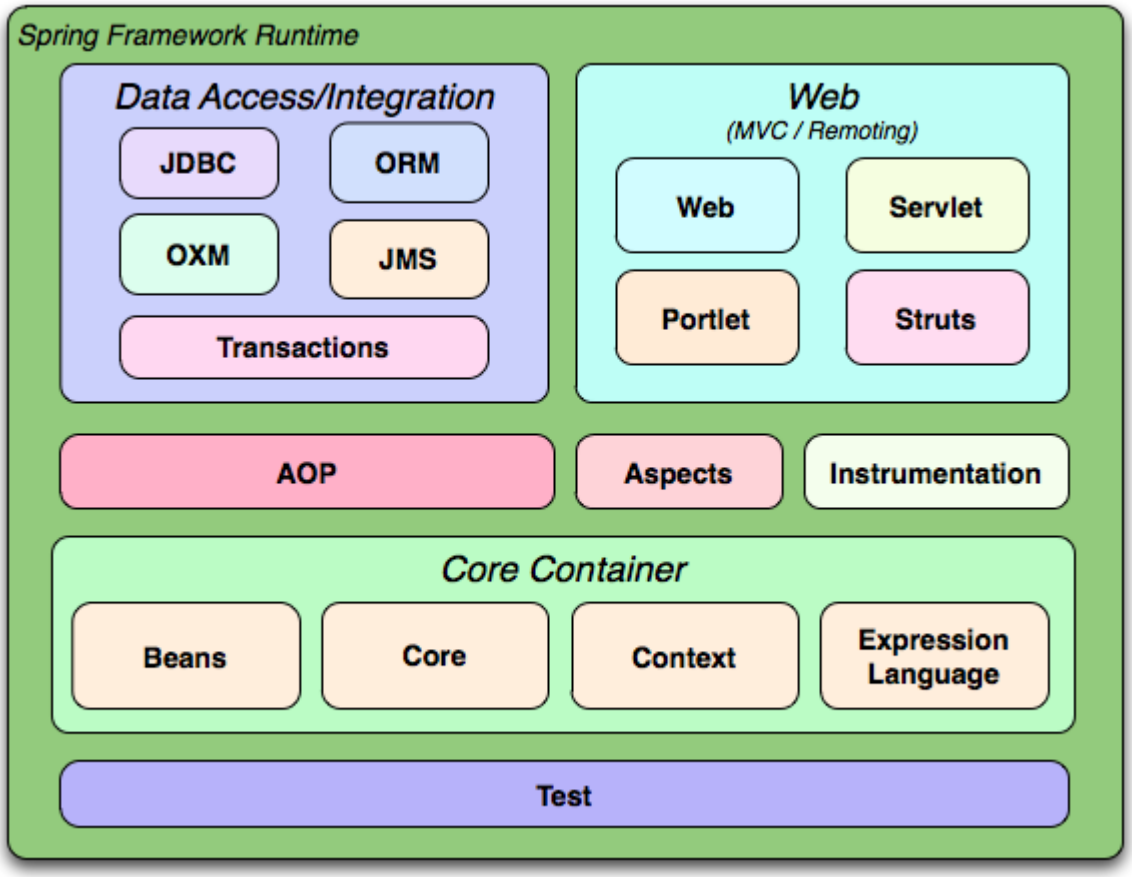
Francisco das Chagas Torres dos Santos

Setembro de 2022

Frameworks Java

Spring Framework



Documentação: https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/overview.html

Spring Framework

É é uma **plataforma Java** que fornece suporte abrangente de infraestrutura para o desenvolvimento de aplicativos Java.

- Lida com a infraestrutura para que você possa se concentrar em seu aplicativo.
- Permite **criar aplicativos a partir de POJOs** e **aplique serviços corporativos** de forma não invasiva a POJOs. Esse recurso se aplica ao modelo de programação **Java SE** e ao **Java EE** total e parcial.

Injeção de Dependência e Inversão de Controle

Inversão de Controle (IoC) – padrão de projeto em que o objeto apenas declara suas dependências sem criá-las e delega a tarefa de construir tais dependências a um Container IoC (Core Containers).

Injeção de Dependência – é a implementação utiliza pelo Spring Framework para aplicar o IoC quando necessário.

Bean – é um objeto que é instanciado, montado e gerenciado por um container do Spring através do IoC e Injeção de Dependência.

POJO - Plain Old Java Objects -

Módulos do Spring

Os módulos são agrupados em:

- Core Container
- Data Access/Integration
- Web
- AOP (Aspect Oriented Programming)
- Instrumentação
- Teste

■ Core Container

Formado pelo **Core**, **Beans**, **Context** e **Expression Language**

Core e **Beans** fornecem as partes fundamentais do framework, incluindo os **recursos IoC e Dependency Injection**.

O **BeanFactory** é uma implementação sofisticada do padrão Factory. Ele elimina a necessidade de programação Singletons e permite **desacoplar a configuração e a especificação de dependência**s da lógica real do programa.

Context é um meio de acessar objetos em uma maneira de estrutura semelhante a um registro JNDI. **Herda recursos do Beans e adiciona** suporte para internacionalização (usando, por exemplo, pacotes de recursos), propagação de eventos, carregamento de recursos e criação transparente de contextos por,

por exemplo, um contêiner de servlet. Também suporta recursos Java EE, como EJB, JMX e comunicação remota básica. A interface ApplicationContext é o ponto focal do módulo Context.
Expression Language fornece uma linguagem de expressão poderosa para consultar e manipular um gráfico de objeto em tempo de execução. É uma extensão da unified EL conforme especificado na JSP 2.1. A linguagem suporta a configuração e obtenção de valores de propriedade, atribuição de propriedade, invocação de método, acesso ao contexto de arrays, coleções e indexadores, operadores lógicos e aritméticos, variáveis nomeadas e recuperação de objetos por nome do contêiner IoC do Spring. Ele também suporta projeção e seleção de listas, bem como agregações de listas comuns.

■ Data Acess/Integration

Formado pelos módulos JDBC, ORM, OXM, JMS e Transations.

JDBC fornece uma camada de abstração JDBC que elimina a necessidade de codificação JDBC.

ORM fornece camadas de integração para APIs populares de mapeamento objeto-relacional, incluindo JPA, JDO, Hibernate e iBatis.

OXM fornece uma camada de abstração que suporta implementações de mapeamento Object/XML para JAXB, Castor, XMLBeans, JiBX e XStream.

JMS - Java Messaging Service contém recursos para produzir e consumir mensagens.

Transaction suporta o gerenciamento de transações programáticas e declarativas para classes que implementam interfaces especiais e para todos os seus POJOs.

■ Web

Formada pelos módulos Web, Web-Servlet, Web-Struts e Web-Portlet.

Web fornece recursos básicos de integração orientados à web, como a funcionalidade de upload de arquivos em várias partes e a inicialização do contêiner IoC usando ouvintes de servlet e um contexto de aplicativo orientado à web. Ele também contém as partes relacionadas à web do suporte remoto do Spring.

Web-Servlet contém a implementação do modelo MVC (Model-View-Controller) do Spring para aplicativos da web.

Web-Struts contém as classes de suporte para integrar uma camada da web clássica do Struts em um aplicativo Spring. Observe que esse suporte agora está obsoleto a partir do Spring 3.0. Considere migrar sua aplicação para o Struts 2.0 e Spring ou para Spring MVC.

Web-Portlet fornece a implementação MVC para ser usada em um ambiente de portlet e espelha a funcionalidade do módulo Web-Servlet.

■ AOP and Instrumentation

AOP do Spring fornece uma implementação de programação orientada a aspectos compatível com AOP Alliance, permitindo que você defina, por exemplo, interceptadores de método e pontos de corte para desacoplar de maneira limpa o código que implementa a funcionalidade que deve ser separada. Usando a funcionalidade de metadados de nível de origem, você também pode incorporar informações comportamentais em seu código, de maneira semelhante à dos atributos .NET.

Aspects separado fornece integração com o **AspectJ**.

Instrumentation fornece suporte a instrumentação de classe e implementações de carregador de classe a serem usadas em determinados servidores de aplicativos.

Messaging

■ Test

Suporta o teste de componentes Spring com **JUnit** ou **TestNG**. Fornece carregamento consistente de Spring ApplicationContexts e armazenamento em cache desses contextos. Ele também fornece objetos simulados que você pode usar para testar seu código isoladamente.

Gerenciamento de Dependência ≠ Injeção de Dependência

Para obter os recursos Spring como IoC, é preciso montar todas as bibliotecas necessárias.

A dependências podem ser:

- **Diretas**: minha aplicação depende do Spring em tempo de execução
- **Indiretas/Transitiva**: minha aplicação depende de commons-dbcp que depende de commons-pool. São mais difíceis de identificar e gerenciar.

O Spring é empacotado como um conjunto de módulos que separam as dependências o máximo possível. Ex.: para um aplicativo da Web tem o módulo spring-web.

Para fazer referência aos módulos da biblioteca Spring, usamos spring-* ou spring-*.jar, onde "*" representa o nome abreviado do módulo (Ex.: exemplo spring-core, spring-webmvc, spring-jms, etc.). Normalmente tem um número, como spring-core-3.0.0.RELEASE.jar.

Geralmente, utiliza-se sistemas automatizados para gerenciar as dependências como o Maven ou Ivy. Mas é posśivel fazer manualmente, baixando todos os jars. É importante saber de onde (repositório) baixar as dependências, não misturar.

Repositórios:

- **Maven Central**
- **SpringSource EBR**

Gerenciamento de dependências Maven

```
<dependencies>
  <dependency>
     <groupId> org.springframework </groupId>
     <artifactId> spring-context </artifactId>
     <version> 3.0.0.RELEASE </version>
     <scope> runtime </scope>
   </dependency>
</dependencies>
```

Um pouco mais de Spring. Caso veja questões em algum concurso ou tenha tempo para estudar, visitar o site para obter as informações sobre:

- **Ivy Dependency Management**
- **Logging**: commons-logging
- **SLF4J**: log4j.properties ou log4j.xml

Saindo um pouco da documentação:

Esteriótipos do Spring

O Spring se baseia no princípio de esteriótipos, que é dividir de acordo com as responsabilidades.

Exemplos de esteriótipos do Spring:

- @Component
- @Service
- @Repository
- @Controller

Spring Boot

Spring Boot = Spring Framework + Servidor embutido – XML<bean> configuration ou @Configuration

O servidor embutido:

- **Tomcant** no Spring MVC
- **Netty** no Spring Cloud

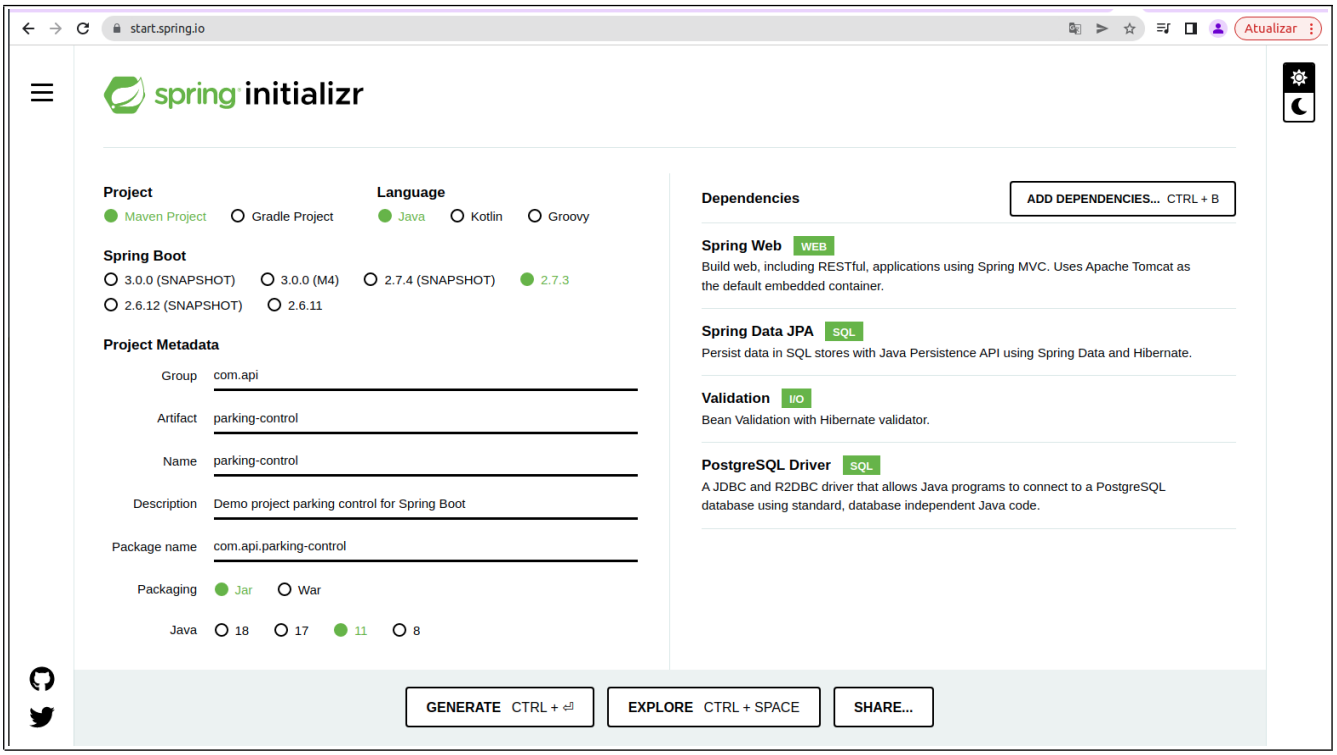
Spring é uma extensão do framework Spring que elimina as configurações padrão necessárias para configurar uma aplicação Spring.

Criando um projeto no Spring Boot

Como exemplo, vamos imaginar uma aplicação de um estacionamento, o Parking Control API Project com Spring conforme abaixo:

- Spring Boot – para iniciar a aplicação
- Spring MVC – construira aplicação web
- Spring Data JPA – para a persistência
- Spring Validation – para validações iniciais

No caso, seria necessário acessar o **Spring Initializr** no site start.spring.io e criar o projeto. A imagem abaixo mostrar como é simples criar um projeto Spring Boot e adicionar as dependências necessárias. Observe que além das dependências do Spring foi adicionado o JDBC do PostgreSQL.



Depois disso, é só fazer o download, extrair e importar para a IDE Java desejada. Na IDE IntelliJ, essas configurações/dependências ficam em pom.xml.

Se olhar na IDE, em External Libraries tem várias dependências que o Maven baixou, estas são dependência das dependências, como: Spring Bean, …

A classe que vem pronta e fica na raiz tem a anotação @SpringBootApplication.

Alguns dos recursos do Spring Boot:

- Dependências “iniciais” para simplificar a configuração da compilação e do aplicativo
- Servidor incorporado para evitar complexidade na implantação de aplicativos
- Métricas, verificação de integridade e configuração externa
- Configuração automática para funcionalidade Spring – sempre que possível

application.properties

Arquivo onde ficam as definições da base de dados:

1. spring.datasource.url = jdbc:postgresql://localhost:5432/parking-control-db
2. spring.datasource.username = postgres
3. spring.datasource.password = banco123
4. spring.jpa.hibernate.ddl-auto = update
5. spring.jpa.properties.hibernate.jdbc.lob.non_contextual_creation = true

A linha 4 define o mapeamento JPA automaticamente

Configuração JPA em https://www.thomasvitale.com/spring-data-jpa-hibernate-java-configuration/

Ao acessar o http:localhost:8080 não parece nada.

Algumas anotações:

- **@SpringBootApplication** – diz que esta classe é o ponto de entrada de uma aplicação Spring Boot
- **@RestController** – define a classe como Bean do tipo Controller
- **@GetMapping(“/”)** - mapea uma URL a um método

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

```
@GetMapping(“/”)
public String index(){
	return “Olá, mundo!”;
}
```

Algumas observações:

- Ao estender o JpaRepository não é necessário anotar explicitamente com @Repository, pois está anotação já está no JpaRepository.
- O Controller acionar o Service e este aciona o Repository
- Dentro do Service adiciona o ponto de injeção, ou seja, coloca o Repository dentro do Service.

```
-- Packingcontrol
	-- ParkingControlApplication
	-- Controllers
	-- Repositories
	-- Models
	-- Services
```

Spring x Spring Boot

Para mais detalhes: https://www.baeldung.com/spring-vs-spring-boot

Dependências do Maven

Dependências mínimas necessárias para criar uma aplicação web usando Spring:

- spring-web de org.springframework
- spring-webmvc de org.springframework

Dependencia Spring Boot para colocar uma palicação web em funcionamento:

- sprig-boot-starter-web de org.springframework.boot

O Spring Boot fornece várias dependências iniciais para diferentes módulos Spring.

Alguns dos mais usados são:

- spring-boot-starter-data-jpa
- spring-boot-starter-security
- spring-boot-starter-test
- spring-boot-starter-web
- spring-boot-starter-thymeleaf

Configuração MVC

Configuração necessária para criar um aplicativo Web JSP usando Spring e Spring Boot.

O Spring requer a definição do dispatcher servlet, mapeamentos e outras configurações de suporte. Podemos fazer isso usando o arquivo **web.xml** ou uma classe **Initializer. Também precisa adiconar @EnableWebMvc a uma classe @Configuration e definri um view-resolver para resolver as views retornadas dos controladores.**

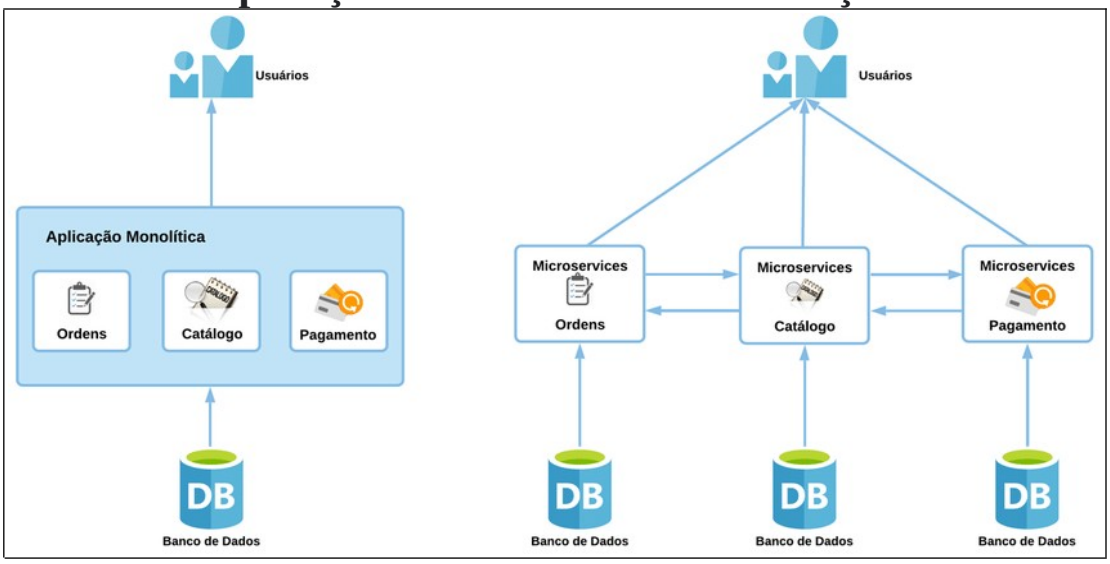
```
@EnableWebMvc
@Configuration
public class ClientWebConfig implements WebMvcConfigurer {
    @Bean
    public ViewResolver viewResolver() {
        InternalResourceViewResolver bean = new InternalResourceViewResolver();
        bean.setViewClass(JstlView.class);
        bean.setPrefix("/WEB-INF/view/");
        bean.setSuffix(".jsp");
        return bean;
    }
```

O Spring Boot precisa apenas de algumas propriedades para fazer as coisas funcionarem quando adicionamos web:

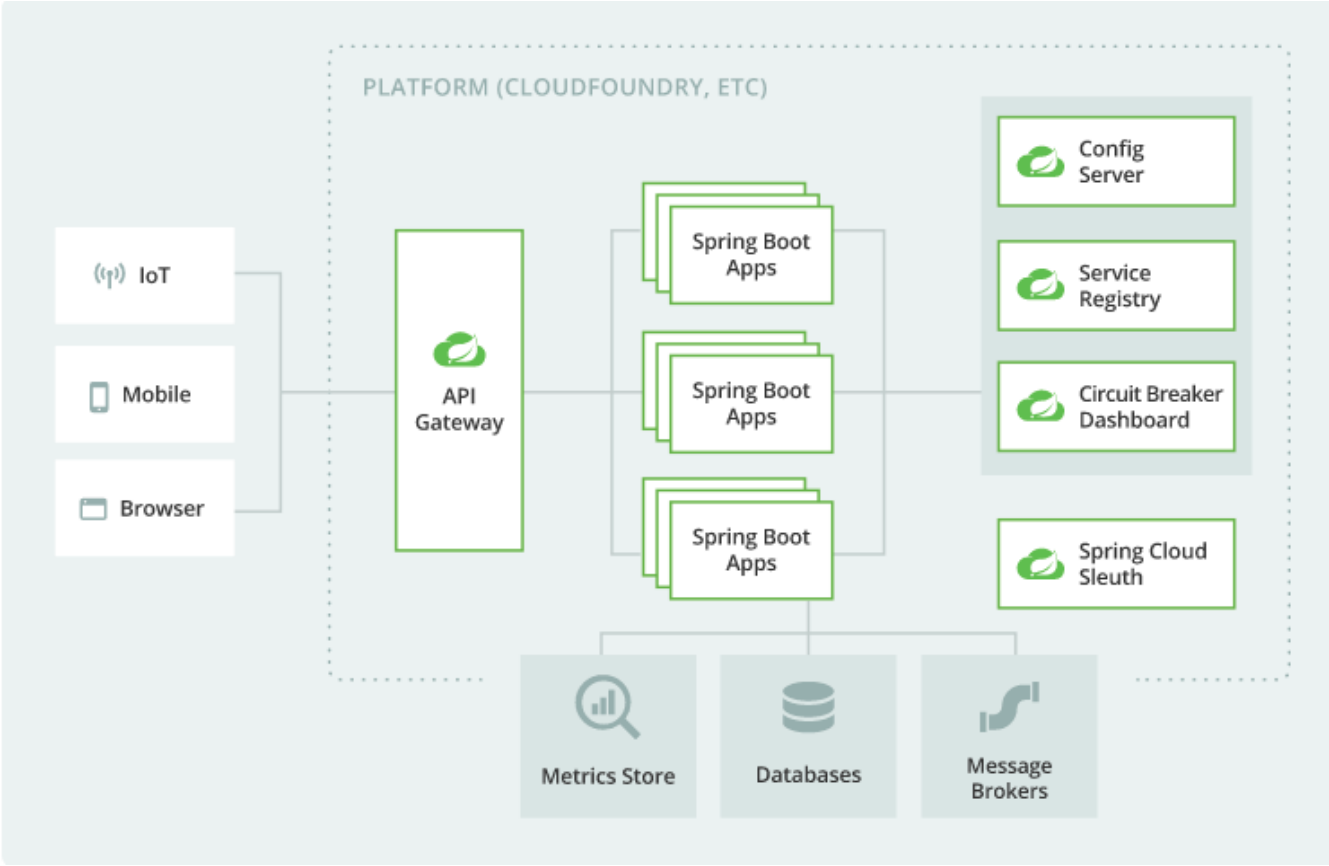
- spring.mvc.view.prefix=/WEB-INF/jsp/
- spring.mvc.view.suffix=.jsp

Toda a configuração do Spring acima é incluída automaticamente ao adicionar o Boot Web de inicialização por meio de um processo chamado auto-configuration. Assim, o Spring Boot examinará as dependências, propriedades e beans que existem no aplicativo e habilitará a configuração com base neles. Se quisermos adicionar nossa configuração personalizada, a configuração automática do Spring Boot será retirada.

Aplicação Monolítica x Microserviços



Microserviços com Spring Cloud



API GatewaySpring Cloud

Trás implementado nos melhores padrões de projetos os serviços para implementar/utilizar microserviços.

API Gateway

- Roteamento e filtros
- Integra balanceamento de carga (Spring Cloud Load Balancer)
- Integra serviços de descoberta
- Segurança/autenticação
- Implementações:
  - **Spring Cloud Gateway**
    - Criado no Spring Framework 5, Project Reactor e Spring Boot 2.0. ou seja, permite trabalhar reativo e não bloqueante. API Web Flux trabalha de forma reativa.
    - Utiliza o Netty Server

Service Registry/Discovery

Implementações:

- Spring Cloud Netflix Eureka
- Spring Cloud Consul

O microserviço precisa saber o URI e a porta para se comunicar.
Todos os serviços que vão se comunicar precisam se registrar. O Eureka Server monitora as intâncias, URI dinâmica e as portas.
O cliente se registra no Eurelka Server e passa a ser um Eureka Client, após descobrir o microserviço, pode fazer a requisição de outro Eureka Client.

Circuit Break

Adiciona e impletmentado em chamada de métodos com alto potencial de latência a falhas.

- Mitiga falhas em cascata
- Métodos fallback durante falhas.
- Hystrix era a implementação padrão do Circuit Break (netflix Hystrix)

Spring Cloud Circuit Break:

- Netflix Hystrix – parou de receber atualizações
- Resilience4J
- Sentinel
- Circuit Breaker (TAS)

Config Server

Serviço que centraliza as configurações dos microserviços
Usa o Git para armazenar as configurações

Spring Cloud Config:

- Spring Cloud Config Server
- Spring Cloud Config Client

Spring Cloud Sleuth

Rastreamento (Tracking distribuído) – ver requisições de um client a um service A que faz requisição a um service B, que faz a um service C.

Adiciona identificadores em cada log:

- **TraceId**: único em todas as chamadas
- **SpanId**: individual para cada chamada

Zipkin → ferramenta para rastrear.

Para mais detalhes: https://dev.to/adevintaspain/spring-cloud-sleuth-in-action-10k2

Spting Cloud Stream

Messagerias – Processos assíncronos e orientados a eventos

Para mais detalhes: https://docs.spring.io/spring-cloud-stream-binder-rabbit/docs/current/reference/html/spring-cloud-stream-binder-rabbit.html

Para mais detalhes de Microserviços: https://spring.io/microservices

Zuul e Eureka

https://emmanuelneri.com.br/2018/05/02/criando-proxy-de-apis-com-spring-cloud-zuul-e-eureka/

Anotações Spring

Anotações Spring Boot

Para mais detalhes: https://www.baeldung.com/spring-boot-annotations
São as anotações dos pacotes:

- org.springframework.boot.autoconfigure
- org.springframework.boot.autoconfigure.condition

@SpringbootApplication – marcar a classe principal da aplicação Spring Boot. Encapsula as anotações @Configuration , @EnableAutoConfiguration e @ComponentScan com seus atributos padrão.

@EnableAutoConfiguration – habilita a configuração automática. Isso significa que o Spring Boot procura beans de configuração automática em seu caminho de classe e os aplica automaticamente. Usada com @Configuration.

```
@Configuration
@EnableAutoConfiguration
class VehicleFactoryConfig {}
```

@ConditionalOnClass e @ConditionalOnMissingClass – usando essas condições, o Spring usará apenas o bean de configuração automática marcado se a classe no argumento da anotação estiver presente/ausente.

```
@Configuration
@ConditionalOnClass(DataSource.class)
class MySQLAutoconfiguration { //... }
```

@ConditionalOnBean e @ConditionalOnMissingBean – para definir condições com base na presença ou ausência de um bean específico.

```
@Bean
@ConditionalOnBean(name = "dataSource")
LocalContainerEntityManagerFactoryBean entityManagerFactory() { // ... }
```

@ConditionalOnProperty – para fazer condições sobre os valores das propriedades

```
@Bean
@ConditionalOnProperty(
        name = "usemysql",
        havingValue = "local"
)
```

```
DataSource dataSource() { // ... }
```

@ConditionalOnResource – faz o Spring usar uma definição apenas quando um recurso específico estiver presente.

```
@ConditionalOnResource(resources = "classpath:mysql.properties")
Properties additionalProperties() { // ... }
```

@ConditionalOnWebApplication e @ConditionalOnNotWebApplication – para criar condições com base em se o aplicativo atual é ou não um aplicativo da web.

@CondicionalExpression – usada em situações mais complexas. O Spring usará a definição marcada quando a expressão **SpEL** for avaliada como true.

```
@Bean
@ConditionalOnExpression("${usemysql} && ${mysqlserver == 'local'}")
DataSource dataSource() { // ... }
```

@Condicional – para condições ainda mais complexas, podemos criar uma classe avaliando a condição personalizada.

```
@Conditional(HibernateCondition.class)
Properties additionalProperties() { //... }
```

Anotações Spring Web

Para mais detalhes: https://www.baeldung.com/spring-mvc-annotations
Pacote: org.springframework.web.bind.annotation

@RequestMapping – marca os métodos do manipulador de solicitação dentro das classes @Controller. Pode ser configurado usando: path, method, params, geaders, consumes ou produces.

Variantes do @RequestMapping o método HTTP: @GetMapping , @PostMapping , @PutMapping , @DeleteMapping e @PatchMapping. Disponíveis desde o lançamento do Spring 4.3.

```
@Controller
class VehicleController {
    @RequestMapping(value = "/vehicles/home", method = RequestMethod.GET)
    String home() {
        return "home";
    }
}
```

```
@Controller
@RequestMapping(value = "/vehicles", method = RequestMethod.GET)
class VehicleController {
   @RequestMapping("/home")
```

```
    String home() {
        return "home";
```

@RequestBody – que mapeia o corpo da requisição HTTP para um objeto. A desserialização é automática e depende do tipo de conteúdo da solicitação.

```
@PostMapping("/save")
void saveVehicle(@RequestBody Vehicle vehicle) { // ... }
```

@PathVariable – indica que um argumento de método está vinculado a uma variável de URI . Podemos especificar o template de URI com a anotação @RequestMapping e vincular um argumento de método a uma das partes do modelo com @PathVariable .

```
@RequestMapping("/{id}")
Vehicle getVehicle(@PathVariable("id") long id) { // ... }
```

Se o nome da parte no modelo corresponder ao nome do argumento do método, não precisamos especificá-lo na anotação:

```
@RequestMapping("/{id}")
Vehicle getVehicle(@PathVariable long id) { // ... }
```

Além disso, podemos marcar uma variável de caminho opcional definindo o argumento obrigatório como false:

```
@RequestMapping("/{id}")
Vehicle getVehicle(@PathVariable(required = false) long id) { // ... }
```

@RequestParam – para acessar os parâmetros de solicitação HTTP. Tem as mesmas opções de configuração que a anotação @PathVariable.

```
@RequestMapping
Vehicle getVehicleByParam(@RequestParam("id") long id) { // ... }
```

Além dessas configurações, podemos especificar um valor injetado quando o Spring não encontrar nenhum valor ou valor vazio na solicitação com o argumento defaultValue.

```
@RequestMapping("/buy")
Car buyCar(@RequestParam(defaultValue = "5") int seatCount) { // ... }
```

Podemos acessar cookies e cabeçalhos da solicitação HTTP com **@CookieValue** e **@RequestHeader**.

Anotações de Tratamento de Respostas

Anotações mais comuns para manipular respostas HTTP no Spring MVC.

@ResponseBody – o Spring tratará o resultado do método como a própria resposta :

```
@ResponseBody
@RequestMapping("/hello")
String hello() {
    return "Hello World!";
}
```

Se anotarmos uma classe @Controller com essa anotação, todos os métodos do manipulador de solicitações a usarão.

@ExceptionHandler – declarar um método de tratamento de erros personalizado . O Spring chama esse método quando um método de manipulador de solicitação lança qualquer uma das exceções especificadas. A exceção capturada pode ser passada para o método como um argumento.

```
@ExceptionHandler(IllegalArgumentException.class)
void onIllegalArgumentException(IllegalArgumentException exception) { // ... }
```

@ResponseStatus – especificar o status HTTP desejado da resposta. Pode ser usada com @ExceptionHandler.

```
@ExceptionHandler(IllegalArgumentException.class)
@ResponseStatus(HttpStatus.BAD_REQUEST)
void onIllegalArgumentException(IllegalArgumentException exception) { // ... }
```

Outras Anotações Web

Algumas anotações não gerenciam solicitações ou respostas HTTP diretamente.

@Controlador – define um controlador Spring MVC com @Controller. Spring Bean Annotations

@RestController – combina @Controller e @ResponseBody. As declarações abaixo são equivalentes.

```
@Controller
@ResponseBody
class VehicleRestController { // ... }
```

```
@RestController
class VehicleRestController { // ... }
```

@ModelAttribute – acessar elementos que já estão no modelo de um MVC @Controller, fornecendo a chave do modelo.

```
@PostMapping("/assemble")
void assembleVehicle(@ModelAttribute("vehicle") Vehicle vehicleInModel) { // ... }
```

Assim como @PathVariable e @RequestParam , não precisamos especificar a chave do modelo se o argumento tiver o mesmo nome.

```
@PostMapping("/assemble")
void assembleVehicle(@ModelAttribute Vehicle vehicle) { // ... }
```

Além disso, @ModelAttribute tem outro uso: se anotarmos um método com ele, o Spring adicionará automaticamente o valor de retorno do método ao modelo.

```
@ModelAttribute("vehicle")
Vehicle getVehicle() { // ... }
```

Como antes, não precisamos especificar a chave do modelo, o Spring usa o nome do método por padrão:

```
@ModelAttribute
Vehicle vehicle() { // ... }
```

Antes que o Spring chame um método de manipulador de solicitação, ele invoca todos os métodos anotados @ModelAttribute na classe.

@CrossOrigin – habilita a comunicação entre domínios para os métodos de manipulador de solicitação anotados. Se marcarmos uma classe com ela, ela se aplicará a todos os métodos do manipulador de solicitações nela.

```
@CrossOrigin
@RequestMapping("/hello")
String hello() {
        return "Hello World!";
}
```

Anotações do Spring Core

Pacotes:

- org.springframework.beans.factory.annotation
- org.springframework.context.annotation packages.

Anotações relacionadas a Injeção de Dependência

@Autowired – marca uma dependência que o Spring vai resolver e injetar. Podemos usar essa anotação com um construtor, setter ou injeção de campo.

```
class Car {
    Engine engine;
    @Autowired
    Car(Engine engine) {
        this.engine = engine;
    }
}
```

```
class Car {
    Engine engine;
    @Autowired
    void setEngine(Engine engine) {
        this.engine = engine;
    }
}
```

```
class Car {
    @Autowired
    Engine engine;
}
```

@Autowired tem um argumento booleano chamado **required** com um valor padrão de **true** . Ele ajusta o comportamento do Spring quando ele não encontra um bean adequado para conectar. Quando true, uma exceção é lançada, caso contrário, nada é conectado. Se usarmos injeção de construtor, todos os argumentos do construtor são obrigatórios.

A partir da versão 4.3, não precisamos anotar construtores com @Autowired explicitamente, a menos que declaremos pelo menos dois construtores.

@Bean – marca um método de fábrica que instancia um Spring bean

```
@Bean
Engine engine() {
    return new Engine();
}
```

O Spring chama esses métodos quando uma nova instância do tipo de retorno é necessária. O bean resultante tem o mesmo nome que o método de fábrica. Se quisermos nomeá-lo de forma diferente, podemos fazê-lo com os argumentos name ou value (alias para name) desta anotação.

```
@Bean("engine")
Engine getEngine() {
    return new Engine();
}
```

Todos os métodos anotados com @Bean devem estar nas classes @Configuration.

@Qualificador – usado com @Autowired para fornecer o id do bean ou o nome do bean que queremos usar em situações ambíguas. Por exemplo, os dois beans a seguir implementam a mesma interface:

```
class Bike implements Vehicle {}

class Car implements Vehicle {}
```

Se o Spring precisar injetar um bean Vehicle , ele terminará com várias definições correspondentes. Nesses casos, podemos fornecer o nome de um bean explicitamente usando a anotação @Qualifier .

```
@Autowired
Biker(@Qualifier("bike") Vehicle vehicle) {  // constructor injection
        this.vehicle = vehicle;
}
```

```
@Autowired
void setVehicle(@Qualifier("bike") Vehicle vehicle) { // setter injection
        this.vehicle = vehicle;
}
```

```
@Autowired
@Qualifier("bike")
void setVehicle(Vehicle vehicle) { // setter injection, outra maneira
       this.vehicle = vehicle;
}
```

```
@Autowired
@Qualifier("bike")
 Vehicle vehicle; // field injection
```

@Required – em métodos setter para marcar dependências que queremos preencher por meio de XML. Caso contrário, BeanInitializationException será lançada.

```
@Required
void setColor(String color) {
     this.color = color;
}
```

```
<bean class="com.baeldung.annotations.Bike">
      <property name="color" value="green" />
</bean>
```

@Value – para injetar valores de propriedade em beans. É compatível com construtor, setter e injeção de campo.

```
Engine(@Value("8") int cylinderCount) {
      this.cylinderCount = cylinderCount;
}
```

```
@Autowired
void setCylinderCount(@Value("8") int cylinderCount) {
      this.cylinderCount = cylinderCount;
}
```

```
@Value("8")
void setCylinderCount(int cylinderCount) {
      this.cylinderCount = cylinderCount;
}
```

```
@Value("8")
int cylinderCount;
```

Injetar valores estáticos não é útil. Podemos usar strings de marcador em @Value para conectar valores definidos em fontes externas, por exemplo, em arquivos .properties ou .yaml. Supondo o arquivo .properties.

```
engine.fuelType=petrol
```

Podemos injetar o valor de engine.fuelType com o seguinte. Podemos usar @Value mesmo com SpEL.

```
@Value("${engine.fuelType}")
String fuelType;
```

@DependsOn – faz o Spring inicializar outros beans antes do anotado. Normalmente, esse comportamento é automático, baseado nas dependências explícitas entre beans. Só precisamos dessa anotação quando as dependências estão implícitas , por exemplo, carregamento de driver JDBC ou inicialização de variável estática. Podemos usar @DependsOn na classe dependente especificando os nomes dos beans de dependência. O argumento value da anotação precisa de um array contendo os nomes dos bean de dependência

```
@DependsOn("engine")
class Car implements Vehicle {}
```

Alternativamente, se definirmos um bean com a anotação @Bean , o factory método deve ser anotado com @DependsOn:

```
@Bean
@DependsOn("fuel")
Engine engine() {
	return new Engine();
}
```

@Lazy – inicializar nosso bean lentamente. Por padrão, o Spring cria todos os beans singleton avidamente na inicialização/bootstrapping do contexto do aplicativo. No entanto, há casos em que precisamos criar um bean quando o solicitamos, não na inicialização do aplicativo .

Essa anotação se comporta de maneira diferente dependendo de onde exatamente a colocamos.

Podemos colocar:

- Um @Bean em um factory method para atrasar a chamada do método (daí a criação do bean).
- Uma classe @Configuration e todos os métodos @Bean contidos serão afetados.
- Uma classe @Component, que não é uma classe @Configuration, este bean será inicializado lentamente.
- Um construtor, setter ou campo @Autowired para carregar a própria dependência preguiçosamente (via proxy).

Esta anotação tem um argumento chamado value com o valor padrão de true. É útil substituir o comportamento padrão.

Por exemplo, marcar beans para serem carregados antecipadamente quando a configuração global for lenta ou configurar métodos @Bean específicos para carregamento antecipado em uma classe @Configuration marcada com @Lazy.

```
@Configuration
@Lazy
class VehicleFactoryConfig {
	@Bean
	@Lazy
	Engine engine() {
		return new Engine();
	}
}
```

@Lookup – diz ao Spring para retornar uma instância do tipo de retorno do método quando o invocamos.

@Primary – Ao definir **vários beans do mesmo tipo**, a injeção não será bem-sucedida porque o Spring não tem ideia de qual bean precisamos. Podemos marcar todos os pontos de fiação com @Qualifier e especificar o nome do bean necessário. No entanto, na maioria das vezes precisamos de um bean específico e raramente dos outros. Podemos usar @Primary para simplificar este caso: se **marcarmos o bean usado com mais frequência** com @Primary ele será escolhido em pontos de injeção não qualificados.

```
@Component
@Primary
class Car implements Vehicle {}

@Component
class Bike implements Vehicle {}

@Component
class Driver {
        @Autowired
        Vehicle vehicle;
}

@Component
class Biker {
        @Autowired
        @Qualifier("bike")
        Vehicle vehicle;
}
```

No exemplo anterior, o carro é o veículo principal. Portanto, na classe Driver, o Spring injeta um bean Car. É claro que, no bean Biker, o valor do veículo de campo será um objeto Bike porque ele é qualificado.

@Scope – definir o escopo de uma classe @Component ou uma definição de @Bean. Pode ser singleton, prototype, request, session, globalSession ou algum escopo personalizado.

```
@Component
@Scope("prototype")
class Engine {}
```

Anotações de configuração de contexto

@Profile – se quisermos que o Spring use uma classe @Component ou um método @Bean apenas quando um perfil específico estiver ativo. Podemos configurar o nome do perfil com o argumento value da anotação.

```
@Component
@Profile("sportDay")
class Bike implements Vehicle {}
```

@Import – usar classes @Configuration específicas sem varredura de componentes

```
@Import(VehiclePartSupplier.class)
class VehicleFactoryConfig {}
```

@ImportResource – importar configurações XML. Podemos especificar as localizações dos arquivos XML com o argumento location , ou com seu alias, o argumento value :

```
@Configuration
@ImportResource("classpath:/annotations.xml")
class VehicleFactoryConfig {}
```

@PropertySource – definir arquivos de propriedades para configurações do aplicativo.

```
@Configuration
@PropertySource("classpath:/annotations.properties")
class VehicleFactoryConfig {}
```

@PropertySource aproveita o recurso de anotações repetidas do Java 8, o que significa que podemos marcar uma classe com ele várias vezes.

```
@Configuration
@PropertySource("classpath:/annotations.properties")
@PropertySource("classpath:/vehicle-factory.properties")
class VehicleFactoryConfig {}
```

@PropertySources – especificar várias configurações @PropertySource :

```
@Configuration
@PropertySources({
        @PropertySource("classpath:/annotations.properties"),
        @PropertySource("classpath:/vehicle-factory.properties")
})
class VehicleFactoryConfig {}
```

Observe que, desde o Java 8, podemos obter o mesmo com o recurso de anotações repetidas, conforme descrito acima.

Anotações Spring Bean

https://www.baeldung.com/spring-bean-annotations

Existem várias maneiras de configurar beans em um contêiner Spring. Em primeiro lugar, podemos declará-los usando a configuração XML. Também podemos declarar beans usando a anotação @Bean em uma classe de configuração. Finalmente, podemos marcar a classe com uma das anotações do pacote **org.springframework.stereotype** e deixar o resto para a varredura de componentes.

Componente Scanning

O Spring pode escanear automaticamente um pacote em busca de beans se o escaneamento de componentes estiver habilitado.

@ComponentScan configura quais pacotes procurar por classes com configuração de anotação . Podemos especificar os nomes dos pacotes base diretamente com um dos argumentos basePackages ou value (value é um alias para basePackages.

```
@Configuration
@ComponentScan(basePackages = "com.baeldung.annotations")
class VehicleFactoryConfig {}
```

Além disso, podemos apontar para classes nos pacotes base com o argumento basePackageClasses :

```
@Configuration
@ComponentScan(basePackageClasses = VehicleFactoryConfig.class)
class VehicleFactoryConfig {}
```

Ambos os argumentos são arrays para que possamos fornecer vários pacotes para cada um.

Se nenhum argumento for especificado, a verificação ocorrerá a partir do mesmo pacote em que a classe anotada @ComponentScan está presente.

@ComponentScan aproveita o recurso de anotações repetidas do Java 8, o que significa que podemos marcar uma classe com ele várias vezes:

```
@Configuration
@ComponentScan(basePackages = "com.baeldung.annotations")
@ComponentScan(basePackageClasses = VehicleFactoryConfig.class)
class VehicleFactoryConfig {}
```

Alternativamente, podemos usar @ComponentScans para especificar várias configurações de @ComponentScan.

```
@Configuration
@ComponentScans({
      @ComponentScan(basePackages = "com.baeldung.annotations"),
      @ComponentScan(basePackageClasses = VehicleFactoryConfig.class)
})
class VehicleFactoryConfig {}
```

Ao usar a configuração XML , a verificação do componente de configuração é igualmente fácil:

```
<context:component-scan base-package="com.baeldung" />
```

@Component – anotação de nível de classe. Durante a varredura do componente, o Spring Framework detecta automaticamente as classes anotadas com @Component.

Por padrão, as instâncias de bean desta classe têm o mesmo nome que o nome da classe com uma inicial minúscula. Além disso, podemos especificar um nome diferente usando o argumento de valor opcional desta anotação.

Como **@Repository , @Service , @Configuration e @Controller são todas meta-anotações de @Component**, eles compartilham o mesmo comportamento de nomenclatura de bean. O Spring também os coleta automaticamente durante o processo de digitalização do componente.

@Repository – As classes DAO ou Repository geralmente representam a camada de acesso ao banco de dados em um aplicativo.

Uma vantagem de usar esta anotação é que ela tem a tradução de exceção de persistência automática habilitada. Ao usar uma estrutura de persistência, como Hibernate, exceções nativas lançadas dentro de classes anotadas com @Repository serão automaticamente traduzidas em subclasses de DataAccessExeption do Spring .

Para habilitar a tradução de exceção, precisamos declarar nosso próprio bean PersistenceExceptionTranslationPostProcessor.

@Service – indica que uma classe pertence a a camada de seriço, onde a lógica de negócios de um aplicativo geralmente reside.

@Controller – é uma anotação de nível de classe, que informa ao Spring Framework que esta classe serve como um controlador no Spring MVC.

@Configuration – As classes de configuração podem conter métodos de definição de bean anotados com @Bean :

```
@Configuration
class VehicleFactoryConfig {
    @Bean
     Engine engine() {
        return new Engine();
     }
}
```

Anotações Spring Data

https://www.baeldung.com/spring-data-annotations

Parecidas com a do JPA

Eureka

É uma solução de **service discovery**, que em conjunto com outras ferramentas possibilita gerenciamento dinâmico e escalabilidade para as aplicações, o Eureka também faz pode ser encontrado no sub-projeto **spring-cloud-netflix**. É responsável por registar as aplicações através de apelidos, o que torna todo roteamento dinâmico.

A única dependência a ser adicionada para configurar o Eureka é a spring-cloud-netflix-eureka-server, que já carrega outras dependências necessárias para inicializar o Spring Boot no projeto (spring-boot-starter).

pom.xml

```
org.springframework.cloud
spring-cloud-netflix-eureka-server
```

Para tornar a aplicação um Eureka Server basta adicionar a anotação **@EnableEurekaServer** em conjunto com a anotação do Spring Boot (SpringBootApplication) que no startup da aplicação será inicializado o Eureka.

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

@SpringBootApplication
@EnableEurekaServer
public class EurekaAppConfig {

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

E por fim, algumas configurações precisam ser feitas nos arquivos de configuração do Spring Boot

application.properties

```
server.port=8761

eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
```

- **erver.port**: Define a porta que o Eureka vai estar disponível, por padrão o Eureka usa a porta 8761;
- **eureka.client.register-with-eureka**: Indica que a instância do Eureka não precisa ser registrada, até mesmo porque ela é a responsável pelos registros, assim não precisa registrar ela mesmo;
- **eureka.client.fetch-registry**: Indica que essa aplicação não precisa buscar informações de registro no Eureka, pelo mesmo motivo anterior.

As dependências adicionadas são: spring-boot-starter-web para fazer a disponibilização da **API Rest** e **spring-cloud-starter-eureka** para registrar a aplicação no Eureka.

```
org.springframework.boot
spring-boot-starter-web

org.springframework.cloud
spring-cloud-starter-eureka
```

A classe de configuração é igual qualquer aplicação Spring Boot, apenas adicionando a anotação **@EnableDiscoveryClient** para fazer o registro da aplicação no Eureka.

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;

@SpringBootApplication
@EnableDiscoveryClient
public class CustomersAppConfig {

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

O registro no Eureka é realizado no startup da aplicação, onde a aplicação se registra no service discovery.

Para mais sobre Eureka e Zuul: https://emmanuelneri.com.br/2018/05/02/criando-proxy-de-apis-com-spring-cloud-zuul-e-eureka/

Zuul

É uma solução de roteamento dinâmico que possibilita monitoramento, resiliência e segurança para aplicações, que também pode ser encontrada no sub-projeto spring-cloud-netflix. É responsável por ser a porta de entrada das chamadas e direcionar as chamadas para as aplicações registradas no Eureka.

As dependências são: **spring-cloud-starter-zuul** para inicializar o Zuul na aplicação e a do **Eureka** para registrar a aplicação no service discovery.

```
org.springframework.cloud
spring-cloud-starter-zuul

org.springframework.cloud
spring-cloud-starter-eureka
```

@EnableZuulProxy – ativar o Zuul na classe de configuração do spring Boot, logo o Zuul é inicializado no startup da aplicação.

```java
@SpringBootApplication
@EnableZuulProxy
@EnableDiscoveryClient
public class ZuulAppConfig {

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

As configurações também são as propriedades para definir porta e registar no Eureka, mas também as configurações do Zuul para mapear os caminho das APIs e qual seu destino.

A configuração das rotas do Zuul no application.properties segue o seguinte padrão: zuul.routes. + nome do serviço + . propriedade a ser configurada para aquele serviço. Essa configuração apenas é valida quando utilizado no Zuul no Spring Boot.

- **zuul.prefix**: Configura para que o contexto de entrada seja no /api, assim todos os serviços vão ser acessados pela URI /api;
- **zuul.ignored-services**: Quando configurado como '*', todos os serviços são ignorados por padrão, assim nenhum serviço vai ser acessado pelo Zuul, apenas vão estar disponível os que estiverem mapeados explícitamente como os de customers e products;
- **zuul.routes.customers.path**: Define a URI para acessar os dados de customer, no caso /api/customers;
- **zuul.routes.customers.serviceId**: Informa o ID da aplicação registrada no Eureka, que é o valor atribuído em cada aplicação na propriedade spring.application.name do application.properties;
- **zuul.routes.products.strip-prefix**: Configurado com false, o prefixo do serviço configurado no path não tera nenhum efeito no path original da sua chamada, assim quando requisitado /api/customers será redirecionado para /customers.