

Spring and Spring Boot

Fundamentals

07.09.2023 Marius Reusch

About Me

About You

Name

(Company)

Java experience (in years)

Spring (Boot) experience (in years)

Experience with other enterprise frameworks, like Java EE (in years)

I know the basic concepts of REST services

Breaks? Coffee ?
Lunch?

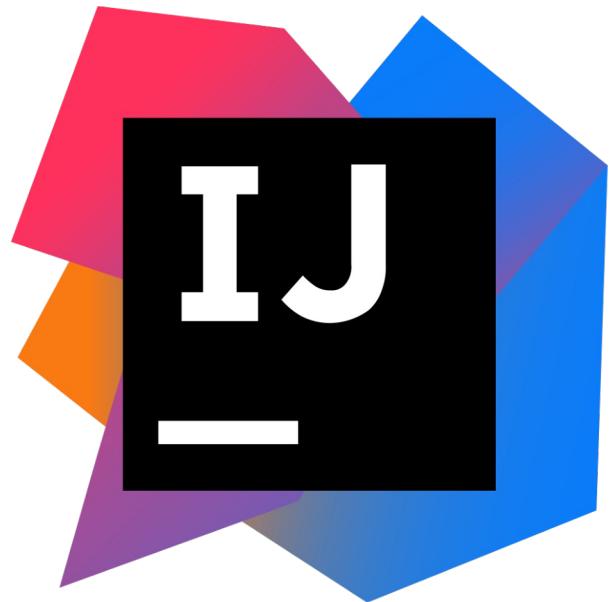
Agenda

- Introduction
- Spring?
- Spring Basics
- Spring Web / MVC
- Spring Boot
- Wrap-Up

Introduction / Objectives For Today

- Understand the core concepts of the Spring Framework
- Know the difference between Spring Framework, Spring MVC and Spring Boot
- Demystify the “magic” behind Spring and Spring Boot
- (Develop Rest Services with Spring)

Introduction / Versions (beside Spring)



2023.2.X



JDK 17



Gradle

8.3

Spring?

“Spring makes it easy to create **Java enterprise applications**. It provides everything you need to embrace the Java language in an enterprise environment. [...]”

Spring is **open source**.

(<https://docs.spring.io/spring/docs/current/spring-framework-reference/overview.html>)

Spring? / Spring Projects

<https://spring.io/projects>

Spring? / Spring Projects



Spring Framework

Provides core support for dependency injection, transaction management, web apps, data access, messaging, and more.

6.0.11



Spring Boot

Takes an opinionated view of building Spring applications and gets you up and running as quickly as possible.

3.1.3

Spring Basics

Spring Basics / Spring Framework Modules

Data Access / Integration

JDBC

ORM

OXM

JMS

Transactions

Web / MVC

Portlet

Web

Servlet

AOP

Instrumentation

Core Container

Beans

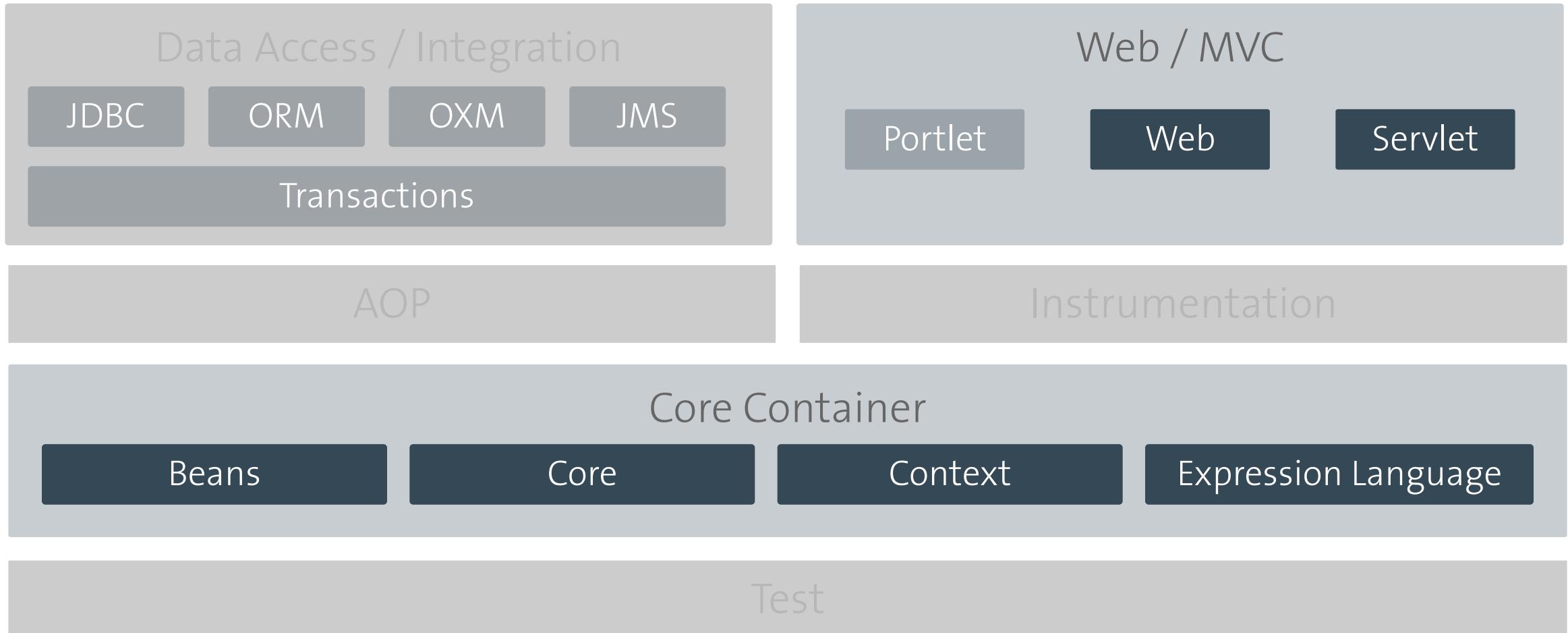
Core

Context

Expression Language

Test

Spring Basics / Spring Framework Modules



Spring Basics / Topics

- Dependency Injection
- Beans
- Bean Stereotypes
- Bean Scopes
- Profiles and other Conditions
- Configuration
- Application Context Summary
- Wrap-Up

Spring Basics

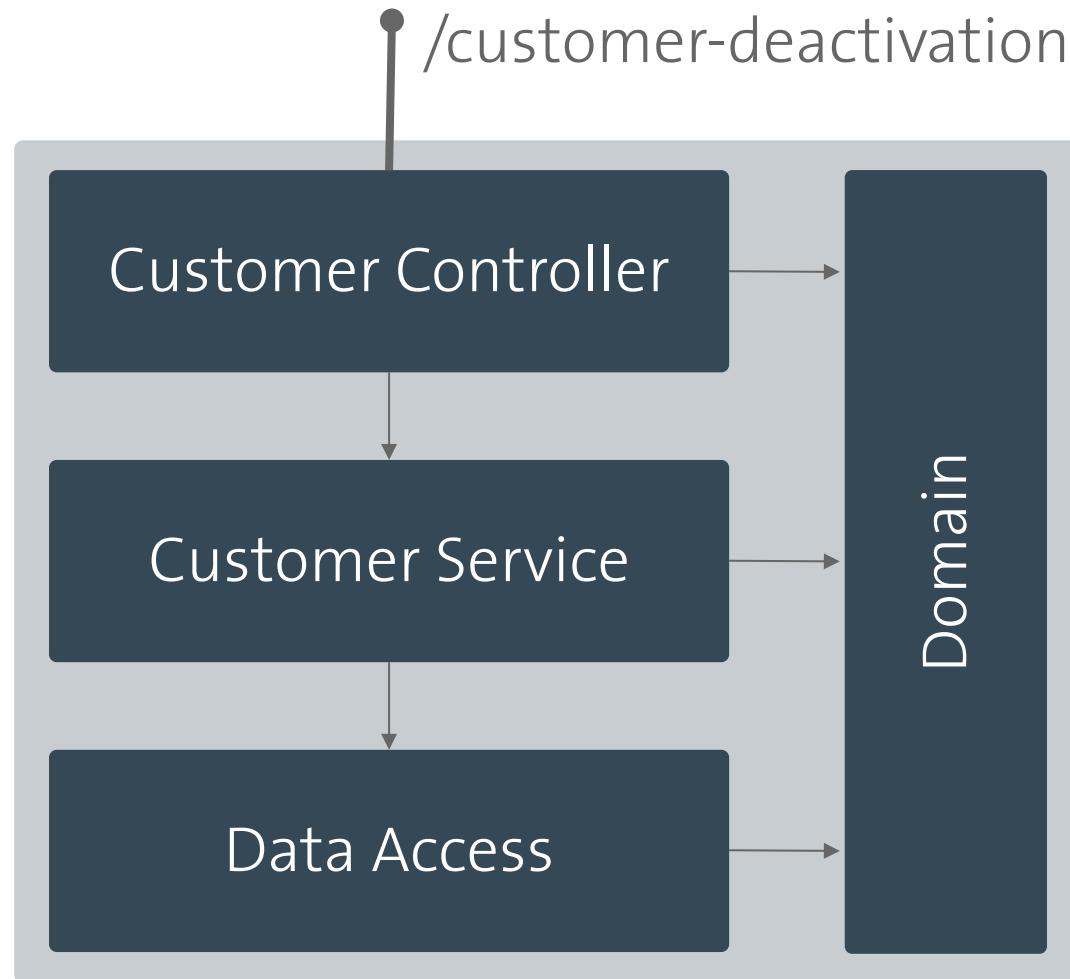
Dependency Injection

Spring Basics / Dependency Injection

Use Case: Deactivate customer account

In case of a successful deactivation, a letter should be sent to the customer. If an e-mail address is available, an e-mail should also be sent to the customer as confirmation.

Spring Basics / Dependency Injection



```
public class CustomerService {  
  
    static String DEACTIVATION_MESSAGE = "Your customer account has been successfully removed";  
  
    public void deactivateCustomer(String customerId) {  
        CustomerLoader customerLoader = new CustomerLoader();  
        Customer customer = customerLoader.findById(customerId);  
  
        if (customer.hasEmailAddress()) {  
            String emailAddress = customer.getEmailAddress();  
            EmailService emailService = new EmailService();  
            emailService.send(emailAddress, "Customer Account", DEACTIVATION_MESSAGE);  
        }  
  
        PostalService postalService = new PostalService();  
        postalService.sendLetter(customer.getMailingAddress(), DEACTIVATION_MESSAGE);  
    }  
}
```

Spring Basics / Dependency Injection

```
public class CustomerService {  
  
    private final CustomerLoader customerLoader;  
    private final EmailService emailService;  
  
    public CustomerService(CustomerLoader customerLoader, EmailService emailService) {  
        this.customerLoader = customerLoader;  
        this.emailService = emailService;  
    }  
}
```

Constructor Injection

Spring Basics / Dependency Injection

```
public class CustomerService {  
  
    private CustomerLoader customerLoader;  
    private EmailService emailService;  
  
    public void setCustomerLoader(CustomerLoader customerLoader) {  
        this.customerLoader = customerLoader;  
    }  
  
    public void setEmailService(EmailService emailService) {  
        this.emailService = emailService;  
    }  
}
```

Setter Injection

Spring Basics / Dependency Injection

```
public class CustomerService {  
  
    public CustomerLoader customerLoader;  
    public EmailService emailService;  
    public PostalService postalService;  
}
```

Field Injection

**Naive approach with
public fields. The
dependency would be
set by assigning it to
the field**

“Dependency Injection is a
25-dollar term for a 5-cent
concept”

(James Shore)

Spring Basics / Dependency Injection

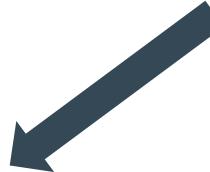
```
public static void main(String[] args) {  
    EmailService emailService = new EmailService();  
    PostalService postalService = new PostalService();  
    CustomerLoader customerLoader = new CustomerLoader();  
    CustomerService customerService = new CustomerService(emailService,  
        postalService, customerLoader);  
  
    CustomerController customerController = new CustomerController(customerService);  
  
    customerController.deactivateCustomerAccount("1");  
}
```

**Do you see any
problems here?**

What could help us here?

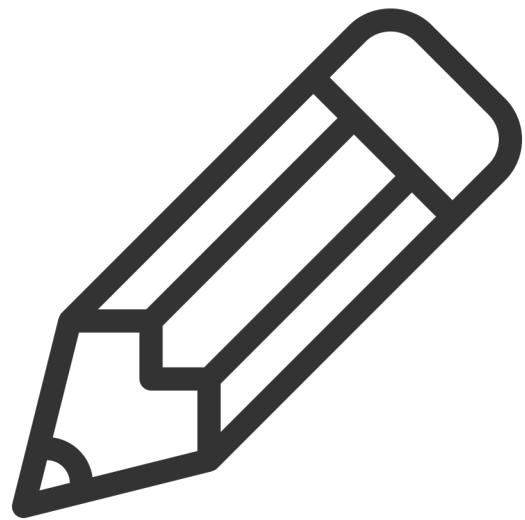
Dependency Injection Frameworks to the rescue!

Dependency Injection Framework



Target: Spring should take care of “creating” and “assembling” our classes (and a lot of other stuff).

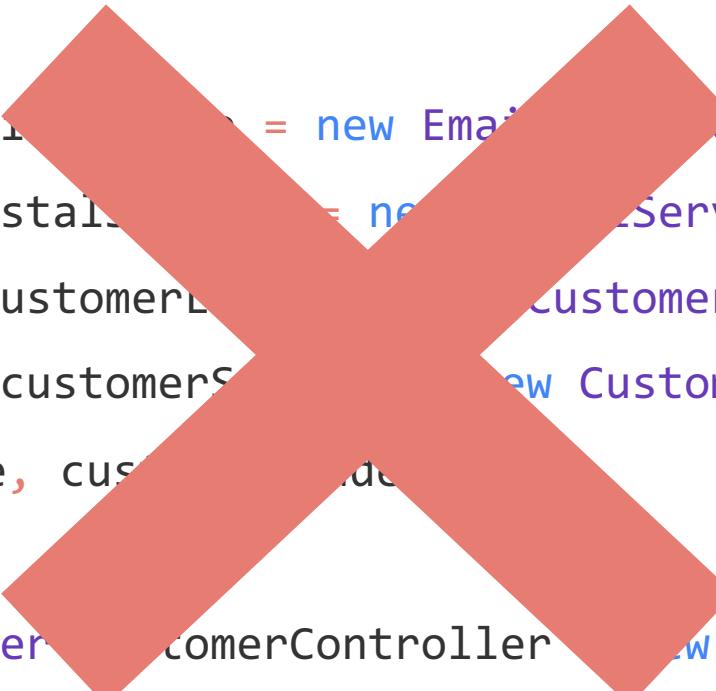
Imagine a box.



What do we need from the box?

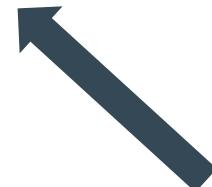
Spring Basics / Dependency Injection

```
public static void main(String[] args) {  
    EmailService emailService = new EmailService();  
    PostalService postalService = new PostalService();  
    CustomerLoader customerLoader = new CustomerLoader();  
    CustomerService customerService = new CustomerService(emailService,  
        postalService, customerLoader);  
  
    CustomerController customerController = new CustomerController(customerService);  
  
    customerController.deactivateCustomerAccount("1");  
}
```



Spring Basics / Dependency Injection

```
public static void main(String[] args) {  
    Box box = new Box();  
  
    CustomerController customerController = box.get(CustomerController.class);  
  
    customerController.deactivateCustomerAccount("1");  
}
```



Hej Box, please give me
an instance of my
CustomerController!

Spring Basics

Beans

Beans?

Managed?

“Object managed by an IoC
container”

Rod Johnson

This is our “box”

Managed!

“A bean is an object that is **instantiated, assembled**, and **otherwise managed** by a Spring IoC container.”

Later

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

This is our “box”

@Component

This annotation transforms a basic Java class (like the CustomerService) into a **Bean** managed by a container (our “**box**”). You can imagine the box as a container in which we put all the classes in that should be managed by Spring



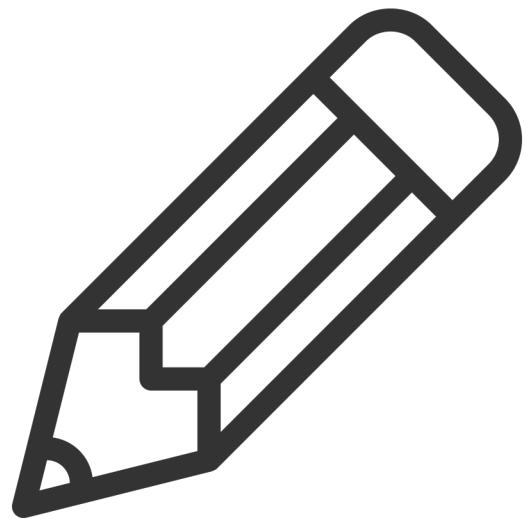
But how does the “box”
know where to look for the
Beans annotated with
@Component?

The “box” doesn’t know!

```
@ComponentScan("com.springfundamentals")
```



This annotation tells the “box”, where it can search for components. It will search within this package and all its sub-packages. You can also omit all package information, then the package of the class annotated with this annotation will be the root package.



But in our project we
use this **@Autowired**
annotation...

@Autowired

This annotation is used to tell the “box” where it should inject a bean.

Spring Basics / Dependency Injection

```
@Component  
public class CustomerService {  
  
    private final CustomerLoader customerLoader;  
    private final EmailService emailService;  
  
    @Autowired  
    public CustomerService(CustomerLoader customerLoader, EmailService emailService) {  
        this.customerLoader = customerLoader;  
        this.emailService = emailService;  
    }  
}
```

Constructor Injection

can be omitted (in case of a single constructor)

The Spring team generally
advocates constructor
injection.

Spring Basics / Dependency Injection

```
@Component  
public class CustomerService {  
    private CustomerLoader customerLoader;  
    private EmailService emailService;  
  
    @Autowired  
    public void setCustomerLoader(CustomerLoader customerLoader) {  
        this.customerLoader = customerLoader;  
    }  
  
    @Autowired  
    public void setEmailService(EmailService emailService) {  
        this.emailService = emailService;  
    }  
}
```

Setter Injection

Setter injection can
be useful for optional
dependencies

Spring Basics / Dependency Injection

```
public class CustomerService {  
    @Autowired  
    private CustomerLoader customerLoader;  
    @Autowired  
    private EmailService emailService;  
}
```

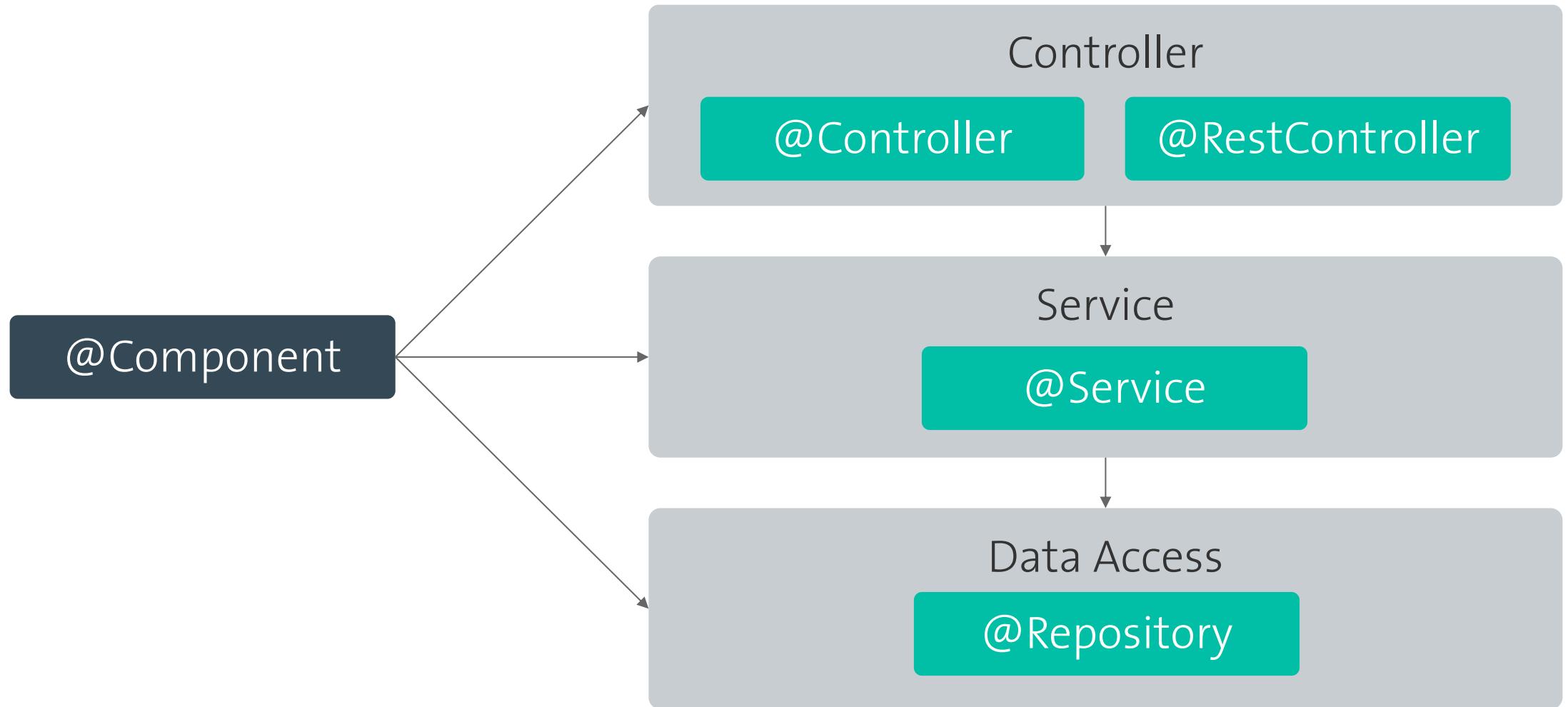
**Not recommended!
(at least not in production code)**

Field Injection

Spring Basics

Bean Stereotypes

Spring Basics / Bean Stereotypes



Spring Basics / Component Hierarchy

@Configuration

@Component

@Controller

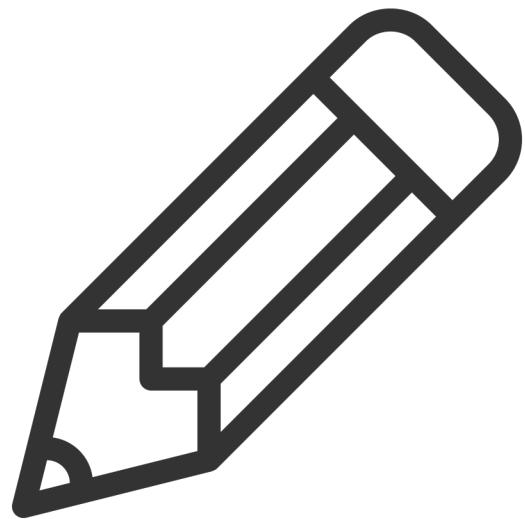
@Service

@Repository

@ControllerAdvice

@RestController

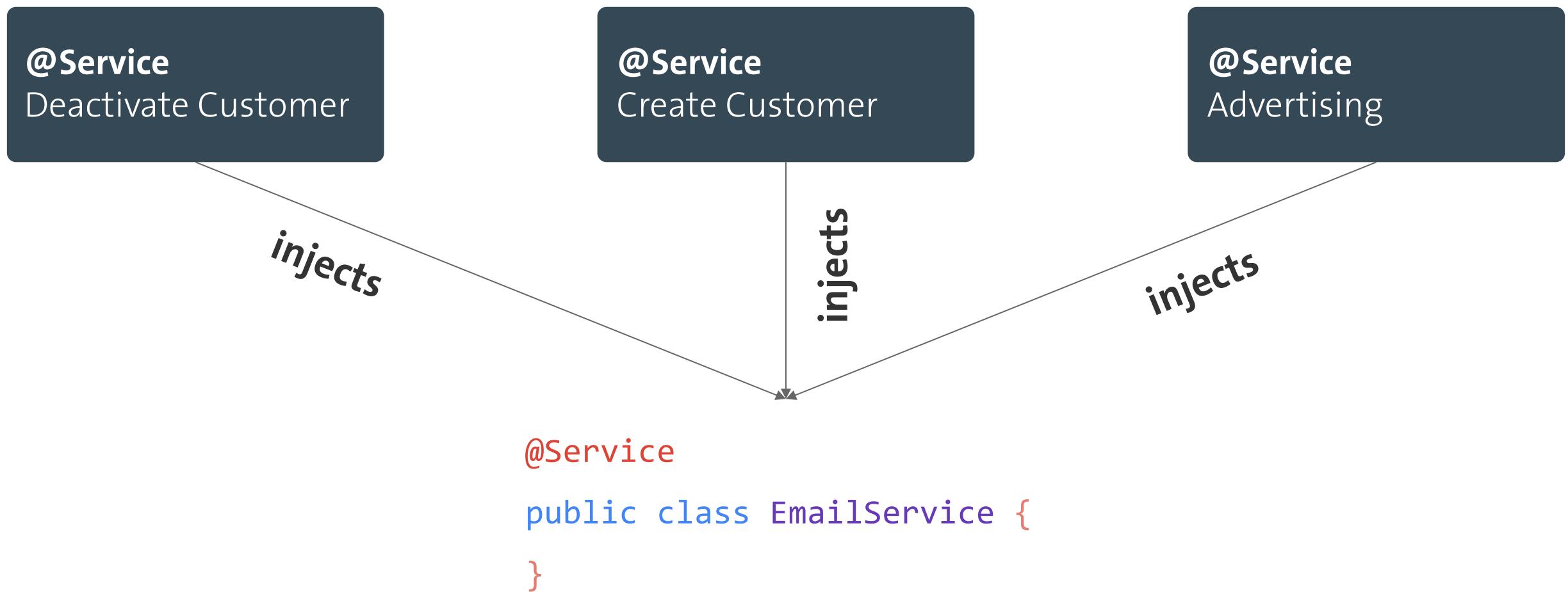
@Repository, @RestController, @Configuration and @ControllerAdvice have additional functionalities. Classes annotated with @Controller and @Service behave exactly as if they had been annotated with @Component. However, it is strongly recommended that you use the specific annotations instead of @Component.



Spring Basics

Bean Scopes

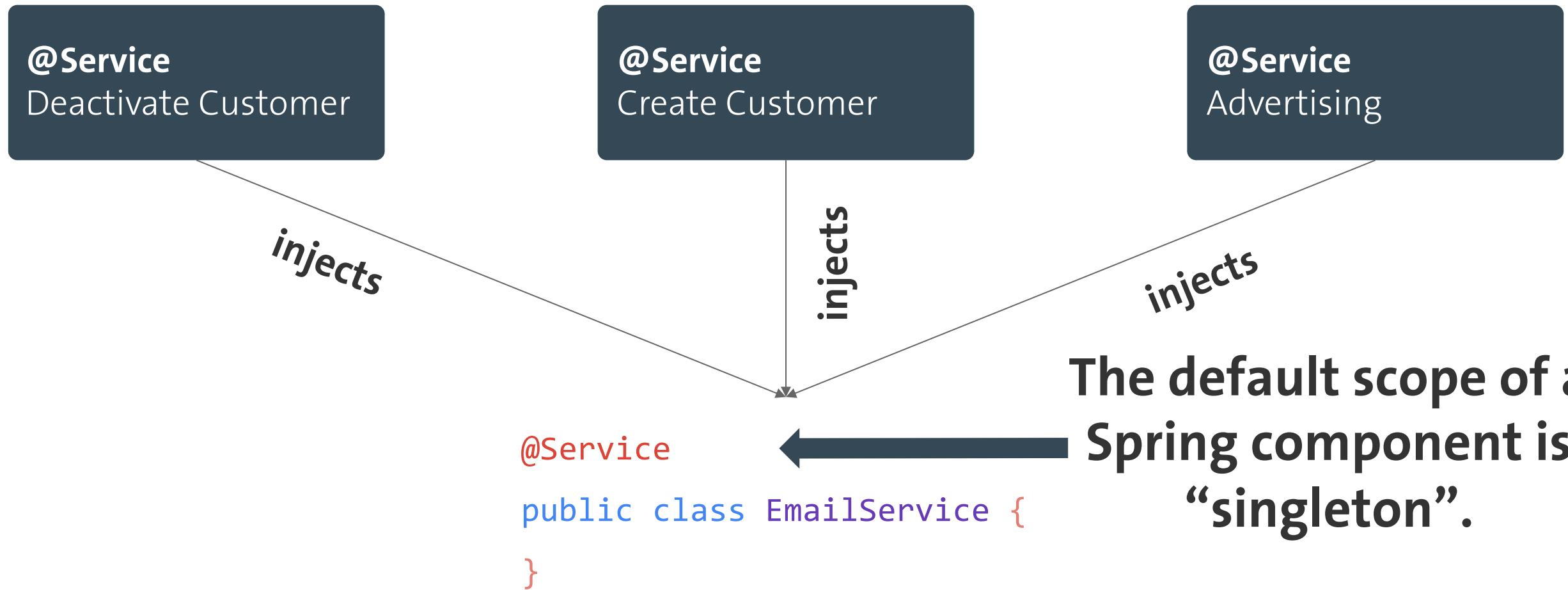
Spring Basics / Bean Scopes



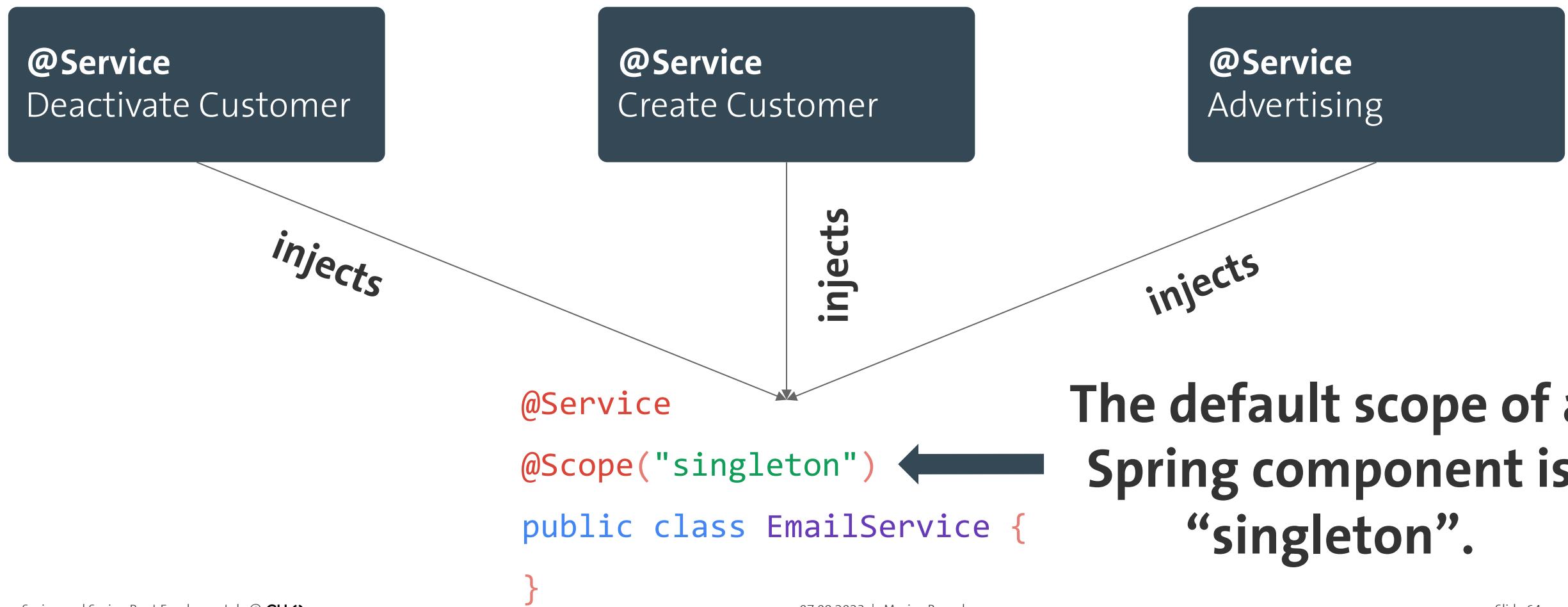
How many instances of the EmailService Bean do I have?

1

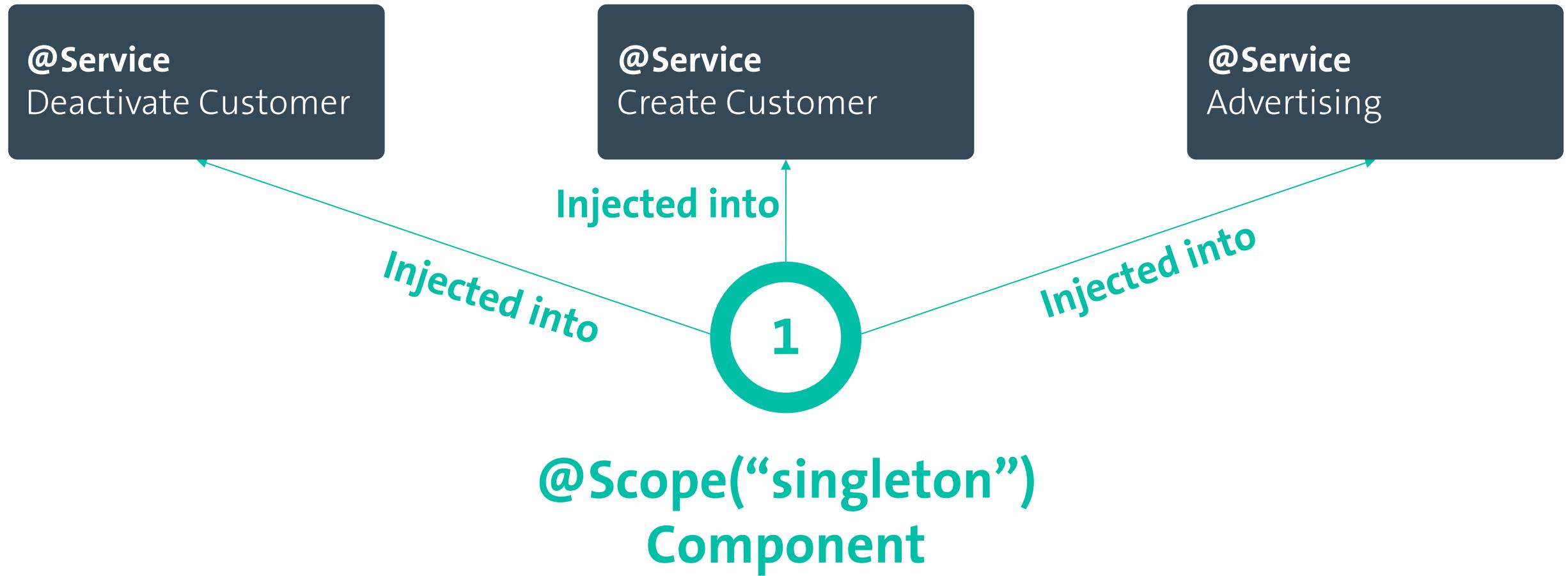
Spring Basics / Bean Scopes



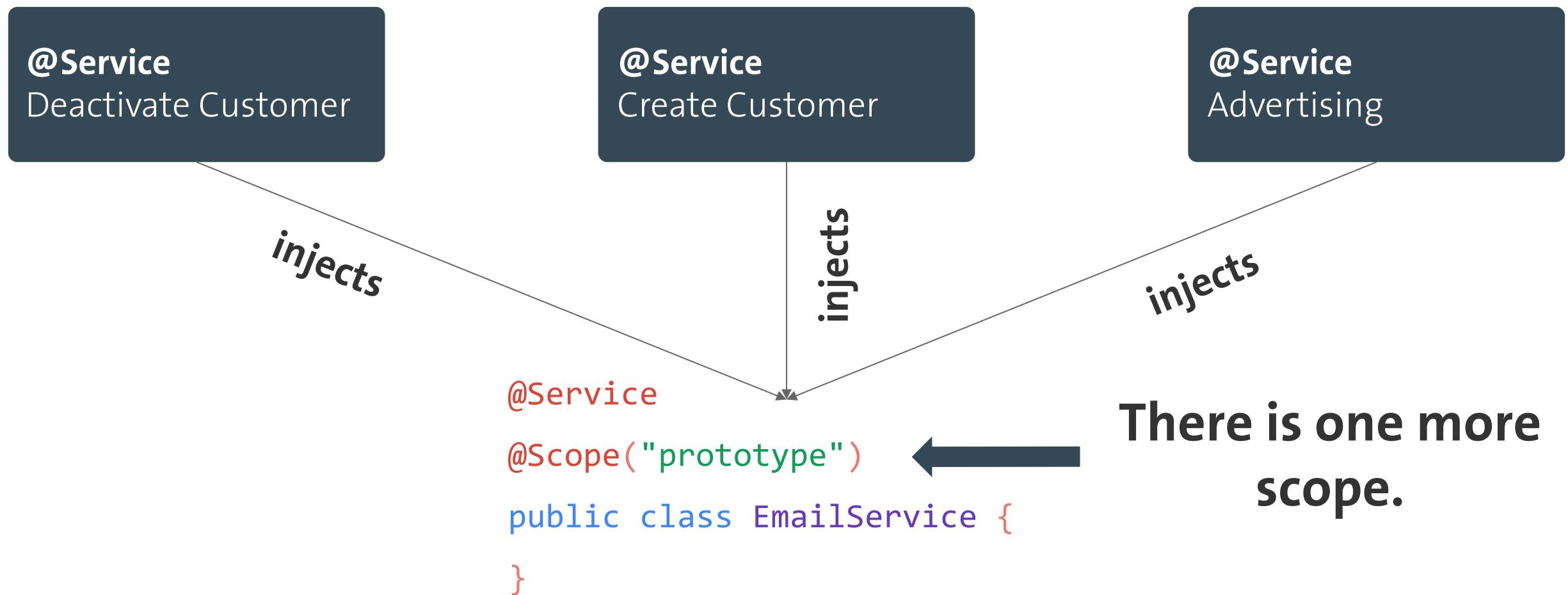
Spring Basics / Bean Scopes



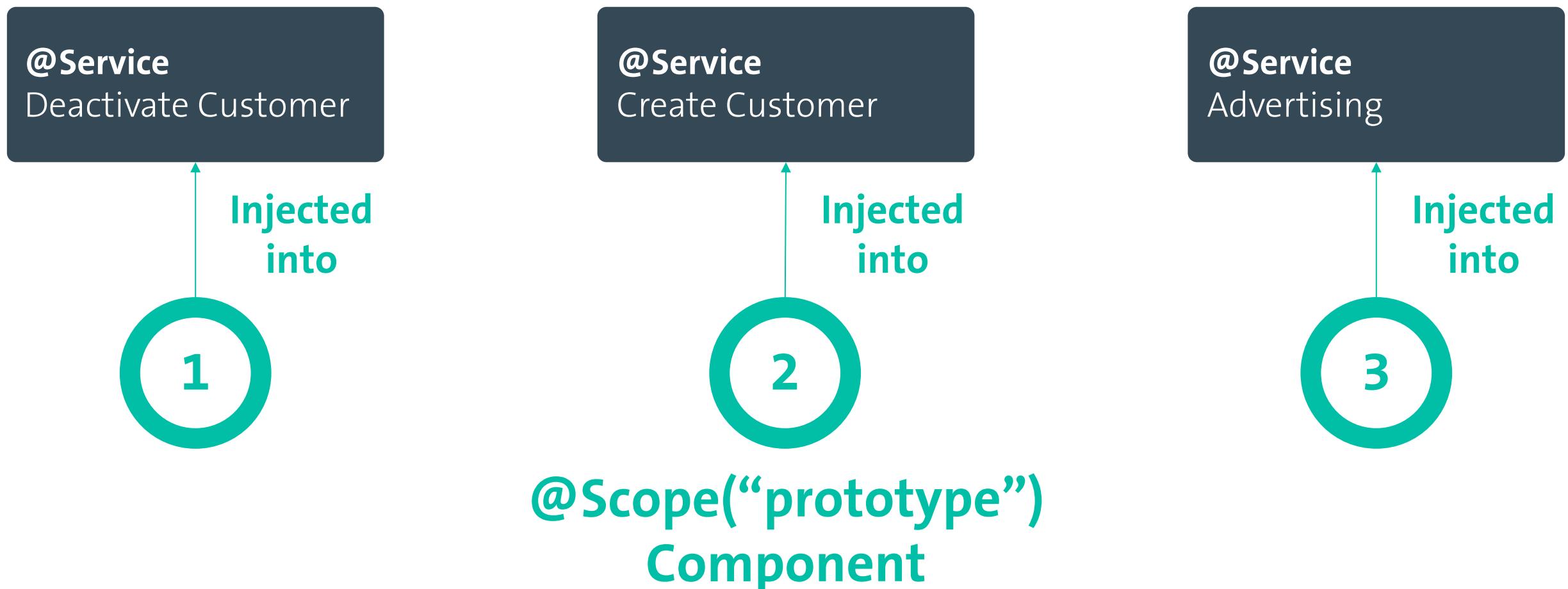
Spring Basics / Bean Scopes



Spring Basics / Bean Scopes



Spring Basics / Bean Scopes



Spring Basics / Bean Scopes



Basic Spring Bean scopes. In most cases you need `@Scope("singleton")`.



Basically you can use these four additional scopes in a Spring MVC project. But you will need them very rarely.

“As a rule, you should use the prototype scope for all stateful beans and the singleton scope for stateless beans.”

(<https://docs.spring.io/spring/docs/5.1.3.RELEASE/spring-framework-reference/core.html#beans-factory-scopes-prototype>)

“As a rule, you  should use the prototype scope for all stateful beans and the singleton scope for stateless beans.”

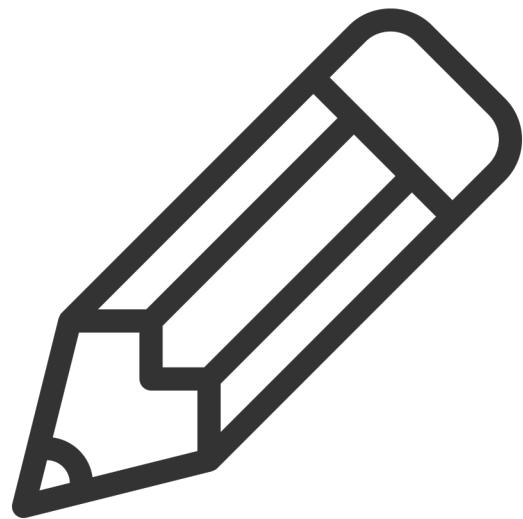
(<https://docs.spring.io/spring/docs/5.1.3.RELEASE/spring-framework-reference/core.html#beans-factory-scopes-prototype>)

What is your experience?

99.5% Singleton

0.49 % RequestScope

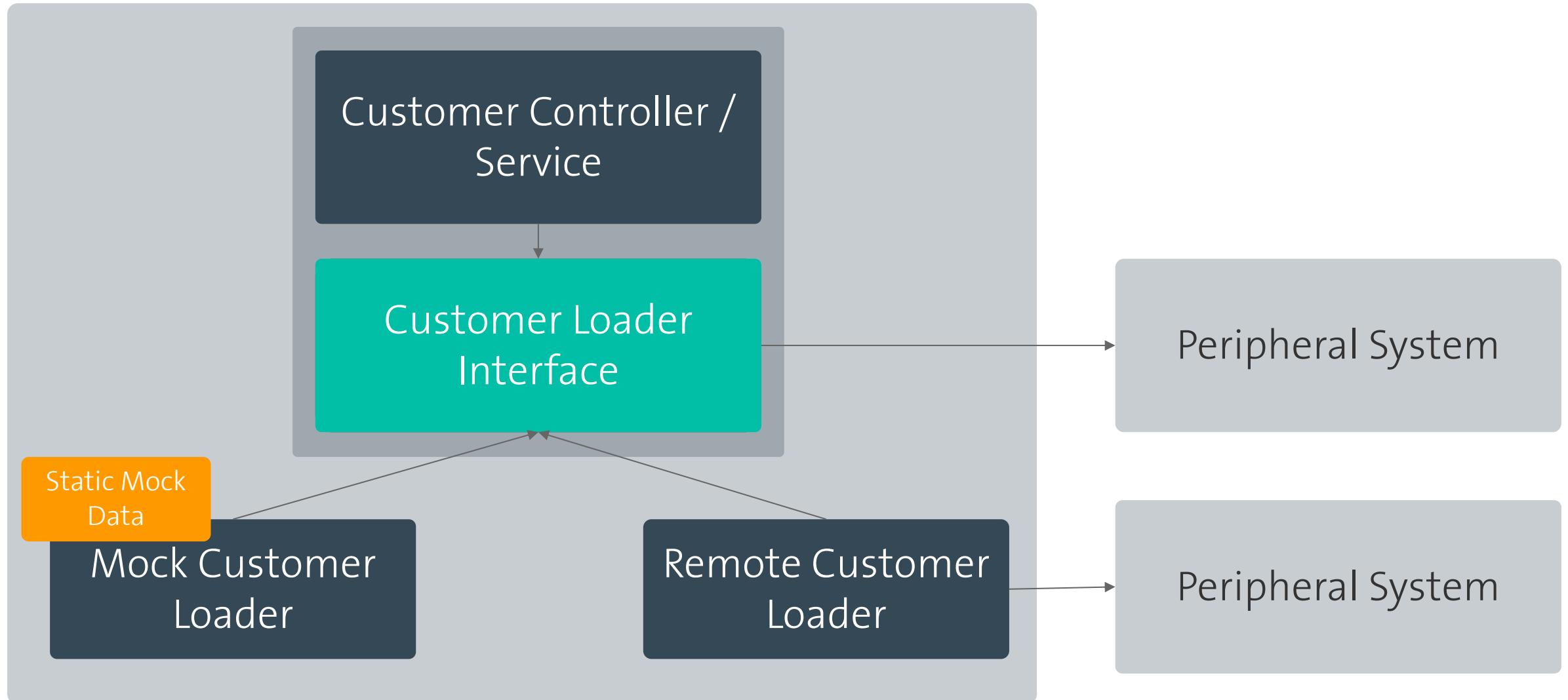
0.01 % The rest



Spring Basics

Profiles and other Conditions

Spring Basics / Profiles and other Conditions



Spring Basics / Profiles and other Conditions

```
@Component
```

```
@Profile("prod")
```

```
public class RemoteCustomerLoader implements CustomerLoader {
```

```
}
```

This component
is only loaded if
one of the active
profiles is
“prod”.

Yes, you can
have more
than one
active profile.

Spring Basics / Profiles and other Conditions

```
@Component
```

```
@Profile("default")
```

```
public class MockCustomerLoader implements CustomerLoader {
```

```
}
```

If no active profile is set, the “default” profile is active.

But you can also set the default profile explicitly.

Spring Basics / Profiles and other Conditions

```
@Component
```

```
@Profile({"dev", "local"})
```

```
public class MockCustomerLoader implements CustomerLoader {
```

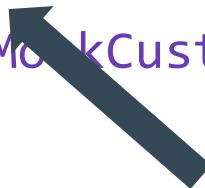
```
}
```

This component
is only loaded if
one of the active
profiles is “dev”
OR “local”.

No, @Profile
does not
support AND.

Spring Basics / Profiles and other Conditions

```
@Component  
@Profile("!prod")  
public class MockCustomerLoader implements CustomerLoader {  
}
```



**This component
is only loaded if
none of the
active profiles is
“prod”.**

Spring Basics / Profiles and other Conditions

@Component

```
public class RemoteCustomerLoader implements CustomerLoader {
```

}

This would lead to a
NoUniqueBeanDefinitionException

@Component

```
public class MockCustomerLoader implements CustomerLoader {
```

}



Spring Basics / Profiles and other Conditions

```
@Component  
@Profile("prod")  
public class RemoteCustomerLoader implements CustomerLoader {
```

```
}
```

**Furthermore you can
define custom
conditions**

```
@Component  
@Conditional(LocalAndMockProfileCondition.class)  
public class MockCustomerLoader implements CustomerLoader {  
  
}
```

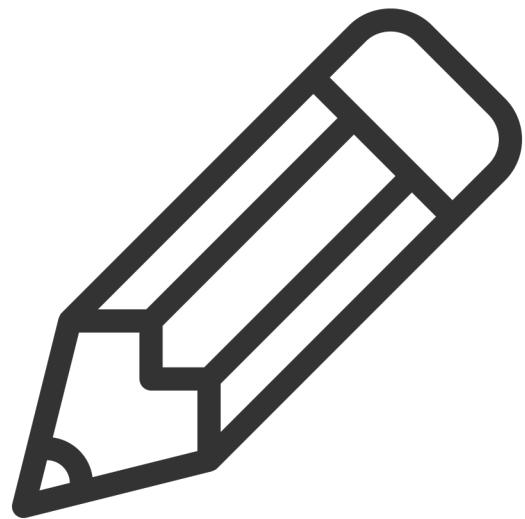
Spring Basics / Profiles and other Conditions

```
public class LocalAndMockProfileCondition implements Condition {  
    @Override  
    public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {  
        List<String> activeProfiles = asList(context.getEnvironment().getActiveProfiles());  
        return activeProfiles.contains("local") && activeProfiles.contains("mock");  
    }  
}
```

**And you have access
to the container**

**You can define any
logic here**

Summary: With conditions
we can control whether a
`@Component` is in our box
or not.



Spring Basics

Configuration

Configuration in Spring

==

XML?

No!

We just developed a Spring application without a single line of XML!

But how did we do the configuration?

We used the **annotation based configuration**:

`@ComponentScan`

`@Component` (and its subtypes)

`@Autowired`

So there are **XML (avoid it)**
and **annotations (prefer it)**,
to configure Spring / our
box...

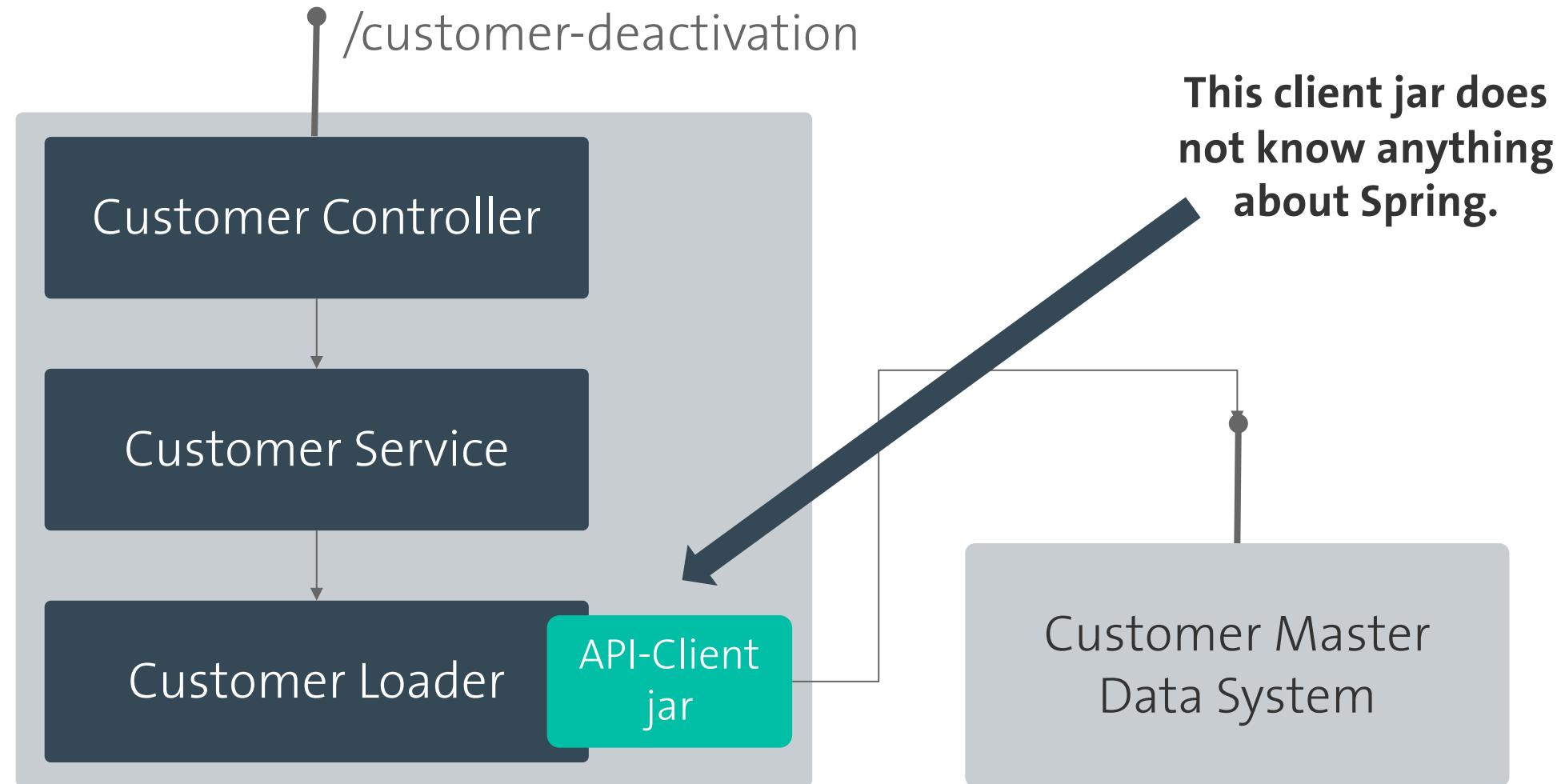
...but there is a third option.

...but there is a third option.

Java-based configuration

Why do I need a third option?

Spring Basics / @Configuration



We can define further Beans in
theses classes with Bean
factory methods (annotated
with `@Bean`)

This annotation is
also a
`@Component`

`@Configuration`

Each Spring application can
have multiple classes
annotated with
`@Configuration`.

There are many more annotations to
configure Spring (`@Enable...`). These
annotations should be placed on
`@Configuration` classes

Spring Basics / Configuration

```
import com.acme.customermasterdata.api.CustomerMasterDataClient;
```

```
@Configuration
```

```
public class CustomerMasterDataApiConfiguration {
```

```
    @Bean
```

```
    public CustomerMasterDataClient cmdClient() {
```

```
        return new CustomerMasterDataClient();
```

```
}
```

```
}
```



From third
party library
jar



Can be any
name.

Spring Basics / Configuration

```
import com.acme.customermasterdata.api.CustomerMasterDataClient;  
  
@Service  
public class CustomerLoader {  
    private final CustomerMasterDataClient customerMasterDataClient;  
  
    @Autowired  
    public CustomerLoader(CustomerMasterDataClient customerMasterDataClient) {  
        this.customerMasterDataClient = customerMasterDataClient;  
    }  
}
```

And now we can inject the client class from the 3rd party lib with our regular mechanisms



@Component (or rather @Service,...)

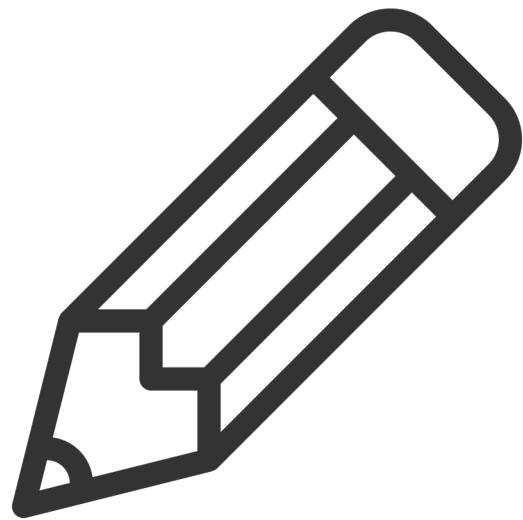
- Class-level annotation
- @Component is used to auto-detect beans using classpath scanning.
- Under normal circumstances we use this approach for defining our bean.

annotation-based config

@Bean

- Method-level annotation
- Decouples bean declaration and class definition.
- With @Bean you can add classes from non Spring-aware libraries to the application context (i.e. third party libraries).
- Can be used to configure Spring-aware libraries.

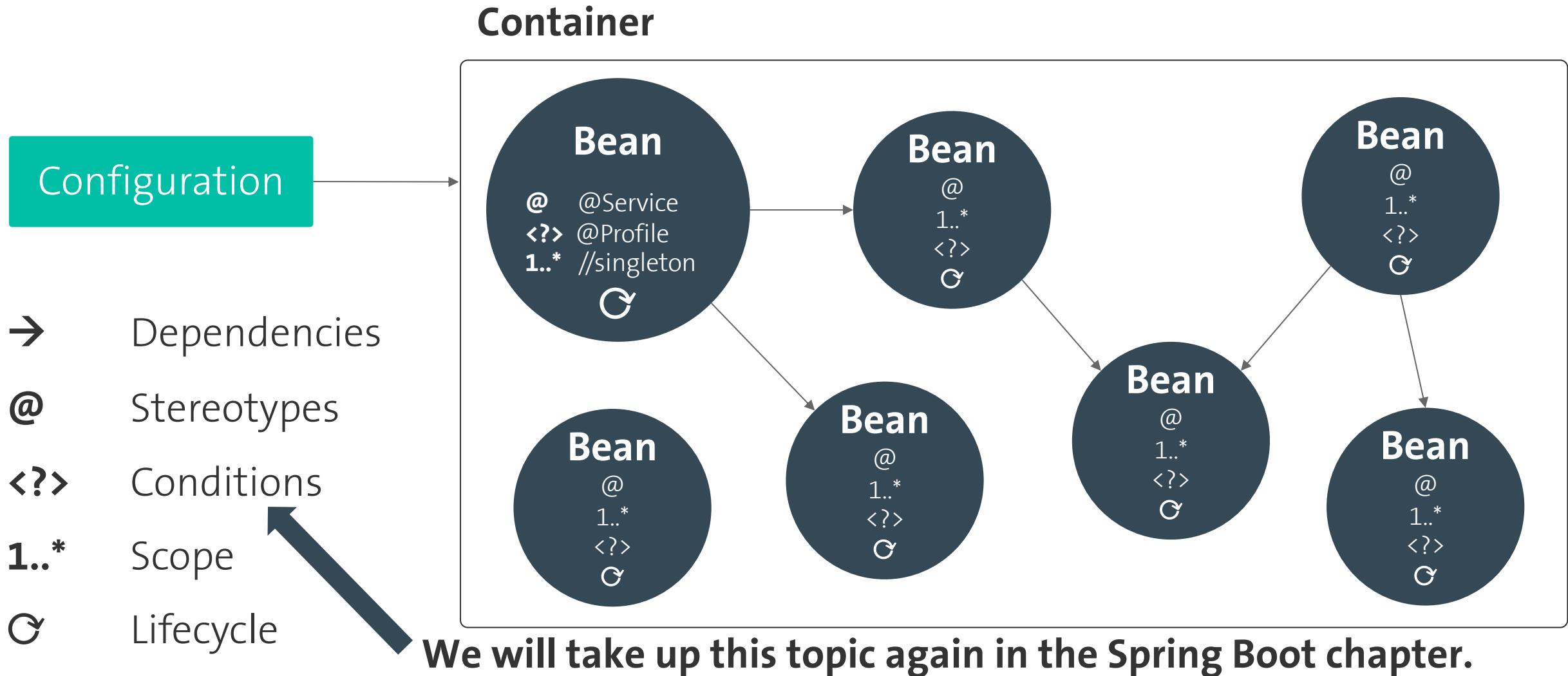
Java-based config



Spring Basics

ApplicationContext Summary

Spring Basics / Imagine a box



Inversion of
Control

Not the bean itself is controlling
the instantiation or location of its
dependencies.

The container will manage that
and injects those dependencies
when it creates the bean
(Dependency Injection).

General name for the
“box” in Dependency
Injection frameworks

(IoC) Container

VS.

(Application) Context

In the Spring world, the
container is represented by
the so called application
context.

FOR TODAY

Box == (IoC) Container == (Application) Context

Inversion of Control == Dependency Injection

Spring Basics

Wrap-up

What is a bean?

An object managed by the
container.

Constructor-, Setter-, or Field Injection? And why?

Constructor Injection! Testing!

@Service or @Component? And Why?

@Service

- Annotation specific behavior
- Readability / Intention revealing
- You can use this it for your own purpose

I want to add a class (that is not under my control) to the application context. What annotation can support me here?

@Bean and @Configuration

Spring Basics / Wrap-up

- Prefer constructor injection over setter and field injection (especially in terms of unit testing).
- Always use the concrete stereotype (like @Service) instead of @Component.
- Use the request scope / prototype scope for all stateful beans and the singleton scope for stateless beans.
- There are three kinds of configuration: XML (**avoid this**), Annotations (**prefer this**) and Java (@Configuration and @Bean) (**use if necessary**)

?

Spring Web / MVC

Spring Web / MVC / Topics

- Introduction
- REST Services with Spring

Spring Web / MVC

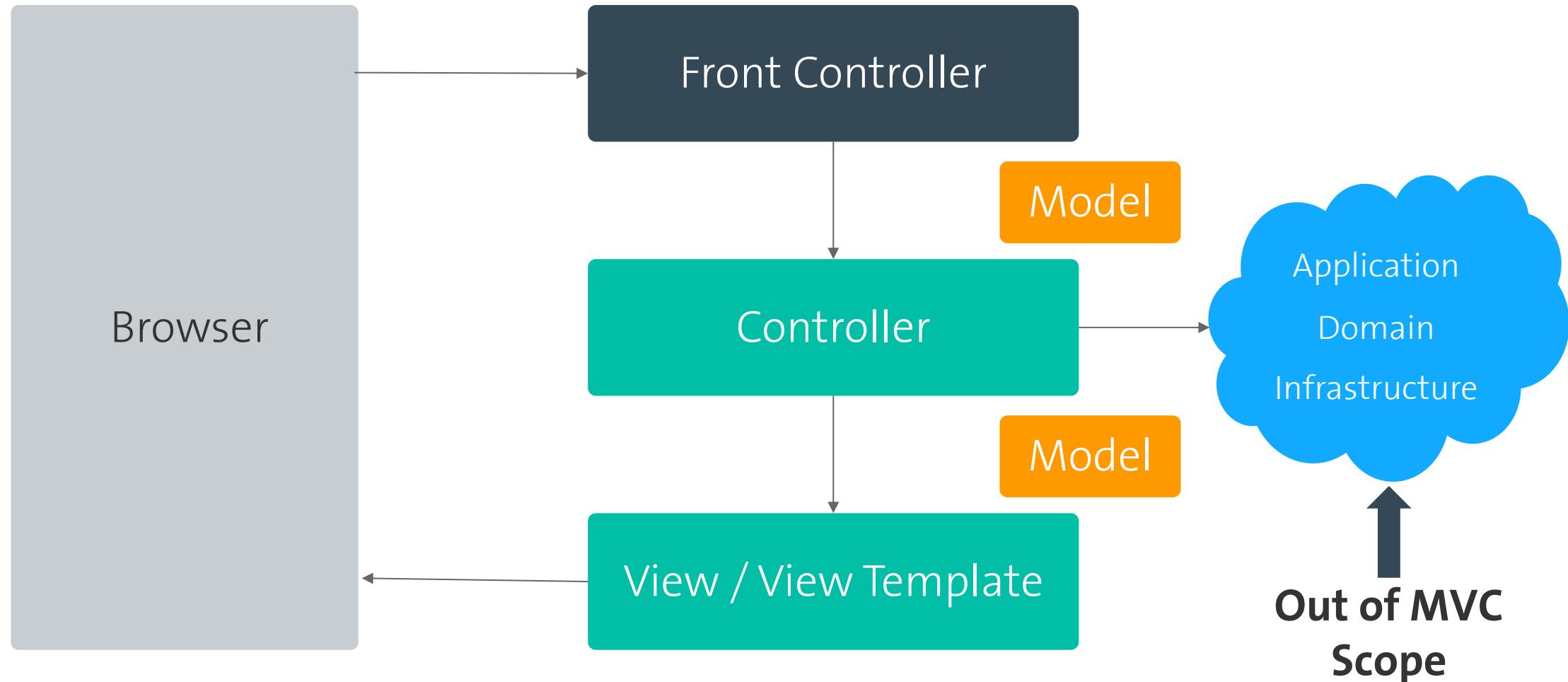
Introduction

“Spring Web MVC is the original **web framework** built on the **Servlet API** and has been included in the Spring Framework from the very beginning.”

(<https://docs.spring.io/>)

Spring Web / MVC / Model View Controller

**Dispatcher
Servlet**
(Provided by Spring)



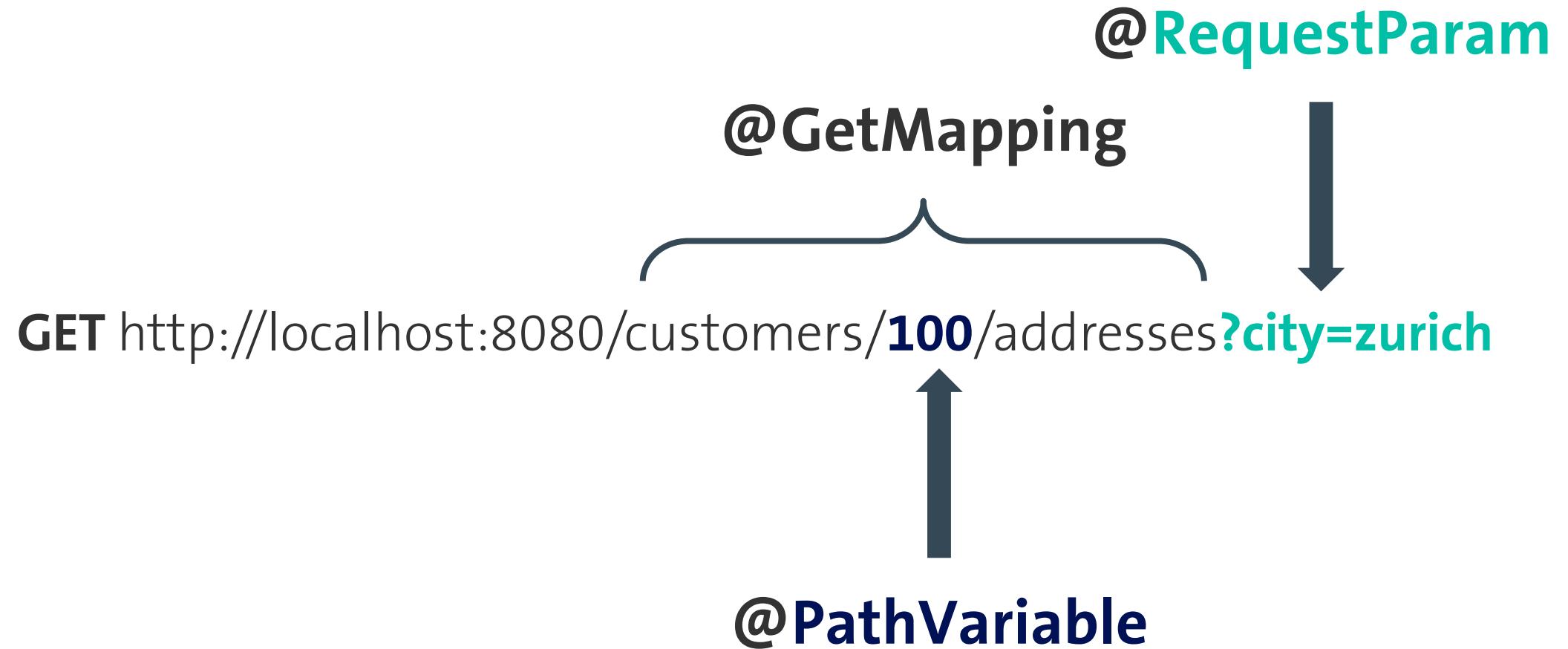
Spring Web / MVC

REST Services with Spring

Spring Web / MVC / Important Annotations

- `@RestController`
- `@RequestMapping`
- `@GetMapping`
- `@PostMapping`
- `@PutMapping`
- `@DeleteMapping`
- `@ResponseStatus`
- `@PathVariable`
- `@RequestParam`
- `@RequestBody`

Spring Web / MVC / Important Annotations



GET `http://localhost:8080/customers/100/addresses?city=zurich`

```
@RestController
```

```
public class CustomerRestController {
```

```
    @GetMapping("/customers/{id}/addresses")
```

```
    public List<CustomerDto> findBy(@PathVariable String id, @RequestParam String city) {
```

```
        //implementation
```

```
}
```

```
}
```

Spring Web / MVC / Important Annotations

@PostMapping

POST http://localhost:8080/customers

Content-Type: application/json

{

"name" = "ACME"

}

@RequestBody

POST `http://localhost:8080/customers`

```
@RestController
```

```
public class CustomerRestController {
```

```
    @PostMapping
```

```
        public void create(@RequestBody CustomerDto customer) {
```

```
            //implementation
```

```
}
```

```
}
```

Spring Web / MVC / Important Annotations

@PutMapping

PUT http://localhost:8080/customers/**100**

Content-Type: application/json

```
{  
    "name" = "ACME Inc."  
}
```

@PathVariable

@RequestBody

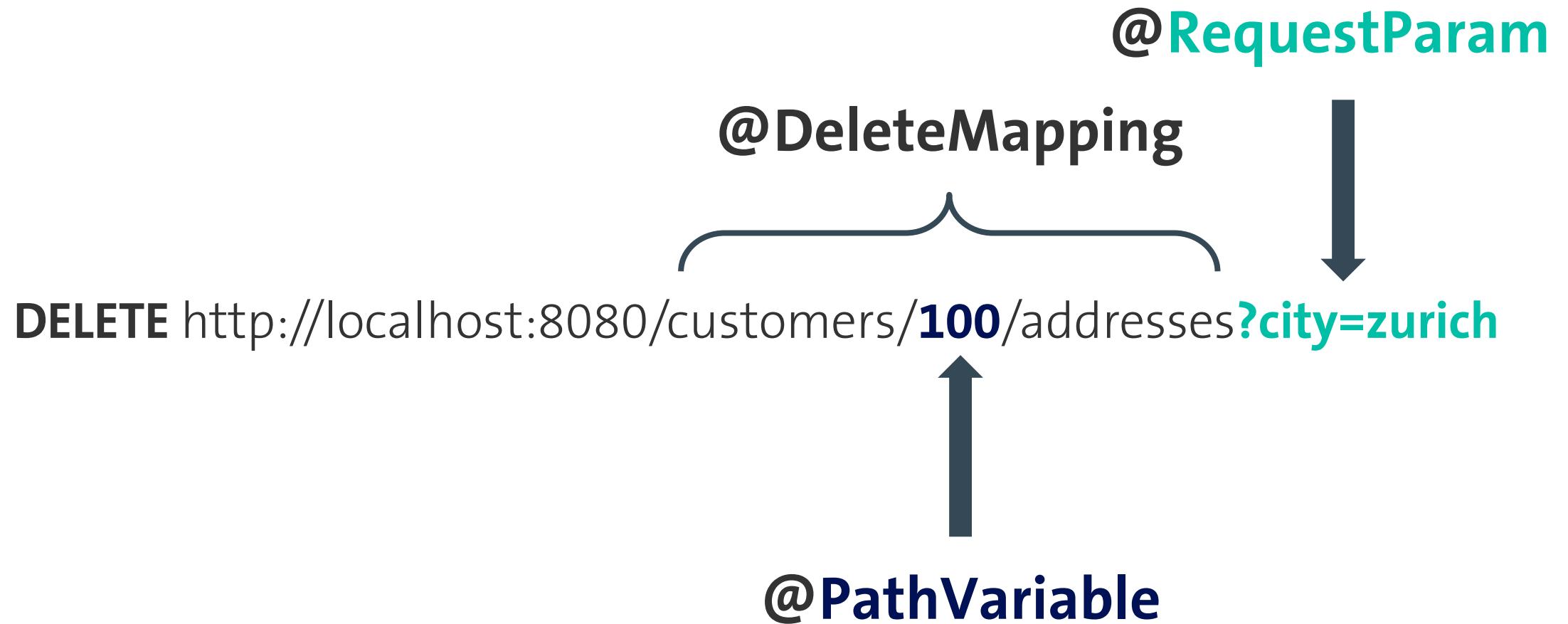
The diagram illustrates the mapping of a Spring REST API endpoint. It shows a **PUT** request to **http://localhost:8080/customers/100**. The path variable **100** is annotated with **@PathVariable**, indicated by a curly brace above it. The request body, which contains a JSON object with the key **"name"** and value **"ACME Inc."**, is annotated with **@RequestBody**, indicated by a curly brace below it. An arrow points from the **Content-Type** header to the JSON object, indicating the media type of the request body.

PUT http://localhost:8080/customers/**100**

@RestController

```
public class CustomerRestController {  
  
    @PutMapping("/customers/{id}")  
    public CustomerDto update(@PathVariable String id, @RequestBody CustomerDto customer) {  
        //implementation  
    }  
}
```

Spring Web / MVC / Important Annotations



DELETE http://localhost:8080/customers/**100**

```
@RestController
```

```
public class CustomerRestController {
```

```
    @DeleteMapping("/customers/{id}/addresses")
```

```
    public void delete(@PathVariable String customerId, @RequestParam String city) {
```

```
        //implementation
```

```
}
```

```
}
```

```
@RestController  
@RequestMapping("/customers")  
  
public class CustomerRestController {  
  
    @PostMapping  
    public CustomerDto create(@RequestBody CustomerDto customer) { ... }  
  
    @GetMapping  
    public List<CustomerDto> findBy(@RequestParam Optional<String> name) { ... }  
  
    @PutMapping("/{id}")  
    public CustomerDto update(@PathVariable String id, @RequestBody CustomerDto customer) { ... }  
  
    @DeleteMapping("/{id}")  
    @ResponseStatus(HttpStatus.NO_CONTENT)  
    public void delete(@PathVariable String customerId) { ... }  
}
```

@RequestMapping can be omitted (if no common path)

Optional indicates an optional request parameter

placeholder is required for @PathVariable

default is 200 (OK)

quick demo

Spring Web / MVC

Wrap-up

?

Spring Boot

Spring Boot / Topics

- What is Spring Boot?
- The first Spring Boot application
- Starters
- Auto Configuration
- Profiles, Properties and Externalized Configuration

Spring Boot?

Spring Boot is a framework for web applications!

No!

Spring Boot works with code generation!

No!

Spring Boot is a framework for building microservices!



Spring Boot / What is Spring Boot?

Create stand-alone Spring applications

Provide opinionated ‘starter’ dependencies to simplify your build configuration

Provide production-ready features such as metrics, health checks and external configuration

Embed Tomcat, Jetty or Undertow directly (no need to deploy WAR files)

Automatically configure 3rd party libraries whenever possible

Absolutely no code generation and no requirements for XML configuration

Spring Boot / What is Spring Boot?

“Spring embraces flexibility and is **not opinionated** about how things should be done. It supports a wide range of application needs with different perspectives.”

(<https://docs.spring.io/spring/docs/5.1.3.RELEASE/spring-framework-reference/overview.html#overview-philosophy>)



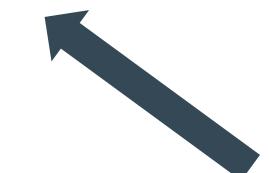
Spring Framework

“We take an **opinionated** view of the Spring platform and third-party libraries, so that you can get started with minimum fuss.” (<https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#getting-started-introducing-spring-boot>)



Spring Boot

“Spring Boot is [...] a configuration wrapper around all of the Spring framework.”



And a lot of other
spring projects

Spring Boot

The first Spring Boot application

Spring Boot /Initializer

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

Spring Boot / @SpringBootApplication

```
package com.springfundamentals;

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

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

Spring Boot / `@SpringBootApplication`

`@SpringBootApplication`

`@ComponentScan`

- Scans for components in current package and all child packages
- Same as we already saw in the Spring Basics chapter

`@EnableAutoConfiguration`

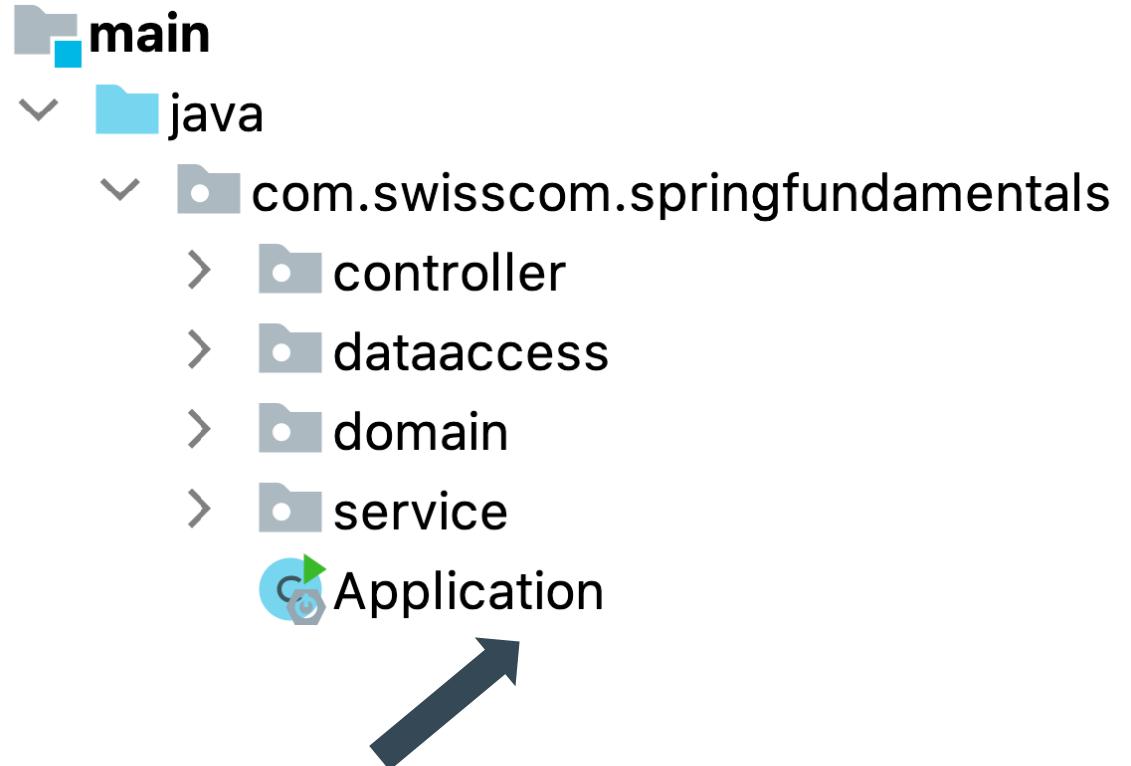
- Spring Boot specific annotation
- Enables the auto configuration of Spring Boot
- See Chapter “Spring Boot - AutoConfiguration”

`@SpringBootConfiguration`

- Spring Boot specific annotation
- Wraps `@Configuration`
- It is a bit more convenient than `@Configuration`: i.e. in spring boot tests the configuration can be found automatically

Spring Boot / Project Structure

- Locate your main application in a root package above other classes.
- Don't use the default package!



This class should contain the main method as well as the `@SpringBootApplication` annotation

Spring Boot / and Gradle

Provides Spring Boot support in Gradle, allowing you to package executable jar or war archives and run Spring Boot applications

```
plugins { id 'org.springframework.boot' version '3.1.3.' }  
plugins { id 'io.spring.dependency-management' version '1.1.3' }
```

This version also serves as spring boot version for the dependency management plugin

Provides Maven-like dependency management / BOM

<https://github.com/spring-projects/spring-boot/blob/v3.1.3/spring-boot-project/spring-boot-dependencies/build.gradle>

BON?

Spring Boot

Starters

“Provide **opinionated**
'starter' dependencies to
simplify your build
configuration”

(<https://spring.io/projects/spring-boot>)



Spring Boot / Starters

```
dependencies {  
  
    implementation 'org.springframework:spring-core:5.1.3.RELEASE' ←  
    implementation 'org.springframework:spring-context:5.1.3.RELEASE'  
    implementation 'org.springframework:spring-webmvc:5.1.3.RELEASE'  
    implementation 'com.fasterxml.jackson.core:jackson-core:2.9.8' ←  
    implementation 'com.fasterxml.jackson.core:jackson-databind:2.9.8'  
    implementation 'com.fasterxml.jackson.datatype:jackson-datatype-jsr310:2.9.8'  
  
    providedCompile 'javax.servlet:javax.servlet-api:4.0.1' ←  
}
```

**Do you remember the build.gradle
file from the MVC project?**

To make a long story short...

Starters are **collections** of **dependencies** that are compatible with each other and serve a **common purpose**. Starters **should not contain logic or code**.

Spring Boot Starter Overview

hands-on

<https://github.com/spring-fundamentals/hands-on>

How much "Spring Boot" did
you do during the exercise?

Spring Boot

Auto Configuration

Spring Boot / Auto Configuration

“Spring embraces flexibility and is **not opinionated** about how things should be done. It supports a wide range of application needs with different perspectives.”

(<https://docs.spring.io/spring/docs/5.1.3.RELEASE/spring-framework-reference/overview.html#overview-philosophy>)

Do you remember this?

“We take an **opinionated** view of the Spring platform and third-party libraries, so that you can get started with minimum fuss.” (<https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#getting-started-introducing-spring-boot>)

 **Spring Framework**

 **Spring Boot**

Spring Boot / Auto Configuration

```
@Configuration  
@EnableWebMvc  
public class AppConfig implements WebMvcConfigurer {  
  
    @Override  
    public void configureDefaultServletHandling(DefaultServletHandlerConfigurer configurer) {  
        configurer.enable();  
    }  
  
    @Override  
    public void configureMessageConverters(List<HttpMessageConverter<?>> converters) {  
        Jackson2ObjectMapperBuilder builder = new Jackson2ObjectMapperBuilder();  
        builder.indentOutput(true).dateFormat(new SimpleDateFormat("yyyy-MM-dd"));  
        converters.add(new MappingJackson2HttpMessageConverter(builder.build()));  
    }  
}
```

And do you remember this configuration class?



So we can summarize...

- Spring Framework offers a lot of freedom.
- But in many applications we have to configure exactly the same things with exactly the same settings.
- And for all these configurations we have approximately an infinite number of possibilities to do so.

And this is where the auto configuration of Spring Boot comes in.

Let's start our application
with enabled debug output.

Spring Boot / Auto Configuration

JmsAutoConfiguration:

Did **not** match:

- @ConditionalOnClass did not find required class
`'javax.jms.Message'` (OnClassCondition)

Spring Boot / Auto Configuration

JacksonAutoConfiguration matched:

- @ConditionalOnClass found required class
`'com.fasterxml.jackson.databind.ObjectMapper'`
(OnClassCondition)

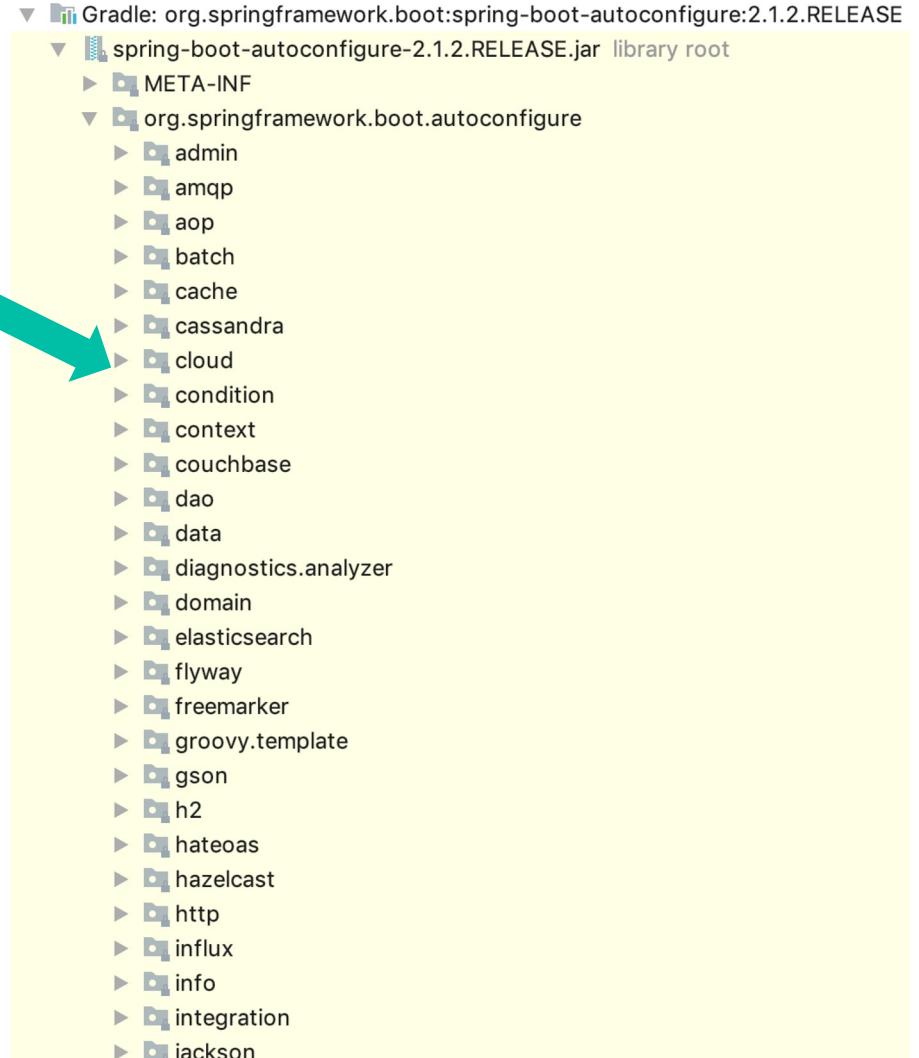
But where are all these conditions
and configurations are from?

Let's have a look in to the
spring-boot-autoconfigure
dependency

Spring Boot / Auto Configuration

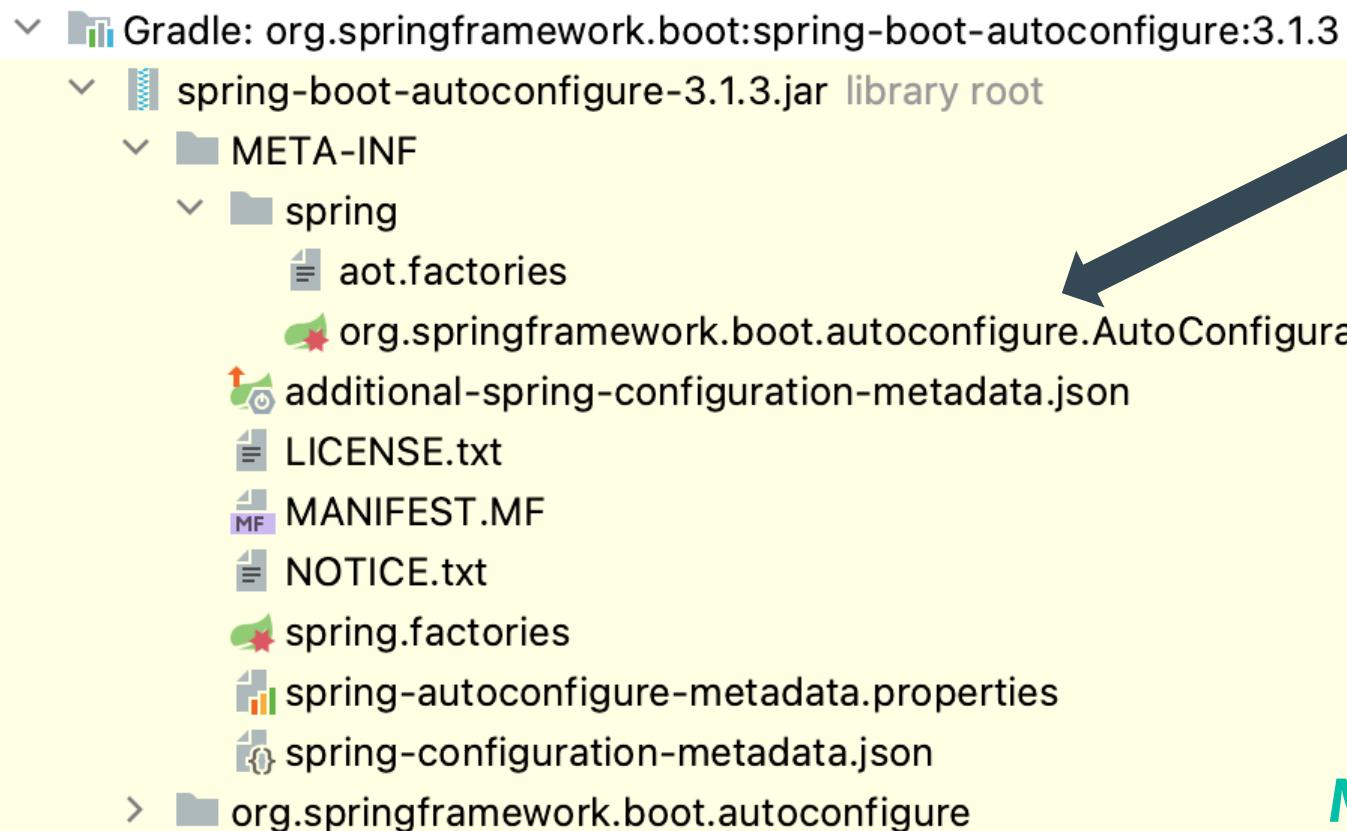
In addition, there is package that contains further annotations that are based on the `@Conditional` approach, that we discussed in the basic chapter.

There is a whole bunch of configuration classes for various frameworks. These classes are more or less reflecting the opinionated view on the Spring framework.



And how does Spring know that it
should consider all these
@Configuration classes during
context initialization?

Spring Boot / Auto Configuration



Spring checks for the presence of `META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports` files in your classpath.

Yes, you can create your own dependencies with such a file in the **META-INF** folder, where you can define configuration classes that should be auto-configured.

Spring Boot / Auto Configuration

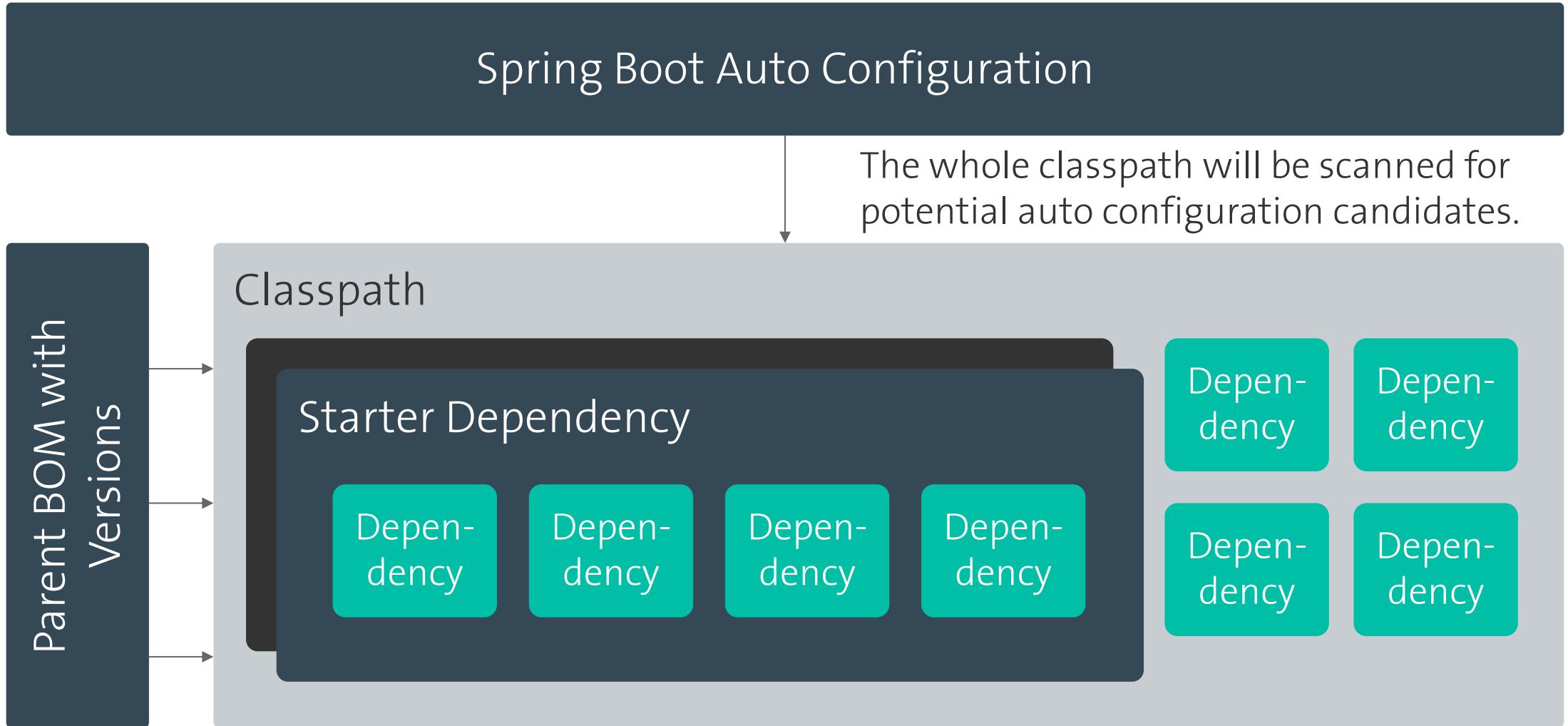


And in this file, there is one section with a whole bunch of configuration classes. All these classes will be considered and their conditions will be evaluated during context initialization

```
# Auto Configure
org.springframework.boot.autoconfigure.admin.SpringApplicationAdminJmxAutoConfiguration
org.springframework.boot.autoconfigure.aop.AopAutoConfiguration
...
org.springframework.boot.autoconfigure.data.elasticsearch.ElasticsearchAutoConfiguration
org.springframework.boot.autoconfigure.flyway.FlywayAutoConfiguration
...
org.springframework.boot.autoconfigure.gson.GsonAutoConfiguration
...
org.springframework.boot.autoconfigure.jackson.JacksonAutoConfiguration
...
org.springframework.boot.autoconfigure.jms.JmsAutoConfiguration
```

Starters vs. Auto Configuration

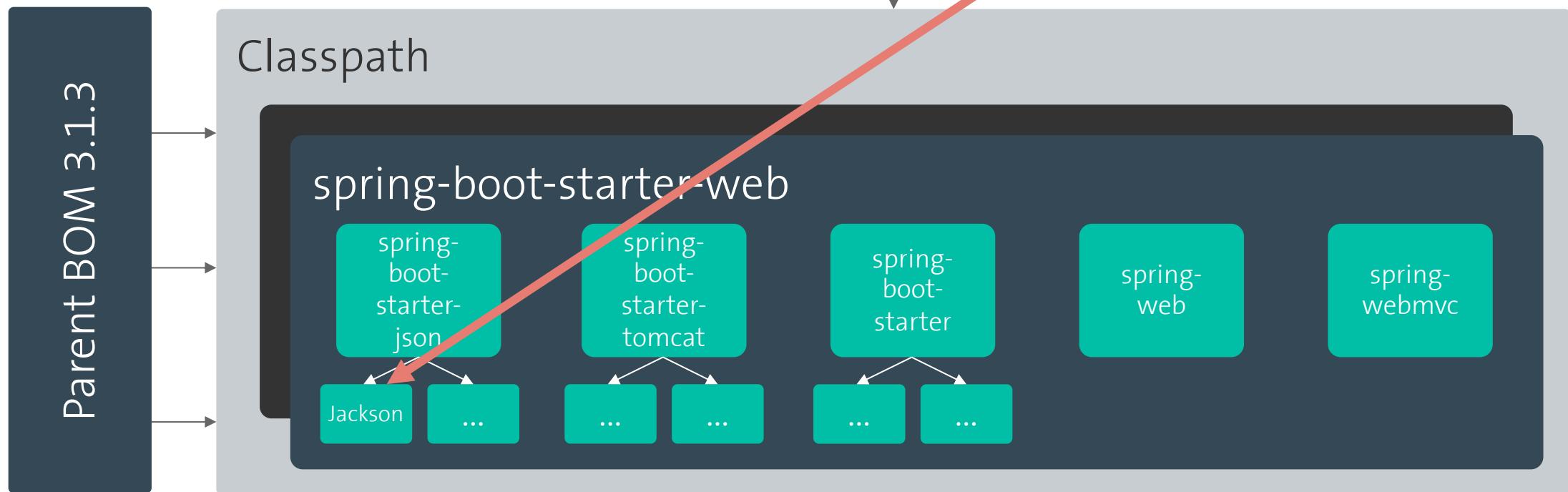
Spring Boot / Auto Configuration



Spring Boot / Auto Configuration

Spring Boot Auto Configuration

```
@ConditionalOnClass(ObjectMapper.class)  
class JacksonAutoConfiguration
```



Starters and auto configuration
are decoupled mechanisms,
but in combination they are
pretty elegant.

Spring Boot “Magic”

=

BOM + Starters + Auto Configuration

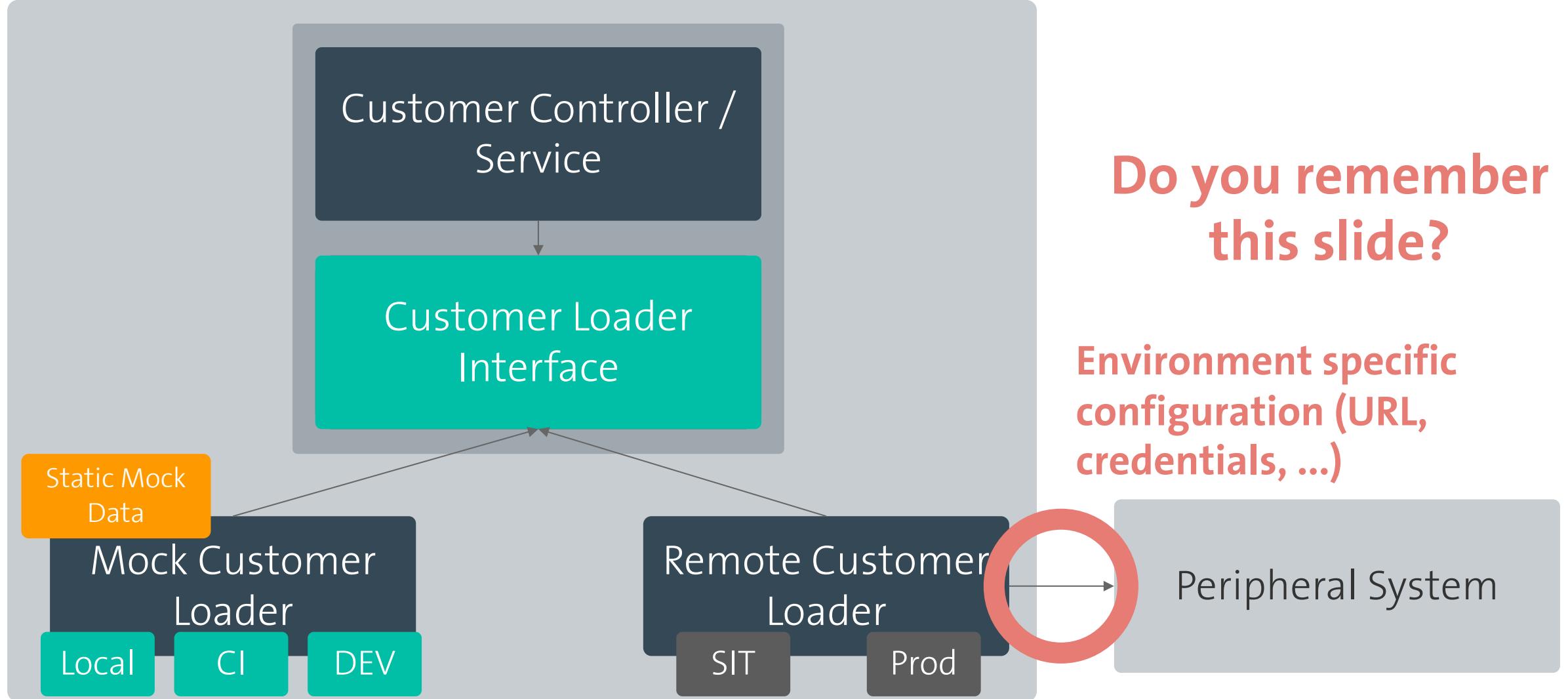
Spring Boot

Profiles, Properties and Externalized Configuration

Spring Boot / Profiles, Properties and Externalized Configuration

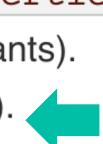
“Spring Boot lets you **externalize your configuration** so that you can work with the **same application** code in **different environments**.”

Spring Boot / Profiles, Properties and Externalized Configuration



But where can I define these
externalized configuration
things?

Spring Boot / Profiles, Properties and Externalized Configuration

1. Devtools global settings properties on your home directory (`~/.spring-boot-devtools.properties` when devtools is active).
2. `@TestPropertySource` annotations on your tests.
3. `properties` attribute on your tests. Available on `@SpringBootTest` and the test annotations for testing a particular slice of your application.
4. Command line arguments.
5. Properties from `SPRING_APPLICATION_JSON` (inline JSON embedded in an environment variable or system property).
6. `ServletConfig` init parameters.
7. `ServletContext` init parameters.
8. JNDI attributes from `java:comp/env`.
9. Java System properties (`System.getProperties()`).
10. OS environment variables.
11. A `RandomValuePropertySource` that has properties only in `random.*`.
12. Profile-specific application properties outside of your packaged jar (`application-{profile}.properties` and YAML variants).
13. Profile-specific application properties packaged inside your jar (`application-{profile}.properties` and YAML variants). 
14. Application properties outside of your packaged jar (`application.properties` and YAML variants).
15. Application properties packaged inside your jar (`application.properties` and YAML variants). 
16. `@PropertySource` annotations on your `@Configuration` classes.
17. Default properties (specified by setting `SpringApplication.setDefaultProperties()`).

**Spring Boot consider
property sources in the
following order**

We will have a closer
look into these property
sources

Spring Boot / Profiles, Properties and Externalized Configuration

.properties

vs

.yml

Spring Boot / Profiles, Properties and Externalized Configuration

```
pizza.inventory.mock=false
pizza.inventory.url=https://api.pizza-inventory.com
pizza.inventory.user=admin
pizza.inventory.password=tester11
```

.properties

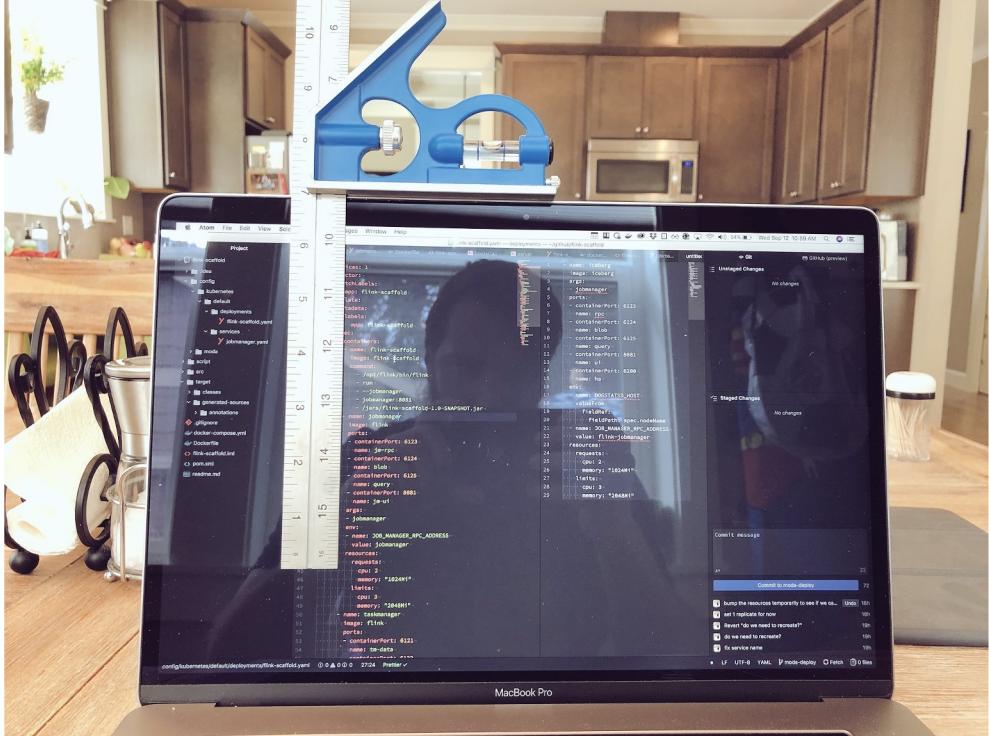
```
pizza:
  inventory:
    mock: false
    url: https://api.pizza-inventory.com
    user: admin
    password: tester11
```

.yml

And what should I
use?

.properties!

Spring Boot / Profiles, Properties and Externalized Configuration



YAML files doesn't provide
a good UX

The screenshot shows an IDE interface with a search results window titled "Find in Files". It displays three matches found in three files:

- RemoteCustomerLoader.java**:
@Value("\${customer.loader.url}")
customer.loader.url=http://customer-system/customers
customer.loader.url=http://customer-system-vega/customers
- application-prod.properties**:
customer.loader.url=http://customer-system/customers
- application-vega.properties**:
customer.loader.url=http://customer-system-vega/customers

The code editor below shows the `RemoteCustomerLoader.java` file with the `@Value("${customer.loader.url}")` annotation highlighted. The IDE also shows other annotations like `@Override` and methods like `findbyId`.

You can search for properties in
.properties files

Spring Boot / Profiles, Properties and Externalized Configuration

application.properties

Examples: App information like name, context or port.

The properties in **application.properties** file are loaded independently of the active profile. However, the properties can be overwritten in the profile specific files (see property source order).

profile name



application-prod.properties

Examples: Url and credentials from peripheral systems.

If there are explicitly activated profiles, then properties from **application-{profile}.properties** are loaded.

application-default.properties

No specific usage. Default can be used as local profile.

If no profiles are explicitly activated, then properties from **application-default.properties** are loaded.

Spring Boot / Profiles, Properties and Externalized Configuration

All **application.properties** and **application-{profile}.properties** files should be saved to **src/main/resources**. Spring Boot will find them there automatically.

Spring Boot / Profiles, Properties and Externalized Configuration

Property values (independently from the property source) can be injected directly into your beans by using the **@Value** annotation.

Spring Boot / Profiles, Properties and Externalized Configuration

```
@Service  
public class PizzaInventoryService {  
  
    @Value("${pizza.inventory.url}")  
    private String url;  
  
    @Value("${pizza.inventory.user}")  
    private String user;  
  
    @Value("${pizza.inventory.password}")  
    private String password;  
}
```

**Field injection
with @Value**

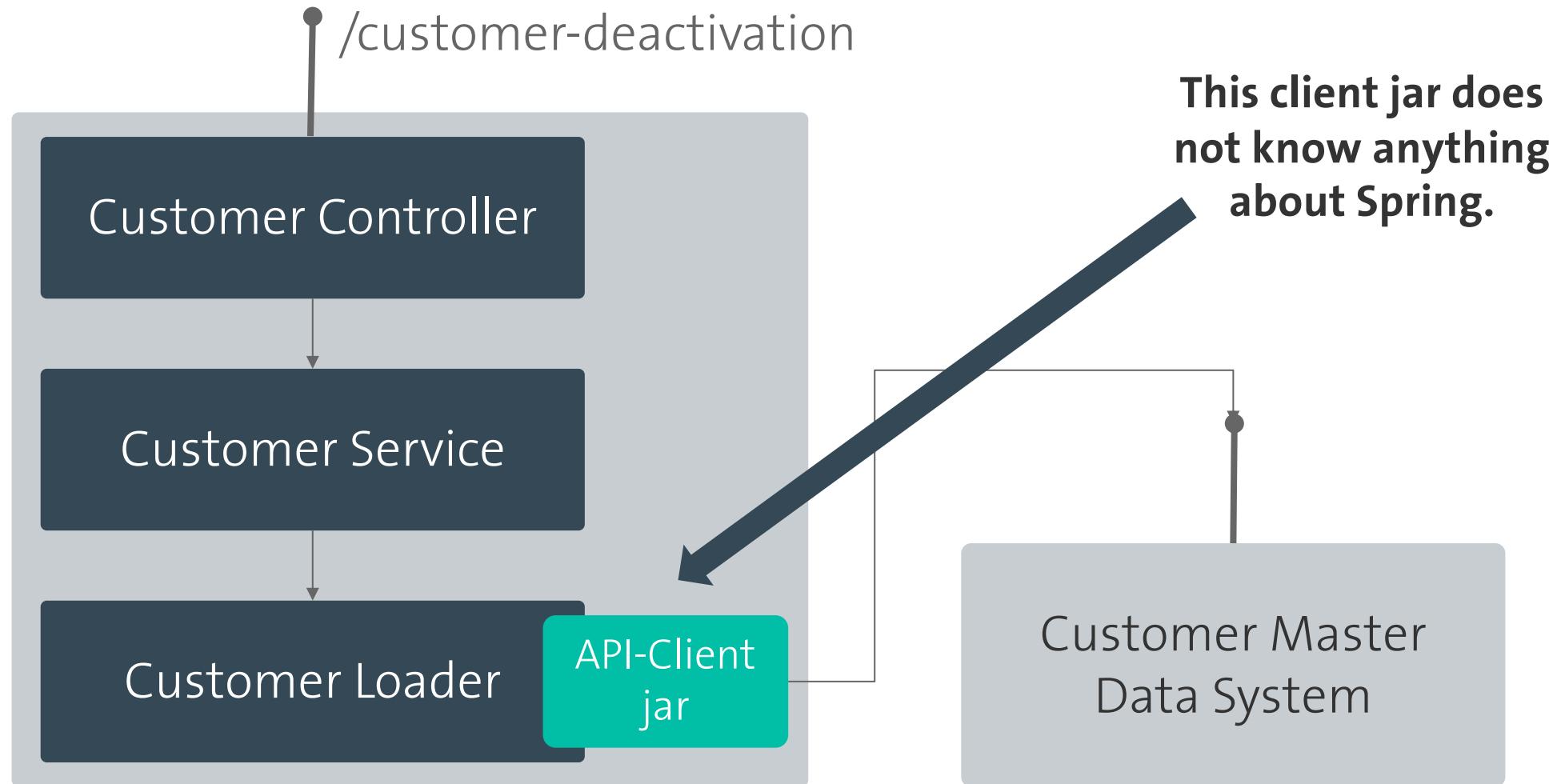
hands-on

<https://github.com/spring-fundamentals/hands-on>

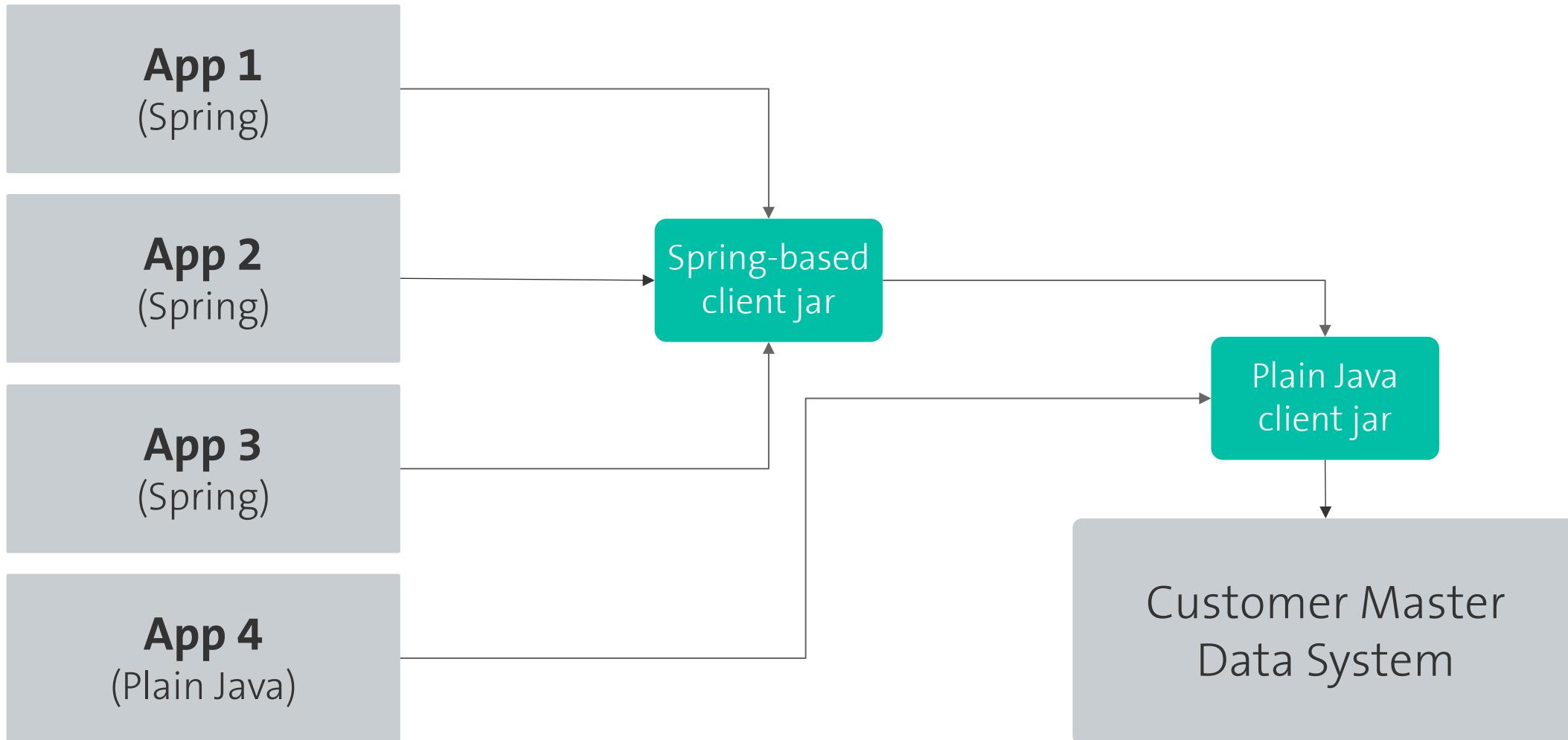
Spring Boot

(Auto) Configuration Sample

Spring Boot / Auto Configuration Sample



Spring Boot / Auto Configuration Sample



Spring Boot

Wrap-up

Spring Boot / Wrap-Up

- Put your `@SpringBootApplication` in your Top-Level package
- Don't use the default package
- If you have a common problem, there will be a solution for Spring (Boot). So check it out before building your own proprietary solution.
- It is helpful to understand at least the basics of Spring Boot Auto Configuration, because sometimes it really looks like magic (but it's not).
- Use Property files instead of yaml files.
- Use Starters
- Consider Actuator and DevTools

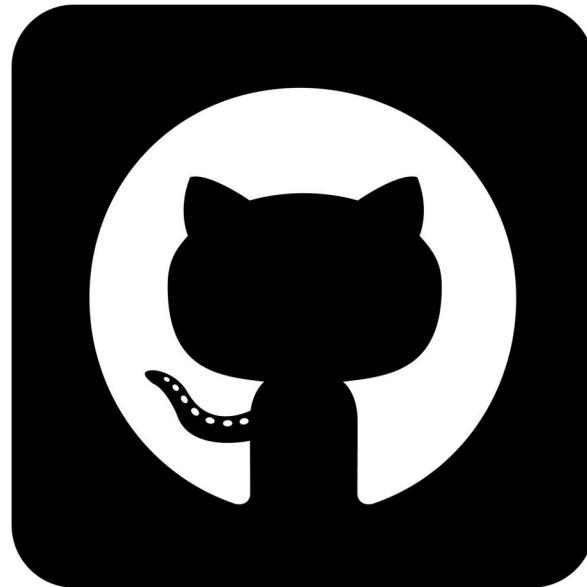
?

Wrap-up

Wrap-up / And why should I use Spring?

- Good and well-proven practices
- Huge ecosystem
- Good testing support
- No proprietary self-made solutions
- Easy knowledge transfer
- Easier to find new developers

Wrap-up / Resources



<https://github.com/spring-fundamentals>

Wrap-up / Further Links

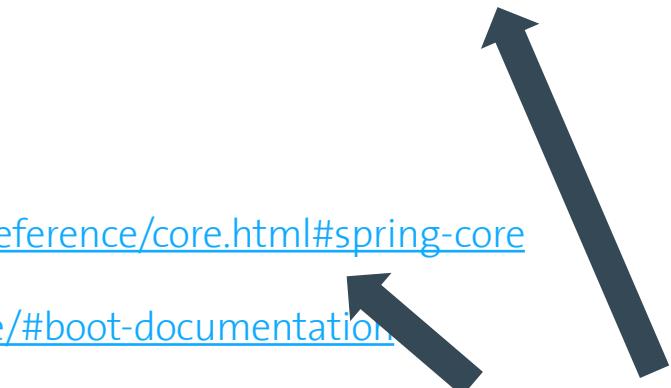
- <https://docs.spring.io/spring-boot/docs/current/reference/html/executable-jar.html>
- <https://www.baeldung.com/spring-web-contexts>
- <http://www.baeldung.com/integration-testing-in-spring>
- <https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-testing.html>
- <http://www.baeldung.com/spring-boot-testing>
- <https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-external-config.html#boot-features-external-config>
- <https://docs.developer.swisscom.com/buildpacks/java/getting-started-deploying-apps/gsg-spring.html>
- <https://docs.developer.swisscom.com/buildpacks/java/configuring-service-connections/spring-service-bindings.htm>
- <https://spring.io/blog/2015/04/27/binding-to-data-services-with-spring-boot-in-cloud-foundry>



**Baeldung is a great source
for various Spring topics.**

Wrap-up / Further Links

- <https://www.innoq.com/de/articles/2018/02/ddd-und-tdd-mit-spring-boot-2/#gezielteteststechnischerschichten>
- <https://www.webjars.org/>
- <https://www.baeldung.com/spring-boot-jasypt>
- <https://docs.spring.io/spring/docs/5.1.3.RELEASE/spring-framework-reference/core.html#spring-core>
- <https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-documentation>



A pretty good summary of the concepts of Spring and Spring Boot is provided by the Spring and Spring Boot docs itself (but I do not recommend it for learning Spring from scratch)

Wrap-up / Good Spring Talks

- Building better monoliths – Implementing modular applications with Spring by Oliver Gierke
<https://youtu.be/OSw4EthrT9E>
- REST beyond the obvious API design for ever evolving systems by Oliver Gierke
https://youtu.be/x_9OJKAv-ic
- Bootiful Testing by Josh Long
https://youtu.be/1W5_tOiwEAc
- Getting started with Spring Cloud by Josh Long
<https://youtu.be/SFDYdsIOvu8>
- Spring Boot - Behind the curtains: Autoconfiguration
<https://youtu.be/Ybfo8Dwactg>

Wrap-up / Good Spring Talks

- Reactive Spring by Josh Long
<https://youtu.be/zVNIZXf4BG8>
- Consuming Data Services with Spring Apps on Cloud Foundry - Scott Frederick
<https://youtu.be/g3DbtW5lwqY>
- Testing Spring Boot Applications - Phil Webb
<https://youtu.be/QjaoAWLIGGs>
- Talks4Nerds: Oliver Gierke - Cloud Native Applications
<https://youtu.be/sjSxOdV84gM>
- Talks4Nerds: Oliver Gierke - Spring Framework und Spring Boot 2.0
https://youtu.be/aTIY_RjTUv0