Relatório Desafio Prático BRy

João Pedro Adami do Nascimento

13 de março de 2023

Introdução

O Desafio Prático foi implementado em 2 projetos diferentes: SigningUtilities e SignApp. SigningUtilities possui a implementação das etapas 1, 2 e 3, enquanto que o SignApp possui a implementação da etapa 4.

O projeto SigningUtilities

Neste projeto as principais classes definidas foram `SigningUtilities` e `Main`. A classe `Main` possui o método `main()`, no qual as etapas de 1 à 3 são executadas com auxílio dos métodos estáticos `etapa1`, `etapa2` e `etapa3`. Além destes métodos, a classe conta com um método não estático, o `getResourceBytes`, que é responsável por ler arquivos presentes no diretório `resources/` e retornar estes na forma de *byte arrrays*. Os métodos estáticos não implementam as operações criptográficas, apenas fazem chamadas para métodos da classe `SigningUtilities` e tratam as exceções provocadas por estes retornando um único tipo de exceção, o `EtapaDesafioException`, que é tratado pelo método `main` a fim de definir se o programa pode continuar ou deve interromper a execução.

```
1  public static void main( String[] args )
2  {
3      // ...
4      try {
5          // Etapa 1: Resumo criptografico
6          etapa1(signingUtilities, docBytes);
7          System.out.println("Etapa 1 success!");
8      } catch (EtapaDesafioException e) {
9          System.out.printf("Etapa 1 error: %s\nEtapa 1 failed.\n\n%n", e.
               getMessage());
10     }
11
12     try {
13         // Etapa 2: Realizar uma assinatura digital
14         signature = etapa2(signingUtilities, docBytes, pkcs12Bytes);
15         System.out.println("Etapa 2 success!");
16     } catch (EtapaDesafioException e) {
17         System.out.printf("Etapa 2 error: %s\nEtapa 2 failed. Skipping Etapa 3
               due to Etapa 2 error...\nTerminating program.\n\n%n", e.getMessage()
               );
18         System.exit(1);
19     }
20
```

```
21      try {
22          // Etapa 3: Verificar a assinatura gerada
23          etapa3(signingUtilities, signature);
24          System.out.println("Etapa 3 success!");
25      } catch (EtapaDesafioException e) {
26          System.out.printf("Etapa 3 error: %s\nEtapa 3 failed.\n\n%n", e.
                getMessage());
27      }
28  }
```

Listing 1: Trecho de código do método main da classe Main

A classe `SigningUtilities` implementa as operações criptográficas da aplicação, estando em contato diretamente com as classes da biblioteca BouncyCastle. Foi necessária fazer esta separação entre as duas classes para que o código das operações criptográficas pudesse ser reutilizado pela API Rest da etapa 4.

A fim de prover desacoplamento do código desta classe em relação aos *digest provider*, algoritmos de assinatura, algoritmos de resumo criptográfico, entre outros, foram definidos atributos relacionados à estas informações. Desta forma, a classe que instancia `SigningUtilities` é quem é responsável por decidir o algoritmo de assinatura, o *digest provider* para operações de *hashing*, etc. Isto desacopla o código de `SigningUtilities` de objetos e decisões que podem mudar durante o desenvolvimento de um projeto.

Etapa 1

A etapa 1 não apresentou nenhuma dificuldade, sendo necessária apenas a leitura dos bytes do recurso doc.txt, o uso dos métodos `update` e `doFinal` da interface `Digest` por meio do objeto `messageDigest`, atributo de SigningUtilities, e finalmente a escrita do resumo criptográfico obtido em forma de string hexadecimal no arquivo `doc_hex_digest.txt`. O método de SigningUtilities utilizado nesta etapa é o `digestData`.

O uso de uma interface para o atributo `messageDigest` ao invés de uma classe (como `SHA256Digest`) possibilita o desacoplamento do código em relação ao algoritmo de *hash* utilizado, podendo desta forma instanciar a classe `SignignUtilities` com diferentes algoritmos de *hash*.

Etapa 2

Inicialmente faz-se o uso do método `loadCertKeyFromPKCS12` para recuperar o certificado e a chave privada do arquivo `desafio.p12`, e em seguida é chamado o método `signData`.

Esta etapa foi um pouco mais trabalhosa, nela busquei revisar o padrão CMS a fim de entender a estrutura interna deste, investiguei sobre a API do Bouncy Castle para a geração de assinaturas CMS e por fim consultei exemplos de uso desta API.

Os códigos de exemplo então foram adaptados para o método signData da seguinte forma:

- Foi implementado um tratamento de exceções mais robusto.
- Objetos do tipo Builder (de acordo com o *Builder Design Pattern*) foram movidos do código do método para os atributos da classe SigningUtilities, tornando responsabilidade de quem instancia um objeto SigningUtilities de fornecer os objetos builder em seu construtor com a configuração desejada (algoritmo de assinatura, algoritmo de hash, etc). Isto proporciona os seguintes benefícios:

– Melhor desempenho do método, pois evita-se a instanciação de um conjunto de objetos toda vez que o método é executado;

– Desacoplamento do código da aplicação em relação as configurações das primitivas criptográficas, uma vez que objetos do tipo Builder são objetos que configuram e constroem outros objetos mais complexos.

Desacoplamento de SigningUtilities

```
1 public class SigningUtilities {
2     private final Digest messageDigest;
3     private final List<X509Certificate> certList;
4     private final JcaSignerInfoGeneratorBuilder jcaSignerInfoGeneratorBuilder;
5     private final JcaSimpleSignerInfoVerifierBuilder
          jcaSimpleSignerInfoVerifierBuilder;
6     private final JcaContentSignerBuilder jcaContentSignerBuilder;
```

Listing 2: Atributos de SigningUtilities

```
1 public static void main( String[] args )
2     {
3         // ...
4         DigestCalculatorProvider digestProvider = null;
5         try {
6             digestProvider = new JcaDigestCalculatorProviderBuilder().
                  setProvider("BC").build();
7         }
8         // ...
9         SigningUtilities signingUtilities = new SigningUtilities(new
              SHA256Digest(), new JcaContentSignerBuilder("SHA256WithRSA"), new
              JcaSignerInfoGeneratorBuilder(digestProvider), new
              JcaSimpleSignerInfoVerifierBuilder());
```

Listing 3: Instanciação de SigningUtilities em Main.main

Etapa 3

A etapa 3 é implementada pelo método `verifySignature` de `SigningUtilities`. Neste método, o byte array que é passado como argumento é interpretado como um tipo `SignedData` do CMS, do qual são extraídos os campos `signerInfos` e `certificates` (RFC 5652 5.1). As informações do assinante juntamente com a assinatura ficam em um objeto do tipo `SignerInformation`, e por meio do método `verify` (passando como parâmetro o certificado do assinante), é feita a verificação da assinatura.

Não houveram grandes dificuldades nessa etapa além da revisão da RFC do CMS.

Etapa 4

Na etapa 4, a fim de reutilizar o código implementado nas etapas anteriores, declarei o projeto SigningUtilities como dependência no arquivo `pom.xml` de `SignApp`, e adicionei o JAR de `SigningUtilities` no diretório `libs/`.

```
<dependency>
    <groupId>org.desafiobry</groupId>
    <artifactId>signingutilities</artifactId>
    <version>1.0</version>
    <scope>system</scope>
    <systemPath>${project.basedir}/libs/signingutilities-1.0.jar</systemPath>
</dependency>
```

Listing 4: Trecho de código do pom.xml

Neste processo, a principal dificuldade foi fazer esta dependência externa ser reconhecida durante o processo de compilação. Para isto foi necessário adicionar a configuração `<includeSystemScope>true </includeSystemScope>` no `pom.xml` para que scopes do tipo system fossem considerados. Além disto, tive problemas do tipo "Invalid signature file digest for Manifest main attributes"provocados pelo fato do JAR de `SigningUtilities` ser um JAR assinado. Este problema foi resolvido deletando os arquivos de assinatura com o comando `zip -d signingutilities-1.0.jar 'META-INF/.SF' 'META-INF/.RSA' 'META-INF/*.DSA'` .

Fora estes empecilhos, a implementação foi simples, sendo necessária a criação apenas de uma classe além dá classe "main"do Spring Boot, a `SignAppController`. No controller, o principal esforço foi no tratamento de exceções, retornando respostas HTTP com código de status condizente à exceção que foi levantada, isto é, para erros provocados pelos dados fornecidos pelo cliente HTTP retornou-se respostas com status 400 (*Bad Request*), enquanto que para exceções internas da aplicação retornou-se respostas com status 500 (*Internal Server Error*).