ChatGPT

BitSealer – Documentación Técnica

1. Nombre del Proyecto y Descripción General

BitSealer es una aplicación web desarrollada con Spring Boot que permite a los usuarios "sellar" digitalmente archivos mediante la generación y almacenamiento de un hash criptográfico único por archivo. En otras palabras, el sistema calcula la huella digital (hash SHA-256) de cada archivo subido y la guarda junto con los datos del usuario, creando un registro permanente en la base de datos. De este modo, BitSealer proporciona una forma de verificar la integridad y existencia de un archivo en un momento dado sin necesidad de conservar el archivo original. La aplicación ofrece una interfaz web segura donde los usuarios pueden registrarse, iniciar sesión, subir archivos para sellado y consultar un historial de sus archivos sellados, todo ello dentro de un entorno corporativo protegido.

BitSealer está diseñada como una herramienta interna en una empresa tecnológica para evidenciar contenido de archivos (por ejemplo, para propósitos de integridad de datos, certificación de autoría o control de versiones). Gracias a su arquitectura basada en Spring Boot, la aplicación es **autosuficiente** (contiene servidor web embebido) y fácilmente desplegable en diversos entornos, incluyendo contenedores Docker. Su objetivo general es brindar a los desarrolladores y sistemas de IA colaboradores un contexto completo del proyecto, abarcando desde la lógica funcional hasta los detalles de implementación técnica.

2. Objetivo Funcional y Justificación Técnica

Objetivo Funcional: El objetivo principal de BitSealer es ofrecer un servicio sencillo pero seguro para generar un "sello digital" de archivos. Un usuario autenticado puede subir un archivo y obtener su hash SHA-256, el cual queda almacenado en la base de datos asociado a su cuenta y marca temporal. Esto permite comprobar posteriormente que el archivo original no ha sido modificado (integridad) y demostrar que dicho archivo existía en el momento del sellado. Es útil, por ejemplo, en casos donde se requiera certificar la existencia de un documento, código fuente o evidencia digital en una fecha determinada sin almacenar el contenido completo. El sistema proporciona funcionalidades de registro de usuarios, autenticación, carga de archivos y consulta de historial, todas integradas de forma coherente para cumplir este propósito.

Justificación Técnica: Para lograr lo anterior de manera eficiente y mantenible, se han elegido tecnologías y un diseño arquitectónico acordes:

- Se utiliza **Spring Boot** (versión 3, sobre Java 17) como base, ya que **facilita el desarrollo de aplicaciones Spring** con mínima configuración y despliegue automatizado [1]. Esto nos permite enfocarnos en la lógica de negocio (el sellado de archivos) sin invertir tiempo excesivo en configurar servidores o dependencias manualmente. Spring Boot también soporta crear aplicaciones **monolíticas autosuficientes** (con servidor web embebido), ideales para una herramienta interna de este alcance.

- La aplicación sigue una arquitectura **monolítica de múltiples capas** (ver sección de Arquitectura) que simplifica el desarrollo y despliegue. Dado el tamaño del proyecto y su uso interno, una arquitectura monolítica es suficiente y evita la complejidad innecesaria de un

enfoque de microservicios. Todos los componentes (UI, lógica de negocio y acceso a datos) residen en un solo artefacto, comunicándose mediante llamadas directas en memoria, lo que reduce la latencia y la complejidad operativa.

- Se eligió **PostgreSQL** como base de datos relacional para almacenar de forma fiable la información de usuarios y hashes de archivos. PostgreSQL es robusto y ampliamente utilizado en entornos empresariales, garantizando consistencia en transacciones y soporte para consultas SQL avanzadas que podrían requerirse (por ejemplo, búsquedas de hashes duplicados, históricos por usuario, etc.). Además, el **driver JDBC de PostgreSQL** se integra fácilmente con Spring Data JPA.

- Para acceder a la base de datos de manera eficiente y con poco código "boilerplate", se emplea **Spring Data JPA** (Hibernate) a través de repositorios. Esto permite mapear las tablas de la base de datos a entidades Java y realizar operaciones CRUD y consultas con mínimo código, manteniendo la capa de persistencia limpia y fácil de mantener.

- La decisión de **no almacenar los archivos binarios** en la base de datos (sólo sus hashes) está justificada técnicamente para mantener el sistema ligero y escalable. Al guardar únicamente el hash (y nombre) de cada archivo, el almacenamiento requerido es mínimo y las operaciones de comparación/verificación son muy rápidas. Si fuera necesario conservar el archivo completo, podría integrarse en el futuro un almacenamiento externo (por ejemplo, un servicio de objetos tipo S3) o un sistema de archivos en servidor, pero el alcance actual del proyecto prioriza la **huella digital** sobre el contenido en bruto.

- Se implementó **Spring Security** para la autenticación y control de acceso. Dado que BitSealer maneja datos sensibles (hashes que corresponden a archivos potencialmente confidenciales), es fundamental que sólo usuarios autorizados puedan acceder a ciertas funciones (subir archivos o ver historiales). Spring Security proporciona un marco robusto para manejar sesiones, credenciales y restricciones de URL de forma declarativa, reduciendo riesgos de seguridad.

- La elección de **BCrypt** para codificar las contraseñas de usuarios se debe a que es un algoritmo de hashing unidireccional seguro y ampliamente adoptado. BCrypt incorpora sal aleatoria y factor de costo, lo que lo hace resistente a ataques de fuerza bruta y de diccionario (es deliberadamente computacionalmente más costoso para un atacante) [2]. De este modo, las contraseñas se guardan **hashed** en la BD, nunca en texto plano, alineándose con las buenas prácticas de seguridad.

- Para gestionar la evolución del esquema de base de datos de forma automatizada y segura, se integra **Flyway** como herramienta de migraciones. Flyway aplica en el arranque de la aplicación cualquier cambio pendiente en el esquema mediante scripts versionados, garantizando que todos los entornos tengan la estructura de datos consistente. Esto evita errores manuales y hace más sencillo el mantenimiento evolutivo del proyecto. Flyway facilita tanto el montaje inicial de la base de datos como las actualizaciones en siguientes versiones [3], aportando control de versiones al esquema y trazabilidad de cambios.

- Se adoptó una estrategia de **despliegue con contenedores Docker** para lograr portabilidad y facilidad de configuración en distintos entornos (desarrollo, pruebas, producción). El proyecto incluye un **Dockerfile** multi-stage que construye la aplicación y genera una imagen ligera de producción, así como un **docker-compose.yml** que orquesta el despliegue conjunto de la aplicación Spring Boot y un contenedor PostgreSQL. Esto asegura que los desarrolladores

puedan levantar la aplicación localmente en un entorno idéntico al de producción con un solo comando, eliminando problemas de "funciona en mi entorno". Además, Docker simplifica la colaboración con herramientas de CI/CD y despliegues en cloud o en servidores on-premise, aislando dependencias del sistema operativo.

En resumen, cada decisión técnica (frameworks, base de datos, seguridad, migraciones, contenedores) ha sido tomada para respaldar el objetivo funcional de BitSealer: brindar un servicio seguro de sellado de archivos, que sea fácil de desarrollar, **mantenible a largo plazo** y reproducible en diferentes entornos por un equipo de desarrollo profesional o incluso por sistemas automatizados.

3. Tecnologías Utilizadas

BitSealer hace uso de un **stack tecnológico moderno** centrado en el ecosistema Spring de Java, complementado con herramientas de base de datos, seguridad y despliegue. A continuación se listan las principales tecnologías y bibliotecas empleadas, junto con una breve explicación de su rol y la razón de su elección:

- **Java 17**: Lenguaje de programación utilizado. Se opta por Java 17 por ser una versión LTS (soporte de largo plazo) que aprovecha las mejoras modernas del lenguaje manteniendo estabilidad. Es además requerida para Spring Boot 3.x.

- **Spring Boot 3.x**: Framework principal para la aplicación. Spring Boot permite crear aplicaciones standalone con configuración por convenciones, servidor web embebido y dependencias preconfiguradas. Simplifica el arranque y despliegue de la app al **proporcionar auto-configuración** y múltiples starters (módulos preintegrados) [1]. En este proyecto se usan varios *starters*:

- **spring-boot-starter-web**: Incluye Spring MVC y servidor Tomcat embebido, facilitando la creación de controladores web y APIs REST (aunque en BitSealer principalmente se devuelven vistas Thymeleaf).
- **spring-boot-starter-data-jpa**: Provee integración con JPA/Hibernate para la capa de persistencia. Permite definir entidades y repositorios para operaciones con la base de datos de forma declarativa.
- **spring-boot-starter-security**: Añade Spring Security al proyecto para gestionar autenticación y autorización de usuarios.
- **spring-boot-starter-thymeleaf**: Incorpora el motor de plantillas Thymeleaf para generar las vistas HTML en el servidor.
- **spring-boot-starter-validation**: Ofrece soporte a la especificación Bean Validation (Jakarta Validation) para anotaciones de validación en entidades y DTOs (por ejemplo, `@NotNull` en el formulario de subida de archivos).

- Adicionalmente, se utilizan **spring-boot-devtools** (para agilizar el desarrollo con recarga automática) y **spring-boot-configuration-processor** (para facilitar la documentación de propiedades configurables).

- **Thymeleaf 3 + Thymeleaf SpringSecurity Dialect**: Motor de plantillas HTML utilizado para las páginas de la aplicación (login, registro, home, subir archivo, historial). Thymeleaf permite enlazar fácilmente datos del modelo enviados por los controladores al markup HTML de las vistas. Además, con la extensión Spring Security para Thymeleaf, es posible mostrar/ocultar elementos en función de la autenticación o rol del usuario (por ejemplo, mostrar el nombre del usuario logueado, opciones de logout, etc.). Se eligió Thymeleaf por su integración fluida con

Spring Boot y por permitir la creación de vistas mantenibles usando fragmentos y dialectos personalizados.

- **PostgreSQL 14**: Sistema de gestión de bases de datos relacional utilizado para almacenar de forma persistente la información de usuarios y hashes de archivos. PostgreSQL fue elegido por su fiabilidad, consistencia ACID, soporte de tipos avanzados y porque es bien soportado por Hibernate (dialecto estable). La estructura de datos es sencilla (dos tablas principales), por lo que un RDBMS encaja perfectamente, aprovechando claves primarias, foráneas y consultas SQL estándar para relacionar usuarios con sus archivos. La versión 14 es usada en los contenedores Docker (imagen oficial **postgres:14-alpine**), garantizando ligereza y buen rendimiento.

- **Spring Data JPA (Hibernate)**: Capa de acceso a datos mediante ORM. Las clases del paquete `model` están anotadas con JPA (`@Entity`) representando las tablas **users** y **file_hashes**, y los repositorios (interfaces que extienden `JpaRepository`) proveen métodos CRUD y consultas derivadas de sus nombres (por ejemplo, `findByOwnerOrderByCreatedAtDesc`). La ventaja de usar JPA es que el desarrollador se abstrae de escribir SQL repetitivo; se confía en Hibernate para generar las consultas necesarias. Esto agiliza el desarrollo y reduce errores manuales. No obstante, cuando se requiera, es posible definir consultas específicas mediante @Query o usar el API Criteria de JPA. En BitSealer, JPA simplifica operaciones como registrar un nuevo usuario o listar los hashes de un usuario, con una sola línea de invocación al repositorio correspondiente.

- **Flyway 9**: Herramienta de migración de base de datos utilizada para versionar el esquema de manera automatizada. Al iniciar la aplicación, Flyway verifica la base de datos y aplica cualquier script pendiente ubicado en `classpath:db/migration`. En este proyecto, la migración **V1__init.sql** crea las tablas iniciales (`users` y `file_hashes`) y establece sus restricciones (ver sección de Base de Datos). Flyway fue incluido para asegurar que todos los entornos pueden reproducir el esquema exacto necesario y que futuros cambios (nuevas tablas, columnas, etc.) se apliquen ordenadamente. **Flyway facilita mantener y montar entornos** de forma consistente, reduciendo el costo de errores humanos en cambios de BD [3]. Su configuración en Spring Boot se realiza vía propiedades (p. ej. `spring.flyway.enabled=true`, rutas de scripts por defecto) y la dependencia `flyway-core`.

- **Spring Security 6**: Framework de seguridad integrado para autenticación y autorización. Se utiliza para implementar el inicio de sesión de usuarios (formulario de login) y la protección de rutas internas. Spring Security maneja automáticamente aspectos como hashing de contraseñas, gestión de sesión (JSESSIONID), contexto de seguridad y prevención de accesos no autorizados. La configuración personalizada (ver `SecurityConfig`) establece las políticas de acceso: páginas públicas vs. privadas, página de login personalizada, URL de logout, etc. También se provee un servicio (`CustomUserDetailsService`) que adapta nuestros usuarios de la BD al modelo de *UserDetails* que entiende Spring Security para la autenticación. Esto garantiza que sólo usuarios válidos puedan acceder a las funciones de sellado, y evita la exposición de recursos sin autenticar. Spring Security fue la elección natural dado que Spring Boot lo integra fácilmente y provee un **marco robusto** probado en entornos de producción empresariales.

- **BCrypt (PasswordEncoder)**: Algoritmo de hash de contraseñas utilizado a través de la implementación `BCryptPasswordEncoder` de Spring Security. BCrypt aplica hashing con sal aleatoria incorporada y múltiple rondas, haciendo las contraseñas derivadas muy difíciles de revertir o precomputar. Spring Security recomienda BCrypt como mecanismo por defecto para almacenar contraseñas de forma segura [2], por lo que en BitSealer todas las contraseñas de

usuarios se almacenan cifradas con BCrypt (string de 60 caracteres que incluye el *salt* y la versión de algoritmo). Esto agrega una capa esencial de seguridad: aunque un atacante obtuviera acceso a la tabla de usuarios, las contraseñas reales no podrían recuperarse fácilmente desde los hashes.

- **Apache Commons Codec 1.17**: Biblioteca auxiliar usada para calcular el hash SHA-256 de los archivos. En concreto, se emplea `DigestUtils.sha256Hex(inputStream)` de Commons Codec para generar la huella digital en hexadecimal de cada archivo subido, de forma sencilla y fiable. Esta librería es ligera y evita tener que implementar manualmente la lógica de hashing, apoyándose en implementaciones probadas de algoritmos criptográficos (SHA-256 en este caso). Se eligió Commons Codec por conveniencia (es ampliamente usado para tareas de codificación y hashing) y para mantener el código conciso.

- **JUnit 5 & Testcontainers**: Para asegurar la calidad y correctitud del código, el proyecto incluye pruebas automáticas. Se usa **JUnit 5** (incluido en `spring-boot-starter-test`) para la estructura de los tests, y **Testcontainers** con JUnit para pruebas de integración que requieren una base de datos real. Testcontainers permite levantar un contenedor Docker de PostgreSQL durante la ejecución de los tests, de forma que las pruebas de repositorio/servicio se ejecutan contra un entorno de base de datos idéntico al real [4]. Esto aumenta la confianza en que el código funcionará correctamente en producción, ya que se prueban escenarios completos (por ejemplo, la secuencia de guardar un archivo y luego recuperar el historial) en condiciones muy similares a las de runtime. Además, se aprovechan utilidades de Spring Boot Test, como @SpringBootTest para cargar el contexto completo en tests de integración y @DataJpaTest para pruebas enfocadas en la capa de persistencia con una BD en memoria o contenedor.

- **Docker & Docker Compose**: Herramientas de contenedorización y orquestación usadas para despliegue. El **Dockerfile** del proyecto define un *build* multi-etapa: primero compila el proyecto usando Maven en una imagen Maven (fase builder) y luego empaqueta el jar generado en una imagen liviana basada en Eclipse Temurin 17 JRE. Esto produce una imagen final más pequeña, apta para producción. Dentro de la imagen, se expone el puerto 8080 y se establece la variable de entorno `SPRING_PROFILES_ACTIVE=docker` para que la aplicación use la configuración de perfil Docker. Por su parte, **Docker Compose** (archivo `docker-compose.yml`) configura dos servicios:

  - `db`: un contenedor PostgreSQL 14 que inicia con un **volumen** persistente (`db_data`) para almacenar los datos. Se le pasan variables de entorno para crear la base de datos `filehashdb` con el usuario `filehashuser` y contraseña `1234`.
  - `app`: el contenedor de BitSealer construido desde el Dockerfile. Este depende del servicio `db` (garantizando que la BD arranque primero), y se le proveen las variables de conexión (`SPRING_DATASOURCE_URL`, `SPRING_DATASOURCE_USERNAME`, `SPRING_DATASOURCE_PASSWORD`) apuntando al contenedor de BD. También se publica el puerto **8080** del contenedor a **8080** del host para acceder a la aplicación web.

Con Compose, se puede levantar todo el stack con un solo comando, asegurando que la aplicación siempre se ejecuta bajo las mismas condiciones (mismas versiones de Java, DB, configuraciones) sin importar el entorno host. Esto es crítico en un contexto empresarial para evitar inconsistencias entre entornos de desarrollo, prueba y producción.

- **Maven 3.9**: Sistema de construcción y gestión de dependencias. El proyecto utiliza Maven para compilar, ejecutar pruebas y empaquetar el artefacto (jar). Maven facilita reproducir el build en cualquier máquina (gracias al pom.xml con todas las dependencias declaradas) y se integra con

el Spring Boot Maven Plugin para tareas como ejecutar la aplicación (`mvn spring-boot:run`) o construir la imagen de Docker (por ejemplo, usando *Spotify docker plugin* o similares si se hubiera configurado, aunque en este caso Dockerfile + Compose es suficiente). Maven también se emplea en la fase 1 del Dockerfile para compilar dentro del contenedor de builder.

Cabe destacar que, adicionalmente, se incluyen **herramientas front-end** a un nivel básico: la aplicación carga recursos estáticos como CSS y JavaScript para la interfaz. En particular, se incorporó la plantilla de estilo **SB Admin 2** (basada en Bootstrap 4) y librerías como **Font Awesome** (iconos) en la carpeta `static/`. Esto permite que las páginas Thymeleaf tengan un diseño moderno y responsivo sin desarrollar una capa front-end desde cero. Dado que es una aplicación interna, se priorizó reutilizar un template existente (SB Admin 2) ajustándolo a las vistas de BitSealer para ahorrar tiempo de diseño web.

4. Arquitectura del Sistema

Tipo de Arquitectura: BitSealer adopta una arquitectura **monolítica de capas (n-tier)**. Esto significa que todos los componentes de la aplicación (interfaz, lógica de negocio, acceso a datos, etc.) se despliegan juntos en un solo artefacto (una aplicación Spring Boot ejecutándose en un proceso Java), pero lógicamente el código está separado por responsabilidades en capas bien definidas. La arquitectura monolítica se consideró adecuada dado el alcance del proyecto: simplifica el desarrollo y la implementación al evitar la sobrecarga de comunicaciones distribuidas, y favorece la coherencia al estar todo el código en un mismo proyecto.

Capas Principales:

- **Capa de Presentación (Web):** Incluye los *Controllers* Spring MVC y las vistas Thymeleaf. En esta capa se gestionan las interacciones con el usuario: formularios de registro/login, páginas para subir archivos y visualizar historiales. Los controladores reciben las peticiones HTTP, interactúan con la capa de servicio para procesar la lógica y finalmente retornan nombres de vistas (Thymeleaf templates) o redirecciones. Las vistas (archivos `.html`) definen la estructura que verá el usuario en el navegador y están cargadas de forma dinámica con los datos provistos por los controladores mediante el modelo. Esta capa también abarca recursos estáticos (CSS/JS) necesarios para la presentación.

- **Capa de Lógica de Negocio (Servicios):** Implementada por los componentes del paquete `service`. Aquí reside la lógica central de la aplicación: por ejemplo, el proceso de registrar un nuevo usuario (verificar unicidad de email/username, codificar contraseña, asignar rol) o el proceso de sellar un archivo (calcular hash y guardar el registro). Las clases de servicio orquestan operaciones que pueden involucrar múltiples pasos o interacciones con la capa de datos. Mantener esta lógica en servicios (y no en los controladores) permite una mejor organización, facilita la prueba unitaria y posibilita reuso de la lógica en otros contextos (por ejemplo, si en un futuro se expone un API REST, los servicios pueden ser utilizados sin duplicar código).

- **Capa de Acceso a Datos (Persistencia):** Compuesta por las entidades JPA (`model`) y los repositorios (`repository`). Esta capa se encarga de la interacción con la base de datos PostgreSQL. Las entidades son mapeos de las tablas y definen las relaciones (por ejemplo, `FileHash` tiene una relación `@ManyToOne` con `AppUser`). Los repositorios proporcionan operaciones de alto nivel para consultar y modificar los datos (por ejemplo, `save`, `findByOwnerOrderByCreatedAtDesc`). Gracias a Spring Data, muchas de estas operaciones

son abstractas y no requieren implementación explícita. La capa de datos está aislada de la lógica de negocio a través de la interfaz de los servicios, lo que significa que los controladores/servicios no manejan directamente `ResultSet` ni SQL, sino objetos de dominio, lo cual mejora la mantenibilidad.

- **Capa de Configuración y Seguridad:** Adicionalmente, hay componentes de configuración técnica que podrían considerarse una capa transversal. Por ejemplo, `SecurityConfig` y `EncoderConfig` definen aspectos de seguridad y beans globales (como el codificador de contraseñas). Estos componentes configuran el marco (framework) para que las otras capas operen correctamente (p. ej., indicando qué URLs requieren autenticación, o proporcionando el bean `PasswordEncoder` que usarán los servicios).

Patrones de Diseño Relevantes: La arquitectura de BitSealer sigue varios patrones comunes en aplicaciones empresariales Java:

- *MVC (Model-View-Controller):* Los controladores manejan la entrada del usuario, las vistas presentan la salida y los modelos (entidades/DTOs) transportan los datos. Este patrón separa claramente la UI de la lógica y de los datos.
- *DAO/Repositorio:* Encapsulación del acceso a la BD mediante repositorios que actúan como *Data Access Objects*. Esto permite cambiar la implementación de persistencia (por ejemplo, a otro tipo de storage) sin afectar al resto de la aplicación mientras se respeten las interfaces.
- *Dependency Injection:* Spring IoC container inyecta dependencias (servicios en controladores, repositorios en servicios, etc.), promoviendo bajo acoplamiento y alta cohesión. Las anotaciones como `@Service`, `@Repository`, `@Controller` y `@Autowired` se usan para declarar estos componentes y sus inyecciones.
- *DTO (Data Transfer Object):* Se utilizan objetos DTO (por ejemplo `FileHashDto`, `UserDto`) para transferir datos hacia las vistas u otras capas sin exponer directamente las entidades JPA. Esto mejora la seguridad (p.ej. no exponer el campo `password` de AppUser) y la flexibilidad (permite formatear o calcular campos derivados para la vista). En BitSealer, los servicios usan un **mapper** (`FileHashMapper`) para convertir entidades a DTOs antes de pasarlos al controlador/vista.

Comunicación entre Componentes: Al ser monolítica, la comunicación es **local, en memoria**: - Los controladores llaman métodos de servicios directamente dentro del mismo proceso (invocación de métodos Java). - Los servicios interactúan con la base de datos a través de los repositorios, que internamente utilizan la API de JPA/Hibernate para generar SQL y comunicarse vía JDBC con PostgreSQL. Esta comunicación con la BD es síncrona: el servicio espera la respuesta (p. ej., confirmación de guardado o resultados de una consulta) antes de continuar. - Spring Security introduce un *filtro* en la cadena de servlets que intercepta llamadas HTTP antes de llegar al controlador, para verificar autenticación/autorización. Esto sucede dentro del mismo servidor embebido (Tomcat), a nivel de aplicación web. - No existen llamadas remotas ni servicios externos en la arquitectura actual. Todas las funcionalidades residen en la propia aplicación, salvo la conexión a la base de datos.

Arquitectura de Seguridad: En cuanto a seguridad, la aplicación sigue una arquitectura estándar de Spring Security: - Durante el login, Spring Security delega la carga del usuario a nuestro `CustomUserDetailsService` (capa de servicio) y luego compara la contraseña proporcionada con la almacenada usando `PasswordEncoder` (capa de configuración). - Una vez autenticado, la sesión del usuario se mantiene vía cookie HTTP (JSESSIONID) y un *SecurityContext* asociado a la sesión. Cada request posterior verifica ese contexto para decidir si el usuario puede acceder o no, según las reglas definidas (por ejemplo, cualquier URL bajo `/upload` o `/history` requiere estar autenticado). - La aplicación utiliza **sesiones HTTP tradicionales** para mantener la autenticación (estado en el servidor).

No se implementa JWT u OAuth en este proyecto, dado que el caso de uso es un servicio web para usuarios humanos dentro de la empresa, donde el enfoque de sesión con cookie es adecuado.

Diagrama de Flujo Interno (Resumen): Cuando un usuario sube un archivo, por ejemplo, el flujo es: Controlador `FileUploadController` recibe el archivo -> Servicio `FileHashService` calcula hash y guarda entidad -> Repositorio `FileHashRepository` escribe en BD -> Servicio retorna a Controlador -> Controlador redirige a vista de historial -> Controlador de historial (`verHistorial`) obtiene datos -> Servicio/Repositorio leen de BD -> DTOs se pasan a la vista Thymeleaf -> se genera HTML final para el usuario. Todo esto ocurre dentro de la misma aplicación, con transacciones de BD manejadas por Spring (por anotaciones @Transactional por defecto en operaciones JPA). La arquitectura en capas asegura que cada paso esté aislado y sea fácilmente modificable sin romper otras partes (por ejemplo, se podría cambiar la forma de calcular el hash en el servicio sin afectar al controlador, o cambiar la consulta de histórico en el repositorio sin cambiar la vista).

En síntesis, la arquitectura de BitSealer es **monolítica, modular y orientada a capas**, aprovechando las fortalezas del framework Spring Boot para auto-configurar muchas de estas capas y permitir que los desarrolladores se centren en la lógica del "sellado" de archivos. Es una arquitectura simple pero sólida, adecuada para la colaboración entre desarrolladores y su eventual escalamiento o particionado si el proyecto creciera en complejidad.

5. Estructura del Código y Paquetes

El código fuente del proyecto se organiza siguiendo las convenciones estándar de Spring Boot, lo que facilita a cualquier desarrollador familiarizado con Spring ubicarse rápidamente. A continuación se detalla el mapa de directorios y paquetes principales, junto con la responsabilidad de cada uno:

```
bitsealer/  (directorio raíz del proyecto)
├── src/
│   ├── main/
│   │   ├── java/
│   │   │   └── com.bitsealer/
│   │   │       ├── BitSealerApplication.java       # Clase main
(@SpringBootApplication)
│   │   │       ├── config/                         # Configuración de
seguridad y beans
│   │   │       │   ├── SecurityConfig.java           # Configura Spring
Security (HTTP, login, logout)
│   │   │       │   └── EncoderConfig.java            # Define el
PasswordEncoder (BCrypt)
│   │   │       ├── controller/                     # Controladores web
(MVC)
│   │   │       │   ├── AuthController.java           # Controla registro y
login (vistas login.html, register.html)
│   │   │       │   ├── HomeController.java           # Controla la página
de inicio (vista home.html)
│   │   │       │   ├── FileUploadController.java     # Controla formulario
de subida y muestra historial (upload.html, history.html)
│   │   │       │   └── ViewController.java           # (Actualmente sin
mappings; reservado para futuras vistas)
```

```
│   │   │       ├── dto/                             # Objetos de
transferencia de datos (Data Transfer Objects)
│   │   │       │   ├── FileUploadRequest.java         # DTO para datos del
formulario de subida (contiene MultipartFile)
│   │   │       │   ├── FileHashDto.java               # DTO para exponer
hash de archivo (id, nombre, hash, fecha)
│   │   │       │   └── UserDto.java                   # DTO para datos de
usuario (id, username, email) – no incluye contraseña
│   │   │       ├── mapper/                           # Mapeadores entidad-
>DTO
│   │   │       │   └── FileHashMapper.java            # Convierte FileHash
(Entity) a FileHashDto (Record)
│   │   │       ├── model/                            # Modelos de datos
(Entidades JPA)
│   │   │       │   ├── AppUser.java                  # Entidad Usuario ->
tabla "users"
│   │   │       │   └── FileHash.java                 # Entidad Hash de
archivo -> tabla "file_hashes"
│   │   │       ├── repository/                       # Repositorios de base
de datos (DAO)
│   │   │       │   ├── UserRepository.java           # Interfaz CRUD para
AppUser (incluye búsquedas por username/email)
│   │   │       │   └── FileHashRepository.java       # Interfaz CRUD para
FileHash (incluye búsqueda por usuario ordenada por fecha)
│   │   │       └── service/                          # Servicios (lógica de
negocio y utilidades)
│   │   │           ├── UserService.java              # Lógica de registro de
usuarios, búsqueda de usuario
│   │   │           ├── FileHashService.java          # Lógica de procesar
archivo (calcular hash, guardar, listar historial)
│   │   │           └── CustomUserDetailsService.java # Implementación de
UserDetailsService (Spring Security) para autenticación
│   │   ├── resources/
│   │   │   ├── templates/                            # Plantillas Thymeleaf
(vistas web)
│   │   │   │   ├── layout/
│   │   │   │   │   └── main.html                     # Plantilla layout
principal (cabecera, cuerpo, scripts)
│   │   │   │   ├── home.html                         # Página de inicio
(pública)
│   │   │   │   ├── login.html                        # Página de login
(formulario de ingreso)
│   │   │   │   ├── register.html                     # Página de registro de
nuevo usuario
│   │   │   │   ├── upload.html                       # Página con formulario
para subir archivo (usuarios logueados)
│   │   │   │   └── history.html                      # Página de historial
de archivos subidos (usuarios logueados)
│   │   │   ├── static/                               # Recursos estáticos
(accesibles como /css/**, /js/**, /img/**, etc.)
```

```
│   │   │   │   ├── css/                              # Hojas de estilo CSS
│   │   │   │   │   ├── sb-admin-2.css                    # Estilos
principales (tema SB Admin 2)
│   │   │   │   │   └── sb-admin-2.min.css                # Versión
minificada
│   │   │   │   ├── js/                               # Scripts JavaScript
│   │   │   │   │   ├── sb-admin-2.js                     # Scripts del
tema (incluye interacciones UI básicas)
│   │   │   │   │   └── sb-admin-2.min.js                 # Versión
minificada
│   │   │   │   ├── img/                              # Imágenes usadas en
la web (logos, ilustraciones)
│   │   │   │   │   ├── hero-banner.jpg                   # Imagen banner
en portada
│   │   │   │   │   ├── login-side.jpg                   # Imagen
decorativa en pantalla login
│   │   │   │   │   └── ... (otros SVG/PNG para la UI)
│   │   │   │   └── vendor/                           # Librerías externas
estáticas (ej: FontAwesome)
│   │   │   │       └── fontawesome-free/ ...          # Iconos FontAwesome
usados en la interfaz
│   │   │   ├── application.properties               # Configuración por
defecto (perfil "default")
│   │   │   └── application-docker.yml               # Configuración
específica para perfil "docker"
│   │   └── ... (otros archivos de configuración si los hubiera)
│   └── test/                                         # Código de pruebas
(JUnit)
│       ├── java/com.bitsealer/
│       │   ├── BitSealerApplicationTests.java        # Prueba básica de
contexto Spring Boot
│       │   ├── service/FileHashServiceTest.java      # Pruebas unitarias del
servicio de hash de archivos
│       │   └── integration/UploadIntegrationTest.java# Prueba de integración
del flujo de subida (con Testcontainers)
│       └── resources/
│           └── application-test.properties           # Config. específica
para tests (si aplica, por ej. desactivar seguridad en tests)
├── Dockerfile                                       # Definición de imagen
Docker multi-stage (build de Maven + runtime)
├── docker-compose.yml                               # Definición de
servicios Docker (app y db, con red y volumen)
├── .dockerignore                                    # Archivos/dirs
excluidos del contexto Docker (por ejemplo, target/, .git/, etc.)
└── pom.xml                                          # Descripor de proyecto
Maven (dependencias, build plugins, propiedades)
```

Detalles destacados de la estructura:

- La clase principal `BitSealerApplication.java` se encuentra en el paquete base `com.bitsealer`. Esta clase está anotada con `@SpringBootApplication` y contiene el método `main` que arranca la aplicación. Al estar en el nivel raíz de `com.bitsealer`, habilita el escaneo de componentes para todos los subpaquetes (config, controller, service, etc.) automáticamente.

- El paquete `config` contiene la configuración de seguridad. `SecurityConfig.java` define la política de seguridad de la aplicación (URLs permitidas, formulario de login, logout, etc.) y expone un bean `SecurityFilterChain` que Spring Boot autoconfigura para proteger las rutas. También contiene un método (con `@Autowired` en AuthenticationManagerBuilder) para registrar `CustomUserDetailsService` y `PasswordEncoder` en el proceso de autenticación. Por su parte, `EncoderConfig.java` declara el bean `PasswordEncoder` (usando BCrypt). Centralizar esto en un paquete de config sigue el principio de separación de concerns, manteniendo la seguridad aislada de la lógica de negocio.

- El paquete `controller` agrupa los **controladores MVC** de Spring. Todos están anotados con `@Controller` (ya que retornan vistas Thymeleaf en lugar de JSON). Los controladores manejan las rutas web definidas:

- **AuthController:** Maneja la pantalla de login y el registro de usuarios. Tiene métodos `@GetMapping("/login")` (devuelve la vista de login) y `@GetMapping("/register")` (devuelve formulario de registro). Además, un método `@PostMapping("/register")` procesa el formulario de registro: valida la entrada, invoca a `UserService.registerUser` y redirige al login con un indicador de éxito. Es importante destacar que el login en sí (POST a `/login`) lo maneja Spring Security internamente, pero se personalizó la vista de login para que coincida con el estilo de la aplicación.
- **HomeController:** Controla la página de inicio accesible por cualquiera (`@GetMapping({"/", "/home"})`). Carga la vista `home.html`, que sirve de portada (podría incluir información general o un CTA para loguearse/registrarse).
- **FileUploadController:** Gestiona las funcionalidades principales de la aplicación, que requieren autenticación. Tiene:
  - `@GetMapping("/upload")` que prepara un `FileUploadRequest` vacío y retorna la vista `upload.html` (formulario para seleccionar archivo).
  - `@PostMapping("/upload")` que procesa el formulario de subida. Recibe el DTO `FileUploadRequest` con el archivo (MultipartFile) adjunto. Aplica validación (`@Valid`) y, si hay errores o no se seleccionó archivo, redirige de nuevo a `/upload` mostrando un mensaje de error (usando RedirectAttributes para flash attributes). Si todo está correcto, obtiene el usuario autenticado actual, invoca a `fileHashService.saveForUser(owner, file)` para realizar el sellado, y después de guardar exitosamente añade un mensaje de éxito al modelo. Finalmente, hace una redirección Post/Redirect/Get a `/history` para mostrar el historial actualizado (esto previene re-envío del formulario si el usuario refresca la página).
  - `@GetMapping("/history")` que arma la página de historial de archivos para el usuario actual. Recupera del contexto de seguridad el usuario logueado, obtiene mediante `fileHashService.findDtosByUser(user)` la lista de archivos (como DTOs) y las pasa al modelo bajo el atributo `"hashes"`. Luego retorna la vista `history.html`, donde se listarán esos datos. Este método es invocado tras la

redirección del upload, pero también podría accederse manualmente vía URL si el usuario desea ver su historial en otro momento.

- **ViewController:** Actualmente es un controlador vacío (sin métodos) reservado para "vistas muertas" o páginas estáticas que no requieren lógica. En el presente código no tiene ningún mapping activo – posiblemente se creó para manejar rutas adicionales en el futuro sin recargar lógica en otros controladores. Por ahora, no se utiliza y no afecta al funcionamiento (Spring lo detecta pero al no tener anotaciones @RequestMapping, no interviene).

- El paquete `dto` contiene **objetos simples para transferencia de datos**. Estos DTOs actúan como una capa intermedia entre las entidades y las vistas:

- `FileUploadRequest` es una clase tradicional con un campo `MultipartFile file` anotado con `@NotNull`. Es usada como el objeto atado al formulario de subida en `upload.html` (mediante `th:object="${fileUploadRequest}"`). Al enviar el formulario, Spring vincula el archivo subido a esta clase para que el controlador lo reciba ya encapsulado.
- `FileHashDto` es un Java Record que define los datos que se mostrarán por cada archivo en el historial: id, nombre original del archivo, hash SHA-256, y fecha de creación (LocalDateTime). Se utiliza un record para simplificar (inmutabilidad y métodos auto-generados) ya que este DTO es meramente un contenedor de datos.

- `UserDto` es otro record definido para contener id, username y email de un usuario. Actualmente no se usa en las vistas ni controladores, pero podría servir en futuros requisitos (por ejemplo, para exponer datos de usuario vía API o mostrarlos en el frontend sin incluir información sensible como la contraseña). Es parte de la infraestructura preparada para seguir la buena práctica de no exponer entidades directamente.

- El paquete `mapper` tiene el mapeador `FileHashMapper`. Este componente ofrece métodos estáticos o de instancia (según implementación) para convertir listas de entidades FileHash en listas de FileHashDto. Así, en `FileHashService.findDtosByUser` se delega la conversión a este mapper. La presencia de un mapper separado sugiere que en caso de aumentar la complejidad del mapeo (por ejemplo, formateo de fechas, combinación de campos, etc.), la lógica estaría aislada aquí. Si el proyecto creciera, podría integrarse una librería de mapeo automático como MapStruct, pero por ahora la conversión parece trivial (propiedades con mismos nombres).

- El paquete `model` contiene las **entidades JPA**:

- `AppUser` corresponde a la tabla `users`. Incluye campos: id, username, email, password, role. Usa anotaciones JPA como `@Entity`, `@Table(name="users")`, `@Id` con generación IDENTITY, y constraints a nivel objeto (`@Column(unique=true)` para username/email, etc.). Esta clase implementa `Serializable` y define constructores, getters/setters básicos. No implementa `UserDetails` directamente (se optó por el servicio custom para adaptarlo), lo cual está bien ya que desacopla la entidad de la seguridad. **Nota:** No se define en AppUser la relación OneToMany con FileHash; es una entidad *padre* unidireccionalmente no consciente de sus hijos para simplicidad (la relación inversa se maneja vía consultas en el repositorio de FileHash).
- `FileHash` corresponde a la tabla `file_hashes`. Campos: id, sha256 (hash en hex), fileName (nombre original), createdAt (fecha/hora en que se crea el registro) y `owner` (usuario dueño). Está mapeado con `@ManyToOne` hacia `AppUser` usando la columna `user_id` como foreign

key. La propiedad `createdAt` se inicializa con `LocalDateTime.now()` al instanciar el objeto, para registrar automáticamente la marca temporal de creación. Esta entidad tampoco define la relación inversa (No `mappedBy` hacia AppUser), lo que simplifica operaciones de guardado (se guarda el FileHash asociándole un AppUser existente y JPA se encarga de la clave foránea). Cabe destacar que *no* se almacena el archivo en sí, solo el hash y el nombre, cumpliendo con el diseño de mantener la BD ligera.

- El paquete `repository` contiene las **interfaces de repositorio** que extienden de Spring Data JPA:

- `UserRepository` extiende `JpaRepository<AppUser, Long>` y añade métodos de búsqueda derivados: `Optional<AppUser> findByUsername(String username)` y `Optional<AppUser> findByEmail(String email)`, así como `boolean existsByEmail(String email)`. Estos métodos permiten, respectivamente, buscar usuarios por nombre o email (para autenticación y validaciones) y comprobar existencia de email (para validar registro único). Spring Data implementa automáticamente estas consultas a partir del nombre del método.
- `FileHashRepository` extiende `JpaRepository<FileHash, Long>`. Define un método custom: `List<FileHash> findByOwnerOrderByCreatedAtDesc(AppUser owner)`, que recupera todos los hashes de un determinado usuario ordenados por fecha descendente (los más recientes primero). Esto es utilizado en la vista de historial. Nuevamente, Spring Data provee la implementación del query basándose en el nombre del método. Con estas interfaces, los servicios pueden realizar operaciones de persistencia sin escribir SQL ni implementar clases DAO manualmente.

- El paquete `service` incluye los **servicios de negocio**:

- `UserService`: Proporciona funcionalidad relacionada con usuarios. Su método principal es `registerUser(AppUser user)` que encapsula la lógica de registro de un nuevo usuario. Esta función realiza varias tareas atómicas:

    1. Verifica que el email no exista ya (`userRepository.existsByEmail`) – si existe, lanza `IllegalArgumentException` con mensaje de error.
    2. (Opcional) Verifica si el username ya está tomado (`findByUsername`) – si sí, lanza excepción similar. (En la práctica, el username también es único por la base de datos).
    3. Cifra la contraseña en el objeto usuario usando `passwordEncoder.encode(...)`.
    4. Asigna el rol por defecto "ROLE_USER" si no se había proporcionado uno (en este caso, desde el formulario no se provee, así que siempre asignará ROLE_USER).
    5. Guarda el usuario en la base de datos a través del `userRepository.save` y devuelve el usuario persistido.

  Adicionalmente, `UserService` tiene métodos como `getByUsername(String username)` que devuelve un Optional<AppUser> usando `UserRepository.findByUsername`, para buscar usuarios existentes (esto se usa en FileUploadController al obtener el usuario actual por nombre).

- `FileHashService`: Contiene la lógica para manejar archivos y hashes. Principales funciones:
    - `saveForUser(AppUser owner, MultipartFile file)`: Calcula el hash SHA-256 del archivo usando **DigestUtils** de Commons Codec y luego llama a la sobrecarga `saveForUser(owner, String sha256, String fileName)`. La sobrecarga crea un nuevo objeto FileHash, setea sus campos (propietario, hash y nombre de archivo) y lo guarda con `hashRepo.save(fh)`. Devuelve la entidad guardada (aunque en este caso el controlador no la usa directamente, sino que confía en que se guardó).
    - `findDtosByUser(AppUser owner)`: Recupera desde la BD todos los FileHash de ese usuario (`hashRepo.findByOwnerOrderByCreatedAtDesc(owner)`) y luego usa `FileHashMapper` para convertirlos en una lista de FileHashDto, que será enviada a la vista. Este método se utiliza tanto justo después de subir un archivo (para obtener lista actualizada) como al entrar a la página de historial.

  También podría haber métodos auxiliares como `listMineDtos()` o similares (según el código, parece que existía la intención de listarlos sin pasar el usuario, quizás usando el contexto de seguridad internamente), pero lo fundamental son los mencionados. **Nota:** Actualmente no hay control de duplicados de hash en este servicio; es decir, si el mismo usuario sube dos veces el mismo archivo, se almacenarán dos registros con el mismo hash. Podría ser un punto de mejora (evitar duplicados), pero el diseño actual opta por registrar cada intento de subida como entrada separada para mayor trazabilidad.

- `CustomUserDetailsService`: Implementa `UserDetailsService` de Spring Security. Esta clase adapta nuestros datos de usuario para el proceso de autenticación de Spring Security. El método `loadUserByUsername(String usernameOrEmail)` intenta encontrar un AppUser bien sea por username o por email (acepta cualquiera de los dos como identificador de login). Si no encuentra, lanza `UsernameNotFoundException`. Si encuentra, construye un objeto `org.springframework.security.core.userdetails.User` (clase de Spring Security) usando:
    - `user.getUsername()` como nombre,
    - `user.getPassword()` como contraseña (ya cifrada en BD, que Spring Security comparará usando PasswordEncoder automáticamente),
    - `List.of(new SimpleGrantedAuthority(user.getRole()))` para asignar la autoridad/rol del usuario (p. ej. "ROLE_USER").

  Este UserDetails resultante es lo que Spring Security utiliza internamente para validar la contraseña y para representar al usuario en el contexto de seguridad. El hecho de permitir autenticación por email o username mejora la usabilidad. Esta clase se marca como `@Service` y es inyectada en `SecurityConfig` para ser usada durante el login.

- En `resources/templates` se ubican los archivos Thymeleaf que corresponden a las vistas:

- `home.html` es la página de inicio (pública). Puede contener un mensaje de bienvenida y botones/enlaces a Login o Registro.
- `login.html` contiene el formulario de inicio de sesión (campo de usuario o email, campo de contraseña y un botón). Utiliza las características de Spring Security Thymeleaf para mostrar errores de login o mensajes (por ejemplo, tras registrarse, se redirige con `?registered` y posiblemente la página muestra "Registro exitoso, por favor inicia sesión").
- `register.html` contiene el formulario de registro (campos para username, email, contraseña, posiblemente repetición de contraseña si se implementó en la vista, y botón de registro). Esta

vista está asociada al model attribute `"userForm"` que es un objeto AppUser vacío en el GET, y en el POST se liga a los campos enviados. Muestra errores de validación en caso de que falte algún campo o violen restricciones (por ejemplo, en la entidad AppUser podrían haberse anotado constraints como @NotBlank en password, etc., de ser así Thymeleaf los desplegaría).
- `upload.html` contiene el formulario para subir un archivo (campo de tipo file). Está asociado al model attribute `"fileUploadRequest"` y aprovecha la validación: si el controlador redirige con error "Debes seleccionar un archivo", la vista podría mostrar ese mensaje (usando `th:if="${#fields.hasErrors('file')}"` o flash attributes).
- `history.html` muestra la lista de archivos sellados del usuario. Itera sobre la colección `"hashes"` proporcionada por el controlador, y por cada FileHashDto muestra, por ejemplo, el nombre de archivo, el hash SHA-256 (posiblemente truncado para mostrar parcial y copiar completo al portapapeles, etc.), y la fecha/hora de creación. También incluye mensajes flash como éxito tras subir un archivo (e.g., "Archivo sellado correctamente").

- `layout/main.html` es una plantilla base usando Thymeleaf Layout Dialect o inclusión fragmentada. Probablemente define el encabezado HTML, la barra de navegación (que podría mostrar el nombre del usuario logueado y un botón de logout) y un contenedor donde insertar el contenido específico de cada página. Así las vistas `home.html`, `history.html`, etc., extenderían este layout para mantener una apariencia consistente (logo de la empresa, menú, pie de página, etc.).

- En `resources/static` están los archivos estáticos servidos directamente. Como se mencionó, `css/sb-admin-2.css` es la hoja de estilo principal (parte de un tema basado en Bootstrap). Hay imágenes en `img/` usadas en la interfaz (gráficos ilustrativos en login o home). En `vendor/` se encuentra *FontAwesome* (íconos) y posiblemente otras librerías front-end. El `SecurityConfig` está configurado para permitir el acceso anónimo a `"/css/**", "/js/**", "/img/**", "/vendor/**"`, de manera que el navegador pueda cargar estos recursos sin necesidad de estar autenticado.

- Los archivos de configuración:

- `application.properties` es la configuración por defecto (perfil *default*). Allí se define, entre otras cosas, la URL de la base de datos local (ej. `jdbc:postgresql://localhost:5432/filehashdb`), el usuario/contraseña de BD para desarrollo (`filehashuser/1234`), y propiedades JPA como `spring.jpa.hibernate.ddl-auto=none` (para que Hibernate no intente crear/tocar el esquema, dejando esa tarea a Flyway), `open-in-view=false` (para evitar problemas de sesión de Hibernate en vistas debido a lazy loading, forzando a obtener todos los datos necesarios en la capa de servicio) y `show-sql=true` (útil en desarrollo para ver en consola las consultas SQL ejecutadas). También se define `spring.application.name=BitSealer` para identificar la aplicación.
- `application-docker.yml` es una configuración específica para cuando el perfil activo es "docker". En YAML se redefinen principalmente las propiedades de datasource: en lugar de valores fijos, usa variables de entorno (`SPRING_DATASOURCE_URL`, etc.) que Docker Compose inyecta. Así, la cadena de conexión apunta al host `db` (nombre del contenedor PostgreSQL) en lugar de localhost. También en este perfil se ajustan algunos valores para entornos productivos: por ejemplo, `spring.jpa.show-sql=false` (no loguear SQL en prod), `spring.jpa.hikari.maximum-pool-size=5` (limitar el pool de conexiones a BD a 5, suficiente para carga baja-moderada), etc. Este perfil asegura que la aplicación se conecte correctamente a la BD dentro del entorno Docker. La activación de este perfil la hace la variable de entorno `SPRING_PROFILES_ACTIVE=docker` en el Dockerfile al construir la imagen.

- No se incluye un `.env` en el repositorio, pero Docker Compose podría leer variables de un fichero `.env` externo si se quisiera ocultar las credenciales. En este caso, las credenciales de BD de desarrollo están explícitamente en el compose (usuario `filehashuser`, pass `1234`), dado que no son sensibles en un entorno local. En un entorno real, se aconsejaría usar un .env o manejadores de secretos para esas variables.

- Archivos de soporte:

- El `pom.xml` lista todas las dependencias mencionadas (Spring Boot starters, PostgreSQL driver, Flyway, Thymeleaf extras, Testcontainers, etc.), las versiones (por ejemplo, Spring Boot versión 3.5.0, PostgreSQL driver 42.7.x, Flyway core y postgresql, etc.) y plugins de construcción. Contiene también la propiedad `java.version` establecida a 17 para compilar con ese nivel de lenguaje.
- El `Dockerfile` ya comentado define cómo se construye la imagen de la aplicación. Es notable que gracias al `.dockerignore` (que incluye probablemente `target/`, `.git/`, etc.), el contexto de build de Docker es más ligero (no se envían artefactos innecesarios al demonio Docker). La fase 1 ejecuta `mvn package -DskipTests` para generar el jar. La fase 2 copia ese jar al contenedor final minimalista.
- `docker-compose.yml` ya detallado orquesta la aplicación. El volumen `db_data` asegura persistencia de la base de datos entre reinicios (en entorno de desarrollo, para no perder datos de prueba).
- `.gitignore` (no mostrado arriba, pero generalmente presente) seguramente lista archivos a ignorar en el control de versiones (carpeta target, credenciales, etc.), manteniendo el repositorio limpio de binarios.

En resumen, la estructura del proyecto sigue las mejores prácticas de Spring Boot: paquetes separados por capa funcional, nombres autoexplicativos y configuración externalizada. Esta organización permite que un nuevo desarrollador pueda navegar y entender dónde realizar cambios: por ejemplo, cambios en la UI se harían en templates o controllers, cambios en lógica en servicios, cambios en modelo de datos en entidades/migraciones, etc. La separación en capas y el uso de estándares de facto (MVC, DTOs, etc.) también facilita la colaboración con herramientas de IA, ya que éstas pueden razonar sobre módulos cohesivos con responsabilidades claras.

6. Flujo General de Uso

A continuación, se describe el **flujo típico de uso** de BitSealer, cubriendo desde el registro de un usuario hasta la generación de hashes y consulta de historial. Este flujo asume la perspectiva de un **usuario final** (por ejemplo, un desarrollador dentro de la empresa) interactuando con la aplicación web:

1. **Registro de Usuario:** Un nuevo usuario accede a la aplicación (por ejemplo, mediante su navegador en `http://<servidor>:8080/`). En la página de inicio (*Home*), se le presentará la opción de **Iniciar Sesión** o **Registrarse**. Como no tiene cuenta aún, el usuario navega a la página de registro (URL `/register`).
2. En `register.html`, se muestra un formulario solicitando datos como **nombre de usuario**, **correo electrónico** y **contraseña** (BitSealer requiere estos tres campos; la contraseña se introducirá en texto secreto).
3. El usuario rellena sus datos. Al enviar el formulario (botón "Registrarse"), el navegador realiza una petición POST a `/register`.

4. El **AuthController** recibe esta petición y valida los datos. Si algún campo obligatorio está vacío o no cumple los requisitos (por ejemplo, formato de email inválido), el formulario se redisplay con mensajes de error de validación (usando BindingResult y Thymeleaf para marcarlos).
5. Si los datos básicos son válidos, AuthController invoca a `userService.registerUser(userForm)`. Este servicio aplicará las reglas de negocio: verificar que el email no esté ya en uso, que el username sea único, etc. Si alguna de estas condiciones falla (p. ej., el email ya existía), `registerUser` lanzará una excepción y AuthController capturará eso, agregando un mensaje de error ("El e-mail ya está registrado.", etc.) que se muestra en la vista de registro. El usuario tendría que corregir el dato y reenviar.
6. En caso de que todo esté correcto, `registerUser` guarda el nuevo usuario en la base de datos (con la contraseña cifrada) y retorna el usuario creado. El AuthController entonces redirige al usuario a la pantalla de login con un indicador de éxito. Concretamente, hace `return "redirect:/login?registered"` – la presencia del parámetro `registered` en la URL puede ser usada por la vista de login para mostrar un mensaje tipo "Registro exitoso, ahora puedes iniciar sesión".

7. *Resultado:* El usuario ha creado su cuenta. Aún no está autenticado automáticamente después del registro, por lo que el siguiente paso es iniciar sesión.

8. **Inicio de Sesión (Login):** El usuario, tras registrarse o al visitar la app posteriormente, accede a la página de login (`/login`).

9. La vista `login.html` despliega un formulario pidiendo **usuario o email** y **contraseña**. BitSealer permite que en el campo de usuario se introduzca indistintamente el nombre de usuario elegido o la dirección de correo, para mayor comodidad.
10. El usuario introduce sus credenciales. Al enviar, se hace POST a `/login` (la misma URL, manejada internamente por Spring Security, *no* por AuthController).
11. Spring Security intercepta esta petición a través del filtro de autenticación. Utiliza `CustomUserDetailsService` para cargar el usuario con el identificador proporcionado. Nuestro servicio buscará primero por username, luego por email. Si no encuentra coincidencias, Spring Security rechaza la autenticación y recarga la página de login con un mensaje de error (“Bad credentials” o equivalente, que la vista puede mostrar).
12. Si encuentra el usuario, Spring Security entonces compara la contraseña ingresada con la almacenada. Gracias a que configuramos BCrypt, el framework aplicará el hash BCrypt al password proporcionado y lo comparará con el hash guardado en la BD. Si coinciden, la autenticación es **exitosa**. En ese momento, Spring Security inicia una sesión HTTP para el usuario y almacena en ella los detalles de autenticación (Security Context).
13. Tras un login exitoso, la configuración (`SecurityConfig`) indica que se debe redirigir al usuario a la página principal "/" (parámetro `.defaultSuccessUrl("/", true)`). De este modo, el usuario autenticado vuelve a la Home pero ahora con acceso autorizado a las áreas protegidas. La home podría reflejar el estado logueado, por ejemplo mostrando su nombre y nuevas opciones de menú (subir archivo, ver historial, cerrar sesión).

14. *Resultado:* El usuario está autenticado en la aplicación. Puede navegar por las secciones públicas y también acceder a las funciones privadas sin necesidad de volver a loguearse mientras dure la sesión.

15. **Acceso a Funcionalidad Protegida – Subida de Archivo:** Autenticado ya, el usuario procede a utilizar la funcionalidad principal. Va a la página de **subida de archivos** (por ejemplo, mediante un botón "Sellar Archivo" visible solo tras login, que apunta a `/upload`).

16. Al acceder a `/upload`, el **FileUploadController** verifica (vía Spring Security) que el usuario esté autenticado. Dado que lo está, permite el acceso y ejecuta el método `mostrarFormulario`. Este método añade al modelo un objeto `FileUploadRequest` vacío y retorna la vista `upload.html`.
17. La página `upload.html` muestra un formulario con un campo para seleccionar un archivo del sistema local (input de tipo file). El usuario escoge el archivo que desea sellar (puede ser cualquier tipo de fichero: texto, imagen, ZIP, código, etc., no hay restricción de tipo definida en la aplicación por ahora).
18. El usuario envía el formulario (botón "Subir/Sellar"). El navegador realiza una petición POST a `/upload` con el fichero adjuntado (multipart request). Spring MVC vincula automáticamente el archivo subido al objeto `FileUploadRequest` que el controlador espera, pasando además por la validación (@Valid).
19. El **FileUploadController** procesa la solicitud en el método `procesarArchivo`. Hace primero las validaciones: si el objeto tiene errores de validación (por ejemplo, si `req.getFile()` estaba vacío porque no se seleccionó nada, o si se puso alguna anotación adicional de validación de tamaño) o si `req.getFile().isEmpty()` retorna true (archivo sin contenido o no seleccionado), entonces añade un mensaje de error usando `RedirectAttributes` – por ejemplo, "Debes seleccionar un archivo" – y redirige de vuelta a `"/upload"` (Post/Redirect/Get en caso de error). Esto hará que la página de subida se recargue y muestre el mensaje flash de error, manteniendo al usuario en la misma página para que intente nuevamente.
20. Si el archivo fue proporcionado correctamente, el controlador procede a obtener el usuario autenticado actual. Para ello, accede al **SecurityContext** y llama `auth.getName()` – esto nos devuelve el username (o email) del principal. Luego, vía `UserService.getByUsername(username)`, recupera el objeto AppUser completo de la base de datos. (En una mejora se podría guardar el ID de usuario en sesión para evitar esta consulta, pero dado que la BD es pequeña, esta búsqueda no impacta significativamente).
21. Con el usuario `owner` en mano y el archivo en el DTO, el controlador invoca a `fileHashService.saveForUser(owner, file)`. Aquí se delega la lógica de *sellado* al servicio:
    - Dentro de `FileHashService.saveForUser(AppUser, MultipartFile)`, se lee el InputStream del archivo y se calcula su **hash SHA-256** mediante `DigestUtils.sha256Hex` (de Apache Commons Codec). Esto genera una cadena hexadecimal de 64 caracteres representando la “huella digital” única del fichero.
    - El servicio luego crea un nuevo objeto FileHash, asigna el usuario como owner, asigna el hash calculado y el nombre original del fichero (`file.getOriginalFilename()`), y guarda esta entidad con `fileHashRepository.save()`. Este método ejecuta un `INSERT` en la tabla `file_hashes`, persistiendo el registro. Flyway ya se aseguró de que la tabla exista con las columnas adecuadas; la columna `user_id` almacenará la referencia al id del usuario propietario.
    - Si por alguna razón el proceso de guardar falla (por ejemplo, una falla de conexión a BD, o cualquier excepción inesperada), el servicio lanza una excepción. El controlador tiene envuelto el call en un bloque try-catch, de modo que si ocurre una excepción, captura esa situación, agrega un mensaje flash de error ("ERROR al procesar el archivo: ...") con la causa, y seguiría adelante. En condiciones normales, no se esperan excepciones aquí salvo errores de E/S leyendo el archivo o problemas de BD, pero está contemplado.
22. De vuelta en el controlador, si todo fue bien, tras el `fileHashService.saveForUser` el registro ya está almacenado. El controlador añade un mensaje de éxito usando `redirect.addFlashAttribute("success", "Archivo sellado correctamente")` para notificar al usuario en la siguiente pantalla. Luego decide qué hacer a continuación. En lugar de mostrar inmediatamente una vista, aplica Post/Redirect/Get para evitar envíos duplicados: hace

`return "redirect:/history"`. Esto redirige al navegador a la URL `/history` mediante un GET.

23. *Resultado:* El archivo del usuario ha sido procesado: su hash SHA-256 está calculado y almacenado en la base de datos junto con la referencia al usuario. El usuario es redirigido a la página de historial para ver el resultado.

24. **Consulta de Historial:** Tras subir un archivo (o cualquier momento que el usuario lo desee), la página de **historial** muestra todos los archivos que ese usuario ha sellado en el pasado. La redirección anterior activó una petición GET a `/history`, que maneja el mismo `FileUploadController` (método `verHistorial`).

25. Este método de controlador de historial, al igual que antes, obtiene el nombre del usuario actualmente autenticado del contexto de seguridad, busca el objeto AppUser en BD (`userService.getByUsername`) y luego invoca `fileHashService.findDtosByUser(owner)`.
26. El `FileHashService.findDtosByUser` a su vez consulta el repositorio: `fileHashRepository.findByOwnerOrderByCreatedAtDesc(owner)`. Esto retorna una lista de FileHash (entidades) correspondientes al usuario, ordenadas de la más reciente a la más antigua. Esa lista se convierte a una lista de `FileHashDto` mediante el mapper para desprender cualquier información innecesaria (por ejemplo, evita exponer internamente el objeto AppUser anidado, etc., y convierte la fecha/hora a LocalDateTime que Thymeleaf puede formatear fácilmente).
27. El controlador recibe la lista de DTOs y la añade al modelo como atributo `"hashes"`. Luego retorna la vista `history.html`.
28. La vista Thymeleaf iterará sobre `"hashes"` y generará probablemente una tabla o tarjetas donde cada entrada muestra: nombre del archivo original, el hash SHA-256 calculado y la fecha/hora de creación. Es común que el hash, al ser tan largo, se muestre truncado con posibilidad de copiarlo completo al portapapeles si el usuario quiere usarlo externamente. También podría haber en el historial alguna indicación del número total de archivos sellados.
29. Adicionalmente, como esta petición vino tras un redirect con flash attributes, Thymeleaf puede mostrar el mensaje de éxito ("Archivo sellado correctamente") en la parte superior de la página para informar al usuario que la última operación fue exitosa. Ese mensaje desaparecerá al refrescar (por ser flash).

30. Desde esta página de historial, el usuario puede optar por sellar otro archivo (habrá un enlace o botón de "Subir otro archivo" que lo lleva de nuevo a `/upload`) o simplemente navegar (quizá un menú donde también está la opción de Home o Logout).

31. **Cierre de Sesión (Logout):** En cualquier momento, el usuario autenticado puede cerrar su sesión (por ejemplo, mediante un botón "Cerrar Sesión" en la barra de navegación). Al hacer click, típicamente se envía una petición POST a `/logout` (Spring Security recomienda que el logout sea POST por seguridad, pero a nivel de UX puede ser un formulario oculto que se envía en segundo plano). Spring Security intercepta `/logout` y realiza:

32. Invalidación de la sesión HTTP del usuario (removiendo su SecurityContext).
33. Opcionalmente, eliminación de la cookie JSESSIONID en el cliente (según configuración por defecto).
34. Redirección a una URL pública. Según nuestra configuración en `SecurityConfig`, tras logout se redirige al usuario a `/home`. De esta forma, el usuario ve la página de inicio como si fuera un

visitante no autenticado. También podríamos mostrar un mensaje "Has cerrado sesión" mediante query param si se deseara.

35. Ahora, si el usuario intenta acceder a rutas protegidas (/upload, /history) de nuevo, Spring lo redirigirá al login, ya que su sesión ya no es válida.

36. **Manejo de Errores en el Flujo:** En cada paso, la aplicación maneja adecuadamente errores típicos:

37. Si el usuario intenta registrarse con un email o nombre ya existente, se le notifica para que elija otro.
38. Si ingresa credenciales incorrectas en login, la misma página de login mostrará un mensaje de error (gestionado por Spring Security automáticamente).
39. Si intenta acceder a una URL protegida sin autenticarse (por ejemplo, `/upload` sin login), Spring Security intercepta y lo envía al login automáticamente (y tras autenticarse, puede enviarlo a la página que quiso acceder originalmente).
40. Si durante la subida de archivo no selecciona ninguno, se le indica que debe elegir un archivo antes de enviar.
41. Errores inesperados (excepciones) durante el procesamiento de archivo generan un mensaje flash de error genérico. En un entorno profesional, estos incidentes también quedarían registrados en logs del servidor para diagnosticar.

En conjunto, el flujo de uso de BitSealer es sencillo e intuitivo: **Registro -> Login -> Subir archivo(s) -> Ver historial**. Cada funcionalidad está integrada de manera consistente con la experiencia de usuario esperada en aplicaciones web (uso de flashes para mensajes, redirección tras operaciones para evitar duplicación, validaciones de formulario, etc.). Esto facilita la adopción interna, ya que los desarrolladores de la empresa encontrarán familiar la interacción (similar a muchas aplicaciones web con registro/login). Además, desde la perspectiva de mantenimiento, cada paso del flujo corresponde a un controlador y vista específicos, lo que permite modificar o extender flujos (por ejemplo, agregar confirmación por correo en el registro, paginación en el historial, etc.) de forma localizada sin repercutir negativamente en otros componentes.

7. Esquema de Base de Datos

El esquema de base de datos de BitSealer es **relativamente simple** y consta de dos tablas principales: una para usuarios del sistema y otra para los registros de archivos sellados. A continuación se describen estas tablas, sus columnas y relaciones:

- **Tabla `users`:** Almacena los datos de cada usuario registrado.
- **id** – BIGINT, Primary Key, auto-generada (IDENTITY). Identificador único de cada usuario.
- **username** – VARCHAR(255), Único, No Nulo. Nombre de usuario elegido, usado para autenticación (también se permite usar el email como alternativa). Un índice único garantiza que no haya dos usuarios con el mismo username.
- **email** – VARCHAR(255), Único, No Nulo. Dirección de correo del usuario. También se usa para login alternativo. Índice único para evitar duplicados (el sistema fuerza emails únicos para propósitos de contacto único y como identificación adicional).
- **password** – VARCHAR(255), No Nulo. Contraseña cifrada del usuario. Aquí se almacena el hash BCrypt de la contraseña original (normalmente un string de 60 caracteres, aunque el campo permite hasta 255 para acomodar posibles algoritmos más largos). **Importante:** nunca se guarda la contraseña en claro; siempre en forma codificada.

- **role** – VARCHAR(255), Puede ser Nulo (según entidad es not null con default). Rol de seguridad del usuario (por ejemplo "ROLE_USER" por defecto, u "ROLE_ADMIN" si hubiera administradores). En la migración se ve que este campo no se marcó unique ni tiene check, simplemente es texto. La aplicación asigna "ROLE_USER" a todos los nuevos registros, y ese prefijo `ROLE_` es requerido por Spring Security para las autoridades. Aunque actualmente todos los usuarios comparten el mismo rol, este campo permitiría en un futuro distinguir diferentes tipos de usuarios o permisos.

Esta tabla es creada por Flyway en la migración inicial. Tiene además las siguientes **restricciones** definidas: - Clave primaria en `id` (implícita por ser identity). - Un índice de unicidad combinada en `username` y en `email` para reforzar la restricción de unicidad (según el script, se especificó `NOT NULL UNIQUE` en cada columna). - Las columnas `username` y `email` se permiten nulas en la definición de entidad, pero en la migración SQL están como NOT NULL (debe haber consistencia: seguramente deberían ser NOT NULL a nivel entidad también). Asumiremos que ambas son NOT NULL en la base de datos, lo cual es lo deseable.

- **Tabla** `file_hashes`**:** Registra cada archivo subido (sellado) por los usuarios, junto con su hash.
- **id** – BIGINT, Primary Key, auto-generada. Identificador único de cada registro de archivo.
- **file_name** – VARCHAR(255). Nombre original del archivo tal como lo subió el usuario (incluye extensión). Esto es útil para referencia humana, ya que un hash por sí solo no indica qué archivo era. No se impone unicidad ni not null estricto (podría ser null si no quisiéramos guardarlo, pero en la práctica la aplicación siempre establece un nombre). Se podría en teoría tener dos entradas con el mismo nombre si dos archivos distintos se llamaban igual.
- **sha256** – VARCHAR(255). Cadena hexadecimal representando el hash SHA-256 del archivo. En teoría son 64 caracteres hexadecimales; se usa 255 por convención/seguridad. Guarda la huella digital que sirve como "sello". No se definió como UNIQUE, dado que archivos distintos (de distintos usuarios o incluso de un mismo usuario en subidas distintas) podrían coincidentemente tener el mismo hash si el contenido es igual. Sin embargo, este campo en combinación con user_id podría haber sido unique si se quisiera evitar duplicados por usuario. Actualmente no lo es, permitiendo registro histórico incluso de subidas repetidas.
- **created_at** – TIMESTAMP(6). Fecha y hora en que se creó el registro (precisión microsegundos). La aplicación asigna este valor en la entidad al momento de instanciarla (LocalDateTime.now()). En la BD podría quedar null si no se envía (no se especificó NOT NULL en el script, pero la aplicación siempre lo rellena). Sirve para ordenar y mostrar al usuario cuándo fue sellado cada archivo.
- **user_id** – BIGINT, Foreign Key, No Nulo. Referencia al usuario dueño del archivo. Este campo realiza la relación muchos-a-uno hacia `users.id`. Tiene un índice implícito por ser clave foránea, útil para consultar todos los hashes de un usuario.

La **relación** entre `file_hashes` y `users` está expresada mediante la clave foránea `fk_filehash_user`: `FOREIGN KEY (user_id) REFERENCES public.users(id)`. Esto garantiza integridad referencial: no puede existir un registro de hash asociado a un usuario inexistente. Si un usuario se elimina (no hay funcionalidad de borrado de usuarios expuesta actualmente, pero a nivel BD), los registros asociados deberían eliminarse o quedar huérfanos según la política (no definida explícitamente, asumiríamos restrict por defecto, impidiendo borrar un usuario si tiene historiales). En este diseño, lo esperado es no borrar usuarios activos, sino quizá marcar inactivos si fuera necesario.

Además de estas tablas principales, **Flyway** crea una tabla auxiliar llamada `flyway_schema_history` (en el esquema `public` por defecto) para llevar el control de las migraciones aplicadas. En ella se registra cada script ejecutado, con su versión (V1, V2, ...), descripción, fecha de aplicación, éxito/fallo, etc. [5] . Esto permite auditoría de cambios en la base de datos y evita re-aplicar la misma migración

dos veces. Para BitSealer, tras la primera ejecución, `flyway_schema_history` contendrá un registro de la versión `1` (script `V1__init.sql` aplicado). Si en el futuro se añade, por ejemplo, `V2__add_audit_table.sql`, Flyway verá que hay una versión nueva y la aplicará, dejando otro registro.

Consideraciones de Diseño de Datos: - La combinación de `users` y `file_hashes` implementa una relación uno-a-muchos: un usuario puede tener muchos hashes (archivos) asociados. Desde la perspectiva de JPA, en la entidad `FileHash` esto está representado con `@ManyToOne(fetch = LAZY) private AppUser owner`. No se mapeó el `@OneToMany` del lado de `AppUser` posiblemente para simplificar consultas (así se evita cargar accidentalmente todos los hashes cuando se carga un usuario, dado que `open-in-view` está false). - El tamaño de los campos: 255 caracteres para username/email podría ser más de lo necesario, pero sigue un estándar común. Los emails suelen limitarse a 254 por RFC, así que 255 está bien. Para el hash 64 hex, 255 es sobredimensionar, pero no da problema y ofrece margen si se quisiera usar otro algoritmo más largo (por ejemplo SHA-512) en el futuro. - No se almacenan los archivos en sí ni su tamaño/tipo MIME. Si esto se necesitara, podría ampliarse la tabla `file_hashes` con columnas como `file_size`, `content_type` o incluso una columna BYTEA para el contenido (no recomendado para archivos grandes). Dado el enfoque de BitSealer, se decidió no guardar los binarios para mantener la BD liviana. - La tabla de usuarios no incluye campos como nombre real, apellidos, etc., ya que no eran necesarios para la funcionalidad. Solo los esenciales para autenticación y contacto. - **Índices:** Además de los índices únicos en email/username, conviene notar que la clave primaria de cada tabla tiene su índice cluster (id). La foreign key `user_id` en `file_hashes` normalmente también crea un índice no único para eficientar las búsquedas por usuario (depende de la configuración de Postgres, pero es habitual). Esto ayuda justamente a la consulta `findByOwnerOrderByCreatedAtDesc`, pues la BD puede rápidamente filtrar por user_id. En la migración dada, no se ve explícito `CREATE INDEX` para user_id, pero PostgreSQL automáticamente indexa las claves primarias y foráneas a menudo (FK no siempre auto-indexa en Postgres, así que en optimización futura podríamos crear un índice en file_hashes(user_id) manualmente si se notara degradación con muchos registros). - **Migraciones:** La migración V1 proporcionada parece provenir de un `pg_dump` inicial, con comentarios y comandos de configuración (por ejemplo, configuración de `search_path`, etc.). Tras crear las tablas, hace un comentario de que se retiraron posibles inserts iniciales. No se insertan datos por defecto (no hay usuarios pre-cargados ni hashes de ejemplo). Por tanto, tras despliegue, la BD arranca vacía en cuanto a datos de negocio. El primer usuario se creará vía la aplicación (registro) y así sucesivamente.

En resumen, el esquema de base de datos es **pequeño pero sólido**: tiene las restricciones necesarias para mantener consistencia (integridad referencial y unicidad donde corresponde) y es fácilmente extensible. Por ejemplo, si se quisiera llevar un log de auditoría de accesos o un registro de cambios, se pueden agregar tablas adicionales en nuevas migraciones. La simplicidad del modelo facilita también la comprensión por parte de nuevos desarrolladores y la colaboración con herramientas de IA, ya que las relaciones son directas y el dominio (usuarios y archivos con hash) es claro.

8. Seguridad

La seguridad de BitSealer se apoya en Spring Security para cubrir autenticación, autorización y protección básica contra accesos no deseados. A continuación se detallan los aspectos más relevantes de la implementación de seguridad: cómo se autentican los usuarios, cómo se gestionan las sesiones, cómo se almacenan las contraseñas y qué políticas de acceso se aplican.

- **Autenticación de Usuarios:** La aplicación utiliza un esquema de autenticación **Username/Password** tradicional mediante formulario de login. Cuando un usuario intenta iniciar sesión,

Spring Security delega en nuestra clase `CustomUserDetailsService` la tarea de cargar los detalles del usuario dado un identificador. Como vimos, este servicio permite ingresar tanto el **nombre de usuario** como el **email** para autenticarse. Internamente:

- Se busca un registro en la tabla `users` cuyo `username` coincida con lo ingresado; si no se halla, se intenta con `email`. Esto es útil en entornos corporativos donde a veces el email es más fácil de recordar.
- Si se encuentra el usuario, se construye un objeto de seguridad con su nombre, contraseña cifrada y rol (BitSealer maneja por defecto el rol "ROLE_USER" para todos). Este objeto es de tipo UserDetails e indica a Spring Security cómo validar la contraseña y qué autorizaciones tiene el usuario.
- Spring Security entonces compara la contraseña proporcionada con la almacenada usando el `PasswordEncoder` (BCrypt). Gracias a la configuración en `EncoderConfig`, la plataforma sabe que la contraseña en BD está cifrada con BCrypt, y procede a **hacer el hash de la contraseña ingresada y verificar coincidencia**. BCrypt incorpora su propio salt, por lo que cada comparación implica recalcular el hash con el salt guardado y comparar hashes resultantes [2].
- Si la comparación es exitosa, el usuario es autenticado con éxito; de lo contrario, la autenticación falla. Los mensajes de error de login son manejados automáticamente por Spring Security en la página de login (por ejemplo, setting `BadCredentialsException` produce un mensaje de "Invalid username or password", que podemos internacionalizar si quisiéramos).

- No existe por el momento un mecanismo de bloqueo de cuenta tras múltiples intentos fallidos ni segundo factor de autenticación. Dado que es una herramienta interna con número limitado de usuarios, se consideró suficiente el esquema básico. Sin embargo, la infraestructura con Spring Security permitiría agregar tales características si fueran requeridas (por ejemplo, usando *UserDetails* campos como accountNonLocked, etc., o integrando con un proveedor SSO corporativo).

- **Gestión de Sesiones:** Una vez autenticado, Spring Security crea una sesión HTTP estándar para el usuario. Los detalles:

- La aplicación, al no ser stateless, usa **cookies de sesión (JSESSIONID)**. Esto significa que el servidor mantiene en memoria (o en un store distribuido, si se configurara) una sesión por cada usuario logueado, identificada por una cookie en el navegador. Cada petición subsecuente del mismo navegador incluye esa cookie, lo que permite a Spring Security saber quién es el usuario (vía *Session ID* -> contexto de seguridad asociado).
- El `SecurityConfig` no define explícitamente configuración de sesiones, por lo que se aplica la política por defecto de Spring Session: una sesión concurrente por usuario (no se prohibe iniciar sesión desde dos lugares diferentes con las mismas credenciales, pero cada una tendría su sesión).
- No se establece un timeout personalizado en `application.properties`; se heredaría el default del contenedor (que suele ser 30 minutos). En un entorno productivo, se podría ajustar `server.servlet.session.timeout` si se desease un tiempo de expiración específico.

- Spring Security se encarga también de asociar la sesión a un *SecurityContext* que contiene el objeto UserDetails del usuario y sus roles. Este contexto es accesible en cualquier parte de la aplicación mediante utilidades estáticas (SecurityContextHolder) o inyección de Authentication en métodos de controlador, etc. En BitSealer, se usa `SecurityContextHolder.getContext().getAuthentication()` para obtener el nombre del usuario actual cuando se necesita en controladores, lo cual es posible gracias a esta gestión de sesión.

- **Autorización y Control de Acceso:** La configuración en `SecurityConfig` establece las reglas de qué recursos son públicos y cuáles requieren autenticación:

- Se **deshabilitó CSRF** en la configuración para simplificar peticiones POST desde formularios (especialmente porque en entorno dev/herramienta interna puede no considerarse crítico, aunque es un aspecto a revisar para producción). Al desactivar CSRF, no se exige el token CSRF en formularios POST, lo que facilita pruebas con Postman o similares, pero hay que tener cuidado pues expone teóricamente a ataques CSRF si la app estuviera expuesta públicamente. En un entorno corporativo cerrado, el riesgo es menor, pero la recomendación oficial es habilitar CSRF para formularios web. De hecho, un punto de mejora será habilitarlo y agregar los tokens en los forms Thymeleaf con `${_csrf.token}` y `${_csrf.parameterName}`.
- **URLs públicas permitidas:** usando `.authorizeHttpRequests` se configuran rutas específicas como accesibles sin autenticación:
    - `/` y `/home` – la página de inicio (portada) es pública.
    - `/login` y `/register` – las páginas de autenticación y registro deben ser públicas, de lo contrario nadie podría registrarse o loguearse.
    - Recursos estáticos: `/css/**`, `/js/**`, `/img/**`, `/vendor/**`, e incluso `/webjars/**` (este último es común cuando se usan librerías vía webjars, en nuestro caso FontAwesome se incluyó manualmente en vendor). Esto garantiza que archivos estáticos (estilos, scripts, imágenes, bibliotecas) se carguen aunque el usuario no esté logueado, evitando que la pantalla de login aparezca sin estilos por bloqueos de seguridad.
    - Estas rutas se configuran con `.permitAll()`, otorgando acceso irrestricto.
- **Protección de URLs internas:** se establece `.anyRequest().authenticated()`, lo que significa que cualquier petición que no haya coincidido con las anteriores debe requerir autenticación. En otras palabras, rutas como `/upload`, `/history`, cualquier otra no listada públicamente, sólo estarán accesibles si el usuario ha hecho login.
- Actualmente, **no se diferencian roles** en la autorización. Todos los usuarios registrados comparten rol "USER" y la regla es simplemente estar autenticado. En el futuro, si se introdujeran roles adicionales (ej. ADMIN), se podrían agregar reglas como `.requestMatchers("/admin/**").hasRole("ADMIN")` para restringir ciertos endpoints. Spring Security facilita esto de forma declarativa.

- **Logout:** Está configurado un logout exitoso que redirige a `/home`. El logout se activa al hacer POST a `/logout` (como se definió con `.logoutUrl("/logout")`). Spring Security se encarga de invalidar la sesión. Además, `.permitAll()` se aplica por defecto a `/logout` (aunque en la config actual quizás faltó `.permitAll()` en la sección de logout, Spring Security suele permitirlo para no bloquear la acción de logout mismo).

- **Almacenamiento de Contraseñas (Hashing):** Como se explicó, las contraseñas de usuarios se almacenan usando el algoritmo **BCrypt**. Detalles técnicos:

- Al registrar un usuario, `UserService.registerUser` toma la contraseña en texto plano proporcionada y llama `passwordEncoder.encode(rawPassword)`. Dado que en `EncoderConfig` definimos `new BCryptPasswordEncoder()`, este método genera un **hash BCrypt** que incluye sal aleatoria incorporada. El resultado es algo como `$2a$10$<22-caracteres-salt><31-caracteres-hash>` (60 caracteres aprox, comenzando con `$2a$10$` donde 10 es el *cost factor* por defecto) [6] [2].
- Este hash se guarda en la columna `password` de la tabla `users`. Ni la aplicación ni los administradores humanos pueden ver la contraseña original a partir de ese hash (no es reversible).

- En autenticación, Spring Security utiliza la misma clase BCryptPasswordEncoder para comprobar las credenciales: internamente sabe que un hash que comienza con `$2a$` es BCrypt [7], así que aplica el algoritmo correspondiente. Si el desarrollador hubiera querido soportar múltiples esquemas, Spring Security tiene `DelegatingPasswordEncoder` que permite almacenar el hash con un identificador `{bcrypt}` o `{noop}` etc. En nuestro caso, siempre usamos bcrypt, así que no es necesario prefijo, pero Spring Boot 3+ suele almacenar con prefijo `{bcrypt}` por seguridad. (Se podría configurar para que encode inserte el identificador).

- **¿Por qué BCrypt?** BCrypt es recomendado porque es **adaptativo**: el factor de costo se puede subir si las máquinas se vuelven más potentes, haciendo el hash más lento y dificultando ataques de fuerza bruta [8]. Además, genera un salt único por contraseña, protegiendo contra ataques con tablas de hashes precomputadas (rainbow tables). En contraste, algoritmos rápidos como SHA-256 (sin salt) no son apropiados para contraseñas, por eso no reutilizamos el mismo SHA-256 de archivos para las passwords. Esta diferenciación está implementada correctamente en BitSealer (BCrypt para contraseñas, SHA-256 para huella de archivos, cada uno en su contexto adecuado).

- **Protección contra amenazas comunes:**

- *SQL Injection:* Al usar JPA y consultas parametrizadas derivadas de métodos, la aplicación está en gran medida protegida frente a inyecciones SQL. No se arman strings SQL manualmente con entrada del usuario en nuestro código, todo pasa por el framework que utiliza prepared statements. Las únicas entradas del usuario que llegan a la BD son el username/email en login (usado en repos findByUsername/Email de forma segura) y los datos de registro (save de AppUser con valores seteados, manejado internamente por Hibernate).
- *XSS (Cross-Site Scripting):* Thymeleaf escapa automáticamente las expresiones de variables en las vistas, previniendo inyección de HTML/JS en caso de mostrar datos que provienen del usuario. Ej: si mostráramos el username en la UI, Thymeleaf lo escaparía por defecto. Además, no estamos mostrando nunca el hash de forma procesable en un script, solo como texto, así que es seguro. Aún así, se revisaría no imprimir directamente contenidos no confiables sin escape.
- *CSRF:* Como se mencionó, actualmente está **deshabilitado** (`http.csrf().disable()` en SecurityConfig). Esto significa que Spring Security no exige el token CSRF en formularios POST, lo que simplifica las cosas pero abre la puerta a que, si un atacante conociera la sesión del usuario y le engañara para hacer submit a /upload por ejemplo, podría ocurrir (dado el ámbito interno es improbable, pero existe la posibilidad). Para un producto en producción orientado a web, **deberíamos habilitar CSRF** y añadir en los formularios Thymeleaf el campo oculto `${_csrf.token}`. Esto impediría peticiones forjadas desde otros orígenes. Dado que es un detalle importante, en entornos reales se corregiría; para nuestra docencia/herramienta interna, se dejó así para evitar confusiones con tokens durante pruebas manuales.
- *Autorización horizontal/vertical:* Actualmente no hay distinción de roles más allá de usuario logueado/no logueado, por lo que no hay riesgo de escalación de privilegios: todos los logueados tienen las mismas capacidades. Si en un futuro hubiera usuarios con mayor privilegio (ADMIN), habría que asegurarse de proteger rutas administrativas con `.hasRole("ADMIN")` y de no filtrar datos indebidamente (por ejemplo, un usuario no admin no debería ver archivos de otro usuario). Ahora mismo, la lógica de negocio siempre filtra por usuario actual (por ejemplo, al listar historiales, se usa el usuario obtenido del contexto de seguridad, así que un usuario A no puede pedir el historial de B porque el código no expone tal funcionalidad).

- *Protección de contraseñas en tránsito:* Se asume que la aplicación estará desplegada en entornos seguros (intranet) o detrás de HTTPS. En las configuraciones actuales no se fuerza HTTPS (eso

suele manejarse en el servidor frontal o config adicional). Para un despliegue internet, se debe servir siempre sobre TLS para que las credenciales no viajen en claro.

- **Implementación de Logout:** Como comentado, la aplicación define `.logoutUrl("/logout")` y `.logoutSuccessUrl("/home")`. No se especificó `.invalidateHttpSession(true)` o `.deleteCookies("JSESSIONID")` pero por defecto Spring invalidará la sesión. Tras logout, el usuario es redirigido a Home donde verá la versión pública. El logout se activa típicamente mediante un pequeño formulario o enlace con JavaScript que haga POST (ya que los métodos de logout con GET están desaprobados, aunque podrían configurarse con .logoutRequestMatcher). Dado que en SB Admin 2 hay ejemplos de menú con logout, es probable que se haya incluido un formulario de logout en alguna parte del template principal.

- **Auditoría de Seguridad:** Actualmente no se implementó, pero al usar Spring Security uno puede aprovechar las clases de auditoría de Spring Data (por ejemplo `@CreatedBy`, `@LastModifiedBy`) para marcar quién creó ciertos registros. En este proyecto, podría ser útil a futuro añadir campos en FileHash que registren el user que lo creó (aunque ya existe la relación owner) o logs de acceso. Por ahora, se confía en que la base de datos misma y el historial cumplen esa función (pues cada FileHash ya está ligado a quién lo creó).

En conclusión, BitSealer adopta un enfoque de seguridad **estándar pero efectivo** para una aplicación web interna: - Autenticación mediante formulario con credenciales, apoyada por un esquema robusto de almacenamiento de contraseñas (BCrypt). - Autorización simple basada en sesión iniciada, protegiendo todas las operaciones sensibles. - Integridad de datos mantenida por validaciones en registro y restricción de acciones (no se puede subir/ver si no está logueado). - Áreas de mejora identificadas, como reactivar CSRF y considerar mecanismos adicionales según necesidades (p.ej., límite de sesiones o MFA, si el contexto lo demandara).

Al ser un sistema interno, estas medidas brindan un **buen nivel de seguridad** sin añadir complejidad innecesaria. Los desarrolladores nuevos en el proyecto encontrarán familiar la configuración de Spring Security y podrán ajustarla o ampliarla (por ejemplo, agregando OAuth2 para SSO corporativo, o implementando roles admin) de forma relativamente sencilla.

9. Configuración y Perfiles de Entorno

BitSealer utiliza el mecanismo de **perfiles de Spring Boot** para manejar diferentes configuraciones según el entorno de despliegue. En particular, se define un perfil personalizado llamado **"docker"** para cuando la aplicación corre dentro de un contenedor Docker, mientras que la configuración por defecto (perfil *default* o *dev*) se usa durante desarrollo local. A continuación se detallan los archivos de configuración, las propiedades más importantes y cómo se gestionan las variables de entorno:

- `application.properties` **(perfil por defecto):** Este archivo contiene la configuración que se aplica cuando no se activa un perfil específico, es decir, típicamente en desarrollo local o ejecución de pruebas manuales. Algunas entradas clave:
- `spring.application.name=BitSealer` – Nombre de la aplicación (puede aparecer en logs, etc., o ser usado para registro en Service Discovery si aplicara).
- **Configuración de DataSource (BD local):**

```
spring.datasource.url=jdbc:postgresql://localhost:5432/filehashdb
spring.datasource.username=filehashuser
spring.datasource.password=1234
```

Esto indica que en entorno de desarrollo la app intentará conectarse a una instancia local de PostgreSQL en el puerto 5432, base de datos `filehashdb`, con credenciales `filehashuser/1234`. Es responsabilidad del desarrollador tener esta BD en marcha (por ejemplo, mediante Docker local o una instalación). Alternativamente, el developer puede también ejecutar `docker-compose` para levantar la BD y la app incluso en dev, pero entonces usaría perfil docker.
Tener estas credenciales en *plain text* en application.properties está bien para dev, pero en producción se evitan. Aquí se usa porque en la máquina local normalmente no es crítico, y se facilita la ejecución out-of-the-box.

- **Configuración JPA/Hibernate:**

```
spring.jpa.hibernate.ddl-auto=none
spring.jpa.open-in-view=false
spring.jpa.show-sql=true
```

La propiedad `ddl-auto=none` deshabilita la auto-creación o actualización de esquemas por parte de Hibernate. Esto es deliberado ya que Flyway se encarga de la migración, evitando así potenciales conflictos (por ejemplo, que Hibernate intente crear tablas ya creadas por Flyway, o borrar algo).
`open-in-view=false` es una recomendación de desempeño y claridad: esta opción, cuando está en false, cierra la sesión de Hibernate al finalizar la transacción del servicio, antes de generar la vista. Así, si en la vista intentáramos acceder a una colección lazy no inicializada, daría excepción en lugar de silenciosamente cargar tardíamente. Esto fuerza a cargar todo lo necesario en la capa de servicio (lo cual se hace, al convertir a DTO y salir del contexto). En aplicaciones web modernas suele ser buena práctica desactivarlo para prevenir problemas de N+1 queries en la vista [9].
`show-sql=true` hace que Hibernate loguee en la consola cada sentencia SQL que ejecuta (junto con sus parámetros). En dev es muy útil para depurar queries y verificar que consultas derivadas funcionan como se espera. En producción se suele apagar (por ruido y pequeñas implicaciones de rendimiento).

- **Otros:** Podría haber más propiedades no mostradas en snippet, por ejemplo:
  - `spring.flyway.enabled=true` – para asegurarse de que Flyway migre al inicio. (Spring Boot por convención activa Flyway si encuentra la dependencia y una BD configurada).
  - `spring.flyway.locations=classpath:db/migration` – normalmente por defecto ya es esa ruta, no haría falta especificar.
  - Configuraciones de logging (no se ven, probablemente no se cambió el nivel por defecto INFO).
  - `server.port` – no está en el archivo, lo cual significa que por defecto la app usará el puerto 8080. En dev se podría cambiar si 8080 estuviera ocupado, pero en general 8080 es estándar.

- `application-docker.yml` **(perfil "docker"):** Este archivo YAML contiene las configuraciones que se aplican cuando la aplicación se ejecuta con el perfil activo "docker". De hecho, el

Dockerfile añade `SPRING_PROFILES_ACTIVE=docker`, activando todo lo aquí definido y sobreescribiendo lo de application.properties donde corresponda. Puntos clave en este perfil:

- **Parámetros de conexión a BD mediante variables de entorno:**

```
spring:
  datasource:
    url: ${SPRING_DATASOURCE_URL}
    username: ${SPRING_DATASOURCE_USERNAME}
    password: ${SPRING_DATASOURCE_PASSWORD}
    hikari.maximum-pool-size: 5
  jpa:
    hibernate.ddl-auto: none
    open-in-view: false
    show-sql: false
  flyway:
    enabled: true
```

  Aquí, en lugar de codificar la URL y credenciales, se referencia a variables de entorno del sistema operativo (las sintaxis `${VAR}` se resolverán en runtime). Docker Compose proporciona estas variables al contenedor:
  - `SPRING_DATASOURCE_URL=jdbc:postgresql://db:5432/filehashdb`
  - `SPRING_DATASOURCE_USERNAME=filehashuser`
  - `SPRING_DATASOURCE_PASSWORD=1234`
    Como se ve, dentro del contenedor la BD no es "localhost" sino `db` (que es el hostname del servicio de BD en la red Docker), para apuntar correctamente. Esta parametrización permite cambiar credenciales o host sin modificar el código ni el YAML – solo cambiando variables. En un entorno productivo, en lugar de 1234 se pondría una contraseña segura inyectada desde un gestor de secretos o compose override.
    El tamaño máximo de pool de conexiones Hikari se establece en 5 para Docker, asumiendo carga baja/media (por ser una herramienta interna, 5 concurrentes está bien; en dev no se especificó, tomando default 10).
- **JPA/Hibernate/Flyway:** Similar a dev: se asegura ddl-auto none, open-in-view false. `show-sql` se pone **false** en Docker para no llenar los logs de SQL – en producción suele mantenerse en false por limpieza. Flyway se habilita explícitamente (aunque con la dependencia, Boot suele ejecutarlo de todas formas, es redundante indicar enabled:true si la dependencia está presente).
- **Perfil docker vs default:** Notar que el perfil docker *hereda* también lo que no esté sobreescrito. Por ejemplo, `spring.application.name` no se redefinió en docker.yml, así que seguirá siendo "BitSealer". Lo mismo con cualquier propiedad no listada: el perfil activo complementa al default. En Spring Boot, `application-docker.yml` solo contiene diferencias. Al arrancar con profile docker, se cargan ambos y las del perfil tienen precedencia en caso de conflicto.

- En este YAML no se ven configuraciones de servidor (como `server.port`); sin embargo, el Dockerfile no define otra cosa, así que seguirá usando puerto 8080 (lo cual coincide con el mapeo en compose). Si quisiéramos exponerlo en otro puerto en Docker, podríamos ajustar Compose en la sección ports, sin cambiar la config interna.

- **Variables de Entorno (.env):** El archivo `.env` no está presente en el repo, pero Docker Compose permite tenerlo para separar las configuraciones. En nuestro caso, definieron las env directamente en el YAML de compose:

```
environment:
  POSTGRES_DB: filehashdb
  POSTGRES_USER: filehashuser
  POSTGRES_PASSWORD: 1234
  SPRING_DATASOURCE_URL: jdbc:postgresql://db:5432/filehashdb
  SPRING_DATASOURCE_USERNAME: filehashuser
  SPRING_DATASOURCE_PASSWORD: 1234
```

Las primeras tres son para configurar la BD Postgres inicial (crea la BD y usuario al inicio del contenedor). Las últimas tres son las que lee la aplicación para conectarse. Si se quisiera ocultar `1234` y otros datos, podríamos moverlas a un archivo `.env` no versionado, y en el compose referenciarlas (Docker Compose automáticamente carga .env para sustituir variables). En entornos corporativos, es común manejarlo así o vía herramientas de orquestación/CI que inyectan las secrets.

Para ejecución local sin Docker, esas variables no son necesarias; la app leerá de application.properties. Pero si un desarrollador quisiera usar el perfil docker localmente (por ejemplo, ejecutar `java -jar app.jar --spring.profiles.active=docker` sin Docker), tendría que proporcionar esas variables en su entorno de consola para que la app arranque (de lo contrario, fallaría al no encontrar SPRING_DATASOURCE_URL). Por eso, normalmente uno no activa perfil docker fuera de su contenedor donde se asegura la presencia de variables.

- **Perfiles de Spring Boot adicionales:** En el proyecto no hay otros perfiles definidos (como "prod", "test", etc.), salvo el implicado *test* durante pruebas automáticas:
- Spring Boot activa el perfil `test` cuando ejecutamos tests con SpringBootTest si se desea. Vemos que hay `application-test.properties` (posiblemente vacío o para override de algunas cosas). Es común en tests desactivar seguridad o usar BD H2 para rapidez. Sin embargo, dado que se usan Testcontainers, probablemente en tests se sigue usando Postgres real, con quizás la única diferencia de string de conexión proporcionada por Testcontainers programáticamente.

- Si quisiéramos un perfil "prod" específico, podríamos crearlo similar a docker pero con ciertas diffs (por ejemplo, más pool de conexiones, logging más restricto, etc.), pero en muchos casos el perfil docker ya *es* el productivo, asumiendo que en prod se corre en contenedor. Podríamos tener un perfil "dev" explícito en lugar del default, pero no es necesario.

- **Configuración de Logging:** No se incluyó un archivo logback-spring.xml ni similar, lo que indica que la configuración de logging es la predeterminada: nivel INFO para la mayoría, WARN para algunos frameworks, output por consola. En entornos docker, ese output de consola va al log del contenedor (que se puede ver con `docker-compose logs`). En producción, a veces se redirige a un archivo o a STDOUT según prácticas (12-factor app sugiere STDOUT). Podríamos parametrizar el nivel de logs con env var `logging.level.com.bitsealer=DEBUG` en dev si queremos debug en nuestras clases, por ejemplo.

- **Otras Configuraciones Potenciales:**

- No se ven configuraciones de **correo** ni de otros servicios, puesto que la app no envía emails ni integra APIs externas. De necesitarlo, se agregarían en properties (por ej., SMTP host, etc.).
- **CORS:** Dado que es uso interno, no se configuró. Spring Security por defecto permite CORS si se habilita, pero no es relevante aquí.

- **Tamaño máximo de archivo:** Spring Boot tiene propiedades `spring.servlet.multipart.max-file-size` y `max-request-size` para limitar peso de uploads. No se establecieron, así que se aplican los defaults (1MB por default en Tomcat?). Esto podría ser importante: si se espera sellar archivos grandes, quizás haya que ajustar esos límites. Los desarrolladores deben notar este detalle para configurar según necesidad (por ahora, sin explicitarlos, se usarán los valores por defecto integrados).

Resumen del uso de perfiles:

- En **desarrollo local**, típicamente un desarrollador puede:
- Levantar una base de datos PostgreSQL local (manualmente o vía `docker-compose up db` para solo BD).
- Ejecutar la app desde su IDE o con Maven (`mvn spring-boot:run`). Esto usará application.properties (perfil default) y conectará a `localhost:5432/filehashdb`. Flyway migrará la estructura la primera vez. El dev podrá probar la app en `http://localhost:8080`.
- Alternativamente, el dev puede usar Docker Compose entero: `docker-compose up`. Esto construye la imagen y corre app+db con perfil docker. Para debuggear, quizá es más cómodo sin docker, pero ambas opciones están disponibles.

- En **producción**, lo esperado es usar la imagen Docker generada (por CI o manual con Dockerfile) y desplegarla quizá con Kubernetes, ECS u otra plataforma. En tal caso se usaría igualmente el perfil docker con variables de entorno apropiadas (y contraseñas seguras). El docker-compose proporcionado sirve para entornos controlados o pruebas integrales, pero en prod se puede traducir a un stack más robusto. La configuración externalizada (env vars) facilita eso.

- Para **pruebas automáticas**, probablemente Spring Boot Test auto-configuró una conexión distinta. Sin embargo, dado que integran testcontainers, es probable que en los tests de integración arranquen un contenedor Postgres y ajusten `spring.datasource.url` dinámicamente. Testcontainers suele detectar si hay Flyway y lo ejecuta en ese contenedor. Las migraciones se aplican en un entorno aislado de test para no ensuciar la BD dev.

En conclusión, la gestión de configuración de BitSealer sigue la filosofía de **externalizar parámetros dependientes del entorno**. Mediante perfiles y variables de entorno, la misma aplicación puede correr en contextos distintos sin cambios de código: por ejemplo, apuntar a otra BD, mostrar u ocultar ciertos logs, etc. Los desarrolladores pueden agregar nuevos perfiles (p. ej. un perfil `staging` con BD de prueba) simplemente creando un nuevo archivo de propiedades y activándolo en esa instancia. Esto hace que el proyecto sea **flexible y fácil de desplegar** en múltiples entornos dentro de la empresa, garantizando al mismo tiempo que en cada uno use los valores adecuados (por ejemplo, en producción jamás se usaría `show-sql=true` ni credenciales hardcodeadas, gracias a este mecanismo).

10. Despliegue y Entorno de Ejecución

La aplicación BitSealer está preparada para ejecutarse tanto directamente en un host local (modo desarrollo) como dentro de contenedores Docker (modo producción o integración continua). En esta sección se explican los pasos y requisitos para desplegar la aplicación, así como algunos **comandos útiles** para su operación. También se describen los **requisitos mínimos de entorno** para correr BitSealer eficazmente.

Entorno de Ejecución – Requisitos mínimos:

- **Java:** Se requiere Java 17 (JDK) para compilar y ejecutar la aplicación fuera de Docker. Todos los desarrolladores que quieran correr la app en local deben tener JDK 17 instalado. En caso de usar la imagen Docker, esta ya incluye un JRE 17, por lo que no se necesita Java instalado en el host, sólo Docker.
- **Maven:** Para construir el proyecto desde el código fuente, se necesita Maven 3.6+ (el proyecto usa 3.9.x en Docker). Alternativamente, se puede utilizar Maven Wrapper incluido (`mvnw` script) para no instalar Maven global.
- **Base de Datos:** En modo local, se espera una instancia de PostgreSQL accesible. Mínimo PostgreSQL 14 (versiones ligeramente distintas pueden funcionar, pero se probó con 14). Debe tenerse un usuario `filehashuser` con contraseña `1234` y una base de datos llamada `filehashdb`, o modificar application.properties para otros valores. El esquema inicial se creará automáticamente al correr la app (Flyway migrará).
- **Docker:** Para despliegue con contenedores, se necesitan Docker Engine y Docker Compose (Compose v1 o v2) instalados en el host. Versiones recientes de Docker Desktop ya incluyen Compose v2. El host debe tener suficiente recursos para 2 contenedores (la app Java y Postgres). En términos de **memoria**, recomendamos al menos 512 MB para la app Java (puede ajustarse con flags JAVA_OPTS si se desea) y ~100 MB para Postgres con pocos datos, así que con 1GB RAM total libre es más que suficiente para uso normal. Almacenamiento: la imagen Docker de la app puede pesar ~100 MB; el contenedor DB usará espacio según datos, pero unos pocos MB para tablas e indices iniciales.

Construcción del Artefacto:

- Para compilar y empaquetar la aplicación manualmente, ejecutar en la raíz del proyecto:

  ```
  mvn clean package
  ```

  Esto generará el archivo `target/bitsealer-<version>.jar`. La versión la define Maven (por ejemplo 0.0.1-SNAPSHOT o similar).
- Alternativamente, en entorno con Docker disponible, se puede usar el **Dockerfile** para compilar sin instalar JDK/Maven localmente. El Dockerfile fase 1 hace `mvn package -DskipTests`. Para lanzar la build con Docker manualmente:

  ```
  docker build -t bitsealer-app:latest .
  ```

  Este comando (ejecutado en la raíz donde está Dockerfile) producirá la imagen final etiquetada "bitsealer-app:latest".

Ejecución en Entorno Local (sin Docker):

1. Asegurarse de tener PostgreSQL corriendo en `localhost:5432` con la base de datos y credenciales configuradas según application.properties. Si es la primera vez:
2. Iniciar Postgres.
3. Crear una base de datos `filehashdb`.
4. Crear el usuario `filehashuser` con contraseña `1234` y otorgarle permisos sobre `filehashdb`.

(En un entorno dev se puede simplificar usando un superusuario, pero es preferible simular las credenciales reales).

5. Opcional: Editar `application.properties` si se quiere usar otras credenciales o puerto. O definir variables de entorno `SPRING_DATASOURCE_URL` etc. para sobreescribir los valores (sin perfil docker, Spring Boot respetará las env vars si se proveen).

6. Ejecutar la aplicación:

7. Desde un IDE (Eclipse, IntelliJ) ejecutando la clase `BitSealerApplication` como Java Application.
8. O con Maven: `mvn spring-boot:run` (esto compila e inicia la app en un solo paso).
9. O usando el jar: `java -jar target/bitsealer-...jar`. Por defecto, levantará en puerto 8080. Se puede comprobar en la consola que aparece el banner de Spring Boot y logs indicando la conexión a la BD y la ejecución de migraciones Flyway (veremos en logs "Successfully applied 1 migration" la primera vez).

10. Probar accediendo a `http://localhost:8080/`. Debería aparecer la página de inicio (pública). Desde ahí, flujo de registro -> login -> etc.

11. Comandos útiles en este modo:

12. Para *detener* la aplicación: si se corrió en foreground, Ctrl+C en la terminal termina el proceso. En IDE, presionar el botón de stop.
13. Para limpiar la BD entre pruebas: se puede dropear las tablas o limpiar registros manualmente. Alternativamente, para re-aplicar migraciones de cero, dropear schema public y volver a lanzar la app (no hacerlo en entornos con datos reales sin respaldo).
14. Logs: en este modo van a la consola estándar. Se podría configurar un logging.file para guardar en archivo, pero por defecto se leen de STDOUT.

Despliegue con Docker Compose (recomendado para integraciones y producción sencilla):

El Docker Compose facilita levantar tanto la app como la BD en un solo paso, con la configuración adecuada:

1. Asegurarse de haber construido la imagen Docker de la app. En desarrollo se puede usar Compose mismo para que la build (Dockerfile) ocurra automáticamente. Compose tiene la directiva `build: .` para el servicio app, así que lo hará. No obstante, se puede ejecutar manualmente `docker-compose build app` para forzar la construcción previa (útil si se cambió código y se quiere recrear la imagen).

2. Ejecutar:

   ```
   docker-compose up
   ```

   Esto iniciará ambos contenedores. La primera vez descargará la imagen de Postgres:14-alpine (unos ~5MB compressed) y construirá la imagen de BitSealer (descargando la imagen base JDK para compile, etc.). Puede tardar un par de minutos la primera vez.

Veremos en la consola los logs combinados de DB y app. La BD primero inicializa (creando el usuario y base segun env). Luego la app arranca, intentará conectarse a `db:5432`. Puede que retrase unos segundos hasta que Postgres esté listo; en Compose `depends_on` garantiza orden de arranque pero no espera readiness, aunque Spring Boot por defecto reintenta conexiones un par de veces. En logs de la app se observará Flyway migrando y luego Tomcat arrancando en port 8080.

3. Una vez ambos servicios estén "Up", la aplicación estará accesible en `http://localhost:8080` (porque Compose publica 8080 del contenedor app a 8080 host). Probar registro/login como de costumbre. Los datos persistirán en el volumen `db_data` incluso si paramos los contenedores, a menos que explícitamente se borre el volumen.

4. Comandos útiles con Docker Compose:

5. `docker-compose up -d`: Levanta en *detached mode*, es decir, en segundo plano. La consola devuelve control inmediatamente. Podemos luego ver logs con `docker-compose logs -f` (follows logs en tiempo real) o ver estado con `docker-compose ps`.
6. `docker-compose down`: Para y elimina los contenedores. Con `-v` además eliminaría el volumen `db_data` (¡cuidado!, sin `-v` mantiene el volumen, lo que usualmente queremos para no perder datos entre reinicios).
7. `docker-compose restart app` / `docker-compose restart db`: Reinicia un servicio si por alguna razón se necesita (por ejemplo, aplicamos nueva imagen de app tras cambios).
8. Si se actualiza el código y se quiere desplegar la nueva versión: hay que reconstruir la imagen (ej. `docker-compose build app` o `docker-compose up --build` directo). Compose recreará el contenedor app con la nueva imagen.
9. Para entrar a la base de datos (por depuración): `docker-compose exec db psql -U filehashuser filehashdb` abrirá un prompt psql dentro del contenedor.

10. Para inspeccionar la BD desde afuera, se puede conectar con PgAdmin o psql local a localhost:5432 (ya que se mapeó ese puerto) usando las credenciales. De esa forma se pueden ver tablas `users`, `file_hashes` y `flyway_schema_history`.

11. **Despliegue en producción real:** Si se usara este proyecto en producción, probablemente se integraría en un pipeline CI/CD:

12. El pipeline construiría el artefacto (ejecutando tests).
13. Luego construiría la imagen Docker y la subiría a un registro privado.
14. Después, se desplegaría en un servicio de contenedores (Kubernetes, Docker Swarm, AWS ECS, etc.). Como el app es una *12-factor app* (config via env), bastaría con pasar las env correctas de producción (contraseñas seguras, etc.).
15. En prod, se montaría un volumen o base de datos administrada para persistir los datos Postgres más allá del contenedor.
16. También se pondría la app detrás de un proxy con HTTPS.

Consideraciones de rendimiento y tamaño: - La imagen Docker final basada en Alpine con JRE es bastante ligera (decenas de MB). Sumado Postgres, el footprint es bajo. En cuanto a CPU/memoria, la aplicación al ser Java puede llegar a consumir 100-200MB RAM en funcionamiento dependiendo de la carga (Spring Boot tiene overhead). Con pocos usuarios concurrentes, 256MB-512MB de heap deberían sobrar. Postgres con esta carga también usará poco (<100MB). - Escalabilidad: Al ser monolítica, escalar implicaría replicar la instancia de la app y usar un balanceador; la BD seguiría única (pues es el punto de verdad). Dado el uso interno, no se espera alta concurrencia, pero es bueno saber que el contenedor

app se puede replicar (session sticky or external session store sería necesario si se replicara, por el uso de sesiones). - **Comandos de mantenimiento de BD:** Flyway se encargará de migraciones en cada arranque. Si por alguna razón una migración falla, la app no iniciará correctamente (Flyway marca error). En tal caso, revisar el script, corregir y reintentar (Flyway no aplicará migración fallida hasta que se limpie su registro o se reemplace por una con mayor versión, etc.). Siempre probar migraciones en staging antes de prod para evitar sorpresas.

Acceso por desarrolladores e IA: - Un nuevo desarrollador puede levantar todo con `docker-compose up` sin conocer detalles internos, gracias a la configuración proporcionada. Esto agiliza onboarding. - Para pruebas manuales de API (aunque no hay API REST pública, solo navegación web), se puede usar curl o herramientas de test con la cookie de sesión tras login, pero dado que es web con formularios, es más sencillo interactuar vía navegador. - Integración con herramientas de IA: Podrían desplegar la app en un contenedor sandbox para que el agente AI interactúe con ella vía HTTP para pruebas end-to-end (por ejemplo, un test de login y upload). La documentación de los endpoints (como la dada en flujo de uso) es suficiente para que sepan qué llamadas hacer.

En resumen, el despliegue de BitSealer es **straightforward** gracias al uso de Docker Compose. Para entornos profesionales, se recomienda usar la misma imagen Docker en todos los entornos (dev, QA, prod) cambiando solo configuración, asegurando paridad. Los comandos proporcionados permiten gestionar la aplicación con facilidad. Los requisitos de sistema son modestos, y cualquier máquina capaz de correr Docker o Java 17 y Postgres puede alojar BitSealer. Esto hace que la aplicación sea **portátil y de fácil mantenimiento** en cuanto a infraestructura.

11. Notas de Mantenimiento y Calidad del Código

En esta sección final se abordan consideraciones de mantenimiento del proyecto BitSealer, señalando posibles mejoras, refactorizaciones y aspectos de calidad de código. Si bien la aplicación es plenamente funcional, siempre hay áreas que se pueden optimizar para hacerla más robusta, legible o extensible. También se mencionan brevemente algunas partes de código que no se usan actualmente, para clarificar su propósito o si podrían eliminarse para evitar confusión.

Clases y Código No Utilizado:

- La clase `ViewController` en el paquete `controller` actualmente no contiene métodos ni mappings activos. Parece haber sido creada con la intención de manejar rutas de vistas estáticas ("vistas muertas") para evitar duplicar lógicas en otros controladores. Dado que quedó vacía, se podría:
- O bien eliminarla del proyecto para reducir ruido,
- O bien usarla para su propósito original en el futuro si se agregan páginas simples.
  Dejar código muerto puede confundir a nuevos desarrolladores que podrían preguntarse si falta implementar algo allí. Por lo tanto, si no se prevé usarlo en el corto plazo, convendría remover `ViewController` del registro de componentes (o al menos documentar en un comentario su posible uso futuro).
- La clase `UserDto` no está siendo utilizada en ningún punto del flujo actual. Todos los usos de información de usuario se manejan con la entidad AppUser directamente (por ejemplo, en registro se pasa AppUser al form, en login se usa UserDetails de Spring Security). `UserDto` pudo haber sido pensado para exponer datos de usuario sin contraseña en alguna API REST o funcionalidad (por ejemplo, una página de perfil de usuario). Al no estar integrada actualmente,

es **código muerto**. Se puede mantener si se prevé su uso en próximas iteraciones, pero habría que evitar instanciarlo o actualizarlo fuera de contexto. Si no, eliminarlo hasta necesitarlo podría mantener el códigobase más limpio.

- Comentarios de código evidencian ciertas partes no usadas:

- En `FileUploadController`, inicialmente parece que se consideró recibir directamente un `FileHash` en vez de `FileUploadRequest` (hay un import comentado y un comentario `// (si ya cambiaste a entity)`). Esto sugiere que en una versión anterior se populaba una entidad desde el formulario (lo cual es posible pero no recomendable, por riesgo de sobreescritura de campos no deseados). Finalmente se optó por el DTO FileUploadRequest. El código comentado asociado a ese enfoque antiguo debería eliminarse para evitar confusión, ya que mantener imports comentados o líneas obsoletas hace más difícil leer el código. El control de versiones (git) ya preserva el historial, no es necesario tener código comentado legado.
- Comentarios como `// Todo OK → volvemos al login con indicador ?registered` en AuthController indican tareas completadas pero no actualizadas en comentarios. Es decir, el código ya hace el redirect con `?registered`, por lo que el `// Todo` podría inducir a pensar que falta algo cuando en realidad está hecho. Sería conveniente limpiar esos comentarios o marcarlos como resueltos para evitar malentendidos a futuros mantenedores.

Duplicación y Refactorización:

- Actualmente, para obtener el usuario autenticado en los controladores `FileUploadController`, se repite la secuencia: `Authentication auth = SecurityContextHolder.getContext().getAuthentication(); String username = auth.getName(); userService.getByUsername(username)`. Esta lógica aparece tanto en `procesarArchivo` como en `verHistorial`. Es mínima duplicación, pero se podría factorizar. Opciones:
- Utilizar la anotación `@AuthenticationPrincipal` en los métodos del controlador para recibir directamente el principal (UserDetails) como parámetro. Spring Security permite algo como: `public String verHistorial(@AuthenticationPrincipal User userDetails, Model model)`. Entonces se tendría el username con `userDetails.getUsername()` sin acceder manualmente al SecurityContext. Esto reduce acoplamiento a SecurityContextHolder y hace más testeable el método.
- Crear quizás un método utilitario estático en una clase de utilidades de seguridad (o aprovechar directamente `auth.getPrincipal()` que en nuestro caso devuelve un `UserDetails`). Se optó por la vía simple pero repetitiva. Un refactor pequeño podría mejorar la claridad.
- Uso de entidades vs DTO en formularios: En registro, se está usando `AppUser` entidad directamente como `@ModelAttribute("userForm")`. Esto puede ser práctico pero conlleva que todos los campos de AppUser (incluido id, role) están vinculados al formulario. Por suerte, Thymeleaf form probablemente solo mapea username, email, password, ignorando `id` y `role` a menos que estuvieran en el form. Aun así, es una buena práctica usar un **DTO específico para registro** (ej. `RegisterRequest` con solo username, email, password, confirmPassword si se desea) en lugar de la entidad. Esto desacopla totalmente la capa web de la capa de datos. Dado que implementaron UserDto para salidas, quizás se les pasó crear uno para entradas. Un refactor futuro podría introducir `RegisterDto` y convertirlo a AppUser en el servicio, aplicando validaciones (por ejemplo, confirmar contraseña igual a confirmación, etc.). No es obligatorio, pero mejora la claridad de intención.
- Manejo de excepciones: Actualmente, UserService lanza IllegalArgumentException para duplicados de email/username, y AuthController lo atrapa para mostrar mensaje en la vista. Esto funciona, pero usar excepciones para control de flujo en validación no es siempre ideal.

Alternativa: podría validar antes de llamar al servicio en el controlador (llamando a userService.existsByEmail y existsByUsername y usando BindingResult.addError(...) para que Thymeleaf marque el campo como error). Sin embargo, concentrarlo en el servicio garantiza una sola verificación en la lógica. Es aceptable. Si se quisiera mejorar, se podrían definir excepciones personalizadas (e.g., `EmailAlreadyUsedException`) para distinguir casos, en lugar de IllegalArgumentException genérica. También se podría internacionalizar los mensajes de error en messages.properties para soportar otros idiomas sin cambiar el texto en código.

- **Código repetido o redundante:**
- El mapper FileHashMapper es trivial (mapea todos campos 1-1). Podría eliminarse en favor de usar Stream map con constructor de record: `fileHashes.stream().map(f -> new FileHashDto(f.getId(), f.getFileName(), f.getSha256(), f.getCreatedAt())).toList()`. No obstante, tener el mapper separado es más limpio si en algún momento cambia la estructura. Quizás se podría marcar el mapper como `@Component` e inyectarlo, en lugar de usarlo estáticamente (depende implementación actual, en el code snippet parecía usarlo instanciado vía `mapper.toDto(...)`). Este es un detalle menor.
- CustomUserDetailsService: Combina la autoridad como `new SimpleGrantedAuthority(user.getRole())`. Dado que en la BD se almacenó "ROLE_USER", está bien. A veces se podría almacenar solo "USER" y prefix en memoria, pero eso es cuestión de estilo. No hay duplicación real ahí.

- Podría considerarse un *refactor* futuro separar la lógica de hashing a un componente específico (por ejemplo, un servicio HashingService que encapsule DigestUtils) para posibilitar cambiar de algoritmo más fácilmente o añadir logging alrededor de hashing. Actualmente, se llama DigestUtils directamente dentro de FileHashService. Es simple y está bien; una abstracción adicional no es necesaria a este tamaño de proyecto.

- **Posibles mejoras de funcionalidades:**

- **Tamaño máximo de archivo y validación de tipo:** Actualmente cualquier archivo se puede subir. Podría valer la pena limitar el tamaño (para que no saturen el servidor con un archivo gigante cuyo hash igualmente se puede calcular pero tardaría). Spring permite configurar un límite global; también se podría validar en `FileUploadController` si `file.getSize()` excede X MB y rechazarlo con mensaje. Dado entorno controlado, quizás no era prioritario.
- **Historial paginado:** Si un usuario sube cientos de archivos, `history.html` puede volverse muy larga. Una mejora es implementar paginación o al menos limitar los mostrados y ofrecer descargar CSV del historial completo si fuera necesario. Spring Data JPA ofrece métodos que devuelven Page<FileHash> fácilmente. Por ahora, para pocos registros, una lista simple está bien.
- **Búsqueda en historial:** Podría agregarse capacidad de buscar un hash en específico o filtrar por nombre de archivo, etc., si el proyecto lo requiriera. Esto sería un feature nuevo más que refactor.
- **Roles y administración:** Actualmente no hay interfaz para distinguir roles ni para que un administrador vea todos los usuarios o sus archivos (por políticas internas podría ser útil tener un admin que audite). Implementarlo requeriría: un rol ADMIN, endpoints solo accesibles a ADMIN, vistas para listar usuarios y/o todos los files. Esto implicaría bastantes añadidos pero está encaminado con la infraestructura actual (ya almacenamos role, Spring Security puede manejar roles).

- **Internacionalización (i18n):** Todos los mensajes están en español incrustados (ya que la app es para contexto hispanohablante interno, eso tiene sentido). Si se quisiera hacer multilingüe, se

podrían externalizar textos a `messages.properties`. No es prioridad en un entorno corporativo cerrado con un idioma principal, pero es una mejora de calidad general.

- **Calidad del Código:**

- En general, el código es claro y sigue convenciones Java (nombres descriptivos, paquetes bien organizados). El uso de comentarios es abundante (sobre todo en SecurityConfig explicando cada sección). Eso es positivo para aprendizaje, aunque en un entorno puramente profesional se podría limpiar un poco esos comentarios formativos. Sin embargo, dado que esta documentación pretende también servir a herramientas de IA y nuevos devs, tener comentarios en el código es beneficioso.
- No se detectan problemas de estilo mayores (posiblemente falta formatear consistentemente llaves o indentación en algún archivo, pero nada crítico).
- Sería recomendable agregar más **pruebas unitarias/integración** si el proyecto evolucionará: actualmente hay tests de FileHashService y un test de UploadIntegration. Faltan tests de UserService (registro) y quizá de AuthController (usando MockMvc). Incluir esos aseguraría que futuras modificaciones en registro o seguridad no rompan funcionalidades (por ejemplo, se podría testear que no se pueda registrar duplicados, etc.).
- Dependencias: No hay dependencias innecesarias en el pom; quizás `flyway-database-postgresql` es prescindible porque `flyway-core` suele bastar (flyway-core detecta Postgres sin necesidad de plugin específico en las últimas versiones). Si no es necesaria, quitarla reduce un poco el peso.

- La estructura del proyecto es coherente. Podría evaluarse en el futuro modularizarlo (por ejemplo, separar un módulo core y otro web) si creciera mucho, pero actualmente es innecesario.

- **Gestión de la calidad y análisis estático:** Sería bueno integrar herramientas como SonarQube o Checkstyle/SpotBugs en la etapa de CI para detectar code smells o potenciales bugs. Por ejemplo:

- Sonar podría advertir sobre el uso de `SecurityContextHolder.getContext().getAuthentication().getName()` repetido, o sobre métodos públicos en entidades (podrían ser package-private si no usados fuera, etc.).
- Checkstyle podría imponer formateo consistente.
- SpotBugs posiblemente no encontraría mucho al ser código straightforward, pero siempre es útil.

- Estas medidas aseguran mantenibilidad a largo plazo, especialmente si más colaboradores se unen.

- **Mantenimiento de Dependencias:** Mantener Spring Boot y otras librerías actualizadas es parte de la calidad. Actualmente usan Spring Boot 3.5.0 (posiblemente milestone ya que Boot 3.1 es estable, 3.5 suena a versión futura). Habría que estar atentos a actualizaciones de seguridad en Spring Security, etc. Flyway, Hibernate y PostgreSQL driver conviene actualizarlos conforme salgan versiones, siempre probando migraciones en un entorno de prueba.

- **Documentación y Comentarios:** A nivel de documentación, este **README** técnico proporciona un amplio contexto. Sería útil complementarlo con:

- Un README más breve para el repositorio con pasos rápidos de ejecución (para devs que solo quieran instrucciones rápidas).
- Documentación del API (aunque no hay API pública REST, podríamos documentar las URLs web, lo cual ya hicimos en flujo).
- Comentarios JavaDoc en clases y métodos principales (por ejemplo en servicios) explicando su propósito. Algunos existen (AuthController tiene JavaDoc en métodos). Esto es bueno para cuando alguien lee el código en IDE con tooltips.
- Posiblemente diagramas UML sencillos (de entidad-relación o de secuencia) en el futuro para explicar el flujo, aunque no son imprescindibles dado que el código es manejable.

En conclusión, desde una perspectiva de **mantenimiento y calidad** el proyecto BitSealer está bien encaminado: presenta una estructura clara y utiliza prácticas comunes en Spring Boot. Las mejoras sugeridas (remover código no usado, factorizar pequeñas duplicaciones, endurecer la seguridad de CSRF, agregar validaciones adicionales) pueden ser abordadas en próximas iteraciones para incrementar la robustez. Ninguna de estas cuestiones es crítica, por lo que el proyecto es **estable** en su estado actual. Sin embargo, atenderlas incrementará la legibilidad y reducirá riesgos futuros, facilitando que nuevos desarrolladores (humanos o IA) entiendan el código sin tropiezos innecesarios. Continuar con un enfoque de limpieza de código constante y pruebas automatizadas garantizará que BitSealer mantenga un alto estándar de calidad a medida que evolucione.

---

1 Qué es Spring Boot - Qindel: Consultoría IT
https://www.qindel.com/que-son-los-microservicios-spring-boot/

2 6 7 8 Password Storage :: Spring Security
https://docs.spring.io/spring-security/reference/features/authentication/password-storage.html

3 4 Flyway: formas de configurar y usos - Adictos al trabajo
https://adictosaltrabajo.com/2022/12/21/flyway-formas-de-configurar-y-usos/

5 9 www.elpuntojs.com
https://www.elpuntojs.com/blog/spring-flyway