

Digitalización PetSociety

Integrantes:

Vania Vargas

Alexis Ramirez

Alan Astudillo

Felipe Navarro

Docente: Tomas Ignacio Opazo Toro

Asignatura: Desarrollo FullStack

Proyecto: PetSociety

Fecha: 12/07/2025

Índice

1. Introducción.....	3
2. Métodos y herramientas utilizadas.....	4
3. Descripción del proyecto.....	6
4. Desarrollo.....	7
5. Pruebas y calidad.....	11
6. Documentación.....	15
7. Aspectos éticos.....	21
8. Conclusiones y recomendaciones.....	22
9. Anexo.....	23
10. Bibliografía.....	43

1. Introducción

El presente informe documenta el desarrollo técnico de una solución moderna basada en microservicios para Petsociety, una empresa chilena en expansión dedicada a la venta de productos para mascotas. Con el objetivo de optimizar su infraestructura tecnológica y responder al aumento de la demanda, se diseñó e implementó una arquitectura compuesta por tres microservicios independientes: Usuario, Inventario y Carrito.

Este documento se enfoca principalmente en el microservicio Usuario, el cual fue complementado con pruebas automatizadas, documentación técnica bajo el estándar OpenAPI, y navegación enriquecida mediante HATEOAS, lo que mejora la experiencia de desarrollo y el consumo de la API.

La empresa ha experimentado un rápido crecimiento a nivel nacional, lo que evidenció las limitaciones de su sistema monolítico original, especialmente en rendimiento, disponibilidad y escalabilidad. Frente a este escenario, se adoptó una arquitectura basada en microservicios, permitiendo desacoplar funcionalidades, escalar servicios de forma independiente y facilitar su mantenimiento.

Cada microservicio fue desarrollado como una unidad autónoma con lógica de negocio específica y endpoints dedicados, siguiendo principios de diseño orientado a servicios. El microservicio Usuario, además de cubrir las operaciones CRUD esenciales, integra prácticas avanzadas como pruebas con JUnit y TestRestTemplate, generación de datos realistas con Faker, y documentación dinámica mediante Swagger UI.

Objetivos específicos del proyecto

- Implementar tres microservicios independientes siguiendo principios de arquitectura modular.
- Garantizar operaciones CRUD completas, soporte para filtros y parámetros de ruta en cada servicio.
- Integrar pruebas automatizadas y documentación técnica en al menos uno de los microservicios (Usuario).
- Simular un entorno profesional de desarrollo con herramientas como Git, Postman y Swagger UI.
- Utilizar JUnit y TestRestTemplate para validar la funcionalidad del microservicio Usuario.
- Aplicar principios RESTful y HATEOAS para mejorar la navegabilidad y el desacoplamiento de la API.

2. Métodos y herramientas utilizadas

El desarrollo del sistema se apoyó en una serie de herramientas y metodologías que facilitaron la construcción de una arquitectura limpia, probada y documentada. La siguiente tabla resume los recursos empleados durante el ciclo de vida del proyecto.

Herramienta	Función	Aporte al desarrollo
Java + Spring Boot	Desarrollo de microservicios	Facilita la creación de APIs REST robustas y escalables
Jira	Planificación y gestión de tareas	Se utilizó para crear y organizar historias de usuario, asignar tareas entre integrantes y seguir el progreso del equipo durante el desarrollo del proyecto.
Canva	Diseño colaborativo y modelado visual	Herramientas utilizadas para representar gráficamente la arquitectura de los microservicios, flujos de trabajo y modelos conceptuales de manera visual e intuitiva para todo el equipo.
JUnit5 + Mockito	Pruebas unitarias	Ayuda a validar la lógica del servicio y asegurar el correcto funcionamiento del código.
TestRestTemplate	Pruebas de integración	Permite verificar el comportamiento completo de los endpoints REST simulando peticiones reales.
Swagger (OpenAPI)	Documentación técnica de la API	Facilita la comprensión y el uso de la API por parte de desarrolladores externos e internos, permitiendo probar y visualizar los endpoints de forma intuitiva y sin necesidad de herramientas adicionales.

HATEOAS	Navegabilidad dentro de la API	Mejora la interacción entre el cliente y la API al incluir en cada respuesta información adicional sobre las posibles acciones disponibles.
Git + GitHub	Control de versiones	Trabajo colaborativo y seguimiento del código fuente
Postman	Pruebas manuales de endpoints	Herramienta gráfica para probar, validar y depurar las rutas expuestas por la API de manera rápida e interactiva.
Faker	Generación de datos de prueba	Automatiza la creación de datos realistas para pruebas, reduciendo la repetitividad y mejorando la cobertura.
Lombok	Reducción de código repetitivo	Simplifica la escritura de código eliminando la necesidad de escribir manualmente getters, setters, constructores, etc.
Visual Studio Code	Entorno de desarrollo	Entorno ágil y personalizable que permite trabajar cómodamente con Java, Spring y Git en simultáneo.
MySQL	Base de datos relacional	Almacenó los datos persistentes del sistema; se configuró una instancia exclusiva para testing.
Gradle	Gestión de dependencias y construcción	Organizó compilación, perfiles de ejecución (dev, test) y ejecución desde terminal.

3. Descripción del proyecto

El proyecto Petsociety surge como respuesta a la necesidad de digitalizar y modernizar las operaciones de una empresa chilena en crecimiento dedicada a la venta de productos para mascotas. Su plataforma web, originalmente construida bajo una arquitectura monolítica, comenzó a evidenciar limitaciones significativas en rendimiento, escalabilidad y disponibilidad, especialmente ante el aumento sostenido de usuarios y sucursales a nivel nacional.

Frente a este contexto, se diseñó una solución basada en arquitectura de microservicios, con el objetivo de lograr un sistema desacoplado, modular y de fácil mantenimiento. La estructura propuesta contempla tres microservicios principales: Usuario, Inventario y Carrito, cada uno con responsabilidades claramente definidas y capacidad de operar de forma independiente.

- Usuario: Administra la información de usuarios registrados en la plataforma, permitiendo operaciones CRUD y entregando identificadores únicos para interacción con otros servicios.
- Inventario: Gestiona el catálogo de productos disponibles en la tienda, incluyendo stock, descripciones y categorías, con soporte para búsqueda y actualización en tiempo real.
- Carrito: Controla el flujo de compra de los usuarios, permitiendo gestionar carritos activos, agregar o quitar productos, y finalizar transacciones.

El objetivo principal del proyecto fue construir un sistema distribuido que permitiera escalar según las demandas del negocio, mejorar la eficiencia del desarrollo y facilitar futuras extensiones. Se prioriza la implementación de buenas prácticas, pruebas automatizadas y documentación técnica en el microservicio Usuario, el cual sirve como núcleo del ecosistema y modelo de referencia para los demás componentes.

4. Desarrollo

Diseño y Planificación

El sistema se diseñó bajo una arquitectura de microservicios, dividiendo sus responsabilidades en tres componentes independientes: Usuario, Inventario y Carrito. Cada servicio cuenta con su propia lógica de negocio, base de datos y endpoints REST, siguiendo el patrón Modelo-Vista-Controlador (MVC). Esta estructura modular permite escalar servicios de forma individual, agilizar el mantenimiento y distribuir el trabajo en paralelo.

Las decisiones técnicas se apoyaron en dos herramientas de planificación clave:

- **Jira:** Para organizar las tareas por microservicio, asignar responsables y visualizar el avance mediante tableros ágiles.
- **Canva:** Para crear diagramas de arquitectura, modelado de datos y flujos de comunicación entre servicios, facilitando la toma de decisiones y la alineación del equipo.

La transición desde el sistema monolítico original respondió a las limitaciones de rendimiento, disponibilidad y escalabilidad que surgieron con la expansión nacional de Petsociety. Al desacoplar los componentes, se logró una mayor reutilización de lógica compartida, una detección de errores más rápida y un desarrollo más colaborativo.

Implementación

Para la construcción de los microservicios se eligió un conjunto de tecnologías por su robustez y compatibilidad con Spring Boot:

- **Java 21 y Spring Boot:** Se utilizaron como base para la construcción de cada microservicio. Spring Boot permitió estructurar el código bajo el patrón Modelo-Vista-Controlador (MVC), facilitando la separación de responsabilidades en capas bien definidas (controladores, servicios, repositorios y modelos). Gracias a su ecosistema, se logró integrar de forma fluida herramientas de documentación, pruebas y gestión de dependencias.

- **Gradle:** Fue empleado como gestor de construcción para compilar el proyecto, manejar dependencias y ejecutar los microservicios desde la terminal. Su configuración modular permitió definir perfiles de ejecución para desarrollo y pruebas, lo cual fue clave para el entorno del microservicio Usuario.
- **MySQL:** Actuó como base de datos relacional para almacenar la información del sistema. Aunque todos los servicios compartieron la misma base en tiempo de ejecución, el microservicio Usuario implementó una base de datos exclusiva para pruebas, configurada mediante `application-test.properties`, lo que permitió ejecutar los tests de forma aislada.
- **Swagger (springdoc-openapi):** Se integró al microservicio Usuario para generar documentación interactiva de sus endpoints. Esto permitió validar manualmente la funcionalidad, simular peticiones sin Postman y facilitar la comprensión de los recursos disponibles.
- **JUnit 5, Mockito y TestRestTemplate:** Se aplicaron exclusivamente al microservicio Usuario para validar la lógica de negocio. JUnit permitió definir casos de prueba estructurados, Mockito facilitó la simulación de dependencias internas, y TestRestTemplate permitió verificar el comportamiento completo de los endpoints como si se tratara de un cliente real.
- **Faker:** Se incorporó en los tests para generar datos dinámicos (usuarios con nombre, correo y dirección aleatorios), reduciendo la repetición de código y aumentando la variedad de escenarios evaluados.
- **Git y GitHub:** Fueron esenciales para el control de versiones. El proyecto se organizó en un repositorio central con commits frecuentes y ramas por microservicio, lo que facilitó el seguimiento de cambios y la colaboración.
- **Spring HATEOAS:** Se aplicó únicamente en el microservicio Usuario para enriquecer las respuestas REST con enlaces automáticos. Esta implementación permitió que cada recurso incluyera referencias a acciones disponibles, mejorando la navegabilidad sin necesidad de hardcodear rutas.
- **Postman:** Usado en las primeras fases para validación manual de endpoints.

Estas tecnologías no sólo fueron integradas por su compatibilidad con Java y Spring, sino también porque permitieron cumplir los objetivos funcionales y técnicos del proyecto de forma progresiva, estructurada y validable.

Procedimientos técnicos

El desarrollo de los microservicios siguió una serie de pasos estandarizados para garantizar coherencia, aislamiento de entornos y validación continua:

1. Configuración de perfiles de entorno

- En `application.properties` se definieron los perfiles dev, prod y test.
- El perfil test apuntó a la base de datos aislada `bdpetsociety_test` para ejecutar pruebas sin afectar datos reales.

2. Estructura del proyecto y organización por paquetes

El código fuente se distribuyó en paquetes que reflejan claramente la separación de responsabilidades. En el microservicio Usuario, la estructura incluye:

- `controller`: expone los endpoints REST.
- `service`: define la lógica del negocio y la precarga de datos (`DataLoader`).
- `repository`: acceso a datos mediante `JpaRepository`.
- `model`: entidades como `Usuario` y modelos auxiliares como `ApiErrorModel`.
- `assemblers`: implementación de HATEOAS mediante `UsuarioModelAssembler`.
- `config`: configuración de Swagger (`SwaggerConfig.java`).
- `test`: clases que contienen pruebas unitarias e integración.
- `resources`: archivos de configuración por entorno (`application-dev.properties`, `application-test.properties`, etc.).

3. Ejecución local y validación funcional

Los servicios se ejecutaron localmente con Gradle (`./gradlew bootRun`). La validación inicial se hizo mediante Swagger UI y Postman, verificando la estructura y comportamiento esperado de las respuestas JSON.

4. Documentación interactiva con Swagger

El microservicio Usuario integra springdoc-openapi, que genera documentación dinámica y navegable basada en la especificación OpenAPI. Esto permite probar los endpoints desde el navegador y visualizar sus esquemas, parámetros y respuestas esperadas.

5. Pruebas automatizadas (usUsuario)

El microservicio Usuario incluyó pruebas de integración con:

- @SpringBootTest para levantar el contexto completo de la aplicación.
- TestRestTemplate para simular peticiones HTTP reales.
- Faker para generar datos dinámicos y evitar entradas estáticas.
- Limpieza de base de datos antes de cada prueba usando @BeforeEach.

6. Implementación de HATEOAS

Se empleó el ensamblador UsuarioModelAssembler, que agrega enlaces contextuales a cada respuesta, guiando al cliente API sobre las operaciones disponibles sin necesidad de conocer rutas estáticas.

Ejemplos de código y explicaciones técnicas

A continuación, se presenta el ensamblador utilizado en el microservicio Usuario para implementar navegación HATEOAS:

```
@Override
public EntityModel<Usuario> toModel(Usuario usuario) {
    EntityModel<Usuario> model = EntityModel.of(usuario);

    if (usuario.getId() != null) {

        model.add(linkTo(methodOn(UsuarioController.class).obtenerUsuario
        PorId(usuario.getId())).withSelfRel());

        model.add(linkTo(methodOn(UsuarioController.class).obtenerTodosUs
        uarios()).withRel("usuarios"));
    }
}
```

```
model.add(linkTo(methodOn(UsuarioController.class).actualizarUsua-
rio(usuario.getId(), null)).withRel("actualizar"));

model.add(linkTo(methodOn(UsuarioController.class).eliminarUsuari-
o(usuario.getId())).withRel("eliminar"));
    }

    return model;
}
```

Este patrón RepresentationModelAssembler transforma una entidad Usuario en un EntityModel, añadiendo automáticamente enlaces relevantes al recurso actual y a operaciones relacionadas. Así, cada respuesta HTTP no solo entrega datos, sino también instrucciones para navegar el API, promoviendo un diseño RESTful desacoplado, flexible y autoexplorable.

5. Pruebas y calidad

Durante el desarrollo del microservicio Usuario, se implementaron pruebas automatizadas con el objetivo de asegurar la estabilidad y confiabilidad de las funcionalidades principales, como el registro, edición y recuperación de usuarios en diferentes escenarios.

Pruebas unitarias

Las pruebas del microservicio Usuario se desarrollaron dentro de la clase PetsocietyApplicationTests.java, utilizando **JUnit 5** como framework principal y **TestRestTemplate** para simular llamadas HTTP reales a los endpoints REST. En algunos casos, también se empleó **Mockito** para realizar pruebas aisladas de componentes individuales y **Faker** para la generación de datos dinámicos (nombres, correos, direcciones) que enriquecieron la variedad y realismo de los escenarios evaluados.

Para garantizar que cada prueba partiera desde un entorno limpio y evitar interferencias entre ejecuciones, se incorporó una limpieza automática de la base de datos con `@BeforeEach`:

```
@BeforeEach
void limpiarBaseDeDatos() {
    usuarioRepository.deleteAll();
}
```

Esta práctica permitió mejorar la confiabilidad de los resultados y mantener la integridad de los datos en cada ciclo de prueba. El entorno test fue configurado con una base de datos exclusiva (`bdpetsociety_test`), aislada de los entornos dev y prod.

Uno de los casos de prueba más representativos fue el de actualización de usuario. Este test automatizado valida el flujo completo: creación de un usuario de prueba, modificación de datos, envío de la solicitud PUT y verificación del cambio mediante una consulta GET posterior.

```
@Test
@Order(6)
void updateUsuarioShouldSucceed() {
    Long id = crearUsuarioDePrueba("Update");

    assertThat(id)
        .withFailMessage("No se encontró el ID del usuario de prueba. Asegúrate de que el UsuarioTest se haya ejecutado.")
        .isNotNull();

    // Creamos un objeto Usuario con nuevos datos para actualizar
    Usuario actualizado = new Usuario();
    actualizado.setPrimerNombre("NombreActualizado");
    actualizado.setSegundoNombre("SegundoActualizado");
    actualizado.setPrimerApellido("ApellidoActualizado");

    actualizado.setSegundoApellido("SegundoApellidoActualizado");
```

```
actualizado.setEmail("actualizado_" + id + "@email.com");
actualizado.setContrasenia("nuevaContrasenia123");
actualizado.setDireccion("Nueva Dirección 456");

// Creamos la solicitud HTTP que lleva el objeto
actualizado en el cuerpo
        HttpEntity<Usuario> request = new
HttpEntity<>(actualizado);

// Enviamos una solicitud PUT para actualizar el usuario
por ID
        ResponseEntity<Void> response = restTemplate.exchange(
            "http://localhost:" + port + "/api/v1/usuarios/" +
id,
            HttpMethod.PUT,
            request,
            Void.class
        );

// Verificamos que el status devuelto sea 204 (sin
contenido) o 200 (éxito con contenido)
        assertThat(response.getStatusCode())
            .withFailMessage("Se esperaba un código 204
NO_CONTENT o 200 OK, pero se recibió: %s",
response.getStatusCode())
            .isIn(HttpStatus.NO_CONTENT, HttpStatus.OK);

// Recuperamos el usuario actualizado desde el backend
        ResponseEntity<Usuario> getResponse =
restTemplate.getForEntity(
            "http://localhost:" + port + "/api/v1/usuarios/" +
id,
            Usuario.class
        );

// Validamos que el usuario devuelto no sea null
        Usuario usuarioActualizado = getResponse.getBody();
        assertThat(usuarioActualizado).isNotNull();
```

```
// Comprobamos que los cambios realmente se hayan  
aplicado  
  
assertThat(usuarioActualizado.getPrimerNombre()).isEqualTo("NombreActualizado");  
  
assertThat(usuarioActualizado.getEmail()).contains("actualizado_");  
}
```

- Las pruebas ejecutadas fueron exitosas, demostrando estabilidad en las funcionalidades implementadas. **Ver anexo** para visualizar el resultado.
- Se cubrieron los métodos más críticos del servicio, incluyendo inserción de datos, consulta y validación de errores.
- Las pruebas sirvieron como respaldo para refactorizar el código sin afectar el comportamiento esperado.

Todas las pruebas se ejecutaron sobre una base de datos independiente, configurada especialmente para el entorno de testeo. Los resultados fueron exitosos, evidenciando estabilidad en los métodos críticos del servicio, como inserción, consulta y validación de errores. Este respaldo permitió refactorizar el código sin comprometer la funcionalidad existente.

Aseguramiento de calidad

Además de las pruebas automatizadas, se aplicaron las siguientes estrategias para mantener altos estándares de calidad:

- Separación de responsabilidades: cada clase cumple una función específica según la arquitectura MVC.
- Validaciones de datos: en la capa de servicio se manejan errores comunes, como entradas nulas o inexistentes.
- Mensajes de error personalizados: se proporciona retroalimentación clara ante fallos en la solicitud.
- Revisión de código grupal: los cambios se revisaron entre integrantes del equipo antes de ser integrados al repositorio.

Adicionalmente, se utilizó **Postman** para realizar pruebas manuales de los endpoints durante las primeras fases del desarrollo, lo que facilitó la detección

temprana de errores y mejoró el ritmo de iteración antes de automatizar los casos críticos.

Estas prácticas contribuyeron a una base de código más robusta, mantenible y lista para su evolución futura.

6. Documentación

Para garantizar la claridad y mantenibilidad del sistema, se implementó documentación técnica automatizada y navegable para los microservicios utilizando herramientas estándar de la industria como Swagger. La generación de la documentación se integró dentro de la configuración de Spring Boot, lo que permitió que la especificación de los endpoints y sus respuestas estuviera siempre sincronizada con el desarrollo.

Uso de estándares: OpenAPI Specification (OAS)

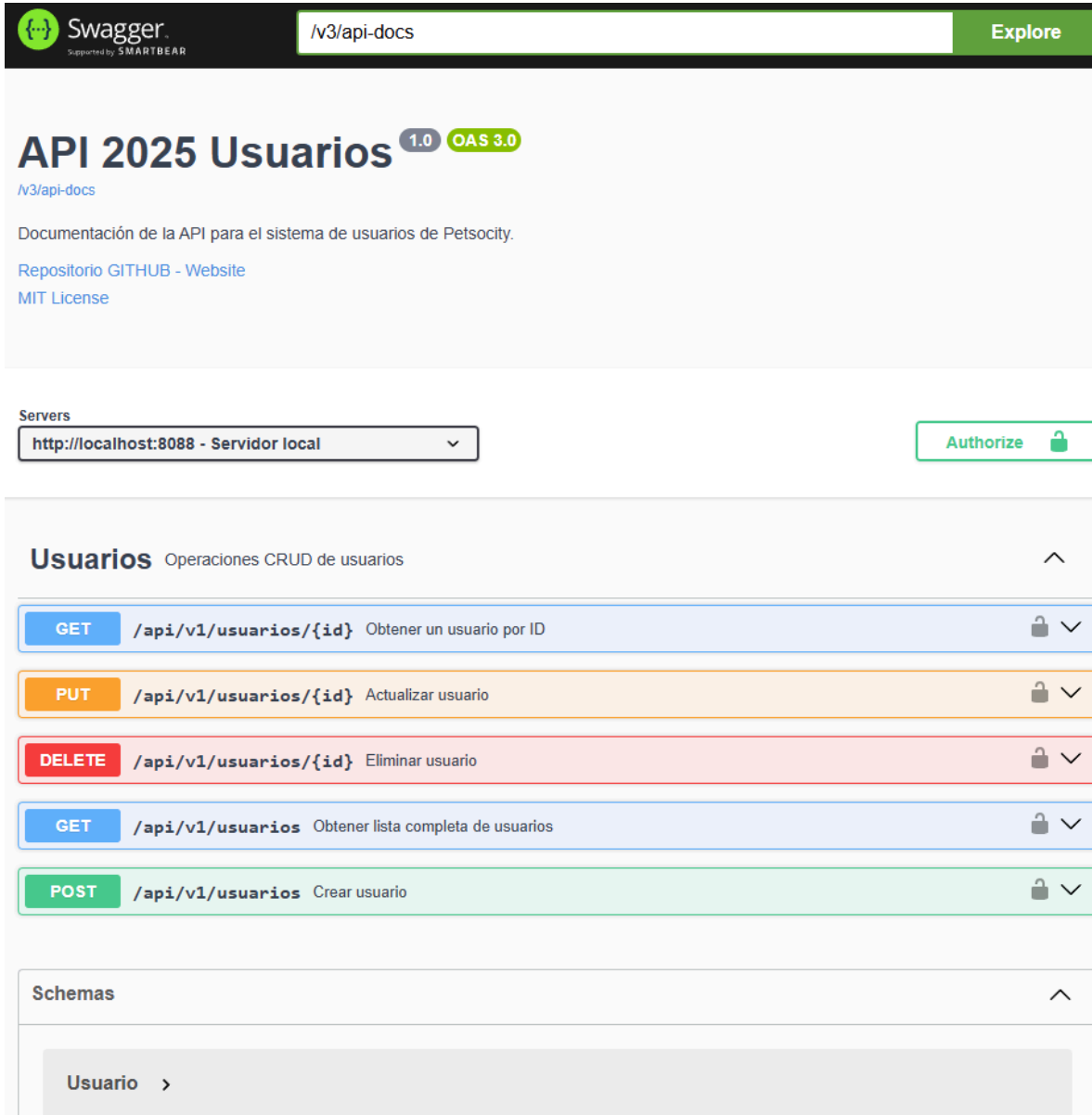
Se utilizó el estándar OpenAPI Specification (OAS) para documentar los endpoints RESTful del microservicio usuario. Esto permitió:

- Estandarizar la descripción de recursos, operaciones, parámetros y respuestas.
- Facilitar pruebas manuales a través de Swagger UI.
- Generar documentación interactiva accesible desde el navegador (/swagger-ui.html).

La documentación incluye detalles como:

- Métodos HTTP disponibles (GET, POST, PUT, DELETE).
- Parámetros requeridos en las rutas y el cuerpo de la petición.
- Respuestas esperadas con códigos de estado HTTP.

Ejemplos de documentación generada:



The image shows a Swagger UI interface for an API. At the top, there's a Swagger logo and a search bar containing "/v3/api-docs". Below this, the API title "API 2025 Usuarios" is displayed with version "1.0" and "OAS 3.0" tags. A description states it's for the Petsociety user system. Links to the GitHub repository and MIT license are provided. The "Servers" section shows a local host URL. The "Authorize" button is visible. The "Usuarios" section lists five endpoints: GET for user by ID, PUT for update, DELETE for delete, GET for list, and POST for create. The "Schemas" section shows a "Usuario" schema.

Swagger
Supported by SMARTBEAR

/v3/api-docs **Explore**

API 2025 Usuarios ^{1.0} OAS 3.0

/v3/api-docs

Documentación de la API para el sistema de usuarios de Petsociety.

[Repositorio GITHUB - Website](#)
[MIT License](#)

Servers

http://localhost:8088 - Servidor local

Authorize

Usuarios Operaciones CRUD de usuarios

Method	Endpoint	Description	Lock	Dropdown
GET	/api/v1/usuarios/{id}	Obtener un usuario por ID	🔒	▼
PUT	/api/v1/usuarios/{id}	Actualizar usuario	🔒	▼
DELETE	/api/v1/usuarios/{id}	Eliminar usuario	🔒	▼
GET	/api/v1/usuarios	Obtener lista completa de usuarios	🔒	▼
POST	/api/v1/usuarios	Crear usuario	🔒	▼

Schemas

Usuario >

GET
/api/v1/usuarios/{id}
Obtener un usuario por ID

Busca un usuario específico usando su ID

Parameters
Try it out

Name	Description
id * required integer(\$int64) (path)	<input type="text" value="1"/>

Responses

Curl

```
curl -X 'GET' \
'http://localhost:8088/api/v1/usuarios/1' \
-H 'accept: application/json'
```

Request URL

```
http://localhost:8088/api/v1/usuarios/1
```

Server response

Code
Details

200

Response body

```
{
  "id": 1,
  "primerNombre": "Calista",
  "segundoNombre": "Jerald",
  "primerApellido": "Bartell",
  "segundoApellido": "Renner",
  "email": "jefferey.goyette@yahoo.com",
  "contrasenia": "w3w732i5s47a",
  "direccion": "516 Cassandra Via, Gleichnerchester, WV 99575",
  "fechaCreacion": "2025-07-03T02:58:46.035358",
  "_links": {
    "self": {
      "href": "http://localhost:8088/api/v1/usuarios/1"
    },
    "usuarios": {
      "href": "http://localhost:8088/api/v1/usuarios"
    },
    "actualizar": {
      "href": "http://localhost:8088/api/v1/usuarios/1"
    },
    "eliminar": {
      "href": "http://localhost:8088/api/v1/usuarios/1"
    }
  }
}
```

Download

Response headers

```
connection: keep-alive
content-type: application/json
date: Sat, 12 Jul 2025 01:09:14 GMT
keep-alive: timeout=60
transfer-encoding: chunked
```

Responses

Code	Description	Links
200	Usuario encontrado Media type <input type="text" value="application/json"/> Controls Accept header. Example Value Schema <pre> { "id": 0, "primerNombre": "string", "segundoNombre": "string", "primerApellido": "string", "segundoApellido": "string", "email": "hXToTAHZh_vjiERtN4DmXN6pYORVvFH31-IftPN7s1Mr_T1q000mqIzrsEtrclky-@VYTVnMzi6-emke5ew1019mz9iRfN3WDCUP.g1kP", "contrasenia": "string", "direccion": "string", "fechaCreacion": "2025-07-12T01:10:40.996Z" }</pre>	No links
404	Usuario no encontrado Media type <input type="text" value="application/json"/> Example Value Schema <pre> { "error": "string", "detalles": "string", "status": 0, "path": "string", "timestamp": "2025-07-12T01:10:40.997Z" }</pre>	No links

En Swagger UI se puede visualizar y probar de forma interactiva todos los endpoints relacionados al microservicio de gestión de usuarios. Entre ellos destacan:

- GET `/api/v1/usuarios/{id}`: Recupera un usuario por su identificador único. La respuesta incluye los enlaces HATEOAS, lo cual permite navegar hacia otras operaciones disponibles para ese recurso, como actualizar, eliminar o consultar la lista completa.
- POST `/api/v1/usuarios`: Permite crear un nuevo usuario. En esta operación se visualizan claramente las restricciones de validación, como campos obligatorios (`@NotBlank`), formato de correo electrónico (`@Email`) y longitud mínima de caracteres (`@Size`), definidas en los esquemas del modelo de entrada.
- PUT `/api/v1/usuarios/{id}`: Actualiza los datos de un usuario existente. La documentación muestra los campos esperados, validaciones, y respuestas posibles como éxito (200 OK) o error (404 Not Found, si el ID no existe).
- DELETE `/api/v1/usuarios/{id}`: Elimina un usuario por ID. Swagger detalla los posibles códigos de estado, incluyendo éxito (204 No Content) o error en caso de inexistencia del recurso.

- GET /api/v1/usuarios: Lista todos los usuarios registrados. Al tratarse de una colección, se presentan múltiples enlaces HATEOAS por cada entidad, facilitando la navegación entre elementos individuales.

Cada una de estas operaciones fue documentada de forma automática utilizando el estándar OpenAPI Specification (OAS), gracias a la configuración del proyecto en Spring Boot. Esto permite que la documentación esté siempre actualizada con el estado real del desarrollo y sea fácilmente navegable desde el navegador a través de la interfaz gráfica de Swagger UI.

La implementación de HATEOAS refuerza la navegabilidad de la API. Por ejemplo, al consultar un usuario individual, el sistema responde con un objeto enriquecido que incluye enlaces a otras acciones relevantes, sin necesidad de que el cliente conozca previamente las rutas exactas. Esta estrategia promueve el diseño desacoplado y la capacidad de autodescubrimiento del API por parte del consumidor.

Documentación HATEOAS

Se implementó una documentación dinámica y contextual usando HATEOAS (Hypermedia as the Engine of Application State). Esto se logró mediante el ensamblador UsuarioModelAssembler, el cual:

- Añade enlaces a las respuestas tipo EntityModel y CollectionModel.
- Permite navegar entre operaciones sin depender de rutas estáticas.

Ejemplos y explicación de la navegabilidad

Cuando se accede a un recurso mediante GET /api/usuarios/1, la respuesta incluye enlaces como:

```
{
  "_embedded": {
    "usuarioList": [
      {
        "id": 1,
        "primerNombre": "Calista",
        "segundoNombre": "Jerald",
        "primerApellido": "Bartell",
        "segundoApellido": "Renner",
        "email": "jefferey.goyette@yahoo.com",
        "contrasenia": "w3w732i5s47a",
        "direccion": "516 Cassandra Via, Gleichnerchester, WV 99575",
        "fechaCreacion": "2025-07-03T02:58:46.035358",
        "_links": {
          "self": { "href": "http://localhost:8088/api/v1/usuarios/1" },
          "usuarios": { "href": "http://localhost:8088/api/v1/usuarios" },
          "actualizar": { "href": "http://localhost:8088/api/v1/usuarios/1" },
          "eliminar": { "href": "http://localhost:8088/api/v1/usuarios/1" }
        }
      }
    ]
  },
  "_links": {
    "self": { "href": "http://localhost:8088/api/v1/usuarios" }
  }
}
```

Estos enlaces permiten al cliente recorrer el API sin necesidad de conocer las rutas, promoviendo desacoplamiento y autodescubrimiento.

7. Aspectos éticos

La implementación de soluciones tecnológicas conlleva una serie de responsabilidades éticas, especialmente cuando se manejan datos personales, procesos automatizados y decisiones que pueden impactar a usuarios y colaboradores. En el contexto del proyecto Petsociety, se consideraron diversas dimensiones éticas que afectan tanto al desarrollo como a la operación del sistema basado en microservicios.

A continuación, se presentan los principales temas éticos identificados, sus implicancias y las soluciones propuestas por el equipo:

Tema Ético	Descripción del riesgo	Enfoque para mitigar riesgo
Privacidad de Datos	Almacenar datos personales implica el riesgo de filtraciones o mal uso.	Se implementarán técnicas como cifrado AES-256, validaciones de entrada y autenticación segura.
Responsabilidad Técnica	Fallos en un microservicio pueden provocar errores en cadena en todo el sistema.	Se aplicaron pruebas automatizadas y un enfoque de despliegue progresivo para minimizar riesgos.
Impacto en el Empleo	La digitalización podría reemplazar ciertas tareas manuales, afectando a los trabajadores.	Se propone reentrenar al personal afectado hacia roles de análisis, soporte y administración.
Trazabilidad y Auditoría	Es necesario registrar las acciones realizadas para garantizar transparencia y control.	Se habilitarán logs individuales por microservicio y monitoreo centralizado mediante dashboards.

Estas consideraciones no fueron abordadas únicamente desde lo técnico, sino también desde una perspectiva de responsabilidad social. Al integrar prácticas como el cifrado de datos, la autenticación reforzada y la planificación del impacto

organizacional, se busca que el sistema no solo sea eficiente, sino también confiable, transparente y justo para todos sus usuarios y colaboradores.

8. Conclusiones y recomendaciones

El desarrollo de los microservicios Usuario, Inventario y Carrito marcó un paso importante en la modernización de la plataforma Petsociety, reemplazando la arquitectura monolítica por una arquitectura distribuida, escalable e independiente. Esta decisión permitió mejorar la mantenibilidad del sistema, facilitar el trabajo en equipo y adaptarse al crecimiento progresivo del negocio.

Durante el proceso, se logró cumplir los objetivos principales del proyecto: implementar servicios con operaciones CRUD completas, aplicar principios RESTful y HATEOAS, automatizar pruebas con JUnit y TestRestTemplate, y generar documentación técnica bajo estándares como OpenAPI Specification (OAS) usando Swagger UI. El microservicio Usuario fue el más completo en cuanto a funcionalidades, pruebas automatizadas, documentación interactiva y calidad técnica.

Como grupo, este proyecto representó un gran avance en nuestra formación como desarrollador backend. Aprendimos a enfrentar desafíos reales como la configuración de entornos de testeo aislados, la integración de documentación dinámica, la generación de datos de prueba realistas y la estructuración de controladores bajo buenas prácticas de diseño. También reforcé la importancia del desacoplamiento, el trabajo colaborativo, y la trazabilidad técnica.

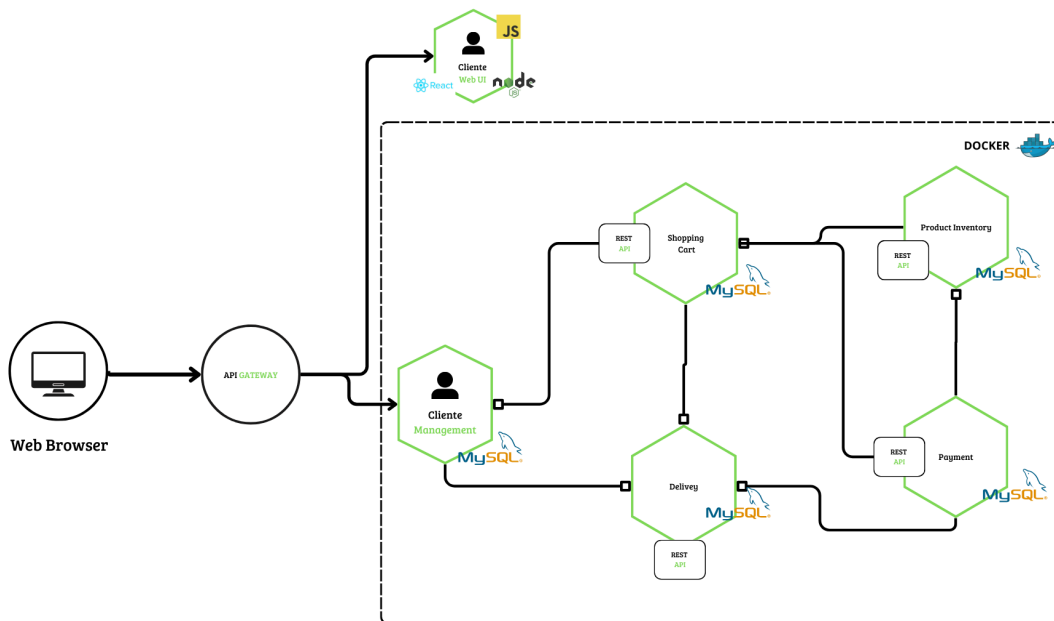
Para futuros proyectos similares, se recomienda:

- Extender la lógica de negocio y pruebas en los microservicios Inventario y Carrito para igualar la madurez alcanzada en Usuario.
- Fortalecer los mecanismos de seguridad, incluyendo autenticación, validación de datos y cifrado.
- Definir desde el inicio un esquema de validación técnica y ética, incluyendo documentación automatizada y planificación de impacto organizacional.
- Mantener un enfoque modular que permita escalar y evolucionar el sistema sin afectar su núcleo.
- Implementar otros microservicios que contemplen la parte de Delivery y Pagos

Más allá del código, este proyecto reafirma que una solución bien construida no solo mejora el rendimiento técnico, sino también la experiencia del usuario y la confianza en la plataforma.

9. Anexo

Diagrama de arquitectura del sistema: Se incluye un esquema visual que representa la división por microservicios (Usuario, Inventario y Carrito), junto con sus interacciones y dependencias tecnológicas. Este diagrama fue elaborado en Canva durante la fase de planificación



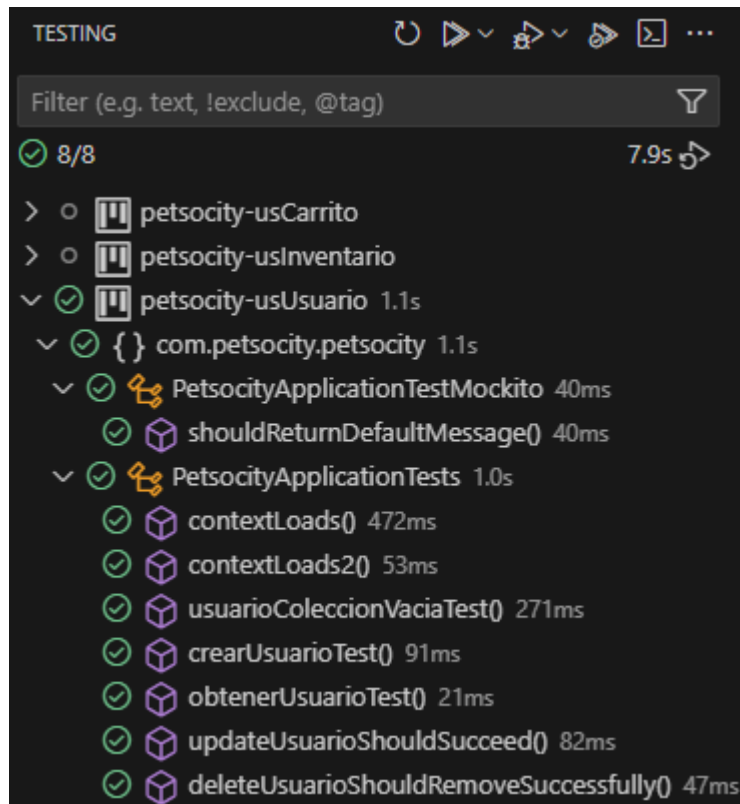
Repositorio del proyecto en Github: El código completo del proyecto se encuentra disponible en el repositorio:

- [Repositorio Petsociety](#)

En el microservicio Usuario, el repositorio incluye:

- Clases principales: `UsuarioController.java`, `UsuarioService.java`, `UsuarioRepository.java`, `Usuario.java`.
- Clase ensambladora HATEOAS: `UsuarioModelAssembler.java`.
- Clases de prueba: `PetsocietyApplicationTests.java`, `PetsocietyApplicationTestMockito.java`.
- Archivos de configuración: `build.gradle`, `application-test.properties`, y ajustes Swagger.

Capturas de pruebas automatizadas: A continuación se presenta un extracto del registro generado por Gradle tras ejecutar las pruebas automatizadas. Este log evidencia la correcta ejecución de los casos definidos para los microservicios Usuario:



Este registro confirma que todas las pruebas fueron ejecutadas de forma exitosa, incluyendo validaciones de inserción, actualización, eliminación y recuperación de datos. El entorno de testeo fue aislado (`bdpetsociety_test`), lo que refuerza la fiabilidad de los resultados y permite refactorizar el sistema sin comprometer funcionalidades existentes.

Copias de código fuente incluidas: A continuación se detallan las clases relevantes del proyecto petsociety, agrupadas por su función dentro del sistema:

Controladores:

- `UsuarioController.java`: gestiona las solicitudes HTTP relacionadas con operaciones de usuario.

```
@RestController
@RequestMapping("/api/v1/usuarios")
@Tag(name = "Usuarios", description = "Operaciones CRUD de usuarios")
public class UsuarioController {

    private final UsuarioService usuarioService;
    private final UsuarioModelAssembler assembler;

    public UsuarioController(UsuarioService usuarioService,
        UsuarioModelAssembler assembler) {
        this.usuarioService = usuarioService;
        this.assembler = assembler;
    }

    // Leer todo
    @GetMapping(produces = {MediaTypes.HAL_JSON_VALUE,
        MediaType.APPLICATION_JSON_VALUE})
    @Operation(summary = "Obtener lista completa de usuarios", description =
        "Retorna todos los usuarios registrados")
    @ApiResponses(value = {
        @ApiResponse(responseCode = "200",
            description = "Operacion exitosa",
            content = @Content(mediaType = "application/json",
                schema = @Schema(implementation = Usuario.class))),
        @ApiResponse(responseCode = "400",
            description = "Error al obtener usuarios",
            content = @Content(mediaType = "application/json",
                schema = @Schema(implementation = ApiErrorModel.class)))
    })
    public CollectionModel<EntityModel<Usuario>> obtenerTodosUsuarios() {
        List<Usuario> usuarios = usuarioService.obtenerTodosUsuarios();
        return assembler.toCollection(usuarios);
    }
}
```

```
}

// Leer por ID
@GetMapping(value =("/{id}", produces = {MediaTypes.HAL_JSON_VALUE,
MediaType.APPLICATION_JSON_VALUE})
@Operation(summary = "Obtener un usuario por ID", description = "Busca
un usuario especifico usando su ID")
@ApiResponses(value = {
    @ApiResponse(responseCode = "200",
        description = "Usuario encontrado",
        content = @Content(mediaType = "application/json",
            schema = @Schema(implementation = Usuario.class))),
    @ApiResponse(responseCode = "404",
        description = "Usuario no encontrado",
        content = @Content(mediaType = "application/json",
            schema = @Schema(implementation = ApiErrorModel.class)))
})
public ResponseEntity<EntityModel<?>>
obtenerUsuarioPorId(@PathVariable (name = "id") Long id) {
    Usuario usuario = usuarioService.obtenerPorIdUsuario(id);

    if (usuario == null) {
        ApiErrorModel error = new ApiErrorModel(
            "Usuario no encontrado",
            "No existe el usuario con ID " + id,404,
            "/api/v1/usuarios/" + id,
            LocalDateTime.now()
        );
        return ResponseEntity.status(HttpStatus.NOT_FOUND)
            .contentType(MediaTypes.HAL_JSON)
            .body(assembler.wrapError(error));
    }
    return ResponseEntity.ok(assembler.toModel(usuario));
}

// Crear usuario
@PostMapping(produces = {MediaTypes.HAL_JSON_VALUE,
MediaType.APPLICATION_JSON_VALUE})
@Operation(summary = "Crear usuario", description = "Registra un nuevo
usuario")
@ApiResponses(value = {
    @ApiResponse(responseCode = "201",
        description = "Usuario creado",
        content = @Content(mediaType = "application/json",
```

```
        schema = @Schema(implementation = Usuario.class))),
    @ApiResponse(responseCode = "400",
        description = "Datos invalidos o duplicados",
        content = @Content(mediaType = "application/json",
            schema = @Schema(implementation = ApiErrorModel.class)))
    })
    public ResponseEntity<EntityModel<?>> crearUsuario(@RequestBody @Valid
        Usuario usuario) {
        try {
            Usuario creado = usuarioService.crearUsuario(usuario);
            URI location =
                linkTo(methodOn(UsuarioController.class).obtenerUsuarioPorId(creado.getId(
                ))).toUri();
            return ResponseEntity.created(location)
                .contentType(MediaType.HAL_JSON)
                .body(assembler.toModel(creado));
        } catch (IllegalArgumentException e) {
            ApiErrorModel error = new ApiErrorModel(
                "Error de validacion",
                e.getMessage(), 400,
                "/api/v1/usuarios",
                LocalDateTime.now()
            );
            return ResponseEntity.badRequest()
                .contentType(MediaType.HAL_JSON)
                .body(assembler.wrapError(error));
        }
    }

    // Actualizar usuario
    @PutMapping(value =("/{id}", produces = {MediaType.HAL_JSON_VALUE,
        MediaType.APPLICATION_JSON_VALUE})
    @Operation(summary = "Actualizar usuario", description = "Modifica
        atributos del usuario por su ID")
    @ApiResponses(value = {
        @ApiResponse(responseCode = "200",
            description = "Usuario actualizado",
            content = @Content(mediaType = "application/json",
                schema = @Schema(implementation = Usuario.class))),
        @ApiResponse(responseCode = "400",
            description = "ID invalido",
            content = @Content(mediaType = "application/json",
                schema = @Schema(implementation = ApiErrorModel.class)))
    })
    })
```

```
public ResponseEntity<EntityModel<?>>
actualizarUsuario(@PathVariable(name = "id") Long id, @RequestBody Usuario
usuario) {

    Usuario actualizado = usuarioService.actualizarUsuario(id,
usuario);
    if (actualizado == null) {
        ApiErrorModel error = new ApiErrorModel(
            "Usuario no encontrado",
            "No se pudo actualizar el usuario con ID " + id, 400,
            "/api/v1/usuarios/" + id,
            LocalDateTime.now()
        );
        return ResponseEntity.badRequest()
            .contentType(MediaType.HAL_JSON)
            .body(assembler.wrapError(error));
    }
    return ResponseEntity.ok()
        .contentType(MediaType.HAL_JSON)
        .body(assembler.toModel(actualizado));
}

// Borrar usuario por ID
@DeleteMapping(value =("/{id}", produces = {MediaType.HAL_JSON_VALUE,
MediaType.APPLICATION_JSON_VALUE})
@Operation(summary = "Eliminar usuario", description = "Borra el
usuario indicado por ID")
@ApiResponses(value = {
    @ApiResponse(responseCode = "200",
        description = "Usuario eliminado",
        content = @Content(mediaType = "application/json",
            schema = @Schema(implementation = Usuario.class))),
    @ApiResponse(responseCode = "404",
        description = "Usuario no encontrado",
        content = @Content(mediaType = "application/json",
            schema = @Schema(implementation = ApiErrorModel.class)))
})
public ResponseEntity<?> eliminarUsuario(@PathVariable(name = "id")
Long id) {
    boolean eliminado = usuarioService.eliminarUsuario(id);

    if (!eliminado) {
        ApiErrorModel error = new ApiErrorModel(
            "Usuario no encontrado",
```

```
        "No se pudo eliminar el usuario con ID " + id, 404,  
        "/api/v1/usuarios/" + id,  
        LocalDateTime.now()  
    );  
    return ResponseEntity.status(HttpStatus.NOT_FOUND)  
        .contentType(MediaType.HAL_JSON)  
        .body(assembler.wrapError(error));  
    }  
    Map<String, String> mensaje = Map.of("mensaje", "Usuario eliminado  
correctamente");  
    EntityModel<Map<String, String>> respuesta =  
    EntityModel.of(mensaje,  
  
    linkTo(methodOn(UsuarioController.class).obtenerTodosUsuarios()).withRel("usuarios")  
    );  
    return ResponseEntity.ok()  
        .contentType(MediaType.HAL_JSON)  
        .body(respuesta);  
    }  
}
```

Servicios:

- UsuarioService.java: contiene la lógica de negocio para crear, actualizar y recuperar usuarios.

```
@Service  
@Transactional  
public class UsuarioService {  
  
    @Autowired  
    private final UsuarioRepository usuarioRepository;  
  
    public UsuarioService(UsuarioRepository usuarioRepository) {  
        this.usuarioRepository = usuarioRepository;  
    }  
  
    public List<Usuario> obtenerTodosUsuarios() {  
        return usuarioRepository.findAll();  
    }  
  
    public Usuario obtenerPorIdUsuario(Long id) {
```

```
        return usuarioRepository.findById(id).orElse(null);
    }

    public Usuario crearUsuario(Usuario usuario) {
        if (usuario.getId() != null) {
            throw new RuntimeException("El ID debe ser nulo");
        }
        if (usuarioRepository.existsByEmail(usuario.getEmail())){
            throw new IllegalArgumentException("El correo ingresado ya
esta registrado");
        }
        if (!usuario.getPrimerNombre().matches("^[A-Za-zÁÉÍÓÚÑáéíóúñ
]+$")) {
            throw new IllegalArgumentException("El primer nombre solo debe
contener letras");
        }
        if (!usuario.getSegundoNombre().matches("^[A-Za-zÁÉÍÓÚÑáéíóúñ
]+$")) {
            throw new IllegalArgumentException("El segundo nombre solo debe
contener letras");
        }
        if (!usuario.getPrimerApellido().matches("^[A-Za-zÁÉÍÓÚÑáéíóúñ
]+$")) {
            throw new IllegalArgumentException("El primer apellido solo debe
contener letras");
        }
        if (!usuario.getSegundoApellido().matches("^[A-Za-zÁÉÍÓÚÑáéíóúñ
]+$")) {
            throw new IllegalArgumentException("El segundo apellido solo debe
contener letras");
        }
        if
(!usuario.getEmail().matches("^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\\.[A-Za-z-
]{2,}$")) {
            throw new IllegalArgumentException("El correo debe tener un
formato válido");
        }
        return usuarioRepository.save(usuario);
    }

    public Usuario actualizarUsuario(Long id, Usuario datosActualizados) {
        return usuarioRepository.findById(id).map(usuario -> {
            if (datosActualizados.getPrimerNombre() != null) {
```

```
usuario.setPrimerNombre(datosActualizados.getPrimerNombre());
    }
    if (datosActualizados.getSegundoNombre() != null) {

usuario.setSegundoNombre(datosActualizados.getSegundoNombre());
    }
    if (datosActualizados.getPrimerApellido() != null) {

usuario.setPrimerApellido(datosActualizados.getPrimerApellido());
    }
    if (datosActualizados.getSegundoApellido() != null) {

usuario.setSegundoApellido(datosActualizados.getSegundoApellido());
    }
    if (datosActualizados.getEmail() != null) {
        usuario.setEmail(datosActualizados.getEmail());
    }
    if (datosActualizados.getDireccion() != null) {
        usuario.setDireccion(datosActualizados.getDireccion());
    }
    if (datosActualizados.getContrasenia() != null){

usuario.setContrasenia(datosActualizados.getContrasenia());
    }
    return usuarioRepository.save(usuario);
}).orElseThrow(() -> new RuntimeException("Usuario no
encontrado"));
    }

    public boolean eliminarUsuario(Long id) {
Optional<Usuario> usuario = usuarioRepository.findById(id);
    if (usuario.isPresent()) {
        usuarioRepository.deleteById(id);
        return true;
    } else {
        return false;
    }
    }
}
```

Repositorio:

- UsuarioRepository.java: accede a la base de datos usando JPA y define consultas personalizadas si aplica.

```
public interface UsuarioRepository extends JpaRepository<Usuario, Long>{  
  
    boolean existsByEmail(String email);  
  
}
```

Modelos de dominio:

- Usuario.java: define la estructura de datos y restricciones de la entidad Usuario.

```
@Entity  
@Table(name = "usuarios")  
@Data  
@NoArgsConstructor  
@AllArgsConstructor  
public class Usuario {  
  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private Long id;  
  
    @NotBlank(message = "Campo obligatorio")  
    @Column(name = "primer_nombre", nullable = false)  
    private String primerNombre;  
  
    @NotBlank(message = "Campo obligatorio")  
    @Column(name = "segundo_nombre", nullable = false)  
    private String segundoNombre;
```



```
@NotBlank(message = "Campo obligatorio")
@Column(name = "primer_apellido", nullable = false)
private String primerApellido;

@NotBlank(message = "Campo obligatorio")
@Column(name = "segundo_apellido", nullable = false)
private String segundoApellido;

>Email(message = "El correo no tiene un formato valido")
@Pattern(
    regexp = "^[A-Za-z0-9+_.-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,6}$",
    message = "Debe tener el formato nombre@dominio.com")
@NotBlank(message = "Campo obligatorio")
@Column(name = "email", nullable = false, unique = true)
private String email;

@NotBlank(message = "Campo obligatorio")
@Size(min = 6, max = 255, message = "La contraseña debe tener al menos
6 caracteres")
@Column(name = "contrasenia", nullable = false)
private String contrasenia;

@NotBlank(message = "Campo obligatorio")
@Column(name = "direccion", nullable = false, columnDefinition =
"TEXT")
private String direccion;

@Column(name = "fecha_creacion", updatable = false)
private LocalDateTime fechaCreacion;
}
```

HATEOAS (si está implementado):

UsuarioModelAssembler.java: construye respuestas enriquecidas siguiendo el principio HATEOAS.

```
@Component
public class UsuarioModelAssembler implements
RepresentationModelAssembler<Usuario, EntityModel<Usuario>> {

    @Override
```

```
public EntityModel<Usuario> toModel(Usuario usuario) {
    EntityModel<Usuario> model = EntityModel.of(usuario);

    if (usuario.getId() != null) {

model.add(linkTo(methodOn(UsuarioController.class).obtenerUsuarioPorId(usuario.getId())).withSelfRel());

model.add(linkTo(methodOn(UsuarioController.class).obtenerTodosUsuarios()).withRel("usuarios"));

model.add(linkTo(methodOn(UsuarioController.class).actualizarUsuario(usuario.getId(), null)).withRel("actualizar"));

model.add(linkTo(methodOn(UsuarioController.class).eliminarUsuario(usuario.getId())).withRel("eliminar"));
    }

    return model;
}

/**
 * Método auxiliar para envolver mensajes de error dentro de una
 * estructura HATEOAS.
 * Recibe una clave (por ejemplo, "error") y un mensaje explicativo,
 * construyendo
 * un EntityModel que permite incluir enlaces útiles para guiar al
 * cliente en su navegación.
 *
 * Esto facilita que los errores no sólo se presenten como texto
 * plano, sino también
 * como recursos enriquecidos y navegables, manteniendo la coherencia
 * con el diseño RESTful.
 *
 * @param key clave del error (e.g. "error")
 * @param message mensaje explicativo o detalle del fallo
 * @return EntityModel que contiene el cuerpo del error y enlaces
 * relacionados
 */

public EntityModel<ApiErrorModel> wrapError(ApiErrorModel error) {
    return EntityModel.of(error,

linkTo(methodOn(UsuarioController.class).obtenerTodosUsuarios()).withRel("
```

```
usuarios")
    );
}

/**
 * Envoltorio de lista de usuarios con enlaces para navegación.
 * Este método toma una lista de objetos Usuario y la transforma en
una
 * CollectionModel, aplicando HATEOAS a cada elemento mediante el
método toModel(),
 * y añadiendo un enlace al recurso raíz de usuarios.
 *
 * Permite que el cliente consuma una colección estructurada en
formato HAL,
 * donde cada usuario tiene sus propios enlaces (self, actualizar,
eliminar, etc.)
 * y la colección tiene su propio enlace de navegación (self) que
apunta al endpoint completo.
 *
 * @param listaUsuarios lista completa de usuarios obtenida desde el
servicio
 * @return CollectionModel que contiene los EntityModel individuales
con enlaces HATEOAS
 */

public CollectionModel<EntityModel<Usuario>>
toCollection(List<Usuario> listaUsuarios) {
    List<EntityModel<Usuario>> modelos = listaUsuarios.stream()
        .map(this::toModel)
        .collect(Collectors.toList());

    return CollectionModel.of(modelos,

linkTo(methodOn(UsuarioController.class).obtenerTodosUsuarios()).withSelfRel()
    );
}
}
```

Configuraciones:

- application.properties: define parámetros de configuración por entorno (dev, test, prod).

```
spring.application.name=petsociety
spring.profiles.active=test

server.address=0.0.0.0
server.port=8088

springdoc.api-docs.enabled=true
springdoc.swagger-ui.enabled=true
springdoc.swagger-ui.path=/doc/swagger-ui.html
```

Pruebas:

- PetsocietyApplicationTests.java: pruebas de integración usando TestRestTemplate y JUnit.

```
// @SpringBootTest
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
class PetsocietyApplicationTests {
    @Autowired
    private UsuarioController usuarioController;
    @Autowired
    private UsuarioRepository usuarioRepository;

    @LocalServerPort
    private int port;

    @Autowired
```

```
private TestRestTemplate restTemplate;

private final Faker faker = new Faker();

// Limpia la base de datos test si se encuentran datos
@BeforeEach
void limpiarBaseDeDatos() {
    usuarioRepository.deleteAll();
}

// Método privado de usuario reutilizable
private Long crearUsuarioDePrueba(String emailExtra) {
    Usuario nuevo = new Usuario();
    nuevo.setPrimerNombre(faker.name().firstName());
    nuevo.setSegundoNombre(faker.name().firstName());
    nuevo.setPrimerApellido(faker.name().lastName());
    nuevo.setSegundoApellido(faker.name().lastName());
    nuevo.setEmail(faker.internet().emailAddress());
    nuevo.setContraseña(faker.internet().password());
    nuevo.setDireccion(faker.address().fullAddress());

    ResponseEntity<EntityModel<Usuario>> response =
restTemplate.exchange(
    "http://localhost:" + port + "/api/v1/usuarios",
    HttpMethod.POST,
    new HttpEntity<>(nuevo),
    new ParameterizedTypeReference<>() {}
);

    EntityModel<Usuario> entityModel = response.getBody();
    if (entityModel == null) {
        throw new IllegalStateException("La respuesta del
servidor fue nula");
    }

    Usuario contenido = entityModel.getContent();
    if (contenido == null) {
        throw new IllegalStateException("El contenido del
```

```
EntityModel fue nulo");
    }

    return contenido.getId();
}

@Test
@Order(1)
void contextLoads() {
    System.out.println("Testing the context loading...");
    // System.out.println("Server running on port: " + port);
}

@Test
@Order(2)
void contextLoads2() throws Exception {
    System.out.println("Testing the context loading. and the
controller...");
    assertNotNull(usuarioController);
}

@Test
@Order(3)
void usuarioColeccionVacíaTest() throws Exception {
    String response =
this.restTemplate.getForObject("http://localhost:" + port +
"/api/v1/usuarios", String.class);
    // Afirmamos que el cuerpo contiene los enlaces
(self)
    assertNotNull(response).contains("_links");
}

@Test
@Order(4)
void crearUsuarioTest() {
```

```
Usuario usuario = new Usuario();
usuario.setPrimerNombre(faker.name().firstName());
usuario.setSegundoNombre(faker.name().firstName());
usuario.setPrimerApellido(faker.name().lastName());
usuario.setSegundoApellido(faker.name().lastName());
usuario.setEmail("Test4" +
faker.internet().emailAddress());
usuario.setContrasenia(faker.internet().password());
usuario.setDireccion(faker.address().fullAddress());

ResponseEntity<Usuario> response =
restTemplate.postForEntity(
    "http://localhost:" + port + "/api/v1/usuarios",
    usuario,
    Usuario.class);

assertThat(response.getBody().getEmail()).isEqualTo(usuario.getEmail());
assertThat(response.getBody()).isNotNull();
}

@Test
@Order(5)
void obtenerUsuarioTest() throws Exception {
    System.out.println("port: " + port);

    assertThat(this.restTemplate.getForObject("http://localhost:" +
port +
        "/api/v1/usuarios",
        String.class)).toString().contains("Test4");
}

@Test
@Order(6)
void updateUsuarioShouldSucceed() {
    Long id = crearUsuarioDePrueba("Update");

    assertThat(id)
        .withFailMessage("No se encontró el ID del usuario de
```

```
prueba. Asegúrate de que el UsuarioTest se haya ejecutado.")
        .isNull();

        // Creamos un objeto Usuario con nuevos datos para
actualizar
        Usuario actualizado = new Usuario();
        actualizado.setPrimerNombre("NombreActualizado");
        actualizado.setSegundoNombre("SegundoActualizado");
        actualizado.setPrimerApellido("ApellidoActualizado");

actualizado.setSegundoApellido("SegundoApellidoActualizado");
        actualizado.setEmail("actualizado_" + id + "@email.com");
        actualizado.setContrasenia("nuevaContrasenia123");
        actualizado.setDireccion("Nueva Dirección 456");

        // Creamos la solicitud HTTP que lleva el objeto
actualizado en el cuerpo
        HttpEntity<Usuario> request = new
HttpEntity<>(actualizado);

        // Enviamos una solicitud PUT para actualizar el usuario
por ID
        ResponseEntity<Void> response = restTemplate.exchange(
            "http://localhost:" + port + "/api/v1/usuarios/" +
id,
            HttpMethod.PUT,
            request,
            Void.class
        );

        // Verificamos que el status devuelto sea 204 (sin
contenido) o 200 (éxito con contenido)
        assertThat(response.getStatusCode())
            .withFailMessage("Se esperaba un código 204
NO_CONTENT o 200 OK, pero se recibió: %s",
response.getStatusCode())
            .isin(HttpStatus.NO_CONTENT, HttpStatus.OK);

        // Recuperamos el usuario actualizado desde el backend
```



```
        ResponseEntity<Usuario> getResponse =
restTemplate.getForEntity(
            "http://localhost:" + port + "/api/v1/usuarios/" +
id,
            Usuario.class
        );

        // Validamos que el usuario devuelto no sea null
        Usuario usuarioActualizado = getResponse.getBody();
        assertThat(usuarioActualizado).isNotNull();

        // Comprobamos que los cambios realmente se hayan
        aplicado

        assertThat(usuarioActualizado.getPrimerNombre()).isEqualTo("NombreActualizado");

        assertThat(usuarioActualizado.getEmail()).contains("actualizado_");
    }

    @Test
    @Order(7)
    void deleteUsuarioShouldRemoveSuccessfully() {
        Long id = crearUsuarioDePrueba("delete");

        // Enviar la solicitud DELETE para eliminar el usuario
        con ese ID
        ResponseEntity<Void> deleteResponse =
restTemplate.exchange(
            "http://localhost:" + port + "/api/v1/usuarios/" +
id,
            HttpMethod.DELETE,
            null,
            Void.class
        );
    }
```

```
// Verificar que la respuesta sea exitosa: 204 No Content
o 200 OK
    assertThat(deleteResponse.getStatusCode())
        .withFailMessage("Se esperaba un código 204
NO_CONTENT o 200 OK, pero se recibió: %s",
deleteResponse.getStatusCode())
        .isin(HttpStatus.NO_CONTENT, HttpStatus.OK);

    // Intentar recuperar el usuario eliminado (debería
    devolver 404 Not Found)
    ResponseEntity<String> getAfterDelete =
restTemplate.getForEntity(
        "http://localhost:" + port + "/api/v1/usuarios/" +
id,
        String.class
    );

    // Verificar que ya no se puede encontrar el usuario
    assertThat(getAfterDelete.getStatusCode())
        .withFailMessage("Se esperaba que el usuario fuera
eliminado, pero aún se encuentra.")
        .isEqualTo(HttpStatus.NOT_FOUND);
}

}
```

10. Bibliografía

Spring Boot Documentation - <https://spring.io/projects/spring-boot>

Spring Data JPA - <https://spring.io/projects/spring-data-jpa>

SpringDoc OpenAPI - <https://springdoc.org/>

JUnit 5 - <https://junit.org/junit5/>

Mockito - <https://site.mockito.org/>