Relatório Desafio da Bry

Eduardo de Moraes

29 de setembro de 2025

1 Considerações iniciais

Seguindo as orientações do arquivo de desafio:

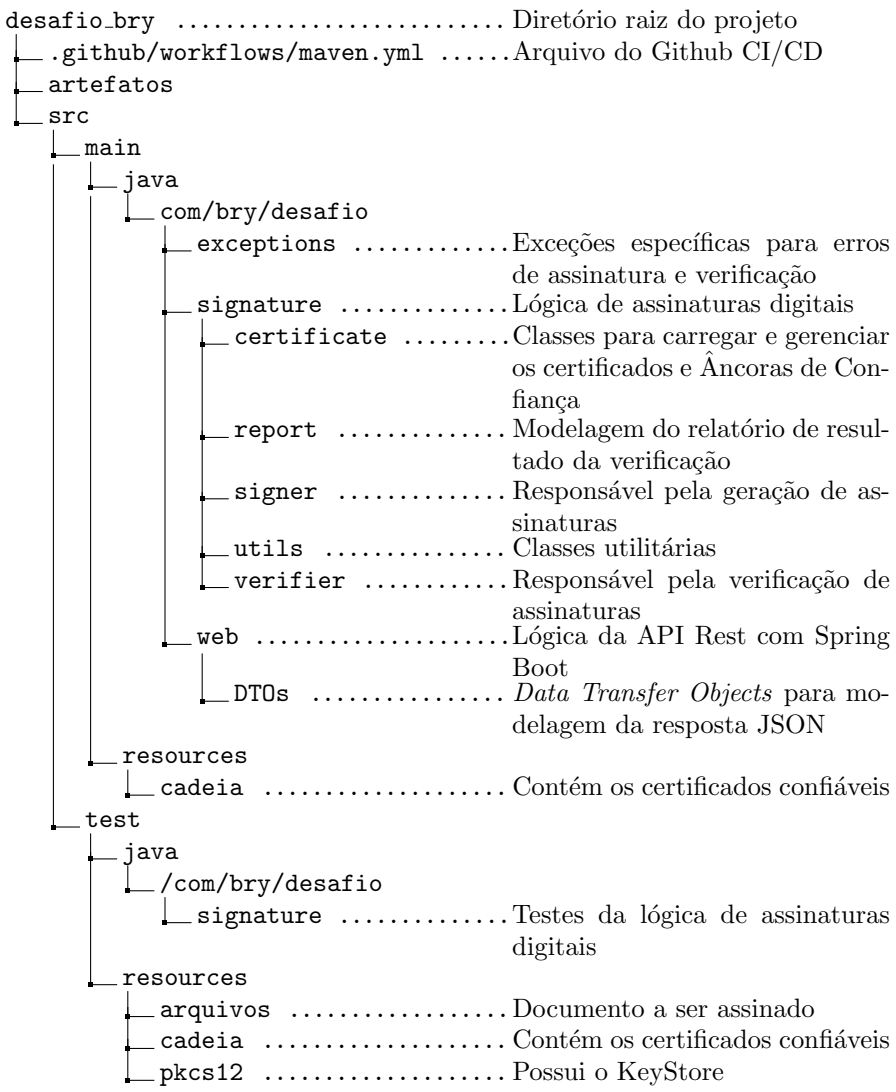
- Projeto do desafio está presente neste link;
- Orientações de como rodar o projeto estão contidas no arquivo `README.md`;
- Resumo criptográfico em hexadecimal da etapa 1 e assinatura da etapa 2 estarão no diretório artefatos, na raiz do projeto.

Para gerar os arquivos iniciais do projeto foi utilizada o Spring Initializr, ferramenta web para automatizar a geração de projetos Spring Boot, escolhendo a seguinte configuração:

- **Project**: Maven
- **Language**: Java
- **Spring Boot**: 3.5.6
- **Packaging**: JAR
- **Java**: 17
- **Dependencies**: Spring Web

2 Estrutura do projeto

O intuito foi separar a lógica de geração e verificação da lógica do Spring Boot, simulando um cenário no qual o serviço de assinatura da API consome uma biblioteca/framework de assinaturas digitais. A ideia por trás de cada classe está documentada nas mesmas. Abaixo, um pouco da lógica por trás da organização do projeto:



Essa foi a forma de fazer um código pensado para ser modular e expansível diante da existência de outros tipos de assinaturas básicas, como XMLDsig, JWS e PDFSignature, e nos diversos perfis que uma assinatura pode ter. Desta forma, qualquer feature poderia ser implementada com poucas ou sem mudanças estruturais e no código.

Por outro lado, para assinaturas avançadas, como XAdES, PAdES, CAdES e JAdES, é difícil deixar uma estrutura pronta, há muito mais processos do que uma assinatura básica, atributos da assinatura, Time Stamps, verificação da revogação dos certificados, Listas confiáveis, LPAs e PAs.

3 Etapas 1, 2 e 3

A geração do hash foi implementada utilizando o algoritmo SHA-512, conforme solicitado. Sem nenhum problema ou dificuldade. De forma idêntica, a geração de assinaturas foi bem simples e sem muitos problemas. Os passos implementados para gerar uma assinatura com alguns comentários:

1. Carrega a KeyStore para obter a chave privada e o certificado do signatário. A String passada pelo desafio estava incorreta, faltavam colchetes no início e fim (`{e2618a8b-20de-4dd2-b209-70912e3177f4}`), mas depurando passo-a-passo pude perceber esse problema e dar continuidade a essa tarefa;

2. Gera o Signer Information usando a chave privada, o certificado e o algoritmo de resumo;

3. Encapsula o Signer Information e o certificado do assinante em um CMSSignedDataGenerator. Inicialmente eu tinha esquecido de adicionar o certificado, fui me tocar disso na etapa de verificação, me fazendo lançar uma versão 0.2.1;

4. Gera a assinatura attached a partir do documento `doc.txt`.

Por fim, na verificação foi um pouco mais demorado, é um processo que possui mais particularidades. No geral, ficou dessa forma:

1. Obtém-se o SignerInformation do assinador e seu certificado, onde tem a chave pública para obter a mensagem assinada, de dentro da assinatura;

2. Verifica a integridade da assinatura, ou seja, confere se o conteúdo anexado na assinatura é igual à mensagem descriptografada pela chave pública;

3. Carrega as âncoras de confiança e as ACs intermediárias. Na verdade, essa etapa acontece assim que se inicia os testes ou é iniciado a aplicação com o Spring Boot (`TrustAnchorInitializer`);

4. Gera o caminho de certificação e a valida.

5. Se a assinatura estiver íntegra e confiável, adiciona no relatório de verificação informações adicionais, como o nome do signatário, data da assinatura, o hash do documento assinado e o algoritmo de resumo criptográfico usado para obter o hash do documento;

6. Encapsula todas as informações e coloca em uma classe `Report`, que será consumido pelos testes ou aplicação.

4 Etapa 4

Talvez foi a etapa mais desafiadora para mim, nunca desenvolvi nada com o Spring Boot. Apesar de ser um framework relativamente fácil de usar, tudo era novo. No fim, a arquitetura ficou dessa forma:

1. (`SignatureController`) para a camada web e um (`SignatureService`) para a lógica de negócio, reutilizando o código das etapas anteriores;

2. `TrustAnchorInitializer` como `@Component` para deixar disponível as âncoras e ACs intermediárias desde o início da aplicação;

3. `POST /desafio/assinar`: Endpoint que recebe dados no formato multipartform-data, como solicitado, contendo o documento a ser assinado, o arquivo PKCS12 do assinante e sua senha. Ele utiliza o serviço de assinatura e retorna o CMS codificado em Base64 no corpo da resposta;

4. `POST /desafio/verificar`: Endpoint que recebe um multipartform-data, contendo o arquivo de assinatura (.p7s). Ele consome o serviço de verificação e retorna um objeto JSON, com um status geral, "Válido"ou "Inválido", e um objeto com os detalhes da validação, como integridade, confiança e as informações extras do certificado.

Nesta etapa foi tentada a implementação da documentação automática com a biblioteca springdoc-openapi. No entanto, foi encontrado um problema em que a biblioteca não conseguia interpretar corretamente os endpoints multipart/form-data, não gerando as especificações para os `MultipartFile` e nem os DTOs de resposta.

5 Etapa 6

Etapa divertida, foi configurado uma pipeline de CI/CD utilizando o GitHub Actions. Nunca imaginei que seria tão simples automatizar a geração de releases que nem um projeto sério. Houveram frustações como o caso de eu não encontrar um jeito de gerar o pacote .jar contendo no nome a versão da release, persistia em ficar com a versão que está no `pom.xml`, coisa que eu sempre esquecia de mudar.

Entretanto, no geral, foi satisfatório criar um mecanismo em que a cada push ou pull request na branch principal, a pipeline aciona, compila o código e

executa todos os testes, garantindo a integração contínua. Da mesma forma que foi interessante criar tags e ativar triggers para que o projeto fosse empacotado em um JAR e publicasse uma release, garantindo a entrega contínua.

6 Testes e tratamento de erros

O tratamento de erro ficou bem detalhista, foram modeladas exceções no âmbito da geração e verificação de assinaturas, manipulação do certificado do signatário e processamento e gerenciamento de KeyStores, criando mensagens de erro específicas para ser fácil o entendimento do erro no relatório de verificação ou de geração de uma assinatura.

Sobre a testagem, foram feito apenas testes dos módulos criptográficos, geração de hash com uma referência, geração e posterior verificação de assinatura. Infelizmente, faltaram testes de cenários negativos, afinal, pode ser que as funcionalidades sempre dão positivo, independente se a comparação ser entre duas partes que obviamente são diferentes.

Não foi implementado teste para a camada de API, mas inicialmente foi desenhado uma forma para validar o `SignatureController` de forma isolada. Era para utilizar o `MockMvc` do Spring Test junto com mocks para o `SignatureService`. O objetivo era testar exclusivamente a responsabilidade do controller em lidar com as requisições HTTP, chamar corretamente os métodos do serviço e serializar as respostas, sem executar a lógica de criptografia em si a cada teste.