**Spring Cloud, Spring Boot ve Docker kullanarak Mikro Servis Projesi Hazırlamak**

Bu makalede Spring ailesini kullanarak ve çok ayrıntıya girmeden bir mikro servis projesi hazırlayacağız. Başlamadan önce mikro servis nedir neden bu kadar yaygınlaşmaya başlamıştır kısaca ondan bahsetmek istiyorum. Daha ayrıntılı bilgi için <https://microservices.io> adresini ziyaret edebilirsiniz.

Mikro servis, bir yazılım geliştirme tekniğidir. Sadece belirli biri işten sorumlu, ölçeklenebilir, “loose-coupled” yapıya sahip servistir.

Faydalar:

* **Modüler**

Geliştirilmesi, anlaması ve test edilmesi kolay.

* **Ölçeklendirilebilir**

Birbirlerinden bağımsız süreçler olarak çalıştıklarından dolayı ölçeklendirirken daha özgür olabiliriz.

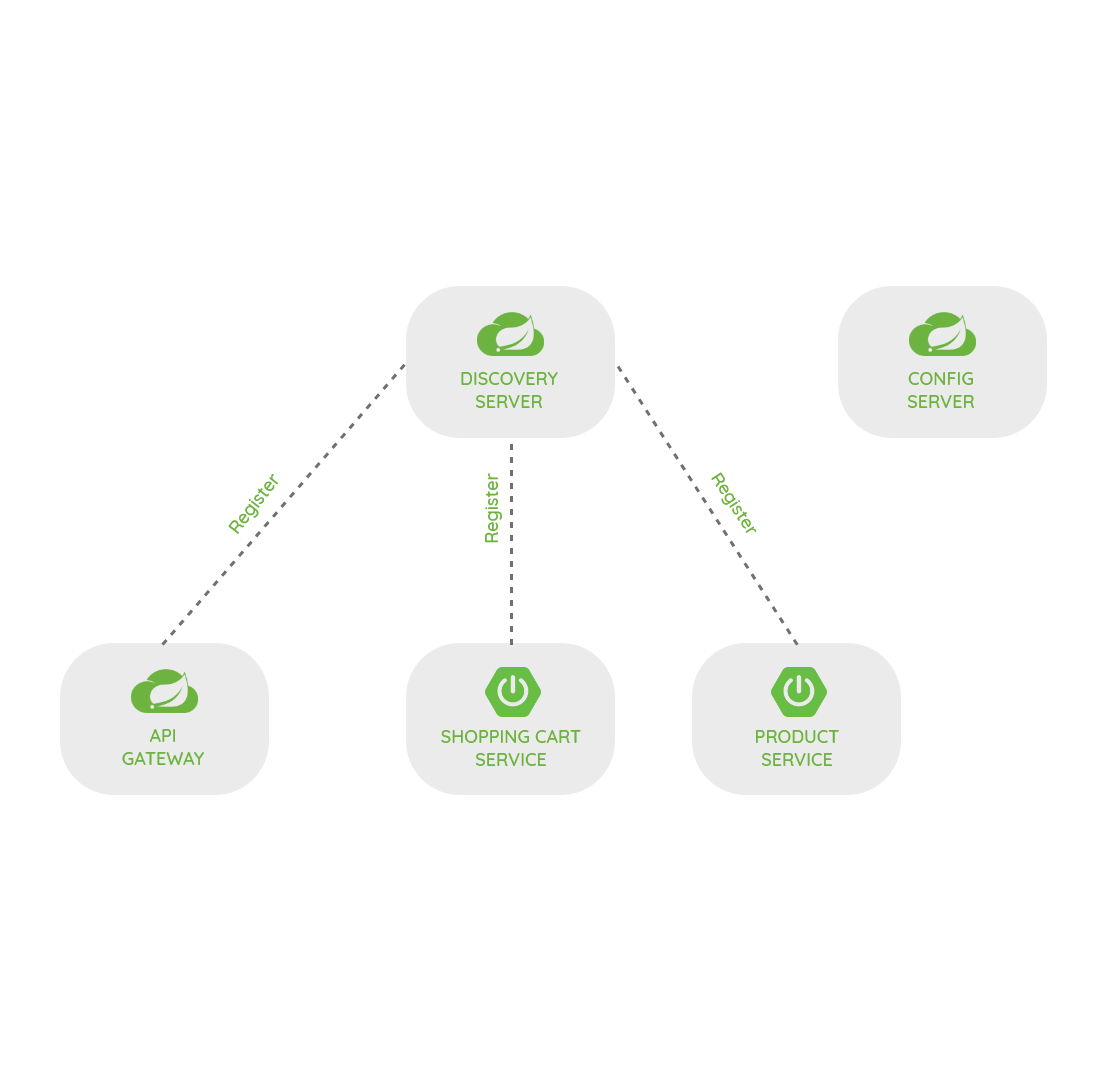
* **Kolay dağıtılabilir**

Küçük ve hızlı projeler oldukları için deploy edilmesi de daha kolay.

Şimdi projemize geçelim. Projemiz aşağıdakiler bileşenleri içermektedir. Yeri geldikçe bileşenler hakkında biraz bilgi ve projedeki görevlerini anlatmaya çalışacağım.

* Discovery Server (Spring Eureka)
* Config Center (Spring Cloud Config)
* API Gateway (Spring Cloud Netfliz - Zuul)
* Service
* Product Service

Proje sonlandığı zaman elimizde böyle bir ekosistem olacak.



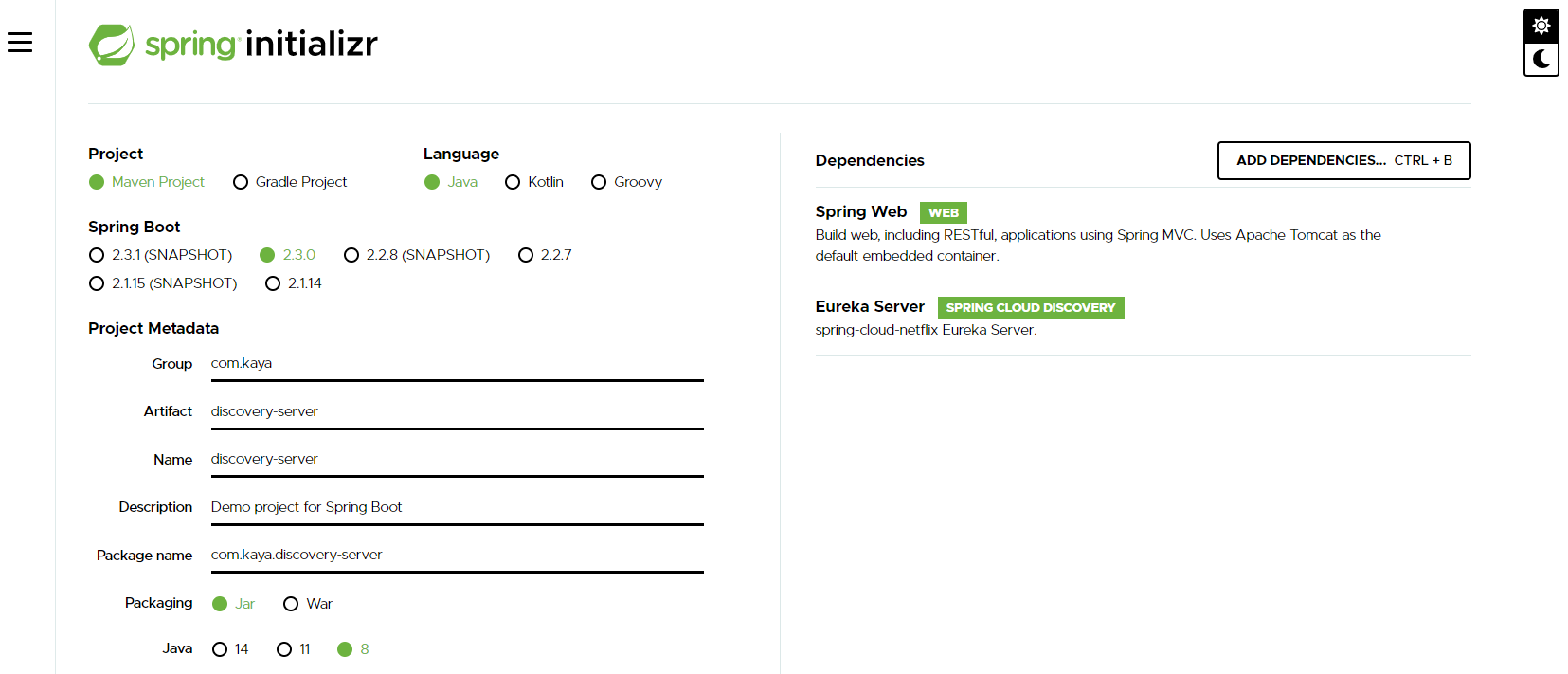
1. **Discovery Server**

Discovery Server için **Spring Eureka** projesini kullanacağız.

Discovery Server’ın amacı kendisine kayıt yapmış servislerin bilgisini tutmaktır. Bir mikro servis ekosistemde birden fazla servis ve bu servislerin birden fazla kopyası bulunabilir. Bu servislerin birbirleriyle bir noktada iletişim kurması gerekecektir. Ama bir servis başka bir servisin hangi ip ve portta çalıştığını bilemez. Discovery servis bu sorunu çözmektedir.

Çalışan her servis, Discovery Server’a kayıt olur. Ve böylelikle Discovery Server hangi servislerin ayakta olduğunu, onların ip ve port bilgileri gibi çeşitli veriyi kendinde saklar.

Spring initializr ([start.spring.io](https://start.spring.io/)) sayfasına girerek projemizi oluşturuyoruz.



Projeyi oluşturduktan sonra **bootstrap.properteis** dosyasının içine bazı ayarlar eklememiz gerekiyor.

spring.application.name=discovery-server

server.port=5858

eureka.client.registerWithEureka=false

eureka.client.fetchRegistry=false

eureka.client.serviceUrl.defaultZone=http://discovery-server:5858/eureka

Daha sonra **DiscoveryServerApplication.java** içine girip. Class’ımıza **@EnableEurekaServer** annotation’ını eklememiz gerekiyor. Böylelikle projemizi bir Eureka Server haline getiriyoruz.

@SpringBootApplication

@EnableEurekaServer

public class DiscoveryServerApplication {

public static void main(String[] args) {

SpringApplication.run(DiscoveryServerApplication.class, args);

}

}

Son adım olarak Discovery Server’ımız dockerize edeceğiz. Böylelikle projemiz Docker’ın kurulu olduğu herhangi bir ortamda rahatlıkla ayağa kalkabilicek ve istersek Kubernetes gibi container-orchestration orkestrasyon toolarıyla server’imizi deploy edebileceğiz.

**Dockerfile** isimli bir dosya oluşturup içine ayarlarımız ekliyoruz.

FROM openjdk:8-jdk-alpine

ARG JAR\_FILE=target/\*.jar

COPY ${JAR\_FILE} app.jar

ENTRYPOINT ["java","-jar","/app.jar"]

Projenin dizinine gidip projemizi build ediyoruz.

mvnw package

Artık elimizde deploy edebileceğimiz bir ürün var. Bir sonrki adımda docker image dosyamızı oluşturacağız. Image’leri çalıştırmanın birden fazla yolu var ancak ben bir docker-compose özelliğini kullanacağım. Böylelikle projemizdeki bütün servislerimizi build ettikten sonra birkaç satır komut ile projemizin tamamını ayağa kaldırabiliceğiz.

docker-compose.yml dosyasını oluşturup içini dolduruyoruz

version: '3'

services:

discovery-server:

build:

context: ./discovery-server

dockerfile: Dockerfile

image: discovery-server-docker

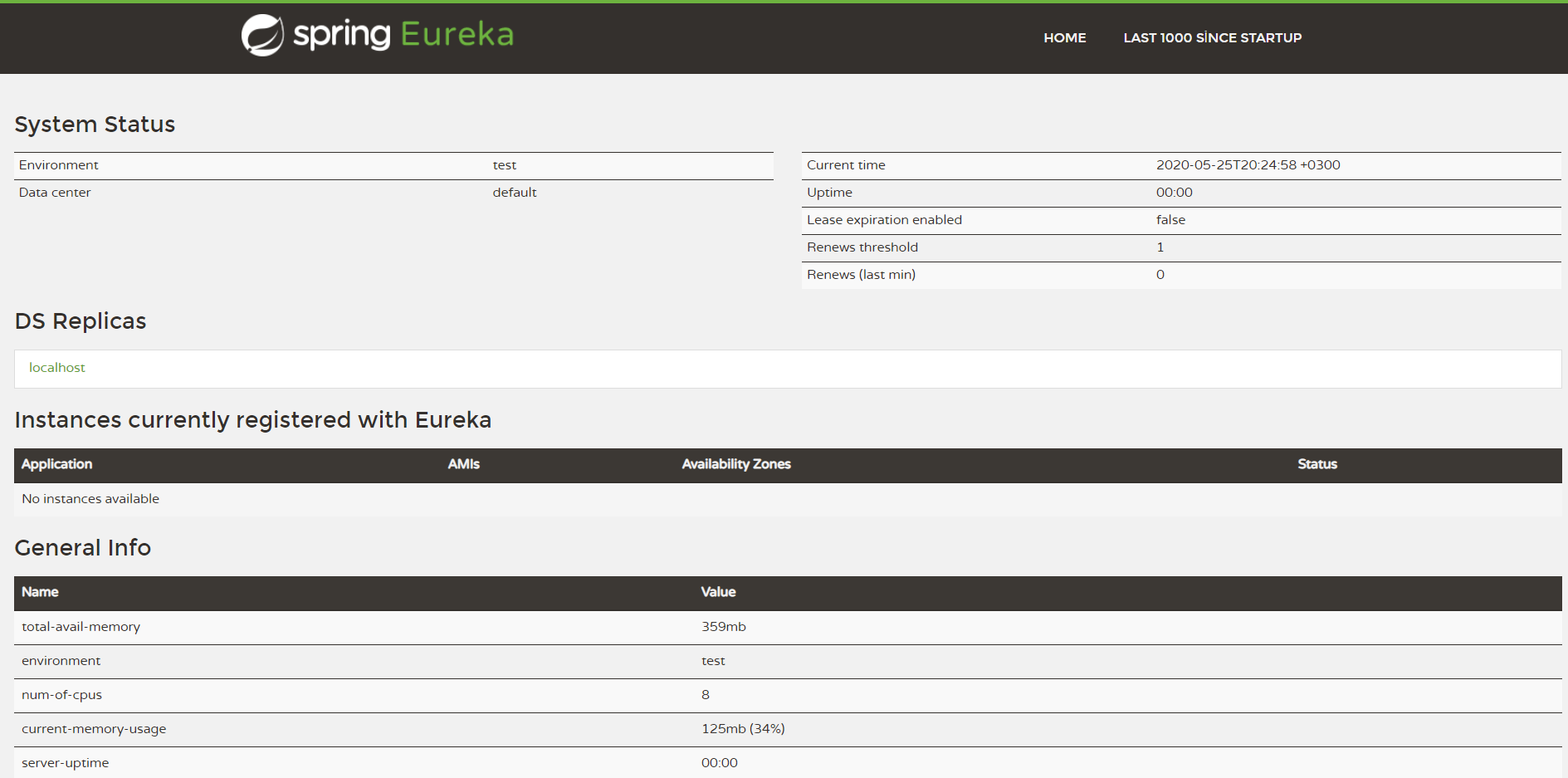
ports:

- 5858:5858

Şimdi docker-compose komutunu kullanarak projemizi ayağa kaldırırıyoruz.

docker-compose up –d discovery-server

Sorunsuz bir şekilde çalıştırdıktan sonra <http://localhost:5858> adresinden Discovery Server arayüzüne ulaşabiliriz.

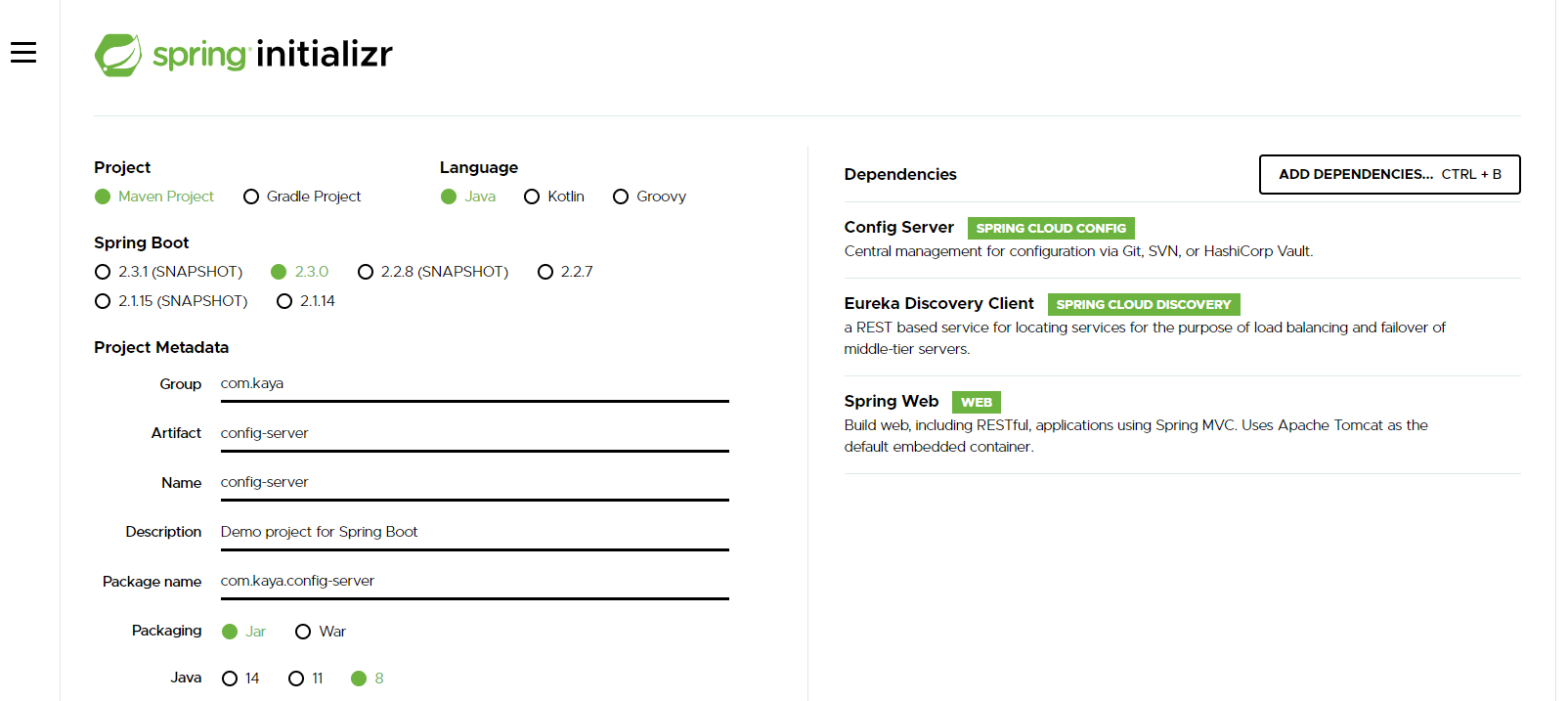


Buradaki **“Instances currently registered with Eureka”** bölümünün altında serverımıza kayıt olmuş servisleri görebiliriz.

Projemizdeki diğer servislerimizi yazmaya başladıktan sonra, burada ekranda servislerimizi göremeye başlayacağız.

1. **Config Server**

Spring initializr ile projemizi oluşturuyoruz.



Oluşturduktan sonra **bootstrap.properties** dosyasına ayarlarımızı gireceğiz.

spring.application.name=config-server

server.port=5859

spring.profiles.active=native

eureka.client.serviceUrl.defaultZone=http://discovery-server:5858/eureka

**spring.profiles.active:** Config dosyalarının nasıl sunulacağını gösterir. Native diyerek configlerimizin local’de olduğunu söylüyoruz.

**eureka.client.serviceUrl.defaultZone:** Discovery Server adresi. Eğer ortam değişkenlerinde DISCOVERY\_SERVER tanımlı değilse default değere olarak localhost:5858/eureka adresine bakmasını söylüyoruz. Ama bir docker-compose dosyamızın içinde ortam değişkeni olarak Discovery Server’imizi tanımlayacağız.

**ConfigServerApplication.java** dosyamıza girip **@EnableConfigServer** ve **@EnableEurekaClient** anotasyonlarını ekliyoruz. Böylece Spring Cloud Config Server özelliğini açıyoruz ve Discovery Server’ımıza kayıt oluyoruz.

Daha sonra config dosyalarımız ekliyoruz. **Resources** klasörünün altına **config** isminde bir klasör açıyoruz ve config dosyalarımız ekliyoruz. İçerilerine şimdilik sadece Discovery Server’imizin adresini giriyoruz.

shopping-cart-service.yml:

eureka:

client:

serviceUrl:

defaultZone: "http://discovery-server:5858/eureka"

product-service.yml:

eureka:

client:

serviceUrl:

defaultZone: "http://discovery-server:5858/eureka"

api-gateway.yml:

eureka:

client:

serviceUrl:

defaultZone: "http://discovery-server:5858/eureka"

Son olarak servisimizi Dockerize edeceğiz. Bu işlem için Discovery Server’in image dosyasını oluştururken izlediğimiz adımları tekrar yapacağız.

Dockerfile

FROM openjdk:8-jdk-alpine

ARG JAR\_FILE=target/\*.jar

COPY ${JAR\_FILE} app.jar

ENTRYPOINT ["java","-jar","/app.jar"]

Projenin dizinine gidip projemizi build ediyoruz.

mvnw package

docker-compose.yml dosyamıza ekliyoruz

…

config-server:

build:

context: ./config-server

dockerfile: Dockerfile

image: config-server-docker

ports:

- 5859:5859

depends\_on:

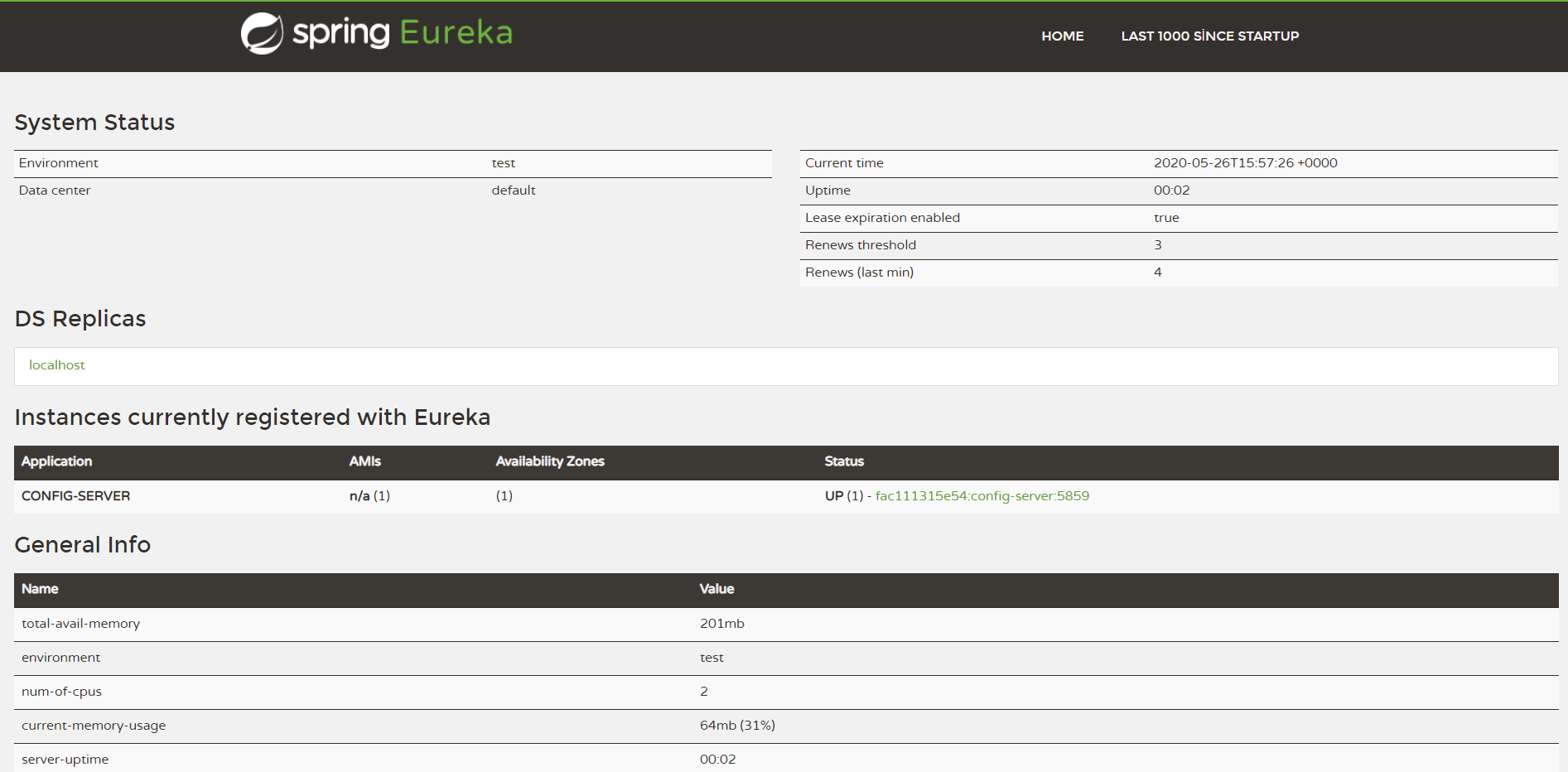
- discovery-server

…

Projemizi çalıştırıyoruz

docker-compose up –d config-server

Daha sonra Dicovery Server’imizi kontrol ediyoruz.



Config Server’a istek atarak configlerimiz fetch edebiliriz. Formatın bu şekilde olması gerekir:

[http://localhost:5859/{application-name}/{profile}/{label}](http://localhost:5859/%7bapplication-name%7d/%7bprofile%7d/%7blabel%7d)

Biz default profili kullanacağız: Bu durum şöyle bir istek atarak configlerimize ulaşabiliriz:

<http://localhost:5859/product-service/default>

{

"name": "product-service",

"profiles": [

"default"

],

"label": null,

"version": null,

"state": null,

"propertySources": [

{

"name": "classpath:/config/product-service.yml",

"source": {

"eureka.client.serviceUrl.defaultZone": "http://localhost:5858/eureka"

}

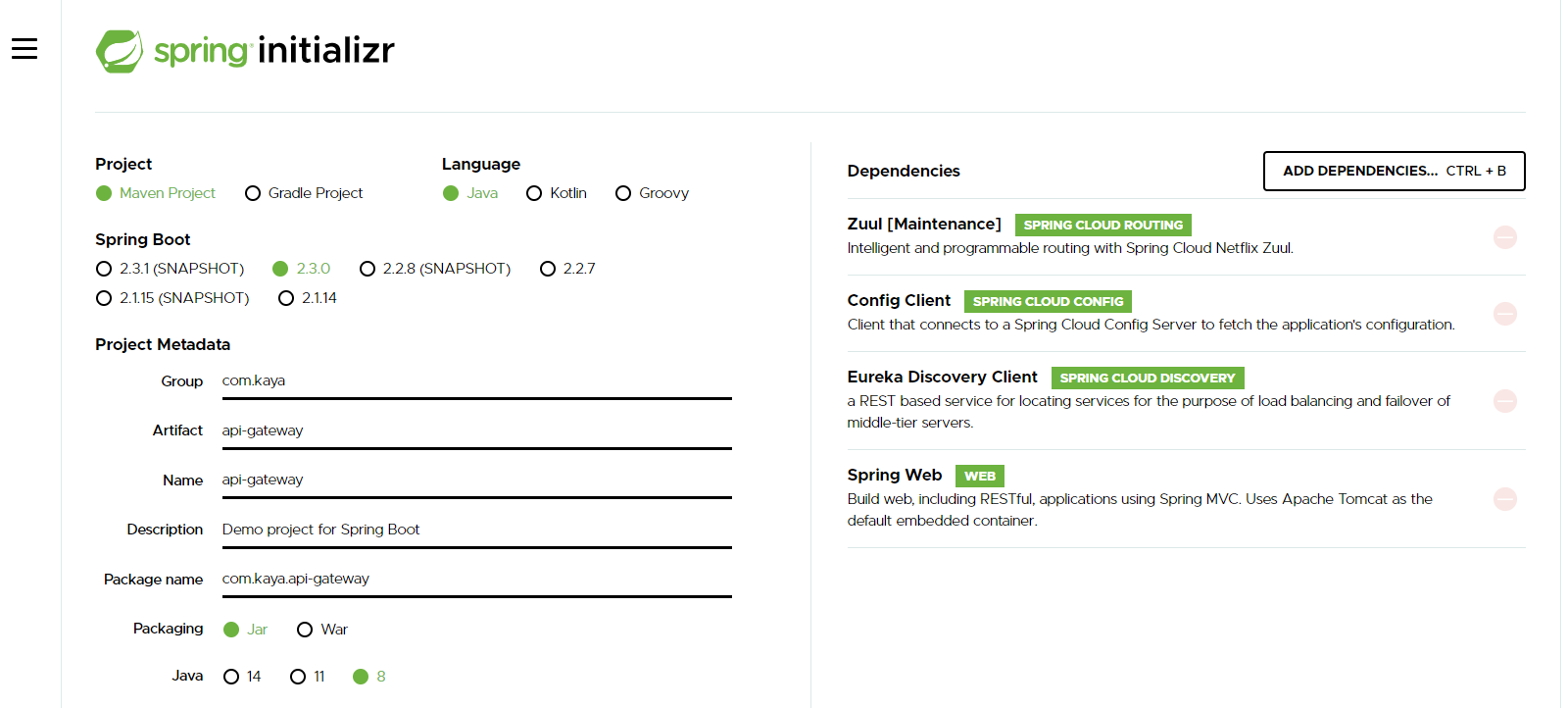
}

]

}

1. **API Gateway**

Spring initializr ile projemizi oluşturuyoruz



**bootstrap.properties** ayarlarını ekliyoruz.

spring.application.name=api-gateway

server.port=5860

spring.cloud.config.uri=http://config-server:5859

spring.cloud.config.uri: Config Server’imizin adresi. Böylelikle configurationlarımızı Config Server’den fetch ediyor olacağız (config-server: docker servisimizin adı).

**ApiGatewayApplication.java** classımıza **@EnableEurekaClient** ve **@EnableZuulProxy** anotasyonlarımız ekliyoruz. Böylelikler projemiz Biz API gateway özelliği kazacanak ve yapılan istekleri Discovery Server’a kayıt yaptırmış servislere yönlendirebilecek.

@SpringBootApplication

@EnableEurekaClient

@EnableZuulProxy

public class ApiGatewayApplication {

public static void main(String[] args) {

SpringApplication.run(ApiGatewayApplication.class, args);

}

}

Dockerfile’ımı oluşturuyoruz

FROM openjdk:8-jdk-alpine

ARG JAR\_FILE=target/\*.jar

COPY ${JAR\_FILE} app.jar

ENTRYPOINT ["java","-jar","/app.jar"]

Projenin dizinine gidip projemizi build ediyoruz.

mvnw package

docker-compose.yml dosyamıza ekliyoruz

…

api-gateway:

build:

context: ./api-gateway

dockerfile: Dockerfile

image: api-gateway-docker

depends\_on:

- discovery-server

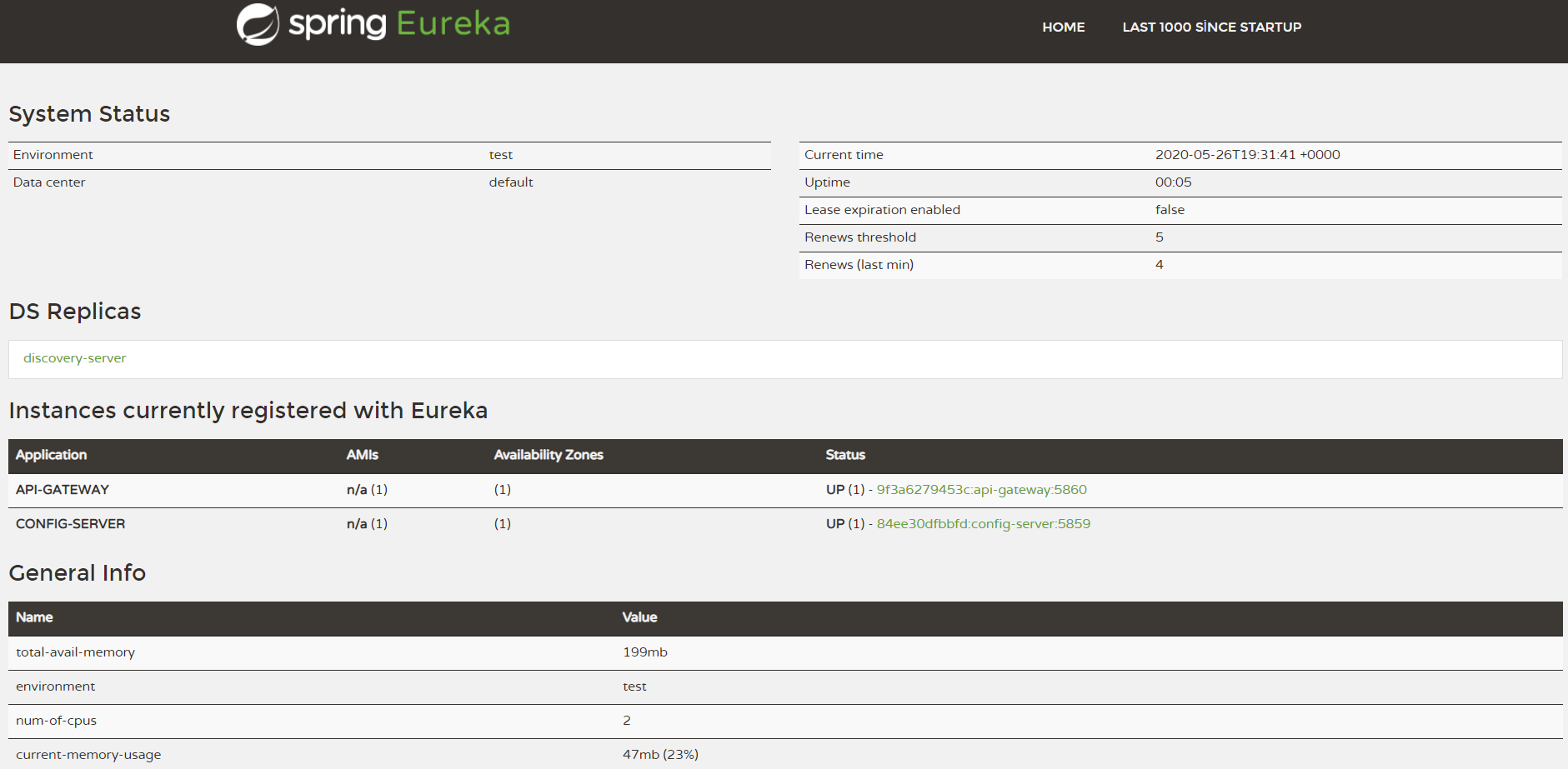
- config-server

…

Projemizi çalıştırıyoruz

docker-compose up –d api-gateway

Discovery Server’i kontrol ederek projenin ayağa kalktığını anlayabiliriz.



Artık isteklerimizi API Gateway üzerinden yönlendireceğiz. Daha önce test amacıyla Config Server’a istek yollamıştık. Şimdi aynı isteği gateway üzerinden deneyelim.

<http://localhost:5860/config-server/product-service/default>

Response:

