API Gateway

1. Introdução - Retomando o que é o API Gateway

Olá! Te damos boas-vindas à nossa aula sobre API Gateway com foco em segurança. Em aulas anteriores, apresentamos o que é o API Gateway, mas aqui falaremos mais especificamente sobre Autenticação, spring cloud e spring security.

O próprio nome desse recurso já antecipa a sua funcionalidade: em português, gateway significa “portão de entrada”. Ficando entre o cliente e os serviços de back end, a API Gateway funciona como um proxy reverso que filtra o tráfego de todas as requisições e direciona os dados e chamadas ao local adequado.

O gateway atua como um mecanismo de segurança que garante a proteção por meio da autenticação do usuário, limitando conexões e permissões de usuários. Isso facilita o controle de acesso ao back end e ao serviço web, pois centraliza as requisições num único ponto de entrada, possibilitando o roteamento do tráfego.

Além disso, a API principal acaba tendo seus custos e complexidade reduzidos e torna-se muito mais simples de se manter, visto que a função do controle de acesso é concentrada na API gateway.

No vídeo a seguir, abordaremos um pouco mais sobre o tema e falaremos sobre Autenticação.

Spring Security - OAuth2

Intuitivamente, podemos relacionar o nome com autenticação e segurança. OAuth pode ser considerado um protocolo, uma estrutura de autorização que tem um padrão aberto para autenticação em sistemas utilizando HTTP. Empresas bastante grandes e famosas, como Twitter, Google, Facebook Github usam dessa tecnologia.

É um protocolo padrão para autorização, permite que aplicativos como web, mobile e desktop obtenham acesso limitado às informações de usuários através do protocolo HTTP.

É comum encontrarmos a utilização dessa tecnologia em telas de login, como por exemplo, em sites das empresas citadas acima. Porém, iremos sobretudo nos deparar muito com esse protocolo em autenticação em APIs, e ele será responsável por gerar tokens de acesso às APIs, sendo considerada uma forma de segurança bastante eficiente e segura.

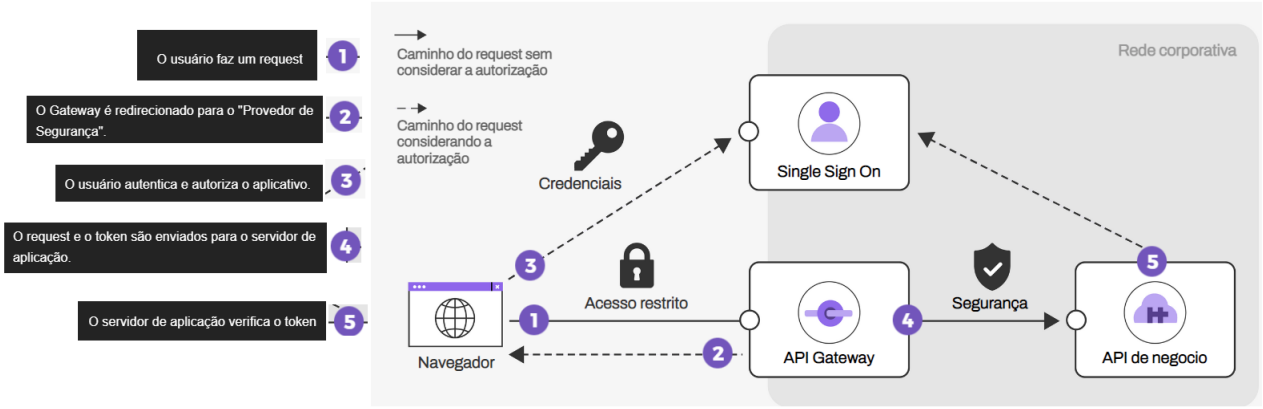
Por que OAuth?

Na prática, a maioria dos sites tem seus próprios mecanismos de autenticação (quando você se registra, é solicitado um endereço de e-mail e uma senha). Por outro lado, elas também permitem que você faça login com credenciais de outros fornecedores (Google, por exemplo). Muitos usuários utilizam o segundo mecanismo e, por sua vez, é conveniente para as empresas do ponto de vista comercial, pois automatiza o processo de onboarding (acesso dos usuários a uma nova plataforma) e, do ponto de vista tecnológico, delega o processo de autenticação - com seus custos e riscos associados - a um fornecedor validado.

É muito importante que você estude o material no Playground antes de cada reunião on-line para que você possa aproveitar ao máximo estas sessões com seus professores e colegas estudantes.

Você acha que o Google, Facebook e Twitter compartilham as credenciais de seus usuários?

A resposta a isto é OAuth, que é um padrão que permite compartilhar informações de autorização específicas de um provedor (Google, Facebook, etc.) para um consumidor (qualquer aplicativo), sem revelar a identidade do usuário.



Os 4 papéis do OAuth2

O OAuth 2 é estruturado em cima de quatro papéis (roles), que são:

Proprietário do Recurso (Resource Owner):
é a pessoa/entidade, que concede o acesso aos seus dados, literalmente, o dono do recurso. É como o OAuth2 classifica o usuário;

Servidor de Recurso (Resource Server):
a API que está exposta na internet precisa de proteção dos dados. Para conseguir acesso ao seu conteúdo, é necessário um token emitido pelo Authorization Server;

Cliente (Client):
é a aplicação que interage com o Resource Owner. No caso de uma aplicação web, seria a aplicação do browser;

Servidor de Autorização (Authorization Server):
é responsável por autenticar o usuário e emitir os tokens de acesso. É ele que possui as informações do Resource Owner (usuário). Autentica e interage com o usuário após a identificação do cliente.

Fluxo do processo de autorização

Agora que conhecemos os pilares que estruturam o OAuth 2.0, vamos entender o fluxo durante o processo de autorização. Entretanto, lembremos que o fluxo mostrado é uma descrição geral, pois pode ser ligeiramente diferente, dependendo do tipo de concessão e uso da autorização. Para entendê-lo, vamos rever os conceitos de autenticação e autorização:

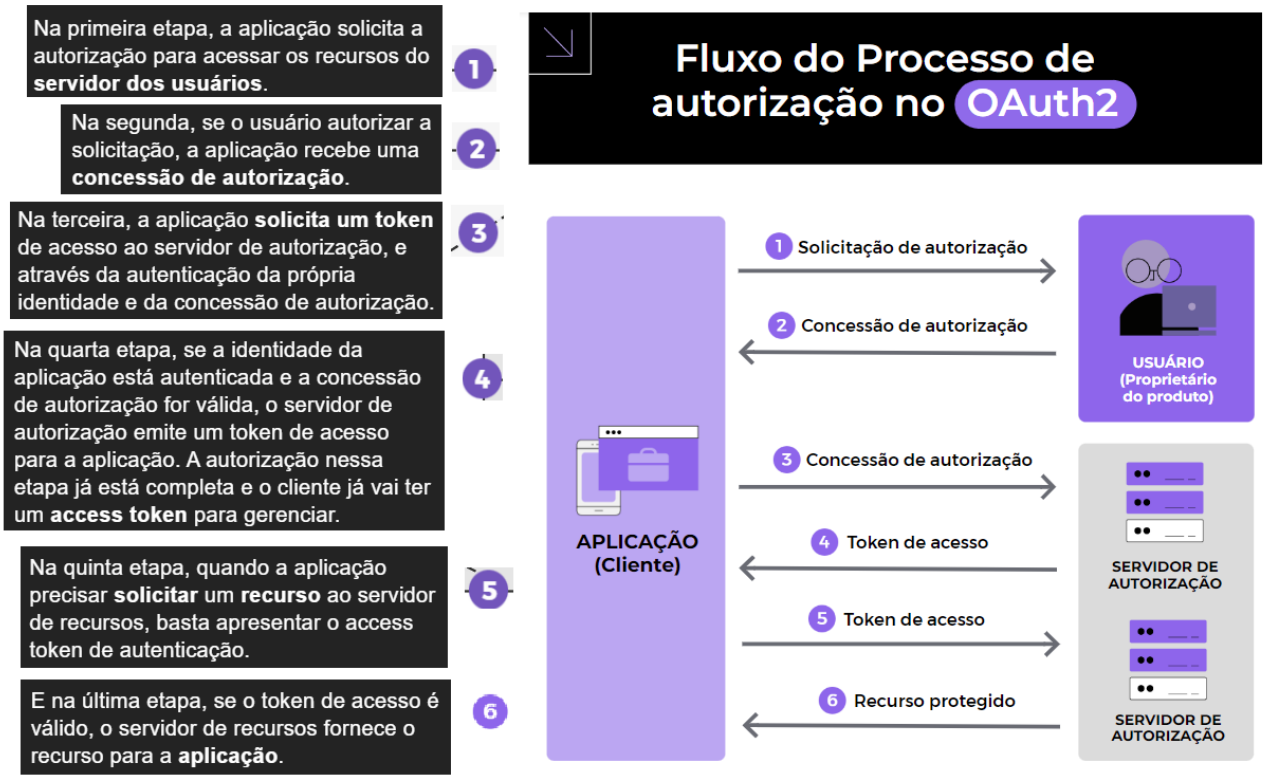
- **Autenticação**: é o processo que determina que um usuário é quem ele diz ser.

- **Autorização**: é o processo que determina que um determinado usuário pode realizar uma determinada ação.

Na prática, um usuário fornece um nome de usuário, uma senha e, em alguns casos, uma segunda senha (chamada de segundo fator de autenticação). A partir disto, um token de sessão (uma string criptografada) é obtido e verificado no servidor. O "uso malicioso" deste símbolo é conhecido como session hijacking.

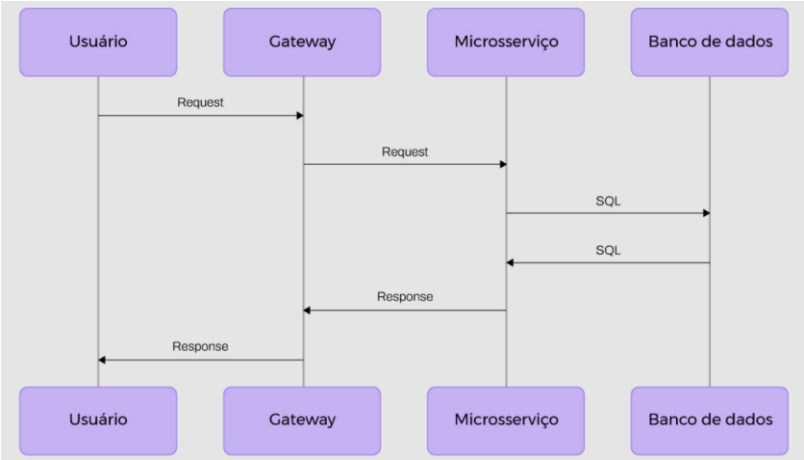
Com relação à autorização, um processo posterior à autenticação, cada vez que uma request é recebida, um objeto é obtido com os dados do usuário associados a essa sessão. Estes dados incluem as chamadas claims ou permissions, que geralmente são strings com um certo protocolo, por exemplo, JWT. Na prática, estas permissões nada mais são do que "funções". São atribuídas aos usuários estas funções e as aplicações ou microsserviços controlam que um usuário tenha uma ou mais funções, dependendo do caso.

A seguir, veremos o fluxo do processo de autorização do OAuth 2.0.

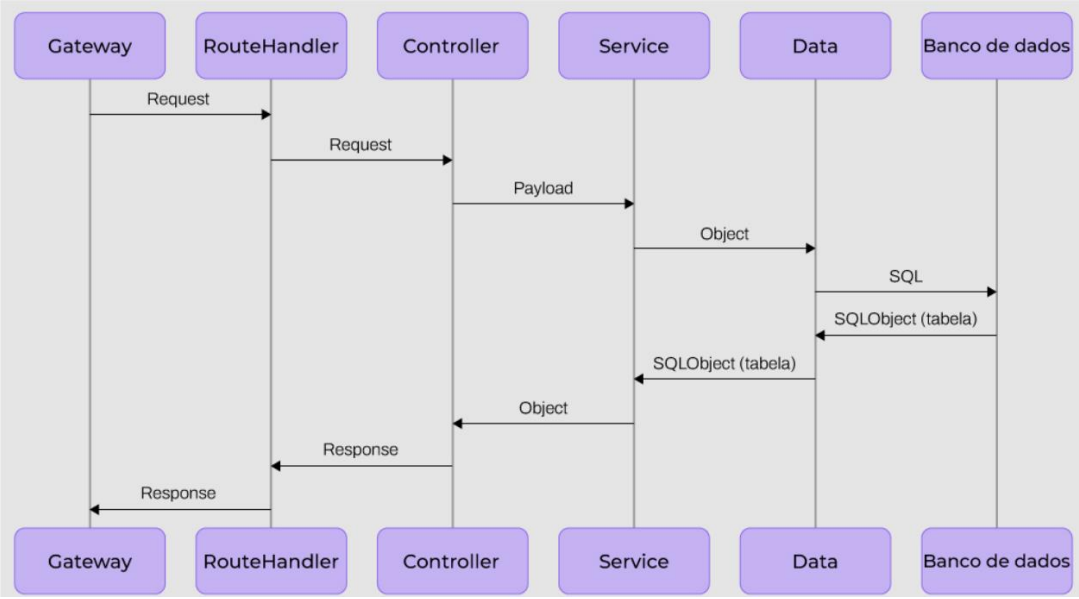


Sobre a segurança dos endpoints

Primeiro, vamos ver como é o fluxo de um request, assumindo que ele passou em todas as verificações de segurança.



Este tipo de diagrama é conhecido como um diagrama sequencial UML. As datas geralmente têm chamadas de método. Na prática, o microsserviço não é uma caixa preta e, além disso, o banco de dados faz parte do microsserviço. Vamos ver como seria em detalhes.



De acordo com o que vimos no gráfico, podemos ver que o gateway recebe um pedido para uma rota específica e sabe para qual instância do microsserviço redirecionar o pedido. Esta ação é executada pelo gateway através de um manipulador de rotas (RouteHandler), que, por sua vez, deriva o pedido a um controlador (Controller), que finalmente invoca o serviço comercial solicitado.

Depois, como em qualquer aplicação em camadas, a camada de serviço executa as transformações comerciais (DTO) e validações necessárias para enviar para a camada de dados apenas a informação de que necessita. A camada de dados traduz a informação recebida em instruções SQL a serem enviadas para a base de dados (padrão DAO).

Da mesma forma, a resposta gera o caminho inverso. O banco de dados responde à camada de dados a requisição gerada através da sentença SQL recebida. Em seguida, envia para a camada de serviço, para finalmente direcionar a informação ao controlador, que transforma a informação recebida em resposta e devolve essa resposta à origem, que (aos olhos de quem gerou a solicitação) é o gateway.

Segurança

No diagrama anterior, vimos com algum detalhe o caminho com o qual um request é executado. Quanto mais próximo um request se aproxima do banco de dados, mais caro ele é em termos de consumo de recursos.

Se pensarmos na segurança como um filtro, o ponto mais claro para aplicar a segurança é o gateway. Isto não é absoluto, mas é o caso típico, se um usuário externo só pode acessar através do gateway, não há sentido em aplicar segurança em microsserviços individuais. No entanto, os riscos são medidos de acordo com "probabilidade x impacto", e existem até mesmo certos regulamentos (SOX, por exemplo) e melhores práticas. Isto sugere que controles de segurança redundantes podem ser aplicados a fim de mitigar os riscos. Por exemplo, um endpoint no gateway tem uma falha de segurança que permite que um request seja passado para microsserviços sem autenticação. Este tipo de falha é conhecido como "back door".

Outro caso típico é o controle de bancos de dados, a restrição de certos comandos e até mesmo backups e rollbacks de transações não autorizadas. Vamos supor que um usuário logado consiga realizar um ataque de injection SQL em um campo de texto. O request, sendo autenticado, passaria pelo firewall e iria para o banco de dados, então, é necessário aplicar um "DROP ALL TABLES".

Nesta aula, vamos ver como aplicar oAuth em um gateway de Spring.

Spring Security

Revendo o que vimos nas aulas anteriores, o mecanismo de segurança padrão do Spring é Spring Security. Isto proporciona mecanismos de autenticação (um usuário é quem ele diz ser), autorização (um determinado usuário pode realizar certas ações) e gerenciamento de fichas de sessão. Há diferentes maneiras de representar isto, mas na prática acaba sendo uma série de tabelas no banco de dados contendo informações e permissões dos usuários. A mágica do Spring permite gerenciar automaticamente estas informações, o que se traduz em logins, permissões, acesso, ABM de usuário, etc.

Como funciona a autenticação baseada na web?

Toda vez que usamos o navegador e abrimos uma página web, fazemos internamente um request contra essa página. Ao fazer esta solicitação, além do caminho e dos parâmetros, o navegador envia informações na forma de "headers", que são valores de “key-value” que podem ser lidos por quem recebe a solicitação. Há vários tipos de header: de idiomas, locais, de gerenciamento de cache, mas estamos interessados nos de segurança.

Entre estes valores, temos o header “authorization”, que tem dois valores (separados por um espaço, lembre-se que o valor é uma string codificada): o primeiro é o tipo de autenticação e o segundo seriam as credenciais. Há muitas implementações e normas, e mesmo estas implementações são geralmente personalizadas por razões de segurança e desempenho.

É muito importante que você estude o material no Playground antes de cada reunião on-line para que você possa aproveitar ao máximo estas sessões com seus professores e colegas estudantes.

Antes de continuarmos, o que precisamos saber é que em cada request enviamos informações que devem ser verificadas pelo servidor.

O caso mais simples seria pedir o nome de usuário e a senha, armazená-la no lado do cliente e enviá-la codificada com cada request não criptografado. De fato, este tipo de autenticação é chamado "basic". Exemplo de header:

```
Authorization: Basic AXVubzpwQDU1dzByYM==
```

Isto é muito inseguro, pois qualquer pessoa que pudesse ler os headers teria automaticamente o nome de usuário e a senha do usuário.

Então, o próximo passo seria o servidor gerar um token, que normalmente é uma função hash (se você se lembrar de Java, a função hashCode gera uma string única a partir do valor das propriedades de um objeto), então, quando o usuário faz o login na primeira vez, o servidor retorna o token para ser usado a partir de agora. Este tipo de autenticação é chamado de Bearer.

```
Authorization: Bearer <token>
    </token>
```

Vamos supor que temos várias aplicações (ou várias APIs) e queremos diferenciar o acesso a uma ou outra API. Então, começamos a jogar com outro parâmetro que vem em combinação com o anterior. Este tipo de autenticação é conhecido como autenticação API Key.

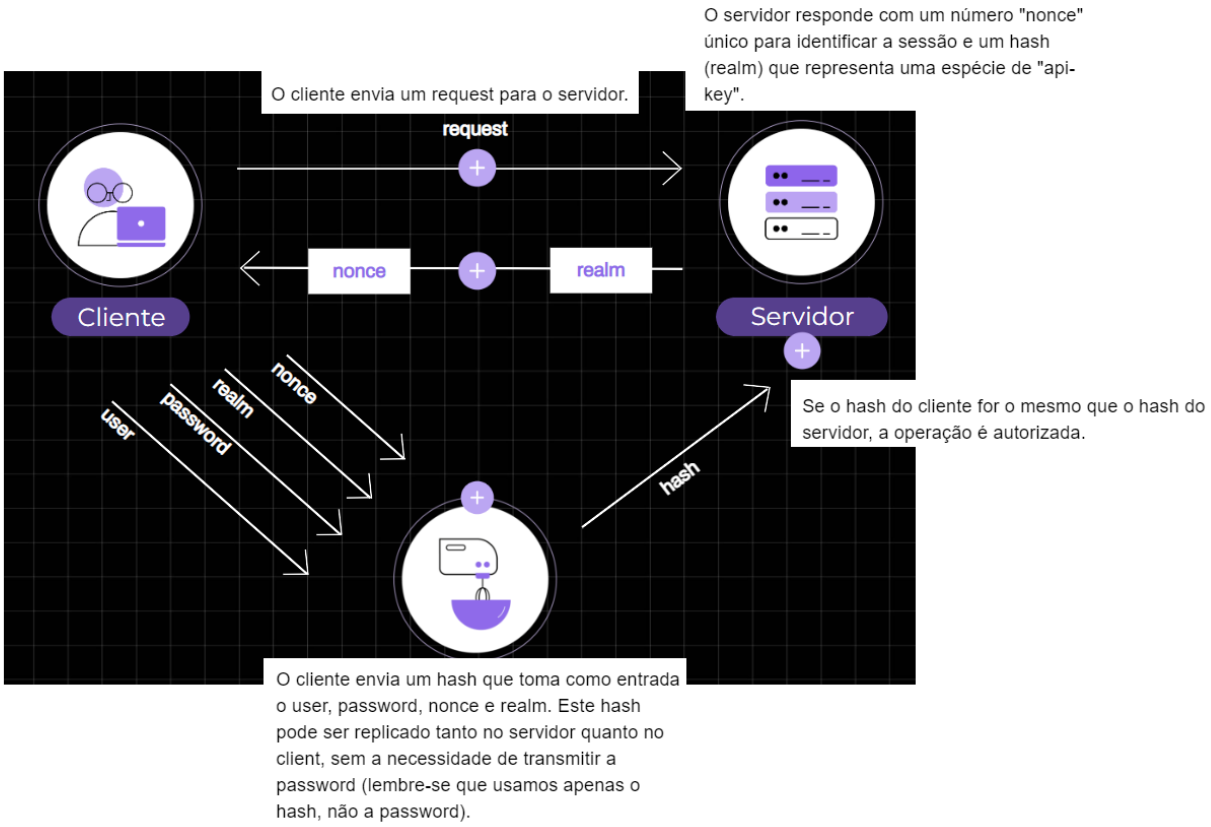
```
X-API-Key: abcdefgh123456789
```

Avancemos um pouco mais. Nos casos anteriores, você tinha que enviar nome de usuário e senha em texto "simples" (não estritamente falando, mas para fins ilustrativos, vamos supor que sim) pelo menos uma vez e armazená-los da mesma forma. Desta forma, o servidor deve saber nossa password. Mas será que poderíamos fazer o servidor não saber nossa password e ainda ser capaz de autenticá-la?

Estamos de volta ao início da função hashCode Java. Lembre-se que - quando você viu objetos - se você sobrescreveu a função equal, você deve fazer o mesmo com a função hashCode, porque "objetos iguais devem retornar o mesmo hashcode". Portanto, seguindo o raciocínio, se eu tiver duas strings e quiser saber se elas são iguais, poderia fazê-lo apenas comparando seus hashcodes. Assim, eu não teria que conhecer as strings originais. Este mecanismo é chamado digest, e nós compartilhamos com você um pedido de exemplo:

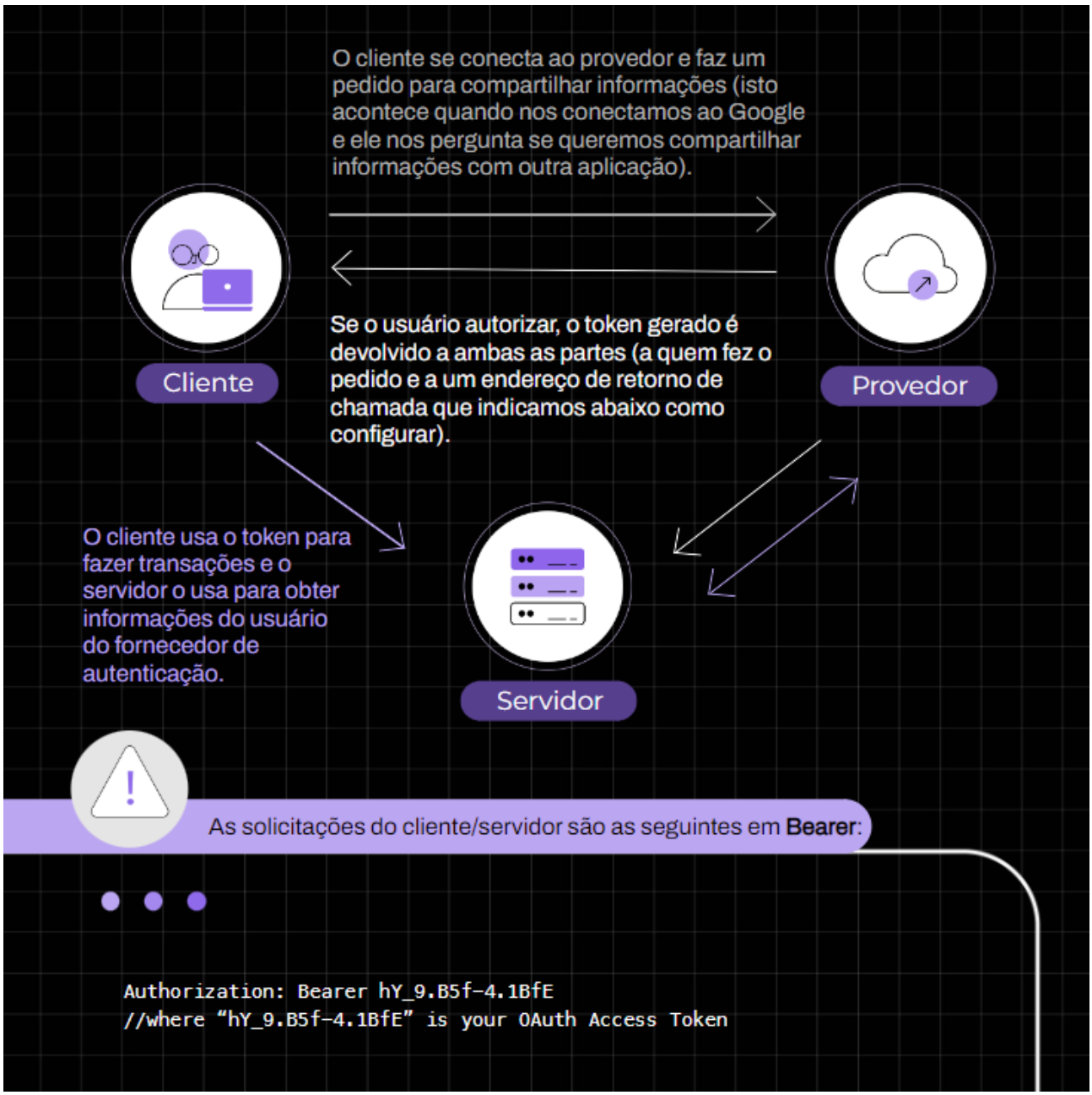
```
Authorization: Digest username="admin" Realm="abcxyz" nonce="474754847743646",
uri="/uri" response="7cffhfr54685gnnfgerg8"
```

Este mecanismo é um pouco mais complexo que os outros. Vamos esquecer o parâmetro URI que é o endereço de retorno. Isto funciona da seguinte forma:



Como funciona a autenticação web ou OAuth 2.0?

O OAuth 2.0 é usado para delegar o processo de login. Um usuário faz login em um site de terceiros e esse site deve retornar uma forma de autenticação. Temos três partes: o cliente, o servidor e o provedor de autenticação. O cliente deve entrar com o provedor de autenticação e o provedor de autenticação deve devolver um token para cliente e servidor, com o objetivo de combinar esses tokens para pedidos futuros um com o outro.



Conclusão

O uso do Spring Cloud tem a vantagem de garantir que esses serviços funcionarão bem em qualquer ambiente distribuído (incluindo a própria máquina do desenvolvedor, servidores e plataformas de cloud como AWS, IBM Cloud, Google Cloud).

A opção de trabalhar em aplicações de Spring Security é provavelmente a melhor alternativa para o desenvolvimento back-end visando o ecossistema da linguagem Java, já que esta tecnologia se tornou uma escolha fundamental para desenvolvedores e empresas em todo o mundo quando querem implementar recursos de segurança em suas aplicações.