

# Spring Framework





# Topics

- ❖ **Introduction**
  - What is Spring?
  - Basics – Beans and Containers
  - Beans Lifecycle
  - IOC - Inversion Of Control
  - Spring Application Design
  - Spring Modules
- ❖ **Spring Beans**
  - Bean lifecycle
  - Prototypes
  - Singletons
  - Application properties
  - Constructors
  - Callbacks
  - Collections
  - Configurations
  - Beans
  - Component Scan
  - Combining Lombok
- ❖ **Spring IoC**
  - Autowired & Resource
  - Post construct
  - Pre destroy
  - Lifecycle interceptors
- ❖ **Spring AOP**
  - Introduction
  - aspectJ
  - Creating aspects
  - Custom aspects
  - Annotation based aspects
  - External aspects
- ❖ **SpringJDBC & DAO Template**
- ❖ **SpringBoot**
  - Introduction
  - Start.spring.io
  - Boot starters
  - Default configuration & properties
  - @SpringBootApplication



# Topics

## ❖ SpringData

- Using Hibernate
- RDBMS Boot properties
- Spring JPA Repositories
- Querying
- Using NoSQL
- Spring MongoDB Repository

## ❖ SpringMVC - REST

- Introduction
- SpringBoot Web ext.
- Embedded web server
- REST mapping annotations
- Handling parameters
- Handling responses
- Handling errors
- RestTemplate
- Using Swagger UI for testing

## ❖ Spring Messaging

- Intro
- Kafka
- RabbitMQ

## ❖ Spring testing

- Introduction
- SpringBoot testing
- Testing annotations
- Unit & integration testing
- Tesing DB repositories
- Testing web endpoints
- Mocking & using alternative DB

## ❖ Spring Cloud

- Introduction to Microservices
- Introduction to Spring cloud
- Configuration server & GIT
- Discovery server and load-balancers
- API-Gateway & Circuit breakers
- Monitoring



# Introduction

# What is Spring?

The Spring Framework is a Java platform that provides comprehensive infrastructure support for developing Java applications.



## **MAIN GOAL:**

Spring handles the infrastructure so developers can focus on application business logic.

# What is Spring?

## Basics – *Beans* and Containers

Object that are managed by Spring container are called Beans.

Beans are the java objects which form a Spring application and are managed by Spring container.

The Spring container is responsible for instantiating, configuring, and assembling the beans.

The container gets its information on what objects to instantiate, configure, and manage by reading configuration **metadata** we define for the application.



# What is Spring?

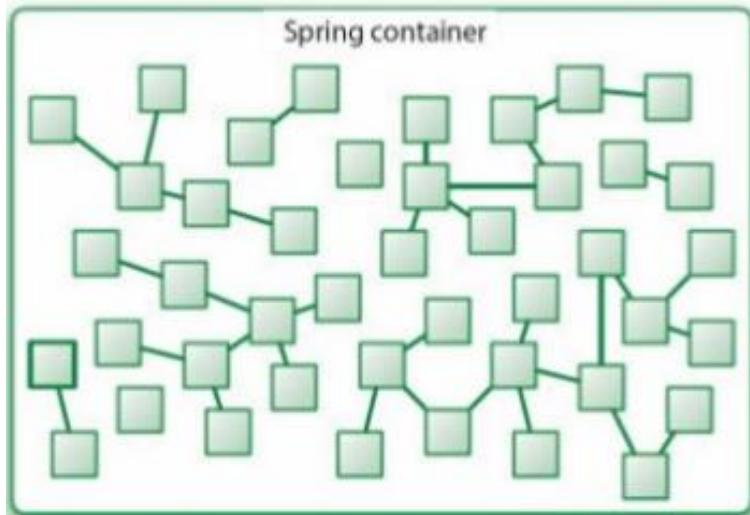
## Basics – Beans and Containers

The Container manages the Beans creation, configuration and management.

Bean that is instantiated by the Container has a lifecycle.

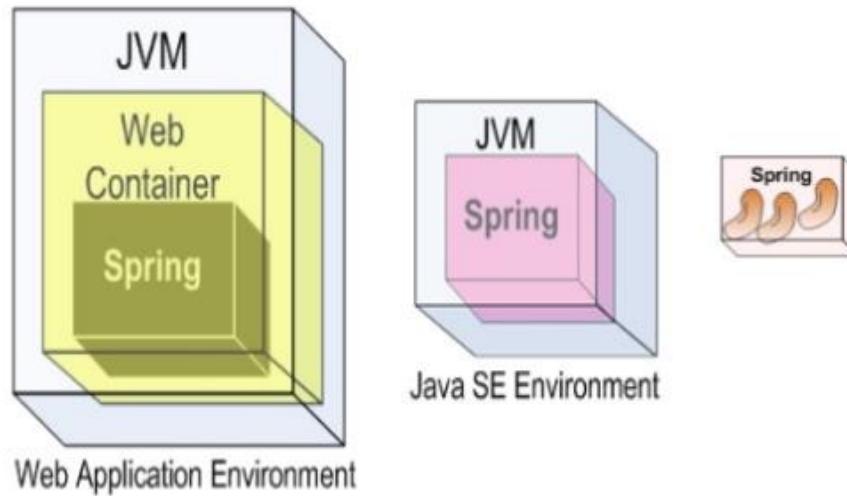
# What is Spring?

Spring Beans are created, wired together and live within a Container.



# What is Spring?

The Spring Container runs within a container (ie JVM or WEB container)

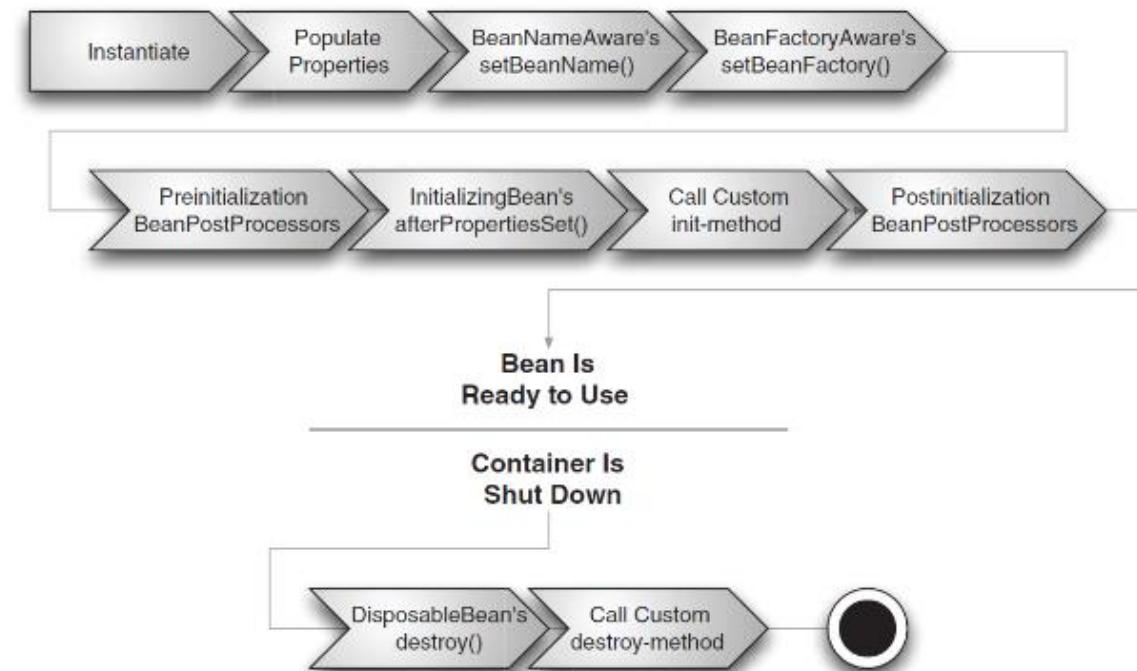


# What is Spring?

## Beans Lifecycle

When a Bean is instantiated, it might have to perform some actions to get into its initial state.

The container assists the Bean to achieve all required resources to get into initial state.





# What is Spring?

## IOC - Inversion Of Control & DI - Dependency Injection

Inversion of control (IoC) is a design principle in which components of a computer program/application receive the flow of control from a generic framework.

callbacks, schedulers, event loops and dependency injection are examples of design patterns that follow the inversion of control principle.



# What is Spring?

## Spring IOC - Inversion Of Control

The Container creates & manages the objects/Beans creation.

The contained INJECTS all the required dependencies into a Bean to bring it to its **initial state** using:

DI - @Autowired, @Resource

@PostConstruct method

Inversion of control is used to increase **modularity** of the program.

Inversion of control is used to increase **reuse** of components.

# What is Spring?

## Beans Lifecycle

As a part of each Bean initialization:

1. Bean resources (dependencies) are fulfilled - Injected, called DI
2. Init method is called after DI.

```
@Service
public class CustDocsServiceImpl extends ServiceBase implements CustDocsService{

    @Autowired
    private CustDocsRepository custDocsRepository;

    @Autowired
    private CustDocsTemplateRepository custDocsTemplateRepository;

    @Autowired
    private CustWalletService custWalletService;

    @Autowired
    private CustomerService customerService;

    @Autowired
    CrmService crmService;

    @Value("${mail.server.to.email}")
    private String mailServerToEmail;

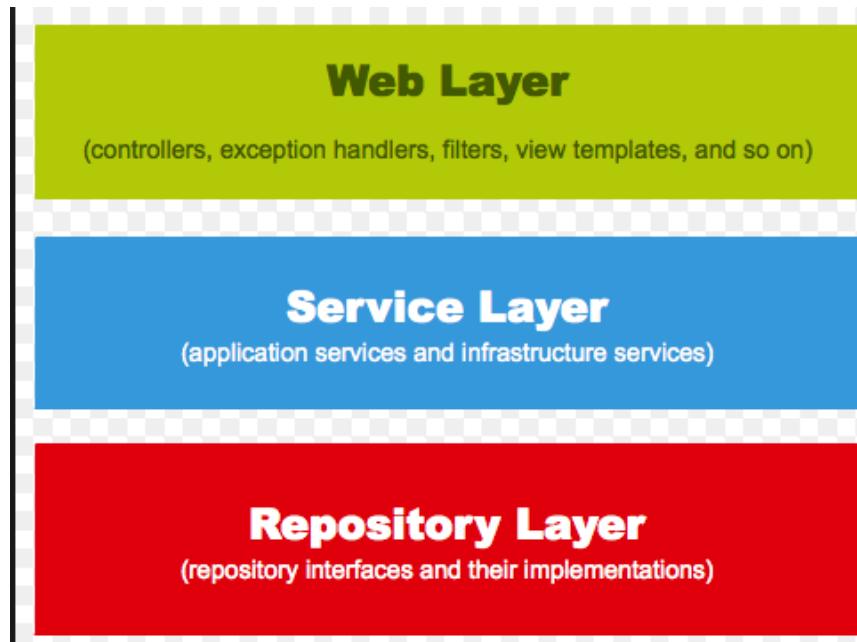
    @Value("${crm.get.user.document.bytes.url}")
    private String crmGetUserDocumentBytesUrl;

    //-----
    @PostConstruct
    public void init(){
        logger.debug("CrmServiceImpl.init() - in.");
    }
    //-----
```

# What is Spring?

## Spring Application Design

Best practice when building a Spring application, is to split application into layers:



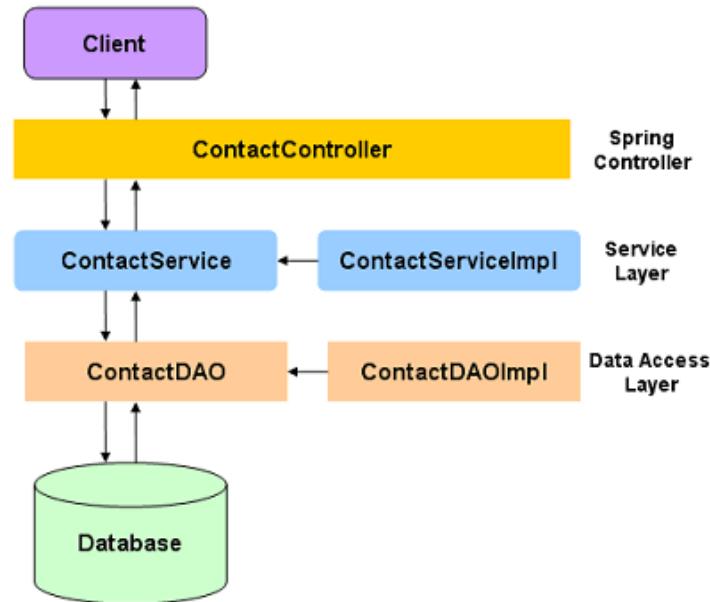
# What is Spring?

## Application design

Every layer is composed of Spring Beans.

Components of a ‘lower’ level are injected to higher layer components.

Injection on the interfaces gains  
**Loose coupling and separation of**  
implementation.





# Spring Modules



# What is Spring?

## Framework Modules

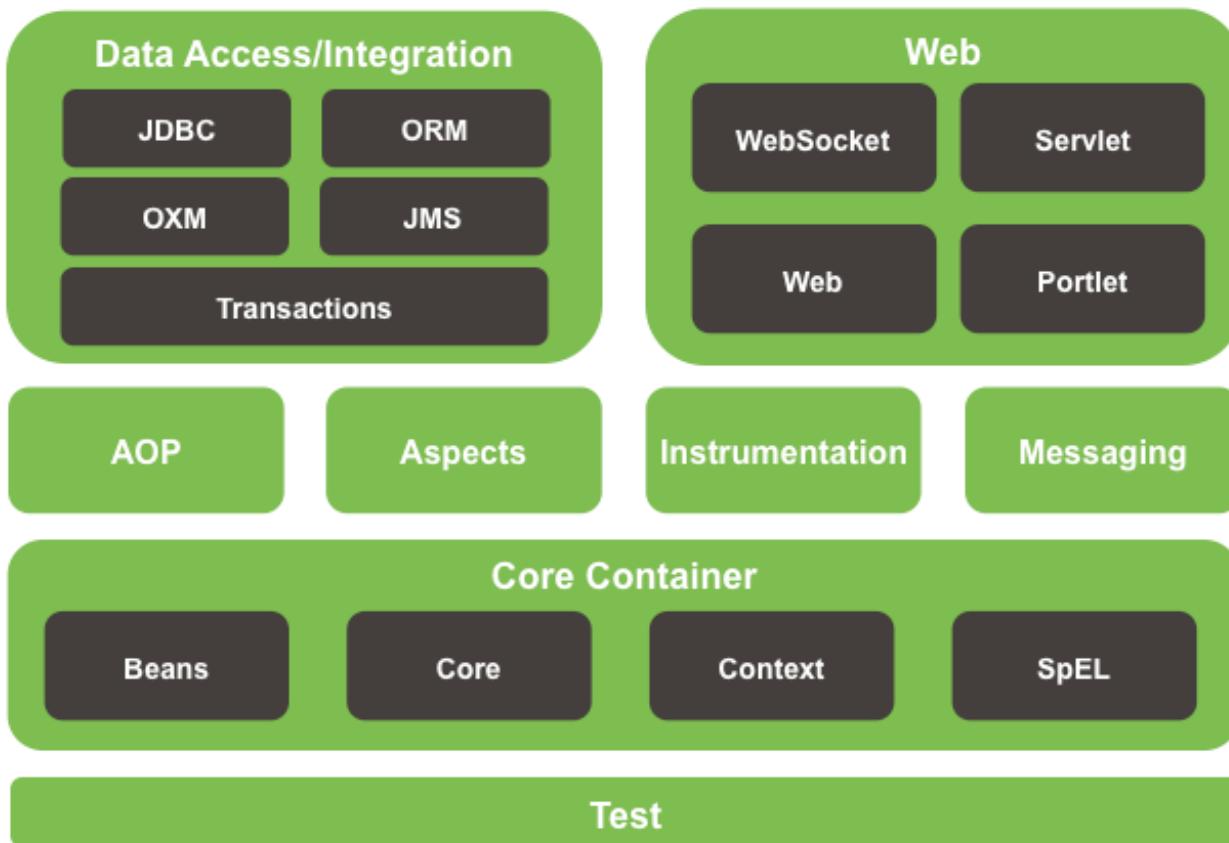
The Spring Framework consists of features, organized into about 20 modules. These modules are grouped into:

- Core Container
- Data Access/Integration
- Web
- AOP (Aspect Oriented Programming)
- Instrumentation
- Messaging
- Test

# What is Spring?



## Spring Framework Runtime





# What is Spring?

## Core Container

The Core Container consists of :

spring-core, spring-beans, spring-context, spring-context-support, and spring-expression (Spring Expression Language) modules.

The spring-core and spring-beans modules provide the fundamental parts of the framework, including the IoC and Dependency Injection features.



# What is Spring?

## BeanFactory

BeanFactory is a central registry of application components.

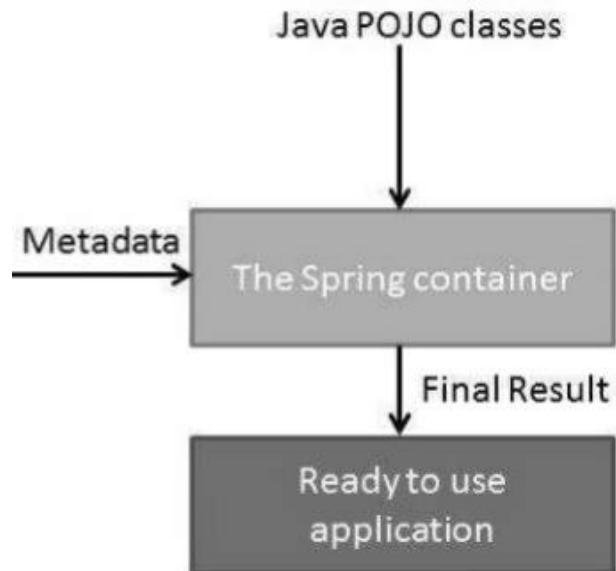
The BeanFactory is a sophisticated implementation of the *factory* pattern.

Normally a BeanFactory loads bean definitions stored in configuration METADATA source (such as Annotations or XML documents) and generates the Bean's context.

# What is Spring?

## BeanFactory

- The container gets its instructions on what objects to instantiate, configure, and assemble by reading the configuration **metadata** provided.
- The configuration metadata can be represented either by **XML**, Java **annotations**, or Java code.



# What is Spring?



## Conclusions

- Spring is an alternative framework to J2EE.
- Spring is lightweight container and development infrastructure.
- Spring is, in a way, a kind of Object-Factory.
- Spring provides Object lifecycle management.
- Simplifies development process.



# Spring provides solutions for:

Spring provides solutions for:

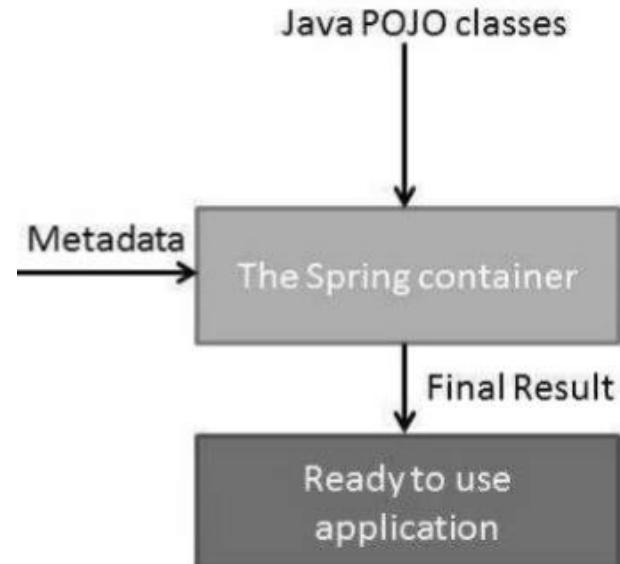
- Object lifecycle initialization & management
- Data conversion
- Expression Language
- DB Access (JDBC/JPA)
- Transactions
- Aspect Programming
- MVC
- JMS

and more..

# Basic Spring supports :

## Basic Concepts

- Objects initialization
  - Dependency Injection
  - Wiring/Auto-wiring
- All done by Declarative Programming
- Metadata Declares/describes relationships and DI.
- .



# Spring Declarative-Metadata obtained by:

## Spring Declarative-Metadata

- Annotations
- XML Deployment Descriptor

..or combination of them both..



# spring project





# Spring Beans



# The Spring Managed Bean

## The Basic Unit

# The Bean basic Unit

As mentioned earlier, Spring is an objects factory.

IOC - The developer does not create class instances explicitly,  
but let Spring create and manage new instances.

Creating an Instance in Traditional Java Programming :

```
public static void main(String[] args){  
    ...  
    Person p1 = new Person();  
    ...  
}
```

# The Bean basic Unit - Creating Beans

Creating a Person Instance - **Stereotype Annotations** approach:

Simply by Marking / Annotating POJO classes with **Stereotype Annotations** :

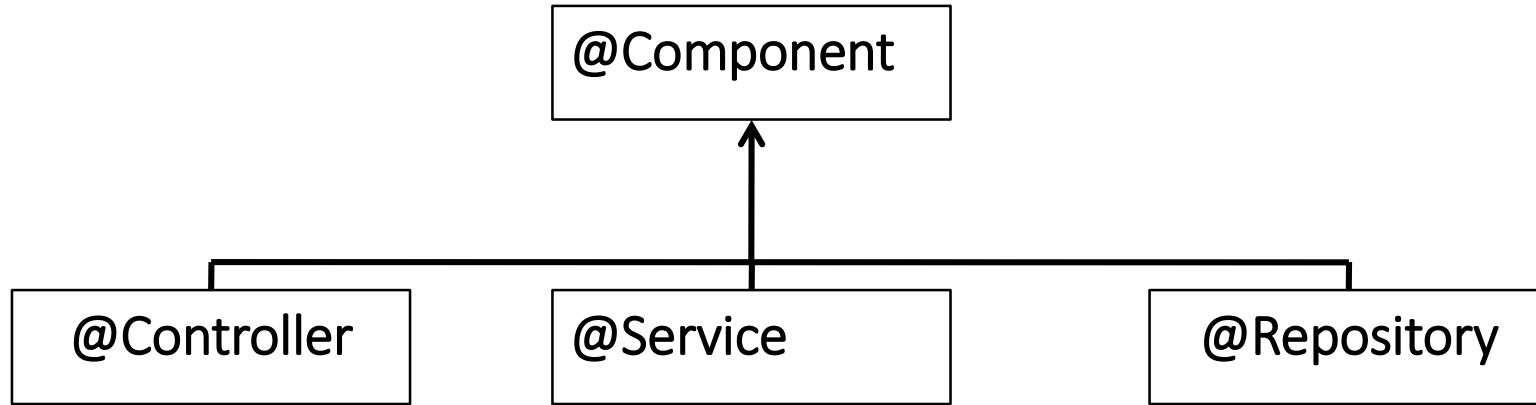
```
@Component("thePerson")
public class Person{
    ..
}
```

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



# The Bean basic Unit - Creating Beans

Creating a Person Instance - Stereotype Annotations approach:

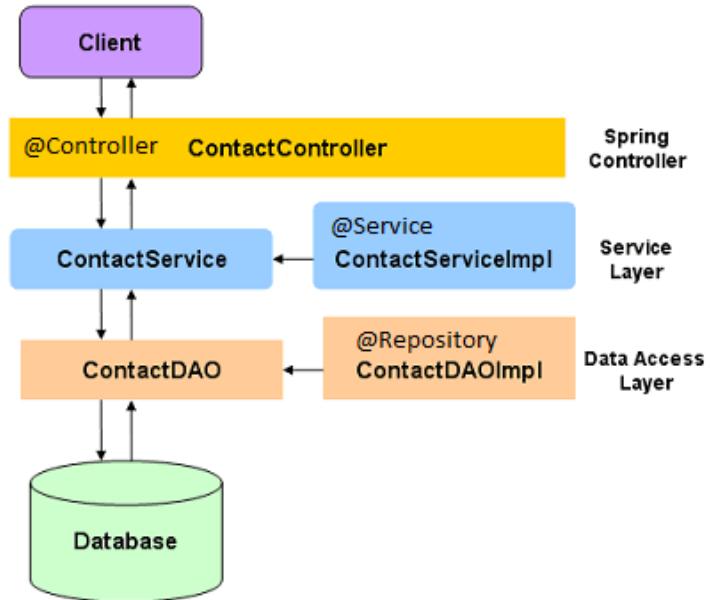


Domain-Driven Design Convention

# The Bean basic Unit - Creating Beans

Domain Driven Application design:

Each layer is composed of Spring Beans.  
Annotated with '@Component like'  
Stereotype Annotation





# The Bean basic Unit - Creating Beans

## Stereotype Annotations approach – Scanning..

```
public static void main(String[] args) {  
    ApplicationContext ctx = new  
        AnnotationConfigApplicationContext(SpringMain.class);  
}
```

```
@Configuration // ← annotates a special Spring-config. class  
@ComponentScan (basePackages="com.jbt.spring.example.beans,  
                com.jbt.spring.example.main")  
public class SpringMain {  
}
```

```
@Controller  
public class PersonController{  
    ..  
}
```

```
@Service  
public class PersonServiceImpl implements PersonService{ ..  
}
```

```
@Repository  
public class PersonDao{  
    ..  
}
```

# The Bean basic Unit - Creating Beans

Creating a Person Instance - Spring Configuration class approach

```
public static void main(String[] args) {  
    ApplicationContext ctx = new AnnotationConfigApplicationContext(SpringMain.class);  
}
```

```
@Configuration // ← annotate a special Spring-config. class  
@ComponentScan (basePackages="com.jbt.spring.example.model,  
                com.jbt.spring.example.main")  
public class SpringMain {  
    @Bean // ← id="thePerson" as the method name  
    public Person thePerson(){  
        return new Person(1234,"Joe", 24, null, null);  
    }  
}
```

# The Bean basic Unit - Creating Beans

In any Factoring approach - Spring creates & manages a global **map** (Context) that includes the beans ids and reference to the newly created object..

1  

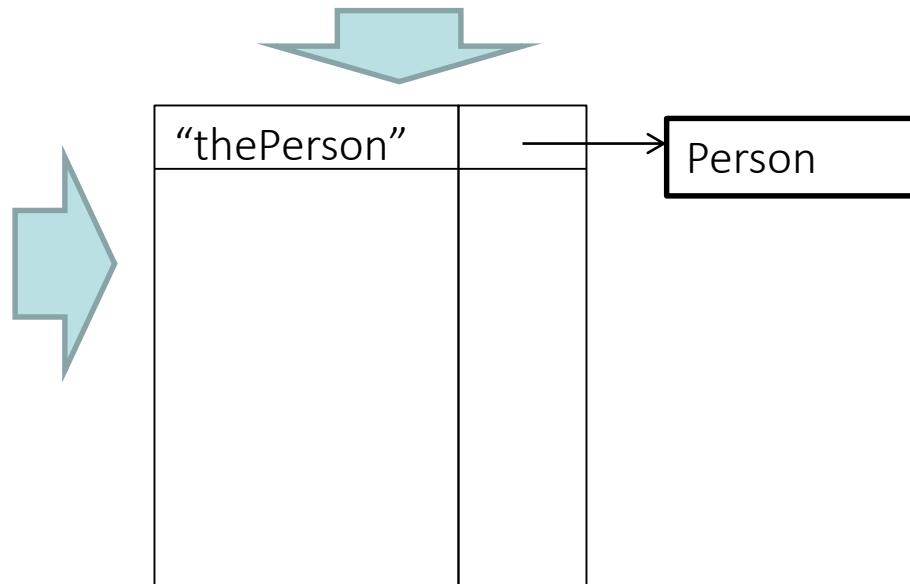
```
<bean id="thePerson"
      class="com.jbt.spring.example.model.Person">
</bean>
```

2  

```
@Component("thePerson")
public class Person{
    ..
}
```

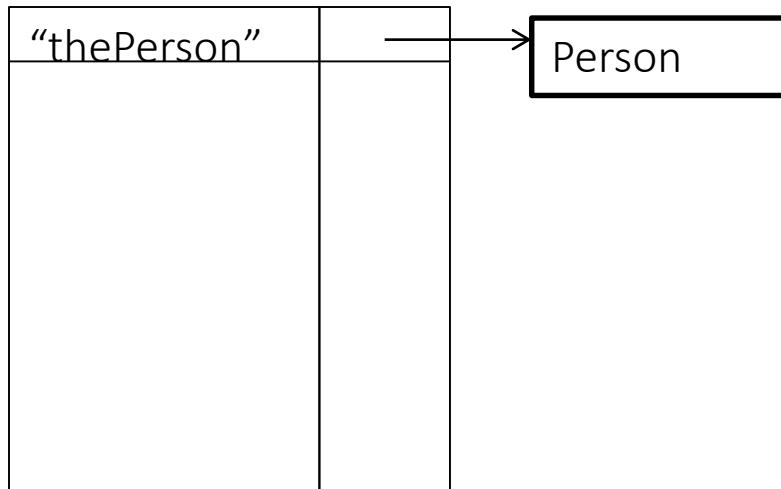
3  

```
@Configuration
public class SpringMain{
    @Bean(name="thePerson")
```



# The Bean basic Unit - Creating Beans

After creation, retrieving the bean by its id is simple..



```
public class SpringMain {  
    public static void main(String[] args) {  
        ApplicationContext ctx = new  
            AnnotationConfigApplicationContext(SpringMain.class);  
        Person p1 = (Person)ctx.getBean("thePerson"); // ← referencing Bean by id  
        ...  
    }  
}
```

# The Bean basic Unit - Creating Beans

Multiple implementations of the same bean type - defaults:

```
@Component  
@Primary  
public class ComponentA{...}
```

```
@Configuration  
public class CustConfiguration{  
    @Bean  
    public ComponentA compA2(){  
        ComponentA a2=new ComponentA();  
        a2.set()...  
        return a2;  
    }  
    ....
```

- `getBean(ComponentA.class)` – takes the `@Primary` or fails if no primary is specified
- `getBean("componentA", ComponentA.class)` – using default bean name
- `getBean("compA2", ComponentA.class)` – using qualifier
- Both `@Component` & `@Bean` accepts custom qualifiers (bean names)

```
@Component("myBean")  
@Bean("myBean")
```

# Dependencies Injection

## Dependency Injection

# Dependencies Injection

```
@Service
```

```
public class CustWalletServiceImpl extends ServiceBase implements CustWalletService{
```

```
    @Autowired
```

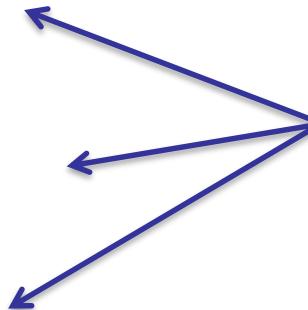
```
    private CustDocsService custDocsService;
```

```
    @Autowired
```

```
    private CustWalletService custWalletService;
```

```
    @Resource
```

```
    private CustomerService customerService;
```



## By Type Injection

Will Inject a 'ServiceImpl'  
In case there is only one implementation

## @Resource vs. @Autowired

- Basically, does the same.
- @Resource was adopted since it is used for DI in JEE
- Prefer @Autowired – this way is it clear we are using Spring-based injections

# Dependencies Injection

```
@Service
```

```
public class CustWalletServiceImpl implements CustWalletService{
```

```
    private CustDocsService custDocsService;
```

```
@Autowired
```

```
    public void setCustDocsService(CustDocsService custDocsService) {  
        this.custDocsService = custDocsService;  
    }
```

```
    public CustDocsService getCustDocsService() {  
        return custDocsService;  
    }
```



# Dependencies Injection

Suppose we have created 2 beans via a Configuration class:

```
@Configuration  
@ComponentScan("com.baeldung.spring")  
public class Config {
```

```
@Bean  
public Engine engine() {  
    return new Engine("v8", 5);  
}
```

```
@Bean  
public Transmission transmission() {  
    return new Transmission("sliding");  
}
```

# Dependencies Injection

The two beans can be injected to 3<sup>rd</sup> bean by Constructor Injection:

```
@Component
```

```
public class Car {
```

```
    @Autowired
```



```
        public Car(Engine engine, Transmission transmission) {
```

```
            this.engine = engine;
```



```
            this.transmission = transmission;
```

```
}
```

```
}
```

## Java Singleton'S VS Spring Singleton'S

Examples:

com.spring.java.singleton  
com.spring.spring.singleton



# Singleton

Question: Will spring create this Singleton @Component ?

**@Component**

```
public class Singleton1 {  
    public static Singleton1 INSTANCE;  
    private int indx = (int)(Math.random()*11);;  
  
    private Singleton1(){  
        System.out.println("Singleton1.CTOR-" + indx);  
    }  
    public static Singleton1 getInstance(){  
        if ( INSTANCE == null ){  
            INSTANCE = new Singleton1();  
        }  
        return INSTANCE;  
    }  
  
    @PostConstruct  
    public void init(){  
        System.out.println("Singleton1.init() - indx:[" + indx + "]");  
    }  
}
```



# Singleton

Yes it will !

Spring @Component Annotation breaks privacy

Running main:

```
public static void main(String[] args) {  
    ApplicationContext ctx = new AnnotationConfigApplicationContext(SpringConfig.class);  
}
```

With Configuration Class (to scan packages):

```
@Configuration  
@ComponentScan(basePackages="com.spring")  
public class SpringConfig {....}
```

Will print

```
Singleton1.CTOR-9  
Singleton1.init() - indx:[9]
```

# Singleton

What about

```
@Configuration  
public class SpringConfig {
```

```
@Bean  
public Singleton1 bean1(){  
    return new Singleton1(); // ??  
}
```

# Singleton

This will not work as it is a java syntax error:

```
@Bean  
public Singleton1 bean1(){  
    return new Singleton1(); // syntax error  
}
```

Fix to:

```
@Bean  
public Singleton1 bean1(){  
    return Singleton1.getInstance(); // OK  
}
```

# Spring Singleton

Every Bean in Spring is a Singleton  
unless explicitly specified:

```
@Component
public class MyBean {

    @Resource
    private String s1;
    @Resource
    private String s2;

    @PostConstruct
    public void init(){
        System.out.println("MyBean.init() - s1:[" + s1 +
"']");
        System.out.println("MyBean.init() - s2:[" + s2 +
"']");
    }
}
```

The diagram shows two annotations, `@Resource`, pointing from the code to a small white box containing the letters "DI".

```
@Configuration
@ComponentScan(
    basePackages="com.spring.spring.singleton")
public class SpringConfig {
```

```
@Bean
public String stringBean(){
    String str = "Hello "
    + (int)(Math.random()*11);
    return str ;
}
}
```

Will print:  
MyBean.init() - s1:[Hello 9]  
MyBean.init() - s2:[Hello 9]



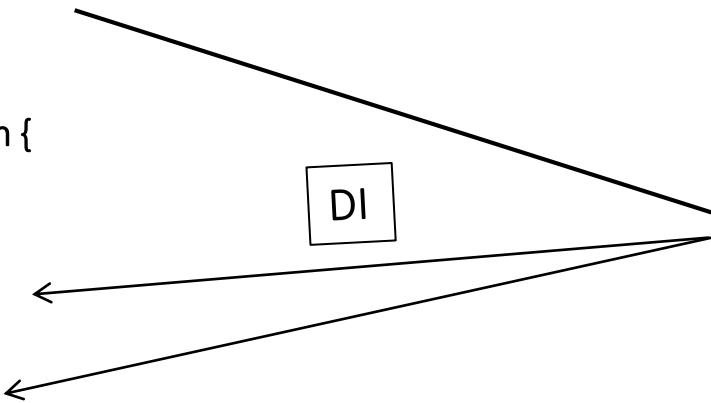
# Spring Singleton

Every Bean in Spring is a Singleton unless explicitly specified:

'prototype'

```
@Component
public class MyBean {
    @Resource
    private String s1;
    @Resource
    private String s2;

    @PostConstruct
    public void init(){
        System.out.println("MyBean.init() - s1:[" + s1 + "]");
        System.out.println("MyBean.init() - s2:[" + s2 + "]");
    }
}
```



```
@Configuration
@ComponentScan(
    basePackages="com.spring.spring.singleton")
public class SpringConfig {

    @Bean
    @Scope("prototype")
    public String stringBean(){
        String str = "Hello "
            + (int)(Math.random()*11);
        return str ;
    }
}
Will print:
MyBean.init() - s1:[Hello 9]
MyBean.init() - s2:[Hello 3]
```

# Singleton Lazy Initialization

- Spring eagerly loads singletons
- @Lazy allows delaying singleton creation until actually used
  - By context.getBean()
  - By @Autowired to other lazily loaded singleton bean
  - Examples:

```
@Component  
@Lazy  
public class MySingleton {  
    ...  
}
```

```
@Configuration  
public class MyConfiguration {  
  
    @Bean  
    @Lazy  
    public MySingleton(){  
        ...  
    }  
    ...  
}
```

# Circular Injection

Spring detects unresolvable circular reference and cancels context initiation

- Runtime message: “Error creating bean with name 'proto': Requested bean is currently in creation: Is there an unresolvable circular reference?”
- Example scenario:

```
@Component
public class BeanA {

    private BeanB b;

    public BeanA(BeanB b){
        this.b=b;
    }...
```

```
@Component
public class BeanB {

    private BeanA a;

    public BeanB(BeanA a){
        this.a=a;
    }...
```

# Circular Injection

Spring detects unresolvable circular reference and cancels context initiation

- when using the same scenario with @Autowired injection – it's OK!
- Auto-wiring is done after default constructor has been invoked

```
@Component
public class BeanA {

    @Autowired
    private BeanB b;
    ....
}
```

```
@Component
public class BeanB {

    @Autowired
    private BeanA a;
    ....
}
```

# Using Lombok with Spring Components

Lombok allows you to shorten sources by saving the need to code the obvious & repetitive pieces:

- Default constructors
- Fully argument constructors
- Getters and setters
- `toString`
- `hashcode()` & `equals()`

```
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.16.8</version>
    <scope>provided</scope>
</dependency>
```

- In order to install and enable code completion on your Eclipse IDE:
  - Add maven dependency / or download Lombok jar manually
  - Navigate to Lombok jar and type: `java -jar lombok.version.jar`
  - In the appearing window – navigate to your `eclipse.exe` and click Install
  - Launch Eclipse and that's it!

# Using Lombok with Spring Components

Lombok main annotations:

- Constructor generation:
  - `@NoArgsConstructor` – empty constructor
  - `@RequiredArgsConstructor` – accepts all non-null members
- Getters & Setters
  - `@Getter` & `@Setter` on class level – generates get/set to all members
    - Boolean members `getXXX` becomes: `isXXX()`
  - `@Getter` & `@Setter` on field level – generates for that field only
    - Are overridden by class level `@Getter`&`@Setter`
- `@Data`
  - Is a combination of: `@Getter`, `@Setter` & `@RequiredArgsConstructor`
  - Is also generates `toString()`, `hashcode()` and `equals()`



# Using Lombok with Spring Components

Lombok main annotations:

- Common methods generation:
  - `@ToString`
  - `@EqualsAndHashCode`
  - All accepts 'of' attribute to specify counted fields for generation
  - Can accept 'exclude' for excluded fields instead
- Specifying required fields
  - Is done via `@NonNull` annotation on field level
  - `@RequiredArgsConstructor` counts in only 'non-null' fields

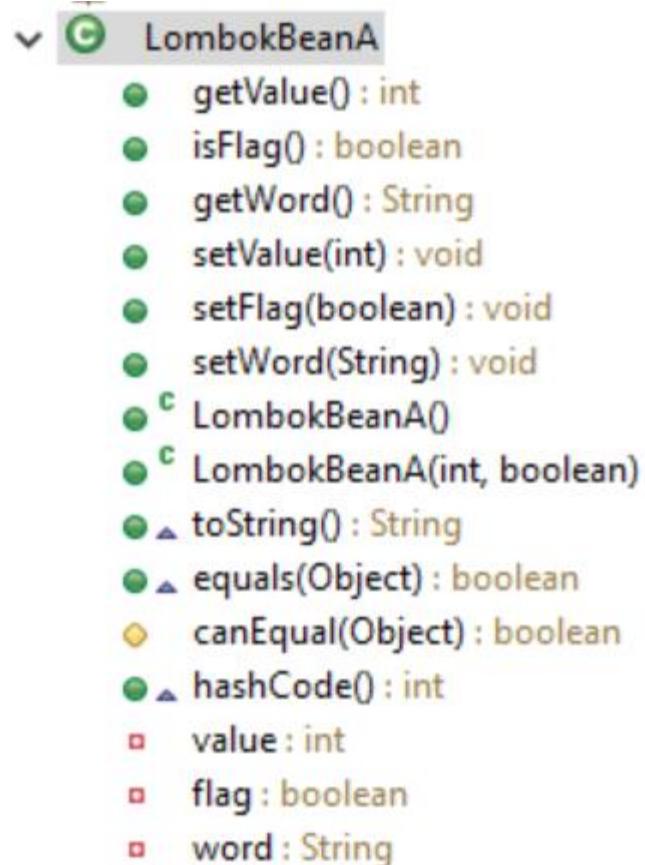
# Using Lombok with Spring Components

Examples:

```
@Component
@Getters @Setters
@NoArgsConstructor @RequiredArgsConstructor
@ToString()
@EqualsAndHashCode(of={"value", "flag"})
public class LombokBeanA {

    private @NotNull int value;
    private @NotNull boolean flag;
    private String word;

}
```



# Using Lombok with Spring Components

Examples:

```
@Component
@Data
public class LombokBeanB {

    private @NonNull int value;
    private @NonNull boolean flag;
    private @NonNull String word;

}
```

- LombokBeanB
- getValue() : int
- isFlag() : boolean
- getWord() : String
- setValue(int) : void
- setFlag(boolean) : void
- setWord(String) : void
- equals(Object) : boolean
- canEqual(Object) : boolean
- hashCode() : int
- toString() : String
- LombokBeanB(int, boolean, String)
- value : int
- flag : boolean
- word : String

# Beans Lifecycle & Interceptors

So far in this course we got familiar with 2 types of annotations:

Stereotype Annotations - @Component, @Controller, @Service, @Repository

Rendering Annotations - @Resource, @Autowired

Lifecycle Annotations - Additional type of annotations assigned to a bean creation & destruction events.

@PostConstruct

@PreDestroy

# Beans Lifecycle & Interceptors

@PostConstruct

@PreDestroy

There are three phases in Spring bean's life:

- Created/Ready - A creation of a Spring bean can be a complex action.
- Used - Beans are the performers/actors of the application
- Destroyed - At Spring container shutdown, beans are destroyed.

# Beans Lifecycle & Interceptors

@PostConstruct

called after Bean construction..

1. Constructor
2. DI
3. @PostConstruct

@Componenet

```
public class Person{
```

....

**@PostConstruct**

```
public void init() {  
    System.out.println("after Construction init Method - @PostConstruct");
```

```
}
```

....

....

# Beans Lifecycle & Interceptors

@PreDestroy  
called before Bean destruction..

```
@Componenet  
public class Person{  
....  
@PreDestroy  
public void beforeDestruction() {  
    System.out.println("beforeDestruction - @PreDestroy");  
}  
....
```



***NOTE:** To enable destroy-method action , registerShutdownHook() should be called*

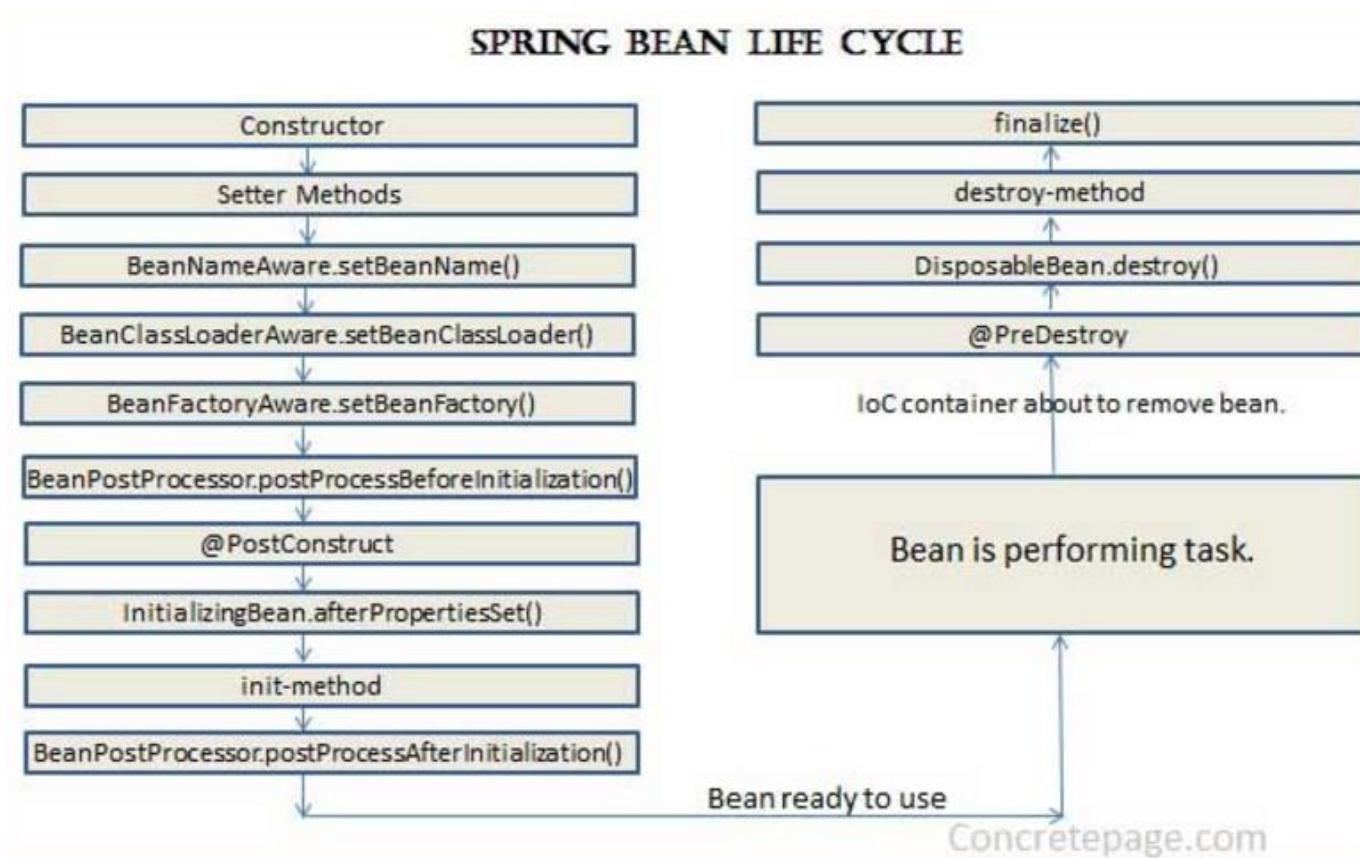
# Beans Lifecycle & Interceptors

## Awareness Interfaces – Bean lifecycle interceptors

Examples: com.spring.spring.beans.lifecycle

# Beans Lifecycle & Interceptors

## Awareness Interfaces – Bean lifecycle interceptors



Concretepage.com

# Beans Lifecycle & Interceptors

Awareness Interfaces – Bean lifecycle interceptors

During bootstrapping, Spring examines each bean to determine if it implements any of the xxxAware interfaces.

When one is found, it invokes the interface method, providing the piece of information that is being asked for.

# Beans Lifecycle & Interceptors

By implementing the **BeanFactoryAware** or **ApplicationContextAware** interfaces we can obtain a knowledge at the factory level

```
@Component
public class MyBeanFactoryAware
```

```
    implements BeanFactoryAware {
        public void setBeanFactory(BeanFactory factory) throws BeansException {
            /*1*/ System.out.println("FACTORY: " + factory.getClass().getSimpleName() );
            /*2*/ String [] names = ((DefaultListableBeanFactory)factory).getBeanDefinitionNames();
            int i = 1;
            System.out.println("BeanFactoryAware.setBeanFactory() - Application BEAN LIST:");
            for (String name : names) {
                System.out.println("BeanFactoryAware.setBeanFactory() - " + (i++) + ")
                    BeanFactoryAware: name: " + name);
            }
            /*3*/ System.out.println("\nBeanFactoryAware.setBeanFactory() - BEANS Annotated with @Configuration:");
            Map<String, Object> map = ((DefaultListableBeanFactory)factory).getBeansWithAnnotation(Configuration.class);
            i = 1;
            Object bean = null;
            for (String key : map.keySet()) {
                bean = map.get(key);
                System.out.println(" " + (i++) + ") BeanFactoryAware: Bean name: " + bean.getClass().getSimpleName() );
            }
        }
    }
```



# Beans Lifecycle & Interceptors

## OUTPUT:

**FACTORY:** DefaultListableBeanFactory

BeanFactoryAware.setBeanFactory() - Application **BEAN LIST:**

- 1) BeanFactoryAware: name:  
org.springframework.context.annotation.internalConfigurationAnnotationProcessor
- 2) BeanFactoryAware: name:  
org.springframework.context.annotation.internalAutowiredAnnotationProcessor
- 3) BeanFactoryAware: name:  
org.springframework.context.annotation.internalRequiredAnnotationProcessor
- 4) BeanFactoryAware: name:  
org.springframework.context.annotation.internalCommonAnnotationProcessor
- 5) BeanFactoryAware: name:  
org.springframework.context.annotation.internalPersistenceAnnotationProcessor
- 6) BeanFactoryAware: name: springConfig
- 7) BeanFactoryAware: name: myBean
- 8) BeanFactoryAware: name: myBeanFactoryAware
- 9) BeanFactoryAware: name: myBeanPostProcessor
- 10) BeanFactoryAware: name: stringBean

BeanFactoryAware.setBeanFactory() - **BEANS Annotated with @Configuration:**

- 1) BeanFactoryAware: Bean name: SpringConfig\$\$EnhancerBySpringCGLIB\$\$69a2f60e

# Beans Lifecycle & Interceptors

## BeanPostProcessor

Gets notification for any newly created beans.

Allows intrusive interference around bean creation.

```
public Object postProcessBeforeInitialization(Object bean, String beanName)
```

Applies the BeanPostProcessor to a given new bean instance before any bean initialization callbacks (init-method, @PostConstruct)

```
public Object postProcessAfterInitialization(Object bean, String beanName)
```

Applies the BeanPostProcessor to a given new bean instance after any bean initialization callbacks (destroy-method, @PreDestroy)

# Beans Lifecycle & Interceptors

## BeanPostProcessor Example

```

@Service("myBean")
public class MyBean {
    @Resource
    private String s1;
    @Resource
    private String s2;

    public MyBean() { }

    @PostConstruct
    public void init(){
        System.out.println("MyBean.init() - s1:[" + s1 + "]");
        System.out.println("MyBean.init() - s2:[" + s2 + "]");
    }
    @PreDestroy
    public void destroy(){ System.out.println("MyBean.destroy()"); }
}
    
```

DI

```

@Configuration
public class SpringConfig {
    @Bean
    public String stringBean(){
        String str = "Hello " +
        (int)(Math.random()*11);
        return str ;
    }
}
    
```

# Beans Lifecycle & Interceptors

## BeanPostProcessor Example

(Example: com.spring.spring.beans.lifecycle)

```
@Component
public class MyBeanPostProcessor implements BeanPostProcessor {
    @Override

    public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {
        System.out.println("BeanPostProcessor.postProcessBeforeInitialization() - beanName:[" + beanName + "],
                           beanClass" + bean.getClass().getName());
        if (bean instanceof com.spring.spring.beans.lifecycle.MyBean) {
            try {
                System.out.println("\t\tbean is MyBean - changing s1 field value");

                changeFieldsValue(bean);

            } catch (IllegalAccessException | RuntimeException e) {
                e.printStackTrace();
            }
        }
        return bean;
    }
}
```

# Beans Lifecycle & Interceptors

## BeanPostProcessor Example – CONT.

```
private void changeFieldsValue(Object bean) throws RuntimeException, IllegalAccessException {  
    Field [] fields = bean.getClass().getDeclaredFields();  
    if ( fields != null ){  
        for (Field field : fields) {  
            if (field.getName().equals("s1")){  
                field.setAccessible(true);  
                System.out.println("\t\changeFieldsValue() s1 field value: " + field.get(bean) );  
                field.set(bean, "THE AWARENESS INTERFACES POWER!");  
            }  
        } //for  
    } //fields  
}
```

# Beans Lifecycle & Interceptors

## BeanPostProcessor Example – CONT.

```
@Override
public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {
    System.out.println("BeanPostProcessor.postProcessAfterInitialization() - beanName:[" + beanName + "],
                       beanClass" + bean.getClass().getName());
    return bean;
}
```

# Beans Lifecycle & Interceptors

## BeanPostProcessor Example – CONT.

### OUTPUT:

```
BeanPostProcessor.postProcessBeforeInitialization() - beanName:[springConfig],  
    beanClass com.spring.spring.beans.lifecycle.SpringConfig$$EnhancerBySpringCGLIB$$5e31e0dc  
BeanPostProcessor.postProcessAfterInitialization() - beanName:[springConfig],  
    beanClass com.spring.spring.beans.lifecycle.SpringConfig$$EnhancerBySpringCGLIB$$5e31e0dc  
BeanPostProcessor.postProcessBeforeInitialization() - beanName:[stringBean], beanClass java.lang.String  
BeanPostProcessor.postProcessAfterInitialization() - beanName:[stringBean], beanClass java.lang.String  
BeanPostProcessor.postProcessBeforeInitialization() - beanName:[myBean],  
    beanClass com.spring.spring.beans.lifecycle.MyBean  
        bean is MyBean - changing s1 field value  
        changeFieldsValue() s1 field value: Hello 9  
MyBean.init() - s1:[THE AWARENESS INTERFACES POWER!]  
MyBean.init() - s2:[Hello 9]  
BeanPostProcessor.postProcessAfterInitialization() - beanName:[myBean], beanClass  
com.spring.spring.beans.lifecycle.MyBean  
Test.main() - bean:MyBean [s1=THE AWARENESS INTERFACES POWER!, s2=Hello 9]  
MyBean.destroy()
```



# spring beans





# AOP

## Aspect Oriented Programming



## Aspect-oriented programming (AOP)

Is an approach to programming that allows global properties of a program to determine how it is compiled into an executable program.

An *aspect* is a subprogram that is associated with a specific property of an existing program.

AOP can be used with object-oriented programming.



## Use case - 1:

Suppose we have to develop a banking application with a very simple method for transferring an amount from one account to another:

```
public void transfer(Account fromAct, Account toAct, int amount) {  
    if (fromAct.getBalance() - amount <= 0) {  
        throw new InsufficientFundsException();  
    }  
    fromAct.withdraw(amount);  
    toAct.deposit(amount);  
}
```



## Use case - 1:

However, this transfer method lacks :

- User Authentication & Authorization checks to verify that the user authorized to perform this operation.
- Open database transaction to encapsulate the operation.
- The operation should be logged to the system log



# AOP

## Use case - 1:

A version with all those new concerns, could look somewhat like this:

```
void transfer(Account fromAct, Account toAct, int amount, User user, Logger logger) {  
    logger.info("Transferring money...");  
    if (!checkUserPermission(user)){  
        logger.info("User has no permission.");  
        throw new UnauthorizedUserException();  
    }  
    if (fromAct.getBalance() - amount <= 0) {  
        logger.info("Insufficient funds.");  
        throw new InsufficientFundsException();  
    }  
    fromAct.withdraw(amount);  
    toAct.deposit(amount); //get database connection //save transactions  
    logger.info("Successful transaction.");  
}
```

In this code version we mixed business-logic with the basic code.



# AOP

- *What if we have in our banking application many business methods as: deposit(), withdraw(), openAccount() ?*
- *What if the same logic has to be used also from WEB module as well as from batch application ?*

AOP allows us to hook the calls to business methods and interfere in the middle with a common logic we need to embed



## Java - Dynamic Proxy

### Use case-2:

We have to develop a List of strings that switches any new added word to lowercase.

We can develop something as the following:



# AOP

## Java - Dynamic Proxy

### Use case-2:

```
public class LowercaseList<String> implements List<String> {
```

```
    List<String> list = null;  
  
    public LowercaseList(){  
        list = new ArrayList<String>();  
    }
```

And override the add method in such way turning any added string to lowercase:

```
@Override  
  
public boolean add(String str) {  
    return list.add(str.toLowerCase());  
}
```



## Java - Dynamic Proxy

**Use case-2:**

BUT ALSO....

We'll be forced to implement the rest 23 or more methods of the List interface as: remove(), size(), removeAll (), clear() , etc.

Not Sounds great..

What can we do ?



## Java - Dynamic Proxy

Starting with java 1.3, Sun introduced the Dynamic Proxy support allows you to Hook method calls of specific object.

We develop a class implements InvocationHandler and implement the invoke () method:

```
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;

public class LowercaseProxy implements InvocationHandler {
    private Object obj;
    public LowercaseProxy(Object obj) {
        this.obj = obj;
    }
}
```



# AOP

The **invoke()** method will be called with any call to the tracked object methods:

```
public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  
    if (method.getName().equals("add") && args != null) {  
        for (int i = 0; i < args.length; i++) {  
            if (args[i] instanceof String) {  
                args[i] = ((String)args[i]).toLowerCase();  
            } //if  
        } //for  
    } //if  
    Object returnObject = method.invoke(obj, args);  
    return returnObject;  
} //invoke()  
} //class
```



## The main() method:

```
public static void main(String[] args) {  
    List<String> list = new ArrayList<String>();  
  
    ClassLoader classLoader = java.util.List.class.getClassLoader();  
    Class<?>[] interfaces = new Class[] { java.util.List.class };  
    InvocationHandler lowerCaseProxy = new LowercaseProxy(list);  
  
    Object proxy = Proxy.newProxyInstance(classLoader, interfaces, lowerCaseProxy);  
  
    List<String> theList = (List<String>) proxy; // cast from Object to List  
    theList.add("Hello"); // this will call the invoke() method..  
}
```



There are two kind of proxy patterns :

## Static Proxy :

Where we create a proxy object for every class.

This is not feasible and practical

## Dynamic Proxy :

In this , proxies are created dynamically through reflection .

This functionality is added from JDK 1.3 .

Dynamic Proxy forms the basic building block of Spring AOP .

Other frameworks implementing aop other than Spring AOP are [AspectJ](#) and [JBoss AOP](#)

The original class must implement a interface ,only those method declared in interface get proxied and then we cast the proxy to the interface type .

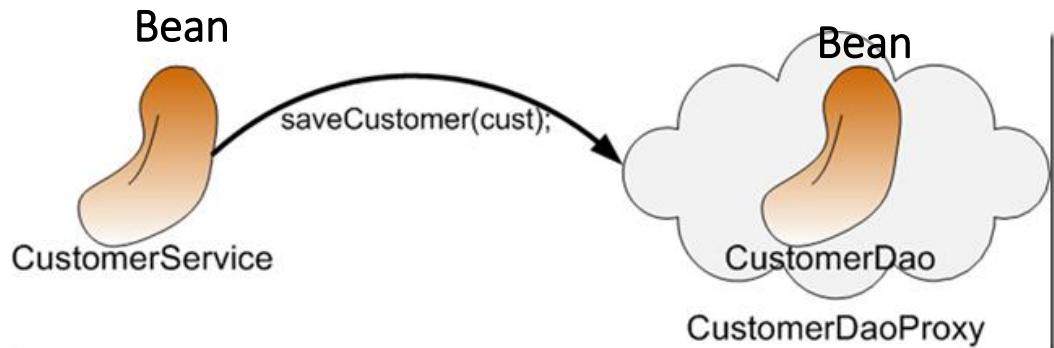
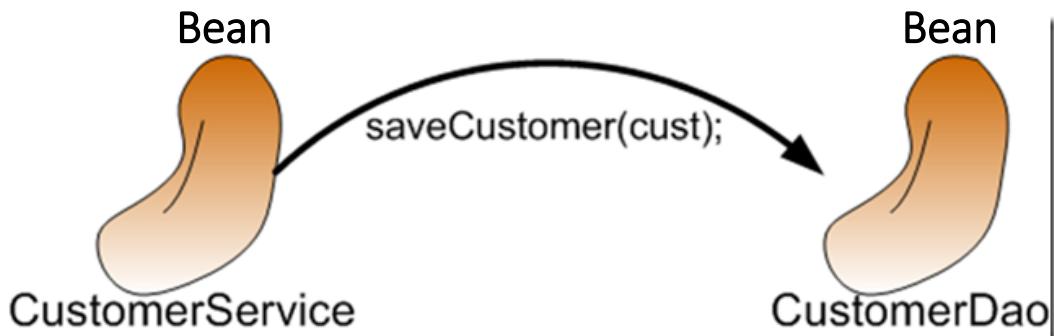


# AOP

## Dynamic Proxy

As an example, a dao bean with the saveCustomer( ) method on a DAO.

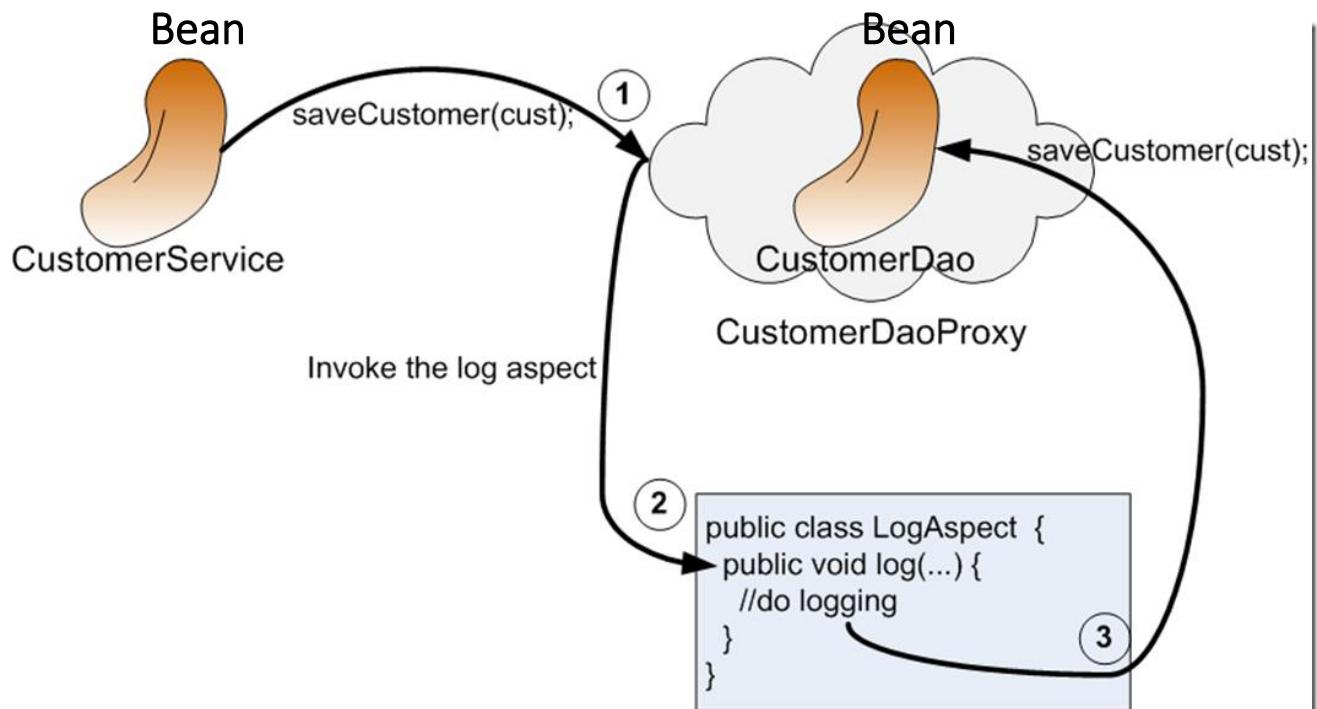
Spring detects your need to call aspect through your AOP configuration or annotations. When it does, it builds a proxy around the "target" object.



# AOP

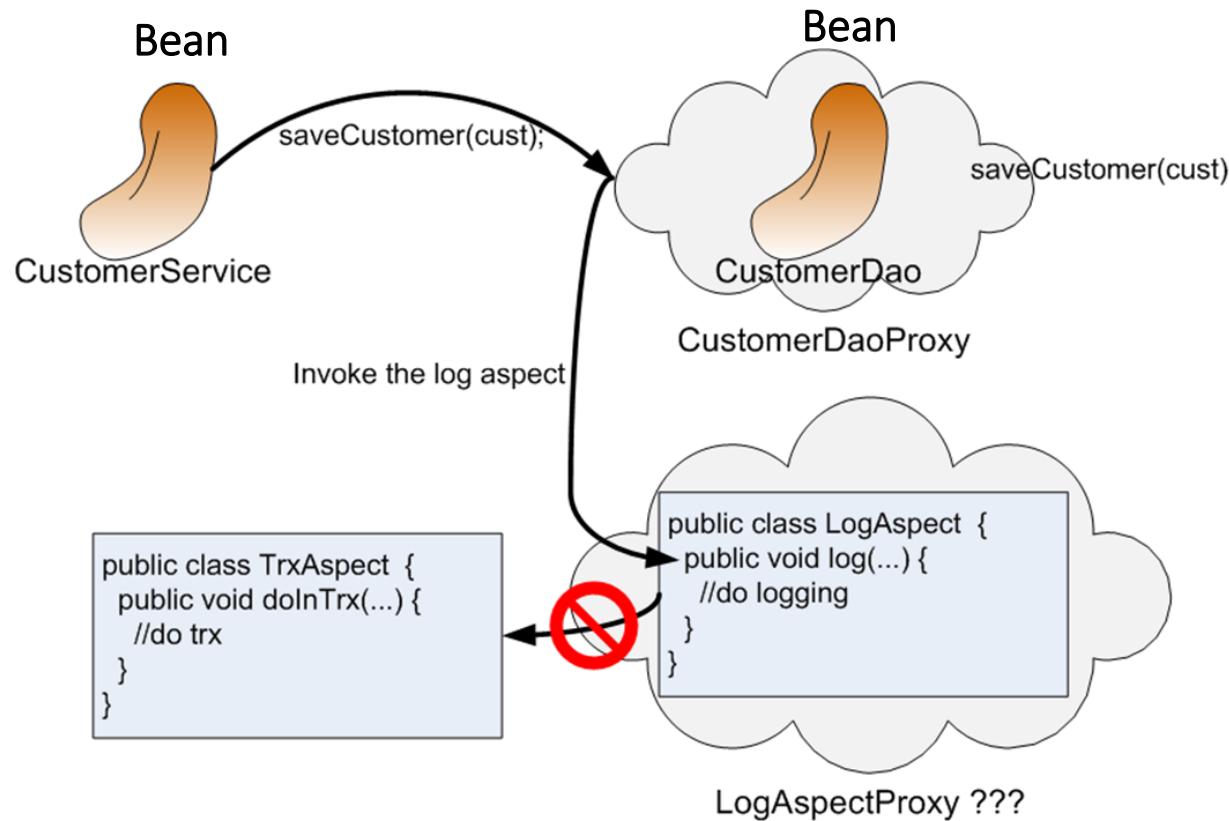
## Dynamic Proxy

Now, on a call to a save method in the DAO, the proxy intercepts the call and routes it to the appropriate advice method in the aspect class.



# AOP

In Spring AOP, it is not possible to have aspects themselves be the target of advice from other aspects.





# AOP

Pretty complicated..

Spring AOP comes to make it easier





# AOP

Spring provides AOP by Annotations or XML configuration

Aspect-oriented programming may include:

## Cross-cutting concerns

For example, we may want to add logging information to classes within the data-access layer (DAO) and also to classes in the UI layer

## Target-method(s)

Method(s) require additional logic (advice) to be embedded in.



## Advice - The additional external code

The additional external code we want to apply to our existing model.

In our Banking example, this is the security and logging code we want to apply whenever banking methods are called.

This is the external method we need to add to our target methods.

## Jointcut (destination) – Defines candidates (targets) for aspects

These might be single or multiple methods in a class or all/some methods in given classes or all/some methods in a given package

## Pointcut (criteria) – Defines Where to apply the advice

A capturing term/condition or expression defines which cross-cutting concern needs to be targeted with an advice code + timing (before, after, around method) to a Joinpoint

In our example, a pointcut is matched when calling some banking method as deposit() , withdraw(), openAccount() , transfer() etc.

## Aspect

The combination of the pointcut and the advice



## ASPECT Configuration

There are **5 Pointcut** designators:

- execution - which methods, with which signatures, in what classes and packages advices will be applied to
- within - Applies advice to all classes within a specified package.
- target - Applies advice to all methods in a class:
- args - Applies advice to all methods that accept the specified argument[s] of specified type
- bean - Applies advice to a specific bean identified specific id
- @annotation - Applies advice to a custom annotation



## execution

“**execution**” pointcut-designator is the most popular and powerful Pointcut-Designator, specify which methods, with what signatures, in what classes and packages advices will be applied to.

The “**execution**” **pointcut format** is as follows:

execution (return-type-pattern naming-pattern(params-pattern) [throws-pattern])

**Wildcards** are widely supported: \*(..)



## execution

More “execution” Pointcut-designator definition EXAMPLES:

1. Any method: execution(\* \*(..))
2. Any method in any \*Db class in the com.jbt.db package:  
    execution(\* com.jbt.db.\*Db. \*(..))
3. Any method that throws an exception with Number in its name:  
    execution(\* \*(..) throws \*.Number\*)
4. Any method that starts with “set..”:  
    execution(\* set\*(..))
5. Any get method in any class within the com.jbt.db package that takes long as its first parameter:  
    execution( \* com.jbt.db.\*.get\*(long, ..) )



Spring AOP allows us surround some basic logic execution

### The method parameter pattern

- "()" - method matches only if it takes no parameters.
- "(..)" - method with zero or more parameters.
- "(\*)" - only methods with one parameter of any type matches the criteria.
- "(\*, long)" - only methods with two parameters:
  - first of any type
  - second long type.



## execution examples:

More “execution” Pointcut-designator definition EXAMPLES:

1. execution(\* com.howtodoinjava.EmployeeManager.\*(..))

Matches :

- All methods declared in the EmployeeManager interface/class.
- And any return type.
- The two dots in the argument list match any number of arguments.

2. execution(public \* EmployeeManager.\*(..))

Matching all public methods in EmployeeManager



## execution examples:

More “execution” Pointcut-designator definition *EXAMPLES*:

1. Matching all public methods in EmployeeManager with return type

EmployeeDTO

execution(public EmployeeDTO EmployeeManager.\*(..))

2. Matching all public methods in EmployeeManager with return type

EmployeeDTO and first parameter as EmployeeDTO

execution(public EmployeeDTO EmployeeManager.\*(EmployeeDTO, ..))

3. Intercept everything but ‘get’ methods in a EmployeeDTO

target(com.jbt.EmployeeDTO) && !execution(\* get\*(..))“

target Applies advice to all methods in a class



## execution - timing

Spring AOP allows us surround some basic logic execution with additional external code (called ADVICE) in 5 execution timings :

- before                   - advice method will be executed before target method
- after                   - advice method will be executed after target method
- around                  - advice method will be executed before and after target method
- after-returning        - advice method will be executed after target method returned. Advice method will receive the target method's returned value.
- after-throwing         - advice method will be executed after target method throw an exception.



## execution - timing

Spring AOP AspectJ Annotation style allow you to intercept method easily with Common AspectJ annotations :

- @Before** – Runs before the method execution
- @After** – Runs after the method returned a result
- @AfterReturning** – Runs after the method returned a result, intercept the returned result as well.
- @AfterThrowing** – Run after the method throws an exception
- @Around** – Run around the method execution, combine all three advices above.



# AOP

## Enabling @AspectJ Support with Java configuration

To enable @AspectJ support with Java @Configuration add the **@EnableAspectJAutoProxy** annotation:

```
@Configuration  
@EnableAspectJAutoProxy  
public class AppConfig {  
....  
}
```

To enable @AspectJ support with XML based configuration use the aop:aspectj-autoproxy element: <aop:aspectj-autoproxy/>



## Spring Aspect Required Maven dependencies

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-aspects</artifactId>
        <version>${spring.version}</version>
    </dependency>
</dependencies>
```



# AOP

## The Bean (target) Class

### Simple AOP Example

```
@Component("myBean")
public class MyBean {

    @Resource
    private String s1;
    @Resource
    private String s2;

    @PostConstruct
    public void init(){
        System.out.println("MyBean.init() - s1:[" + s1 + "]");
        System.out.println("MyBean.init() - s2:[" + s2 + "]");
    }
}
```

CONT

```
@PreDestroy
public void destroy(){
    System.out.println("MyBean.destroy()");
}

@Override
public String toString() {
    return "MyBean [s1=" + s1 + ", s2=" + s2 + "]";
}
public String testAspectCall(int i){
    return "Hello-" + testAspectCallInt(i);
}
public Integer testAspectCallInt(int i){
    System.out.println("MyBean.testAspectCallInt(i="
                      + i + ")");
    return i + 1000;
}
```



## The Configuration Class

```
@Configuration  
@ComponentScan(basePackages="com.spring.spring.aop")  
@EnableAspectJAutoProxy  
public class SpringConfig {  
}
```



# AOP

@Component

@Aspect

```
public class SystemAspect {
```

```
@Before("execution( java.lang.Integer com.spring.spring.aop.MyBean.*(..))")
```

```
public void doAccessCheck(org.aspectj.lang.JoinPoint jp) {
```

```
    System.out.println("Before ASPECT: " + jp);
```

```
    Signature sign = jp.getSignature();
```

```
    System.out.println("\t\tMethod signature:" + sign );
```

```
    System.out.println("\t\tMethod signature modifiers:" + sign.getModifiers() ); //java.lang.reflect.Modifier
```

```
    System.out.println("\t\tMethod signature DeclaringTypeName:" + sign.getDeclaringTypeName() );
```

```
    System.out.println("\t\tMethod signature DeclaringType:" + sign.getDeclaringType() );
```

```
    System.out.println("\t\tkind: " + jp.getKind());
```

```
    System.out.println("\t\tTarget: " + jp.getTarget().getClass().getSimpleName());
```

```
    System.out.println("\t\tThis: " + jp.getThis().getClass().getSimpleName());
```

```
    Object [] args = jp.getArgs();
```

```
    for (int i = 0; i < args.length; i++) {
```

```
        System.out.println("\t\targ:" + args[i].getClass().getSimpleName() + " value=" + args[i]);
```

```
        args[i] = new Integer(99);
```

```
    }
```

```
    jp.proceed();
```

## The Aspect Classes

# The Aspect Classes

`@Component`  
`@Aspect`

```
public class AfterReturningAspect {
```

## Method Result



## The Aspect Classes

```
@Component
@Aspect
public class AroundAspect {
    @Around("execution(java.lang.Integer com.spring.spring.aop.basic.MyBean.*(..)) &&
        args(java.lang.Integer)")
    public void checkAroundSetter(final ProceedingJoinPoint pjp) throws Throwable {
        System.out.println("AroundAspect befor:" + pjp.getSignature().toLongString());
        Object[] args = pjp.getArgs();
        printAndChange(args);
        Object targetMethodResult = pjp.proceed(args);
        System.out.println("AroundAspect - AFTER - targetMethodResult:" + targetMethodResult);
        print(args);
        pjp.proceed();
    }
}
```

### NOTE

- The @Arround method should accept a **ProceedingJoinPoint** as first parameter.
- The @Arround method **should call the proceed method** of this object to execute the target method or the next Aspect method on the chain.



# AOP

## The Aspect Classes

### Around Aspect CONT

```
private void print(Object[] args) {  
    Arrays.stream(args).forEach(a->{System.out.println("AroundAspect AFTER arg:" + a);});  
}
```

```
private void printAndChange(Object[] args) {  
    for (int i = 0; i < args.length; i++) {  
        System.out.println("AroundAspect BEFORE - args[" + i + "]: " + args[i] + " changing to 99..");  
        if (args[i] instanceof Integer ){  
            args[i] = new Integer(99); // CHANGE THE VALUE  
        }  
    } /*for*/  
} /*method*/  
} /*class*/
```



## The main Class

```
public class Test {  
    public static void main(String[] args) {  
        ApplicationContext ctx = new AnnotationConfigApplicationContext(SpringConfig.class);  
  
        MyBean bean = (MyBean)ctx.getBean("myBean");  
        String str = bean.testAspectCall(12); // calling method  
        System.out.println(str);  
        bean.testAspectCallInt(12); // calling method  
        System.out.println("Test.main() - bean:" + bean);  
    }  
}
```



# AOP

## Program Output

AroundAspect befor:public java.lang.Integer com.spring.spring.aop.basic.MyBean.testAspectCallInt(int)

AroundAspect BEFORE - args[0]:12 changing to 99..

**Before ASPECT:** execution(Integer com.spring.spring.aop.basic.MyBean.testAspectCallInt(int))

Method signature:Integer com.spring.spring.aop.basic.MyBean.testAspectCallInt(int)

Method signature modifiers:1

Method signature DeclaringTypeName:com.spring.spring.aop.basic.MyBean

Method signature DeclaringType:class com.spring.spring.aop.basic.MyBean

kind: method-execution

Target: MyBean

This: MyBean\$\$EnhancerBySpringCGLIB\$\$a0d40555

arg:Integer value=99

MyBean.testAspectCallInt(i=99)

AroundAspect - AFTER - targetMethodResult:1099

AroundAspect AFTER arg:99

**AfterReturningAspect** ASPECT: execution(Integer com.spring.spring.aop.basic.MyBean.testAspectCallInt(int))

**AfterReturningAspect** Returned null as return value.

to get results Disable AroundAspect

Test.main() - bean:MyBean [s1>Hello 3, s2>Hello 3] MyBean.destroy()



## private/protected methods interception

Due to the proxy-based nature of Spring's AOP framework, protected methods are by definition not intercepted, neither for JDK proxies (where this isn't applicable) nor for CGLIB proxies (where this is technically possible but not recommendable for AOP purposes).

As a consequence, any given pointcut will be matched against **public methods only!**

If your interception needs include protected/private methods or even constructors, consider the use of Spring-driven **native AspectJ** weaving instead of Spring's proxy-based AOP framework.

This constitutes a different mode of AOP usage with different characteristics, so be sure to make yourself familiar with weaving first before making a decision.



# AOP

## AOP Custom Annotation Capturing



## AOP Custom Annotation Capturing

Spring AOP supports METHOD and CLASS level Custom Annotation Capturing  
Annotation capturing is performed ONLY when calling a method of the class

EXAMPLE - A Custom Annotation :

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.METHOD})
public @interface MyAnnotation {
    String value()    default "";
    String key()     default "";
    String condition() default "";
}
```



## AOP Custom Annotation Capturing

```
@Component("myBean")
public class MyBean {

    public MyBean() { }

    public String testAspectCall(int i){
        return "Hello-" + testAspectCallInt(i);
    }

    @MyAnnotation(value="valtest", key="keytest", condition="contest")
    public Integer testAspectCallInt(int i){
        System.out.println("MyBean.testAspectCallInt(i=" + i + ")");
        return i+1000;
    }
}
```



## AOP Custom Annotation Capturing

```
@Component
```

```
@Aspect
```

```
public class MyArroundAspect{
```

```
    @Around("@annotation(MyAnnotation)")
```

```
    public Object testAspectCallInt (ProceedingJoinPoint jointPoint) throws Throwable {
```

```
        System.out.println("MyArroundAspect in." );
```

```
        Annotation [] annotations = jointPoint.getTarget().getClass().getDeclaredAnnotations();
```

```
        for (Annotation annotation : annotations) {
```

```
            System.out.println("MyArroundAspect: annotation" + annotation.toString() );
```

```
}
```

```
        return jointPoint.proceed();
```

```
}
```

```
}
```



## AOP Custom Annotation Capturing

```
public class Test {  
    public static void main(String[] args) {  
        ApplicationContext ctx = new AnnotationConfigApplicationContext(SpringConfig.class);  
  
        MyBean bean = (MyBean)ctx.getBean("myBean");  
        MyBean2 bean2 = (MyBean2)ctx.getBean("myBean2");  
        MyBean3 bean3 = (MyBean3)ctx.getBean("myBean3");  
  
        bean.testAspectCallInt(12); // calling method  
        bean2.testAspectCallInt(12); // calling method  
        bean3.testAspectCallInt(12); // calling method  
        System.out.println("Test.main() - bean:" + bean);  
    }  
}
```



## AOP Custom Annotation Capturing OUTPUT

MyArroundAspect in.

MyArroundAspect: annotation@org.springframework.stereotype.Component(value=myBean)

MyBean.testAspectCallInt(i=12)

MyBean2.testAspectCallInt(i=12)

MyBean3.testAspectCallInt(i=12)

Test.main() - bean:com.spring.spring.aop.annotate.MyBean@7d20d0b



# AOP

## AOP Custom Annotation - get Annotation attributes values..



# AOP

To get annotation attributes values, instead of @Around("@annotation(MyAnnotation)") do..

```
@Around("execution(@com.spring.spring.aop.annotate.MyAnnotation * *(..))  
      &&@annotation(myAnnotation)")
```

```
@Component
```

```
@Aspect
```

```
public class MyArroundAspect2{
```

```
@Around("execution(@com.spring.spring.aop.annotate.MyAnnotation * *(..)) && @annotation(myAnnotation)")  
public Object process(ProceedingJoinPoint jointPoint, MyAnnotation myAnnotation) throws Throwable {  
    System.out.println("MyArroundAspect-2: myAnnotation:" + jointPoint.getTarget().getClass().getSimpleName());  
    System.out.println("MyArroundAspect-2: myAnnotation condition:" + myAnnotation.condition());  
    System.out.println("MyArroundAspect-2: myAnnotation key:" + myAnnotation.key());  
    System.out.println("MyArroundAspect-2: myAnnotation value:" + myAnnotation.value());
```

*CONT..*



# AOP

## AOP Custom Annotation - get Annotation attributes values..

*CONT..*

```
Annotation [] annotations = jointPoint.getTarget().getClass().getDeclaredAnnotations();
for (Annotation annotation : annotations) {
    System.out.println("MyArroundAspect-2: annotation" + annotation.toString() );
}
return jointPoint.proceed();
}
```



# AOP

AOP Custom Annotation - get Annotation attributes values..

## OUTPUT

MyArroundAspect-2: myAnnotation:MyBean

MyArroundAspect-2: myAnnotation condition:contest

MyArroundAspect-2: myAnnotation key:keytest

MyArroundAspect-2: myAnnotation value:valtest

MyArroundAspect-2: annotation@org.springframework.stereotype.Component(value=myBean)

MyBean.testAspectCallInt(i=12)

MyBean2.testAspectCallInt(i=12)

MyBean3.testAspectCallInt(i=12)

Test.main() - bean:com.spring.spring.aop.annotate.MyBean@33c911a1



# AOP

## AOP **class level** Custom Annotation



## AOP class level Custom Annotation

```
@Component
```

```
@Aspect
```

```
public class MyArroundAspect3{
```

```
    @Around("@target(myAnnotation)")
```

```
    public Object process(ProceedingJoinPoint jointPoint, MyAnnotation myAnnotation) throws Throwable {
```

```
        System.out.println("MyArroundAspect-3: myAnnotation condition:" +
```

```
                           jointPoint.getTarget().getClass().getSimpleName() );
```

```
        System.out.println("MyArroundAspect-3: myAnnotation condition:" + myAnnotation.condition());
```

```
        System.out.println("MyArroundAspect-3: myAnnotation key:" + myAnnotation.key());
```

```
        System.out.println("MyArroundAspect-3: myAnnotation value:" + myAnnotation.value());
```

```
        return jointPoint.proceed();
```

```
}
```

```
}
```



## AOP class level Custom Annotation

```
@Component("myBean2")
@MyAnnotation(value="valtest-classLevel2", key="keytest-classLevel2", condition="contest-classLevel2")
public class MyBean2 {
    public Integer testAspectCallInt(int i){
        System.out.println("MyBean2.testAspectCallInt(i=" + i + ")");
        return i+1000;
    }
}

@Component("myBean3")
@MyAnnotation(value="valtest-classLevel3", key="keytest-classLevel3", condition="contest-classLevel3")
public class MyBean3 {
    public Integer testAspectCallInt(int i){
        System.out.println("MyBean3.testAspectCallInt(i=" + i + ")");
        return i+1000;
    }
}
```



## AOP class level Custom Annotation

### OUTPUT

```
MyBean.testAspectCallInt(i=12)
MyArroundAspect-3: myAnnotation condition:MyBean2
MyArroundAspect-3: myAnnotation condition:contest-classLevel2
MyArroundAspect-3: myAnnotation key:keytest-classLevel2
MyArroundAspect-3: myAnnotation value:valtest-classLevel2
MyBean2.testAspectCallInt(i=12)
```

```
MyArroundAspect-3: myAnnotation condition:MyBean3
MyArroundAspect-3: myAnnotation condition:contest-classLevel3
MyArroundAspect-3: myAnnotation key:keytest-classLevel3
MyArroundAspect-3: myAnnotation value:valtest-classLevel3
MyBean3.testAspectCallInt(i=12)
Test.main() - bean:com.spring.spring.aop.annotate.MyBean@131ef10
```



## Ordering aspects

How can we specify interception order when multiple aspects share same pointcuts?

- Each Aspect may have ordinal value
- It is set via @Order annotation
- Example:

```
@Aspect  
@Order(1)  
public class AspectA  
{  
    @Before(".....")  
    public void doit() {}  
}
```

```
@Aspect  
@Order(2)  
public class AspectB  
{  
    @Before(".....")  
    public void doit() {}  
}
```



# AOP

## External pointcut:



## External pointcut:

- Sometimes we need to externalize the pointcut-designator to be common for multiple

Aspect methods.

- In that case we define a **@Poincut** annotation above an empty method in the Aspect class annotated by **@Poincut** annotation and use this method name in multiple pointcut-designators..

**@Component**

**@Aspect**

```
public class AspectClass{  
    @Poincut ("execution(* com.jbt.hr.Person.*(..))") // pointcut designator  
    private void myEmptyVoidMethod(){} // Empty void method
```



## External pointcut:

- Using the externalize the pointcut-designator

```
@Around("myEmptyVoidMethod()") // ← the name of the empty void method uses as pointcut-designator
public Object someArroundMethod (ProceedingJoinPoint pjp) throws Throwable {
    System.out.println("Before calling target method:" + pjp.toString());
    Object o = pjp.proceed();
    System.out.println("After calling target method:" + pjp.toString());
    return o;
}
```

```
@Before("myEmptyVoidMethod()") // ← again the name of the empty void method
public Object someArroundMethod (JoinPoint jp) throws Throwable {
    System.out.println("Before calling target method:" + pjp.toString());
}
```



## External pointcut:

Example

```
@Component("myBean")
public class MyBean {
    public String testAspectCall(int i){
        return "Hello-" + i;
    }
    public Integer testAspectCallInt(int i){
        Integer ret = i + 1000;
        System.out.println("MyBean.testAspectCallInt(got i=" + i + "), returning ret:[" + ret + "]");
        return ret;
    }
}
```



## External pointcut:

Example

```
@Component
@Aspect
public class MyExtrnPoincutAspect{

    @Pointcut(value="execution(public Integer *(..))")
    public void anyPublicMethod() { } // Empty method

    @Before("anyPublicMethod()") // Usage-1
    public void processBefore(JoinPoint jp) throws Throwable {
        System.out.println("MyExtrnPoincutAspect.processBefore(). args: " +
                           Arrays.toString(jp.getArgs()));
    }
}
```



## External pointcut:

Example - CONT

```
@Around("anyPublicMethod()") // Usage-2
public void processAround(ProceedingJoinPoint pjp) throws Throwable {
    System.out.println("MyExtrnPoincutAspect.processAround() - BEFORE target method");
    Object targetMethodRetVal = pjp.proceed();
    System.out.println("MyExtrnPoincutAspect.processAround() - AFTER target method.
        targetMethodRetVal:[" + targetMethodRetVal + "]");
}
```



# JDBC

## SPRING DAO TEMPLATES

Examples : springbasic1 project, package: com.jbt.spring.db.jdbc.templates



# Jdbc

There is a lot of code to write prior to implement a simple database query..

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Connection con = DriverManager.getConnection("jdbc:odbc:personDBLogicName");
Statement stm = con.createStatement();
ResultSet rs = stm.executeQuery("Select * from person");

while (rs.next()) {
    System.out.println(rs.getInt(1)
        + " " + rs.getString(2)
        + " " + rs.getInt(3));
}
// while
```



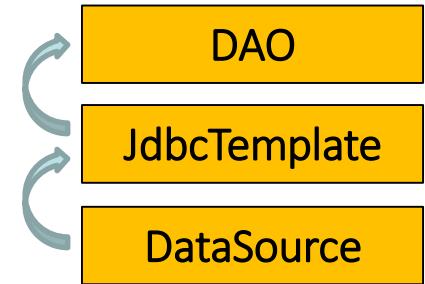
# Jdbc- JdbcTemplate

Spring offers Template Classes to simplify JDBC tasks

- JdbcTemplate – basic, with ‘?’ Jdbc –like Place holders
- NamedParameterJdbcTemplate - replaces ‘?’ With ‘:name1’, ‘:name2’
- SimpleJdbcTemplate

The idea is to :

1. Create a DS Bean
2. Create a templates @Bean and inject the DS right to this bean.
3. Create your DAO @Bean and inject the template bean to your DAO.
4. From your DAO methods, call the template methods for insert, delete, update, select activities.



The whole difference between these templates is in the way arguments are provided to the template's C.R.U.D methods.



# Jdbc

## STEP 1

```
CREATE TABLE employees(  
    id      int(11) NOT NULL AUTO_INCREMENT,  
    age     int(11) NOT NULL,  
    name   varchar(255) DEFAULT NULL,  
    PRIMARY KEY (id)  
)
```



# Jdbc

## STEP 2 - (Add Spring JDBC and MySQL dependencies)

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
    <version> ... </version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version> ...</version>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version> ... </version>
</dependency>
```



# Jdbc

## STEP 3: (Create Data Transfer Object)

```
public class Employee {  
  
    private int id;  
    private String name;  
    private int age;  
  
    public int getId() {  
        return id;  
    }  
    public void setId(int id) {  
        this.id = id;  
    }  
  
    public String getName() {  
        return name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
    public int getAge() {  
        return age;  
    }  
    public void setAge(int age) {  
        this.age = age;  
    }  
}
```



# Jdbc

## STEP 4: (Create a configuration class)

```
@Configuration
@ComponentScan(basePackages = "com.jbt.spring.db.jdbc.template")
public class SpringConfig {

    @Bean
    public DataSource dataSource() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName("com.mysql.jdbc.Driver");
        dataSource.setUrl("jdbc:mysql://localhost:3306/TestDB");//change url
        dataSource.setUsername("root"); //change userid
        dataSource.setPassword("root"); //change pwd
        return dataSource;
    }
}
```



# Jdbc

## STEP 4: (Create a configuration class)

```
@Bean
```

```
public JdbcTemplate jdbcTemplate() {  
    JdbcTemplate jdbcTemplate = new JdbcTemplate();  
    jdbcTemplate.setDataSource(dataSource());  
    return jdbcTemplate;  
}
```

```
}
```



# Jdbc

## Step 5: (Create DAO classes)

```
public interface EmployeeDAO {  
    public String          getEmployeeNameById(int id);  
    public List<Employee> getAllEmployees();  
    public List<Employee> getEmployeesByNameLike(String name);  
    public int             savePerson(Employee e);  
}
```



# Jdbc

## Step 5: (Create DAO classes)

```
@Repository
public class EmployeeDAOImpl implements EmployeeDAO {

    private JdbcTemplate jdbcTemplate;

    @Resource
    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    public String getEmployeeNameById(int id) {
        String sql = "select name from employees where id = ?";
        String name = jdbcTemplate.queryForObject(sql, new Object[] { id }, String.class);
        return name;
    }
}
```



# Jdbc

## Step 5: (Create DAO classes)

```
public List<Employee> getAllEmployees() {  
    String sql = "select * from Employees";  
    Object[] args = null; // { id } ;  
    List<Employee> list = jdbcTemplate.query(sql, args, new EmployeeRowMapper());  
    return list;  
}  
  
public List<Employee> getEmployeesByNameLike(String name) {  
    String sql = "select * from Employees where name like ?";  
    name = "%" + name + "%";  
    Object[] args = { name };  
    List<Employee> list = jdbcTemplate.query(sql, args, new EmployeeRowMapper());  
    return list;  
}
```



# Jdbc

## Step 5: (Create DAO classes)

```
public int savePerson(Employee e) {  
    String sql = "insert into Employees(name, age) values (?,?)";  
    Object[] args = { e.getName(), e.getAge() };  
    int[] types = { Types.VARCHAR, Types.INTEGER };  
    int insertCount = jdbcTemplate.update(sql, args, types);  
    return insertCount;  
}  
  
public int updatePerson(Employee e) {  
    String sql = "update Employees set name=?, age=? where id=?";  
    Object[] args = { e.getName(), e.getAge(), e.getId() };  
    int[] types = { Types.VARCHAR, Types.INTEGER, Types.INTEGER };  
    int insertCount = jdbcTemplate.update(sql, args, types);  
    return insertCount;  
}}
```



# Jdbc

RowMapper – maps data from ResultSet to Value's object (bean's) attributes

PersonRowMapper is a Mapper class

- implements the org.springframework.jdbc.core.RowMapper Spring interface
- map a ResultSet row to a Person object structure by implementing **mapRow(..)** method

```
public class EmployeeRowMapper implements RowMapper<Employee> {  
  
    public Employee mapRow(ResultSet rs, int index) throws SQLException {  
        Employee employee = new Employee();  
        employee.setId ( rs.getInt ("id") );  
        employee.setName( rs.getString("name"));  
        employee.setAge ( rs.getInt ("age") );  
        return employee;  
    }  
}
```



# Jdbc

## Step 6: (Main Application class)

```
public static void main(String[] args) {
    ApplicationContext ctx=
        new AnnotationConfigApplicationContext(SpringConfig.class);
    EmployeeDAO empDAO = ctx.getBean(EmployeeDAO.class);
    String empName = empDAO.getEmployeeNameById(111);
    System.out.println("Employee 111 name is " + empName);

    System.out.println("\nEmployees List:");
    List<Employee> list = empDAO.getAllEmployees();
    for (Employee employee : list) {
        System.out.println(employee);
    }

    System.out.println("\nGet Employees List by name like:");
    list = empDAO.getEmployeesByNameLike("v");
    System.out.println(list);
}

}
```



# Jdbc

## Step 6: (Main Application class)

```
System.out.println("\nGet Employees List by name like:");
list = empDAO.getEmployeesByNameLike("v");
System.out.println(list);
}
}
```



# Jdbc

## Additional JdbcTemplate capabilities

To retrieve person name you can use `queryForObject` method:

```
int id = 123;
String name = (String) jdbcTemplate.queryForObject("select name from Employees where id = ?",
                                                 new Object [] {id},
                                                 String.class);
```

Or table row count :

```
int numRows = jdbcTemplate.queryForInt("select count(*) from Person");
```



Jdbc -

## NamedParameterJdbcTemplate

In multiple parameter long queries It is confusing to use many '?' marks in a query  
Common mistakes occur when number of values do not match the exact number of '?

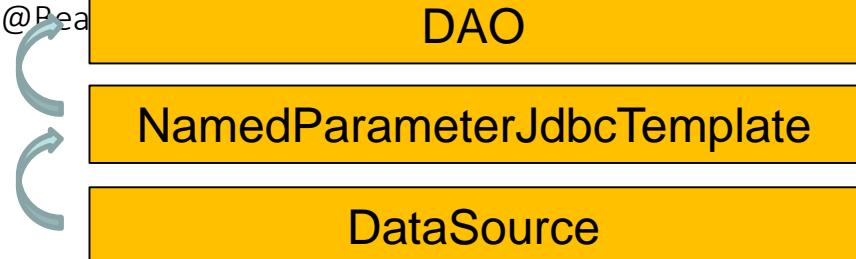
To help with that, Spring comes out with the second template :  
**NamedParameterJdbcTemplate**

The entire configuration doesn't change a lot:

Declare your DataSource as a @Bean

Declare NamedParameterJdbcTemplate as a @Bean and inject the DS as constructor args

Inject the NamedParameterJdbcTemplate @Bean to your DAO @Bean





Jdbc -

# NamedParameterJdbcTemplate

Change to Configuration class:

```
@Configuration
```

```
...
```

```
...
```

```
@Bean
```

```
public NamedParameterJdbcTemplate namedParameterJdbcTemplate() {  
    NamedParameterJdbcTemplate namedParameterJdbcTemplate =  
        new NamedParameterJdbcTemplate( dataSource() );  
    return namedParameterJdbcTemplate;  
}
```



## Jdbc -

# NamedParameterJdbcTemplate

@Repository

```
public class EmployeeDAOImpl implements EmployeeDAO {
```

```
    private NamedParameterJdbcTemplate namedParameterJdbcTemplate;
```

@Resource

```
    public void setNamedParameterJdbcTemplate(NamedParameterJdbcTemplate  
                                              namedParameterJdbcTemplate) {  
        this.namedParameterJdbcTemplate = namedParameterJdbcTemplate;  
    }
```

@Override

```
    public int saveEmployee(Employee employee) {  
        String SQL = "INSERT INTO Employees (name, age) VALUES (:name, :age)";  
        Map namedParameters = new HashMap();  
        namedParameters.put("name", employee.getName());  
        namedParameters.put("age", employee.getAge());  
        int countEffected = namedParameterJdbcTemplate.update(SQL, namedParameters);  
        System.out.println("Created Row Name = " + employee.getName() + " Age = " +  
                           employee.getAge());  
        return countEffected;  
    }
```



## Jdbc -

# NamedParameterJdbcTemplate

```
@Override
public Employee getEmployeeById(int empid) {
    String SQL = "SELECT * FROM Employees WHERE id = :id";
    SqlParameterSource namedParameters = new MapSqlParameterSource("id", Integer.valueOf(empid));
    Employee employee = (Employee) namedParameterJdbcTemplate.queryForObject(SQL,
namedParameters, new EmployeeRowMapper());
    return employee;
}
```

```
@Override
public List<Employee> getAllEmployees() {
    String SQL = "SELECT * FROM Employees";
    List<Employee> employees = (List<Employee>) namedParameterJdbcTemplate.query(SQL,
                                            new EmployeeRowMapper() );
    return employees;
}
```



## Jdbc -

# NamedParameterJdbcTemplate

```
@Override
public void deleteEmployee(Integer empid) {
    String SQL = "DELETE FROM Employees WHERE id = :empid";
    SqlParameterSource namedParameters = new MapSqlParameterSource("empid", Integer.valueOf(empid));
    namedParameterJdbcTemplate.update(SQL, namedParameters );
    System.out.println("Deleted Record with EMPID = " + empid);
}

@Override
public void update(Integer empid, Integer age) {
    String SQL = "UPDATE Employees SET age = :age WHERE id = :empid";
    SqlParameterSource namedParameters = new MapSqlParameterSource();
    ((MapSqlParameterSource) namedParameters ).addValue("age", age);
    ((MapSqlParameterSource) namedParameters ).addValue("empid", empid);
    int countEffected = namedParameterJdbcTemplate.update(SQL, namedParameters );
    System.out.println("Updated " + countEffected + " rows with EMPID = " + empid);
}
```



# spring data jdbc





# SPRING BOOT

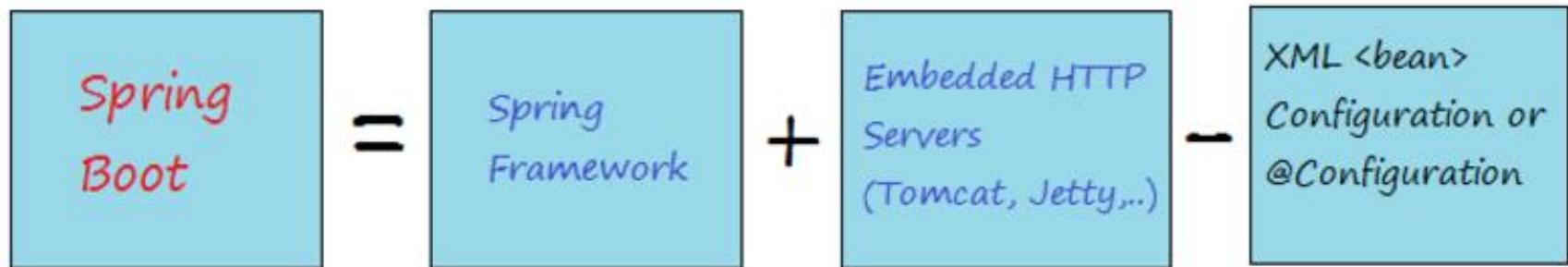
Examples: project: springbasic3-Boot

## What is Spring Boot ?

- **Spring Boot** is the next step of Spring to make Spring easier setting-up and developing applications.
- For Spring Boot, Spring configuration are minimized.
- **Spring Boot** supports embedded containers (Tomcat, Jetty) in allowing web applications to be able to run independently without deploying on **Web Server**.
- You can use spring boot to launch Java Web application via command line '`java -jar`' or export to war file to deploy on Web Server as usual.

# Spring Boot

Spring Boot can be explained simply by the illustration below:



*Spring Boot is formed from the idea of **simplifying** the Spring Framework.*

- It is very easy to develop Spring Based applications with Java or Groovy.
- It reduces lots of development time and increases productivity.
- It avoids writing lots of boilerplate Code, Annotations and XML Configuration.
- It provides Embedded HTTP servers like Tomcat, Jetty etc. to develop and test our web applications very easily.

*Spring Boot is formed from the idea of **simplifying** the Spring Framework.*

- It is very easy to develop Spring Based applications with Java or Groovy.
- It reduces lots of development time and increases productivity.
- It avoids writing lots of boilerplate Code, Annotations and XML Configuration.
- It provides Embedded HTTP servers like Tomcat, Jetty etc. to develop and test our web applications very easily.

## pom.xml

There are 3 key points in pom.xml (1),(2), (3) as shown below,

Spring Boot supports you in simplifying the declaration of Spring libraries.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2<project xmlns="http://maven.apache.org/POM/4.0.0"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
5                           http://maven.apache.org/xsd/maven-4.0.0.xsd">
6
7   <modelVersion>4.0.0</modelVersion>
8
9   <groupId>org.o7planning</groupId>
10  <artifactId>HelloSpringBoot</artifactId>
11  <version>0.0.1-SNAPSHOT</version>
12  <packaging>jar</packaging>
13
14 <name>HelloSpringBoot</name>
15 <description>Demo project for Spring Boot</description>
16
17<parent>
18   <groupId>org.springframework.boot</groupId>
19   <artifactId>spring-boot-starter-parent</artifactId>
20   <version>1.4.1.RELEASE</version>
21   <relativePath/> <!-- lookup parent from repository -->
22 </parent>
23
24<properties>
25   <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
26   <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
27   <java.version>1.8</java.version>
28 </properties>
```

pom.xml

(1)

## 1 - spring-boot-starter-parent

- **spring-boot-starter-parent** is an available project in **Spring Boot**.
- The dependent libraries is declared in **spring-boot-starter-parent**, your project only inherit it.
- You only need to declare **<parent>** in file **pom.xml** of your project.

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>1.4.1.RELEASE</version>
</parent>
```

## 2 - spring-boot-starter-web

Since we are developing a web application, we will add a **spring-boot-starter-web** dependency

Other “**Starters**” simply provide dependencies that you are likely to need when developing a specific type of application.

```
29
30<
31<dependencies>
32     <dependency>
33         <groupId>org.springframework.boot</groupId>
34         <artifactId>spring-boot-starter-web</artifactId>
35     </dependency>
36
37     <dependency>
38         <groupId>org.springframework.boot</groupId>
39         <artifactId>spring-boot-starter-test</artifactId>
40         <scope>test</scope>
41     </dependency>
42 </dependencies>
43
44<build>
45     <plugins>
46         <plugin>
47             <groupId>org.springframework.boot</groupId>
48             <artifactId>spring-boot-maven-plugin</artifactId>
49         </plugin>
50     </plugins>
51 </build>
52
53 </project>
```

(2)

(3)



# Spring Boot

## 3 - spring-boot-maven-plugin

is the plugin providing necessary libraries that helps your project to be able to run directly without deploying on a Web Server. It helps to create an executable jar file.

```
29
30    <dependencies>
31        <dependency>
32            <groupId>org.springframework.boot</groupId>
33            <artifactId>spring-boot-starter-web</artifactId>
34        </dependency>
35
36        <dependency>
37            <groupId>org.springframework.boot</groupId>
38            <artifactId>spring-boot-starter-test</artifactId>
39            <scope>test</scope>
40        </dependency>
41    </dependencies>
42
43    <build>
44        <plugins>
45            <plugin>
46                <groupId>org.springframework.boot</groupId>
47                <artifactId>spring-boot-maven-plugin</artifactId>
48            </plugin>
49        </plugins>
50    </build>
51
52
53 </project>
```

(2)

(3)

## How Spring Boot runs?

Spring Boot starts within a main() method of a POJO class annotated with @SpringBootApplication

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

```
@SpringBootApplication  
public class HelloSpringBootApplication {  
  
    public static void main(String[] args) {  
        SpringApplication.run(HelloSpringBootApplication.class, args);  
    }  
  
}
```

The `@SpringBootApplication` annotation

Is equivalent to using

- `@Configuration`
- + `@EnableAutoConfiguration`
- + `@ComponentScan`

with their default attributes:

## @EnableAutoConfiguration

Enables auto-configuration of the Spring Application Context, attempting to guess and configure beans that you are likely to need.

Auto-configuration classes are usually applied based on your classpath and what beans you have defined.

For example, If you have tomcat-embedded.jar on your classpath you are likely to want a TomcatEmbeddedServletContainerFactory  
(unless you have defined your own EmbeddedServletContainerFactory bean).

Therefore, **@SpringBootApplication** supports you in automatically configuring Spring, and automatically scanning entire project in order to find out Spring components (Controller, Bean, Service,...)

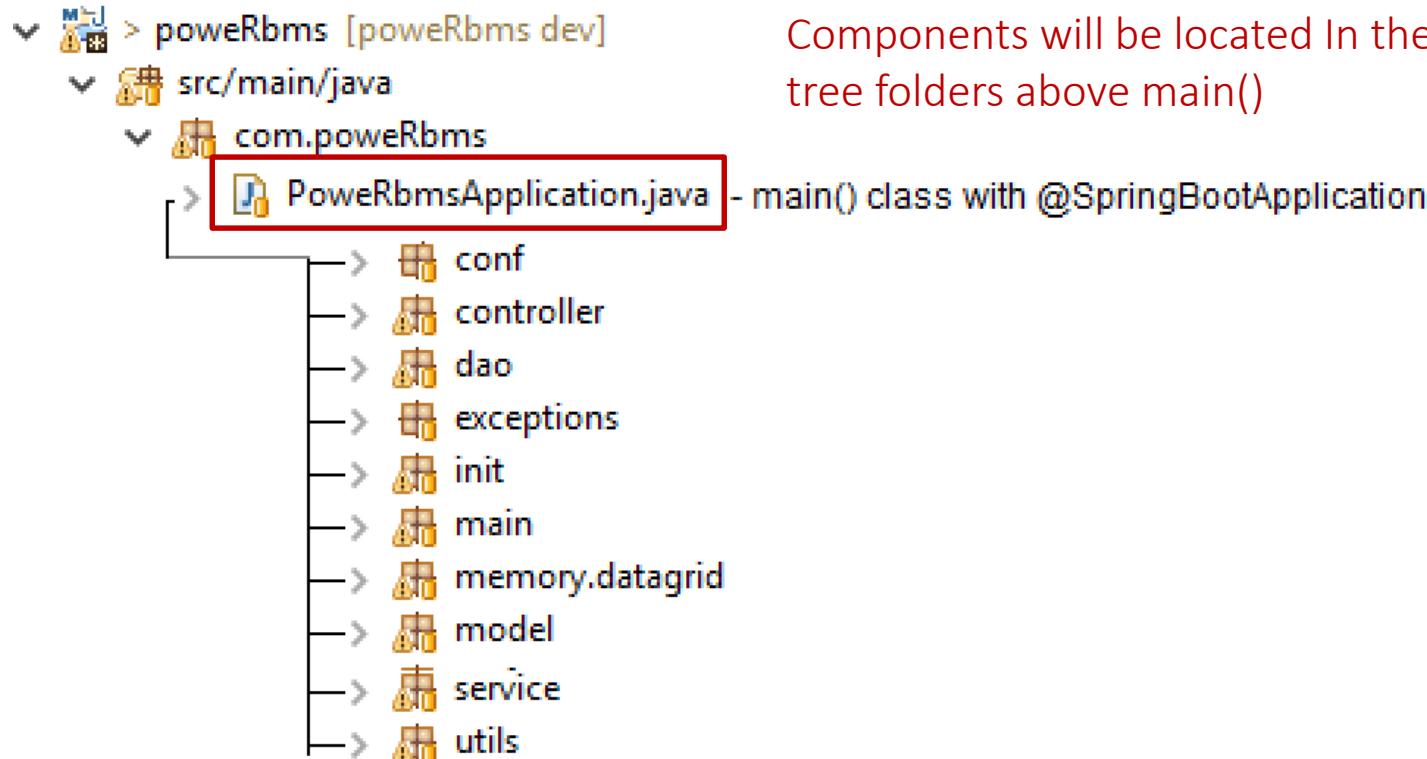
To reduce configuration :

Spring Boot Makes a lot of assumption / guesses..

The `@SpringBootApplication` annotated main class should be placed in the top/head of the application folders tree.

It will then scan the underneath directory tree for Spring components...

e.g...



## Spring Boot Assumption:

Components will be located In the relatively upper tree folders above main()



# Spring Boot

## Building & Running Spring Boot

### Building

Spring Boot jar can be built using maven clean/package or clean/install commands.

### Running

Spring Boot jar can be ran from command line simply by typing

Java –jar <name of the jar file>

e.g. Java –jar myapplication.jar

### WEB Application

If the pom.xml contains spring-boot-starter-web dependency, spring will start

Embedded Tomcat listens by default to port 8080

## Spring Boot Jar Structure

### Building

Spring Boot jar is an executable jar.

It has all the required dependencies wrapped into it..

Project classes

Spring/Spring Boot Libraries  
+ Project Dependencies

Spring Boot loader classes

META-INF\MANIFEST.MF

Spring-Boot-Version: 1.5.2.RELEASE  
Implementation-Vendor: Pivotal Software, Inc.

Main-Class: org.springframework.boot.loader.JarLauncher

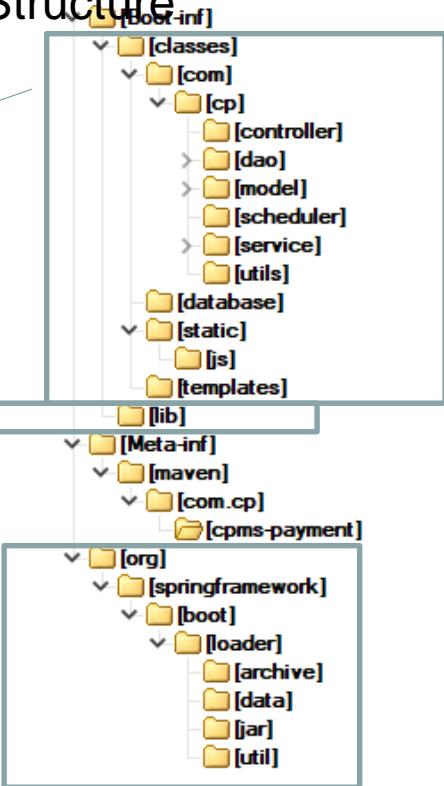
Start-Class: com.cp.CpMainApplication

Spring-Boot-Classes: BOOT-INF/classes/

Spring-Boot-Lib: BOOT-INF/lib/

Created-By: Apache Maven 3.2.1

### Spring Boot Jar Structure



The Spring Boot starters starts by default **Tomcat** embedded container for you when declaring “spring-boot-starter-web”

This can be changed to spring-boot-starter-jetty and spring-boot-starter-undertow etc..

## Use another Web server

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
  <exclusions>
    <exclusion> <!-- Exclude the Tomcat dependency -->
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-tomcat</artifactId>
    </exclusion>
  </exclusions>
</dependency>

<dependency> <!-- Use Jetty instead -->
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-jetty</artifactId>
</dependency>
```

## Adding Servlets, Filters, and Listeners using classpath scanning

@WebServlet, @WebFilter, and @WebListener annotated classes can be automatically registered with an embedded servlet container by **annotating a @Configuration class with @ServletComponentScan** and specifying the package(s) containing the components that you want to register.

By default, @ServletComponentScan will scan from the package of the annotated class.

## Spring Boot Minimal Configuration file

Spring Boot support one general configuration file called ‘application.properties’

The file is located in:

Maven Project – under src/main/resources

Runtime – in classpath of the jar

Out of the jar - under the ‘config’ sub-folder.



# Spring Boot

## Spring Boot Minimal Configuration file

Importing configuration file parameters into the application managed-beans is simply done by the @Values annotation

```
operational.country.code=972
```

```
operational.country.ID=IL
```

---

```
@Service
```

```
public class CustWalletServiceImpl extends ServiceBase implements CustWalletService{
```

```
    @Value("${operational.country.code}")
```

```
    private String countryCode;
```

```
    @Value("${operational.country.ID}")
```

```
    private String countryId;
```

```
server.port=3060          application.properties file
server.address=127.0.0.1

# ----- DB MYSQL -----
spring.datasource.url=jdbc:mysql://localhost:3306/cppym?useSSL=false
autoReconnect=true&useEncoding=true&characterEncoding=UTF-8
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.driver-class-name=com.mysql.jdbc.Driver

# ----- JPA -----
spring.jpa.open-in-view=false
spring.jpa.database=MYSQL
spring.jpa.hibernate.dialect=org.hibernate.dialect.MySQL5Dialect
spring.jpa.database-platform=org.hibernate.dialect.MySQL5Dialect
spring.jpa.show-sql=true
spring.jpa.hibernate.ddl-
auto=nonespring.jpa.properties.hibernate.hbm2ddl.auto=none
spring.data.jpa.repositories.enabled=true
spring.jpa.hibernate.naming_strategy=org.hibernate.cfg.EJB3NamingStrategy
```

## application.properties file

```
# ----- LOG -----
logging.pattern.file= "%d{yyyy-MM-dd HH:mm:ss} %-5level %logger{36} - %msg%n"
spring.application.name=cppym
logging.file=log/${spring.application.name}_PROP.log
logging.level.=DEBUG

logging.level.org.springframework : OFF

logging.level.org.hibernate: ERROR

logging.level.my.package1.name : OFF
logging.level.my.package2.name : OFF
logging.level.org.apache : OFF
```



## application.properties file

```
# ----- MONGODB -----
spring.data.mongodb.host=fw.dev.secureregion.com
spring.data.mongodb.port=27017
spring.data.mongodb.database=servicewall
spring.data.mongodb.username=servicewallmongo
spring.data.mongodb.password=servicewallmongopassword

# ----- SSL -----
server.ssl.enabled=true
server.ssl.client-auth=false
server.ssl.protocol=TLS
server.ssl.key-store=c:/cert/jjj.jks
server.ssl.key-store-password=Aa123456
server.ssl.trust-store=c:/cert/jjj.jks
server.ssl.trust-store-password=Aa123456
```

## Sample Application

```
@SpringBootApplication
public class Main1 {

    public static void main(String[] args) {
        SpringApplication.run(Main1.class, args);
    }
}
```

## @Controller

```
@RequestMapping(value="/employee")
public class EmployeeController {

    @Autowired
    EmployeeService employeeService;

    @RequestMapping(value="/findById", method=RequestMethod.GET)
    public @ResponseBody Employee findById(int id){
        return employeeService.findById(id);
    }

    @RequestMapping(value="/findByName", method=RequestMethod.GET)
    public @ResponseBody List<Employee> findByName( String name){
        return employeeService.findByName(name);
    }
}
```

```
public interface EmployeeService {  
    public Employee    findById (Integer id);  
    public List<Employee> findByName(String name);  
}
```

```
@Service
@Transactional
public class EmployeeServiceImpl implements EmployeeService{

    @Autowired
    private EmployeeRepository employeeRepository;

    @Override
    public Employee findById(Integer id) {
        return employeeRepository.findById(id);
    }

    @Override
    public List<Employee> findByName(String name) {
        return employeeRepository.findByName(name);
    }
}
```



# Spring Boot

```
@Service
@Transactional
public class EmployeeServiceImpl implements EmployeeService{

    @Autowired
    private EmployeeRepository employeeRepository;

    @Override
    public Employee findById(Integer id) {
        return employeeRepository.findById(id);
    }

    @Override
    public List<Employee> findByName(String name) {
        return employeeRepository.findByName(name);
    }
}
```



# Spring Boot

```
@Service
@Transactional
public class EmployeeServiceImpl implements EmployeeService{

    @Autowired
    private EmployeeRepository employeeRepository;

    @Override
    public Employee findById(Integer id) {
        return employeeRepository.findById(id);
    }

    @Override
    public List<Employee> findByName(String name) {
        return employeeRepository.findByName(name);
    }
}
```

## SPRING INITIALIZR

<https://start.spring.io/>



The screenshot shows the Spring Initializr landing page. At the top, there's a navigation bar with links like 'Secure' and 'Bookmarks'. Below it, a main heading says 'SPRING INITIALIZR bootstrap your application now'. To the right, a callout box says 'Fill in maven details'. In the center, there's a form with the text 'Generate a Maven Project with Java and Spring Boot 1.5.6'. Below this, under 'Project Metadata', there are fields for 'Group' (containing 'com.example') and 'Artifact' (containing 'demo'). On the right, under 'Dependencies', there's a search bar with 'Web, Security, JPA, Actuator, Devtools...' and a 'Selected Dependencies' section. A large green button at the bottom right says 'Generate Project alt + ⌘'. A callout box points to this button with the text 'Click here to expand'.

Generate a  with  and Spring Boot

### Project Metadata

Artifact coordinates

Group

com.example

Artifact

demo

### Dependencies

Add Spring Boot Starters and dependencies to your application

Search for dependencies

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

#### Selected Dependencies

Click here to expand

Generate Project alt + ⌘

Don't know what to look for? Want more options? [Switch to the full version.](#)



# Spring Boot

Generate Project alt + ↴

## Core

- Security  
Secure your application via spring-security
- Aspects  
Create your own Aspects using Spring AOP and AspectJ
- Atomikos (JTA)  
JTA distributed transactions via Atomikos
- Bitronix (JTA)  
JTA distributed transactions via Bitronix
- Narayana (JTA)  
JTA distributed transactions via Narayana
- Cache

## Web

- Web  
Full-stack web development with Tomcat and Spring MVC
- Reactive Web  
Reactive web development with Netty and Spring WebFlux  
requires Spring Boot >=2.0.0.M1
- Websocket  
Websocket development with SockJS and STOMP
- Web Services  
Contract-first SOAP service development with Spring Web Services
- Jersey (JAX-RS)  
RESTful Web Services framework with support of JAX-RS

Select required items  
And press “generate”



## Review Example: project: springbasic3-Boot



# spring boot





# **SPRING JPA**

# **SPRING DATA JPA**

# **HIBERNATE IMPLEMENTATION**

Examples : springbasic1 project, package: com.jbt.spring.db.jpa



# Spring JPA

Spring JPA/Spring Data JPA data access technologies:

- Spring Data/Spring Data JPA, is a part of the larger Spring Data family
- Makes it easy to access data using ORM
- The newest part - Spring Data JPA uses JPA based repositories makes it even easier.
- This module deals with enhanced support for JPA based data access layers.
- It makes it easier to build Spring-powered applications that use data access technologies.



# Spring JPA

There are a lot of ORM/JPA commercial packages available

[ActiveJDBC](#), Java implementation inspired by Ruby on Rails

[ActiveJPA](#), open-source Java ORM

[Apache Cayenne](#), open-source for Java

[Apache Gora](#), open-source software framework provides an in-memory data model and persistence for big data focused on [NoSQL](#) and SQL stores

[Athena Framework](#), open-source Java ORM

[Carbonado](#), open-source framework, backed by [Berkeley DB](#) or [JDBC](#)

[DataNucleus](#), open-source JPA implementation (formerly known as JPOX)

[Ebean](#), open-source ORM framework

[EclipseLink](#), Eclipse persistence platform



# Spring JPA

Enterprise Objects Framework, Mac OS X/Java, part of Apple WebObjects

- Hibernate, open-source ORM framework, widely used
- Apache OpenJPA - used by IBM 'W.A.S.'
- TopLink by (BEA) Oracle

Java Data Objects (JDO)

Java Object Oriented Querying (jOOQ)

Kodo, commercial implementation of both Java Data Objects and Java Persistence API

Kundera, open-source framework, JPA compliant, polyglot object-datastore mapping library for NoSQL datastores

MyBatis, free open-source, formerly named iBATIS

QuickDB ORM, open-source ORM framework



# Spring JPA

- The whole idea is to manipulate database operation using POJO Classes/Entities.
- The POJO's are mapped by annotations to assign it to a database table and database table columns.

No SQL involved.

No persistence.xml file is required - application.properties can be used instead

```
@Entity
@Table(name="Employees")
public class Employee {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    @Column(name = "id")
    private int id;

    ...
}
```



# Spring JPA

To Implement JPA in Spring, 3 Beans should be created in the configuration class

- [DataSource](#)
- [EntityManagerFactory](#)
- [JpaTransactionManager](#)



# Spring JPA

Configuration class example:

```
@Configuration
@ComponentScan(basePackages = "com.jbt.spring.db.jpa")
@EnableTransactionManagement

public class SpringConfig {
    @Bean
    public DataSource dataSource() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName("com.mysql.jdbc.Driver");
        dataSource.setUrl("jdbc:mysql://localhost:3306/TestDB");//change url
        dataSource.setUsername("root"); //change userid
        dataSource.setPassword("root");//change pwd
        return dataSource;
    }
}
```



# Spring JPA

Configuration class *example* - CONT:

```
@Bean
public LocalContainerEntityManagerFactoryBean entityManagerFactory() {
    LocalContainerEntityManagerFactoryBean entityManagerFactory =
        new LocalContainerEntityManagerFactoryBean();
    entityManagerFactory.setDataSource(dataSource());
    entityManagerFactory.setPackagesToScan( "com.jbt.spring.db.jpa" );
    entityManagerFactory.setJpaVendorAdapter(new HibernateJpaVendorAdapter() );
    Properties properties = new Properties();
    properties.setProperty("hibernate.dialect", "org.hibernate.dialect.MySQL5Dialect");
    entityManagerFactory.setJpaProperties( properties );

    return entityManagerFactory;
}
```



# Spring JPA

Configuration class *example* - CONT:

```
@Bean
public JpaTransactionManager transactionManager() {
    JpaTransactionManager transactionManager = new JpaTransactionManager();
    transactionManager.setEntityManagerFactory(entityManagerFactory().getObject()); // javax.persistence.EntityManagerFactory
    return transactionManager;
}
```



# Spring JPA

## The POJO Entity

```
@Entity
@Table(name="Employees")
public class Employee {
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    @Column(name = "id")
    private int id;

    @Column(name = "name")
    private String name;

    @Column(name = "age")
    private int age;
    ....
    ....get/set
}
```



# Spring JPA

## The DAO interface..

```
public interface EmployeeDAO {  
    public Employee    getEmployeeById(int id);  
    public List<Employee> getAllEmployees();  
    public void        saveEmployee(Employee e);  
    public void        deleteEmployee(Integer empid);  
}
```



# Spring JPA

## The DAO Impl - uses Injected EntityManager

```
@Repository
public class EmployeeDAOImpl implements EmployeeDAO {

    @PersistenceContext
    private EntityManager _em;

    @Override
    @Transactional
    public void saveEmployee(Employee employee) {
        _em.persist(employee);
    }

    @Override
    public Employee getEmployeeById(int empid) {
        Employee employee = (Employee) _em.find(Employee.class, empid);
        return employee;
    }
}
```



# Spring JPA

## The DAO Impl – CONT

```
@Override
public List<Employee> getAllEmployees() {
    String SQL = "SELECT e FROM Employee e";
    Query query = _em.createQuery(SQL);
    List<Employee> employees = (List<Employee>)query.getResultList();
    return employees;
}

@Override
@Transactional
public void deleteEmployee(Integer empid) {
    Employee employee = _em.find(Employee.class, empid);
    if ( employee != null){
        _em.remove(employee);
    }else{
        System.out.println("empid:[" + empid + "] was not found, nothing to delete");
    }
}
```



# Spring JPA

## Test main

```
public static void main(String[] args) {  
    ApplicationContext ctx = new AnnotationConfigApplicationContext(SpringConfig.class);  
  
    EmployeeDAO empDAO = ctx.getBean(EmployeeDAO.class);  
    Employee emp = empDAO.getEmployeeById(111);  
    System.out.println("Employee 111 is: " + emp );  
  
    System.out.println("\nEmployees List:");  
    List<Employee> list = empDAO.getAllEmployees();  
    for (Employee employee : list) {  
        System.out.println(employee);  
    }  
    empDAO.saveEmployee(new Employee("John", 20));  
    empDAO.deleteEmployee(335);  
}  
}
```



# Spring JPA

## Maven dependencies

```
<properties>
    <org.springframework.version>4.0.6.RELEASE</org.springframework.version>
</properties>
```

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-tx</artifactId>
    <version>${org.springframework.version}</version>
</dependency>
```

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-orm</artifactId>
    <version>${org.springframework.version}</version>
</dependency>
```



# Spring JPA

## Maven dependencies

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>${hibernate.version}</version>
</dependency>
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-entitymanager</artifactId>
    <version>${hibernate.version}</version>
</dependency>
```

```
<properties>
    <hibernate.version>4.3.6.Final</hibernate.version>
</properties>
```



# STILL.. TOO COMPLICATED



# SPRING DATA JPA

Examples: com.jbt.spring.db.datajpa



# Spring Data JPA

- Still - Too much code has to be written to execute simple queries
- Spring Data JPA aims to significantly improve the implementation of data access layers by reducing the effort to the amount that's actually needed.
- As a developer you write your repository interfaces, including custom finder methods, and Spring will provide the implementation automatically.



# Spring Data JPA

## The Change..

- The DAO Layer classes are replaced by just interfaces that **extend**  
`org.springframework.data.jpa.repository.JpaRepository`  
With the Generics<**Entity**, **Entity-PK**> parameters

```
public interface EmployeeRepository extends JpaRepository<Employee, Integer>{  
  
    public Employee findById (int empid);  
    public List<Employee> findByName (String name);  
    public List<Employee> findByNameLike (String name);  
    public List<Employee> findByAge (int age);  
    public List<Employee> findByAgeLessThan (int age);  
    public List<Employee> findByAgeGreaterThan (int age);  
}
```



# Spring Data JPA

The ServiceImpl class injects and uses the Repository..

```
@Service
public class EmployeeServiceImpl implements EmployeeService {

    @Resource
    private EmployeeRepository employeeRepository;

    @Override
    @Transactional
    public Employee saveEmployee(Employee employee) {
        Employee e = employeeRepository.save(employee);
        return e;
    }
}
```



# Spring Data JPA

## Vocabulary

### Keyword Sample

And	findByLastnameAndFirstname
Or	findByLastnameOrFirstname
Between	findByStartDateBetween
LessThan	findByAgeLessThan
GreaterThan	findByAgeGreaterThan
After	findByStartDateAfter
Before	findByStartDateBefore
IsNull	findByAgeIsNotNull
IsNotNull,NotNull	findByAge(Is)NotNull
Like	findByFirstnameLike
NotLike	findByFirstnameNotLike
StartingWith	findByFirstnameStartingWith
EndingWith	findByFirstnameEndingWith



# Spring Data JPA

## Keyword Sample

Containing	findByFirstnameContaining
OrderBy	findByAgeOrderByLastnameDesc
Not	findByLastnameNot
In	findByAgeIn(Collection<Age> ages)
NotIn	findByAgeNotIn(Collection<Age> age)
TRUE	findByActiveTrue()
FALSE	findByActiveFalse()



# Spring Data JPA

## Required Dependencies:

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



# Spring Data JPA

## @Query - HQL Type Queries

In case an explicit query should be precisely keyed in the @Query annotation takes place:

```
public interface UserRepository extends JpaRepository<User, Long> {  
    @Query("select u from User u where u.emailAddress = ?1")  
    User findByEmailAddress(String emailAddress);  
}
```

```
public interface UserRepository extends JpaRepository<User, Long> {  
    @Query("select u from User u where u.firstname like %?1")  
    List<User> findByFirstnameEndsWith(String firstname);  
}
```



# Spring Data JPA

## @Query – HQL Type Queries

```
public interface FileUploadRepository extends JpaRepository<FileUpload, Long>
{
```

```
    @Query("Select a from VInboxMessage a where a.accountNumber = :accountNumber "
        + " And a.accountBranchNumber = :accountBranchNumber And a.idType Is Null "
        + " And a.idNumber Is Null And a.archiveInboxDate Between :fromDate And :toDate ")
    public List<VInboxMessage> findByFewParams (@Param("accountNumber")
```

```
        String accountNumber,
```

```
        @Param("accountBranchNumber") String accountBranchNumber,
```

```
        @Param("fromDate") Date fromDate,
```

```
        @Param("toDate") Date toDate );
```

```
}
```



# Spring Data JPA

## @Query – Native Queries

A Native Query can be also presented by specifying the `nativeQuery = true` parameter:

```
public interface FileUploadRepository extends JpaRepository<FileUpload, Long>
{
    @Query(value = "select * from fileuploadtable where "
        + " fileGuid= :fileGuid AND SanitizeStatus is not null ", nativeQuery = true)
    public FileUpload CheckGuid(@Param("fileGuid") String fileGuid);

}
```



Review Example on: com.jbt.spring.db.datajpa



# NOSQL WITH SPRING DATA



# NoSQL with Spring Data

Spring Data provides additional projects that help you access a variety of NoSQL technologies including

- MongoDB
- Neo4J
- Elasticsearch
- Solr
- Redis
- Gemfire
- Cassandra
- Couchbase
- LDAP



# NoSQL with Spring Data

## Mongo DB Required Dependencies

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



# NoSQL with Spring Data

## Mongo DB application.properties

```
spring.data.mongodb.host=com.jbt.com
spring.data.mongodb.port=27017
spring.data.mongodb.database=empdb
spring.data.mongodb.username=empdbuser
spring.data.mongodb.password=empdbpassword
```



# NoSQL with Spring Data

## Mongo DB Entity Class

```
@Document(collection="people") ← MongoDB collection
public class Person{                                No @Entity required
    @Id
    protected String id;                           No @Column required
    protected String name;                         No @xToN relations
    protected int age;
    get / set...
    ....
}
```



# NoSQL with Spring Data

## Mongo DB Repository Class

public interface PersonRepository extends **MongoRepository<Person, String>**

```
List<Person> findByName(String name);  
List<Person> findByAge(int age);  
}
```

The rest classes: @Controller's, @Service's – are the same as with JPA repositories..



# SPRING MVC & REST

Examples: springbasic-Mvc project, TestController

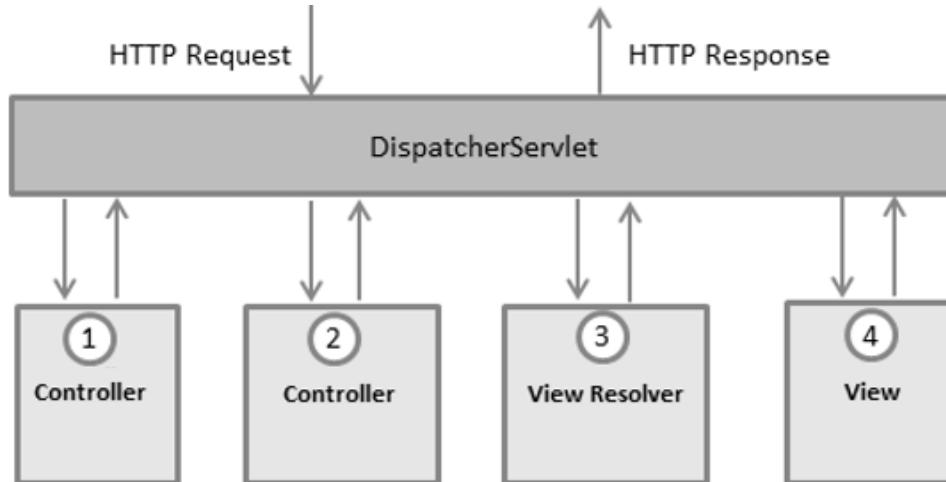
The Spring Web MVC framework provides Model-View-Controller (MVC) architecture.

The MVC pattern results in separating the different aspects of the application (input logic, business logic, and UI logic), while providing a loose coupling between these elements.

The Spring Web framework is designed around the spring **DispatcherServlet**

## Dispatcher Servlet

- Dispatches requests to handlers denoted with @Controller Annotation
- Supports injections data structures are handles in JSON format by default





## Controllers

- Controllers are POJO classes annotated by class level '@Controller' Annotation.
- Controllers are the web Gateways into the application.

```
@RestController
```

```
@RequestMapping(value="/inbox")
```

```
public class InboxController {
```

```
    @RequestMapping(value="/test", method=RequestMethod.GET)
```

```
    public @ResponseBody String test() {
```

```
        return "InboxController.test() - I'm OK ! ! helloStr:[" + helloStr +"]";
```

```
}
```

```
}
```

Target URL: <http://localhost:8080/inbox/test>



Application has to launch DispatcherServlet by configuration.

Launching the Dispatcher Servlet can be done:

- Via WEB-INF/web/xml
- Via WebApplicationInitializer



# Spring MVC

## Registering DispatcherServlet via web.xml DD

```
<web-app>
  <servlet>
    <servlet-name>dispatcher1</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>dispatcher1</servlet-name>
    <url-pattern>/*</url-pattern>
  </servlet-mapping>
</web-app>
```

Logical name

## From DispatcherServlet web.xml – to Spring <Beans>

- When launched from web.xml, DispatcherServlet tries to locate a file with a special name, identical some to its logical name ended with “-servlet.xml”..  
e.g: in the above example the file should be called **dispatcher1-servlet.xml**
- This file is the gateway to Spring framework world, loading bean'S in the old fashion:

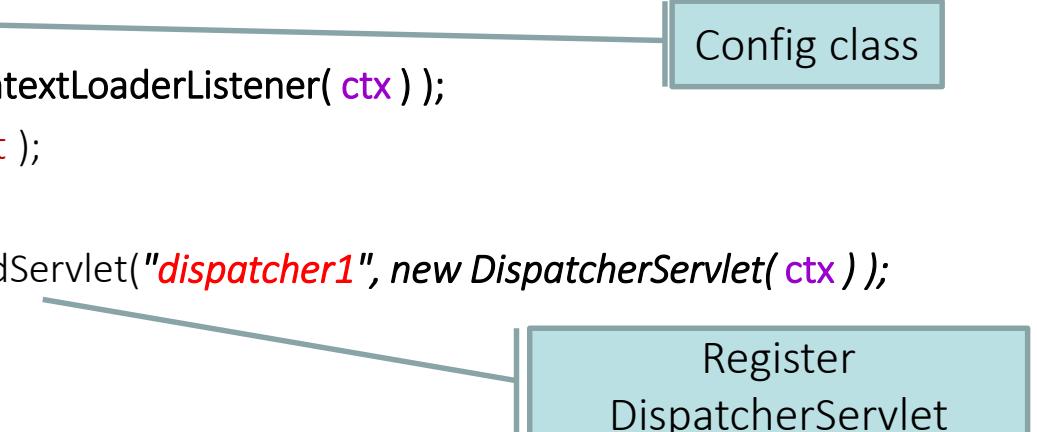
```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans".....>
    <import resource="spring-beans.xml" />
    <mvc:annotation-driven />
    <context:component-scan base-package="com.jbt.examples.people" />
    <bean id="myBean1" class="com.jbt.beans.MyBean1"/>
</beans>
```



# Spring MVC

## Registering DispatcherServlet Via WebApplicationInitializer:

```
public class Initializer implements WebApplicationInitializer {  
  
    public void onStartup(ServletContext servletContext ) throws ServletException {  
        AnnotationConfigWebApplicationContext ctx = new AnnotationConfigWebApplicationContext();  
        ctx.register(WebAppConfig.class);  
        servletContext.addListener( new ContextLoaderListener( ctx ) );  
        ctx.setServletContext( servletContext );  
  
        Dynamic servlet = servletContext.addServlet("dispatcher1", new DispatcherServlet( ctx ) );  
        servlet.addMapping("/");  
        servlet.setLoadOnStartup(1);  
    }  
}
```

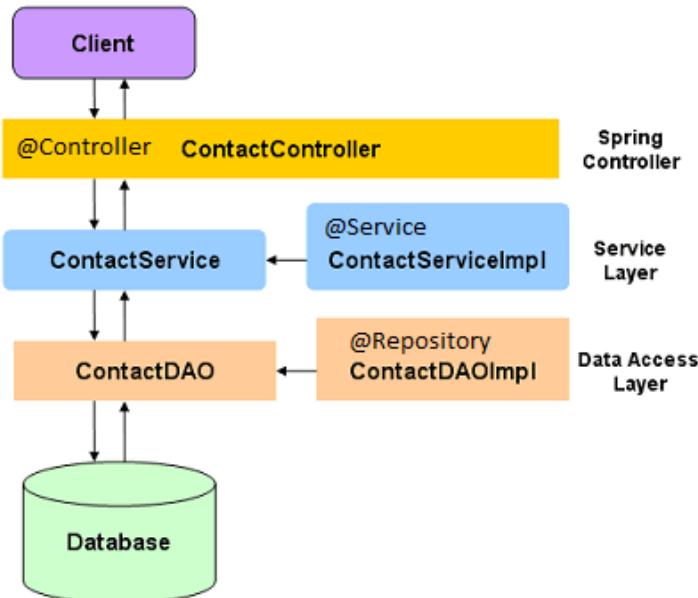




# Spring MVC

## Controllers

- Controllers part/goal is to be a gateway to the application logic
- Controllers should not do any business logic, but only deal with the http layer
- Business logic should be done by the Service / Serviceml Layer.





# Spring MVC

The `@RestController` Annotation (extends `@Controller`)

`@Controller`'S are Spring Managed beanS As they are also `@Component's..`

```
@Target({ElementType.TYPE})  
@Retention(RetentionPolicy.RUNTIME)  
@Documented  
@Component
```

```
public @interface Controller {  
    String value() default "";  
}
```



# Spring MVC

## Controllers URL mapping - `@RequestMapping`

`@Controller`'S class and methods are entitled to be mapped by the  
“`@RequestMapping`” annotation..

`@RequestMapping` maps HTTP call to a class/method

Class Level mapping

`@RestController`

`@RequestMapping(value = "/inbox")`

public class InboxController {

...

...

`@RequestMapping(value = "/test", method = RequestMethod.GET)`

public @ResponseBody String test() {

...

Method Level mapping



# Spring MVC

## Controllers Methods Parameters and Response format

### Methods Parameters

Controllers Methods can accept multiple **user defined parameters**

Controllers Methods can accept multiple **Spring parameters**

```
@RequestMapping(value = "/setEmployee", method = RequestMethod.POST)
public @ResponseBody<Employee> setEmployee(@RequestBody Employee emp,
                                             HttpServletRequest request,
                                             HttpServletResponse response) {
```

### Response format

Controller Methods returning Objects will be converted into JSON format unless otherwise is explicitly specified.



# Spring MVC

## Controllers URL mapping

@RequestMapping attributes

method:

`@RequestMapping(value = "/test", method=RequestMethod.GET)`

Will match the “/test” url + HTTP GET requests only

headers:

`@RequestMapping(value="/something", headers = "content-type=text/*")`

Will match requests to “/domething” url with a Content-Type header of “text/html”, “text/plain”, etc.



# Spring MVC

## @RequestMapping attributes - CONT

### *consumes*

```
@RequestMapping(value = "/test", method = RequestMethod.GET,  
    consumes={MediaType.APPLICATION_JSON_VALUE})
```

The consumes media types of the mapped request, narrowing the primary mapping.

The format is a single media type or a sequence of media types, with a request only mapped if the “Content-Type” header matches one of these media types.

Examples: consumes = "text/plain" consumes = {"text/plain", "application/\*"}

Expressions can be negated by using the "!" operator, as in "!"text/plain", which matches all requests with a Content-Type other than "text/plain".



# Spring MVC

@RequestMapping attributes - CONT

produces

```
@RequestMapping(value = "/test", method = RequestMethod.GET,  
    produces=MediaType.APPLICATION_JSON_VALUE )
```

The produces media types of the mapped request, narrowing the primary mapping.

The format is a single media type or a sequence of media types, with a request only mapped if the “Accept header” matches one of these media types.

*Examples:* produces = "text/plain" produces = {"text/plain", "application/\*"}

Expressions can be negated by using the "!" operator, as in "**!text/plain**", which matches all requests with a Accept other than "text/plain".



## Controller parameters annotations:

### @RequestParam

The @RequestParam annotation is used to bind parameter values of query string

to the controller method parameters.

```
public String handleEmployeeRequestByDept (@RequestParam("dept") String dept) {..}
```

Called by: <http://localhost:8080//springbasicMvc/...?dept=SALES>

Basically, Spring 4.x Controller methods receive the parameters automatically when they match with names with no need of adding this annotation.

### @PathVariable

```
@RequestMapping(value="/action3/id/{id}/name/{name}")  
public @ResponseBody String action3(@PathVariable("id") int id,  
                                    @PathVariable("name") String name){
```

Called by: <http://localhost:8080//springbasicMvc/actions/action3/id/123/name/Dani>



Controller parameters annotations:

## @RequestHeader

Injects a web request header as a method parameter

e.g. public String hello(@RequestHeader(value="User-Agent") String userAgent)

## @ResponseEntity – (later)

A wrapper for response which allows to return different types of contents, including error statuses and messages

e.g. public @ResponseBody<Employee> getEmployee()



## Controller parameters annotations:

```
@PathVariable
```

```
@RequestMapping(value="/action3/id/{id}/name/{name}")
```

```
public String action3(@PathVariable("id") int id,  
                      @PathVariable("name") String name){
```

Called by: <http://localhost:8080//springbasicMvc/actions/action3/id/123/name/Dani>

Review Example project : `springbasic-Mvc`, `TestController` class

## @ResponseBody & HttpStatus

Sets the HTTP response content & status code



# REST based Web Services

## Manipulating HTTP Response via ResponseEntity

status header – HttpStatus

headers - HttpHeaders

body - <T>

```
@RestController
public class Hello {
    @RequestMapping(...)
    public ResponseEntity<Entity> setEntry (...) {
        if (ok) {
            HttpHeaders responseHeaders = new HttpHeaders();
            responseHeaders.set("MyResponseHeader", "MyValue");
            Entity data = new Entity();
            data.set..(...);
            return new ResponseEntity<Entity>( data , responseHeaders, HttpStatus.OK);
        }else{
            return new ResponseEntity(HttpStatus.BAD_REQUEST);
        }
    }
}
```

Resulting mixed ResponseEntity contents:

```
@RestController
public class Hello {
    @RequestMapping(... )
    public ResponseEntity<?> setEntry (...) {
        if (ok) {
            HttpHeaders responseHeaders = new HttpHeaders();
            responseHeaders.set("MyResponseHeader", "MyValue");
            Entity data = new Entity();
            data.set...( ... );
            return new ResponseEntity<Entity>( data , responseHeaders, HttpStatus.OK);
        }else{
            return new ResponseEntity<String>("Error setting entity",HttpStatus.BAD_REQUEST);
        }
    }
}
```



## Controllers as application Gateways

Controller methods serve as application gateway.

Controller methods will delegate request handling to injected @Service methods

**@Controller**

```
public class EmployeeController {
```

**@Autowired**

```
private EmployeeService employeeService;
```

```
@RequestMapping(value = "/getEmployee", method = RequestMethod.GET)
```

```
public @ResponseBody Employee getEmployee(int empld) {
```

```
    return employeeService.getEmployee(empld);
```

```
}
```



# Spring MVC – Additional Scopes

Additional Scopes with SpringMVC:

- request
- session
- globalsection – relevant for Portlet based pages in which several web modules, each with its own session, are combined in a single view and might need to share ‘global’ session data
- All can be set via @Scope Spring annotation

Review Example project : **springbasic-Mvc**, ExceptionController class

# Spring MVC – Combining Swagger

## Swagger

- Allows to rapidly generate web UI for SpringMVC rest endpoints
- Provides simple and useful testing utilities



# Spring MVC – Combining Swagger

## Swagger2

- First, we add swagger and swagger-ui Maven dependencies

```
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger2</artifactId>
    <version>2.9.2</version>
</dependency>

<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger-ui</artifactId>
    <version>2.9.2</version>
</dependency>
```

# Spring MVC – Combining Swagger

## Swagger2

- Then, we add a swagger configuration:
  - Enabling Swagger
  - Setting Docket bean to specify mapped endpoints

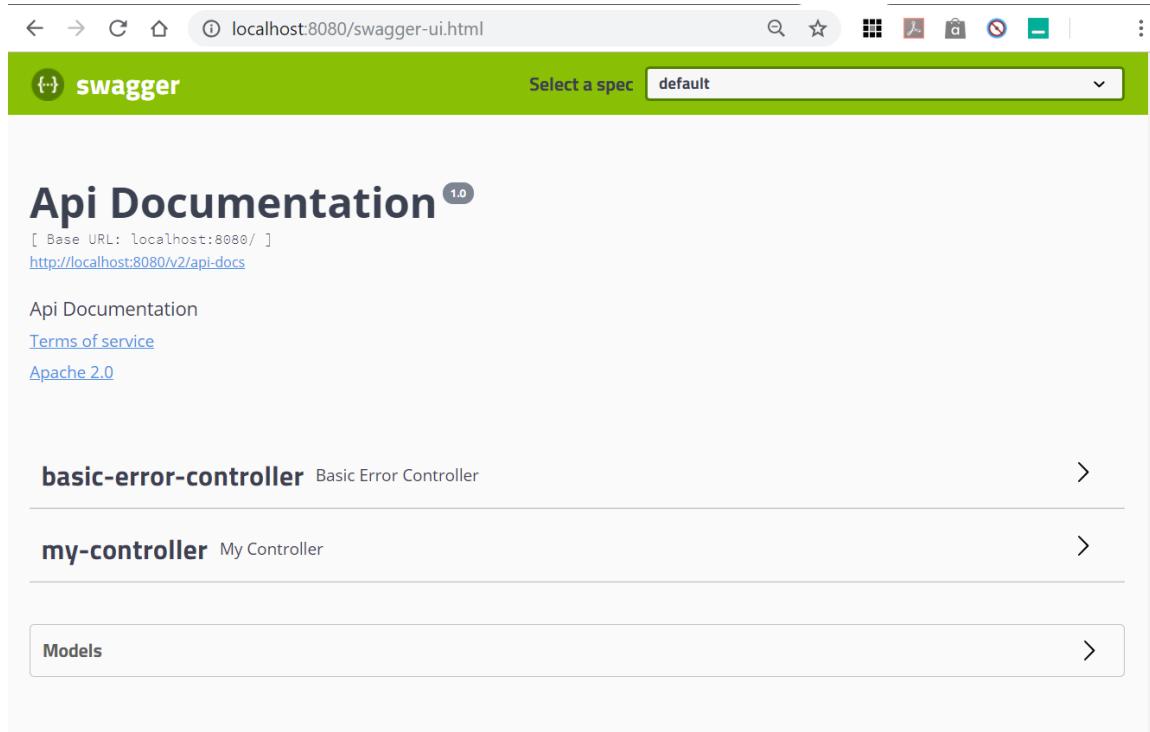
```
@Configuration
@EnableSwagger2
public class SwaggerConfiguration {

    @Bean
    public Docket api() {
        return new Docket(DocumentationType.SWAGGER_2)
            .select()
            .apis(RequestHandlerSelectors.any())
            .paths(PathSelectors.any())
            .build();
    }
}
```

# Spring MVC – Combining Swagger

## Swagger2

- Swagger UI can be found on this URL:  
<http://localhost:8080/swagger-ui.html>



The screenshot shows the Swagger UI interface for an API. At the top, there's a green header bar with the word "swagger" and a dropdown menu labeled "Select a spec" set to "default". Below the header, the title "Api Documentation" is displayed with a "1.0" badge. A note indicates the base URL is "localhost:8080" and provides a link to "http://localhost:8080/v2/api-docs". The main content area lists three sections: "basic-error-controller" (Basic Error Controller), "my-controller" (My Controller), and "Models". Each section has a right-pointing arrow indicating further details. To the right of the screenshot, two boxes are connected by arrows pointing towards the "basic-error-controller" section: a top box labeled "SpringBoot error controller" and a bottom box labeled "Your controller".

SpringBoot error controller

Your controller

# Spring MVC – Combining Swagger

## Swagger2

- Swagger UI

Exploring endpoints

The screenshot shows the Swagger UI interface at `localhost:8080/swagger-ui.html#/my-controller`. The interface lists ten API endpoints under the heading "my-controller My Controller". Each endpoint is represented by a colored button indicating the HTTP method and a link to the detailed documentation.

Method	Path	Description
GET	/greet	greet
POST	/handle	handlePerson
GET	/people	getPeople
HEAD	/people	getPeople
POST	/people	getPeople
PUT	/people	getPeople
DELETE	/people	getPeople
OPTIONS	/people	getPeople
PATCH	/people	getPeople
GET	/person/{name}/{age}	getPerson

# Spring MVC – Combining Swagger

## Swagger2

- Swagger UI

Testing endpoints

The screenshot shows the Swagger UI interface for a Spring MVC application. The URL in the browser is `localhost:8080/swagger-ui.html#/my-controller/greetUsingGET`. The main content area displays the `my-controller` endpoint details:

- Method:** GET /greet
- Parameters:**
  - Name:** name (string, query)
  - Description:** Default value : No One
- Responses:**
  - Code:** 200 **Description:** OK
  - Code:** 401 **Description:** Unauthorized
  - Code:** 403 **Description:** Forbidden
  - Code:** 404 **Description:** Not Found

At the bottom, other endpoints are listed:

- POST** /handle handlePerson
- GET** /people getPeople



# SPRING-DATA-REST & HAL-BROWSER

Examples: project: springbasic3-Boot

# SPRING-Data-Rest & Hal-Browser

To make development even easier, SPRING-Data-Rest & Hal-Browser were developed.

HAL – stands for: Hypertext Application Language

Decorating the JPA repositories with `@RepositoryRestResource` annotation will grant them A `@Controller` web-interface capabilities with full REST features:

```
@RepositoryRestResource(path = "employee")
public interface EmployeeRepository extends JpaRepository<Employee, Integer>{
```

This can provide the basic CRUD requirements for the basic entities.

# SPRING-Data-Rest & Hal-Browser

## Maven Dependencies:

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

OR

```
<dependency>
  <groupId>org.springframework.data</groupId>
  <artifactId>spring-data-rest-webmvc</artifactId>
</dependency>
```

# SPRING-Data-Rest & Hal-Browser

Spring Data REST officially supports:

- Spring Data JPA
- Spring Data MongoDB
- Spring Data Neo4j (Graph Database)
- Spring Data GemFire
- Spring Data Cassandra

# SPRING-Data-Rest & Hal-Browser

REPOSITORY With HAL annotations:

```
@RepositoryRestResource(path = "emp.jpa.repos")
```

```
public interface EmployeeRepository extends JpaRepository<Employee, Integer> {
```

```
    public Employee findById(Integer id);
```

```
@RestResource(path = "/findByName", rel = "findByName")
```

```
    public List<Employee> findByName(@Param("name") String name);
```

```
@RestResource(path = "/findByAgeGreaterThan", rel = "findByAgeGreaterThan")
```

```
    public List<Employee> findByAgeGreaterThan(@Param("age") int age);
```

```
}
```

CALL:

```
curl -i http://localhost:8080/emp.jpa.repos/search/findByAgeGreaterThan?age=22
```

```
curl -i http://localhost:8080/emp.jpa.repos/search/findByName?name=John
```



# SPRING-Data-Rest & Hal-Browser

http://localhost:8080/

The screenshot shows the HAL Browser interface for Spring Data REST. The top navigation bar includes links for 'localhost:8080/agent', 'yosi', 'comp', 'J9', 'jv8', 'Mongo-Mapping', 'jointJS', 'react', 'spring5', 'Etcd', and 'swagger-samples/java'. Below the navigation is a header with 'The HAL Browser (for Spring Data REST)', 'Go To Entry Point', and 'About The HAL Browser (for Spring Data REST)'.

**Explorer**

- Custom Request Headers:** A text input field with a 'Go!' button.
- Properties:** A JSON object field containing '{}'.
- Links:** A table showing relationships between entities:

rel	title	name / index	docs	GET	NON-GET
Employees					
profile					

**Inspector**

**Response Headers:**

```
200 success
date: Tue, 29 Aug 2017 22:51:30 GMT
transfer-encoding: chunked
content-type: application/hal+json; charset=UTF-8
```

**Body:**

```
{
  "_links": {
    "Employees": {
      "href": "http://localhost:8080/emp.jpa.repos?page,size,sort",
      "templated": true
    },
    "profile": {
      "href": "http://localhost:8080/profile"
    }
  }
}
```



# SPRING-Data-Rest & Hal-Browser

Pressing the 'GET' button:

**Expand URI Template**

URI Template:

```
http://localhost:8080/emp.jpa.repos{?page,size,sort}
```

Input (JSON):

```
{  
  "page": "",  
  "size": "",  
  "sort": ""  
}
```

Expanded URI:

```
http://localhost:8080/emp.jpa.repos
```

**Follow URI**

The diagram illustrates the process of expanding a URI template. On the left, a modal window titled 'Expand URI Template' shows a 'URI Template' input field containing 'http://localhost:8080/emp.jpa.repos{?page,size,sort}'. Below it is an 'Input (JSON)' field containing a JSON object with three empty string values for 'page', 'size', and 'sort'. A callout box labeled 'Parameters' points from the JSON input area to the placeholder '{?page,size,sort}' in the expanded URI field. The expanded URI field shows the resulting URL 'http://localhost:8080/emp.jpa.repos'. At the bottom right of the modal is a blue 'Follow URI' button.

# SPRING-Data-Rest & Hal-Browser

Pressing the 'NON-GET' button:

**Create/Update**

**Employee**

Name

Id

Age

Action:  
POST  
<http://localhost:8080/emp.jpa.repos>

**Make Request**

POST/PUT  
Create  
Or  
Update

# SPRING-Data-Rest & Hal-Browser

`http://localhost:8080/emp.jpa.repos/search/findByAgeGreaterThan?age=22`

```
{
    - _embedded: {
        - employees: [
            - {
                name: "Steven",
                age: 32,
                - _links: {
                    - self: {
                        href: "http://localhost:8080/emp.jpa.repos/222"
                    },
                    - employee: {
                        href: "http://localhost:8080/emp.jpa.repos/222"
                    }
                }
            },
            - {
                name: "Cheng",
                age: 33,
                - _links: {
                    - self: {
                        href: "http://localhost:8080/emp.jpa.repos/333"
                    },
                    - employee: {
                        href: "http://localhost:8080/emp.jpa.repos/333"
                    }
                }
            }
        ]
    - _links: {
        - self: {
            href: "http://localhost:8080/emp.jpa.repos/search/findByAgeGreaterThan?age=22"
        }
    }
}
```

# SPRING-Data-Rest & Hal-Browser

@Id fields are missing from the above search results.

To expose the entities ID fields - create a new class as follows

```
import org.springframework.context.annotation.Configuration;
import org.springframework.data.rest.core.config.RepositoryRestConfiguration;
import
org.springframework.data.rest.webmvc.config.RepositoryRestConfigurerAdapter;
```

```
import com.jbt.model.Employee;
```

```
@Configuration
```

```
public class ExposeEntityIdRestConfig extends RepositoryRestConfigurerAdapter {
```

```
    @Override
```

```
        public void configureRepositoryRestConfiguration(RepositoryRestConfiguration
config) {
```

```
            config.exposIdsFor(Employee.class);
```

```
}
```

```
}
```

# SPRING-Data-Rest & Hal-Browser

## SPRING-Data-Rest Data Manipulations

### INSERT

```
curl -i -X POST -H "Content-Type:application/json; charset=utf-8"  
http://localhost:8080/emp.jpa.repos -d "{ \"name\":\"nm444\", \"age\":\"44\"}"
```

### SEARCH

```
curl -i http://localhost:8080/emp.jpa.repos/334  
curl -i http://localhost:8080/emp.jpa.repos/search/findByAgeGreaterThan?age=22  
curl -i http://localhost:8080/emp.jpa.repos/search/findByName?name=John
```

### UPDATE

```
curl -X PUT -H "Content-Type:application/json; charset=utf-8" -d "{ \"name\":\"Yosi\",  
\"age\":\"66\"}" http://localhost:8080/emp.jpa.repos/444
```

### DELETE

```
curl -i -X DELETE http://localhost:8080/emp.jpa.repos/339  
curl -i http://localhost:8080/emp.jpa.repos/339 ← Not found
```



# SPRING REST TEMPLATE

Examples: project: springbasic1, package:  
com.jbt.spring.rest.clients



# Spring RestTemplate

## RestTemplate

Spring's central class for synchronous client-side HTTP access.

It simplifies communication with HTTP servers, and enforces RESTful principles.



# Spring RestTemplate

## RestTemplate

Spring's central class for synchronous client-side HTTP access.

It simplifies communication with HTTP servers, and enforces RESTful principles.

`org.springframework.web.client.RestTemplate`



# Spring RestTemplate

## RestTemplate

Simple methods as:

- getForObject
- postForObject
- put
- delete



# Spring RestTemplate

## RestTemplate Examples

Get XML representation of employees collection in String format

### SERVER

```
@RequestMapping(value = "/employees", produces =
        MediaType.APPLICATION_XML_VALUE, method = RequestMethod.POST)
public String getAllEmployeesXML(Model model)
```

### CLIENT

```
final String uri = "http://localhost:8080/employees";
RestTemplate restTemplate = new RestTemplate();
String result = restTemplate.getForObject(uri, String.class);
```



# Spring RestTemplate

## RestTemplate Examples

### Getting Objects

```
String uri = "http://localhost:8080/emp.jpa.repos/343";
Employee emp = restTemplate.getForObject(uri, Employee.class);
System.out.println("EMPLOYEE:" + emp);
```

*OUTPUT:* EMPLOYEE:Employee [id=343, name=Yosi, age=66]



# Spring RestTemplate

## RestTemplate Examples

### Passing parameters in URL

#### SERVER

```
@RequestMapping(value = "/employees/{id}")
public ResponseEntity<EmployeeVO> getEmployeeById (@PathVariable("id") int id)
```

#### CLIENT

```
final String uri = "http://localhost:8080/springrestexample/employees/{id}";
Map<String, String> params = new HashMap<String, String>();
params.put("id", "1");
```

```
RestTemplate restTemplate = new RestTemplate();
EmployeeVO result = restTemplate.getForObject(uri, EmployeeVO.class, params);
```



# Spring RestTemplate

## RestTemplate Examples

### Passing parameters in URL

#### SERVER

```
@RequestMapping(value = "/employees/{id}")
public ResponseEntity<EmployeeVO> getEmployeeById (@PathVariable("id") int id)
```

#### CLIENT

```
final String uri = "http://localhost:8080/springrestexample/employees/{id}";
Map<String, String> params = new HashMap<String, String>();
params.put("id", "1");
```

```
RestTemplate restTemplate = new RestTemplate();
EmployeeVO result = restTemplate.getForObject(uri, EmployeeVO.class, params);
```



# Spring RestTemplate

## HTTP PUT Method *Example*

### SERVER

```
@RequestMapping(value = "/employees/{id}", method = RequestMethod.PUT)
public ResponseEntity<Employee> updateEmployee(@PathVariable("id") int id,
                                                @RequestBody Employee employee)
```

### CLIENT

```
final String uri = "http://localhost:8080/springrestexample/employees/{id}";
Map<String, String> params = new HashMap<String, String>();
params.put("id", "2");
```

```
Employee updatedEmployee = new Employee(2, "New Name", "Gilly", "test@email.com");
RestTemplate restTemplate = new RestTemplate();
restTemplate.put ( uri, updatedEmployee, params);
```



# Spring RestTemplate

## HTTP DELETE Method Example

### SERVER

```
@RequestMapping(value = "/employees/{id}", method = RequestMethod.DELETE)
public ResponseEntity<String> updateEmployee(@PathVariable("id") int id)
```

### CLIENT

```
final String uri = "http://localhost:8080/springrestexample/employees/{id}";
```

```
Map<String, String> params = new HashMap<String, String>();
params.put("id", "2");
```

```
RestTemplate restTemplate = new RestTemplate();
restTemplate.delete( uri, params );
```



# SPRING MESSAGING



# Spring Messaging

## Messaging

- Relates to asynchronous communication
- Requires message broker to be available at any time
- Message broker maintains message queues & topics
- Producers publishes messages to queues / topics
- Consumers consumes messages from queues / topics



# Spring Messaging

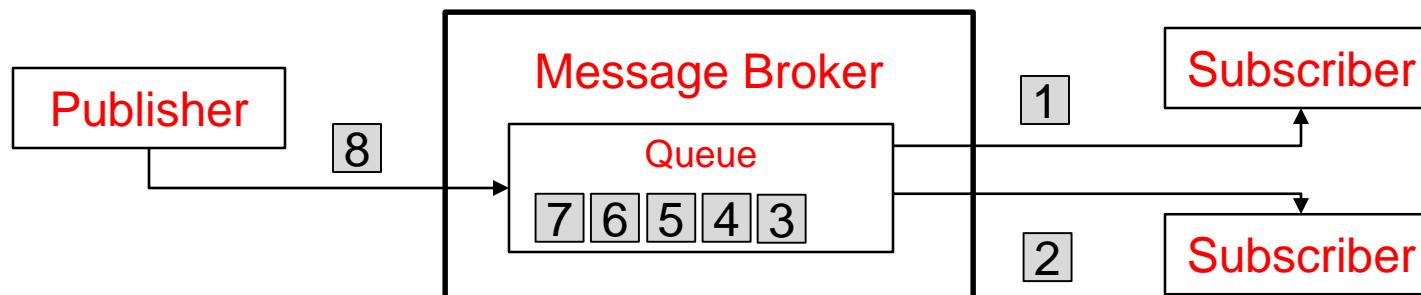
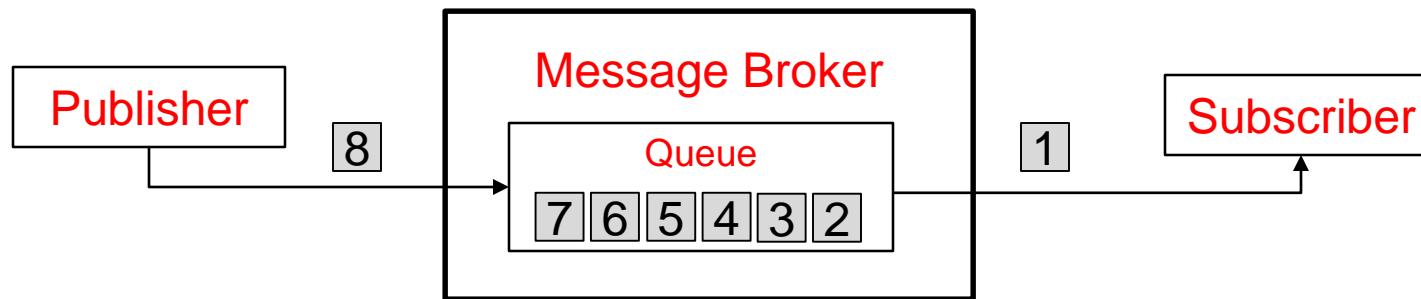
## Messaging

- Message distribution strategies:
  - P2P
    - Uses queue
    - One point sends message
    - One point receives it
    - Multiple consumer & producers queues
      - Production goes to the same queue – no ordering
      - Every message is still distributed to only one consumer
        - round-robin
        - fixed or other...

# Spring Messaging

## Messaging

- Message distribution strategies:
  - P2P - Queue





# Spring Messaging

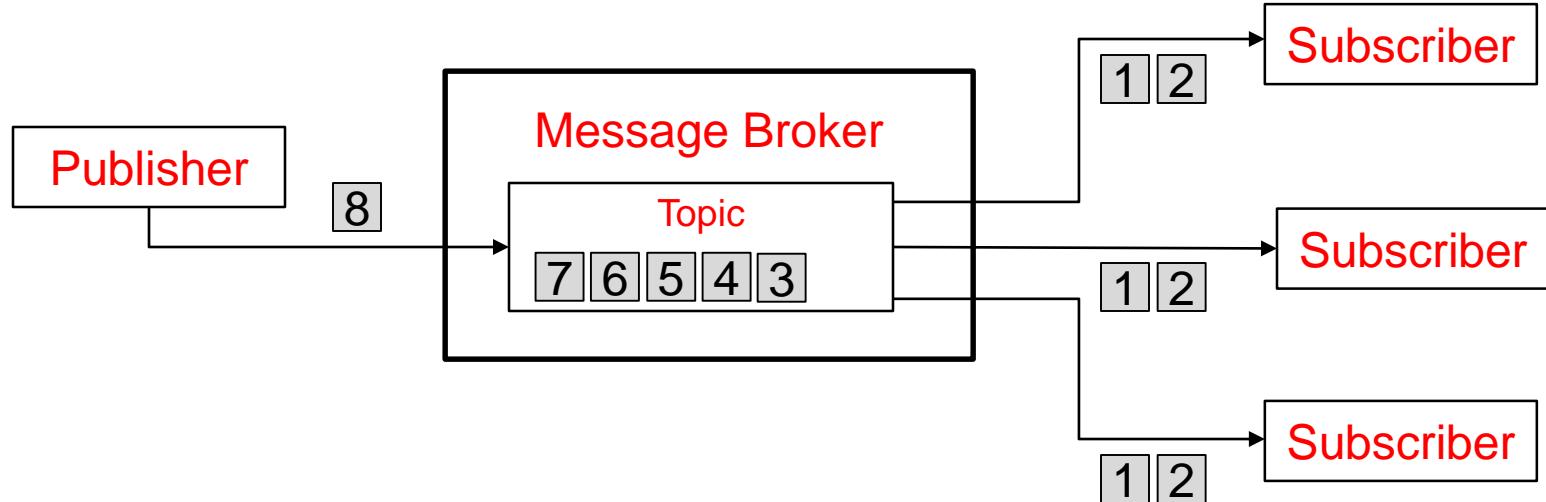
## Messaging

- Message distribution strategies:
  - PUB-SUB
    - Uses topics
    - One/multiple point/s sends messages
    - Each message is distributed to all connected subscribers

# Spring Messaging

## Messaging

- Message distribution strategies:
  - PUB-SUB Topics





# Spring Messaging

## Common message properties

- id / key
- timestamp
- expiration (absolute values – not timestamps)
- queue – origin queue
- replyTo – reply queue (optional)
- redelivered flag
- type / description – saves time when handling messages



# Spring Messaging

## Acknowledges

- Usually consumers acknowledge messages upon retrieval
- Message broker can track and trace consumers
- The broker will re-send messages if not been acknowledged
- Acknowledgement strategies:
  - Eager
  - Lazy (A.K.A Dup-OK)
  - Manual (risky)



# SPRING MESSAGING WITH KAFKA

# Spring Messaging with kafka

## Kafka

- Apache Kafka is a distributed streaming platform
- Supports
  - Publishing & subscribing streams of records (messages)
  - Stream backup (uses file system)
  - Message processing & real-time events

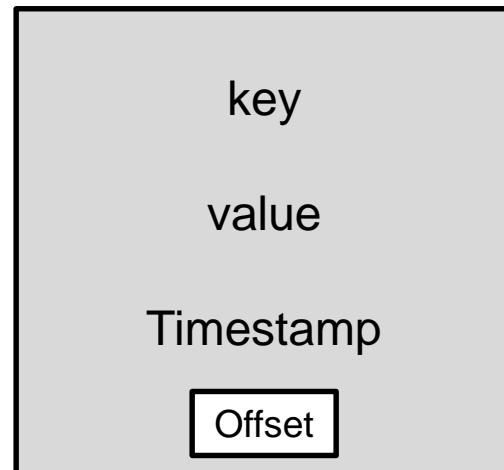




# Spring Messaging with kafka

## Kafka

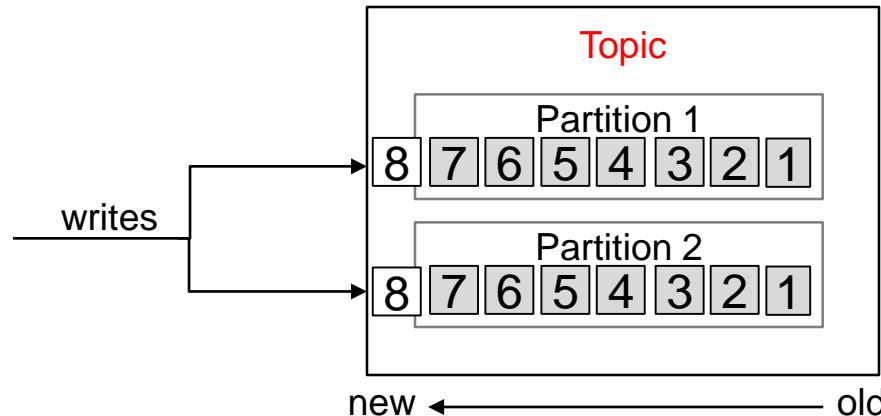
- Kafka supports server cluster, therefore can span over multiple data centers
- Uses high-performance TCP communication
- Kafka provides Topics
- Kafka message infrastructure:
  - Key – message identifier
  - Value – the actual value
  - Timestamp
  - Offset - later



# Spring Messaging with kafka

## Kafka - Topics

- Topics may have 0,1 or multiple consumers
- Kafka distributes topics to multiple partitions which allows:
  - High availability
  - Parallel consumption





# Spring Messaging with kafka

## Kafka – backup & HA

- Topics log/backup
  - Kafka backups messages to secondary storage
  - Backup quota can be configured with time & sizes
  - Consumers may load ‘old’ data as long as it is still stored
  - When expires – old data is discarded and overridden by new data

# Spring Messaging with kafka

## Kafka – Partitions

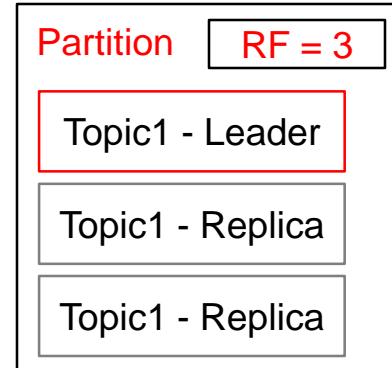
- Topics may be partitioned
  - Each partition may hold multiple topic replicas
  - Consumers and producers may specify partition to work with
  - Topic replicas are better to be hosted on separate physical machines
- Partitioned Topics N factor (replication factor - RF)
  - Defines number of in-memory topic replications in each partition
  - Means that a Kafka topic setup with RF=3 keeps 3 copies of the topic in every dedicated partition



# Spring Messaging with kafka

## Kafka - Partitions

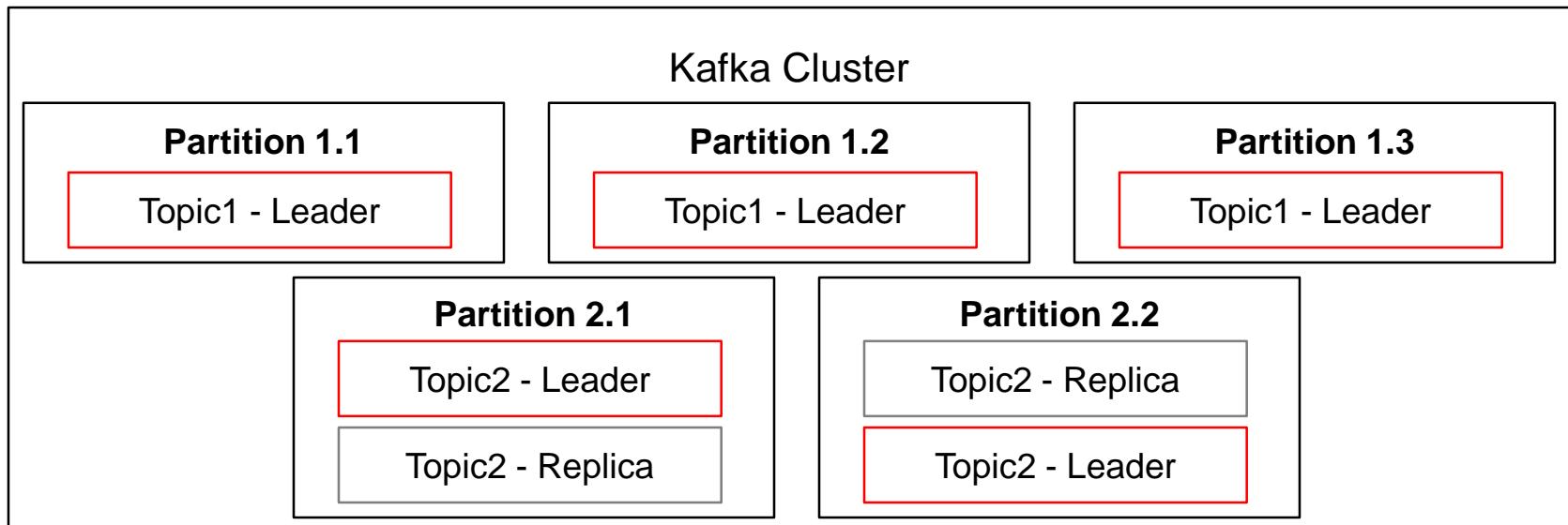
- In each partition, Topic replicas are divided into 2 types:
  - Leader
    - all writes & reads goes through it
    - responsible for synchronizing all other relevant replicas
  - Replicas
    - copy of the data – in-memory replication
    - may become Leader if current leader fails
- Offset is the sequential location of a message in a specific partition
- Kafka remembers offset for each consumer as it consumes messages



# Spring Messaging with kafka

## Kafka - Broker

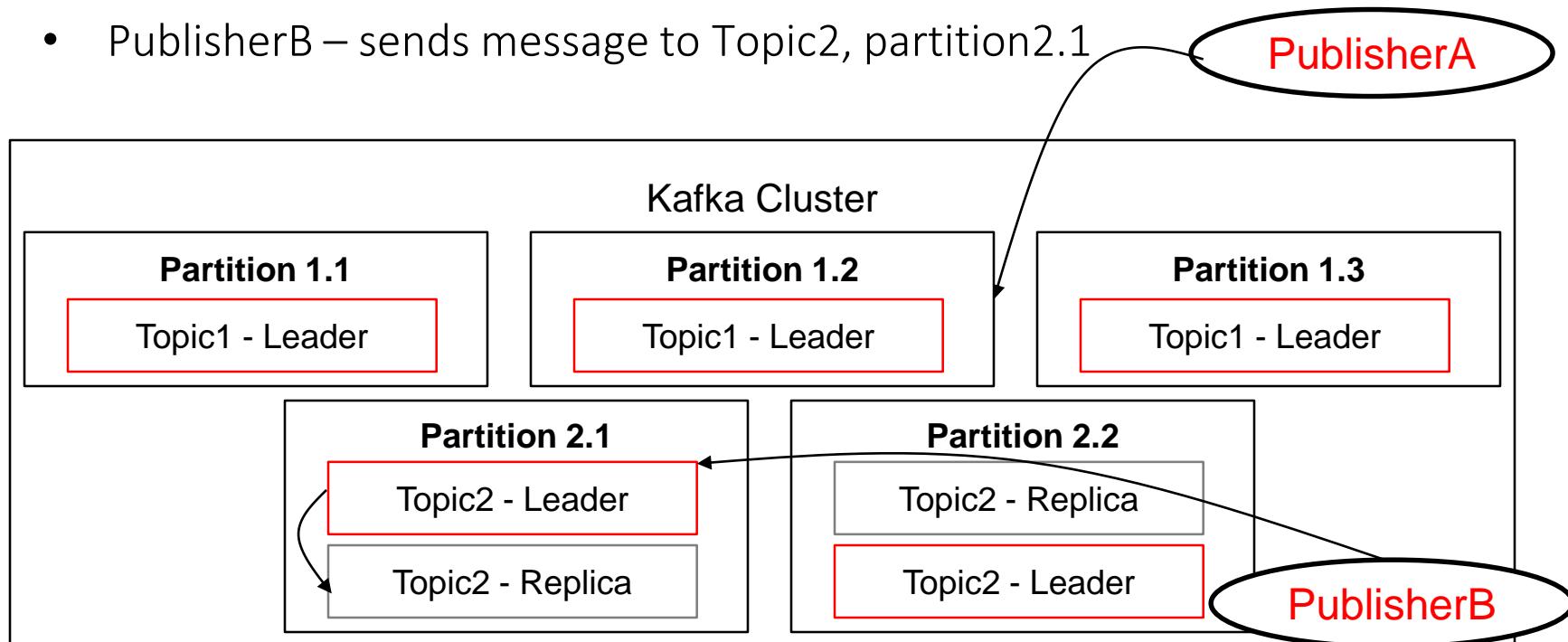
- Here is an example of:
  - 2 partitioned topics: topic1, topic2.
  - Topic1 uses 3 partitions
  - Topic1 uses 2 partitions
  - Topic1 RF is set to 1 – so there is only one copy of each topic in each partition
  - Topic2 RF is set to 2 – so there are 2 copies of each topic in each partition



# Spring Messaging with kafka

## Kafka - Producers

- Producers publish to the leader on a specific partition
- Here's an example of 2 publishers
  - PublisherA – sends message to Topic1, partition1.2
  - PublisherB – sends message to Topic2, partition2.1





# Spring Messaging with kafka

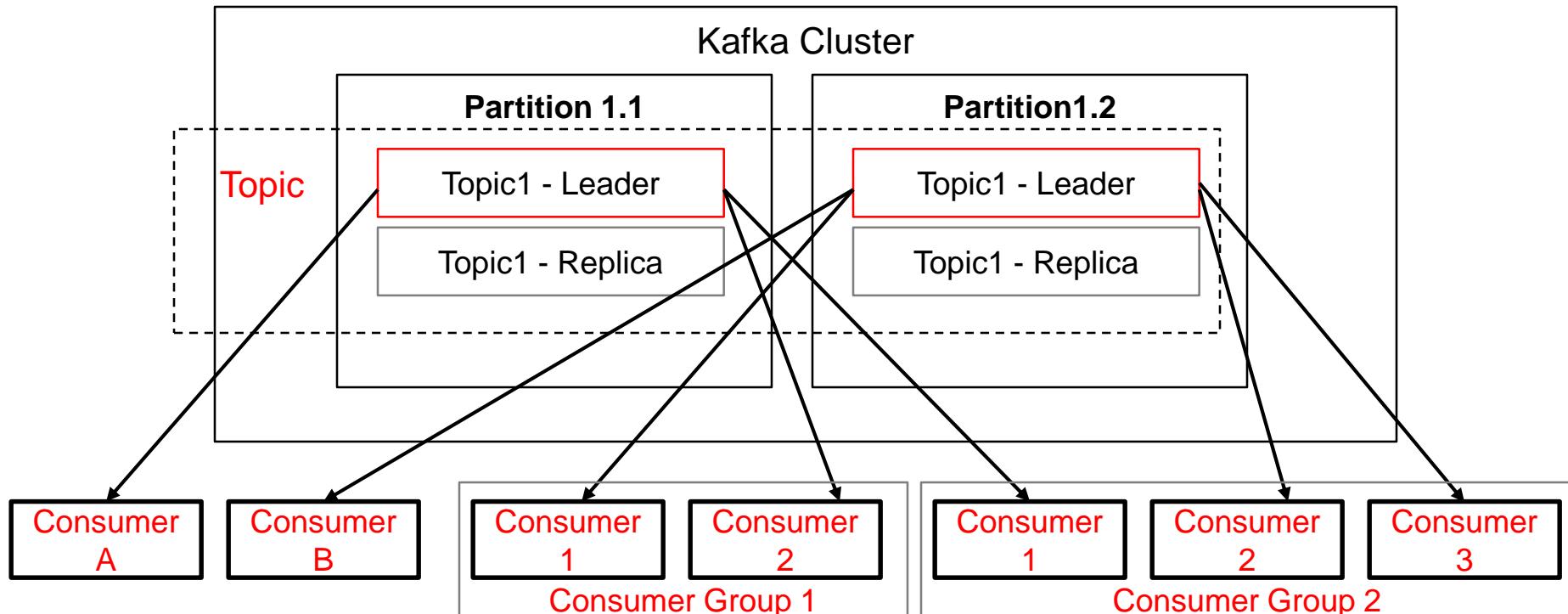
## Kafka - Consumers

- Consumer receives data from topics / partitioned topics
- When multiple consumers are connected to the same topic
  - Each consumer receives published messages
- Consumer may be a member in a consumer-group
  - Messages are load-balanced among consumers
  - Each consumer get part of the whole data
  - If there are more consumers than partitions – some will not receive anything...
- Kafka optimizes work among available cluster members according to connected consumers and consumer-groups

# Spring Messaging with kafka

## Kafka - Consumers

- Example – consumers and consumer-groups receives messages from a clustered broker with a partitioned topic:



# Spring Messaging with kafka

Kafka uses Zookeeper



**Apache ZooKeeper™**

Zookeeper helps in:

- leader detection
- configuration management
- synchronization
- detecting when a new node joins or leaves the cluster
- Future kafka releases are planned to be Zookeeper free

# Spring Messaging with kafka

## Installing & running kafka server

- Download kafka-2.0.0-src.tgz from <https://kafka.apache.org/downloads>
- Extract file using tar utility

```
cd C:\kafka  
C:\kafka>tar -xzf kafka_2.11-2.0.0.tgz
```

- Open cmd and launch zookeeper

```
C:\kafka>cd kafka_2.11-2.0.0  
C:\kafka\kafka_2.11-2.0.0>cd bin  
C:\kafka\kafka_2.11-2.0.0\bin>cd windows  
C:\kafka\kafka_2.11-2.0.0\bin\windows>zookeeper-server-start.bat ..\..\config\zookeeper.properties
```

- Open cmd and launch kafka – default port: 9092

```
C:\kafka>cd kafka_2.11-2.0.0  
C:\kafka\kafka_2.11-2.0.0>cd bin  
C:\kafka\kafka_2.11-2.0.0\bin>cd windows  
C:\kafka\kafka_2.11-2.0.0\bin\windows>kafka-server-start.bat ..\..\config\server.properties
```

# Spring Messaging with kafka

## Defining topics

- New topic arguments
  - Zookeeper address (stands behind Kafka which uses localhost:9092)
  - Replication factor
  - Number of partitions
  - Topic name

```
C:\kafka\kafka_2.11-2.0.0\bin\windows>kafka-topics.bat --create --zookeeper localhost:3030 --replication-factor 1 --partitions 1 --topic test
```

- List topics running on zookeeper

```
C:\kafka\kafka_2.11-2.0.0\bin\windows>kafka-topics.bat --list --zookeeper localhost:3030
```

# Spring Messaging with kafka

## Publishing messages

- Using kafka command-line sender client

```
C:\kafka\kafka_2.11-2.0.0\bin\windows>kafka-console-producer.bat --broker-list localhost:9092 --topic test
Message 1
Message 2
```

## Receiving messages

- Using kafka command-line receiver client

```
C:\kafka\kafka_2.11-2.0.0\bin\windows>kafka-console-consumer.bat –bootstrap-server localhost:9092
                                         --topic test
                                         --from-beginning
```

Output:

Message 1  
Message 2

# Spring Messaging with kafka

## SpringBoot support for kafka

- Create topics
- Use pre-configured producers
- Use pre-configured consumers
- KafkaTemplate – powerful and simple publishers
- Publish listeners



# Spring Messaging with kafka

Maven dependencies:

```
<dependency>
    <groupId>org.springframework.kafka</groupId>
    <artifactId>spring-kafka</artifactId>
    <version>${spring-kafka.version}</version>
</dependency>
<dependency>
    <groupId>org.springframework.kafka</groupId>
    <artifactId>spring-kafka-test</artifactId>
    <version>${spring-kafka.version}</version>
    <scope>test</scope>
</dependency>
```

# Spring Messaging with kafka

## application.properties

- group-id
  - consumers group to share consumption
- offset-reset
  - start-from strategy
    - earliest – from oldest available value
    - latest – from last published value
- Kafka hosts
  - default is localhost:9092
  - use spring.kafka.bootstrap-servers property to set custom hosts
- app.topic.<name>
  - custom topic names

```
spring.kafka.consumer.group-id= foo
spring.kafka.consumer.auto-offset-reset= earliest
app.topic.foo= foo.t
```

# Spring Messaging with kafka

## SpringKafka Sender – KafkaTemplate

- maintains weak-refs of publisher configuration
- creates topics
- provides simple methods for sending messages
- lists topic partitions
- binds producer listener
- sets message converter

```
@Component
public class Sender {

    @Autowired
    private KafkaTemplate<String, String> kafkaTemplate;

    @Value("${app.topic.foo}")
    private String topic;

    public void send(String message){
        kafkaTemplate.send(topic, message);
    }
}
```

# Spring Messaging with kafka

## SpringKafka Sender – KafkaTemplate

- KafkaTemplate<K,V>
- send (...)
  - send(Message)
    - Message contains key, value & timestamp
  - send(String topic, V data)
    - sends ‘null’ key message with given value
  - send(String topic, int partition, V data)
    - same but to specified partition
    - use partitionFor(String topic) : List<PartitionInfo>



# Spring Messaging with kafka

## SpringKafka Sender – KafkaTemplate

- All send (...) methods results with `ListenableFuture<K,V>`
- `ListenableFuture` allows to track sending process
  - `get()` - blocks
  - `isDone()`, `isCanceled()`
  - `cancel()` – discards tracking
  - `addCallback(ListenableFutureCallback<SendResult<K,V>>)`
    - `SendResult` has `onSuccess(K,V)` & `onError(Throwable)`

# Spring Messaging with kafka

## SpringKafka Sender – KafkaTemplate

- ListenableFuture

```
public class Sender{

    public void send(){
        kafkaTemplate.setDefaultTopic("topic-name");
        ListenableFuture<SendResult<String, String>> future = kafkaTemplate.send(topic, value);
        future.addCallback(new ListenableFutureCallback<SendResult<String, String>>() {
            @Override
            public void onSuccess(SendResult<String, String> result) {
                System.out.println("success");
            }
            @Override
            public void onFailure(Throwable ex) {
                System.out.println("failed");
            });
    });
}
```

# Spring Messaging with kafka

## SpringKafka Receiver – @KafkaListener

- Registers consumer to given topic/s
- Denotes asynchronous call when message is dispatched by broker
- @KafkaListener keeps SpringBoot application waiting

```
@Component
public class Receiver {

    @KafkaListener(topics = "${app.topic.foo}")
    public void receive(@Payload String message, @Headers MessageHeaders headers) {
        System.out.println("message: " + message);
    }
}
```

# Spring Messaging with kafka

## Running

```
@SpringBootApplication
public class Main implements CommandLineRunner{

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

    @Autowired
    private Sender sender;

    @Override
    public void run(String... strings) throws Exception {
        sender.send("*** MESSAGE 1 ***");
        sender.send("*** MESSAGE 2 ***");
        try{Thread.sleep(5000);}catch(Exception e){}
        sender.send("*** MESSAGE 3 ***");
    }
}
```

This delay might cause sender to discard current configuration  
So, it might be forced to create new one when thread awakes



# SPRING MESSAGING WITH RABBITMQ

# Spring Messaging with RabbitMQ

## RabbitMQ

- Is also a message broker platform
- Like kafka, messages are much lighter than in JMS
- Messages can be everything and by default converted into byte[]
- Custom converters (e.g JSON converter) can be set
- Uses TCP communication – AMQP (Advanced Message Queuing Protocol)
  - TCP connections – the connection between application and RabbitMQ
  - Channels – channels are hosted within a connection to communicate with Queues





# Spring Messaging with RabbitMQ

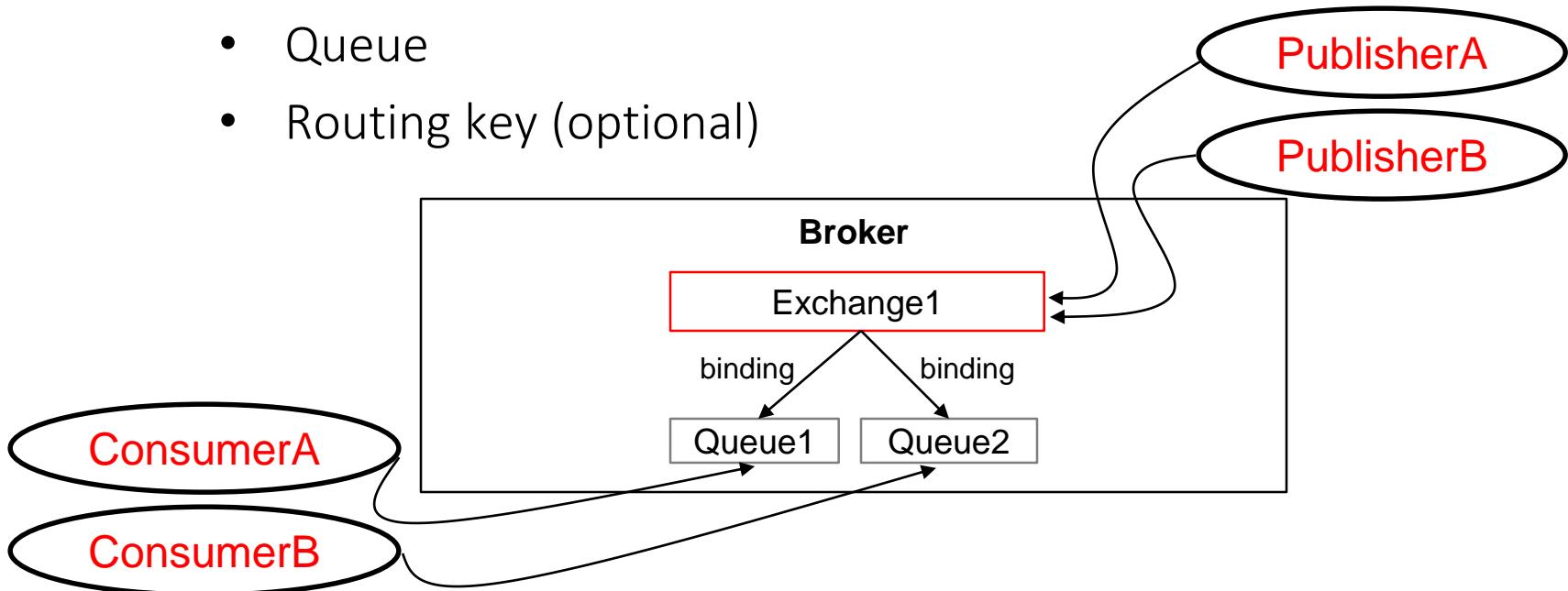
## RabbitMQ

- RabbitMQ exchanges
  - Exchange is the unit that receives messages sent via producers
  - Each exchange is bounded with queue(s)
  - Exchange delegates the message to its bounded queues
- RabbitMQ queues
  - Allows consumers to listen and wait for messages distribution
- Routing keys
  - Queues are bounded to exchanges
  - Binding may have a routing key used by publishers
  - This way exchanges may perform sophisticated queue routing

# Spring Messaging with RabbitMQ

## RabbitMQ

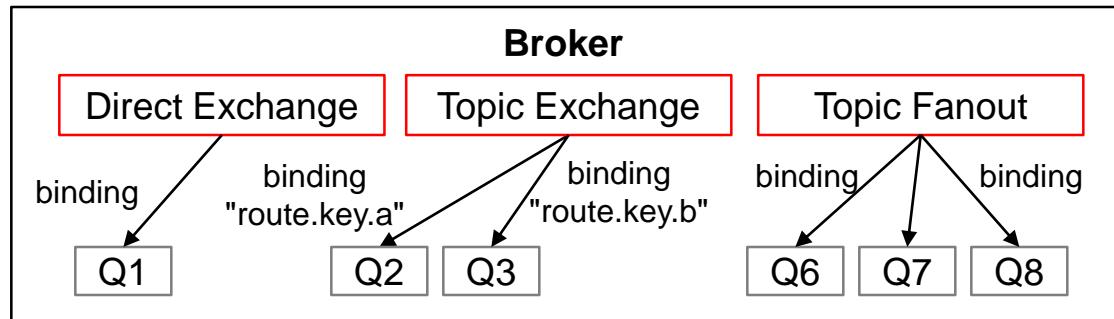
- RabbitMQ Binding
  - Binding describes the linkage between
    - Exchange
    - Queue
    - Routing key (optional)



# Spring Messaging with RabbitMQ

## RabbitMQ

- Types of exchanges:
  - Direct – for P2P, binding only one queue with a single exchange
  - Topic – for PUB-SUB,
    - binding multiple queues with single exchange
    - counts routing-keys when delegating to queues
  - Fanout – for PUB-SUB
    - Delegates to all bounded queues
    - Doesn't count routing-keys



# Spring Messaging with RabbitMQ

## RabbitMQ – installing and running

- Download & install
  - otp\_win64\_21.0.1\*.exe – used by RabbitMQ for communication
  - rabbitmq-server-3.\*.\*.exe – RabbitMQ installer (depends on previous)
- In order to start RabbitMQ server
  - Go to C:\...\RabbitMQ Server\rabbitmq\_server-3.7.7\sbin
  - Run: rabbitmq-server.bat

```
C:\Administrator: RabbitMQ Command Prompt (sbin dir) - rabbitmq-server

C:\Development\Java\rabbitMQ\RabbitMQ Server\rabbitmq_server-3.7.7\sbin>rabbitmq-server

##  ##
## ##      RabbitMQ 3.7.7. Copyright (C) 2007-2018 Pivotal Software, Inc.
##### Licensed under the MPL. See http://www.rabbitmq.com/
##### ##
##### Logs: C:/Users/Rony/AppData/Roaming/RabbitMQ/log/RABBIT~1.LOG
          C:/Users/Rony/AppData/Roaming/RabbitMQ/log/rabbit@Rony-PC_upgrade.log

          Starting broker...
completed with 0 plugins.
```

# Spring Messaging with RabbitMQ

## SpringBoot and RabbitMQ – Spring AMQP

- provides RabbitTemplate to abstract and simplify work
  - Sending
  - Receiving
  - Attach listeners
  - Obtain and set exchange
  - Obtain and set routing key
  - Configure reply address



# Spring Messaging with RabbitMQ

Maven dependencies:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-amqp</artifactId>
    <version>${spring-kafka.version}</version>
</dependency>
```

# Spring Messaging with RabbitMQ

## RabbitMQ Bean Configuration

- RabbitTemplate depends on the following Beans:
  - For sending messages:
    - Queue
    - Exchange
    - Binding & routing keys
  - For subscribing:
    - Message Listener Adapter
      - Takes a receiver implementation and uses it to handle messages
    - Message Listener Container
      - Binds MessageListenerAdapter with a queue(s)

# Spring Messaging with RabbitMQ

## Configuring RabbitMQ Beans:

```
@Configuration
public class RabbitConfig {

    @Value("${ex.topic.exchange}") private String topicExchangeName;
    @Value("${example.rabbitmq.queue}") private String queueName;
    @Value("${ex.routing.key}") private String routingKey;

    @Bean
    public Queue queue() {
        return new Queue(queueName, false); //false means not-durable
    }

    @Bean
    public TopicExchange exchange() {
        return new TopicExchange(topicExchangeName);
    }

    @Bean
    public Binding binding(Queue queue, TopicExchange exchange) {
        return BindingBuilder.bind(queue).to(exchange).with("boot.rabbit.#");
    }
}
```

The code block shows three annotated beans: `queue()`, `exchange()`, and `binding()`. A callout bubble from the `queue()` annotation points to a box labeled "Queue". Another callout bubble from the `exchange()` annotation points to a box labeled "Exchange". A curved arrow from the `binding()` annotation points to two boxes: one labeled "Binding" and another labeled "Routing Key".

# Spring Messaging with RabbitMQ

## Supported Exchange Beans:

- DirectExchange
  - Can be set with queue name
  - Supports routing keys - with(...)

```
@Bean
public DirectExchange exchange() {
    return new DirectExchange(directExchangeName);
}
@Bean
public Binding binding(Queue queue, DirectExchange exchange) {
    return BindingBuilder.bind(queue).to(exchange).withQueueName();
}
```

# Spring Messaging with RabbitMQ

## Supported Exchange Beans:

- FanoutExchange
  - Is set without routing key

```
@Bean
public FanoutExchange exchange() {
    return new FanoutExchange(fanoutExchangeName);
}
@Bean
public Binding binding(Queue queue, FanoutExchange exchange) {
    return BindingBuilder.bind(queue).to(exchange);
}
```

# Spring Messaging with RabbitMQ

## Supported Exchange Beans:

- TopicExchange
  - Is set with routing key

```
@Bean
public TopicExchange exchange() {
    return new TopicExchange(topicExchangeName);
}
@Bean
public Binding binding(Queue queue, TopicExchange exchange) {
    return BindingBuilder.bind(queue).to(exchange).with("boot.rabbit.#");
}
```

# Spring Messaging with RabbitMQ

## Supported Exchange Beans:

- HeadersExchange
  - Are for filtering
  - All the following results with a Binding that can filter:
    - to(HeadersExchange).where(String key).matches(Object value)
    - to(HeadersExchange).where(String key).exists()
    - to(HeadersExchange).whereAll(Map<String, Object> allKeyValues).match()
    - to(HeadersExchange).whereAll(String...keys).exist()
    - to(HeadersExchange).whereAny(Map<String, Object> allKeyValues).match()
    - to(HeadersExchange).whereAny(String...keys).exist()

```
@Bean  
public HeadersExchange exchange() {  
    return new HeadersExchange(exchangeName);  
}
```

# Spring Messaging with RabbitMQ

## Creating senders with RabbitTemplate

```
@Component
public class Producer {

    @Autowired
    private RabbitTemplate rabbitTemplate;

    @Value("${ex.routing.key}")
    private String routingKey;

    public void send(String message){
        System.out.println("Sender - Sending message: "+message);
        rabbitTemplate.convertAndSend(RabbitConfig.topicExchangeName, routingKey , message);
    }
}
```

- convertAndSend accepts T and generates Message which contains T but there are other options
- Sender may use routing key

# Spring Messaging with RabbitMQ

## RabbitTemplate send(...)

- send(Message message)
  - Message contains:
    - byte[]
    - MessageProperties (id, timestamp, replyTo....)
- convertAndSend(Object data)
  - Creates a Message with given data
- Both allows assigning of
  - routing key – String
  - exchange - String
  - processor - MessagePostProcessor interface
    - postProcessMessage(Message msg)

# Spring Messaging with RabbitMQ

## Receiving messages

- You may use **@RabbitListener** which allows you to specify
  - queues - String[] – queues names

```
@Component
public class Consumer {

    @RabbitListener(queues="${example.rabbitmq.queue}")
    public void receivedMessage(String msg) {
        System.out.println("Received Message: " + msg);
    }
}
```



# Spring Messaging with RabbitMQ

RabbitTemplate supports message receiving as well

- `receive(...)`
  - Returns Message
- `receiveAndConvert(...)`
  - Returns Object
- Both allows assigning of
  - routing key – String
  - exchange – String
  - timeout – long (millis)
  - `ReceiveAndReplyCallback(T message, R reply)`
    - `handle(T):R`
    - Converts origin message body (T) into R and publishes back to sender

# Spring Messaging with RabbitMQ

## Running

```
@SpringBootApplication
public class Main implements CommandLineRunner{

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

    @Autowired
    private Producer sender;

    @Override
    public void run(String... strings) throws Exception {
        sender.send("**** MESSAGE 1 ****");
        sender.send("**** MESSAGE 2 ****");
        try{Thread.sleep(5000);}catch(Exception e){}
        sender.send("**** MESSAGE 3 ****");
    }
}
```



# SPRING TESTING

SpringBoot supports the following test-oriented packages:

- **JUnit** - standard for unit testing Java applications
- **Spring Test & Spring Boot Test**: Utilities and integration test support for Spring Boot applications
- **Mockito** - A Java mocking framework
- **JSONassert** - An assertion library for JSON
- **JsonPath** - XPath for JSON
- **AssertJ**
- **Hamcrest** - A library of object comparison and matching



# SpringBoot Testing

SpringBoot Testing main annotations:

- `@SpringBootTest`
  - Creates an `ApplicationContext` based on your `SpringApplication`
  - Scans up from the test package to track `@SpringBootApplication` class
  - Main attributes:
    - configuration class – specifies configuration classes
      - Usually specifies the main configuration – `SpringApplication` class
      - Not required if `SpringApplication` is up the test package hierarchy
    - properties – allows defining key-value settings to your test
    - webEnvironment – determines the server policy for Web tests
      - MOCK (default) – embedded servers are not started
      - RANDOM\_PORT & DEFINED\_PORT – launched embedded servers
      - NONE – disables web environment if present

## SpringBoot Testing main annotations – cont.

- `@RunWith`
  - Must be set with `SpringRunner.class`
    - Extends `SpringJUnit4ClassRunner` which plays JUnit

```
@RunWith(SpringRunner.class)
@SpringBootTest(classes = Application.class)
public class MyTests {

    ...
}
```

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class MyTests {

    ...
}
```



# SpringBoot Testing

## SpringBoot Testing main annotations – cont.

- `@TestConfiguration & @Import`
  - Like `@Configuration` – allows to configure more beans to the context
  - Unlike `@Configuration` – is added to `ApplicationContext` and not replaces it
  - Disables component scan - (which might be a good thing when `SpringContext` loads testing beans we don't need for current test....)
  - Use `@Import` to enable specific configuration

## Web testing specifics

- Web applications might be request-oriented or reactive
- Since different containers are used – it must be set as part of the test
- Use @SpringBootTest.properties attribute to
  - set your web-app type: mvc /reactive
  - Define your embedded server policy via webEnvironment property

```
@RunWith(SpringRunner.class)
@SpringBootTest(properties = "spring.main.web-application-type=reactive",
               webEnvironment=WebEnvironment.DEFINED)
public class WebFluxTests { ... }
```



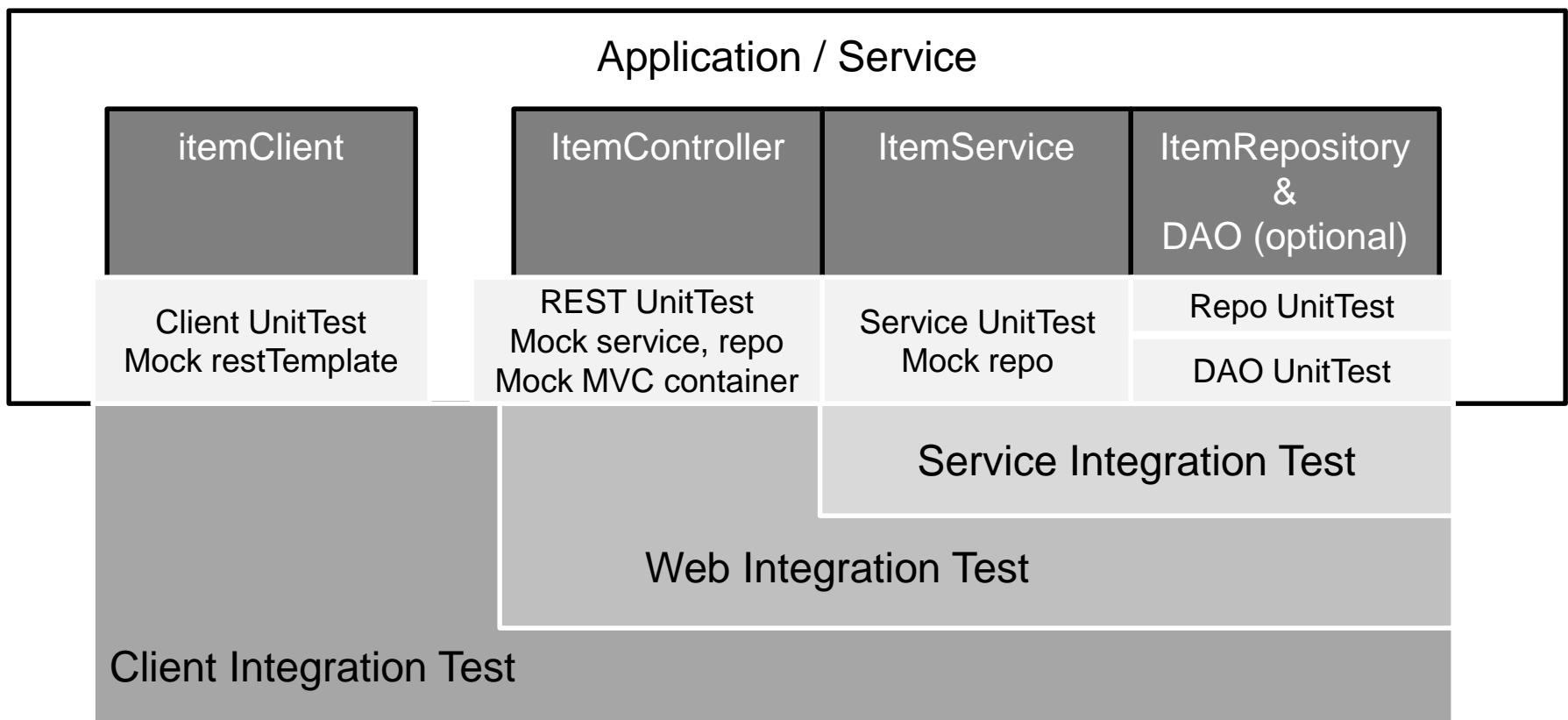
# SpringBoot Testing

## Unit and integration testing

- Unit testing
  - Testing ‘flat’ components
  - For components with dependencies – we mock dependencies
- Integration testing
  - Testing components with dependencies
  - Dependencies are partially mocked or not mocked at all
  - Can be done in any level – tier down testing

# SpringBoot Testing

## Unit and integration testing





# SpringBoot Testing

## Unit testing

- Unit testing – simple example
  - Testing ‘flat’ components

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class ServiceAUnitTests{

    @Autowired
    private ServiceA serviceA;

    @Test
    public void test1(){...}

    @Test
    public void test2(){...}

    ...
}
```

# SpringBoot Testing

## Unit testing - Mocking

- Unit testing – components with dependency injection

- Mock dependencies

- Spring can mock component bounded to ApplicationContext

- Assume serviceA is injected to serviceB

- When using **@MockBean** on serviceA – we mock all its instances on the context. Therefore serviceB gets it mocked already !

ItemService

Service UnitTest  
Mock repo

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class ServiceBUnitTests{
```

```
@MockBean
private ServiceA serviceA;
```

```
@Autowired
private ServiceB service;
```

```
@Test
public void test(){...}
```

```
...
```

## Unit testing – Testing DB

- When testing DB we usually check on our Repositories
  - Since most of the code is generated – not much to test..
  - But still – custom queries should be tested (custom findBy\*)
- In some cases we enhances repositories via DAO components
  - Allows to put some logic before actually going to the DB
  - DAOs are denoted as Spring components via @Repository
  - Are optional – but if exists – must be tested

ItemRepository  
&  
DAO (optional)

Repo UnitTest

DAO UnitTest

## Unit testing – Testing DB

- Testing Spring Repositories
  - For these cases Spring is capable of working with in-memory DB
  - Saves time in communicating with actual DB server
  - Doesn't update tables with testing data & activity
  - In order to use an alternative in-memory DB for testing:
    - Add DB dependency to pom.xml
    - Set in-memory DB properties - src/test/resources/application.properties
    - Enable alternative test DB via - @DataJpaTest
    - Inject & test your repositories (only Spring repositories)
    - NOTE: @Components, @Repository.. – are not injected!
  - If none of these configurations is set – real DB is used

ItemRepository  
&  
DAO (optional)

Repo UnitTest

# SpringBoot Testing

## Unit testing – Testing DB

- Testing Spring Repositories with alternative DB – example:

```
@SpringBootTest
@RunWith(SpringRunner.class)
@DataJpaTest
public class ItemRepositoryUnitTest {

    @Autowired
    private ItemRepository repo;

    @Test
    public void testItemBetweenPrices(){
        //insert some items...
        assertTrue(repo.findItemBetweenPrice(100,0).size() == 0);
    }
}
```

*pom.xml*

```
...
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <version>1.4.194</version>
</dependency>
...
```

*src/test/resources/application.properties*

```
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.url=jdbc:h2:mem:db;DB_CLOSE_DELAY=-1
spring.datasource.username=sa
spring.datasource.password=sa
```

## Unit testing – Testing DB

- Testing your DAOs
  - DAOs are basically denoted as @Repository
  - Therefore it is disabled when working with alternative DBs
  - You may:
    - Create new configuration to alternative DB
      - Define properties in custom file
      - Create @Configuration
      - Use @PropertySource("your-custom-props.properties")
    - Use the real DB
  - NOTE: if you have in-memory DB setting for tests – you need to disable it. This is done via: @AutoConfigureTestDatabase

ItemRepository  
&  
DAO (optional)

DAO UnitTest

# SpringBoot Testing

## Unit testing – Testing DB

- Testing custom DAOs – example:

```
@SpringBootTest
@RunWith(SpringRunner.class)
@AutoConfigureTestDatabase(replace=Replace.NONE)
public class ItemDAOUnitTest {

    @Autowired
    private ItemDAO dao;

    @Test
    public void testAddItem(){
        Item t=new Item();
        t.setName("Item2998");
        t.setAmount(-10); //wrong value...
        assertEquals(-999,dao.addItem(t));
    }
}
```

```
@Repository
public class ItemDAO {

    @Autowired
    private ItemRepository repo;

    @Test
    public long addItem(Item t){
        if(t.getAmount()<0)
            return -999;
        //else – persist item...
    }
}
```

## Unit testing – Web Unit testing

- Web unit tests are usually focused on:
  - A – check controller logic
    - verify it invokes the correct services
    - check data conversions
    - check custom validations
  - B – check service interface
    - url mappings
    - response statuses
    - response content types
- In order to perform ‘B’ tests you may mock your MVC container

ItemController

REST UnitTest  
Mock service, repo  
Mock MVC container

## Unit testing – Web Unit testing

- Mocking MVC container
  - Lightweight servlet container
  - For testing request mappings – counted as UnitTest..
  - Spring provides:
    - MockMvc – uses lightweight servlet container
    - request builders (content, headers, cookies, URL, redirect..)
    - response handlers & matchers
      - supports content based assertions (good for integration tests - later)
      - supports content-type based assertions (good for ‘B’ types)



# SpringBoot Testing

## Unit testing – Web Unit testing

- ResultActions MockMVC.perform(RequestBuilder builder)
  - RequestBuilder
    - Use MockMvcRequestBuilders utility class to factor builders
    - get(), post(), put(), delete()...
  - ResultActions
    - andExpect(ResultMatcher m)
    - Asserts the result of an executed request

```
...
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;
...
```



# SpringBoot Testing

## Unit testing – Web Unit testing

- ResultActions MockMVC.perform(RequestBuilder builder)
  - ResultMatcher
    - Use MockMvcResultMatchers.class utility class to factor Matchers
      - content(), status(), cookie(), header()...
      - Each results with a specific matcher to test specific values



# SpringBoot Testing

## Unit testing – Web Unit testing

- ResultActions MockMVC.perform(RequestBuilder builder)
  - ResultMatcher
    - Use MockMvcResultMatchers.class utility class to factor Matchers
      - content(), status(), cookie(), header()...
      - Each method results with a specific matcher
    - Use BaseMatcher<T> to create your own
      - override matches(Object value) with your own logic
  - Each matcher allows testing by focusing on specific HTTP parts



# SpringBoot Testing

## Unit testing – Web Unit testing

- ResultMatchers – MockMvcResultMatchers.class

### **Response Content (Body)**

- Method: content()
- Result: ContentResultMatcher
  - bytes(bytes[] expected)
  - contentType(MediaType expected)
  - encoding(String expected)
  - string(String expected)
  - json(String expected)
  - xml(string expected)

### **Response Headers**

- Method: header()
- Result: HeaderResultMatcher
  - dataValue(String name, String expected)
  - longValue(String name, String expected)
  - string(String name, String expected)
  - stringValues(String name, String ...expected)

## Unit testing – Web Unit testing

- ResultMatchers – MockMvcResultMatchers.class – cont.

### Response Cookies

- Method: status()
- Result: StatusResultMatcher
  - is(String expected)
  - isAccepted(), isFound(), isForbidden()
  - isBadGateway(), isVBadRequest()
  - isGone(), isOK, isNoContent()...

### Response Cookies

- Method: cookie()
- Result: CookieResultMatcher
  - comment(String name, String expected)
  - doesNotExist(String name) / exist(...)
  - domain(String expected) / path(...)
  - maxAge(String name, int expected)
  - value(String expected)
  - version(string expected)



# SpringBoot Testing

## Unit testing – Web Unit testing

- Mocking MVC container & using matchers

```
...
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;

...
@RunWith(SpringRunner.class)
@SpringBootTest
@AutoConfigureMockMvc
public class ServiceAControllerUnitTests{

    @Autowired
    private MockMVC mvc;

    @Test
    public void test1(){
        mvc.perform(get("/items/all").andExpect(status().isOk())
            .andExpect(content().contentType(MediaType.APPLICATION_JSON_UTF_8));
    }
}
```

## Unit testing – Web Unit testing

- Custom Matchers

```
...
    @Test
    public void test2(){
        BaseMatcher<String> matcher=new BaseMatcher<String>() {
            @Override
            public void describeTo(Description d) {
                d.appendText("Testing item name String format");
            }
            @Override
            public boolean matches(Object value) {
                if(value instanceof String){
                    return ((String)value).endsWith("###");
                }
                return false;
            }
        }
        mvc.perform(get("/item/name/1234").andExpect(status().isOk())
                    .andExpect(content().string(matcher)));
    }
    ...
}
```



# SpringBoot Testing

## Unit testing – Web Unit testing

- In cases where tested web-service invokes another one
  - You might want to mock the other service(s)
  - Use @MockBean on RestTemplate

```
...
@RunWith(SpringRunner.class)
@SpringBootTest
@AutoConfigureMockMvc
public class ServiceAControllerUnitTests{

    @Autowired
    private MockMVC mvc;

    @MockBean
    private RestTemplate restTemplate;
    ...

}
```

## Unit testing – Web Unit testing - JSON

- We might be willing to test JSON format & content as well
  - Assume binding in Java & Spring is bug free
  - Just verify you receive the JSON you expect
  - Denote your test with `@JsonTest`
  - Use `JacksonTester<T>` to test and assert
  - You may use Matchers here



# SpringBoot Testing

## Unit testing – Web Unit testing - JSON

- Example:

```
@RunWith(SpringRunner.class)
@JsonTest
public class ServiceAClientUnitTests{

    @Autowired
    private JacksonTester<Item> tester;

    @Autowired
    private RestTemplate restTemplate;

    public void testItemJson(){
        URL url=new URL("http://localhost:8080/entity/info/189675");
        String json="";
        try(BufferedReader in=new BufferedReader(new InputStreamReader(url.openStream()))){
            json=in.readLine();
        }catch(Exception e){...}
        test.parse(json).assertThat().hasFieldOrProperty("id").hasFieldOrProperty("price")
            .isEqualTo(someTestItem);
    }
}
```



# SPRING CLOUD MICROSERVICES



# Microservices

Definition:

“Microservices is a variant of the service-oriented architecture (SOA) architectural style that structures an application as a collection of loosely coupled services” (wiki)



# Microservices

Break application into small, fine-grained, standalone services

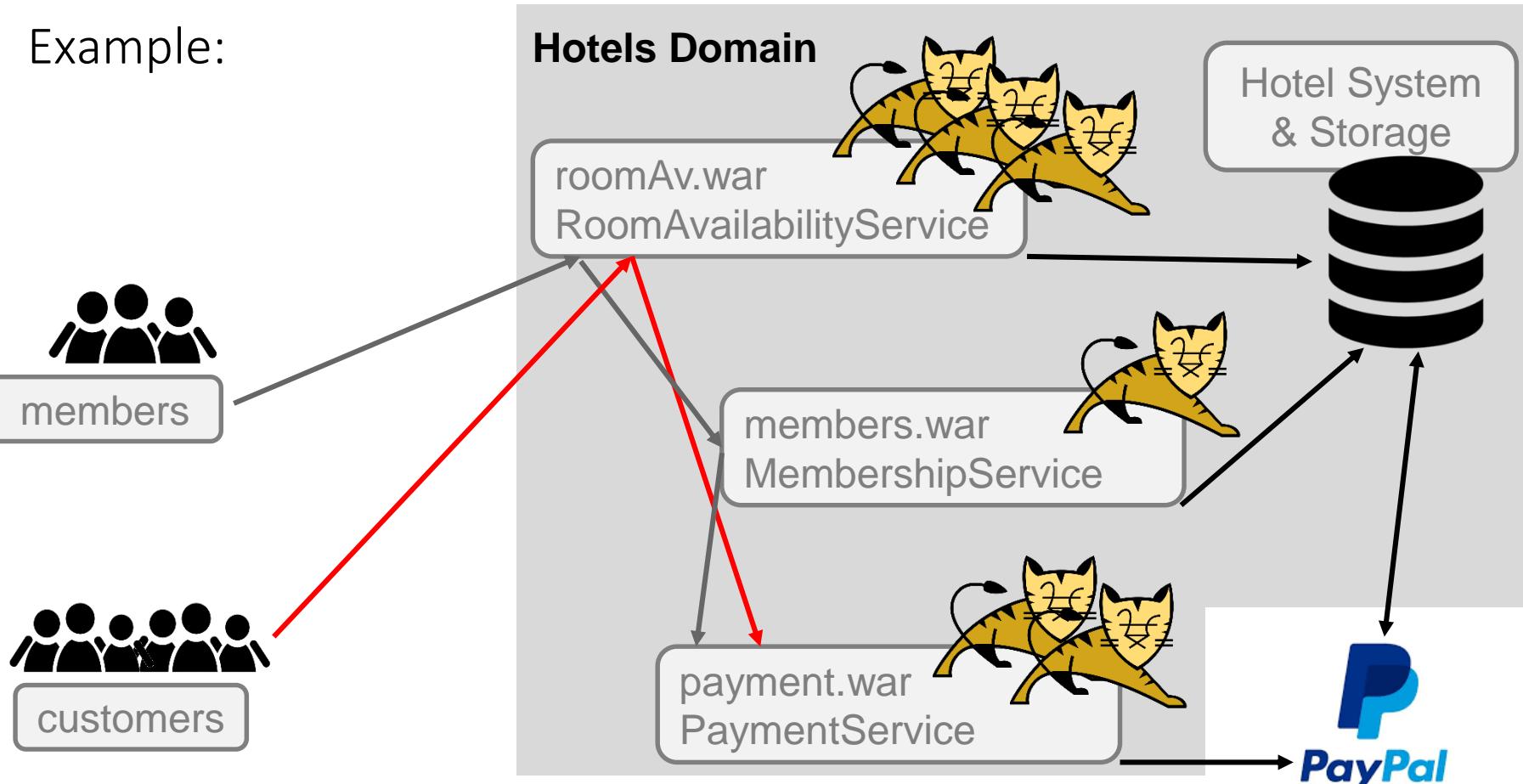
- Each service has
  - its contract (interface)
  - its own deployment settings
  - Its own container / server instance(s)

For new developments – design for service

For existing application - decompose

# Microservices

Example:





# Microservices

## Benefits:

- Each business component maintained separately
- Business flow based on services may use both internal and external business logic transparently
- Intensive releases and continuous developments are easier
- Integration mechanism is a non-issue
- Service may count on multiple alternative services for failover
- Service deployment, availability and maintainability can be automated via DevOps tools



# Microservices

## Drawbacks:

- Intensive HTTP communication (still text based...)
- Going stateless? Not always... and then what?
- 2-phase commit is not supported in any microservice – so can't be used...
- Higher costs – each microservice comes with its infrastructure....
- When microservices span over different systems or divisions few challenges emerges:
  - Team communication
  - Need in good documentation
  - Testing is hard when more than single service is involved



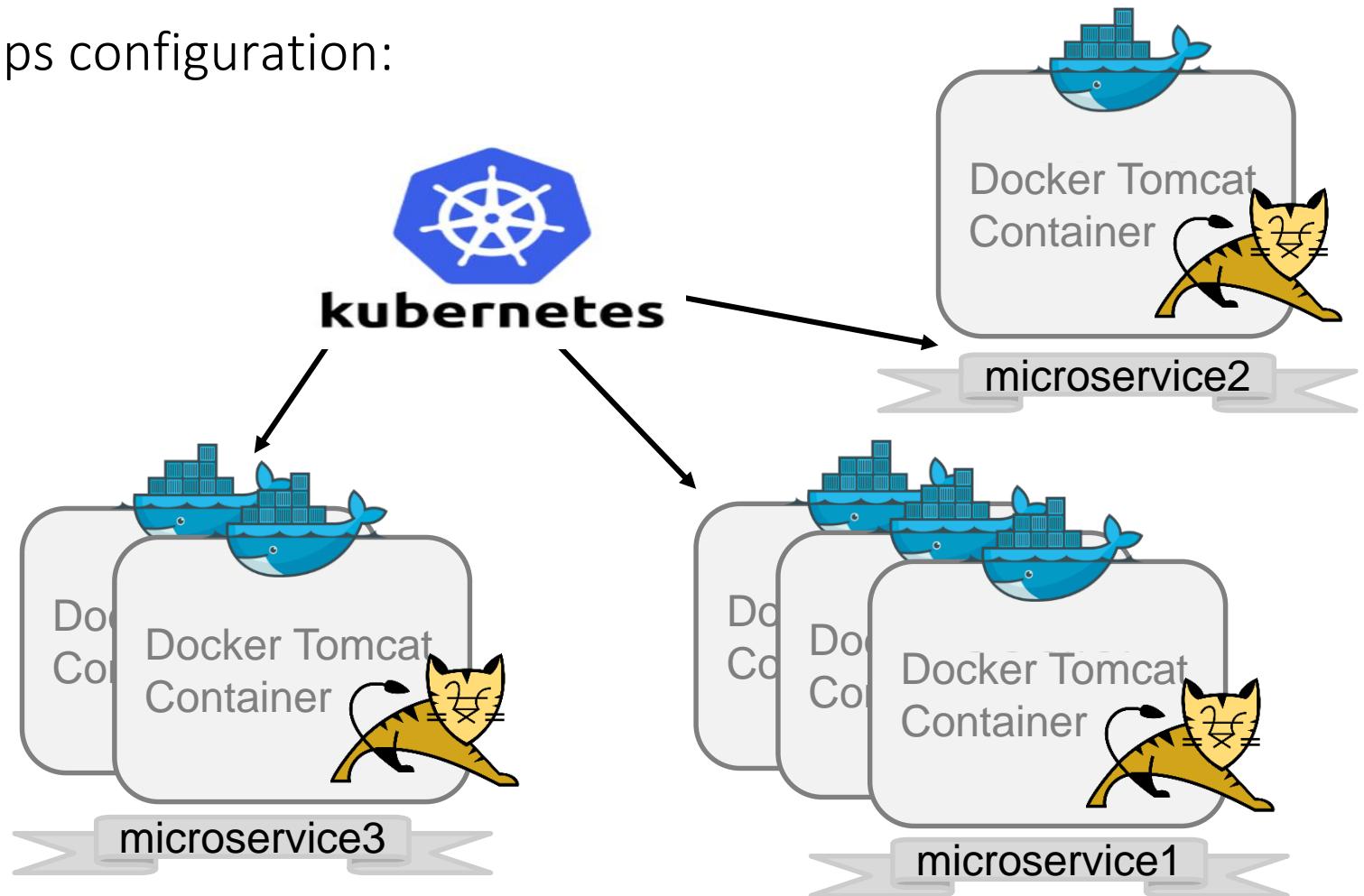
# Microservices

## Best practices:

- Create separate data-store per microservice
- Provide global data management system
  - Runs in the backgrounds
  - Fixes data inconsistencies
- Update code by creating a new microservice
  - Combine with version control
- Use separate builds for each service
- Run services in containers. Container is NOT a services
- Keep server configuration plain when administrating microservices
  - Any complex configuration should be set internally by the server

# Microservices

DevOps configuration:





## Microservice Ecosystem minimal requirements

### Configuration server

Loads the configuration information for all services

Usually obtains info from a GIT repository

### Service discovery

Responsible for maintaining a map of all active services

Checks heartbeat every few seconds

### API Gateway

Entry façade to the Microservice domain

Proxies requests to actual services

### Circuit breaker

Provides cross-service invocation tracking

Supports failover executions

### Monitoring

# SpringCloud - Introduction

- provides tools for out-of-the-box Microservice platform solutions
- Built on Spring-boot & Spring MVC
- Provides both abstractions & implementations
- Allows rapidly wrap a micro-service with all / most important capabilities in order to integrate into Microservice ecosystems
- Done mostly via annotations



Spring Cloud provides Cloud Native application style



# Features

- Distributed/versioned configuration
- Service registration and discovery
- Routing
- Service-to-service calls
- Load balancing
- Circuit Breakers
- Global locks
- Leadership election and cluster state
- Distributed messaging

- Configuration server with GIT
- Netflix (Eureka, Hystrix, Zuul...)
- Cloud Bus
- Cloud Foundry (SSH & oAuth2 support)
- Cloud Cluster (Zookeeper, Redis, Hazelcast, Consul)
- Cloud Data Flow (data composing)
- Cloud Stream (external resources – Kafka & RabbitMQ)
- Cloud Zookeeper (service discovery)
- Cloud Gateway (based on Project Reactor programmatic routing)
- Load balancing - Ribbon



# Spring Cloud Contexts

- Bootstrap application context
  - Is the absolute root context
  - Is the parent of the main application
  - Populates and shares environmental information (Environment)
    - Bootstrap properties cannot be overridden by application
    - Holds configuration info
    - Configuration files: bootstrap.properties / bootstrap.yml (instead of 'application')
  - Usually, loads remote configurations from Configuration Server
- ApplicationContext
  - Usually a direct sibling of Bootstrap Application Context
    - Use parent() to obtain bootstrap from application context
  - By default, application configuration is used when not found in bootstrap

- Generally, client Microservices use bootstrap.yml / bootstrap.properties
  - Bootstrap configuration allows getting conf. info from a remote repository
  - Bootstrap context will help in generating service application context
- Bootstrap configuration holds local configuration as well
  - Used in a case where remote configuration is unavailable
- The process:

Client-Microservice → configuration-server → GIT

if config server not available:

Client-Microservice → local configuration

# Spring Cloud + Netflix

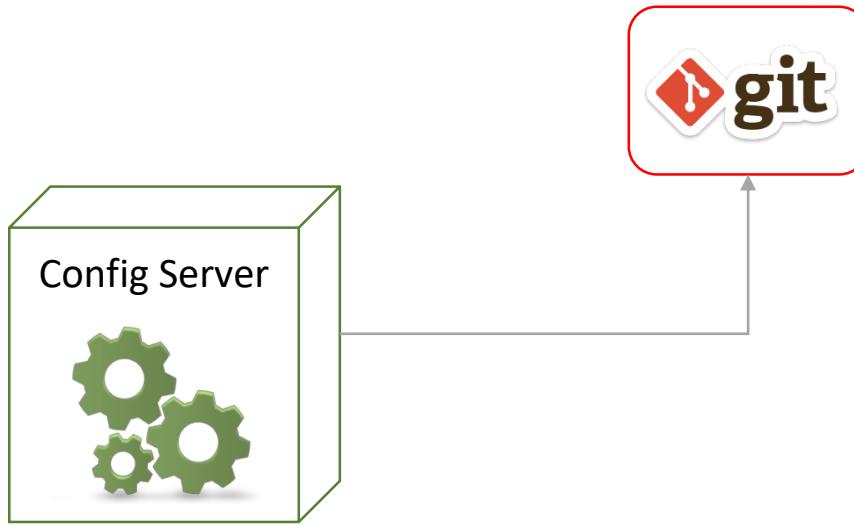
Common out-of-the-box solutions:

- Configuration server
- Service discovery - Eureka 
- API gateway - Zuul 
- Circuit breaker (load-balancing) – Hystrix 
- Monitoring – Hystrix 
- oAuth2 support Zuul 
- Client-side Load balancing - Ribbon



# Configuration Server

- Serves externalized configuration in a distributed system
- Provides server and client-side support
- Default implementation uses GIT to download configuration info



# Configuration Server

- Config Server

Maven dependencies:

```
pom.xml
...
<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-config-server</artifactId>
    </dependency>
</dependencies>
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>Edgware.SR2</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
...
```

dependency management  
takes care for the jar  
dependency chain

# Configuration Server

application.yml

```
spring:  
  cloud:  
    config:  
      server:  
        git :  
          uri: https://github.com/spring-cloud-repo  
  
server:  
  port: 9999
```

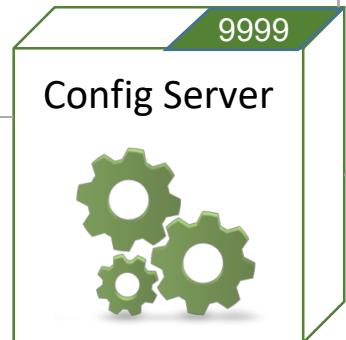
@SpringBootApplication

**@EnableConfigServer**

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

ConfigServer.java

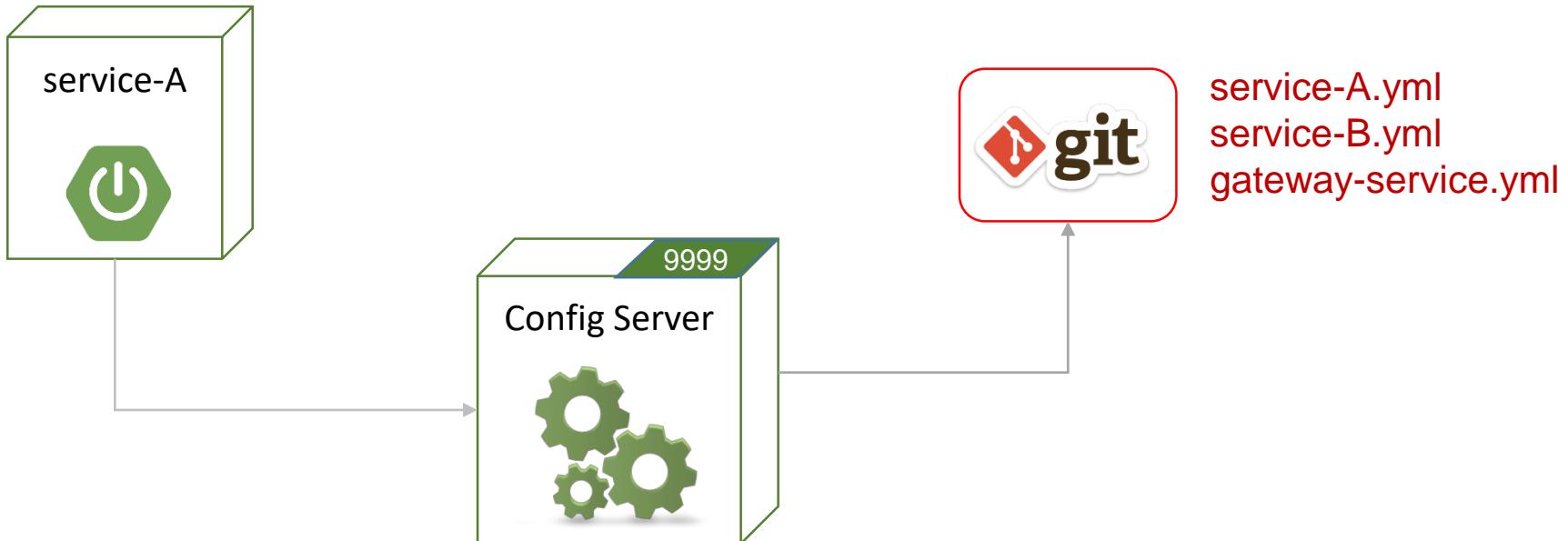
service-A.yml  
service-B.yml  
gateway-service.yml



# Configuration Server

Any service that uses Config-server will:

- Try to get \*.yml / \*.properties loaded from GIT
- If fails to connect to the Config-Server, uses its own local configuration



# Configuration Server

Microservices are set to connect to config server or use local configuration



**bootstrap.yml**

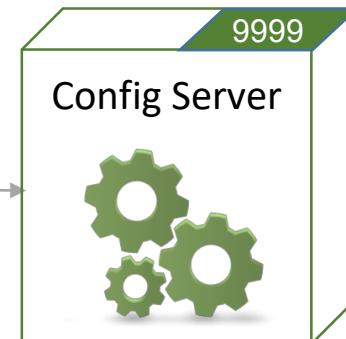
```
spring:
  application:
    name: service-A
  cloud:
    config:
      uri: http://localhost:9999
  server:
    host: localhost
    port: 7001
```



**service-A.yml (port:8001)**

```
@SpringBootApplication
public class MicroserviceA {
  public static void main(String[] args) {
    SpringApplication.run(MicroserviceA.class, args);
  }
}
```

**MicroserviceA.java**

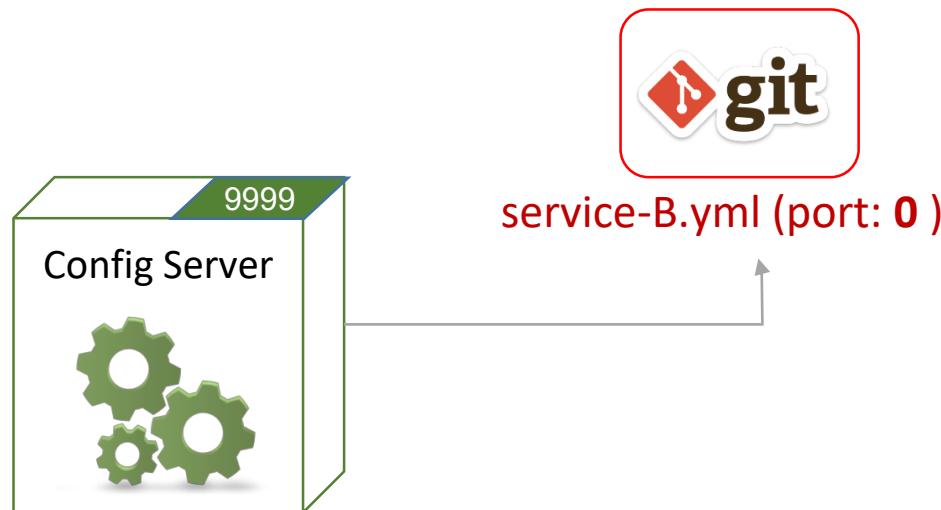


# Configuration Server

Running multiple Microservice instances

Assign zero value to port

Configuration server randomizes values



# Configuration Server

- Client Microservice  
Maven dependencies:

pom.xml

...

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

- Config Server
  - test server configuration:

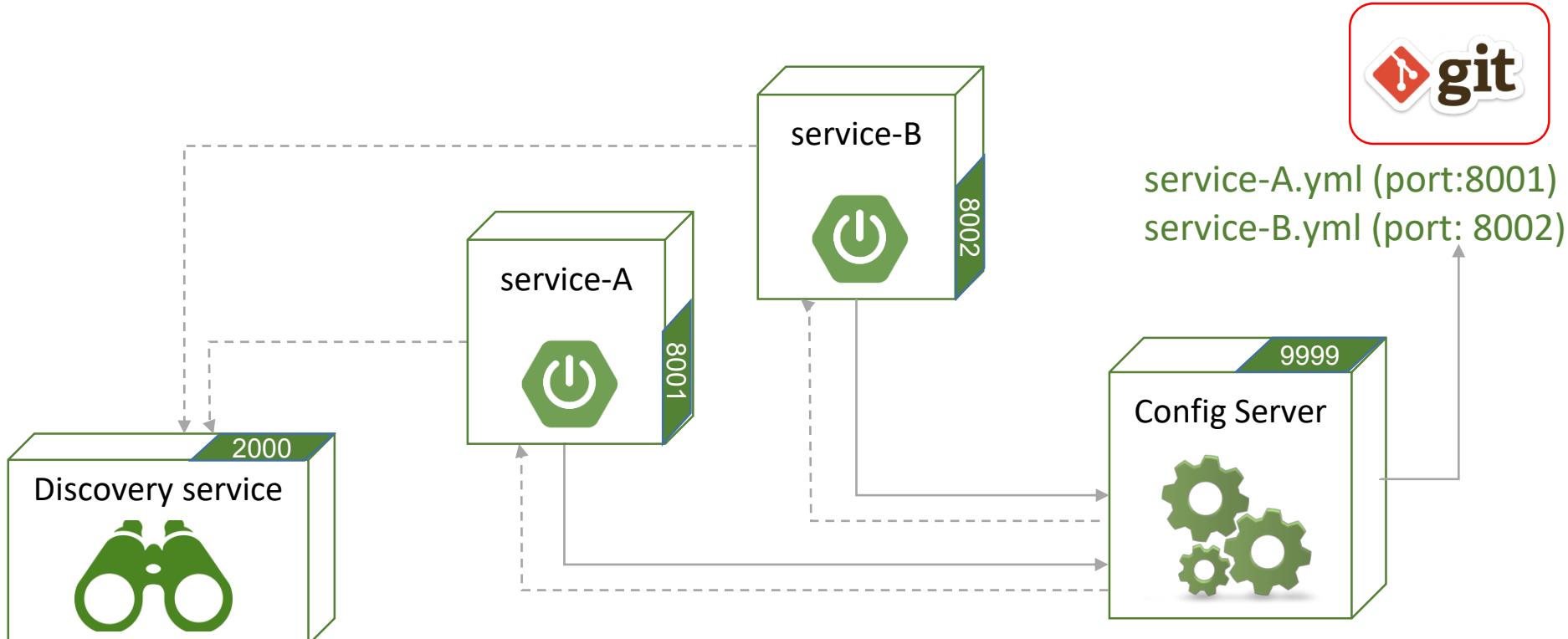
`http://localhost:9999/service-A/master`

`http://localhost:9999/service-B/master`

- Information is returned in JSON format

# Service Discovery - Eureka

- Service discovery receives notifications from other services
- Maintains health & state



# Service Discovery - Eureka

- Discovery server
- Maven dependencies:

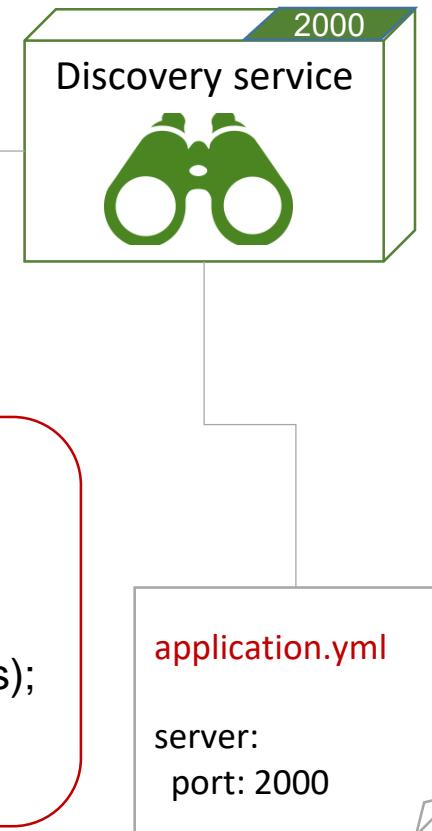
```
pom.xml
...
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-netflix-eureka-server</artifactId>
</dependency>
...
```

# Service Discovery - Eureka

Enabling Eureka discovery service in Spring Boot

```
@EnableEurekaServer
@SpringBootApplication
public class DiscoverServerApplication {
    public static void main(String[] args) {
        SpringApplication.run(DiscoverServerApplication.class, args);
    }
}
```

DiscoverServerApplication.java



# Service Discovery - Eureka

- Eureka clients are
  - Our Microservices
  - Eureka client – embedded client (requires URL configuration)



- Discovery server may be configured as:
  - Instance & Client
    - default
    - Runs continues intensive checks with embedded client
  - Instance –maintains available Microservices client list

Configuration:

```
application.yml
```

```
eureka:
  client:
    registerWithEureka: false
    fetchRegistry: false
    service-url:
      defaultZone: http://localhost:2000
  server:
    port: 2000
```

# Service Discovery - Eureka

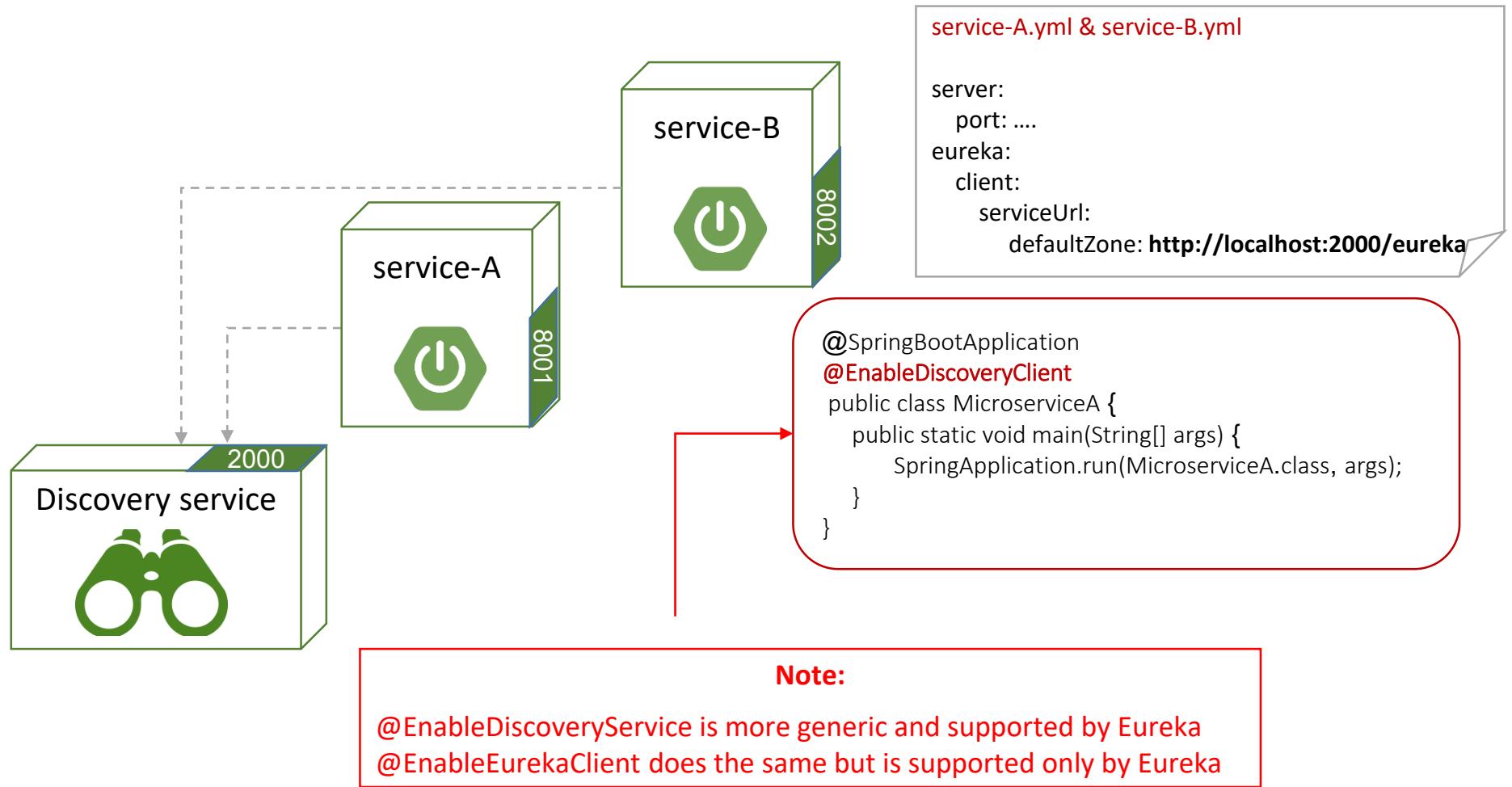
- Make your Microservice discoverable  
Maven dependencies:

pom.xml

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

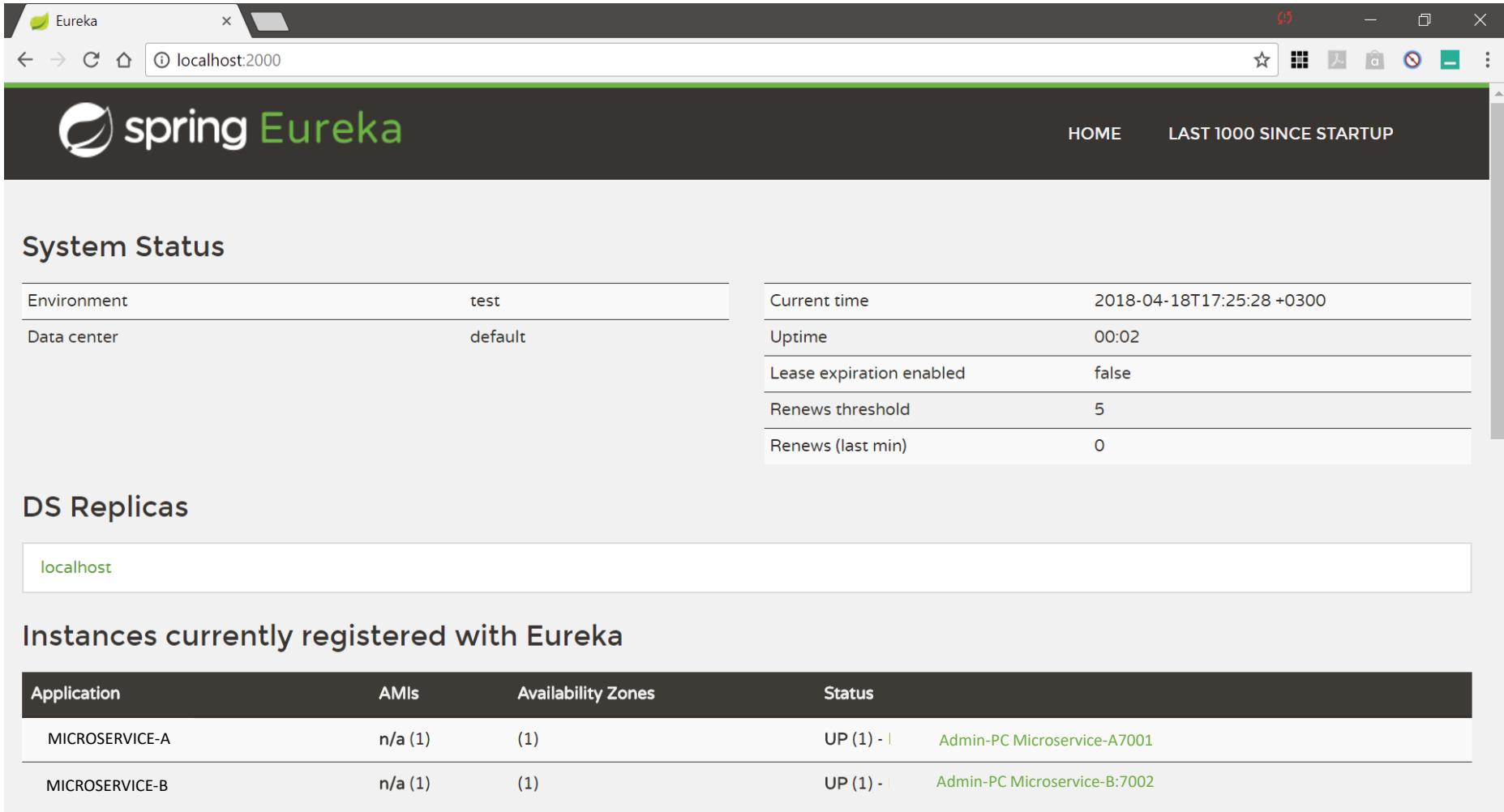
# Service Discovery - Eureka

- Microservices register to Discovery service
  - Configuration (local and/or on GIT):



# Service Discovery - Eureka

After starting discovery server we can monitor running services



The screenshot shows a web browser window with the title "Eureka". The address bar displays "localhost:2000". The page content is the Spring Eureka dashboard.

**System Status**

Environment	test
Data center	default

Current time	2018-04-18T17:25:28 +0300
Uptime	00:02
Lease expiration enabled	false
Renews threshold	5
Renews (last min)	0

**DS Replicas**

localhost

**Instances currently registered with Eureka**

Application	AMIs	Availability Zones	Status	
MICROSERVICE-A	n/a (1)	(1)	UP (1) -	Admin-PC Microservice-A:7001
MICROSERVICE-B	n/a (1)	(1)	UP (1) -	Admin-PC Microservice-B:7002

# Service Discovery - Eureka

- Eureka uses 90 sec as default ‘refresh’ interval

- In order to customize it, for example to 10 sec

- add the following props to

- Eureka’s application.yml:

## application.yml

```
eureka:  
  instance:  
    lease-renewal-interval-in-seconds: 10  
    lease-expiration-duration-in-seconds: 10  
  server:  
    response-cache-auto-expiration-in-seconds: 10  
  client:  
    registry-fetch-interval-seconds: 10  
  ...
```

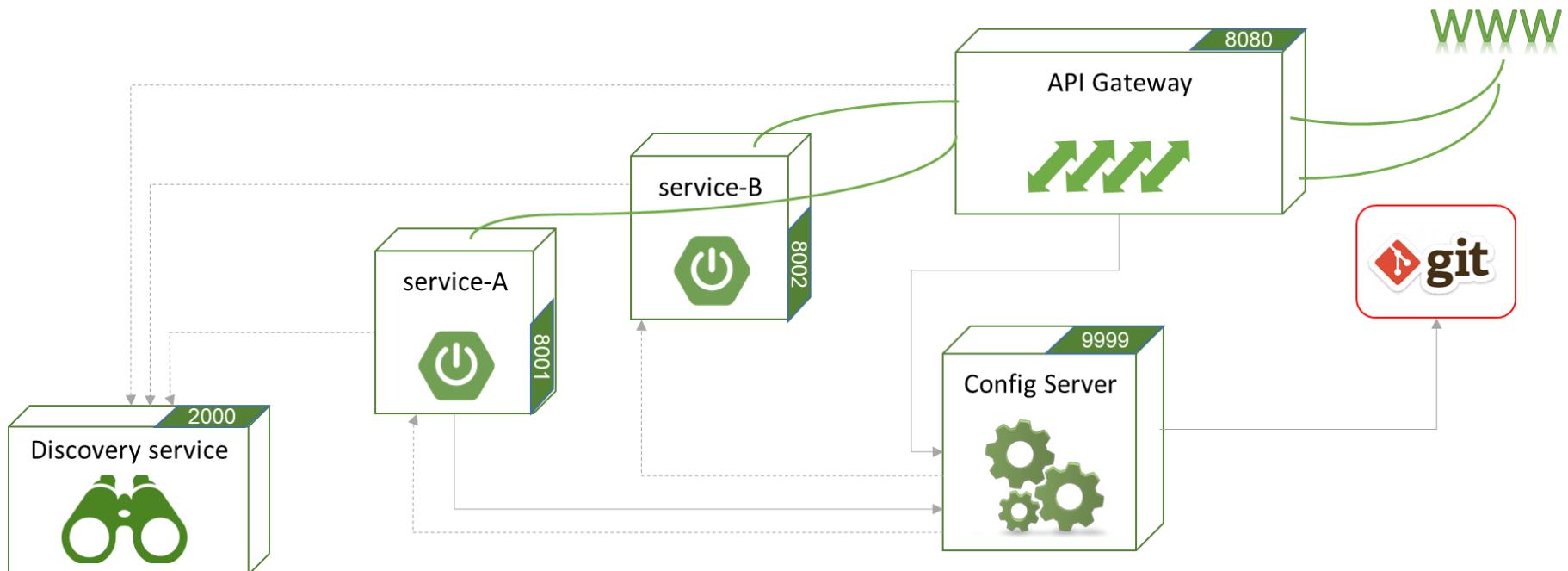
## bootstrap.yml

```
eureka:  
  instance:  
    lease-renewal-interval-in-seconds: 10  
    lease-expiration-duration-in-seconds: 10  
  client:  
    registry-fetch-interval-seconds: 10  
  ...
```

- And the following to each Microservice bootstrap.yml:

# API Gateway - Zuul

- Spring Cloud provides an Embedded Zuul proxy
  - Acts as API Gateway



# API Gateway - Zuul

- Gateway server Zuul  
Maven dependencies:

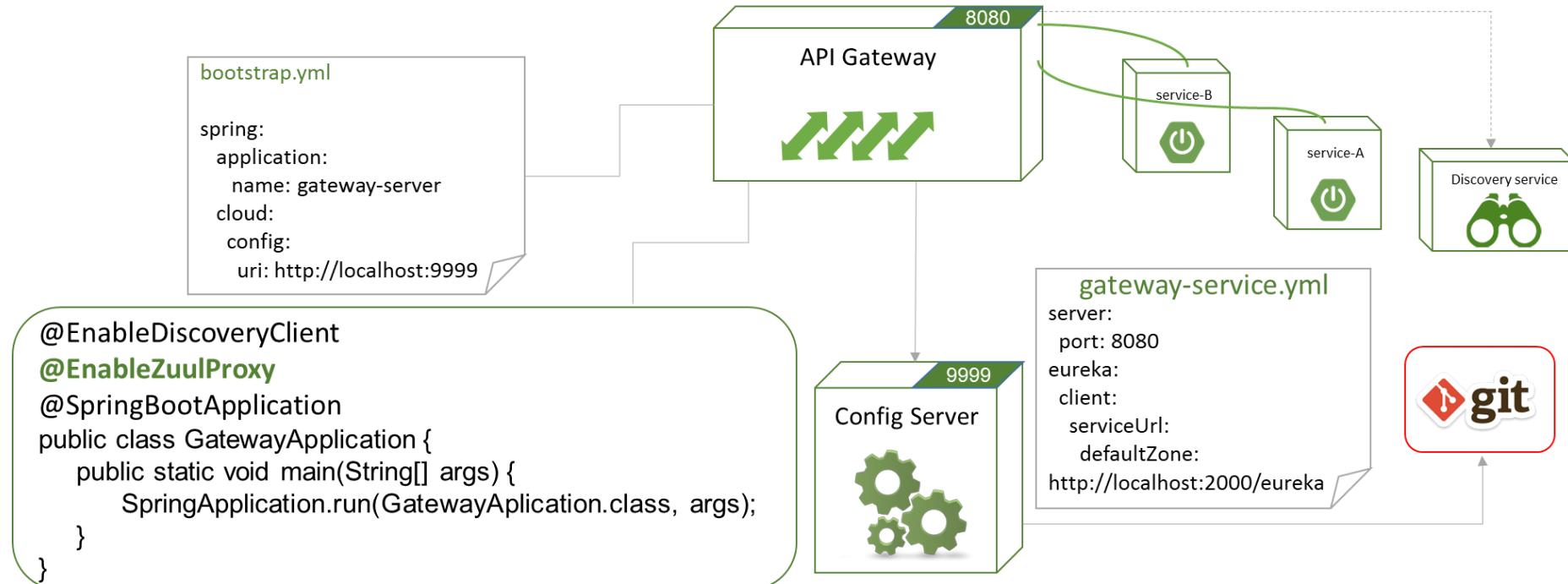
pom.xml

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

```

# API Gateway - Zuul

- Spring Cloud provides an Embedded Zuul proxy
  - Acts as API Gateway



# API Gateway - Zuul

- When starting Gateway server it:
  - downloads configuration from Config Server (port 8080)
    - Port (8080)
    - discovery server registration info
  - Queries Discovery Server for all running services
  - Before running Gateway:
    - `http://localhost:8001/..microserviceA`
    - `http://localhost:####/..microserviceB`
  - After running Gateway – services also available on:
    - `http://localhost:8080/service-a/..microserviceA`
    - `http://localhost:8080/service-b/..microserviceB`

**Note:**

Zuul uses LOWER-CASE proxy names regardless actual application name as specified in bootstrap.yml  
e.g: 'service-A' is tracked via 'service-a'

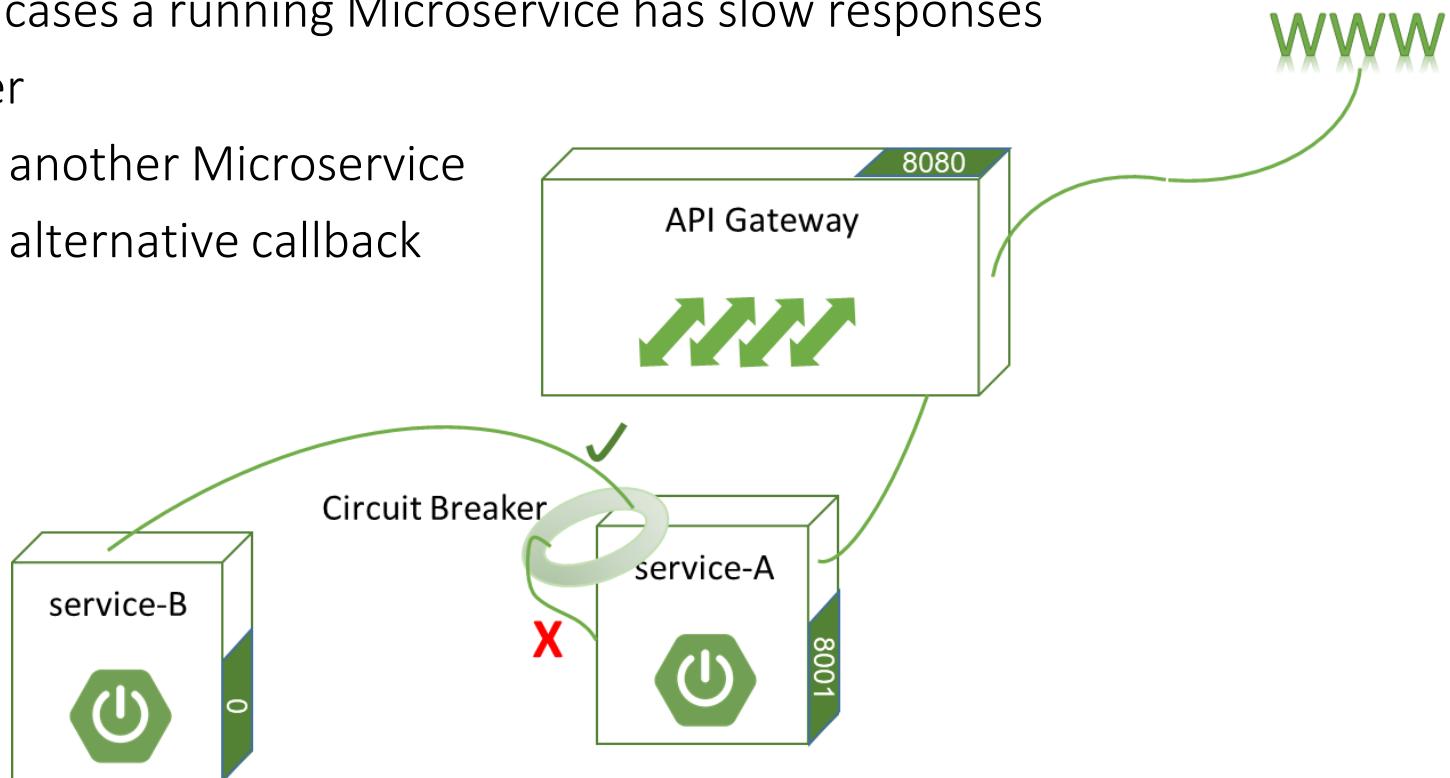


service-A.yml  
service-B.yml

- Zuul uses Ribbon implementation
  - Ribbon is used for load-balancing when multiple Microservice instances exists
  - Default load-balance algorithm is Round-Robin
- Currently, we can run multiple instance but only one will be discovered by Eureka
  - This is because our instances registers to Eureka with the same instance Id
  - Eureka keeps track of the last instance and ignores all others....
  - Later we will solve this issue by assigning different instance Id to Eureka

# Circuit Breaker – Hystrix

- Hystrix implements Circuit breaker DP
  - Tracks requests
  - Cancel policy (5 failures in 20 sec.)
    - In cases a running Microservice has slow responses
  - Failover
    - ✗ to another Microservice
    - ✗ to alternative callback

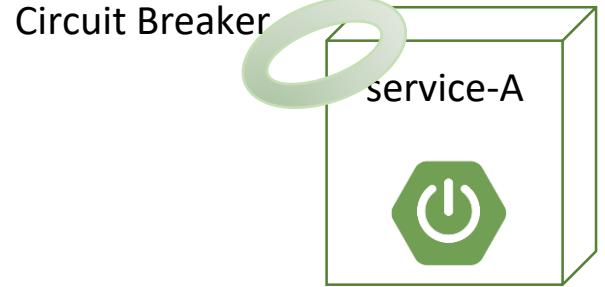


# Circuit Breaker – Hystrix

- Circuit Breaker is configured on client Microservice:

```
@SpringBootApplication
@EnableDiscoveryClient
@EnableCircuitBreaker
public class MicroserviceA {
    public static void main(String[] args) {
        SpringApplication.run(MicroserviceA.class, args);
    }

    @Configuration
    class Config {
        @LoadBalanced ←
        @Bean
        public RestTemplate testTemplate(){
            return new RestTemplate();
        }
    }
}
```



**Note:**

@LoadBalanced puts all resource activity on a Ribbon  
Circuit Breaker works on load balanced resources only (but not vice versa)

# Circuit Breaker – Hystrix

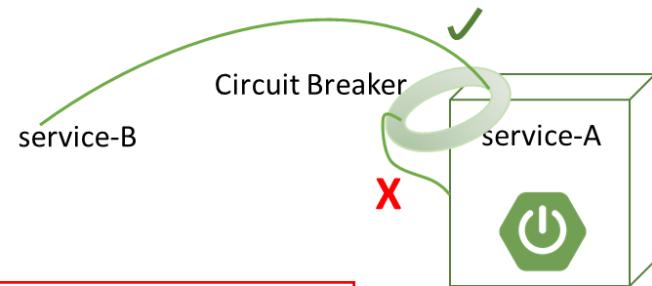
- Client Microservice Hystrix circuit breaker Maven dependencies:

```
pom.xml
...
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-hystrix</artifactId>
</dependency>
...
```

# Circuit Breaker – Hystrix

- Using load-balanced resources:
    - `@HystrixCommand` – defines failover callback & meta data
    - URL is resolved by Gateway Server (“/service-B/”)

```
@Autowired  
private RestTemplate restTemplate;  
  
@HystrixCommand(fallbackMethod = "fallback", groupKey = "srvA",  
    commandKey = "srvA", threadPoolKey = "srvAThread")  
@GetMapping("/serviceA")  
public String method() {  
    String url = "http://service-B/some/url";  
    return restTemplate.getForObject(url, String.class);  
}  
  
public String fallback(Throwable hystrixCommand) {  
    return "Fall Back Message";  
}
```



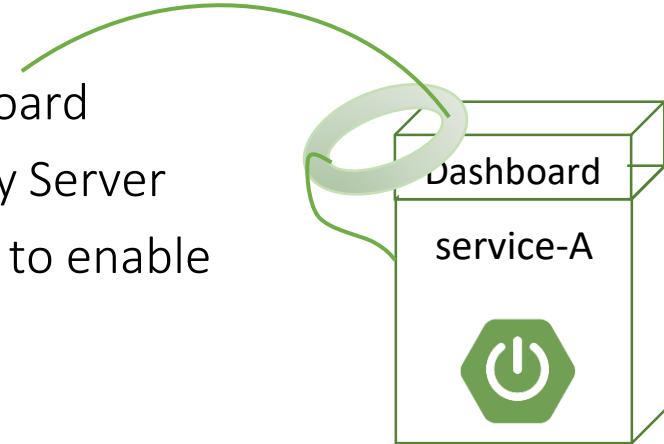
## Note:

Pay attention to the fact that we use a URL without mentioning its host:port

Hystrix counts on information taken from Config & Discovery and maintains a valid server-list

# Monitoring – Hystrix Dashboard

- Hystrix commands are gathered
- Command can be monitored via Hystrix Dashboard
- In most cases Dashboard is enabled on Gateway Server
- Denote service with `@EnableHystrixDashboard` to enable
- Tracks all endpoints in a Microservice
  - Success, failure, short-circuited, timed-out
  - Execution time
  - Traffic



```
@EnableDiscoveryClient  
@EnableZuulProxy  
@EnableHystrixDashboard  
@SpringBootApplication  
public class GatewayApplication {  
    public static void main(String[] args) {  
        SpringApplication.run(GatewayApplication.class, args);  
    }  
}
```

# Monitoring – Hystrix Dashboard

Gateway / client Microservice

Hystrix dashboard Maven dependencies:

pom.xml

```
...
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-hystrix-dashboard</artifactId>
</dependency>
<!-- for hystrix dashboard streaming --&gt;
&lt;dependency&gt;
    &lt;groupId&gt;org.springframework.boot&lt;/groupId&gt;
    &lt;artifactId&gt;spring-boot-starter-actuator&lt;/artifactId&gt;
&lt;/dependency&gt;
...</pre>
```

# Monitoring – Hystrix Dashboard

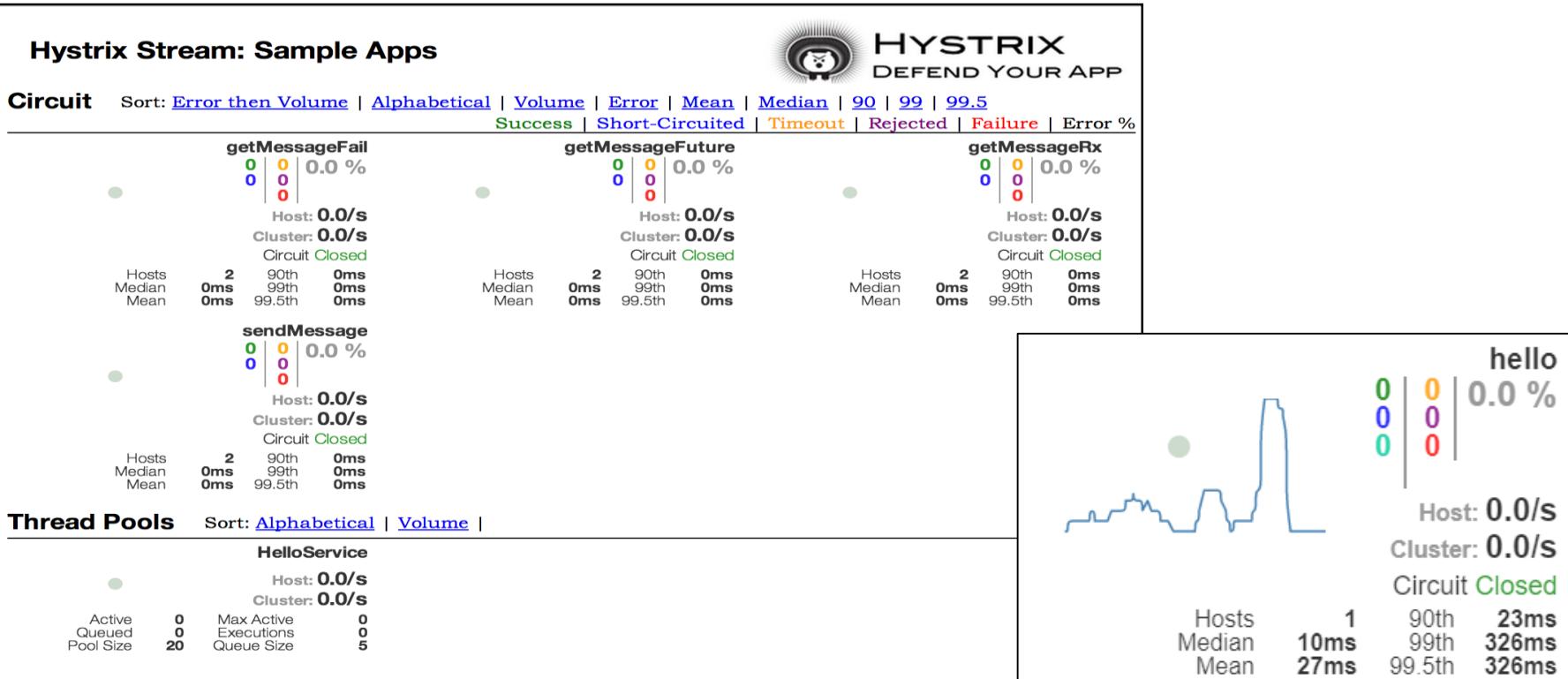
- Logging to Hystrix Dashboard on running service
  - On Gateway Server : <http://localhost:8080/hystrix>

The screenshot shows a web browser window displaying the Hystrix Dashboard. The address bar at the top contains the URL [localhost:8080/hystrix](http://localhost:8080/hystrix). The main content area features a large, stylized logo of a white bear with a shocked expression, set against a black sunburst background. Below the logo, the text "Hystrix Dashboard" is centered. A blue rectangular input field contains the URL <http://hostname:port/turbine/turbine.stream>. Below this field, there are three lines of text providing alternative URLs:  
*Cluster via Turbine (default cluster):* <http://turbine-hostname:port/turbine.stream>  
*Cluster via Turbine (custom cluster):* [http://turbine-hostname:port/turbine.stream?cluster=\[clusterName\]](http://turbine-hostname:port/turbine.stream?cluster=[clusterName])  
*Single Hystrix App:* <http://hystrix-app:port/hystrix.stream>

Delay:  ms    Title:

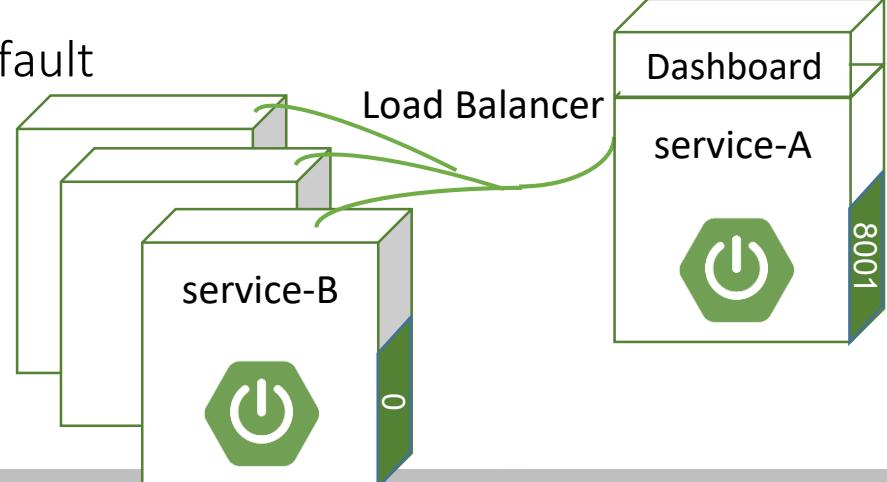
# Monitoring – Hystrix Dashboard

- In order to monitor all Microservice endpoints specify its location and press ‘Start Monitor’
  - Location ends with /hystrix.stream
  - <http://localhost:8080/service-A/hystrix.stream> (Gateway URL)



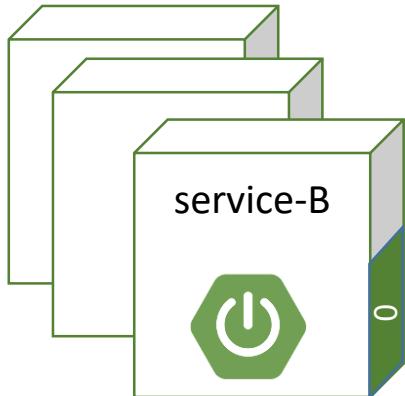
# Client Side Load Balancing

- Spring Cloud Ribbon
  - Ribbon maintains load-balancing for domain-intra-communication
    - Tracks living instances & ignores failed instances
    - Maintains valid available server list
    - Can be fully configured both programmatically & via configuration files
    - Can be easily wrap any RestTemplate activity done from one Microservice to another
    - Round-robin is used by default



# Client Side Load Balancing

- For load-balancing multiple service-B instances should be running
  - Random ports by assigning zero value allows multiple instances on the same host
  - Problem is that Eureka uses <application-name>:<port> as default instance ID
    - These values are the same for every service-B instance: ‘service-B:0’
  - In order to provide each instance a unique name – edit service-B.yml on GIT:

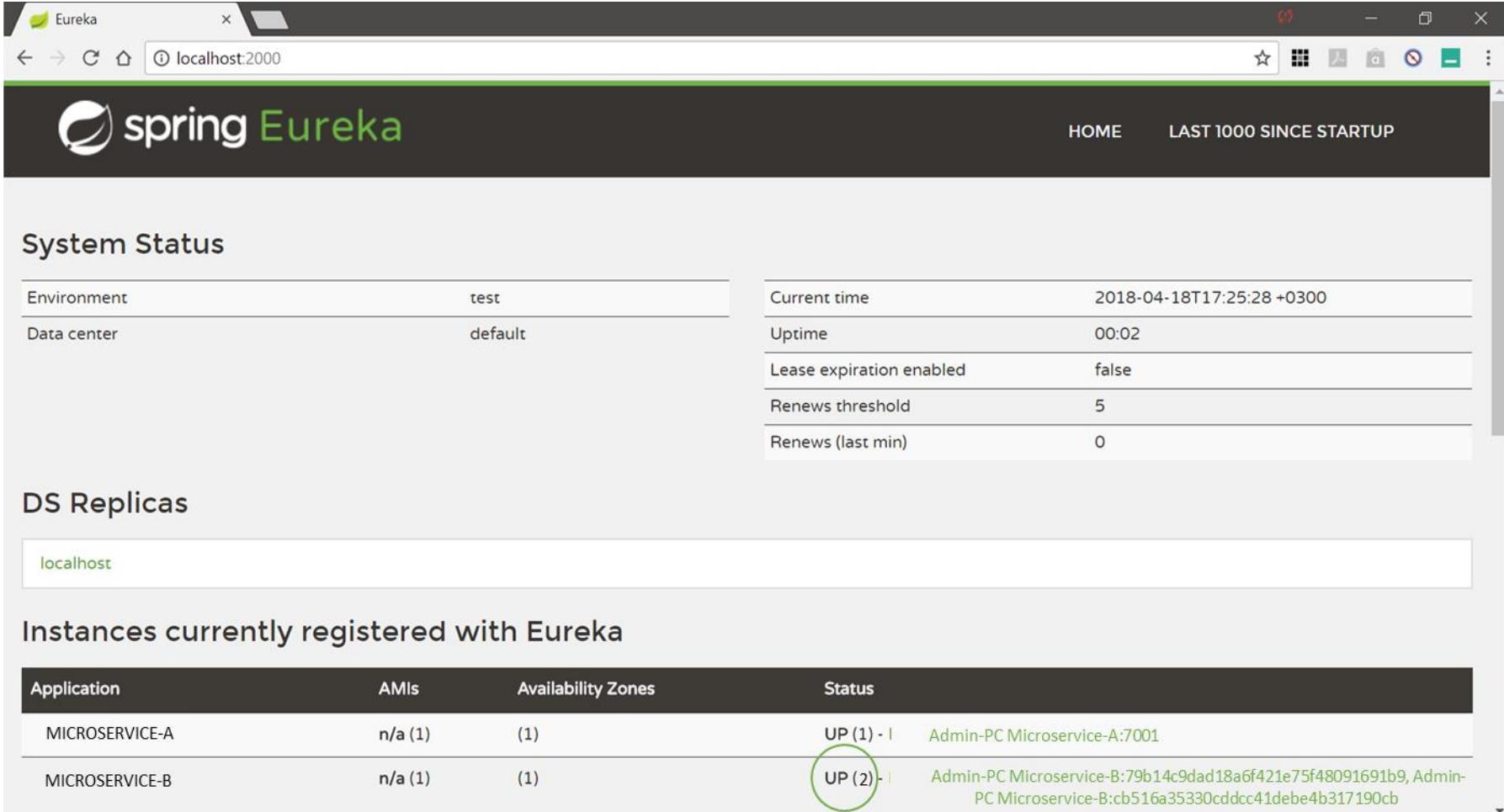


```
service-B.yml

server:
  port: ....
eureka:
  instance:
    instanceId: ${spring.application.name}:${spring.application.instance_id:${random.value}}
  client:
    serviceUrl:
      defaultZone: http://localhost:2000/eureka
```

# Client Side Load Balancing

- Monitoring instances in Eureka:



The screenshot shows the Spring Eureka dashboard running on localhost:2000. The top navigation bar includes links for HOME and LAST 1000 SINCE STARTUP. The main content area is divided into sections: System Status, DS Replicas, and Instances currently registered with Eureka.

**System Status**

Environment	test	Current time	2018-04-18T17:25:28 +0300
Data center	default	Uptime	00:02
		Lease expiration enabled	false
		Renews threshold	5
		Renews (last min)	0

**DS Replicas**

localhost
-----------

**Instances currently registered with Eureka**

Application	AMIs	Availability Zones	Status	Details
MICROSERVICE-A	n/a (1)	(1)	UP (1) - I	Admin-PC Microservice-A:7001
MICROSERVICE-B	n/a (1)	(1)	UP (2) - I	Admin-PC Microservice-B:79b14c9dad18a6f421e75f48091691b9, Admin-PC Microservice-B:cb516a35330cddcc41debe4b317190cb



# Client Side Load Balancing

- Spring Cloud Ribbon Load Balancer & Hystrix Circuit Breaker
  - Ribbon is for load balancing
  - Hystrix is for circuit breaker
- Gateway & client load balancing uses ribbon
- Any load balanced call may use circuit breaker as well
- How this combination behaves?

# Client Side Load Balancing

- We can already enjoy load-balancing by denoting `restTemplate` as `@LoadBalanced`
- A much cleaner & rapid way is to use Feign Clients which load-balance natively
  - `@FeignClient` – declarative rest client generation
    - Generates `RestTemplate` based implementation
    - Wraps `RestTemplate` with client load-balancing ribbon
    - Uses URL or logical (proxy) names as base client URL
  - Feign Clients can be reused by different Microservices
  - Note: Zuul already uses ribbon. Add `@HystrixCommand` in addition to ribbon-load-balancer for adding Circuit-breaking capabilities

# Client Side Load Balancing

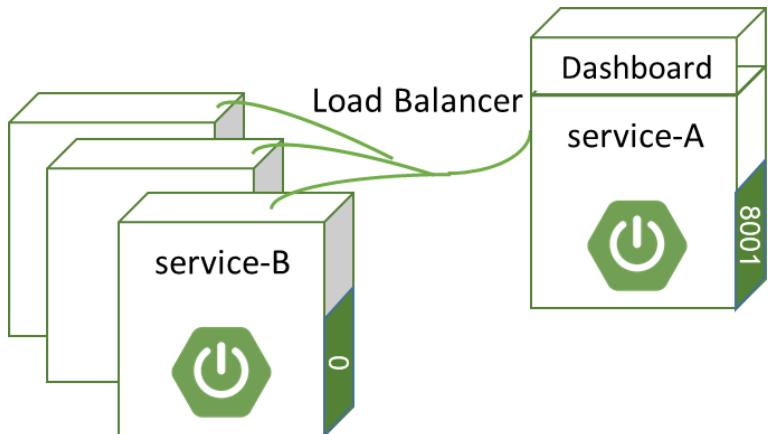
- Client Microservice Ribbon  
Feign Starter Maven  
dependencies:

pom.xml

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

# Client Side Load Balancing

- Spring Cloud Ribbon



```
@FeignClient(name="service-B")
public interface ServiceBFeignClient {
    @GetMapping("/some/url")
    String method();
}
```

```
@RestController
public class ServiceAController {
    @Autowired
    private ServiceBFeignClient client;

    @GetMapping("/call/service")
    public String callServiceB(){
        return client.method();
    }
}
```

```
@EnableCircuitBreaker
@EnableDiscoveryClient
@SpringBootApplication
@EnableFeignClients
public class MicroserviceA{ ... }
```

- When calling `http://service-A/call/service`, microservice-B calls (`http://service-B/some/uri`) are load-balanced

# Client Side Load Balancing

- Ribbon can be totally configured by creating Ribbon-Configurations
- What can be set?

Bean Type	Bean Name	Class Name
IClientConfig	ribbonClientConfig	DefaultClientConfigImpl
IRule	ribbonRule	ZoneAvoidanceRule
IPing	ribbonPing	DummyPing
ServerList<Server>	ribbonServerList	ConfigurationBasedServerList
ServerListFilter<Server>	ribbonServerListFilter	ZonePreferenceServerListFilter
ILoadBalancer	ribbonLoadBalancer	ZoneAwareLoadBalancer
ServerListUpdater	ribbonServerListUpdater	PollingServerListUpdater

# Client Side Load Balancing

- Creating Ribbon Configuration:

```
@Configuration
protected static class FooConfiguration {

    @Bean
    public ZonePreferenceServerListFilter serverListFilter() {
        ZonePreferenceServerListFilter filter = new ZonePreferenceServerListFilter();
        filter.setZone("....");
        return filter;
    }

    @Bean
    public IPing ribbonPing() {
        return new CustomPingUrl();
    }
}
```

# Spring Cloud Microservice Ecosystem

