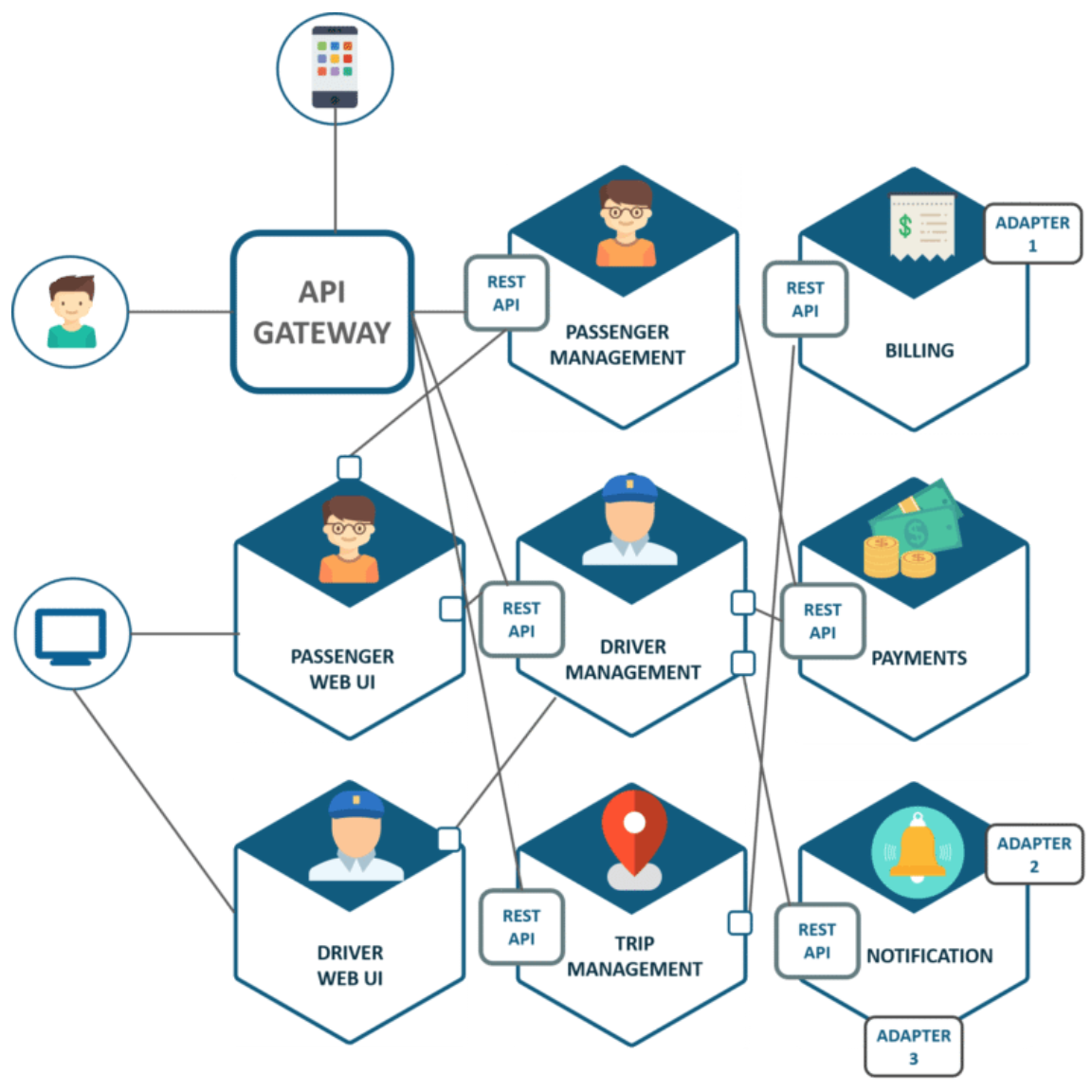
**Workshop**

**Microservices bouwen en testen met REST-Assured**

Deze workshop is opgebouwd uit een aantal lessen. We kijken vandaag hoever we komen. De volgorde waarop we ermee aan de slag gaan is als volgt:

1. Algemeen: wat zijn Microservices en hoe ziet een microservice architectuur eruit?
2. Het bouwen van één microservice met behulp van Micronaut
3. Het lokaal testen van de microservice met behulp van Rest-assured
4. Het bouwen van een tweede microservice die requesten doorstuurt naar de eerste microservice
5. Een derde microservice die gebruik maakt van Lombok en het builder pattern
6. Logging
7. De microservices en testprojecten in een Docker container draaien
8. Het lokaal testen van de tweede microservice met behulp van Rest-assured

Hieronder een overzicht van hoe een microservice architectuur er uit zou kunnen zien:

  
*Microservice Architecture of Uber*

**1. Wat zijn microservices?**

In deze workshop zullen wij een klein stukje van de architectuur van microservices onder handen nemen.  
Namelijk:  
- Wat is een API Gateway

Een API-gateway neemt alle API-aanroepen van clients en leidt ze vervolgens naar de juiste microservice met requestrouting en protocolvertaling.

Doorgaans behandelt het een verzoek door meerdere microservices op te roepen en de resultaten terug te koppelen.

Het kan vertalen tussen webprotocollen en webonvriendelijke protocollen die intern worden gebruikt.

Een app bouwer kan een API-gateway gebruiken om mobiele clients een dienst te bieden voor het ophalen van alle productdetails met één verzoek.

Het roept verschillende services aan, combineert de resultaten zoals persoonsinformatie en uitkering gegevens en koppelt deze terug naar de app.  
  
Wil je er meer over lezen zoals de implementatie van netflix?  
[Lees dan dit stuk](https://freecontent.manning.com/the-api-gateway-pattern/)

- Wat is de rol van een microservice?

Dit hangt af van de rol die vervult moet worden.

Moeten er gegevens over een persoon worden opgezocht?

Moeten er bedragen worden berekend?

Moeten er elementen onttrokken worden uit een bericht?

Etc.

De rol van een microservice is doorgaans een service die slechts 1 taak voor zijn rekening neemt zoals het filteren van gegevens uit een bericht of het verschillende persoonsgegevens samenvoegen het resultaat terug te geven aan de aanvrager.

**2. Je eerste Hello World! microservice**

In dit voorbeeld zullen we gebruikmaken van een framework genaamd “Micronaut” die het makkelijk maakt om een microservice te bouwen.

1. Installeer Homebrew:  
   */usr/bin/ruby -e "$(curl -fsSL* <https://raw.githubusercontent.com/Homebrew/install/master/install>)"
2. Installeer Micronaut m.b.v. de CLI:

2.1 *brew update*

2.2 *brew install micronaut*

2.3 controleer of micronaut succesvol is geïnstalleerd:  
 in de CLI: *mn* [ENTER]

Wat je te zien krijgt als het succesvol is geïnstalleerd:

Starting interactive mode...

Enter a command name to run. Use TAB for completion:

mn>

Je bevindt je nu in micronaut (mn>). Gebruik control + c om micronaut weer af te sluiten in je command line.

**EXTRA INFO:**  
*Bij het aanmaken van een project is de naamgeving belangrijk. De naam van de microservice komt terug in de repository naam binnen Gitlab, de groupId en artifactId van [Maven](https://maven.apache.org/guides/mini/guide-naming-conventions.html" \t "_blank) en de [Java package name](https://docs.oracle.com/javase/tutorial/java/package/namingpkgs.html" \t "_blank). Ook het Docker image voor de microservice krijgt deze naam.*

*Met behulp van de Micronaut CLI kan een project met bepaalde eigenschappen en functionaliteiten worden aangemaakt. In het onderstaande voorbeeld wordt er voor de applicatie* ***my-micronaut-service*** *een Maven project aangemaakt (met ondersteuning voor [Java](https://www.oracle.com/technetwork/java/javase/overview/index.html" \t "_blank))*

1. Genereer je microservice project
   1. Navigeer naar je .m2 folder en maak een kopie van je settings.xml
   2. Plaats onderstaande in een nieuw settings.xml bestand:

<settings xmlns="http://maven.apache.org/SETTINGS/1.1.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.1.0 http://maven.apache.org/xsd/settings-1.1.0.xsd">

<localRepository>/Users/$USER/.m2/repository</localRepository>

<profiles>

<profile>

<id>external</id>

<repositories>

<repository>

<id>central</id>

<url>https://repo1.maven.org/maven2</url>

<releases>

<enabled>true</enabled>

</releases>

<snapshots>

<enabled>true</enabled>

</snapshots>

</repository>

<repository>

<id>eclipse\_repo\_releases</id>

<url>https://repo.eclipse.org/content/groups/releases</url>

<releases>

<enabled>true</enabled>

</releases>

<snapshots>

<enabled>false</enabled>

</snapshots>

</repository>

</repositories>

<pluginRepositories>

<pluginRepository>

<id>central</id>

<url>https://repo1.maven.org/maven2</url>

<releases>

<enabled>true</enabled>

</releases>

<snapshots>

<enabled>true</enabled>

</snapshots>

</pluginRepository>

<pluginRepository>

<id>eclipse\_repo\_releases</id>

<url>https://repo.eclipse.org/content/groups/releases</url>

<releases>

<enabled>true</enabled>

</releases>

<snapshots>

<enabled>true</enabled>

</snapshots>

</pluginRepository>

</pluginRepositories>

</profile>

</profiles>

<activeProfiles>

<activeProfile>external</activeProfile>

</activeProfiles>

</settings>

let op dat je $USER vervangt met je eigen naam.

3.3 Navigeer met de CLI naar de locatie waar je dit project wil hebben draaien.

Het makkelijkst is als we dit allemaal doen binnen je workspace (cd $ws)

3.4 Met micronaut genereer je het project in de CLI:

*mn create-app nl.bkwi.workshop.mymicronautservice.my-micronaut-service \*

*--build=maven --features=java,junit*

We hebben nu een project aangemaakt met de naam “my-micronaut-service”

3.5 Open het project met IntelliJ

Onder src>main>java zie je nu de package nl.bkwi.workshop.mymicronautservice met daarin een java klasse genaamd “Application”.  
Dit bestand is gegenereerd en wordt gebruikt door het micronaut framework.  
Wij zullen hier tijdens deze workshop verder geen aandacht meer aan schenken.  
Het enige wat belangrijk is om te weten is dat micronaut dit bestand nodig heeft om de microservice te draaien.  
Daarnaast werkt micronaut met annotaties.

Wat zijn annotaties?  
Annotaties zijn labels die je bijvoorbeeld aan een klasse of een methode kunt hangen.  
Op compile- of runtime worden alle klassen doorlopen en gekeken naar de labels.  
Labels hebben opzichzelf geen waarde maar hangt er een actie aan vast.

Dat betekend dat er wat mee gedaan kan worden maar dat hoeft niet.

Bijvoorbeeld de actie dat er objecten gegenereerd moeten worden waar de klasse of methode gebruik van kan maken of bijvoorbeeld een configuratie die meegestuurd wordt naar de implementatie.  
Op de achtergrond wordt er voor de implementatie gezorgd en hoeven wij ons daar verder niet druk om te maken.

Door een klasse te annoteren met de [@Controller](https://docs.micronaut.io/latest/api/io/micronaut/http/annotation/Controller.html" \t "_blank) annotatie en vervolgens de methoden te annoteren met de [@Get](https://docs.micronaut.io/latest/api/io/micronaut/http/annotation/Get.html" \t "_blank) of [@Post](https://docs.micronaut.io/latest/api/io/micronaut/http/annotation/Post.html" \t "_blank) annotatie worden de http requesten die naar de applicatie worden gestuurd via deze methoden vertaald naar java code. Voorbeelden zijn te vinden in de [Micronaut documentatie](https://docs.micronaut.io/latest/guide/index.html" \l "creatingServer" \t "_blank) en [Creating your first Micronaut app](https://guides.micronaut.io/creating-your-first-micronaut-app/guide/index.html" \l "controller" \t "_blank) guide.

Het is goed om te weten dat een microservice bestaat uit een controller, een cliënt, een service en een model van verschillende object klassen.  
  
De controller is het component dat de inkomende REST API calls kan vertalen naar Java en dat doorsturen naar de service laag dat vervolgens “iets” doet met de call.   
Dit zou bijvoorbeeld het aanmaken of retourneren van een persoon kunnen zijn die is opgebouwd uit het model van object klassen die gezamenlijk tot een persoon komen met blauwe ogen en groen gelakte nagels.

Maar de service laag binnen een microservice kan bijvoorbeeld ook de business logica verwerken dat er 4 elementen uit een bericht worden gefilterd en dat die als nieuw bom bericht worden geretourneerd door de controller.

In deze workshop laten wij de complexiteit van de service laag en het model achterwege en retourneren wij gewoon een simpele “Hello world” String terug.

Het enige doel dat wij nu willen bereiken is dat de Controller een “Hello world!” terug geeft wanneer wij in de browser een api (get) call doen aan <http://localhost:9090/hello>

Om dit te realiseren hebben we twee dingen nodig.

* Deze micro service draaien op poort 9090
* Een Controller service die antwoord geeft wanneer wij naar bovenstaande URL gaan.

1. Pas de poort aan waarop de applicatie draait.

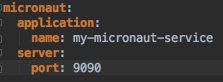
4.1 Open de “application.yml” onder src>main>resources

4.2 Voeg onderstaande toe aan de applicatie configuratie

server:

port: 9090

**LET OP!**   
De woorden dienen correct ingesprongen te zijn. Dit kan betekenen dat de puntjes niet netjes onder elkaar staan! Indien dat niet het geval is zal de applicatie niet starten.



1. Draai een mvn clean-package om te zorgen dat alle dependencies worden gedownload. Er mogen geen rood onderstreepte elementen meer zijn in de code.
2. Start de microservice om te controleren of de configuratie geslaagd is.

5.1 Rechtermuis op “Application” en klik dan op de groene pijl

(Run ‘Application.main()’)

5.2 Open je browser en ga naar <http://localhost:9090>

5.3 Controleer of je de volgende foutmelding ziet staan:

{"\_links":{"self":{"href":"/","templated":false}},"message":"Page Not Found"}

Wanneer je deze foutmelding ziet staan betekend dit dat de configuratie geslaagd is en dat de service draait, maar dat de pagina niet gevonden is. En dat klopt, want we hebben nog geen service gemaakt die een antwoord geeft wanneer we een bevraging doen op deze url.

6. Bouw de Hello World controller klasse.

6.1 Maak een nieuw java klasse bestand aan in dezelfde folder waar het “Application” bestand staat (Rechtermuis op de myfirstmicronautservice package > New > Java Class) en noem dit bestand “HelloWorldController”

Zoals eerder aangegeven werkt micronaut met Annotaties.  
In de HelloWorldController klasse zullen we nu een aantal annotaties opnemen die er voor zorgen dat het framework zijn werk kan doen.

7. Definieer de controller klasse:

7.1 Importeer de Controller annotatie:

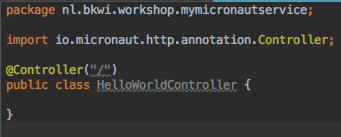
*import io.micronaut.http.annotation.Controller;*

7.2 Door boven de klasse definitie de Controller annotatie aan te geven, geven we aan het framework door dat dit een Controller klasse betreft.  
Deze annotatie wil graag weten wat het pad is waarop deze controller luistert en daarom zullen we deze als volgt doorgeven:

@Controller("/v1")

(Dit kan uiteraard ook @Controller("/hoi") of @Controller("/hi") etc. zijn)

Je klasse zal er nu als volgt uitzien:



8. Implementeer de “Hello world” service

8.1 Importeer de Get annotatie:

*import io.micronaut.http.annotation.Get;*

8.2 Implementeer de Get response:

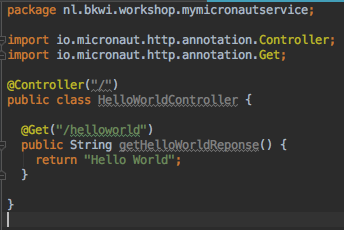
@Get("/helloworld")

public String getHelloWorldReponse() {

return "Hello World";

}

Als het goed is ziet je controller klasse er nu zo uit:

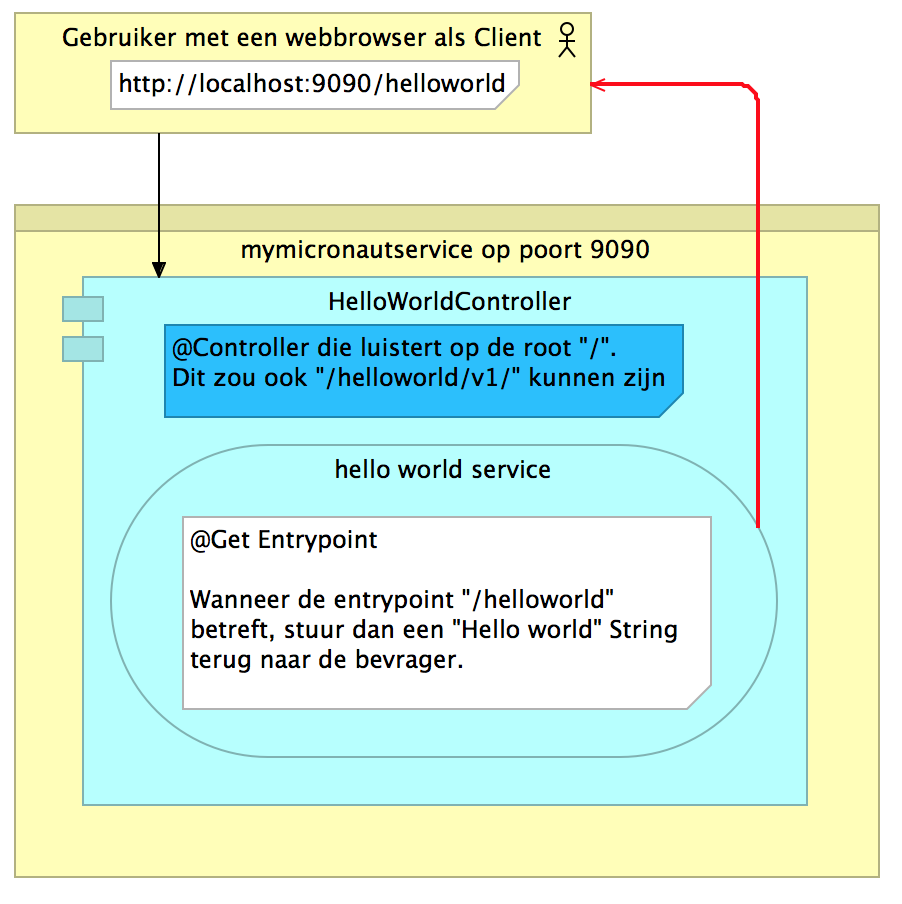


Start de service nu opnieuw op door linksonder- of boven in op de stop knop (rode vierkant) en vervolgens op de start knop (groene driehoek) te drukken.

Navigeer in je browser naar:  
localhost:9090/helloworld

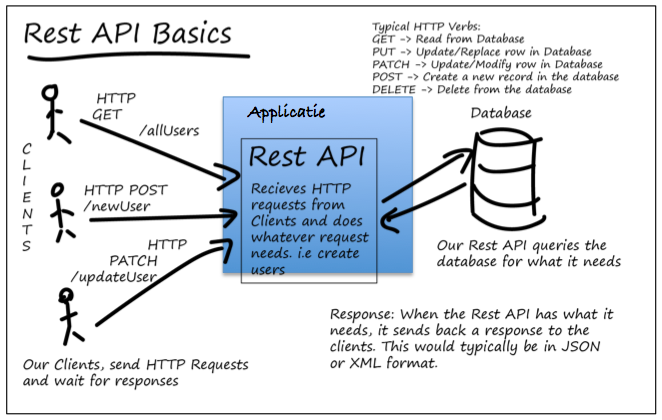
Als het goed is zie je nu de door jou opgegeven “Hello world” response in de browser verschijnen. Omdat de response in JSON formaat wordt aangeleverd kun je hetzelfde request natuurlijk ook uitvoeren met andere tools zoals Postman.

Wat hebben we bereikt?



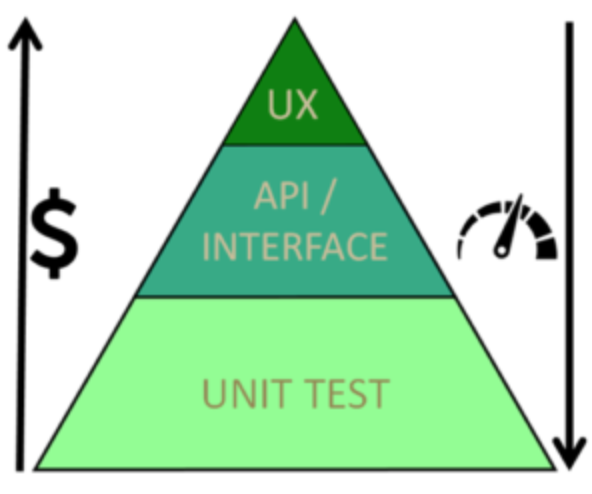
**3. Rest-assured testen bij de hello world service**

Nu we de “Hello world” service hebben draaien kunnen we deze ook testen met Rest-assured.

Rest-assured wordt veelal gebruikt voor het aftesten van API’s.  
  


Maar waarom?  
  
Er zijn meerdere redenen maar de voornaamste zijn toch wel:

* De mogelijkheid om vanuit de wens van de klant (business logic – oftewel Gherkin/BDD-Style) af te kunnen testen.
* Er kunnen al testen gemaakt worden voor server side code, waarvoor nog geen userinterface is gebouwd of wanneer die nog in ontwikkeling is.
* De belangrijkste: wanneer je test vanuit de user interface, dan test je in eerste instantie en eigenlijk alleen maar de validaties aan de client side. Een applicatie heeft echter ook validaties aan de server kant. Om zeker te weten dat de server ook juist valideert is het daarom erg nuttig om direct de API aan te spreken, door middel van http requests. Een hacker kan dit namelijk ook doen, dus je wilt zeker zijn dat daar geen informatie wordt geaccepteerd die de broker onnodig belast of dat de hacker gevoelige informatie teruggestuurd krijgt uit of over de applicatie.
* Binnen TWI is de belangrijkste reden om rest-assured te gebruiken simpelweg omdat we niet over een user interface beschikken waar we tegenaan kunnen testen. De microservices kunnen alleen getest worden op het REST niveau.
* Daarnaast is het testen van de API stukken goedkoper en sneller dan het aftesten tegen een GUI zoals een web browser.



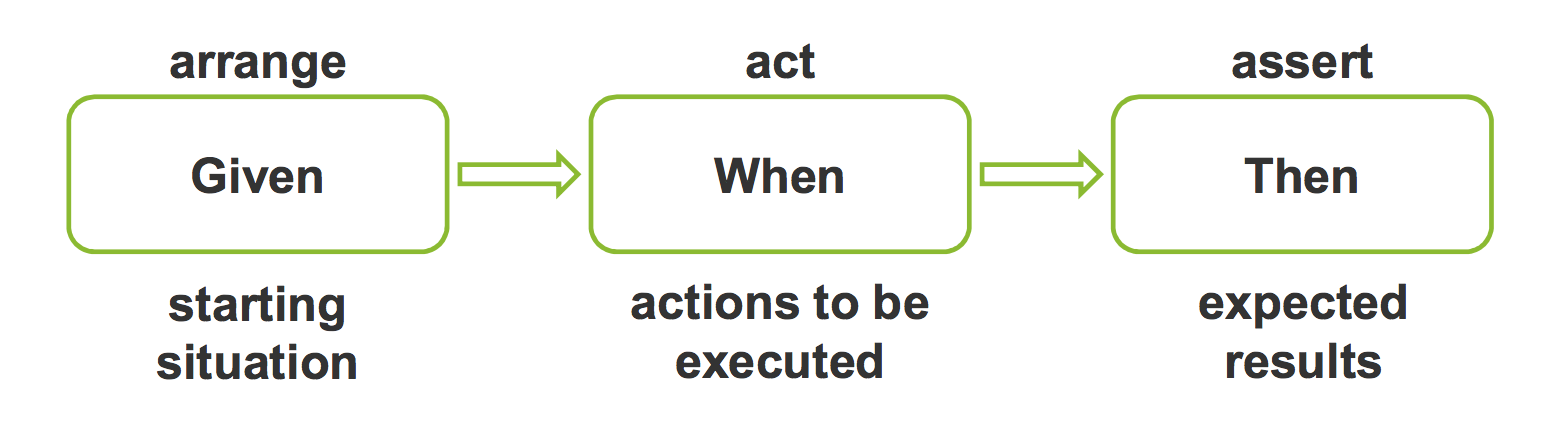
**De structuur**

Hoe een Rest-Assured test wordt opgebouwd, sluit sterk aan bij wat in Gherkin wordt beschreven. Voor meer informatie over het gebruik van Gherkin en syntax zie verder https://docs.cucumber.io/gherkin.

given()

.when()

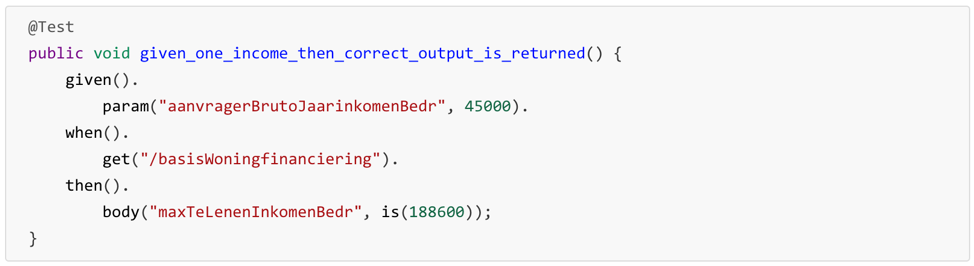
.then()



Binnen de given/when/then structuur kun je ook op verschillende plekken and() inzetten om de leesbaarheid te vergroten, maar deze is optioneel. Het werkt ook om meerdere given/when/then’s te gebruiken wanneer je geen and() tussenvoegt.

Dankzij deze opbouw wordt je gedwongen de testen structureel op te pakken, net als bij Cucumber. Een Rest-Assured test laat zich door deze structuur makkelijk lezen, ook als je minder Java kennis hebt.

Een voorbeeld:



Rest-Assured biedt tevens ondersteuning voor logging op de volgende niveaus:

* Alles
* Body van de response
* Statuscode van de response
* Parameters die meegegeven worden
* Cookies die meegegeven worden
* Headers

Omdat je dankzij deze tool onder andere assertions kunt doen op basis van bijvoorbeeld de response body en de status code hebben wij er voor gekozen om deze tool in te zetten binnen BKWI.

**EXTRA INFO:**

*Soms wordt er ook gebruik gemaakt van:*

*given()*

*.expect()*

*.when()*

*Deze tweede variant wordt ook wel ‘legacy syntax’ genoemd en is ontwikkeld op het moment dat Rest-Assured de given/when/then structuur nog niet ondersteunde. De given/expect/when-variant is minder helder en eenduidig en kan daardoor verwarrend zijn. Daarbij heeft het wel één voordeel ten opzichte van given/when/then: alle verwachte waarden kunnen in één keer worden getoond. Dit is niet mogelijk bij de eerste variant, omdat deze variant de test afbreekt op het moment dat er een validatie faalt. De eventuele andere validaties in de test worden dan niet meer uitgevoerd.*

*Los van dit voordeel wordt de given/when/then variant aangeraden door de makers van Rest-Assured, omdat deze aansluit bij BDD (Behavioural Driven Development).*

Genoeg theorie… Laten we maar eens wat gaan doen…

9. Maak een nieuw Maven project aan in IntelliJ:

9.1 File > New > Project

9.2 Select Maven en je SDK (Java versie)

9.3 Klik op Next

9.4 Voer “nl.bkwi.workshop “ in bij GoupId.

9.5 Voer “api-restassured” in bij ArtifactId.

9.6 Voer “1.0-SNAPSHOT” in bij Version.

9.7 Klik op Next

9.10 Voer “restassuredworkshop” in bij Project name.

9.11 Selecteer je workspace folder als Project location

9.12 Klik op Finish

We hebben zojuist een nieuw Maven project aangemaakt.

Maven is een gereedschap voor Java-projectmanagement en geautomatiseerde softwarebouw.

Door gebruik te maken van maven wordt de projectstructuur alvast voor je gereed gemaakt.

Het is goed om te weten dat we met REST Assured de API calls kunnen doen tegen een webserver en informatie kunnen onttrekken uit de response, maar dat we daar verder nog niet veel aan hebben wanneer je wilt gaan testen.  
  
Voor het testen hebben wij een framework nodig waarmee je testen kunt uitvoeren.  
  
Voor deze workshop heb ik gekozen voor het JUnit Framework.  
JUnit maakt het mogelijk om onder andere assertions uit te voeren.

Omdat wij binnen dit project afhankelijk zijn van REST Assured voor de API calls en JUnit voor de assertions zullen wij deze als afhankelijkheid moeten toevoegen aan de pom.xml.

Voor degene die niet weten wat de pom is:  
In het kort kun je zeggen dat de pom.xml alle informatie bevat om een project te kunnen bouwen.

Bijvoorbeeld: Wanneer iemand jou een pom.xml bestand verstuurt per mail en deze pom bevat voldoende informatie dan heb je niet meer nodig om het project op te kunnen bouwen. Wanneer je dit bestand importeert in je IDE als IntelliJ of Eclipse, dan zal maven alle source code en alle dependencies downloaden. De Testcases draaien, de jar/war bestanden bouwen, de applicatie deployen, een rapport opbouwen van de code kwaliteit etc.

Het micronaut commando mn create-app … om een applicatie te maken heeft een pom.xml bestand gegenereerd binnen het project.

De toevoeging --features=junit in het commando om het project aan te maken heeft er voor gezorgd dat de jUnit dependencies zijn toegevoegd.

Ik ga er in deze workshop niet dieper op in maar we zullen er wel gebruik van maken.

10. Voeg REST Assured als afhankelijkheid toe aan de pom:

10.1 Open de pom in IntelliJ

10.2 Voeg onderstaande toe:

<properties>

<junit.version>4.12</junit.version>

<rest-assured.version>4.0.0</rest-assured.version>

</properties>

<dependencies>

<dependency>

<groupId>io.rest-assured</groupId>

<artifactId>rest-assured</artifactId>

<version>${rest-assured.version}</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>${junit.version}</version>

</dependency>

</dependencies>

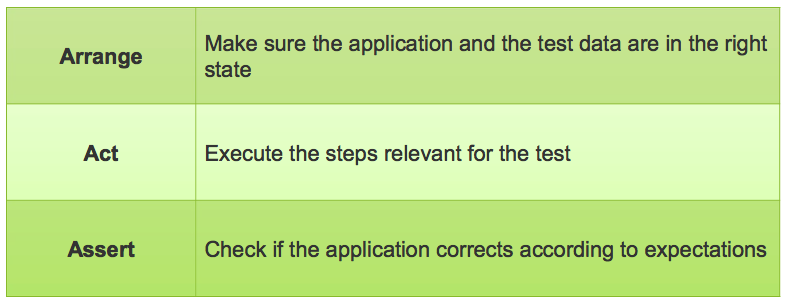
10.3 Doe vervolgens een maven clean-compile zodat alle benodigde dependencies

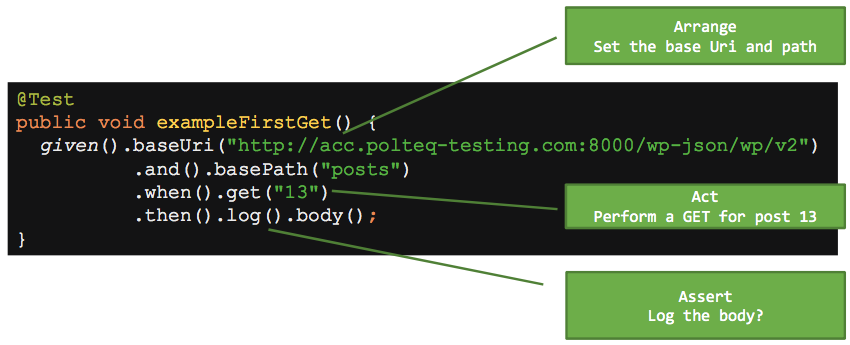
op basis van de pom.xml worden gedownload.

We hebben zojuist het project gereed gemaakt om testen uit te kunnen voeren.  
We hebben ondersteuning voor JUnit 4 en we hebben REST assured tot onze beschikking.

Nu we ons gereedschap hebben klaargezet kunnen we beginnen met het opbouwen van onze testen.

Voor het bouwen van onze testen is het goed om te weten dat we deze opbouwen met een bepaalde structuur.

De structuur van testen:  




Tijdens deze workshop zullen we ons beperken tot het testen van een aantal status codes en de “Hello world” response.

**Extra info:**  
*http kent verschillende categorieën voor status codes:*

*1xx: Informatief*

*2xx: Succesvol*

*3xx: Redirect*

*4xx: Client error*

*5xx: Server error*

11. maak je eerste REST assured test

11.1 Maak binnen je project onder src > test > java een nieuwe Package aan en noem deze “nl.bkwi.workshop”. (rechtermuis op de java folder > New > package)

11.2 Maak binnen deze package een nieuw Java klasse aan en geef deze een naam.

Bijvoorbeeld “MyFirstRestAssuredTestClass”

Rechtermuisknop op de package > New > Java Class

11.3 Voeg onderstaande import toe aan de zojuist aangemaakte testklasse:

*import static io.restassured.RestAssured.get;*

11.4. Voeg onderstaande testmethode toe aan de zojuist aangemaakte testklasse:

*@Test*

*public void endpointHelloWorldRespondsWithStatuscode(){*

*given()*

*.baseUri("http://localhost:9090")*

*.when()*

*.get("/helloworld")*

*.then().statusCode(200);*

*}*

Kunnen jullie mij vertellen wat er hier gebeurt en of deze test slaagt wanneer je deze uitvoert in IntelliJ?

We gaan nog een aantal extra testen toevoegen die controleert of de correcte String (tekst) terug komt.

12: Voeg een string assertion test toe:

12.1 Voeg onderstaande testmethoden toe aan de zojuist aangemaakte testklasse:

*@Test*

*public void endpointHelloWorldRespondsWithCorrectString(){*

*Response response = given()*

*.baseUri("http://localhost:9090")*

*.when()*

*.get("/helloworld!");*

*assertEquals("Hello World", response.getBody().asString());*

*}*

*@Test*

*public void endpointHelloWorldRespondsWithCorrectStringWithThen(){*

*given()*

*.baseUri("http://localhost:9090")*

*.when()*

*.get("/helloworld")*

*.then().body(equalTo("Hello World!"));*

*}*

Kunnen jullie mij het verschil uitleggen tussen bovenstaande twee testmethoden en of deze zullen slagen?

Tot nu toe hebben we een microservice gemaakt die zelf de hello world string terug geeft.  
Je zou kunnen zeggen dat dit betekend dat deze microservice zowel de front als de backend implementatie voor zijn rekening neemt.

Met microservices is het natuurlijk de bedoeling dat je een verzoek kunt sturen naar service A (Front end) en dat service B (Back end) de afhandeling doet.  
De API gateway is bekend met alle verzoeken die binnen kunnen komen maar laat de verwerking (afhankelijk van het verzoek, op basis van endpoint) uitvoeren door de service die daarvoor verantwoordelijk is.  
  
**4. Het bouwen van een tweede microservice**

13. Bouw met behulp van micronaut je tweede microservice

13.1 Ga in de terminal naar dezelfde folder als waar je eerste service staat en voer onderstaand commando uit:

*mn create-app nl.bkwi.workshop.mymicronautservice.my-second-micronaut-service \*

*--build=maven --features=java,junit*

13.2 Open dit project met IntelliJ.

14. Bouw de Hello World By Name controller klasse.

14.1 Maak een nieuw java klasse bestand aan in dezelfde folder waar het “Application” bestand staat en noem dit bestand “HelloWorldByNameController”

14.2 Importeer de Controller en Get annotatie:

*import io.micronaut.http.annotation.Controller;*

*import io.micronaut.http.annotation.Get;*

14.3 Voeg de controller annotatie toe aan de klasse en het pad van de endpoint:

@Controller(”/wereldvan”)

14.4 Voeg de Get annotatie toe met een geparametriseerd endpoint:

@Get("/{name}")

Wat doen we met /{name} ?

14.5 Implementeer de afhandeling voor dit endpoint

*public String getWorldByName(String name) {*

*String helloWorldName = "Hoi " + name + "'s wereld";*

*return helloWorldName;*

*}*

Hoe kunnen we bovenstaande return statement ook uitvoeren?

14.6 Pas de poort aan waarop de applicatie luisterd

src > main > resources > application.yml

voeg de server poort “7070” toe zoals we dat ook voor de andere microservice hadden gedaan.

14.7 Start de microservice en controleer de werking

<http://localhost:7070/wereldvan/wesley>

Als het goed is zie je nu een mooie string in de browser terugkomen.

Op dit moment hebben we dus twee microservices draaien die beiden een string terug kunnen geven.  
Het is natuurlijk de bedoeling dat we microservice 1 kunnen aanspreken en dat microservice 2 de afhandeling doet.

Dit wordt mogelijk gemaakt door op microservice 1 een “Client” interface te introduceren.  
Deze client is nodig om het verzoek uitzetten bij microservice 2 en de response terugkoppelen aan de bevragende applicatie.

15 Introduceer de Client op microservice 1.

15.1 Open de eerste microservice met IntelliJ.

15.2 Maak een nieuwe Java klasse genaamd “HelloWorldByNameClient” van het type “Interface”.

15.3 Voeg de Client annotatie toe aan de interface die verwijst naar de tweede microservice.

*@Client("http://localhost:7070/wereldvan")*

*public interface HelloWorldByNameClient {*

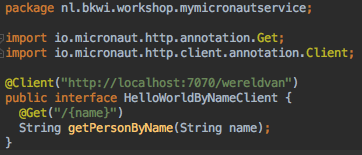
*}*

15.4 voeg de @Get annotatie toe binnen de interface en hang deze aan een methode

@Get("/{name}")

String getPersonByName(String name);

Je Client interface zal er nu zo uitzien:



16. Koppel de client aan de controller zodat er via deze microservice de andere microservice kan worden bevraagd.

16.1 Injecteer de client *binnen de controller klasse* en koppel deze aan een variabele:

*@Inject*

*HelloWorldByNameClient externeHelloService;*

16.2 Maak binnen de controller een endpoint die op de achtergrond de tweede microservice bevraagd

*@Get("/remote/{naam}")*

*public String getPersonsName(String naam) {*

*return externeHelloService.getPersonByName(naam);*

*}*

Als het goed is ziet je controller klasse er op dit moment (ongeveer) zo uit:

@Controller("/")  
public class HelloWorldController {

@Inject  
 HelloWorldByNameClient externeHelloService;

@Get("/helloworld")  
 public String getHelloWorldReponse() {  
 return "Hello World";  
 }  
  
 @Get("/remote/{naam}")  
 public String getPersonsName(String naam) {  
 return externeHelloService.getPersonByName(naam);  
 }  
  
}

Nu je de code zo ziet, kun je nu de verbanden leggen en mij vertellen wat er zou moeten gebeuren wanneer we dit /remote/<naam> bevragen in de browser?

Hieronder kun je zien hoe de bevragingstroom loopt.

De gebruiker bevraagt in de browser een webserver op poort 9090 met daarbij een entrypoint /remote/piet.

De controller op deze webserver heeft een Get annotatie die luistert naar /remote/<string>.

De onderliggende methode binnen deze Get annotatie heeft een return waarde die de @Get methode binnen de client interface aanspreekt.  
Deze client interface staat zo ingesteld dat alle bevragingen gedaan moeten worden naar een webserver die draait op poort 7070 en de aangesproken methode heeft een Get annotatie met daaraan een endpoint /name.

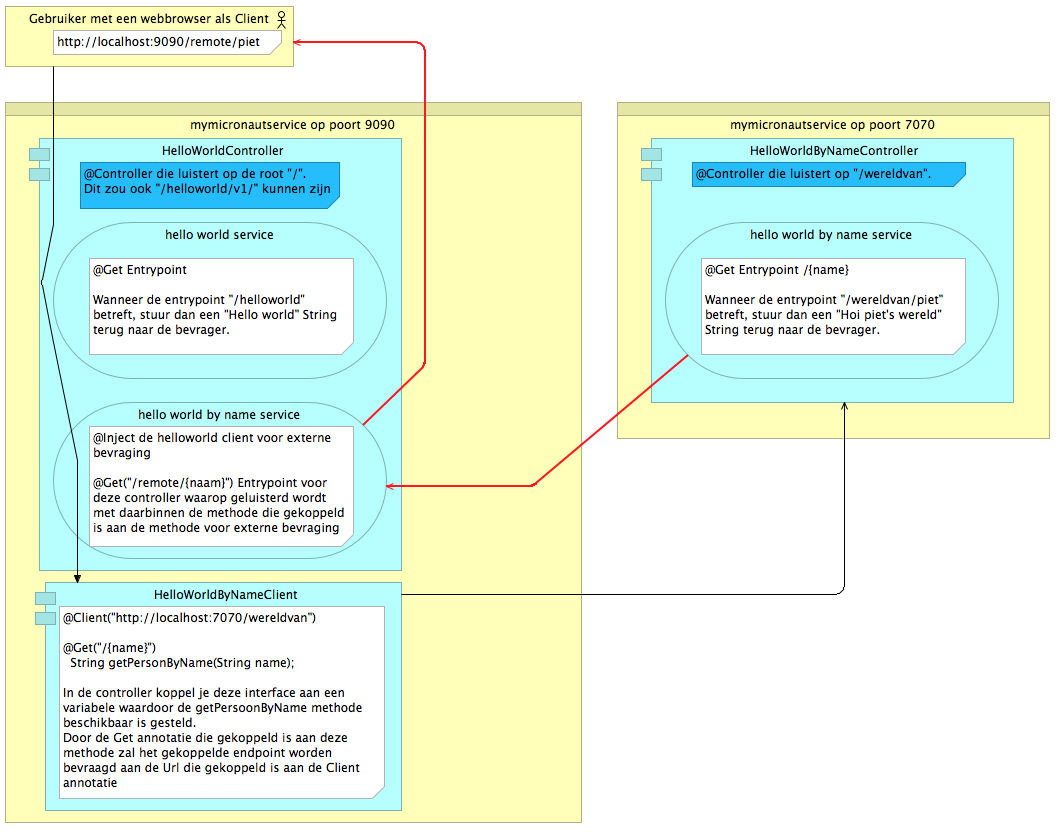
Op het moment dat de gebruiker een localhost:9090/remote/name opgeeft

Wordt er onderwater een localhost:7070/wereldvan/name bevraging gedaan.

Het verzoek komt eerst binnen bij de controller, in de controller is de client geïnjecteerd en die zet het verzoek door naar het backend systeem.  
  
Qua routing van het netwerkverkeer zou je het kunnen vergelijken met een tunnel maar daar is geen sprake van. Het lijkt eerder op een interne redirect waarbij de binnenkomende controller als bemiddelaar fungeert.

Wanneer het verzoek binnen komt op HOST-X/aap/noot/mies, zet dan het verzoek door naar HOST-Y/gijs/lam/kees/bok.

De waarde die daar uit komt wordt terug gegeven aan de browser van de gebruiker.



Nu we een simpele hello service hebben gemaakt met front- en backend kunnen we een stapje verder gaan door in plaats van hello world, een dier terug te geven.  
Om dit te realiseren maken we een derde microservice die dit gaat doen.

5. Maak een nieuwe microservice

mn create-app nl.bkwi.workshop.mymicronautservice.my-animal-service \

--build=maven --features=java,junit

Zoals jullie weten heb je bij het maken van eigen objecten te maken met “getters” en “setters”.

Daarnaast heb je tijdens het programmeren te maken met bepaalde ontwerp keuzes.

Met onderstaande service zullen we gebruik maken van het [Builder (design) Pattern](https://dzone.com/articles/design-patterns-the-builder-pattern).  
  
Met een builder kun je een object bouwen zonder dat je bij het aanmaken van het nieuwe object, tig parameters moet doorgeven. Een voorbeeld van het aanmaken van een nieuw dier object:

*static Dog dog = new MutableAnimals.DogBuilder()  
 .candy("Honden snoepjes")  
 .color("bruin")  
 .ras("herder")  
 .barkSound("WHOOF!!!")  
 .speeltje("Rubberen eend en touw")  
 .build();*  
  
Hierbij zijn de properties van het dier ‘candy’, ‘color’, enzovoorts. Deze kun je ‘setten’ door deze poperties aan te roepen (met .) bij het dier en een waarde te geven, zoals in bovenstaand voorbeeld.

Wanneer je te maken krijgt met grote objecten zorgt het implementeren van alle getters en setters voor onoverzichtelijke java klassen en het bouwen van een eigen builder implementatie kost veel tijd.

Om de code overzichtelijk te houden en veel werk uit handen te nemen zullen we in deze microservice gebruik maken van [Project Lombok](https://projectlombok.org/features/Builder).  
Lombok maakt het leven van een ontwikkelaar gemakkelijk, omdat deze (onder andere) een builder en alle getters en setters voor je kan genereren.

Dit gebeurd op basis van de Annotaties @Data en @Builder.  
  
5.1 Voeg het Lombok project toe aan de dependencies

<dependency>  
 <groupId>org.projectlombok</groupId>  
 <artifactId>lombok</artifactId>  
 <version>1.18.8</version>  
 <scope>provided</scope>  
</dependency>

5.2 Voeg het pad toe van het Lombok project aan de annotationProcessorPaths:

<path>  
 <groupId>org.projectlombok</groupId>  
 <artifactId>lombok</artifactId>  
 <version>1.18.0</version>  
 </path>

5.3 Installeer ook de lombok plugin binnen de plugins van Intellij.

5.4 Maak een nieuwe java klasse genaamd “MutableAnimals”, plak daar onderstaande code in en analyseer of je door hebt wat er in deze klasse gebeurt:

@Builder(builderMethodName = "dogBuilder")  
public static MutableAnimals.Dog newDog(String ras, String color, String barkSound, String candy, String speeltje) {  
 final MutableAnimals.Dog dog = new MutableAnimals.Dog();  
 dog.setBarkSound(barkSound);  
 dog.setColor(color);  
 dog.setFavoriteCandy(candy);  
 dog.setRasType(ras);  
 dog.setFavoriteToy(speeltje);  
 dog.setNote(MutableAnimals.PetService.*treatService*(dog));  
 return dog;  
}  
  
@Builder(builderMethodName = "catBuilder")  
public static MutableAnimals.Cat newCat(String rasSoort, String color, String meowSound, String candy,String toy) {  
 final MutableAnimals.Cat cat = new MutableAnimals.Cat();  
 cat.setMeowSound(meowSound);  
 cat.setColor(color);  
 cat.setFavoriteCandy(candy);  
 cat.setRasType(rasSoort);  
 cat.setFavoriteToy(toy);  
 cat.setNote(MutableAnimals.PetService.*treatService*(cat));  
 return cat;  
}  
  
@Data  
public static class Cat implements MutableAnimals.Animal {  
 String color;  
 String meowSound;  
 String name;  
 String favoriteCandy;  
 String rasType;  
 String favoriteToy;  
 String note;  
}  
  
@Data  
public static class Dog implements MutableAnimals.Animal {  
 String color;  
 String barkSound;  
 String favoriteCandy;  
 String rasType;  
 String favoriteToy;  
 String note;  
}  
  
public interface Animal {  
 String getRasType();  
 String getFavoriteCandy();  
 String getFavoriteToy();  
 String getNote();  
}  
  
@Data  
public static class PetService {  
 public static String treatService(MutableAnimals.Animal dier) {  
 return "Een " + dier.getRasType() + " Snoept graag " + dier.getFavoriteCandy() + " en speelt graag met een " + dier.getFavoriteToy();  
 }  
}

Extra info:  
Mutable wil zeggen dat het object nog aangepast kan worden.  
Bij een immutable object kan dat niet.  
  
Een immutable dier object zou er zo uit kunnen zien:  
 *@Builder(builderMethodName = "dogBuilder")*

*public static Dog newDog(String color, String barkSound) {*

*return new Dog(color, barkSound);*

*}*

5.5 Nu we deze klasse hebben ingericht, gaan we er voor zorgen dat de controller er wat mee kan doen.  
Voeg onderstaande code toe aan de Controller klasse:

static Cat *cat* = new MutableAnimals.CatBuilder()  
 .candy("Katten snoepjes en vis")  
 .color("Wit")  
 .rasSoort("Blauwe Rus")  
 .meowSound("Prr")  
 .toy("Nep muis")  
 .build();  
  
static Dog *dog* = new MutableAnimals.DogBuilder()  
 .candy("Honden snoepjes")  
 .color("bruin")  
 .ras("herder")  
 .barkSound("WHOOF!!!")  
 .speeltje("Rubberen eend en touw")  
 .build();  
  
@Get("/dieren/v1/dier/{dier}")  
public Animal dier(String dier) {  
  
 dier = dier.toLowerCase();  
  
 if (dier.equals("cat")){  
 return *cat*;  
 } else if (dier.equals("dog")) {  
 return *dog*;  
 } else  
 return null;  
}

//Ik zou hier ‘kat’ en ‘hond’ van maken.

In bovenstaande code wordt er bij het opstarten van de applicatie een hond en een kat object aangemaakt.  
Wanneer het endpoint wordt bevraagd, zal er aan de hand van de informatie die beschikbaar is een kat, hond of null waarde worden terug geleverd aan de bevrager.  
  
Voeg voor jezelf een paar dieren toe aan de klasse zodat je kunt merken dat het werken met builders prettig werkt en dat Lombok wel erg mooi is om te gebruiken binnen je project.   
  
5.6 Pas de poort aan van deze microservice en doe eens een bevraging richting deze microservice.  
Voldoet het antwoord aan je verwachting?  
  
5.7 Pas de client service aan zodat ook deze microservice via de front end bevraagd kan worden.

5.7.1 Maak een nieuwe interface en voeg de Client en Get annotaties toe zoals hieronder als voorbeeld wordt gegeven.

@Client("http://localhost:6060/dieren/v1/dier/")  
public interface AnimalClient {  
  
 @Get("/{catOrDog}")  
 List<String> getAnimal(String catOrDog);  
}

5.7.2 Injecteer deze interface binnen de controller zoals hieronder als voorbeeld wordt weergeven.

@Inject  
AnimalClient animal;  
  
@Get("/remote/animalservice/{catOrDog}")  
public List<String> getAnimal(String catOrDog) {  
  
 return animal.getAnimal(catOrDog);}

5.8 Herstart de front end microservice en doe een bevraging.

Is de response zoals je had verwacht?

**6. Voeg een logger toe om te kunnen zien wat er met de verzoeken gebeurt op de backend dieren microservice.**

Om te kunnen trouble shooten kan het handig zijn om logging toe te voegen.   
Micronaut heeft standaard een logger in zijn arsenaal aan tools tot zijn beschikking genaamd

“Logback”.

6.1 Onder src>main>resources is het logback.xml bestand te vinden waarin de configuratie is opgenomen.

Dit bestand kun je naar wens aanpassen, maar ziet er zo uit:

<configuration>  
  
 <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">  
 <withJansi>true</withJansi>  
 <!-- encoders are assigned the type  
 ch.qos.logback.classic.encoder.PatternLayoutEncoder by default -->  
 <encoder>  
 <pattern>%cyan(%d{HH:mm:ss.SSS}) %gray([%thread]) %highlight(%-5level) %magenta(%logger{36}) - %msg%n</pattern>  
 </encoder>  
 </appender>  
 <logger name="io.micronaut.http.client" level="TRACE"/>  
 <root level="info">  
 <appender-ref ref="STDOUT" />  
 </root>  
</configuration>

Om gebruik te maken van deze logger dien je deze toe te voegen aan de gewenste klasse waarin je logging wilt toepassen.

In dit voorbeeld zullen we deze toevoegen aan de AnimalController klasse.

6.2 voeg de private static final Logger regel toe aan de controller klasse.

@Controller("/")  
public class AnimalController {  
 private static final Logger *LOG* = LoggerFactory.*getLogger*(AnimalController.class);

Deze logger kun je binnen methoden aanspreken om je eigen logregels toe te voegen maar standaard zal er vanaf nu ook nuttige informatie naar de console worden uitgespuugd.

6.3 Herstart de dieren microservice en doe nogmaals een bevraging via de front end.

Zie je de logregels nu in de dieren microservice console voorbij komen en geven deze nuttige informatie?

Dat we de microservices hebben draaien via de IDE is natuurlijk leuk, maar we willen deze het liefst als Docker containers hebben draaien zodat we ze ook naar een register gepusht zouden kunnen worden.

Om een docker image te kunnen maken hebben we een Dockerfile nodig die we moeten inrichten om dit mogelijk te maken.  
  
Normaal gesproken kan dit best een frustrerend werkje zijn maar heeft micronaut daar zelf al aan gedacht.  
Tijdens het aanmaken van het project is er ook een Dockerfile gegenereerd.  
  
Het is dan ook gelijk mogelijk om binnen je werkfolder van je project een “docker build –t <tagname> .” uit te voeren, dat er voor zorgt dat er een docker container wordt gebouwd.

Met de –t flag geef je een naam mee aan de image zodat je deze makkelijk kunt starten.

Wanner je de –t flag niet meegeeft zal er een eigen naam aan gegeven worden.  
  
De . (punt) aan het eind geeft aan dat je de docker image in de huidige directory wilt aanmaken.

Het commando “Docker build –t test-container .” bouwt dan ook een container in de huidige folder.

Het commando “docker run test-container” zal de container starten en de webserver die we hebben gebouwd zal op deze container worden gestart op de opgegeven poort.  
  
Omdat de container op zn eigen netwerk draait kunnen we de webserver niet bereiken.  
Om dit wel te realiseren zullen we een portmapping/port forward moeten doorgeven bij het starten van de container.  
Dit doe je door de –p flag mee te geven bij het starten van de container, gevolgd door <interne poort>:<container poort>.  
Het commando ziet er dan als volgt uit “docker run –p 8080:6050 test-container”  
Wanneer je nu in de browser naar <http://localhost:8080> gaat wordt het netwerk verkeer doorgestuurd naar de test-container op poort 6050.

Wanneer je de applicatie start met een docker container en de regressietest ook vanuit een container zullen de testen falen.

Dit komt doordat je op je eigen machine wel een port forward hebt richting de applicatie container maar daar heeft de test container geen weet van.

Dit komt omdat elke container op zijn eigen netwerk draait.

Je zult de test container dus moeten starten op het netwerk vanaf je eigen machine of door beide containers op hetzelfde docker netwerk te starten.

Start de test container met: "docker run --net=host " en de docker container zal op hetzelfde netwerk starten als je eigen computer.

De test container zal nu starten en de test uitvoeren tegen de api die draait op de andere container.

Dit wordt mogelijk gemaakt door de "--net=host" tag.

Wat je ook kunt doen is een virtuele netwerk laag creeren en beide containers op dat netwerk starten:

$ docker network create mynet $ docker run --name foo --net mynet img $ docker run --name bar --net mynet img

Bijvoorbeeld: "docker network create virtuele\_netwerk\_naam"

Start de api container op dat netwerk: "docker run --name netwerknaam\_api --net virtuele\_netwerk\_naam -p 8081:8080 <applicatie\_container>"

start de testcontainer: docker run --name netwerknaam\_test --net virtuele\_netwerk\_naam --env applicatie=netwerknaam\_api test\_image\_name

--env geef je mee zodat de testclasse deze variabele over kan nemen en tegen de juiste container de test start. "String eigenHostname = System.getenv("applicatie");""

Aan jullie de opdracht om beide mogeijkheden uit te testen.

Veel succes en vooral VEEL PLEZIER!