Documentación del Sistema de Gestión de Inventarios con QAS
Descripción General
Objetivo: Desarrollar un sistema de gestión de inventarios para pequeñas empresas que implemente todas las etapas del ciclo de vida del aseguramiento de calidad de software (QAS).

Características clave:
- Gestión completa de productos (CRUD)
- Control de stock básico
- API RESTful para integración
- Enfoque en calidad, seguridad y usabilidad

Funcionalidades Principales

1. Gestión de Productos

Funcionalidad	Descripción
Agregar Producto	Registrar nuevos productos con: nombre, descripción, categoría, precio, cantidad
Editar Producto	Modificar información de productos existentes
Eliminar Producto	Remover productos del inventario
Visualizar Productos	Listado con búsqueda y filtrado por categoría

2. API de Integración
Endpoints principales:
- POST /api/products - Crear producto
- GET /api/products - Listar productos
- PUT /api/products/{id} - Actualizar producto
- DELETE /api/products/{id} - Eliminar producto

Ciclo de Vida QAS Implementado
1. Planificación y Gestión
Riesgos identificados:
1. Cambios en requisitos (Probabilidad: Media, Impacto: Alto)
 - Mitigación: Reuniones semanales con stakeholders

2. Problemas de compatibilidad (Probabilidad: Baja, Impacto: Medio)
 - Mitigación: Pruebas tempranas en múltiples navegadores

2. Pruebas de Calidad
Estrategia de pruebas:
Distribución de Pruebas
Unitarias (JUnit)
Integración (Cucumber)
Aceptación (Cucumber)

Detalle de Pruebas Automatizadas

Pruebas BDD con Cucumber

1. Configuración de Entorno de Pruebas

CucumberSpringConfiguration.java
```java
@CucumberContextConfiguration
@ActiveProfiles("test")
@SpringBootTest(
    classes = Main.class,
    webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT
)
public class CucumberSpringConfiguration {
}
```
- Función: Configura el contexto de Spring para las pruebas de Cucumber
- Detalles:
  - @CucumberContextConfiguration: Marca esta clase como configuración para Cucumber
  - @ActiveProfiles("test"): Activa el perfil "test" para usar configuraciones específicas
  - @SpringBootTest: Inicia la aplicación Spring Boot en un puerto aleatorio para pruebas

TestSecurityConfig.java
```
@TestConfiguration
@Profile("test")
@EnableWebSecurity
public class TestSecurityConfig {
    @Bean
    @Primary
    public SecurityFilterChain testSecurityFilterChain(HttpSecurity http) throws Exception {
        return http
                .csrf(csrf -> csrf.disable())
                .authorizeHttpRequests(auth -> auth
```

```
                .anyRequest().permitAll()
            )
            .build();
    }
}
```

- Función: Configuración de seguridad para el entorno de pruebas
- Detalles:
  - Deshabilita CSRF para simplificar las pruebas
  - Permite todas las solicitudes sin autenticación
  - Se activa solo en el perfil "test"

2. Ejecutor de Pruebas

ApiTestRunner.java

```
@RunWith(Cucumber.class)
@CucumberOptions(
    features = "src/test/resources/features",
    glue = {"com.inventory.cucumber.steps", "com.inventory.cucumber.config"},
    plugin = {
        "pretty",
        "html:target/cucumber-reports.html",
        "json:target/cucumber-reports/Cucumber.json",
        "junit:target/cucumber-reports/Cucumber.xml"
    },
    publish = true,
    tags = "not @ignore"
)
public class ApiTestRunner {
}
```

- Función: Configura y ejecuta las pruebas de Cucumber
- Detalles:
  - features: Ubicación de los archivos .feature
  - glue: Paquetes donde buscar implementaciones de pasos
  - plugin: Formatos de reporte (HTML, JSON, JUnit)
  - tags: Filtra escenarios (excluye los marcados con @ignore)

3. Implementación de Pasos (Step Definitions)

ProductApiStepDefinitions.java

Configuración Inicial

```
@Before
public void setup() {
   // Configura MockMvc y limpia la base de datos antes de cada escenario
   // Elimina todos los productos existentes
}
```
- Función: Preparación del entorno antes de cada escenario
- Detalles:
  - Crea instancia de MockMvc para simular peticiones HTTP
  - Limpia la base de datos eliminando todos los productos

Pasos Given (Precondiciones)
```
@Given("the system has the following products:")
public void systemHasProducts(DataTable dataTable) {
   // Crea productos iniciales usando datos de la tabla
}

@Given("the product {string} has ID {int}")
public void productHasId(String productName, int id) {
   // Asigna un ID específico a un producto
}
```
- Función: Establece el estado inicial del sistema
- Detalles:
  - Crea productos con datos específicos para el escenario
  - Permite configurar IDs predefinidos para pruebas

Pasos When (Acciones)
```
@When("I send a GET request to {string}")
public void sendGetRequest(String endpoint) {
   // Realiza petición GET al endpoint especificado
}

@When("I send a POST request to {string} with:")
public void sendPostRequest(String endpoint, DataTable dataTable) {
   // Envía petición POST con datos del producto
}

@When("I send a PUT request to {string} with:")
public void sendPutRequest(String endpoint, DataTable dataTable) {
   // Actualiza un producto existente
```

```
}

@When("I send a DELETE request for product {string}")
public void sendDeleteRequestForProduct(String productName) {
    // Elimina un producto por nombre
}
```
- Función: Ejecuta acciones sobre la API
- Detalles:
  - Simulan todas las operaciones CRUD
  - Usan MockMvc para interactuar con los endpoints
  - Manejan datos en formato JSON

Pasos Then/And (Verificaciones)
```
@Then("I should receive status {int}")
public void verifyStatusCode(int expectedStatus) {
    // Verifica el código de estado HTTP
}

@And("the response should contain {int} products")
public void verifyProductCount(int expectedCount) {
    // Verifica cantidad de productos en la respuesta
}

@And("the product {string} should exist in the system")
public void verifyProductExists(String productName) {
    // Verifica existencia de producto
}

@And("the total inventory value should be {int}")
public void verifyTotalInventoryValue(int expectedValue) {
    // Calcula y verifica valor total del inventario
}
```
- Función: Validación de resultados
- Detalles:
  - Verifican códigos de estado HTTP
  - Comprueban contenido de respuestas JSON
  - Validan cálculos de negocio (valor del inventario)

Pruebas Unitarias con JUnit

ProductServiceTest.java

Configuración Inicial

```
@ExtendWith(MockitoExtension.class)
class ProductServiceTest {
    @Mock
    private ProductRepository productRepository;

    @InjectMocks
    private ProductServiceImpl productService;

    private Product product;

    @BeforeEach
    void setUp() {
        product = new Product();
        product.setId(1L);
        product.setName("Laptop HP");
        // ... inicializar otros campos
    }
}
```

- Función: Prepara el entorno para cada prueba
- Detalles:
  - Usa Mockito para simular el repositorio
  - Inyecta dependencias automáticamente
  - Crea instancia de producto para pruebas

Pruebas de Validación

```
@Test
void save_throwsExceptionForInvalidPrice() {
    Product invalidProduct = new Product();
    invalidProduct.setPrice(BigDecimal.valueOf(-10));

    IllegalArgumentException exception = assertThrows(
        IllegalArgumentException.class,
        () -> productService.save(invalidProduct)
    );

    assertEquals("Price must be positive", exception.getMessage());
    verify(productRepository, never()).save(any(Product.class));
}
```

- Función: Verifica validación de precios negativos
- Detalles:
  - Prueba que el servicio rechace precios inválidos
  - Verifica el mensaje de error
  - Confirma que no se llama al repositorio

Pruebas de Operaciones CRUD

```
@Test
void save_savesAndReturnsProduct() {
    when(productRepository.save(any(Product.class))).thenReturn(product);

    Product savedProduct = productService.save(product);

    assertNotNull(savedProduct);
    assertEquals("Laptop HP", savedProduct.getName());
    verify(productRepository, times(1)).save(product);
}
```

- Función: Prueba creación exitosa de producto
- Detalles:
  - Configura mock para devolver el producto guardado
  - Verifica que el servicio devuelva el producto correcto
  - Confirma que se llamó al método save del repositorio

```
@Test
void update_updatesAndReturnsProduct() {
    Product updatedProduct = new Product();
    updatedProduct.setDescription("Updated description");

    when(productRepository.findById(1L)).thenReturn(Optional.of(product));
    when(productRepository.save(any(Product.class))).thenReturn(updatedProduct);

    Product result = productService.update(1L, updatedProduct);

    assertEquals("Updated description", result.getDescription());
    verify(productRepository, times(1)).findById(1L);
}
```

- Función: Prueba actualización de producto
- Detalles:
  - Simula búsqueda y actualización
  - Verifica que los cambios se apliquen correctamente

- Confirma las interacciones con el repositorio

Pruebas de Consultas

```
@Test
void findAll_returnsProductList() {
    List<Product> products = Collections.singletonList(product);
    when(productRepository.findAll()).thenReturn(products);

    List<Product> result = productService.findAll();

    assertEquals(1, result.size());
    assertEquals("Laptop HP", result.getFirst().getName());
    verify(productRepository, times(1)).findAll();
}
```

- Función: Prueba recuperación de todos los productos
- Detalles:
  - Verifica que el servicio devuelva la lista correcta
  - Comprueba el tamaño y contenido de la lista
  - Confirma la interacción con el repositorio

```
@Test
void findById_throwsExceptionForInvalidId() {
    when(productRepository.findById(2L)).thenReturn(Optional.empty());

    RuntimeException exception = assertThrows(
        RuntimeException.class,
        () -> productService.findById(2L)
    );

    assertEquals("Product not found", exception.getMessage());
    verify(productRepository, times(1)).findById(2L);
}
```

- Función: Prueba manejo de producto no encontrado
- Detalles:
  - Simula búsqueda fallida
  - Verifica que se lance la excepción adecuada
  - Confirma el mensaje de error

Cobertura alcanzada:

- Pruebas unitarias: 100%
- Pruebas de aceptación: 71%

3. Documentación
Documentos generados:
1. Documento de requisitos (funcionales y no funcionales)
2. Guía de pruebas con:
   - 18 casos de prueba documentados
   - 3 defectos identificados y corregidos
3. Manual de usuario básico

CI/CD y Contenedorización
Implementación:
- Docker para empaquetado de la aplicación
- Flyway para migraciones de base de datos
- Pipeline básico en GitHub Actions

Gestión de Calidad
Tecnologías Implementadas
Backend:
- Spring Boot 3.5.2
- Spring Security (JWT)
- Hibernate Envers (auditoría)

Frontend:
- React.js (interfaz básica)
- keycloak.js

Base de Datos:
- PostgreSQL

Pruebas:
- JUnit 5
- Cucumber 7

DevOps:
- Docker
- GitHub Actions
- Flyway

Conclusiones
- Sistema cumple con todos los requisitos funcionales básicos
- Implementación completa del ciclo QAS según lo requerido
- Cobertura de pruebas supera los objetivos planteados

- Arquitectura preparada para futuras expansiones

Recomendaciones

1. Implementar módulo avanzado de control de stock en futuras versiones
2. Expandir pruebas de estrés para escenarios de alta demanda

Contribución Equitativa

Estrategia de trabajo:

- Commits verificados en GitHub por ambos miembros
- Revisiones de código en pareja
- Distribución equitativa de tareas