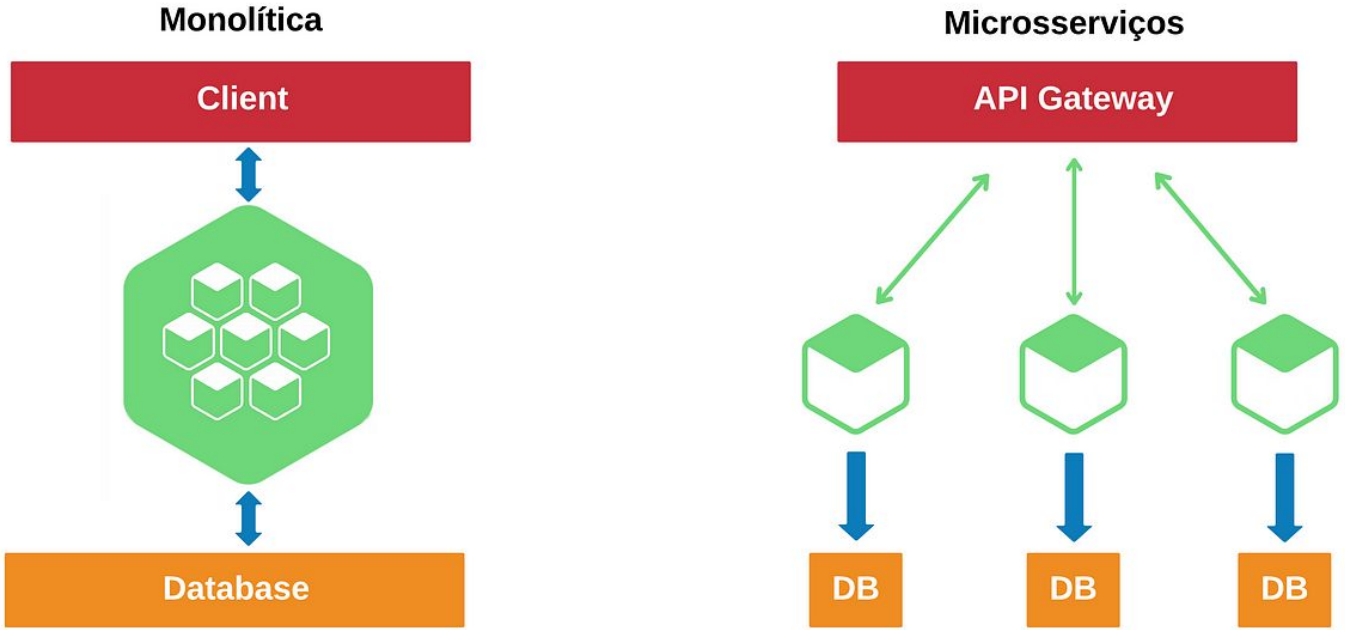
Micro Serviços com Spring boot

Por Manoel C M Neto

Arquitetura de Microservices x Monólitos

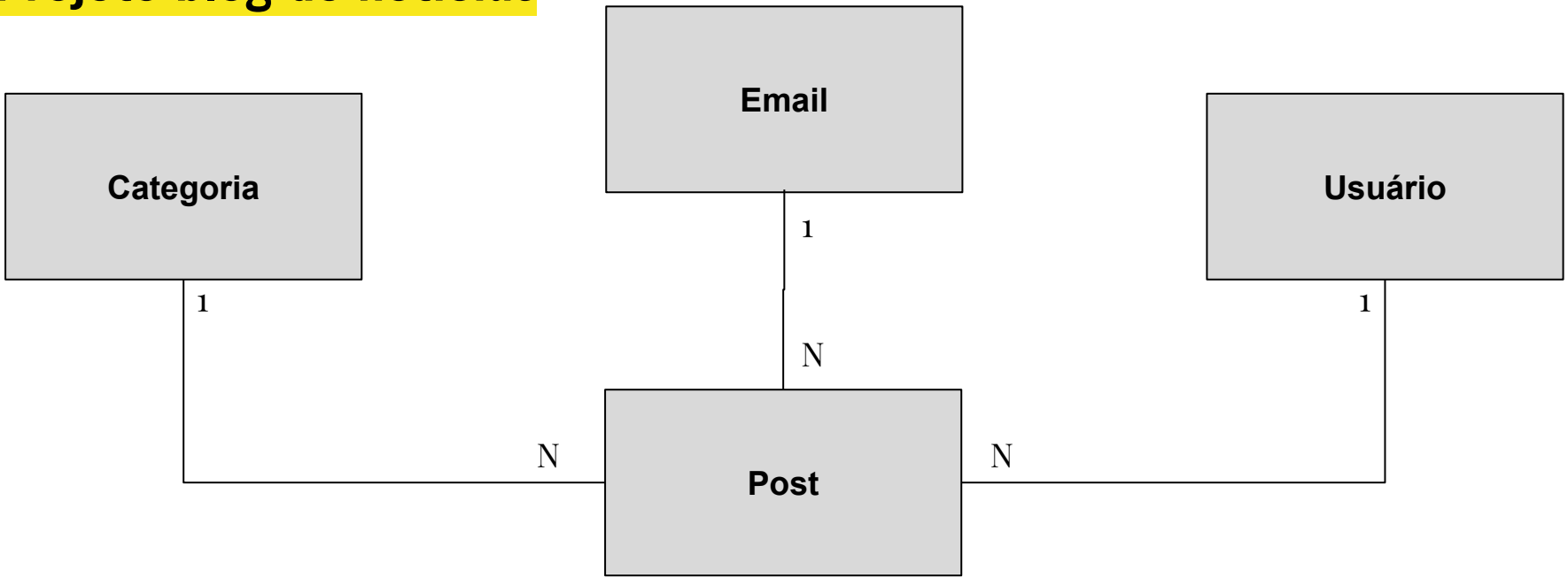


Arquitetura de Microsserviços

Arquitetura de Microserviços x Monólitos

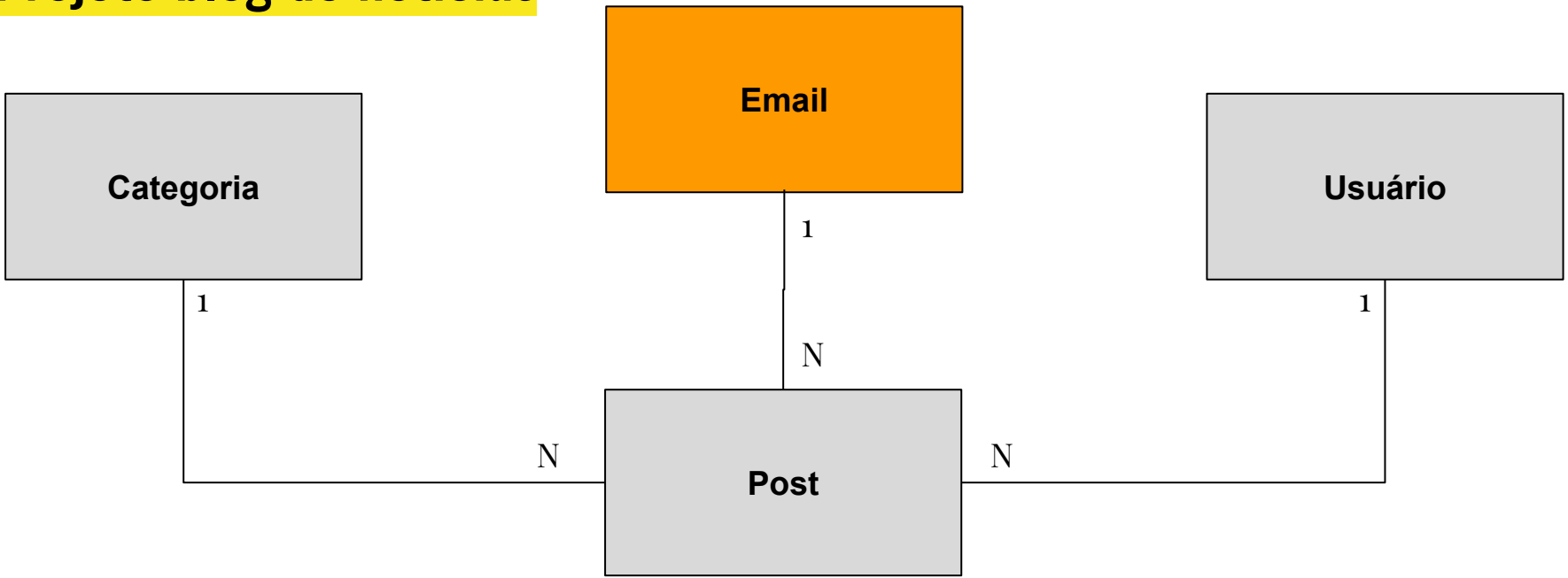
- Facilidade de mudança
- Times menores
- Reuso
- Diversidade tecnológica e experimentação (trocas de tecnologias)
- Maior isolamento de falhas
- Escalabilidade independente e flexível

Projeto blog de notícias

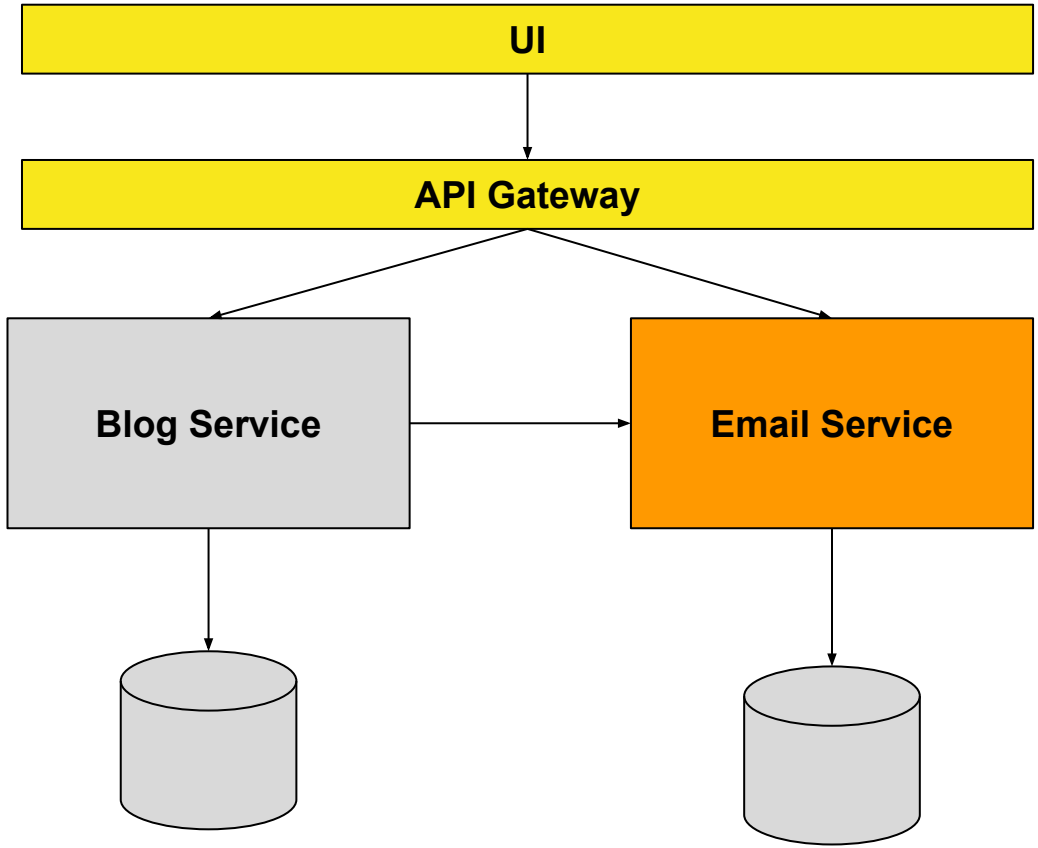


Projeto blog de notícias

Enviar email pode ser útil para várias outras APis



Projeto blog de notícias baseado em microverviços



Reusar Projeto Blog e Criar um novo Projeto Email Service

- Para exemplificar o uso de microserviços vamos criar duas Apis em projetos separados: Blog App (essa é a mesma que fizemos em sala) e EmailServices
- Crie um projeto no https://start.spring.io/ e inclua como dependências:
    - JPA
    - Devtools
    - Java Mail Sender
    - Postgres
    - Web

Spring Data: No diretório resources edite o application.properties

```
1.  spring.application.name=email
2.  #postgresql
3.  spring.datasource.url=jdbc:postgresql://localhost:5432/postgres
4.  spring.datasource.username=postgres
5.  spring.datasource.password=alunoifba
6.  spring.datasource.driver-class-name=org.postgresql.Driver
7.  #hibernate
8.  spring.jpa.database-platform=org.hibernate.dialect.PostgreSQLDialect
9.  spring.jpa.hibernate.ddl-auto=update
10. spring.jpa.show-sql=true
11.
```

Spring Data: Anotando os models para o padrão JPA

```
1.  @Entity(name="emails")
2.  public class Email {
3.      @Id
4.      @GeneratedValue(strategy = GenerationType.IDENTITY)
5.      private Long id;
6.      private String mailFrom;
7.      private String mailTo;
8.      private String mailSubject;
9.      private String mailText;
10.     private LocalDateTime sendDateEmail;
11.     @Enumerated(EnumType.STRING)
12.     private EmailStatus status=EmailStatus.SENT;
```

EmailStatus

```
1.  public enum EmailStatus {
2.
3.          SENT,
4.          ERROR
5.  }
6.
```

EmailDTO

```
1. public record EmailDTO(String mailFrom, String mailTo, String
   mailSubject, String mailText) {
2.         public EmailDTO(Email email) {
3.                 this( email.getMailFrom(),email.getMailTo(),
   email.getMailSubject(),email.getMailText());
4.         }
5. }
```

Criando um Email Repository

```
1. public interface EmailRepository extends JpaRepository<Email, Long>{
2.
3. }
```

EmailController

```
1.  @RestController
2.  @RequestMapping("/email")
3.  public class EmailController {
4.
5.          @Autowired
6.          private EmailService service;
7.
8.          @PostMapping("/send")
9.          public ResponseEntity<Email> sendEmail(@RequestBody EmailDTO data){
10.
11.                 return new ResponseEntity<Email>(service.sendEmail(data),HttpStatus.CREATED);
12.         }
13.
14. }
```

EmailService

```
1.  @Service
2.  public class EmailService {
3.          @Autowired
4.          private EmailRepository emailRepository;
5.          @Autowired
6.          private JavaMailSender emailSender;
7.          public Email sendEmail(EmailDTO dto) {
8.                  Email data=new Email(dto);
9.                  data.setSendDateEmail(LocalDateTime.now());
10.                 SimpleMailMessage message=new SimpleMailMessage();
11.                 message.setFrom(dto.mailFrom());
12.                 message.setTo(dto.mailTo());
13.                 message.setSubject(dto.mailSubject());
14.                 message.setText(dto.mailText());
15.                 data.setStatus(EmailStatus.SENT);
16.                 emailSender.send(message);
17.                 emailRepository.save(data);
18.                 return data;
19.         }
```

Edite o application.properties para incluir configuraçoes de envio de email

1. spring.mail.host=smtp.gmail.com
2. spring.mail.port=587
3. spring.mail.username=********@gmail.com
4. spring.mail.password=**********
5. spring.mail.properties.mail.smtp.auth=true
6. spring.mail.properties.mail.smtp.starttls.enable=true

Guia para gerar código de 16 dígitos para configurar o smtp do gmail:
https://support.google.com/accounts/answer/185833

Para Testar abra o Postman

1. POST : http://localhost:8080/email/send

```
{
    "mailFrom": "manoelnetom@gmail.com",
    "mailTo": "manoelnetom@gmail.com",
    "mailSubject": "Dúvida",
    "mailText": "Erro ao criar projeto"
}
```

Como Integrar os dois micro serviços a qualquer API?

- Até aqui temos dois serviços isolados.
- Como integrar/coordenar esses serviços de forma eficiente?
- Primeiro é preciso criar uma forma de fazer com que o serviços sejam localizados evitando decorar urls e portas diferentes de cada serviço.
- Para isso vamos criar uma estrutura chamada de **Service Discovery**
- No Spring isso é bem simples. Basta criar um novo projeto e incluir a dependência **Eureka**.
- O Eureka é um serviço que permite registrar e encontrar múltiplos micro serviços.

Configurando o Eureka Server

- Vá no https://start.spring.io/ e crie um projeto chamado eurekaserver.
- Inclua apenas a dependência Eureka Server
- Edite o application.properties:
  - server.port=8081
  -
  - spring.application.name=server
  - eureka.client.register-with-eureka=false
  - eureka.client.fetch-registry=false
  - eureka.client.serviceUrl.defaultZone=http://localhost:8081/eureka
- No **main inclua a anotação** @EnableEurekaServer
- **Rode e acesse** http://localhost:8081

Como Registrar Blog Service e Email Service do Erureka?

- O próximo passo é registrar os dois micro serviços no servidor eureka.
- Para isso em vamos precisar inclui a dependência Eureka Client **tanto no pow.xml de blog quanto no de email services.**
- Cuidado aqui! Use o starter Spring.io
- Anote o main com @EnableDiscoveryClient
- Edite o App.properties de cada MS e inclua**:**
  - **spring.application.name=<NOME DO MS>**
  - **eureka.client.serviceUrl.defaultZone=http://localhost:8081/eureka**
  - **server.port=0**
  - **eureka.instance.prefer-ip-address=true**

O gateway

- Agora que temos cada micro serviço acessível em uma URL diferente de forma automatizada como consequência temos um outro problema: **como centralizar todas as chamadas a API em uma única URL?**
- Para isso vamos criar complemento chamado gateway que fará o papel semelhante a roteador**.**
- Vá no https://start.spring.io/ e crie um projeto chamado gateway
- inclua as dependências: .
  - **Reactive Gateway**
  - **Eureka Client**

O gateway

- Agora edite o application.properties

- server.port=8082
- eureka.client.serviceUrl.defaultZone=http://localhost:8081/eureka
- eureka.instance.prefer-ip-address=true
-
- spring.application.name=gateway
- **spring.cloud.gateway.discovery.locator.enabled=true**
- spring.cloud.gateway.discovery.locator.lowerCaseServiceId=true

Padão da URL de Chamada

- O gateway aparece na lista de serviços do servidor eureka.
- O papel dele é padronizar as chamadas aos MS e deixar isso transparente para os clientes desses serviços.
- Dessa forma, para chamar o seu micro serviço use o padrão:
  - http://<IP>:<PORTA do Gateway>/<nome do MS>/<endpoint>
  - ex: POST via postman em http://10.25.30.2:8082/emial-ms/send/email
  - ex: GET via postman em http://10.25.30.2:8082/blog-ms/posts
  - Perceba que nos exemplos acima usamos a mesma URL base para micro serviços diferentes
  - Para um frontend o uso deles é transparente e igual a uma API monolítica

Balanceamento de Carga

- O gateway pode atuar também como um gerente que faz balanceamento de carga de forma automática.
- Para isso basta identificar cada instância do MS com um id único e rodar/subir quantas vezes você quiser esse serviço.
- Depois disso cada nova chamada ao MS será direcionada para uma instância específica.
- Edite o APP.PROPERTIES de cada MS e inclua:
  - **eureka.instance.instance-id=${spring.application.name}:${random.int}**
  - Para testar inclua no controlador esse método que exibe a porta de cada instância:

```
@GetMapping("/porta")

 public String retornaPorta(@Value("${local.server.port}") String porta){

        return String.format("Requisição respondida pela instância executando na porta %s", porta);

}
```

Comunicação Síncrona entre Micro Serviços

- Além da comunicação feita entre um frontend e um micro serviço existe também a necessidade de permitir a comunicação entre eles.
- Por exemplo: Como fazer o MS de blog enviar um e-mail (via o MS de e-mail) quando um post for cadastrado?
- Para isso vamos usar uma dependência chamada de **Open Feign**
- Add o **openfeign** no pow.xml de blog-ms via spring.start.io
- Depois disso add **@EnableFeignClients** no main do blog-ms

Comunicação Síncrona entre Micro Serviços

- Crie um pacote chamado **clients e dentro dele copie o EmailDTo de email-ms e implementa:**

```
@FeignClient("email-ms")

public interface EmailClient {

    @RequestMapping(method = RequestMethod.POST, value = "/email/send")

    public ResponseEntity<EmailDto> sendEmail(@RequestBody EmailDto dto);

}
```

Comunicação Síncrona entre Micro Serviços

- Em PostService add:

@Autowired

private EmailClient emailClient;

- **E ainda faça a chamada do método, ex:**

emailClient.sendEmail(new EmailDto("manoelnetom@gmail.com", "manoelnetom@gmail.com", post.getTitulo(), post.getTexto()));