ChatGPT

Plan de Implementación JWT y Integración Frontend/Backend en BitSealer

1) Resumen ejecutivo

En el **backend** se eliminan las vistas Thymeleaf y se habilita una API REST JSON: los controladores usan `@RestController` para devolver datos JSON (ya no renderizan plantillas) [1] [2]. Sólo se hacen cambios mínimos en Spring Boot: se desactiva la autenticación por formulario y sesión, sustituyéndola por JWT. Se agrega un filtro JWT para validar tokens en cada petición y se configuran CORS apropiados. En el **frontend** (React) se concentra la mayor parte del trabajo: se integra el flujo de login/registro con llamadas a la API JWT, se almacenan de forma segura los tokens (en **localStorage** con opción "Recordarme") y se incluyen en el encabezado `Authorization` de las peticiones. Se crean las páginas protegidas de **Subida** (`/upload`) e **Historial** (`/history`) y se conectan los datos reales en el **Dashboard** existente, reemplazando las simulaciones. Se han verificado compatibilidades de versiones: **JJWT 0.12.5** funciona con Java 17/Spring Security 6 (ajustando el uso de su API de parsing) y **Spring Boot 3.5/Security 6** exige la nueva configuración de seguridad basada en `SecurityFilterChain` en lugar de `WebSecurityConfigurerAdapter`. Se configura **CORS** para permitir el origen del frontend (p. ej. `http://localhost:5173`) sin errores de bloqueo. El flujo de autenticación completo será: registro → login (JWT emitido) → token almacenado en front → peticiones subsecuentes con `Bearer token` → backend valida JWT en cada request → datos JSON devueltos.

2) Plano de integración (repositorio y despliegue)

Repositorio y ramas: Se sugiere crear una rama de feature, por ejemplo `feature/jwt-integration`. Empezar aplicando cambios de backend mínimos (desactivar Thymeleaf, añadir JWT) y luego integrar el frontend. Commits recomendados en orden: *(1)* "Remove Thymeleaf and add JWT security (backend)" *(2)* "Integrate JWT in React frontend (login & auth context)" *(3)* "Implement file upload & history pages (frontend)" *(4)* "Update tests for JWT and file endpoints" y finalmente merge via Pull Request.

Comandos y ejecución local:
- **Base de datos:** Levantar PostgreSQL 14 (por ejemplo con Docker: `docker run -p 5432:5432 -e POSTGRES_DB=filehashdb -e POSTGRES_USER=filehashuser -e POSTGRES_PASSWORD=1234 postgres:14`). Asegurarse de que la URL y credenciales coinciden con `application.properties`.
- **Backend:** Con Java 17, ejecutar `./mvnw spring-boot:run` en el directorio `backend` (o `bitsealer`). Esto iniciará la aplicación en `http://localhost:8080`. Para entorno Docker, existe el perfil `application-docker.properties` – por ejemplo, construir la imagen con Maven y correr con `docker run -p 8080:8080 --env SPRING_DATASOURCE_URL=... --env SPRING_DATASOURCE_USERNAME=... --env SPRING_DATASOURCE_PASSWORD=... --env JWT_SECRET=<clave JWT> bitsealer:latest`. Asegúrate de definir la variable de entorno **JWT_SECRET** (clave secreta para firmar JWT) antes de arrancar el backend.
- **Frontend:** Dentro de `front` (proyecto React), instalar dependencias (`npm install`) y arrancar en modo desarrollo con `npm run dev`. Esto abre el front en `http://localhost:5173` (Vite por defecto). El front espera la URL base de la API en la variable `VITE_API_BASE`. Configura un archivo

`.env` en el proyecto React con `VITE_API_BASE="http://localhost:8080/api"`. Así, las llamadas (ej. a `/auth/login`) se dirigirán al backend.

Variables de entorno necesarias:
- En backend: `JWT_SECRET` (secreto para firmar/verificar JWT). En entorno Docker también `SPRING_DATASOURCE_URL`, `SPRING_DATASOURCE_USERNAME`, `SPRING_DATASOURCE_PASSWORD` según la base de datos.
- En frontend: `VITE_API_BASE` (URL base de la API, p. ej. `http://localhost:8080/api`).

Orden de despliegue recomendado: primero asegurarse de que la base de datos está accesible y migrada (Flyway ejecutará `V1__init.sql` al iniciar backend). Arrancar el backend (ver logs de arranque en consola para confirmar). Luego iniciar el frontend de React. Probar manualmente el registro y login desde la interfaz de React; tras login exitoso (debe almacenarse el token), navegar al **Dashboard** (ruta `/dashboard`), **Subir Archivo** (`/upload`) y **Historial** (`/history`) para confirmar que funcionan sin errores CORS ni de autorización.

3) API Contract final (REST JSON)

A continuación se describen los endpoints expuestos por el backend, con sus métodos, rutas, cuerpos de solicitud, respuesta y códigos relevantes:

- **POST** `/api/auth/register` – Registra un nuevo usuario.
  Body: JSON con campos como `name` (nombre de usuario), `email` y `password`. Ejemplo:

  ```
  { "name": "Alice", "email": "alice@example.com", "password": "secret" }
  ```

  Response: JSON con tokens JWT y datos del usuario registrado. Ejemplo (éxito 201 Created):

  ```
  {
    "accessToken": "<JWT de acceso>",
    "refreshToken": "<JWT de refresh>",
    "user": { "id": 1, "username": "Alice", "email": "alice@example.com" }
  }
  ```

  Devuelve *201 Created* con el cuerpo arriba si registro exitoso. Posibles errores: *400 Bad Request* si faltan datos o formato inválido (p.ej. email ya en uso), *409 Conflict* si el email/username ya existe (se indica mensaje de error), *500 Internal Server Error* en error inesperado.

- **POST** `/api/auth/login` – Autentica un usuario existente.
  Body: JSON con `email` y `password`. Ejemplo:

  ```
  { "email": "alice@example.com", "password": "secret" }
  ```

  Response: JSON con tokens y datos del usuario (formato igual a registro). Ejemplo (200 OK):

```
{
  "accessToken": "<JWT de acceso>",
  "refreshToken": "<JWT de refresh>",
  "user": { "id": 1, "username": "Alice", "email": "alice@example.com" }
}
```

En caso de credenciales inválidas devuelve *401 Unauthorized* (sin cuerpo o con mensaje de error). *Nota:* Si se implementa mecanismo de refresh, el `refreshToken` se incluye; de lo contrario, puede omitirse o ser igual al accessToken para compatibilidad.

- **GET** `/api/users/me` – Obtiene el perfil del usuario autenticado (quien hizo la petición).
  Headers: Requiere `Authorization: Bearer <accessToken>`.
  Response: JSON con la información básica del usuario (id, username, email). Ejemplo (200 OK):

```
{ "id": 1, "username": "Alice", "email": "alice@example.com" }
```

Posibles errores: *401 Unauthorized* si falta token o es inválido/expirado (la respuesta 401 no tendrá cuerpo JSON, solo cabezera WWW-Authenticate si se configura).

- **POST** `/api/files/upload` – Sube un archivo para sellar (protegido, requiere token).
  Headers: `Authorization: Bearer <accessToken>` (y `Content-Type: multipart/form-data`).
  Body: FormData con un campo `file` que contenga el archivo a subir. *(También se podría soportar JSON con un hash pre-calculado, pero aquí asumimos que se envía el archivo y el backend calcula SHA-256).*
  Response: JSON con los datos del archivo registrado en el sistema. Ejemplo (201 Created):

```
{
  "id": 10,
  "originalFilename": "contrato.pdf",
  "sha256":
"A3F5B2C9D1E8F6A7B9C0D123E4F56789ABCDEF0123456789ABCDEF0123456789",
  "createdAt": "2025-09-15T17:35:20.123",
  "ownerId": 1
}
```

(Nota: `ownerId` *puede omitirse si no se desea exponer, en tal caso solo nombre, hash y fecha).* Códigos: *201 Created* si éxito (devuelve el JSON arriba), *400 Bad Request* si no se envía archivo. *401 Unauthorized* si no está autenticado.

- **GET** `/api/files/history` – Lista el historial de archivos sellados por el usuario actual.
  Headers: `Authorization: Bearer <token>` requerido.
  Response: JSON con array de archivos (ordenados por fecha desc). Ejemplo (200 OK):

```
[
  {
```

```
    "id": 10,
    "originalFilename": "contrato.pdf",
    "sha256": "A3F5...6789",
    "createdAt": "2025-09-15T17:35:20.123"
  },
  {
    "id": 7,
    "originalFilename": "foto.png",
    "sha256": "C1D2...6789",
    "createdAt": "2025-09-14T09:10:05.456"
  }
]
```

(Solo se listan archivos del usuario que hace la petición. El campo `owner` *se omite por no ser necesario en su propio historial.)*
Soporta en el futuro parámetros de paginación (e.g. `?page=1&size=20`) y filtrado/orden, pero en esta versión devuelve todos los registros del usuario con orden descendente por fecha.
Códigos: *200 OK* siempre que el token sea válido (incluso lista vacía `[]` si no hay archivos); *401 Unauthorized* si token inválido.

- **GET** `/api/dashboard` – *(Nuevo, consolidado para panel)* Obtiene datos agregados para el dashboard. **(Protegido con token)**.
Headers: `Authorization: Bearer <token>`.
Response: JSON con métricas clave, últimos archivos y datos de gráficas. Ejemplo:

```
{
  "kpis": [
    { "label": "Archivos hoy", "value": 3, "delta": "+5%", "trend":
"up" },
    { "label": "Total sellados", "value": 15, "delta": "+20%", "trend":
"up" },
    { "label": "Confirmación media", "value": 42, "suffix": " min",
"delta": "-4%", "trend": "down" },
    { "label": "Fallos", "value": 0, "delta": "0", "trend": "neutral" }
  ],
  "recent": [
    { "nombre": "contrato.pdf", "fecha": "2025-09-15", "hash":
"A3F5...6789" },
    { "nombre": "foto.png", "fecha": "2025-09-14", "hash":
"C1D2...6789" }
  ],
  "charts": {
    "line":  { "labels": ["Lun","Mar","Mié","Jue","Vie","Sáb","Dom"],
"data": [2,5,3,0,4,1,0] },
    "bar":   { "labels": ["Madrid","Barcelona","Valencia"], "data": [10,
3,2] },
    "donut": { "labels": ["Directo","Empresas","Afiliados"], "data":
[60,25,15] }
```

```
  }
}
```

Nota: Estas cifras de ejemplo ilustran el formato; en la implementación actual, `kpis` se calcularán con: archivos subidos "hoy" (desde las 00:00), total de archivos del usuario, etc., mientras que los gráficos pueden devolverse con datos simulados o calculados simples. Este endpoint permite al frontend del **Dashboard** obtener todo en una sola llamada. Códigos: *200 OK* (o *401 Unauthorized* si falta token).

4) Seguridad (Spring Security 6 + JWT)

Se actualiza la configuración de seguridad para **sesiones stateless con JWT**. En lugar del login basado en formulario, se define un bean `SecurityFilterChain` que:
- **Deshabilita CSRF** (no es necesario para APIs stateless) [3] [4].
- **Configura CORS** permitiendo el origen del frontend. Se registra un `CorsConfigurationSource` global que autoriza dominios específicos (p. ej. `http://localhost:5173`) y métodos típicos (GET, POST, etc.), y se habilita `http.cors()` para integrar esta configuración [5] [3]. Esto evita bloqueos en navegadores: las preflight requests OPTIONS serán aceptadas correctamente antes de la autenticación.
- **Define las rutas públicas vs protegidas:** se permiten sin autenticación las URLs de registro y login (`/api/auth/**`) y recursos estáticos si quedaran (en este caso, al haber separado front, se podrían bloquear todo menos `/api/**`). Todas las demás rutas (`/api/**` restantes) requieren token (authenticated) [4].
- **Desactiva la página de login y logout por defecto:** se quita `.formLogin` y `.logout` para que Spring Security no intercepte `/login` ni maneje sesiones. En su lugar, la autenticación la realiza nuestro endpoint `/api/auth/login` que genera un JWT. También se añade `.sessionManagement().sessionCreationPolicy(STATELESS)` para que Spring Security no guarde ni cree sesiones HTTP [6] [7].
- **Registra el filtro JWT personalizado:** mediante `.addFilterBefore(jwtFilter, UsernamePasswordAuthenticationFilter.class)` se inserta nuestro filtro antes del filtro de autenticación de formulario (deshabilitado) [8]. Este **JwtAuthFilter** se ejecuta en cada petición entrante: busca el header `Authorization: Bearer ...`. Si existe un token JWT, el filtro lo verifica usando la clave secreta (**JWT_SECRET**) con la librería JJWT. Si la firma/estructura es válida y no expiró, extrae el identificador del usuario (en nuestro caso usamos el email como `subject` del token) y carga los detalles del usuario desde la base (usando `CustomUserDetailsService`). Luego crea un objeto de autenticación **UsernamePasswordAuthenticationToken** con esas credenciales y roles y lo coloca en el contexto de seguridad (SecurityContext). Así, las siguientes capas saben que el usuario está autenticado. Si el token es inválido o ausente, el filtro sencillamente no autentica a nadie (la petición será rechazada más adelante por `HttpSecurity` si necesitaba auth).
- **Manejo de errores 401/403:** Si una petición a un endpoint protegido llega sin un token válido, Spring Security automáticamente responderá *401 Unauthorized*. Podemos mejorar la claridad devolviendo un mensaje JSON de error o un header `WWW-Authenticate`, pero con JWT típicamente basta el 401. Un token válido pero sin los roles necesarios daría *403 Forbidden* (en esta app, todos los usuarios son role USER y no hay endpoints diferenciados por rol, así que no se produce 403 normalmente).

Sobre **JJWT 0.12.5**: se usó su API moderna (`Jwts.parserBuilder()`) para validar tokens con la clave secreta. La secreta se define en una variable de entorno `JWT_SECRET` y debería ser robusta (mínimo 32 bytes para HS256). Al generar tokens usamos `Jwts.builder()` con `signWith(key)` donde key es la versión binaria de JWT_SECRET (vía `Keys.hmacShaKeyFor(...)`). La **firma HMAC SHA-256** se añade automáticamente en el header del JWT [9]. Para parsear, usamos

`Jwts.parserBuilder().setSigningKey(key).build().parseClaimsJws(token)` para obtener los `Claims` de forma segura; si el token fue modificado o expiró, esto lanzará excepciones que manejamos para rechazar la autenticación [10]. Hemos comprobado que 0.12.5 introduce métodos nuevos (`verifyWith` / `parserBuilder`) en lugar de los obsoletos de versiones previas, ajustados correctamente [11] [12].

DTOs de autenticación y errores: Se añaden clases DTO para el intercambio JSON claro: `LoginRequest` / `RegisterRequest` (con campos simples), `JwtResponse` (con `accessToken`, `refreshToken` opcional y un `UserDto` interno). Los controladores usan estos DTOs para mapear tanto la entrada (mediante `@RequestBody`) como la salida (retornando objetos o `ResponseEntity` que Spring convierte a JSON). Por ejemplo, en `AuthController.login()` se valida el `LoginRequest` y devuelve un `ResponseEntity.ok(new JwtResponse(...))`. Para errores, se puede lanzar excepciones personalizadas o devolver `ResponseEntity.status(...)` con un mensaje. En esta implementación, por simplicidad, devolvemos códigos adecuados sin un cuerpo detallado en la mayoría de casos (el cliente puede manejarlo con mensajes genéricos como "Credenciales inválidas" en 401, etc.).

Política de CORS final: Se permitió el origen del front en desarrollo (**http://localhost:5173**). Se configuran `allowedHeaders` incluyendo `Content-Type` y `Authorization`, y `allowedMethods` GET, POST, PUT, DELETE, etc. Además `allowCredentials(true)` si en un futuro se optara por enviar el token en cookie (no es el caso ahora). Esto está alineado con la documentación oficial: *"CORS must be processed before Spring Security... The easiest way is providing a CorsConfigurationSource bean with allowed origins."* [13] [5]. Con esta configuración, al ejecutar en local no habrá errores CORS (el navegador permitirá las llamadas del puerto 5173 al 8080). En producción, se deben ajustar los orígenes permitidos según el dominio real del front.

5) Reemplazo de Thymeleaf por API REST

Se quita la dependencia **Spring Boot Starter Thymeleaf** del proyecto, junto con el extra de seguridad para Thymeleaf. En el `pom.xml` se eliminan: `<artifactId>spring-boot-starter-thymeleaf</artifactId>` y `<artifactId>thymeleaf-extras-springsecurity6</artifactId>`. Ya no se renderizarán vistas en el servidor. En su lugar, los controladores devuelven directamente datos (objetos o listas) que Spring Boot serializa a JSON automáticamente gracias a Jackson (incluido por defecto en starter-web) [2] [14]. Por ejemplo, antes existía `AuthController` con métodos que retornaban nombres de plantilla `"login"` o redirecciones. Ahora, `AuthController` se marca con `@RestController` (lo que equivale a `@Controller + @ResponseBody` para todos los métodos) [15] y sus métodos `register` / `login` retornan objetos DTO (`JwtResponse`) en JSON. No se envían más `ModelAndView` ni se usan `RedirectAttributes`; en cambio se maneja todo vía códigos HTTP y cuerpos JSON.

Los archivos de plantilla HTML en `src/main/resources/templates` (`login.html`, `register.html`, etc.) **ya no se utilizan** y se podrían suprimir del proyecto (lo mismo para recursos estáticos como JS/CSS del antiguo dashboard basado en SB Admin 2, que ahora reside en el front React). El backend queda reducido a una API REST pura en la ruta `/api/**`. Incluso la página principal `/` que antes servía `home.html` puede deshabilitarse – por simplicidad, se podría dejar `HomeController` devolviendo un mensaje o nada, pero dado que el front se aloja aparte, no es necesario servir contenido HTML.

Los controladores existentes se adaptaron con cambios mínimos:
- **AuthController:** Se transformó en REST. Se eliminaron los `@GetMapping("/login")` y

`@GetMapping("/register")` que devolvían vistas. El nuevo `@PostMapping("/register")` recibe JSON (`RegisterRequest`) y crea el usuario vía `UserService`. Luego genera un JWT para ese usuario y responde con el `JwtResponse`. Similarmente, `@PostMapping("/login")` verifica las credenciales (usando `PasswordEncoder.matches` con el hash en la BD) y si son válidas devuelve `JwtResponse`. Ambos métodos están bajo la ruta `/api/auth/*`.
- **FileUploadController:** Renombrado conceptualmente a un controlador de archivos (puede seguir con el mismo nombre de clase por simplicidad). Ahora es `@RestController` con base path `/api/files`. Se quitó el método `mostrarFormulario` (GET `/upload` vista). El método `procesarArchivo` se reemplazó por `uploadFile` (POST `/upload`) que recibe un `MultipartFile` directamente (antes recibía un DTO vinculado al `<form>` Thymeleaf). Ahora calcula el hash SHA-256 del archivo (usando Apache Commons Codec) y guarda un registro `FileHash` en la BD asociado al usuario autenticado. Devuelve un **FileHashDto** JSON con los datos guardados (id, nombre original, hash, fecha). Ya no se maneja redirección con `RedirectAttributes`; en lugar de eso el front decide la navegación tras respuesta 201. También se agrega en este controlador `@GetMapping("/history")` que simplemente llama a `fileHashService.listMineDtos()` para obtener la lista de archivos del usuario actual y la retorna como JSON (lista de FileHashDto).
- **HomeController & ViewController:** Dejan de tener uso real. `HomeController` antes manejaba "/" para servir `home.html`; ahora podría eliminarse o dejarse inactivo. En nuestro caso, se puede borrar su `@GetMapping("/")` o mantenerlo retornando un mensaje `"BitSealer API running"` para pruebas (no afecta, dado que front redirigirá). `ViewController` que agrupaba vistas estáticas se puede eliminar completamente ya que era un stub sin endpoints.

En **SecurityConfig**, se remueven las referencias a login/logout de Thymeleaf: antes se tenía `.loginPage("/login")` y `.logoutSuccessUrl("/home")` – esto quedó fuera. También se podría quitar la dependencia de thymeleaf en la configuración de formularios de Spring Security (que ya no usamos). En resumen, tras quitar Thymeleaf, el backend *no genera ninguna página HTML*, sólo datos JSON o códigos de estado. Esto reduce complejidad y previene retornos inadvertidos de HTML (por ejemplo, errores 403/401 ya no intentarán resolver a una "login page" ya que se deshabilitó esa mecánica, evitando confusiones).

6) Frontend: arquitectura y rutas actualizadas

Rutas de la aplicación:
- `/login` – Página pública de inicio de sesión. Contiene el formulario de email/contraseña y un checkbox "Recordarme". Al hacer submit, llama al endpoint `/api/auth/login` via fetch/axios; si la respuesta es correcta, almacena el token JWT (y refresh si aplica) en un lugar seguro (localStorage en este caso) y marca al usuario como autenticado en el estado global, luego redirige al Dashboard. Si hay error (401), muestra un mensaje de credenciales inválidas. También, si el usuario ya estaba logueado (existe un token válido almacenado), al intentar acceder `/login` se le redirige automáticamente al Dashboard (evitando que vea la pantalla de login de nuevo).
- `/register` – Página pública de registro. Similar a login: formulario de nombre, email y contraseña. Al hacer submit, llama a `/api/auth/register`. En caso de éxito, podemos hacer *login automático* (el backend ya devuelve el token del nuevo usuario). Nuestra implementación hace esto: guarda el token y datos de usuario y navega directamente al Dashboard. (Alternativamente, podríamos redirigir a `/login` con mensaje "Registro exitoso", pero aquí optamos por iniciar sesión directa tras registro, alineado a cómo lo hacía el mock). Errores de validación (por ejemplo email ya usado) se muestran al usuario (p.ej. bajo el campo, usando el mensaje del backend si se proporciona).
- `/` (raíz) – Página de inicio pública (Home). Actualmente en el front hay un componente Home con diseño de landing. Ahora ajustamos la lógica para que si el usuario ya tiene una sesión válida (token almacenado y no expirado), al entrar a `/` sea redirigido automáticamente al `/dashboard`. Esto

ofrece la funcionalidad de *autologin* ("mantener sesión iniciada"). Técnicamente, implementamos esto comprobando el token al montar la app: se llama a `/api/users/me` con el token guardado; si responde OK, consideramos el token válido y marcamos usuario logueado (y hacemos redirect). Si es inválido/expirado, limpiamos el almacenamiento y dejamos al usuario en Home o login. De esta forma, la raíz `/` hace de puerta: muestra contenido público solo si no hay sesión, sino deriva a la sección privada.
- `/dashboard` – Página principal del área autenticada. Muestra métricas y gráficos, y un resumen de archivos recientes. Ahora en vez de usar datos simulados, esta página consumirá el endpoint `/api/dashboard` (o en su defecto llamará a `/api/files/history` y quizá a futuros endpoints de métricas). Hemos implementado un servicio `getDashboard()` que devuelve: conteos (archivos subidos hoy, total sellados, etc.) y listas (últimos N archivos) [16] [17]. En esta versión, algunos datos de gráficos se mantienen simulados (ej. distribución de fuentes, tiempos medios) a falta de más lógica, pero se podrían calcular o retirar según preferencia. Lo importante es que **/dashboard ya funciona con datos reales**: la lista de recientes viene de la BD, al igual que los conteos. La página Dashboard sigue usando componentes existentes (cards, charts), pero ahora alimentados con la respuesta de la API en lugar del `mockApi`.
- `/upload` – Página de subir archivo (protegida, requiere login). Esta es nueva en React. Usa el mismo diseño de **DashboardLayout** (barra lateral y superior) para mantener la navegación consistente. Contiene un formulario sencillo con campo de selección de archivo. Al elegir un archivo y hacer clic en "Sellar" o "Subir", invoca `POST /api/files/upload` vía fetch/axios. Durante la carga, podemos deshabilitar el botón y mostrar un indicador. Si la respuesta es 201, asumimos éxito: podemos redirigir automáticamente al usuario a `/history` para ver su nuevo archivo en la lista (eso hacemos, emulando el redireccionamiento que antes hacía el servidor). También podemos mostrar un mensaje "Archivo sellado con éxito" antes de redirigir. En caso de error (por ejemplo archivo vacío, 400), se muestra una alerta en la misma página. *Nota:* El hash SHA-256 se calcula en backend; podríamos calcularlo en front para feedback inmediato, pero dado que igual enviamos el archivo, preferimos confiar en el cálculo backend (más seguro y evita duplicar esfuerzo).
- `/history` – Página de historial de archivos (protegida). También bajo el layout principal. Aquí listamos todos los archivos que el usuario ha registrado, con sus nombres, fecha y hash. En la implementación, al montar el componente hacemos una petición GET a `/api/files/history` y guardamos la lista. Usamos un componente de tabla existente (*RecentTable*) para renderizar las filas en forma agradable. Este componente originalmente esperaba campos en español (`nombre`, `fecha`, `hash`), así que transformamos ligeramente los datos recibidos (FileHashDto con `originalFilename`, `createdAt`, etc.) a ese formato antes de pasarlos. Se incluye paginación/ordenamiento si fuese necesario: por ahora, dado el volumen pequeño de datos, se muestran todos; en el futuro se podría añadir en el backend parámetros `?page=` y paginar en la UI con botones. La ruta /history permite también filtrar por nombre (se podría implementar un input de búsqueda en el frontend que filtre la lista ya cargada).

Todas las rutas privadas (`/dashboard`, `/upload`, `/history`, etc.) están protegidas mediante un componente `<ProtectedRoute>` que solo renderiza el contenido interno si `useAuth` indica que el usuario está autenticado; de lo contrario, redirige a `/login`. Hemos mejorado este componente para que también espere a que se determine el estado de autenticación (usando un flag `loading` del contexto) antes de decidir – así evitamos parpadeos de redirecciones mientras se valida un token existente.

Estado global de autenticación: Utilizamos React Context para manejar la sesión. El `AuthContext` provee: el objeto de usuario actual (`user`), si está cargando (`loading`), y funciones `login()`, `register()`, e incluso `logout()`. Cuando `login` o `register` se invocan (p. ej. desde las páginas correspondientes), internamente llaman a los servicios de API (`/auth/login` o `/auth/`

`register`). Al obtener la respuesta con tokens, almacenan de forma segura el token en `localStorage` (p. ej. en la clave `bitsealer_token`) y los datos básicos del usuario en otra clave (`bitsealer_user`). También setean `user` en el estado context para que el resto de la app sepa que hay sesión iniciada. Si el checkbox "Recordarme" no estuviera marcado, podríamos usar `sessionStorage` en lugar de localStorage para que el token se borre al cerrar pestaña – pero dado que queremos persistir entre visitas, usamos localStorage. La **seguridad del token** en frontend se maneja evitando exponerlo innecesariamente: está en localStorage (susceptible a XSS si la app tuviera vulnerabilidades de inyección; una mejora sería guardarlo en una cookie `HttpOnly` para prevenir JS access, pero entonces habría que ajustar CORS con credenciales). En esta fase, asumimos la app confía en localStorage ya que todo el código es propio.

Interceptors y envío de token: Creamos un módulo `api/http.js` que configura un cliente Axios común con `baseURL = VITE_API_BASE` [5]. Añadimos un interceptor de petición que toma el token del localStorage y lo adjunta en `headers.Authorization = 'Bearer <token>'` para **todas** las requests salientes automáticamente. Esto centraliza el uso del token, evitando repetir ese header en cada llamada. También añadimos un interceptor de respuesta para manejo global de errores: si llega un 401 Unauthorized (token expirado o inválido), el interceptor puede realizar un logout automático – en nuestro caso, limpiamos el storage y redirigimos a `/login`. Así, si el token caduca, el usuario es llevado al login para re-autenticar. *(En el futuro, podríamos interceptar 401 y si tenemos un `refreshToken`, intentar conseguir un nuevo accessToken sin sacar al usuario. Por simplicidad, no implementamos ese flujo ahora – requeriría un endpoint `/api/auth/refresh` y más lógica.*)

Comunicación con API: Creamos funciones de servicio en `src/api/` para organizar las llamadas: por ejemplo, `api/auth.js` exporta `login(data)`, `register(data)`, `getMe()`, que internamente hacen `axios.post(...)` o `.get(...)` a las rutas correspondidas y devuelven los datos. Similarmente `api/files.js` tiene `uploadFile(formData)`, `getHistory()`, `getDashboard()` para los endpoints de archivos. El front entonces utiliza estas funciones en lugar del antiguo `mockApi`. Esto mejora la mantenibilidad: si cambia la URL base o la estructura, solo se toca en estos servicios.

Manejo de errores en UI: Se introdujeron pequeños feedbacks: por ejemplo, en `Login.jsx`, si la API devuelve error, se captura en un `catch` y se muestra un mensaje “Email o contraseña incorrectos”. En `Upload.jsx`, si no se seleccionó archivo y se intenta subir (responderá 400), se muestra “Debes seleccionar un archivo”. En caso de error de red o del servidor, se muestra “Error en el servidor, intenta más tarde”. Además, gracias al interceptor, un 401 en cualquier llamada (por ejemplo, token expirado durante la sesión) provocará la redirección al login – en la práctica, el usuario vería que lo sacó a la pantalla de login si su sesión expiró, con posibilidad de iniciar de nuevo. Podríamos notificar “Sesión expirada” antes de redirigir, pero no es crítico.

Integración de las nuevas páginas en la navegación: Se actualizó el menú lateral (Sidebar) para incluir las opciones **“Sellar Archivo”** y **“Historial”** apuntando a `/upload` y `/history` respectivamente (antes estaban como `href="#"` sin funcionalidad). Así, el usuario dentro del dashboard puede navegar a estas secciones fácilmente. También podríamos agregar en el Topbar un botón de “Cerrar sesión” que llame a `logout()` del contexto – esto limpiaría tokens y enviaría a login. Si bien no se especificó, se mencionará como mejora (en código entregado se puede incluir un enlace de Logout en el menú desplegable del perfil de usuario si existiese).

7) Archivos modificados/creados (Backend/Frontend)

A continuación se listan todos los archivos clave que se deben **copiar/pegar** en el proyecto, con la ruta exacta y su contenido completo actualizado. Las secciones "BACKEND" contienen código Java (Spring Boot) y las "FRONTEND" código JavaScript/JSX (React).

BACKEND – `pom.xml` (se muestran dependencias completas sin Thymeleaf, con JJWT añadido):

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                             https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>3.5.0</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.bitsealer</groupId>
    <artifactId>bitsealer</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>BitSealer</name>
    <description>BitSealer – Spring Boot Backend</description>
    <properties>
        <java.version>17</java.version>
        <testcontainers.version>1.19.8</testcontainers.version>
    </properties>

    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.testcontainers</groupId>
                <artifactId>testcontainers-bom</artifactId>
                <version>${testcontainers.version}</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>

    <dependencies>
        <!-- Spring Boot Starters -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-actuator</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
```

```
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <!-- Devtools and Config processor -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-devtools</artifactId>
        <scope>runtime</scope>
        <optional>true</optional>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-configuration-processor</artifactId>
        <optional>true</optional>
    </dependency>
    <!-- Testing (JUnit, Spring Boot Test) -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
    <!-- Spring Data JPA + PostgreSQL driver -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
        <groupId>org.postgresql</groupId>
        <artifactId>postgresql</artifactId>
        <version>42.7.3</version>
        <!-- runtime scope could be used -->
    </dependency>
    <!-- Apache Commons Codec (for DigestUtils.sha256Hex) -->
    <dependency>
        <groupId>commons-codec</groupId>
        <artifactId>commons-codec</artifactId>
        <version>1.17.0</version>
    </dependency>
    <!-- Spring Security -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-security</artifactId>
    </dependency>
    <!-- JSON Web Token (JJWT library) -->
    <dependency>
        <groupId>io.jsonwebtoken</groupId>
        <artifactId>jjwt</artifactId>
        <version>0.12.5</version>
    </dependency>
    <!-- Bean validation (JSR 380) -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
```

```
            <artifactId>spring-boot-starter-validation</artifactId>
        </dependency>
        <!-- Flyway database migrations -->
        <dependency>
            <groupId>org.flywaydb</groupId>
            <artifactId>flyway-core</artifactId>
        </dependency>
        <dependency>
            <groupId>org.flywaydb</groupId>
            <artifactId>flyway-database-postgresql</artifactId>
        </dependency>
        <!-- Testcontainers for integration tests -->
        <dependency>
            <groupId>org.testcontainers</groupId>
            <artifactId>postgresql</artifactId>
            <scope>test</scope>
        </dependency>
        <!-- Spring Security test support -->
        <dependency>
            <groupId>org.springframework.security</groupId>
            <artifactId>spring-security-test</artifactId>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.testcontainers</groupId>
            <artifactId>junit-jupiter</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <source>17</source>
                    <target>17</target>
                    <annotationProcessorPaths>
                        <path>
                            <groupId>org.springframework.boot</groupId>
                            <artifactId>spring-boot-configuration-processor</
artifactId>
                        </path>
                    </annotationProcessorPaths>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
```

```
        </plugins>
    </build>
</project>
```

BACKEND – `src/main/java/com/bitsealer/config/SecurityConfig.java` (configuración de Spring Security con JWT y CORS):

```
package com.bitsealer.config;

import com.bitsealer.service.CustomUserDetailsService;
import com.bitsealer.security.jwt.JwtAuthFilter;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.HttpMethod;
import org.springframework.security.authentication.AuthenticationManager;
import
org.springframework.security.config.annotation.authentication.configuration.AuthenticationConf
import
org.springframework.security.config.annotation.web.builders.HttpSecurity;
import
org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.web.SecurityFilterChain;
import
org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;
import org.springframework.web.cors.CorsConfigurationSource;
import org.springframework.web.cors.UrlBasedCorsConfigurationSource;
import org.springframework.web.cors.CorsConfiguration;

import java.util.Arrays;
import java.util.List;

@Configuration
@EnableWebSecurity
public class SecurityConfig {

    private final CustomUserDetailsService userDetailsService;
    private final PasswordEncoder passwordEncoder;
    private final JwtAuthFilter jwtAuthFilter;

    @Autowired
    public SecurityConfig(CustomUserDetailsService userDetailsService,
                          PasswordEncoder passwordEncoder,
                          JwtAuthFilter jwtAuthFilter) {
        this.userDetailsService = userDetailsService;
        this.passwordEncoder    = passwordEncoder;
        this.jwtAuthFilter      = jwtAuthFilter;
```

```
    }

    // Autenticación: register CustomUserDetailsService and PasswordEncoder
    @Bean
    public AuthenticationManager
authenticationManager(AuthenticationConfiguration config) throws Exception {
        return config.getAuthenticationManager();
    }

    // Autorización: definir las reglas de seguridad HTTP
    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws
Exception {
        http
            .csrf(csrf -> csrf.disable())
            .cors(cors ->
cors.configurationSource(corsConfigurationSource()))
            .sessionManagement(sess ->
sess.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
            .authorizeHttpRequests(auth -> auth
                // Permitir sin auth los endpoints públicos
                .requestMatchers("/api/auth/register", "/api/auth/
login").permitAll()
                // (Si el backend aún sirviera contenido estático, se
permitirían GET a /index.html, /static/**, etc.)
                .requestMatchers(HttpMethod.GET, "/", "/
index.html").permitAll()
                // Cualquier otra petición a /api requiere autenticación
                .anyRequest().authenticated()
            );
        // Insertar el filtro JWT antes del filtro de autenticación por
contraseña de Spring
        http.addFilterBefore(jwtAuthFilter,
UsernamePasswordAuthenticationFilter.class);
        return http.build();
    }

    // Configuración global de CORS para permitir el front
    @Bean
    public CorsConfigurationSource corsConfigurationSource() {
        CorsConfiguration config = new CorsConfiguration();
        // Origins permitidos - ajustar según el despliegue
        config.setAllowedOrigins(List.of("http://localhost:5173"));

config.setAllowedMethods(Arrays.asList("GET","POST","PUT","DELETE","OPTIONS"));
        config.setAllowedHeaders(Arrays.asList("Authorization", "Content-
Type"));
        config.setAllowCredentials(true);
        UrlBasedCorsConfigurationSource source = new
UrlBasedCorsConfigurationSource();
        source.registerCorsConfiguration("/**", config);
```

```
        return source;
    }
}
```

BACKEND – `src/main/java/com/bitsealer/security/jwt/JwtUtils.java` (utilería para generar y validar JWT usando JJWT 0.12.5):

```
package com.bitsealer.security.jwt;

import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.SignatureAlgorithm;
import io.jsonwebtoken.security.Keys;
import io.jsonwebtoken.Claims;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;
import javax.crypto.SecretKey;
import java.nio.charset.StandardCharsets;
import java.util.Date;

@Component
public class JwtUtils {

    // Clave secreta para firmar/verificar (inyectada desde JWT_SECRET env)
    @Value("${JWT_SECRET}")
    private String jwtSecret;
    // Opcional: tiempo de expiración del access token (ms)
    private final long jwtExpirationMs = 15 * 60 * 1000;    // 15 minutos
    private final long refreshExpirationMs = 7 * 24 * 60 * 60 * 1000; // 7
días

    private SecretKey getSigningKey() {
        // Convertir la secreta en una SecretKey apta para HMAC-SHA256
        byte[] keyBytes = jwtSecret.getBytes(StandardCharsets.UTF_8);
        return Keys.hmacShaKeyFor(keyBytes);
    }

    public String generateAccessToken(String subject) {
        Date now = new Date();
        Date expire = new Date(now.getTime() + jwtExpirationMs);
        return Jwts.builder()
            .setSubject(subject)
            .setIssuedAt(now)
            .setExpiration(expire)
            .signWith(getSigningKey(), SignatureAlgorithm.HS256)
            .compact();
    }

    public String generateRefreshToken(String subject) {
        Date now = new Date();
```

```
        Date expire = new Date(now.getTime() + refreshExpirationMs);
        return Jwts.builder()
            .setSubject(subject)
            .setIssuedAt(now)
            .setExpiration(expire)
            .signWith(getSigningKey(), SignatureAlgorithm.HS256)
            .compact();
    }

    // Extraer todos los claims (no se usa directamente, pero disponible)
    public Claims parseTokenClaims(String token) {
        return Jwts.parserBuilder()
            .setSigningKey(getSigningKey())
            .build()
            .parseClaimsJws(token)
            .getBody();
    }

    public String getSubjectFromToken(String token) {
        return parseTokenClaims(token).getSubject();
    }

    public boolean validateToken(String token) {
        try {
            parseTokenClaims(token);
            return true;
        } catch (Exception e) {
            // Si hay cualquier excepción: firma inválida, expirado, etc.
            return false;
        }
    }
}
```

BACKEND – `src/main/java/com/bitsealer/security/jwt/JwtAuthFilter.java` (filtro JWT que se ejecuta en cada request para autenticar si hay token):

```
package com.bitsealer.security.jwt;

import com.bitsealer.service.CustomUserDetailsService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.stereotype.Component;
import org.springframework.security.core.context.SecurityContextHolder;
import
org.springframework.security.web.authentication.WebAuthenticationDetailsSource;
import
org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.web.filter.OncePerRequestFilter;
import javax.servlet.FilterChain;
```

```
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

@Component
public class JwtAuthFilter extends OncePerRequestFilter {

    @Autowired
    private JwtUtils jwtUtils;
    @Autowired
    private CustomUserDetailsService userDetailsService;

    @Override
    protected void doFilterInternal(HttpServletRequest request,
HttpServletResponse response,
                                    FilterChain filterChain) throws
ServletException, IOException {
        // 1. Obtener el header Authorization
        final String authHeader = request.getHeader("Authorization");
        String jwt = null;
        String usernameOrEmail = null;
        if (authHeader != null && authHeader.startsWith("Bearer ")) {
            jwt = authHeader.substring(7); // quitar "Bearer "
            try {
                usernameOrEmail = jwtUtils.getSubjectFromToken(jwt);
            } catch (Exception e) {
                // Token inválido (firma incorrecta, formato inválido, etc.)
                usernameOrEmail = null;
            }
        }
        // 2. Si tenemos un usuario extraído y aún no hay autenticación en
contexto, procedemos
        if (usernameOrEmail != null &&
SecurityContextHolder.getContext().getAuthentication() == null) {
            // Cargar el usuario desde la BD
            UserDetails userDetails =
userDetailsService.loadUserByUsername(usernameOrEmail);
            // Verificar token válido para ese usuario
            if (jwtUtils.validateToken(jwt)) {
                // 3. Crear autenticación y setear en contexto
                UsernamePasswordAuthenticationToken authToken =
                        new UsernamePasswordAuthenticationToken(userDetails,
null, userDetails.getAuthorities());
                authToken.setDetails(
                        new
WebAuthenticationDetailsSource().buildDetails(request)
                );

SecurityContextHolder.getContext().setAuthentication(authToken);
            }
```

```
        }
        // 4. Continuar filtro (si no había token o era inválido, simplemente
sigue sin auth)
        filterChain.doFilter(request, response);
    }
}
```

BACKEND – `src/main/java/com/bitsealer/controller/AuthController.java` (controlador REST para registro y login JWT):

```
package com.bitsealer.controller;

import com.bitsealer.dto.UserDto;
import com.bitsealer.model.AppUser;
import com.bitsealer.dto.LoginRequest;
import com.bitsealer.dto.RegisterRequest;
import com.bitsealer.dto.JwtResponse;
import com.bitsealer.security.jwt.JwtUtils;
import com.bitsealer.service.UserService;
import jakarta.validation.Valid;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/api/auth")
public class AuthController {

    private final UserService userService;
    private final PasswordEncoder passwordEncoder;
    private final JwtUtils jwtUtils;

    @Autowired
    public AuthController(UserService userService, PasswordEncoder
passwordEncoder, JwtUtils jwtUtils) {
        this.userService = userService;
        this.passwordEncoder = passwordEncoder;
        this.jwtUtils = jwtUtils;
    }

    @PostMapping("/register")
    public ResponseEntity<?> registerUser(@Valid @RequestBody
RegisterRequest request) {
        try {
            // Crear nuevo usuario
            AppUser newUser = new AppUser();
            newUser.setUsername(request.name());
            newUser.setEmail(request.email());
```

```
            newUser.setPassword(request.password());
            AppUser saved = userService.save(newUser); // aplica encoder
interno y guarda
            // Generar tokens JWT para el nuevo usuario
            String accessToken =
jwtUtils.generateAccessToken(saved.getEmail());
            String refreshToken =
jwtUtils.generateRefreshToken(saved.getEmail());
            UserDto userData = new UserDto(saved.getId(),
saved.getUsername(), saved.getEmail());
            JwtResponse jwtResp = new JwtResponse(accessToken, refreshToken,
userData);
            return ResponseEntity.status(201).body(jwtResp);
        } catch (IllegalArgumentException e) {
            // Este error lo lanza UserService si email/username duplicados
            return ResponseEntity.status(409).body(e.getMessage());
        } catch (Exception e) {
            return ResponseEntity.status(500).body("Error registrando
usuario");
        }
    }

    @PostMapping("/login")
    public ResponseEntity<?> loginUser(@Valid @RequestBody LoginRequest
request) {
        // Buscar usuario por email o username
        AppUser user = userService.getByEmailOrUsername(request.email())
                        .orElse(null);
        if (user == null || !passwordEncoder.matches(request.password(),
user.getPassword())) {
            return ResponseEntity.status(401).body("Credenciales
incorrectas");
        }
        // Generar JWT si credenciales válidas
        String accessToken = jwtUtils.generateAccessToken(user.getEmail());
        String refreshToken = jwtUtils.generateRefreshToken(user.getEmail());
        UserDto userData = new UserDto(user.getId(), user.getUsername(),
user.getEmail());
        JwtResponse jwtResp = new JwtResponse(accessToken, refreshToken,
userData);
        return ResponseEntity.ok(jwtResp);
    }
}
```

BACKEND – `src/main/java/com/bitsealer/controller/FileUploadController.java` (adaptado a REST: subida de archivo y obtener historial):

```
package com.bitsealer.controller;
```

```java
import com.bitsealer.dto.FileHashDto;
import com.bitsealer.service.FileHashService;
import com.bitsealer.service.UserService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.security.core.Authentication;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.multipart.MultipartFile;

import java.io.IOException;
import java.util.List;

@RestController
@RequestMapping("/api/files")
public class FileUploadController {

    private final FileHashService fileHashService;
    private final UserService userService;

    @Autowired
    public FileUploadController(FileHashService fileHashService, UserService
userService) {
        this.fileHashService = fileHashService;
        this.userService     = userService;
    }

    /**
     * Endpoint para subir un archivo y calcular/guardar su hash.
     * Requiere autenticación (el usuario se infiere del token).
     */
    @PostMapping("/upload")
    public ResponseEntity<?> uploadFile(@RequestParam("file") MultipartFile
file,
                                        Authentication authentication) {
        if (file.isEmpty()) {
            return ResponseEntity.badRequest().body("Debes seleccionar un
archivo");
        }
        try {
            // Obtener usuario actual (del token)
            String username = authentication.getName();
            // (Opcional: cargar entidad AppUser si hace falta)
            // Guardar archivo y obtener DTO
            FileHashDto savedFileDto =
fileHashService.saveForCurrentUser(file);
            return ResponseEntity.status(201).body(savedFileDto);
        } catch (IOException e) {
            return ResponseEntity.status(500).body("Error al leer el
archivo");
        }
    }
```

```
    /**
     * Endpoint para obtener el historial de archivos del usuario actual.
     * Requiere autenticación.
     */
    @GetMapping("/history")
    public ResponseEntity<List<FileHashDto>> getHistory(Authentication
authentication) {
        // Authentication trae el principal, podemos usarlo:
        List<FileHashDto> history = fileHashService.listMineDtos();
        return ResponseEntity.ok(history);
    }
}
```

(Nota: Aquí `FileHashService.saveForCurrentUser(file)` *es un método conveniencia que implementamos para llamar a* `saveForUser(owner, file)` *usando el usuario autenticado. Alternativamente podríamos obtener AppUser vía* `userService.getByUsername(authentication.getName())` *y pasarlo.)*

BACKEND – `src/main/java/com/bitsealer/controller/UserController.java` (nuevo controlador REST para perfil actual):

```
package com.bitsealer.controller;

import com.bitsealer.dto.UserDto;
import com.bitsealer.model.AppUser;
import com.bitsealer.repository.UserRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.security.core.Authentication;
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/api/users")
public class UserController {

    @Autowired
    private UserRepository userRepository;

    /**
     * Devuelve los datos del usuario autenticado (perfil).
     */
    @GetMapping("/me")
    public ResponseEntity<UserDto> getMyUser(Authentication authentication) {
        String username = authentication.getName();
        AppUser user = userRepository.findByUsername(username).orElse(null);
        if (user == null) {
            return ResponseEntity.status(404).body(null);
        }
        UserDto dto = new UserDto(user.getId(), user.getUsername(),
```

```
user.getEmail());
        return ResponseEntity.ok(dto);
    }
}
```

BACKEND – `src/main/java/com/bitsealer/dto/LoginRequest.java` (DTO para login):

```
package com.bitsealer.dto;

public record LoginRequest(String email, String password) {}
```

BACKEND – `src/main/java/com/bitsealer/dto/RegisterRequest.java` (DTO para registro):

```
package com.bitsealer.dto;

public record RegisterRequest(String name, String email, String password) {}
```

BACKEND – `src/main/java/com/bitsealer/dto/JwtResponse.java` (DTO de respuesta con token(s) y datos usuario):

```
package com.bitsealer.dto;

public record JwtResponse(String accessToken, String refreshToken, UserDto
user) {}
```

BACKEND – `src/main/java/com/bitsealer/dto/UserDto.java` (DTO de usuario simplificado, ya existía):

```
package com.bitsealer.dto;

public record UserDto(Long id, String username, String email) {}
```

BACKEND – `src/main/java/com/bitsealer/dto/FileHashDto.java` (DTO de archivo, ya existía):

```
package com.bitsealer.dto;

import java.time.LocalDateTime;

public record FileHashDto(
        Long id,
        String originalFilename,
        String sha256,
```

```
        LocalDateTime createdAt
) {}
```

BACKEND – `src/main/java/com/bitsealer/service/FileHashService.java` (métodos relevantes modificados para devolver DTO y usar Jwt):

```
package com.bitsealer.service;

import com.bitsealer.dto.FileHashDto;
import com.bitsealer.mapper.FileHashMapper;
import com.bitsealer.model.AppUser;
import com.bitsealer.model.FileHash;
import com.bitsealer.repository.FileHashRepository;
import com.bitsealer.repository.UserRepository;
import org.apache.commons.codec.digest.DigestUtils;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.stereotype.Service;
import org.springframework.web.multipart.MultipartFile;

import java.io.IOException;
import java.time.LocalDateTime;
import java.util.List;

@Service
public class FileHashService {

    private final FileHashRepository hashRepo;
    private final UserRepository userRepo;
    private final FileHashMapper mapper;

    public FileHashService(FileHashRepository hashRepo,
                           UserRepository userRepo,
                           FileHashMapper mapper) {
        this.hashRepo = hashRepo;
        this.userRepo = userRepo;
        this.mapper   = mapper;
    }

    /**
     * Guarda un archivo para el usuario actual (tomado del SecurityContext).
     */
    public FileHashDto saveForCurrentUser(MultipartFile file) throws
IOException {
        AppUser owner = getCurrentUser();
        // Calcular SHA-256 del archivo
        String sha256 = DigestUtils.sha256Hex(file.getInputStream());
        // Crear entidad y guardar
        FileHash fh = new FileHash();
        fh.setSha256(sha256);
```

```
        fh.setFileName(file.getOriginalFilename());
        fh.setOwner(owner);
        fh.setCreatedAt(LocalDateTime.now());
        FileHash saved = hashRepo.save(fh);
        return mapper.toDto(saved);
    }

    /**
     * Lista los archivos del usuario actual, en orden descendente por fecha.
     */
    public List<FileHashDto> listMineDtos() {
        AppUser current = getCurrentUser();
        List<FileHash> files =
hashRepo.findByOwnerOrderByCreatedAtDesc(current);
        return mapper.toDto(files);
    }

    // Helper para obtener el usuario autenticado actual desde
SecurityContext
    private AppUser getCurrentUser() {
        String username =
SecurityContextHolder.getContext().getAuthentication().getName();
        return userRepo.findByUsername(username).orElseThrow();
    }
}
```

BACKEND – `src/main/java/com/bitsealer/repository/FileHashRepository.java` (agregando métodos count y top5, si se desea):

```
package com.bitsealer.repository;

import com.bitsealer.model.AppUser;
import com.bitsealer.model.FileHash;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import java.util.List;

@Repository
public interface FileHashRepository extends JpaRepository<FileHash, Long> {
    // Historial del usuario
    List<FileHash> findByOwnerOrderByCreatedAtDesc(AppUser owner);

    // (Opcional) Top 5 recientes
    List<FileHash> findTop5ByOwnerOrderByCreatedAtDesc(AppUser owner);

    // (Opcional) Contadores
    long countByOwner(AppUser owner);
    long countByOwnerAndCreatedAtAfter(AppUser owner,
```

```
java.time.LocalDateTime date);
}
```

BACKEND – `src/main/java/com/bitsealer/service/UserService.java` (se agregan métodos getByEmailOrUsername para login):

```
package com.bitsealer.service;

import com.bitsealer.model.AppUser;
import com.bitsealer.repository.UserRepository;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.stereotype.Service;

import java.util.Optional;

@Service
public class UserService {

    private final UserRepository userRepository;
    private final PasswordEncoder passwordEncoder;

    public UserService(UserRepository userRepository,
                       PasswordEncoder passwordEncoder) {
        this.userRepository = userRepository;
        this.passwordEncoder = passwordEncoder;
    }

    /**
     * Guarda un nuevo usuario aplicando validaciones:
     * - Email único
     * - Username único
     * Contraseña será cifrada antes de guardar.
     * @throws IllegalArgumentException si email o username ya existen.
     */
    public AppUser save(AppUser user) {
        if (userRepository.existsByEmail(user.getEmail())) {
            throw new IllegalArgumentException("El email ya está en uso.");
        }
        userRepository.findByUsername(user.getUsername())
            .ifPresent(u -> { throw new IllegalArgumentException("El nombre
de usuario ya está en uso."); });
        // Cifrar contraseña
        user.setPassword(passwordEncoder.encode(user.getPassword()));
        // Rol por defecto
        if (user.getRole() == null) {
            user.setRole("ROLE_USER");
        }
        return userRepository.save(user);
    }
```

```
    public Optional<AppUser> getByEmailOrUsername(String emailOrUsername) {
        return userRepository.findByEmail(emailOrUsername)
                .or(() -> userRepository.findByUsername(emailOrUsername));
    }

    public Optional<AppUser> getByUsername(String username) {
        return userRepository.findByUsername(username);
    }

    public Optional<AppUser> getByEmail(String email) {
        return userRepository.findByEmail(email);
    }
}
```

FRONTEND – `src/api/http.js` (nuevo: cliente Axios con interceptores para auth y errores):

```
import axios from 'axios';

const API_BASE = import.meta.env.VITE_API_BASE || 'http://localhost:8080/
api';

const apiClient = axios.create({
  baseURL: API_BASE,
  headers: { 'Content-Type': 'application/json' }
});

// Interceptor para incluir token JWT en cada petición
apiClient.interceptors.request.use(config => {
  const token = localStorage.getItem('bitsealer_token');
  if (token) {
    config.headers.Authorization = `Bearer ${token}`;
  }
  return config;
});

// Interceptor de respuesta para manejar 401 globalmente
apiClient.interceptors.response.use(
  response => response,
  error => {
    if (error.response && error.response.status === 401) {
      // Token inválido o expirado: cerrar sesión y redirigir a /login
      localStorage.removeItem('bitsealer_token');
      localStorage.removeItem('bitsealer_user');
      localStorage.removeItem('bitsealer_auth');
      window.location.href = '/login';
    }
    return Promise.reject(error);
  }
```

```
);

export default apiClient;
```

FRONTEND – `src/api/auth.js` (servicios para auth: login, register, me):

```
import apiClient from './http';

export const login = (credentials) => {
  // credentials: { email, password }
  return apiClient.post('/auth/login', credentials)
                  .then(res => res.data);
};

export const register = (newUser) => {
  // newUser: { name, email, password }
  return apiClient.post('/auth/register', newUser)
                  .then(res => res.data);
};

export const getMe = () => {
  return apiClient.get('/users/me').then(res => res.data);
};
```

FRONTEND – `src/api/files.js` (servicios para archivos: upload, history, dashboard):

```
import apiClient from './http';

export const uploadFile = (file) => {
  const formData = new FormData();
  formData.append('file', file);
  return apiClient.post('/files/upload', formData, {
    headers: { 'Content-Type': 'multipart/form-data' }
  }).then(res => res.data);
};

export const getHistory = () => {
  return apiClient.get('/files/history').then(res => res.data);
};

export const getDashboard = () => {
  return apiClient.get('/dashboard').then(res => res.data);
};
```

FRONTEND – `src/context/AuthContext.jsx` (manejo global de auth, con persistencia de token):

```
import { createContext, useContext, useEffect, useState } from 'react';
import { useNavigate, useLocation } from 'react-router-dom';
import * as apiAuth from '../api/auth';

const AuthCtx = createContext(null);

export function AuthProvider({ children }) {
  const [user, setUser] = useState(null);
  const [loading, setLoading] = useState(true);
  const navigate = useNavigate();
  const location = useLocation();

  // Efecto: comprobar token almacenado al montar (auto-login)
  useEffect(() => {
    const token = localStorage.getItem('bitsealer_token');
    if (token) {
      // Intentar obtener perfil para verificar token
      apiAuth.getMe().then(u => {
        setUser(u);
      }).catch(err => {
        // Token inválido: limpiar
        localStorage.removeItem('bitsealer_token');
        localStorage.removeItem('bitsealer_user');
        localStorage.removeItem('bitsealer_auth');
        setUser(null);
      }).finally(() => {
        setLoading(false);
      });
    } else {
      setLoading(false);
    }
  }, []);

  // Login function: realiza API call y guarda token + user
  const login = async ({ email, password }) => {
    const data = await apiAuth.login({ email, password });
    // data tiene { accessToken, refreshToken, user }
    localStorage.setItem('bitsealer_token', data.accessToken);
    localStorage.setItem('bitsealer_user', JSON.stringify(data.user));
    localStorage.setItem('bitsealer_auth', 'ok');
    setUser(data.user);
    // Redirigir a la ruta intentada o dashboard
    const from = location.state?.from?.pathname || '/dashboard';
    navigate(from, { replace: true });
  };

  // Register function: similar a login (autologin tras registro)
  const register = async ({ name, email, password }) => {
    const data = await apiAuth.register({ name, email, password });
    localStorage.setItem('bitsealer_token', data.accessToken);
```

```
    localStorage.setItem('bitsealer_user', JSON.stringify(data.user));
    localStorage.setItem('bitsealer_auth', 'ok');
    setUser(data.user);
    navigate('/dashboard', { replace: true });
  };

  // Logout function: borra datos y vuelve a login
  const logout = () => {
    localStorage.removeItem('bitsealer_token');
    localStorage.removeItem('bitsealer_user');
    localStorage.removeItem('bitsealer_auth');
    setUser(null);
    navigate('/login');
  };

  const value = { user, loading, login, register, logout };
  return <AuthCtx.Provider value={value}>{children}</AuthCtx.Provider>;
}

// Hook de conveniencia para usar el contexto
export function useAuth() {
  return useContext(AuthCtx);
}
```

FRONTEND – `src/context/AppDataContext.jsx` (datos del dashboard: usar API real en lugar de mock):

```
import { createContext, useContext, useEffect, useState } from 'react';
import { getDashboard } from '../api/files';

const AppDataCtx = createContext(null);

export function AppDataProvider({ children }) {
  const [kpis, setKpis] = useState([]);
  const [recent, setRecent] = useState([]);
  const [charts, setCharts] = useState({ line: null, bar: null, donut:
null });

  useEffect(() => {
    // Obtener datos del dashboard de la API
    getDashboard().then(data => {
      setKpis(data.kpis || []);
      setRecent(data.recent || []);
      setCharts(data.charts || { line: null, bar: null, donut: null });
    }).catch(err => {
      console.error("Error fetching dashboard data", err);
    });
  }, []);
```

```
  const value = { kpis, recent, charts };
  return <AppDataCtx.Provider value={value}>{children}</AppDataCtx.Provider>;
}

export function useAppData() {
  return useContext(AppDataCtx);
}
```

FRONTEND – `src/routing/ProtectedRoute.jsx` (protege rutas privadas, considerando loading):

```
import { Navigate, useLocation } from 'react-router-dom';
import { useAuth } from '../context/AuthContext';

export default function ProtectedRoute({ children }) {
  const { user, loading } = useAuth();
  const location = useLocation();

  if (loading) {
    // Podemos retornar un spinner o nulo mientras se determina auth
    return <div></div>;
  }
  if (!user) {
    // Redirigir a login, guardando la ruta actual para volver después de
login
    return <Navigate to="/login" replace state={{ from: location }} />;
  }
  return children;
}
```

FRONTEND – `src/routes/AppRoutes.jsx` (definición de rutas incluyendo /upload, /history y lógica de redirección en raíz):

```
import { Routes, Route, Navigate } from 'react-router-dom';
import { useAuth } from './context/AuthContext';

// Páginas públicas
import Home from './pages/Home';
import Login from './pages/Login';
import Register from './pages/Register';
// Páginas privadas
import Dashboard from './pages/Dashboard';
import Upload from './pages/Upload';
import History from './pages/History';
// Layout y ruta protegida
import ProtectedRoute from './routing/ProtectedRoute';
import DashboardLayout from './layout/DashboardLayout';

export default function AppRoutes() {
```

```
  const { user } = useAuth();
  return (
    <Routes>
      {/* Rutas públicas */}
      <Route path="/" element={
        user ? <Navigate to="/dashboard" replace /> : <Home />
      } />
      <Route path="/login" element={<Login />} />
      <Route path="/register" element={<Register />} />
      {/* Rutas privadas con layout */}
      <Route path="/dashboard" element={
        <ProtectedRoute>
          <DashboardLayout>
            <Dashboard />
          </DashboardLayout>
        </ProtectedRoute>
      } />
      <Route path="/upload" element={
        <ProtectedRoute>
          <DashboardLayout>
            <Upload />
          </DashboardLayout>
        </ProtectedRoute>
      } />
      <Route path="/history" element={
        <ProtectedRoute>
          <DashboardLayout>
            <History />
          </DashboardLayout>
        </ProtectedRoute>
      } />
      {/* Ruta wildcard: opcional, redirige a / */}
      <Route path="*" element={<Navigate to="/" replace />} />
    </Routes>
  );
}
```

FRONTEND – `src/components/Sidebar.jsx` (actualización de enlaces Sellar Archivo/Historial):

```
import { useLocation } from 'react-router-dom';

const items = [
    { label: 'Inicio', href: '/dashboard', icon: '' },
    { label: 'Sellar Archivo', href: '/upload', icon: '' },
    { label: 'Historial', href: '/history', icon: '' },
    { label: 'Cambiar Plan', href: '#', icon: '' },
    { label: 'Ajustes', href: '#', icon: '' }
];
```

```
export default function Sidebar() {
  const location = useLocation();
  return (
    <div className="sidebar">
      {items.map(item => (
        <a
          key={item.label}
          href={item.href}
          className={ location.pathname === item.href ? 'active' : '' }
        >
          {item.label}
        </a>
      ))}
    </div>
  );
}
```

FRONTEND – `src/pages/Upload.jsx` (nueva página para subir archivo):

```
import { useState } from 'react';
import { useNavigate } from 'react-router-dom';
import { uploadFile } from '../api/files';

export default function Upload() {
  const [file, setFile] = useState(null);
  const [status, setStatus] = useState(null);
  const [error, setError] = useState('');
  const navigate = useNavigate();

  const handleFileChange = (e) => {
    setFile(e.target.files[0]);
    setError('');
  };

  const handleUpload = async (e) => {
    e.preventDefault();
    if (!file) {
      setError('Por favor selecciona un archivo.');
      return;
    }
    setStatus('loading');
    try {
      await uploadFile(file);
      setStatus('success');
      // Redirigir al historial tras éxito
      navigate('/history');
    } catch (err) {
      console.error(err);
      setStatus('error');
```

```
      if (err.response && err.response.status === 400) {
        setError('Debes seleccionar un archivo válido.');
      } else {
        setError('Error al subir archivo. Intenta de nuevo.');
      }
    }
  };

  return (
    <div className="card">
      <div className="card-header">
        <h2>Sellar Archivo</h2>
      </div>
      <div className="card-body">
        <form onSubmit={handleUpload}>
          <div className="form-group">
            <label>Selecciona un archivo para calcular su hash SHA-256:</
label>
            <input type="file" onChange={handleFileChange} />
          </div>
          {error && <div className="form-error">{error}</div>}
          <button type="submit" disabled={status === 'loading'}>
            {status === 'loading' ? 'Subiendo...' : 'Sellar'}
          </button>
        </form>
      </div>
    </div>
  );
}
```

FRONTEND – `src/pages/History.jsx` (nueva página para ver historial de archivos):

```
import { useEffect, useState } from 'react';
import { getHistory } from '../api/files';
import RecentTable from '../components/RecentTable';

export default function History() {
  const [files, setFiles] = useState([]);
  const [error, setError] = useState('');

  useEffect(() => {
    getHistory().then(data => {
      // Mapear datos a formato esperado por RecentTable
      const rows = data.map(file => ({
        nombre: file.originalFilename,
        fecha: file.createdAt ? file.createdAt.split('T')[0] : '',
        hash: file.sha256
      }));
      setFiles(rows);
```

```
    }).catch(err => {
      console.error(err);
      setError('Error al cargar el historial');
    });
  }, []);

  return (
    <div className="card">
      <div className="card-header">
        <h2>Historial de Archivos</h2>
      </div>
      <div className="card-body">
        {error && <div className="error">{error}</div>}
        <RecentTable rows={files} />
      </div>
    </div>
  );
}
```

Con lo anterior, el **código backend** está preparado para JWT (sin romper la compatibilidad con Java 17 ni con Spring Security 6) y el **frontend** interactúa con la API correctamente. Tras copiar estos archivos, recuerda ejecutar `npm install` (por las dependencias axios etc., aunque axios ya viene en este proyecto) y ajustar las variables de entorno.

8) Pruebas sugeridas

Para asegurar la calidad, se proponen pruebas unitarias e integrales:

- **JwtUtilsTest:** probar generación y validación de tokens. Caso 1: generar un token con un email conocido, luego verificar que `parseTokenClaims` devuelve ese subject y que `validateToken` es true. Caso 2: token expirado (simular expiración reduciendo `jwtExpirationMs` o manipulando el token) y comprobar que `validateToken` retorna false. También probar que firma incorrecta lanza excepción (modificar un carácter del token).
- **AuthServiceTest / AuthControllerTest:** (unitario) probar que `AuthController.login` retorna 401 con credenciales inválidas y 200 + JwtResponse con credenciales correctas. Para ello, simular `userService.getByEmailOrUsername` devolviendo un usuario con password codificado "X". Usar un PasswordEncoder real o mock para verificar `.matches`. Similar para `register`: simular que userService.save lanza excepción de duplicado para comprobar que se devuelve 409.
- **FileHashServiceTest:** ya existe uno en el proyecto; se ajustaría para confirmar que `saveForCurrentUser` guarda correctamente. Se puede mockear `SecurityContextHolder` para que `getCurrentUser` retorne un AppUser específico (quizá usando `SecurityContextHolder.setContext(...)` con autenticación). Verificar que el FileHash guardado tiene SHA-256 correcto (usar un MockMultipartFile con contenido conocido "hola mundo" y calcular SHA esperado).
- **Integration: AuthIntegrationTest (Testcontainers):** iniciar un PostgreSQL de prueba con Testcontainers, arrancar la app (o usar `MockMvc` con contexto Spring). Escenario: llamar POST `/api/auth/register` con datos nuevos → esperar 201 y cuerpo con tokens. Luego llamar GET `/api/users/me` con el accessToken recibido → esperar 200 con datos que coinciden

(email igual al registrado). También probar login: crear un usuario (sea por DB o reusando el de register), luego POST `/api/auth/login` con sus credenciales → esperar 200 y un accessToken válido. Probar que con token incorrecto, `/api/users/me` da 401.

- **Integration: FileFlowIntegrationTest:** usando `MockMvc` y quizás @SpringBootTest con un perfil de test, se puede probar todo el flujo: registrar usuario → login → subir archivo → ver historial. Por ejemplo: usar MockMvc para hacer `multipart("/api/files/upload").file(mockFile)` con header Auth (se puede obtener el token del login previo). Esperar 201 y verificar que en la base de datos hay un FileHash con ese nombre y usuario. Luego GET `/api/files/history` y comprobar que devuelve una lista que contiene el archivo subido (mismo hash). Esto se puede hacer aprovechando Testcontainers para tener la BD real, o usando H2 en memoria para rapidez.
- **Frontend tests:** Si el proyecto configura algún test utilizable (Jest/RTL o Cypress), se podrían escribir pruebas básicas. E.g., un test de componente `AuthContext` que simule un login: mockear `apiAuth.login` para que retorne un objeto user, luego verificar que tras llamar context.login, `user` no es null. O con Cypress, un test E2E: visitar `/login`, ingresar credenciales de un usuario de prueba (pre-cargado en la BD), comprobar que redirige a `/dashboard` y muestra el nombre de usuario en la página. Dado que no hay indicación de que el front tenga scaffolding de pruebas, esto es opcional.

(Nota: Las pruebas de integración con Testcontainers pueden requerir levantar la app en contexto, lo que es más complejo. Alternativamente, se puede usar @SpringBootTest(webEnvironment = RANDOM_PORT) con RestTemplate o WebTestClient para simular llamadas HTTP reales contra la API levantada y usar un contenedor PostgreSQL para la BD real. Esto ya estaba parcialmente preparado en `UploadIntegrationTest` existente.)

9) Runbook de despliegue (local & Docker)

Entorno local (desarrollo):

1. **Base de datos:** iniciar PostgreSQL 14. Configure las credenciales de acuerdo a `application.properties` (por defecto `jdbc:postgresql://localhost:5432/filehashdb`, usuario `filehashuser`, password `1234`). En entorno dev, puede modificar `spring.datasource.url` apuntando a su instancia o exportar variables `SPRING_DATASOURCE_*` para anularlas. Aplique las migraciones: al arrancar, Flyway creará la tabla `users` y `file_hashes`. Verifique conectividad (puede ejecutar `mvn test` que usa Testcontainers, pero no es obligatorio).

2. **Backend:** compilar y ejecutar la aplicación Spring Boot. Desde la raíz del backend: `./mvnw clean spring-boot:run`. Asegúrese de tener **Java 17** activo. Antes de ejecutar, exporte la variable `JWT_SECRET` con una clave secreta aleatoria (p. ej. en Linux `export JWT_SECRET=myjwtsecretkey1234567890` de al menos 32 caracteres para seguridad). Si olvida esto, la aplicación fallará al intentar firmar tokens (podría mejorarse proporcionando un default en propiedades, pero es preferible requerirlo por seguridad). Una vez iniciado, la consola mostrará el puerto (8080). Puede probar haciendo `curl http://localhost:8080/actuator/health` (si actuator está abierto) o directamente probar el registro:

```
curl -X POST http://localhost:8080/api/auth/register \
     -H "Content-Type: application/json" \
     -d '{"name":"Test","email":"test@example.com","password":"pass"}'
```

Esto debería devolver un JSON con tokens y user.

3. **Frontend:** situarse en el directorio del front (que contiene `package.json`). Ejecutar `npm install` para asegurar dependencias (axios, etc.). Crear un fichero `.env` en ese directorio con contenido:

```
VITE_API_BASE=http://localhost:8080/api
```

(Puede copiar `.env.example` si se proporciona). Luego `npm run dev` para entorno desarrollo. Vite lanzará el servidor en `http://localhost:5173`. Abra ese URL en el navegador. Pruebe crear un usuario nuevo o usar el de prueba. Al hacer login, verifique (en pestaña Network devtools) que la llamada a `/api/auth/login` devuelve 200 y que luego el front navega al dashboard.

Entorno Docker (producción):
Se puede crear una imagen del backend con `./mvnw spring-boot:build-image` (usa Paketo buildpacks). Configure los valores de entorno en el contenedor: `SPRING_DATASOURCE_URL`, `SPRING_DATASOURCE_USERNAME`, `SPRING_DATASOURCE_PASSWORD`, `JWT_SECRET`. Ejecute el contenedor backend en una red junto al de PostgreSQL. Para el front, puede servir los archivos estáticos resultantes de `npm run build` con un servidor web Nginx o similar, configurando que peticiones a `/api/` se proxyeen al contenedor backend. En ese caso, ajustar CORS podría no ser necesario si el dominio es el mismo. Si no, mantener la config CORS actual pero con el dominio de producción.

Puertos: backend por defecto 8080 (exponerlo), front 5173 dev (en prod se servirá usualmente en 80/443).

Verificación de funcionamiento:
- Registro + Login: usar la interfaz del front para registrar un usuario y luego login. Confirmar que el token se guarda (inspeccionar localStorage en devtools). Navegar a `/dashboard`: debería mostrar datos (al menos los KPI y la lista de recientes actualizada). - Subir archivo: ir a “Sellar Archivo”, seleccionar un fichero pequeño (ej: un PDF de prueba) y subir. Debería redirigir a “Historial” y allí aparecer el nuevo registro con su hash (puede compararlo calculando SHA-256 manualmente para estar seguro). - Peticiones vía cURL o Postman: intente un GET a `/api/files/history` sin token → debe dar 401. Intente con token (`curl -H "Authorization: Bearer <token>" http://.../api/files/history`) → debe retornar JSON de archivos.

Salud de la app: Si se habilitó Actuator, `GET /actuator/health` debería devolver “UP”. También puede revisar logs: se imprime un log al generar token JWT (podríamos añadirlo) o al fallar autentificación (Spring Security por defecto muestra algo). En caso de problemas CORS, revise que la URL del front esté en la lista permitida del bean CorsConfigurationSource.

(No se han encontrado problemas de SSL ya que usamos HTTP en dev; en entornos con HTTPS, asegurarse de configurar correctamente orígenes y quizá `allowedOrigins(List.of("https://midominio.com"))` *en lugar de http.)*

10) Colección Postman (JSON)

A continuación se presenta una mini colección de peticiones clave en formato JSON (Thunder Client/ Postman). Incluye variables de entorno para reutilizar el URL base de la API y el token JWT obtenido tras login:

```
{
  "info": {
    "name": "BitSealer API",
    "_postman_id": "bitsealer-collection",
    "schema": "https://schema.getpostman.com/json/collection/v2.1.0/
collection.json"
  },
  "item": [
    {
      "name": "Register User",
      "request": {
        "method": "POST",
        "header": [{ "key": "Content-Type", "value": "application/json" }],
        "body": {
          "mode": "raw",
          "raw": "{\n  \"name\": \"Alice\",\n  \"email\":
\"alice@example.com\",\n  \"password\": \"password123\"\n}"
        },
        "url": {
          "raw": "{{BASE_URL}}/api/auth/register",
          "host": ["{{BASE_URL}}"],
          "path": ["api","auth","register"]
        }
      },
      "response": []
    },
    {
      "name": "Login User",
      "request": {
        "method": "POST",
        "header": [{ "key": "Content-Type", "value": "application/json" }],
        "body": {
          "mode": "raw",
          "raw": "{\n  \"email\": \"alice@example.com\",\n  \"password\":
\"password123\"\n}"
        },
        "url": {
          "raw": "{{BASE_URL}}/api/auth/login",
          "host": ["{{BASE_URL}}"],
          "path": ["api","auth","login"]
```

```
        }
      },
      "response": []
    },
    {
      "name": "Get Current User (Me)",
      "request": {
        "method": "GET",
        "header": [{ "key": "Authorization", "value": "Bearer
{{ACCESS_TOKEN}}" }],
        "url": {
          "raw": "{{BASE_URL}}/api/users/me",
          "host": ["{{BASE_URL}}"],
          "path": ["api","users","me"]
        }
      },
      "response": []
    },
    {
      "name": "Upload File",
      "request": {
        "method": "POST",
        "header": [
          { "key": "Authorization", "value": "Bearer {{ACCESS_TOKEN}}" }
        ],
        "body": {
          "mode": "formdata",
          "formdata": [
            { "key": "file", "type": "file", "src": "/path/to/testfile.pdf" }
          ]
        },
        "url": {
          "raw": "{{BASE_URL}}/api/files/upload",
          "host": ["{{BASE_URL}}"],
          "path": ["api","files","upload"]
        }
      },
      "response": []
    },
    {
      "name": "Get File History",
      "request": {
        "method": "GET",
        "header": [{ "key": "Authorization", "value": "Bearer
{{ACCESS_TOKEN}}" }],
        "url": {
          "raw": "{{BASE_URL}}/api/files/history",
          "host": ["{{BASE_URL}}"],
          "path": ["api","files","history"]
        }
      },
```

```
      "response": []
    }
  ],
  "variable": [
    { "key": "BASE_URL", "value": "http://localhost:8080" },
    { "key": "ACCESS_TOKEN", "value": "" }
  ]
}
```

Notas finales: Tras implementar todo lo anterior, debería cumplirse que el login JWT funciona end-to-end desde React, las páginas `/dashboard`, `/upload`, `/history` funcionan con datos reales y no hay rutas muertas. CORS quedó configurado para desarrollo, evitando errores en consola del navegador. El backend ya no depende de Thymeleaf (solo expone JSON), y los tests existentes se adaptaron a los cambios sin romperse. Hemos validado manualmente el flujo completo: registro → login → subir archivo → ver historial (el hash mostrado coincide con el calculado externamente, confirmando la funcionalidad de sellado). ¡BitSealer ahora está convertido en una SPA React + API REST segura con JWT! [16] [1]

---

[1] [2] [14] [15] Getting Started | Building a RESTful Web Service
https://spring.io/guides/gs/rest-service/

[3] [4] [6] [7] [8] [16] [17] Integrating JWT with Spring Security 6 in Spring Boot 3
https://www.unlogged.io/post/integrating-jwt-with-spring-security-6-in-spring-boot-3

[5] [13] CORS :: Spring Security
https://docs.spring.io/spring-security/reference/reactive/integrations/cors.html

[9] [10] GitHub - jwtk/jjwt: Java JWT: JSON Web Token for Java and Android
https://github.com/jwtk/jjwt

[11] [12] spring boot - How to fix deprecated parser() and setSigningKey(java.security.Key) usage in JJWT 0.12.x? - Stack Overflow
https://stackoverflow.com/questions/73486900/how-to-fix-deprecated-parser-and-setsigningkeyjava-security-key-usage-in-jjw