Anexo III

Manual del programador

Sistema de Prestación de Servicios de Certificación Electrónica

Trabajo de Fin de Grado

Grado en Ingeniería Informática

Autor

Tomás Calderón López

Tutor/a

Ángel Luis Sánchez Lázaro

Tabla de contenido

Introducción

El objetivo primordial de este Anexo es proporcionar documentación técnica exhaustiva que sirva como una herramienta esencial para que los desarrolladores puedan comprender y trabajar con el código fuente del proyecto. Durante el transcurso de este documento, se llevará a cabo un análisis detallado de la estructura general del sistema, así como de las diversas dependencias y componentes.

La documentación técnica desempeña un papel fundamental en el proceso de desarrollo de software, ya que no solo facilita la comprensión del código existente, sino que también agiliza la colaboración entre los miembros del equipo de desarrollo. Proporciona una visión completa y organizada del sistema, permitiendo a los desarrolladores identificar rápidamente las áreas relevantes, las interacciones clave y las dependencias críticas.

A lo largo del trabajo se revisará:

- Dependencias
- Estructura
- Explicación del código

1. Dependencias

Las dependencias son gestionadas por Maven, una herramienta de gestión de proyectos utilizada en el desarrollo de software. Su objetivo principal es simplificar la construcción, gestión y distribución de proyectos de software. Se declaran en el archivo *pom.xml* y son las siguientes:

thymeleaf-extras-springsecurity6: Proporciona integración de Thymeleaf con Spring Security 6 para la generación de vistas seguras.

- **Java Mail:** Proporciona capacidades de envío de correo electrónico.

- **My SQL Connectort:** Permite la conexión a una base de datos MySQL desde tu aplicación Spring Boot.

- **Thymeleaf:** Incluye Thymeleaf como motor de plantillas para las vistas web.

- **Spring Security:** Agrega el soporte de Spring Security para la autenticación y autorización.

- **Spring JPA:** Facilita el uso de JPA (Java Persistence API) para acceder y gestionar datos relacionales.

- **Spring Web** Proporciona las dependencias necesarias para desarrollar aplicaciones web con Spring Boot.

- **Spring DevTools** Herramientas de desarrollo de Spring Boot que facilitan la recarga automática y la configuración.

- **h2:** Una base de datos en memoria que se utiliza comúnmente para pruebas y desarrollo.

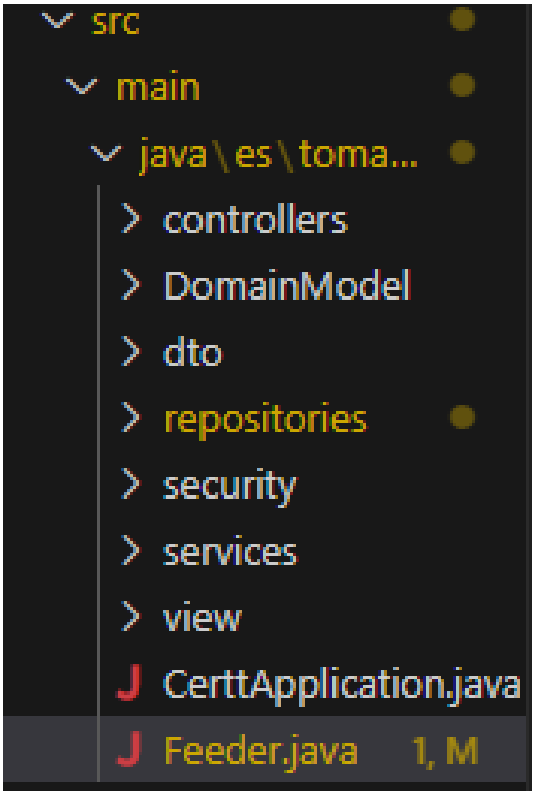
- **Spring Starter** Contiene dependencias para realizar pruebas unitarias y de integración en Spring Boot.

- **BouncyCatstle:** Bibliotecas de código abierto de Java de algoritmos criptográficos, útiles para operaciones de seguridad, como cifrado y firma digital.

2. Estructura

La estructura del proyecto se divide en las siguientes secciones:

- La carpeta *src*:

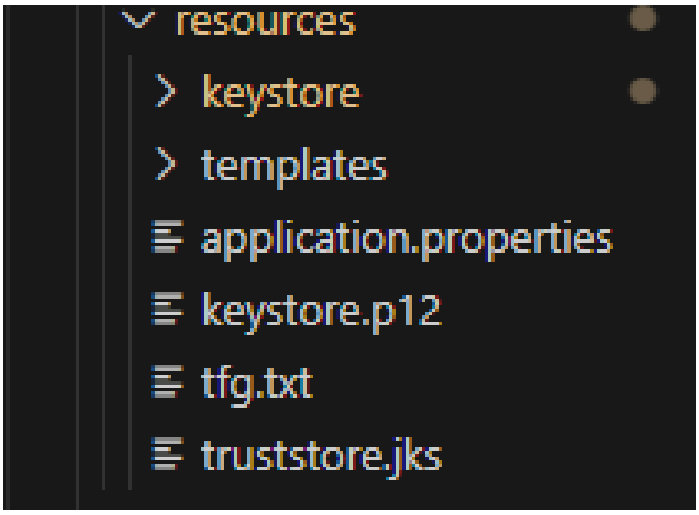


En ella se encuentra el código fuente, donde a su vez están:

- **Controllers:** En esta carpeta se encuentran las clases del controlador.
- **DomainModel:** En esta se encuentra el modelo de dominio, en el que solamente está el Usuario.
- **Dto:** Esta carpeta es de "Data Transfer Objects", clases que sirven como plantilla de los datos para las peticiones y respuestas entre controlador y modelo.
- **Repositories:** Esta carpeta contiene la interfaz con la base de datos de las entidades del dominio, en este caso el Usuario otra vez.
- **Security:** Toda la configuración relacionada con la seguridad, permisos y autenticación, así como gestión de los usuarios.

- **Services:** Servicios que implementan toda la lógica del proyecto.
- **View:** Solo hay un archivo View que gestiona las peticiones de la vista.
- **CertAplication.java:** Es el main, es autogenerado al crear el proyecto.
- **Feeder.java:** Entorno de pruebas para probar implementaciones del modelo de negocio en primeras fases.

- La carpeta *resources*:



En esta carpeta se encuentran los recursos que va a utilizar el sistema para funcionar:

- **Keystore:** En esta carpeta se guardan los keystores (almacenes de claves) de los usuarios, que contienen certificados personales y sus claves privadas, para crear y dejar a disposición de descarga.
- **Templates:** en esta carpeta se encuentra el conjunto de las plantillas html que componen la interfaz del usuario con el sistema.
- **Application.properties:** archivo de configuración de propiedades del sistema. (SSL, truststore…).
- **Keystore.p12:** Este es el almacén de claves donde se encuentran los certificados que utiliza el servidor para habilitar HTTPS y para expedir certificados. Tiene dos certificados con extensión de CA de las mismas características, cuya diferencia es el algoritmo de creación (uno RSA y el otro ECDSA) que permiten crear certificados personales creados con sendos algoritmos.

- **Truststore.jks:** Es el almacén de confianza del sistema, hace que el sistema solo acepte la autenticación ssl proveniente de los certificados en los que confíe. En este se confía solamente en los dos certificados mencionados anteriormente.

3. Explicación del código

Se presenta en el mismo orden anterior:

Controllers

Los controladores conectan el modelo de negocio con la vista. Con frecuencia aparecerán los argumenos *Authentication* que sirve para obtener las credenciales del usuario autenticado en el sistema, y *Model*, que es la clase donde se asignan valores de la forma {clave, valor}, y son devueltos a la vista en cada respuesta, donde luego pueden ser acceidos usando las claves de la forma ${clave} con Thymeleaf. No se explicarán de forma densa e individual, pues por lo general son mediadores entre el cliente y el modelo de negocio.

AlmacenController.java

```
public String getCertificados(Authentication
authentication, Model model)
```

Esta función es la llamada que devuelve la lista de certificados de un usuario en el modelo.

```
public String crearCertificadoPersonal(Authentication
authentication, @ModelAttribute DatosCertificadoDTO
campos)
```

Esta función es la llamada que envía los atributos de un certificado recibidos por parte del usuario en la solicitud de una creación al modelo de negocio.

```
public ResponseEntity<Resource>
descargarCertificado(Authentication authentication,
@RequestParam("alias") String alias)
```

Esta función es algo más compleja, ya que se le solicita la descarga de un certificado, buscado por el alias, cuando el modelo de negocio termina la búsqueda de un certificado, se recibe en esta función y lo pone a disposición de descarga como un Resource.

UsuarioController.java

En este controlador se puede observar la tónica de recibo de parámetros por parte del cliente con los DTO. Simplemente con la etiqueta @ModelAttribute, Spring se encarga de relacionar los datos enviados desde el cliente con su correspondiente atributo en el servidor.

```
public String registrarUsuario(@ModelAttribute
UsuarioDTO usuario)
```

Esta llamada es la petición de registro de un usuario

```
public String verificarUsuario(@RequestParam String c)
```

Esta llamada es la petición de verificación de un usuario. Esta en concreto recoge el parámetro por URL. La razón fuera que el usuario pudiera enviar el código simplemente pinchando en el enlace generado del correo.

```
public String logoutPage (HttpServletRequest request,
HttpServletResponse response)
```

Esta llamada es la petición de fin de sesión de un usuario

FirmaController.java

```
public String postCrearFirma(@ModelAttribute
CrearFirmaDTO cf, Model model)
```

```
public String postComprobarFirma(@ModelAttribute
ComprobarFirmaDTO cf, Model model)
```

Estas llamadas son las que gestionan la creación y comprobación de las firmas

DomainModel

Del modelo de dominio no hace falta destacar mucho, ya que solo lo compone el usuario.

Usuario.java

```
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(unique = true)
    private String email;
    private String nombre;
    private String contrasena;

    @ElementCollection(fetch = FetchType.EAGER)
    private List<String> roles;

    @Column(unique = true)
    private String codigo;
```

DTO

DTO es la abreviatura de "Data Transfer Object". Un DTO es un patrón de diseño utilizado en desarrollo de software para representar un objeto que transporta datos entre diferentes componentes de un sistema. Su propósito principal es encapsular un conjunto de datos y enviarlos de un lado a otro dentro de una aplicación sin exponer la estructura interna de los datos ni su lógica de negocio.

Al usar DTOs, se puede controlar qué datos se exponen y cómo se presentan, lo que ayuda a mejorar la eficiencia y seguridad de la transferencia de datos, así como a reducir la complejidad y la dependencia entre las capas del sistema.

- **CertificadoDTO.java:** Gestiona los datos de búsqueda de un certificado al descargarlo
- **ComprobarFirmaDTO.java:** Gestiona los datos de la comprobación de una firma
- **CrearFirmaDTO.java:** Gestiona los datos de la creación de una firma
- **DatosCertificado.DTO.java:** Gestiona los datos de la creación de un certificado

- **UsuarioDTO.java:** Gestiona los datos de registro de un usuario

Repositories

UsuarioRepository.java

```
@Repository
public interface UsuarioRepository extends
JpaRepository<Usuario, Long> {

    Optional<Usuario> findByEmail(String email);

    Optional<Usuario> findByCodigo(String codigo);
}
```

Los repositorios son interfaces que sirven para gestionar las bases de datos, en este caso del usuario se han implementado dos funciones, búsqueda por email y búsqueda por código para verificación.

Security

Este apartado se explica de forma extensa en la memoria principal, pues tiene que ver con la configuración general de la seguridad del sistema.

Services

En este proyecto, es en los servicios donde está implementada la lógica y el modelo de negocio. Todos ellos son llamados por controladores, procesan los datos y devuelven un resultado.

CertificadoService.java

```
public KeyStore
crearCertificadoPersonal(DatosCertificadoDTO dc, String
alias)
```

Esta función es la que expide un certificado personal con el certificado de CA del sistema, el usuario introduce los datos y el algoritmo a elegir y se crea aquí con esas características.

ClaveService.java

```
public KeyPair generarClaves(int longitudClave,
String algoritmo)
```

Este servicio se ha creado como una simplificación de la gestión de claves en java, solo tiene una función y es la de generar un par de claves dado un algoritmo y una longitud.

EmailService.java

Este servicio únicamente gestiona el cliente de correo de Java para enviar un mail de confirmación generado con un código único para que el usuario entre en un enlace con el código generado y se verifique.

```
public void correoConfirmacion(String destinatario,
String codigoConfirmacion, String nombre)
```

FirmaService.java

Este servicio realiza las operaciones de crear y verificar las firmas. Tiene dos operaciones auxiliares que convierte las claves en formato legible introducidas por el usuario en operables por la biblioteca BouncyCastle.

```
public String crearFirma(CrearFirmaDTO cf)
public boolean comprobarFirma(ComprobarFirmaDTO cf)
private PrivateKey stringKeyToPrivateKey(String pk)
private PublicKey stringKeyToPublicKey(String
publicKeyString)
```

GestorAlmacenesService.java

Esta interfaz simplifica las operaciones de instanciar Keystores y abrir y cerrar archivos, permite cargarlos en memoria y guardar en almacenamiento. Las dos ultimas son específicas de los usuarios, que les permite recuperar sus certificados o añadir uno a su almacén. Por cada usuario, se gestiona un fichero único y protegido con la contraseña cifrada del usuario.

```
public void cargarAlmacen(String email, String
rutaAlKeyStore, String contrasena)
public void guardarAlmacen(String contrasena)
public List<X509Certificate> getCertificados()
public void anadirCertificadoPersonal(String alias,
KeyStore ks, String contraseña)
```

LeerPKCSService.java

Este servicio es utilizado por los demás para extraer y leer las claves de los certificados de autoridad del servidor, de ahí el por qué de que sean estáticos. Se pueden extraer tanto ambas claves como el certificado. El argumento algoritmo existe por el hecho de que son dos los certificados que hay en el sistema, uno firmado con RSA y otro con ECDSA, por lo que la lectura ha de ajustarse según el tipo de algoritmo.

```
public static PrivateKey
leerClavePrivadaKeyStore(String algoritmo)
public static PublicKey leerClavePublicaKeyStore(String
algoritmo)
public static X509Certificate
leerCertificadoKeyStore(String algoritmo)
```

UsuarioService.java

Estas funciones son las que gestionan el registro completo, desde el almacenamiento en la base de datos y la comprobación del email hasta que el usuario se verifica correctamente con su código.

```
public static boolean isValidEmail(String email)
public boolean registroUsuario(Usuario u)
public boolean verificarCuentaUsuario(String codigo)
```

View

En este último apartado, se va a poner como ejemplo la operación de Crear una Firma para explicar le tónica de los flujos de ejecución, ejecutados de forma similar durante toda la aplicación web:

Interfaz (HTML):

```
        <h2>Crear firma digital</h2>
        <form method="post" action="/crear-firma">
          <div class="form-group">
            <label for="textoAFirmar">Texto a firmar</label>
            <textarea
              class="form-control"
              name="textoAFirmar"
              rows="4"
              placeholder="TEXTO A FIRMAR"
            ></textarea>
          </div>
          <div class="form-group">
```

```
            <label for="clavePrivada">Clave privada</label>
            <textarea
              class="form-control"
              name="clavePrivada"
              rows="4"
              placeholder="CLAVE PRIVADA:
              -----BEGIN PRIVATE KEY-----
              -----END PRIVATE KEY-----
                          o
              -----BEGIN RSA PRIVATE KEY-----
              -----BEGIN RSA PRIVATE KEY-----"
            ></textarea>
          </div>
          <div class="form-group">
            <label>Resultado</label>
            <textarea
              th:text="${firma}"
              class="form-control"
              rows="4"
              placeholder="RESULTADO"
              readonly
            ></textarea>
          </div>
          <button type="submit" class="btn btn-primary">Firmar</button>
        </form>
      </div>
    </div>
```

En primer lugar, se muestra un formulario HTML que consta de tres campos: "Texto a firmar", "Clave privada" y "Resultado". El usuario puede ingresar un texto que desea firmar y una clave privada en los campos correspondientes.

Envío del Formulario:

Cuando el usuario hace clic en el botón "Firmar" del formulario, se envía una solicitud POST al servidor. La acción de formulario está definida como /crear-firma, lo que significa que la solicitud POST se enviará a esa URL en el servidor.

Controlador Spring (@PostMapping):

```
    @PostMapping("/crear-firma")
```

```
    public String postCrearFirma(@ModelAttribute
CrearFirmaDTO cf, Model model) {
        String firma = fs.crearFirma(cf);
        model.addAttribute("firma", firma);
        return "crear-verificar-firmas";
    }
```

En el controlador de Spring, se tiene un método anotado con @PostMapping("/crear-firma"). Este método espera recibir los datos del formulario, que se mapearán automáticamente al objeto *CrearFirmaDTO (@ModelAttribute CrearFirmaDTO cf).*

CLAVE: Los datos se mapean según la coincidencia entre los nombres de las propiedades del DTO y los identificadores introducidos en el *name* de HTML.

El método *fs.crearFirma(cf)* es invocado, donde *fs* es un servicio que se encarga de crear la firma digital utilizando el *CrearFirmaDTO* proporcionado.

Generación de Firma Digital:

El servicio *fs.crearFirma(cf)* toma los datos del CrearFirmaDTO (que incluye el texto a firmar y la clave privada) y realiza el proceso de generación de la firma digital.
El resultado de este proceso es una firma digital, que se almacena en la variable firma.

Actualización del Modelo:

La firma digital generada se agrega al modelo utilizando *model.addAttribute("firma", firma)*. Esto significa que la firma estará disponible para su presentación en la vista.
Vista HTML (crear-verificar-firmas):

Después de que el controlador ha terminado de procesar la solicitud, redirige al usuario a una vista llamada "crear-verificar-firmas".

En la vista, se utiliza th:text="${firma}" para mostrar la firma digital generada al usuario.
El usuario verá la firma digital en el campo de "Resultado" del formulario.

Resumen de la técnica

Esta técnica de recibir datos del formulario, procesarlos en el controlador y presentar los resultados en la vista es un patrón común en aplicaciones web basadas en Spring Framework. Se aplica en múltiples partes del proyecto donde se necesite interactuar con los usuarios a través de formularios web y procesar los datos ingresados por ellos.