative adapters.



SPRING BATCH

Simplifies and optimizes the work
of processing high-volume batch
operations.



SPRING SECURITY

Protects your application with
comprehensive and extensible
authentication and authorization
support.



SPRING HATEOAS

Simplifies creating REST
representations that follow the
HATEOAS principle.



SPRING SOCIAL

Easily connects your applications
with third-party APIs such as
Facebook, Twitter, LinkedIn, and
more.



SPRING AMQP

Applies core Spring concepts to
the development of AMQP-
based messaging solutions.

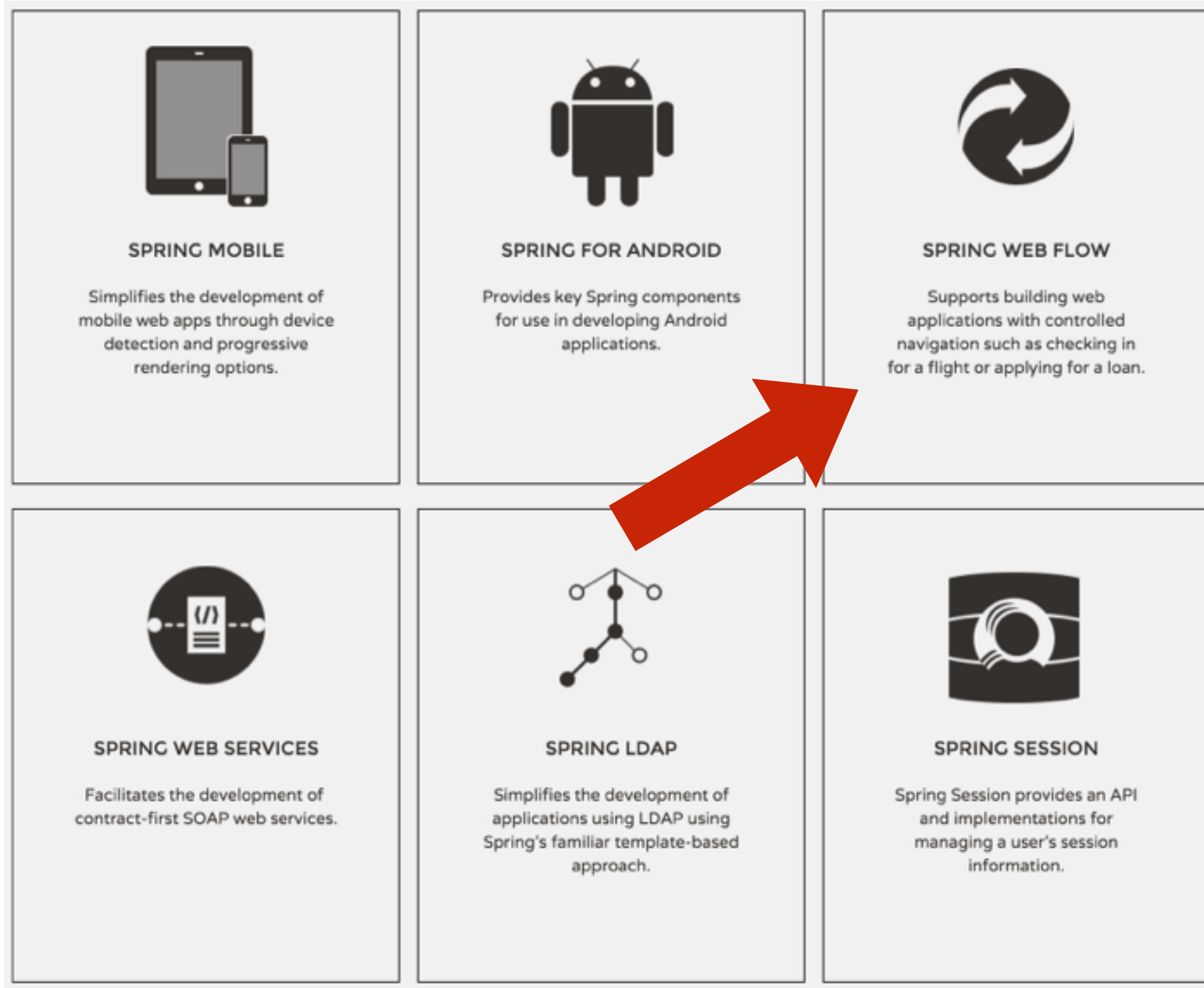
4

4

Spring.io Projects

heig-vd

Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud



Spring.io Projects

heig-vd

Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud



Getting Started Guides

If you're just getting to know Spring or tackling building something new, these are for you! All you need is 15-30 minutes, a JDK and a text editor.

[Try one of the Getting Started Guides >](#)



Tutorials

Designed to be completed in 2-3 hours, tutorials provide deeper, in-context explorations of enterprise application development topics.

[See the Tutorials >](#)



Reference Documentation

Looking for in-depth knowledge on a particular Spring project? Here you'll find quick access to javadoc APIs and reference documentations.

[Read the Reference Documentation >](#)



On top of Spring Boot...

heig-vd

Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

Greetings, Java Hipster!



JHipster is a
Yeoman generator,



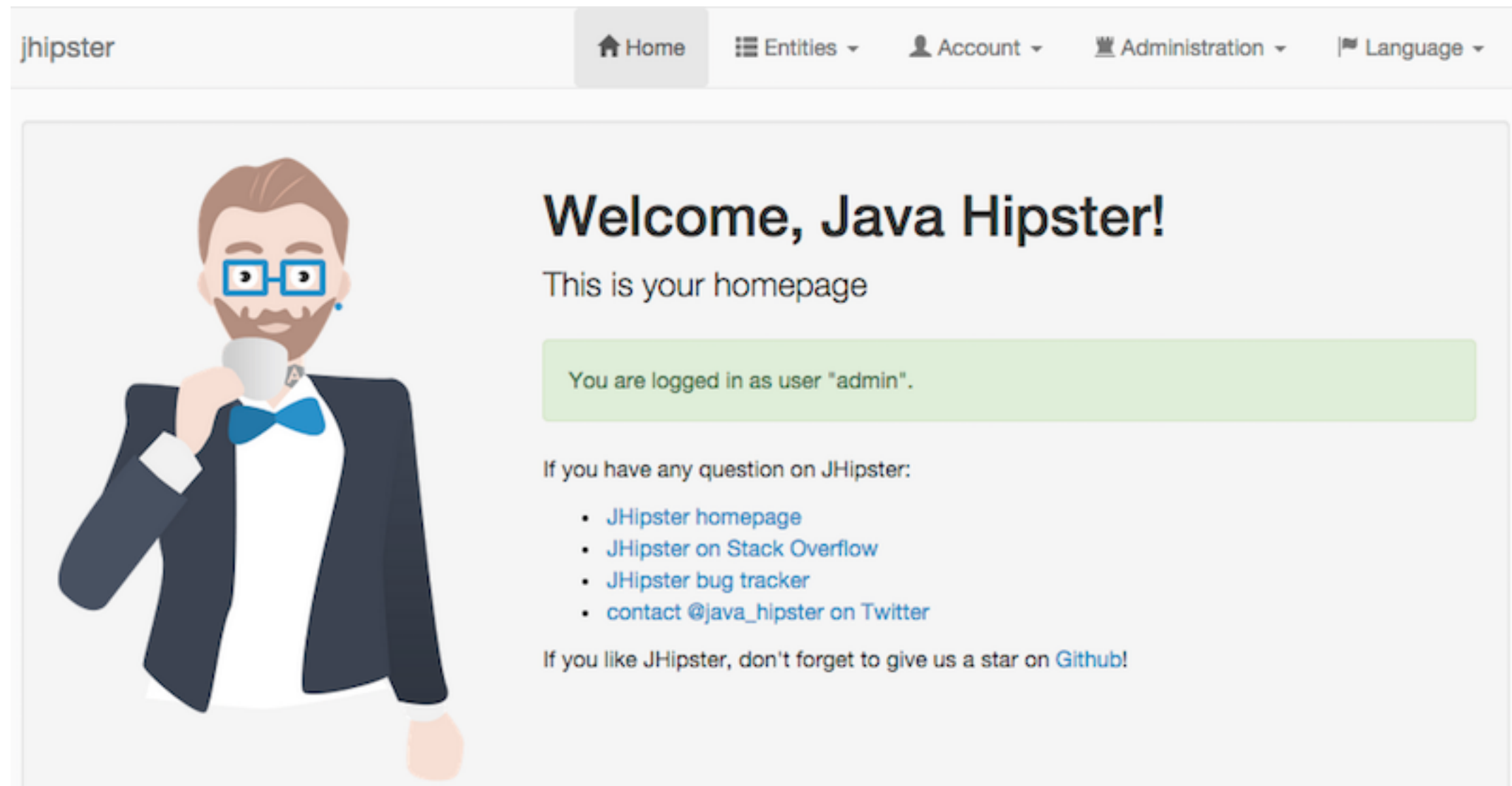
used to create a
Spring Boot + AngularJS
project.



On top of Spring Boot...

heig-vd

Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud



On top of Spring Boot...

heig-vd

Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

jhipster

Home Entities Account Administration Language

Logs

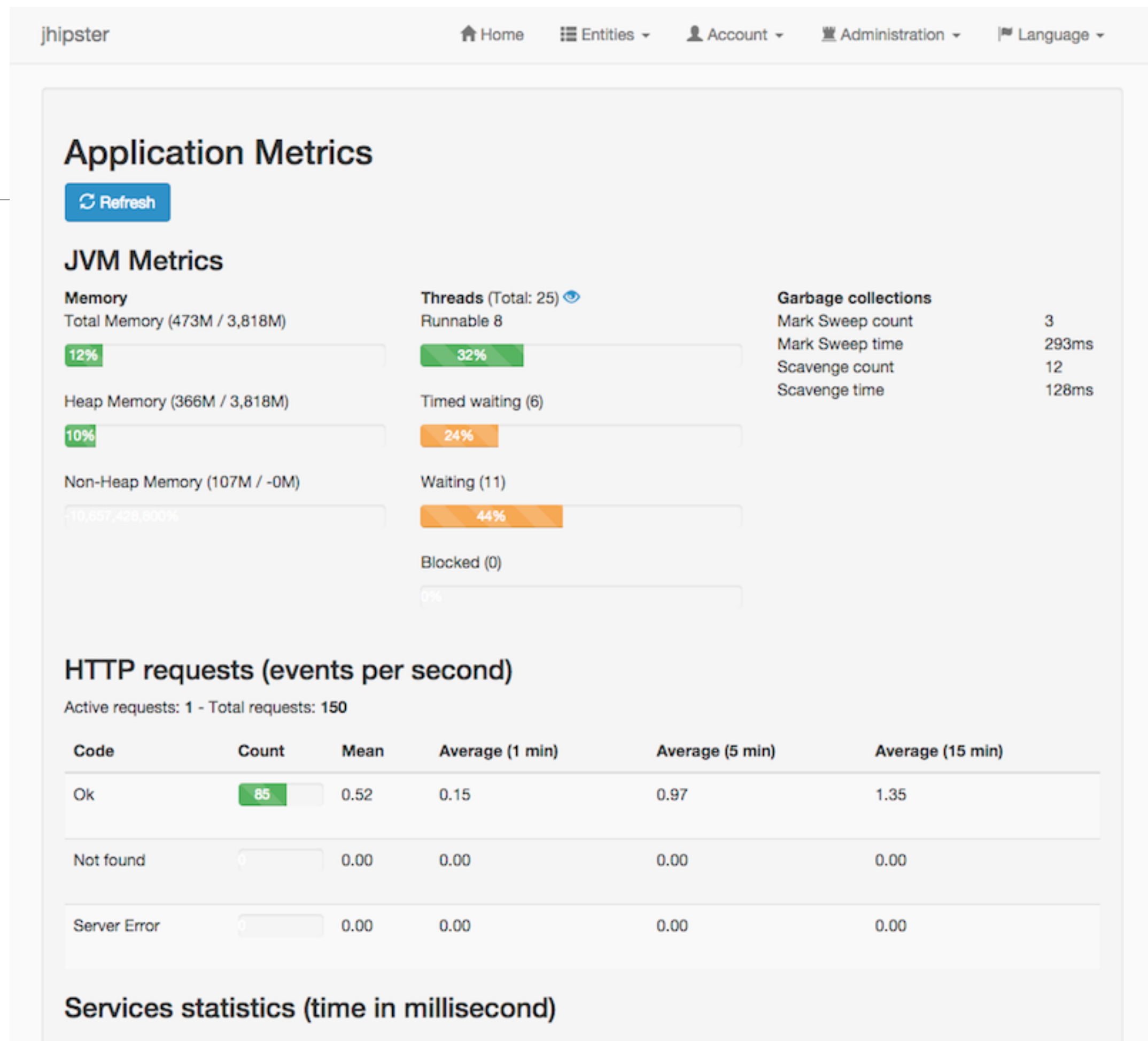
There are 1063 loggers.

Filter

springframework.security

Name	Level
org.springframework.security	TRACE DEBUG INFO WARN ERROR
org.springframework.security.access	TRACE DEBUG INFO WARN ERROR
org.springframework.security.access.annotation	TRACE DEBUG INFO WARN ERROR
org.springframework.security.access.annotation.Jsr250MethodSecurityMetadataSource	TRACE DEBUG INFO WARN ERROR
org.springframework.security.access.expression	TRACE DEBUG INFO WARN ERROR
org.springframework.security.access.expression.DenyAllPermissionEvaluator	TRACE DEBUG INFO WARN ERROR
org.springframework.security.access.expression.method	TRACE DEBUG INFO WARN ERROR
org.springframework.security.access.expression.method.DefaultMethodSecurityExpressionHandler	TRACE DEBUG INFO WARN ERROR
org.springframework.security.access.expression.method.ExpressionBasedPostInvocationAdvice	TRACE DEBUG INFO WARN ERROR
org.springframework.security.access.intercept	TRACE DEBUG INFO WARN ERROR
org.springframework.security.access.intercept.AsyncInvocationInterceptor	TRACE DEBUG INFO WARN ERROR

On



et de Gestion

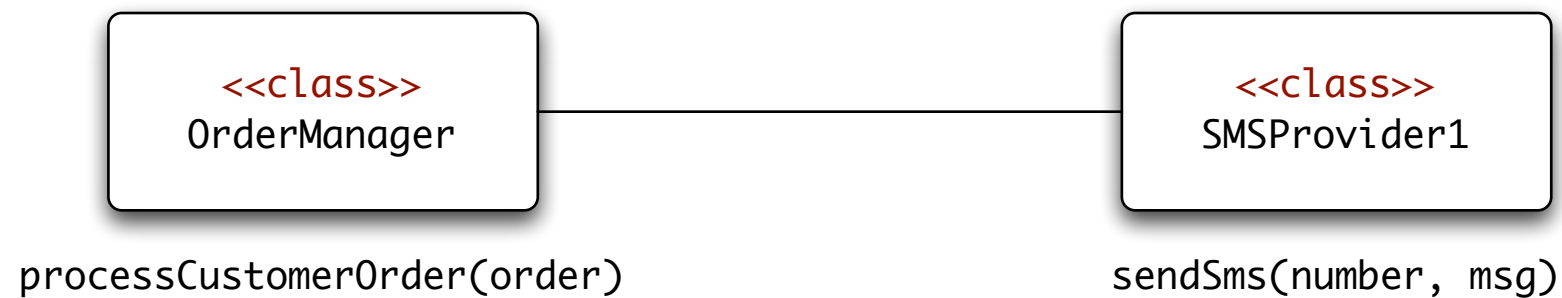


Dependency Injection

Example

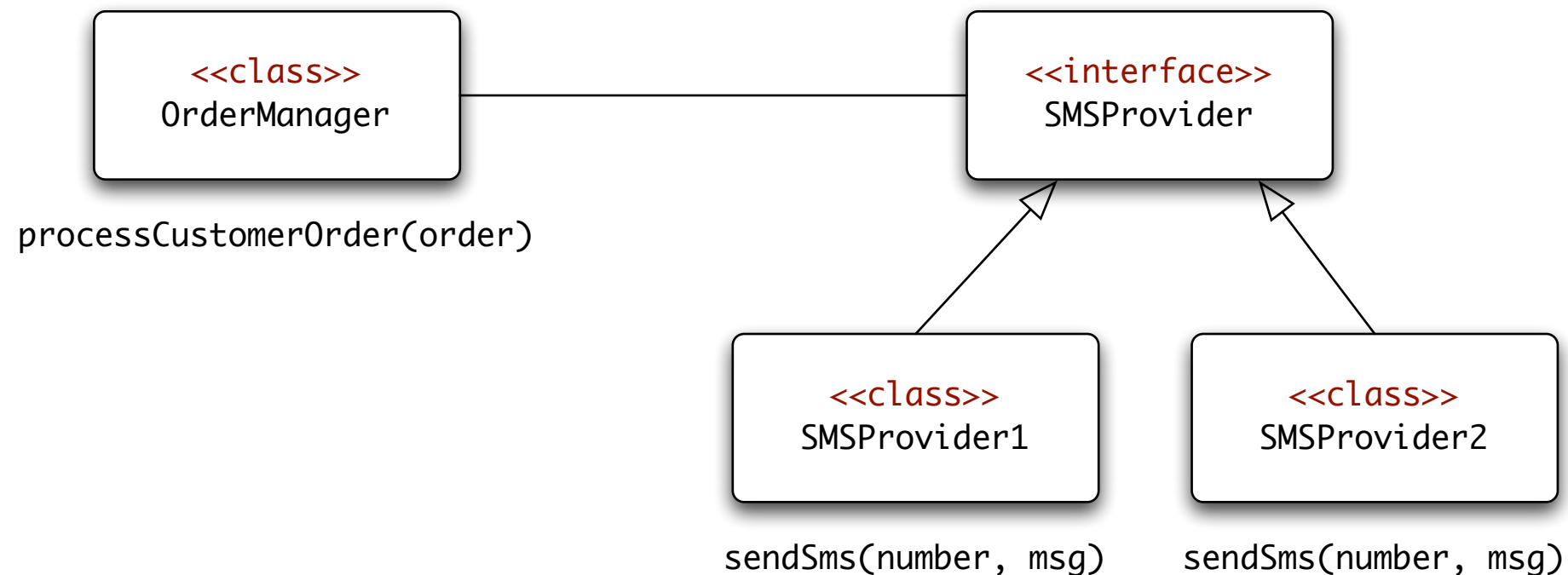
- You develop an **online web store** and design an **order processing** workflow.
- During the order processing, you want to **send a SMS** to the customer.
- You can use “SMS Providers” also called “**SMS Brokers**”, who make it easy to send SMS through an HTTP interface.
- Question: how do you **design** the underlying system?

Step 1: share responsibilities



Ok... but what if want to change
the SMS provider?

Step 2: program against interfaces



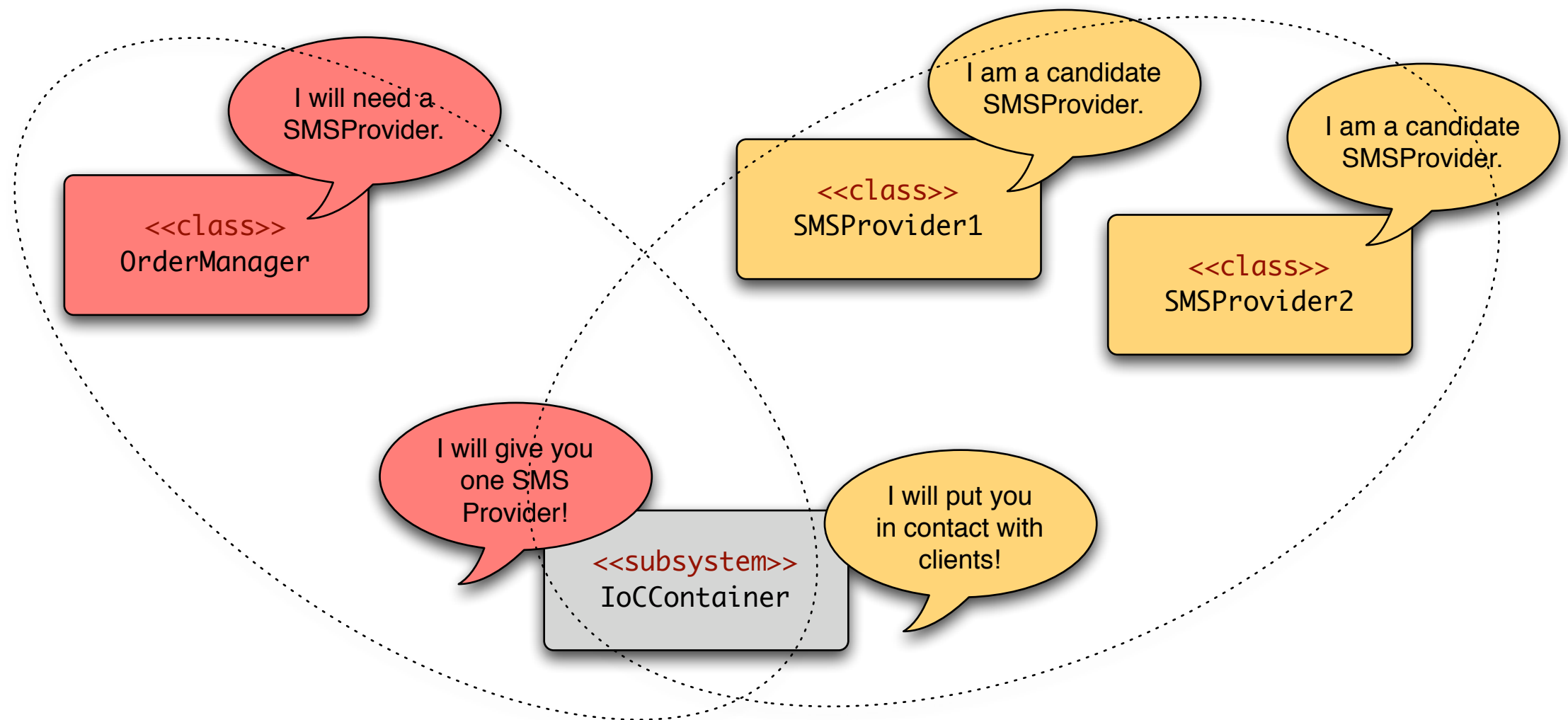
Ok... but how does the OrderManager get a reference to the proper SMS provider?

Step 2: program against interfaces

```
public class OrderManager {  
  
    public void processOrder(Order o) {  
        ...  
        SMSProvider p1 = new SMSProvider1();  
        p1.sendSms(o.getCustPhoneNumber(), o.getTotal(), message);  
        ...  
    }  
  
}
```

Not ideal... if we need to change the SMSProvider implementation, we need to go back to code.

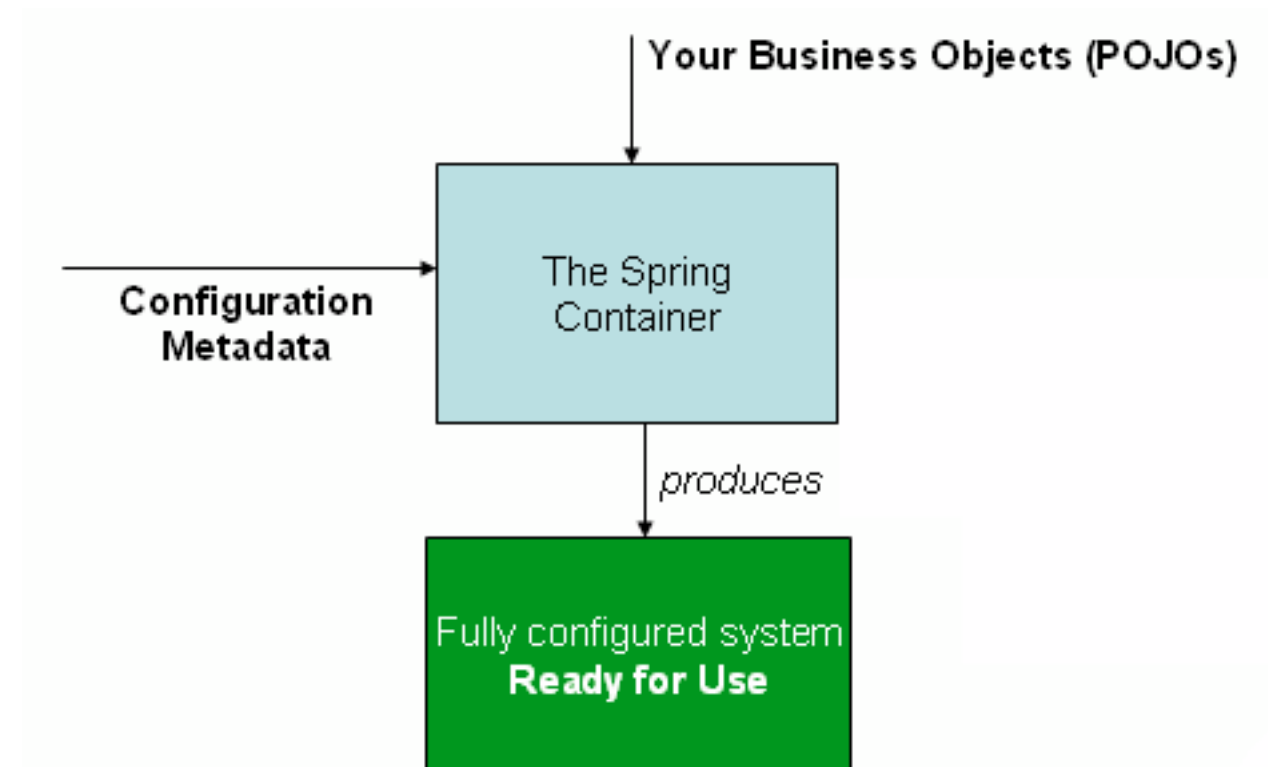
Step 3: dependency injection with IoC



Oh... so the IoC container simply is a mediator that wires OO components together?

Dependency Injection with Spring

- The IoC Container manages “**Spring Beans**”, which are standard classes (POJOs).
- The IoC knows about all “**components**” or “**services**” defined in the system.
- The BeanFactory interface provides methods for interacting with the IoC Container.
- `ApplicationContext` extends `BeanFactory` and adds lots of “magic” behind the scenes (e.g. lifecycle management). **Use this!**



<http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/beans.html>

BeanFactory vs ApplicationContext

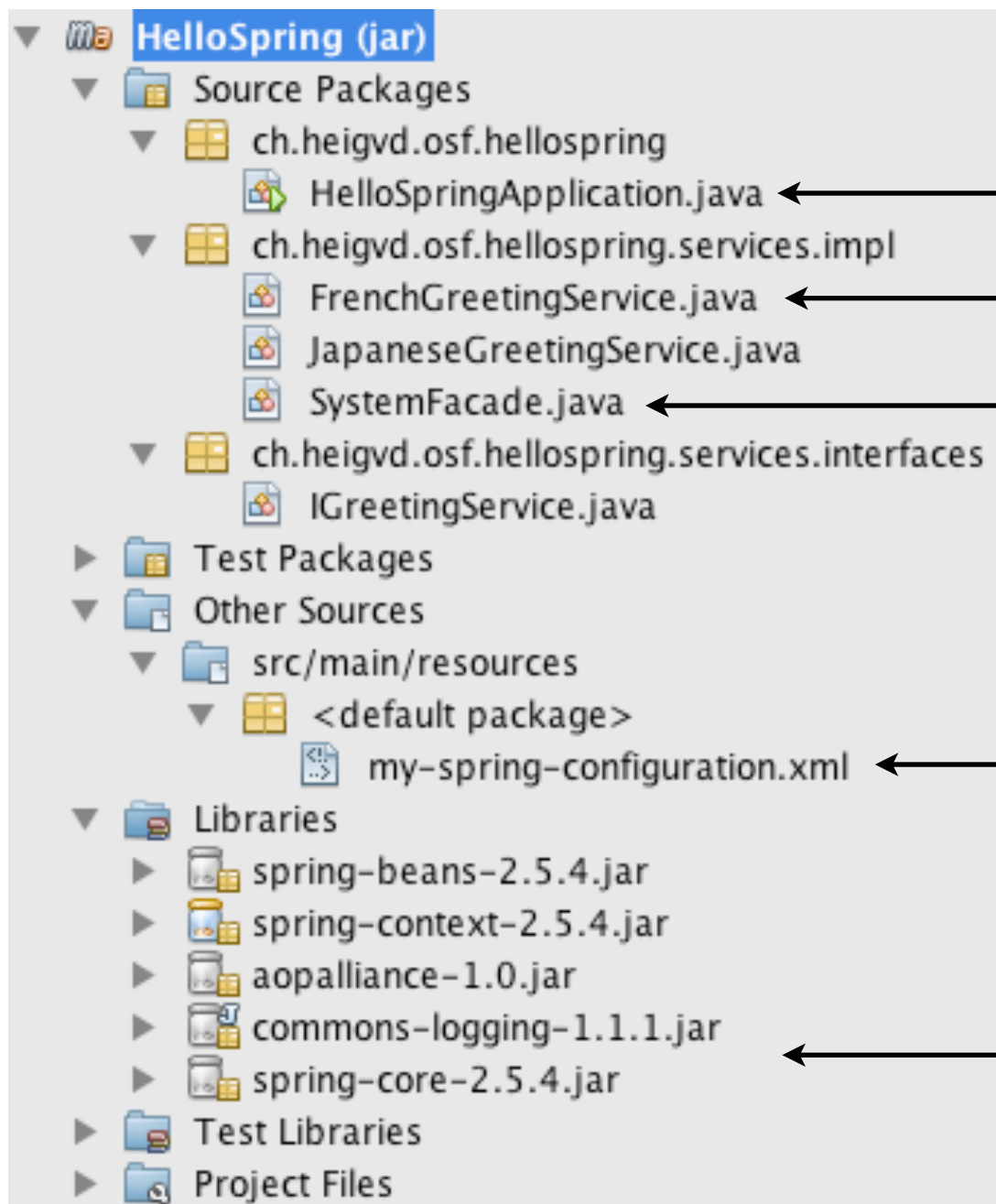
BeanFactory or ApplicationContext?

Users are sometimes unsure whether a BeanFactory or an ApplicationContext is best suited for use in a particular situation. A BeanFactory pretty much just instantiates and configures beans. An ApplicationContext also does that, and it provides the supporting **infrastructure** to enable lots of enterprise-specific features such as **transactions** and **AOP**.

In short, favor the use of an ApplicationContext.

From the doc: <http://static.springsource.org/spring/docs/2.5.6/reference/beans.html>

Sample project (Java SE)



the entry point that creates the IoC container

2 spring beans implementing the same interface

1 spring bean that needs a
IGreetingService implementation

the configuration file that declares
and wires the spring beans

the core spring libraries

Sample project: the entry point

```
package ch.heigvd.osf.hellospring;

import ch.heigvd.osf.hellospring.services.impl.SystemFacade;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class HelloSpringApplication {

    private Log log = LogFactory.getLog(HelloSpringApplication.class);

    public void startApplication() {
        log.info("Starting HelloSpring application... Welcome!");
        ApplicationContext context = new ClassPathXmlApplicationContext(new String[]{"my-spring-configuration.xml"});

        log.info("Getting a reference to the 'mySystem' bean (facade)");
        SystemFacade facade = (SystemFacade)context.getBean("mySystem");
        log.info("Invoking doStuff method on the facade; processing will be delegated to wired IGreetingService bean");
        facade.doStuff();

        log.info("Done.");
    }

    public static void main(String[] args) {
        new HelloSpringApplication().startApplication();
    }
}
```

This is where we launch the IoC container

The Spring Beans are accessible through the IoC container

Sample project: the configuration file

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       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-2.5.xsd">
```

```
  <bean id="aFrenchService" class="ch.heigvd.osf.hellospring.services.impl.FrenchGreetingService">
  </bean>
```

```
  <bean id="aJapaneseService" class="ch.heigvd.osf.hellospring.services.impl.JapaneseGreetingService">
  </bean>
```

```
  <bean id="mySystem" class="ch.heigvd.osf.hellospring.services.impl.SystemFacade">
    <property name="greetingService" ref="aFrenchService"/>
  </bean>
```

```
</beans>
```

we use this name when we do the
lookup in the Java code

```
SystemFacade facade =
(SystemFacade)context.getBean("mySystem");
```

this is a “dependency injection”: we
provide a IGreetingService
implementation to the mySystem bean

Sample project: bean implementation

```
package ch.heigvd.osf.hellospring.services.impl;
```

```
import ch.heigvd.osf.hellospring.services.interfaces.IGreetingService;
```

```
import org.apache.commons.logging.Log;
```

```
import org.apache.commons.logging.LogFactory;
```

```
public class SystemFacade {
```

```
    private Log log = LogFactory.getLog(IGreetingService.class);
```

```
    private IGreetingService greetingService;
```

```
    /**
```

```
     * Setter for the greetingService property. This is what makes dependency
```

```
     * injection possible: the IoC container will call this method and pass
```

```
     * an instance of the bean defined in the XML configuratin file
```

```
     * @param greetingService a spring bean defined in the XML configuratinf file
```

```
     */
```

```
    public void setGreetingService(IGreetingService greetingService) {
```

```
        this.greetingService = greetingService;
```

```
    }
```

```
    /**
```

```
     * This method will be called from the applicatin. It does not do much, except
```

```
     * for delegating processing to the wired beans (thank you dependency injection)
```

```
     */
```

```
    public void doStuff() {
```

```
        log.info("System facade invoked... delegating work to the wired greeting service");
```

```
        log.info("Wired greeting service says: " + greetingService.greet());
```

```
        log.info("System facade done with processing.");
```

```
    }
```

```
}
```

This method enables “setter” based
dependency injection



Spring IoC Configuration

- Different ways to declare Spring Beans, both in XML and through annotations.
- These different ways have been added over time, so they are not available in all versions of the Spring Framework.
- Essentially, you need to declare your components, how they are instantiated and how they depend on each other.
- For some frameworks, you have lots of components. XML schemas make configuration less verbose.

`<bean></bean>`

XML with
DTD

`<mvc:view-controller>`

XML with
schemas

bean wiring

Annotation
based

bean declaration

Java based

XML Schema Based Configuration

```
<!-- creates a java.util.List instance with values loaded from the supplied 'sourceList' -->
<bean id="emails" class="org.springframework.beans.factory.config.ListFactoryBean">
  <property name="sourceList">
    <list>
      <value>pechorin@hero.org</value>
      <value>raskolnikov@slums.org</value>
      <value>stavrogin@gov.org</value>
      <value>porfiry@gov.org</value>
    </list>
  </property>
</bean>
```



before

```
<!-- creates a java.util.List instance with the supplied values -->
<util:list id="emails">
  <value>pechorin@hero.org</value>
  <value>raskolnikov@slums.org</value>
  <value>stavrogin@gov.org</value>
  <value>porfiry@gov.org</value>
</util:list>
```



after

XML Schema Based Configuration

```
<bean id="dataSource" class="org.springframework.jndi.JndiObjectFactoryBean">  
  <property name="jndiName" value="jdbc/MyDataSource"/>  
</bean>
```

```
<bean id="userDao" class="com.foo.JdbcUserDao">  
  <!-- Spring will do the cast automatically (as usual) -->  
  <property name="dataSource" ref="dataSource"/>  
</bean>
```



before

```
<jee:jndi-lookup id="dataSource" jndi-name="jdbc/MyDataSource"/>
```

```
<bean id="userDao" class="com.foo.JdbcUserDao">  
  <!-- Spring will do the cast automatically (as usual) -->  
  <property name="dataSource" ref="dataSource"/>  
</bean>
```



after

XML Schema Based Configuration

```
<bean id="simple"  
      class="org.springframework.ejb.access.LocalStatelessSessionProxyFactoryBean">  
  <property name="jndiName" value="ejb/RentalServiceBean"/>  
  <property name="businessInterface" value="com.foo.service.RentalService"/>  
</bean>
```

before

```
<jee:local-slsb id="simpleSlsb" jndi-name="ejb/RentalServiceBean"  
               business-interface="com.foo.service.RentalService"/>
```

after



Aspect Oriented Programming

- **Aspect Oriented Programming**
 - Separation of concerns, cross-cutting concerns
 - Terminology
 - AOP frameworks
- **Aspect Oriented Programming & Java EE**
 - Declarative enterprise services
 - EJBs and interceptors
- **AOP with Spring**
 - Spring AOP vs. Spring with AspectJ
 - XML vs. Annotations

Aspect Oriented Programming (AOP)

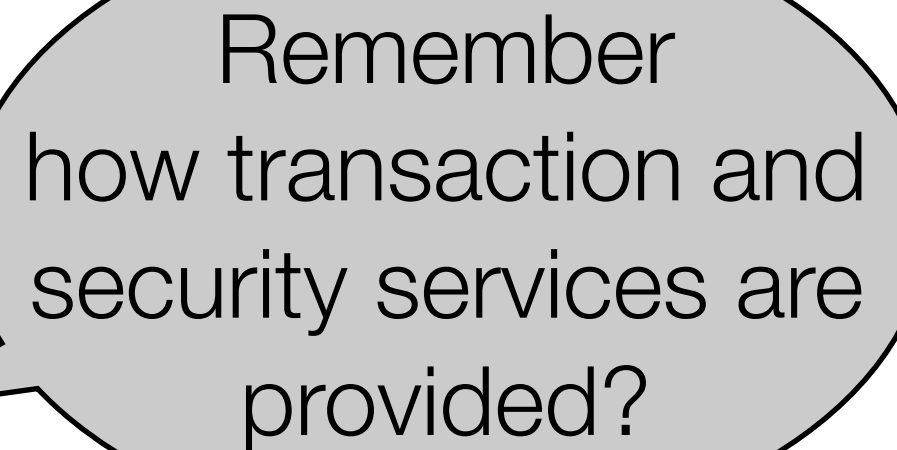
- In all applications, there are “things” that need to be done over and over and that are **orthogonal** to **business logic**.
- Examples:
 - Logging and auditing
 - Security checks (authorization)
 - Transaction management
- In traditional object-oriented design, the common approach is to implement the pure business logic and these orthogonal functions **at the same place** (in class methods).

Separation of concerns: business logic vs. other “aspects”

- AspectJ created at Xerox PARC in 2001 (Gregor Kiczales)
- Several other frameworks and projects have been developed (e.g. AspectWerkz), for different languages.
- The Spring Framework makes it possible to use AOP concepts and relies itself on AOP for some of its features.
- As always with Spring, there are lots of different ways to use AOP (pure Spring vs. AspectJ integration, XML vs. annotations, etc.)



Remember the
notion of “container”



Remember
how transaction and
security services are
provided?



Remember EJB3
interceptors?

Aspect Oriented Programming (AOP)

- Where is my business logic? It's hard to find... What do I have to bother with all these infrastructure concerns?
- How can I get a global view for security management in my application?
- What if I need to change the way I do the auditing? I will have to go in every single method...

- *What a nightmare!!*



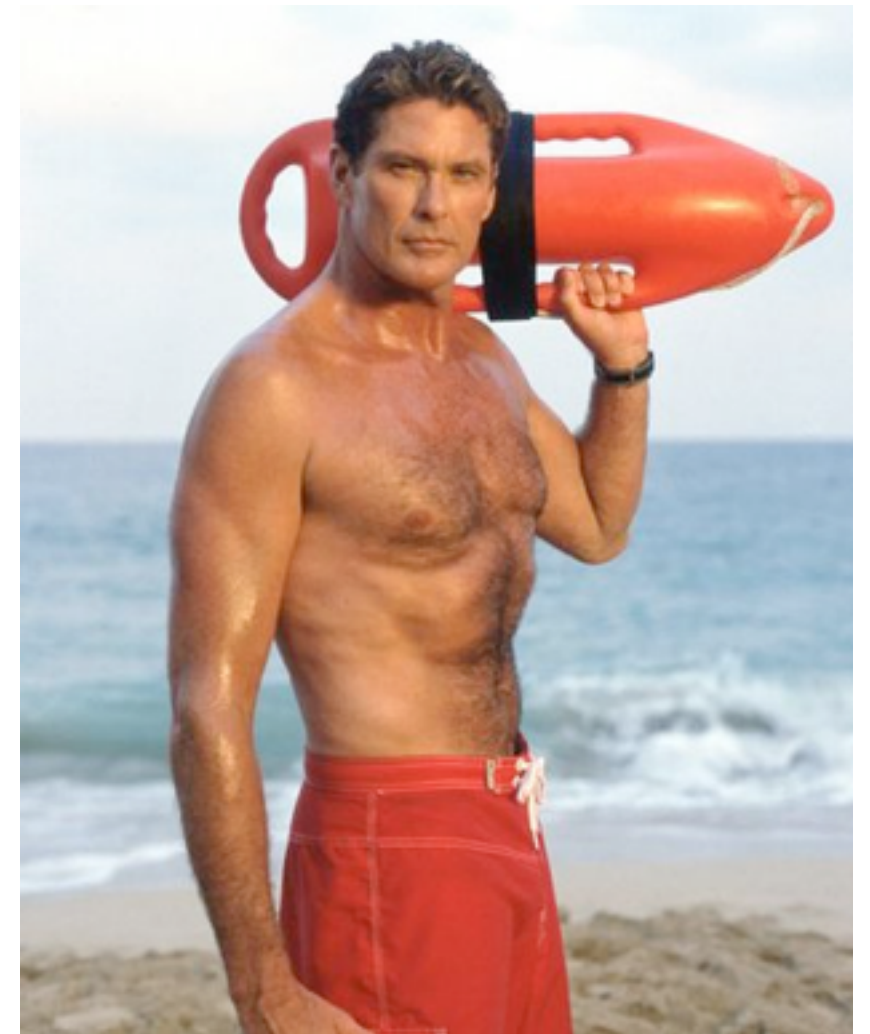
```
<<class>>  
ProductManager
```

```
public void addProduct(Product p) {  
    // check if the user is authenticated and authorized  
    ...  
    // start transaction  
    ...  
    // finally, some business logic  
    ...  
    // commit transaction  
    ...  
    // leave a trace in the audit trail  
    ...  
}
```

```
public void removeProduct(Product p) {  
    // check if the user is authenticated and authorized  
    ...  
    // start transaction  
    ...  
    // finally, some business logic  
    ...  
    // commit transaction  
    ...  
    // leave a trace in the audit trail  
    ...  
}
```

AOP to the rescue

- **AOP supports the separation of concerns.** In other words, it gives a way to split the implementation of the business logic from the implementation of system-level functions.
- **Terminology**
 - An **aspect** or **cross-cutting concern** refers to **something** that needs to be done throughout the application code. Security, logging and transaction management are examples of cross-cutting concerns.
 - An **advice** is the **orthogonal logic** that is executed when a certain join point is executed (advice can be executed **before**, **after** or **around** the join point).
 - A **pointcut** is an **expression** used to define a set of join points. With a pointcut, one can specify which join points (i.e. which methods)
 - A **join point** defines **when** the orthogonal logic could be executed. For instance, the execution of a `processOrder()` **method** is a join point.



AOP to the rescue

- **AOP supports the separation of concerns.** In other words, it gives a way to split the implementation of the business logic from the implementation of system-level functions.

- **Terminology**

- An **aspect** or **cross-cutting concern** refers to **something** that needs to be done throughout the application code, e.g. management.

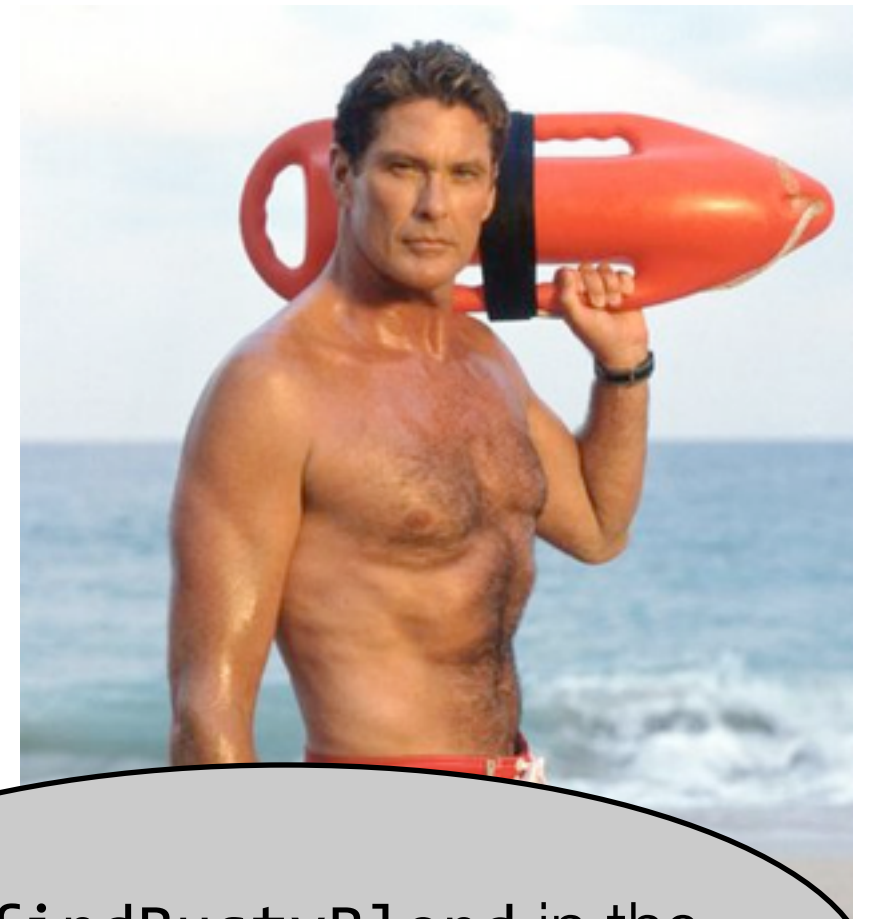
- An **advice** is a piece of code that is executed at a certain join point, **before, after** or **around** the execution of the target code.

All methods that start with
“find” in the `ch.heigvd.osf`
package

- A **pointcut** is an **expression** used to define a set of join points. With a pointcut, one can specify which join points (i.e. which methods)

- A **join point** defines **when** the orthogonal logic can be executed. For instance, the execution of a `processOrder()` **method** is a join point.

the `findBustyBlond` in the
`ch.heigvd.osf.BayWatch` class



Join Point

<<class>>

ch.heigvd.osf.service.OrderManager

```
public void processOrder(Order o) {} ←
public void processOrder(Order o, Params p) {} ←
public void processOrder(Order o, Helper h) {} ←
public void cancelOrder(Order o) {} ←
public void candelOrder(Order o, Params p) {} ←
public Stats getStatistics() {} ←
public List listLargeOrders() {} ←
```

These are all join points...

<<class>>

ch.heigvd.osf.service.ProductManager

```
public void createProduct(Product p) {} ←
public void updateProduct(Product p) {} ←
public void deleteProduct(Product p) {} ←
public void listAllProducts() {} ←
public List listExpensiveProducts() {} ←
```

...surprise, surprise,
these too!

Pointcut

Pointcuts can be declared with an annotation (or with XML...)

```
@PointCut(expression)
private void aNameForThisSetOfMethods {}
```

The **expression** is based on the AspectJ pointcut language. Here are some examples:

the execution of any public method:

```
execution(public * *(..))
```

the execution of any method with a name beginning with "set":

```
execution(* set*(..))
```

the execution of any method defined by the AccountService interface:

```
execution(* com.xyz.service.AccountService.*(..))
```

the execution of any method on a Spring bean named 'tradeService':

```
bean(tradeService)
```

the execution of any method on a Spring bean with a name matching the wildcard expression

```
bean(*Service)
```

`execution(modifiers-pattern? ret-type-pattern declaring-type-pattern? name-pattern(param-pattern) throws-pattern?)`

<http://static.springsource.org/spring/docs/2.5.6/reference/aop.html#aop-pointcuts>

How Can it Work?

- There are different ways to implement AOP.
- Remember that we want to “**combine**” two pieces of orthogonal code - located in two different artifacts (a “business” class and an “advice class”).
- One possibility is to use a **special compilation process**. This is called “**weaving**”, since the aspect code is weaved into the main business logic. As an alternative, it is possible to do the weaving as an **after-compilation** process. “Weaving” is what the AspectJ framework and toolset is doing.
- Another approach is to use proxies that are dynamically generated. This is what Spring is doing - so there is no special compilation process

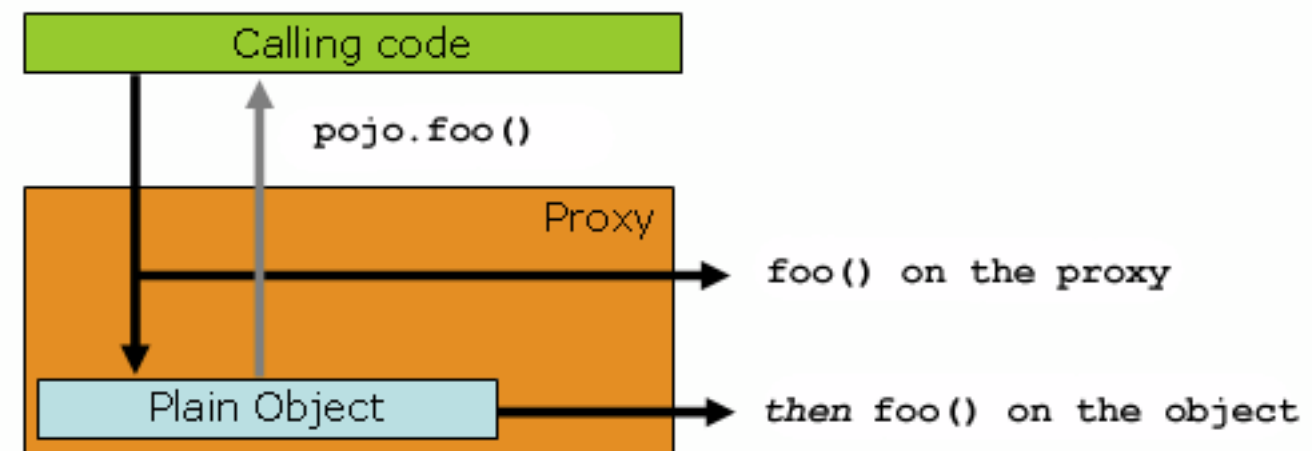
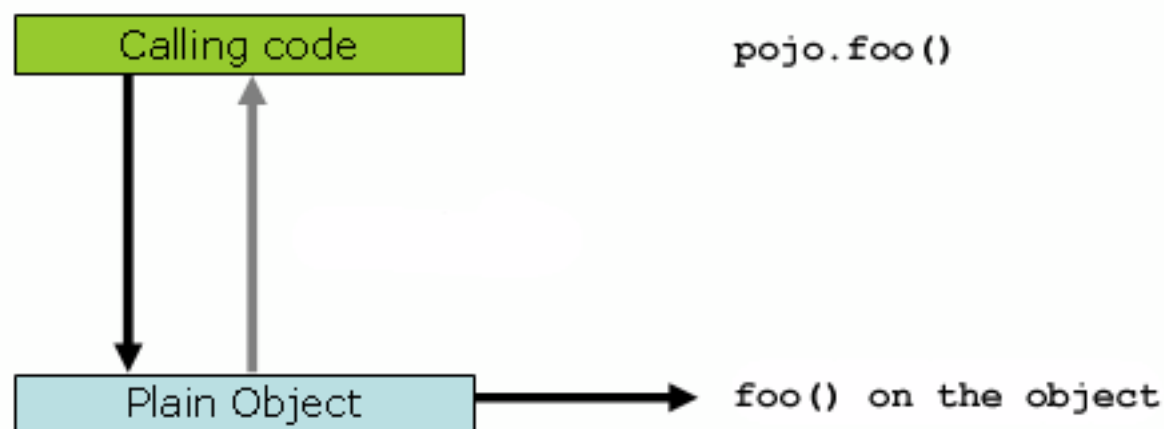
AOP in the Spring Framework

- AOP is used in the Spring Framework to:
 - provide **declarative** enterprise services, especially as a replacement for EJB declarative services. The most important such service is declarative **transaction management**
 - allow users to implement **custom aspects**, complementing their use of OOP with AOP

“If you are interested only in generic declarative services or other pre-packaged declarative middleware services such as pooling, you do not need to work directly with Spring AOP, and can skip most of this chapter.”

Spring AOP Capabilities and Goals (1)

- Spring AOP is implemented in **pure Java**
- There is **no need for a special compilation process**
- Spring AOP does not need to control the class loader hierarchy, and is thus **suitable for use in a Java EE web container or application server**



Spring AOP Capabilities and Goals (2)

- Spring AOP currently supports only **method execution join points** (advising the execution of methods on Spring beans).
- Field interception is not implemented, although support for field interception could be added without breaking the core Spring AOP APIs.
- If you need to advise field access and update join points, **consider a language such as AspectJ**.

Spring AOP Capabilities and Goals (3)

- Spring AOP's approach to AOP differs from that of most other **AOP frameworks**.
- The aim is **not** to provide the most complete AOP implementation (although Spring AOP is quite capable);
- It is rather to provide a **close integration** between AOP implementation and Spring IoC to help solve common problems in enterprise applications.
- Spring AOP will never strive to compete with **AspectJ** to provide a comprehensive AOP solution.

Spring AOP or full AspectJ?

- **Use the simplest thing that can work.**
- **Spring AOP is simpler than using full AspectJ** as there is no requirement to introduce the AspectJ compiler / weaver into your development and build processes.
- If you only need to **advise the execution of operations on Spring beans**, then Spring AOP is the right choice.
- If you need to **advise objects not managed by the Spring container** (such as domain objects typically), then you will need to use **AspectJ**.
- You will also need to use **AspectJ** if you wish to advise join points other than simple method executions (for example, **field get or set join points**, and so on).
- When using **AspectJ**, you have the choice of the **AspectJ** language syntax (also known as the "code style") or the @AspectJ annotation style.

Example

Project Setup



```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-context</artifactId>
  <version>3.0.2.RELEASE</version>
</dependency>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-aop</artifactId>
  <version>3.0.2.RELEASE</version>
</dependency>
<dependency>
  <groupId>org.aspectj</groupId>
  <artifactId>com.springsource.org.aspectj.runtime</artifactId>
  <version>1.6.6.RELEASE</version>
</dependency>
<dependency>
  <groupId>org.aspectj</groupId>
  <artifactId>aspectjweaver</artifactId>
  <version>1.6.6</version>
</dependency>
<dependency>
  <groupId>org.aspectj</groupId>
  <artifactId>aspectjrt</artifactId>
  <version>1.6.6</version>
</dependency>
<dependency>
  <groupId>cglib</groupId>
  <artifactId>cglib</artifactId>
  <version>2.2</version>
</dependency>
```

Pointcut

Pointcuts can be declared with an annotation (or with XML...)

```
@PointCut(expression)
private void aNameForThisSetOfMethods {}
```

The **expression** is based on the AspectJ pointcut language. Here are some examples:

the execution of any public method:

```
execution(public * *(..))
```

the execution of any method with a name beginning with "set":

```
execution(* set*(..))
```

the execution of any method defined by the AccountService interface:

```
execution(* com.xyz.service.AccountService.*(..))
```

the execution of any method on a Spring bean named 'tradeService':

```
bean(tradeService)
```

the execution of any method on a Spring bean with a name matching the wildcard expression

```
bean(*Service)
```

`execution(modifiers-pattern? ret-type-pattern declaring-type-pattern? name-pattern(param-pattern) throws-pattern?)`

<http://static.springsource.org/spring/docs/2.5.6/reference/aop.html#aop-pointcuts>

Defining a Pointcut “Inline”

```
@Aspect
public class MyFirstAspect {

    @Before("execution(public * ch.heigvd.osf..*.*(..))")
    public void myMethod(JoinPoint jp) {
        System.out.println("My advice has been applied...");
        System.out.println("target: " + jp.getTarget());
        System.out.println("this: " + jp.getThis());
        System.out.println("signature: " + jp.getSignature());
    }
}
```

```
<aop:aspectj-autoproxy/>
```

```
<bean id="myFirstAspect" class="ch.heigvd.osf.hellospringaop.aspects.MyFirstAspect">
</bean>
```

Notes:

- myMethod will be executed before **any public method** in **any class** in the `ch.heigvd.osf` package (or in a **sub-package**) is called.
- myMethod has access to runtime information

Using an @Aspect to Define Pointcuts

```
package ch.heigvd.osf.system;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
public class SystemPointCuts {

    @Pointcut("execution(* create*(..))")
    public void createMethods() {}

    @Pointcut("execution(* update*(..))")
    public void updateMethods() {}

    @Pointcut("execution(* delete*(..))")
    public void deleteMethods() {}

    @Pointcut("createMethods() && updateMethods() && deleteMethods()")
    public void allCRUDMethods() {}

}
```

Using an @Aspect to Implement Advices

```
package ch.heigvd.osf.system.logging;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class MyLoggingAspect {
    @Before("ch.heigvd.osf.system.SystemPointCuts.allCRUDMethods()")
    public void doLogOperation() {
        log.info("About to call a CRUD method....");
    }
}
```

Here, we work with:

- one pointcut, which is defined in the SystemPointCuts aspect (see previous slide)
- this pointcut defines a set of several join points: all the methods with a name starting with either create, update or delete
- one advice, which states that before every execution of the join points matching the pointcut, we will execute the doLogOperation

References

- <http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/aop.html>
- <http://www.javalobby.org/java/forums/t44746.html>
- http://en.wikipedia.org/wiki/Aspect-oriented_programming