Entendendo o padrão de segurança JWT

E seu uso em Java com Spring

Cleuton Sampaio, M.Sc.

Descrição

Toda aplicação web que retorna recursos dinâmicos deveria se proteger contra acessos anônimos. Há dois motivos para isto:

- Evitar roubo de informação (web scrapping);
- Evitar divulgação de informação corporativa;

Podemos proteger uma aplicação simplesmente adicionando um **CAPTCHA** (como o Google reCaptcha: https://www.google.com/recaptcha/about/)? Sim. Isso protege contra o roubo automático por scripts, mas não contra o acesso indevido manual, com uma pessoa navegando.

É preciso acabar com o acesso anônimo, protegendo rotas web que retornem recursos dinâmicos. Isso não só protege seus dados, como evita sobrecarregar seus servidores (de aplicação, de banco de dados etc) com acessos ilegítimos, o que é um tipo de ataque DOS (Denial of Service).

Existem vários esquemas de autenticação de usuário. Antigamente, quando utilizávamos o conceito de SESSÃO WEB, o cliente recebia um "cookie" identificador de sessão e tinha que sempre acessar o mesmo servidor.

Hoje, com o conceito de recursos **REST**, que são ***Stateless*** por natureza, este esquema não funciona mais, embora seja possível guardar o estado em um banco de dados no ***backend***.

Cookies não são uma forma muito segura de guardar identificações de estado, portanto, um novo esquema de autenticação é necessário.

JASON Web Token

Sempre que criamos um esquema particular de autenticação, ficamos sujeitos a incompatibilidades. O JWT (JASON Web Token) é um mecanismo para transitar informações entre ***frontend*** e ***backend*** de maneira padronizada.

Um token JWT é um conjunto de afirmações (“claims”) sobre um usuário remoto, por exemplo:

```
{
  "fresh": false,
  "iat": 1639743783,
  "jti": "72ad917e-e4f0-4efc-b797-9656e92b215c",
  "type": "access",
  "sub": "test",
  "nbf": 1639743783,
  "exp": 1639744683
}
```

Neste token temos alguns ***claims*** interessantes:

- sub: Subject ou o usuário que está enviando o token;
- exp: Expiration ou intervalo de tempo em que o token deixará de ser válido;

JWT e Sessões

Alguns utilizam tokens JWT para enviar estado de sessão mantido no ***frontend*** para o ***backend***. Para isso, utilizam ***claims*** particulares, fora do esquema padrão do JWT. Eu não considero uma boa prática, pois manter estado no ***frontend*** é inseguro e trafegar grandes quantidades de bytes por JWT é ruim.

Autenticação web

O padrão é utilizar tokens JWT para identificar um usuário autenticado. Você pode ter várias **rotas REST** em seu ***backend***, sendo que algumas exigem que o usuário esteja autenticado. Neste caso ele pode enviar um token JWT para provar isso.

Cabe ao ***backend*** validar o token JWT antes de retornar os recursos para o ***frontend***.

Há várias maneiras para o ***frontend*** enviar um token JWT para o ***backend***:

- Em um **cookie**: Tem a vantagem de ser automático e diminui o trabalho para o ***frontend***, mas há o risco de Cross Site Request Forgery (**CSRF**);
- Em um **header HTTP**: Uma das melhores opções, pois existe o header ***Authorization*** que é exatamente para isto;
- No corpo do **request**: Além de complicar o processamento do ***request***, também está sujeito a certos tipos de ataque;

O fluxo de autenticação seria algo assim:

1. **Frontend**: Tenta acessar uma rota protegida;
2. **Backend**: Retorna HTTP Status 401;
3. **Frontend**: Acessa uma rota de login, enviando credenciais;
4. **Backend**: Se tudo estiver correto, retorna um token JWT no corpo da mensagem;
5. **Frontend**: Acessa uma rota protegida, passando o header “Authorization: Bearer <token recebido>”;
6. **Backend**: Valida o token (vê se a assinatura está ok e se não expirou) e retorna o recurso protegido. Caso haja erro no token, pode retornar erro.

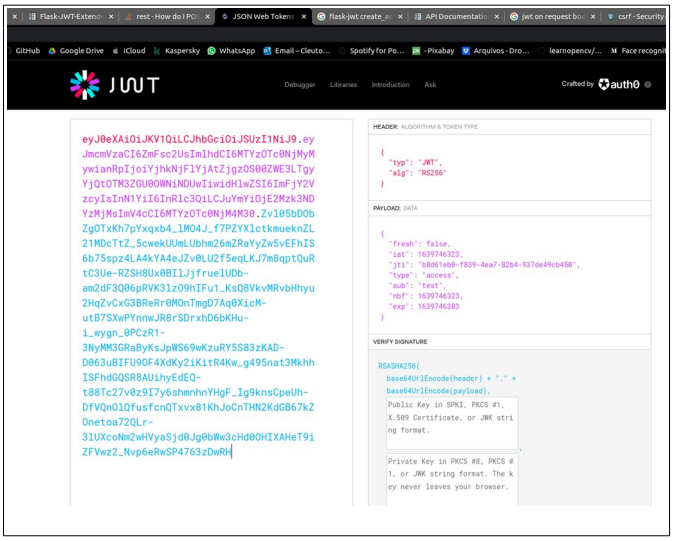
Há muitas variações, inclusive em esquemas de ***Single Sign-On*** e **autenticação federada**, mas este fluxo muda pouca coisa.

Componentes do token JWT

Um token JWT é algo assim:


```
eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJmcmVzaCI6ZmFsc2UsImlhdCI6MTYzOTc0NjMyMywianRpIjoiYjhkNjFlYjAtZjgzOS00ZWE3LTgyYjQtOTM3ZGU0OWNiNDUwIiwidHlwZSI6ImFjY2VzcyIsInN1YiI6InRlc3QiLCJuYmYiOjE2Mzk3NDYzMjMsImV4cCI6MTYzOTc0NjM4M30.Zvl05bDObZgOTxKh7pYxqxb4_lMO4J_f7PZYXlctkmueknZL21MDcTtZ_5cwekUUmLUbhm26mZRaYyZw5vEFhIS6b75spz4LA4kYA4eJZv0LU2f5eqLKJ7m8qptQuRtC3Ue-RZSH8Ux0BIlJjfruelUDb-am2dF3Q06pRVK3lzO9hIFu1_KsQ8VkvMRvbHhyu2HqZvCxG3BReRr0MOnTmgD7Aq0XicM-utB7SXwPYnnwJR8rSDrxhD6bKHu-i_wygn_0PCzR1-3NyMM3GRaByKsJpWS69wKzuRY5S83zKAD-D063uBIFU9OF4XdKy2iKitR4Kw_g495nat3MkhhISFhdGQSR8AUihyEdEQ-t88Tc27v0z9I7y6shmnhnYHgF_Ig9knsCpeUh-DfVQnOlQfusfcnQTxvx81KhJoCnTHN2KdGB67kZOnetoa72QLr-3lUXcoNm2wHVyaSjd0Jg0bWw3cHd0OHIXAHeT9iZFVwz2_Nvp6eRwSP4763zDwRH
```


Há 3 partes separadas por pontos: Header, Payload e Assinatura digital. Se quiser conferir o token, pode ir no site http://jwt.io:



Podemos ver que este JWT contém um header:

```
{
  "typ": "JWT",
  "alg": "RS256"
}
```

E um payload:

```
{
  "fresh": false,
  "iat": 1639746323,
  "jti": "b8d61eb0-f839-4ea7-82b4-937de49cb450",
  "type": "access",
  "sub": "test",
  "nbf": 1639746323,
  "exp": 1639746383
}
```

A última parte é a assinatura digital.

O token não deve conter informações privilegiadas, como senhas ou níveis de autorização, ou mesmo dados de sessão. Se for necessário incluir isso, então é melhor encriptar o token. A própria proteção do HTTPS é suficiente para manter um token JWT protegido. Porém, é preciso verificar se a assinatura digital é válida e se o token não expirou.

Há duas maneiras de assinar um token JWT:

- Chave simétrica: Todas as partes conhecem a chave, que serve para encriptar e decriptar;
- Chave assimétrica: Há uma chave privada e uma chave pública;

A maneira mais segura é assinar um token utilizando a **chave** assimétrica **privada** no ***backend***, e validar com a **chave** assimétrica **pública**.

Refresh de token

O que acontece quando o token expira? Simples: Você precisa se autenticar novamente!

Vários frameworks permitem ***refresh*** de token. Muitos geram novo token a cada request, ou quando o token está próximo de expirar. Também é possível colocar um prazo muito alto para expiração ou mesmo gerar token sem prazo de expiração, o que é muito arriscado.

Um esquema que eu gosto é NÃO FAZER REFRESH! Ao fazer ***refresh***, você revalida o token, permitindo que o usuário fique indefinidamente acessando com o mesmo token.

Você pode gerar um token com duração inicial alta, digamos: 1 dia, e, para operações realmente sensíveis (comprar, deletar etc) pode exigir que o usuário entre com a senha novamente, gerando novo token.

Muitas aplicações usam este esquema. Depois que o usuário se autentica, deixam acessar as rotas menos sensíveis mas, para acessar perfil ou realizar compras, exigem a senha novamente, gerando um token mais recente.

Cancelamento de token (logout)

Há certas histórias de usuário que podem requerer sair da aplicação, processo conhecido como ***logout***, ***logoff*** ou ***sign-off.*** Isso não é comum em aplicações web, mas existe em aplicações móveis. Neste caso é preciso invalidar o token, e não há uma maneira simples de fazer isso. Se o token não expirou, então pode ser usado novamente.

O que podemos fazer é criar um repositório no ***backend*** contendo os tokens cancelados e testar se um token recebido está entre eles.

Ataques de força bruta

Um dos tipos de ataque mais conhecidos (e que funciona até hoje) é tentar logar com diversas senhas conhecidas. A Wikipedia tem uma lista de passwords comuns, utilizadas até hoje: https://en.wikipedia.org/wiki/List_of_the_most_common_passwords

Veja só:

- password
- 123456
- qwerty

E no Brasil também temos uma lista (https://www.tecmundo.com.br/seguranca/229020-lista-mostra-senhas-comuns-vazamentos-brasil.htm):

- 123456
- 123456789
- Brasil
- 12345
- 102030
- senha
- 12345678
- 1234
- 10203
- 123123
- 123
- 1234567
- 654321
- 1234567890
- gabriel
- abc123
- q1w2e3r4t5y6
- 101010
- 159753
- 123321
- senha123
- mirantte
- flamengo

Se você repetir o login com algumas dessas senhas, há grande chance de sucesso. Os ***hackers*** utilizam bases de dados com usuários e senhas capturados em fraudes, criando ***scripts*** que ficam

tentando logar na sua aplicação o tempo todo. Eles podem até configurar tempos de espera aleatórios entre um login e outro, ou mudar de **zumbi** (máquinas infectadas) para te enganar.

Há algumas maneiras de se proteger disso:

1. No primeiro erro, enviar um **CAPTCHA** de volta para o ***frontend***;
2. Sempre exigir **CAPTCHA** para ***login***;
3. Contar a quantidade de tentativas de ***login*** e suspender (temporariamente ou não) aquele usuário;

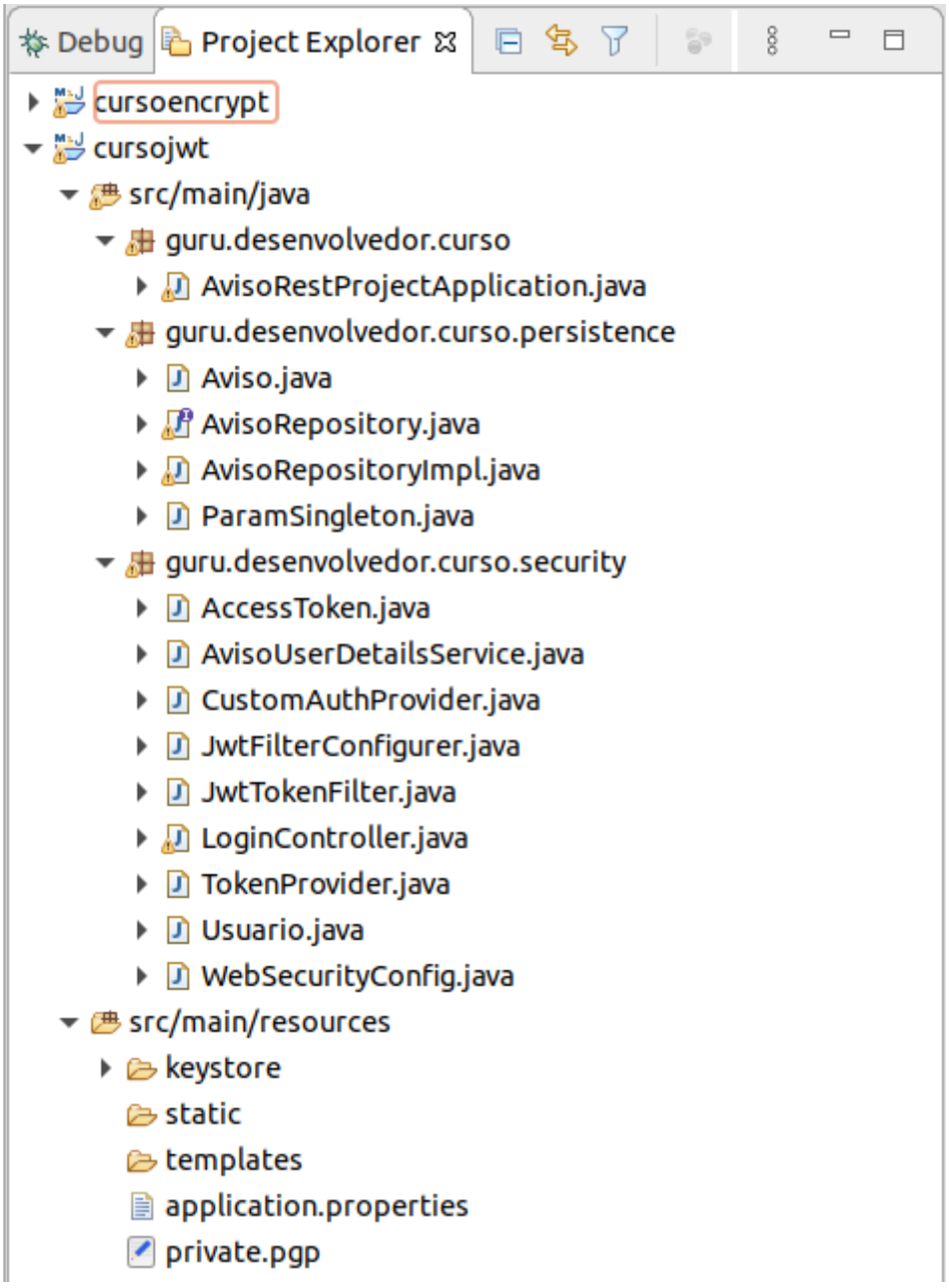
Isso é responsabilidade do ***backend*** e não está previsto no esquema JWT.

Em resumo

1. Tokens JWT servem para identificar um usuário autenticado;
2. Devem ser enviados no header Authorization;
3. Podem expirar, requerendo geração de novo token;
4. Não devem ser utilizados para trafegar nada além dos ***claims*** padrões;
5. Devem ser assinados digitalmente com chaves assimétricas;
6. Cancelamento de tokens devem ser lidados pelo ***backend*** com listas de bloqueio;
7. Cabe ao ***backend*** se precaver de ataques de força bruta;

Uso em Java com Spring

A estrutura do projeto é esta:



Para começar, já que o banco de dados PostgreSQL é encriptado, foi necessário modificar nosso Repositório. Agora, temos esta interface (“AvisoRepository”):

```
@RepositoryRestResource(collectionResourceRel = "api", path = "api")
public interface AvisoRepository extends Repository<Aviso,Long>{

	@Secured("ROLE_AUTENTICADO")
	List<Aviso> findAll();
}
```

Ao estender Repository, fazemos nossa implementação exportar métodos como rotas REST. E notem que eu marquei o método “findAll()”, que retorna a lista de notícias, como protegido (@Secured). Para acessar este método, o usuário deve enviar um token JWT válido no header “Authorization” do request. Por exemplo:

```
curl -i -k --header "Authorization: Bearer
eyJhbGciOiJSUzI1NiJ9.eyJzdWIiOiJmdWxhbm8iLCJhdXRoIjoiQVVURU5USUNBRE8iLCJpYXQiOjE
2NDAwNDI3MDIsImV4cCI6MTY0MDA0Mjc2Mn0.HaTKjbetZje4X7Zg7nDVLkcQs7fg4Z02O3ktbIbl96I
lxJliBJbYxNSwJxVgPzbqEIhJ1Mx0oP-
IiQsH14o4rqO1WyJq2ZKBoWFK5wxqVz4S5tSRqhNBtKwKylpJlzE-
JhocsVstie5P4dO3pOQePLf0rRTwnYCxAZBVZ62B8R7-
c1wAaDBSRoRUK5rbSTlMLAzXpUBe1kMBfkrMAPKqncDS4YjU_JA8Or-ckiZQSXYuPrKBWPAuYpcqJ-
pXNB_tcvMR9_xIfVxrGR5s-g1OvqOnfzENdPHhN7yiA7yYkLjtsQZyG4e6plGIK_cl-u-Gow-
ZFz1jNWkVFRS0nGGf-
EsyrY9TATHj6SRTLueIKsEevhu6V8FgjywqKJFOVW5wGsn_0__hITxbqVPH88xcOzCU0qfIMdLy7FffA
5kdecnvmixWuuS3efItiLlUgepV0agxgqspFJ8soQGpQZacBPsyjA-vLpgHcI2_SA90vPAp53f-
1d5mGclZCR-OsdDk" \
https://localhost:8080/api
```

Nosso repositório está mapeado para “api”, então ele vai invocar este método. Caso não tenha o token ou ele tenha expirado, um erro será retornado (403).

E tive que criar uma implementação dessa interface (“AvisoRepositoryImpl”):

```
public class AvisoRepositoryImpl implements AvisoRepository {
	@Autowired
	private EntityManager entityManager;
	private static String PRIVATE_KEY;
	private static String PASSPHRASE;

	public AvisoRepositoryImpl() throws Exception {
		PRIVATE_KEY = ParamSingleton.getPrivateKey();
		PASSPHRASE = ParamSingleton.getPassphrase();
	}

	@Override
	public List<Aviso> findAll() {
		Query q = entityManager.createNativeQuery("SELECT id, "
		+ " pgp_pub_decrypt(titulo, keys.privkey,'"
		+ PASSPHRASE + "') As titulo,"
		+ " pgp_pub_decrypt(resumo, keys.privkey,'"
```

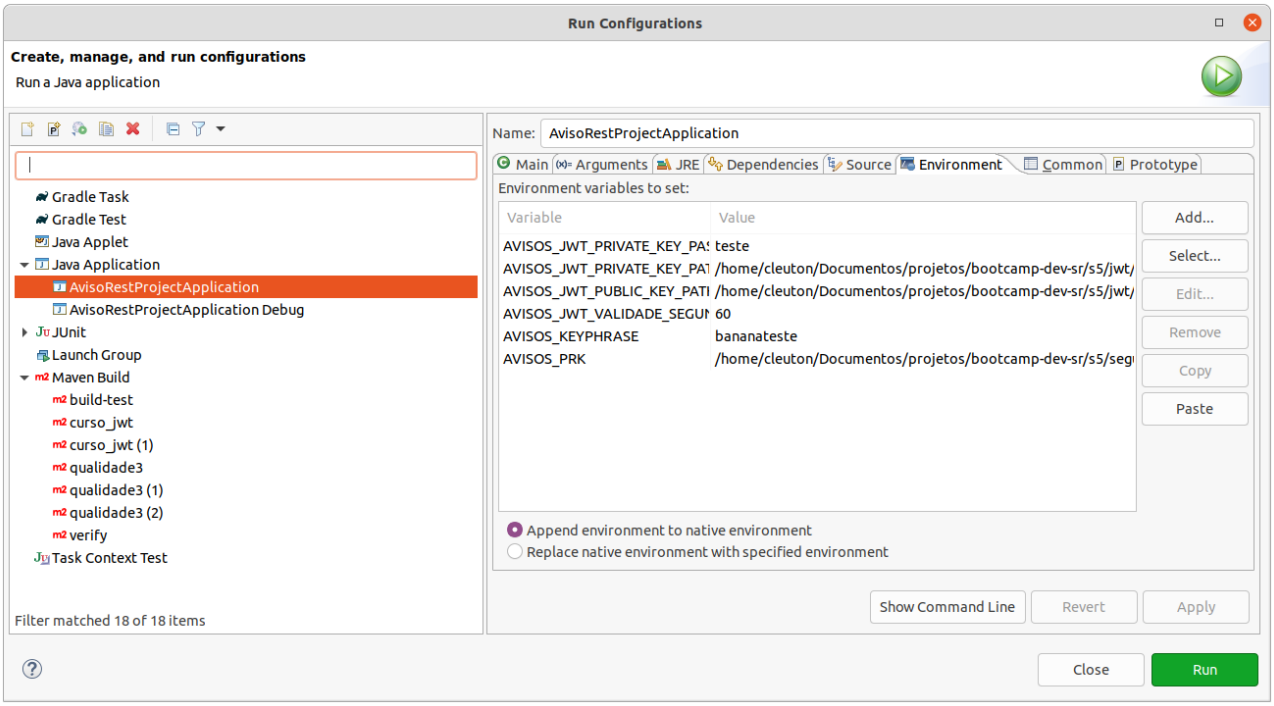
```
                + PASSPHRASE + "') As resumo,"
                + " pgp_pub_decrypt(thumb, keys.privkey,'"
                + PASSPHRASE + "') As thumb,"
                + " pgp_pub_decrypt(imagem, keys.privkey,'" + PASSPHRASE + "') As imagem,"
                + " pgp_pub_decrypt(texto, keys.privkey,'" + PASSPHRASE + "') As texto,"
                + " pgp_pub_decrypt(data, keys.privkey,'" + PASSPHRASE + "') As data "
                + "          "
                + "    FROM aviso "
                + "        CROSS JOIN"
                + "            (SELECT dearmor('" + PRIVATE_KEY
                + "') As privkey) As keys;", Aviso.class);

                List<Aviso> avisos = q.getResultList();

                return avisos;
        }

}
```

Criei uma query em SQL nativo (PostgreSQL) para acessar e decriptar as colunas. A chave privada e a passphrase são carregadas a partir de um Singleton (ParamSingleton) que pega os valores de variáveis de ambiente. Você precisa configurar as variáveis de ambiente no servidor ou dentro das configurações de execução da workspace no Eclipse:



Para representar o token, precisei criar um DTO (“AccessToken”) com uma propriedade chamada “access_token”. Precisei fazer isso para ficar compatível com o Python. Ao ser serializado, gerará uma resposta JSON como essa:

{**"access_token"**:"eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJmcmVzaCI6ZmFsc2UsImlhdCI6MTYzOTc1MjIyMywianRpIjoiODQ4YWFmZTMtYzIwMS00NTU0LTg4ZjktMzJiYTRiZTRkMmNhIiwidHlwZSI6ImFjY2VzcyIsInN1YiI6InRlc3QiLCJuYmYiOjE2Mzk3NTIyMjMsImV4cCI6MTYzOTc1MjI4M30.ZBNcAPW84KuRmXxIzMTWeypjJn6jSPHatlqSYiP0_UY1E22KI1bEmFMhq5OjXKvlibcJh_p2a4sReJ6Yi8dEdIYD9G4Nn6-3tu-dHkwdStt9EMJwGB8GPAtMc31OGDJVT6MuOgtpnXqzqyODiDLP1FEPfcCeyqPHPf35vs44Ks1rznfodUd14r6KMi3N342_eomsv0itq1u30kaEH1rNWIMqPrXSka8VdA1Xv_UP0PTPirU89o4hpCQyPDRNI1rtFX6ReA1wzTqKMFrFq2Cmkrw3CSePptJ2SM5eNGLaZvv37tpk9dL9YgDfYjxOZWa2U2D1EpdccX6h18cdBIObByJgTds5Z_qNX-bkhnX2mH2YCpF6odV4Ru7e7JWkIlZWD-3iv8q_TRp4q3f19RSAUPlIOpZQn17gAomYHUFIVmdZYupNnr3Gbk0YzHj_6GdNdNUL3nQJAUA_c4-mtXak1Zrtjf58P_9Qe0laPAcXe6JodLFteKpxtqY0lPSgVb5"}

Agora, como vamos recuperar os dados do usuário logado? O certo é criar um repositório e pegar isso do Banco de dados, mas, neste exemplo simples, eu só tenho um único usuário, portanto, criei um serviço que implementa UserDetailsService (do Spring Security) e retorna os dados. Muito simples (“AvisoUserDetailsService”):

```
@Service
public class AvisoUserDetailsService implements UserDetailsService {

	@Override
	public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {
		// TODO Auto-generated method stub
		return org.springframework.security.core.userdetails.User
			.withUsername(username)
			.password("senhafulano")
			.authorities("ROLE_AUTENTICADO")
			.accountExpired(false)
			.accountLocked(false)
			.credentialsExpired(false)
			.disabled(false)
			.build();


	}

}
```

E precisei criar um componente para fornecer tokens (“**TokenProvider**”) . Seu papel é criar e validar tokens JWT. Os métodos principais são:

- getToken: Obtém o token a partir do Header do request;
- validateToken: Valida o token JWT, verificando assinatura e expiração com o pacote JsonWebToken. Ele precisa de uma chave pública para isso;
- getAuthentication: É quem utiliza o serviço “AvisoUserDetailsService” para gerar um token de autenticação. É este método que efetivamente autentica o usuário;
- createToken: Se o usuário estiver autenticado, ele cria um token JWT com expiração e assinatura;

Mas não é só isso… Como eu preciso criar uma rota de login, tenho que criar um Rest Controller (“LoginController”) para a rota “/login”. Quando o usuário tentar fazer login, esta rota será invocada. Teoricamente, poderia colocar isso no repositório, mas não funcionaria. O método login é bem simples:

```
@PostMapping("/login")
public ResponseEntity<AccessToken> login(@RequestBody Usuario usuario) throws Exception {
	// Na vida real você vai utilizar um banco de dados ou um serviço OAuth
	try {
		Authentication auth =
		authenticationManager.authenticate(
		new UsernamePasswordAuthenticationToken(usuario.username, usuario.password)
		);
		AccessToken token = new AccessToken();
		token.access_token = tokenProvider.createToken(usuario.username,
			"AUTENTICADO");
		return ResponseEntity.ok(token);
	} catch (AuthenticationException e) {
		throw new Exception("Invalid username/password supplied");
	}
}
```

Depois, precisei criar mais algumas classes que o Spring Security precisa:

- WebSecurityConfig: Configura a autenticação e as outras classes necessárias;
- JwtFilterConfigurer: Configura o filtro de request que vai separar o token JWT;
- JwtTokenFilter: O filtro que verifica e valida o token, utilizando o “**TokenProvider**”;

A rota “login” faz o login do usuário:

```
curl -i -k --header "Content-Type: application/json" \
--request POST   \
--data '{"username":"fulano","password":"senhafulano"}'   \
https://localhost:8080/login
```

E podemos enviar o token obtido dessa forma:

```
curl -i -k --header "Authorization: Bearer eyJhbGciOiJSUzI1NiJ9.eyJzdWIiOi … R-OsdDk" \
https://localhost:8080/api
```