

Test-Driven Development (TDD) Spring Framework





Somkiat Puisungnoen

Search

Somkiat | Home

Update Info 1 View Activity Log 10+ ...

Timeline About Friends 3,138 Photos More

When did you work at Opendream? X

... 22 Pending Items

Post Photo/Video Live Video Life Event

What's on your mind?

Public Post

Intro

Software Craftsmanship

Software Practitioner at สยามชานนาญกิจ พ.ศ. 2556

Agile Practitioner and Technical at SPRINT3r

Somkiat Puisungnoen 15 mins · Bangkok · ...

Java and Bigdata

When did you work at Opendream? X

... 22 Pending Items

Post Photo/Video Live Video Life Event

What's on your mind?

Public Post

Intro

Software Craftsmanship

Software Practitioner at สยามชานนาญกิจ พ.ศ. 2556

Agile Practitioner and Technical at SPRINT3r

Somkiat Puisungnoen 15 mins · Bangkok · ...

Java and Bigdata



somkiat.cc

Page Messages Notifications 3 Insights Publishing Tools Settings Help ▾

Help people take action on this Page. X

+ Add a Button

Home Posts Videos Photos

Liked Following Share ...

somkiat.cc
@somkiat.cc

Home Posts Videos Photos



Agenda



Day 1 (Spring Framework)

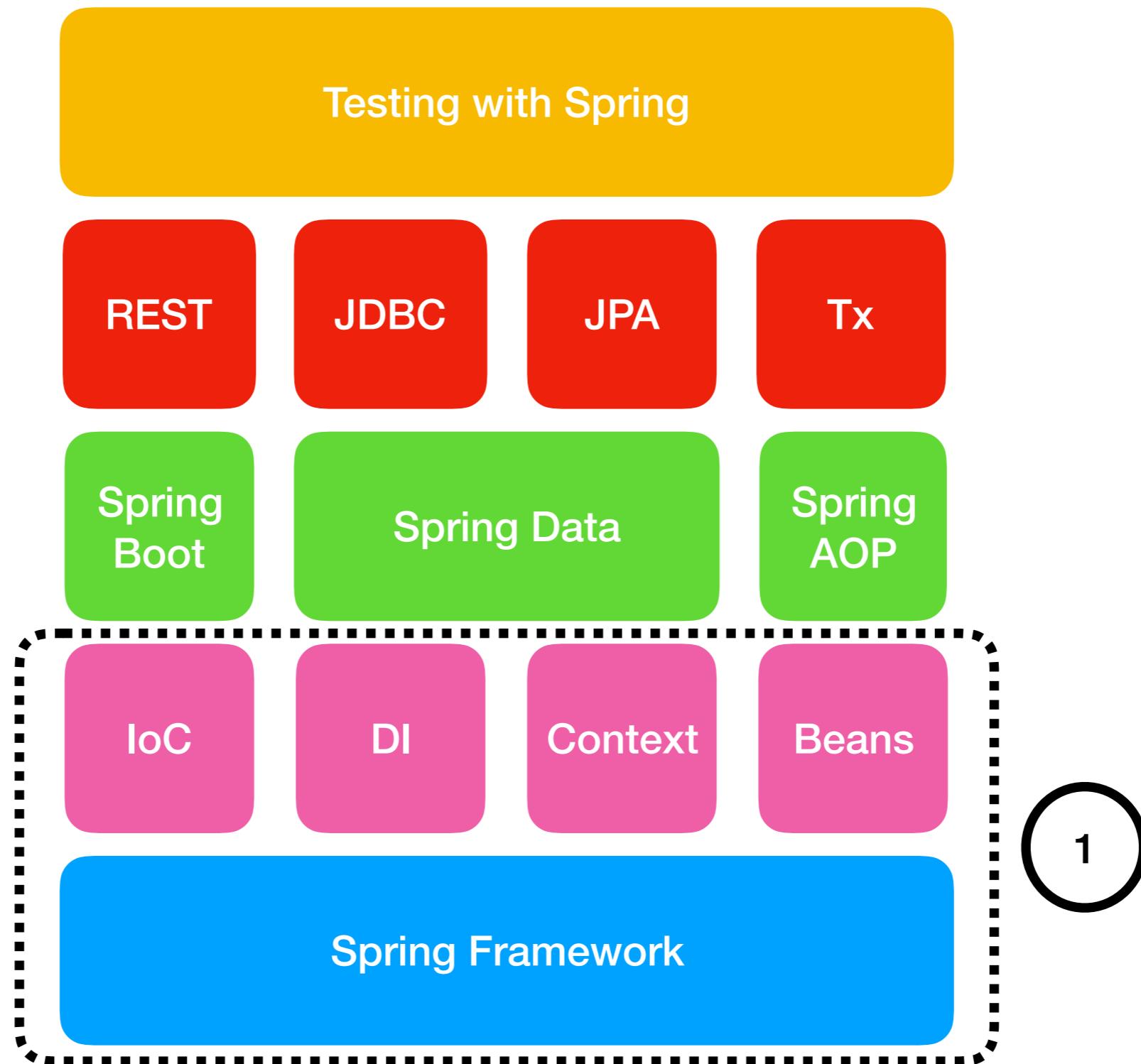


Agenda

Introduction to Spring framework
Dependency Injection (DI)
Inversion of Control (IoC)
Spring Context and Bean
How to build project with Spring ?
Spring Configuration
Why we need Spring framework ?
Spring Modules and Project
SOLID for Object-Oriented Design



Workshop



1



Day 2 (Spring Boot)



Agenda

REST (REpresentational State Transfer)

Introduction to Spring Boot

Goals of Spring Boot

Better project structure of Spring Boot

Develop RESTful APIs

Error handling

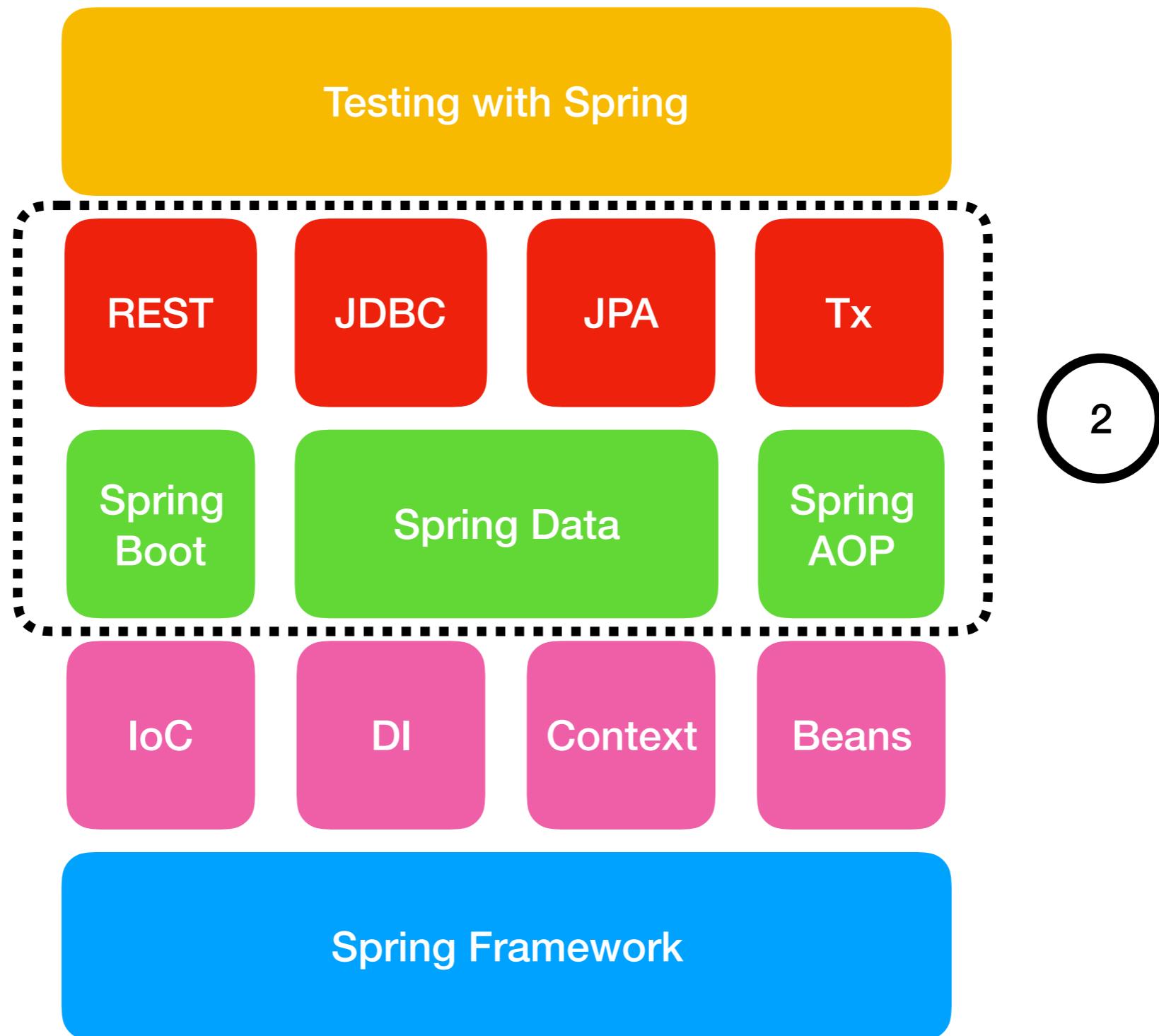
Layer in Spring Boot project

Working with Spring Data (JPA and JDBC)

Manage transaction



Workshop



Day 3 (Spring Testing, TDD)

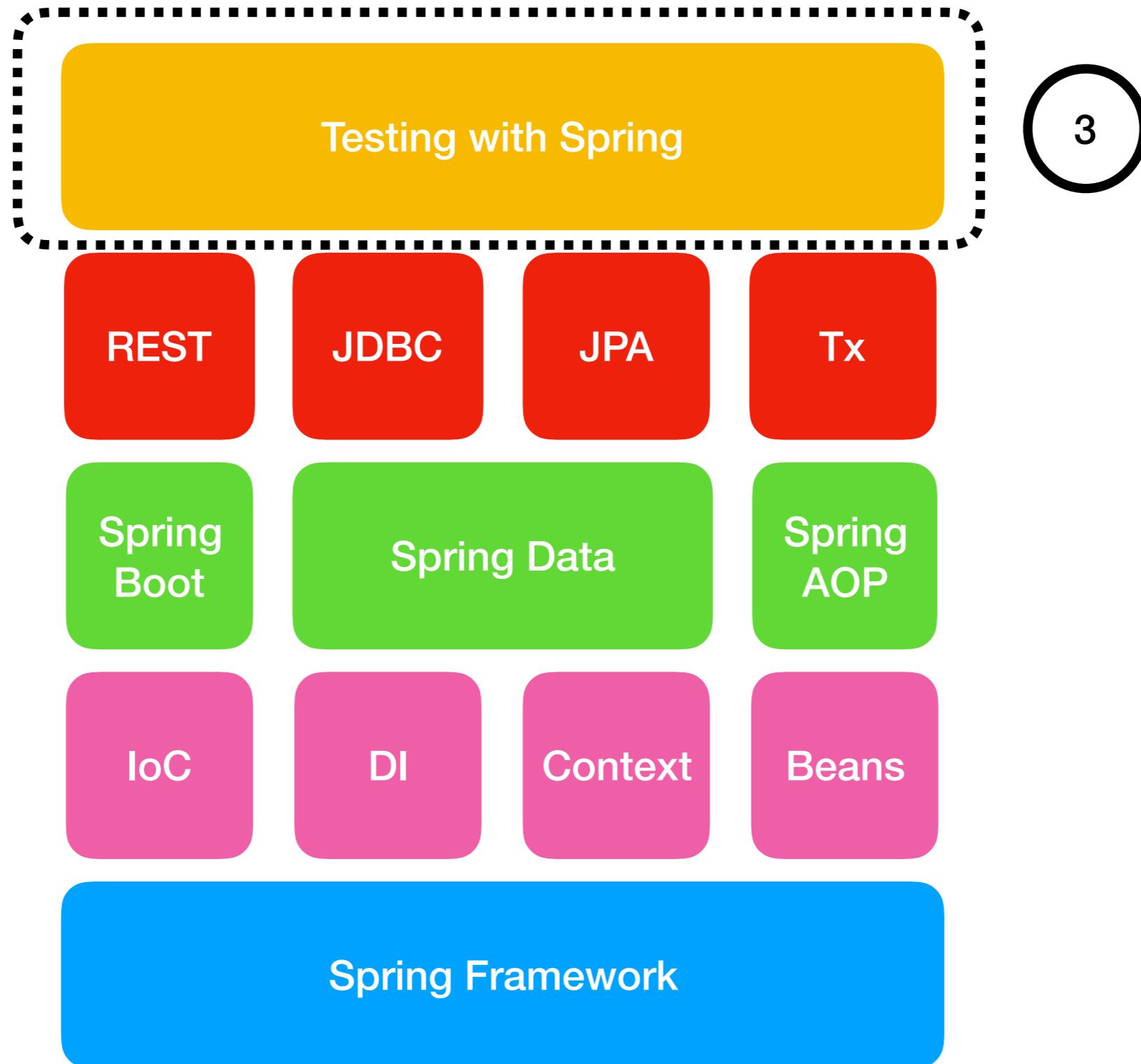


Agenda

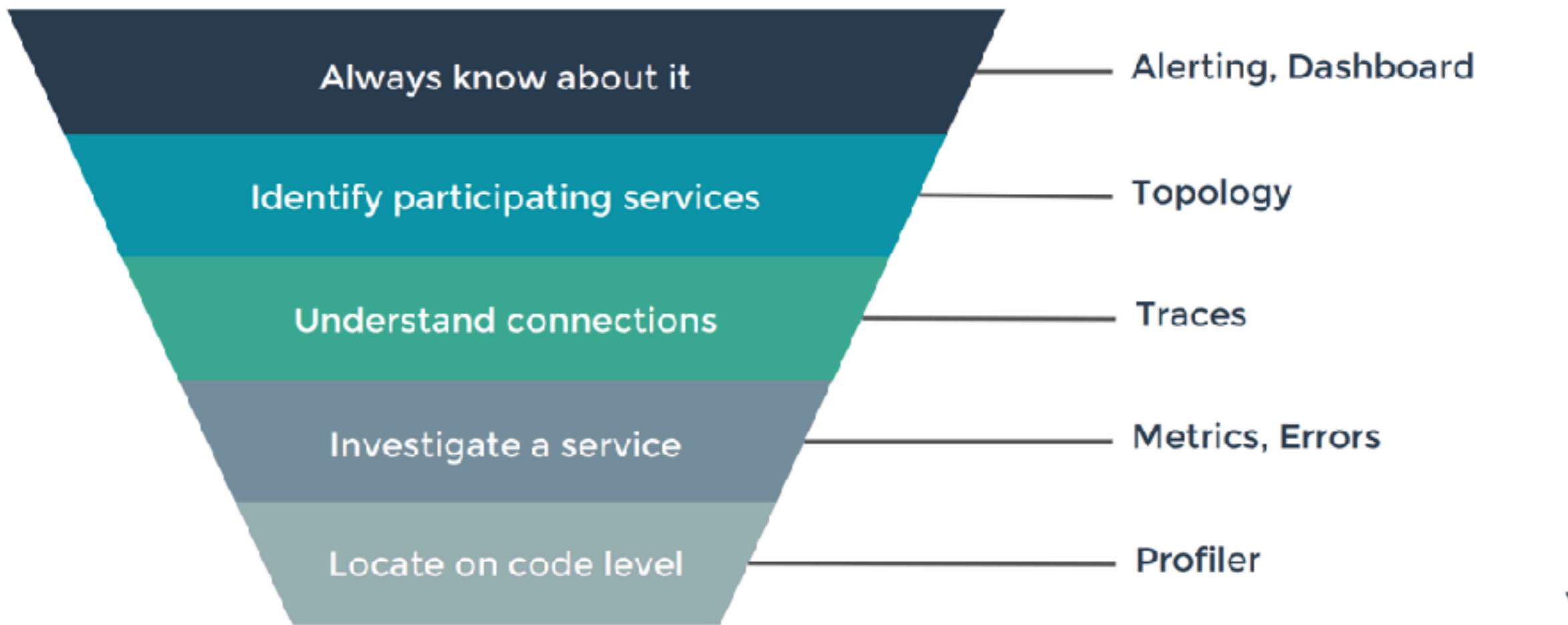
Test-Driven Development (TDD)
Testing with Spring Boot
Controller testing
Service testing
Repository testing
Test double (Stub, Spy and Mocking)
Learn JUnit and Mockito
Code coverage



Workshop



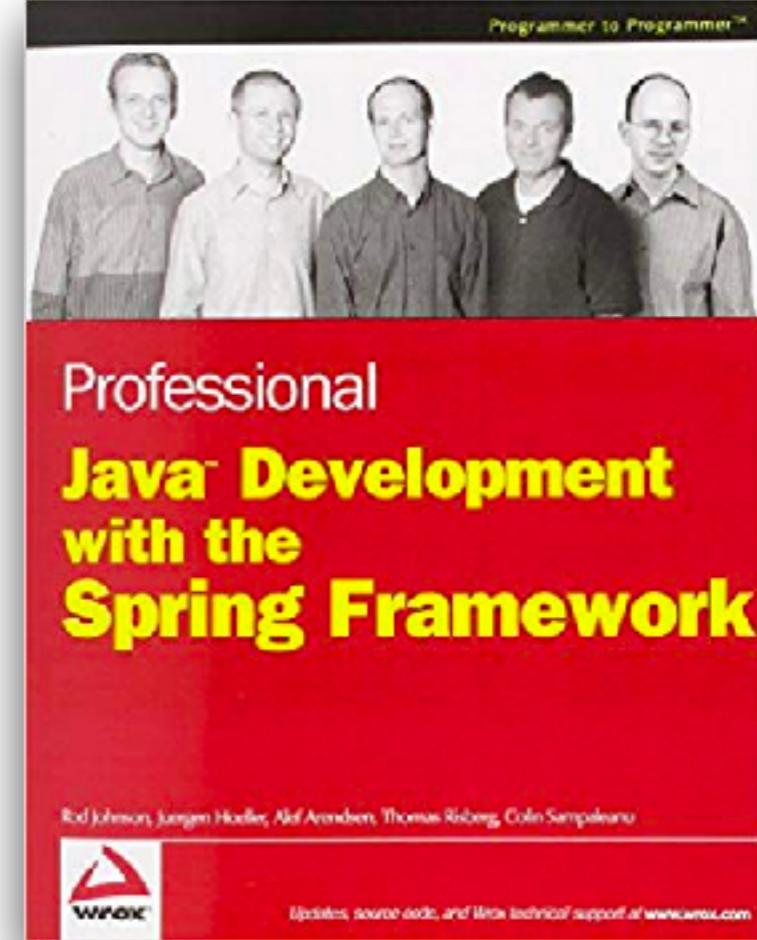
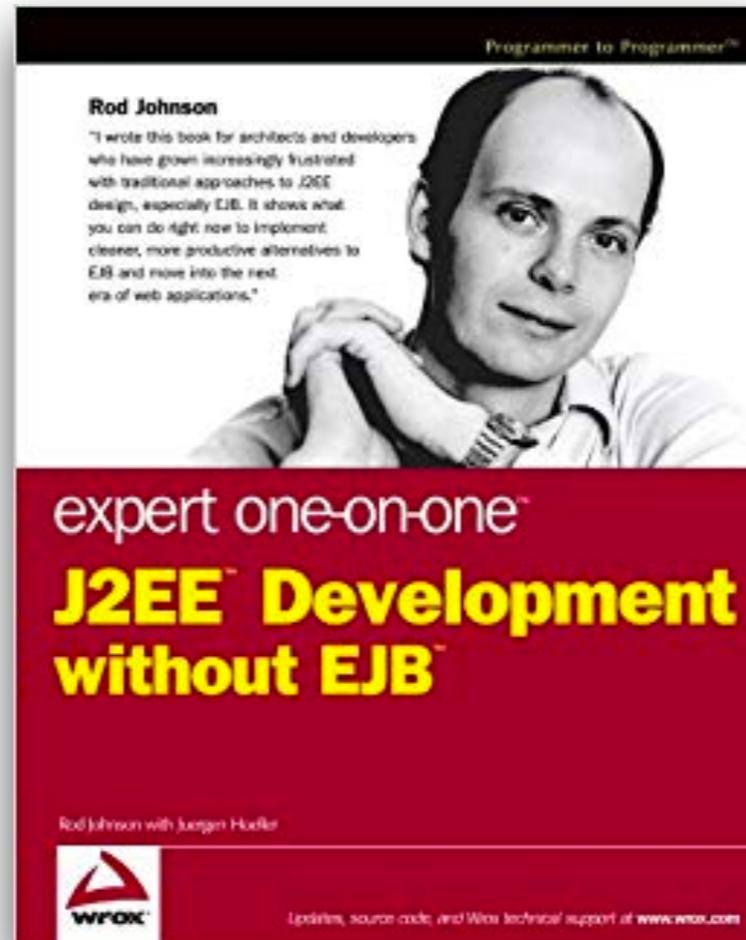
Workshop



Introduction to Spring Framework



History



Spring ?

Spring: the source for modern java



Spring Boot

BUILD ANYTHING

Spring Boot is designed to get you up and running as quickly as possible, with minimal upfront configuration of Spring. Spring Boot takes an opinionated view of building production-ready applications.

Spring Cloud

COORDINATE ANYTHING

Built directly on Spring Boot's innovative approach to enterprise Java, Spring Cloud simplifies distributed, microservice-style architecture by implementing proven patterns to bring resilience, reliability, and coordination to your microservices.

Spring Cloud Data Flow

CONNECT ANYTHING

Connect the Enterprise to the Internet of Anything—mobile devices, sensors, wearables, automobiles, and more. Spring Cloud Data Flow provides a unified service for creating composable data microservices that

<https://spring.io/>



Spring Projects

Modular by design

Spring Framework

Spring Boot

Spring Cloud Data Flow

Spring Cloud

Spring Data

Spring Integration

Spring Batch

Spring Security

Spring AMQP

Spring LDAP

Spring WebFlow

Spring REST Doc

<https://spring.io/projects>



Spring Framework



What ?

Application development framework for Java
Create high performing, testable and reusable
code without any lock-in

Open source since 2003

Reduce code and speed up development

Current version **5.x**

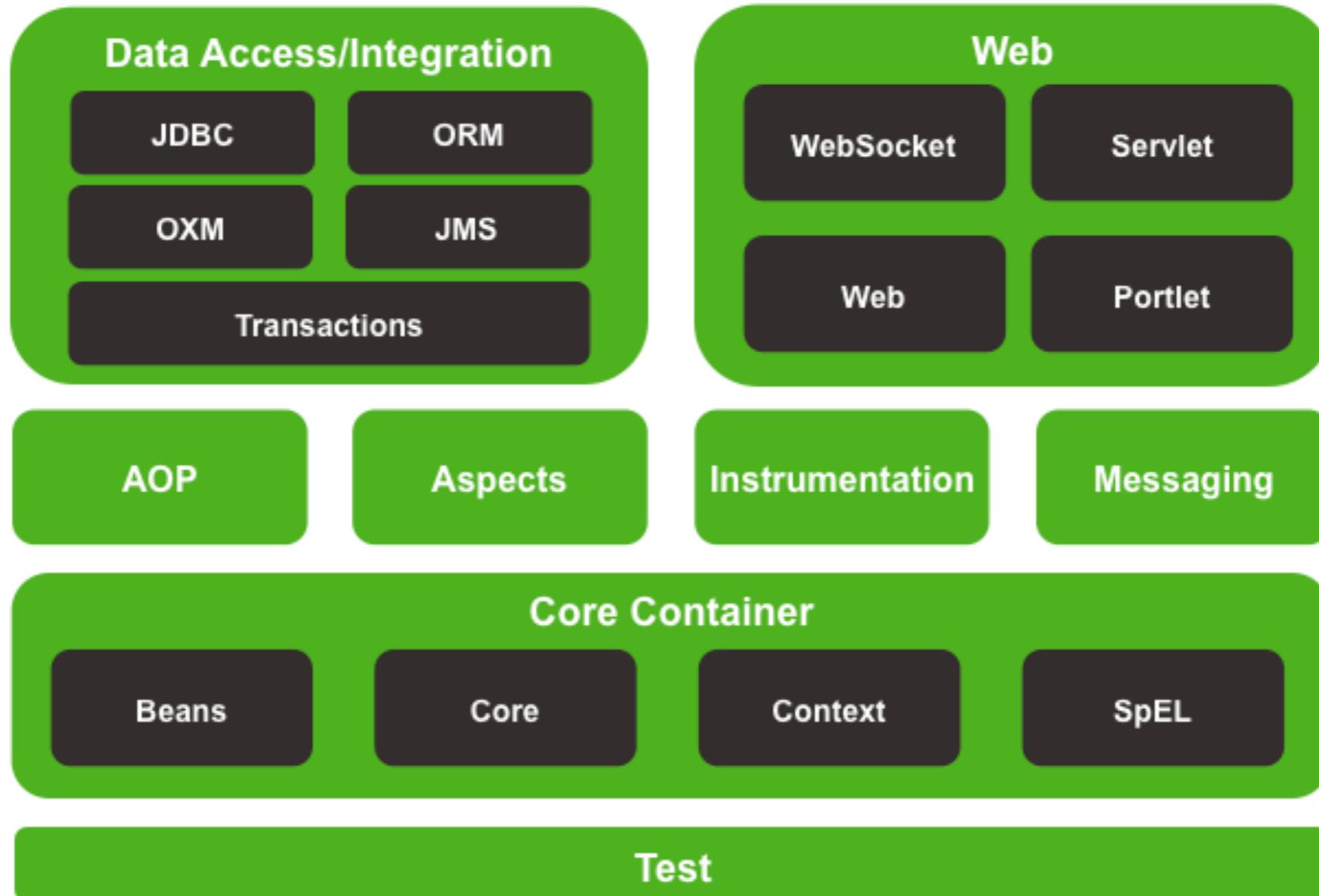
Required **JDK 8+**



Overview of Spring Framework



Spring Framework Runtime



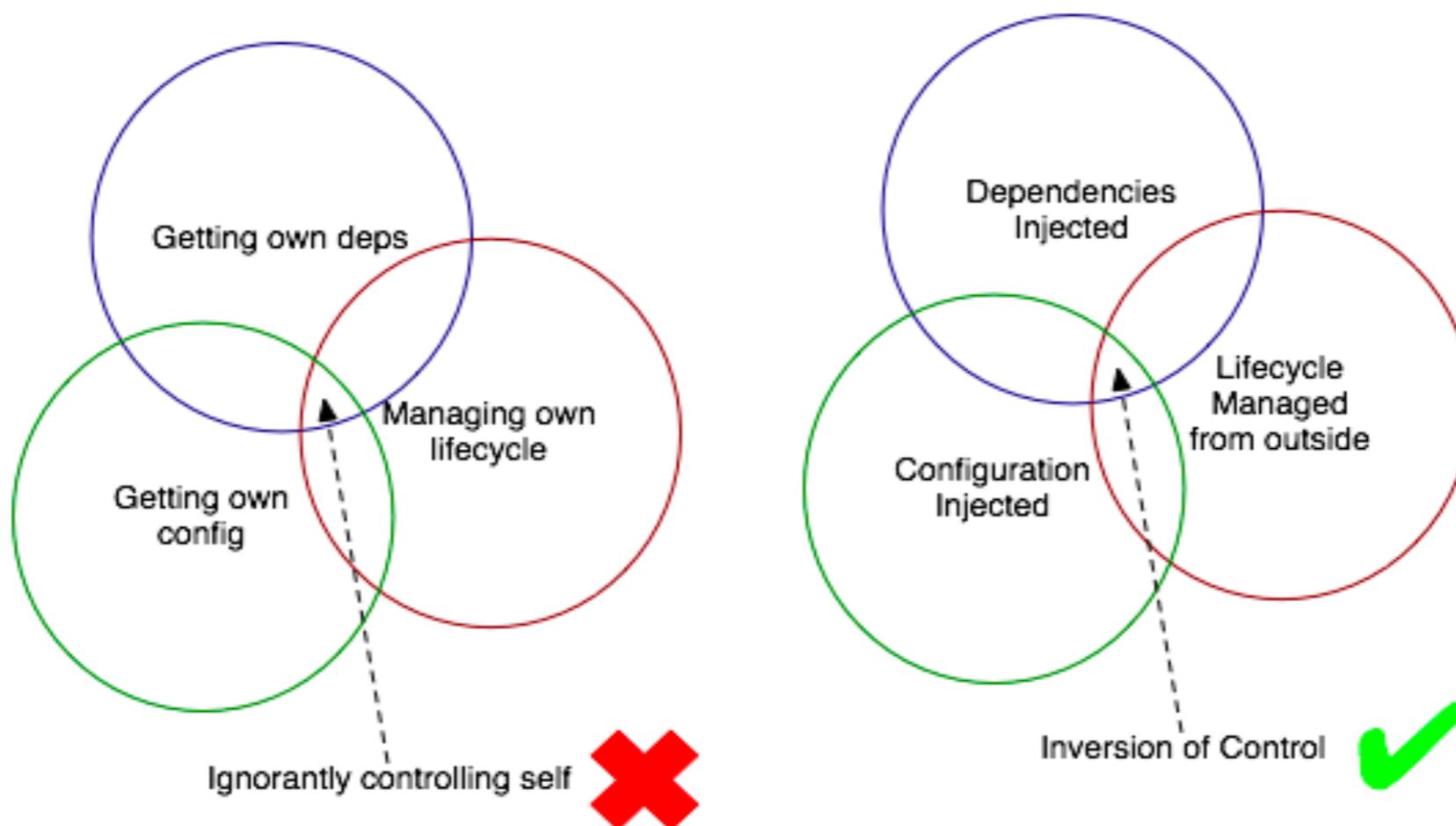
Core Container

Core and Beans
Context
Expression language



Core and Beans

Provide the fundamental parts of framework
Including IoC and Dependency Injection



<https://www.martinfowler.com/articles/injection.html>



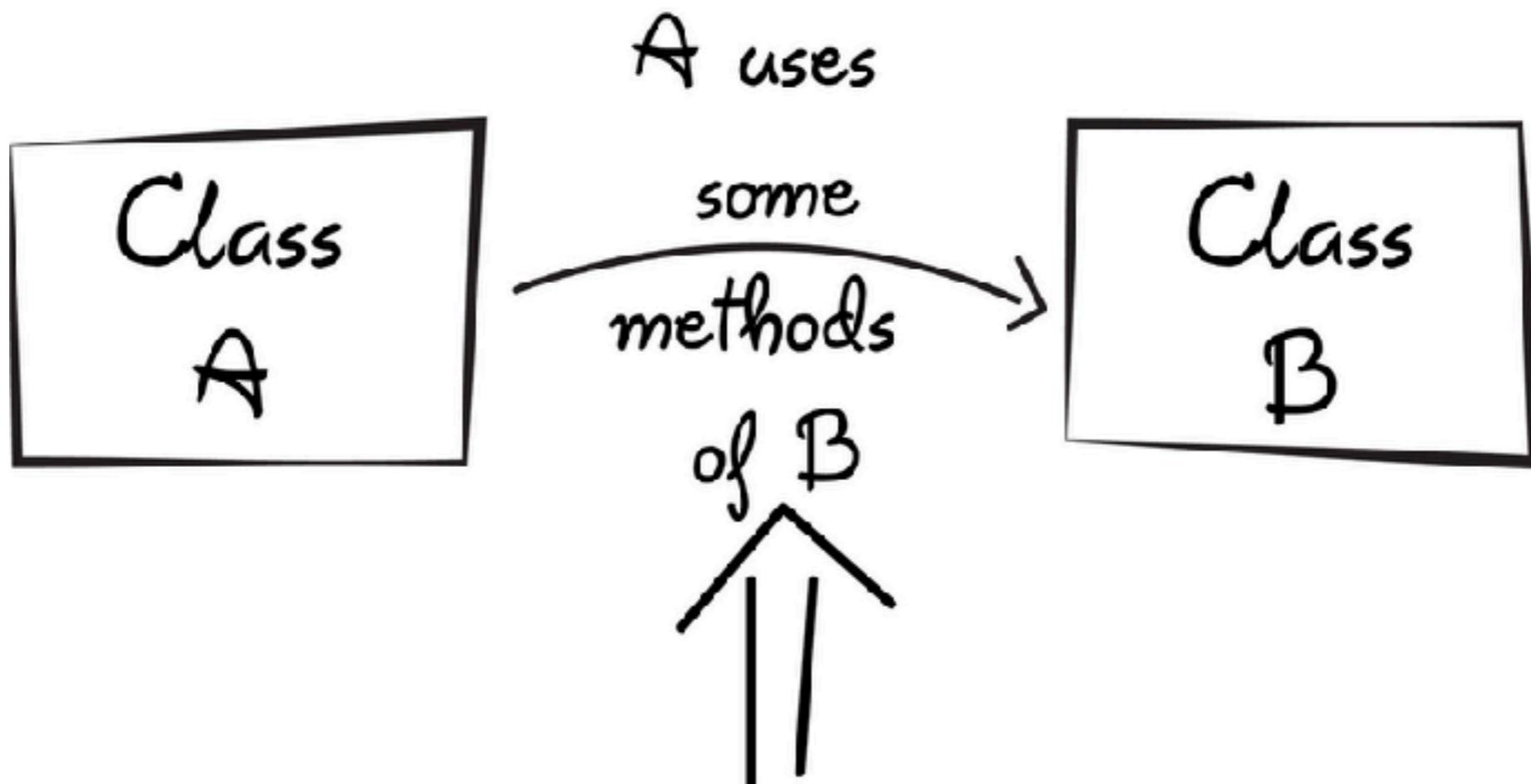
Inversion of Control (IoC)

Concept in application development

Don't call me, I will call you



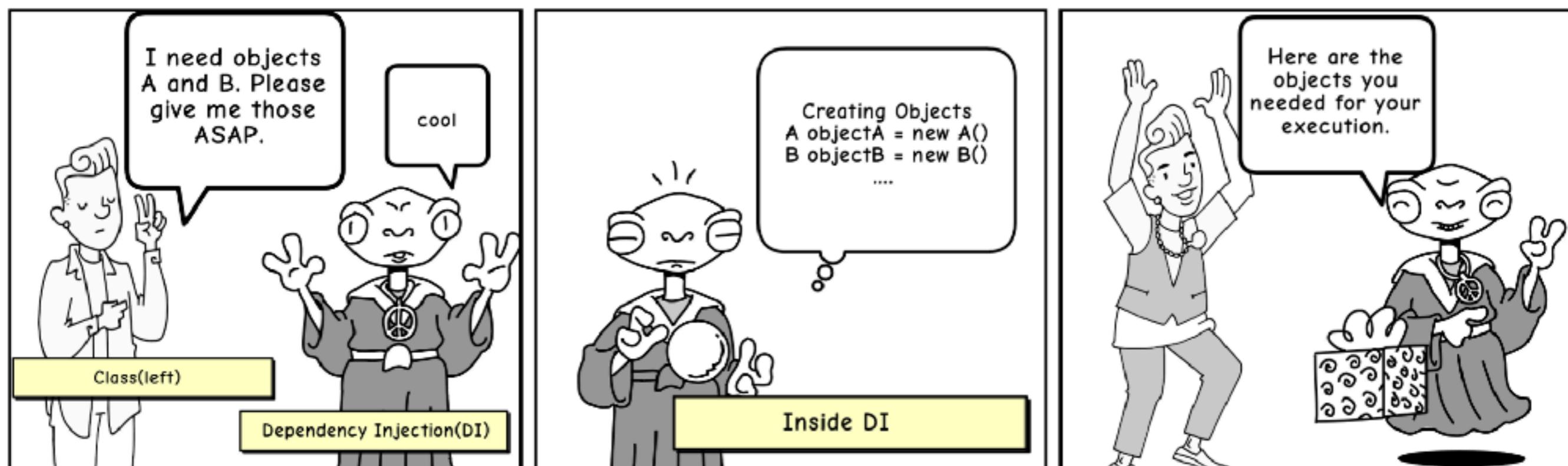
Dependency Injection



Its a dependency



Dependency Injection

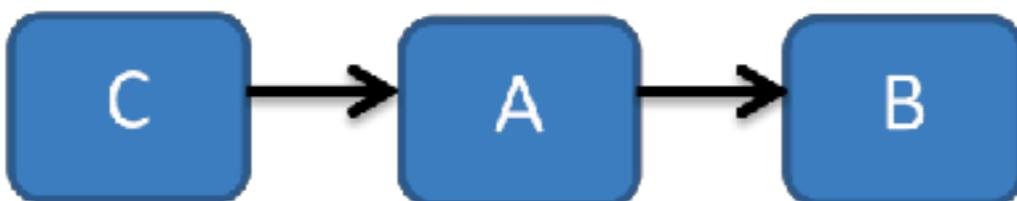


This comic was created at www.MakeBeliefsComix.com. Go there and make one now!

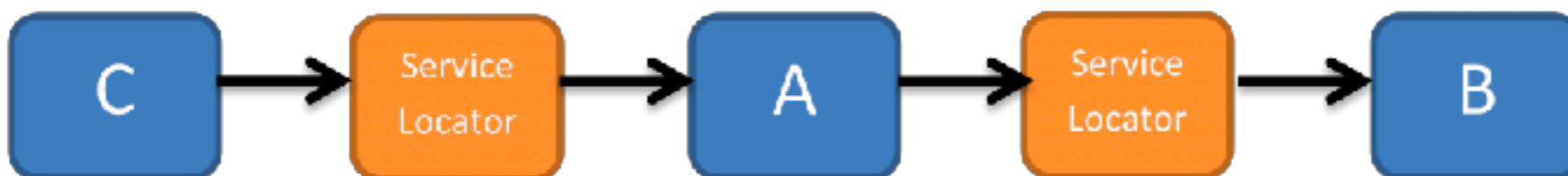


Dependency Injection

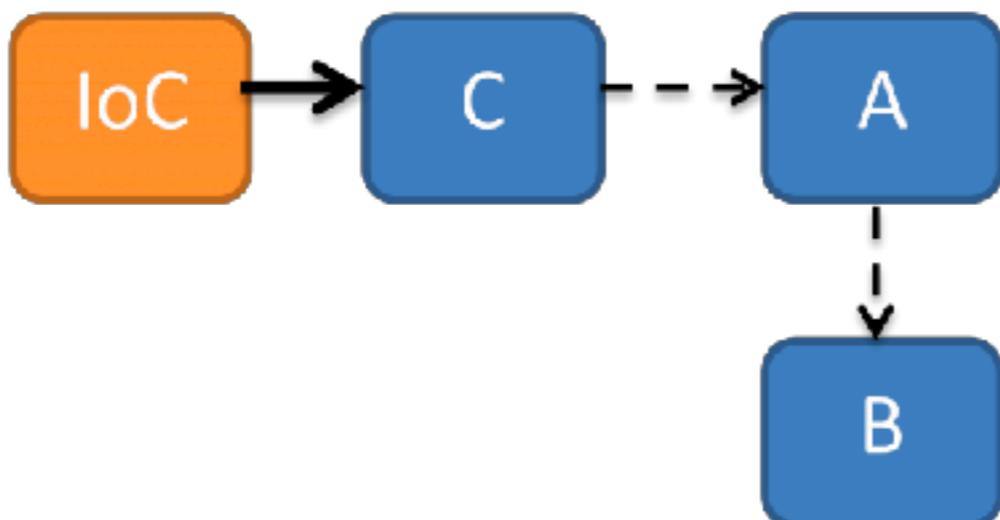
Class Dependencies



Service Location / Active Calling



IoC / DI / Auto-Wiring / Passive Calling



Types of Dependency Injection

Constructor injection

Property/Setter injection

Method injection

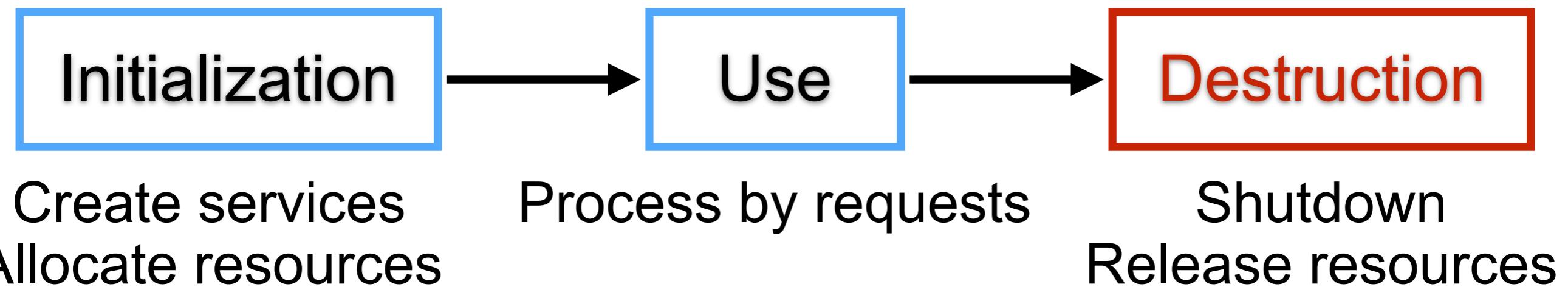
Interface injection



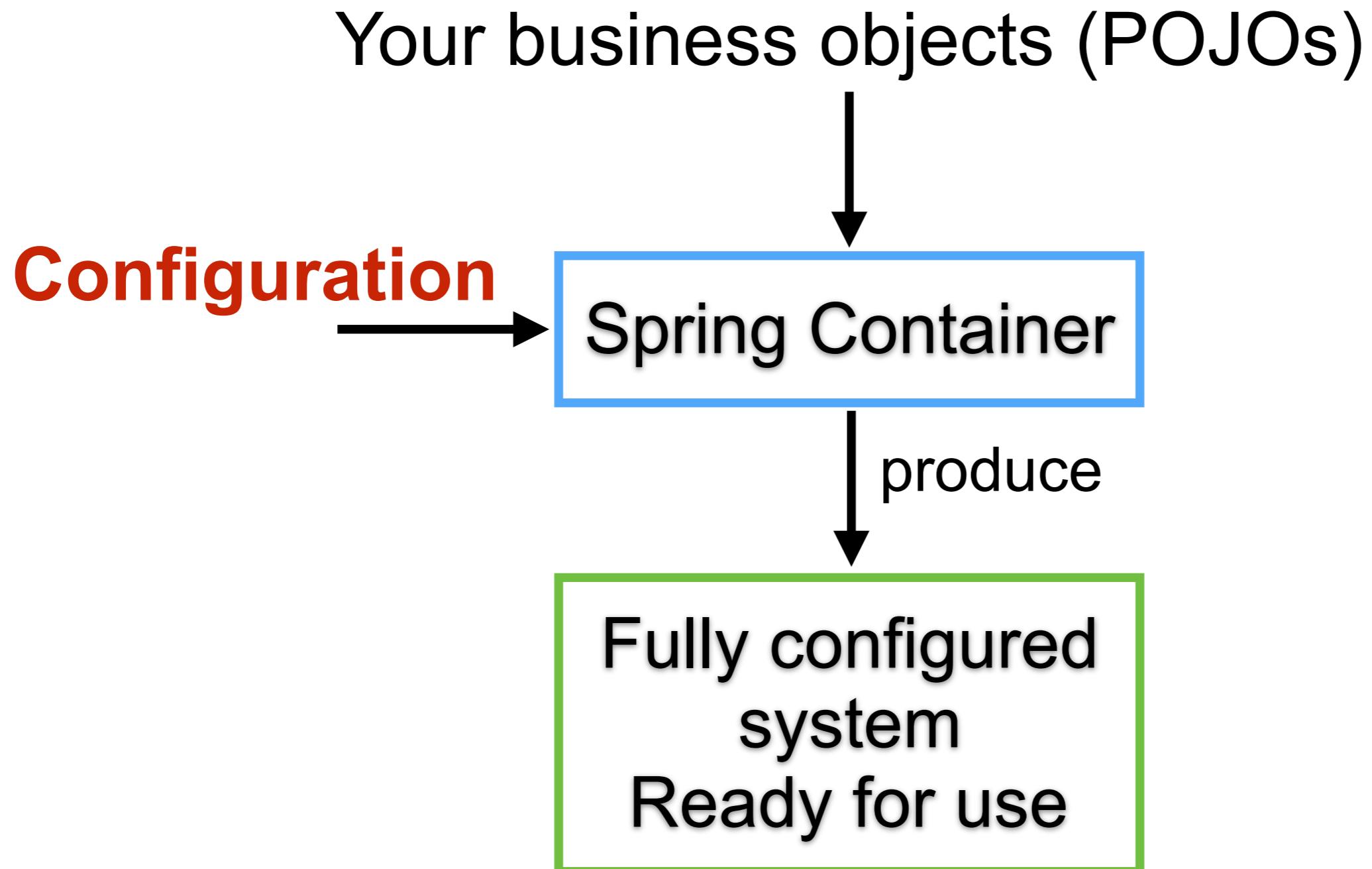
Back to Spring Framework



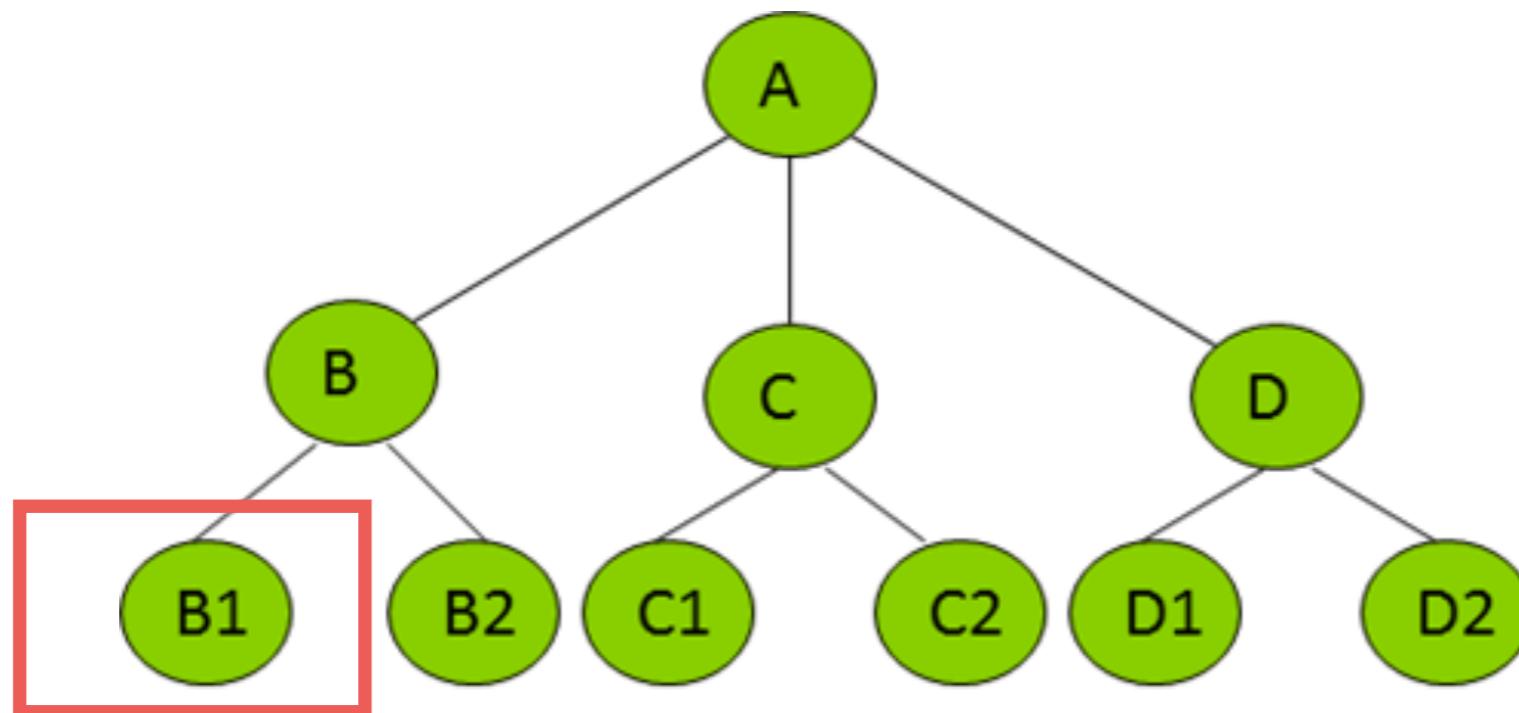
Application Lifecycle



Spring IoC container



Spring IoC container



Bean B1



Configuration

From XML file

Annotation-based configuration (2.5)

Java-based configuration (3.0)



Beans

Spring IoC container manages one or more beans
Beans are created with configuration



Bean Definitions

Package-qualified class name

Bean behavioral (scope, lifecycle, callback)

Reference to other beans

Other configuration setting to create new object



Bean Definitions

Property	Section
Class	Instantiating beans
Name	Naming beans
Scope	Beans scopes
Constructor arguments	Dependency Injection
Properties	Dependency Injection
Autowiring mode	Autowiring collaborators
Lazy initialization mode	Lazy-initialized beans

<https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-definition>



Bean Scopes

Scope	Description
singleton	Single instance for each container
prototype	Single bean definition to any number of object instances.
request	HTTP request
session	HTTP session
application	ServletContext

<https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-scopes>

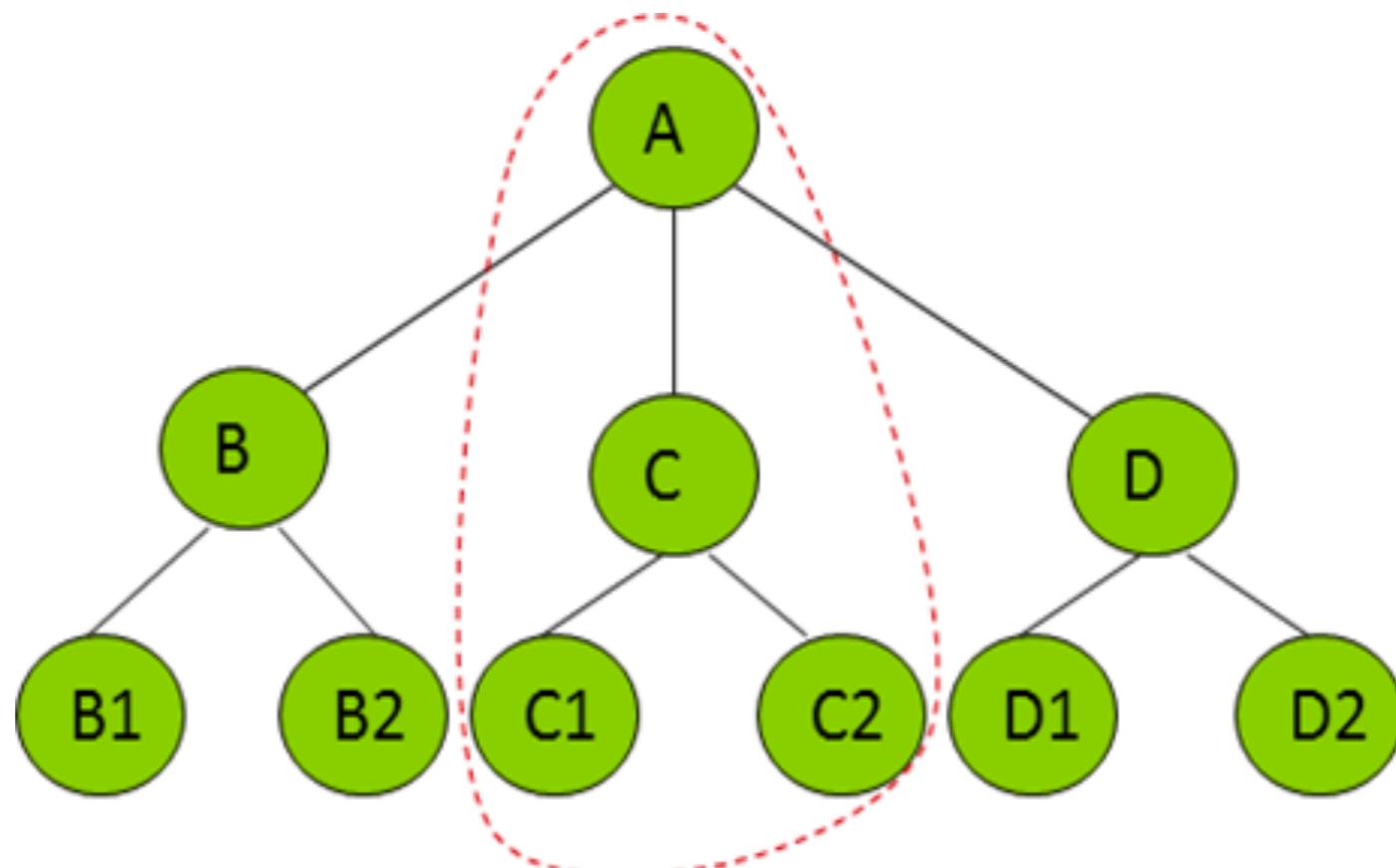


Change scope of bean

```
@Component  
@Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)  
public class RealRandom implements MyRandom {  
  
    public String number;  
  
    @Override  
    public int nextInt(int bound) {  
        return new Random().nextInt(bound);  
    }  
}
```



Lazy-load dependencies



BeanFactory ?

Interface defines basic functionality for Spring container

Factory design pattern

Load beans from configuration source

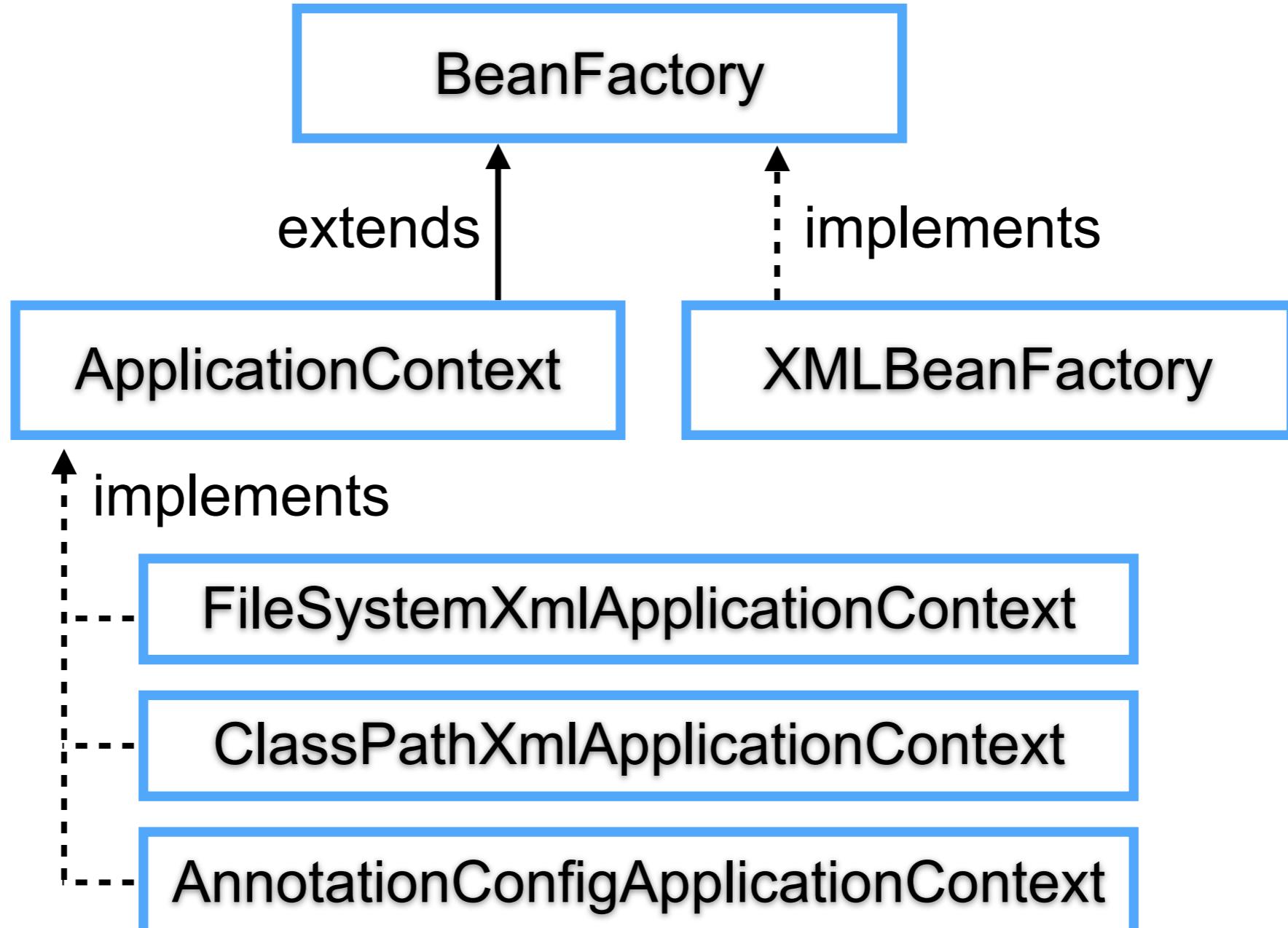
Instantiated the bean when requested

Wire dependencies and properties for beans

Manage the bean lifecycle



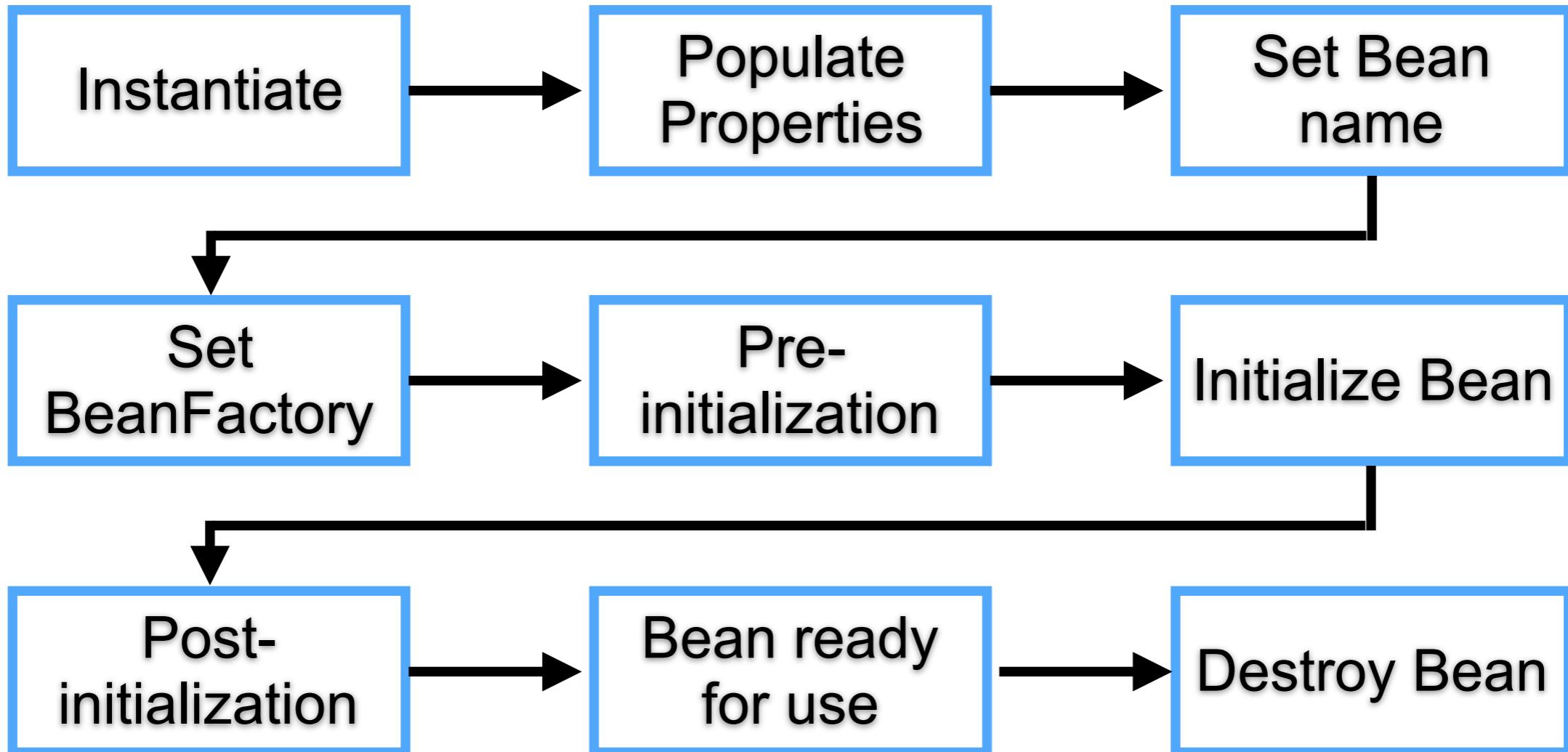
BeanFactory ?



<https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/ApplicationContext.html>



Lifecycle of BeanFactory



Let's start Spring Framework



Workshop



Create new Project

Use spring initializr

SPRING INITIALIZR bootstrap your application now

Generate a with and Spring Boot

Project Metadata

Artifact coordinates

Group

Artifact

Dependencies

Add Spring Boot Starters and dependencies to your application

Search for dependencies

Selected Dependencies

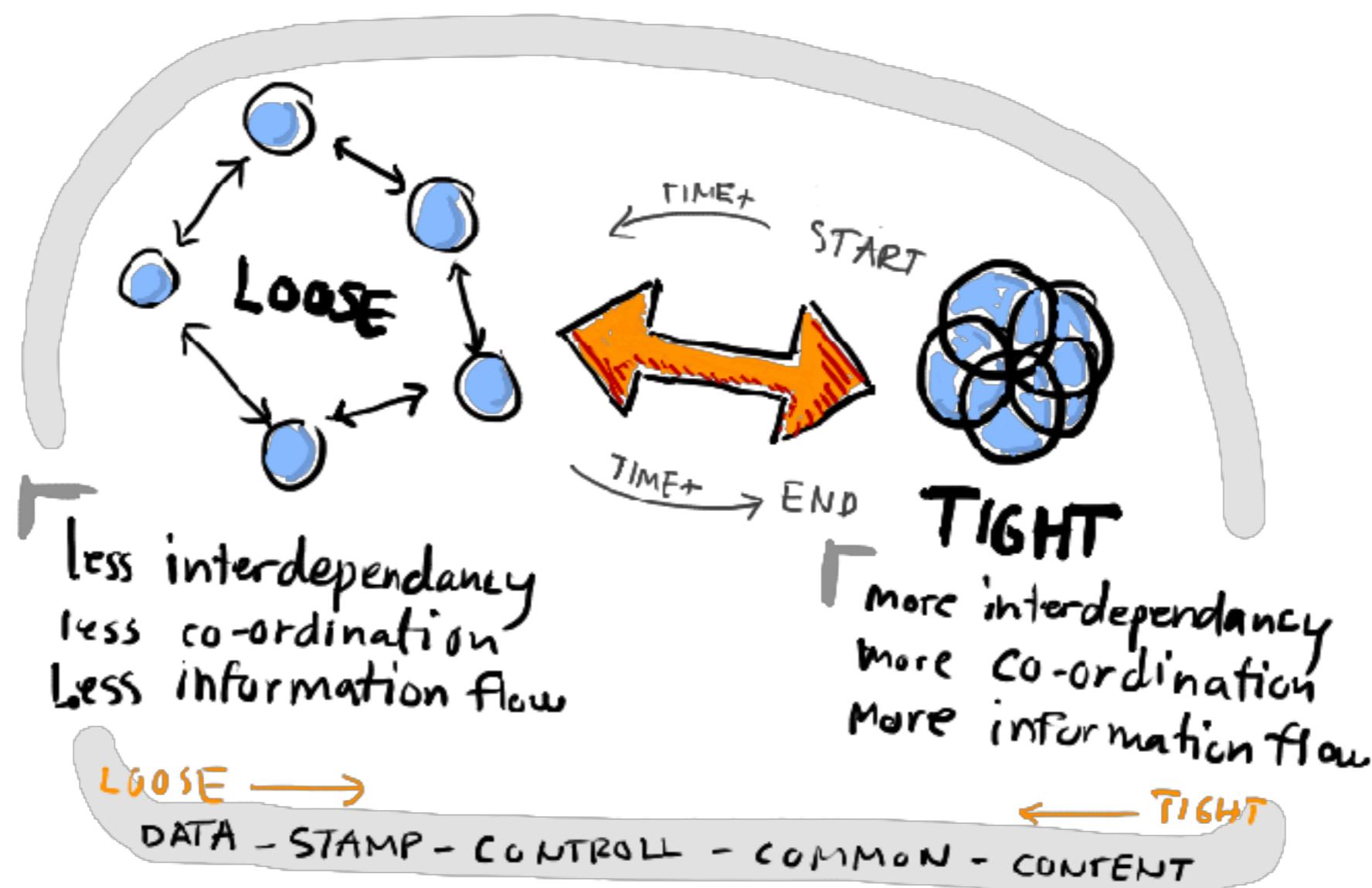
Don't know what to look for? Want more options? Switch to the full version.

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



Understanding Dependency Injection

Tight coupling
Loose coupling



Using Spring to manage dependencies

@Component

@Autowired

Constructor and Setter injection

@Primary

@Qualifier



Scope of beans

Singleton
Prototype

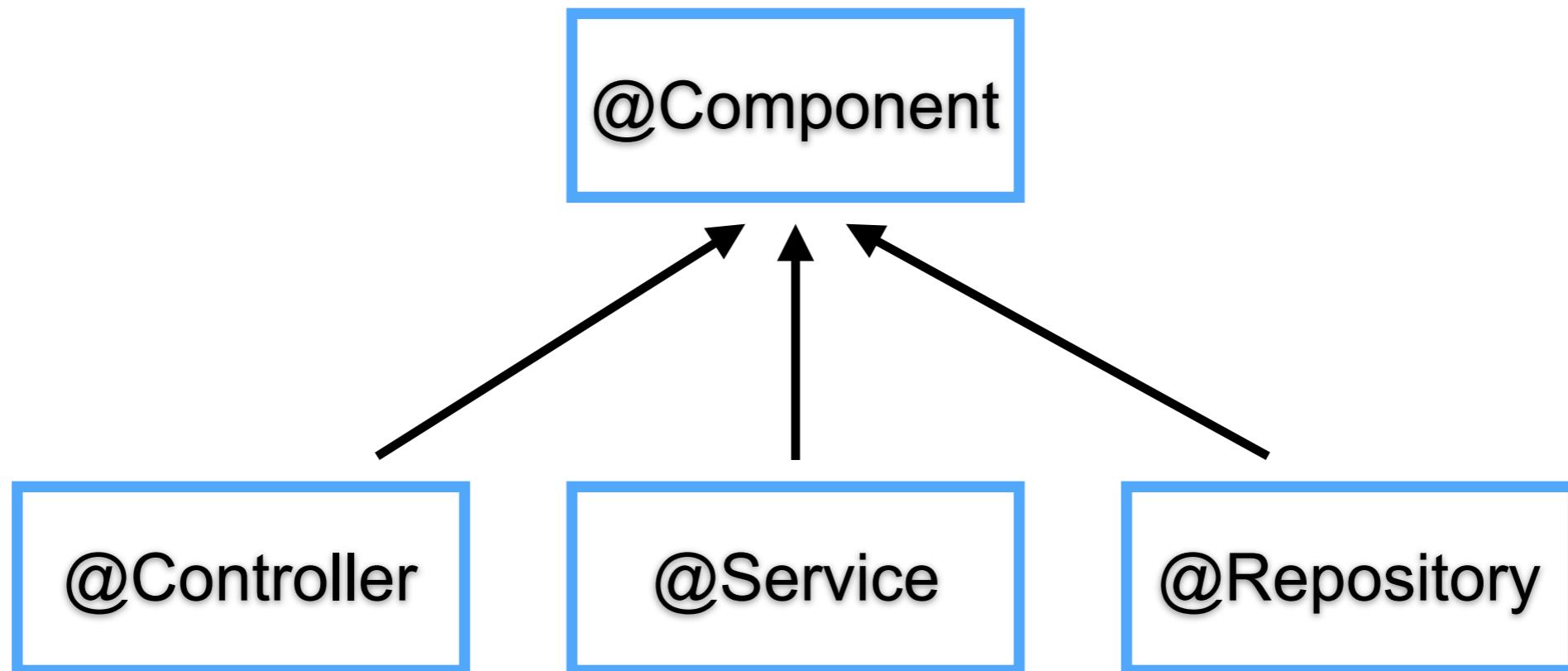


What are difference of ...

@Component
@Controller
@Service
@Repository



What are difference of ...



Layer of application



What are difference of ...

@Component

Generic stereotype for any component or bean

@Controller

Stereotype for the presentation layer (Spring MVC)

@Service

Stereotype for the service layer

@Repository

Stereotype for the persistence layer



Why Spring is popular ?

- Enable testable code
- No plumbing code
- Flexible architecture
- Staying current



Eclipse plug-in

INFINITEST

FOR INTELLIJ FOR ECLIPSE COMMUNITY CONTRIBUTE SPONSORS



infinitest

THE CONTINUOUS TEST RUNNER FOR THE JVM

You like TDD? you'll love Infinitest.
Each time you change the code Infinitest runs the relevant tests for you!!

<https://infinitest.github.io/>



Code Smell

Code Smells

- What? How can code "smell"??
- Well it doesn't have a nose... but it definitely can stink!



Bloaters

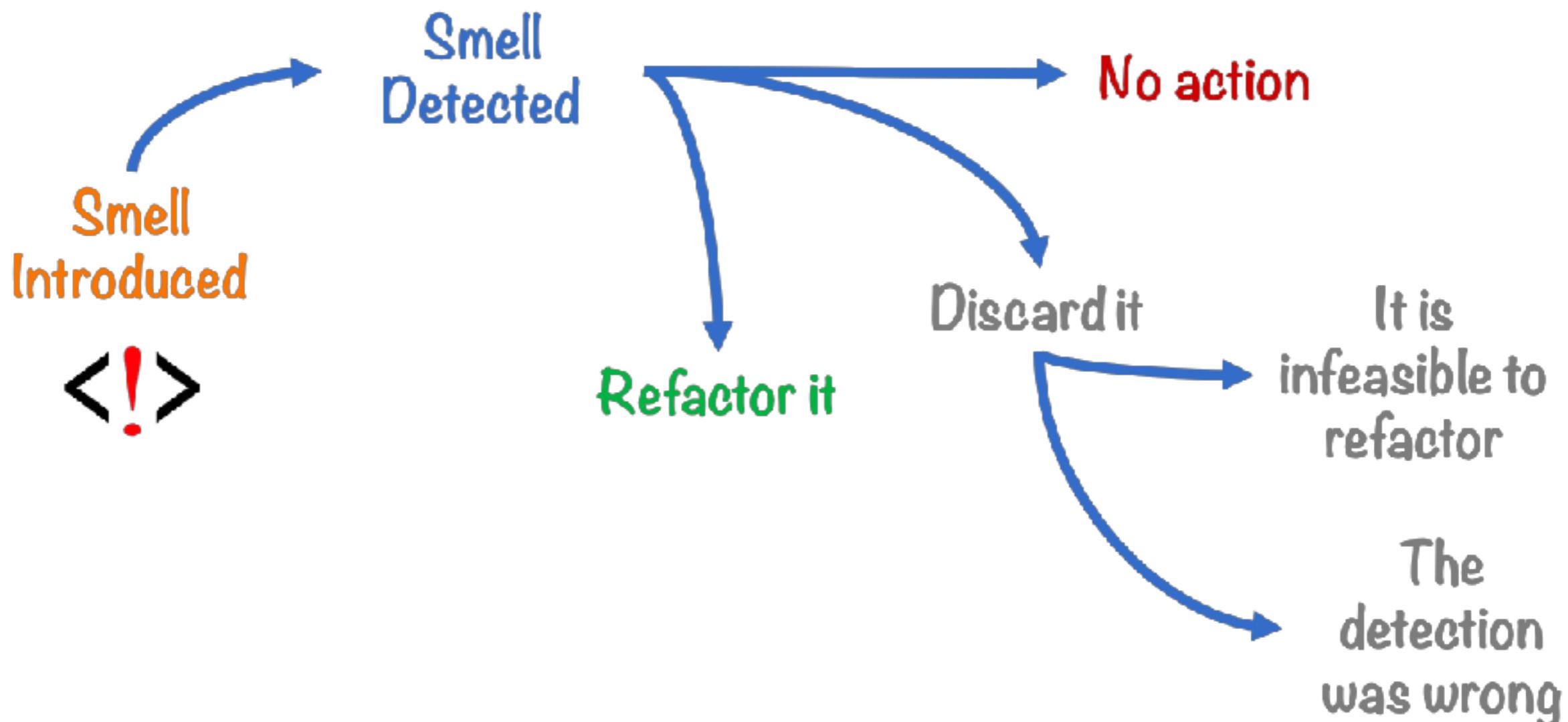
Bloaters are code, methods and classes that have increased to such gargantuan proportions that they are hard to work with. Usually these smells do not crop up right away, rather they accumulate over time as the program evolves (and especially when nobody makes an effort to eradicate them).

- Long Method
- Large Class
- Primitive Obsession
- Long Parameter List
- Data Clumps

<https://sourcemaking.com/refactoring/smells>



Code Smell



Testable structure/application



SOLID



Single Responsibility Principle

Open Closed Principle

Liskov Substitution Principle

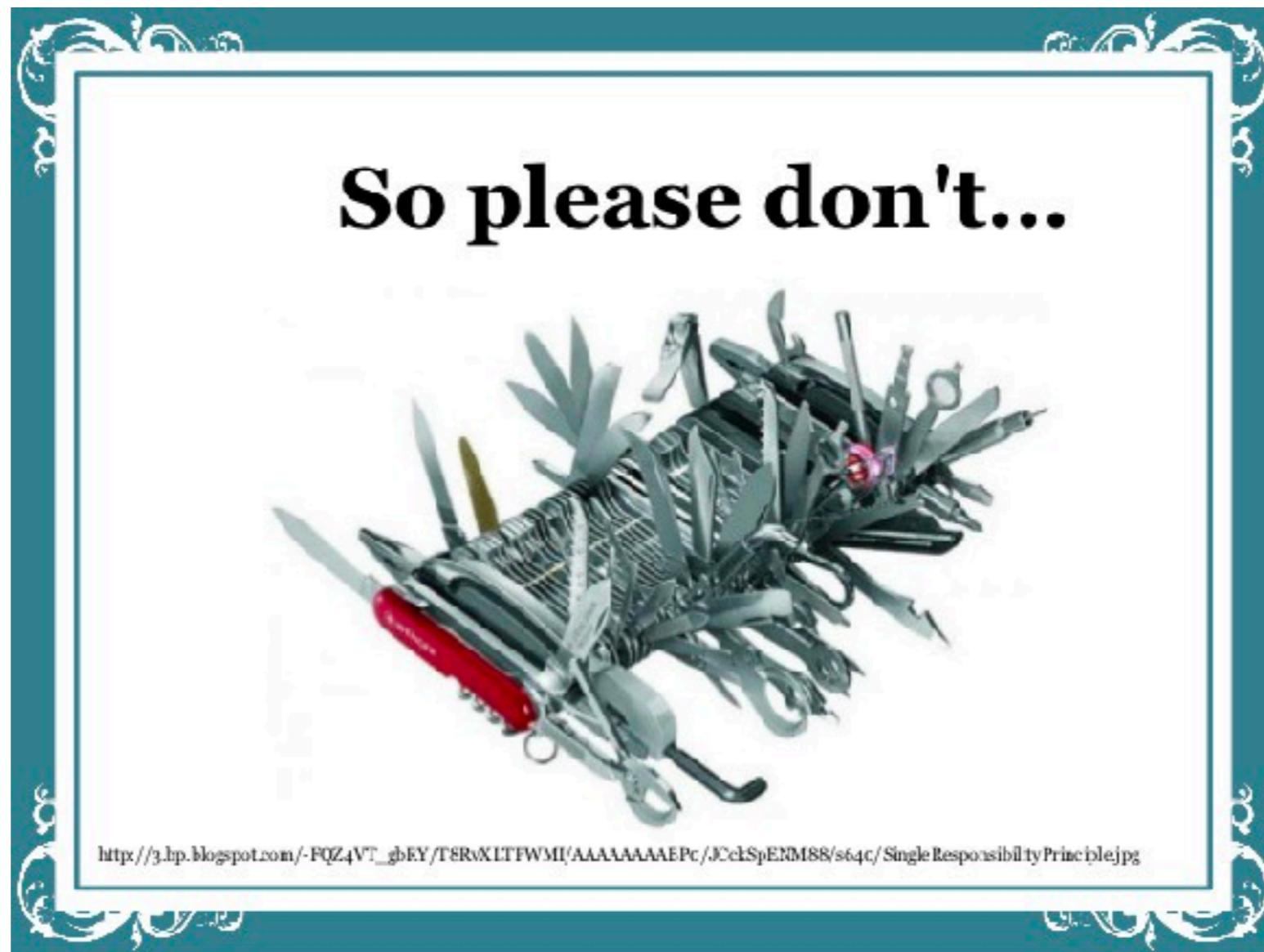
Interface Segregation Principle

Dependency Inversion Principle



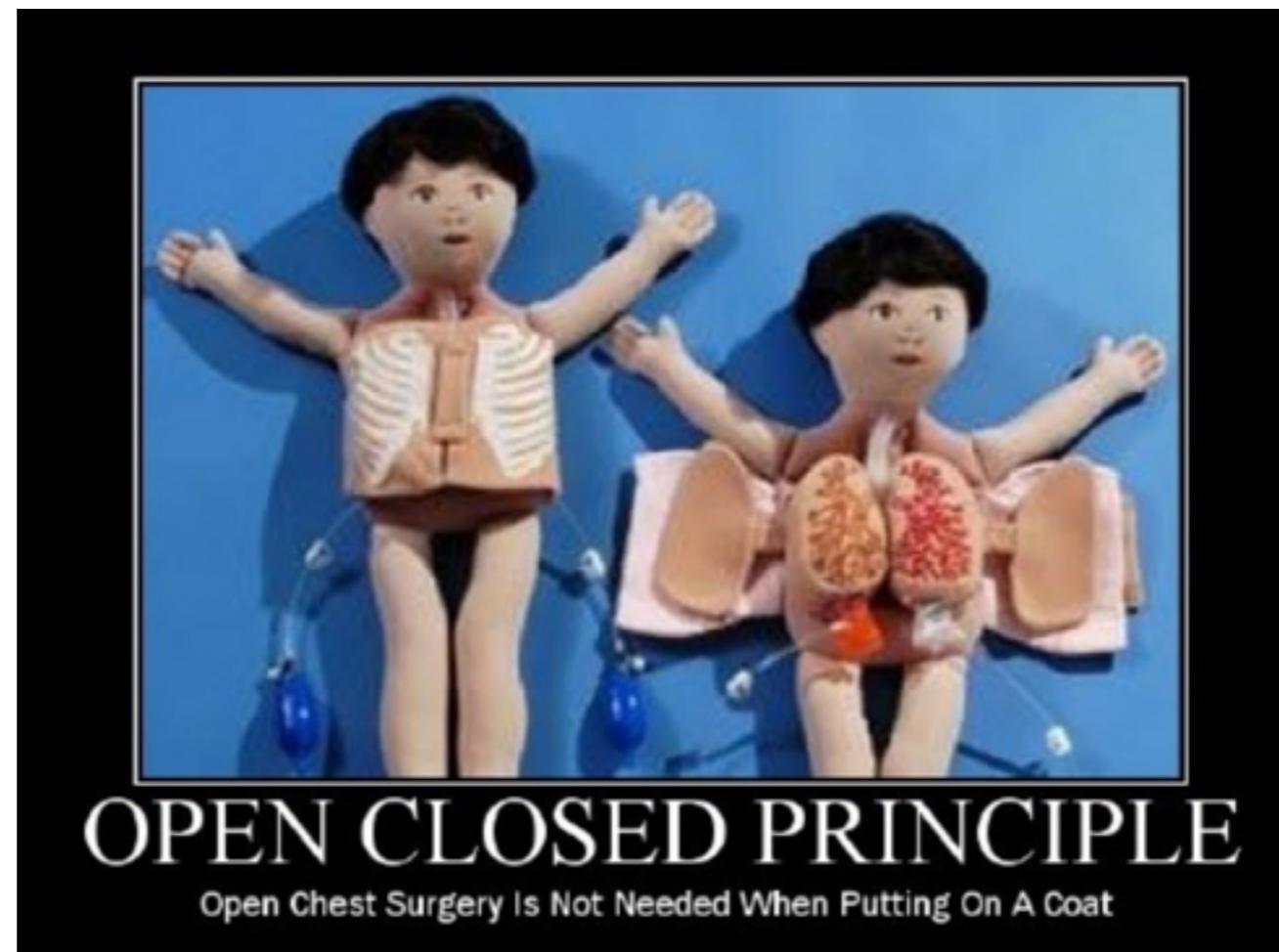
Single Responsibility Principle

A class should have one and only one reason to change



Open Closed Principle

Software entities should open for extension
But closed for modification



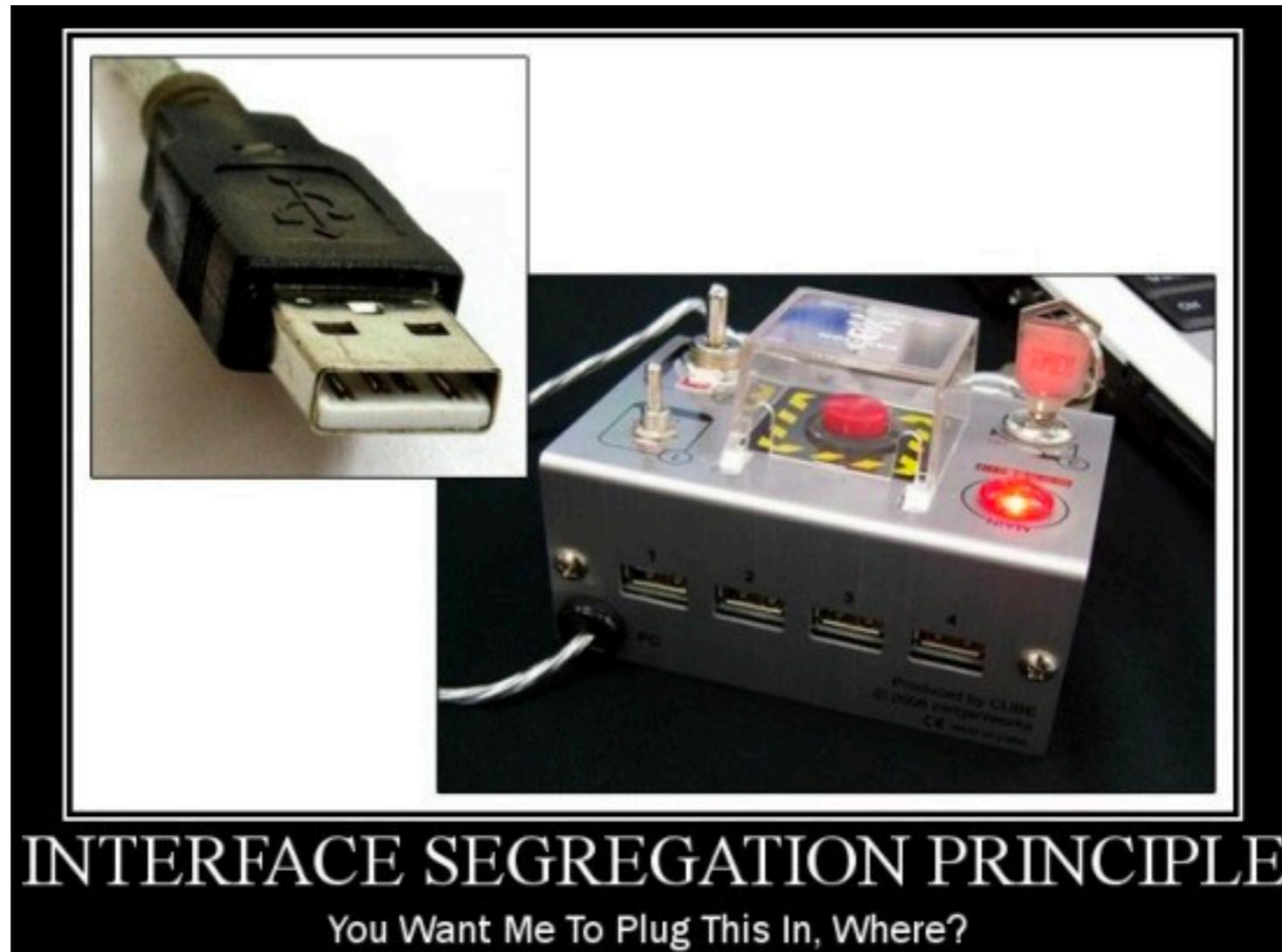
Liskov Substitution Principle

Child classes should never break the parent



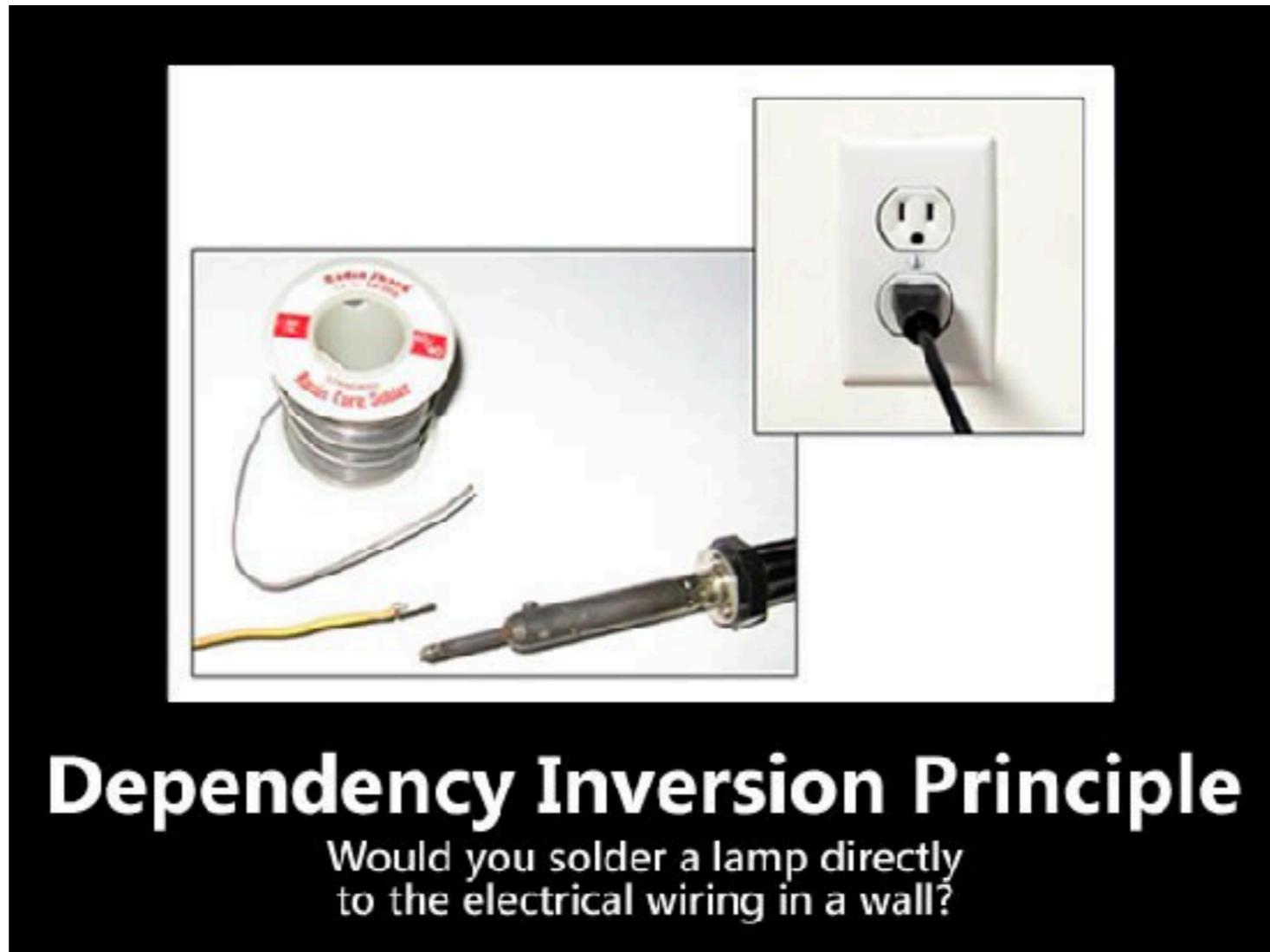
Interface Segregation Principle

Make fine grained interfaces that are client specific



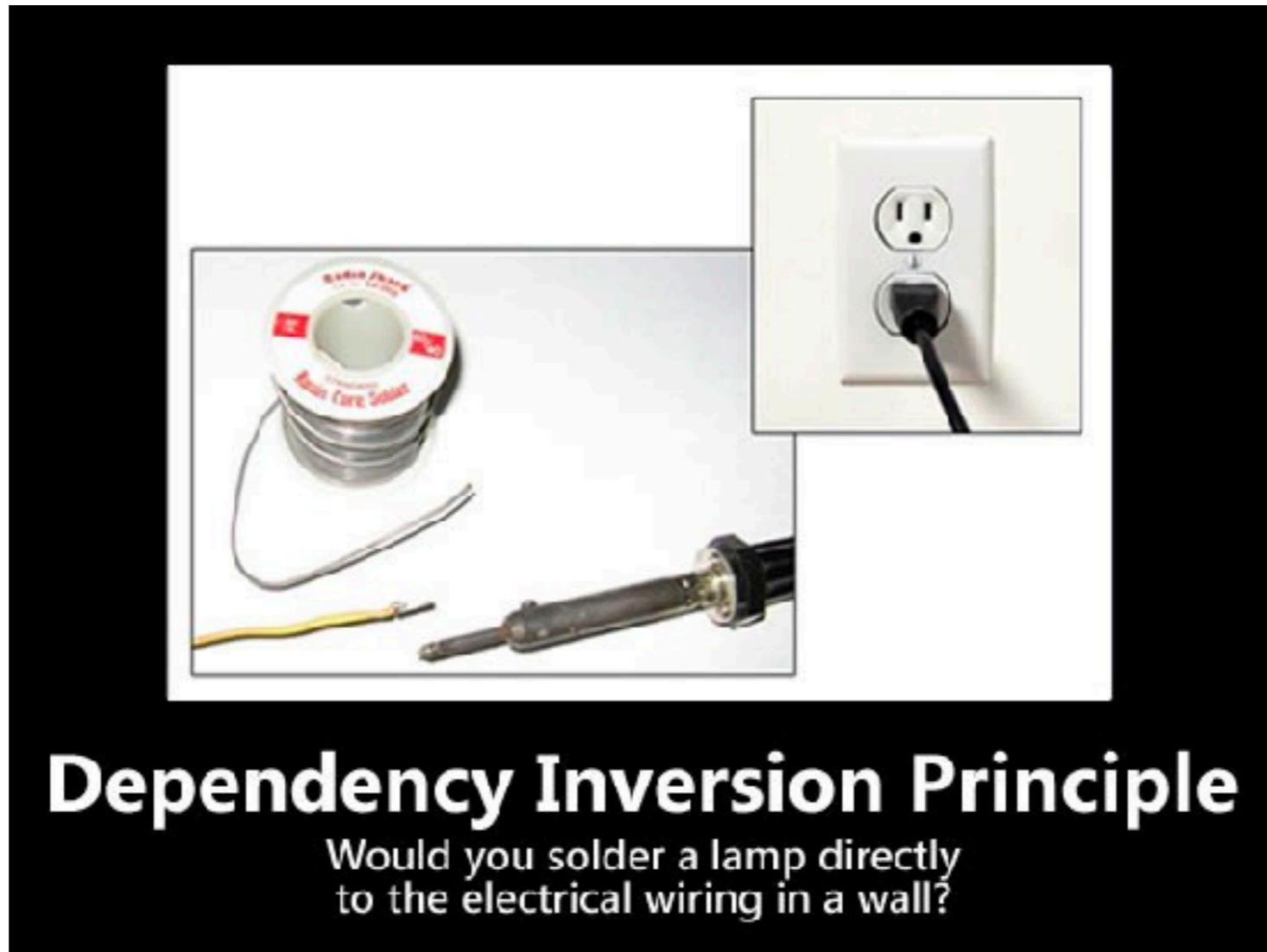
Dependency Inversion Principle

High-level module should not depend on low-level module



Dependency Inversion Principle

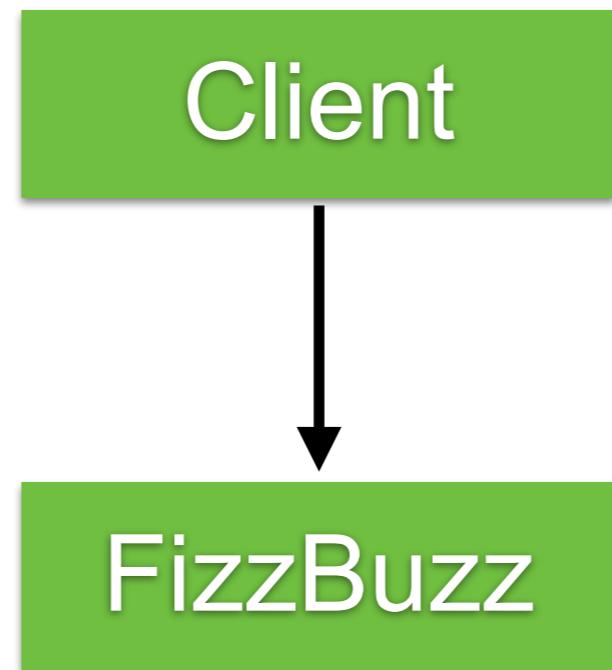
Abstraction layer should not depend on detail



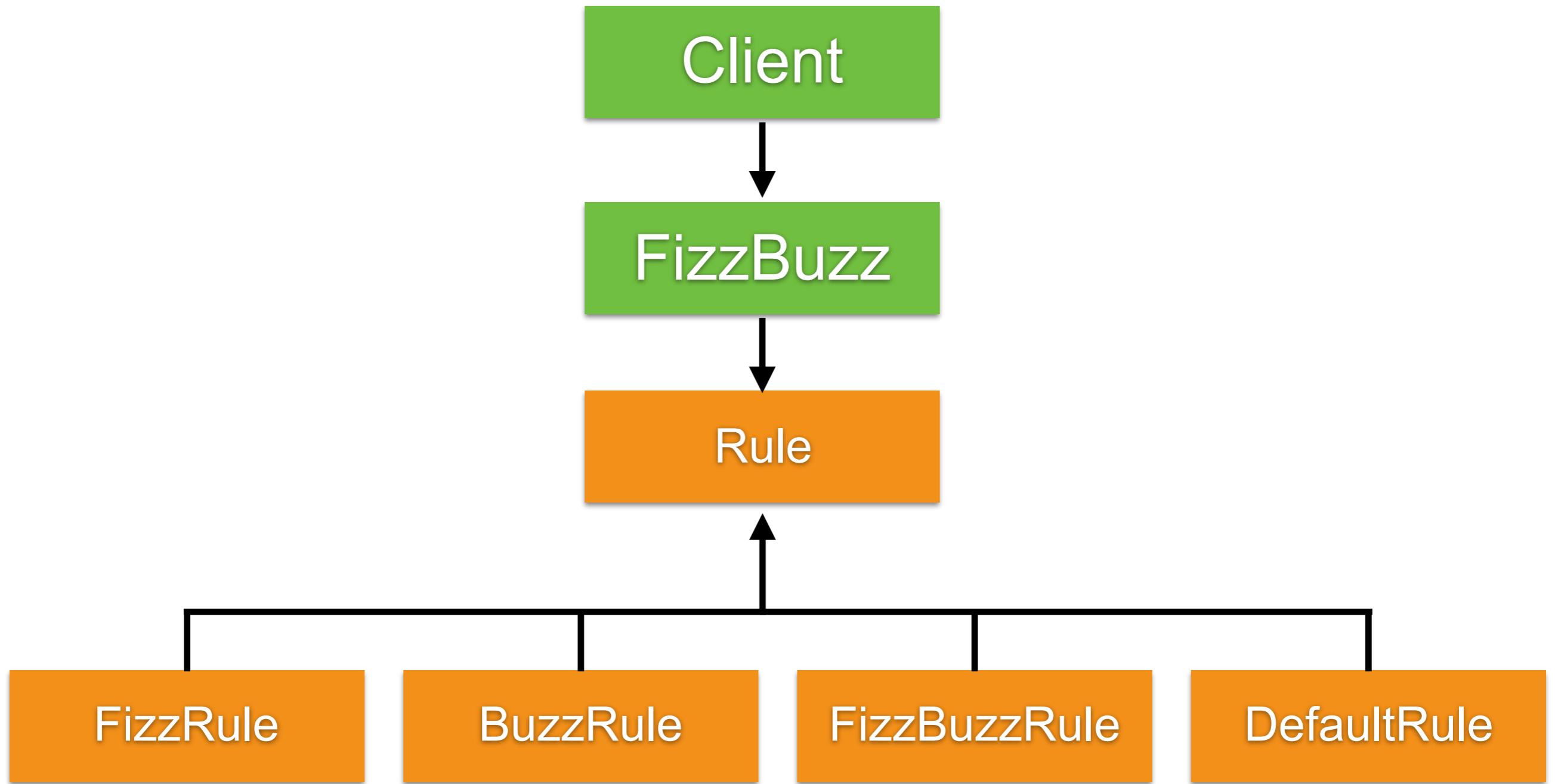
Workshop



FizzBuzz



FizzBuzz



Spring Boot



Spring Framework



Spring Boot



REST



REST

REpresentation **S**tate **T**ransfer

The style of software architecture behind
RESTful services

Defined in 2000 by Roy Fielding

https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm



Goals

Scalability
Generality of interfaces
Independent deployment of components



RESTful service



REST Request Messages

RESTful request is typically in form of
Uniform Resource Identifiers (URI)



REST Request Messages

RESTful request is typically in form of
Uniform Resource Identifiers (URI)

Structure of URI depend on specific service



REST Request Messages

RESTful request is typically in form of
Uniform Resource Identifiers (URI)

Structure of URI depend on specific service

Request can include parameter and data in
body of request as XML, JSON etc.



REST Request & Response

Request Method

GET, POST, PUT, DELETE

Client

Server



Response Format

XML or JSON



HTTP Methods meaning

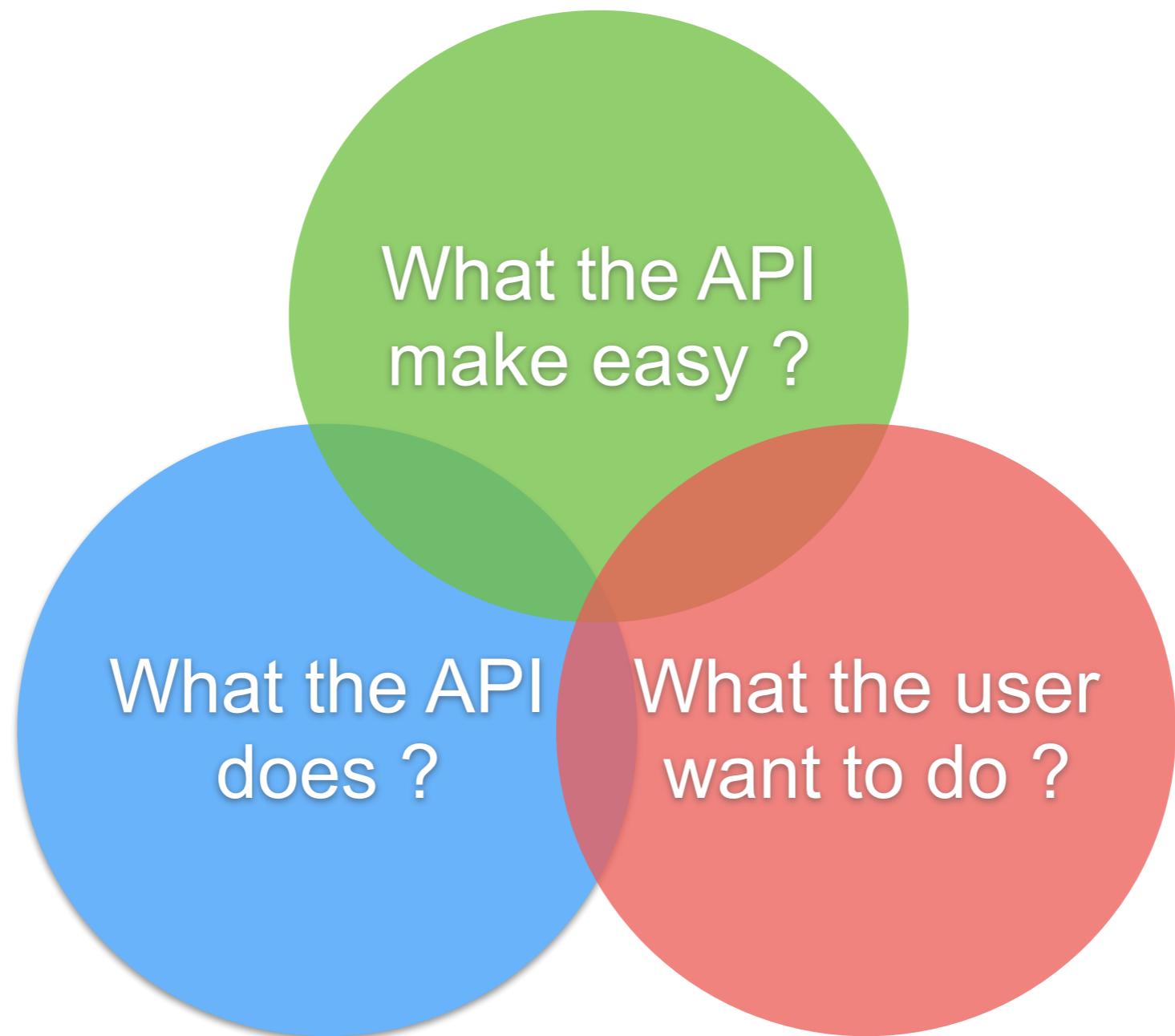
Method	Meaning
GET	Read data
POST	Create/Insert new data
PUT/PATCH	Update data or insert if a new id
DELETE	Delete data



Response format ?



Good APIs ?

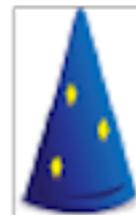


Java Framework for RESTful



unirest

Lightweight HTTP Request Client Libraries



Dropwizard



Restlet



ACT.framework

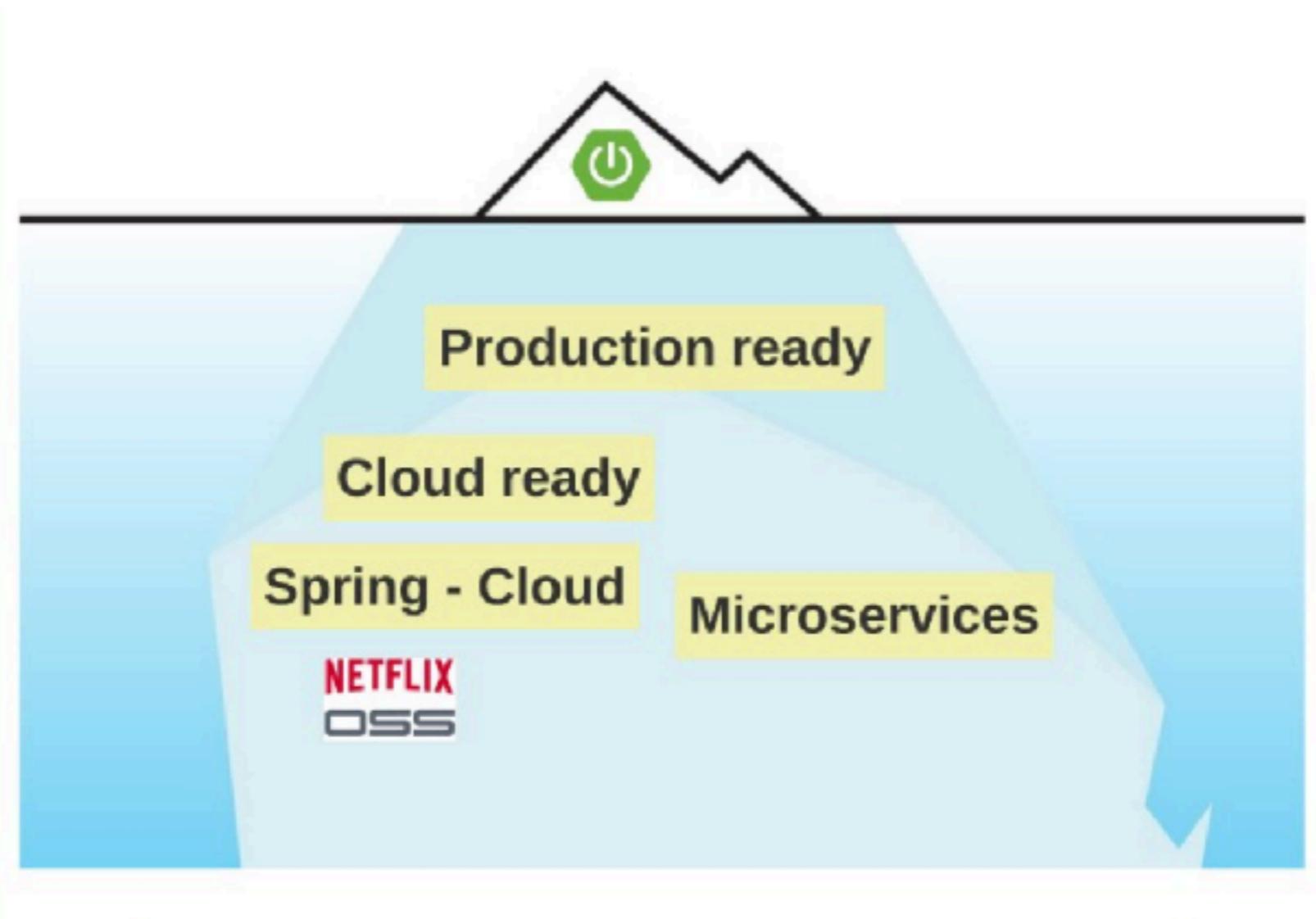


Hello Spring Boot 2.x



Why ?

Application skeleton generator
Reduce effort to add new technologies



What ?

Embedded application server

Integration with tools/technologies (starter)

Production tools (monitoring, health check)

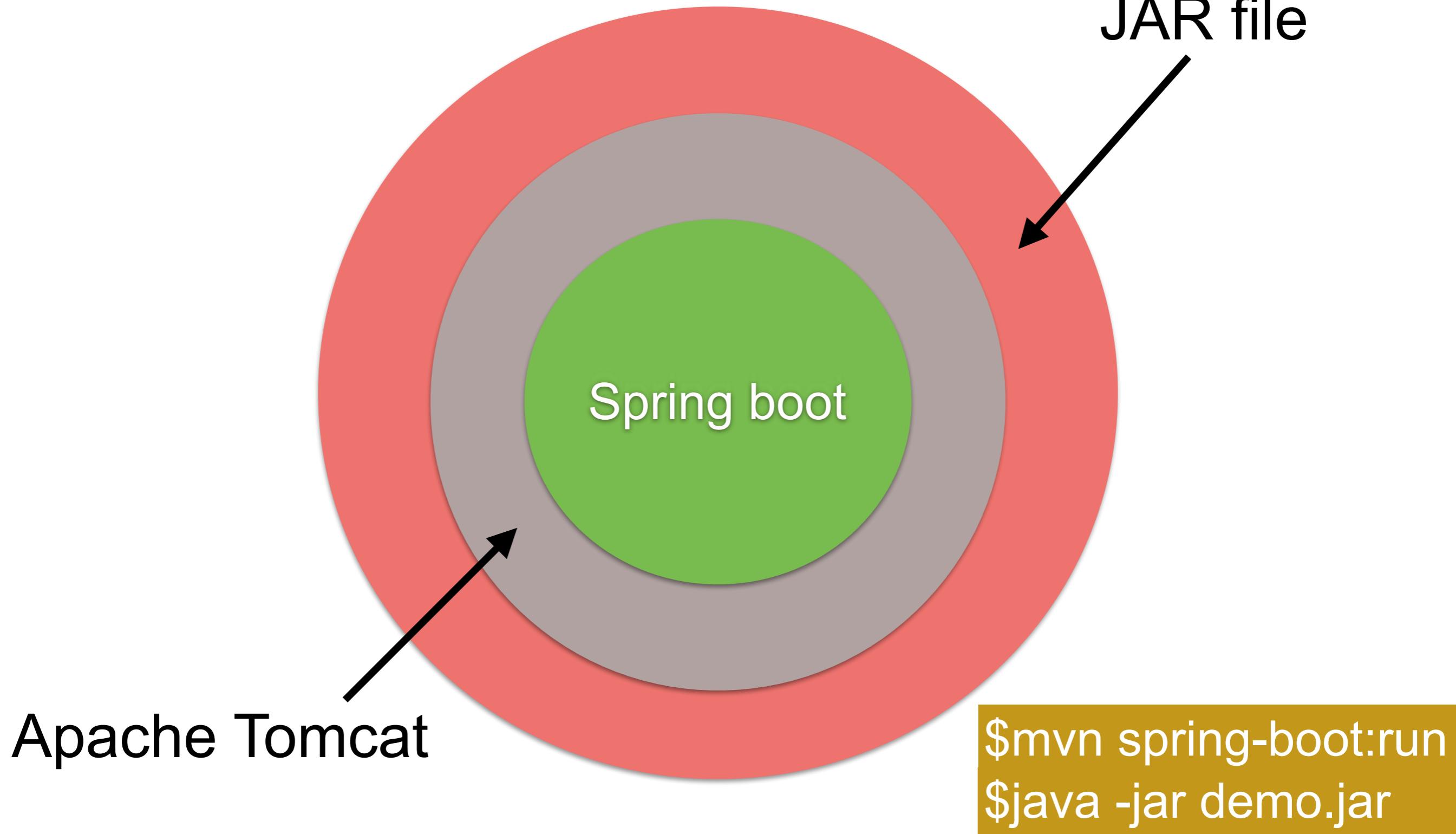
Configuration management

Dev tools

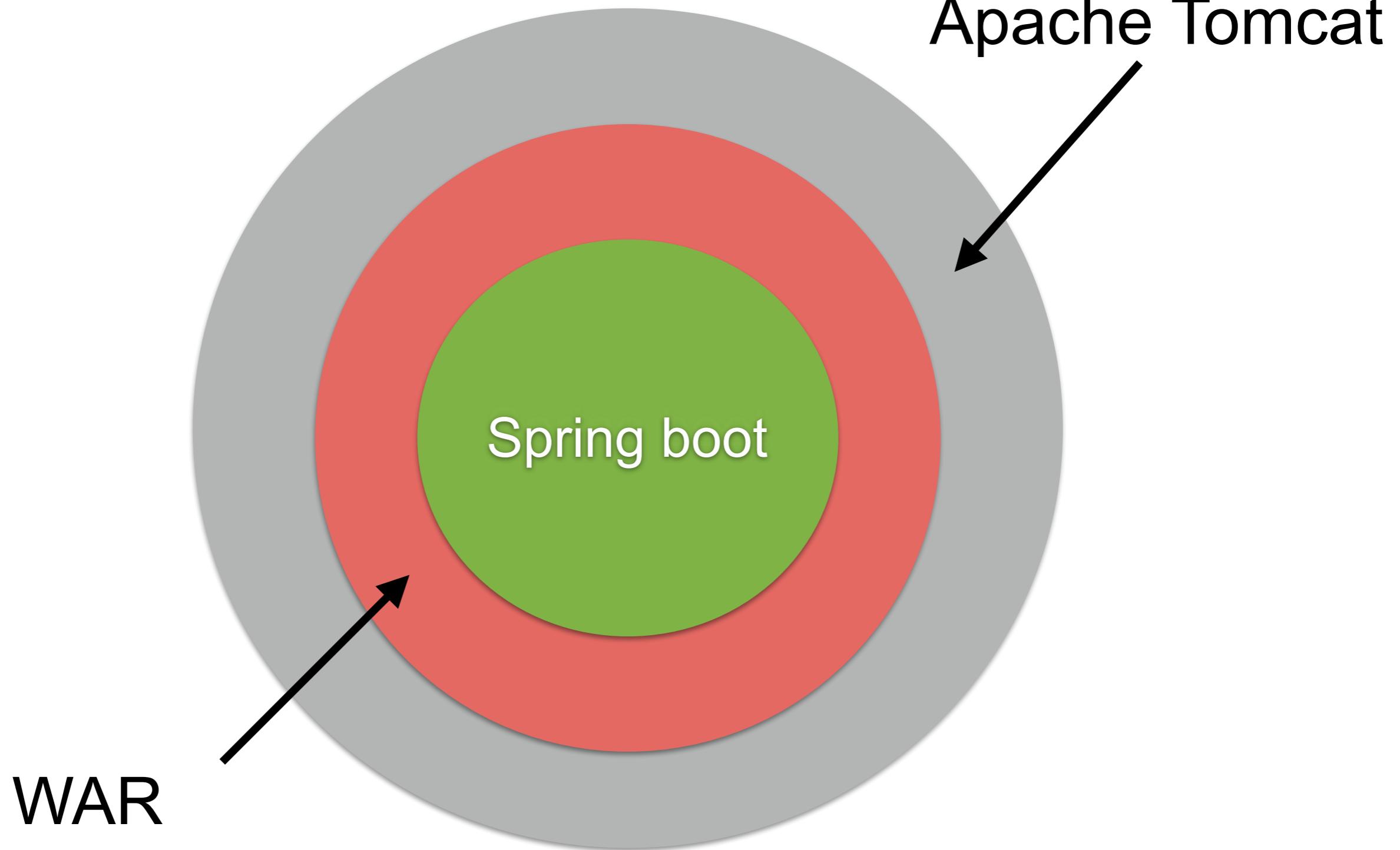
No source code generation, no XML



How ?



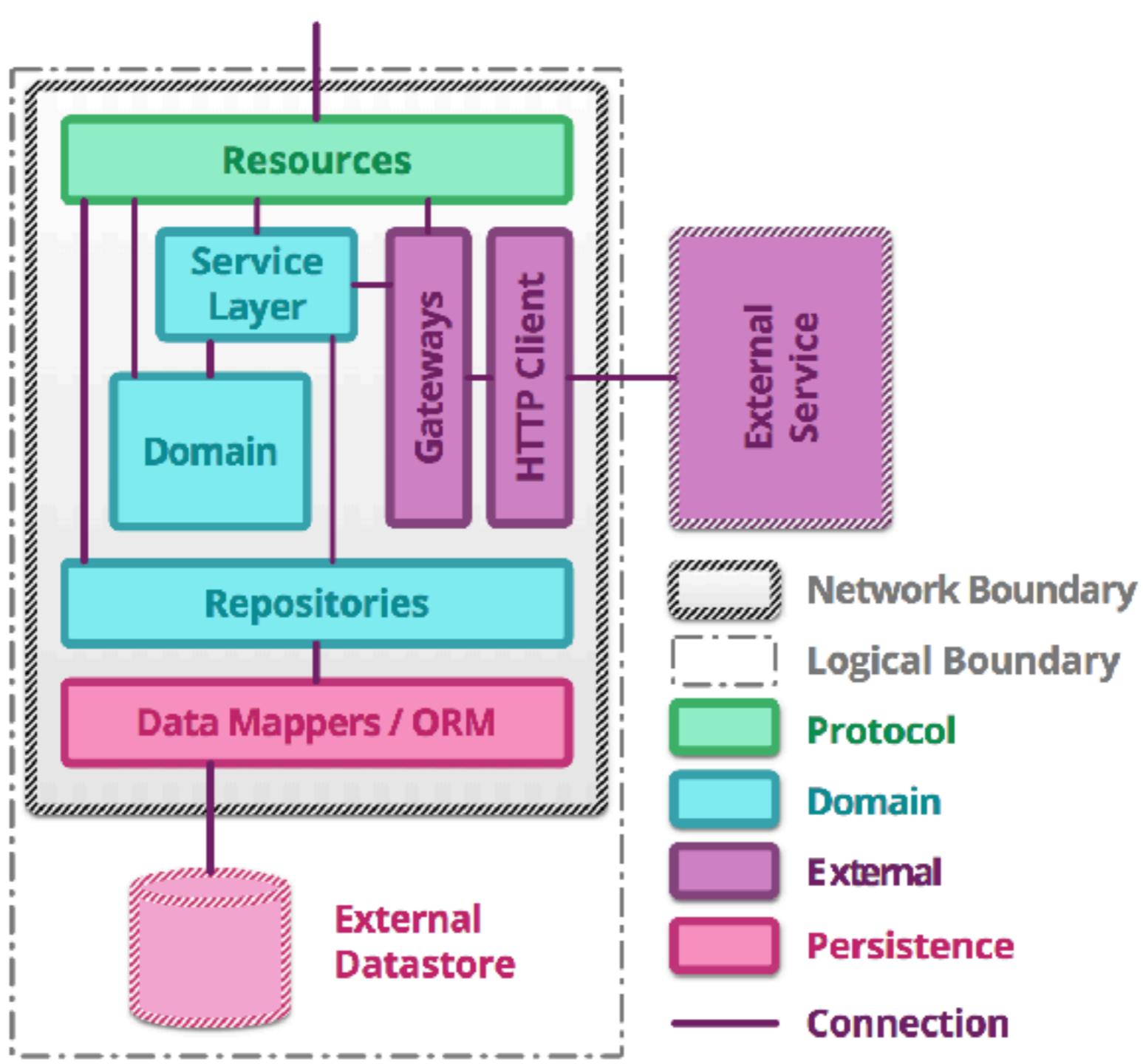
How ?



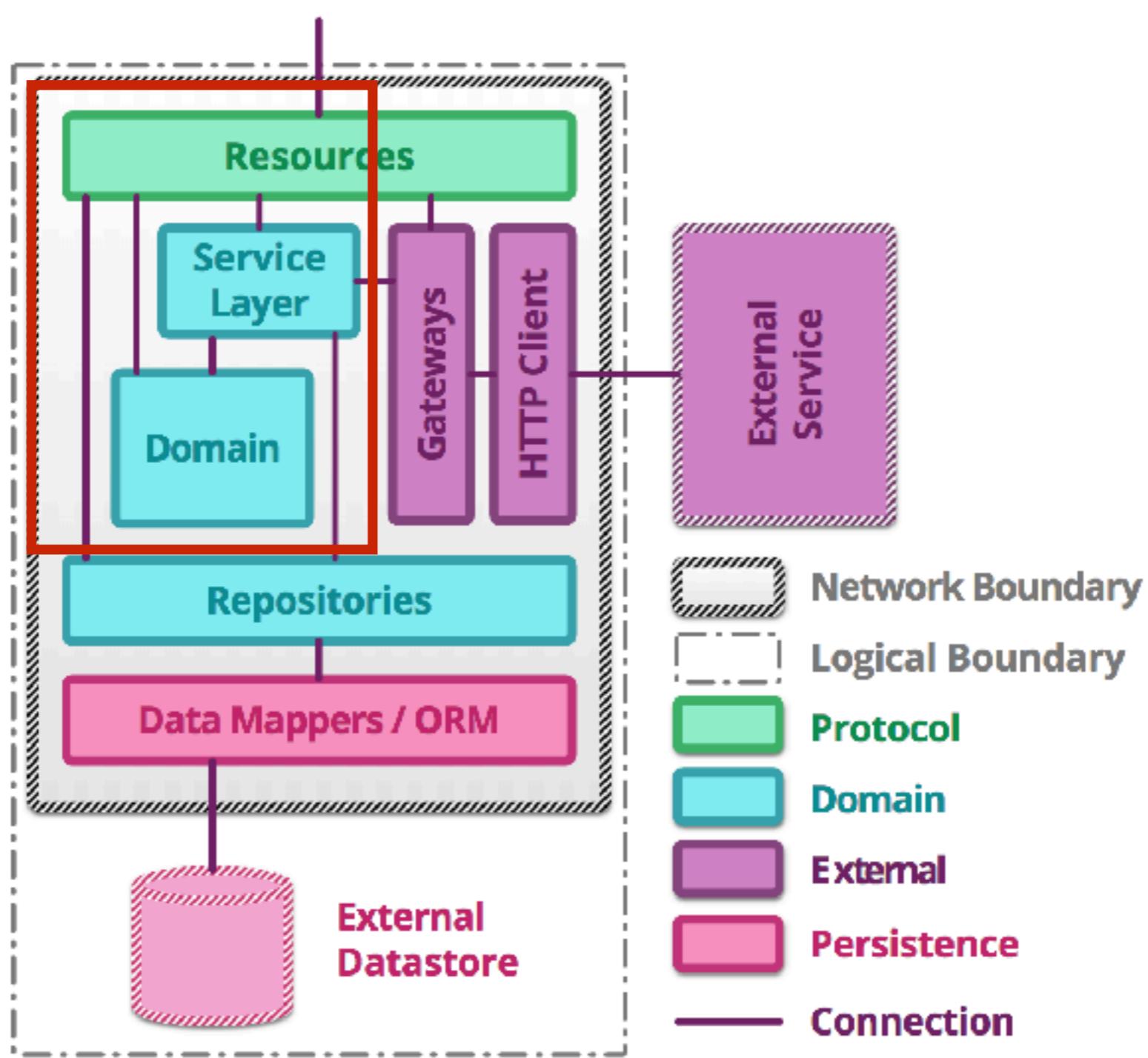
Building RESTful API with Spring Boot



Service Structure



Service Structure



Create project with Spring Initializr



Spring Initializr

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

SPRING INITIALIZR bootstrap your application now

Generate a with and Spring Boot

Project Metadata

Artifact coordinates

Group

Artifact

Dependencies

Add Spring Boot Starters and dependencies to your application

Search for dependencies

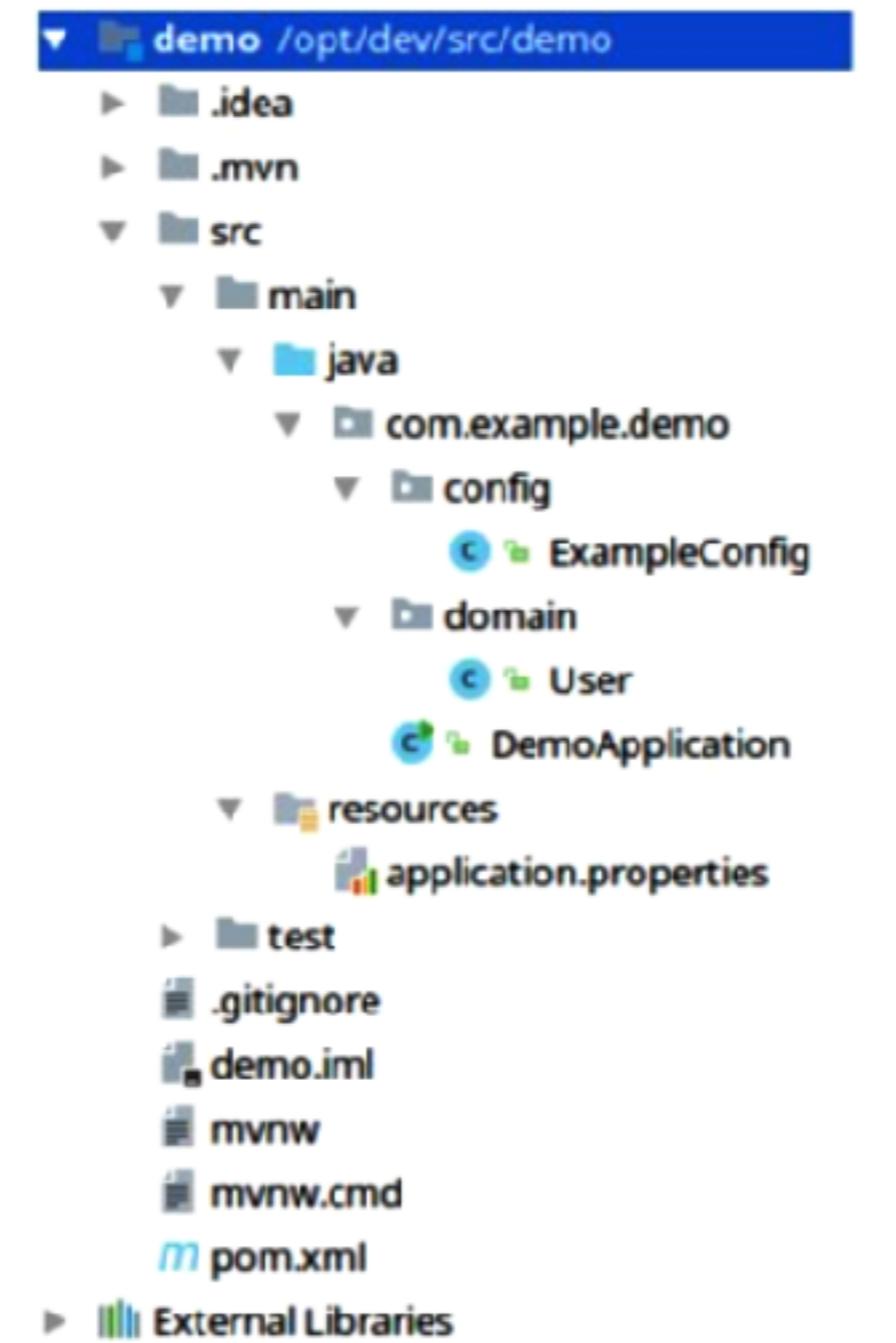
Selected Dependencies

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

start.spring.io is powered by [Spring Initializr](#) and [Pivotal Web Services](#)



Structure of Spring Boot



Spring Boot main class

```
package com.example.demo;

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

@SpringBootApplication
public class DemoApplication {

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



Run project (Dev mode)

```
./mvnw spring-boot:run
```

```
      .--.----( )----\VVV\\
     (( ))\__| | | | | |
    \VVV __)| | | | | |
    ' | | | | | | |
=====|_|=====|_|/_=/_/_/_/_/
:: Spring Boot ::          (v2.0.2.RELEASE)

2018-06-07 13:03:30.412  INFO 12828 --- [
oApplication           : Starting DemoApplication on
D 12828 (started by somkiat in /Users/somkiat/Down
2018-06-07 13:03:30.418  INFO 12828 --- [
oApplication           : No active profile set, fall
```



Run project (production mode)

```
./mvnw package  
$java -jar target/<file name>.jar
```



Display all beans !!

```
@SpringBootApplication
public class DemoApplication {

    public static void main(String[] args) {
        ApplicationContext context
            = SpringApplication.run(DemoApplication.class,
                System.out.println("Hello World"));

        for (String name: context.getBeanDefinitionNames()) {
            System.out.println(name);
        }
    }
}
```



Custom Banner



Custom banner

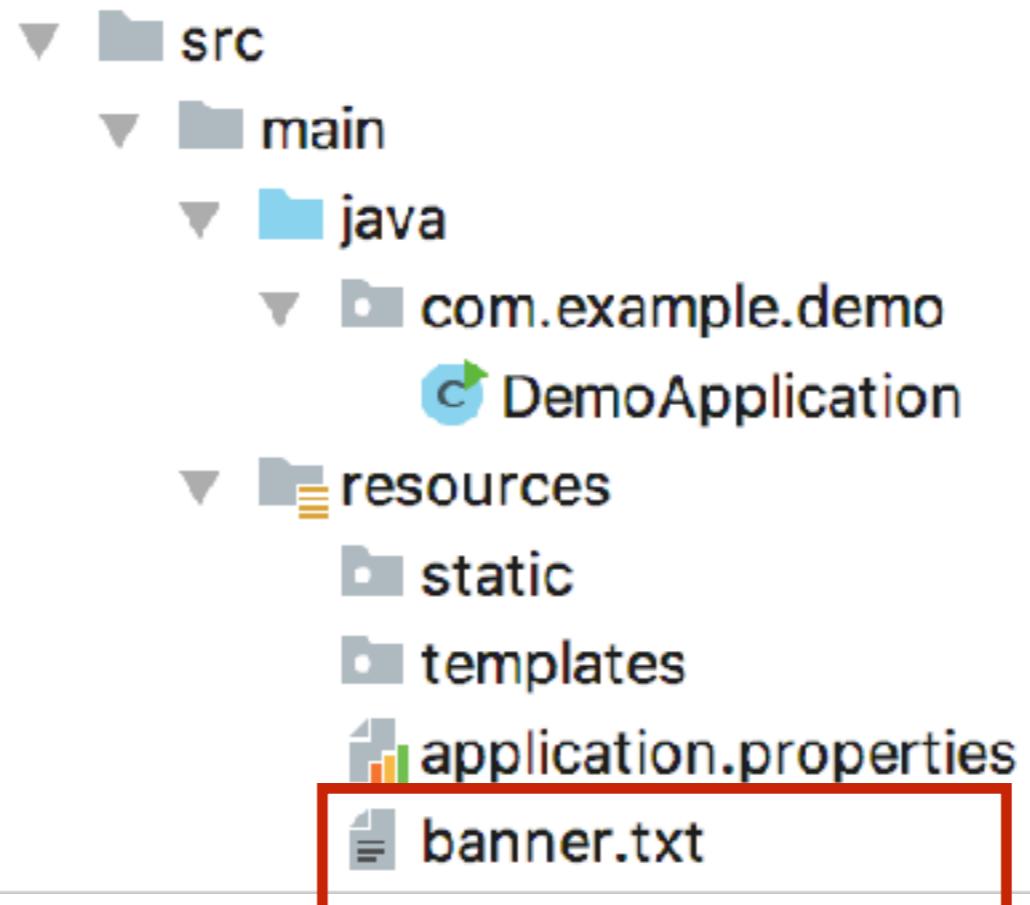
The Spring Boot logo consists of a grid of characters including slashes, parentheses, brackets, and underscores, arranged in a pattern that forms the words "Spring" and "Boot". Below the grid, the text ":: Spring Boot ::" is written in green, followed by "(v2.0.2.RELEASE)" in gray.

```
2018-06-07 13:03:30.412  INFO 12828 --- [  
oApplication           : Starting DemoApplication on  
D 12828 (started by somkiat in /Users/somkiat/Down  
2018-06-07 13:03:30.418  INFO 12828 --- [  
oApplication           : No active profile set, fall
```



Custom banner (1)

Create file banner.txt or banner.png in resources folder



Custom banner (2)

```
@SpringBootApplication
public class DemoApplication {

    public static void main(String[] args) {
        ImageBanner banner = new ImageBanner(
            new ClassPathResource("try.png"));

        new SpringApplicationBuilder()
            .sources(DemoApplication.class)
            .banner(banner)
            .run();
    }
}
```



Disable banner

```
@SpringBootApplication
public class DemoApplication {

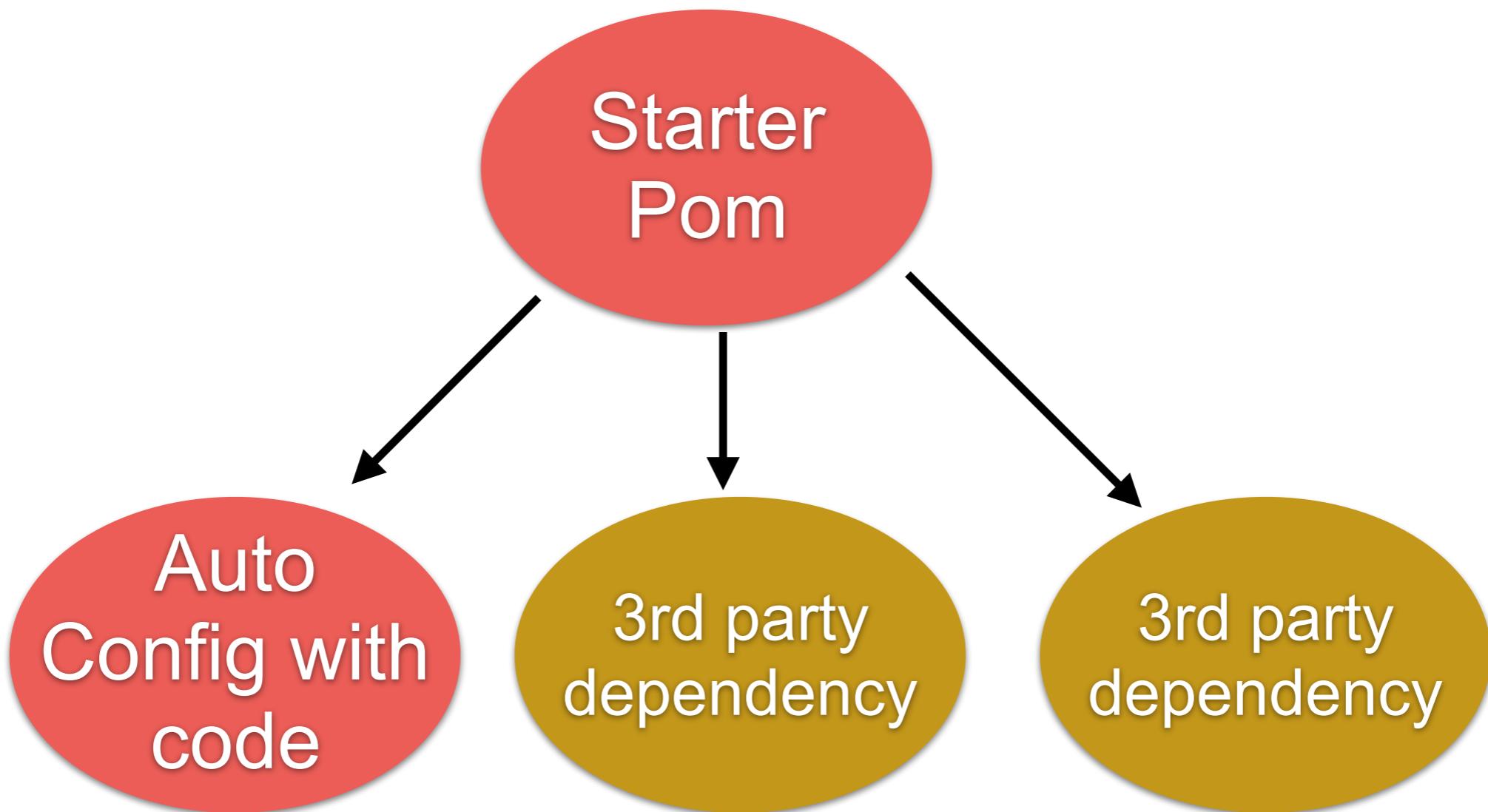
    public static void main(String[] args) {
        new SpringApplicationBuilder()
            .sources(DemoApplication.class)
            .bannerMode(Banner.Mode.OFF)
            .run(args);
    }
}
```



Anatomy of Starter



Anatomy of Starter



Configuration management

Properties/XML/YAML

Config server (App, Spring cloud config, git)

17 ways !!!

<https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-external-config.html>



Look at POM.xml



POM.xml (1)

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.o
<modelVersion>4.0.0</modelVersion>

<groupId>hello</groupId>
<artifactId>hello</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>jar</packaging>

<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.0.0.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
</parent>

<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
    <java.version>1.8</java.version>
</properties>
```



POM.xml (2)

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

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>

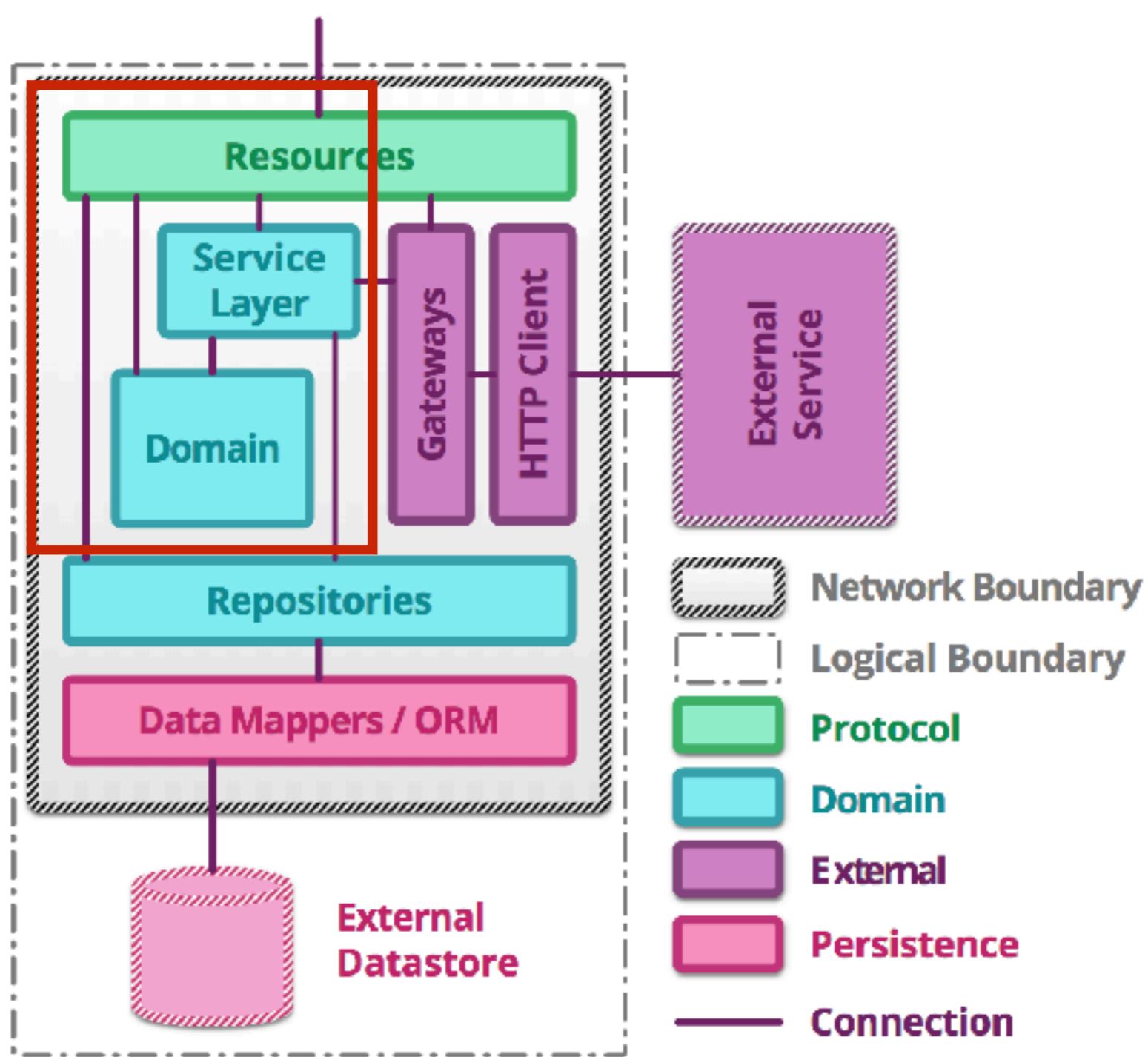
<build>
    <finalName>hello</finalName>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```



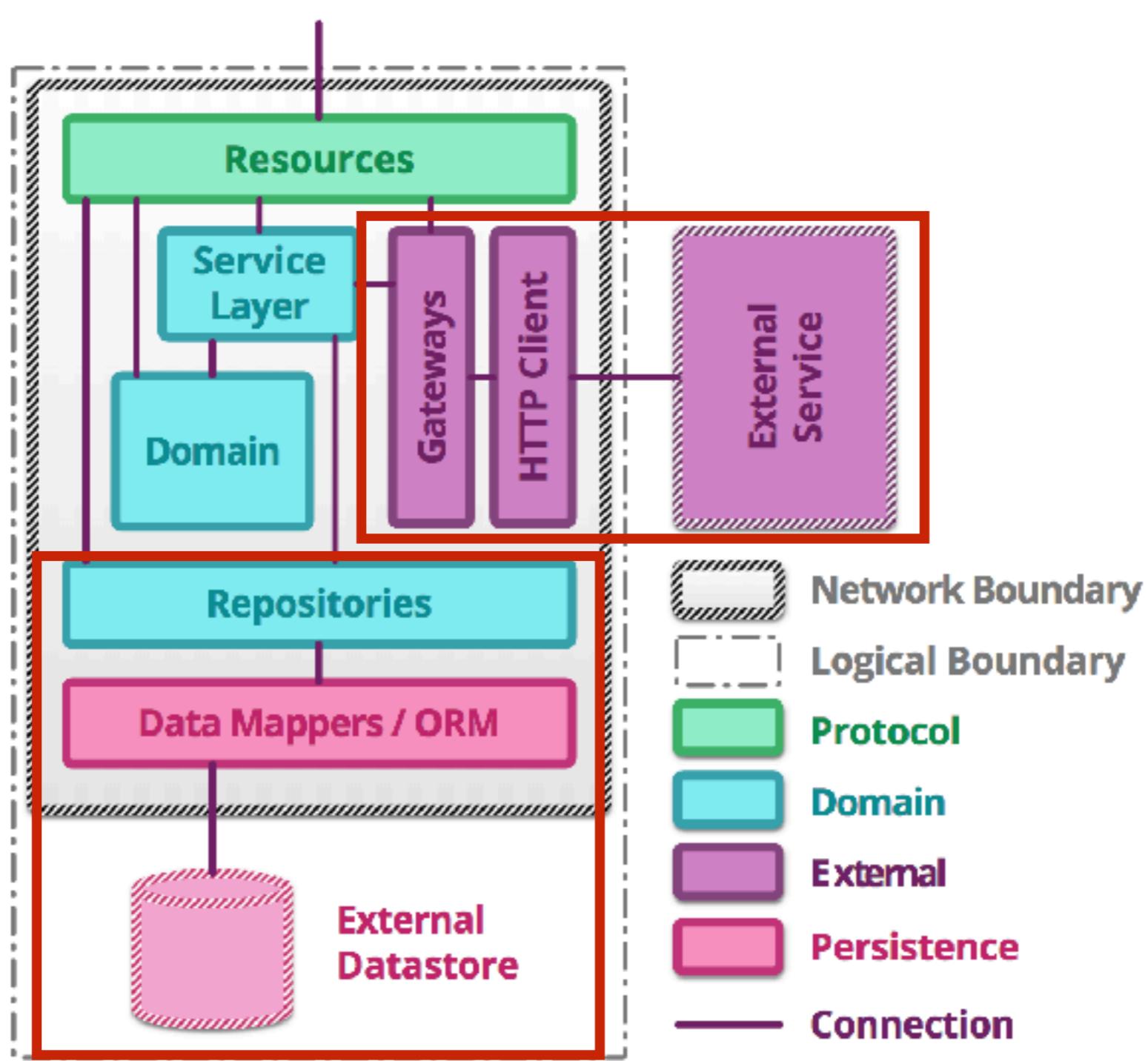
Basic structure of Spring Boot



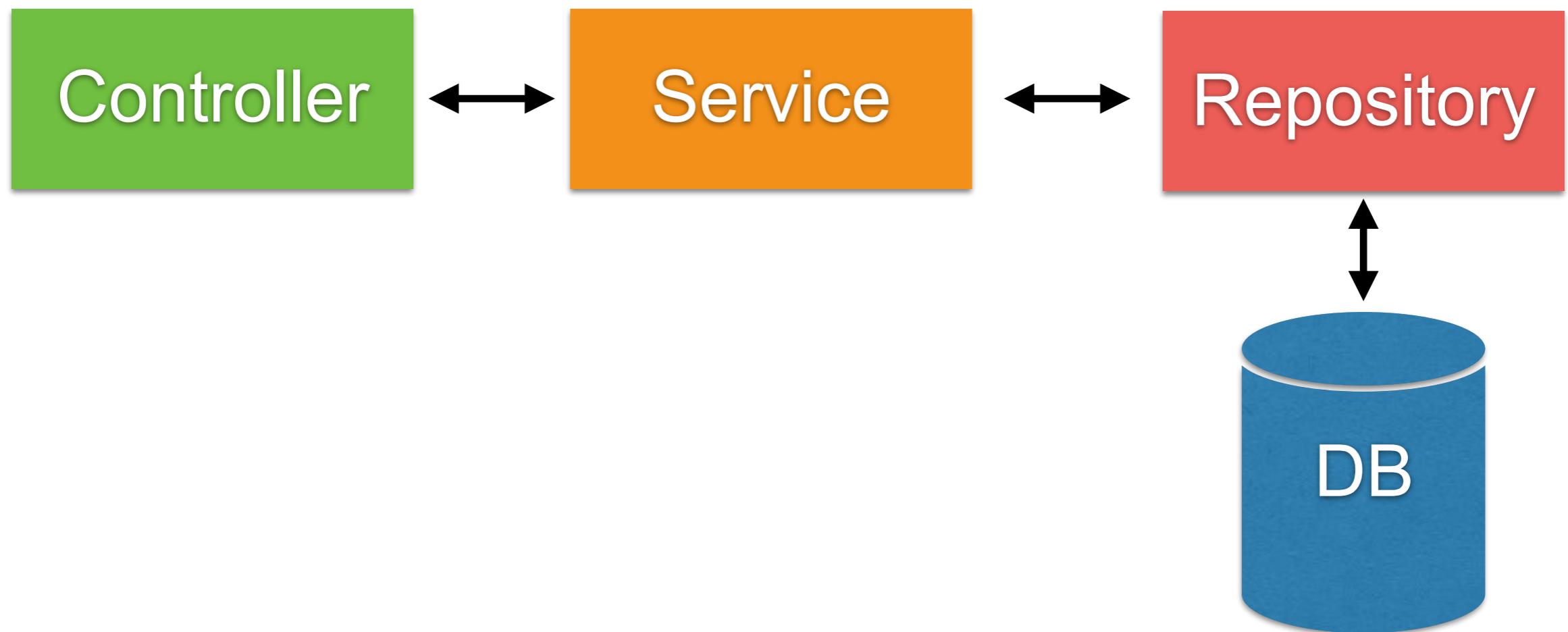
Service Structure



Service Structure



Basic structure of Spring Boot



Controller

Request and Response
Validation input

Delegate to others classes such as service and repository



Service

Control the flow of main business logic
Manage database transaction
Don't reuse service



Repository

Manage data in data store such as RDBMS and
NoSQL



Gateway

Call external service via network such as
WebServices and RESTful APIs



Spring Boot Structure (1)

Separate by function/domain/feature

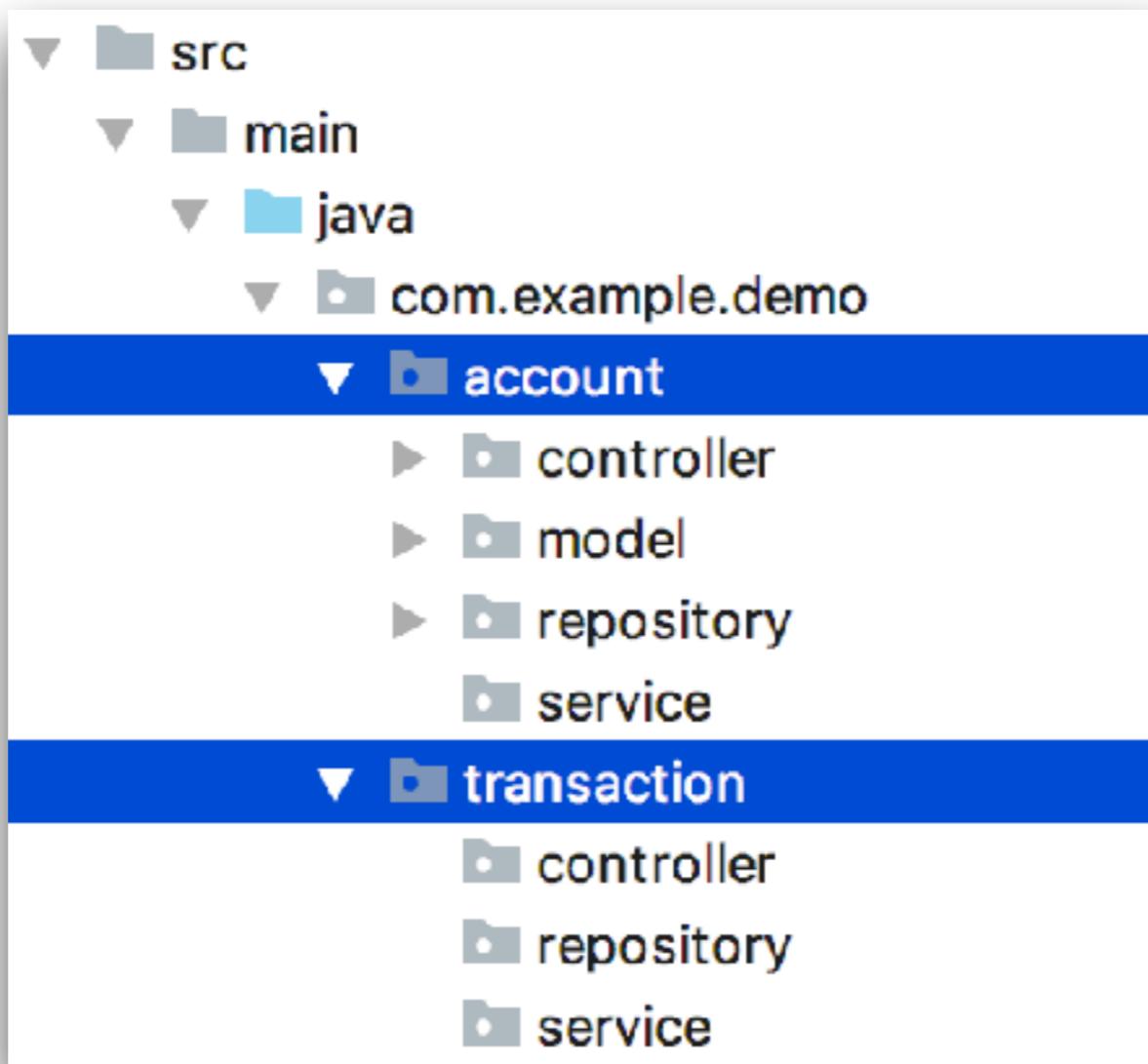
- feature1**
 - controller
 - service
 - repository
- feature2**
 - controller
 - service
 - repository

<https://docs.spring.io/spring-boot/docs/current/reference/html/using-boot-structuring-your-code.html>



Spring Boot Structure (2)

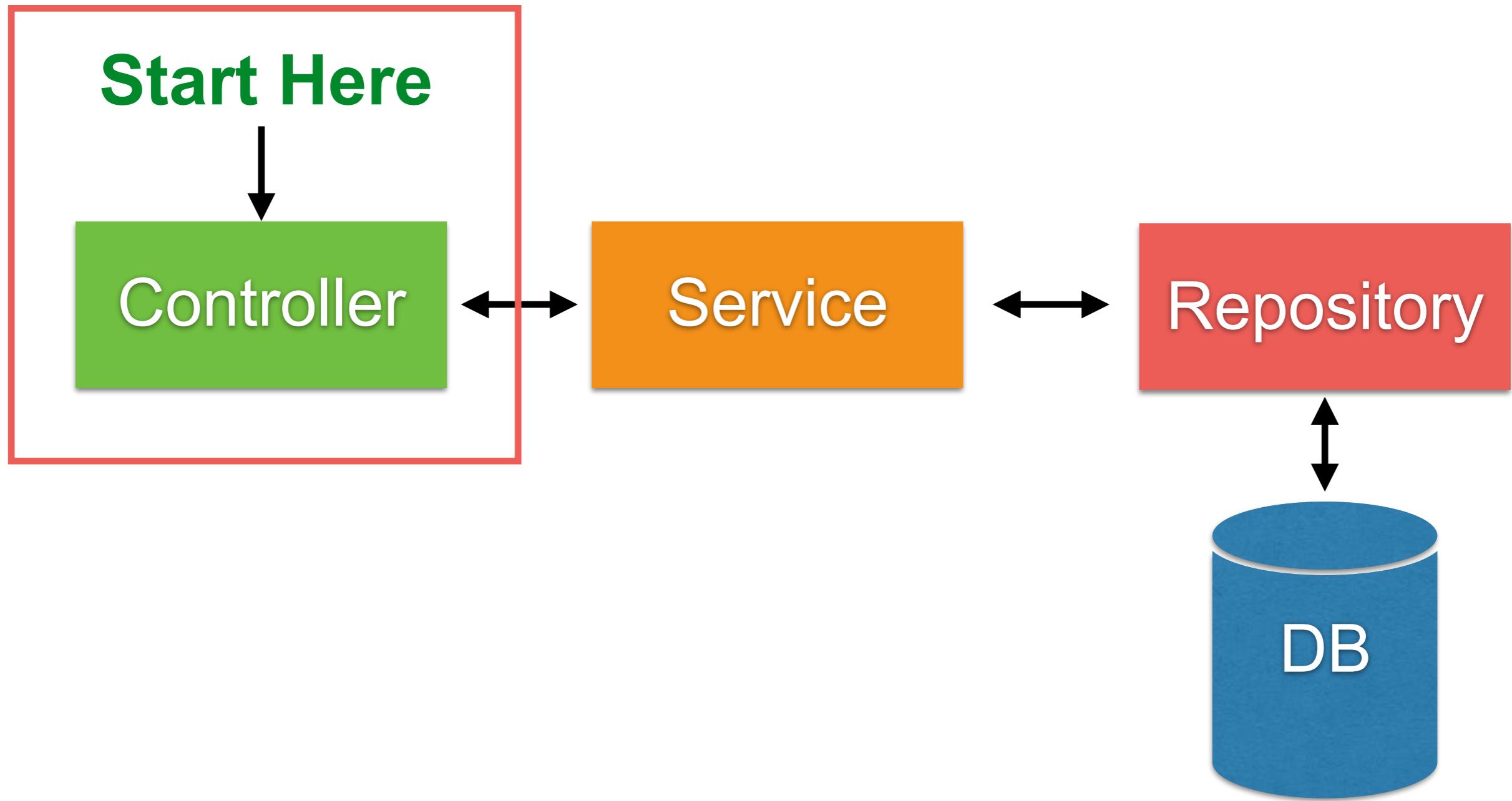
Separate by function/domain/feature



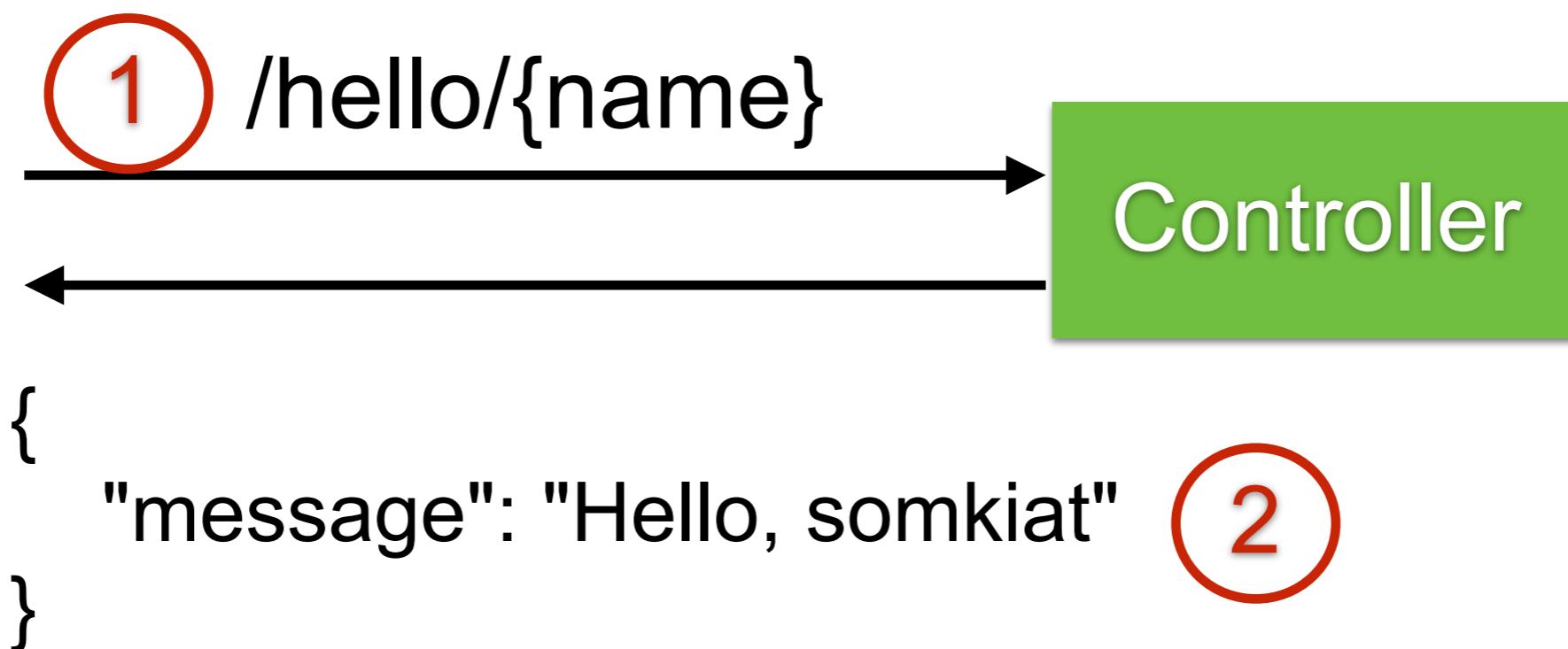
Create first RESTful API



Basic structure of Spring Boot



Hello API



1. Create REST Controller

HelloController.java

```
@RestController
public class HelloController {

    @GetMapping("/hello/{name}")
    public Hello sayHi(@PathVariable String name) {
        return new Hello("Hello, " + name);
    }

}
```



2. Create model class

Hello.java

```
public class Hello {  
  
    private String message;  
  
    Hello(String message) {  
        this.message = message;  
    }  
  
    public String getMessage() {  
        return message;  
    }  
  
    public void setMessage(String message) {  
        this.message = message;  
    }  
}
```



3. Compile and Packaging

\$mvnw clean package



4. Run

```
$java -jar target/hello.jar
```



5. Open in browser

<http://localhost:8080/hello/somkiat>

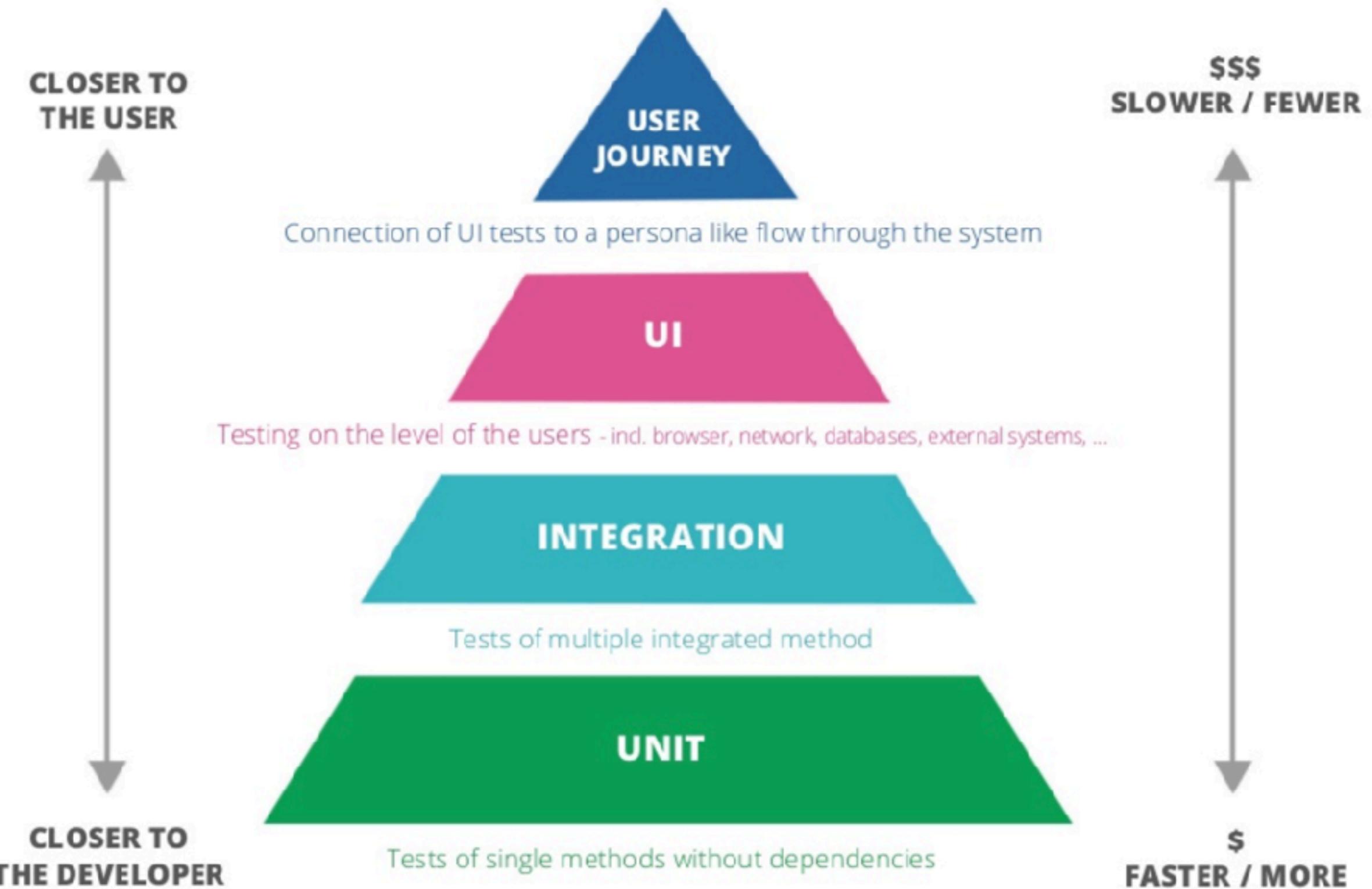


```
{ "message": "Hello somkiat" }
```



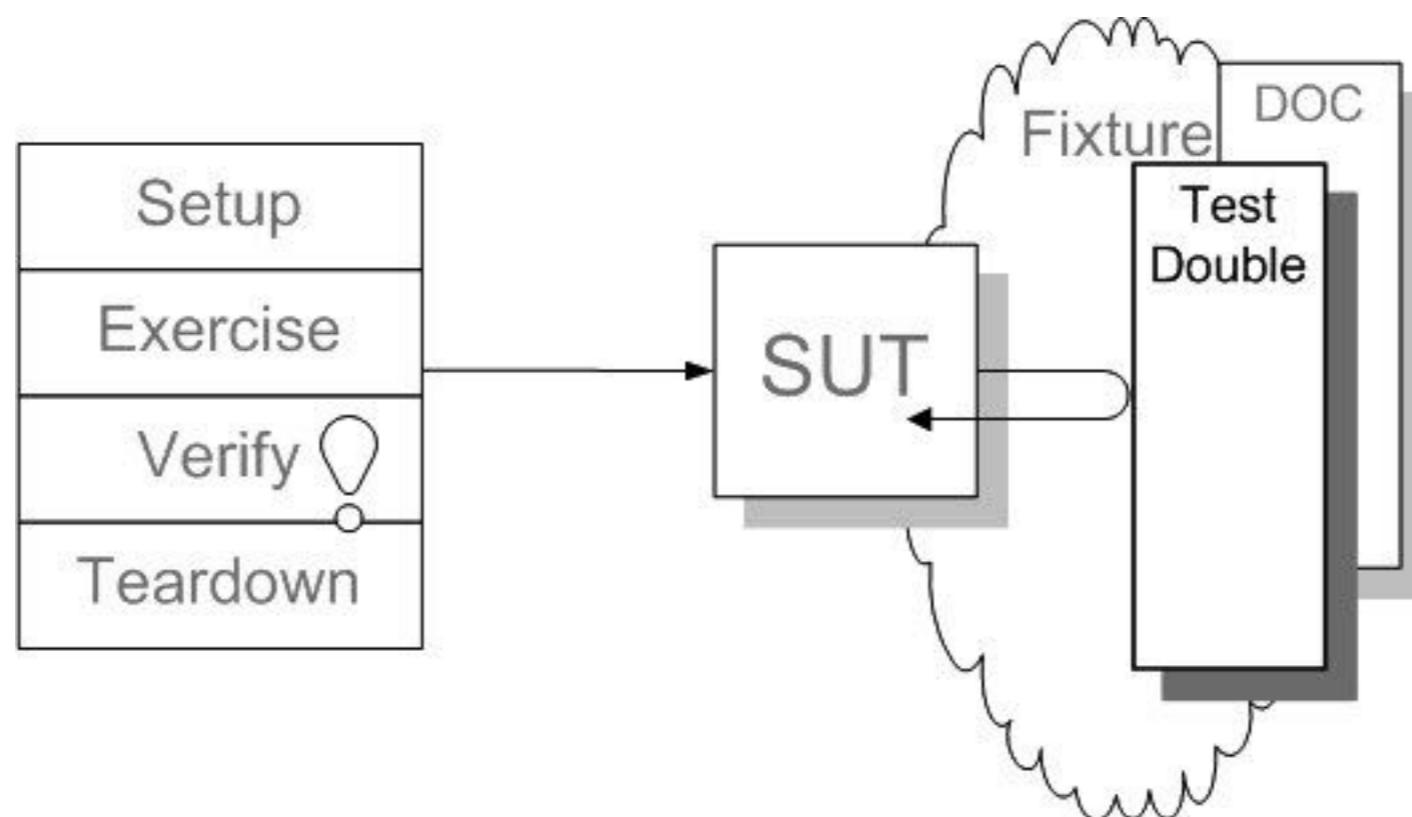
How to test the Hello controller ?





Test Double

How can we verify logic independently ?
How can we avoid Slow tests ?



<http://xunitpatterns.com/Test%20Double.html>



Types of Test Double

Based on **how** and **why** we use it !!

Dummy
object

Test
stub

Test
spy

Mock
object

Fake
object



Controller testing

How to testing with Spring Boot ?



Testing in Spring Boot

@SpringBootTest
@WebMVCTest
@JsonTest
@DataJpaTest
@RestClientTest



Testing in Spring Boot

@SpringBootTest

@WebMVC Test

@JsonTest

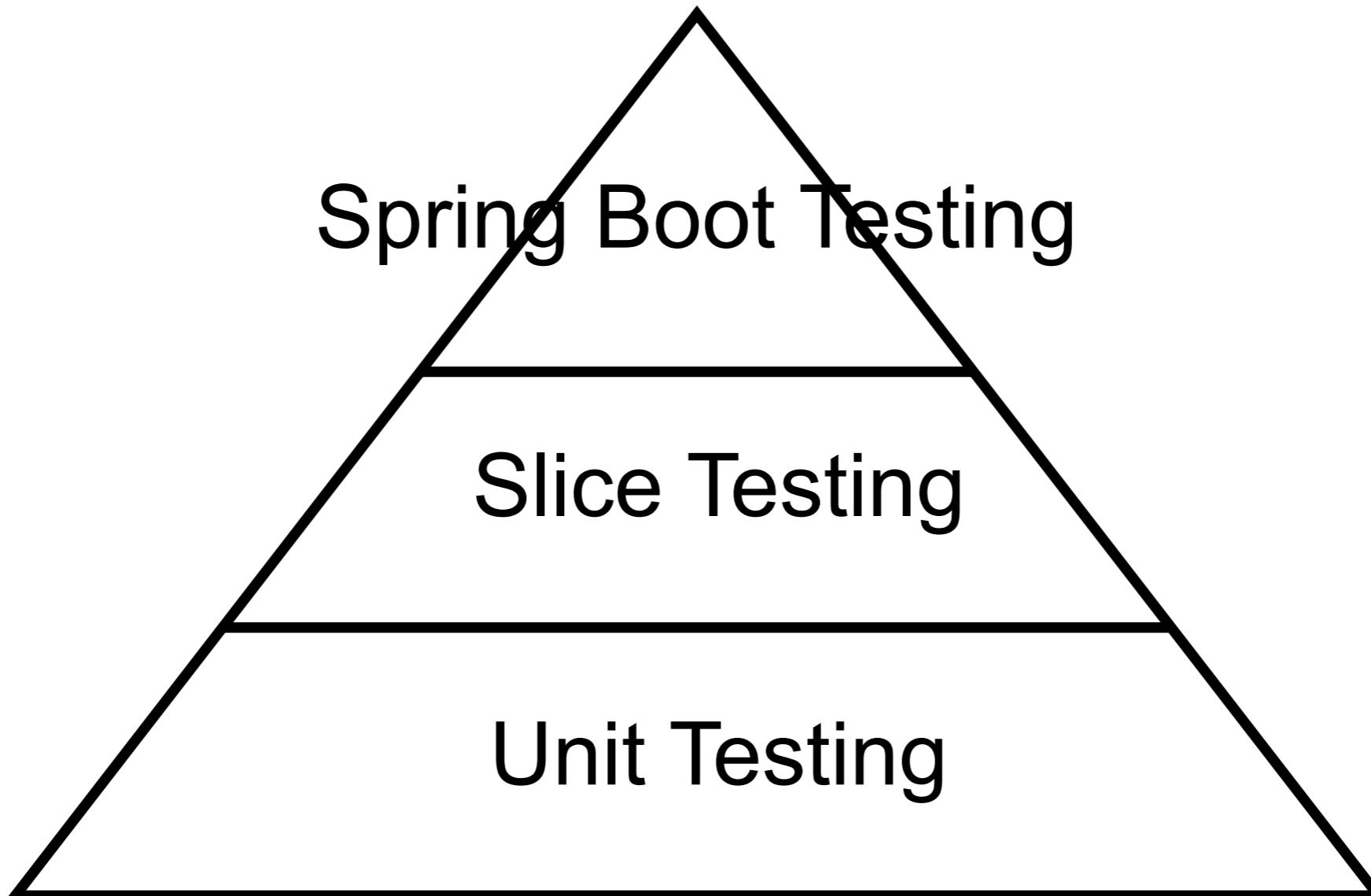
@DataJpaTest

@RestClientTest

Slice testing



Testing in Spring Boot



Controller testing

1. Spring Boot Testing
2. Slice Testing with MockMvc
3. Unit Testing



1. SpringBootTest

Application context

Controller Advice

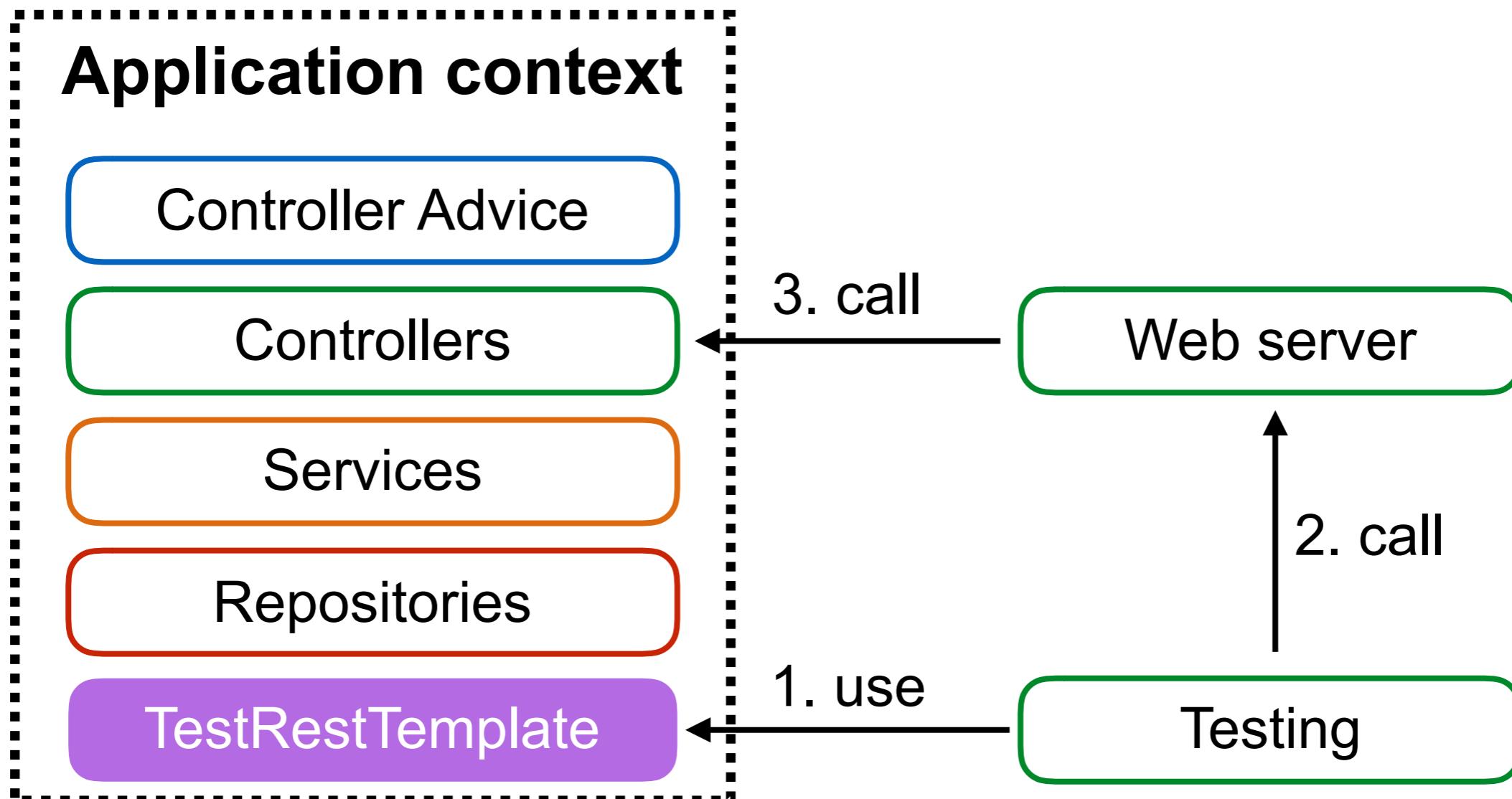
Controllers

Services

Repositories



1. SpringBootTest



SpringBootTest #1

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment
        = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class HelloControllerTest {

    @Autowired
    private TestRestTemplate testRestTemplate;

    @Test
    public void sayHi() {
        // Action :: Call controller
        Hello actualResult
            = testRestTemplate.getForObject("/hello/somkiat",
                                            Hello.class);

        // Assertion :: Check result with expected result
        assertEquals("Hello, somkiat", actualResult.getMessage());
    }
}
```



SpringBootTest #2

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment
        = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class HelloControllerTest {

    @Autowired
    private TestRestTemplate testRestTemplate;

    @Test
    public void sayHi() {
        // Action :: Call controller
        Hello actualResult
            = testRestTemplate.getForObject("/hello/somkiat",
                Hello.class);

        // Assertion :: Check result with expected result
        assertEquals("Hello, somkiat", actualResult.getMessage());
    }
}
```



SpringBootTest #3

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment
        = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class HelloControllerTest {

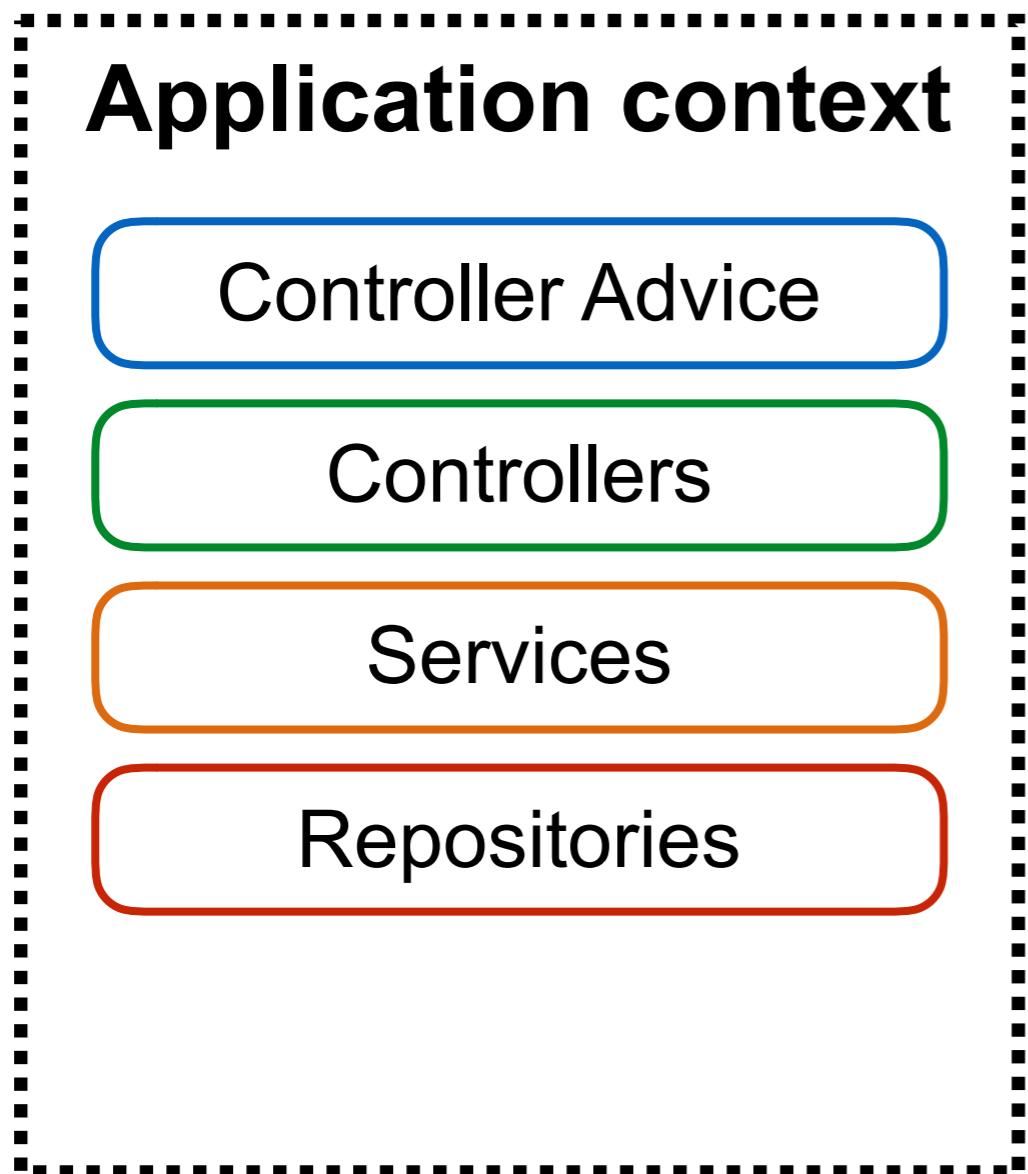
    @Autowired
    private TestRestTemplate testRestTemplate;

    @Test
    public void sayHi() {
        // Action :: Call controller
        Hello actualResult
                = testRestTemplate.getForObject("/hello/somkiat",
                                                Hello.class);

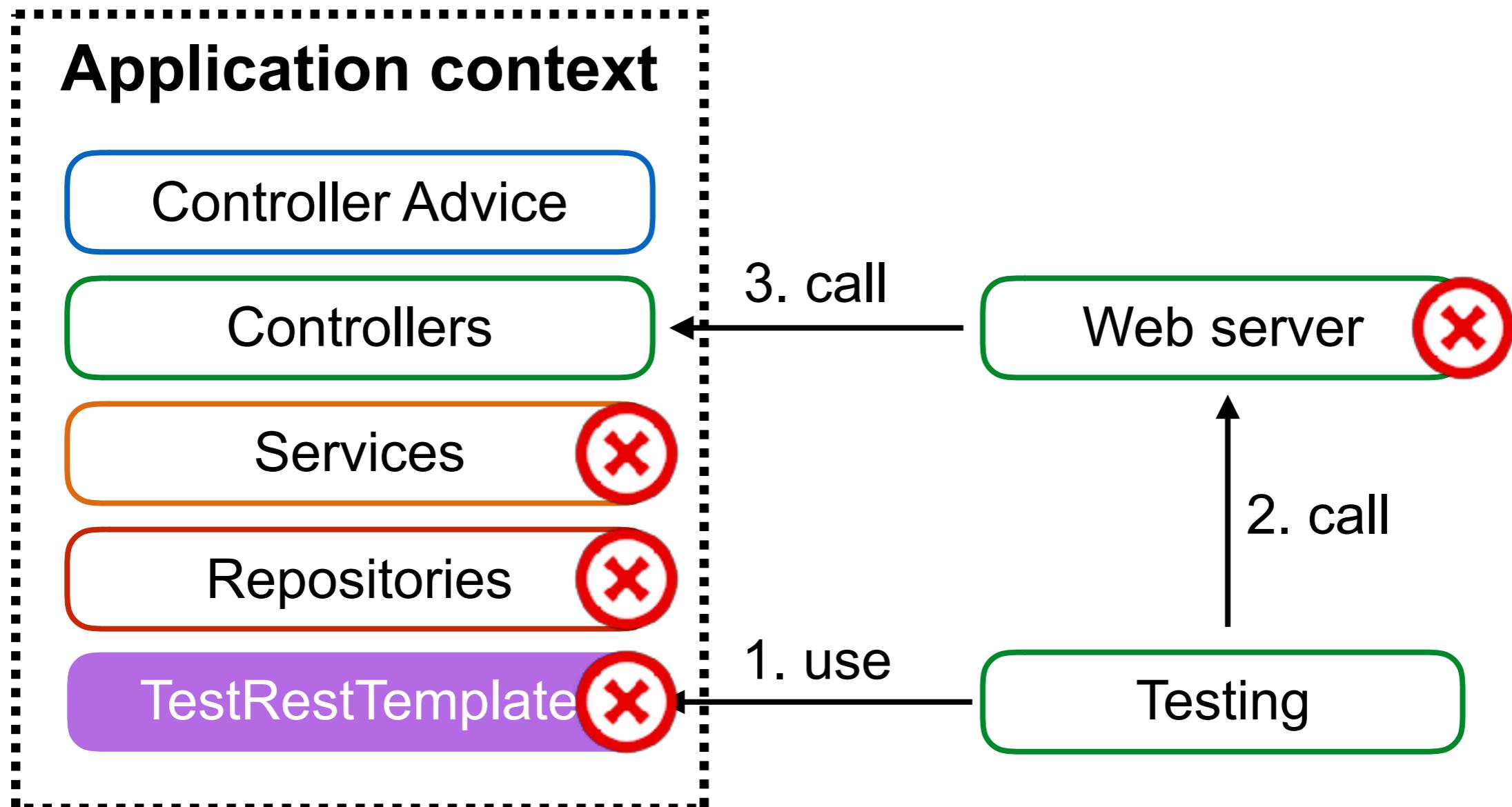
        // Assertion :: Check result with expected result
        assertEquals("Hello, somkiat", actualResult.getMessage());
    }
}
```



2. Slice Testing with MockMVC



2. Slice Testing with MockMVC



MockMvcTest #1

```
@RunWith(SpringRunner.class)
@WebMvcTest(NumberController.class)
public class NumberControllerMockMvcTest {

    @MockBean
    private MyRandom stubRandom;

    @Autowired
    private MockMvc mvc;

    @Test
    public void success() throws Exception {
        NumberControllerResponse expected
            = new NumberControllerResponse("5555");

        // Stub
        given(stubRandom.nextInt(10)).willReturn(5555);
    }
}
```



MockMvcTest #2

```
@RunWith(SpringRunner.class)
@WebMvcTest(NumberController.class)
public class NumberControllerMockMvcTest {

    @MockBean
    private MyRandom stubRandom;

    @Autowired
    private MockMvc mvc;

    @Test
    public void success() throws Exception {
        NumberControllerResponse expected
            = new NumberControllerResponse("5555");

        // Stub
        given(stubRandom.nextInt(10)).willReturn(5555);
```



MockMvcTest #3

Use ObjectMapper to convert JSON to object

```
// Call API HTTP response code = 200
String response =
    this.mvc.perform(get("/number"))
    .andExpect(status().isOk())
    .andReturn()
    .getResponse().getContentAsString();

// Convert JSON message to Object
ObjectMapper mapper = new ObjectMapper();
NumberControllerResponse actual =
    mapper.readValue(response,
        NumberControllerResponse.class);
```



3. Unit testing with Controller



Unit test

Use Test Double

In java, use Mockito library



<http://site.mockito.org/>



Unit testing with Mockito #1

```
@RunWith(MockitoJUnitRunner.class)
public class NumberControllerUnitTest {

    @Mock
    private MyRandom stubRandom;

    @Test
    public void success() throws Exception {
        NumberControllerResponse expected
            = new NumberControllerResponse("5555");

        // Stub
        given(stubRandom.nextInt(10)).willReturn(5555);
    }
}
```



Unit testing with Mockito #2

```
@RunWith(MockitoJUnitRunner.class)
public class NumberControllerUnitTest {

    @Mock
    private MyRandom stubRandom;

    @Test
    public void success() throws Exception {
        NumberControllerResponse expected
            = new NumberControllerResponse("5555");

        // Stub
        given(stubRandom.nextInt(10)).willReturn(5555);
    }
}
```



Unit testing with Mockito #3

```
@Test
public void success() throws Exception {
    NumberControllerResponse expected
        = new NumberControllerResponse("5555");

    // Stub
    given(stubRandom.nextInt(10)).willReturn(5555);

    // Call
    NumberController controller = new NumberController(stubRandom);
    NumberControllerResponse actual = controller.randomNumber();

    // Assert
    assertEquals("5555", actual.getValue());
    assertEquals(expected, actual);
}
```



Error handling



Error handling

```
@Service
public class UserService {

    private AccountRepository accountRepository;

    @Autowired
    public UserService(AccountRepository accountRepository) {
        this.accountRepository = accountRepository;
    }

    public Account getAccount(int id) {
        Optional<Account> account = accountRepository.findById(id);
        if(account.isPresent()) {
            return account.get();
        }
        throw new MyAccountNotFoundException(
            String.format("Account id=[%d] not found", id));
    }
}
```



MyAccountNotFoundException

```
public class MyAccountNotFoundException  
    extends RuntimeException {
```

```
    public MyAccountNotFoundException(String message) {  
        super(message);  
    }
```

```
}
```



Response Status

404 = Not Found

Status	Description
400	Request body doesn't meet API spec
401	Authentication/Authorization fail
403	User can't perform the operation
404	Resource does not exist
405	Unsupported operation
500	Error on server



Handling error in Spring Boot

```
@RestControllerAdvice  
public class AccountControllerHandler {  
  
    @ExceptionHandler(MyAccountNotFoundException.class)  
    public ResponseEntity<ExceptionResponse> accountNotFound(  
        MyAccountNotFoundException exception) {  
  
        ExceptionResponse response =  
            new ExceptionResponse(exception.getMessage(),  
                "More detail");  
  
        return new ResponseEntity<ExceptionResponse>(response,  
            HttpStatus.NOT_FOUND);  
    }  
}
```

1



Handling error in Spring Boot

```
@RestControllerAdvice  
public class AccountControllerHandler {  
  
    @ExceptionHandler(MyAccountNotFoundException.class)  
    public ResponseEntity<ExceptionResponse> accountNotFound(  
        MyAccountNotFoundException exception) {  
  
        ExceptionResponse response =  
            new ExceptionResponse(exception.getMessage(),  
                "More detail");  
  
        return new ResponseEntity<ExceptionResponse>(response,  
            HttpStatus.NOT_FOUND);  
    }  
}
```

2



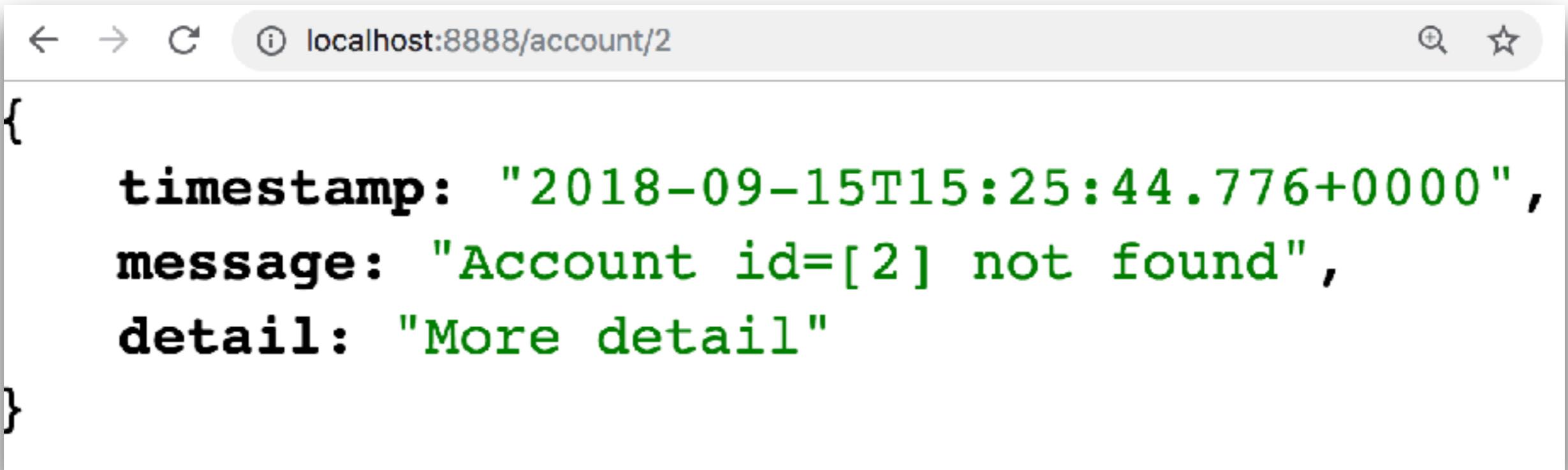
ExceptionResponse

Response format of error

```
public class ExceptionResponse{  
  
    private Date timestamp = new Date();  
    private String message;  
    private String detail;  
  
    public ExceptionResponse(String message, String detail) {  
        this.message = message;  
        this.detail = detail;  
    }  
}
```



Result of API



A screenshot of a web browser window. The address bar shows the URL `localhost:8888/account/2`. The main content area displays a JSON response:

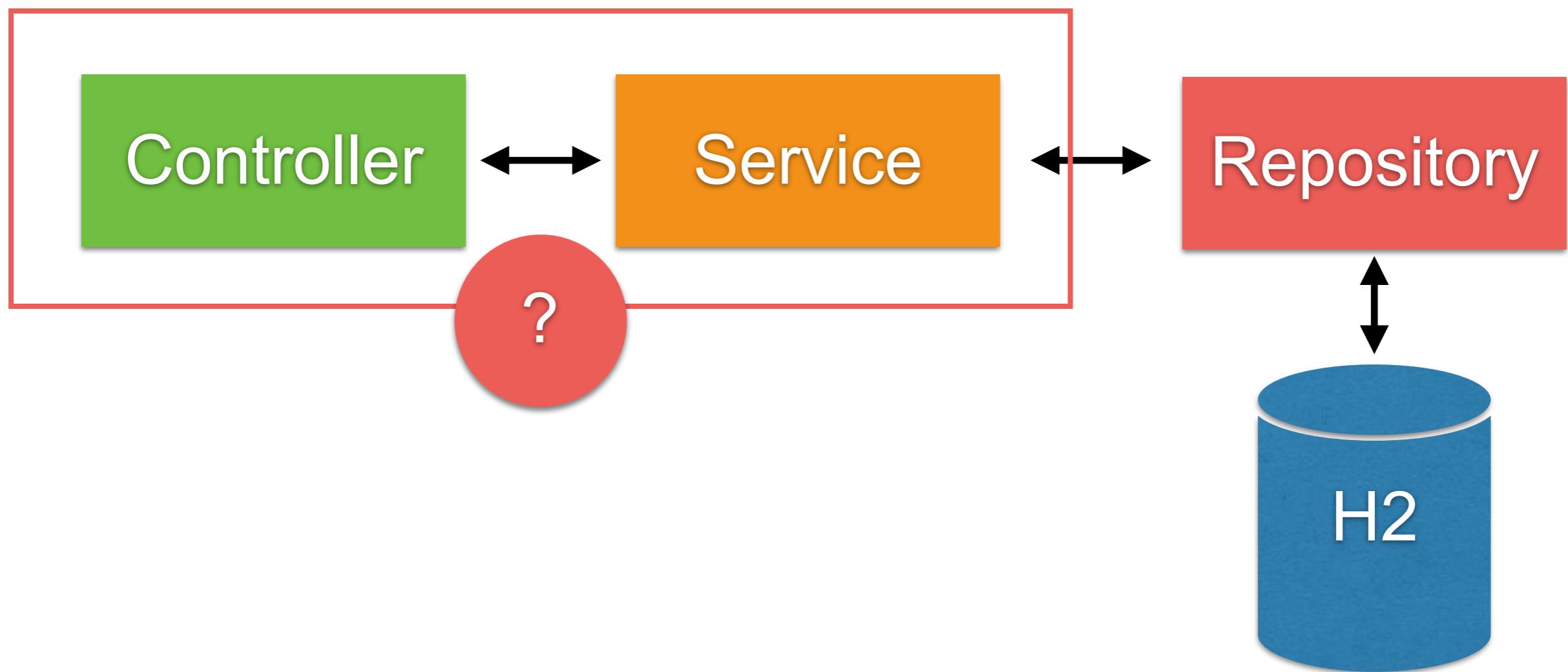
```
{  
  timestamp: "2018-09-15T15:25:44.776+0000",  
  message: "Account id=[ 2 ] not found",  
  detail: "More detail"  
}
```



How to test ?



How to test with Error/Exception ?



Testing with WebMvcTest and MockMvc

Try to check data in response

```
@Test  
public void getByIdWithNotFoundAccount() throws Exception {  
    // Stub  
    given(userService.getAccount(2))  
        .willThrow(new MyAccountNotFoundException("Not found"));  
  
    mockMvc.perform(  
        get("/account/2")  
            .accept(MediaType.APPLICATION_JSON))  
        .andExpect(status().isNotFound())  
        .andExpect(content().contentType(MediaType.APPLICATION_JSON_UTF8))  
        .andExpect(jsonPath("$.message", is("Not found")));  
}
```

1



Testing with WebMvcTest and MockMvc

Try to check data in response

```
@Test  
public void getByIdWithNotFoundAccount() throws Exception {  
    // Stub  
    given(userService.getAccount(2))  
        .willThrow(new MyAccountNotFoundException("Not found"));  
  
    mockMvc.perform(  
        get("/account/2")  
            .accept(MediaType.APPLICATION_JSON))  
        .andExpect(status().isNotFound())  
        .andExpect(content().contentType(MediaType.APPLICATION_JSON_UTF8))  
        .andExpect(jsonPath("$.message", is("Not found")));  
}
```

2



Testing with Service

Try to check exception

```
@RunWith(MockitoJUnitRunner.class)
public class UserServiceWithExceptionTest {

    @Mock
    private AccountRepository accountRepository;

    @Test(expected = MyAccountNotFoundException.class)
    public void getAccountWithException() {
        // Stub
        given(accountRepository.findById(1))
            .willThrow(new MyAccountNotFoundException("Not found"));

        UserService userService = new UserService(accountRepository);
        Account actualAccount = userService.getAccount(1);
    }
}
```



Compile with testing

\$mvnw clean test

```
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO]
```



Code coverage



"Code coverage can show the high risk areas in a program, but never the risk-free."

Paul Reilly, 2018, Kotlin TDD with Code Coverage



Code coverage

A tool to measure how much of your code is covered by tests that break down into classes, methods and lines.



Code coverage

But 100% of code coverage **does not mean** that
your code is 100% correct



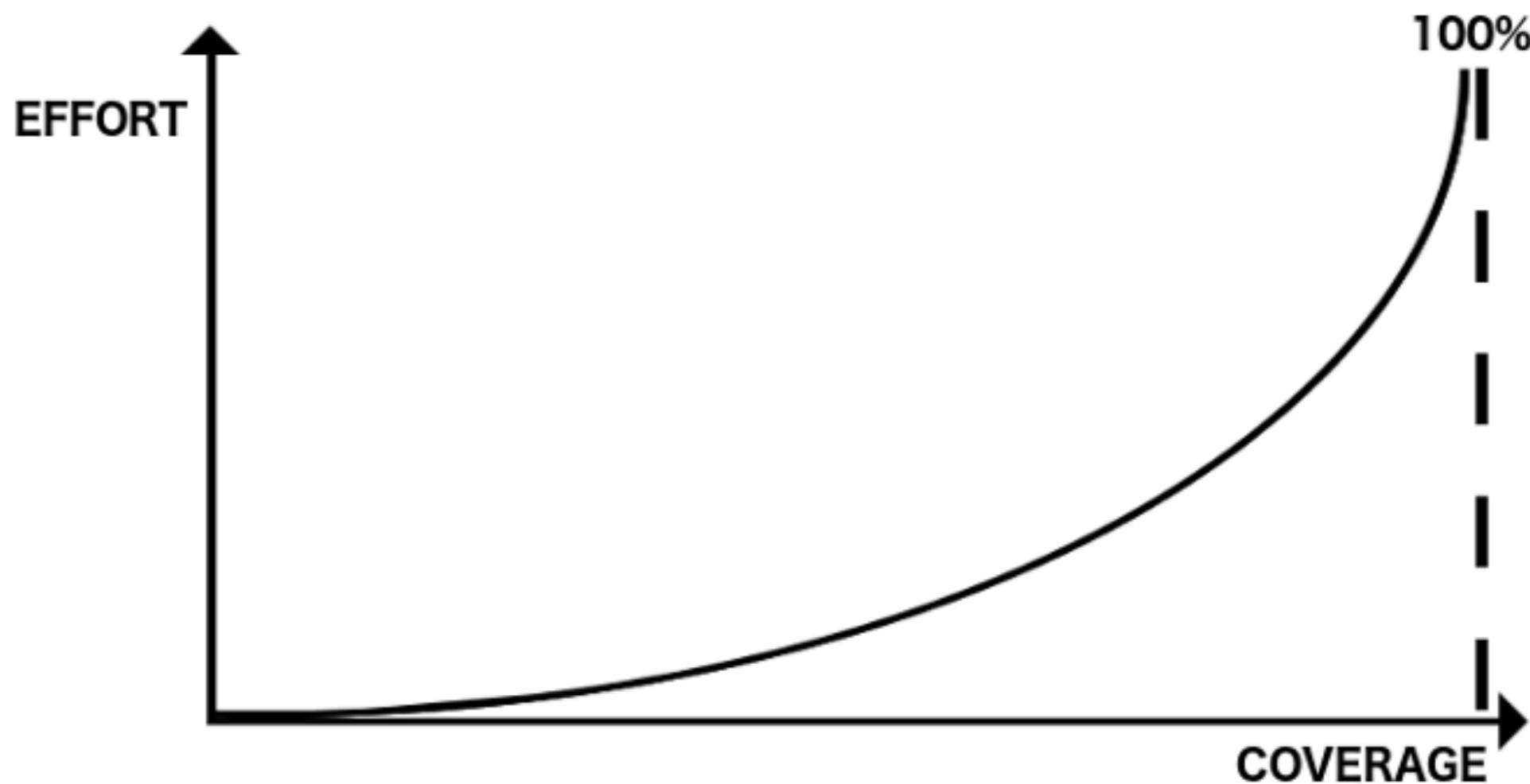
Code coverage

Powerful tool to improve the quality of your code

Code coverage != quality of tests



Code coverage 100% ?



% of Code/Test coverage



Code coverage with Java

Cobertura
Jacoco

<http://bit.ly/2DIlsDeX>



Run test again

\$mvnw clean package

Cobertura Report generation was successful.

Cobertura 2.1.1 - GNU GPL License (NO WARRANTY) - See COPYRIGHT file

Cobertura: Loaded information on 3 classes.

time: 125ms

Cobertura Report generation was successful.

BUILD SUCCESS



Coverage report

open target/site/cobertura/index.html

Packages

All
[hello](#)
[hello.controller](#)
[hello.domain](#)

Coverage Report - All Packages

Package	# Classes	Line Coverage	Branch Coverage	Complexity
All Packages	3	63% 7/11	N/A N/A	1
hello	1	33% 1/3	N/A N/A	1
hello.controller	1	100% 2/2	N/A N/A	1
hello.domain	1	66% 4/6	N/A N/A	1

Report generated by [Cobertura](#) 2.1.1 on 3/6/18 12:40 AM.

All Packages

Classes

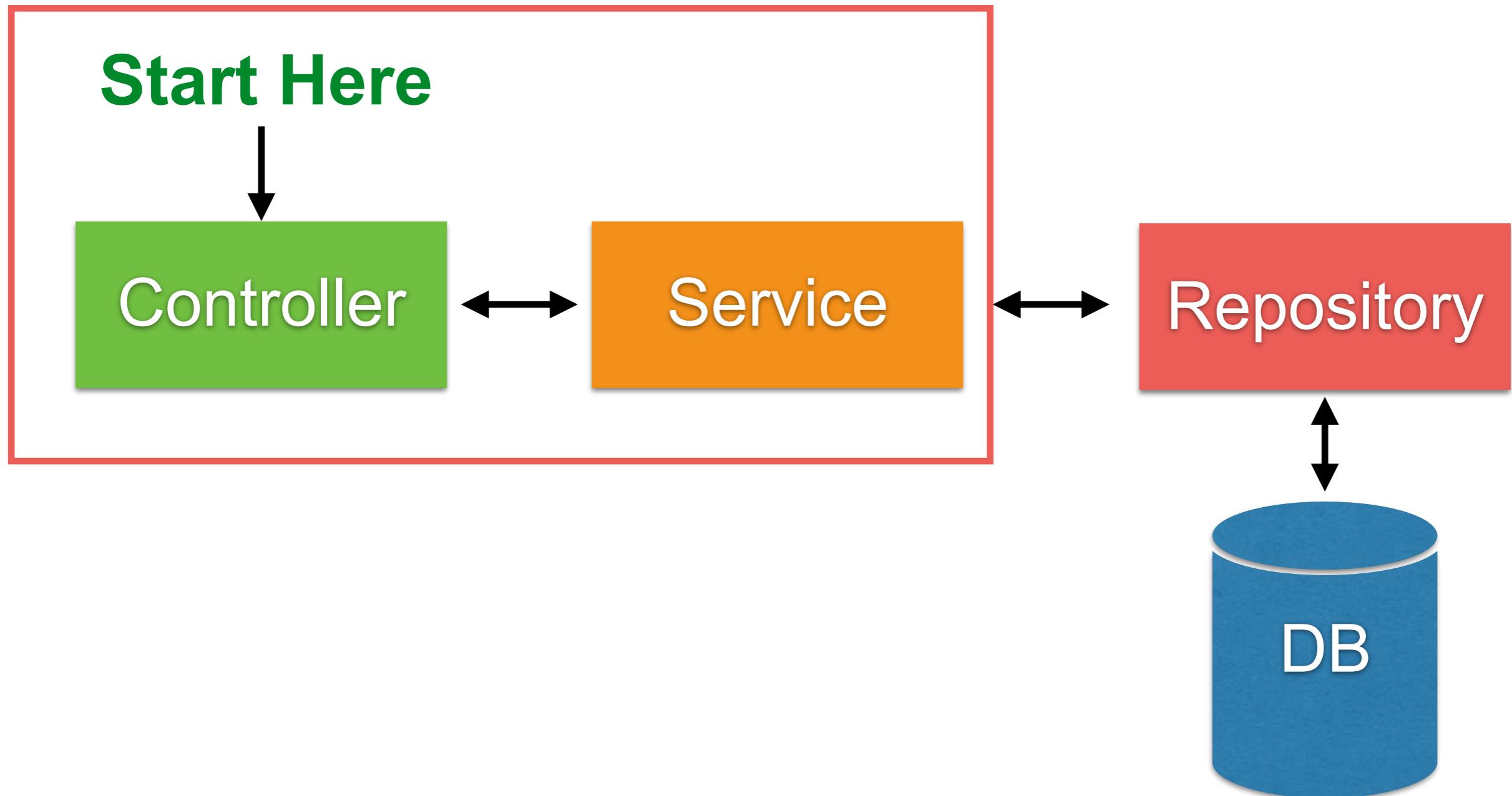
[Hello \(66%\)](#)
[HelloApplication \(33%\)](#)
[HelloController \(100%\)](#)



Move business logic to service

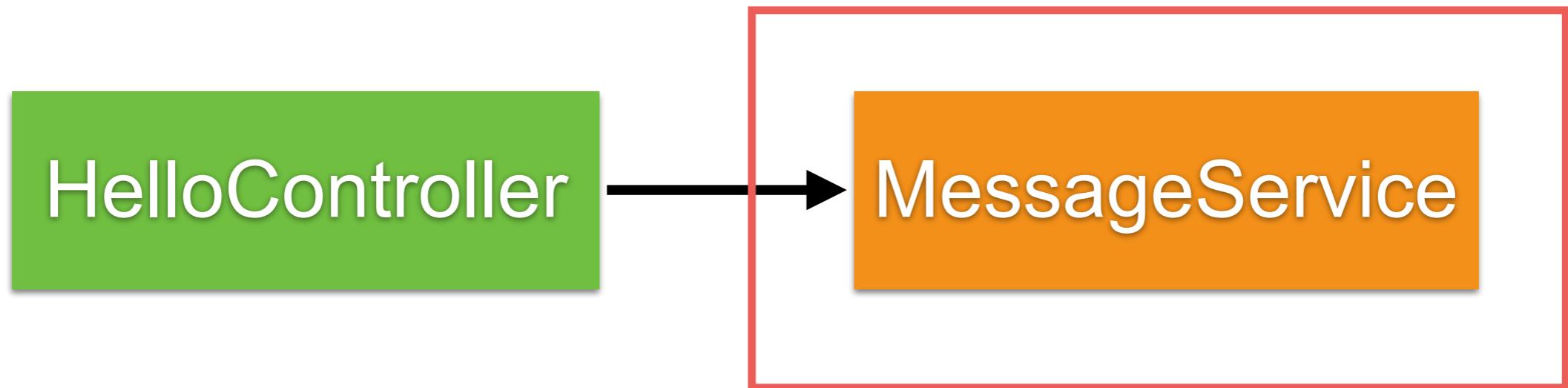


Working with service



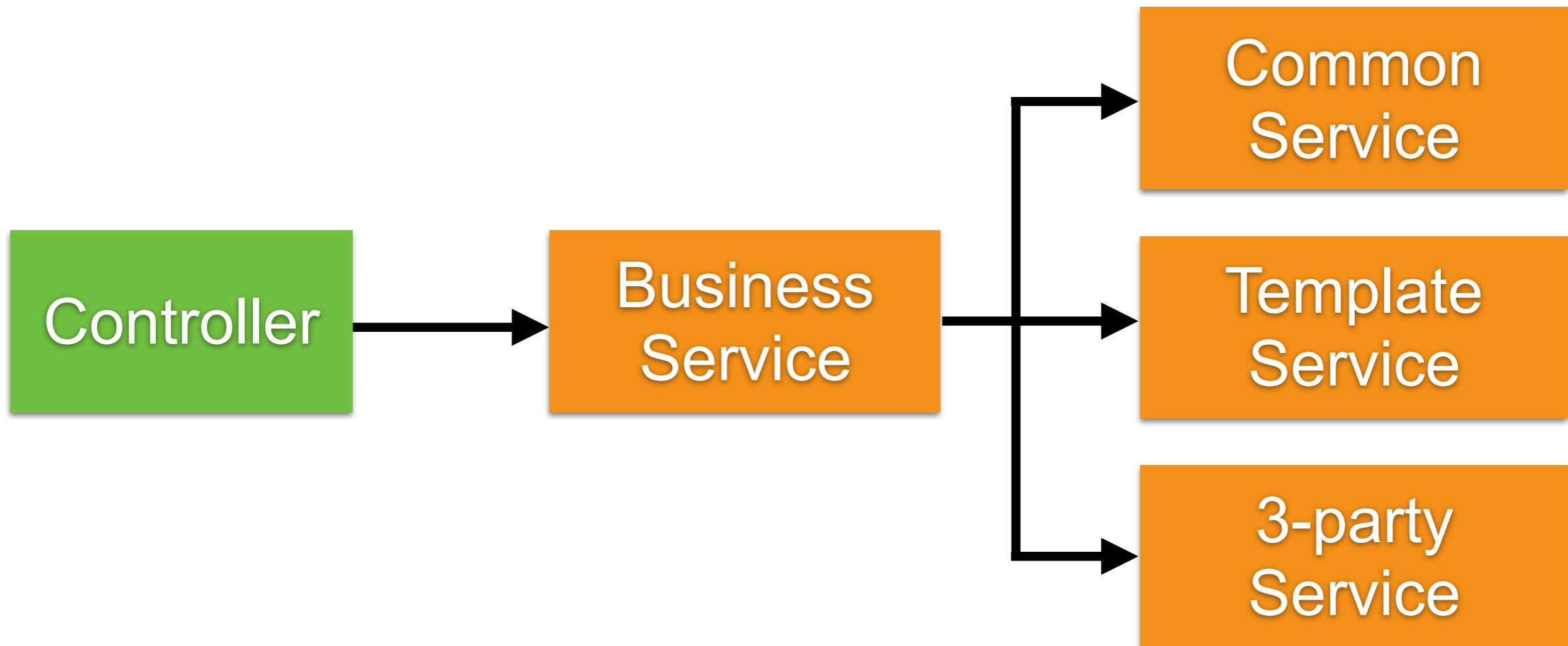
Move business logic to service

Service class or interface ?

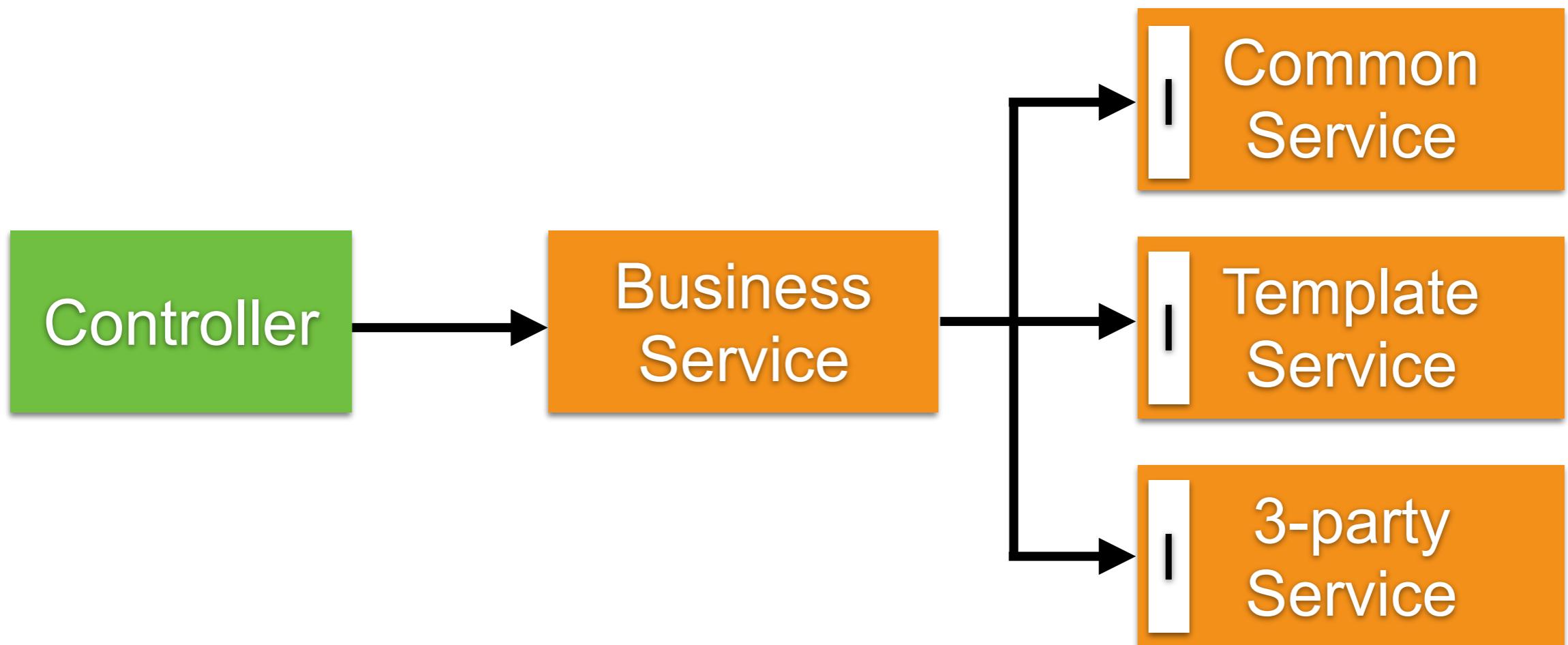


Types of service

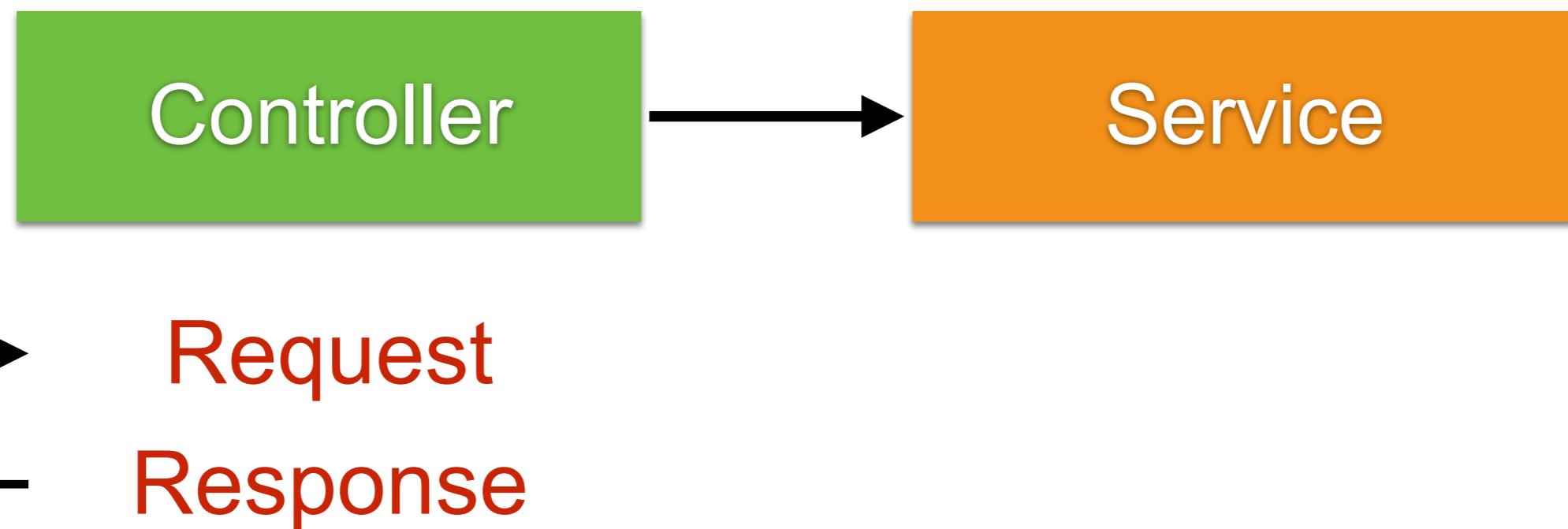
Business services (not reuse)
Common/Template/3-party services



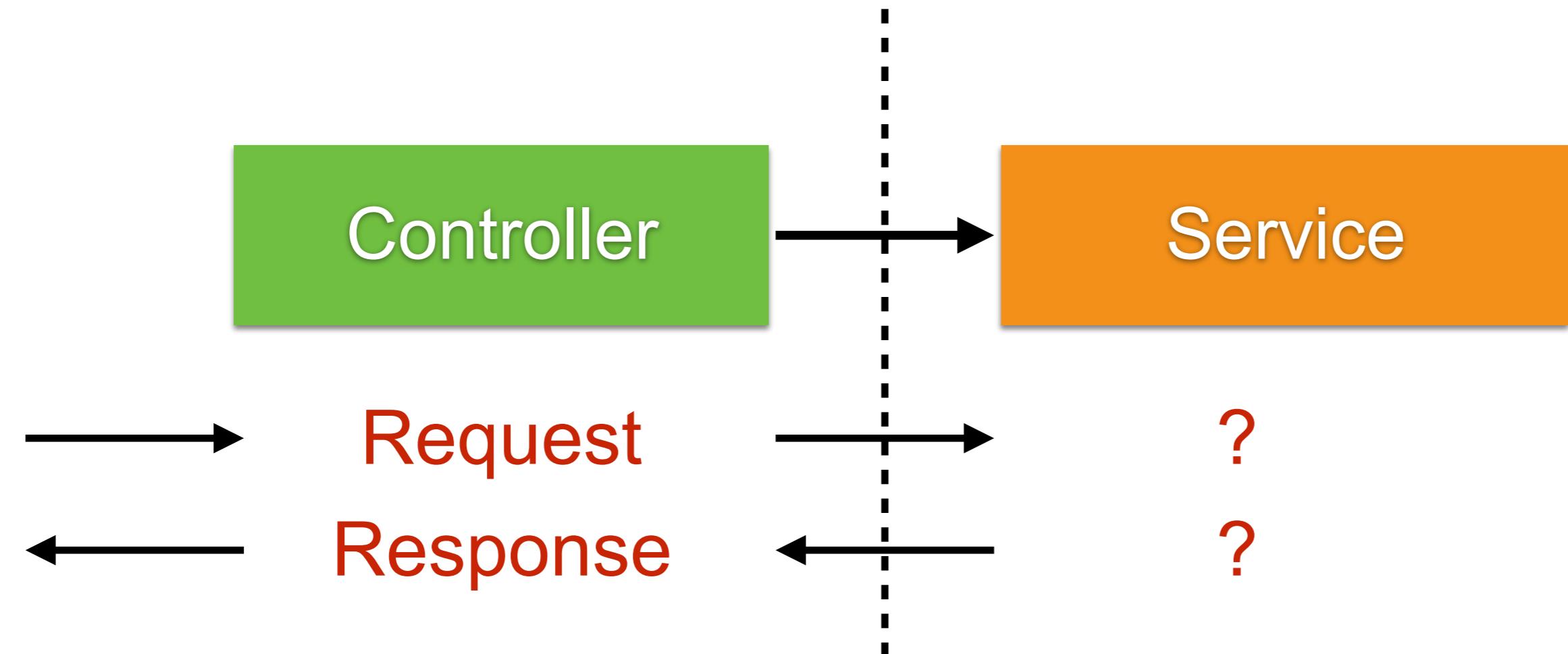
Interface for common service



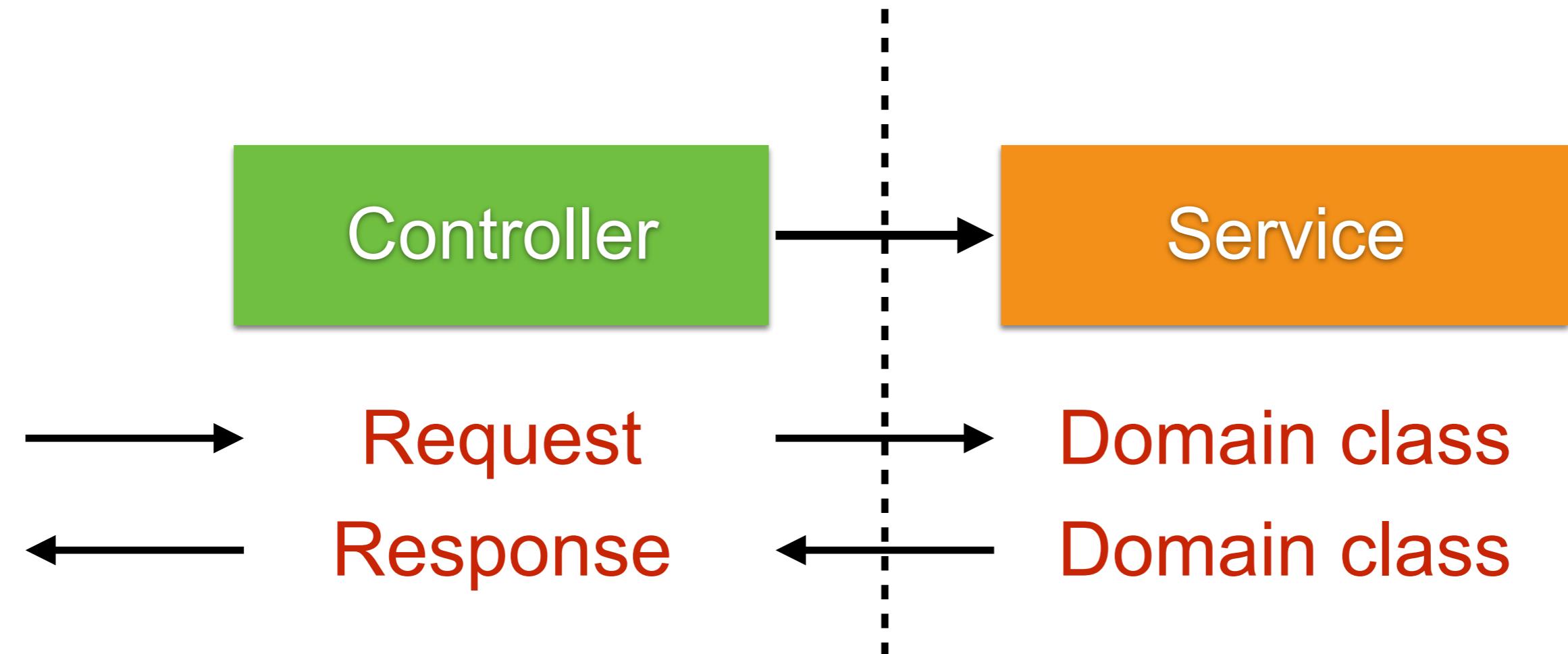
Data Model for service ?



Data Model ?

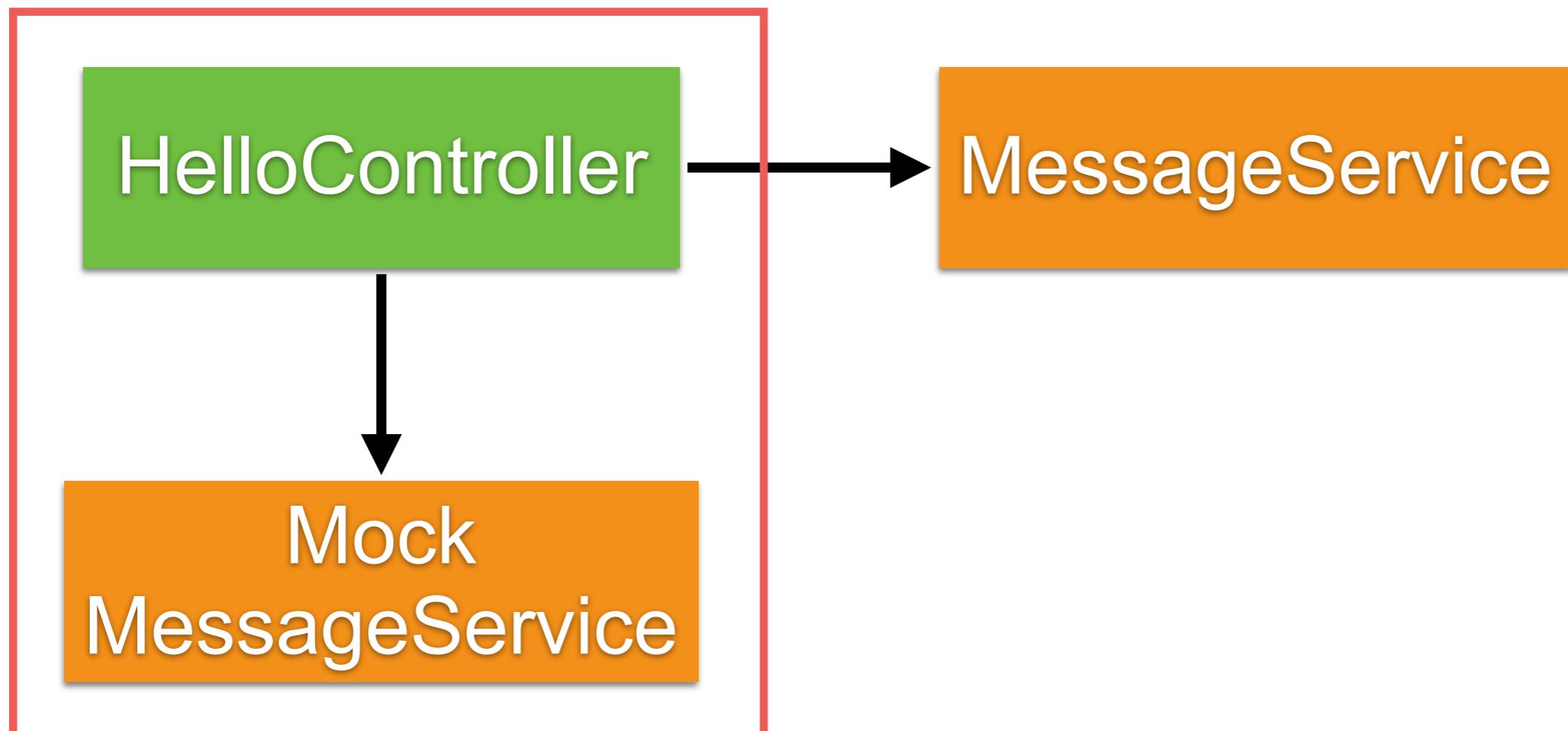


Data Model ?



Testing controller with service

Try to mocking service with Mockito



Run all tests !!

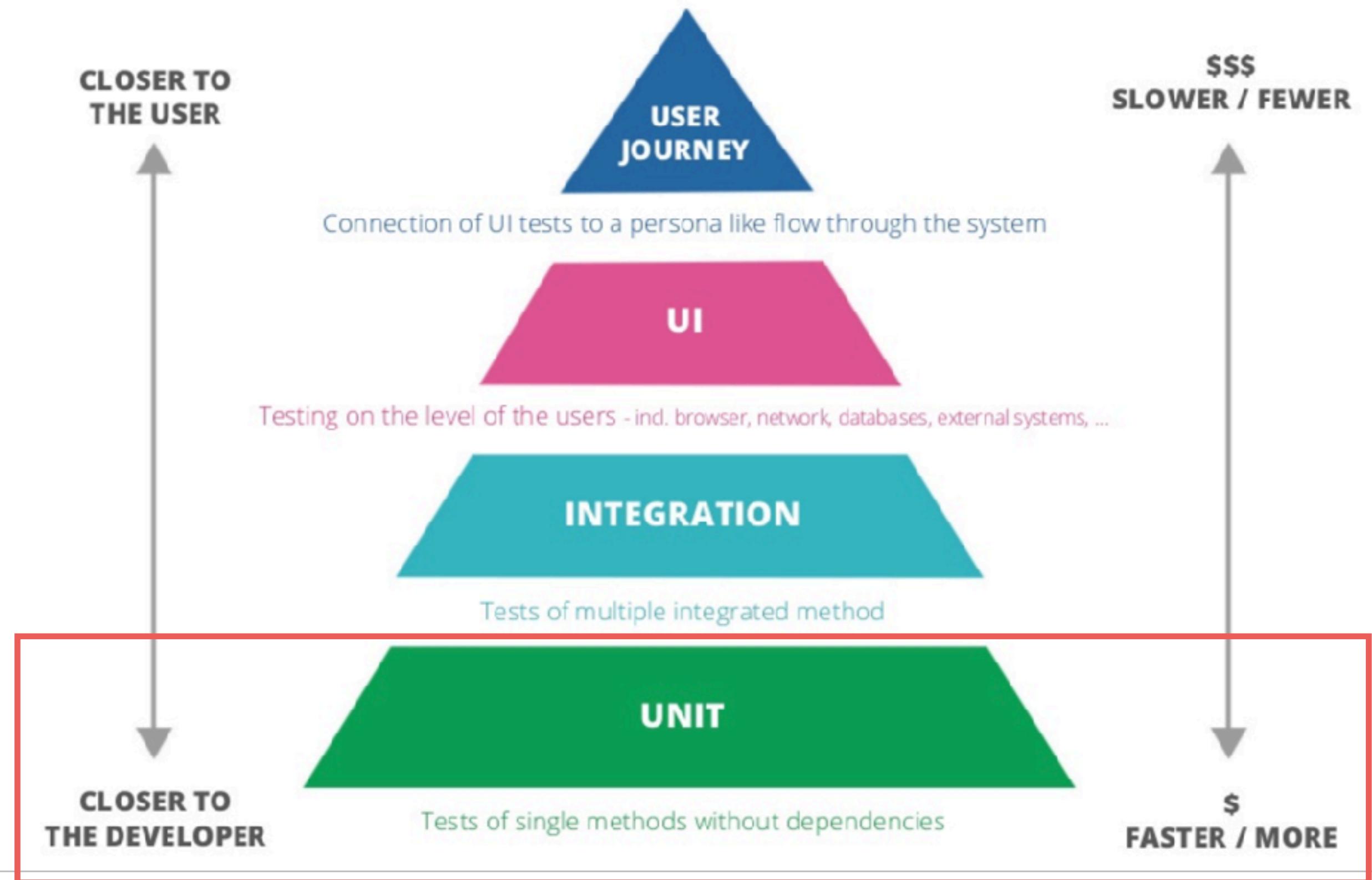
\$mvnw clean test

```
[INFO]
[INFO] Results:
[INFO]
[WARNING] Tests run: 10, Failures: 0, Errors: 0,
[INFO]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 18.299 s
[INFO] Finished at: 2018-08-20T23:36:31+07:00
[INFO] -----
```

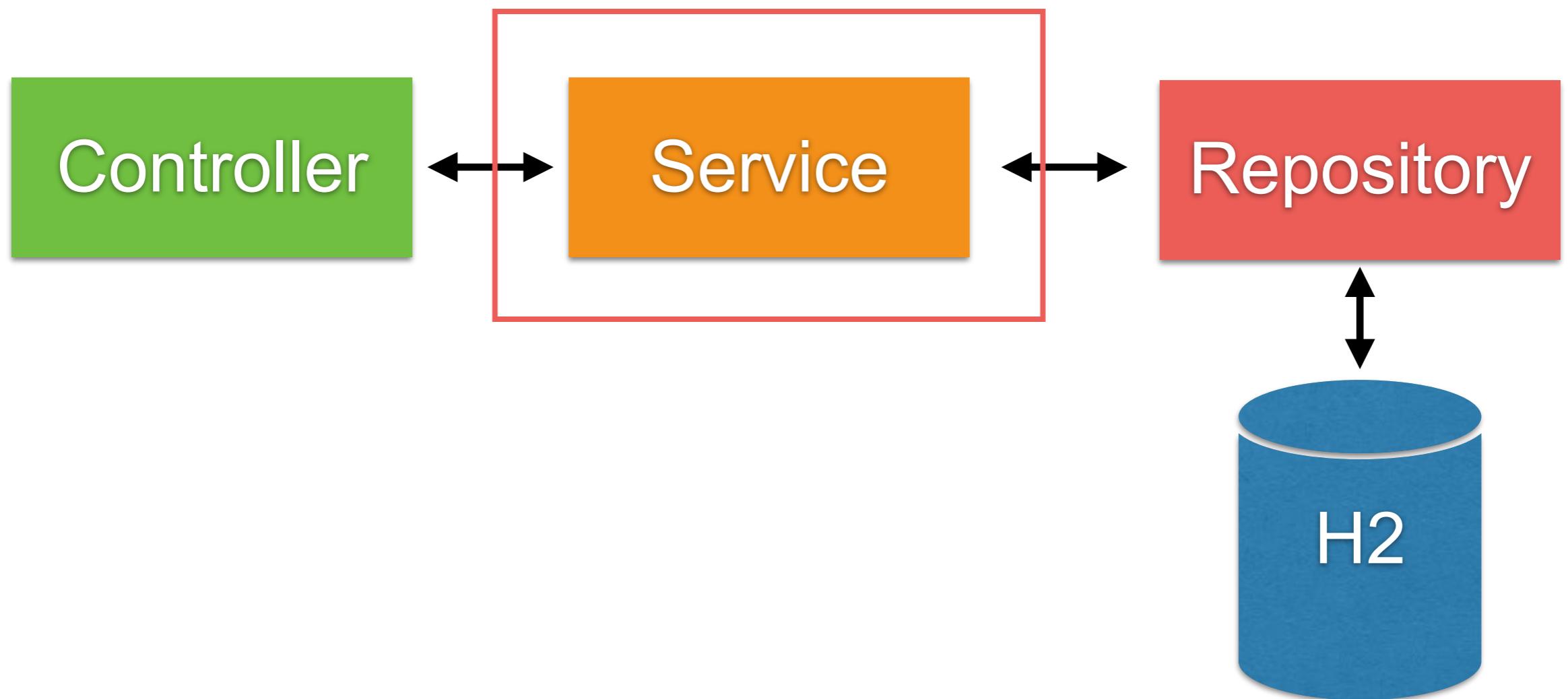


How to improve the speed of testing ?





Service Testing ?



Service Testing

```
@RunWith(MockitoJUnitRunner.class)
public class UserServiceTest {

    @Mock
    private AccountRepository accountRepository;

    @Test
    public void getAccount() {
        // Stub
        Account account = new Account();
        account.setUserName("user");
        account.setPassword("pass");
        account.setSalary(1000);
        given(accountRepository.findById(1))
            .willReturn(Optional.of(account));

        UserService userService = new UserService(accountRepository);
        Account actualAccount = userService.getAccount(1);
        assertNotNull(actualAccount);
    }
}
```

1



Service Testing

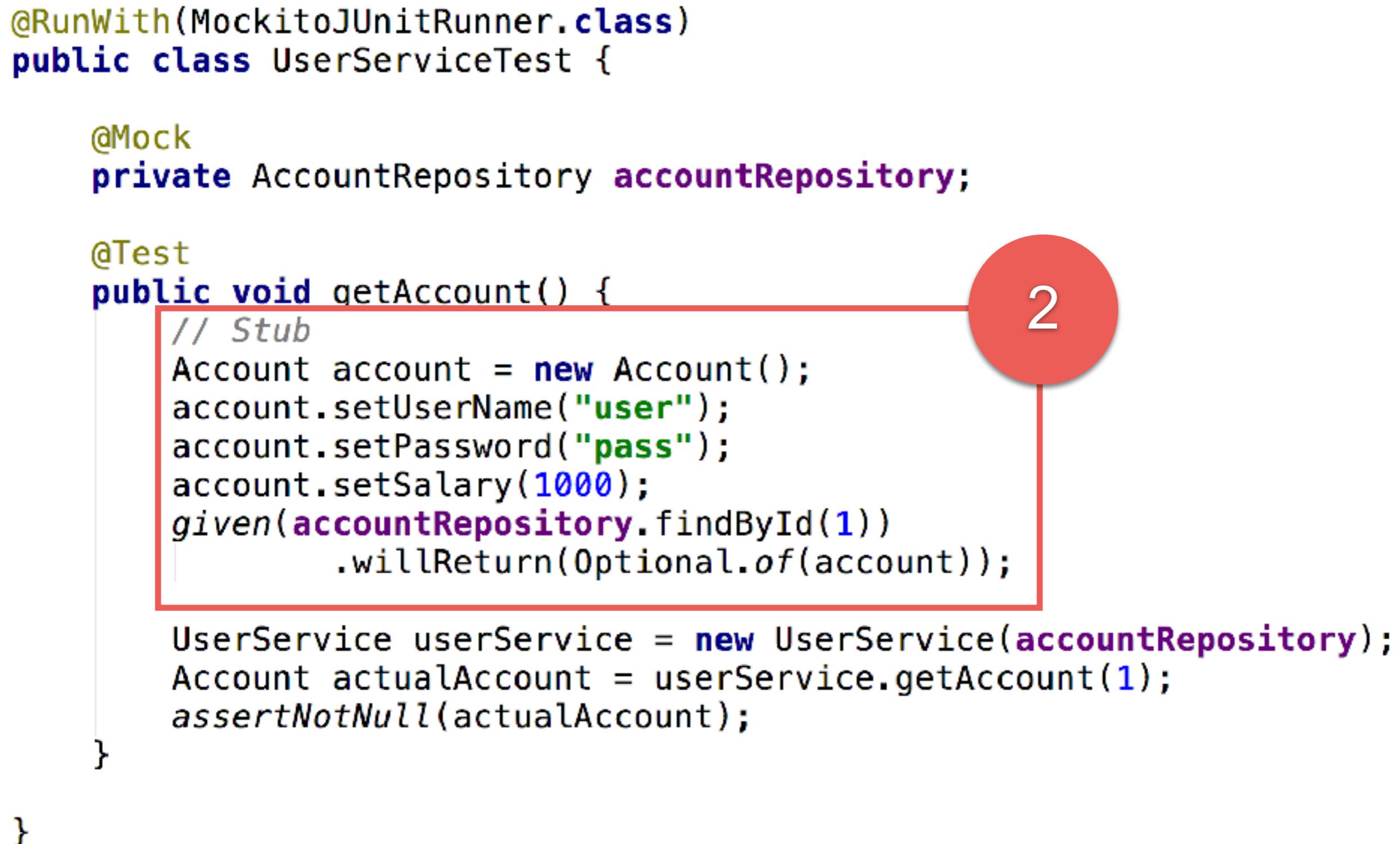
```
@RunWith(MockitoJUnitRunner.class)
public class UserServiceTest {

    @Mock
    private AccountRepository accountRepository;

    @Test
    public void getAccount() {
        // Stub
        Account account = new Account();
        account.setUserName("user");
        account.setPassword("pass");
        account.setSalary(1000);
        given(accountRepository.findById(1))
            .willReturn(Optional.of(account));
    }

    UserService userService = new UserService(accountRepository);
    Account actualAccount = userService.getAccount(1);
    assertNotNull(actualAccount);
}

}
```



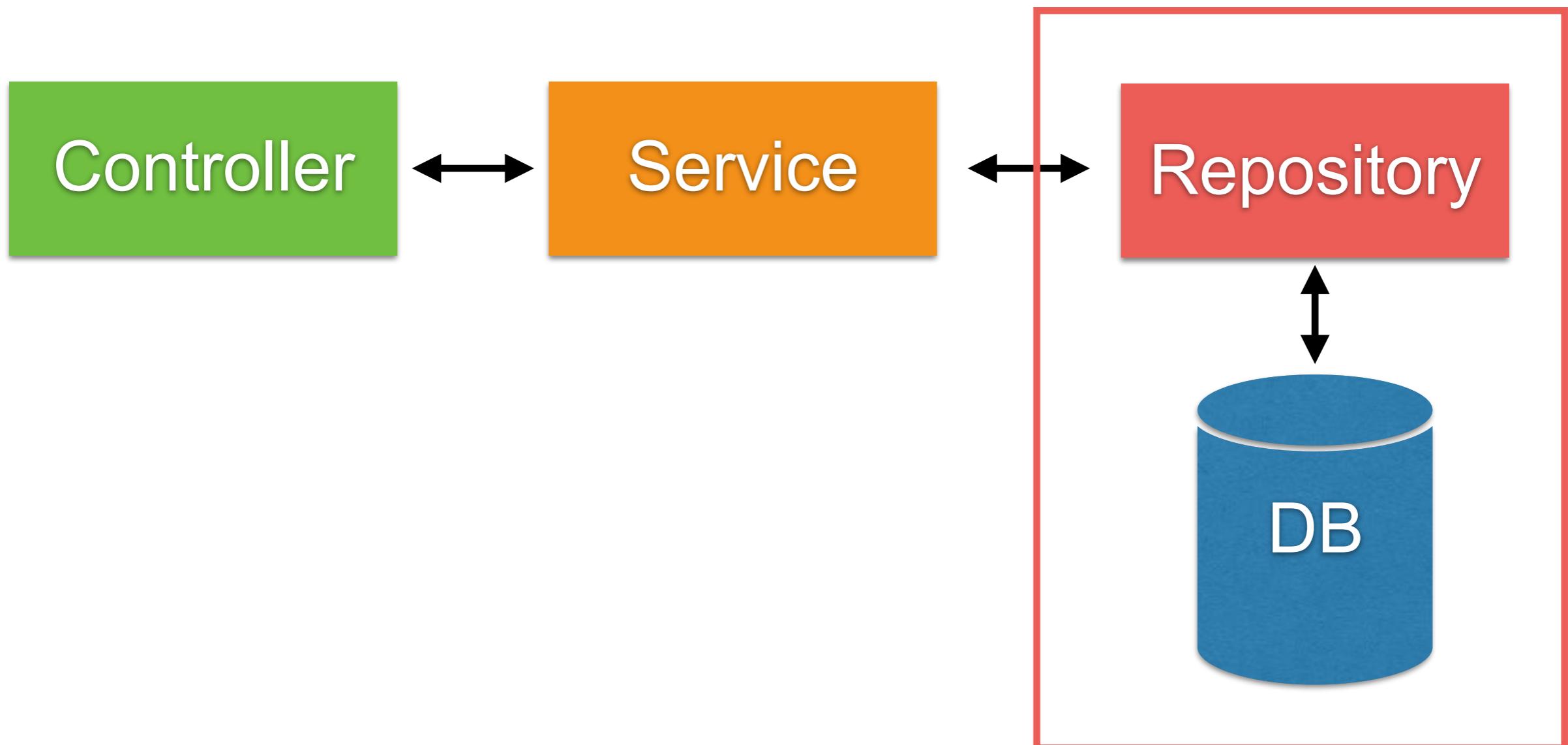
The diagram shows a red rectangular box highlighting the stubbing code within the `getAccount()` method. A red circle containing the number "2" is positioned above the right edge of this red box.



Working with Repository

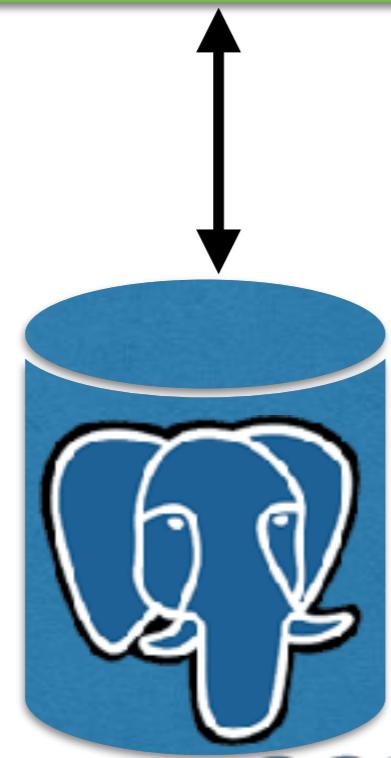


Working with repository



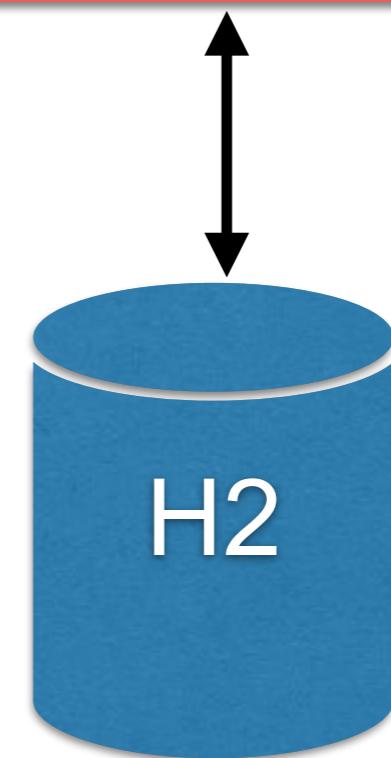
Working with Database ?

Production



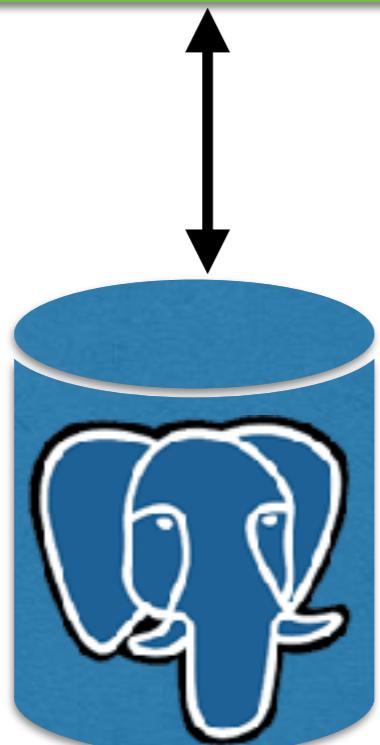
PostgreSQ_L

Testing



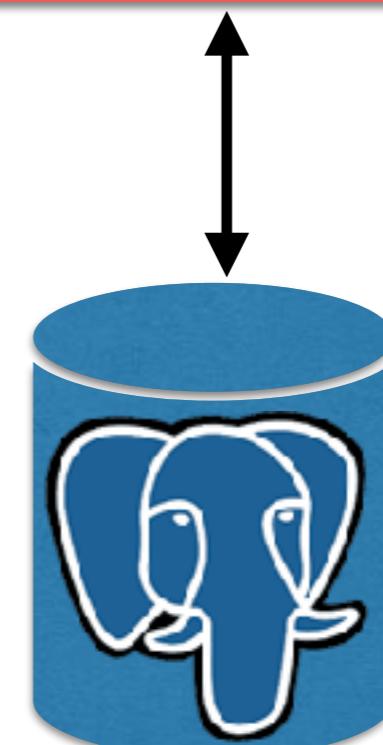
Working with Database ?

Production



PostgreSQL

Testing

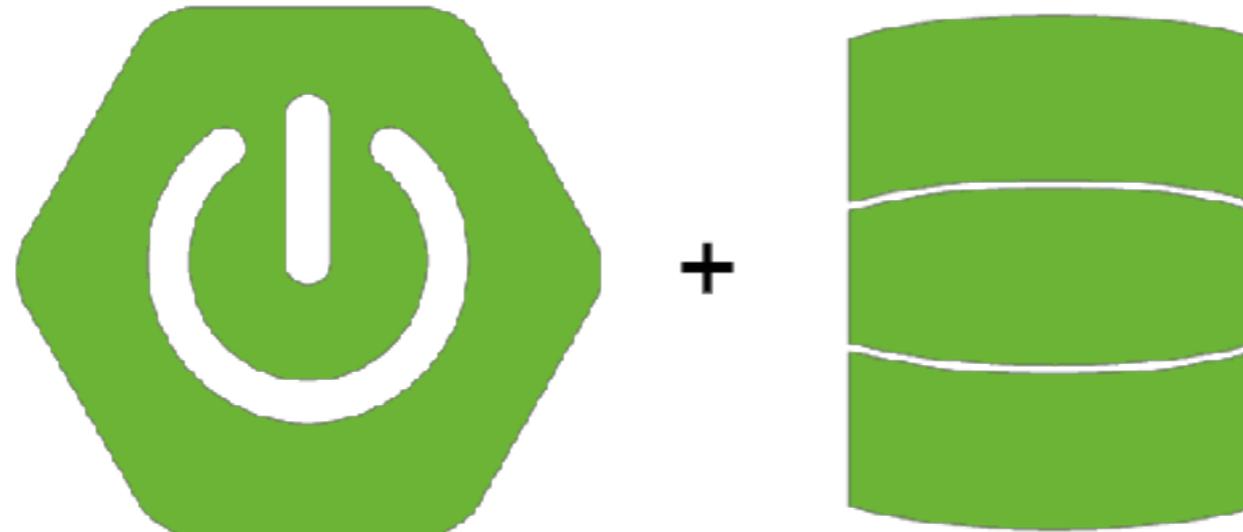


PostgreSQL



Working with repository

We're using Spring Data



<https://spring.io/projects/spring-data>



Spring Data

JDBC

JPA

MongoDB

Redis

more ...

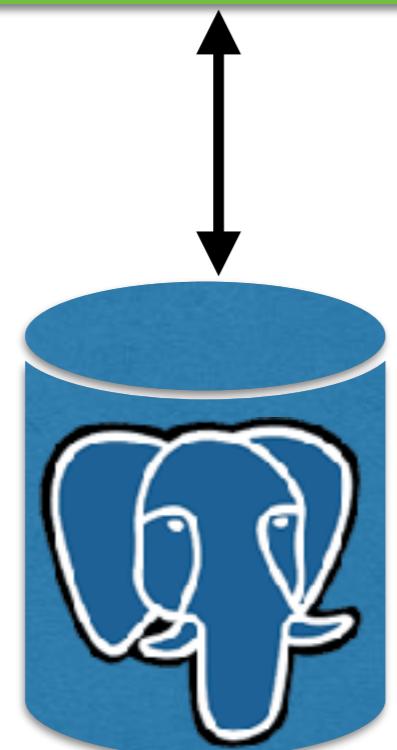


Working with Spring Data JPA



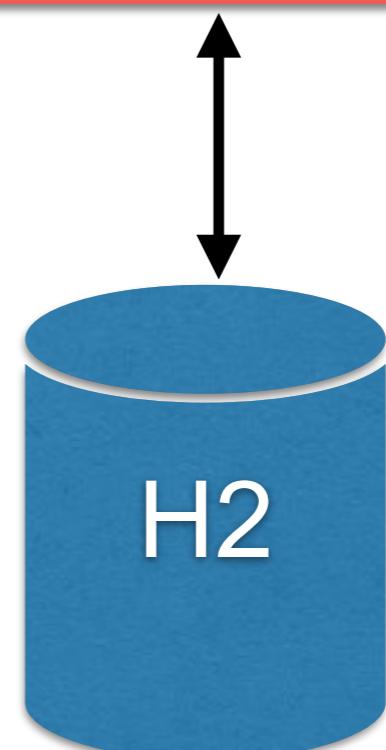
Working with Database

Production



Postgre**SQL**

Testing



Modify pom.xml

Add library of Spring Data JPA, PostgreSQL, H2

The screenshot shows a web-based interface for selecting dependencies. On the left, under 'Dependencies', there is a search bar with placeholder text: 'Search dependencies to add' and 'Web, Security, JPA, Actuator, Devtools...'. To the right, under 'Dependencies selected', three items are listed in boxes: 'JPA [SQL]' (selected), 'PostgreSQL [SQL]', and 'H2 [SQL]'. Each box contains a brief description of the dependency's purpose.

Dependencies selected
JPA [SQL] Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate
PostgreSQL [SQL] PostgreSQL JDBC driver
H2 [SQL] H2 database (with embedded support)

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



Modify pom.xml

H2 for testing

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

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>test</scope>
</dependency>
```

```
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <scope>runtime</scope>
</dependency>
```



Modify pom.xml

PostgreSQL for production

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

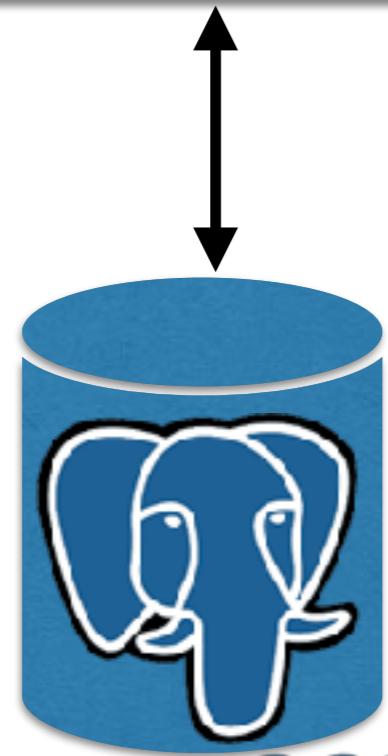
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <scope>runtime</scope>
</dependency>
```



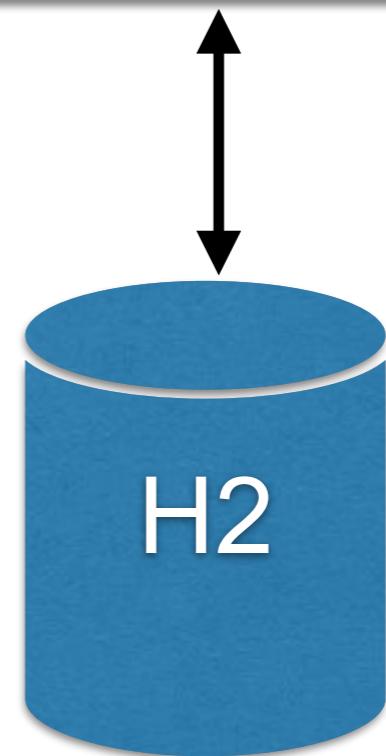
Start in testing scope

Production

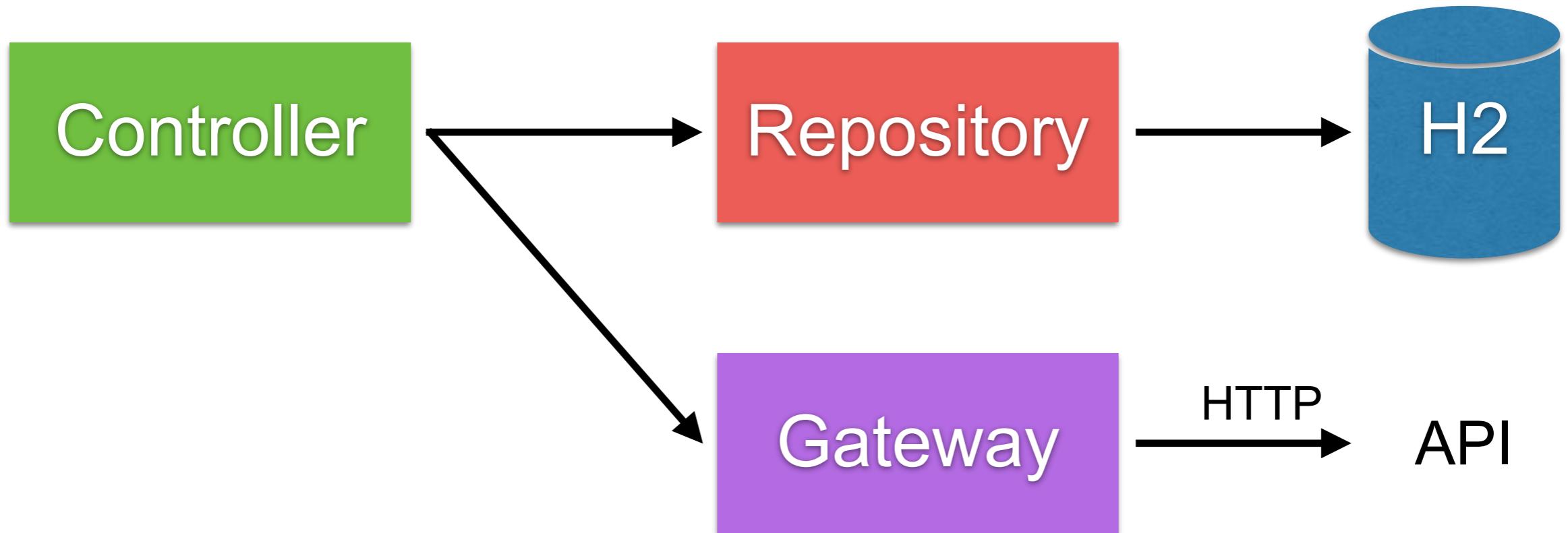


PostgreSQL

Testing

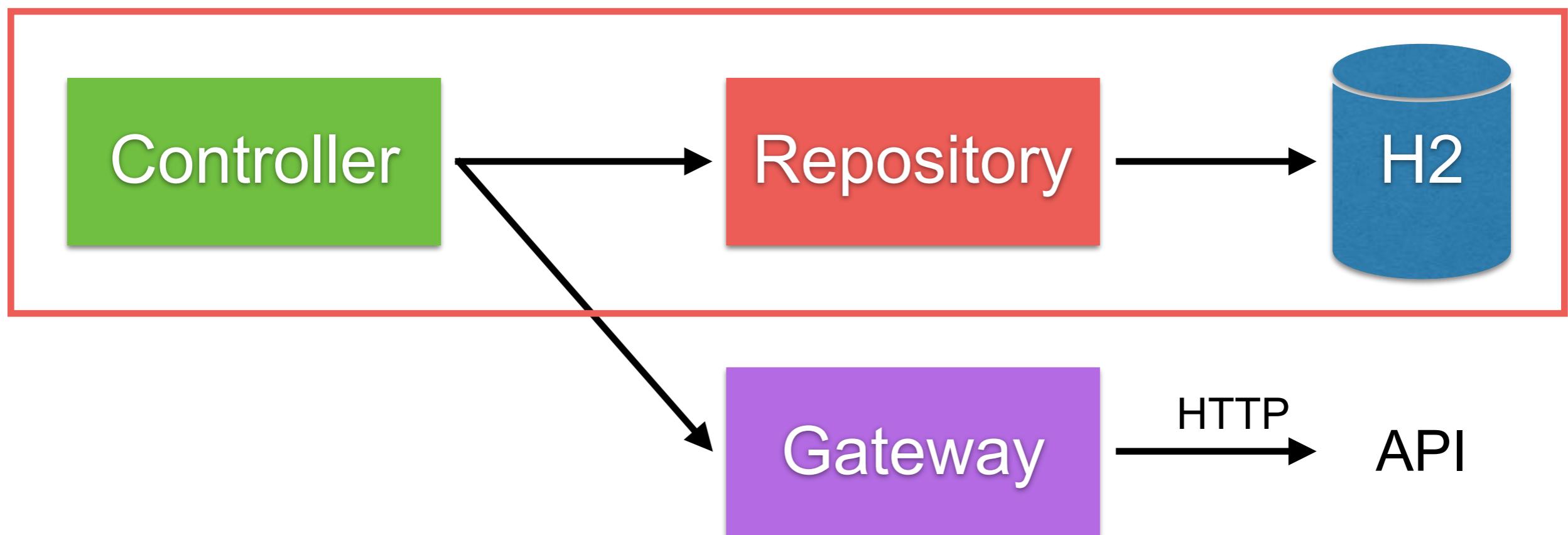


Use cases



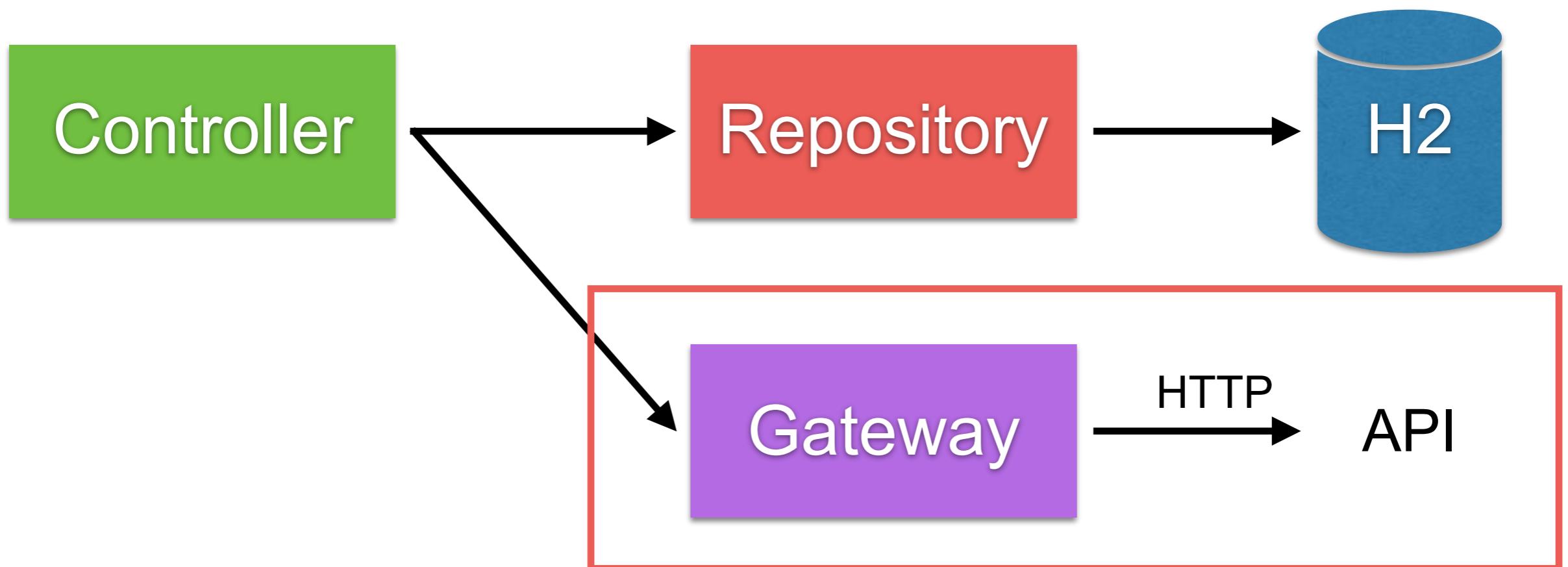
Use case 1

Working with repository



Use case 2

Working with API



Use case 1



Use case 1

Working with repository

3. HelloController



2. PersonRepository



Controller

Repository

H2



4. PersonResponse

1. Person



1. Create Entity class

In package person

```
@Entity  
public class Person {  
  
    @Id  
    @GeneratedValue(strategy = GenerationType.AUTO)  
    private long id;  
    private String firstName;  
    private String lastName;  
  
    public Person() {  
    }  
}
```



2. Create repository with JPA

PersonRepository.java

```
import java.util.Optional;  
  
import org.springframework.data.repository.CrudRepository;  
  
public interface PersonRepository  
    extends CrudRepository<Person, Long> {  
  
    Optional<Person> findByLastName(String lastName);  
  
}
```



2. Create repository with JPA

PersonRepository.java

```
import java.util.Optional;  
  
import org.springframework.data.repository.CrudRepository;  
  
public interface PersonRepository  
    extends CrudRepository<Person, Long> {  
  
    Optional<Person> findByLastName(String lastName);  
}
```

*SELECT * FROM Person WHERE LastName=?*

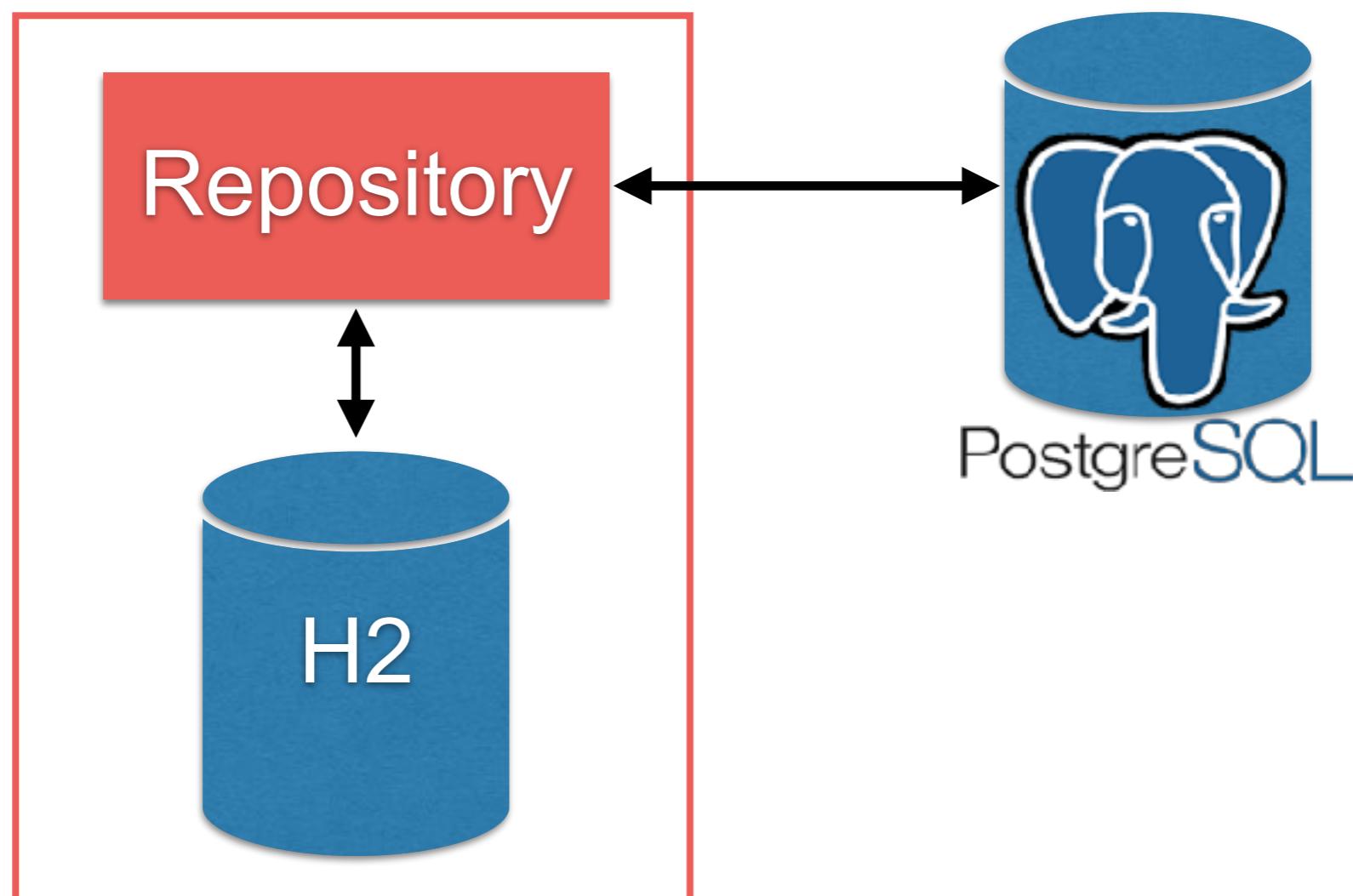


How to testing repository ?



Repository Testing

Using `@DataJpaTest` (slice testing)



Working with In-memory database



Repository Testing #1

Setup test with @DataJpaTest

```
@RunWith(SpringRunner.class)
@DataJpaTest
public class PersonRepositoryTest {

    @Autowired
    private PersonRepository repository;

    @After
    public void tearDown() throws Exception {
        repository.deleteAll();
    }

}
```



Repository Testing #2

Auto wired repository for testing

```
@RunWith(SpringRunner.class)
@DataJpaTest
public class PersonRepositoryTest {

    @Autowired
    private PersonRepository repository;

    @After
    public void tearDown() throws Exception {
        repository.deleteAll();
    }

}
```



Repository Testing #3

Clear data in table after executed each test case

```
@RunWith(SpringRunner.class)
@DataJpaTest
public class PersonRepositoryTest {

    @Autowired
    private PersonRepository repository;

    @After
    public void tearDown() throws Exception {
        repository.deleteAll();
    }

}
```



Repository Testing #3

Write your first test case

```
@Test  
public void should_save_fetch_a_person() {  
  
    Person somkiat = new Person("Somkiat", "Pui");  
    repository.save(somkiat);  
  
    Optional<Person> maybeSomkiat  
        = repository.findByLastName("Pui");  
  
    assertEquals(maybeSomkiat, Optional.of(somkiat));  
}
```



Run test

\$mvnw clean test

Hibernate: drop table person if exists

Hibernate: drop sequence if exists hibernate_sequence

Hibernate: create sequence hibernate_sequence start with 1 increment by 1

Hibernate: create table person (id varchar(255) not null, first_name varchar(255), last_name varchar(255), primary key (id))

Insert data

Hibernate: call next value for hibernate_sequence

Hibernate: insert into person (first_name, last_name, id) values (?, ?, ?)

Hibernate: select person0_.id as id1_0_, person0_.first_name as first_na2_0_, person0_.last_name as last_nam3_0_ from person person0_ where person0_.last_name=?

Hibernate: select person0_.id as id1_0_, person0_.first_name as first_na2_0_, person0_.last_name as last_nam3_0_ from person person0_

2

Query data

Hibernate: drop table person if exists

Hibernate: drop sequence if exists hibernate_sequence



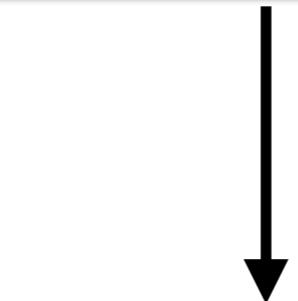
Use case 1

Integrate repository with controller

3. HelloController



2. PersonRepository



4. PersonResponse

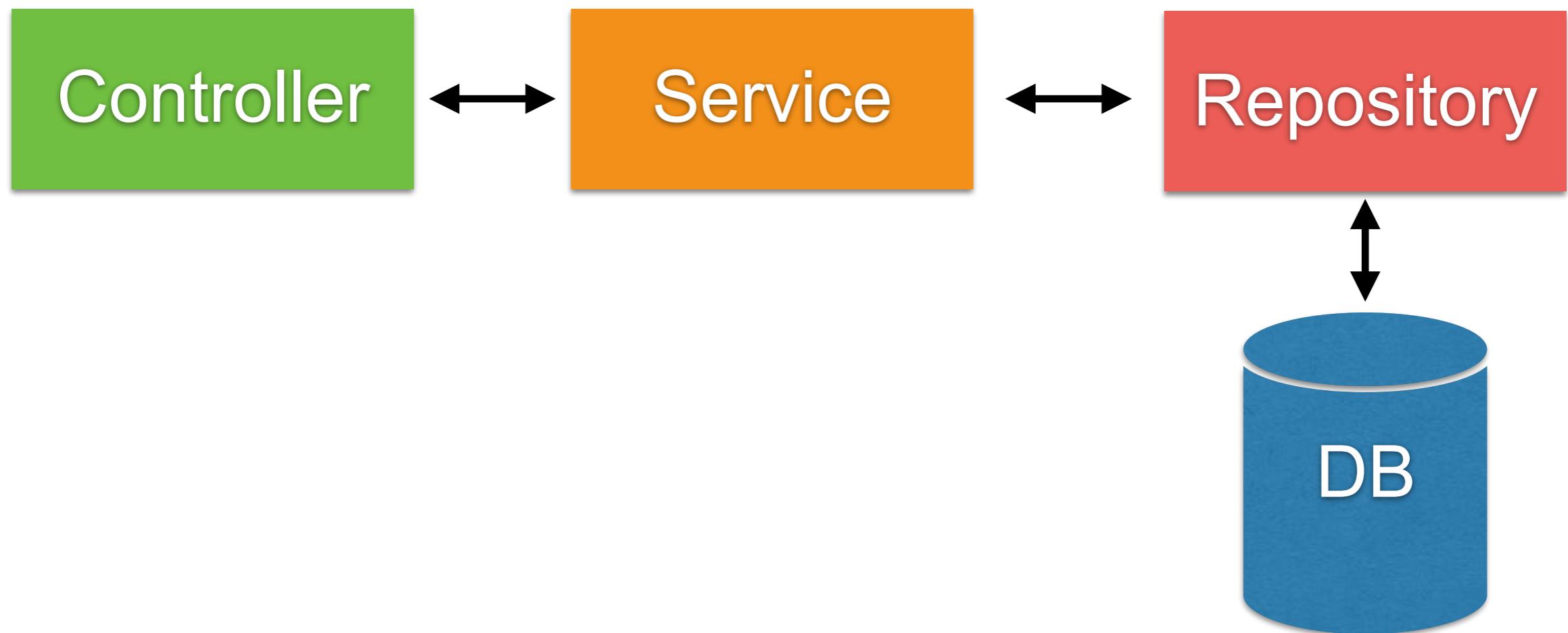
1. Person



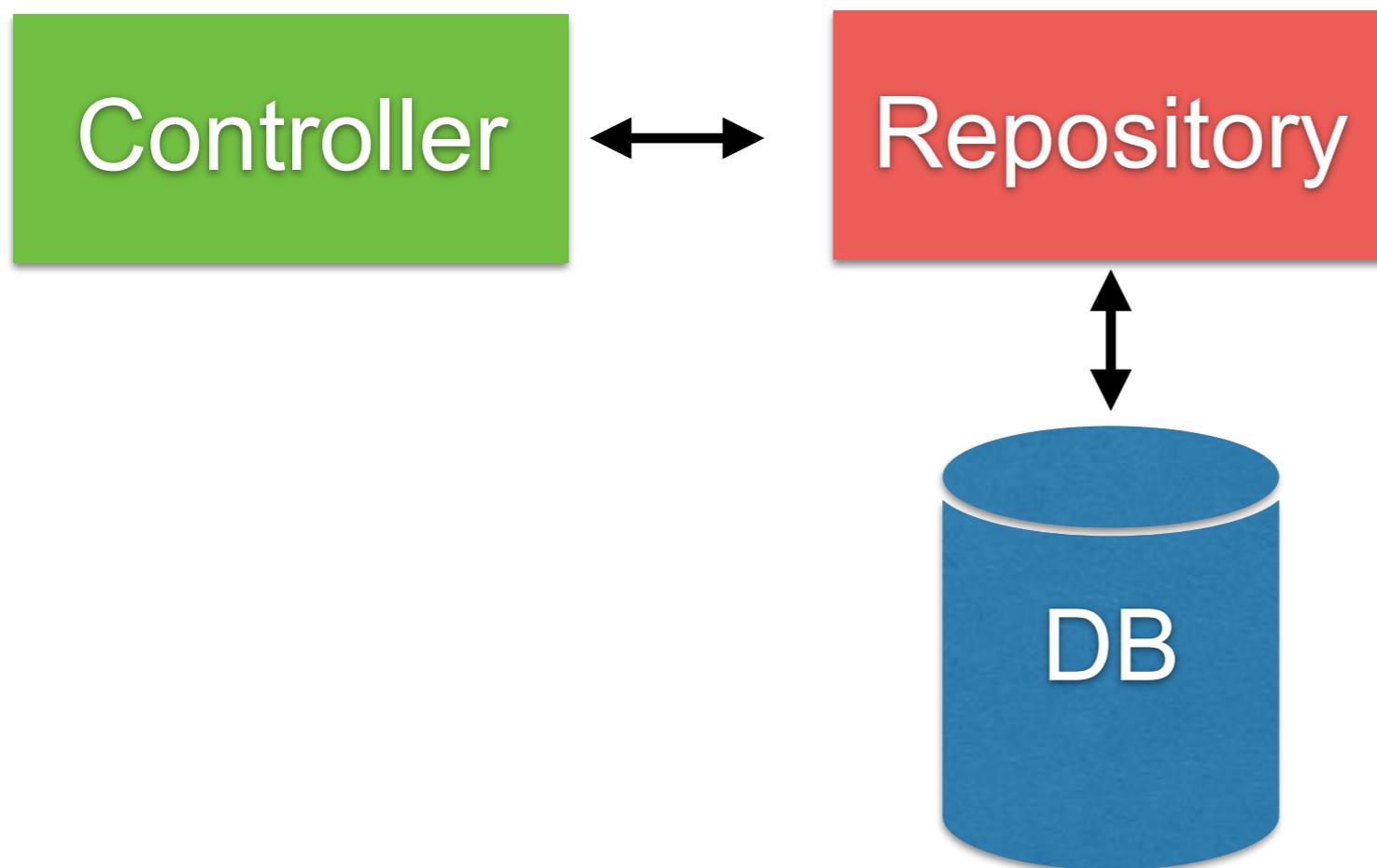
Integrate repository with service/controller



Service use repository ?



Controller use repository ?



Controller call repository

Create HelloController.java

```
@RestController
public class HelloController {

    private final PersonRepository personRepository;

    @Autowired
    public HelloController(final PersonRepository personRepository) {
        this.personRepository = personRepository;
    }
}
```



Controller call repository

Create HelloController.java

```
@GetMapping("/hello/{lastName}")
public HelloResponse hello(@PathVariable final String lastName) {

    Optional<Person> foundPerson
        = personRepository.findByLastName(lastName);

    return foundPerson
        .map(person ->
            new HelloResponse(person.getFirstName(),
                person.getLastName()))
        .orElseThrow(() -> new RuntimeException());
}
```



Run spring boot

\$mvnw spring-boot:run



Fix !!!

Modify src/main/resources/application.properties

server.port=8088

spring.datasource.url=jdbc:postgresql://127.0.0.1:15432/postgres

spring.datasource.username=testuser

spring.datasource.password=password

spring.datasource.platform=POSTGRESQL

spring.jpa.show-sql=true

spring.jpa.hibernate.ddl-auto=create-drop

spring.jpa.database-platform=org.hibernate.dialect.PostgreSQLDialect

Start database server !!



Fix !!!

Modify pom.xml

Delete or comment postgresql dependency

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>

<!-- <dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <scope>runtime</scope>
</dependency> -->
```



Run spring boot

\$mvnw spring-boot:run



Initial data in database



Initial database #1

Using @Bean and CommandLineRunner

```
@Bean  
public CommandLineRunner initData(MessageRepository repository) {  
    return new CommandLineRunner() {  
  
        @Override  
        public void run(String... args) throws Exception {  
            repository.save(new Message("somkiat1"));  
            repository.save(new Message("somkiat2"));  
        }  
    };  
}
```



Initial database #2

Using @PostConstruct

```
@PostConstruct  
public void initData() {  
    Account account1 = new Account();  
    account1.setAccountId("01");  
    accountRepository.save(account1);  
    Account account2 = new Account();  
    account2.setAccountId("02");  
    accountRepository.save(account2);  
}
```



Initial database #3

Schema (resources/schema.sql)

Data (resources/data.sql)

Schema.sql

```
CREATE TABLE account(  
    id BIGINT AUTO_INCREMENT PRIMARY KEY,  
    account_Id VARCHAR(16) NOT NULL UNIQUE,  
    mobile_No VARCHAR(10),  
    name VARCHAR(50),  
    account_Type CHAR(2)  
);
```

Data.sql

```
INSERT INTO account (account_Id) VALUES ('01');  
INSERT INTO account (account_Id) VALUES ('02');
```



Initial database 2

Disable auto generate DDL from JPA in file application.yml

```
spring:  
  jpa:  
    show-sql: true  
    hibernate:  
      ddl-auto: none
```



Initial database 2

Problem with naming strategy !!

```
spring:  
  jpa:  
    show-sql: true  
    hibernate:  
      ddl-auto: none  
      naming:  
        physical-strategy:  
          org.springframework.boot.orm.jpa.hibernate.SpringPhysicalNamingStrategy  
        implicit-strategy:  
          org.springframework.boot.orm.jpa.hibernate.SpringImplicitNamingStrategy
```

<https://github.com/spring-projects/spring-boot/tree/master/spring-boot-project/spring-boot/src/main/java/org/springframework/boot/orm/jpa/hibernate>



Run and see from logging

Execute file schema.sql and data.sql

```
.datasource.init.ScriptUtils      : Executing SQL script from URL [file:/Users/somki...  
.datasource.init.ScriptUtils      : Executed SQL script from URL [file:/Users/somki...  
.datasource.init.ScriptUtils      : Executing SQL script from URL [file:/Users/somki...  
.datasource.init.ScriptUtils      : Executed SQL script from URL [file:/Users/somki...
```



Run spring boot

\$mvnw spring-boot:run



Write controller testing ?

\$mvnw clean test

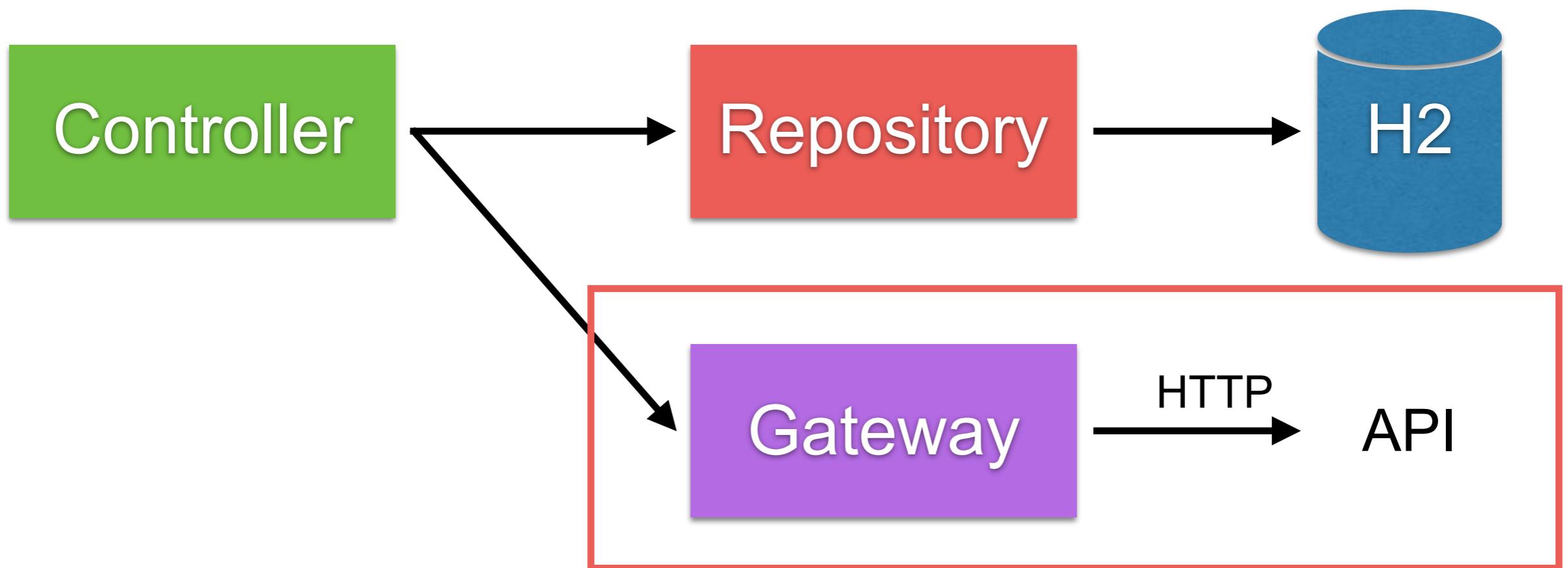


Use case 2

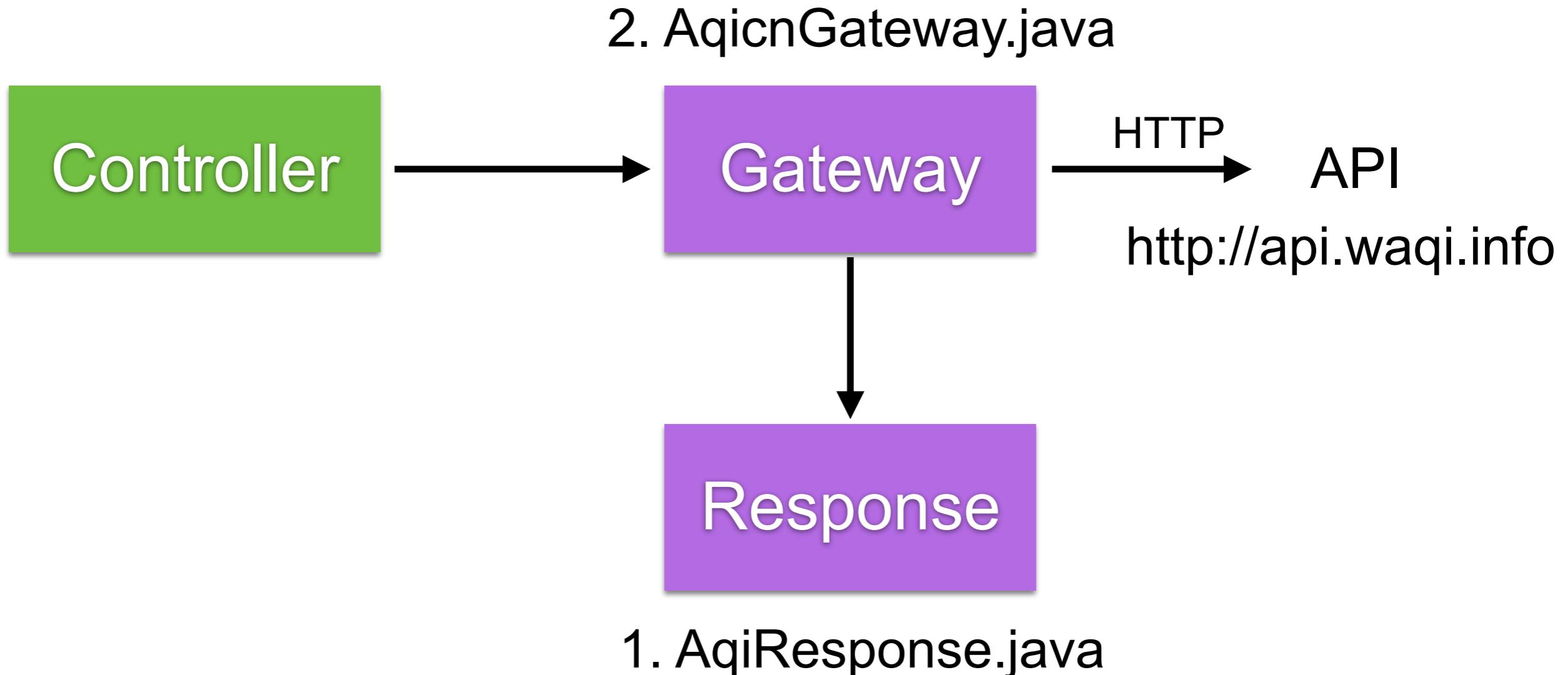


Use case 2

Working with API



Working with API



1. Create Response class

In package ahi

```
public class AqiResponse {  
  
    private Data data;  
  
    public AqiResponse() {  
    }  
  
    public AqiResponse(Data data) {  
        this.data = data;  
    }  
}
```

```
public class Data {  
  
    private int aqi;  
  
    public Data() {  
    }  
  
    public Data(int aqi) {  
        this.aqi = aqi;  
    }  
}
```



2. Create AqicnGateway class #1

In package ahi

```
@Component
public class AqicnGateway {

    private final RestTemplate restTemplate;
    private final String url;
    private final String token;

    @Autowired
    public AqicnGateway(final RestTemplate restTemplate,
                        @Value("${aqicn.org.url}") final String url,
                        @Value("${aqicn.org.token}") final String token) {
        this.restTemplate = restTemplate;
        this.url = url;
        this.token = token;
    }
}
```



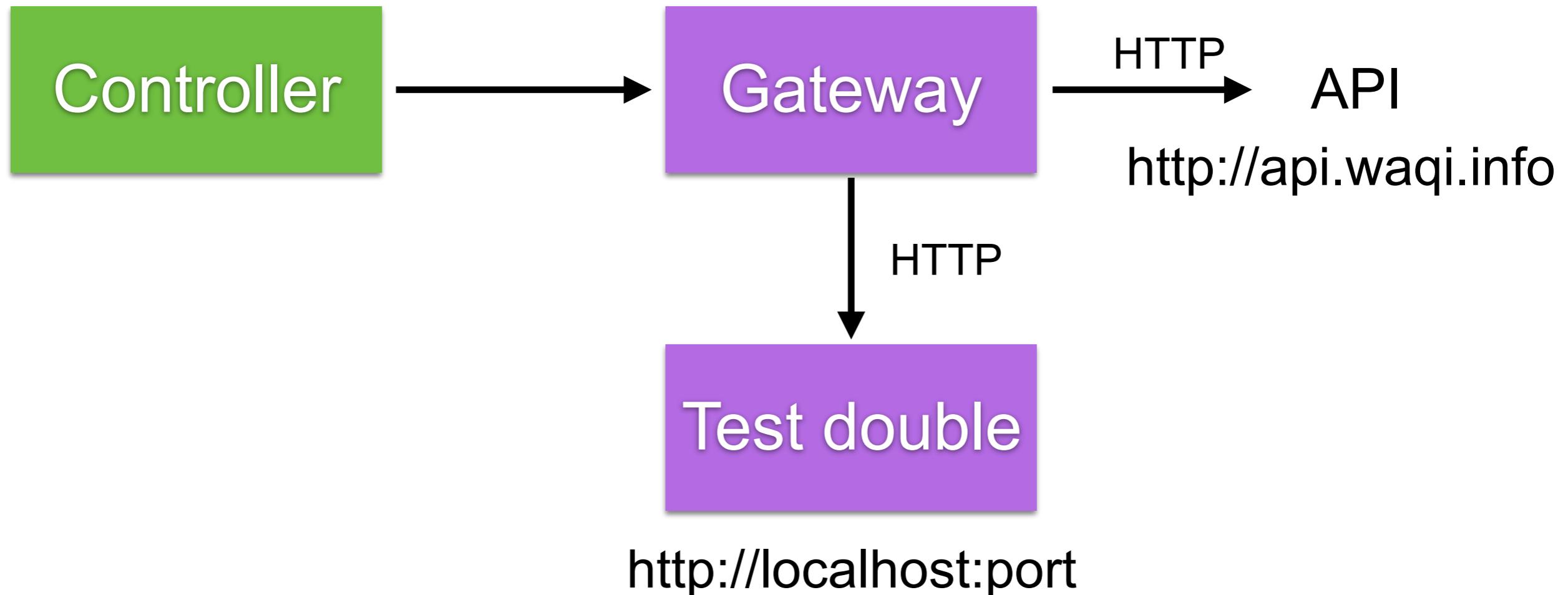
2. Create AqicnGateway class #2

Get data from API

```
public Optional<AqiResponse> fetchData(String target) {  
  
    String targetUrl  
        = String.format("%s/%s/?token=%s", url, target, token);  
  
    try {  
  
        return Optional.ofNullable(restTemplate.getForObject(  
                    targetUrl, AqiResponse.class));  
  
    } catch (RestClientException e) {  
        return Optional.empty();  
    }  
}
```



Testing with API



Testing with API

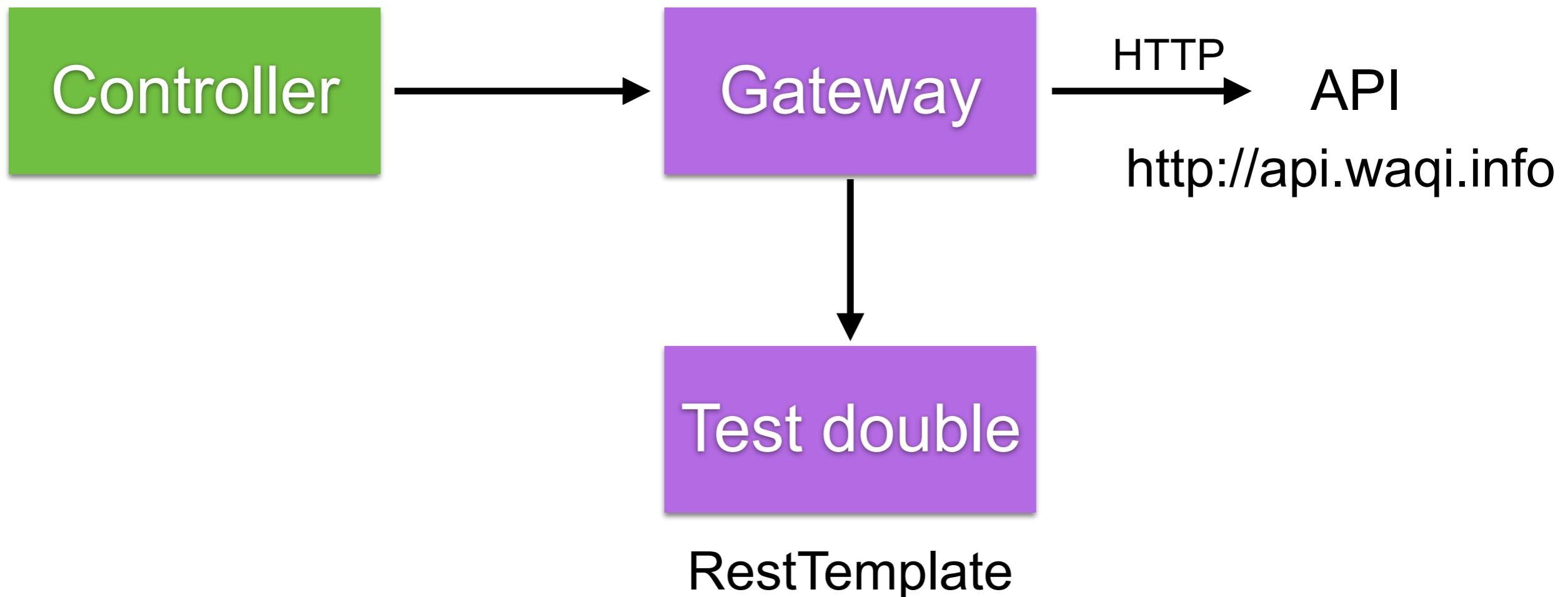
Unit testing with Mockito

Integration testing with WireMock

Consumer testing with Pact



Unit test with Mockito



1. Unit testing #1

Call API with RestTemplate

```
@RunWith(MockitoJUnitRunner.class)
public class AqicnGatewayTest {

    @Mock
    private RestTemplate restTemplate;

    private AqicnGateway gateway;
```



1. Unit testing #2

Success case

```
@Test
public void shouldCallService() throws Exception {
    gateway = new AqicnGateway(restTemplate, "http://localhost", "xxxx");

    String target = "thailand/pathum-thani/bangkok-university-rangsit-campus";
    AqiResponse expectedResponse = new AqiResponse(new Data(1));

    String targetUrl = "http://localhost/" + target + "?token=xxxx";
    given(restTemplate.getForObject(targetUrl, AqiResponse.class))
        .willReturn(expectedResponse);

    Optional<AqiResponse> actualResponse = gateway.fetchData(target);
    assertEquals(actualResponse, Optional.of(expectedResponse));
}
```



1. Unit testing #3

Failure case

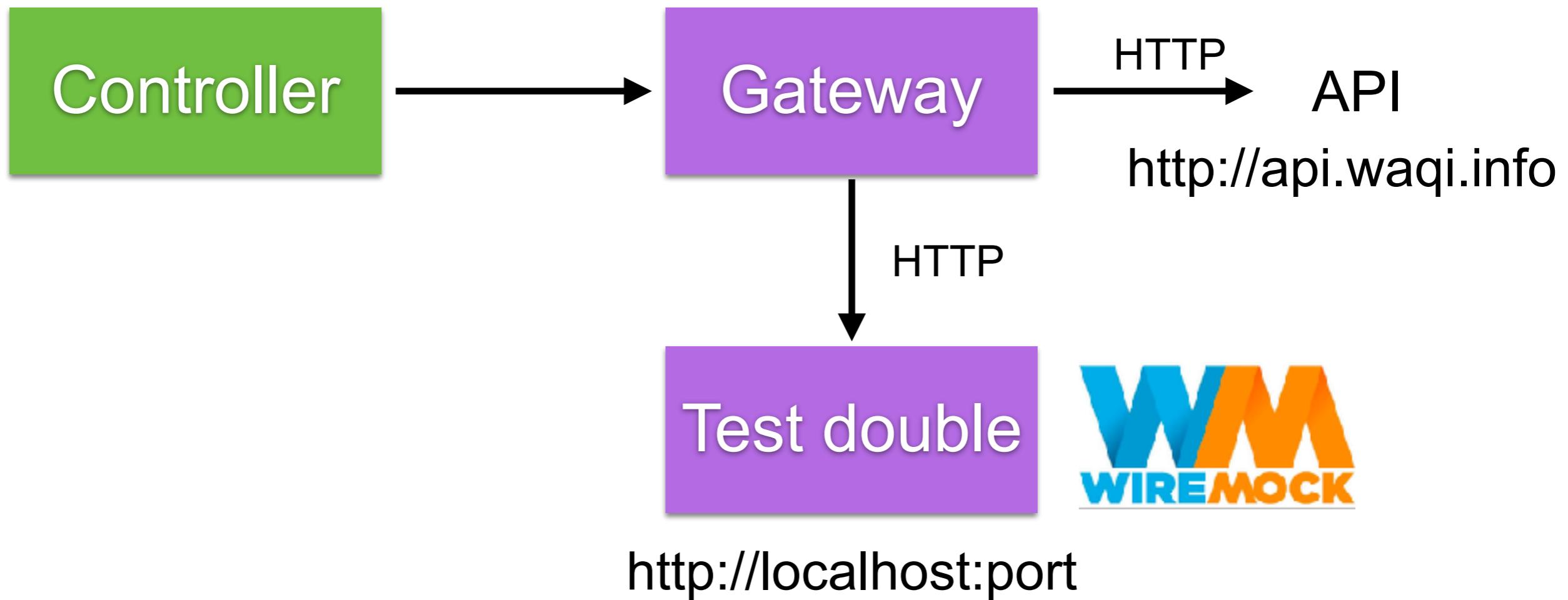
```
@Test
public void shouldReturnEmptyWhenServiceIsUnavailable() throws Exception {
    gateway = new AqicnGateway(restTemplate, "http://localhost", "xxxx");

    String target = "thailand/pathum-thani/bangkok-university-rangsit-campus";
    String targetUrl = "http://localhost/" + target + "?token=xxxx";
    given(restTemplate.getForObject(targetUrl, AqiResponse.class))
        .willThrow(new RestClientException("something went wrong"));

    Optional<AqiResponse> actualResponse = gateway.fetchData(target);
    assertEquals(actualResponse, Optional.empty());
}
```



Integration test with WireMock



/src/test/resources/application.properties

```
aqicn.org.url=http://localhost:8888
aqicn.org.token=xxxx
```



2. Integration testing #1

Working with WireMock

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class AqicnGatewaySpringBootTest {

    @Autowired
    private AqicnGateway gateway;          Default port = 8888

    @Rule
    public WireMockRule wireMockRule = new WireMockRule(8888);
}
```



2. Integration testing #2

Success case

```
@Test
public void shouldCallService() throws Exception {
    String target = "thailand/pathum-thani/bangkok-university-rangsit-campus";

    wireMockRule.stubFor(get(urlPathEqualTo("/*"+target+"/"))
        .withQueryParam("token", equalTo("xxxx"))
        .willReturn(aResponse()
            .withBody(read("classpath:apiResponse.json"))
            .withHeader(CONTENT_TYPE, MediaType.APPLICATION_JSON_VALUE)
            .withStatus(200)));
}

Optional<AqiResponse> response = gateway.fetchData(target);

Data data = new Data(80);
Optional<AqiResponse> expectedResponse = Optional.of(new AqiResponse(data));
assertEquals(expectedResponse, response);
}
```

Stub response



2. Integration testing #3

Read data from resources folder

```
public static String read(String filePath)
    throws IOException {
    File file = ResourceUtils.getFile(filePath);
    return new String(Files.readAllBytes(file.toPath()));
}
```



Write controller testing ?

\$mvnw clean test



Working with transaction !!



Separate tests with jUnit



<https://semaphoreci.com/community/tutorials/how-to-split-junit-tests-in-a-continuous-integration-environment>



Separate test with jUnit



Working with jUnit Category

Create interface in each category

```
package com.lotto.lotto.category;  
  
public interface UnitTest {  
}
```

```
package com.lotto.lotto.category;  
  
public interface SlicingTest {  
}
```

```
package com.lotto.lotto.category;  
  
public interface IntegrationTest {  
}
```



Add category in each test class

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
@Category(IntegrationTest.class)
public class AccountControllerTest {

    @Autowired
    private TestRestTemplate testRestTemplate;

    @MockBean
    private UserService userService;
```

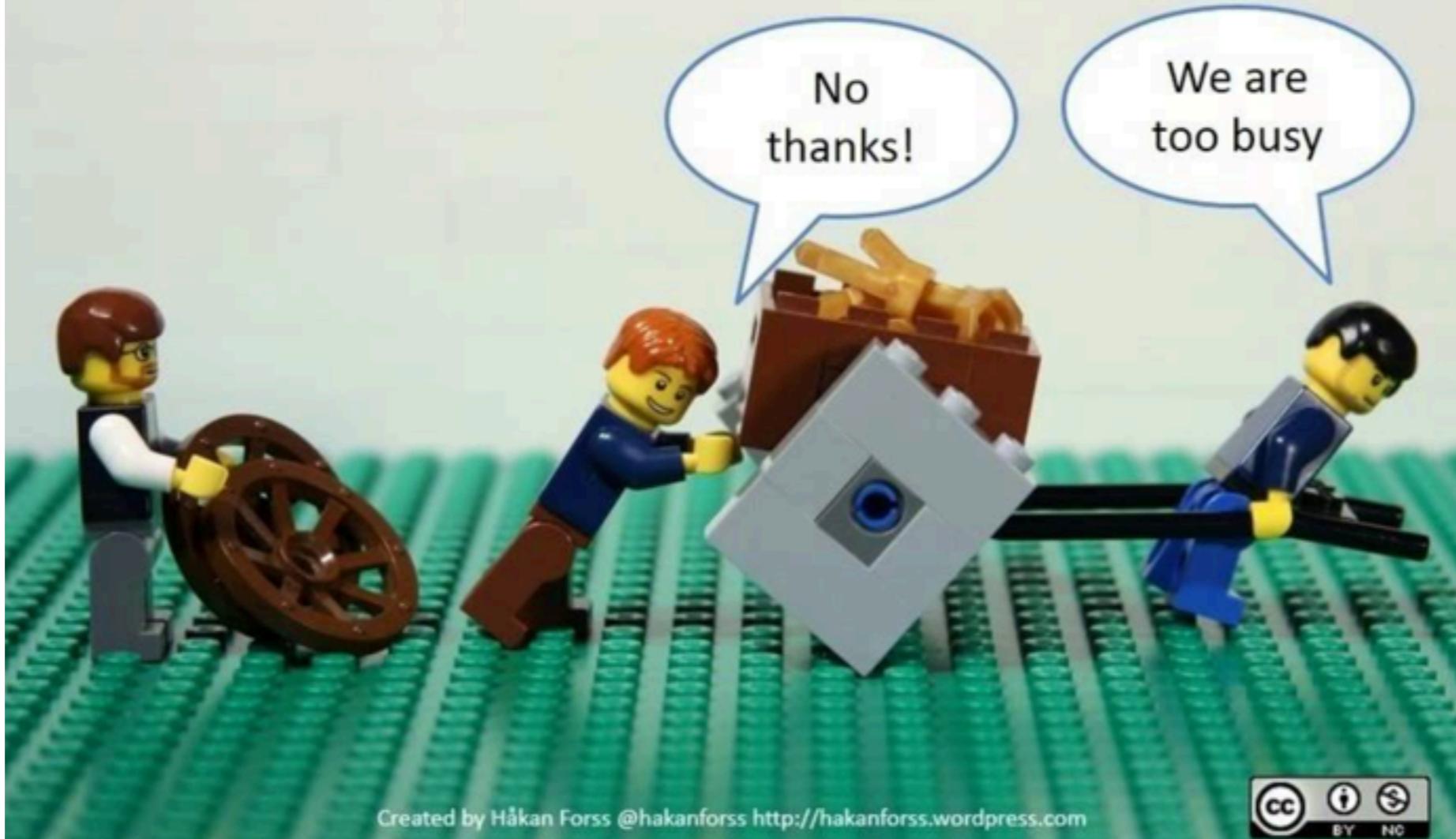


Run with category

```
$mvnw clean test  
-Dgroups="com.lotto.lotto.category.UnitTest"
```



Are you too busy to improve?



Thank you
Q/A

