Memoria de Proyecto Fin de CFGS

Desarrollo de Aplicaciones Web

Marcos Alonso Reyero

IES Bernaldo de Quirós - Mieres

Índice

Contenido

Introducción

Presentación y objetivos

El presente proyecto, titulado "Desarrollo de un Sistema de Control de Stock basado en Angular, Spring Boot y contenedores Docker", tiene como finalidad la creación de una aplicación web moderna y eficiente para la gestión de inventarios. En la actualidad, la gestión eficiente del inventario es fundamental para cualquier empresa que maneje productos físicos. Muchas empresas aún utilizan sistemas obsoletos o registros manuales, lo que puede generar errores y pérdida de información.

Los objetivos principales de este proyecto son:

- Desarrollar una aplicación web para la gestión de stock.
- Implementar un backend robusto con Spring Boot y base de datos relacional (MySQL).
- Diseñar una interfaz intuitiva y dinámica con Angular.
- Utilizar contenedores Docker para facilitar la implementación y escalabilidad.
- Garantizar la seguridad de los datos mediante autenticación (JWT) y autorización.
- Integrar informes y estadísticas sobre los productos en stock.

Contexto

El proyecto se enmarca en la necesidad actual de las empresas de modernizar sus procesos internos, especialmente aquellos críticos como la gestión de inventario. La digitalización ofrece herramientas para optimizar recursos, reducir errores y mejorar la toma de decisiones. Este sistema busca ser una solución que aproveche tecnologías de vanguardia para ofrecer una alternativa superior a métodos tradicionales o sistemas anticuados.

Planteamiento del problema (la idea)

El problema central que aborda este proyecto es la ineficiencia y los riesgos asociados a la gestión de stock manual o con sistemas obsoletos. Estos sistemas son propensos a errores humanos, dificultades en el seguimiento en tiempo real, falta de seguridad en los datos y poca o nula capacidad de generar informes analíticos.
La idea es desarrollar un sistema de control de stock moderno que solucione estas deficiencias, proporcionando una herramienta ágil, segura y escalable.

Análisis de costes

Costes de Desarrollo: Principalmente tiempo del desarrollador. Para un proyecto académico, se considera el tiempo dedicado. En un entorno profesional, se calcularía en horas/hombre.

Software:

- IDE (IntelliJ IDEA Community, VS Code): Gratuito.
- Frameworks (Angular, Spring Boot): Open Source, gratuito.
- Base de Datos (MariaDB): Gratuito.
- Docker Desktop: Gratuito para uso personal/pequeñas empresas.
- Git/GitHub: Gratuito para repositorios públicos/privados limitados.

Hardware: Ordenador personal del desarrollador.

Despliegue (Estimación para un entorno real básico): Coste de un VPS o plataforma PaaS (ej. Heroku, AWS Free Tier inicialmente, luego escalable).

Plan de financiación

Para este proyecto académico, la financiación se basa en recursos propios (tiempo y equipamiento existente). En un contexto comercial, se buscarían inversores, préstamos o se financiaría con fondos de la empresa.

Plan de recursos humanos

El proyecto será desarrollado por un único alumno, Marcos Alonso Reyero, quien asumirá los roles de analista, diseñador, desarrollador full-stack y tester. Se contará con la tutorización del profesorado del ciclo.

Plan de prevención de riesgos

- **Riesgos Técnicos:**
    - Dificultad en la integración de tecnologías: Mitigación mediante investigación previa, prototipado y pruebas incrementales.
    - Errores de programación: Uso de buenas prácticas, revisiones de código (auto-revisión), pruebas unitarias y de integración.
    - Pérdida de datos: Uso de control de versiones (Git/GitHub) para el código. Backups regulares de la base de datos en un entorno de desarrollo/producción.
- **Riesgos de Planificación:**
    - Retrasos en el desarrollo: Metodología ágil (Scrum) para gestionar el tiempo, priorizar tareas y realizar entregas incrementales.
- **Riesgos de Seguridad:**
    - Vulnerabilidades en la aplicación: Aplicación de principios de seguridad en el desarrollo (OWASP), uso de Spring Security, HTTPS, validación de entradas.

Análisis

Introducción

En esta sección se detallan los requisitos que debe cumplir el sistema de control de stock para satisfacer las necesidades identificadas y los objetivos del proyecto. Se dividen en requisitos funcionales y no funcionales.

Descripción general

El sistema será una aplicación web que permitirá a los usuarios (administradores y empleados) gestionar productos, controlar las entradas y salidas de stock, monitorizar niveles de inventario, recibir notificaciones y visualizar estadísticas. La autenticación y autorización garantizarán que cada usuario solo acceda a las funcionalidades permitidas según su rol.

Requisitos específicos

Requerimientos funcionales

Los requerimientos funcionales describen las acciones que el sistema debe ser capaz de realizar:

- **RF01: Gestión de productos:**
    - RF01.1: El sistema permitirá crear nuevos productos (con nombre, descripción, precio, stock inicial, stock mínimo, etc.).
    - RF01.2: El sistema permitirá actualizar la información de los productos existentes.
    - RF01.3: El sistema permitirá eliminar productos (lógica o físicamente, a definir).
- **RF02: Control de stock:**
    - RF02.1: El sistema permitirá registrar entradas de productos al stock, actualizando la cantidad disponible.
    - RF02.2: El sistema permitirá registrar salidas de productos del stock, actualizando la cantidad disponible.

- **RF03: Gestión de usuarios:**
    - RF03.1: El sistema permitirá la autenticación de usuarios.
    - RF03.2: El sistema gestionará roles de usuario (Administrador, Empleado).
    - RF03.3: (Administrador) El sistema permitirá crear, modificar y eliminar cuentas de empleado.
- **RF05: Dashboard con estadísticas:**
    - RF05.1: El sistema mostrará gráficos y reportes sobre el estado del stock.
    - RF05.2: El sistema mostrará gráficos y reportes sobre los movimientos de inventario (entradas/salidas).
- **RF06: Historial de movimientos:**
    - RF06.1: El sistema registrará todas las modificaciones en el stock (entradas, salidas, ajustes) con fecha, usuario responsable y producto afectado.
    - RF06.2: El sistema permitirá consultar el historial de movimientos.

Requerimientos de interfaces externas

- **2.3.2.1. Interfaces de Usuario (UI):**
    - La interfaz será web, accesible a través de navegadores modernos (Chrome, Firefox, Edge).
    - Deberá ser intuitiva y dinámica, facilitando la navegación y el uso de las funcionalidades.
    - Será responsiva, adaptándose a diferentes tamaños de pantalla (escritorio, tablet).
- **2.3.2.2. Interfaces Hardware:** No se prevén interfaces hardware específicas más allá de los dispositivos estándar (PC, tablet) para acceder a la aplicación web.
- **2.3.2.3. Interfaces Software:**
    - El frontend (Angular) se comunicará con el backend (Spring Boot) a través de una API RESTful sobre HTTP/S.
    - El backend interactuará con una base de datos MySQL mediante JPA/Hibernate.

- **2.3.2.4. Interfaces de Comunicaciones:** La comunicación se realizará mediante el protocolo HTTP/S.

Requerimientos de rendimiento

- El sistema deberá responder a las peticiones del usuario en un tiempo aceptable (ej. < 2-3 segundos para cargas de página y operaciones comunes).
- La base de datos deberá ser capaz de manejar un volumen de datos creciente sin degradación significativa del rendimiento para el alcance del proyecto.
- El sistema debe poder gestionar múltiples usuarios concurrentes (definir un número razonable para el alcance, ej. 10-20 usuarios simultáneos sin degradación notable).

Obligaciones del diseño

- **Usabilidad:** La interfaz debe ser fácil de aprender y utilizar. Los flujos de trabajo deben ser lógicos y eficientes.
- **Seguridad:** Se deben implementar medidas para proteger los datos y el acceso al sistema (ver 2.3.5.1).
- **Mantenibilidad:** El código debe ser modular, bien documentado y seguir buenas prácticas para facilitar futuras modificaciones y correcciones.
- **Escalabilidad:** El diseño debe permitir un crecimiento futuro, tanto en volumen de datos como en número de usuarios. El uso de Docker facilitará la escalabilidad horizontal del backend si fuera necesario.

Atributos

- **2.3.5.1. Seguridad:**
    - Autenticación de usuarios mediante JWT (JSON Web Tokens).
    - Autorización basada en roles (Administrador, Empleado) para restringir el acceso a funcionalidades.
    - Protección contra vulnerabilidades comunes (ej. XSS, CSRF, SQL Injection) mediante el uso de Spring Security y buenas prácticas de codificación.
    - Las contraseñas se almacenarán de forma segura (hasheadas y salteadas) utilizando, por ejemplo, BCrypt.
    - Se recomienda el uso de HTTPS en un entorno de producción.
- **2.3.5.2. Facilidades de Mantenimiento:**
    - Código fuente organizado en módulos lógicos tanto en el frontend (componentes, servicios Angular) como en el backend (controladores, servicios, repositorios Spring Boot).
    - Uso de comentarios en el código donde sea necesario.
    - Adhesión a convenciones de codificación para Java/Spring y TypeScript/Angular.
- **2.3.5.3. Portabilidad y Escalabilidad:**
    - **Portabilidad:** El uso de Docker asegura que la aplicación se pueda ejecutar de manera consistente en diferentes entornos (desarrollo, pruebas, producción) que soporten Docker.
    - **Escalabilidad:** La arquitectura permitirá escalar componentes individualmente. El backend Spring Boot puede ser escalado horizontalmente si se diseña como stateless y se utiliza Docker Swarm o Kubernetes. La base de datos MySQL también tiene estrategias de escalado.
- **2.3.5.4. Otros Requerimientos:**
    - **Usabilidad:** (Ya mencionado en 2.3.4) Interfaz clara, consistente y con retroalimentación al usuario.
    - **Fiabilidad:** El sistema debe ser estable y minimizar el riesgo de fallos que provoquen pérdida de datos o interrupción del servicio. Transacciones de base de datos para operaciones críticas.

- **Disponibilidad:** El sistema deberá estar disponible para su uso durante las horas operativas esperadas. Docker ayuda a una rápida recuperación en caso de fallo de un contenedor.

Diseño

Introducción

Esta sección describe la arquitectura general del sistema y el diseño detallado de sus componentes principales, incluyendo las capas de presentación, negocio y persistencia, así como el diseño de la seguridad.

Arquitectura del sistema

El sistema se desarrolla siguiendo una arquitectura de tres capas (Three-Tier Architecture) desacoplada, con una API RESTful como intermediaria:

1. **Capa de Presentación (Frontend):** Desarrollada con Angular. Responsable de la interfaz de usuario, la interacción con el usuario y la comunicación con la capa de negocio a través de la API REST.
2. **Capa de Negocio o Lógica (Backend):** Desarrollada con Spring Boot (Java). Implementará la lógica de negocio, la gestión de las reglas de la aplicación, la seguridad (Spring Security) y expondrá los servicios a través de una API REST.
3. **Capa de Persistencia o Datos (Backend):** Gestionada por Spring Data JPA e Hibernate, interactuando con una base de datos relacional MariaDB (según application.properties). Responsable del almacenamiento y recuperación de los datos.

El despliegue se realizará utilizando contenedores Docker para cada uno de los servicios principales (frontend, backend, base de datos), lo que facilita la portabilidad, el aislamiento y la escalabilidad.

Diseño de la Capa de Presentación

- **Componentes:** Se crearán componentes reutilizables y específicos para las diferentes vistas y funcionalidades (ej. LoginComponent, StockManagementComponent, UserManagementComponent, DashboardViewComponent, HomeComponent, StockMovementsHistoryComponent). La navegación principal y el layout general estarán gestionados por AppComponent.
- **Servicios:** Se utilizarán servicios Angular para encapsular la lógica de comunicación con la API REST del backend (AuthService, ProductService, StockMovementService, UserService, DashboardService), así como para compartir datos y estado entre componentes mediante BehaviorSubject.
- **Módulos:** Aunque los componentes son standalone, la organización lógica se da por carpetas dentro de features (auth, admin, dashboard, stock, home) y core (auth, guards, interceptors, services).
- **Routing:** Se usará el sistema de enrutamiento de Angular (app.routes.ts) para la navegación entre las diferentes vistas de la aplicación, protegido por guards.
- **Gestión de Estado:** Principalmente a través de BehaviorSubject en los servicios para mantener y propagar el estado (ej. usuario actual en AuthService, lista de productos en ProductService).
- **Estilos:** Se utilizará CSS3, Bootstrap 5 y Font Awesome para el diseño visual y la responsividad de la interfaz. HTML5 para la estructura semántica. TypeScript como lenguaje de programación.
- **Interacción con API:** Se usará HttpClient de Angular para realizar peticiones HTTP (GET, POST, PUT, DELETE) a la API REST del backend. El authInterceptor se encargará de añadir los JWT a las cabeceras de autorización.
- **Guards:** authGuard para proteger rutas que requieren autenticación y roleGuard para proteger rutas basadas en roles específicos del usuario (ADMIN, MANAGER, USER).
- **Interceptors:** authInterceptor para adjuntar el token JWT a las peticiones salientes y manejar errores 401.

Diseño de la Capa de Negocio o Lógica

- **Controladores REST (@RestController):**
    - AuthController: Endpoints para /api/auth/login.
    - ProductController: Endpoints CRUD para /api/products y /api/products/stats.
    - StockMovementController: Endpoints para /api/stock-movements y estadísticas de movimientos.
    - UserController: Endpoints CRUD para /api/users.
    - Estos controladores utilizarán @PreAuthorize para la autorización a nivel de método basada en roles (ADMIN, MANAGER).
- **Servicios (@Service):**
    - AuthService (implícito en AuthController y JwtUtil): Lógica de autenticación y generación de tokens.
    - ProductService: Lógica de negocio para productos, incluyendo el registro de movimientos de stock al crear o actualizar productos.
    - StockMovementService: Lógica para consultar movimientos de stock y generar estadísticas.
    - UserService: Lógica de negocio para la gestión de usuarios.
- **DTOs (Data Transfer Objects):** Se utilizarán para transferir datos entre la capa de controladores y la de servicios, y como respuesta/entrada en la API: JwtResponse, LoginRequest, StockMovementResponseDTO, UserRequestDTO, UserResponseDTO.
- **Manejo de Excepciones:** Aunque no hay un @ControllerAdvice explícito en el código proporcionado, los servicios pueden lanzar IllegalArgumentException y los controladores devuelven ResponseEntity con diferentes estados HTTP. AuthEntryPointJwt maneja errores de autenticación no autorizada (401).
- **Seguridad:** Implementada con Spring Security (ver sección 3.6).
- **Inicialización de datos:** DataInitializer para crear datos de prueba (productos y usuarios con contraseñas hasheadas) al arrancar la aplicación si la base de datos está vacía.

Diseño de la Capa de Persistencia o Datos

- **Entidades JPA (@Entity):** Clases Java que mapean las tablas de la base de datos:
    - Product: Con campos como name, description, price, stock, imageUrl.
    - StockMovement: Con campos como product, type (enum ENTRADA, SALIDA, AJUSTE_INICIAL, CORRECCION), quantityChanged, stockBefore, stockAfter, movementDate, reason, user. Usa @PrePersist para la fecha.
    - User: Con campos como username, password, role (enum ADMIN, MANAGER, USER), name.
- **Repositorios Spring Data JPA (JpaRepository):** Interfaces que extienden de JpaRepository:
    - ProductRepository: Con métodos como countByStockLessThan.
    - StockMovementRepository: Con consultas JPQL personalizadas (@Query) para estadísticas y búsquedas específicas (ej. countMovementsByTypeSince, getMovementSummaryBetweenDates).
    - UserRepository: Con métodos como findByUsername, existsByUsername.
- **Configuración de la Base de Datos:** Se configurará la conexión a la base de datos MariaDB en application.properties, especificando URL, usuario, contraseña y dialecto. spring.jpa.hibernate.ddl-auto=update gestionará el esquema.
- **Transaccionalidad (@Transactional):** Los métodos de servicio que realizan operaciones de escritura o lectura se anotarán con @Transactional para asegurar la atomicidad, consistencia y correcta gestión de las sesiones de Hibernate.

Diseño de la Seguridad

- **Configuración de Spring Security (SecurityConfig.java):**
    - Se deshabilita CSRF (csrf(AbstractHttpConfigurer::disable)), apropiado para APIs stateless con JWT.
    - Se configura CORS globalmente mediante WebConfig.java y se referencia en SecurityConfig con cors(Customizer.withDefaults()).
    - Se define AuthEntryPointJwt para manejar errores 401.
    - La política de sesión es STATELESS.
    - Se definen reglas de autorización para los endpoints en authorizeHttpRequests:
        - Permitir acceso público a /api/auth/**, /h2-console/** (si estuviera habilitado), y GET a /api/products y /api/products/{id}.
        - Restringir POST, PUT, DELETE en /api/products/** y acceso a /api/stock-movements/** a roles ADMIN y MANAGER.
        - Restringir /api/users/** al rol ADMIN.
        - Cualquier otra petición requiere autenticación.
    - Se registra DaoAuthenticationProvider utilizando UserDetailsServiceImpl y PasswordEncoder (BCrypt).
    - Se añade el filtro personalizado AuthTokenFilter antes de UsernamePasswordAuthenticationFilter.
- **Autenticación (AuthController, JwtUtil, UserDetailsServiceImpl):**
    - El endpoint /api/auth/login (AuthController) recibe LoginRequest (username, password).
    - AuthenticationManager usa UserDetailsServiceImpl para cargar el usuario y PasswordEncoder para verificar la contraseña.
    - Si la autenticación es exitosa, JwtUtil.generateJwtToken() crea un JWT firmado (HS512) con el username y roles como claims, y una fecha de expiración. El token se devuelve en un JwtResponse.

- **Autorización (AuthTokenFilter, SecurityConfig, Anotaciones @PreAuthorize):**
    - El filtro AuthTokenFilter intercepta cada petición, extrae el JWT de la cabecera Authorization (Bearer <token>) usando jwtUtil.getTokenFromHeader().
    - jwtUtil.validateJwtToken() verifica la firma y expiración del token.
    - Si es válido, se extrae el username y se usa UserDetailsServiceImpl para cargar UserDetails.
    - Se crea un UsernamePasswordAuthenticationToken y se establece en SecurityContextHolder.
    - Spring Security utiliza esta información y la configuración en SecurityConfig o anotaciones @PreAuthorize en los controladores para autorizar o denegar el acceso. Los roles usados son ADMIN, MANAGER, USER directamente como strings.
- **Gestión de Contraseñas:** BCryptPasswordEncoder se usa para hashear contraseñas (ej. en DataInitializer).
- **CORS (WebConfig.java):** Configura CorsRegistry para permitir peticiones desde http://localhost:4200 (frontend Angular) a todos los endpoints bajo /api/**, permitiendo métodos específicos y todas las cabeceras.

Implementación

Tecnologías utilizadas en el desarrollo del proyecto

4.1.1. Frontend:

- **Angular (versión implícita, moderna, uso de standalone: true):** Framework para SPA.
- **TypeScript:** Lenguaje principal para Angular.
- **HTML5, CSS3:** Para estructura y estilo.
- **Bootstrap 5:** Framework CSS para diseño responsivo.
- **Font Awesome:** Para iconos.
- **Ngx-charts:** Para la visualización de gráficos en el dashboard.

4.1.2. Backend:

- **Spring Boot (v3+ implícito por uso de jakarta.persistence y jakarta.validation):** Framework Java.
- **Java (JDK 17+ implícito):** Lenguaje del backend.
- **Spring MVC, Spring Data JPA, Spring Security.**
- **Hibernate:** Implementación ORM.

4.1.3. Base de Datos:

- **MariaDB (según application.properties):** Sistema de gestión de bases de datos relacional.

4.1.4. Autenticación:

- **JWT (JSON Web Token):** Implementado con la librería io.jsonwebtoken.

4.1.5. Control de Versiones:

- **Git, GitHub** (asumido).

4.1.6. Despliegue:

- **Docker, Docker Compose** (como se planeó).

Herramientas de desarrollo:

- **IDE:** IntelliJ IDEA (para Backend), VS Code (para Frontend) (asumido).
- **Gestor de dependencias:** Maven (para Backend), npm (para Frontend) (asumido).

Descripción del Proyecto

- **Proyecto Backend (Spring Boot):**
  Organizado en paquetes dentro de es.marcosar.proyectobackend:
  - config: Clases de configuración global como DataInitializer, SecurityConfig, WebConfig.
  - controller: Controladores REST como AuthController, ProductController, StockMovementController, UserController.
  - dto: Data Transfer Objects (JwtResponse, LoginRequest, etc.) para la comunicación API.
  - entity: Entidades JPA que mapean la base de datos (Product, StockMovement, User).
  - repository: Interfaces Spring Data JPA (ProductRepository, StockMovementRepository, UserRepository).
  - security: Componentes relacionados con la seguridad JWT (AuthEntryPointJwt, AuthTokenFilter, JwtUtil, UserDetailsServiceImpl).
  - service: Clases de servicio con la lógica de negocio (ProductService, StockMovementService, UserService).
  - Archivo principal: ProyectoBackendApplication.java.
  - Configuración: resources/application.properties.
- **Proyecto Frontend (Angular):**
  Estructura típica de un proyecto Angular moderno con componentes standalone:
  - app/core/: Lógica central y transversal.
    - auth/: AuthService.
    - guards/: auth.guard.ts, role.guard.ts.
    - interceptors/: auth.interceptor.ts.
    - services/: user.service.ts (servicio de usuario global).
  - app/features/: Módulos de funcionalidad por característica.
    - admin/user-management/: UserManagementComponent.
    - auth/login/: LoginComponent.

        - dashboard/dashboard-view/: DashboardViewComponent y dashboard.service.ts.
        - home/: HomeComponent.
        - stock/:
            - services/: product.service.ts, stock-movement.service.ts.
            - stock-management/: StockManagementComponent.
            - stock-movements-history/: StockMovementsHistoryComponent.
    - app/shared/models/: Modelos de datos compartidos (product.model.ts, stock-movement.model.ts, user.model.ts).
    - Archivos principales: app.component.ts, app.config.ts, app.routes.ts.
    - Configuración de entorno: environments/environment.ts.
    - Estilos globales: styles.css.

Implementación de la Capa de Presentación

La capa de presentación se ha implementado con Angular, utilizando componentes standalone para una mayor modularidad.

- **Componentes Principales:**
    - AppComponent: Componente raíz que define la estructura general con cabecera (navegación), contenido principal (<router-outlet>) y pie de página. Gestiona los enlaces de navegación según el estado de autenticación y rol del usuario.
    - LoginComponent: Formulario para la autenticación de usuarios, interactúa con AuthService.
    - HomeComponent: Página de bienvenida estática.
    - StockManagementComponent: Permite listar, crear, editar y eliminar productos. Su funcionalidad de escritura está condicionada por el rol del usuario (ADMIN o MANAGER). Utiliza ProductService.
    - StockMovementsHistoryComponent: Muestra el historial de movimientos de stock, con opción de filtrar por productId a través de query params. Utiliza StockMovementService y ProductService.

- DashboardViewComponent: Presenta KPIs y gráficos (usando ngx-charts) sobre productos y movimientos. Utiliza DashboardService.
- UserManagementComponent: Para la gestión CRUD de usuarios, accesible solo por ADMIN. Utiliza UserService.

- **Servicios:**
  - AuthService: Gestiona la autenticación (login, logout), almacenamiento y recuperación del token JWT y datos del usuario en localStorage. Proporciona BehaviorSubject para el estado del usuario actual y autenticación.
  - ProductService: Maneja las operaciones CRUD para productos, comunicándose con la API. Utiliza un BehaviorSubject (products$) para mantener la lista de productos actualizada y reactiva en la UI.
  - StockMovementService: Obtiene los movimientos de stock desde la API.
  - UserService: Gestiona las operaciones CRUD para usuarios (accesible por admin).
  - DashboardService: Agrega llamadas a la API para obtener las estadísticas necesarias para el dashboard.
- **Routing (app.routes.ts):**
  Se definen rutas para cada funcionalidad, utilizando carga diferida (loadComponent) para los componentes de las features.
  - Rutas públicas: /login.
  - Rutas protegidas: '' (Home), /stock, /stock-movements, /dashboard, /admin/users.
  - Se utilizan authGuard para asegurar que el usuario esté autenticado y roleGuard para restringir el acceso basado en roles (ADMIN, MANAGER, USER).
- **Interceptors (auth.interceptor.ts):**
  El authInterceptor añade automáticamente el token JWT a las cabeceras Authorization de las peticiones HTTP dirigidas a la API. También maneja errores 401 (no autorizado) realizando un logout si el token es inválido o ha expirado.
- **Manejo de Formularios:** Se utiliza ReactiveFormsModule para la creación y validación de formularios (ej. LoginComponent, ProductFormComponent dentro de StockManagementComponent, UserManagementComponent).

- **Estilos:** Se utiliza Bootstrap para la maquetación y componentes UI base, complementado con CSS personalizado por componente y estilos globales en styles.css (que incluye Font Awesome).

Implementación de la Capa de Negocio o Lógica

El backend implementa una API RESTful con Spring Boot y Java.

- **Controladores REST (@RestController):**
    - AuthController: Expone el endpoint /api/auth/login para la autenticación. Utiliza AuthenticationManager y JwtUtil para generar el token.
    - ProductController: Proporciona endpoints CRUD para /api/products (ej. GET /api/products, POST /api/products, GET /api/products/{id}, PUT /api/products/{id}, DELETE /api/products/{id}) y un endpoint /api/products/stats para estadísticas de productos. Las operaciones de escritura están protegidas con @PreAuthorize("hasAnyAuthority('ADMIN', 'MANAGER')").
    - StockMovementController: Ofrece endpoints para /api/stock-movements (listar todos), /api/stock-movements/product/{productId} (listar por producto) y endpoints de estadísticas como /api/stock-movements/stats/movements-by-type y /api/stock-movements/stats/summary-last-week. Protegido con @PreAuthorize("hasAnyAuthority('ADMIN', 'MANAGER')").
    - UserController: Gestiona el CRUD de usuarios en /api/users. Totalmente protegido con @PreAuthorize("hasAuthority('ADMIN')").
- **Servicios (@Service):**
    - ProductService: Implementa la lógica de negocio para productos. Destaca el método recordStockMovement que se llama internamente al crear o modificar el stock de un producto, registrando el movimiento en StockMovementRepository.
    - StockMovementService: Proporciona métodos para consultar movimientos de stock y agregar datos para estadísticas.
    - UserService: Lógica para crear, leer, actualizar y eliminar usuarios. Realiza validaciones como la existencia del username. *Nota: el hasheo de contraseñas en createUser y updateUser está pendiente de implementación en el código proporcionado, aunque DataInitializer sí lo hace*.

    - UserDetailsServiceImpl: Implementa UserDetailsService de Spring Security para cargar los detalles del usuario por username durante la autenticación.

- **DTOs:** Se utilizan para la comunicación API, ejemplos: LoginRequest para la entrada de login, JwtResponse para la salida, UserRequestDTO y UserResponseDTO para la gestión de usuarios, y StockMovementResponseDTO para presentar los movimientos de stock con información adicional como el nombre del producto y el usuario.
- **Seguridad (SecurityConfig, JwtUtil, AuthTokenFilter):**
    - SecurityConfig: Define la cadena de filtros de seguridad, reglas de autorización para endpoints, configuración de CORS, y el PasswordEncoder (BCrypt).
    - JwtUtil: Encapsula la lógica para generar, validar y parsear tokens JWT. Los tokens incluyen el username y los roles como claims.
    - AuthTokenFilter: Filtro que se ejecuta en cada petición para validar el JWT y establecer el contexto de seguridad.
    - AuthEntryPointJwt: Maneja los intentos de acceso no autorizado, devolviendo una respuesta JSON con estado 401.
- **Configuración (WebConfig, DataInitializer):**
    - WebConfig: Configura CORS globalmente para permitir solicitudes desde el frontend Angular (http://localhost:4200).
    - DataInitializer: Utiliza CommandLineRunner para insertar productos y usuarios de prueba (con contraseñas hasheadas usando PasswordEncoder) al inicio si la base de datos está vacía.

Implementación de la Capa de Persistencia o de Datos

- **Entidades JPA (@Entity):**
    - User: Define al usuario con id, username (único), password, role (enum ADMIN, MANAGER, USER) y name.
    - Product: Define el producto con id, name, description, price, stock y imageUrl.
    - StockMovement: Registra los movimientos de stock, relacionándose con Product y User (opcionalmente). Incluye type (enum ENTRADA, SALIDA, AJUSTE_INICIAL, CORRECCION), quantityChanged, stockBefore, stockAfter, movementDate (auto-generada con @PrePersist si es nula) y reason.
- **Repositorios Spring Data JPA:**
    - UserRepository: Extiende JpaRepository. Incluye findByUsername(String username) y existsByUsername(String username).
    - ProductRepository: Extiende JpaRepository. Incluye countByStockLessThan(Integer stockThreshold).
    - StockMovementRepository: Extiende JpaRepository. Incluye findByProductIdOrderByMovementDateDesc(Long productId), findAllByOrderByMovementDateDesc(), y consultas JPQL personalizadas (@Query) para obtener estadísticas como countMovementsByTypeSince y getMovementSummaryBetweenDates.
- **Configuración de la Base de Datos (application.properties):**
  Se especifica la conexión a una base de datos MariaDB (jdbc:mariadb://localhost:3306/proyectodb). Se utiliza spring.jpa.hibernate.ddl-auto=update para que Hibernate actualice el esquema de la base de datos según las entidades. Se habilita spring.jpa.show-sql=true y spring.jpa.properties.hibernate.format_sql=true para logging de SQL en desarrollo.

Despliegue del Proyecto

El despliegue de la aplicación se realizará utilizando Docker y Docker Compose para orquestar los diferentes servicios (frontend, backend, base de datos).

- **Dockerfile para Frontend (Angular):**
    - Se utilizará una imagen base de Node para compilar la aplicación Angular (ng build).
    - Luego, los artefactos compilados (dist/) se copiarán a una imagen ligera de servidor web (ej. Nginx) para servir los archivos estáticos.
- **Dockerfile para Backend (Spring Boot):**
    - Se utilizará una imagen base de OpenJDK compatible con la versión de Java del proyecto.
    - Se copiará el archivo .jar ejecutable (generado por mvn package o gradle build) a la imagen.
    - Se expondrá el puerto en el que corre la aplicación Spring Boot (8080).
- **Docker Compose (docker-compose.yml):**
    - Se definirán tres servicios:
        1. frontend-app: Construido a partir del Dockerfile de Angular, exponiendo el puerto del servidor web (ej. 80 o 4200).
        2. backend-app: Construido a partir del Dockerfile de Spring Boot, exponiendo el puerto de la API (8080). Dependerá del servicio de base de datos.
        3. db: Utilizando una imagen oficial de MariaDB (ej. mariadb:latest o una versión específica), configurando variables de entorno para el usuario, contraseña, base de datos (ej. MYSQL_ROOT_PASSWORD, MYSQL_DATABASE, MYSQL_USER, MYSQL_PASSWORD que MariaDB también soporta por compatibilidad), y montando un volumen para la persistencia de los datos. El nombre del servicio de base de datos debe ser db para que la URL jdbc:mariadb://db:3306/proyectodb funcione desde el contenedor del backend.
    - Se configurarán las redes para permitir la comunicación entre los contenedores.

- Variables de entorno para configurar las URLs de conexión, credenciales, y el secreto JWT del backend (JWT_SECRET, JWT_EXPIRATION_MS).

Conclusión

Valoración Personal del Trabajo Realizado (Análisis DAFO + CAME)

- **Reflexión sobre el trabajo realizado:**
  - El desarrollo de este proyecto ha permitido aplicar de forma integrada los conocimientos adquiridos durante el ciclo formativo en tecnologías de frontend (Angular), backend (Spring Boot) y despliegue (Docker). Se ha logrado construir un sistema funcional que cumple con los objetivos iniciales, proporcionando una solución moderna para la gestión de stock.
  - Los principales desafíos han sido: la correcta configuración de Spring Security con JWT, la gestión del estado en Angular, o la orquestación de los contenedores Docker.
  - Los principales aprendizajes han sido: la importancia de una buena API REST, el desacoplamiento entre capas, la agilidad que proporciona Docker.
- **Análisis DAFO:**
  - **Debilidades:**
    - Alcance limitado de algunas funcionalidades avanzadas (ej. predicción de stock, integraciones complejas).
    - Pruebas de rendimiento exhaustivas no realizadas por limitaciones de tiempo/recursos.
    - Interfaz de usuario podría mejorarse con un diseño más profesional si se contara con un diseñador UX/UI.
  - **Amenazas:**
    - Rápida evolución de las tecnologías (nuevas versiones de frameworks, nuevas herramientas).
    - Posibles vulnerabilidades de seguridad si no se mantiene actualizado.

        - Competencia de soluciones de gestión de stock ya establecidas en el mercado (en un contexto comercial).
    - **Fortalezas:**
        - Uso de tecnologías modernas y demandadas (Angular, Spring Boot, Docker).
        - Sistema modular, escalable y portable gracias a Docker.
        - Funcionalidades clave para la gestión de stock cubiertas.
        - Seguridad implementada a nivel de autenticación y autorización.
    - **Oportunidades:**
        - Extender el sistema para cubrir necesidades de nichos específicos de mercado.
        - Integración con otras plataformas (e-commerce, sistemas ERP).
        - Desarrollo de una aplicación móvil complementaria.
        - Ofrecerlo como un producto SaaS (Software as a Service).
- **Análisis CAME (Corregir, Afrontar, Mantener, Explotar):**
    - **Corregir (Debilidades):**
        - Planificar futuras iteraciones para incluir funcionalidades avanzadas.
        - Dedicar tiempo específico a la optimización del rendimiento y a pruebas más exhaustivas en futuras versiones.
    - **Afrontar (Amenazas):**
        - Establecer un plan de mantenimiento para actualizar dependencias y frameworks.
        - Realizar auditorías de seguridad periódicas (en un entorno real).
    - **Mantener (Fortalezas):**
        - Continuar utilizando buenas prácticas de desarrollo para asegurar la modularidad y escalabilidad.
        - Seguir invirtiendo en la seguridad del sistema.

- **Explotar (Oportunidades):**
    - Investigar las posibilidades de integración con plataformas de e-commerce.
    - Considerar el desarrollo de una PWA o una app móvil nativa.

Posibles Ampliaciones y Líneas Futuras

- **Integración con lectores de códigos de barras:** Para agilizar la entrada y salida de productos.
- **Gestión de múltiples almacenes/ubicaciones.**
- **Informes más avanzados y personalizables:** Utilizando librerías de generación de PDFs o exportación a Excel.
- **Módulo de gestión de proveedores y órdenes de compra.**
- **Sistema de predicción de demanda:** Utilizando el historial de movimientos para estimar necesidades futuras.
- **Internacionalización (i18n):** Soporte para múltiples idiomas en la interfaz.
- **Auditoría más detallada:** Registrar no solo movimientos de stock, sino cambios en productos, usuarios, etc.
- **Integración con pasarelas de pago** si se venden productos directamente desde el sistema.
- **Aplicación móvil nativa o PWA (Progressive Web App).**

Bibliografía

Documentación oficial de Angular: https://angular.io/docs

Documentación oficial de Spring Boot: https://spring.io/projects/spring-boot

Documentación oficial de Spring Security: https://spring.io/projects/spring-security

Documentación oficial de Spring Data JPA: https://spring.io/projects/spring-data-jpa

Documentación oficial de Hibernate: https://hibernate.org/orm/documentation/

Documentación oficial de MySQL: https://dev.mysql.com/doc/

Documentación oficial de Docker: https://docs.docker.com/

Documentación oficial de TypeScript: https://www.typescriptlang.org/docs/

Documentación oficial de Bootstrap: https://getbootstrap.com/docs/

JWT.IO (JSON Web Tokens): https://jwt.io/

Baeldung - Spring Security: https://www.baeldung.com/security-spring