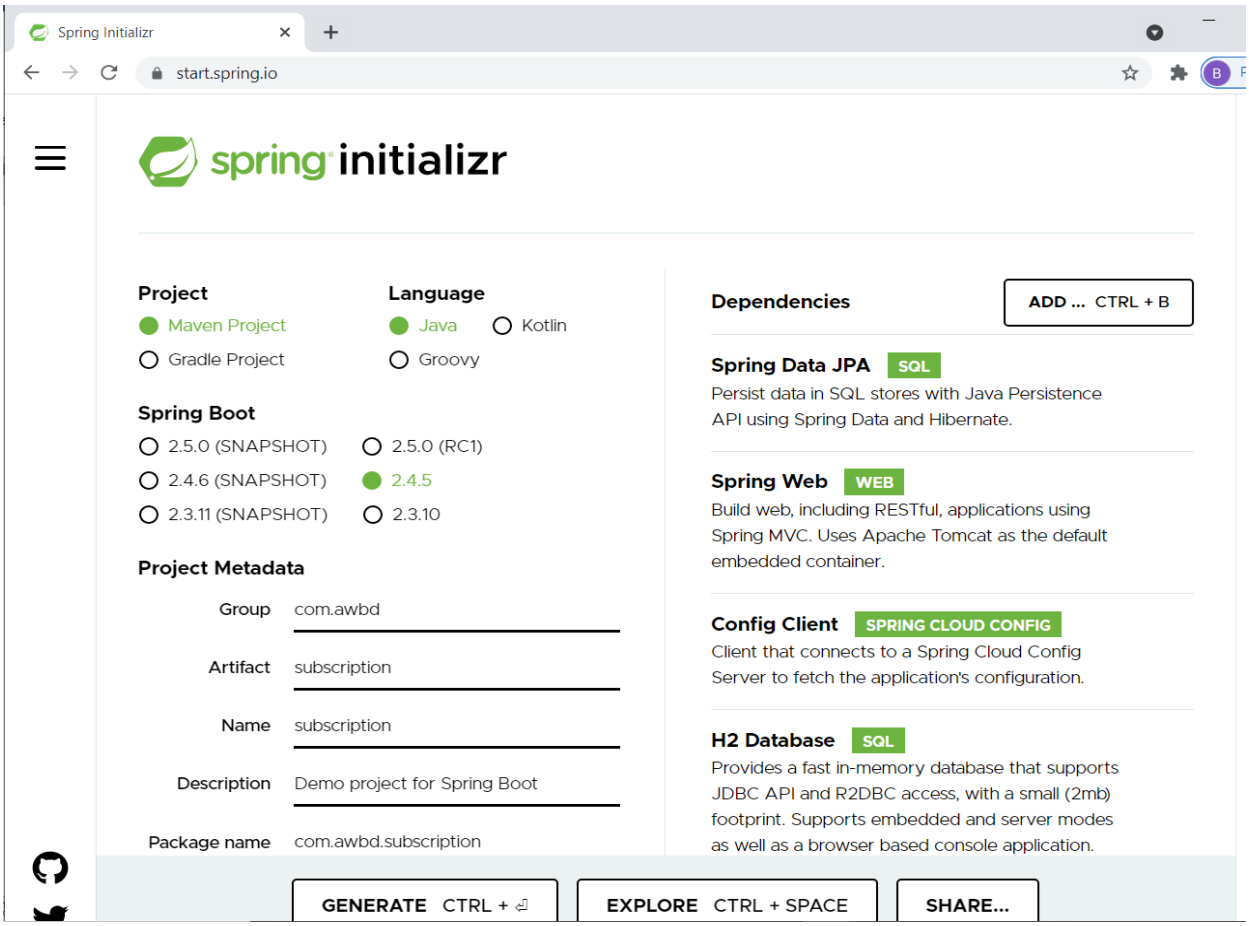
Practice 7: RESTFUL Webservices, Spring Cloud I

Config Servers

1. Open project subscription and skip to step 4 or Use ***Spring initializr*** to generate a maven project. https://start.spring.io/

Add dependencies: Spring Data JPA, Spring WEB, Config Client, H2 Database, Spring Cloud Discovery, Lombok, Spring HATEOAS.

Packaging **jar**



Info **Restful WebServices [1]**

- REST **architectural style** that defines constraints to be used for creating restful web services.
- **Resource driven**: Data or functionality are resources.
- **Lightweight,** Message body in JSON/XML/ HTML/… format.
- Client – server communication, **stateless**.

REST architectural constraints

UNIFORM INTERFACE:

- Each resource has unique URI.
- Uniform requests and representations: naming conventions, link or data format (XML/JSON).
- All resources should be accessible through a common approach (example HTTP).
- A resource should not be too large, when needed it may contain links to related information, **HATEOAS** (*Hypermedia as the Engine of Application State*).

CLIENT SERVER SEPARATION

- Client and server run independently.
- Clients and servers may be replaced independently.
- The only interaction between client and server is through requests and responses.
- The client should only know the resource URI.

STATELESS

- No server-side sessions.
- The server will not store any history of requests and will treat each request as new.
- Each request contains all the information needed.
- The client is managing the state of the application.

CACHEABLE RESOURCES

- If possible, resources should be cached.
- Caching improves performance.
- Cacheable resources should be marked as cacheable and have a version number.
- Requesting the same resource more than once should be avoided.

LAYERED SYSTEM

- Several layers of servers might interact between the server that returns the response and the client.
- Example: API on server A, data on server B, authentication on server C.
- Intermediary servers not visible to the client.

CODE ON DEMAND (optional)

- The response may contain executable code (e.g. JavaScript).

REST verbs

GET

- read resource.
- does not change the resource (read only).
- Using get multiple times for the same resource won't change the result (idempotence).
- Safe operation: does not change state of the resource.

PUT

- Insert if resource not found.
- Update if resource found.
- Using PUT multiple times with the same resource won't change the result (idempotence).
- Not safe operation: does change state of the resource.

POST

- Creates the resource.
- Not safe operation: does change state of the resource
- Using POST multiple times will create multiple resources (**not idempotent**).

DELETE

- Delete a resource.
- Not safe operation: does change state of the resource

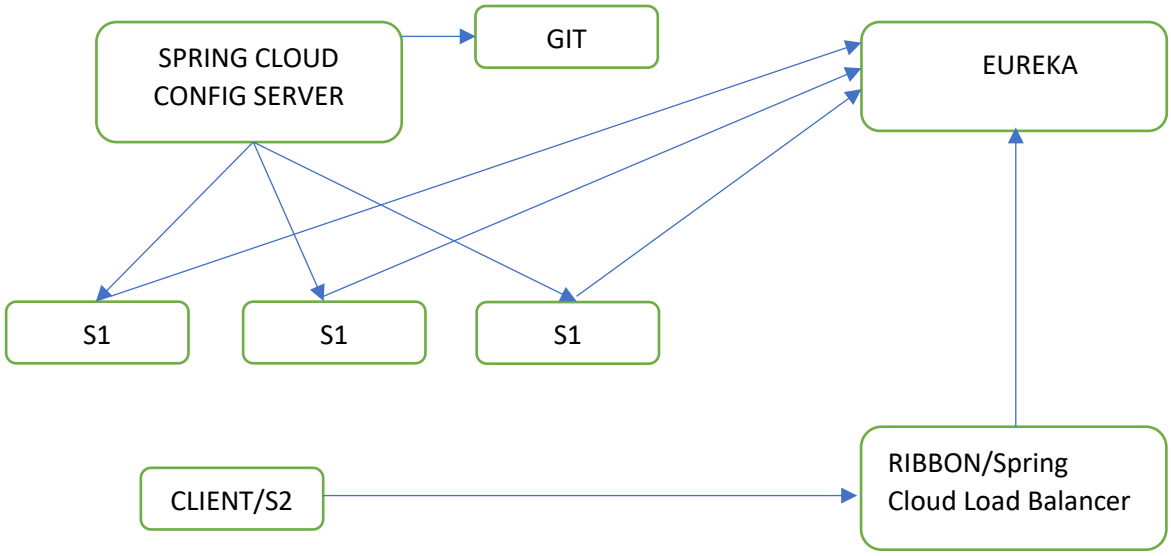
Other REST verbs:

- **HEAD** -- Used to retrieve the same headers as that of GET response. No response body, only headers
- **OPTIONS** -- List of HTTP methods supported by a resource.
- **TRACE** -- Debugging, echo header.

Spring Cloud Architecture [3]

Spring Cloud tools for developing common patterns in distributed systems.

- Configuration management -- **Spring Cloud Config Server**
- Service discovery -- **Naming Server Eureka**
- Intelligent routing, load balancing – **Ribbon**
- Visibility: monitoring services/servers **Zipkin Distributed Tracing Server**.
- Fault tolerance: **Hystrix** defaul behaviour in case of failure.



2. Add the maven dependencies for *swagger* and validations:

```
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
    <version>2.1.0</version>
</dependency>
```

3. Add package com.awbd.subscription and test requests using Postman:

```
http://localhost:8080/subscription/5
http://localhost:8080/subscription/4
http://localhost:8080/subscription/coach/James/sport/tennis
http://localhost:8080/subscription/list


POST http://localhost:8080/subscription
{
    "coach": "Ken",
    "sport": "tennis",
    "price": 200
}
```

4. Configure package *com.awbd.subscriptions.config* and a class to configure swagger, *com.awbd.subscriptions.config.SwaggerConfig*:

```
@Configuration
public class SwaggerConfig {
    @Bean
    public OpenAPI customOpenAPI() {
        return new OpenAPI().info(new Info().title("Spring Cloud
application API")
                .version("1").description("demo Spring Cloud"));

    }
}
```

5. Add documentation delete request from SubscriptionController, for more info read [4][5]:

```
@Operation(summary = "delete subscription by id")
@ApiResponses(value = {
        @ApiResponse(responseCode = "200", description = "subscription
deleted",
                content = {@Content(mediaType = "application/json",
                        schema = @Schema(implementation =
Subscription.class))}),
        @ApiResponse(responseCode = "400", description = "Invalid id",
                content = @Content),
        @ApiResponse(responseCode = "404", description = "Subscription
not found",
                content = @Content)})
@DeleteMapping("/subscription/{subscriptionId}")
public Subscription deleteSubscription(@PathVariable Long
subscriptionId) {

    Subscription subscription =
subscriptionService.delete(subscriptionId);
    return subscription;
}
```

6. Test requests:

http://localhost:8080/swagger-ui.html
http://localhost:8080/v3/api-docs
Notice that there are two accepted types of responses: *application/json* and *application/xml*. Response type must be specified as a request header.

7. Add dependency for Jackson xml converter:

```
<dependency>
    <groupId>com.fasterxml.jackson.dataformat</groupId>
    <artifactId>jackson-dataformat-xml</artifactId>
    <version>2.9.8</version>
</dependency>
```

Send request with header [{"key":"Accept","value":"application/xml","description":""}]
http://localhost:8080/subscription/list

Info **Spring-HATEOAS [6] [7][8]**
Hypertext as the Engine of Application State

Creates API that guides the client through the application by returning relevant information about the next potential steps, along with each response.

Creates links pointing to Spring MVC controller methods rendered into supported hypermedia formats (such as HAL). Hypertext Application Language (HAL) is an Internet Draft (a "work in progress") standard convention for defining hypermedia such as links to external resources within JSON or XML code. **[9]**

8. Add @EnableHypermediaSupport(type = EnableHypermediaSupport.HypermediaType.HAL) for class com.awbd.subscription. SubscriptionApplication

```
@SpringBootApplication
@EnableHypermediaSupport(type =
EnableHypermediaSupport.HypermediaType.HAL)
public class SubscriptionApplication {

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

9. Class subscription should extend *RepresentationModel<Subscription>.*
RepresentationModel is a container for a collection of Links. APIs add those links to the model.

```
public class Subscription extends RepresentationModel<Subscription>
```

10. Add a link to the resource returned by controller method *findByCoachAndSport*:
Test request: http://localhost:8080/subscription/coach/James/sport/tennis

```
Link selfLink =
linkTo(methodOn(SubscriptionController.class).getSubscription(subscripti
on.getId())).withSelfRel();
subscription.add(selfLink);
```

11. Add links to the list of Subscriptions return by *findSubscriptions* method of *SubscriptionController*:

```
@GetMapping(value = "/subscription/list", produces =
{"application/hal+json"})
public CollectionModel<Subscription> findAll() {

    List<Subscription> subscriptions = subscriptionService.findAll();
    for (final Subscription subscription : subscriptions) {
        Link selfLink =
linkTo(methodOn(SubscriptionController.class).getSubscription(subscripti
on.getId())).withSelfRel();
        subscription.add(selfLink);
        Link deleteLink =
linkTo(methodOn(SubscriptionController.class).deleteSubscription(subscri
ption.getId())).withRel("deleteSubscription");
        subscription.add(deleteLink);
    }
    Link link =
linkTo(methodOn(SubscriptionController.class).findAll()).withSelfRel();
    CollectionModel<Subscription> result =
CollectionModel.of(subscriptions, link);
    return result;
}
```

Info **Configuration management**
It is a good practice to store specific configuration external, not embedded into application code.

It is recommended to use a central repository to store properties that are needed by micro services (examples: database credentials). All configurations may be **stored** in a GitHub repository, in a cloud storage or in a database.
Properties may be **injected** from the central repository instead of rebuilding docker images whenever environment specific properties are changed. Microservices should get the needed properties without restarting all instances. Also, one may keep **different versions** of properties for **multiple configurations** (dev, prod configurations)

Spring Cloud Config [10] provides client-side and server-side support for loading and injecting external properties for microservices.

Server-side, embedded in Spring Boot with @EnableConfigServer, exposes external configuration files (key-value, YAML etc.) and encrypt and decrypt property values.

Client-side config binds microservices to config servers, encrypt and decrypt property values.

12. Use Spring initializr to create a new project with Artifact Id: *config-server*. Add dependencies: Config Server (Spring Cloud Config).

13. Annotate class com.awbd.ConfigServerApplication with @EnableConfigServer annotation:

```
@EnableConfigServer
@SpringBootApplication
public class ConfigServerApplication {

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

}
```

14. Run config-server on port 8070, in application.properties add:

```
spring.application.name=config-server
server.port=8070
spring.profiles.active=native
spring.cloud.config.server.native.search-locations=classpath:/config
```

15. Add in src\main\resources\config files discount.properties, discount-dev.properties, discount-prod properties.

```
percentage.month=10
percentage.year=20
```

```
percentage.month=30
percentage.year=40
```

```
percentage.month=50
percentage.year=60
```

16. Test: http://localhost:8070/discount/default
http://localhost:8070/discount/prod
http://localhost:8070/discount/dev

17. Create a new project with Artifact Id: *discount*. Add dependencies: Lombok, Web and Config Client (Spring Cloud Config).

18. Modify application.properties:

```
spring.application.name=discount
percentage.month=5
percentage.year=7
```

19. Create package config and class PropertiesConfig.

```
@Component
@ConfigurationProperties("percentage")
@Getter
@Setter
public class PropertiesConfig {
    private int month;
    private int year;
}
```

Info **@ConfigurationProperties [11]**

Properties may be grouped in POJOs. With @ConfigurationProperties annotation, properties POJOs are mapped to *application.properties* file.

20. Create a package *model* and class *Discount*.

```
@Setter
@Getter
@AllArgsConstructor
public class Discount {
    private int month;
    private int year;
}
```

21. Create a package *controllers* and class *Controller*.

```
@RestController
public class Controller {
    @Autowired
    private PropertiesConfig configuration;

    @GetMapping("/discount")
    public Discount getDiscount(){
    return new
Discount(configuration.getMonth(),configuration.getYear());
    }
}
```

22. Modify file *application.properties* in project discount:

```
spring.application.name=discount
spring.config.import=optional:configserver:http://localhost:8070/
spring.profiles.active=dev


server.port = 8081
```

Run project discount-service on port 8081.

23. Modify config-server so that config files are read from file system:

```
spring.application.name=config-server
server.port=8070
spring.profiles.active=native
spring.cloud.config.server.native.search-locations= file:///C:/../config
```

24. Run config-server on port 8070, in application.properties add:

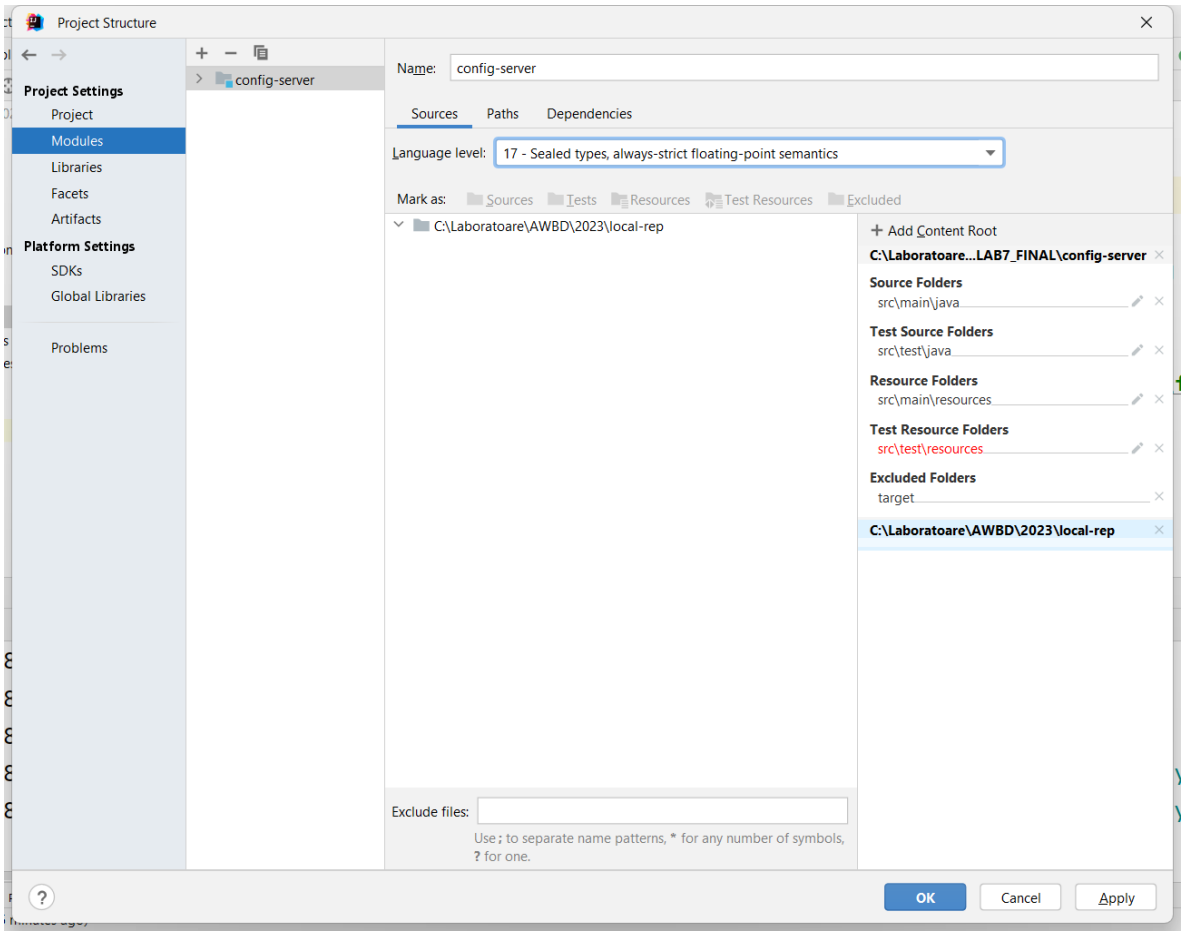
```
spring.application.name=config-server
server.port = 8070
spring.cloud.config.server.git.uri=file:///C:/../local-rep
```

25. Install git https://git-scm.com/downloads.

26. Create git local repository:

```
>> mkdir local-rep
>> cd local-rep
>> git init
```

27. Add git repository in IntelliJ:



28. Commit on git repository **discount.properties** file:

```
percentage.month=15
percentage.year=25
```

```
>> git add -A
>> git config --global user.username awbd
>> git commit -m "config file"
```

29. Test discount-service configuration:

http://localhost:8070/discount/default

30. Modify config-server so that config files are read from hithub:

```
spring.application.name=config-server
server.port=8070
spring.profiles.active=git

spring.cloud.config.server.git.uri=https://github.com/…/config_files.git
spring.cloud.config.server.git.clone-on-start=true
spring.cloud.config.server.git.default-label=main
```

Add dependency for Actuator in discount application:

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

Add annotation @RefreshScope

```
@SpringBootApplication
@RefreshScope
public class DiscountApplication {

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

}
```

31. Expose actuator endpoints. In application.properties add:

```
management.endpoints.web.exposure.include=*
```

Modify discount-dev.properties on Git and reload configuration for discount service with actuator request:

```
http://localhost:8081/actuator/refresh
```

34. Add class *DiscountServiceProxy*:

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

```
@FeignClient(value = "discount", url = "localhost:8081")
public interface DiscountServiceProxy {
    @GetMapping("/discount")
    Discount findDiscount();
}
```

35. Modify Subscription controller:

```
@Autowired
DiscountServiceProxy discountServiceProxy;

@GetMapping("/subscription/coach/{coach}/sport/{sport}")
Subscription findByCoachAndSport(@PathVariable String coach,
                    @PathVariable String sport){
   Subscription subscription = subscriptionService.findByCoachAndSport(coach, sport);
   Discount discount = discountServiceProxy.findDiscount();
   subscription.setPrice(subscription.getPrice()* (1 - discount.getMonth()));

   return subscription;
}
```

Test urls:

http://localhost:8081/discount
http://localhost:8080/subscription/coach/James/sport/tennis

B

[1] https://restfulapi.net/

[2] RESTful Web Services, Leonard Richardson, Sam Ruby 2007

[3] https://spring.io/projects/spring-cloud

[4] https://swagger.io/resources/articles/documenting-apis-with-swagger/

[5] https://www.baeldung.com/spring-rest-openapi-documentation

[6] https://www.baeldung.com/spring-hateoas-tutorial

[7] https://spring.io/projects/spring-hateoas

[8] https://howtodoinjava.com/spring5/hateoas/spring-hateoas-tutorial/

[9] https://en.wikipedia.org/wiki/Hypertext_Application_Language

[10] https://spring.io/projects/spring-cloud-config

[11] https://www.baeldung.com/configuration-properties-in-spring-boot

[12] https://www.baeldung.com/spring-cloud-openfeign

[13] https://springdoc.org/v2/#features