



Spring and Spring Boot

Fundamentals Handout

22.02.2021, Marius Reusch

Spring Basics

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.

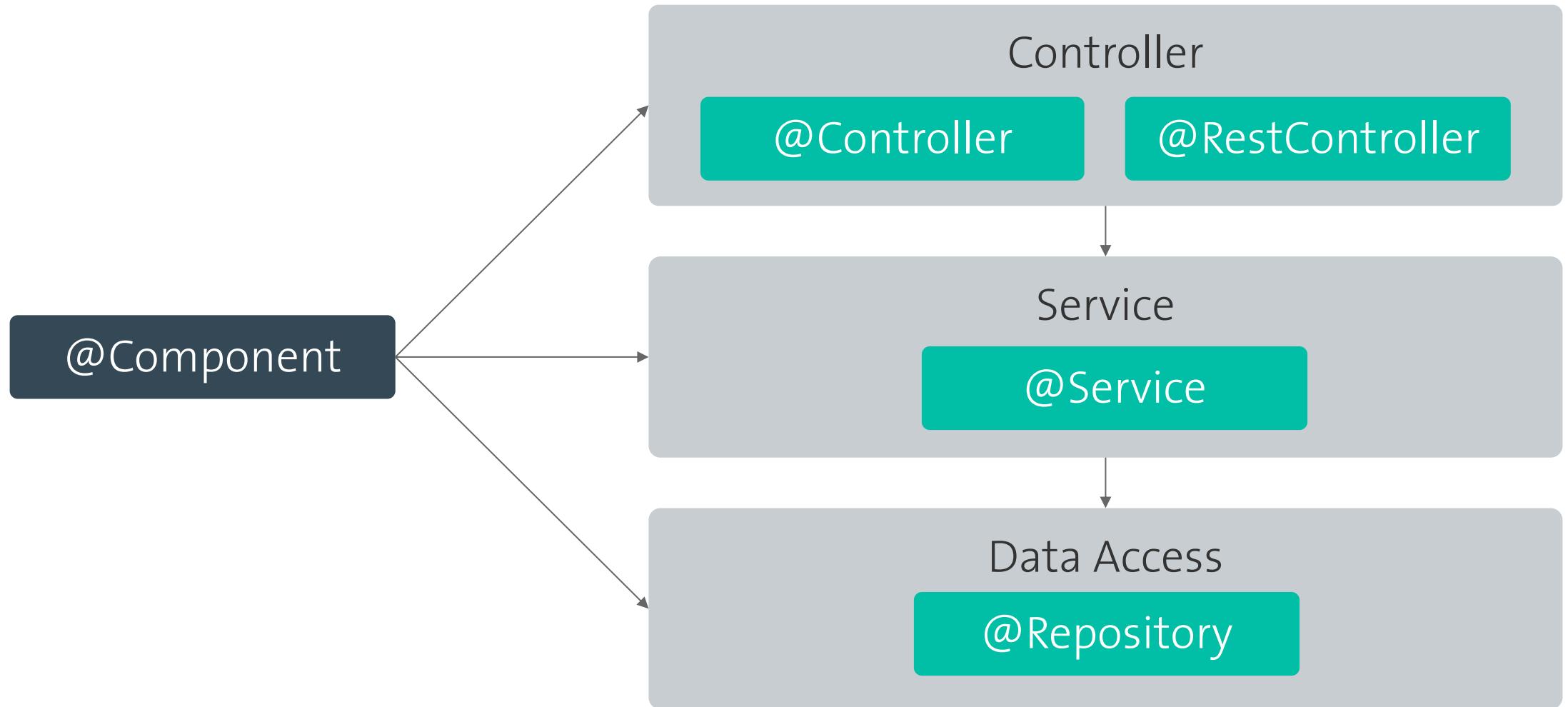
Spring Basics / Dependency Injection

```
@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 / Bean Stereotypes



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 {
```

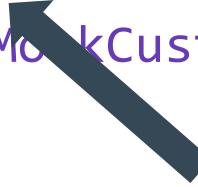
```
}
```

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 {
```

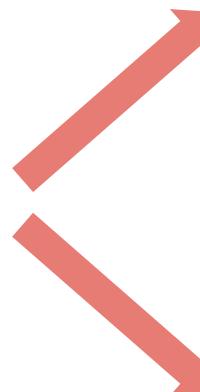
```
}
```

This would lead to a
NoUniqueBeanDefinitionException

@Component

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

```
}
```



Spring Basics / Profiles and other Conditions

@Component

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

}

@Component

@Primary



In case of multiple beans with no or the same profile conditions @Primary can serve as an additional condition. The bean that is annotated with @Primary has a higher precedence.

```
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**

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



Spring Basics / Configuration

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

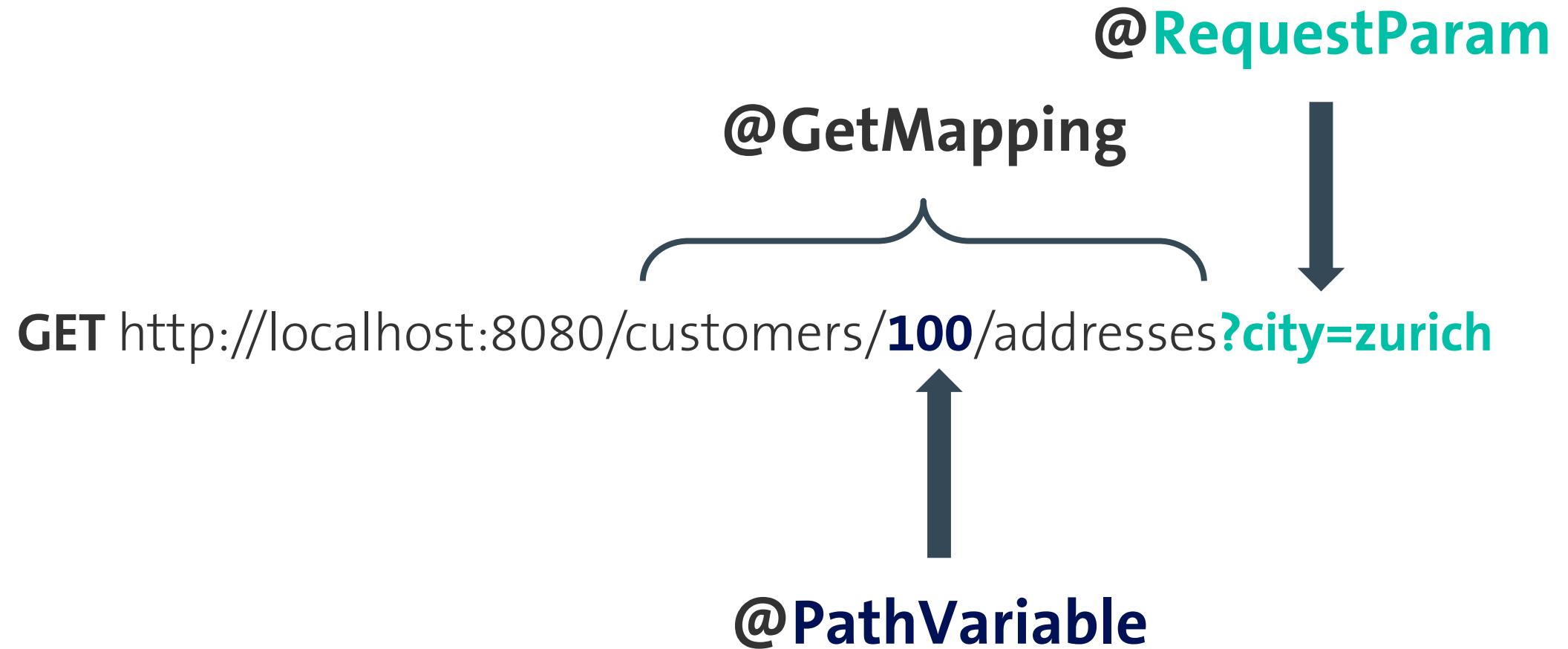
- Class-level annotation
- @Component is used to auto-detect beans using classpath scanning.
- Under normal circumstances we use this approach for defining our bean.

@Bean

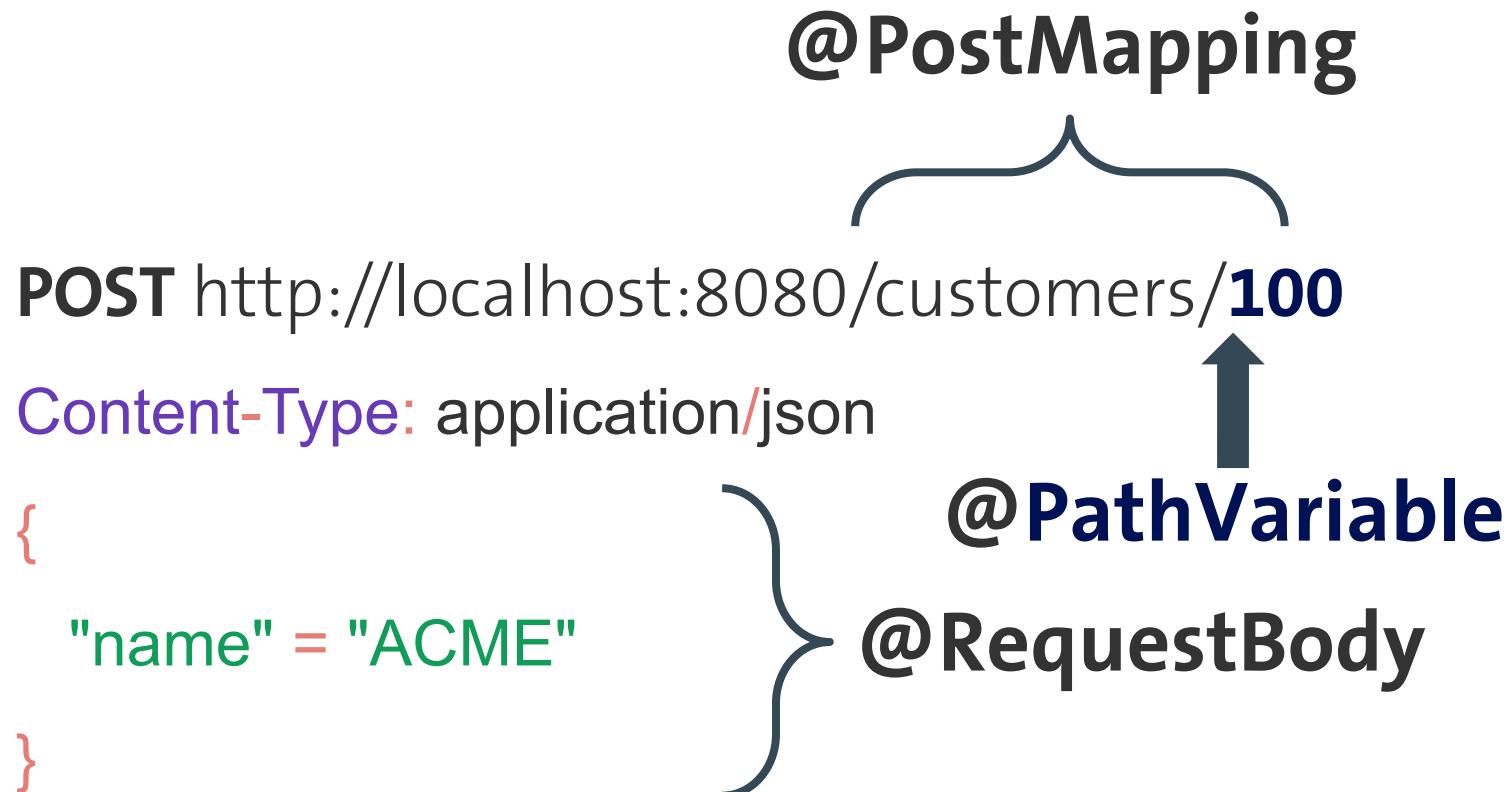
- Method-level annotation
- Decouples bean declaration and class definition.
- With @Bean you can add classes from non Spring-aware libraries to the application context (i.e. third party libraries).
- Can be used to configure Spring-aware libraries.

Spring Web / MVC

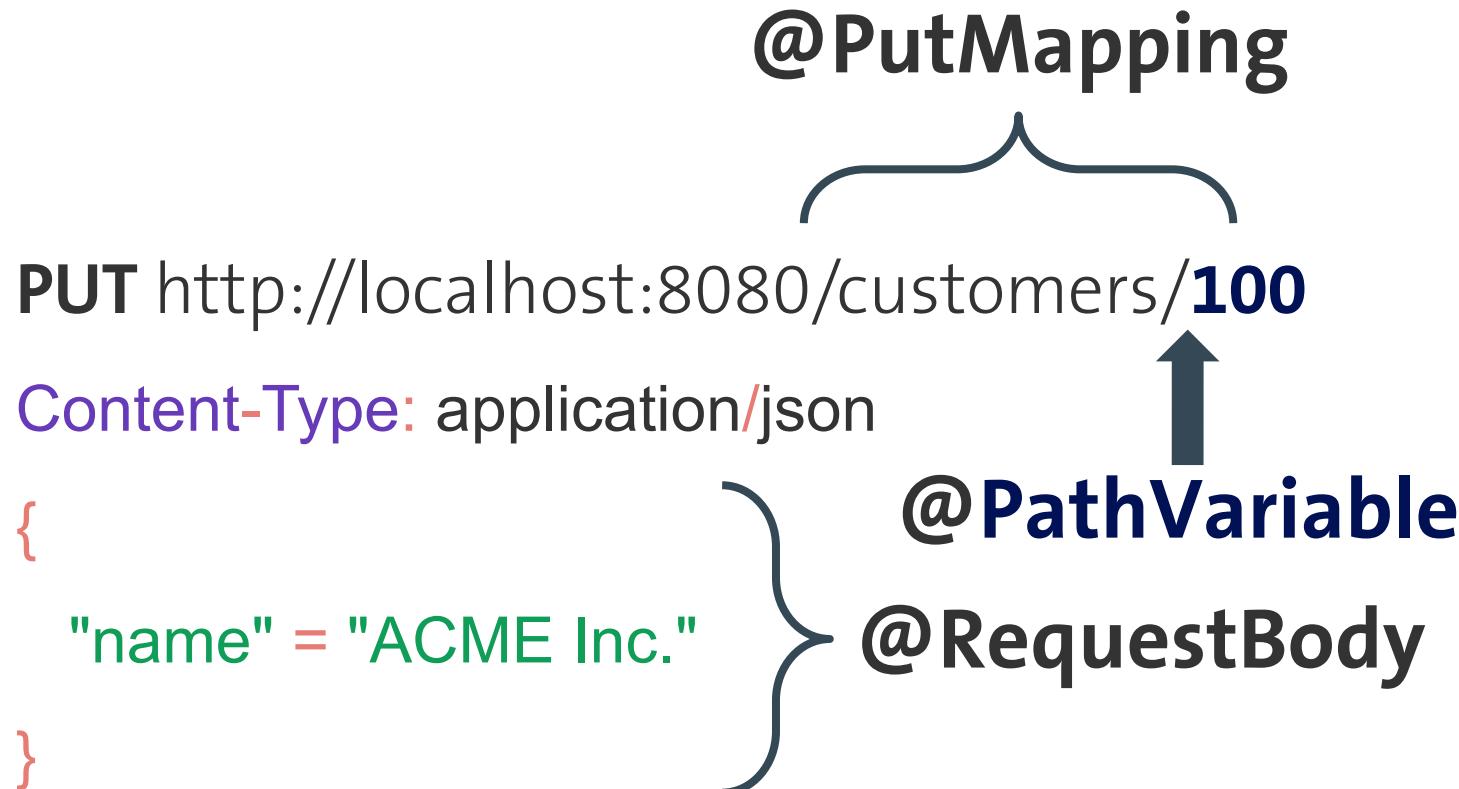
Spring Web / MVC / Important Annotations



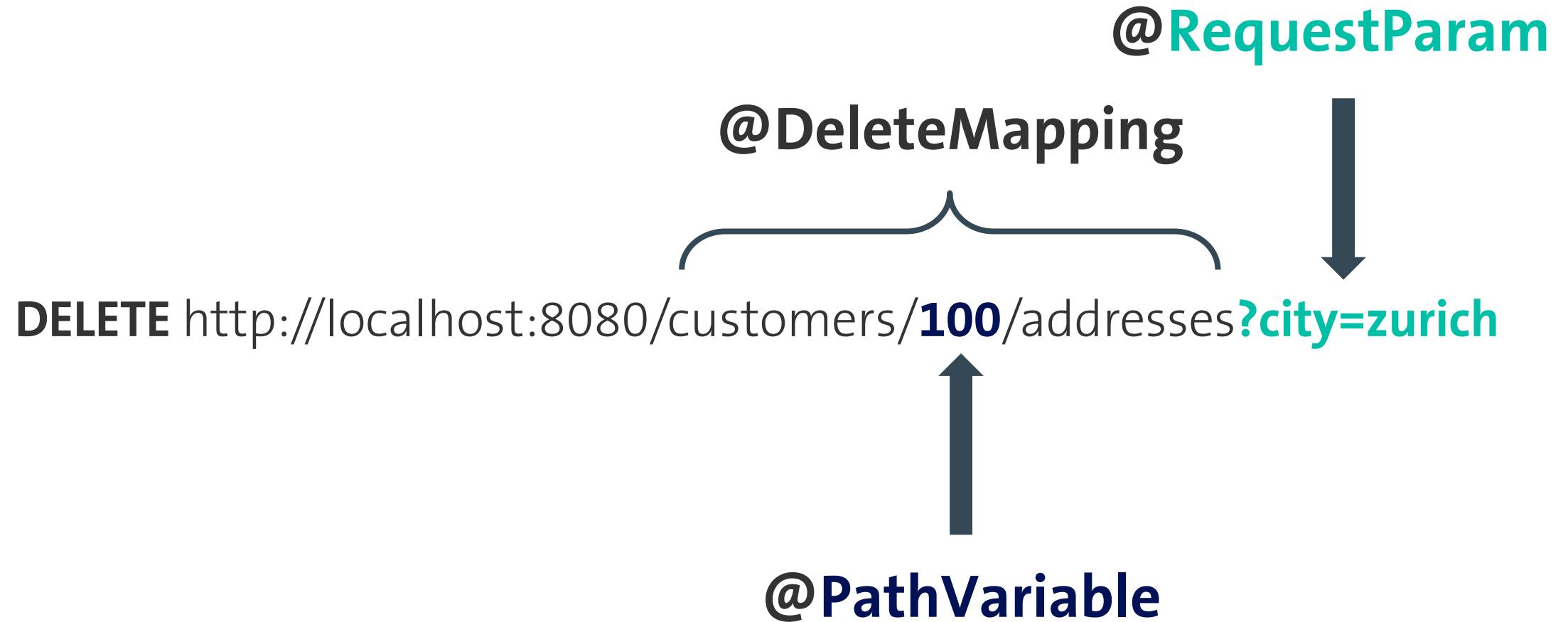
Spring Web / MVC / Important Annotations



Spring Web / MVC / Important Annotations



Spring Web / MVC / Important Annotations



```
@RestController  
@RequestMapping("customers")  
  
public class CustomerRestController {  
  
    @PostMapping()  
    public CustomerDto create(@RequestBody CustomerDto customer)  
  
    @GetMapping(produces = MediaType.APPLICATION_XML_VALUE)  
    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("id") String customerId) { ... }  
}
```

@RequestMapping can be omitted (if no common path)

The parentheses can be omitted (if empty)

application/json is the default media type

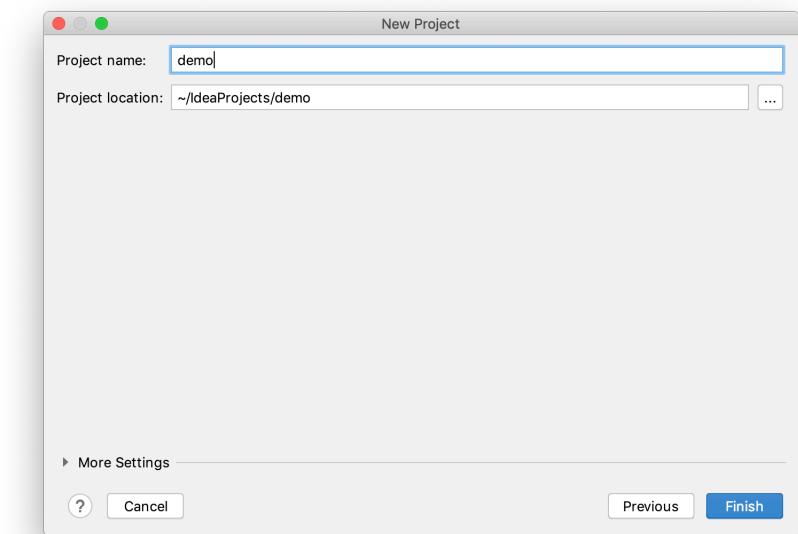
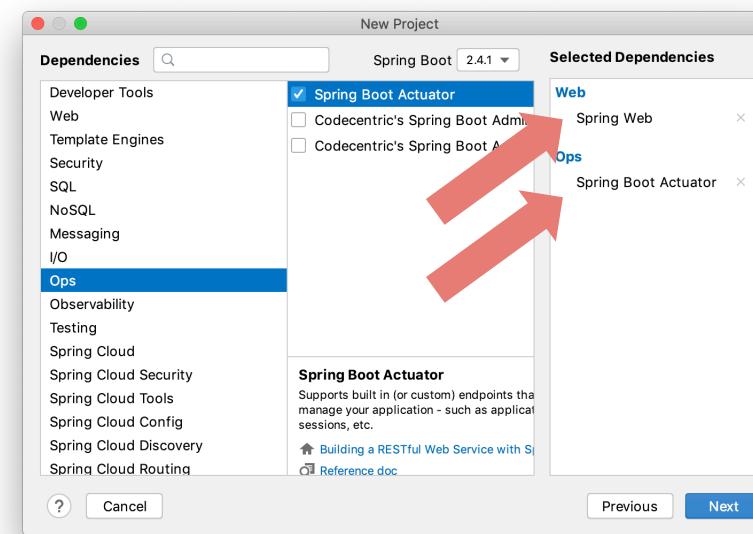
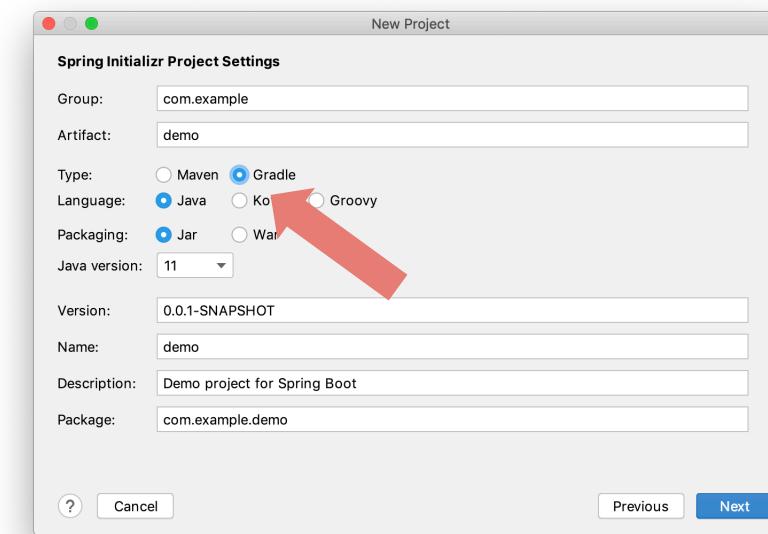
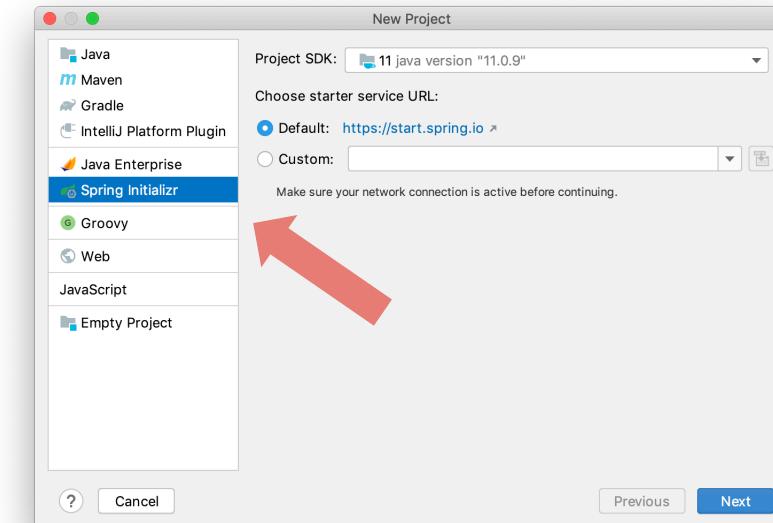
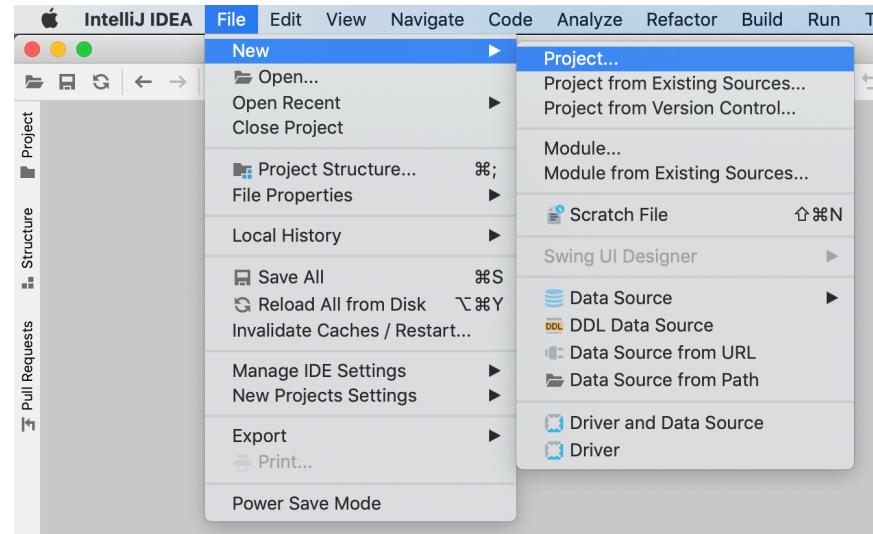
Optional indicates an optional request parameter

placeholder is required for @PathVariable

default is 200 (OK)

if the method parameter (customerId) and the placeholder (id) are different, the placeholder name must be specified.

Spring Boot



Spring Boot / @SpringBootApplication

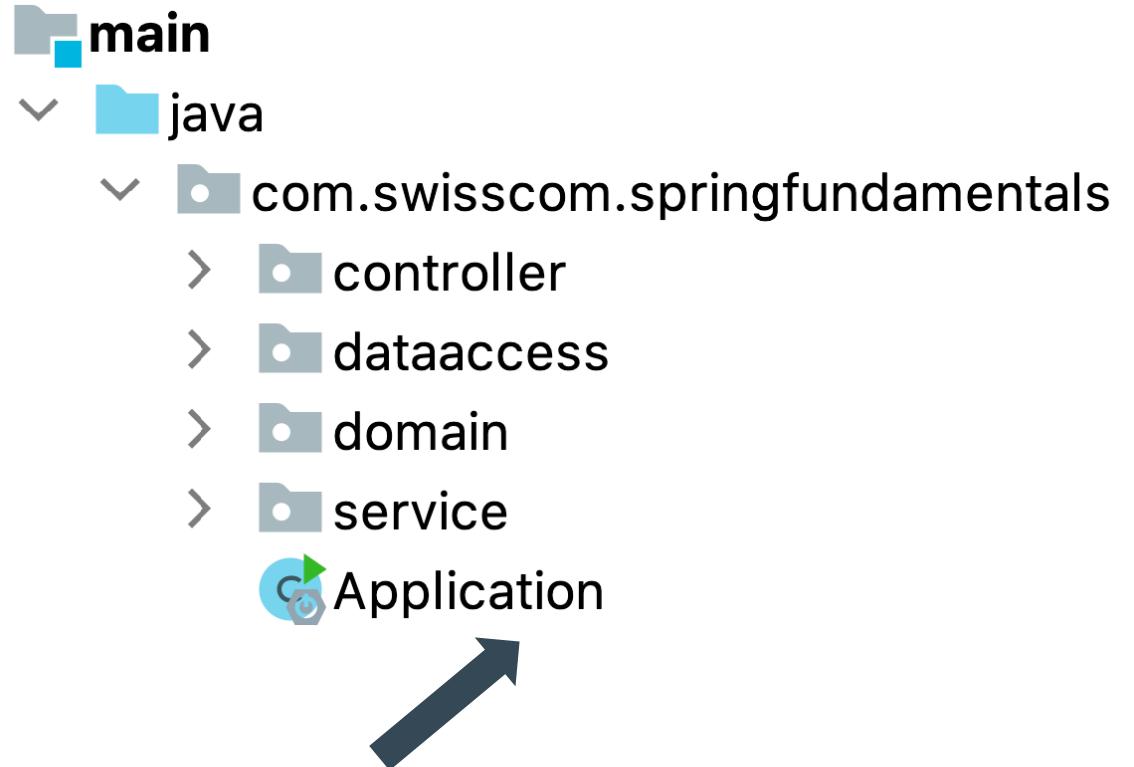
```
package com.swisscom.springfundamentals;

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 / 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 '2.4.1.' }  
plugins { id 'io.spring.dependency-management' version '1.0.10.RELEASE' }
```

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



Provides Maven-like dependency management

<https://github.com/spring-projects/spring-boot/blob/v2.4.1/spring-boot-project/spring-boot-dependencies/build.gradle>

Spring Boot / Hands-On

If you don't have a REST Client installed, you can use IntelliJ by creating a new "HTTP Request" scratch file (Mac OS: ⌘N, Windows: Ctrl+Shift+Alt+Insert) from which you can start HTTP Requests in the following format:

```
GET http://localhost:8080/customers
```

```
PUT http://localhost:8080/customers/1
```

```
Content-Type: application/json
```

```
{
```

```
    "name": "ACME Europe"
```

```
}
```

```
DELETE http://localhost:8080/customers/1
```

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

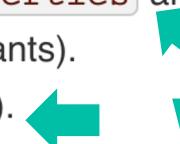
```
Content-Type: application/json
```

```
{
```

```
    "name": "ACME Europe"
```

```
}
```

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()`).

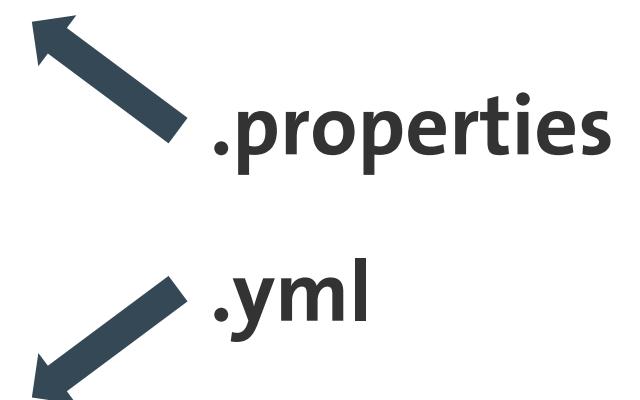
**Spring Boot consider
property sources in the
following order**

**We will have a closer
look into these property
sources**

Spring Boot / Profiles, Properties and Externalized Configuration

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

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



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.

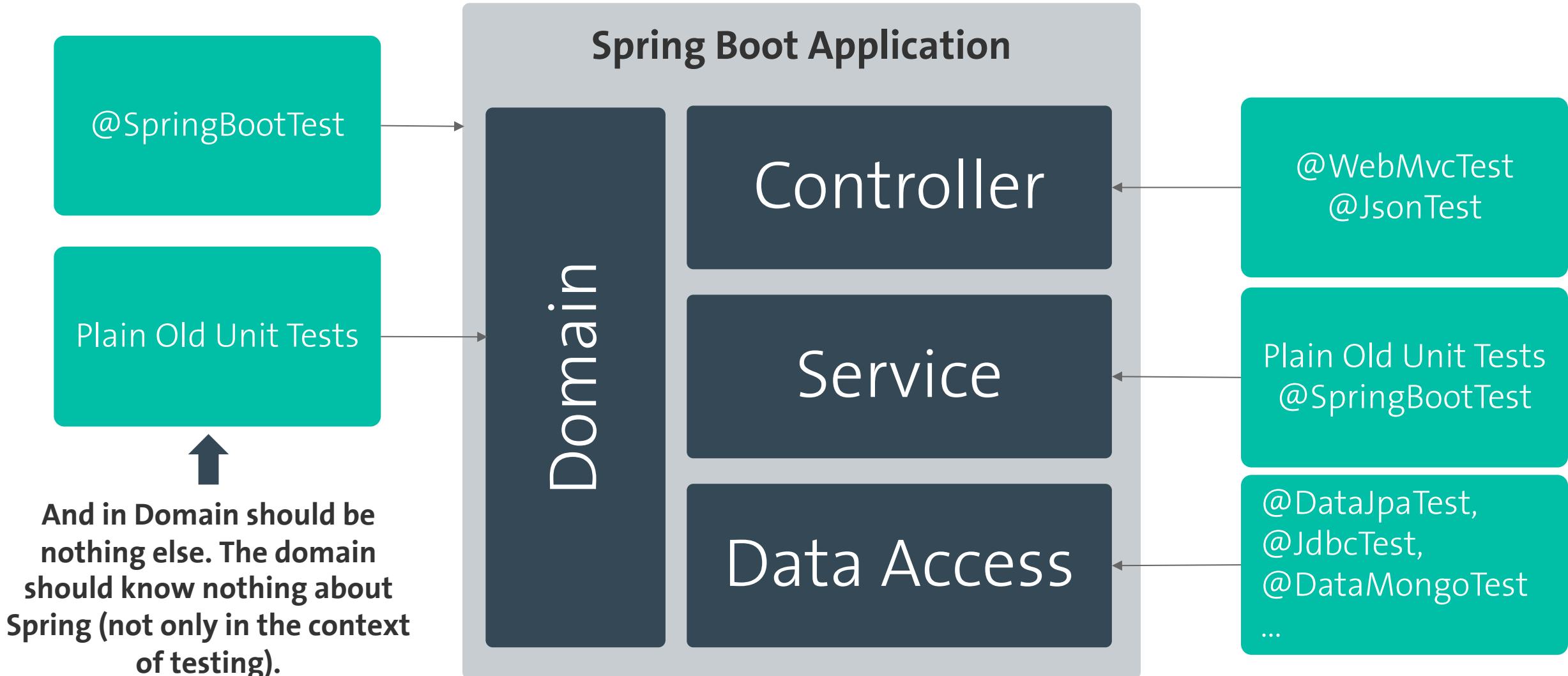
Spring Boot / Profiles, Properties and Externalized Configuration

```
@Service  
public class PizzalInventoryService {  
  
    @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**

Testing with Spring

Testing with Spring / Test Slices



Testing with Spring / @WebMvcTest

```
@WebMvcTest(GreetingRestController.class)
public class GreetingRestControllerTest {
    @Autowired
    private MockMvc mockMvc;
    @MockBean
    private GreetingService greetingService;

    @Test
    public void name() throws Exception {
        doReturn(new GreetDto("Hej", "Joe")).when(greetingService).greet(Optional.of("Joe"));
        mockMvc.perform(get("/greet?name=Joe")).andExpect(status().isOk())
            .andExpect(content().json("{\"name\": \"Joe\", \"greeting\": \"Hej\"}"));
    }
}
```