Demoiselle Components Guide

Demoiselle Components

Humberto Pacheco

<humberto.pacheco@serpro.gov.br>

Sobre o Demoiselle Components

O *Demoiselle Components Guide* agrega em um único lugar toda a documentação referente a todos os componentes disponibilizados e que são compatíveis com a versão mais recente do *Demoiselle Framework*.

Parte I. Demoiselle Applet

O Security Applet simplifica a construção de Applets para manipulação de certificados digitais. Seu objetivo é o carregamento do Repositório de Chaves e Certificados (Keystore) de Certificados A3, cujo armazenamento é realizado por dispositivos seguros como Tokens USB ou Smart Cards, ou de Certificados A1, cujo armazenamento é feito em arquivos.

Configuração do Demoiselle Applet

1.1. Instalação do componente

Para instalar o componente *Demoiselle Applet* na aplicação, basta adicionar a sua dependência no arquivo `pom.xml` do projeto gerenciado pelo Maven:

```
<dependency>
    <groupId>br.gov.frameworkdemoiselle.component</groupId>
    <artifactId>demoiselle-certificate-applet</artifactId>
    <version>1.0.11</version>
</dependency>
```

1.2. Arquivos de configuração

O arquivo `security-applet.properties` define as propriedades do componente. Ele deve ser incluído no pasta de resource do projeto, caso contrário será utilizado o arquivo padrão `security-applet-default.properties` encontrado internamente no jar do demoiselle-applet.

Abaixo temos o conteúdo completo do arquivo de configuração.

```
#fabrica
factory.applet.action="applet-execute"

#Look and Feel
look.and.feel=Nimbus

#Labels
label.dialog.title=Autenticação
label.dialog.label.pin=Informe o seu PIN:
label.dialog.label.table=Lista de Certificados
label.dialog.button.run=Executar
label.dialog.button.cancel=Cancelar
label.dialog.option_pane.title=Mensagem de erro

#Messages
message.error.driver.incompatible=Nenhum driver encontrado ou Dispositivo Desconectado
message.error.invalid.pin=O PIN não foi informado ou é inválido.
message.error.unexpected=Erro inesperado
message.error.pkcs11.not.found=Nenhum Token/Smartcard presente. Por favor, conecte um
 dispositivo.
message.error.driver.not.available=Nenhum driver instalado para acesso ao seu dispositivo

#Configurations
config.dialog.modal=true
config.dialog.visible=true

config.dialog.dimension.width=470
config.dialog.dimension.height=310
```

```
config.dialog.pin-label.x=10
config.dialog.pin-label.y=5
config.dialog.pin-label.width=350
config.dialog.pin-label.height=15
config.dialog.pin-label.font=Dialog
config.dialog.pin-label.font.style=0
config.dialog.pin-label.font.size=12

config.dialog.pin-code.x=10
config.dialog.pin-code.y=25
config.dialog.pin-code.width=220
config.dialog.pin-code.height=20
config.dialog.pin-code.font=Dialog
config.dialog.pin-code.font.style=0
config.dialog.pin-code.font.size=12

config.dialog.button.font=Dialog
config.dialog.button.font.style=0
config.dialog.button.font.size=12

config.dialog.button-run.x=15
config.dialog.button-run.y=250
config.dialog.button-run.width=120
config.dialog.button-run.height=25

config.dialog.button-cancel.x=145
config.dialog.button-cancel.y=250
config.dialog.button-cancel.width=120
config.dialog.button-cancel.height=25

config.dialog.title.label.font=Dialog
config.dialog.title.label.font.style=0
config.dialog.title.label.font.size=12

config.dialog.table.certificates.font=Dialog
config.dialog.table.certificates.font.style=0
config.dialog.table.certificates.font.size=12

config.dialog.table.certificates.x=15
config.dialog.table.certificates.y=30
config.dialog.table.certificates.width=440
config.dialog.table.certificates.height=210
config.dialog.table.certificates.row.heigth=25
```

Funcionalidades

O princípio do componente é prover ao desenvolvedor o Keystore de um dispositivo. A partir do keystore a aplicação pode construir outras funcionalidades como autenticação e assinatura de documentos. Existem duas formas de se obter o Keystore do certificado, descritas a seguir.

2.1. O JKeyStoreDialog

O `JKeyStoreDialog` é um JDialog que solicita o PIN ao usuário retornando o KeyStore através do método `getKeyStore` .

```
JKeyStoreDialog dialog = new JKeyStoreDialog();
KeyStore keystoreDialog = dialog.getKeyStore();
```

2.2. O JKeyStorePanel

O `JKeyStorePanel` é um JPanel que pode ser renderizado junto a qualquer applet e possui o mesmo comportamento do `JKeyStoreDialog` .

```
public class JPanelApplet extends JApplet {
        public void init() {
                keyStorePanel = new JKeyStorePanel();
                ...
                this.getContentPane().add(keyStorePanel);
                ...
        }
}
```

Criando o projeto exemplo

Nesta seção apresentaremos o passo-a-passo para construção de um projeto de exemplo do *demoiselle-applet*. Nele será construída uma página html que executará a applet para obtenção de certificados A1 ou A3 e apresentação das informações do certificado na própria página html.

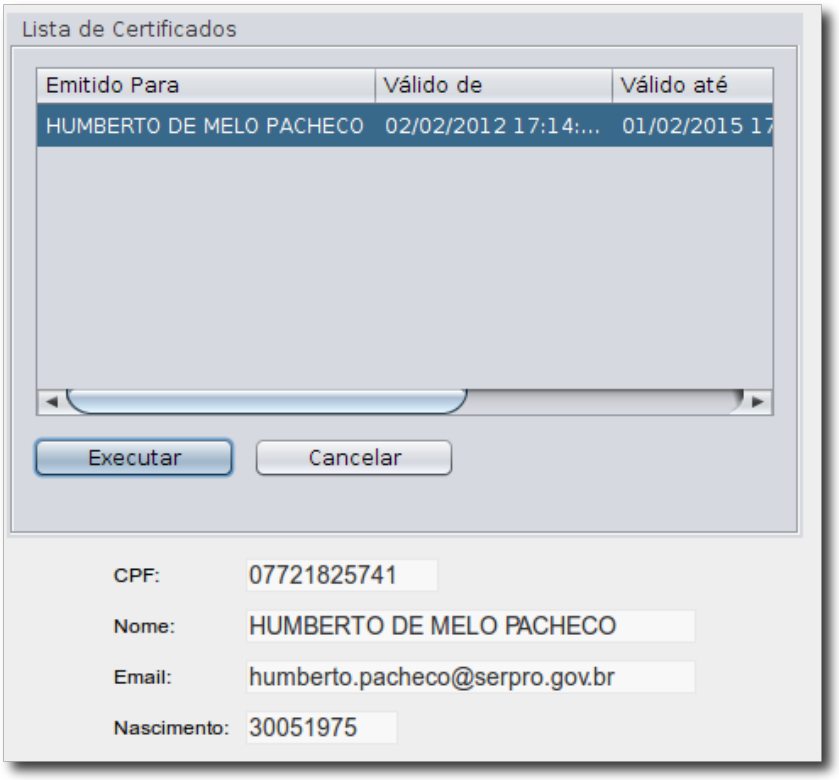

Figura 3.1. Tela Final do Projeto Exemplo

3.1. Criando uma AppletExecute

Crie um novo projeto com o nome de demoiselle-applet-customizada. Em seguida, crie uma classe que chamaremos de `App.java` no `pacote br.gov.frameworkdemoiselle` , estendendo a classe `AbstractAppletExecute` .

```
package br.gov.frameworkdemoiselle;

public class App extends AbstractAppletExecute {

    public void execute(KeyStore keystore, String alias, Applet applet) {
        try {
            ICPBrasilCertificate certificado = super
                .getICPBrasilCertificate(keystore, alias, false);
            super.setFormField(applet, "mainForm", "cpf", certificado.getCpf());
            super.setFormField(applet, "mainForm", "nome", certificado.getNome());
         super.setFormField(applet, "mainForm", "nascimento", certificado.getDataNascimento());
            super.setFormField(applet, "mainForm", "email", certificado.getEmail());
```

```
        } catch (KeyStoreException e) {
            e.printStackTrace();
            JOptionPane.showMessageDialog(applet, e.getMessage(), "Error",
                                          JOptionPane.ERROR_MESSAGE);
        }
    }

    public void cancel(KeyStore keystore, String alias, Applet applet);

    }
}
```

No código acima o método `execute` será acionado logo após o carregamento do keystore do usuário. O método `getICPBrasilCertificate` retorna um objeto do tipo ICPBrasilCertificate que possui todas as informações de um certificado ICPBrasil.

Os métodos `setFormField` escrevem no formulário html chamado de mainForm no qual a applet está sendo executado. O terceiro parâmetro do método informa em qual campo do formulário a informação será registrada.

O método `cancel` pode ser utilizado para implementar uma ação no caso do usuário desistir da ação. No código de exemplo é feito apenas a ocultação da applet.

3.2. Implementando o método Cancel

O componente possui agora uma ação de cancelamento padrão, que oculta e libera os recursos da tela de certificados. Caso seja necessário fazer mais alguma ação além desta, o desenvolvedor pode implementar uma customização através do método cancel.

```
public void cancel(KeyStore keystore, String alias, Applet applet){
    /* Seu codigo customizado aqui... */
}
```

3.3. Integração do applet com páginas web

O componente do applet possui a capacidade se comunicar com campos html de uma página web.

Para prencher um valor vindo do applet em um componente html, vamos considerar o código html abaixo:

```
<form id="mainForm" name="mainForm" method="post">
        <input type="text" name="cpf" value="" size="11">
</form>
```

Para preenchermos o campo html descrito acima com um valor fornecido do applet, utilizamos o código a seguir.

```
super.setFormField(applet, "mainForm", "cpf", "12345678900");
```

Para obtermos o valor do campo html descrito acima para ser utilizado pelo applet, utilizamos o código a seguir.

```
super.getFormField(applet, "mainform", "cpf");
```

3.4. Empacotando a applet customizada

Crie o jar do seu projeto (demoiselle-applet-customizada-1.0.0.jar) que conterá sua classe App.class.

```
demoiselle-certificate-applet-customizada-1.0.0.jar
   |---br
   |   |---gov
   |        |----frameworkdemoiselle
   |                |---App.class
   |---security-applet.properties
```

Nota

A criação do arquivo security-applet.properties é opcional. Sua função é sobrescrever todas as propriedades do componente.

3.5. Criando a Página HTML

Segue o código do formulário html.

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/
loose.dtd">
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Projeto Exemplo</title>
        <style type="text/css">
            table.padrao {
                background-color: #eeeeee;
            }

            td.padrao {
                font-family: "Arial";
                font-size: 12px;
            }

            input.padrao {
                font-family: "Arial";
                font-size: 14px;
                background-color: #F8F8F8;
                border: 1px solid #E8E8E8;
            }
        </style>
    </head>

    <script type="text/javascript">
```

```
        function foo() {
            alert("foo() method called!");
        }
    </script>

    <body>
        <form id="mainForm" name="mainForm" method="post" action="FileUploadServlet">
            <table align="center" border="0" cellspacing="5" cellpadding="1" class="padrao">
                <tr>

   <td   align="center"><applet   codebase="http://localhost:8080/demoiselle-certificate-applet-
customizada-web/"       code="br.gov.frameworkdemoiselle.certificate.applet.view.JPanelApplet"
 width=470 height=310 MAYSCRIPT
                     archive="demoiselle-certificate-applet-customizada-1.0.0-assinado.jar,
                        demoiselle-certificate-applet-1.0.11-assinado.jar,
                        demoiselle-certificate-core-1.0.11-assinado.jar,
                        demoiselle-certificate-criptography-1.0.11-assinado.jar,
                        demoiselle-certificate-signer-1.0.11-assinado.jar,
                        demoiselle-certificate-ca-icpbrasil-1.0.11-assinado.jar,
                        demoiselle-certificate-ca-icpbrasil-homologacao-1.0.11-assinado.jar,
                        bcmail-jdk15-1.45-assinado.jar,
                        bcprov-jdk15-1.45-assinado.jar,
                        log4j-1.2.15-assinado.jar,
                        slf4j-api-1.6.1-assinado.jar",
                        plugin-1.0-assinado.jar>

                   <param name="factory.applet.action" value="br.gov.frameworkdemoiselle.App" /
>
                            <param name="applet.javascript.postaction.failure" value="foo" />
                        </applet>
                    <td>
                </tr>
                <tr>
                    <td>
                             <table align="center" border="0" cellspacing="0" cellpadding="5"
 class="padrao">
                            <tr>
                                <td class="padrao">CPF:</td>
                                    <td><input class="padrao" type="text" name="cpf" value=""
 size="11"></td>
                            </tr>
                            <tr>
                                <td class="padrao">Nome:</td>
                                         <td><input class="padrao" type="text" name="nome"
 value="" size="30"></td>
                            </tr>
                            <tr>
                                <td class="padrao">Email:</td>
                                          <td><input class="padrao" type="text" name="email"
 value="" size="30"></td>
                            </tr>
                            <tr>
                                <td class="padrao">Nascimento:</td>
                                      <td><input class="padrao" type="text" name="nascimento"
 value="" size="8"></td>
                            </tr>
                        </table>
                </tr>
            </table>
```

```
        </form>
    </body>
</html>
```

3.6. Criando um tratamento de erro por JavaScript

Caso ocorra algum problema na recuperação do certificado digital, o componente está preparado para efetuar o tratamento. Entretanto, existem casos em que o desenvolvedor precisa executar ações adicionais. Para estes casos, foi disponibilizado um novo recurso em que o componente efetua a chamada de um método JavaScript que pode ser personalizado conforme a necessidade da aplicação. Abaixo temos um exemplo de método:

```
<script type="text/javascript">
        function foo() {
                alert("foo() method called!");
        }
</script>
```

Nota

A únicas propriedades necessárias ao funcionamento da applet são a `factory.applet.action` , que define qual classe será instanciada no momento do clique do botão Ok e carregamento do Keystore do usuário, e a `applet.javascript.postaction.failure` , que define qual método JavaScript deverá ser chamado. Mesmo que este último recurso não seja usado, é necessário que seja definido um método vazio, do contrário o componente exibirá um erro informando que não foi possível encontrar um método válido.

Importante

A chamada do método JavaScript por padrão só funciona no Linux e Mac, pois neles o acesso é feito diretamente pelo Hardware, e não por uma camada de abstração como no Windows. Para que este recurso funcione também no Windows, é necessário indicar que o acesso deve ser feito diretamente pelo hardware também. Consulte a documentação do componente *demoiselle-core* para detalhes sobre este recurso.

3.7. Assinatura dos jars

Para publicação do projeto será necessário assinar todos os jar necessários a execução da applet, conforme mostrado na tabela abaixo:

Tabela 3.1. Lista dos jars assinados

Jar Original	Jar Assinado
demoiselle-certificate-applet-customizada-1.0.0.jar	demoiselle-certificate-applet-customizada-1.0.0-assinado.jar
demoiselle-certificate-applet-1.0.11.jar	demoiselle-certificate-applet-1.0.11-assinado.jar

Jar Original	Jar Assinado
demoiselle-certificate-core-1.0.11.jar	demoiselle-certificate-core-1.0.11-assinado.jar
demoiselle-certificate-signer-1.0.11.jar	demoiselle-certificate-signer-1.0.11-assinado.jar
demoiselle-certificate-criptography-1.0.11.jar	demoiselle-certificate-criptography-1.0.11-assinado.jar
demoiselle-certificate-ca-icpbrasil-1.0.11.jar	demoiselle-certificate-ca-icpbrasil-1.0.11-assinado.jar
bcprov-jdk15-145.jar	bcprov-jdk15-145-assinado.jar
bcmail-jdk15-145.jar	bcmail-jdk15-145-assinado.jar
log4j-1.2.15.jar	log4j-1.2.15-assinado.jar
slf4j-api-1.6.1.jar	slf4j-api-1.6.1-assinado.jar
plugin-1.0.jar	plugin-1.0-assinado.jar

Para utilizar certificados de homologação, o componente abaixo deve ser incluido. Não esqueça de removê-lo quando a aplicação foi enviada para produção ou o usuário poderar usar certificados de homologação neste ambiente.

Tabela 3.2. Lista dos jars assinados

Jar Original	Jar Assinado
demoiselle-certificate-ca-icpbrasil-homologacao-1.0.11.jar	demoiselle-certificate-ca-icpbrasil-homologacao-1.0.11-assinado.jar

Para mais detalhes sobre os procedimentos para assinatura de jar, consulte o *Capítulo 4, Assinatura de Jars*.

3.8. Publicando no servidor Tomcat

Para testar nosso projeto, crie um novo projeto Web (No exemplo utilizamos o nome demoiselle-certificate-applet-customizada-web) e copie todos os jar assinados e seu index.html para a pasta WebContent. Adicione seu projeto ao servidor Tomcat e acesse a aplicação pelo navegador.

3.9. Executando a applet

Abra seu navegador na url na qual foi publicado os arquivos. Neste exemplo seria http://localhost:8080/demoiselle-certificate-applet-customizada-web. Se o certificado digital necessitar do pin para que seja feito o acesso, a aplicação solicitará imediatamente o pin de seu certificado, conforme a tela abaixo:

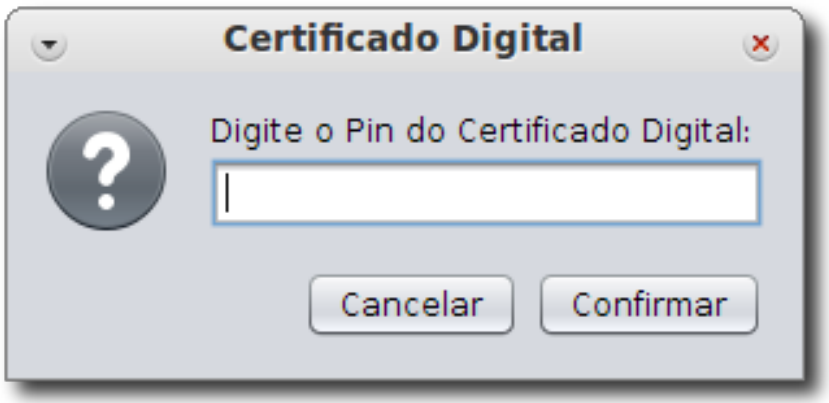

Figura 3.2. Solicitação de Pin do Certificado

O componente exibirá uma tela com os certificados disponíveis, a data inicial de validade, a data final de validade e o emissor deste certificado, fornecendo ao usuário a possibilidade de escolher qual certificado deseja-se utilizar.

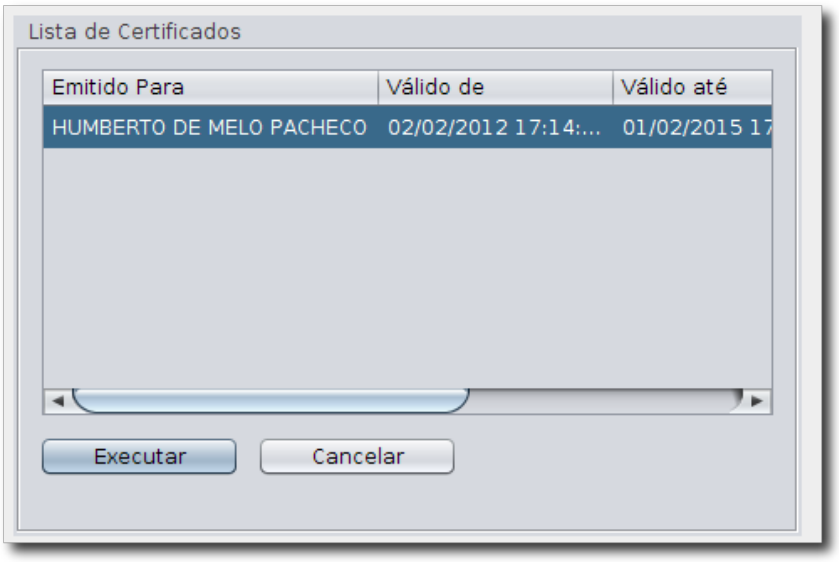

Figura 3.3. Lista com certificado de usuário

3.10. Transformação de um Panel para Dialog

Para transformar a tela que exibe a lista de certificados de um Panel para Dialog, basta realizar a modificação na tag que efetua o carregamento do componente, conforme mostrado abaixo.

```
code="br.gov.frameworkdemoiselle.certificate.applet.view.JPanelApplet"
```

Para:

```
code="br.gov.frameworkdemoiselle.certificate.applet.view.JDialogApplet"
```

3.11. Integração com o componente Demoiselle Signer

A seguir temos um exemplo de implementação utilizando o componente demoiselle-signer, onde o componente utiliza o certificado recebido pelo applet e efetua a assinatura digital de um arquivo.

```
package br.gov.frameworkdemoiselle;

import br.gov.frameworkdemoiselle.certificate.applet.action.AbstractAppletExecute;
import br.gov.frameworkdemoiselle.certificate.applet.certificate.ICPBrasilCertificate;
import br.gov.frameworkdemoiselle.certificate.signer.factory.PKCS7Factory;
import br.gov.frameworkdemoiselle.certificate.signer.pkcs7.PKCS7Signer;
import br.gov.frameworkdemoiselle.certificate.signer.pkcs7.attribute.FileName;
import br.gov.frameworkdemoiselle.certificate.signer.pkcs7.attribute.SigningCertificate;
import br.gov.frameworkdemoiselle.certificate.signer.pkcs7.bc.policies.ADRBCMS_2_1;
import java.applet.Applet;
```

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.security.AuthProvider;
import java.security.KeyStore;
import java.security.KeyStoreException;
import java.security.NoSuchAlgorithmException;
import java.security.PrivateKey;
import java.security.UnrecoverableKeyException;
import java.security.cert.X509Certificate;
import javax.security.auth.login.LoginException;
import javax.swing.JOptionPane;

public class App extends AbstractAppletExecute {

    @Override
    public void execute(KeyStore keystore, String alias, Applet applet) {
        try {
            System.out.println("App.execute()");

            ICPBrasilCertificate certificado = super
                .getICPBrasilCertificate(keystore, alias, false);
            super.setFormField(applet, "mainForm", "cpf", certificado.getCpf());
            super.setFormField(applet, "mainForm", "nome", certificado.getNome());
         super.setFormField(applet, "mainForm", "nascimento", certificado.getDataNascimento());
            super.setFormField(applet, "mainForm", "email", certificado.getEmail());

            /* Carregando o conteudo a ser assinado */

        String documento = AbstractAppletExecute.getFormField(applet, "mainForm", "documento");

            if (documento.length() == 0) {
                JOptionPane.showMessageDialog(applet,
                        "Por favor, escolha um documento para assinar",
                        "Error", JOptionPane.ERROR_MESSAGE);
                return;
            }

            String user_home = System.getProperty("user.home");
            String path = new File(documento).getAbsolutePath();
            byte[] content = readContent(path);

            /* Parametrizando o objeto signer */
            PKCS7Signer signer = PKCS7Factory.getInstance().factoryDefault();
            signer.setCertificates(keystore.getCertificateChain(alias));
            signer.setPrivateKey((PrivateKey) keystore.getKey(alias, null));

            /*
             * Para a politica 1.1, o algoritmo pode ser especificado como SHA-1 ou SHA-256
            * signer.setAlgorithm(SignerAlgorithmEnum.SHA256withRSA); Especificando a politica
 a ser utilizada
             */
            signer.setSignaturePolicy(new ADRBCMS_2_1());
            signer.setAttached(true);

            /* Realiza a assinatura do conteudo */
            System.out.println("Efetuando a  assinatura do conteudo");
```

```
            byte[] signed = signer.signer(content);

            /* Grava o conteudo assinado no disco */
            writeContent(signed, documento.concat(".p7s"));

            /* Valida o conteudo */
            System.out.println("Efetuando a validacao da assinatura.");
            boolean checked = signer.check(content, signed);

            boolean checked = signer.check(content, signed);
            if (checked) {
                logger.info("A assinatura # v#lida.");
                JOptionPane.showMessageDialog(applet,
                        "O arquivo foi assinado e validado com sucesso.",
                        "Mensagem", JOptionPane.INFORMATION_MESSAGE);
            } else {
                System.out.println("A assinatura n#o # v#lida!");
            }

        } catch (KeyStoreException e) {
            JOptionPane.showMessageDialog(applet, e.getMessage(), "Error",
                    JOptionPane.ERROR_MESSAGE);
        } catch (NoSuchAlgorithmException e) {
            JOptionPane.showMessageDialog(applet, e.getMessage(), "Error",
                    JOptionPane.ERROR_MESSAGE);
        } catch (UnrecoverableKeyException e) {
            JOptionPane.showMessageDialog(applet, e.getMessage(), "Error",
                    JOptionPane.ERROR_MESSAGE);
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            System.out.println("Efetuando logout no provider.");
            AuthProvider ap = null;

            if (keystore != null) {
                ap = (AuthProvider) keystore.getProvider();
            }

            if (ap != null) {
                try {
                    ap.logout();
                } catch (LoginException e) {
                    e.printStackTrace();
                }
            }
        }
    }

    @Override
    public void cancel(KeyStore keystore, String alias, Applet applet) {
        System.out.println("App.cancel()");

        try {
            System.out.println(((X509Certificate) keystore.getCertificate(alias)).toString());
        } catch (KeyStoreException e) {
            e.printStackTrace();
        }

        /* Seu codigo customizado aqui... */
```

```
    }

    private byte[] readContent(String arquivo) {
        byte[] result = null;
        try {
            File file = new File(arquivo);
            FileInputStream is = new FileInputStream(file);
            result = new byte[(int) file.length()];
            is.read(result);
            is.close();

        } catch (IOException e) {
            e.printStackTrace();
        }
        return result;
    }

    private void writeContent(byte[] conteudo, String arquivo) {

        try {
            File file = new File(arquivo);
            FileOutputStream os = new FileOutputStream(file);
            os.write(conteudo);
            os.flush();
            os.close();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }

    }
}
```

3.12. Customização

Para modificar no rótulo do botão selecionar modifique a propriedade label.dialog.button.select registrando-a como parâmetro da applet:

```
<param name="label.dialog.button.select" value="Escolher" />
```

Para redimencionar os botões Carregar e Cancelar inclua os seguintes parametros:

```
<param name="config.dialog.button-run.height" value="45" />
<param name="config.dialog.button-cancel.height" value="45" />
```

Segue código completo:

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
```

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/
loose.dtd">
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Projeto Exemplo</title>
        <style type="text/css">
            table.padrao {
                background-color: #eeeeee;
            }

            td.padrao {
                font-family: "Arial";
                font-size: 12px;
            }

            input.padrao {
                font-family: "Arial";
                font-size: 14px;
                background-color: #F8F8F8;
                border: 1px solid #E8E8E8;
            }
        </style>
    </head>

    <script type="text/javascript">
        function foo() {
            alert("foo() method called!");
        }
    </script>

    <body>
        <form id="mainForm" name="mainForm" method="post" action="FileUploadServlet">
            <table align="center" border="0" cellspacing="5" cellpadding="1" class="padrao">
                <tr>

   <td   align="center"><applet   codebase="http://localhost:8080/demoiselle-certificate-applet-
customizada-web/"       code="br.gov.frameworkdemoiselle.certificate.applet.view.JPanelApplet"
 width=470 height=310 MAYSCRIPT
                        archive="demoiselle-certificate-applet-customizada-1.0.0-assinado.jar,
                        demoiselle-certificate-applet-1.0.11-assinado.jar,
                        demoiselle-certificate-core-1.0.11-assinado.jar,
                        demoiselle-certificate-criptography-1.0.11-assinado.jar,
                        demoiselle-certificate-signer-1.0.11-assinado.jar,
                        demoiselle-certificate-ca-icpbrasil-1.0.11-assinado.jar,
                        demoiselle-certificate-ca-icpbrasil-homologacao-1.0.11-assinado.jar,
                        bcmail-jdk15-1.45-assinado.jar,
                        bcprov-jdk15-1.45-assinado.jar,
                        log4j-1.2.15-assinado.jar,
                        slf4j-api-1.6.1-assinado.jar",
                        plugin-1.0-assinado.jar>

                    <param name="factory.applet.action" value="br.gov.frameworkdemoiselle.App" /
>
                            <param name="applet.javascript.postaction.failure" value="foo" />

                            <param name="label.dialog.button.run" value="Escolher" />
                            <param name="config.dialog.button-run.height" value="45" />
                            <param name="config.dialog.button-cancel.height" value="45" />
```

```
                          <param name="config.dialog.table.certificates.x" value="15" />
                          <param name="config.dialog.table.certificates.y" value= "30"/>
                         <param name="config.dialog.table.certificates.width" value="440" />
                        <param name="config.dialog.table.certificates.height" value="210" />

                      </applet>
                  <td>
              </tr>
              <tr>
                  <td>
                           <table align="center" border="0" cellspacing="0" cellpadding="5"
class="padrao">
                          <tr>
                              <td class="padrao">CPF:</td>
                                   <td><input class="padrao" type="text" name="cpf" value=""
size="11"></td>
                          </tr>
                          <tr>
                              <td class="padrao">Nome:</td>
                                          <td><input class="padrao" type="text" name="nome"
value="" size="30"></td>
                          </tr>
                          <tr>
                              <td class="padrao">Email:</td>
                                         <td><input class="padrao" type="text" name="email"
value="" size="30"></td>
                          </tr>
                          <tr>
                              <td class="padrao">Nascimento:</td>
                                    <td><input class="padrao" type="text" name="nascimento"
value="" size="8"></td>
                          </tr>
                      </table>
              </tr>
          </table>
      </form>
   </body>
</html>
```

Abaixo temos a tela com as customizações aplicadas.

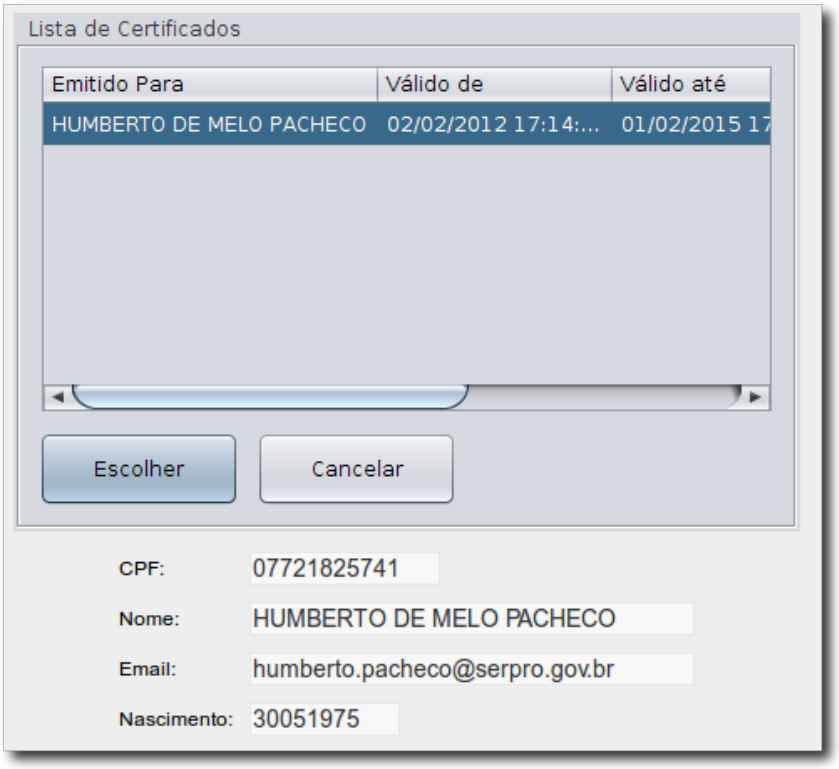

Figura 3.4. Tela com as customizações aplicadas

Assinatura de Jars

4.1. Introdução

O modelo de segurança da plataforma Java é centrado sobre o conceito de sandbox (caixa de areia), no qual um código remoto como um applet por padrão não é confiável e portanto não pode ter acesso ilimitado ao Sistema Operacional. A sandbox garante que mesmo que um usuário carregue um applet malicioso, este não poderá danificar a máquina local executando, por exemplo, o comando de remoção de um arquivo do seu disco local.

Entretanto existem applets confiáveis que necessitam rodar fora da sandbox, como é o caso do componente Demoiselle Applet que necessita acessar certificados A3 armazenados em dispositivos como tokens usb. Nestes casos podemos criar uma permissão para acesso fora da sandbox através da assinatura digital do componente.

Neste parte da documentação é apresentada uma forma para assinar componentes utilizando as ferramentas *keytool* e *jarsigner* nativas na instalação do JDK. Vale lembrar que os certificados criados pelo *keytool* serão úuteis durante as fases de desenvolvimento e teste, entretanto para uso em ambiente de produção deve-se utilizar certificados emitidos por Autoridades Certificadoras (AC) reconhecidas.

4.2. Criando um certificado

Primeiramente criaremos o keystore que armazenará o certificado digital. A ferramenta *keytool* será utilizada para criação simultânea do keystore e do certificado digital que identificaremos pelo alias *applet_alias*.

```
keytool -genkey -alias applet_alias -keyalg RSA -keypass changeit -storepass changeit -
keystore applet_keystore.jks
```

Importante

Utilizaremos a mesma senha *changeit* para acessar o keystore e o certificado contido nele.

Na sequência serão solicitadas algumas informações do certificado:

```
What is your first and last name?
  [Unknown]:  Framework Demoiselle
What is the name of your organizational unit?
  [Unknown]:  Demoiselle
What is the name of your organization?
  [Unknown]:  Demoiselle
What is the name of your City or Locality?
  [Unknown]:  Rio de Janeiro
What is the name of your State or Province?
  [Unknown]:  RJ
What is the two-letter country code for this unit?
  [Unknown]:  BR
Is CN=Framework Demoiselle, OU=Demoiselle, O=Demoiselle, L=Rio de Janeiro, ST=RJ, C=BR correct?
  [no]:  yes
```

Será criado o keystore JKS de nome *applet_keystore.jks* que contém um certificado auto assinado seu par de chaves identificado pelo alias *applet_alias.*

4.3. Assinando um jar com certificado auto-assinado

Neste momento a ferramenta *jarsigner* será utilizada para assinar todos os jars da aplicação. Portanto será necessário informar a localização do keystore, o nome do jar assinado, o nome do jar original e o alias do certificado:

```
jarsigner -keystore applet_keystore.jks -signedjar meujar-assinado.jar meujar.jar applet_alias
```

Importante

Note que o jar assinado (meujar-assinado.jar) define o nome do arquivo jar que será criado, diferente do nome original do jar (meujar.jar).

Dentro do jar, na pasta *META-INF*, foram inseridos os aquivos *APPLET_A.RSA*, *APPLET_A.SF* e *MANIFEST.MF*, que possuem informações como o algoritmo de criptografia utilizado e a chave pública do certificado.

Para verificar a assinatura do jar utilize o comando *jarsigner* conforme abaixo:

```
jarsigner -verify -keystore applet_keystore.jks meujar-assinado.jar
```

4.4. Assinando um artefato com Token ou SmartCard

O certificado auto-assinado nos permite efetuar a homologação necessária ao funcionamento dos componentes, mas para o uso em produção o recomendado é utilizar um certificado reconhecido pela CA, normalmente armazenado em um Token ou SmartCard.

Primeiramente, precisamos criar um arquivo de configuração que determinará a localização do driver do Token ou Smartcard. Crie um arquivo de nome *token.config* na sua pasta de trabalho.

A seguir temos o conteúdo que deve ser colocado neste arquivo. Os campos *name* e *description* podem conter um texto de sua escolha, mas o campo *library* deve apontar para a localização do driver do dispositivo.

```
name = Provedor
description = Token Pro Azul
library = /usr/lib/libeTPkcs11.so
```

O Token ou Smartcard pode conter um ou mais certificados, cada um deles associado a um apelido. Antes de iniciarmos a assinatura, precisamos descobrir qual o apelido do certificado que será utilizado para a assinatura. Para isso, vamos executar a linha de comando a seguir. O campo *PASSWORD* deve ser substituído pelo seu Pin.

```
keytool -keystore NONE -storetype PKCS11 -providerClass sun.security.pkcs11.SunPKCS11 -
providerArg token.config -storepass PASSWORD -list
```

Após a execução deste comando, será exibido a lista de apelidos. abaixo temos um exemplo do resultado esperado.

```
(eTCAPI) HUMBERTO DE MELO PACHECO's ICP-Brasil ID
```

Podemos agora proceder para a assinatura do artefato utilizando a linha de comando abaixo. Os parâmetros a serem alterados são os seguintes.

- *PASSWORD*, o Pin do Token ou SmartCard.
- *DSANAME*, o nome do arquivo que contém as assinaturas das classes. Este atributo é opcional.
- *JARFILESIGNED*, o nome do arquivo gerado após a assinatura.
- *JARFILE*, o nome do arquivo a ser assinado.
- *ALIAS*, o apelido do certificado a ser utilizado, obtido no passo anterior.

```
jarsigner -keystore NONE -storetype PKCS11 -providerClass sun.security.pkcs11.SunPKCS11 -
providerArg token.config -storepass PASSWORD -sigfile DSANAME -signedjar JARFILESIGNED -verbose
 JARFILE "ALIAS"
```

Novas regras de segurança de aplicações JAVA assinadas

5.1. Introdução

texto...

5.2. O Atributo Permissions

O atributo *Permissions* é usado para verificar se o nível de permissões solicitada por um RIA quando ele é executado coincide com o nível de permissões que foi definido quando foi criado o arquivo JAR. Use este atributo para ajudar a impedir alguém de reimplantar um aplicativo que está assinado com o seu certificado e executá-lo em um nível de privilégio diferente. Defina esse atributo para um dos seguintes valores:

Tabela 5.1. Valores para o atributo Permissions

sandbox	Indica que o RIA é executado no domínio de segurança e não requer permissões adicionais.
all-permissions	Indica que o RIA requer acesso aos recursos do sistema do usuário.

Se o nível de segurança no painel de controle do JAVA estiver definido como muito alta ou alta, então o atributo de permissões será necessário no arquivo JAR principal do RIA. Se o atributo não estiver presente, então o RIA será bloqueado. Para o nível de segurança médio, se o atributo de permissões não estiver presente, então será emitido um aviso de segurança com um aviso amarelo sobre o atributo ausente, e o nível de permissões solicitado pelo RIA será usado.

5.3. O Atributo Codebase

O atributo *Codebase* é usado para restringir a base de código do arquivo JAR para domínios específicos. Use este atributo para impedir que alguém reimplante seu aplicativo em um outro site para fins maliciosos.

Importante

Se o atributo *Codebase* não especificar um servidor seguro, como HTTPS, existe o risco de seu código ser reaproveitado em esquemas de ataque Man-in-the-Middle (MITM).

Defina este atributo para o nome de domínio ou endereço IP onde se encontra o arquivo JAR do aplicativo. Um número de porta também pode ser incluído. Para vários locais, separe os valores com um espaço. Um asterisco (*) pode ser usado como um curinga apenas no início do nome do domínio.

O valor do atributo *Codebase* deve coincidir com a localização do arquivo JAR do RIA. Caso contrário, um erro é mostrado e o RIA será bloqueado. Se o atributo não estiver presente, um aviso é escrito no console do JAVA e a base de código especificada para a tag applet ou o arquivo JNLP é usado.

5.4. O Atributo Application-Name

O atributo *Application-Name* é usado em avisos de segurança para fornecer um título para a sua RIA assinada. Recomenda-se o uso deste atributo para ajudar os usuários a tomar a decisão de confiar e executar o RIA. O valor pode ser qualquer cadeia de caracteres válida, por exemplo:

Application-Name: Hello World

Se o atributo de nome do aplicativo não estiver presente no manifesto do arquivo JAR, um aviso é escrito no console JAVA e o valor do atributo Main-Class é usado. Se nenhum atributo estiver presente no manifesto, nenhum título será mostrado no aviso de segurança. Os títulos não são mostrados em RIAs não assinados.

5.5. O Atributo Application-Library-Allowable-Codebase

O atributo *Application-Library-Allowable-Codebase* identifica os locais onde é esperado ser encontrado o seu RIA assinado. Este atributo é usado para determinar o que está listado no campo de localização para o aviso de segurança que é exibido aos usuários quando o arquivo JAR do seu RIA está em um local diferente do arquivo JNLP ou página HTML que inicia seu RIA. Se os arquivos não estiverem nos locais identificados, o RIA será bloqueado. Defina este atributo para os domínios onde se situam o arquivo JAR, arquivo JNLP e página HTML. Para especificar mais de um domínio, separe-os por um espaço, por exemplo:

*Application-Library-Allowable-Codebase: https://host.example.com *.samplehost.com/apps*

Se o atributo estiver presente e coincidir com o local onde o RIA foi iniciado, então um único host é listado no campo de localização do aviso e a opção de ocultar avisos futuros é fornecida. Se este atributo estiver presente, e os arquivos são acessados a partir de um local não incluído no atributo, então o RIA será bloqueado. Se este atributo não estiver presente, então vários hosts que correspondem aos locais de arquivo JAR e o arquivo JNLP ou página HTML serão listados no campo de localização do aviso. Quando múltiplos hosts são mostrados, não é dado ao usuário a opção de ocultar avisos futuros. Recomenda-se o uso deste atributo para que os arquivos do RIA sejam acessados somente a partir de locais conhecidos.

Este atributo não é necessário se o arquivo JAR do RIA estiver no mesmo local que o arquivo JNLP ou página HTML que inicia o RIA.

5.6. O Atributo Caller-Allowable-Codebase

O atributo *Caller-Allowable-Codebase* é usado para identificar os domínios nos quais código JavaScript pode fazer chamadas para seu RIA sem avisos de segurança. Defina este atributo para o domínio que hospeda o código JavaScript. Se for feita uma chamada do código JavaScript que não está localizado em um domínio especificado pelo atributo *Caller-Allowable-Codebase*, a chamada é bloqueada. Para especificar mais de um domínio, separe-os por um espaço, por exemplo:

Caller-Allowable-Codebase: host.example.com 127.0.0.1

Se o atributo não estiver presente, chamadas de código JavaScript para seu RIA serão bloqueadas quando o nível estiver definido como alto, que é o padrão, ou muito alto. Quando o nível de segurança é definido como médio, que não é recomendado, é mostrado um aviso de segurança e usuários, então, tem a opção para permitir a chamada ou bloquear a chamada. Para RIAs não assinados, o código JavaScript que requer acesso para a RIA deve estar no mesmo local que o arquivo JAR principal do seu RIA. Caso contrário, o usuário é solicitado a permitir acesso. Avisos de segurança são mostrados para cada instância do Classloader do applet.

5.7. O Atributo Trusted-Only

O atributo *Trusted-Only* é usado para impedir que as classes não confiáveis ou recursos sejam carregadas em um applet ou aplicativo. Por exemplo:

Trusted-Only: true

Este atributo impede que um aplicativo privilegiado ou applet seja realocado com componentes não confiáveis. Todas as classes e recursos do aplicativo ou da applet devem ser assinados e todas as permissões devem ser solicitadas.

5.8. O Atributo Trusted-Library

O atributo *Trusted-Library* é usado para aplicativos e applets que são projetados para permitir componentes não confiáveis. Nenhuma caixa de diálogo de aviso é mostrada e um aplicativo ou applet pode carregar arquivos JAR que contêm classes não confiáveis ou recursos. Por exemplo:

Trusted-Library: true

Este atributo impede que componentes em um aplicativo privilegiado ou applet seja realocado com componentes não confiáveis. Todas as classes e recursos em um arquivo JAR contendo este atributo devem ser assinados e todas as permissões devem ser solicitadas.

5.9. Incluindo os atributos no manifesto

Para incluir os atributos no manifesto, primeiramente precisamos criar um arquivo de texto chamado *manifest-addiction.txt* contendo os parâmetros abaixo. Os valores são apenas para efetuar a configuração inicial, posteriormente o desenvolvedor deve restringir o máximo possível os valores para aumentar o nível de segurança da aplicação.

```
Permissions: all-permissions
Codebase: *
Application-Name: Applet Customizada
Application-Library-Allowable-Codebase: *
Caller-Allowable-Codebase: *
Trusted-Only: true
Trusted-Library: true
```

Em seguida, imediatamente antes da assinatura digital do arquivo, o comando abaixo deve ser executado.

A variável JAVAHOME deve apontar para a versão mais nova do JAVA, na versão 1.7.0 release 51 ou maior.

```
JAVAHOME/bin/jar uvfm jarfile manifest-addition.txt
```

Ao final da execução deste comando os atributos estarão incorporados ao manifesto.

5.10. Tratando os artefatos jar assinados por terceiros

Alguns artefatos JAR podem vir previamente assinados pelo fornecedor, como é o caso das dependências do BouncyCastle. Neste caso, antes de aplicarmos a adição dos parâmetros do manifesto e efetuar a assinatura, precisamos remover os dados de assinatura do fornecedor. Na figura baixo podemos verificar os arquivos contidos no pacote META-INF correspondentes à assinatura.

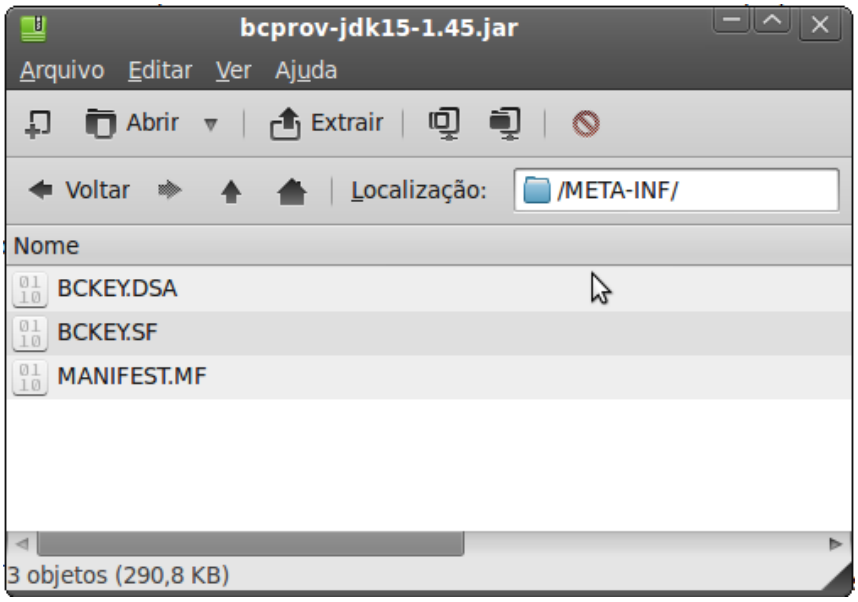

Figura 5.1. Conteúdo do META-INF

Neste caso, para remover os dados da assinatura anterior basta executar o comando abaixo. Após este passo basta executar os procedimentos de adição de atributos e assinatura digital normalmente.

```
zip -d jarfile /META-INF/BCKEY.*
```

O nome “BCKEY” no comando é correspondente ao BouncyCastle. Se o arquivo JAR possuir um nome diferente, o comando deve ser alterado.

5.11. Importação da cadeia de certificados para o JAVA

Mesmo que todos os passos acima sejam executados corretamente, existe a possibilidade do RIA retornar a tela exibida a seguir:

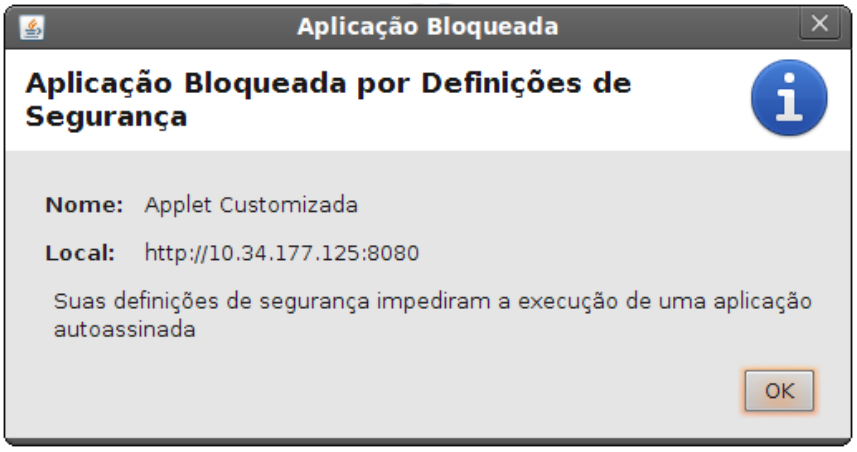

Figura 5.2. Tela de aplicação bloqueada

Esta tela indica que, apesar de todos os atributos estarem corretamente configurados, os artefatos foram assinados por um certificado digital que não é confiado pela máquina virtual do JAVA. Para resolver isto, precisamos executar os procedimentos a seguir no equipamento do usuário.

Primeiramente, precisamos baicar toda a cadeia da CA que emitiu o certificado. SE o certificado foi emitido pelo SERPRO, as cadeias podem ser obtidas no site do *CCD SERPRO* [https://ccd.serpro.gov.br/serproacf/], na opção de menu “Baixar Cadeia”.

Em seguida, abra o painel de controle do JAVA e selecione a aba “Segurança”.

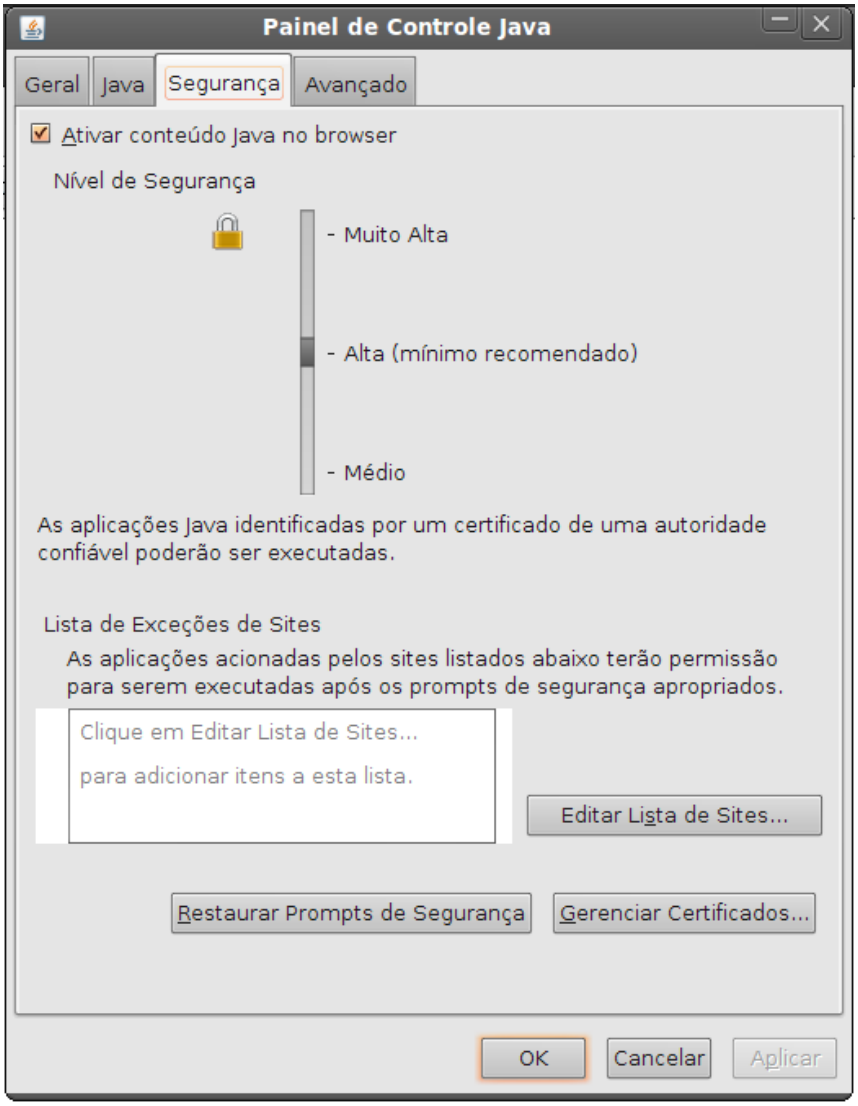

Figura 5.3. Aba "Segurança"

Clique no botão “Gerenciar Certificados” e selecione a opção “CA de Signatário” na combo que aparece no topo da tela.

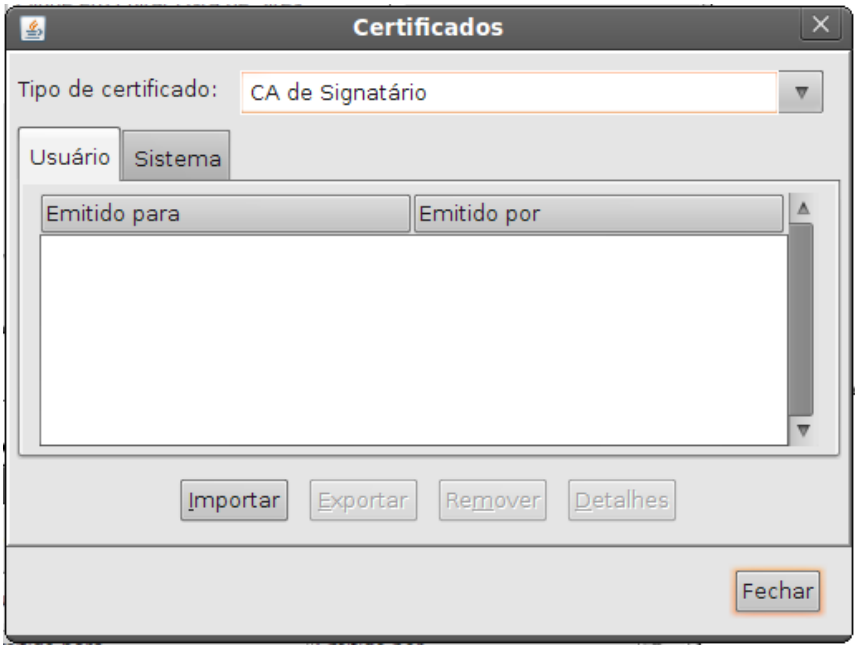

Figura 5.4. CA de Signatário

Clique no botão “Importar” e altere a combo de filtro para “Todos os arquivos”.

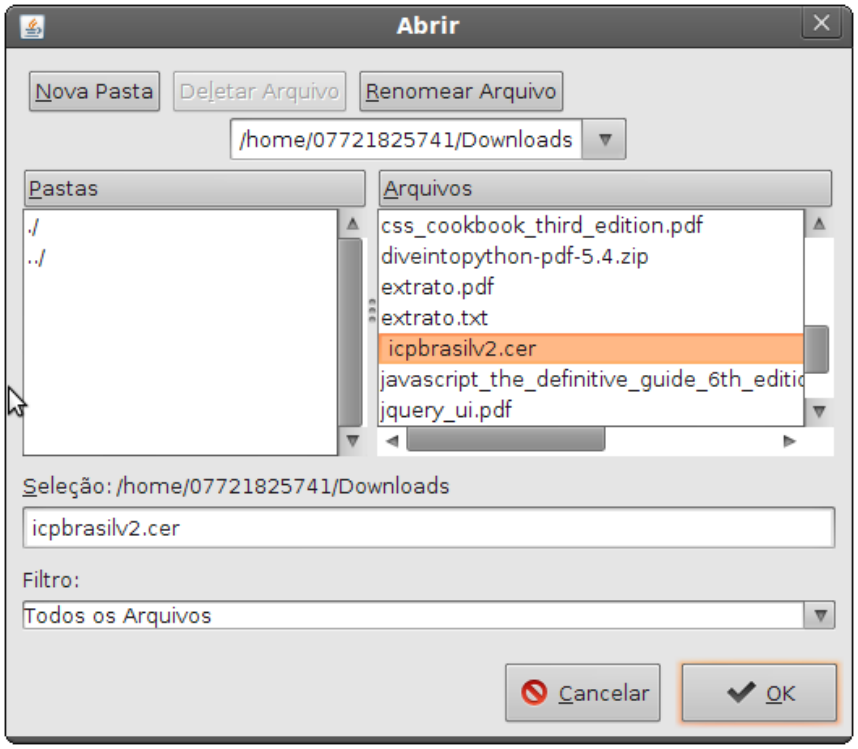

Figura 5.5. Importando os certificados

A importação de cada cadeia deve ser feita separadamente. Após a importação, as ACs devem ser exibidas como na tela a seguir:

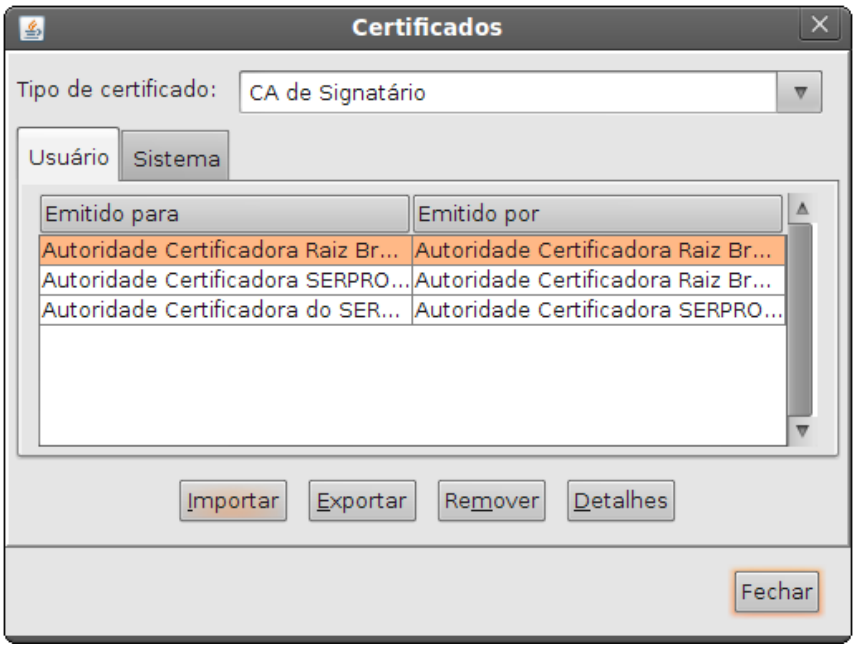

Figura 5.6. Certificados instalados

Parte II. Demoiselle Core

Este componente provê uma API para facilitar o tratamento de Certificados Digitais em aplicações Java. Seus objetivos são o carregamento, validação e obtenção de dados para certificados digitais.

Configuração do Demoiselle Core

6.1. Instalação do componente

Para instalar o componente *Demoiselle Core* na aplicação, basta adicionar a sua dependência no arquivo `pom.xml` do projeto gerenciado pelo Maven:

```
<dependency>
            <groupId>br.gov.frameworkdemoiselle.component</groupId>
            <artifactId>demoiselle-certificate-core</artifactId>
            <version>1.0.11</version>
        </dependency>
```

Funcionalidades relativas ao Certificado

O componente de segurança disponibiliza o CertificateManager que permite manipular objetos de certificado X.509 para extrair informações e validar seu conteúdo. Para trabalhar com o CertificateManager basta instanciá-lo passando o objeto X.509 no construtor. Se não for informado, serão carregados os validadores *CRLValidator* e *PeriodValidator* . A validação ocorre no momento da instanciação do objeto CertificateManager. Segue abaixo a criação do CertificateManager.

```
CertificateManager cm = new CertificateManager(x509);
```

É possível desativar o carregamento dos validadores mudando a instrução para:

```
CertificateManager cm = new CertificateManager(x509, false);
```

Caso seja necessário implementar os próprios validadores de certificado basta mudar a instrução para:

```
/* Neste caso os validadores padrao tambem serao carregados. */
CertificateManager cm = new CertificateManager(x509, validator1, validator2, validatorN);
```

ou

```
/* Neste caso os validadores padrao nao serao carregados. */
CertificateManager cm = new CertificateManager(x509, false, validator1, validator2, validatorN);
```

É possível também criar um CertificateManager e passar um arquivo do tipo PEM que represente um objeto X509Certificate, conforme mostrado abaixo.

```
File certFile = new File("certificado.pem");
CertificateManager cm = new CertificateManager(certFile);
```

Também é possível criar um CertificateManager que carrege um certificado direto de um token.

```
String pinNumer = "pinNumber do token";
CertificateManager cm = new CertificateManager(pinNumber);
```

7.1. O Certificado Digital

7.1.1. Extração de Informações utilizando anotações

Os certificados no formato X.509 podem conter várias informações armazenadas que podem ser obtidas através de um OID (Object Identifier). OID são usados extensivamente em certificados de formato X.509, como por exemplo, para designar algoritmos criptográficos empregados, políticas de certificação e campos de extensão. Cada autoridade certificadora pode definir um conjunto de OID para armazenar suas informações. O componente de segurança implementa extensões de OID para ICP-Brasil e Default.

Para extrair informações basta criar uma classe com os atributos que se deseja preencher com informações do certificado X.509. Cada atributo deve ser anotado com o seu OIDExtension. Para executar a carga das informações basta passar a classe/objeto para o CertificateManager.

```
class Cert {

    @ICPBrasilExtension(type=ICPBrasilExtensionType.CPF)
    private String cpf;

    @ICPBrasilExtension(type=ICPBrasilExtensionType.NOME)
    private String nome;

    @DefaultExtension(type=DefaultExtensionType.CRL_URL)
    private List<String> crlURL;

    public String getCpf() {
        return cpf;
    }

    public String getNome() {
        return nome;
    }

    public List<String> getCrlURL() {
        return crlURL;
    }

}
```

Em seguida basta efetuar o carregamento da classe.

```
CertificateManager cm = new CertificateManager(x509);
Cert cert = cm.load(Cert.class);
```

7.1.1.1. DefaultExtension

Os OIDs default de um certificado que podem ser obtidos por essa anotação são:

- BEFORE_DATE
- AFTER_DATE

- CERTIFICATION_AUTHORITY
- CRL_URL
- SERIAL_NUMBER
- ISSUER_DN
- SUBJECT_DN
- KEY_USAGE
- PATH_LENGTH
- AUTHORITY_KEY_IDENTIFIER
- SUBJECT_KEY_IDENTIFIER

7.1.1.2. ICPBrasilExtension

Os OIDs definidos pela ICP-Brasil que podem ser obtidos são:

- CPF
- CNPJ
- CEI_PESSOA_FISICA
- CEI_PESSOA_JURIDICA
- PIS_PASEP => Ver NIS
- NOME
- NOME_RESPONSAVEL_PESSOA_JURIDICA
- EMAIL
- DATA_NASCIMENTO
- NUMERO_IDENTIDADE
- ORGAO_EXPEDIDOR_IDENTIDADE
- UF_ORGAO_EXPEDIDOR_IDENTIDADE
- NUMERO_TITULO_ELEITOR
- ZONA_TITULO_ELEITOR
- SECAO_TITULO_ELEITOR
- MUNICIPIO_TITULO_ELEITOR
- UF_TITULO_ELEITOR
- NOME_EMPRESARIAL
- TIPO_CERTIFICADO
- NIVEL_CERTIFICADO

7.1.2. Extração de Informações utilizando Classes

Uma outra maneira de obter os valores necessários do certificado é através das classes de apoio fornecidas pelo componente. Caso deseje obter apenas informações, básicas, podemos utilizar a classe `BasicCertificate`.

A seguir temos o exemplo de utilização, onde passamos um certificado para a classe e em seguida obtemos exibimos algumas informações no console.

```
BasicCertificate bc = new BasicCertificate(certificate);
logger.log(Level.INFO, "Nome.....................[{0}]", bc.getNome());
logger.log(Level.INFO, "E-mail...................[{0}]", bc.getEmail());
logger.log(Level.INFO, "Numero de serie..........[{0}]", bc.getSerialNumber());
logger.log(Level.INFO, "Nivel do Certificado....[{0}]", bc.getNivelCertificado());
```

Para obter informações mais específicas de um certificado de um e-CPF, e-CNPJ ou de equipamento, devemos utilizar a classe `CertificateExtra`.

A seguir temos alguns exemplos de de utilização.

O exemplo a seguir recupera o CPF e o número RIC de um certificado digital do tipo e-CPF.

```
CertificateExtra ce = new CertificateExtra(certificate);
logger.log(Level.INFO, "CPF.....................[{0}]", ce.getOID_2_16_76_1_3_1().getCPF());
logger.log(Level.INFO,                                                    "RIC.....................
[{0}]", ce.getOID_2_16_76_1_3_9().getRegistroDeIdentidadeCivil());
```

O exemplo a seguir recupera o CNPJ de um certificado digital do tipo e-CNPJ.

```
CertificateExtra ce = new CertificateExtra(certificate);
logger.log(Level.INFO, "CNPJ.....................[{0}]", ce.getOID_2_16_76_1_3_3().getCNPJ());
```

O exemplo a seguir recupera o nome do responsável de um certificado digital do tipo Equipamento.

```
CertificateExtra ce = new CertificateExtra(certificate);
logger.log(Level.INFO, "Nome.....................[{0}]", ce.getOID_2_16_76_1_3_2().getNome());
```

7.2. Validadores

7.2.1. CRLValidator

O CRLValidator verifica se o certificado está na lista de certificados revogados da autoridade certificadora. Cada certificado pode conter uma ou mais links para os arquivos de CRL. O mecanismo de obtenção dos arquivos de crl é implementado pelos Repositórios de CRL.

7.2.2. PeriodValidator

Verifica a data de validade do certificado.

7.2.3. CAValidator

Verifica se o certificado foi emitido por uma Autoridade Certificadora confiável. O componente consegue executar essa validação pois existe internamente um arquivo com os certificados digitais das autoridades válidas.

Importante

Este validador foi substituído pelo contido no componente *demoiselle-certificate-signer* para realização de assinaturas digitais, pois este já efetua as validações baseado nas novas políticas da ICP-Brasil.

7.3. Repositório de CRL

O Repositório de CRL disponibiliza uma lista de ICPBR_CRL (CRLs padrão ICP Brasil). Esta lista é obtida pelos arquivos de crl referentes a um certificado digital. A obtenção e armazenamentos dos arquivos de crl são implementados de dois modos: Online ou Offine.

7.3.1. Repositório Online

O Repositório Online nao utiliza um diretório para armazenamento dos arquivos crl, efetuando diretamente a consulta no endereço web da crl.

7.3.2. Repositório Offline

O Repositório offline utiliza um diretório onde é mantida uma lista de crl e um arquivo de índice. O arquivos de índice identificam a url do certificado e o nome do arquivos armazenado no file system, como no exemplo abaixo:

```
73bc162ad833c4da45ea60ac8ac016cc=https\://thor.serpro.gov.br/LCR/LCRPRA1.crl
75bc176ad833c4da05ea70ac8ac016ca=http\://ccd.serpro.gov.br/lcr/ACPRv1.crl
43bc194ad833c4da95ea90ac8ac016cb=http\://ccd2.serpro.gov.br/lcr/ACPRv2.crl
```

O diretório e o nome do arquivo de índice devem ser configurados através de chaves informadas em variáveis de ambiente:

- *security.certificate.repository.crl.path*
- *security.certificate.repository.crl.index*

Por padrão essas chaves são inicializadas na seguintes forma:

- *security.certificate.repository.crl.path=/tmp/crls*
- *security.certificate.repository.crl.index=.crl_index*

Programaticamente é possível modificar as propriedades por meio da classe `Configuration.`

```
Configuration config = Configuration.getInstance();
config.setCrlIndex(".crl_index");
```

```
config.setCrlPath("/tmp/crls/");
```

Quando o arquivo de crl se encontra com data vencida ou não existe o arquivo no diretório, o repositório Offline realiza o download do arquivo de crl e o armazena no diretório de crl.

7.3.3. Configuração

Para modificar o modo de uso do repositório (online ou offline) deve ser configurada a chave *security.certificate.repository.online.*

O valor padrão é true, mas é possível modificar programaticamente conforme abaixo.

```
Configuration config = Configuration.getInstance();
config.setOnline(false);
```

Funcionalidades relativas ao Keystore

8.1. Introdução

A RSA Laboratories definiu algumas especificações de uso de criptografia e assinatura digital conhecidas pelo prefixo PKCS. Duas delas estão relacionadas a keystore. São elas PKCS#11 e PKCS#12.

PKCS#11 define uma API genérica para acesso a hardware criptográfico, comumente chamados de Token ou Smartcard.

PKCS#12 define um formato de arquivo usado para guardar chaves privadas acompanhadas de seus certificados digitais e protegidos por meio de senha.

A linguagem Java suporta a utilização desses formatos e com isso define o que chamamos de KeyStore. Um KeyStore é usado para armazenar um ou mais certificados digitais e também par de chaves, com isso é possível utilizar os padrões da RSA através da mesma interface. A partir de um objeto KeyStore instanciado é possível navegar pelos certificados digitais contidos no KeyStore por meio dos apelidos destes certificados.

O componente visa facilitar o uso de KeyStore em diversos ambientes, seja PKCS#11 ou PKCS#12. A maneira como se carrega um KeyStore PKCS#11, ou seja, mantidos em hardware, difere quando trabalhamos com sistemas operacionais diferentes e até mesmo versões de JVM.

No ambiente Windows, é possível utilizar a API padrão do sistema operacional de carregamento de KeyStore PKCS#11, evitando ter de conhecer a marca e o driver do fabricante, mas para isso precisamos também saber a versão da JVM instalada. Isso é necessário porque na versão 1.6 a implementação JCE já comporta o tratamento nativo na plataforma e na versão 1.5 ou inferior é necessário utilizar uma biblioteca para trabalhar com a API nativa do Windows.

Em ambiente Unix-like é possível carregar um KeyStore PKCS#11 a partir de um driver específico, mas é preciso saber o fabricante e o caminho do driver no sistema operacional. Para carregamento de KeyStore formato PKCS#12, ou seja, em arquivo, o processo de carregamento é o mesmo para os diversos sistemas operacionais.

As funcionalidades do componente estão acessíveis por meio da fábrica *br.gov.frameworkdemoiselle.certificate.keystore.loader.factory.KeyStoreLoaderFactory* de objetos do tipo *br.gov.frameworkdemoiselle.certificate.keystore.loader.KeyStoreLoader.*

O uso da fábrica é importante, mas não é obrigatório. A importância dela se deve à funcionalidade de descobrir qual a melhor implementação para o carregamento de KeyStore baseando-se em configurações. Utilizando a fábrica não é necessário escrever códigos específicos para um determinado sistema operacional, pois a fábrica identifica qual o sistema operacional e a versão da JVM para fabricar a melhor implementação.

Exemplo de uso da fábrica de objetos KeyStoreLoader

```
KeyStoreLoader keyStoreLoader = KeyStoreLoaderFactory.factoryKeyStoreLoader();
```

Exemplo de uso da fábrica de objetos KeyStoreLoader para KeyStore PKCS#12

```
KeyStoreLoader keyStoreLoader = KeyStoreLoaderFactory.factoryKeyStoreLoader(new File("/usr/
keystore.p12"));
```

8.2. Carregamento de KeyStore PKCS#12

Para carregar um KeyStore a partir de uma arquivo no formato PKCS#12 basta utilizar a classe *br.gov.frameworkdemoiselle.certificate.keystore.loader.implementation.FileSystemKeyStoreLoader*.

Abaixo temos exemplos de uso.

```
KeyStore keyStore = (new FileSystemKeyStoreLoader(new File("/usr/keystore.p12"))).getKeyStore("password");
```

```
KeyStore keyStore = KeyStoreLoaderFactory.factoryKeyStoreLoader(new File("/usr/keystore.p12")).getKeyStore("password");
```

8.3. Carregamento de KeyStore PKCS#11 em ambiente Linux

Para carregar um KeyStore PKCS#11 basta utilizar a classe *br.gov.frameworkdemoiselle.certificate.keystore.loader.implementation.DriverKeyStoreLoader*.

Para configuração de drivers favor acessar a área de Configuração do componente em *Seção 8.5, "Lista de Drivers"*.

Abaixo temos exemplos de uso.

```
KeyStore keyStore = (new DriverKeyStoreLoader()).getKeyStore("PIN NUMBER");
```

```
KeyStore keyStore = KeyStoreLoaderFactory.factoryKeyStoreLoader().getKeyStore("PIN NUMBER");
```

Caso se queira instanciar um KeyStore a partir de um driver específico que não esteja na lista de driver configurada, é possível informar o driver como parâmetro para a classe, veja o exemplo:

```
KeyStore keyStore = (new DriverKeyStoreLoader()).getKeyStore("PIN NUMBER", "Pronova", "/usr/lib/libepsng_p11.so");
```

```
KeyStore keyStore = (new DriverKeyStoreLoader()).getKeyStore("PIN NUMBER", "/usr/lib/libepsng_p11.so");
```

Importante

Este código também funciona em ambiente Windows, bastando especificar o driver correto a ser utilizado.

8.4. Carregamento de KeyStore PKCS#11 em ambiente Windows

Para carregar um KeyStore utilizando a API nativa do Windows basta utilizar a classe *br.gov.frameworkdemoiselle.certificate.keystore.loader.implementation.MSKeyStoreLoader.*

Abaixo temos exemplos de uso.

```
KeyStore keyStore = (new MSKeyStoreLoader()).getKeyStore(null);
```

```
KeyStore keyStore = KeyStoreLoaderFactory.factoryKeyStoreLoader().getKeyStore(null);
```

Importante

Este recurso só funciona em JVM 1.6 ou superior. Caso deseje executar em um ambiente com o Java mais antigo, desabilite a camada MSCAPI e faça o acesso diretamente pelo driver. Para saber como proceder, consulte *Seção 8.9, "Desabilitar a camada de acesso SunMSCAPI"*.

8.5. Lista de Drivers

Uma das configurações mais importantes desse componente é a lista de drivers PKCS#11 e seus respectivos arquivos. O componente já possui uma lista pré-estabelecida conforme a tabela a seguir.

Tabela 8.1. Drivers pré definidos

Path do Driver
WINDOWS_HOME/system32/ngp11v211.dll
WINDOWS_HOME/system32/aetpkss1.dll
WINDOWS_HOME/system32/gclib.dll
WINDOWS_HOME/system32/pk2priv.dll
WINDOWS_HOME/system32/w32pk2ig.dll
WINDOWS_HOME/system32/eTPkcs11.dll
WINDOWS_HOME/system32/acospkcs11.dll
WINDOWS_HOME/system32/dkck201.dll
WINDOWS_HOME/system32/dkck232.dll
WINDOWS_HOME/system32/cryptoki22.dll
WINDOWS_HOME/system32/acpkcs.dll
WINDOWS_HOME/system32/slbck.dll
WINDOWS_HOME/system32/cmP11.dll
WINDOWS_HOME/system32/WDPKCS.dll

Path do Driver
WINDOWS_HOME/System32/Watchdata/Watchdata Brazil CSP v1.0/WDPKCS.dll
/Arquivos de programas/Gemplus/GemSafe Libraries/BIN/gclib.dll
/Program Files/Gemplus/GemSafe Libraries/BIN/gclib.dll
/usr/lib/libaetpkss.so
/usr/lib/libgpkcs11.so
/usr/lib/libgpkcs11.so.2
/usr/lib/libepsng_p11.so
/usr/lib/libepsng_p11.so.1
/usr/local/ngsrv/libepsng_p11.so.1
/usr/lib/libeTPkcs11.so
/usr/lib/libeToken.so
/usr/lib/libeToken.so.4
/usr/lib/libcmP11.so
/usr/lib/libwdpkcs.so
/usr/local/lib64/libwdpkcs.so
/usr/local/lib/libwdpkcs.so
/usr/lib/watchdata/ICP/lib/libwdpkcs_icp.so
/usr/lib/watchdata/lib/libwdpkcs.so
/opt/watchdata/lib64/libwdpkcs.so
/usr/lib/opensc-pkcs11.so
/usr/lib/pkcs11/opensc-pkcs11.so
/usr/lib/libwdpkcs.dylib
/usr/local/lib/libwdpkcs.dylib
/usr/local/ngsrv/libepsng_p11.so.1.2.2
/usr/local/lib/libetpkcs11.dylib
/usr/local/lib/libaetpkss.dylib

8.6. Configuração de Token / SmartCard em tempo de execução

É possível, porém, adicionar mais drivers em tempo de execução. Para isso é necessário trabalhar com a classe `br.gov.frameworkdemoiselle.certificate.keystore.loader.configuration.Configuration`.

```
Configuration.getInstance().addDriver("Nome do Driver", "Path do Driver");
```

Este código irá procurar pelo driver e caso ele exista, ou seja, o path do arquivo for válido, o driver será colocado a disposição para futuro uso pelas implementações de carregamento de KeyStore.

Caso seja necessário verificar os drivers já informados, podemos usar a seguinte construção:

```
Map<String, String> drivers = Configuration.getInstance().getDrivers();
```

8.7. Configuração de Token / SmartCard por variáveis de ambiente

Em algumas ocasiões pode ser inviável utilizar o Configuration para adicionar um driver diretamente no código. Neste caso, A API do Java permite definir um arquivo de configuração onde pode-se informar o nome do driver e seus parâmetros. O componente permite a definição desse arquivo por meio de váriáveis de ambiente ou variáveis da JVM.

Abaixo temos o exemplo de como declarar essas configurações.

Tabela 8.2. Configurações do PKCS#11

Ambiente	Variável de Ambiente	Variável JVM
Linux	export PKCS11_CONFIG_FILE=/usr/pkcs11/drivers.config	-DPKCS11_CONFIG_FILE=/usr/pkcs11/drivers.config
Windows	set PKCS11_CONFIG_FILE=c:/pkcs11/drivers.config	-DPKCS11_CONFIG_FILE=c:/pkcs11/drivers.config

A estrutura deste arquivo pode ser encontrada *aqui* [http://java.sun.com/j2se/1.5.0/docs/guide/security/p11guide.html] para Java 1.5, *aqui* [http://java.sun.com/javase/6/docs/technotes/guides/security/p11guide.html] para Java 1.6 ou *aqui* [http://docs.oracle.com/javase/7/docs/technotes/guides/security/p11guide.html] para Java 1.7.

Uma alternativa a este arquivo de configuração é informar o driver diretamente. Para isso basta informar na variável, conforme o exemplo abaixo.

Tabela 8.3. Configurações do PKCS#11

Ambiente	Variável de Ambiente	Variável JVM
Linux	export PKCS11_DRIVER=/usr/lib/libepsng_p11.so	-DPKCS11_DRIVER=/usr/lib/libepsng_p11.so
Windows	set PKCS11_DRIVER=/WINDOWS/system32/ngp11v211.dll	-DPKCS11_DRIVER=/WINDOWS/system32/ngp11v211.dll
Linux	export PKCS11_DRIVER=Pronova::/usr/lib/libepsng_p11.so	-DPKCS11_DRIVER=Pronova::/usr/lib/libepsng_p11.so
Windows	set PKCS11_DRIVER=Pronova::/WINDOWS/system32/ngp11v211.dll	-DPKCS11_DRIVER=Pronova::/WINDOWS/system32/ngp11v211.dll

Quando a variável for declarada através da JVM, ela deve ser feita diretamente no painel de controle do JAVA. A seguir demonstramos a configuração para o sistema Windows.

Abra o painel de controle e seleciona e abra o aplicativo "Java".

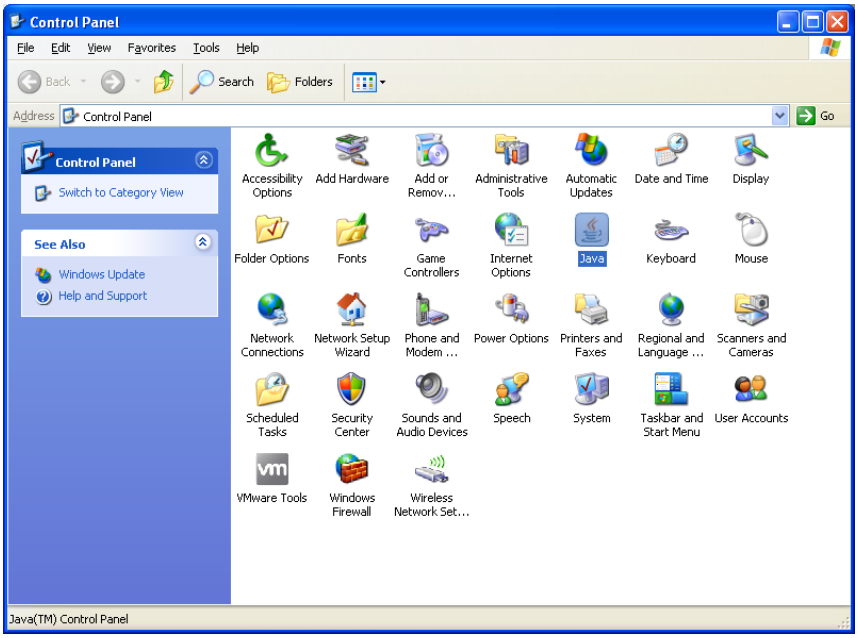

Figura 8.1. Java no Painel de Controle

Selecione a aba "Java" e clique em "View..."

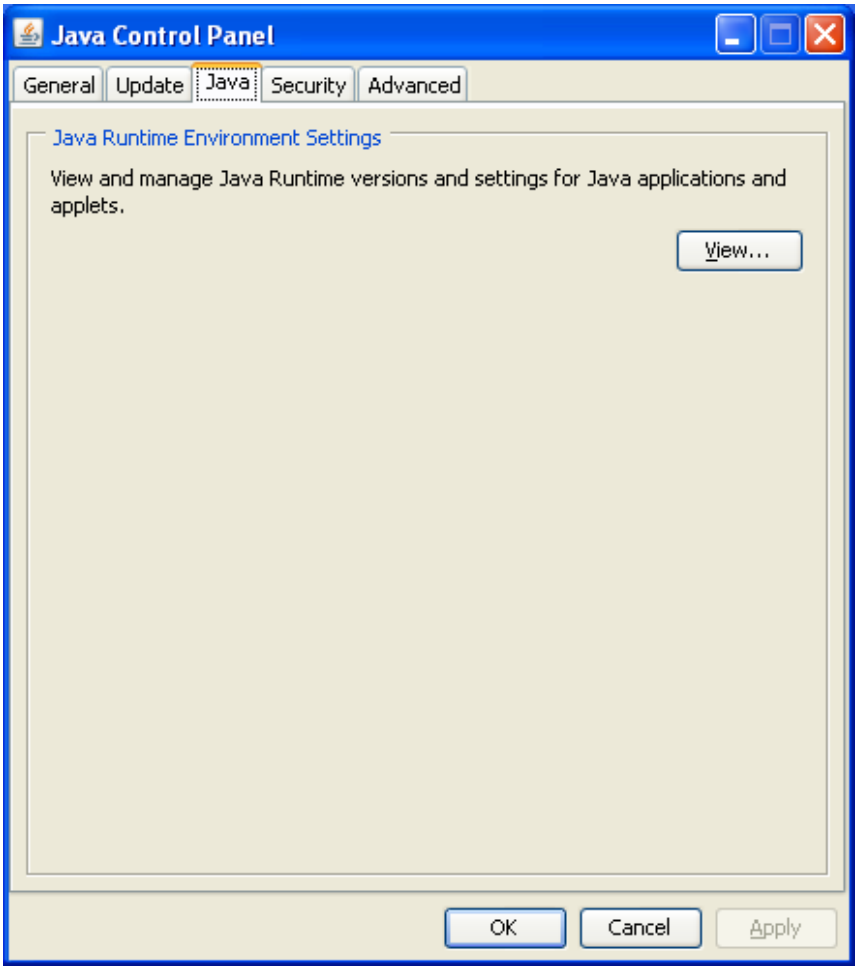

Figura 8.2. Configurações do ambiente Java

Na aba "User", em "Runtime Parameters", coloque a declaração da variável. Em seguida, aplique as alterações.

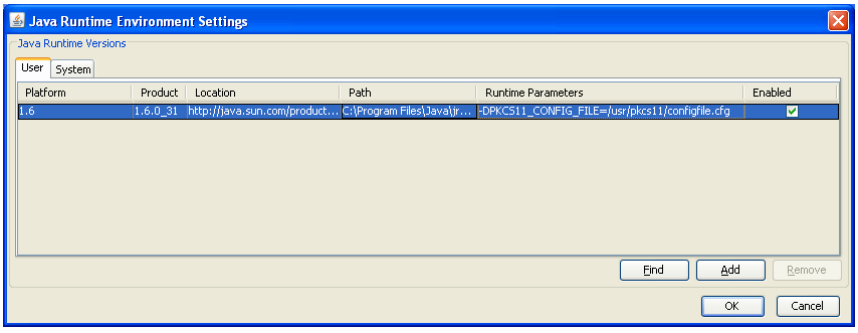

Figura 8.3. Desabilitando a camada MSCAPI

8.8. Configuração de Token / SmartCard por arquivo de configurações

As configurações acima demonstram uma configuração mais refinada para o carregamento de certificados em dispositivos, mas o componente possui um procedimento padrão a ser executado caso se deseje um método mais simplificado. A seguir é explicado como utilizar este mecanismo.

8.8.1. Utilizando certificados armazenados em Disco ou em Token / SmartCard no Windows

O Sistema Operacional Windows fornece uma camada chamada MSCAPI, ou Microsoft CryptoAPI, que facilita o acesso a certificados armazenados em disco ou em dispositivos criptográficos. Neste tipo de acesso, basta que o certificado esteja corretamente instalado e válido, e a própria camada nos fornecerá o driver correto e os meios para acessar os certificados. Até a versão 5 do Java não existia um provedor de acesso para esta camada, mas na versão 6 em diante foi implementado o provedor *SunMSCAPI* para lidar com este tipo de acesso.

8.8.2. Utilizando certificados armazenados em Disco no Linux ou Mac

Ao Contrario do Windows, que utiliza a API da *MS-CAPI* [http://en.wikipedia.org/wiki/Microsoft_CryptoAPI] para abstrair o acesso aos certificados digitais, em outros sistemas operacionais este recurso não existe. Para efetuar o acesso, precisamos criar um arquivo de configuração informando os parâmetros de acesso.

Para viabilizar o acesso em um sistema Não-Windows, deve ser criado um arquivo chamado `drivers.config` dentro do diretório [/home/usuario] com a parametrização mostrada abaixo. Nesta configuração serão carregados todos os certificados A1 que estejam instalados no Firefox.

Para o Ubuntu:

```
name = Provedor
slot = 2
library = /usr/lib/nss/libsoftokn3.so
nssArgs    =    "configdir='/home/<usuario>/.mozilla/firefox/bv3b2x7j.default'    certPrefix=''
 keyPrefix='' secmod='secmod.db'  flags='readWrite'"
```

Para o Fedora:

```
name = Provedor
slot = 2
library = /usr/lib/libsoftokn3.so
nssArgs    =    "configdir='/home/<usuario>/.mozilla/firefox/d3qv5ulo.default'    certPrefix=''
 keyPrefix='' secmod='secmod.db'  flags='readOnly'"
```

Para o Mac OS:

```
```

```
name = Provedor
slot = 2
library = /Applications/Firefox.app/Contents/MacOS/libsoftokn3.dylib
nssArgs        =        "configdir='/Users/usuario/Library/Application        Support/Firefox/
Profiles/5lnc5tyx.default' certPrefix='' keyPrefix='' secmod='secmod.db' flags='readOnly'"
```

Importante

A sequência de caracteres que precede o *.default* , como em *5lnc5tyx.default* é criptografada e, sendo assim, é diferente para cada equipamento e cada usuário.

8.8.3. Utilizando certificados armazenados em Token / SmartCard no Linux ou Mac

Para configurar um token A3, o conteúdo do arquivo `drivers.config` deve ser especificado como mostrado abaixo.

```
name = Provedor
description = Token Pronova ePass2000
library = /usr/local/ngsrv/libepsng_p11.so.1.2.2
```

Importante

Não é possível utilizar certificados A3 e A1 no Linux ou Mac simultaneamente, devendo ser configurado somente UM dos tipos de acesso em um determinado momento.

8.9. Desabilitar a camada de acesso SunMSCAPI

Quando o componente é utilizado em ambiente Windows, o acesso é feito através de uma camada de abstração chamada MSCAPI, que abstrai informações que são particulares de cada token ou smartcard, como os drivers do dispositivo, por exemplo. Este tipo de recurso facilita o uso do componente com dispositivos de diversos fabricantes. Porém, podem existir casos específicos em que o acesso precisa ser feito diretamente ao driver para utilização de funções específicas, como forçar o logout de um token. Para isso, é necessário informar na JVM um parâmetro chamado `mscapi.disabled` passando o valor `true` . Este parâmetro informa que o acesso será feito via PKCS11, sendo necessário informar o arquivo de configuração do token que se deseja acessar. Caso o parâmetro `mscapi.disabled` esteja ausente, o componente fará uso do MSCAPI normalmente.

A seguir demonstramos a configuração para o sistema Windows.

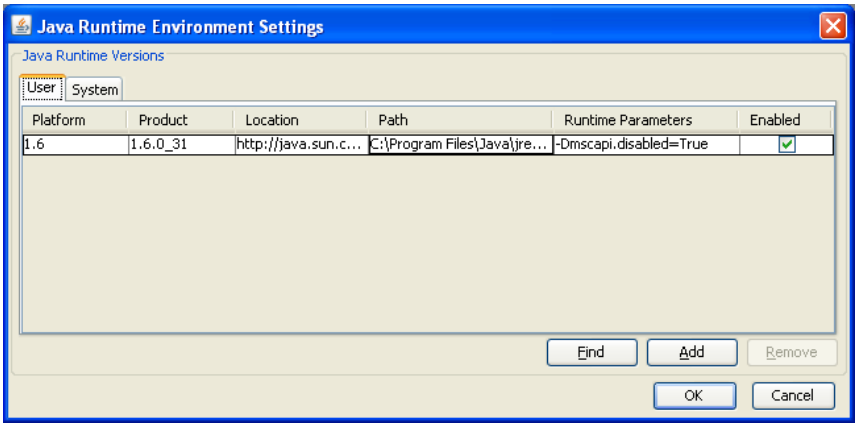

Figura 8.4. Desabilitando a camada MSCAPI

Parte III. Demoiselle Signer

O componente *Demoiselle Signer* foi desenvolvido para atender às necessidades de assinatura digital no âmbito da ICP-Brasil.

Configuração do Demoiselle Signer

9.1. Instalação do componente

Para instalar o componente *Demoiselle Signer* na aplicação, basta adicionar a sua dependência no arquivo `pom.xml` do projeto gerenciado pelo Maven:

```
<dependency>
    <groupId>br.gov.frameworkdemoiselle.component</groupId>
    <artifactId>demoiselle-certificate-signer</artifactId>
    <version>1.0.11</version>
</dependency>
```

Funcionalidades

Este componente provê mecanismos de assinatura digital baseado nas normas ICP-Brasil e implementa mecanismos de assinatura digital em dois formatos: PKCS1 e PKCS7. A maior diferença entre esses dois mecanismos está na forma de envelopamento da assinatura digital, onde o PKCS1 não possui um formato de envelope, sendo o resultado da operação de assinatura a própria assinatura, já o PKCS7 possui um formato de retorno que pode ser binário (especificado na RFC5126) ou XML. A interface `br.gov.frameworkdemoiselle.certificate.signer.Signer` especifica o comportamento padrão dos mecanismos de assinatura digital. O componente especializa essa interface em mais duas, são elas: `br.gov.frameworkdemoiselle.certificate.signer.pkcs1.PKCS1Signer` para implementações de mecanismos PKCS1 e `br.gov.frameworkdemoiselle.certificate.signer.pkcs7.PKCS7Signer` para implementações de mecanismos de envelopamento PKCS7.

Este componente, até a presente versão, permite assinar dados representados por um array de bytes. Então se for necessário a assinatura de um arquivo, por exemplo, a aplicação deverá montar um array de bytes com o conteúdo do arquivo para poder assiná-lo.

Para assinar um dado através do componente demoiselle-signer é preciso executar alguns passos.

- Ter um conteúdo a ser assinado
- Escolher qual formato de assinatura a ser utilizado PKCS1 ou PKCS7
- Fabricar o objeto responsável pela implementação do formato escolhido
- Passar algumas informações para o objeto fabricado como chave criptográfica, algoritmo, etc. O formato PKCS7 necessita de mais informações do que o formato PKCS1.
- Assinar o conteúdo

10.1. Assinatura Digital no Formato PKCS1

A seguir temos um fragmento de código que demonstra uma assinatura no formato PKCS1.

```
/* conteudo a ser assinado */
byte[] content = "conteudo a ser assinado".getBytes();

/* chave privada */
PrivateKey chavePrivada = getPrivateKey(); /* implementar metodo para pegar chave privada */

/* construindo um objeto PKCS1Signer atraves da fabrica */
PKCS1Signer signer = PKCS1Factory.getInstance().factory();

/* Configurando o algoritmo */
signer.setAlgorithm(SignerAlgorithmEnum.SHA1withRSA);

/* Configurando a chave privada */
signer.setPrivateKey(chavePrivada);

/* Assinando um conjunto de bytes */
byte[] sign = signer.signer(content);
```

10.2. Assinatura Digital no Formato PKCS7

O formato PKCS7 permite colocar informações no pacote gerado. Essas informações são úteis para validar a assinatura. A ICP-Brasil define um conjunto mínimo de informações básicas para as assinaturas digitais. São elas: Tipo de conteúdo, data da assinatura, algoritmo de resumo aplicado e a política de assinatura. O componente demoiselle-signer já monta o pacote final com três atributos obrigatórios: tipo de conteúdo, data da assinatura e o algoritmo de resumo. Então, para montar um PKCS7 padrão ICP-Brasil é necessário informar ao objeto PKCS7Signer qual a política de assinatura aplicada.

A seguir temos um fragmento de código que demonstra a utilização do pacote PKCS7 padrão.

```
byte[] content = readContent("texto.txt"); /* implementar metodo de leitura de arquivo */
PKCS7Signer signer = PKCS7Factory.getInstance().factoryDefault();
signer.setCertificate(certificate);
signer.setPrivateKey(privateKey);
byte[] sign = signer.signer(this.content);
```

A seguir temos um fragmento de código que demonstra a utilização do pacote PKCS7 padrão com informação da política de assinatura. Neste caso podemos escolher uma das políticas que já acompanham o componente e referem-se à Assinatura Digital de Referência Básica.

- `ADRBCMS_1_0` , Refere-se à Assinatura Digital de Referência Básica versão 1.0;
- `ADRBCMS_1_1` , Refere-se à Assinatura Digital de Referência Básica versão 1.1;
- `ADRBCMS_2_0` , Refere-se à Assinatura Digital de Referência Básica versão 2.0;
- `ADRBCMS_2_1` , Refere-se à Assinatura Digital de Referência Básica versão 2.1;

```
byte[] content = readContent("texto.txt"); /* implementar metodo de leitura de arquivo */
PKCS7Signer signer = PKCS7Factory.getInstance().factoryDefault();
signer.setCertificate(certificate);
signer.setPrivateKey(privateKey);
signer.setSignaturePolicy(new ADRBCMS_2_1());
byte[] sign = signer.signer(this.content);
```

Importante

Caso não seja especificada nenhuma política, o componente assumirá a política padrão ADRBCMS_1_0 (1.0).

10.3. Criação de Assinatura Digital sem envio do conteúdo original para o assinador

Este procedimento visa facilitar a geração de assinaturas digitais em aplicações Web pois não há necessidade de trafegar o conteúdo original pela rede, sendo necessário apeas o tráfego dos bytes do resumo do conteúdo original

(HASH). Neste caso, é necessário gerar o HASH do conteúdo a ser assinado e passar para o assinador. Ao gerar o HASH, é importante dar atenção ao algoritmo a ser usado, pois na validação da assinatura será considerado o algoritmo da política escolhida. Então, para que esse procedimento funcione, é necessário escolher o algoritmo do HASH igual ao algoritmo da assinatura digital. Não há necessidade de passar o conteúdo original para o assinador, como mostra a última linha do exemplo abaixo:

```
import br.gov.frameworkdemoiselle.certificate.criptography.Digest;
import br.gov.frameworkdemoiselle.certificate.criptography.factory.DigestFactory;
import br.gov.frameworkdemoiselle.certificate.criptography.DigestAlgorithmEnum;
...

byte[] content = readContent("texto.txt"); /* implementar metodo de leitura de arquivo */
/* Gerando o HASH fora do assinador */
Digest digest = DigestFactory.getInstance().factoryDefault();
digest.setAlgorithm(DigestAlgorithmEnum.SHA_256);
byte[] hash = digest.digest(content);

/* Gerando a assinatura a partir do HASH gerado anteriormente */
PKCS7Signer signer = PKCS7Factory.getInstance().factoryDefault();
signer.setCertificate(certificate);
signer.setPrivateKey(privateKey);
signer.addAttribute(new MessageDigest(hash));
signer.setSignaturePolicy(new ADRBCMS_2_1());
byte[] sign = signer.signer(null);
```

Importante

Este procedimento gera o pacote PKCS7 idêntico ao pacote gerado pelo exemplo do tópico 2.2

10.4. Validação de assinatura PKCS7 sem o conteúdo anexado

A seguir temos um fragmento de código que demonstra a validação uma assinatura PKCS7 sem o conteúdo anexado.

```
byte[] content = readContent("texto.txt"); /* implementar metodo de leitura de arquivo */
byte[] signed = readContent("texto.pkcs7"); /* implementar metodo de leitura de arquivo */
PKCS7Signer signer = PKCS7Factory.getInstance().factoryDefault();
boolean checked = signer.check(content, signed);
```

10.5. Validação de assinatura PKCS7 com o conteúdo anexado

A seguir temos um fragmento de código que demonstra a validação uma assinatura PKCS7 com o conteúdo anexado.

```
byte[] signed = readContent("texto.pkcs7"); /* implementar metodo de leitura de arquivo */
PKCS7Signer signer = PKCS7Factory.getInstance().factoryDefault();
/* O componente ira detectar se o conteudo original esta anexado */
boolean checked = signer.check(null, signed);
```

10.6. Leitura do conteúdo anexado a uma assinatura PKCS7

A seguir temos um fragmento de código que demonstra a recuperação do conteúdo de um arquivo anexado a uma assinatura PKCS7.

```
byte[] signed = readContent("texto.pkcs7"); /* implementar metodo de leitura de arquivo */
PKCS7Signer signer = PKCS7Factory.getInstance().factoryDefault();

/* Para extrair o conteudo original validando a assinatura */
byte[] content = signer.getAttached(signed, true);

/* Para extrair o conteudo original sem validar a assinatura */
byte[] content = signer.getAttached(signed, false);
```

Exemplos de Uso

A seguir temos alguns exemplos de tarefas normalmente necessárias na utilização do componente.

11.1. Carregar um array de bytes de um arquivo

```
byte[] result = null;
File file = new File("documento.odp");
FileInputStream is = new FileInputStream(file);
result = new byte[(int) file.length()];
is.read(result);
is.close();
return result;
```

11.2. Gravar um array de bytes em um arquivo

```
byte[] conteudo = "este eh um conteudo de arquivo texto".getBytes();
FileOutputStream out = new FileOutputStream(new File("arquivo.txt");
out.write(sign);
out.close();
```

11.3. Carregar uma chave privada em arquivo

```
File file = new File("private_rsa_1024.pkcs8");
fileContent = new byte[(int) file.length()];
is = new FileInputStream(file);
is.read(fileContent);
is.close();
PKCS8EncodedKeySpec privateKeySpec = new PKCS8EncodedKeySpec(fileContent);
KeyFactory keyFactory = KeyFactory.getInstance("RSA");
PrivateKey chavePrivada = keyFactory.generatePrivate(privateKeySpec);
```

11.4. Carregar uma chave privada de um token

```
KeyStoreLoader keyStoreLoader = KeyStoreLoaderFactory.factoryKeyStoreLoader();
KeyStore keyStore = keyStoreLoader.getKeyStore("pinnumber");
String certificateAlias = keyStore.aliases().nextElement();
PrivateKey chavePrivada = (PrivateKey)keyStore.getKey(certificateAlias, "pinnumber");
```

11.5. Carregar uma chave pública em arquivo

```
File file = new File("public_rsa_1024.pkcs8");
byte[] fileContent = new byte[(int) file.length()];
is = new FileInputStream(file);
is.read(fileContent);
is.close();
X509EncodedKeySpec publicKeySpec = new X509EncodedKeySpec(fileContent);
KeyFactory keyFactory = KeyFactory.getInstance("RSA");
PublicKey chavePublica = keyFactory.generatePublic(publicKeySpec);
```

11.6. Carregar uma chave pública de um token

```
/* Usando os componentes demoiselle-keystore e demoiselle-certificate */
KeyStoreLoader keyStoreLoader = KeyStoreLoaderFactory.factoryKeyStoreLoader();
KeyStore keyStore = keyStoreLoader.getKeyStore("pinnumber");
CertificateLoader certificateLoader = new CertificateLoaderImpl();
certificateLoader.setKeyStore(keyStore);
X509Certificate certificate = certificateLoader.loadFromToken();
PublicKey chavePublica = certificate.getPublicKey();
```

11.7. Carregar um certificado digital de um arquivo

```
/* Usando o componente demoiselle-certificate */
CertificateLoader certificateLoader = new CertificateLoaderImpl();
X509Certificate certificate = certificateLoader.load(new File("certificado.cer");
```

11.8. Carregar um certificado digital de um token

```
/* Usando os componentes demoiselle-keystore e demoiselle-certificate */
KeyStoreLoader keyStoreLoader = KeyStoreLoaderFactory.factoryKeyStoreLoader();
KeyStore keyStore = keyStoreLoader.getKeyStore("pinnumber");
CertificateLoader certificateLoader = new CertificateLoaderImpl();
certificateLoader.setKeyStore(keyStore);
X509Certificate certificate = certificateLoader.loadFromToken();
```

Parte IV. Demoiselle Criptography

O *demoiselle-criptography* é um componente corporativo que provê um mecanismo simplificado de criptografia de dados. Neste contexto o componente atua nos dois principais tipos de algoritmos, os algoritmos de chave simétrica e algoritmos de chave assimétrica.

O componente provê as funções de cifragem e decifragem utilizando algoritmos simétricos ou também chamados de algoritmos de chave-simétrica, ou seja, os que utilizam uma mesma chave para cifrar e decifrar as mensagens.

Além disso o componente utiliza algoritmos de hash para criptografia de dados com a finalidade de criar um valor único que identifique um dado original. Este recurso é recomendado para finalidades de autenticação, nas quais deseja-se armazenar as senhas criptografas por meio de um valor hash. Também é possível construir hash de arquivos no intuito de avaliar sua integridade física.

O componente também realiza as funções de cifragem e decifragem por meio de algoritmos de chave-assimétrica. Neste processo é necessário um par de chaves para realizar a cifragem e decifram das mensagens. A primeira chave é denominada chave privada, ela é de posse exclusiva de seu detentor e ninguém mais a conhece. A segunda chave do par e denominada de chave pública e pode ser enviada a qualquer indivíduo.

Configuração do Criptography

12.1. Instalação do componente

Para instalar o componente *Demoiselle Criptography* na aplicação, basta adicionar a sua dependência no arquivo `pom.xml` do projeto gerenciado pelo Maven:

```
<dependency>
    <groupId>br.gov.frameworkdemoiselle.component</groupId>
    <artifactId>demoiselle-certificate-criptography</artifactId>
    <version>1.0.11</version>
</dependency>
```

12.2. Customização das implementações

O componente Demoiselle Criptography possui implementações padrões às funcionalidades de criptografia, entretanto é possível definir outras implementações. Neste caso é necessário informar ou como variável de ambiente ou com variável da JVM qual a implementação das interfaces: `br.gov.frameworkdemoiselle.certificate.criptography.Criptography` e `br.gov.frameworkdemoiselle.certificate.criptography.Digest`.

Por padrão, as respectivas implementações são: `br.gov.frameworkdemoiselle.certificate.criptography.implementation.CriptographyImpl` e `br.gov.frameworkdemoiselle.certificate.criptography.implementation.DigestImpl`.

Veja a configuração por meio de variável de ambiente para definição da implementação da interface `br.gov.frameworkdemoiselle.certificate.criptography.Criptography`.

Tabela 12.1. Exemplo com Variável de Ambiente

Ambiente	Variável de ambiente
Linux	export criptography.implementation = br.gov.frameworkdemoiselle.certificate.criptography.implementation.CriptographyImpl
Windows	set criptography.implementation= br.gov.frameworkdemoiselle.certificate.criptography.implementation.CriptographyImpl

Tabela 12.2. Exemplo com Variável JVM

Ambiente	Variável JVM
Linux	-criptography.implementation= br.gov.frameworkdemoiselle.certificate.criptography.implementation.MyCriptographyImpl
Windows	-criptography.implementation= br.gov.frameworkdemoiselle.certificate.criptography.implementation.MyCriptographyImpl

Veja a configuração por meio de variável de ambiente para definição da implementação da interface `br.gov.frameworkdemoiselle.certificate.criptography.Digest`.

Tabela 12.3. Exemplo com Variável de Ambiente

Ambiente	Variável de ambiente
Linux	export digest.implementation= br.gov.frameworkdemoiselle.certificate.criptography.implementation.MyDigestImpl
Windows	set digest.implementation= br.gov.frameworkdemoiselle.certificate.criptography.implementation.MyDigestImpl

Tabela 12.4. Exemplo com Variável JVM

Ambiente	Variável JVM
Linux	-messageDigest.implementation= br.gov.frameworkdemoiselle.certificate.criptography.implementation.MyDigestImpl
Windows	-messageDigest.implementation= br.gov.frameworkdemoiselle.certificate.criptography.implementation.MyDigestImpl

Funcionalidades

13.1. A Criptografia Simétrica

A cifragem e decifragem de dados são providas pela interface `Criptography` e o componente se utiliza de uma fábrica dessa interface. Segue abaixo um exemplo ilustrativo:

```
public class App {

        public static void main(String[] args) {
                String frase = "conteudo original";

               Criptography criptography = CriptographyFactory.getInstance().factoryDefault();

                /* Geracao da chave unica */
                Key key = criptography.generateKey();
                criptography.setKey(key);

                /* Cifragem */
                byte[] conteudo_criptografado = criptography.cipher(frase.getBytes());
                System.out.println(conteudo_criptografado);

                /* Decifragem */
             byte[] conteudo_descriptografado = criptography.decipher(conteudo_criptografado);
                System.out.println(new String(conteudo_descriptografado));
        }
}
```

Os métodos `cipher` e `decipher` recebem como entrada um array de bytes e retornam o array de bytes processado.

Para que a criptografia simétrica seja realizada é necessário o uso de uma única chave, para criptografar e descriptografar. Neste caso, é necessário gerar a chave através do método `generateKey`.

Caso não seja informado o algoritmo de criptografia o componente utilizará como padrão o algoritmo *AES (Advanced Encryption Standard)*. Caso necessite utilizar outro algoritmo invoque o método `setAlgorithm` informando um `SymmetricAlgorithmEnum` ou um `AsymmetricAlgorithmEnum.`

```
public class App {

        public static void main(String[] args) {
                String frase = "conteudo original";

               Criptography criptography = CriptographyFactory.getInstance().factoryDefault();

                /* Alterando algoritmo */
                criptography.setAlgorithm(SymmetricAlgorithmEnum.TRI_DES);

                /* Geracao da chave unica */
                Key key = criptography.generateKey();
```

```
                criptography.setKey(key);

                byte[] conteudo_criptografado = criptography.cipher(frase.getBytes());
                System.out.println(conteudo_criptografado);

             byte[] conteudo_descriptografado = criptography.decipher(conteudo_criptografado);
                System.out.println(new String(conteudo_descriptografado));
        }
}
```

Caso as opções de criptografia definidas pelo `SymmetricAlgorithmEnum` não atendam é possível customizar os parâmetros de criptografia através dos métodos `setAlgorithm`, `setKeyAlgorithm` e `setSize`.

```
public static void main(String[] args) {
                String frase = "conteudo original";

               Criptography criptography = CriptographyFactory.getInstance().factoryDefault();

                /* Customizacao de parametros */
                criptography.setAlgorithm("AES/ECB/PKCS5Padding");
                criptography.setKeyAlgorithm("AES");
                criptography.setSize(128);

                /* Cifragem */
                byte[] conteudo_criptografado = criptography.cipher(frase.getBytes());
                System.out.println(conteudo_criptografado);

                /* Decifragem */
             byte[] conteudo_descriptografado = criptography.decipher(conteudo_criptografado);
                System.out.println(new String(conteudo_descriptografado));
        }
```

13.2. A Criptografica Assimétrica

Na criptografia assimétrica é necessário um par de chaves para realizar a cifragem e decifram das mensagens. A primeira chave é denominada chave privada e é de posse exclusiva de seu detentor. A segunda chave do par é denominada de chave pública e pode ser enviada a qualquer indivíduo.

```
        /* Cifragem */
        Criptography cripto = CriptographyFactory.getInstance().factoryDefault();
        cripto.setKey(privateKey);
        byte[] conteudoCriptografado = cripto.cipher("SERPRO".getBytes());
        System.out.println(conteudoCriptografado);

        /* Decifragem */
        Criptography cripto2 = CriptographyFactory.getInstance().factoryDefault();
        cripto2.setKey(publicKey);

        byte[] conteudoDescriptografado = cripto2.decipher(conteudoCriptografado);
        System.out.println(new String(conteudoDescriptografado));
```

Perceba a utilização do método `setKey` para informar qual chave será utilizada no processo de cifragem e decifragem. Vale lembrar que na criptografia assimétrica a cifragem realizada com a chave privada só poderá ser decifrada com a chave pública e vice-versa.

Na sequência, demonstramos a utilização do componente com certificados digitais do tipo A1 e A3.

13.2.1. Certificados A1

O certificado A1 é aquele que encontra-se armazenado no sistema de arquivo do sistema operacional. Para exemplificar sua manipulação, segue o código abaixo:

```
	public static void main(String[] args) {

		try {
			/* Obtendo a chave publica */
			File file = new File("/home/03397040477/public.der");
			byte[] encodedPublicKey = new byte[(int) file.length()];
			InputStream inputStreamPublicKey = new FileInputStream(file);
			inputStreamPublicKey.read(encodedPublicKey);
			inputStreamPublicKey.close();
		   X509EncodedKeySpec publicKeySpec = new X509EncodedKeySpec(encodedPublicKey);
			KeyFactory kf = KeyFactory.getInstance("RSA");
			PublicKey publicKey = kf.generatePublic(publicKeySpec);

			/* Obtendo a chave privada */
			file = new File("/home/03397040477/private.pk8");
			byte[] encodedPrivateKey = new byte[(int) file.length()];
			InputStream inputStreamPrivateKey = new FileInputStream(file);
			inputStreamPrivateKey.read(encodedPrivateKey);
			inputStreamPrivateKey.close();
			PKCS8EncodedKeySpec privateKeySpec = new PKCS8EncodedKeySpec(encodedPrivateKey);
			kf = KeyFactory.getInstance("RSA");
			PrivateKey privateKey = kf.generatePrivate(privateKeySpec);

			/* Cifragem */
		    Criptography cripto = CriptographyFactory.getInstance().factoryDefault();
			cripto.setAlgorithm(AsymmetricAlgorithmEnum.RSA);
			cripto.setKey(privateKey);
			byte[] conteudoCriptografado = cripto.cipher("SERPRO".getBytes());
			System.out.println(conteudoCriptografado);

			/* Decifragem */
		    Criptography cripto2 = CriptographyFactory.getInstance().factoryDefault();
			cripto2.setAlgorithm(AsymmetricAlgorithmEnum.RSA);
			cripto2.setKey(publicKey);

		    byte[] conteudoDescriptografado = cripto2.decipher(conteudoCriptografado);
			System.out.println(new String(conteudoDescriptografado));

		} catch (Exception e) {
			e.printStackTrace();
		    Assert.assertTrue("Configuracao nao carregada: " + e.getMessage(), false);
		}
	}
```

Neste exemplo é demonstrada a obtenção das chaves pública e privada do certificado A1. Note que, apesar do código para manipulação dos certificados, a forma de uso do componente *Demoiselle Criptography* para cifragem e decifragem de mensagens é a mesma.

13.2.2. Certificados A3

O certificado A3 é armazenado em dispositivos eletrônicos como smart card ou tokens usb que criptografam o certificado provendo maior segurança. No exemplo abaixo utilizamos o componente *Demoiselle Core* para obtenção do keyStore a partir de um token usb. Você pode ver mais sobre esse componente em *Capítulo 6, Configuração do Demoiselle Core*

```
	public static void main(String[] args) {

		/* Senha do dispositivo */
		String PIN = "senha_do_token";
		try {

			/* Obtendo a chave privada */
			KeyStore keyStore = KeyStoreLoaderFactory.factoryKeyStoreLoader().getKeyStore(PIN);
			String alias = (String) keyStore.aliases().nextElement();
		 PrivateKey privateKey = (PrivateKey) keyStore.getKey(alias, PIN.toCharArray());

			/* Obtendo a chave publica */
			CertificateLoader cl = new CertificateLoaderImpl();
			X509Certificate x509 = cl.loadFromToken(PIN);
			PublicKey publicKey = x509.getPublicKey();

			/*Configurando o Criptography */
		    Criptography cripto = CriptographyFactory.getInstance().factoryDefault();
			cripto.setAlgorithm(AsymmetricAlgorithmEnum.RSA);
			cripto.setProvider(keyStore.getProvider());

			/* criptografando com a chave privada */
			cripto.setKey(privateKey);
			byte[] conteudoCriptografado = cripto.cipher("SERPRO".getBytes());
			System.out.println(conteudoCriptografado);

			/* descriptografando com a chave publica */
			cripto.setKey(publicKey);
			byte[] conteudoAberto = cripto.decipher(conteudoCriptografado);
			System.out.println(new String(conteudoAberto));

		} catch (UnrecoverableKeyException e) {
			e.printStackTrace();
		} catch (KeyStoreException e) {
			e.printStackTrace();
		} catch (NoSuchAlgorithmException e) {
			e.printStackTrace();
		}
	}
```

O componente utiliza como padrão o provider SUN JCE, mas caso necessite de outro provider utilize o método `setProvider` da classe `Criptography`.

No exemplo acima foi utilizado o método `setProvider` para informar o provedor, ou seja, quem executará os algoritmos de criptografia. Até então, nos exemplos anteriores, o provedor era a biblioteca SUN JCE contida na própria JVM. Como o token é o único a ter acesso a chave privada do certificado ele também será o único capaz de executar os processos de cifragem e decifragem.

Desta forma foi utilizado o objeto KeyStore do próprio token usb para informar o novo provider ao Criptography.

13.3. Geração de Hash

13.3.1. Hash simples

A criptografia de hash tem a finalidade de criar um valor único que identifique um dado original. Este recurso pode ser utilizado por exemplo para finalidades de autenticação nas quais deseja-se armazenar as senhas cifradas por meio de um valor hash.

A fábrica `DigestFactory` do componente constrói um objeto padrão do tipo Digest que calcula o valor hash de um array de bytes retornando outro array de bytes.

```
public static void main(String[] args) {
        Digest digest = DigestFactory.getInstance().factoryDefault();
        byte[] resumo = digest.digest("SERPRO".getBytes());
        System.out.println(resumo);
}
```

Caso queira obter o valor hash no formato caractere hexadecimal utilize o método `digestHex` . Este formato é bastante utilizado para representar o hash de arquivos.

```
public static void main(String[] args) {
        Digest digest = DigestFactory.getInstance().factoryDefault();
        String resumo = digest.digestHex("SERPRO".getBytes());
        System.out.println(resumo);
}
```

13.3.2. Hash de arquivo

O hash de arquivo pode ser utilizado quando se deseja verificar a integridade física de um arquivo. No caso de ferramentas de download é possível ao final do processo de transferência de dados, verificar se o arquivo obtido apresenta o mesmo hash do arquivo original.

A interface `Digest` possui os métodos `digestFile` e `digestFileHex` para retornar respectivamente o valor hash em array de bytes ou caracter hexadecimal:

```
public static void main(String[] args) {
        Digest digest = DigestFactory.getInstance().factoryDefault();
        digest.setAlgorithm(DigestAlgorithmEnum.SHA_256);
    String resumo = digest.digestFileHex(new File("/home/03397040477/relatorio.pdf"));
        System.out.println(resumo);
```

```
        }
```

Os algoritmos de hash recomendados pelo componente são definidos pelo enum `DigestAlgorithmEnum`. Caso não seja informado o componente utilizará o "SHA-1" por ser considerado mais seguro quanto a quebras em relação ao MD5.

Parte V. Demoiselle CA ICP-Brasil

O Demoiselle CA ICP-Brasil fornece uma implementação para validação de um conjunto de Autoridades Certificadoras de certificados digitais de Produção.

Configuração do CA ICP-Brasil

14.1. Instalação do componente

Para instalar o componente *Demoiselle CA ICP-Brasil* na aplicação, basta adicionar a sua dependência no arquivo `pom.xml` do projeto gerenciado pelo Maven:

```
<dependency>
    <groupId>br.gov.frameworkdemoiselle.component</groupId>
    <artifactId>demoiselle-certificate-ca-icpbrasil</artifactId>
    <version>1.0.11</version>
</dependency>
```

14.2. Autoridades Certificadoras

Abaixo mostraremos a lista de autoridades certificadoras atual que está presente no componente:

Tabela 14.1. Lista de Autoridades Certificadoras

Autoridade Certificadora	Arquivo	Data de expiração
AC Certisign-JUS G2	ac_certisign_jus_g2.crt	19/06/2017
Autoridade Certificadora SERPRORFBv4	ac_serpro_rfb_v4.crt	11/10/2021
AC VALID RFB	ac_valid_rfb.crt	23/07/2020
AC SINCOR RFB G3	ac_sincor_rfb_g3.crt	26/12/2019
AC PRODEST RFB v2	ac_prodest_rfb_v2.crt	20/12/2019
AC BOA VISTA CERTIFICADORA	ac_boavista_certificadora.crt	29/11/2021
AC VALID	ac_valid.crt	04/06/2022
Autoridade Certificadora Raiz Brasileira v2	autoridade_certificadora_raiz_brasileira_v2.crt	31/05/2023
AC CAIXA-JUS v2	ac_caixa_jus_v2.crt	28/12/2019
Autoridade Certificadora SERPRO v2	autoridade_certificadora_serpro_v2.crt	18/02/2019
Autoridade Certificadora do SERPRO Final v4	autoridade_certificadora_do_serpro_final_v4.crt	08/10/2021
Autoridade Certificadora da Presidencia da Republica v4	autoridade_certificadora_da_presidencia_da_republica_v4.crt	21/06/2023
AC SERASA-JUS v1	ac_serasa_jus_v1.crt	19/06/2017
AC Imprensa Oficial SP RFB G3	ac_imesp_rfb_g3.crt	27/12/2019
AC Secretaria da Receita Federal do Brasil	secretaria_da_receita_federal_do_brasil.crt	24/10/2018

Autoridade Certificadora	Arquivo	Data de expiração
AC Certisign G6	ac_certisign_g6.crt	20/09/2021
AC Certisign RFB G3	ac_certisign_rfb_g3.crt	29/10/2016
SERASA Autoridade Certificadora Principal v2	serasa_autoridade_certificadora_principal_v2.crt	20/09/2021
AC FENACON Certisign RFB G2	ac_fenacon_certisign_rfb_g2.crt	29/10/2016
AC Certisign Tempo G1	ac_certisign_tempo_g1.crt	25/07/2021
AC BR RFB G3	ac_br_rfb_g3.crt	26/12/2019
AC Imprensa Oficial SP G2	ac_imprensa_oficial_sp_g2.crt	09/11/2019
AC SOLUTI-JUS v1	ac_soluti_jus_v1.crt	22/10/2021
SERASA Certificadora Digital v1	serasa_certificadora_digital_v1.crt	04/08/2016
Autoridade Certificadora Raiz Brasileira v2	icp-brasilv2.crt	21/06/2023
AC Imprensa Oficial SP RFB G3	ac_imprensa_oficial_sp_rfb_g3.crt	27/12/2019
AC SERASA RFB v2	ac_serasa_rfb_v2.crt	16/12/2019
AC PRODEMGE RFB G3	ac_prodemge_rfb_g3.crt	27/12/2019
Autoridade Certificadora SERPRORFB v3	ac_serpro_rfb_v3.crt	13/12/2019
AC SINCOR RFB G2	ac_sincor_rfb_g2.crt	30/10/2016
AC CAIXA v2	ac_caixa_v2.crt	02/12/2021
AC Instituto Fenacon G2	ac_instituto_fenacon_g2.crt	06/11/2019
AC VALID PLUS	ac_valid_plus.crt	31/05/2022
AC CAIXA-JUS v1	ac_caixa_jus_v1.crt	14/01/2019
Autoridade Certificadora do SERPRO Final v3	autoridade_certificadora_do_serpro_final_v3.crt	16/12/2019
Autoridade Certificadora da Presidencia da Republica v3	autoridade_certificadora_da_presidencia_da_republica_v3.crt	23/11/2021
AC Certisign SPB G3	ac_certisign_spb_g3.crt	08/09/2016
AC BOA VISTA RFB	ac_boa_vista_rfb.crt	11/10/2021
AC VALID SPB	ac-valid_spb.crt	18/04/2021
AC Imprensa Oficial SP RFB G2	ac_imesp_rfb_g2.crt	29/10/2016
AC VALID BRASIL	ac_valid_brasil.crt	12/07/2020
SERASA Autoridade Certificadora Principal v1	serasa_autoridade_certificadora_principal_v1.crt	31/07/2018
AC CAIXA PJ v2	ac_caixa_pj_v2.crt	21/12/2019
AC SINCOR RIO RFB G1	ac_sincor_rio_rfb_g1.crt	11/10/2021
AC SOLUTI Multipla	ac_soluti_multipla_v1.crt	20/06/2023
AC OAB G2	ac_oab_g2.crt	15/11/2019
AC BR RFB G2	ac_br_rfb_g2.crt	30/10/2016
AC VALID SPB	ac_valid_spb.crt	18/04/2021

Autoridade Certificadora	Arquivo	Data de expiração
Autoridade Certificadora do PRODERJ v2	autoridade_certificadora_do_proderj_v2.crt	22/12/2019
AC OAB	ac_oab.crt	08/09/2016
AC BOA VISTA	ac_boavista.crt	21/06/2023
AC PRODEMGE RFB G2	ac_prodemge_rfb_g2.crt	30/10/2016
AC PETROBRAS G3	ac_petrobras_g3.crt	27/11/2019
AC CAIXA PF v2	ac_caixa_pf_v2.crt	21/12/2019
AC DIGITALSIGN	ac_digitalsign.crt	17/10/2021
AC Imprensa Oficial G4	ac_imprensa_oficial_g4.crt	16/01/2023
Autoridade Certificadora do PRODERJ	autoridade_certificadora_do_proderj.crt	18/05/2017
Autoridade Certificadora da Casa da Moeda do Brasil	autoridade_certificadora_da_casa_da_moeda_do_brasil.crt	03/02/2020
AC SOLUTI	ac_soluti.crt	20/06/2023
Autoridade Certificadora do SERPRO Final v2	autoridade_certificadora_do_serpro_final_v2.crt	12/12/2016
Autoridade Certificadora do SERPRORFB	ac_serpro_rfb.crt	08/11/2016
Autoridade Certificadora da Presidencia da Republica v2	autoridade_certificadora_da_presidencia_da_republica_v2.crt	13/10/2018
AC Notarial RFB G3	ac_notarial_rfb_g3.crt	26/12/2019
Autoridade Certificadora da Casa da Moeda do Brasil v3	autoridade_certificadora_da_casa_da_moeda_do_brasil_v3.crt	16/10/2022
AC DIGITALSIGN ACP	ac_digitalsign_acp.crt	21/06/2023
AC VALID-JUS v4	ac_valid_jus_v4.crt	22/10/2021
AC PRODEMGE G3	ac_prodemge_g3.crt	23/11/2019
AC SERASA RFB v1	ac_serasa_rfb.crt	28/10/2016
AC DIGITALSIGN RFB	ac_digitalsign_rfb.crt	11/10/2021
AC Certisign Multipla G5	ac_certisign_multipla_g5.crt	22/09/2019
AC Imprensa Oficial	ac_imprensa_oficial.crt	24/10/2018
AC Secretaria da Receita Federal do Brasil v3	ac_secretaria_da_receita_federal_do_brasil_v3.crt	21/10/2021
AC PETROBRAS G2	ac_petrobras_g2.crt	20/11/2016
AC Imprensa Oficial G3	ac_imprensa_oficial_g3.crt	27/12/2019
AC DIGITAL	ac_digital.crt	20/06/2023
AC Notarial RFB G2	ac_notarial_rfb_g2.crt	30/10/2016
Autoridade Certificadora da Casa da Moeda do Brasil v2	autoridade_certificadora_da_casa_da_moeda_do_brasil_v2.crt	28/12/2021
Autoridade Certificadora Raiz Brasileira v1	icp-brasil.crt	29/07/2021

Autoridade Certificadora	Arquivo	Data de expiração
AC Certisign G3	ac_certisign_g3.crt	02/09/2018
AC PRODEMGE G2	ac_prodemge_g2.crt	29/10/2016
AC SERPRO-JUS v4	ac_serpro_jus_v4.crt	23/12/2019
AC SINCOR G3	ac_sincor_g3.crt	06/11/2019
Autoridade Certificadora da Justica v4	autoridade_certificadora_da_justica_v4.crt	22/11/2021
AC Imprensa Oficial SP G4	ac_imprensa_oficial_sp_g4.crt	21/06/2023
AC Imprensa Oficial G2	ac_imprensa_oficial_g2.crt	15/05/2019
SERASA Autoridade Certificadora v2	serasa_autoridade_certificadora_v2.crt	05/10/2019
AC Certisign-JUS G3	ac_certisign_jus_g3.crt	23/12/2019
AC SINCOR RFB G4	ac_sincor_rfb_g4.crt	26/01/2020
AC CAIXA SPB	ac_caixa_spb.crt	02/12/2021
AC CNDL RFB v2	ac_cndl_rfb_v2.crt	11/10/2021
Autoridade Certificadora SERPRO v3	autoridade_certificadora_serpro_v3.crt	21/10/2021
AC Certisign SPB G5	ac_certisign_spb_g5.crt	22/09/2019
Autoridade Certificadora Raiz Brasileira v1	ac_raiz_v1.crt	29/07/2021
AC SERASA-JUS v2	ac_serasa_jus_v2.crt	28/12/2019
AC Imprensa Oficial SP RFB G4	ac_imesp_rfb_g4.crt	11/10/2021
AC Certisign RFB G4	ac_certisign_rfb_g4.crt	22/12/2019
AC SAFEWEB RFB	ac_safeweb_rfb.crt	11/10/2021
AC SERPRO-JUS v3	ac_serpro_jus_v3.crt	19/06/2017
AC SINCOR G2	ac_sincor_g2.crt	29/10/2016
AC FENACON Certisign RFB G3	ac_fenacon_certisign_rfb_g3.crt	22/12/2019
AC Certisign Multipla G3	ac_certisign_multipla_g3.crt	08/09/2016
AC Imprensa Oficial SP G3	ac_imprensa_oficial_sp_g3.crt	21/12/2021
Autoridade Certificadora da Justica v3	autoridade_certificadora_da_justica_v3.crt	12/06/2019
SERASA Certificadora Digital v2	serasa_certificadora_digital_v2.crt	13/10/2019
AC FENACOR v1	ac_fenacor_v1.crt	04/08/2016
AC Imprensa Oficial SP RFB G4	ac_imprensa_oficial_sp_rfb_g4.crt	11/10/2021
SERASA Autoridade Certificadora v1	serasa_autoridade_certificadora_v1.crt	04/08/2016
AC Instituto Fenacon RFB G2	ac_instituto_fenacon_rfb_g2.crt	26/12/2019

Parte VI. Demoiselle CA ICP-Brasil Homologação

O Demoiselle CA ICP-Brasil Homologação fornece uma implementação para validação de um conjunto de Autoridades Certificadoras de certificados digitais de Homologação.

Configuração do CA ICP-Brasil Homologação

15.1. Instalação do componente

Para instalar o componente *Demoiselle CA ICP-Brasil Homologação* na aplicação, basta adicionar a sua dependência no arquivo `pom.xml` do projeto gerenciado pelo Maven:

```
<dependency>
    <groupId>br.gov.frameworkdemoiselle.component</groupId>
    <artifactId>demoiselle-certificate-ca-icpbrasil-homologacao</artifactId>
    <version>1.0.11</version>
</dependency>
```

15.2. Autoridades Certificadoras

Abaixo mostraremos a lista de autoridades certificadoras atual que está presente no componente:

Tabela 15.1. Lista de Autoridades Certificadoras

Autoridade Certificadora	Arquivo
Autoridade Certificadora Raiz de Homologacao SERPRO	RaizdeHomologacaoSERPRO.cer
Autoridade Certificadora Intermediaria HOMv2	IntermediariaHOMv2.cer
Autoridade Certificadora ACSERPROACFv3 Homologacao	ACSERPROACFv3Homologacao.cer