



TECNOLÓGICO NACIONAL DE MÉXICO
INSTITUTO TECNOLÓGICO DE APIZACO

NOMBRE DEL PROYECTO

GEO TRACE HUB - DESARROLLO DE BACKEND REST PARA APP ANDROID

OPCIÓN QUE CORRESPONDA

TITULACIÓN INTEGRAL INFORME DE RESIDENCIA PROFESIONAL

PARA OBTENER EL TÍTULO PROFESIONAL DE:

INGENIERÍA EN TECNOLOGÍAS DE LA INFORMACIÓN Y COMUNICACIONES

PRESENTA:

SANDRA ABIGAIL CARMONA SÁNCHEZ

NOMBRE DEL ASESOR:

ALICIA CORTÉS FERNÁNDEZ



APIZACO, TLAX., OCTUBRE 2025



Instituto Tecnológico de Apizaco
Sistemas y Computación

Apizaco, Tlaxcala, 30/junio/2025

Oficio No. S.C./OPV/0249/25

ASUNTO: Liberación de proyecto
para titulación integral

ISRAEL JORGE SÁNCHEZ SÁNCHEZ
JEFE DEL DEPTO. DE DIVISIÓN DE ESTUDIOS PROFESIONALES
P R E S E N T E.

POR ESTE MEDIO LE INFORMO QUE HA SIDO LIBERADO EL SIGUIENTE PROYECTO PARA LA TITULACIÓN INTEGRAL.

NOMBRE DEL EGRESADO:	CARMONA SÁNCHEZ SANDRA ABIGAIL
CARRERA:	INGENIERÍA EN TECNOLOGÍAS DE LA INFORMACIÓN Y COMUNICACIONES
NO. DE CONTROL:	20370949
NOMBRE DEL PROYECTO:	Geo Trace Hub - Desarrollo de backend REST para App Android
PRODUCTO U OPCIÓN:	TITULACIÓN INTEGRAL INFORME DE RESIDENCIA PROFESIONAL

AGRADEZCO DE ANTEMANO SU VALIOSO APOYO EN ESTA IMPORTANTE ACTIVIDAD PARA LA FORMACIÓN PROFESIONAL DE NUESTROS EGRESADOS.

ATENTAMENTE

Excelencia en Educación Tecnológica.
Pensar para Servir, Servir para Triunfar®

MIQUELINA SÁNCHEZ PULIDO
JEFA DEL DEPTO. DE SISTEMAS Y COMPUTACIÓN
SECRETARÍA DE EDUCACIÓN PÚBLICA
TECNOLÓGICO NACIONAL
DE MÉXICO
INSTITUTO TECNOLÓGICO DE APIZACO
DEPARTAMENTO DE
SISTEMAS Y COMPUTACIÓN

ccp. Coordinación de carrera
Expediente
MSP/mrol*



ALICIA CORTÉS FERNÁNDEZ
ASESORA INTERNA



2025
Año de
La Mujer
Indígena

Av. Instituto Tecnológico No. 418, San Andrés Ahuashuatepec,
Municipio de Tzompantepec, Tlaxcala, Mex. C.P. 90491
Tel. (241) 4172010 Ext. 115, e-mail: sistemas@apizaco.tecnm.mx
tecnm.mx | apizaco.tecnm.mx





Educación
Secretaría de Educación Pública



TECNOLÓGICO
NACIONAL DE MÉXICO



Instituto Tecnológico de Apizaco
División de Estudios Profesionales

Tzompantepec, Tlaxcala 26/septiembre/2025

CARMONA SÁNCHEZ SANDRA ABIGAIL
EGRESADO(A) DE LA CARRERA DE
INGENIERÍA EN TECNOLOGÍAS DE LA INFORMACIÓN
Y COMUNICACIONES.
CON NÚMERO DE CONTROL 20370949.
P R E S E N T E.

Por este medio, me permito informar a usted que por aprobación de la comisión revisora asignada para valorar el trabajo que mediante la opción **TITULACIÓN INTEGRAL INFORME DE RESIDENCIA PROFESIONAL** y cuyo tema es: **GEO TRACE HUB – DESARROLLO DE BACKEND REST PARA APP ANDROID**, conforme a lo establecido en el procedimiento para la obtención del Título Profesional en el Instituto Tecnológico, la División de Estudios Profesionales a mi cargo le otorga la:

AUTORIZACIÓN DE IMPRESIÓN

Debiendo entregar **2 USB** del mismo, apropiadamente empastados, a la brevedad, para presentar su Acto de Recepción Profesional.

Sin otro particular por el momento me despido.

ATENTAMENTE

Excelencia en Educación Tecnológica®
Pensar para Servir, Servir para Triunfar®

CECILIA YARELY ARREGUÍN LARA
JEFA DE LA DIVISIÓN DE ESTUDIOS PROFESIONALES

ccp. Archivo

CYAL/rrc



SECRETARÍA DE EDUCACIÓN PÚBLICA
TECNOLÓGICO NACIONAL
DE MÉXICO
INSTITUTO TECNOLÓGICO DE APIZACO
DIVISIÓN DE
ESTUDIOS PROFESIONALES



2025
Año de
La Mujer
Indígena

Av. Instituto Tecnológico No. 418, San Andrés Ahuashuatepec,
Municipio de Tzompantepec, Tlaxcala, Mex. C.P. 90491 Tel. (241) 4172010
Ext. 304, e-mail: dep_titulacion@apizaco.tecnm.mx | apizaco.tecnm.mx



Agradecimientos

En este momento de mi vida, quiero reconocer las dádivas de Dios que me ha brindado cada día en la que ha llegado a permitirme alcanzar esta meta, que a lo largo del camino, he enfrentado pruebas, también retos, pero su guía me ha llevado a convertirme en profesionista en tecnología, una pasión que me ha acompañado desde siempre. Como dice el libro de Santiago 1:17, 'Toda buena dádiva y todo don perfecto es de lo alto...', para mí, este logro es una de sus dádivas. Gracias Dios por guiarme en mi carrera, vida profesional y mayormente por estar siempre presente en mi vida desde que nací.

Asimismo quiero expresar mi más profundo agradecimiento a mis padres Quintín Carmona Mejorada y María del Carmen Sánchez Hernández, cuyo amor sumado a su apoyo incondicional han sido la base sobre la cual he construido mi vida académica, igualmente la personal, en el que su sacrificio, paciencia además de su constante motivación me han permitido superar obstáculos, además de alcanzar mis metas, recordando las veces en las que me esperaba mi mamá en mis largas noches de estudio, asimismo, mi papá que me acompañaba a la parada para esperar el transporte temprano, siempre estuvieron allí para ofrecerme su apoyo, en el que su influencia en mi vida ha sido invaluable.

Agradezco también a mis amigos quienes han sido un pilar importante en mi camino universitario, han estado conmigo en momentos alegres también de dificultad, dejando un gran aprendizaje en esta etapa. Su amistad incondicional sumadas a las experiencias que juntos hemos vivido las cuales las llevaré siempre en mi corazón. Espero que nuestra amistad perdure a lo largo del tiempo.

También agradezco a mi asesora académica M.C. Alicia Cortés Fernández, quien ha sido una figura clave en mi desarrollo académico así como profesional, que ha visto en mi potencial en este sector, me ha brindado su apoyo, amistad; otorgándome además la visión de poder superarme para destacar en el sector productivo, en el que su confianza en mí ha sido un gran motivador.

Quiero destacar mi gratitud al equipo de Tech4Grow quienes me han brindado la oportunidad de aprender, así mismo agradezco a Josué Ayala Carro por abrirme las puertas de su empresa para poder desarrollar este proyecto que ha sido fundamental para mi crecimiento profesional. Agradezco al Mtro. Juan Manuel Muñoz Pérez quién me brindó su apoyo, paciencia, sobre todo por su experiencia, de las cuales pude adaptarme a un flujo de trabajo en un entorno real de desarrollo de software.

Índice

I.	Generalidades del proyecto.....	1
1.1.	Introducción.....	1
1.2.	Descripción de la empresa	2
1.3.	Problemas a resolver.....	3
1.4.	Objetivos	4
1.4.1.	Objetivo general.....	4
1.4.2.	Objetivos específicos	4
1.5.	Justificación.....	5
1.6.	Delimitación.....	5
II.	Marco teórico	6
2.1.	Metodologías ágiles.....	6
2.2.	Metodología Scrum	9
2.3.	Kanban	12
2.1.	Arquitectura de software.....	14
2.2.	Java.....	16
2.3.	Maven	18
2.4.	Spring Boot.....	20
2.5.	JPA e Hibernate.....	21
2.6.	Clean Code	22
III.	Desarrollo.....	25
3.1.	Procedimiento y descripción de las actividades realizadas.	25
3.1.1.	Instalación del JDK	25
3.1.2.	Instalación de IntelliJ.....	27
3.1.3.	Uso de GitLab.....	33
3.1.4.	Clonar el repositorio del proyecto en la máquina local	36
3.1.5.	Kanban en GitLab	38

3.1.6.	Análisis	39
3.1.7.	Diseño de la arquitectura del sistema	42
3.1.8.	Esquema de base de datos	64
3.1.9.	Estructura del proyecto	67
3.1.10.	Administración de usuarios	70
3.1.11.	Configuración de Flyway:.....	80
3.1.12.	Manejo de control de errores para la administración de usuarios.....	86
3.1.13.	Subir cambios por tickets	89
3.1.14.	Pruebas unitarias para la administración de usuarios	95
3.1.15.	Administración de dispositivos	106
3.1.16.	Pruebas unitarias en la administración de dispositivos	115
3.1.17.	Pruebas de integración de la administración de dispositivos.....	123
3.1.18.	Administración de ubicaciones.....	141
3.1.19.	Pruebas unitarias de ubicaciones	146
3.1.20.	Pruebas de integración de ubicaciones.....	151
3.1.21.	Filtrado de ubicaciones	160
3.1.22.	Pruebas y correcciones.....	163
3.1.23.	Implementar el estándar OpenAPI en los endpoints desarrollados	233
IV.	Resultados	236
4.1.	Colección POSTMAN de la suite de pruebas	236
4.2.	Manual creado mediante OpenAPI	247
V.	Conclusiones.....	269
5.1.	Conclusiones del proyecto.....	269
5.2.	Recomendaciones	269
5.3.	Experiencia profesional adquirida	270
VI.	Competencias desarrolladas	271
VII.	Fuentes de información	272

VIII.	Anexos	275
8.1.	Glosario	275
8.2.	Presentación mostrada en la empresa	277

Índice de Figuras

Figura 1	Página oficial de Oracle.....	25
Figura 2	Listado de JDK instalables.	25
Figura 3	Instalador de JDK.....	26
Figura 4	Finalización de la instalación de Java 21.....	26
Figura 5	Versión de Java en el CMD.	27
Figura 6	Página oficial de IntelliJ IDEA.....	27
Figura 7	Página de agradecimiento por la descarga de IntelliJ IDEA.	28
Figura 8	Instalador de IntelliJ IDEA.	28
Figura 9	Configuración de IntelliJ IDEA.	29
Figura 10	Acceso directo para IntelliJ IDEA.....	29
Figura 11	Finalización de la instalación de IntelliJ IDEA.	30
Figura 12	Ejecución de IntelliJ IDEA.	30
Figura 13	Creación de un nuevo proyecto en IntelliJ IDEA.....	31
Figura 14	Entorno de IntelliJ IDEA.	31
Figura 15	Sección de Plugins en IntelliJ IDEA.....	32
Figura 16	Plugin Sonarlint.	32
Figura 17	Invitación a GitLab por email.	33
Figura 18	Inicio de Sesión en GitLab.....	33
Figura 19	Formulario de registro en GitLab.	34
Figura 20	Panel de GitLab.....	34
Figura 21	Sección de edición del perfil en GitLab.....	35
Figura 22	Sección de SSH Keys.	35
Figura 23	Repositorio del proyecto en GitLab.....	36
Figura 24	Proyecto almacenado de forma local.....	36
Figura 25	Estructura de carpetas del proyecto en IntelliJ IDEA.	37
Figura 26	Tableros Kanban en GitLab.	38
Figura 27	Tickets en los tableros en GitLab.....	38

Figura 28	Vista lógica mediante diagrama de secuencias UML.....	42
Figura 29	Vista de desarrollo mediante diagrama de componentes UML.	43
Figura 30	Vista física mediante diagrama de despliegue UML.	45
Figura 31	Vista de procesos para crear una cuenta.	46
Figura 32	Vista de procesos para obtener las cuentas.	47
Figura 33	Vista de procesos para obtener la información de un usuario.....	48
Figura 34	Vista de procesos para actualizar un usuario.	49
Figura 35	Vista de procesos para cambiar el estado de un usuario.....	50
Figura 36	Vista de procesos para eliminar a un usuario del sistema.	51
Figura 37	Vista de procesos para añadir un nuevo dispositivo al sistema.	52
Figura 38	Vista de procesos en el que se visualizan los dispositivos de un solo usuario.....	53
Figura 39	Vista de procesos para obtener todos los dispositivos.....	54
Figura 40	Vista de procesos que muestra la búsqueda de un dispositivo por ID.	55
Figura 41	Vista de procesos en el que se actualizan detalles de un dispositivo.	56
Figura 42	Vista de procesos que muestra el flujo de desactivar el estado de un dispositivo....	57
Figura 43	Vista de procesos que plasma la eliminación de un dispositivo.	58
Figura 44	Vista de procesos que muestra todas las ubicaciones de un dispositivo.	59
Figura 45	Vista de procesos que muestra como el sistema almacena una nueva posición.	60
Figura 46	Vista de procesos que muestra el filtrado de ubicaciones.....	61
Figura 47	Vista de escenarios mediante diagrama de casos de uso.	62
Figura 48	Stack Tecnológico del proyecto.	63
Figura 49	Esquema relacional de la base de datos.	64
Figura 50	Tabla "users".	64
Figura 51	Tabla "devices".	65
Figura 52	Tabla "positions".	66
Figura 53	Tabla generada por Flyway.....	66
Figura 54	Paquetes y archivos cruciales del proyecto.	67
Figura 55	Declaración del nombre de la aplicación en el archivo .yaml.	68
Figura 56	Configuración de la conexión a la base de datos en el archivo .yaml.	68
Figura 57	Configuración de JPA e Hibernate en el archivo .yaml.....	69
Figura 58	Activación de la depuración en el archivo .yaml.....	69
Figura 59	Modelo User.	70
Figura 60	Dependencia validation en el pom.xml.	70
Figura 61	Declaración de los atributos para la tabla users en el modelo User.	71

Figura 62	Interfaz UserRepository.	72
Figura 63	Interfaz UserService.	72
Figura 64	Clase UserServiceImpl.	73
Figura 65	Método getAllUsers.	73
Figura 66	Método userStatus.	74
Figura 67	Método getUserById.	74
Figura 68	Método saveUser.	75
Figura 69	Método updateUser.	75
Figura 70	Método deleteUser.	76
Figura 71	Clase UserController.	77
Figura 72	Método getAllUsers del controlador.	77
Figura 73	Método getUserById del controlador.	78
Figura 74	Método saveUser del controlador.	78
Figura 75	Método updateUser del controlador.	79
Figura 76	Método userStatus del controlador.	79
Figura 77	Método deleteUser del controlador.	79
Figura 78	Dependencias de Flyway en el archivo pom.xml.	80
Figura 79	Configuración de Flyway en el archivo .yaml.	80
Figura 80	Configuración del conflicto entre JPA y Flyway.	81
Figura 81	Directorio para las migraciones con Flyway.	81
Figura 82	Migraciones creadas dentro del directorio.	82
Figura 83	Estructura de la creación de la tabla users.	82
Figura 84	Estructura de la creación de la tabla devices y positions.	83
Figura 85	Estructura de intercesión de datos en SQL.	83
Figura 86	Base de datos en Workbench con sus tablas.	84
Figura 87	Migraciones almacenadas en la tabla Flyway_schema_history.	84
Figura 88	Estructura de la tercera migración.	85
Figura 89	Estructura de la cuarta migración.	85
Figura 90	Creación del paquete exceptions.	86
Figura 91	Clase ValidationExceptionHandler.	86
Figura 92	Método handleValidationExceptions.	87
Figura 93	Método handleUserAlreadyExistsException.	87
Figura 94	Clase ValidateException.	88
Figura 95	Lanzamiento de la excepción en el método saveUser del service implements.	88

Figura 96	Creación de una rama.....	89
Figura 97	Asignación del nombre de la rama.....	89
Figura 98	Archivos con los cambios realizados.....	89
Figura 99	Selección de los archivos para crear un commit.....	90
Figura 100	Vista previa de los archivos con commit.....	90
Figura 101	Mensaje para crear el merge request.....	91
Figura 102	Creación del merge request.....	91
Figura 103	Merge request enlazado al ticket en GitLab.....	92
Figura 104	Ejemplo de observaciones a modificar.....	92
Figura 105	Fusión de las ramas en el repositorio.....	93
Figura 106	Cambio a la rama principal.....	93
Figura 107	Actualización de la rama principal.....	93
Figura 108	Mensaje de actualización de los cambios.....	94
Figura 109	Página oficial de MVN REPOSITORY.....	95
Figura 110	Versiones de Mockito.....	95
Figura 111	Versión de Mockito 1.10.19.....	96
Figura 112	Dependencia de Mockito en Maven.....	96
Figura 113	Dependencia de Mockito en el pom.xml.....	97
Figura 114	Clase UserServiceImpl para pruebas unitarias.....	97
Figura 115	Selección de los métodos a ejecutar en pruebas unitarias.....	98
Figura 116	Creación de la clase UserServiceImplTest.....	98
Figura 117	Declaración de los atributos.....	99
Figura 118	Declaración del mock a usar.....	99
Figura 119	Declaración de la inyección del mock.....	99
Figura 120	Configuración del método setUp.....	100
Figura 121	Método getAllUsers del test unitario.....	100
Figura 122	Acción de ejecución del método.....	101
Figura 123	Ejecución del test del método getAllUsers.....	101
Figura 124	Prueba aprobada del método getAllUsers.....	101
Figura 125	Método getUserById del test unitario.....	102
Figura 126	Ejecución del test del método getUserById.....	102
Figura 127	Método saveUser del test unitario.....	103
Figura 128	Ejecución del test del método saveUser.....	103
Figura 129	Método updateUser del test unitario.....	104

Figura 130	Ejecución del test del método updateUser.....	104
Figura 131	Método deleteUser del test unitario.	105
Figura 132	Ejecución del test del método deleteUser.	105
Figura 133	Clase Device.	106
Figura 134	Interfaz DispositiveRepository.	107
Figura 135	Interfaz DeviceService.....	108
Figura 136	Métodos getAllDevices y deviceStatus en DeviceServiceImpl.	108
Figura 137	Método getDeviceById.	109
Figura 138	Método saveDevice.	109
Figura 139	Método updateDevice.....	110
Figura 140	Método deleteDevice.....	110
Figura 141	Método getDevicesByUserId.	111
Figura 142	Clase DeviceController.	111
Figura 143	Método getAllDevices del controlador.....	112
Figura 144	Método getDeviceById del controlador.	112
Figura 145	Método saveDevice del controlador.....	113
Figura 146	Método updateDevice del controlador.	113
Figura 147	Método deviceStatus del controlador.....	114
Figura 148	Método deleteDevice del controlador.....	114
Figura 149	Método getDevicesByUser del controlador.	114
Figura 150	Clase devicesServicesImpl para crear la prueba unitaria.	115
Figura 151	Selección de los métodos de la clase DeviceServiceImpl.....	115
Figura 152	Declaración de los atributos en la clase DeviceServiceImplTest.....	116
Figura 153	Método getAllDevices del test unitario.....	116
Figura 154	Ejecución del test del método getAllDevices.	117
Figura 155	Método getDeviceById del test unitario.....	117
Figura 156	Ejecución del test del método getDeviceById.	118
Figura 157	Método saveDevice del test unitario.	118
Figura 158	Ejecución del test del método saveDevice.....	119
Figura 159	Método updatedDevice del test unitario.....	119
Figura 160	Ejecución del test del método updateDevice.	120
Figura 161	Método deleteDevice del test unitario.....	120
Figura 162	Ejecución del test del método deleteDevice.	121
Figura 163	Método getDevicesByUserId del test unitario.	121

Figura 164	Ejecución del test del método <code>getDevicesByUserId</code> .	122
Figura 165	Actualización de la dependencia <code>starter-parent</code> .	123
Figura 166	Adición del <code>surefire.version</code> en el <code>properties</code> .	123
Figura 167	Plugin <code>Maven-surefire-plugin</code> para eliminar los warnings.	124
Figura 168	Uso de <code>DeviceController</code> para las pruebas de integración.	124
Figura 169	Menú de opciones "Generate".	125
Figura 170	Selección de todos los métodos del controlador de dispositivos para el test de integración.	125
Figura 171	Clase <code>DeviceControllerIntegrationTest</code> para test de integración de dispositivos.	126
Figura 172	Uso de las anotaciones <code>@ExtendWith</code> , <code>@WebMvcTest</code> y <code>@AutoConfigureMockMvc</code> en la clase <code>DeviceControllerIntegrationTest</code> .	127
Figura 173	Declaración de las variables con <code>MockMvc</code> .	127
Figura 174	Configuración del método <code>setUp</code> en la prueba de integración de dispositivos.	128
Figura 175	Método <code>getAllDevices_ShouldRequireUser</code> del test de integración.	128
Figura 176	Resultado del método <code>getAllDevices_ShouldRequireUser</code> .	129
Figura 177	Método <code>getDeviceById_ShouldReturnDevice</code> del test de integración.	130
Figura 178	Configuración de la solicitud en el método <code>getDeviceById_ShouldReturnDevice</code> .	130
Figura 179	Ejecución del test del método <code>getDeviceById_ShouldReturnDevice</code> .	131
Figura 180	Método <code>saveDevice_ShouldReturnSavedDevice</code> .	132
Figura 181	Configuración de la solicitud para <code>saveDevice_ShouldReturnSavedDevice</code> .	133
Figura 182	Ejecución del test del método <code>saveDevice_ShouldReturnSavedDevice</code> .	133
Figura 183	Método <code>updateDevice_ShouldReturnUpdatedDevice</code> .	134
Figura 184	Configuración de la solicitud para <code>updateDevice_ShouldReturnUpdatedDevice</code> .	135
Figura 185	Ejecución del test del método <code>updateDevice_ShouldReturnUpdatedDevice</code> .	135
Figura 186	Método <code>updateDeviceStatus_ShouldReturnSuccess</code> .	136
Figura 187	Ejecución del test del método <code>updateDeviceStatus_ShouldReturnSuccess</code> .	137
Figura 188	Método <code>deleteDevice_ShouldReturnNoContent</code> .	137
Figura 189	Ejecución del test del método <code>deleteDevice_ShouldReturnNoContent</code> .	138
Figura 190	Método <code>getDevicesByUser_ShouldReturnUserDevices</code> .	138
Figura 191	Configuración en <code>getDevicesByUser_ShouldReturnUserDevices</code> .	139
Figura 192	Configuración de la solicitud en <code>getDevicesByUser_ShouldReturnUserDevices</code> .	139
Figura 193	Ejecución del test del método <code>getDevicesByUser_ShouldReturnUserDevices</code> .	140
Figura 194	Clase <code>Position</code> .	141
Figura 195	Clase <code>PointConverter</code> .	142

Figura 196	Interfaz PositionsRepository.	142
Figura 197	Interfaz PositionsService.	143
Figura 198	Métodos getPointsByDevice y storePosition.....	143
Figura 199	Clase PositionsController.	144
Figura 200	Método getPositions del controlador.....	144
Figura 201	Método savePosition del controlador.	145
Figura 202	Selección de los métodos en la clase PositionsServiceImpl.	146
Figura 203	Creación de la clase PositionsServiceImplTest para las pruebas unitarias.	146
Figura 204	Configuración del método setUp en la prueba unitarias de ubicaciones.	147
Figura 205	Método getPointsByDevice del test unitario.....	147
Figura 206	Al método getPointsByDevice de la clase positionsServiceImpl se le agrega el log.info.	148
Figura 207	Ejecución del test del método getPointsByDevice.	148
Figura 208	Método storePosition del test unitario.	149
Figura 209	Al método storePosition se le agrega el log.info.	149
Figura 210	Ejecución del test del método storePosition.	150
Figura 211	Clase PositionsController para la prueba de integración de ubicaciones.	151
Figura 212	Selección de los métodos para la prueba de integración.....	151
Figura 213	Clase PositionsControllerIntegrationTest.	152
Figura 214	Creación de las clases JacksonConfig y PointJsonSerializers.....	152
Figura 215	Clase PointJsonSerializers.....	153
Figura 216	Clase JacksonConfig.....	153
Figura 217	Anotaciones @ExtendWith, @WebMvcTest y @AutoConfigureMockMvc en la clase PositionsControllerIntegrationTest.....	154
Figura 218	Declaración de las variables con MockMvc en la clase del test.	154
Figura 219	Configuración del método setUp en la prueba de integración de ubicaciones.	155
Figura 220	Método getPositions_ShouldReturnPositionsList del test de integración.	155
Figura 221	Configuración de la solicitud para getPositions_ShouldReturnPositionsList.	156
Figura 222	Ejecución del test del método getPositions_ShouldReturnPositionsList.	157
Figura 223	Método savePosition_ShouldReturnSavedPosition del test de integración.....	157
Figura 224	Configuración de la solicitud para savePosition_ShouldReturnSavedPosition.....	158
Figura 225	Ejecución del test del método savePosition_ShouldReturnSavedPosition.....	159
Figura 226	Atributo createdAt en la clase Position.....	160
Figura 227	Método findByDeviceAndCreatedAtBetween en la clase PositionsRepository.	161

Figura 228	Método <code>getPositionsByDeviceAndDateRange</code> en la clase <code>PositionsService</code>	161
Figura 229	Método <code>getPositionsByDeviceAndDateRange</code> en <code>PositionsServiceImpl</code>	162
Figura 230	Se agrega el método <code>getFilterPositions</code> en la clase <code>PositionsController</code>	162
Figura 231	Creación de dos dispositivos para la <code>PositionsServiceImplTest</code>	163
Figura 232	Método <code>getPositionsByDeviceAndDateRange</code> del test unitario.	163
Figura 233	Método <code>getPositionsByDeviceAndDateRange</code> se añade un <code>info.log</code>	164
Figura 234	Ejecución del test del método <code>getPositionsByDeviceAndDateRange</code>	164
Figura 235	Dependencia <code>jackson.datatype-jsr310</code> en el <code>pom.xml</code>	165
Figura 236	Configuración del método <code>setUp</code> en la clase <code>PositionsControllerIntegrationTest</code> . .	165
Figura 237	Método <code>getFilteredPositions_ShouldReturnFilteredList</code> del test de integración. ...	166
Figura 238	Se configura el guardado de las ubicaciones.	167
Figura 239	Configuración de la solicitud en <code>getFilteredPositions_ShouldReturnFilteredList</code> . .	167
Figura 240	Ejecución del test del método <code>getFilteredPositions_ShouldReturnFilteredList</code>	168
Figura 241	Colección de POSTMAN de los endpoints creados.	169
Figura 242	Respuesta errónea del criterio 1.1 de listar los dispositivos.	171
Figura 243	Se añade la anotación <code>@JsonIgnore</code> en la clase <code>User</code>	171
Figura 244	Respuesta 200 OK del criterio 1.1 de listar los dispositivos.	172
Figura 245	Resultado 200 OK del criterio 2 de listar los dispositivos.	172
Figura 246	Resultado criterio 1.1 de ver usuario mediante su ID.	173
Figura 247	Resultado erróneo del criterio 2 de ver usuario mediante su ID.	174
Figura 248	Creación de la clase <code>UserNotFoundException</code>	174
Figura 249	Método <code>handleUserNotFoundException</code> en <code>ValidationExceptionHandler</code>	175
Figura 250	Se llama la excepción en <code>getUserById</code> de la clase <code>UserServiceImpl</code>	175
Figura 251	Resultado del criterio 2 de ver usuario mediante su ID.	176
Figura 252	Resultado del criterio 1.1 del registro de usuario.	178
Figura 253	Resultado del criterio 1.2 del registro de usuario.	178
Figura 254	Resultado del criterio 1.3 del registro de usuario.	179
Figura 255	Ejecución nuevamente del criterio 1.3 del registro de usuario.	179
Figura 256	Ejecución del criterio 1.3 con otro parámetro en el registro de usuario.	180
Figura 257	Resultado del criterio 1.4 del registro de usuario.	180
Figura 258	Resultado del criterio 2 del registro de usuario.	181
Figura 259	Resultado del criterio 3 del registro de usuario.	181
Figura 260	Resultado del criterio 1 de la edición de usuarios.	183
Figura 261	Resultado del criterio 2 de la edición de usuarios.	183

Figura 262	Resultado de error del criterio 3 de la edición de usuarios.	184
Figura 263	En el método updateUser se llama la excepción UserNotFoundException.	184
Figura 264	Resultado del criterio 3 de la edición de usuarios.	185
Figura 265	Resultado de error del criterio 4 de la edición de usuarios.	185
Figura 266	Clase UpdateUserDTO	186
Figura 267	Método updateUser invoca la clase DTO.	186
Figura 268	La clase UserServiceImpl usa la clase DTO	187
Figura 269	El método updateUser usa la clase DTO en la clase UserService.	187
Figura 270	Resultado del criterio 4 de la edición de usuarios en el que el nombre de usuario no cumple con el criterio de longitud.	188
Figura 271	Resultado del criterio 4 de la edición de usuarios en el que el nombre de usuario no cumple los criterios de al menos un número en el nombre de usuario.	188
Figura 272	Resultado del criterio 1.1 para cambiar estado activo de la cuenta de usuario. ...	189
Figura 273	Cambio de estado en falso de la cuenta de usuario.	190
Figura 274	Resultado de cambiar el estado en activo.	190
Figura 275	Cambio de estado en verdadero de la cuenta de usuario.	191
Figura 276	Resultado del criterio 2 cambiar estado activo de la cuenta de usuario.	191
Figura 277	Resultado del criterio 1.1 de la eliminación de un usuario.	192
Figura 278	Resultado erróneo del criterio 2 de la eliminación de un usuario.	193
Figura 279	Se llama la excepción UserNotFoundException en el método deleteUser.	193
Figura 280	Resultado del criterio 2 de la eliminación de un usuario.	194
Figura 281	Resultado erróneo del criterio 3 de la eliminación de un usuario.	194
Figura 282	Clase ConflictException.	195
Figura 283	Método handleConflictException en ValidationExceptionHandler.	195
Figura 284	Uso de las excepciones UserNotFoundException y ConflictException en el método deleteUser.	196
Figura 285	Resultado del criterio 3 de la eliminación de un usuario.	196
Figura 286	Resultado del criterio 1 para llamar a todos los dispositivos.	197
Figura 287	Resultado erróneo del criterio 1 para listar un dispositivo mediante su ID.	198
Figura 288	Se añade la anotación @JsonIgnore en la clase Device.	199
Figura 289	Resultado del criterio 1 para listar un dispositivo mediante su ID.	199
Figura 290	Resultado del criterio 1.1 del registro de un dispositivo.	200
Figura 291	Resultado del criterio 2 del registro de un dispositivo en el que faltaron datos para el registro.	201

Figura 292	Uso de la excepción ValidateException en el método saveDevice.....	201
Figura 293	En el criterio 2 se omite el nombre del dispositivo y muestra una advertencia.....	202
Figura 294	En el criterio 2 se omite el nombre del dispositivo y se muestra un mensaje.....	202
Figura 295	Resultado del criterio 2 en el que se muestran los mensajes correspondientes.	203
Figura 296	Resultado del criterio 1.1 de la actualización de información de un dispositivo.....	204
Figura 297	Resultado erróneo del criterio 2 de la actualización de un dispositivo.	205
Figura 298	Clase DeviceNotFoundException.	205
Figura 299	Método handleDeviceNotFoundException en ValidationExceptionHandler.	206
Figura 300	Se llama la excepción en el método updateDevice.....	206
Figura 301	Resultado del criterio 2 de la actualización de información de un dispositivo.	207
Figura 302	Resultado no esperado del criterio 3 de la actualización de un dispositivo.	207
Figura 303	Se agregan validaciones en el método updatedDevice.	208
Figura 304	Se añaden todos los datos nuevos en la actualización.....	208
Figura 305	Resultado del criterio 3 de la actualización de información de un dispositivo.	209
Figura 306	Resultado del criterio 1.1 para cambiar estado de un dispositivo210	
Figura 307	Listado de la información de un dispositivo con el estado en falso.	211
Figura 308	Activación de un dispositivo en estado verdadero.	211
Figura 309	Listado de la información de un dispositivo con el estado en verdadero.....	212
Figura 310	Resultado erróneo del criterio 2 de cambiar estado de un dispositivo.	212
Figura 311	Uso de DeviceNotFound en deviceStatus de la clase DeviceServiceImpl.....	213
Figura 312	Resultado del criterio 2 en cambiar estado de un dispositivo no existente.....	213
Figura 313	Resultado del criterio 1.1 de la eliminación de un dispositivo.	214
Figura 314	Resultado erróneo del criterio 2 de la eliminación de un dispositivo.	215
Figura 315	Uso de DeviceNotFoundException en el método deleteDevice.	215
Figura 316	Resultado del criterio 2 de la eliminación de un dispositivo.	216
Figura 317	Resultado erróneo del criterio 3 en la eliminación de un dispositivo.	216
Figura 318	Modificación del método deleteDevice de la clase DeviceServiceImpl.	217
Figura 319	Resultado del criterio 3 de la eliminación de un dispositivo.	217
Figura 320	Resultado del criterio 1 en mostrar los dispositivos que tiene un usuario.	219
Figura 321	Resultado inesperado del criterio 2 en la mostrar los dispositivos de un usuario.	219
Figura 322	Se llama a DeviceNotFoundException al método getDevicesByUserId.	220
Figura 323	Resultado del criterio 2 en mostrar los dispositivos que tiene un usuario.	220
Figura 324	Resultado del criterio 3 para mostrar los dispositivos que tiene un usuario.	221
Figura 325	Resultado del criterio 4 para mostrar los dispositivos que tiene un usuario.	221

Figura 326	Resultado del criterio 1 para invocar todas las ubicaciones de un dispositivo.	223
Figura 327	Resultado inesperado del criterio 2 para obtener todas las ubicaciones almacenadas en un dispositivo.	223
Figura 328	En getPointByDevice usa la excepción DeviceNotFoundException.	224
Figura 329	Resultado del criterio 2 para invocar todas las ubicaciones de un dispositivo.	224
Figura 330	Resultado del criterio 3 para invocar todas las ubicaciones de un dispositivo.	225
Figura 331	Resultado del criterio 1 para almacenar ubicaciones.	227
Figura 332	Resultado erróneo del criterio 2 para almacenar ubicaciones.	227
Figura 333	Se usa DeviceNotFoundException en el método storePosition.	228
Figura 334	Resultado del criterio 2 para almacenar ubicaciones.	228
Figura 335	Resultado erróneo del criterio 3 para almacenar ubicaciones.	229
Figura 336	Validaciones en el método storePosition de la clase PositionsServiceImpl.	229
Figura 337	Ejecución del criterio 3 para el almacenado de ubicaciones.	230
Figura 338	Resultado del criterio 3 para almacenar ubicaciones.	230
Figura 339	Resultado del criterio 1 para el filtrado de ubicaciones.	231
Figura 340	Listado de ubicaciones en un periodo establecido.	232
Figura 341	Resultado del criterio 2 para el filtrado de ubicaciones.	232
Figura 342	Dependencia OpenAPI en el pom.xml.	233
Figura 343	Clase OpenAPIConfig.	233
Figura 344	Configuración de OpenAPI en el archivo .yaml.	234
Figura 345	Interfaz de OpenAPI.	234
Figura 346	OpenAPI en formato JSON.	235
Figura 347	Introducción a la colección de POSTMAN.	236
Figura 348	Colección de endpoints de usuarios.	236
Figura 349	Endpoint registro de usuarios.	237
Figura 350	Endpoint obtener usuarios por su ID.	237
Figura 351	Endpoint actualizar a un usuario.	238
Figura 352	Endpoint desactivar cuenta de usuario.	238
Figura 353	Endpoint activar cuenta de usuario.	239
Figura 354	Endpoint eliminar usuario.	239
Figura 355	Colección de endpoints de dispositivos.	240
Figura 356	Endpoint llamar dispositivos.	240
Figura 357	Endpoint obtener dispositivo por ID.	241
Figura 358	Endpoint llamar dispositivos por usuario.	241

Figura 359	Endpoint guardar dispositivo.	242
Figura 360	Endpoint actualizar dispositivo.	242
Figura 361	Endpoint desactivar dispositivo.	243
Figura 362	Endpoint activar dispositivo.	243
Figura 363	Endpoint eliminar dispositivo.	244
Figura 364	Colección de endpoints de posiciones.	244
Figura 365	Endpoint obtener todas las posiciones de un dispositivo.	245
Figura 366	Endpoint guardar posición.	245
Figura 367	Endpoint filtrar posición por fecha.	246
Figura 368	Documentación de OpenAPI.	247
Figura 369	Endpoints de ubicaciones en OpenAPI.	248
Figura 370	Primer endpoint de ubicaciones.	248
Figura 371	Resultado del primer endpoint de ubicaciones.	249
Figura 372	Segundo endpoint de ubicaciones.	250
Figura 373	Resultado del segundo endpoint de ubicaciones.	250
Figura 374	Tercer endpoint de ubicaciones.	251
Figura 375	Resultado del tercer endpoint de ubicaciones.	251
Figura 376	Endpoints de dispositivos en OpenAPI.	252
Figura 377	Primer endpoint de dispositivos.	252
Figura 378	Resultado del primer endpoint de dispositivos.	253
Figura 379	Segundo endpoint de dispositivos.	253
Figura 380	Resultado del segundo endpoint de dispositivos.	254
Figura 381	Tercer endpoint de dispositivos.	254
Figura 382	Resultado del tercer endpoint de dispositivos.	255
Figura 383	Cuarto endpoint de dispositivos.	255
Figura 384	Resultado del cuarto endpoint de dispositivos.	256
Figura 385	Quinto endpoint de dispositivos.	256
Figura 386	Resultado del quinto endpoint de dispositivos.	257
Figura 387	Sexto endpoint de dispositivos.	257
Figura 388	Resultado del sexto endpoint de dispositivos.	258
Figura 389	Séptimo endpoint de dispositivos.	258
Figura 390	Resultado del séptimo endpoint de dispositivos.	259
Figura 391	Colección de endpoints de usuarios.	259
Figura 392	Primer endpoint de usuarios.	260

Figura 393	Resultado del primer endpoint de usuarios.....	260
Figura 394	Segundo endpoint de usuarios.	261
Figura 395	Resultado del segundo endpoint de usuarios	261
Figura 396	Tercer endpoint de usuarios.	262
Figura 397	Resultado del tercer endpoint de usuarios.....	262
Figura 398	Cuarto endpoint de usuarios.....	263
Figura 399	Resultado del tercer endpoint de usuarios.....	263
Figura 400	Quinto endpoint de usuarios.....	264
Figura 401	Resultado del quinto endpoint de usuarios.....	264
Figura 402	Sexto endpoint de usuarios.	265
Figura 403	Resultado del sexto endpoint de usuarios.	265
Figura 404	Esquemas creados en el proyecto.....	266
Figura 405	Esquema UpdateUserDTO	266
Figura 406	Esquema User.....	267
Figura 407	Esquema Device.	267
Figura 408	Esquema Point.	268
Figura 409	Esquema Position.	268
Figura 410	Portada de la presentación del proyecto.....	277
Figura 411	Sección de antecedentes en la presentación a la empresa.	277
Figura 412	Sección del objetivo general del proyecto en la presentación a la empresa.	278
Figura 413	Sección de objetivos específicos en la presentación a la empresa.....	278
Figura 414	Continuación de los objetivos específicos en la presentación a la empresa.	279
Figura 415	Sección de alcances del proyecto en la presentación a la empresa.	279
Figura 416	Sección de fuera de alcance en la presentación a la empresa.	280
Figura 417	Sección de metodología de trabajo en la presentación a la empresa.	280
Figura 418	Sección de tecnologías utilizadas en la presentación a la empresa.....	281
Figura 419	Sección de resultados en la presentación a la empresa.	281
Figura 420	Sección de conclusiones en la presentación a la empresa.	282
Figura 421	Sección de agradecimientos en la presentación a la empresa.	282

Índice de Tablas

Tabla 1	Requerimientos funcionales del backend.	39
Tabla 2	Requerimientos no funcionales del backend.	41
Tabla 3	Estructura de la petición "Listar usuarios"	170
Tabla 4	Estructura de la petición "ver usuario mediante su ID".	173
Tabla 5	Estructura de la petición "registro de usuario"	177
Tabla 6	Estructura de la petición "edición de usuarios"	182
Tabla 7	Estructura de la petición "cambiar estado activo de la cuenta de usuario"	189
Tabla 8	Estructura de la petición "eliminación de un usuario".	192
Tabla 9	Estructura de la petición para "llamar a todos los dispositivos"	197
Tabla 10	Estructura de la petición "listar un dispositivo mediante su ID".	198
Tabla 11	Estructura de la petición "registro de un dispositivo"	200
Tabla 12	Estructura de la petición "actualización de información de un dispositivo"	204
Tabla 13	Estructura de la petición "cambiar estado de un dispositivo".	210
Tabla 14	Estructura de la petición "eliminación de un dispositivo"	214
Tabla 15	Estructura de la petición "mostrar los dispositivos que tiene un usuario".	218
Tabla 16	Estructura de la petición "invocar todas las ubicaciones de un dispositivo".	222
Tabla 17	Estructura de la petición "almacenar ubicación".	226
Tabla 18	Estructura de la petición "filtrado de ubicaciones".	231

Resumen

Este proyecto nace a la raíz de contar con un sistema independiente para gestionar ubicaciones geográficas de dispositivos móviles, debido a los cambios en el historial de ubicaciones de Google. Además, muchas aplicaciones utilizan bases de datos no relacionales como Firebase, que con el tiempo resultan poco eficientes para estructuras complejas. Por ello, se planteó desarrollar un backend robusto, escalable como comprensible que ofreciera control total sobre los datos.

El objetivo principal fue desarrollar un backend REST utilizando el framework Spring Boot que permita recibir, almacenar, además de consultar posiciones geográficas en una base de datos MySQL, optimizando la gestión de usuarios, dispositivos así como ubicaciones. Se emplearon tecnologías como Java, Spring Boot, JPA, Hibernate junto con Flyway. La arquitectura del sistema se modeló con el enfoque “4+1” de Kruchten, aplicando principios de Clean Code. Se creó una suite de pruebas que incluían pruebas unitarias, de integración asimismo como de aceptación. La planificación se organizó con metodología Kanban en GitLab bajo la dirección de la metodología Scrum.

El backend desarrollado permite registrar usuarios, dispositivos; también almacenar sus ubicaciones y consultarlas con filtros. Se implementó el estándar OpenAPI para la documentación de los endpoints, además de crear una colección de POSTMAN para validación funcional. El sistema desarrollado cumple con los objetivos planteados, lista para integrarse con aplicaciones móviles o interfaces web permitiendo una gestión eficiente de datos geoespaciales, en el que representa una alternativa viable a herramientas comerciales, brindando autonomía, así como escalabilidad para futuros desarrollos tecnológicos.

I. Generalidades del proyecto

1.1. Introducción

En la era de la transformación digital en la que la sociedad está inmersa día a día, la gestión de datos geoespaciales se ha vuelto un componente esencial para múltiples aplicaciones móviles y sistemas empresariales, con el cual el seguimiento de ubicaciones, en particular, ha cobrado gran relevancia en sectores como la logística, la seguridad, el monitoreo de activos, así como en la experiencia de usuario en tiempo real. Esto permite rastrear, así también analizar el recorrido de los pasos de los usuarios, para tomar decisiones informadas en base al historial registrado. Sin embargo, debido a los recientes cambios en la gestión del historial de ubicaciones de Google, muchas aplicaciones que dependían de este servicio se han visto afectadas.

Ahora, la información se almacena directamente en los dispositivos, eliminándose automáticamente en periodos establecidos. Además, el uso extendido de bases de datos no relacionales como Firebase en la construcción de apps móviles, si bien es útil en ciertos escenarios, pero representa desafíos para la estructuración, e igualmente la escalabilidad de soluciones que requieren consultas complejas como relaciones entre entidades. Ante esta problemática, surge la necesidad de construir un sistema propio que permita a los usuarios gestionar la información geográfica de manera independiente, así como eficiente.

El propósito del proyecto es desarrollar un backend REST utilizando el framework Spring Boot, que permita recibir, almacenar, también consultar posiciones geográficas de dispositivos móviles en una base de datos relacional MySQL. Esta solución busca ofrecer una alternativa robusta, escalable; que esté bien documentada, facilitando su integración con aplicaciones móviles así mismo con aplicaciones web.

1.2. Descripción de la empresa

Nombre: Agilidad Tecnológica para el Crecimiento Corporativo S.A. de C.V.

Dirección: Av. Hidalgo #303. Col Centro C.P. 90300. Apizaco, Tlaxcala.

Correo: josue.ayala@tech4grow.com.mx

Teléfono: 241 106 2154

Sitio web: <https://tech4grow.com.mx/>

Misión: facilitar y potenciar la ruta de transformación digital de las empresas, ofreciendo soluciones innovadoras, accesibles y basadas en tecnologías maduras, estables y en tendencia.

Visión: ser un aliado estratégico para nuestros clientes, acompañándolos en su ruta hacia la digitalización e hiper automatización que abarque toda su cadena de valor.

Valores:

- **Excelencia:** superamos expectativas y ofreceremos resultados sobresalientes en cada proyecto.
- **Innovación:** impulsamos la creatividad y la adopción de nuevas tecnologías que generen en nuestros clientes un valor diferencial.
- **Integridad:** actuamos con honestidad, transparencia y ética en todas nuestras acciones.
- **Colaboración:** fomentamos un ambiente de trabajo en equipo donde se valoran las ideas y se escuchan las diversas perspectivas.
- **Compromiso:** nos enfocamos en entender las necesidades de nuestros clientes y ofrecemos soluciones que generen valor real para su negocio.

Giro de la empresa: ofrecer soluciones innovadoras y personalizadas a través de diversas especialidades, incluyendo Robotic Process Automation, Desarrollo de Software, Oracle, Bases de Datos, Apps Móviles y Empresariales, Integraciones, ERP, Microsoft Power Platform, Power Automate, Power Bi, Power Apps y Power Virtual Agents.

1.3. Problemas a resolver

El proyecto se enfoca en desarrollar un backend REST en Spring Boot, centrado en la creación de un sistema de almacenamiento, al igual que la gestión de posiciones geográficas de dispositivos móviles que sea escalable con facilidad de uso. Se integrará con una base de datos relacional, además de servicios REST para la administración de usuarios, dispositivos móviles y ubicaciones geográficas. A continuación, se presentan los problemas que se buscan resolver:

En primer lugar, un problema categorizado como crítico, es la falta de un sistema de almacenamiento independiente para posiciones geográficas en los dispositivos móviles. Esto se debe a los cambios en la forma en que se almacena y gestiona el historial de ubicaciones en Google. Por ello, es necesario desarrollar un sistema alternativo, que permita a los usuarios almacenar como gestionar posiciones geográficas en los móviles. Resulta importante que los usuarios tengan control sobre sus datos, logrando gestionar eficazmente la información de manera externa, es decir, que esta no se guarde en los dispositivos.

Lo cual permite abordar el segundo problema, categorizado como importante, es la Integración con una base de datos MySQL, para el manejo de la información de usuarios, dispositivos al igual que de ubicaciones geográficas. Esta integración es fundamental para almacenar y gestionar la información de manera óptima. A pesar de que en el desarrollo de aplicaciones móviles suelen almacenar los datos en bases de datos no relacionales como Firebase, lo que es una buena opción, pero con el tiempo su estructura puede volverse compleja al mantenerla. Por lo tanto, la integración con una base de datos relacional se presenta como una alternativa valiosa para mejorar la gestión de datos de ubicaciones.

Por último, un problema de tipo significativo es el desarrollo de servicios REST para la administración de usuarios, dispositivos móviles así como de ubicaciones geográficas. Es necesario desarrollar servicios REST que permitan realizar operaciones CRUD (creación, actualización, consulta, eliminación) de registros de usuarios, dispositivos móviles. Esto incluye consultar, almacenar, también filtrar ubicaciones geográficas de manera flexible, además de escalable. Los servicios REST pueden utilizarse para crear nuevos usuarios, actualizar información de dispositivos móviles, consultar ubicaciones geográficas, o bien eliminar registros de usuarios que ya no sean necesarios. Al consumir la API REST en alguna interfaz creada con posterioridad permite que los usuarios soliciten mediante los *endpoints* creados.

1.4. Objetivos

1.4.1. Objetivo general

Desarrollar un *backend* en Spring Boot que proporcione un servicio REST capaz de recibir, almacenar y gestionar posiciones geográficas de dispositivos móviles en una base de datos MySQL, permitiendo su consulta y administración de manera eficiente.

1.4.2. Objetivos específicos

- Configurar el entorno de desarrollo en Spring Boot para la implementación de un backend REST que interactúe con una base de datos MySQL.
- Diseñar y modelar la base de datos en MySQL para almacenar la información de usuarios, dispositivos y ubicaciones geográficas.
- Desarrollar un servicio REST para la administración de usuarios, permitiendo la creación, actualización, consulta y eliminación de registros de usuarios en la plataforma.
- Implementar un servicio REST para la gestión de dispositivos móviles, que permita registrar, actualizar, consultar y eliminar los dispositivos asociados a los usuarios.
- Crear un servicio REST para la administración de ubicaciones geográficas, permitiendo el almacenamiento y consulta del historial de posiciones de los dispositivos móviles de los usuarios.

1.5. Justificación

El historial de ubicaciones de Google ha experimentado cambios significativos en los últimos años. Si bien no ha desaparecido, la forma en que se almacena y gestiona esta información ha evolucionado, poniendo un mayor énfasis en la privacidad del usuario. Del Vayo (2024) explica que, si bien Google no eliminó el historial de ubicaciones, sí modificó su comportamiento, integrándolo en la memoria del smartphone, además de un plazo de eliminación predeterminado de 18 meses.

Estos cambios han llevado a que algunas aplicaciones, o bien, servicios que dependían de esta información hayan tenido que adaptarse o buscar alternativas. Es por ello, que se desarrollará un servicio *REST* que permita almacenar, así como administrar las posiciones geográficas de un dispositivo móvil.

1.6. Delimitación

Se desarrollará una aplicación backend utilizando Spring Boot, java, REST y MySQL, que permita administrar posiciones geográficas de dispositivos móviles que pertenecen a usuarios específicos. Se desarrollarán únicamente endpoints REST y queda fuera de alcance el desarrollo de la interfaz de la misma manera queda fuera de alcance la implementación en un servidor productivo juntamente con la implementación de seguridad.

II. Marco teórico

El desarrollo del *backend* robusto así como escalable requiere no solo de habilidades técnicas, sino también en la elección adecuada de metodologías, herramientas, además de principios de diseño que garanticen calidad junto con mantenibilidad. En este sentido, el presente marco teórico tiene como propósito fundamentar las bases conceptuales, también técnicas clave que sustentan el desarrollo del *backend* del presente proyecto, centrado en la gestión de posiciones geográficas mediante la creación de una *API REST* construida en Spring Boot.

Es imprescindible conocer, así como aplicar principios de desarrollo ágil, utilizando metodologías como Scrum y *Kanban*, que facilitan la planificación, sumando a la ejecución de tareas en entornos de desarrollo iterativos e incrementales, estas metodologías permiten una mejor organización del equipo de trabajo, favoreciendo la adaptabilidad a los cambios, al igual que la entrega continua de valor. Se examina la arquitectura de software bajo la perspectiva de Kruchten en su modelo “4+1” que es utilizado para estructurar el sistema.

Por otro lado, se profundiza el uso de Java el cual es uno de los lenguajes principales en el desarrollo del *backend*, además de herramientas clave como *Maven*, *JPA* e *Hibernate* para gestión de dependencias y persistencia de datos, asimismo con el uso de la filosofía del Clean Code como pilar en la escritura de código comprensible, mantenible y eficiente, lo que resulta clave en proyectos de software a largo plazo. Este marco teórico servirá como base para comprender los fundamentos que respaldan el desarrollo del sistema, asegurando que las decisiones técnicas tomadas estén sustentadas en prácticas.

2.1. Metodologías ágiles

Las metodologías ágiles son un enfoque para la planificación y gestión de proyectos de software que se caracteriza por su flexibilidad y adaptabilidad a cambios basados en el principio de que es mejor entregar un producto funcional y evolutivo en lugar de esperar a tener todas las especificaciones detalladas al inicio del proyecto (Nache, 2022). Este enfoque surgió por el constante crecimiento del desarrollo de software a inicios del siglo XXI en el que se estaba convirtiendo en algo común en el día a día.

A palabras de Laoyan (2025) describe que en el año del 2001, el método lineal en cascada no resultaba efectivo para los equipos que trabajaban con software porque todo cambiaba constantemente y las necesidades de los clientes también se modificaban, así es como surgieron

las metodologías ágiles, que se concretaron en “**The Agile Manifesto**” (**Manifiesto Ágil**), un compendio de cuatro valores y doce principios que revolucionaron el modo en el que se gestionaban los proyectos de desarrollo de software.

La gestión de proyectos en el desarrollo ágil del software se había convertido en un proceso rígido que terminaba en productos que no atendían a las necesidades del cliente. Entendieron que era clave diseñar un proceso que posibilitara realizar cambios veloces y entregar valor al cliente de forma continua, sin necesidad de esperar a tener el producto completo.

El Manifiesto Ágil es el resultado que cambia la forma de entender la gestión de proyectos. Los cuatro valores principales de la gestión ágil de proyectos son:

1. **Individuos e interacciones sobre procesos y herramientas:** se prioriza la colaboración y comunicación efectiva dentro del equipo, por encima de los procesos rígidos para usar herramientas específicas. De esta manera, se fomenta el trabajo en colaborativo teniendo como resultado la resolución de problemas que llevan a obtener los mejores resultados posibles.
2. **Software funcionando sobre documentación extensiva:** es entregar al consumidor el producto que quieren por encima de otros trabajos.
3. **Colaborar con el cliente en la negociación:** los equipos dejan que los clientes marquen la dirección en la que se debe orientar el software. La colaboración con el cliente es más importante que los detalles más específicos de la negociación contractual.
4. **Respuesta ante el cambio sobre seguir un plan:** el contrato con el cliente debe ser flexible, es más importante entregar un producto con valor al cliente. Ofreciendo la posibilidad de que los equipos cambien rápido de estrategia sin afectar al proyecto entero.

Estos cuatro valores son los pilares de la Metodología Ágil. A partir de estos, se desarrollaron doce principios, que se pueden adaptar fácilmente para las necesidades de cada equipo son:

1. **Satisfacer a los clientes a través de la entrega temprana y continua:** cuando los clientes reciben actualizaciones del proyecto, verán los cambios que desean dentro del producto y la satisfacción del cliente sea positiva. Los clientes son más felices y están más satisfechos. La clave es deleitar al cliente.

2. **Aceptar que los requisitos cambien en el proyecto:** el eje fundamental de los esquemas ágiles es la adaptabilidad. En procesos iterativos la flexibilidad causa más bien que mal.
3. **Hacer entregas valiosas con frecuencia:** se efectúan entregas valiosas a los clientes con frecuencia y por eso es menos probable que se pierdan los clientes.
4. **Romper con el aislamiento en los proyectos:** la colaboración es la clave. Las personas ya no realizan actividades individuales para colaborar juntas con mayor frecuencia.
5. **Desarrollar proyectos con personas motivadas:** funciona mejor cuando los integrantes de los equipos se sienten comprometidos con lo que hacen y trabajan activamente para alcanzar algún objetivo, lo que contribuye a la excelencia técnica del proyecto.
6. **Comunicarse cara a cara:** al trabajar en un equipo descentralizado, se dedica tiempo a comunicarse de forma más personalizada, fomentando la comunicación cara a cara.
7. **El software funcionando indica progreso:** lo más importante es que los equipos se esfuercen por desarrollar los productos. La entrega de software que funciona por sobre cualquier otra cosa.
8. **Mantener un ritmo de trabajo constante:** algunos miembros del equipo pueden tener un ritmo acelerado, pero nada debería ser tan rápido como para que los miembros del equipo terminen padeciendo agotamiento. El objetivo es mantener la constancia a lo largo del proyecto.
9. **Favorecer la agilidad con excelencia continua:** si el equipo desarrolla un código excelente en un sprint, podrá basarse en ese mismo código para el siguiente desarrollo sostenible, permitiendo que los equipos avancen más rápido con el paso del tiempo.
10. **La simplicidad es esencial:** a veces, la solución más simple es la mejor. Es no complicar demasiado las cosas para hallar respuestas simples a problemas complejos.
11. **Resultados valiosos en la organización interna y autónoma del equipo:** los equipos en los que se trabaja con proactividad producen activos más valiosos para la empresa, ya que se esfuerzan por aportar ese valor.
12. **Reflexión y adaptación del equipo para favorecer la efectividad:** las reuniones diarias para análisis retrospectivo son muy comunes en las prácticas ágiles. Los equipos reflexionan sobre su desempeño.

El esquema o metodología ágil es una metodología iterativa, es decir, se realizan entregas cíclicas en cada una de las fases del ciclo: desde toma de requerimientos, diseño, verificación y entrega. La mayor diferencia de las metodologías ágiles, frente a los antiguos modelos waterfall, es que en los procesos ágiles se entrega valor para obtener feedback constantemente durante todo el proyecto. En lugar de depender de un producto final predefinido desde el inicio, promoviendo la creación de software funcional que evoluciona de manera incremental.

2.2. Metodología Scrum

Dentro del amplio espectro de metodologías ágiles, Scrum se ha consolidado como uno de los marcos de trabajo más populares y ampliamente adoptados. De Dios (2024) lo describe como una guía clara para los equipos de desarrollo al afrontar nuevos proyectos, destacando cinco valores fundamentales que sustentan su enfoque:

1. **Coraje:** valentía para hacer lo correcto y resolver problemas complejos.
2. **Foco:** se logra mediante la organización del trabajo en sprints, lo que permite centrar los esfuerzos en objetivos alcanzables.
3. **Compromiso:** implica la responsabilidad de todos los miembros involucrados.
4. **Sinceridad:** se manifiesta en la transparencia del equipo en cuanto al trabajo realizado así mismo con los obstáculos encontrados.
5. **Respeto:** reflejado en el trato adecuado entre compañeros, en valoración de sus ideas y esfuerzos de cada uno es imprescindible.

Scrum se caracteriza por su flexibilidad, permitiendo la adopción de cambios, teniendo como resultado, los nuevos requisitos a lo largo de un proyecto complejo. Además, pone un fuerte énfasis en el factor humano, promoviendo la colaboración continua con el cliente en el que favorece el desarrollo iterativo como mecanismo para alcanzar buenos resultados.

El autor también explica que los tres pilares fundamentales de Scrum son:

1. **Transparencia:** todos los implicados deben tener conocimiento claro de lo que ocurre en el proyecto generando una visión común.
2. **Inspección:** los miembros del equipo revisan periódicamente el progreso para detectar posibles problemas para asegurar que el trabajo fluya de forma efectiva.

3. **Adaptación:** ante la necesidad de ajustes, el equipo modifica su enfoque para alcanzar los objetivos del sprint, siendo esta capacidad de respuesta un factor clave en entornos de alta incertidumbre.

Respecto a los roles dentro del equipo Scrum, se establecen tres funciones principales:

1. El **Product Owner** es el responsable de maximizar el valor del trabajo a través de la gestión del Product Backlog, actuando como el principal interlocutor con el cliente.
2. El **Scrum Master** se encarga de asegurar que se comprendan y apliquen correctamente los principios de Scrum, actuando como un líder que elimina impedimentos.
3. El **Equipo de Desarrollo** es un grupo multifuncional y autoorganizado que se encarga de llevar a cabo las tareas priorizadas por el Product Owner, asegurando la entrega de valor sin requerir supervisión constante.

En cuanto a los eventos en Scrum, el desarrollo iterativo se estructura en ciclos llamados “sprints”, considerados como el núcleo de esta metodología. La duración de un sprint no debe superar un mes.

Cada sprint es conformado por una serie de eventos clave:

- **Sprint Planning:** reunión inicial donde se define el trabajo a realizar así como definir el objetivo del sprint. De ella se obtiene el Sprint Goal y el Sprint Backlog; puede llegar a durar hasta 8 horas en sprints mensuales.
- **Daily Meeting:** breve reunión diaria (máx. 15 minutos) para sincronizar actividades, abordar impedimentos además de revisar avances.
- **Sprint Review:** sesión en la que presentan los resultados al cliente a la vez que se recibe retroalimentación; puede durar hasta 4 horas.
- **Sprint Retrospective:** espacio de reflexión del equipo sobre el sprint, que permite identificar mejoras; puede extenderse hasta 3 horas.
- **Grooming (o Refinamiento del Product Backlog):** dedicado a revisar para preparar el Product Backlog , ocupando hasta el 10% del tiempo del sprint.

Scrum contempla también, tres artefactos clave, diseñados para maximizar la transparencia dentro del proyecto:

1. **Product Backlog:** lista priorizada de tareas que abarca todo el proyecto. Es responsabilidad exclusiva del Product Owner, quien mantiene una comunicación constante con el cliente.
2. **Sprint Backlog:** conjunto de tareas seleccionadas del Product Backlog durante el Sprint Planning, junto con un plan para ejecutarlas. Este artefacto puede ajustarse durante el sprint, pero las tareas seleccionadas no cambian.
3. **Incremento:** representa la suma de todos los elementos del Product Backlog completados en el sprint actual, es el valor acumulado de los sprints anteriores. Se presenta durante la Sprint Review para su validación.

Scrum, puede resultar complejo, en el aspecto de que se requiere compromiso y disciplina por parte de todo el equipo de trabajo. Lo que la convierte en una herramienta efectiva para gestionar proyectos complejos de manera ágil así como adaptable, permitiendo a los equipos entregar valor de forma:

- Rápida,
- identificable,
- resolver problemas de manera temprana,
- mejorar continuamente sus procesos.

Cabe señalar que, más allá de los elementos fundamentales del marco Scrum, algunos autores han propuesto una visión más evolucionada de esta metodología. Menzinsky et al., (2016) plantean que Scrum ha progresado al punto de distinguirse en dos niveles: **Scrum técnico** y **Scrum avanzado**.

El Scrum técnico abarca los roles, eventos y artefactos tal como se definen en el marco original, sirviendo como estructura operativa básica para los equipos de desarrollo. Por su parte, el Scrum avanzado se caracteriza por la incorporación de prácticas más flexibles, como el uso de tableros *Kanban* ajustados a las necesidades del proyecto como técnicas de incremento continuo e iterativo.

Según los autores, una vez que los equipos han alcanzado una implementación efectiva del Scrum técnico, pueden dar el siguiente paso, para una mayor madurez hacia el Scrum avanzado. Esta transición se caracteriza por dejar de aplicar las reglas de forma estricta para enfocarse en

una interpretación más profunda de los valores de Scrum, ajustando sus prácticas a las circunstancias específicas de la organización.

En este nivel avanzado, el rol de Scrum Master se expande más allá de la dinámica del proyecto individual, asumiendo una visión más sistémica de la organización. Como destacan los autores *"la organización, como realidad sistémica debe dar respuesta de forma coordinada y alineada con su visión, a responsabilidades en tres áreas: gerencia, procesos y producción"* (p. 54)

Estas tres áreas representan un marco de responsabilidad distribuido que va más allá de los roles tradicionales del Scrum técnico:

1. **Gerencia:** garantiza el equilibrio sistémico de la organización, la coherencia del modelo ágil adoptado, junto con la provisión de medios e incluso la formación necesarios para su implementación.
2. **Procesos:** abarca la configuración flexible de Scrum, según el tipo de proyecto, para la mejora continua del modelo de trabajo en el que asegura el funcionamiento adecuado del marco, responsabilidades que tradicionalmente recaen en el Scrum Master.
3. **Producción:** se centra en la entrega del valor del producto mediante la gestión del producto, la promoción de la autoorganización del equipo juntamente con la adopción de las tecnologías ágiles, tareas comúnmente asignadas al Product Owner y al equipo en el nivel técnico (p. 55).

2.3. Kanban

Pasando a otro punto, es necesario examinar el enfoque *Kanban*. Al respecto, Martins (2025) la describe como una metodología que ofrece una vía para ayudar a los equipos lograr un balance entre las tareas pendientes y la capacidad real de cada integrante para llevarlas a cabo. Kanban promueve una cultura de mejora continua, donde las actividades se “extraen” progresivamente de un conjunto de tareas pendientes que fluyen a través de un proceso organizado.

Mientras que Kanban pone énfasis en optimizar los procesos, Scrum suele aplicarse con el objetivo de aumentar la productividad del equipo que permite acelerar la finalización del trabajo. En este contexto, Kanban se considera una subcategoría de la gestión ágil de proyectos de software.

Actualmente, la representación más común de Kanban es mediante tableros virtuales, en los cuales se disponen con columnas que reflejan las distintas etapas del flujo de trabajo. No obstante, todavía es posible utilizar tableros físicos con notas adhesivas sobre pizarras blancas. Cada tarea se simboliza mediante una *tarjeta Kanban*, la cual se desplaza de una columna a otra conforme avanza en función de las etapas del proceso.

En este sentido, Lowe et al., (2021) señalan que los elementos de trabajo pueden variar en tipo y tamaño —*desde tareas simples hasta proyectos complejo*-- y que estos fluyen a través del tablero en forma de anotaciones individuales o tickets, siendo el tablero un reflejo fiel del flujo real de trabajo, más allá de una visión idealizada del mismo. También se destacan por el uso de elementos visuales como tarjetas de colores, para distinguir tipos de tareas o clientes. Un tablero Kanban suele construirse al menos con las clásicas columnas de "Por Hacer", "En Progreso" y "Terminado", aunque dependiendo del flujo del trabajo se pueden incluir tantas columnas como se requieran (p. 12).

En el ámbito de las Tecnologías de la Información y la Comunicación (TIC), Menzinsky et al., (2016) afirman que la implementación de tableros Kanban se ha vuelto una práctica común para visualizar y administrar el progreso de los proyectos, facilitando la reordenación de prioridades en la pila del producto. Otros rasgos de esta herramienta es que permite identificar así como mitigar dos obstáculos clave en los procesos: los cuellos de botella en el flujo de trabajo y los periodos de inactividad.

La implementación de Kanban, según Martins (2025), se guía por cuatro principios fundamentales:

1. **Empezar con lo que haces ahora:** Kanban puede aplicarse a cualquier flujo de trabajo existente. A diferencia de otros marcos ágiles más definidos como Scrum, ofrece gran flexibilidad para adaptarse a las prácticas ya establecidas dentro del equipo.
2. **Comprometer a la búsqueda e implementación de cambios progresivos y evolutivos:** introducir cambios abruptos, puede ser perjudicial. Kanban se basa en una filosofía de evolución progresiva, fomentando pequeños ajustes que impulsen una transformación sostenible para lograr que los procesos realmente evolucionen con el tiempo.
3. **Respetar los procesos, los roles y las responsabilidades actuales:** a diferencia de otras metodologías, Kanban no requiere redefinir funciones ni estructuras. De este modo,

permite conservar elementos eficaces del sistema actual, evitando pérdidas derivadas a cambios radicales de un momento a otro.

4. **Impulsar el liderazgo en todos los niveles:** Kanban reconoce que las iniciativas de mejora pueden surgir desde cualquier punto del equipo, no únicamente desde la dirección. Por ello, se alienta a los miembros a participar, proponer formas para lograr que los procesos evolucionen y emprender nuevas iniciativas de trabajo, en el que permite el surgimiento del liderazgo de acciones que fortalezcan el desempeño colectivo.

2.1. Arquitectura de software

La arquitectura de software constituye la base estructural sobre la cual se diseñan los planos para posteriormente desarrollar las aplicaciones, en el que se define la organización general del sistema, los componentes que lo integran, sus relaciones junto con las directrices que rigen su evolución a lo largo del tiempo. Como señalan Bass, Clements y Kazman (2003), la arquitectura de software comprende “la estructura o estructuras del sistema, que incluyen componentes de software, las propiedades visibles externamente de esos componentes y las relaciones entre ellos” (p. 21).

Bajo esta perspectiva, se resalta que una buena arquitectura no solo considera en cómo está organizado el sistema, sino también en cómo soporta e incorpora atributos de calidad como el rendimiento, la escalabilidad, la mantenibilidad e incluso la seguridad.

Una de las propuestas más influyentes para describir la arquitectura de software es el modelo de las "vistas 4+1" propuesto por Philippe Kruchten en (1995), dentro del marco del Proceso Unificado de Rational. Este modelo se basa en la idea de que ningún conjunto único de diagramas puede abordar adecuadamente las necesidades de todos los interesados en un sistema como lo son los usuarios, diseñadores, desarrolladores, testers, por lo que propone cinco vistas complementarias, que permiten representar los distintos aspectos del sistema:

1. **Vista lógica:** apoya principalmente los requisitos funcionales, es decir, lo que el sistema debe ofrecer en términos de servicios a sus usuarios. Generalmente esta vista se construye mediante diagramas de clases.
2. **Vista de procesos:** considera requisitos no funcionales como el rendimiento y la disponibilidad, en el que especifica en qué hilo de control se ejecuta efectivamente las operaciones identificadas en la vista lógica.

3. **Vista física:** describe el mapeo del software en el hardware; se reflejan los aspectos de distribución del sistema.
4. **Vista de desarrollo:** muestra la organización estática del software en su entorno de desarrollo. Se enfoca en la estructura real de módulos, paquetes así como componentes.
5. **Vista de escenarios o vista + 1:** integra las vistas anteriores mediante un conjunto reducido de escenarios o casos de uso representativos que permiten verificar cómo interactúan los elementos de las otras vistas a través de diagramas de interacción.

También resalta que *“los elementos de una vista están conectados a los elementos de las otras vistas siguiendo ciertas reglas y heurísticas de diseño”* (p. 10), lo que permite mantener coherencia además de trazabilidad en el sistema arquitectónico global.

Para complementar la implementación de las vistas 4+1, Moya (2012) señala que, si bien no hay una manera estándar para documentarlas, puede recurrirse al **UML (Lenguaje de Modelado Unificado)**, ya que este permite representar visualmente los diferentes componentes del software mediante sus diagramas, facilitando así la documentación de cada perspectiva. En este marco, un diagrama puede actuar como una vista, al estar estandarizado por UML, se evita comenzar la documentación desde cero.

Partiendo de esta premisa, la vista lógica puede representarse mediante diagramas de clases, de comunicación y de secuencia, los cuales permiten describir lo que el sistema debe hacer, así como las funciones además de los servicios que ofrecerá.

Por su parte, la vista de desarrollo (conocida también por vista de despliegue) logra documentarse con el uso de diagramas de componentes y de paquetes, lo que ayuda al programador a dividir el sistema en módulos para visualizar las dependencias entre ellos.

En lo que respecta a la vista de procesos se ilustra con diagramas de actividades, útiles para mostrar los procesos del sistema así como el flujo de trabajo de la lógica del negocio.

En cuanto a la vista física se sugiere el uso de diagramas de despliegue, los cuales detallan los componentes físicos del sistema, proporcionando una visión técnica que facilita su implementación por parte del ingeniero en sistemas para su construcción.

Finalmente, la vista +1 (conocida como vista de escenarios) es representada mediante diagramas de casos de uso, los cuales permiten relacionar las cuatro vistas anteriores a través de los escenarios, es decir de ejemplos de uso concretos. Por último, el autor concluye que el

modelo 4+1 se presenta como una herramienta eficaz para comunicar e incluso documentar arquitecturas complejas, alineándose con estándares como el IEEE 1471-2000.

2.2. Java

Joyanes Aguilar y Luis Zahonero Martínez (2011), en su libro “Programación en Java 6. Algoritmos, programación orientada a objetos e interfaz gráfica de usuario”, destacan la importancia del lenguaje Java para el desarrollo de aplicaciones contemporáneas además de su relevancia en el entorno de internet. Señalan que *"Java es un lenguaje de programación creado para desarrollar programas que ejecuten tareas de diversos tipos... y se diseñó de manera concreta para realizar una amplia gama de actividades en internet"* (p. 1).

En una revisión histórica del lenguaje, Bernal (2021) lo describe como una herramienta de propósito general de alto nivel, concebida para resolver una amplia variedad de problemas. SUN Microsystems, fundada en 1982 por Andreas Von Bechtolsheim, Vinod Khosla, Scott McNealy y Bill Joy, identificó desde sus inicios la necesidad de un lenguaje capaz de operar en diversos dispositivos electrónicos sin depender del hardware específico.

Motivados por esta necesidad, el equipo liderado por James Gosling creó un lenguaje completamente nuevo, basado en la sintaxis de C/C++ pero con la filosofía de "escribe una vez, ejecuta en cualquier lado" (WORE/WORA). Este proyecto, inicialmente conocido como Green Talk y luego como OAK, adoptó el nombre de Java en 1995 debido a problemas de registro de marca.

La propuesta de este respondía a cinco objetivos principales:

1. Implementar la programación orientada a objetos.
2. Garantizar la portabilidad del programa en múltiples sistemas operativos.
3. Incluir soporte para trabajo en red.
4. Ejecutar código de forma segura en sistemas remotos.
5. Ser fácil de utilizar, aprovechando las fortalezas de lenguajes orientados a objetos como C++.

SUN reconoció el potencial de Java en este nuevo panorama, lo que dio lugar a una reorientación hacia el entorno web. Gosling (como se citó en Bernal, 2021) recuerda: *"Nos dimos cuenta de*

que podíamos construir un navegador realmente genial. Así que creamos un navegador". Este navegador, conocido como WebRunner y luego HotJava, fue el primero en ejecutar *applets* escritos en Java, permitiendo la incorporación de elementos dinámicos mediante animaciones así como sonidos en las páginas web.

Los *applets* son definidos como pequeños programas incrustados en documentos HTML que se ejecutaban dentro de navegadores compatibles, ofrecieron en su momento una experiencia web enriquecida. Sin embargo, requerían que Java estuviera instalado e incluso habilitado. Aunque actualmente están en desuso, fueron clave en su momento para añadir interactividad a las páginas web. Hoy en día, han sido reemplazados por tecnologías web más modernas, seguras a la par de eficientes como HTML5, JavaScript, CSS, así como frameworks de *frontend* como Angular o React Native, y de *backend* como Spring Boot (que utilizan Java).

Este giro estratégico consolidó la posición de Java como un lenguaje versátil y adaptable, fundamental en el desarrollo web, tal como lo indicaron Joyanes y Zahonero (2011). En cuanto a su funcionamiento los autores explican que, a diferencia de otros lenguajes de alto nivel, Java compila el código fuente *.java* en *bytecode*, generando un archivo *.class* mediante el compilador *javac*. Este *bytecode* es luego interpretado por la JVM (Máquina Virtual Java), lo que permite que los programas puedan funcionar de manera consistente en diferentes entornos, sin depender de la arquitectura o el sistema operativo, garantizando así la portabilidad de las aplicaciones.

A medida que Java se consolidaba como un lenguaje robusto, SUN Microsystems decidió subdividirlo en distintas ediciones, cada una orientada a contextos específicos. Como explica Nieva (2019), a diferencia de una versión, una edición no implica una mejora respecto a una anterior, sino un conjunto de características orientadas a una especialización concreta. A partir de 1998, con la introducción de Java 2, SUN comenzó a diferenciar la plataforma base (JDK) de otras ediciones como **J2EE** para el desarrollo empresarial y **J2ME** para dispositivos móviles.

Actualmente las ediciones vigentes son:

- **Java SE (Standard Edition):** enfocada en el desarrollo de aplicaciones de escritorio y servidores. Incluye componentes esenciales como el *JRE*, la *JVM* y el *JDK*.
- **Java EE (Enterprise Edition):** ofrece una plataforma robusta para aplicaciones empresariales distribuidas de gran escala, multicapas, aplicaciones web, *APIs RESTFUL*, compuesta por cerca de 28 especificaciones.

- **Java ME (Micro Edition):** orientada al desarrollo en dispositivos móviles y sistemas embebidos como sensores, microcontroladores y otros elementos del internet de las cosas como interfaces flexibles, protocolos integrados y soporte offline.
- **JavaFX:** una familia de tecnologías destinada al desarrollo de interfaces gráficas enriquecidas (RIAs), presentada en 2008 como competencia de Flash y Silverlight.
- **Java Card:** especializada para ejecutar applets en Smart cards y dispositivos embebidos, con su propia máquina virtual y entorno de ejecución simulado.

Por su parte, Bernal (2021) también contextualiza el panorama de SUN Microsystems antes de ser adquirida por Oracle. SUN se distinguió por su fuerte compromiso con el software libre, los múltiples proyectos que lanzó bajo licencias abiertas como *MySQL*, *Solaris*, *OpenOffice*, *Java* y el entorno de desarrollo *NetBeans*. No obstante, en 2009, Oracle compró SUN Microsystems por 7,400 millones de dólares, lo que marcó un giro importante en la historia de Java. Como consecuencia, figuras clave como James Gosling abandonaron la compañía y a su vez Oracle modificó las políticas de licenciamiento. Desde Java SE 9, el uso comercial requiere licencia, aunque el lenguaje Java y sus APIs permanecen libres de derechos de autor. Sin embargo, componentes del entorno JDK, como el *classpath*, están protegidos legalmente, aunque gran parte del código fuente se distribuye bajo licencia GPL.

2.3. Maven

Si bien Java cuenta con una extensa librería estándar que está en constante evolución, para implementar ciertas funciones en la creación de aplicaciones a menudo se requieren de librerías externas, desarrolladas por terceros que deben ser descargadas e importarlas al proyecto. Este proceso, aunque muy común puede derivar en problemas de compatibilidad con el código afectando especialmente en proyectos de gran escala, lo que hace que la construcción de aplicaciones hechas en Java sea inviable realizar la importación y verificación de cada biblioteca manualmente. Para resolver este problema, surge *Maven*, una herramienta de gestión de proyectos así como dependencias creadas en Java.

Souza (2023) expone los antecedentes que permitieron el surgimiento de Maven, destacando el papel de herramientas como Apache Ant, que fue muy popular en su momento porque facilitaba la compilación ordenada de los archivos del proyecto según sus dependencias; Apache Ivy, que introdujo la gestión de dependencias transitivas mediante repositorios JAR. A partir de estas

herramientas se desarrolló *Apache Maven*, concebido como una solución integral para la gestión y comprensión de proyectos Java.

Esta herramienta se basa mediante el **Modelo de Objeto de Proyecto (POM)**, que permite centralizar la configuración del proyecto en un solo archivo centralizado llamado *pom.xml*. Dicho archivo organiza toda la información relacionada con el proyecto en el que facilita la inclusión automatizada de bibliotecas externas, descargadas desde repositorios centralizados, resolviendo además conflictos de versiones. Cada módulo del proyecto puede contar con su propio *pom.xml*, lo que refuerza la jerarquía como la estructura organizacional del software. Además, permite la generación automática de documentación del proyecto, integrando así múltiples funcionalidades en una sola herramienta.

Complementando esta visión, Álvarez (2024) compara el funcionamiento de Maven con el de una fábrica de coches. Según su analogía, así como una fábrica organiza piezas en almacenes que posteriormente se usarán para ensamblarse, Maven proporciona una estructura estandarizada de carpetas, como *src/main/java* para ubicar el código fuente del proyecto. Del mismo modo que la fábrica solicita componentes a sus proveedores, Maven descarga automáticamente las bibliotecas necesarias desde repositorios remotos.

En este proceso se encuentra alineado con el “**ciclo de vida de Maven**” (**LifeCycle**), incluye fases como validación, compilación, pruebas y despliegue del software. Además, el uso de plugins en Maven, permite realizar tareas específicas durante este ciclo, como si fueran operarios especializados en una línea de montaje, facilitando la personalización de acciones según las necesidades del proyecto.

Por estas razones, Apache Maven se ha consolidado como la herramienta más popular para el desarrollo de proyectos en Java, porque ha permitido mejorar la eficiencia, escalabilidad y la portabilidad entre distintos entornos de desarrollo.

2.4. Spring Boot

Sobre la base de la gestión eficiente de proyectos que ofrece Maven, Spring Boot emerge como un poderoso framework que ha revolucionado el desarrollo de aplicaciones Java. Tiene una interesante historia, pues fue construido sobre el robusto Spring Framework. Como explica Baker (2019), Spring fue concebido por Rod Johnson en 2002 como una alternativa más ligera frente al enfoque tradicional de Java EE, el cual implicaba una gran dependencia de sus componentes para construir aplicaciones. Con el objetivo de simplificar el desarrollo del lado del servidor, su propuesta se basó en el uso de *POJOs* (*Plain Old Java Objects*), así como la inyección de dependencias para reducir la complejidad del código.

En palabras de Maldonado (2024), los *POJOs* pueden definirse como “*clases comunes y corrientes que agrupan atributos (propiedades) que ofrecen métodos getters y setters para acceder a ellos*”. Estas características permitieron a Spring posicionarse como una herramienta flexible así como modular para el desarrollo empresarial en Java.

No obstante, integrar aplicaciones Spring en contenedores de *servlets* seguía siendo un desafío. Como respuesta a esta necesidad, en 2013 surgió Spring Boot, un subproyecto liderado por Phil Webb, cuyo propósito fue simplificar aún más la configuración e incluso el despliegue de aplicaciones Java. Spring Boot introdujo convenciones predeterminadas, integración automática de dependencias y la posibilidad de ejecutar aplicaciones directamente con un método *main()*, eliminando la necesidad de servidores externos.

Desde el punto de vista técnico, Константин (2021) resalta que Spring Boot minimiza la configuración inicial al proporcionar una estructura predefinida con configuración automática de los componentes del sistema. También subraya que incluye marcos como Spring MVC y WebFlux para el desarrollo web, así como herramientas específicas para la construcción de *API REST*, lo que facilita significativamente la creación de servicios web con Java.

Aunque Spring Boot está principalmente orientado al desarrollo de aplicaciones web y sistemas *backend*, también se destaca por su capacidad de integrarse con otros proyectos del ecosistema Spring. Según García (2025), Spring Boot al formar parte de toda la arquitectura Spring, puede interactuar con los demás proyectos como Spring Framework, el cual proporciona soporte para construir aplicaciones web y crear nuestras propias *APIs*. Esto permite desplegar servicios *REST* que pueden ser consumidos por distintos tipos de clientes, ya sea una aplicación móvil, una aplicación web u otro tipo de sistema que se conecte a través del protocolo HTTP.

Esta accesibilidad ha contribuido a su adopción masiva. De hecho, según una encuesta sobre el ecosistema de desarrollo realizada por JetBrains (como se citó en Baker 2019) *“un 56 % de los desarrolladores Java reportaron utilizar Spring Boot para aplicaciones web, lo que representó un crecimiento significativo en comparación con el año anterior. Además, un 61 % de los encuestados indicó preferir Spring Boot como alternativa a los servidores de aplicaciones”*.

Complementando esta perspectiva, Canle (2025) destaca como una de las principales fortalezas de Spring Boot es su capacidad de empaquetar aplicaciones como archivos *.jar* autoejecutables, gracias a servidores embebidos como Tomcat o Jetty. Esto sumado a su sistema interno de gestión de dependencias, permite desplegar aplicaciones de forma autónoma, sin necesidad de servidores externos.

Otro aspecto fundamental es su ciclo de vida, que comienza con la ejecución del método *main()*, seguido por la carga automática de configuraciones y la gestión de *beans*, los cuales representan servicios, controladores o repositorios que son administrados por el contenedor de Spring. Como indica el autor, *“cada bean tiene su propio ciclo de vida, que está directamente relacionado con el ciclo de vida de la aplicación en sí”*. También expresa que Spring Boot está especialmente orientado al desarrollo de arquitecturas de microservicios en el que es considerado un enfoque modular que permite dividir una aplicación en componentes independientes que tiene ventajas de una mayor escalabilidad, despliegue independiente de servicios y mayor facilidad para adaptar las soluciones a las necesidades específicas del cliente.

2.5. JPA e Hibernate

Una vez establecida la estructura y la facilidad de desarrollo proporcionada por Spring Boot, la necesidad de interactuar con sistemas de gestión de bases de datos relacionales se vuelve fundamental en muchas aplicaciones Java. En este contexto, Vergara (2016) explica que surge **“JPA” (Java Persistence API)**, una *API* de persistencia que permite manejar datos relacionales tanto en aplicaciones Java SE como Java EE. A diferencia de un framework tradicional, JPA forma parte de la especificación de **EJB 3**, y se define como un documento que establece los principios básicos para la gestión de la capa de persistencia en aplicaciones Java.

Según Vyas (2024), JPA permite a los desarrolladores utilizar objetos Java en lugar de escribir directamente consultas SQL, proporcionando métodos CRUD para crear, leer, actualizar y eliminar datos en las bases de datos. Entre sus componentes se encuentran las **entidades**

(definidas mediante la anotación `@Entity`) que representan las tablas en la base de datos; el **EntityManager**, que actúa como interfaz principal para interactuar con el contexto de persistencia, este gestiona el ciclo de vida de las entidades con métodos como `persist()`, `find()`, `merge()` y `delete()`. Por su parte, el **contexto de persistencia** supervisa las instancias de entidad y sincroniza sus cambios con la base de datos, funcionando como una caché de primer nivel.

Por otro lado, *Hibernate*, también descrito por Vergara (2016), es un framework de “**mapeo objeto-relacional**” (**ORM**) que implementa la especificación de JPA que la aumenta con capacidades adicionales. Este facilita el mapeo de atributos entre clases de Java y tablas de bases de datos relacionales, mediante archivos XML o anotaciones en los beans de las entidades para establecer dichas relaciones.

Como explica Balazs (2023), Hibernate permite escribir consultas utilizando su propio lenguaje de consulta orientado a objetos llamado “**HQL**” (**Hibernate Query Language**), lo cual representa una ventaja significativa al reducir el uso de SQL tradicional permitiendo facilitar el desarrollo.

Dentro de este ecosistema, *Spring Data JPA* se posiciona como una abstracción de mayor nivel desarrollada por el equipo de Spring. Esta simplifica aún más el trabajo con JPA e Hibernate. De acuerdo con Canle (2025), Spring Boot se integra fácilmente con las bases de datos relacionales o no relacionales gracias a este módulo, que permite realizar operaciones CRUD sin necesidad de escribir SQL manualmente.

Complementando esta idea, GeeksforGeeks (2025) señala que Spring Data JPA permite eliminar el código repetitivo al realizar las operaciones CRUD eficientemente a través de repositorios como *JpaRepository* o *CrudRepository*. Por ejemplo el método `findById()` de *JpaRepository* <> permite recuperar un registro por su clave primaria desde una base de datos como *MySQL*. Esta capacidad convierte a Spring Data JPA en una herramienta clave para desarrollar aplicaciones escalables.

2.6. Clean Code

Si bien Spring Data JPA permite reducir el código repetitivo que optimiza la interacción con bases de datos, la verdadera clave para garantizar aplicaciones escalables radica en la forma en que se construye el código. En este sentido, surge el concepto de **Clean Code (código limpio)**, una filosofía y conjunto de prácticas orientadas a escribir código claro, legible y fácil de modificar. Esta fue promovida principalmente por Robert C. Martin en su libro “Clean Code: A Handbook of

Agile Software Craftsmanship” (2009), el cual se ha convertido en una referencia fundamental para los desarrolladores que buscan construir software de calidad. El autor enfatiza que, para aprender a escribir código limpio no es una tarea fácil porque *“requiere algo más que conocer principios y patrones”* (p. 16).

Como señala López (2023), el propósito del Clean Code es mejorar la productividad del equipo de desarrollo, reducir la deuda técnica y facilitar la evolución del software a largo plazo.

La adopción de esta filosofía no solamente requiere conocimiento técnico sino que tendrá que ir acompañado de buenas prácticas y principios de desarrollo de software. Uno de los problemas más comunes cuando no se aplican principios de Clean Code es la acumulación de deuda técnica, término que hace referencia a los problemas derivados de soluciones apresuradas o mal estructuradas. Esta deuda, puede provocar dificultades para realizar cambios, pérdida de productividad, problemas de escalabilidad provocando un mayor riesgo de errores en producción.

Para mitigar estos riesgos, el Clean Code propone una serie de principios y técnicas aplicables al lenguaje Java, particularmente en versiones modernas como Java 18. Entre los más destacados se encuentran:

- Usar adecuadamente nombres descriptivos para variables y funciones.
- Crear funciones cortas que realicen tareas específicas, siguiendo el principio de “una función, una responsabilidad”.
- Usar moderadamente comentarios, priorizando el código auto explicativo y una refactorización continua.
- Diseñar los sistemas bajo acoplamiento y alta cohesión.

Asimismo, prácticas como la "Regla del Boy Scout", sugiere dejar el código en mejores condiciones de cómo se encontró, fomentan una cultura de mejora continua dentro de los equipos de desarrollo. Estas técnicas no solo promueven la claridad del código, sino que también reducen los costos a la corrección de errores y mantenimiento del software en etapas posteriores del ciclo de vida del proyecto.

Además, Choudhary (2025) ofrece un enfoque complementario que profundiza en cómo aplicar principios de Clean Code específicamente en Java. Reconoce que los lenguajes de programación comparten fundamentos comunes, cada uno presenta matices particulares. En el caso de Java, escribir código limpio implica dominar tanto su sintaxis como sus mejores prácticas

establecidas. Destaca que el código limpio debe ser organizado, libre de errores, bien estructurado y comprensible, incluso para programadores novatos. Entre sus recomendaciones prácticas se incluyen:

- Mantener clases y métodos breves que faciliten la legibilidad e incluso el mantenimiento.
- Comentar el código de forma clara y directa, utilizando tanto comentarios de documentación (*/** */*) como de implementación (*//, /* */*).
- Estructurar adecuadamente el código, separando responsabilidades (por ejemplo, almacenamiento vs. procesamiento de datos) y adoptando estructuras estándar como la sugerida por Maven (carpetas *src/main/java*, *src/test/java*).
- Utilizar nombres claros, descriptivos para clases, métodos y variables, reflejando con precisión su propósito.
- Formatear el código de forma consistente, siguiendo convenciones como sangrías, uso de mayúsculas además de separación lógica entre bloques de código.
- Evitar métodos con muchos parámetros, buscando simplificarlas.
- Evitar la codificación rígida (*hardcoding*) de valores, lo que mejora la mantenibilidad y reduce errores.
- Eliminar código duplicado como fragmentos repetidos en métodos o clases reutilizables.
- Escribir registros claros y eficientes, que faciliten la depuración sin afectar el rendimiento.

En combinación con herramientas y entornos modernos como Java 18 (o versiones más recientes) y frameworks de desarrollo como Spring Boot, estas prácticas permiten a los desarrolladores construir soluciones más robustas, mantenibles y alineadas con estándares profesionales.

III. Desarrollo

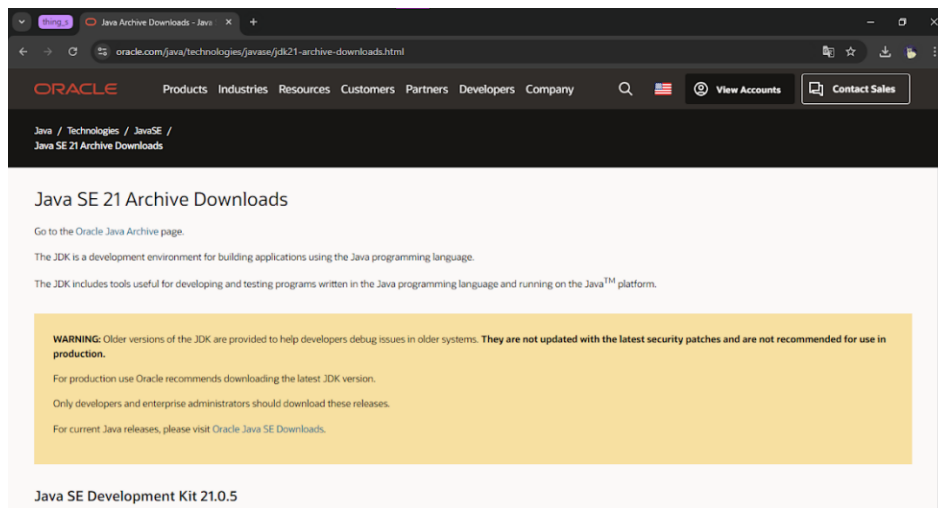
3.1. Procedimiento y descripción de las actividades realizadas.

3.1.1. Instalación del JDK

El proyecto utilizará Java 21. Para descargar el JDK 21, se debe acceder al sitio oficial de Oracle, como se muestra en la Figura 1 en <https://www.oracle.com/java/technologies/javase/jdk21-archive-downloads.html>:

Figura 1

Página oficial de Oracle.



El listado de versiones de JDK disponibles puede observarse en la Figura 2:

Figura 2

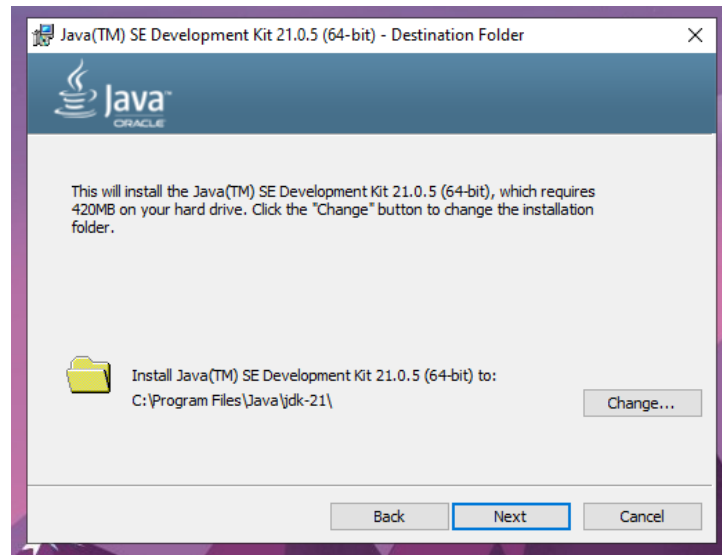
Listado de JDK instalables.

macOS x64 Compressed Archive	184.51 MB	https://download.oracle.com/java/21/archive/jdk-21.0.5_macos-x64_bin.tar.gz (sha256)
macOS x64 DMG Installer	183.85 MB	https://download.oracle.com/java/21/archive/jdk-21.0.5_macos-x64_bin.dmg (sha256)
Windows x64 Compressed Archive	185.91 MB	https://download.oracle.com/java/21/archive/jdk-21.0.5_windows-x64_bin.zip (sha256)
Windows x64 Installer	164.28 MB	https://download.oracle.com/java/21/archive/jdk-21.0.5_windows-x64_bin.exe (sha256)
Windows x64 MSI Installer	163.03 MB	https://download.oracle.com/java/21/archive/jdk-21.0.5_windows-x64_bin.msi (sha256)

El archivo ejecutable de instalación de JDK se presenta en la Figura 3:

Figura 3

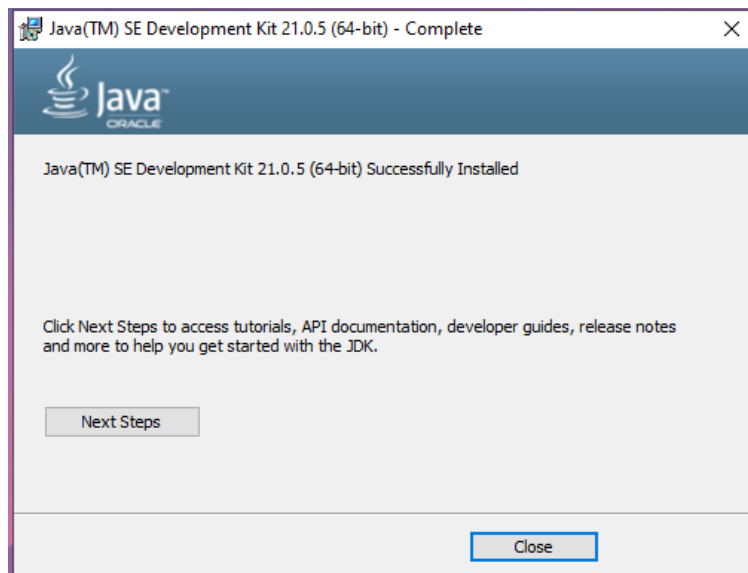
Instalador de JDK.



El proceso de instalación del JDK 21 concluye como se ilustra en la Figura 4:

Figura 4

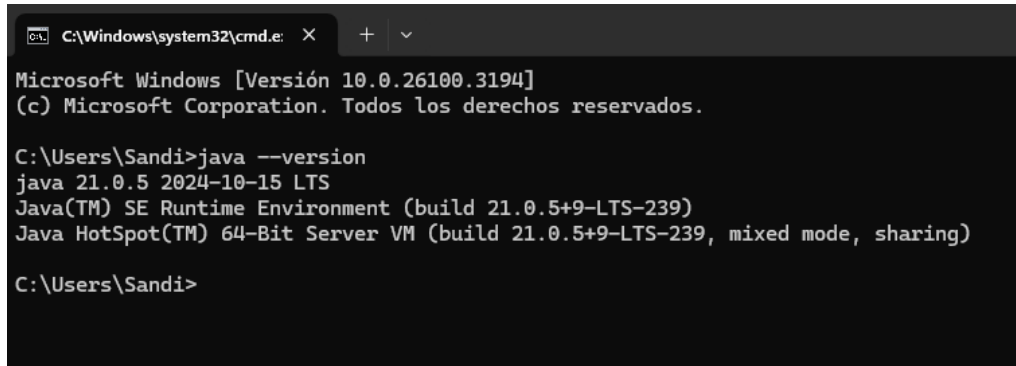
Finalización de la instalación de Java 21.



La correcta instalación se valida ejecutando con el comando **java -version**. Como se puede visualizar en la Figura 5, aparece la versión de Java instalada, en este caso es la versión 21.0.5:

Figura 5

Versión de Java en el CMD.



```
C:\Windows\system32\cmd.e: X + v
Microsoft Windows [Versión 10.0.26100.3194]
(c) Microsoft Corporation. Todos los derechos reservados.

C:\Users\Sandi>java --version
java 21.0.5 2024-10-15 LTS
Java(TM) SE Runtime Environment (build 21.0.5+9-LTS-239)
Java HotSpot(TM) 64-Bit Server VM (build 21.0.5+9-LTS-239, mixed mode, sharing)

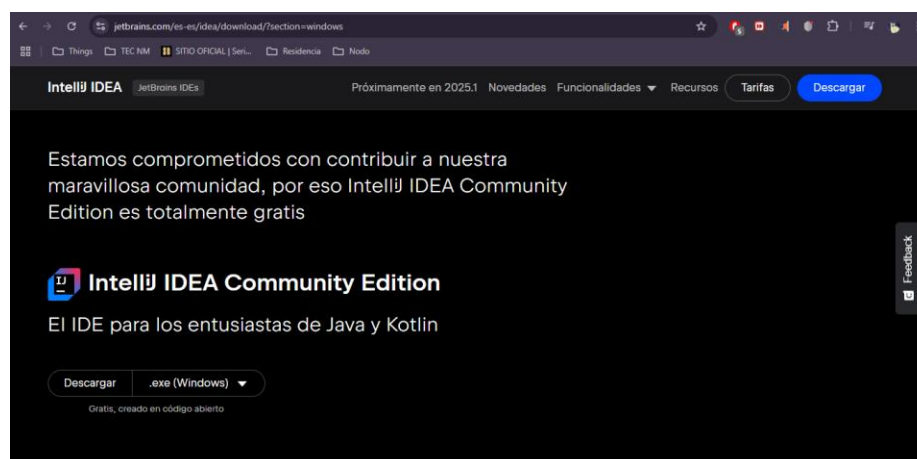
C:\Users\Sandi>
```

3.1.2. Instalación de IntelliJ

La descarga de IntelliJ IDEA inicia desde su sitio oficial, mostrado en la Figura 6 en <https://www.jetbrains.com/es-es/idea/download/?section=windows>:

Figura 6

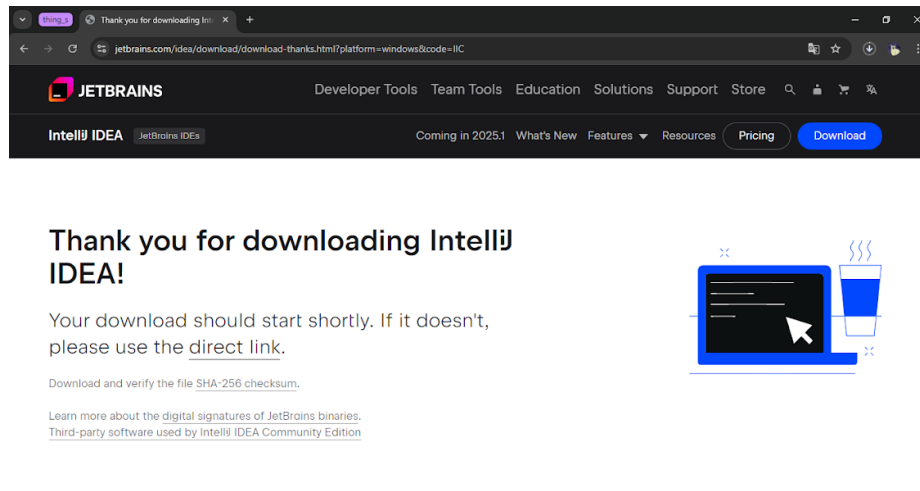
Página oficial de IntelliJ IDEA.



Una vez seleccionado el sistema operativo (en este caso Windows), el archivo se descargará. Al iniciar la descarga, el sitio muestra un mensaje como el de la Figura 7:

Figura 7

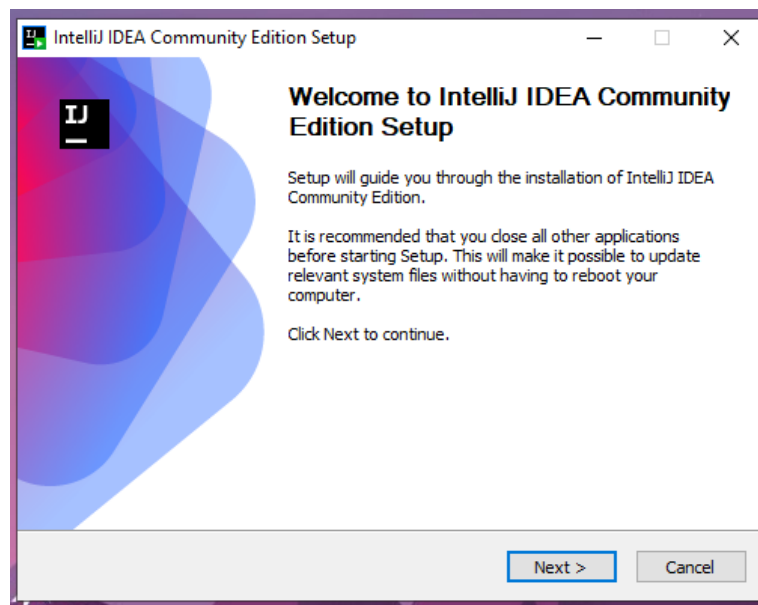
Página de agradecimiento por la descarga de IntelliJ IDEA.



El asistente de instalación de IntelliJ se visualiza en la Figura 8:

Figura 8

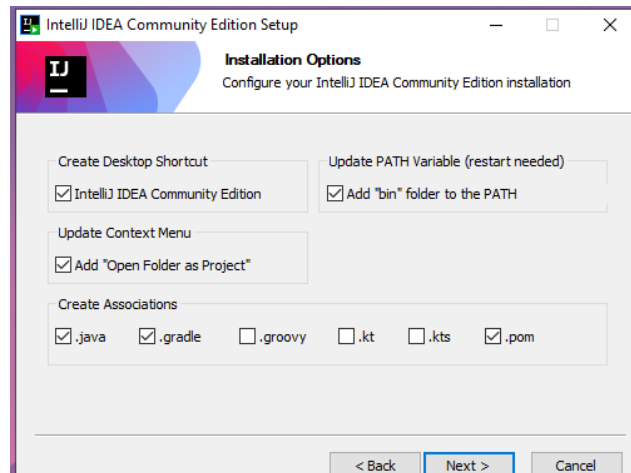
Instalador de IntelliJ IDEA.



La configuración inicial del entorno se puede observar en la Figura 9:

Figura 9

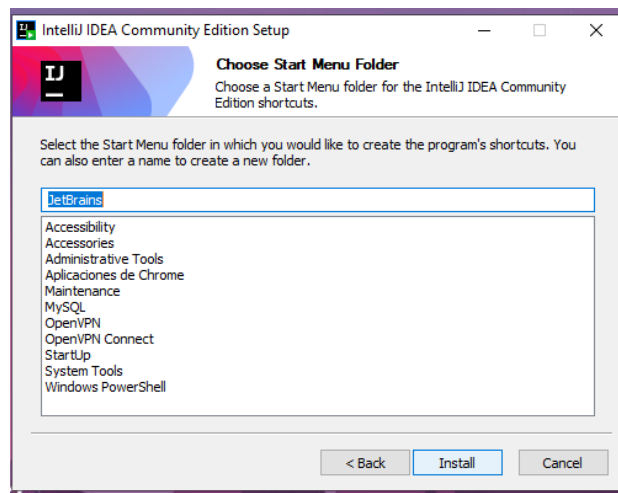
Configuración de IntelliJ IDEA.



Para crear el acceso rápido al entorno es ilustrado en la Figura 10:

Figura 10

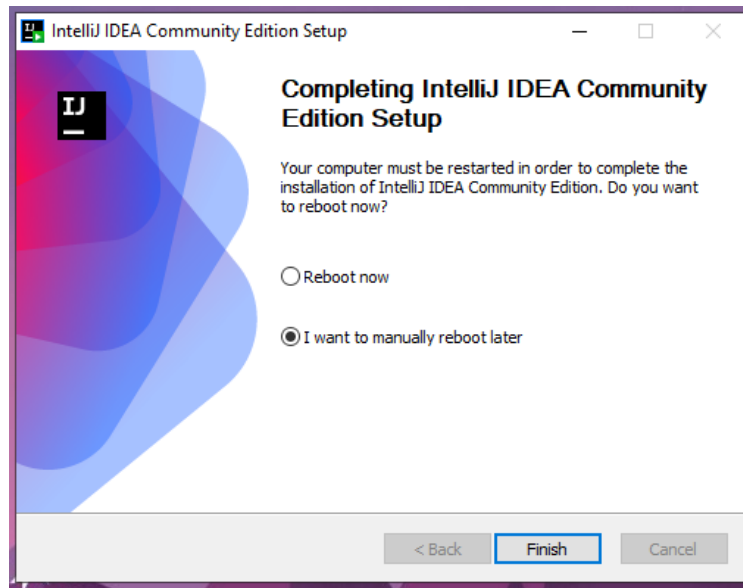
Acceso directo para IntelliJ IDEA.



El proceso de instalación concluye como se aprecia en la Figura 11:

Figura 11

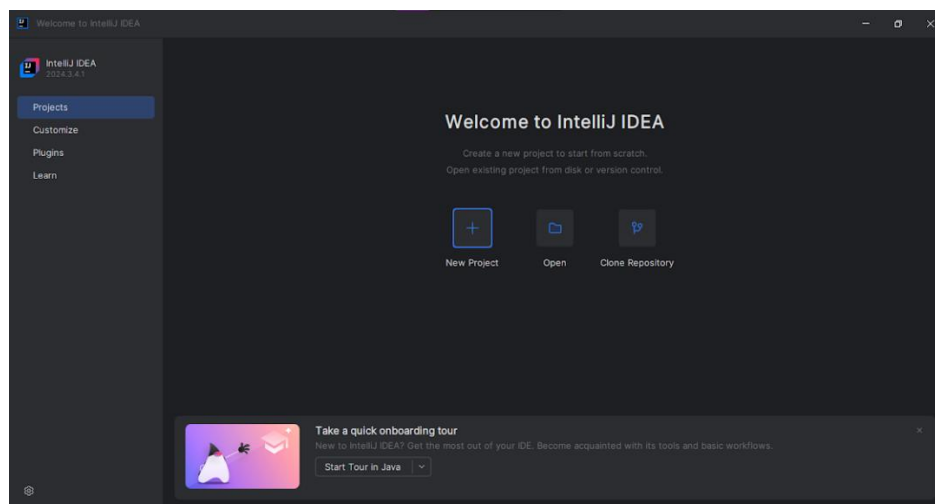
Finalización de la instalación de IntelliJ IDEA.



Al ejecutar IntelliJ IDEA por primera vez, se muestra una ventana mostrada en la Figura 12:

Figura 12

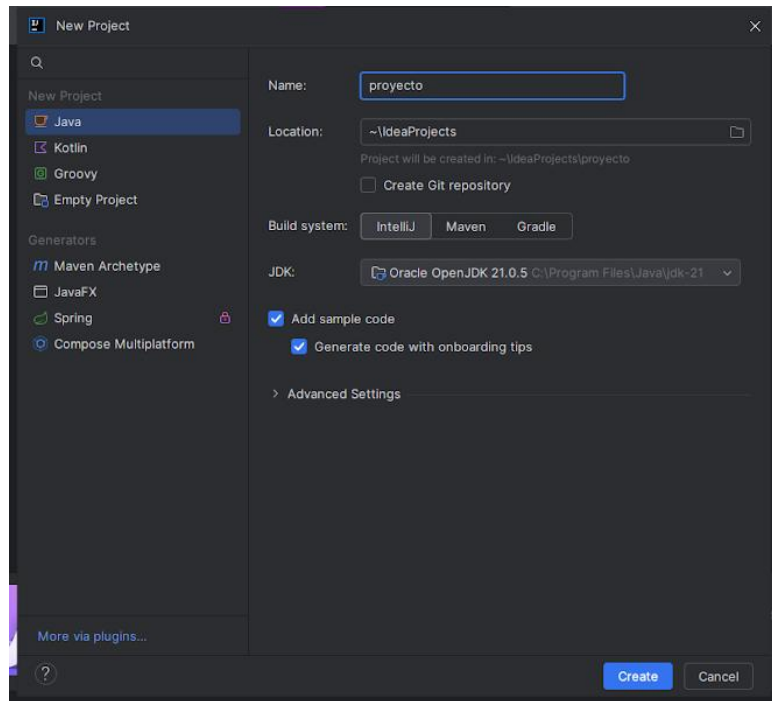
Ejecución de IntelliJ IDEA.



El procedimiento para crear un nuevo proyecto se representa en la Figura 13:

Figura 13

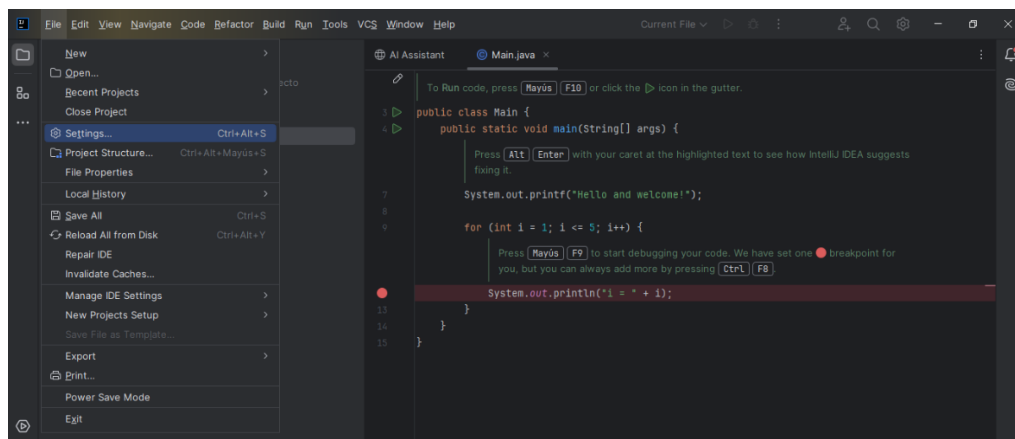
Creación de un nuevo proyecto en IntelliJ IDEA.



Al crear el proyecto, se accede a la barra de tareas en “File” y se busca “Settings”, se visualiza en la Figura 14:

Figura 14

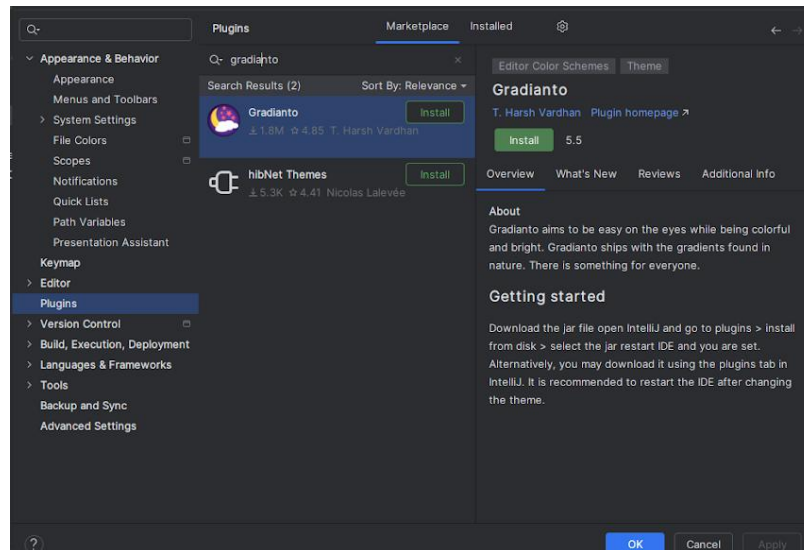
Entorno de IntelliJ IDEA.



La Figura 15 muestra cómo acceder a la sección de plugins para personalizar el IDE, se localiza "Gradianto" dentro de la sección de Plugins:

Figura 15

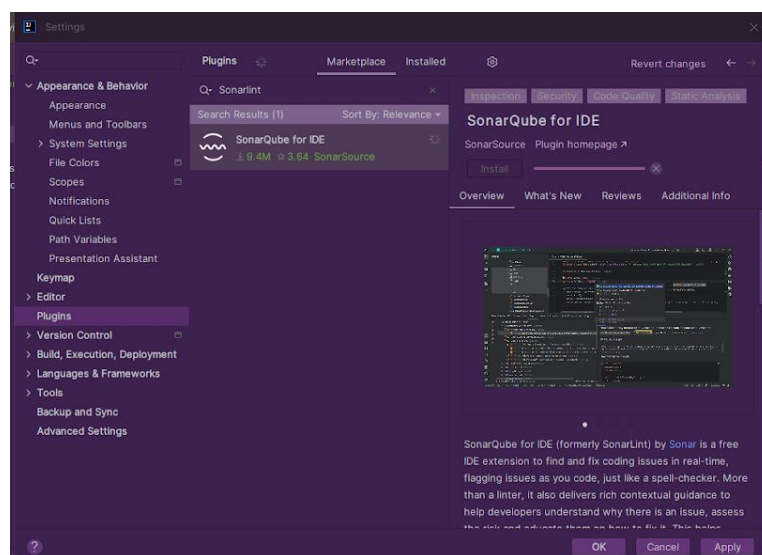
Sección de Plugins en IntelliJ IDEA.



Para mejorar la calidad del código, se instala Sonarlint como se observa en la Figura 16:

Figura 16

Plugin Sonarlint.

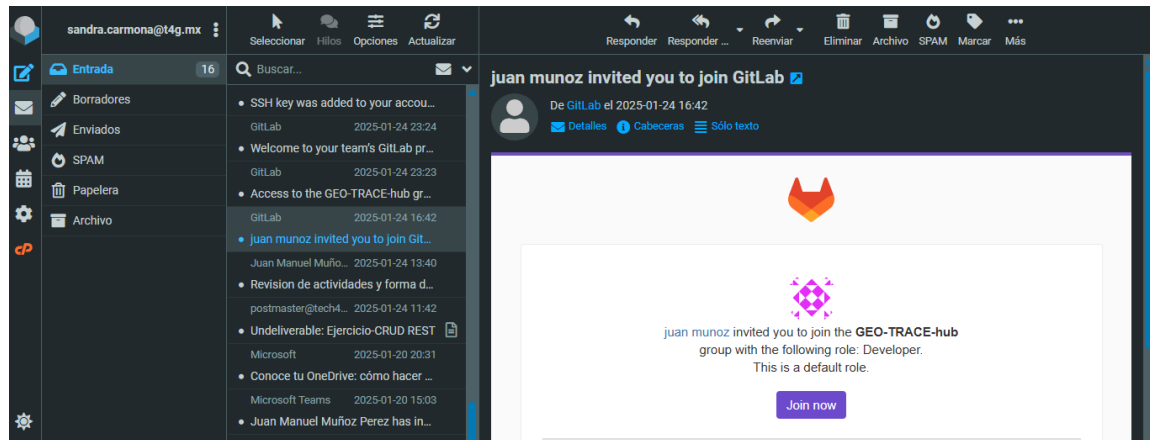


3.1.3. Uso de GitLab

Para utilizar *GitLab*, el Scrum Master, administrador o dueño del proyecto envía una invitación a la cuenta empresarial para otorgar acceso al repositorio como la de la Figura 17:

Figura 17

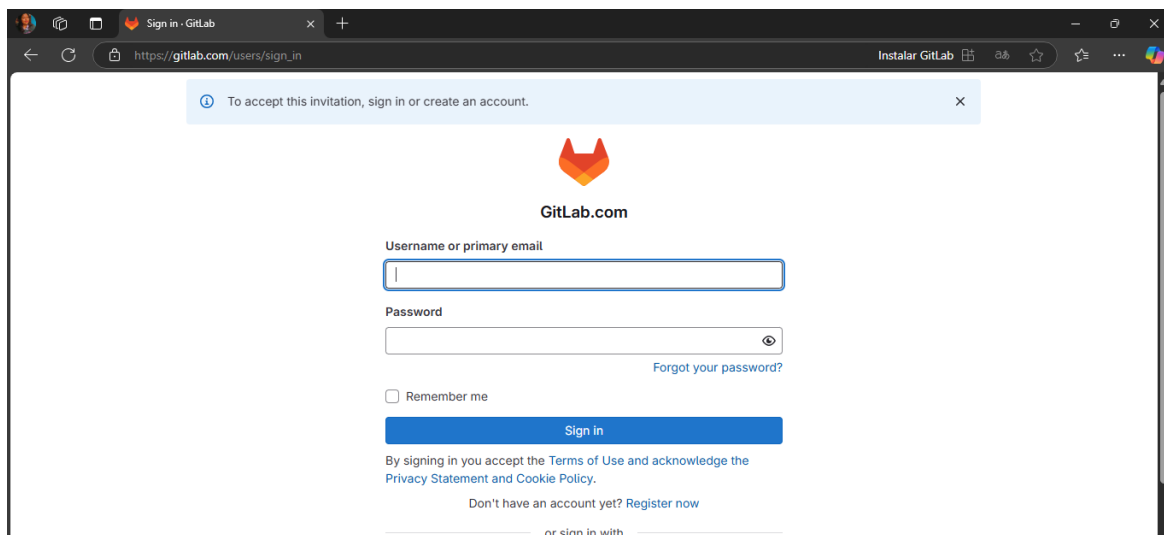
Invitación a GitLab por email.



El acceso a la plataforma se ilustra en la Figura 18:

Figura 18

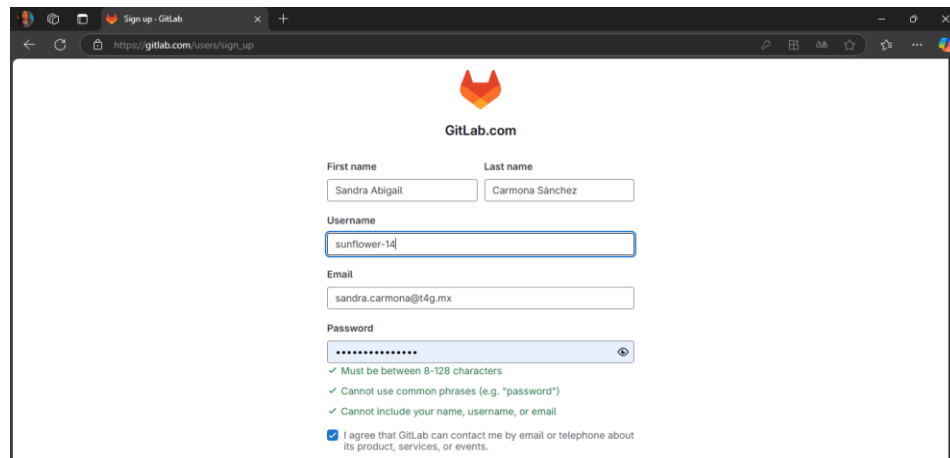
Inicio de Sesión en GitLab.



La ausencia de una cuenta implica que se tendrá que crear una en el apartado "Register now." Esto redirigirá a un formulario de registro donde se ingresan el nombre, apellidos, un nombre de usuario, el correo electrónico proporcionado por la empresa y una contraseña. Una vez que los datos son correctos, se procede a crear la cuenta. El formulario de registro de GitLab se presenta en la Figura 19:

Figura 19

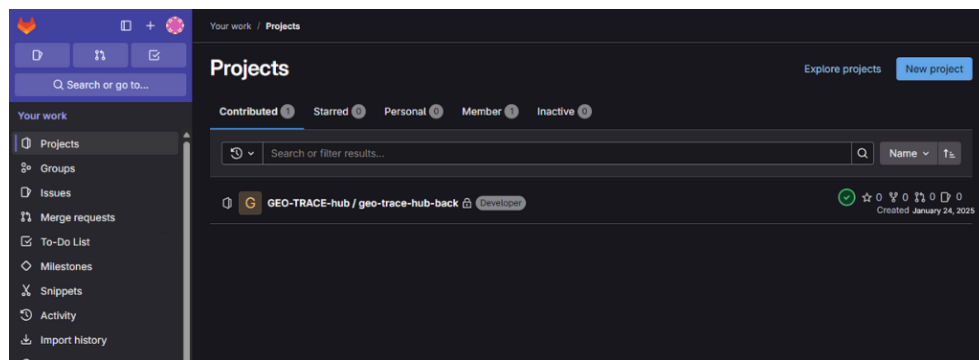
Formulario de registro en GitLab.

A screenshot of the GitLab sign-up page in a web browser. The page has a white background with the GitLab logo at the top center. Below the logo, the text "GitLab.com" is displayed. The form consists of several input fields: "First name" (containing "Sandra Abigail"), "Last name" (containing "Carmona Sánchez"), "Username" (containing "sunflower-14"), "Email" (containing "sandra.carmona@t4g.mx"), and "Password" (containing masked characters). Below the password field, there are three green checkmarks indicating password requirements: "Must be between 8-128 characters", "Cannot use common phrases (e.g. 'password')", and "Cannot include your name, username, or email". At the bottom, there is a checkbox labeled "I agree that GitLab can contact me by email or telephone about its product, services, or events." which is checked.

Acto seguido, el usuario será redirigido al panel principal, en el cual se visualiza el repositorio del proyecto. El panel de inicio de GitLab se muestra en la Figura 20:

Figura 20

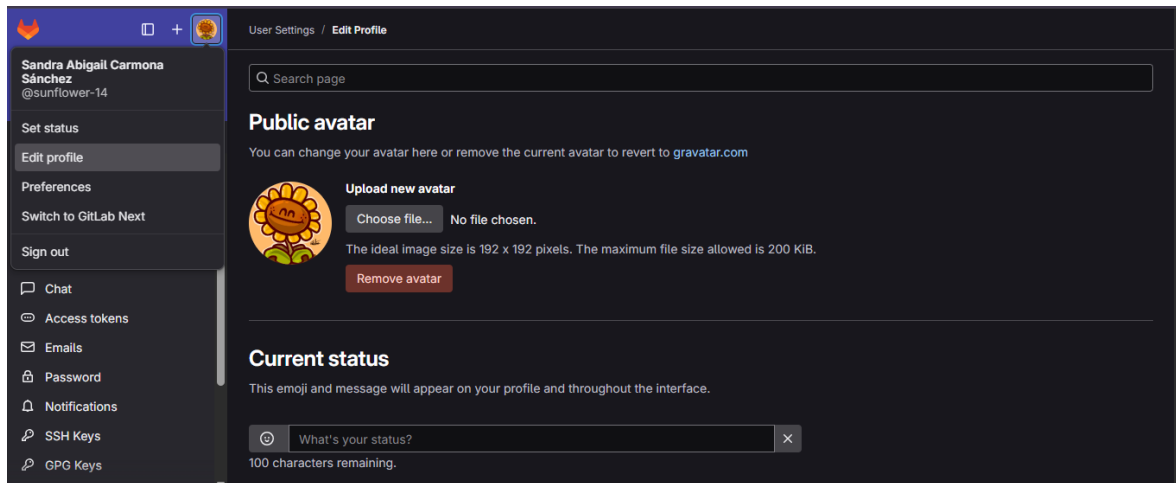
Panel de GitLab.



Para identificar a la cuenta, se cambia la foto de perfil en “Edit profile”. El área para editar el perfil se ubica como se indica en la Figura 21:

Figura 21

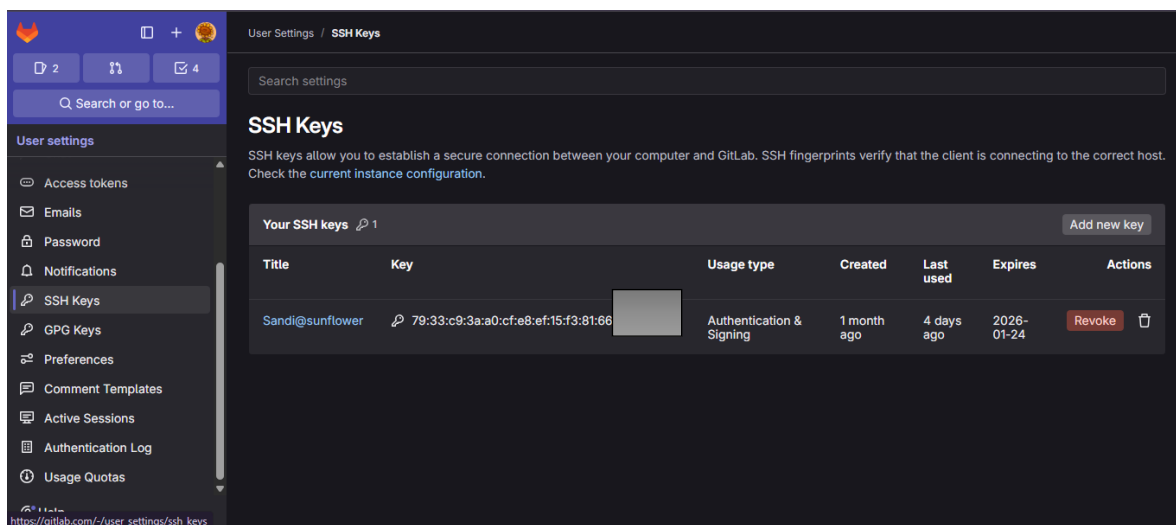
Sección de edición del perfil en GitLab.



La sección de claves SSH se puede observar en la Figura 22 que permite incorporar la llave pública del dispositivo, lo que permite clonar, realizar commits y ejecutar merge request de forma directa:

Figura 22

Sección de SSH Keys.

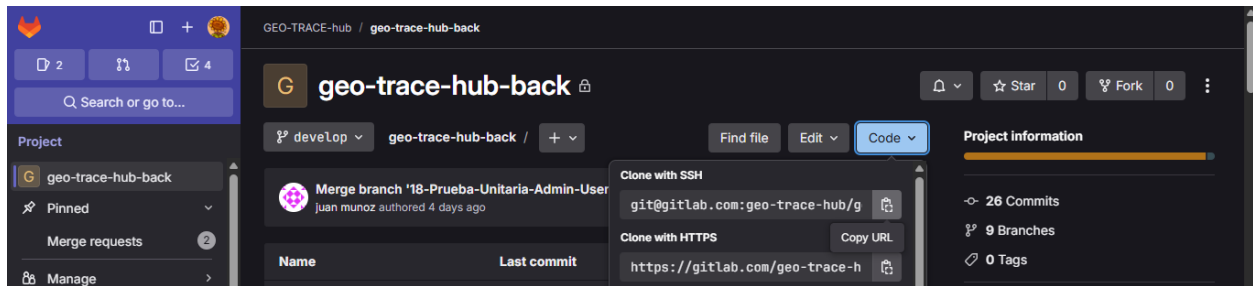


3.1.4. Clonar el repositorio del proyecto en la máquina local

Este proceso comienza al acceder al repositorio y desde el botón "Code" se copia en la sección "Clonar con SSH". El repositorio del proyecto en GitLab se visualiza en la Figura 23:

Figura 23

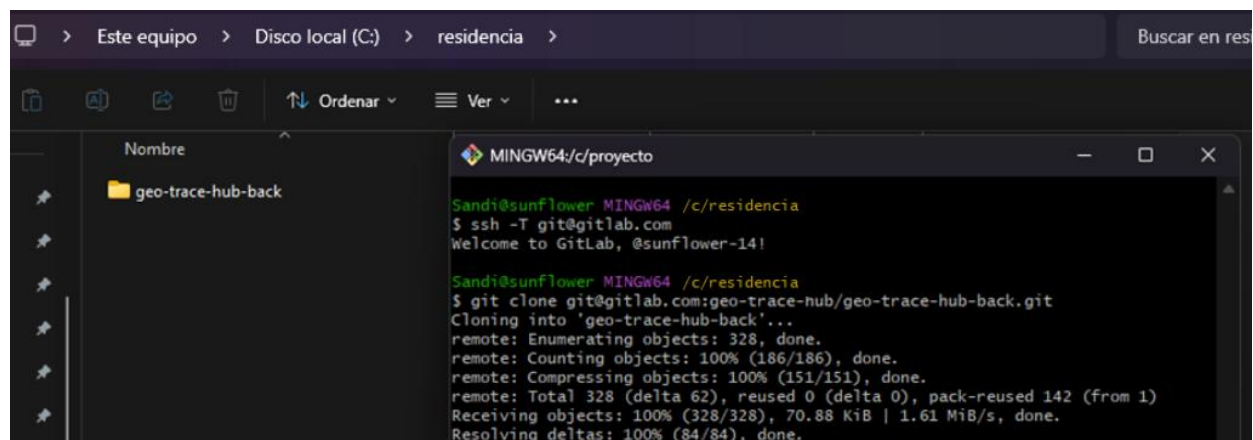
Repositorio del proyecto en GitLab.



Posteriormente en la Figura 24 se crea una carpeta donde se alojará el proyecto. Dentro de esta carpeta, se abre una consola Git Bash y se pega la instrucción para clonar. Una vez clonado, el proyecto se muestra la carpeta del proyecto localmente.

Figura 24

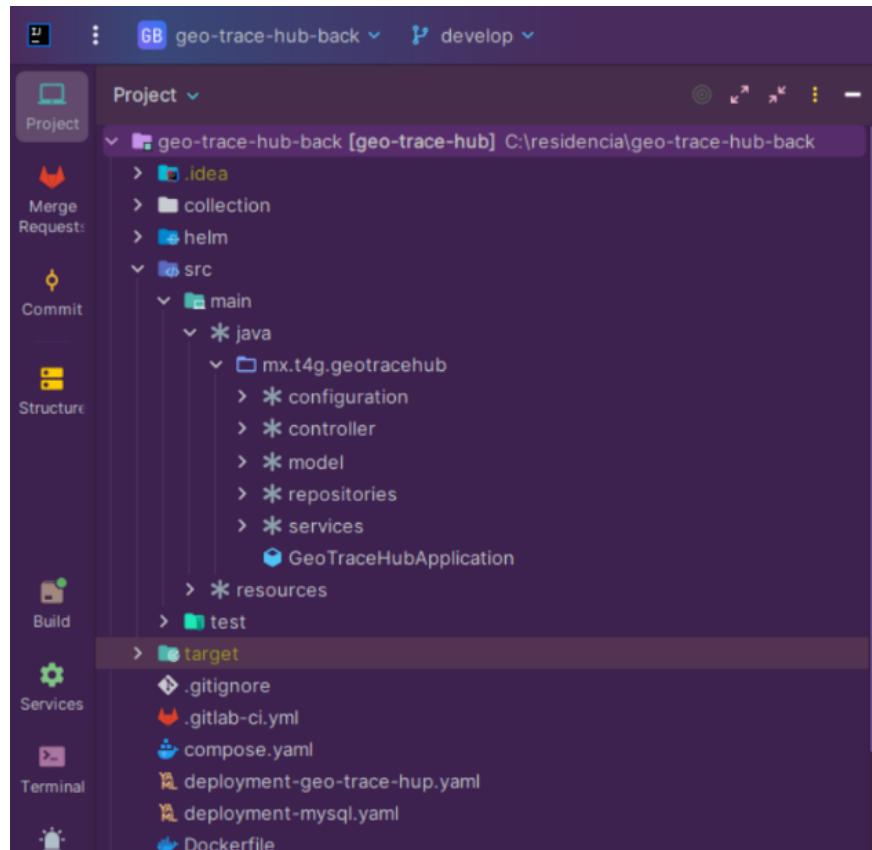
Proyecto almacenado de forma local.



Se abre en IntelliJ el proyecto, visualizando la estructura de carpetas de este. La estructura de carpetas se representa en la Figura 25.:

Figura 25

Estructura de carpetas del proyecto en IntelliJ IDEA.

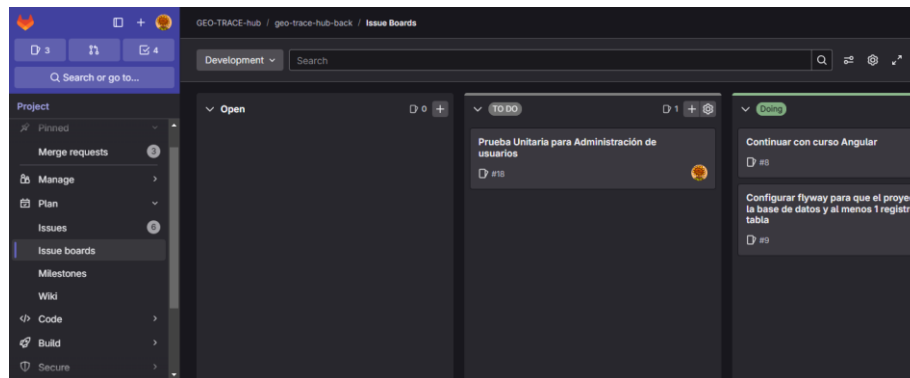


3.1.5. Kanban en GitLab

En GitLab incluye la herramienta “Plan”, para crear Issues o problemas a resolver y las permite visualizar en forma de un tablero Kanban en Issue boards. La organización de tareas se visualiza en la Figura 26:

Figura 26

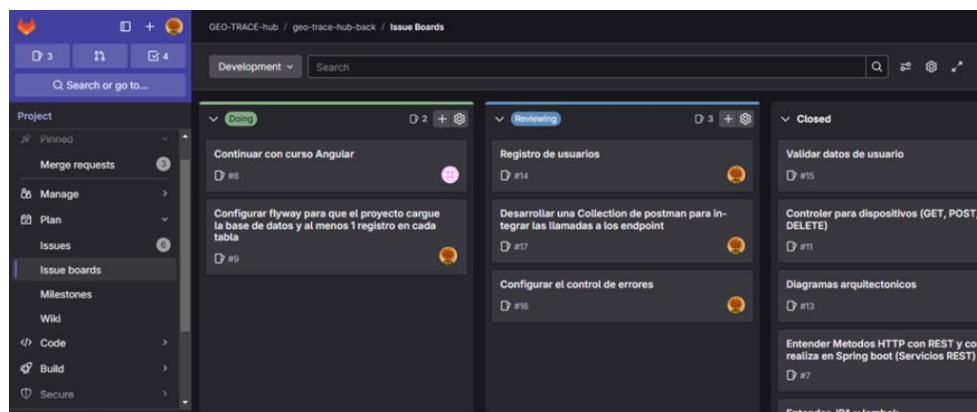
Tableros Kanban en GitLab.



Estos tableros son personalizables, permiten configurar distintas categorías. El proyecto contiene las siguientes: “Open” para tareas a realizar; “To Do” representando la lista de tareas por hacer; “Doing” representan las tareas en curso; “Reviewing” tareas que se finalizaron pero se necesita la revisión del Scrum Master y “Closed” para tareas que fueron aprobadas. Al crear un nuevo problema o asignación, es posible asignar un responsable a la tarea así como enumerarlas, convirtiéndolas en tickets por gestionar. Los tickets de tareas se presentan en la Figura 27:

Figura 27

Tickets en los tableros en GitLab.



3.1.6. Análisis

Como parte del análisis en el diseño del *backend*, se han establecido los requerimientos funcionales y no funcionales que permitirán guiar su desarrollo e implementación. Esta clasificación proporciona una base clara e incluso estructurada para asegurar que la solución sea coherente, escalable así como alineada con los objetivos planteados en el proyecto.

Requerimientos funcionales

Los requerimientos funcionales describen las operaciones esenciales que debe realizar el *backend REST* para cumplir con las necesidades del usuario. La Tabla 1 expone los requerimientos funcionales para el desarrollo del *backend*:

Tabla 1

Requerimientos funcionales del backend.

Clave	Requerimiento	Descripción
RF01	Registro de usuarios	El sistema debe permitir registrar nuevos usuarios, validando los campos ingresados.
RF02	Consulta de usuarios	El sistema debe permitir visualizar todos los usuarios registrados o buscar uno por su ID.
RF03	Actualización de usuarios	Los usuarios deben poder actualizar su información personal.
RF04	Activación/Inactivación de cuenta	El sistema debe permitir desactivar o reactivar cuentas de usuario sin eliminarlas.
RF05	Eliminación de usuario	El administrador debe poder eliminar usuarios, siempre que no tengan dispositivos asociados.
RF06	Registro de dispositivo	Los usuarios pueden agregar nuevos dispositivos al sistema.
RF07	Consulta de dispositivos	El sistema permite consultar únicamente los dispositivos asociados a un usuario específico o mediante el ID de un dispositivo.
RF08	Consulta de dispositivo por ID	Los usuarios deben poder buscar un dispositivo específico mediante su ID.
RF09	Actualización de dispositivo	El sistema debe permitir la modificación de los datos de un dispositivo.

RF10	Activación/Inactivación de dispositivo	Los dispositivos pueden activarse o desactivarse en lugar de ser eliminados.
RF11	Eliminación de dispositivo	El sistema permite eliminar dispositivos registrados.
RF12	Registro de ubicaciones	El sistema debe permitir registrar nuevas posiciones geográficas enviadas por dispositivos.
RF13	Consulta de ubicaciones	El sistema debe permitir consultar el historial completo de ubicaciones de un dispositivo.
RF14	Filtrado de ubicaciones	Las ubicaciones deben poder filtrarse por rangos de fechas.
RF15	<i>API REST</i> accesible	Toda la funcionalidad debe estar expuesta mediante servicios <i>RESTFUL</i> para interacción con apps.

Nota. Esta tabla presenta los requerimientos funcionales identificados para el desarrollo del *backend REST* del sistema Geo Trace Hub, agrupados por operaciones clave del sistema.

Requerimientos no funcionales

Los requerimientos no funcionales definen las condiciones técnicas, de calidad y rendimiento bajo las cuales debe operar. La Tabla 2 muestra los requerimientos no funcionales para el desarrollo del *backend*:

Tabla 2

Requerimientos no funcionales del backend.

Clave	Requerimiento	Descripción
RNF01	Tecnología <i>backend</i>	El sistema debe estar desarrollado usando Spring Boot y Java.
RNF02	Persistencia de datos	Debe usarse MySQL como sistema de gestión de bases de datos.
RNF03	Documentación <i>OpenAPI</i>	Los <i>endpoints</i> deben estar documentados usando el estándar OpenAPI.
RNF04	Pruebas de Calidad	El sistema debe incluir pruebas unitarias, de integración y de aceptación.
RNF05	Control de versiones	Se debe usar GitLab para el control de versiones del código fuente.
RNF06	Escalabilidad	El sistema debe estar diseñado para escalar eficientemente en futuros desarrollos.
RNF07	Interoperabilidad	El <i>backend</i> debe poder integrarse con aplicaciones móviles y frontales a través de la <i>API REST</i> .

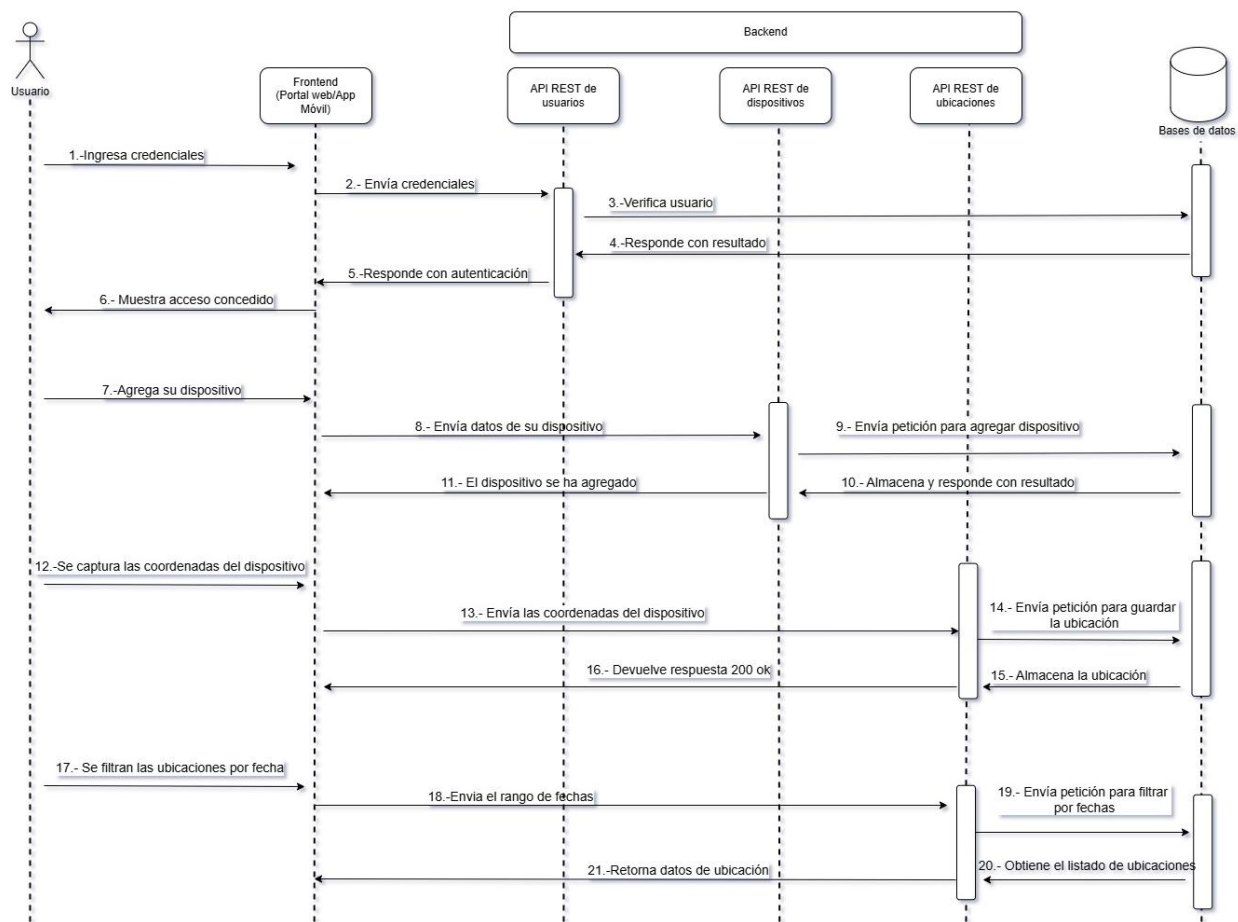
Nota. Esta tabla describe los requerimientos no funcionales que aseguran las condiciones de calidad, escalabilidad, documentación y mantenimiento del sistema desarrollado.

3.1.7. Diseño de la arquitectura del sistema

Antes de comenzar a construir el código, se utiliza la arquitectura de Kruchten “4 + 1”, para crear los planos adecuados del comportamiento del sistema en sus diferentes perspectivas. La primera vista, es la “Vista lógica” plasmada en el diagrama de secuencias UML, en el que se contempla al usuario, al *frontend*, a las *API REST* y la base de datos (*backend*). La lógica del sistema se representa mediante el diagrama de la Figura 28:

Figura 28

Vista lógica mediante diagrama de secuencias UML.



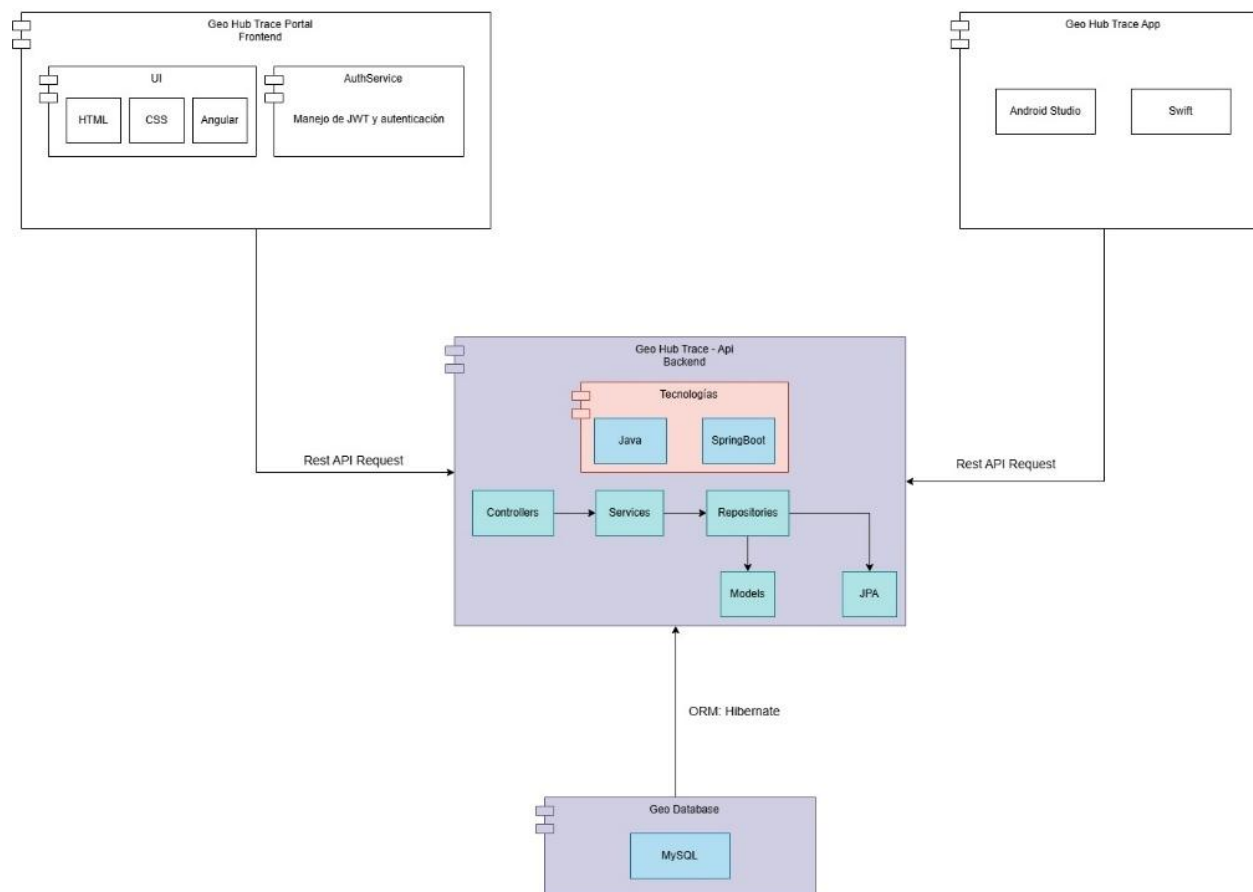
En la que se muestra el flujo de cómo el usuario usará el sistema. El usuario al ingresar al sistema verá su información correspondiente (debido a que la base de datos respondió correctamente) y al mostrarle su información llama al servicio *REST* de usuarios, luego si el usuario ingresa un nuevo dispositivo llamará a la *API* de dispositivos que almacenará en la base de datos este nuevo dispositivo (aplicará para las demás operaciones CRUD para la administración de dispositivos),

luego se capturan las coordenadas de la ubicación que se vayan registrando el dispositivo almacenándolos en la base de datos, en el que será posible por el servicio *REST* de la administración de ubicaciones, luego de que estas posiciones se guarden, posteriormente, se podrán filtrarse por fecha para visualizar el historial de las posiciones guardadas.

La segunda vista es, la “Vista de desarrollo” en la que se construyó usando el diagrama de componentes UML. Se visualiza cómo se organizan y relacionan los diferentes componentes del sistema Geo Trace Hub. La parte que está coloreada es la que se realizará en esta fase del proyecto. La arquitectura se detalla en el diagrama de la Figura 29:

Figura 29

Vista de desarrollo mediante diagrama de componentes UML.



El esquema se compone de cuatro componentes importantes. En primer lugar, se abordará el *backend* junto con la base de datos, ilustrando la comunicación que hay entre ambos, así como las tecnologías que permiten que los componentes funcionen. El componente del *backend* se

desarrolla mediante el lenguaje Java y framework Spring Boot, implementando una arquitectura por capas para garantizar un código organizado.

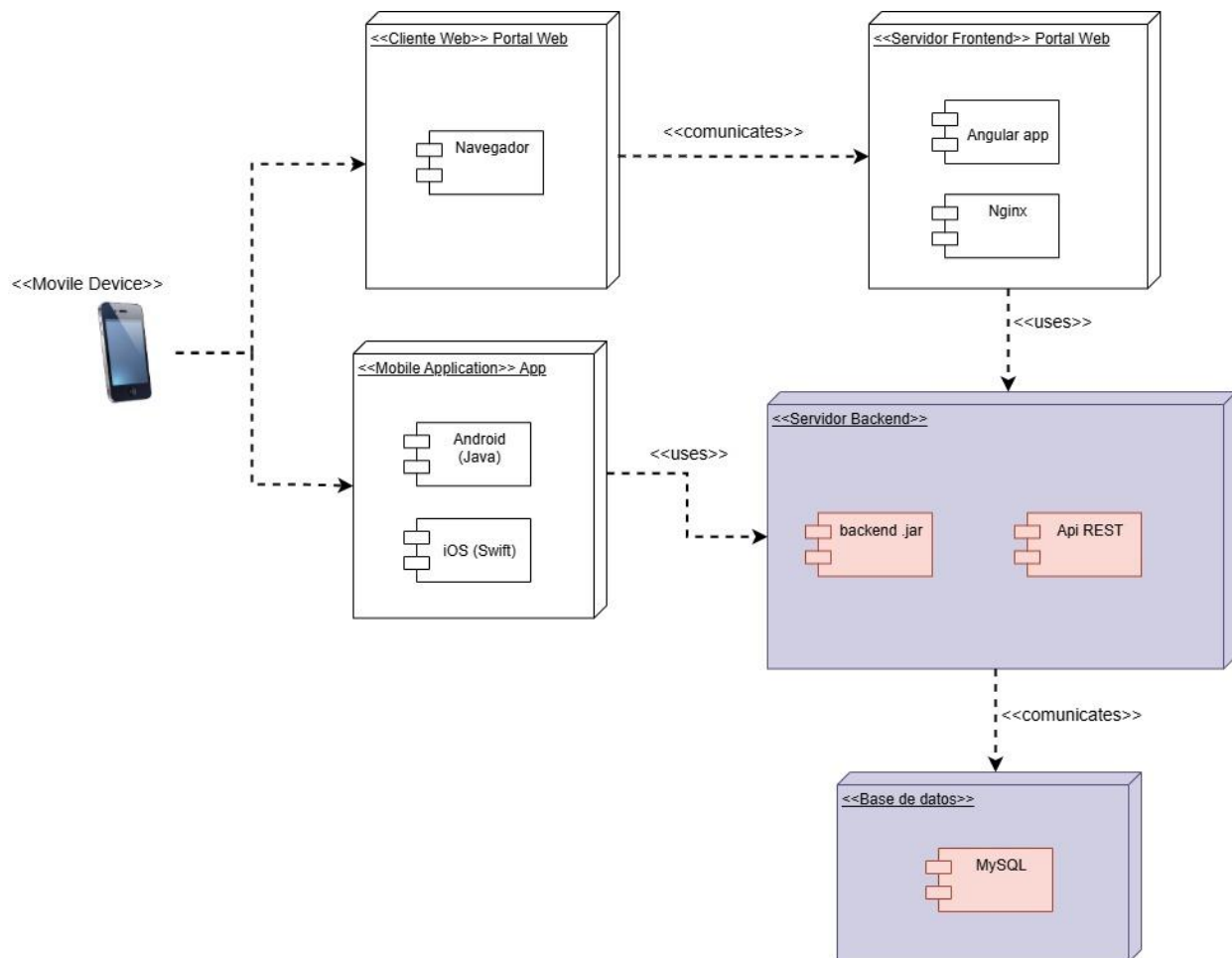
Los controladores interactuarán con los servicios que a su vez utilizarán los repositorios. Estos repositorios extenderán JPA para realizar consultas en la base de datos, mismos que emplearán los modelos de datos para la creación de tablas en MySQL, conectándose de manera segura como eficiente mediante Hibernate al gestor de la base de datos y a la aplicación backend.

Asimismo, en el diagrama se han diseñado los componentes para el frontend y la aplicación móvil, los cuales se implementarán en fases posteriores del proyecto, que se conectarán a este *backend REST*.

La tercera perspectiva es la "Vista física", representada por un diagrama de despliegue UML. Este diagrama consta de cinco nodos principales, de los cuales en la fase actual del proyecto se implementarán el nodo "Servidor Backend" así como el nodo "Base de datos". Es importante destacar que el *backend* es una *API REST* que se comunicará con la base de datos en MySQL. Por su parte, los nodos "Servidor Frontend" y "Mobile Application" consumirán la *API REST* para el almacenamiento de la información de los usuarios. El entorno de ejecución se representa en la Figura 30:

Figura 30

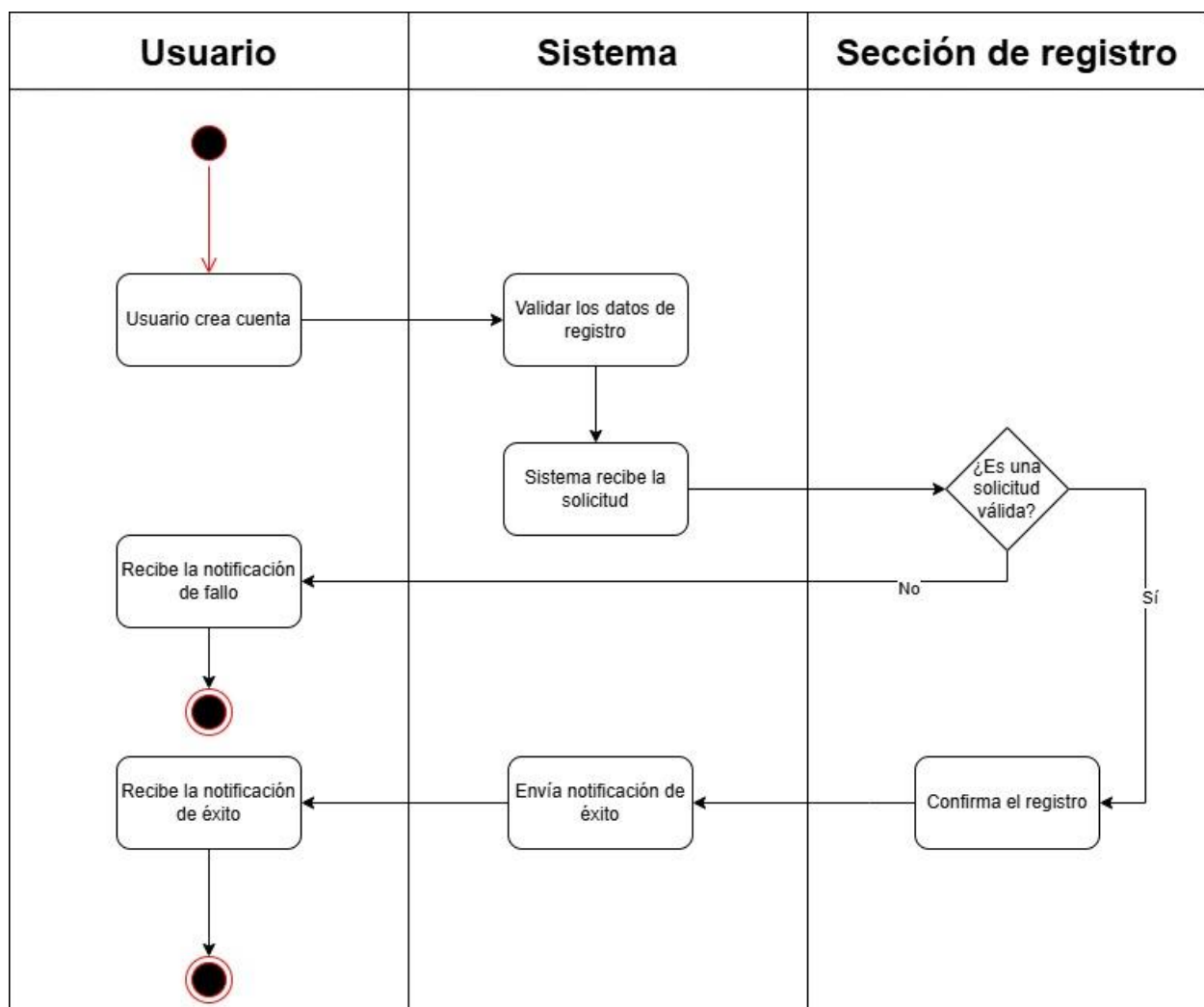
Vista física mediante diagrama de despliegue UML.



La cuarta vista, la “Vista de procesos”, utiliza el diagrama de actividades para plasmar el flujo que se desarrollará en el *backend REST*. Inicialmente, se detalla el proceso para añadir un nuevo usuario al sistema. El sistema validará la solicitud y si no es válida, notificará al usuario que uno de los campos de registro no se recibió correctamente. Si la solicitud es válida, el registro se almacenará y se informará al usuario que su registro fue exitoso. La creación de cuentas se describe en la Figura 31:

Figura 31

Vista de procesos para crear una cuenta.

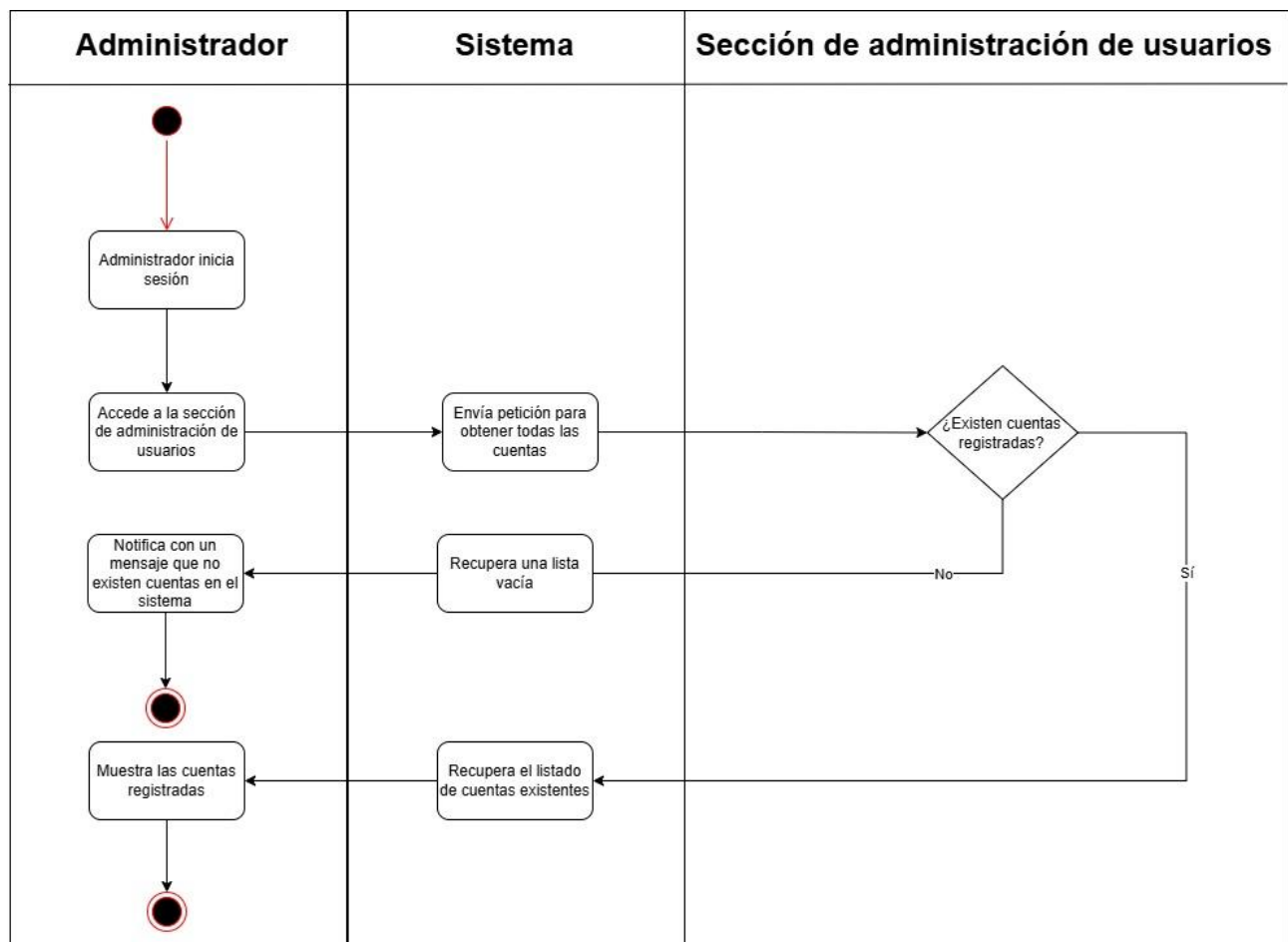


La Figura 32 representa una operación fundamental que se utiliza en el desarrollo del *backend REST*: la recuperación de todos los registros de usuarios existentes. Esta función permitirá visualizar un listado de las cuentas registradas.

Existirá una sección de administración de usuarios que permitirá solicitar al sistema la obtención de estas cuentas. Si no hay cuentas registradas, el sistema notificará al usuario. En el caso contrario, es decir, si existen cuentas, el sistema devolverá una lista de los usuarios registrados:

Figura 32

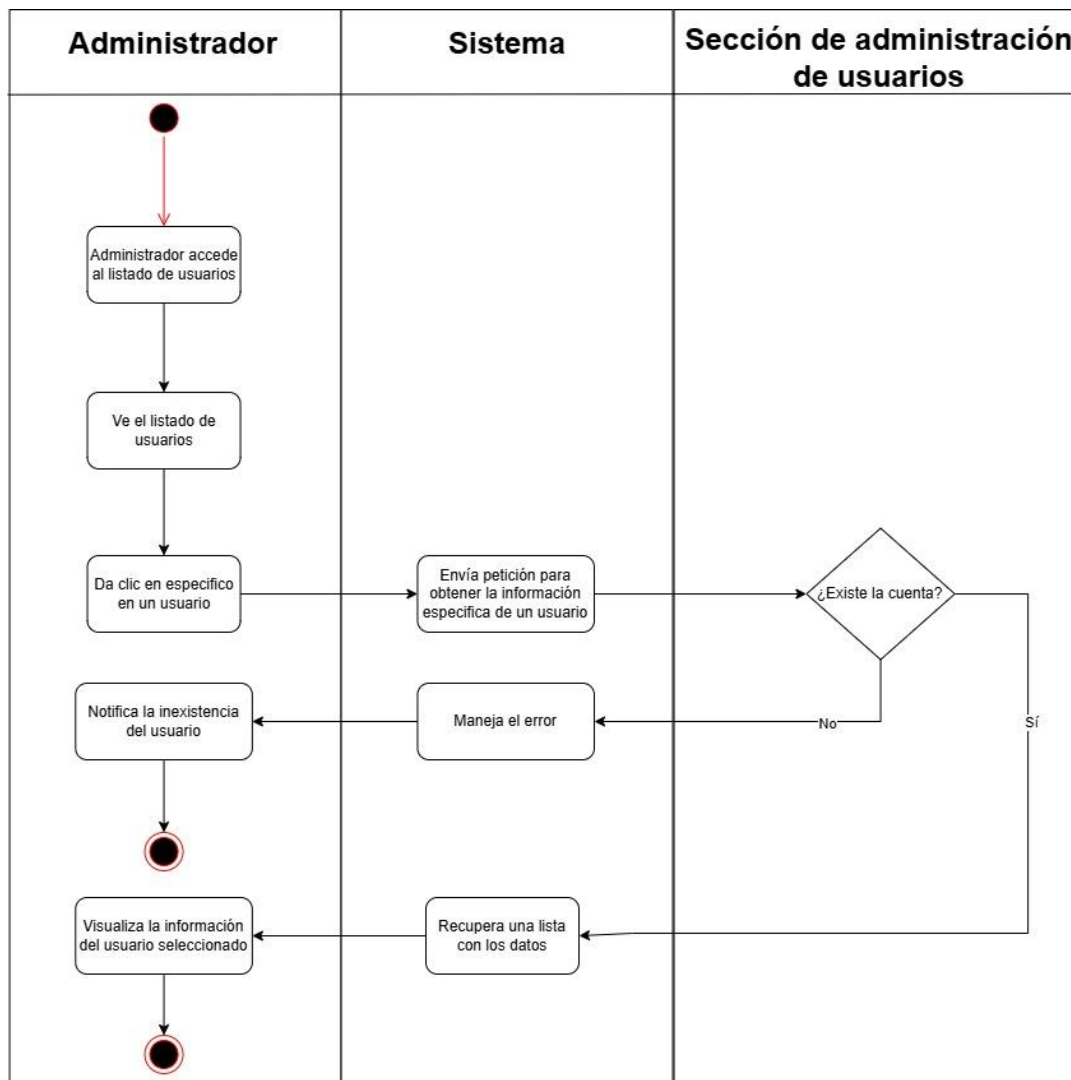
Vista de procesos para obtener las cuentas.



Otra operación crucial en el *backend* es la búsqueda de un usuario por su ID. Esta funcionalidad permitirá al administrador visualizar de forma individual la información de un solo usuario. De igual manera, esta búsqueda es útil para que los usuarios puedan acceder a su propia información, en lugar de ver los datos de todas las cuentas existentes en el sistema. La obtención de datos de un usuario se ilustra en la Figura 33:

Figura 33

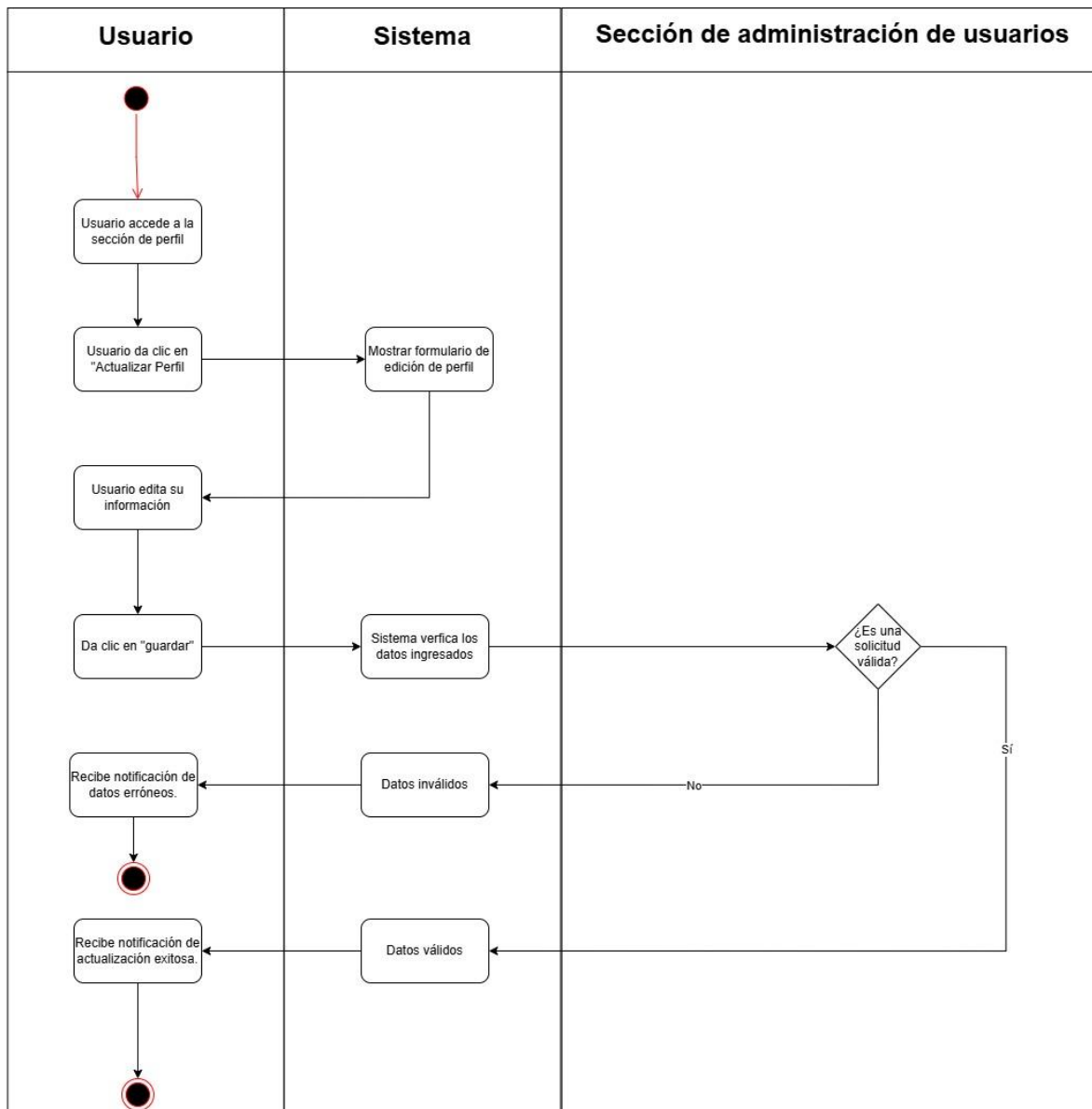
Vista de procesos para obtener la información de un usuario.



La Figura 34 representa el proceso de actualización de un usuario. Establece el flujo de la actualización de la información de un usuario. El usuario podrá editar su información guardada. El sistema verificará que la solicitud sea válida.

Figura 34

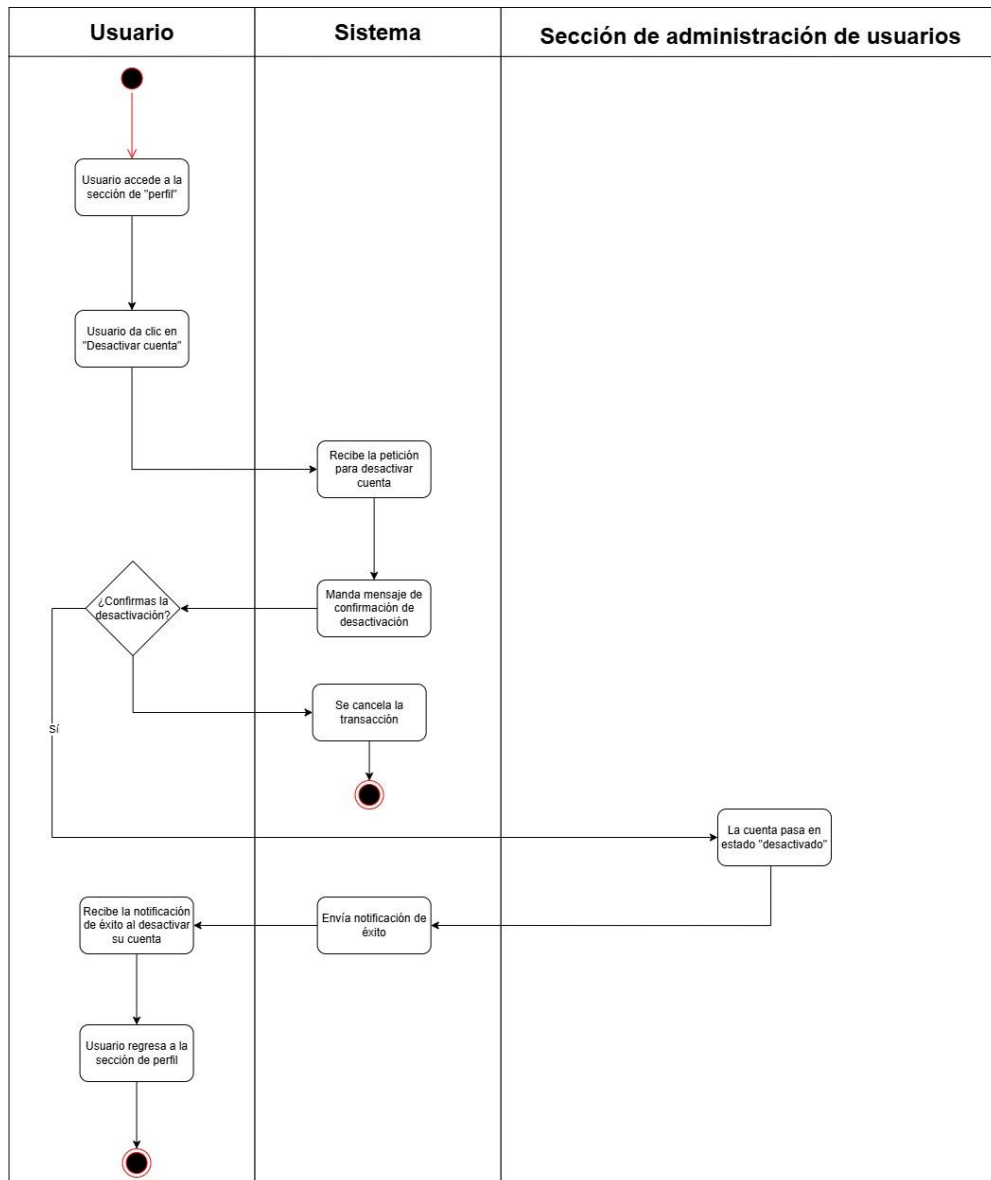
Vista de procesos para actualizar un usuario.



El cambio de estado de un usuario se visualiza en la Figura 35 que muestra el flujo mediante el cual un usuario puede desactivar su cuenta, en lugar de eliminarla por completo. Esto cambia el estado de la cuenta a "inactivo". El mismo flujo se aplicaría para volver a habilitar la cuenta del usuario:

Figura 35

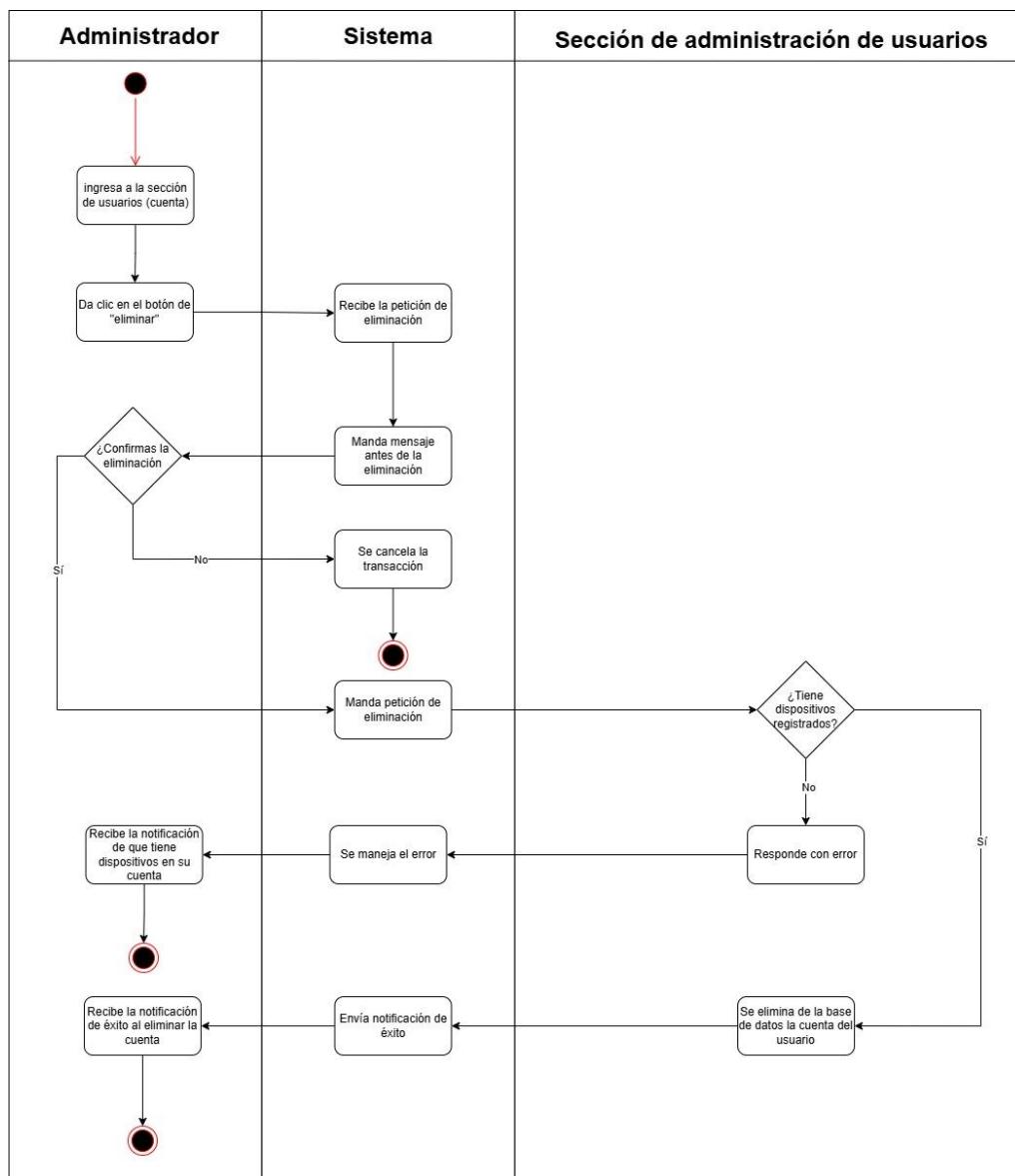
Vista de procesos para cambiar el estado de un usuario.



Para finalizar la sección de administración de usuarios, la Figura 36 detalla el proceso de eliminación de un usuario. Únicamente el administrador tendrá la facultad de eliminar cuentas. Si la cuenta del usuario tiene dispositivos asociados (debido a llaves foráneas), no se podrá eliminar. Sin embargo, si no hay información ligada, la cuenta se eliminará sin inconvenientes:

Figura 36

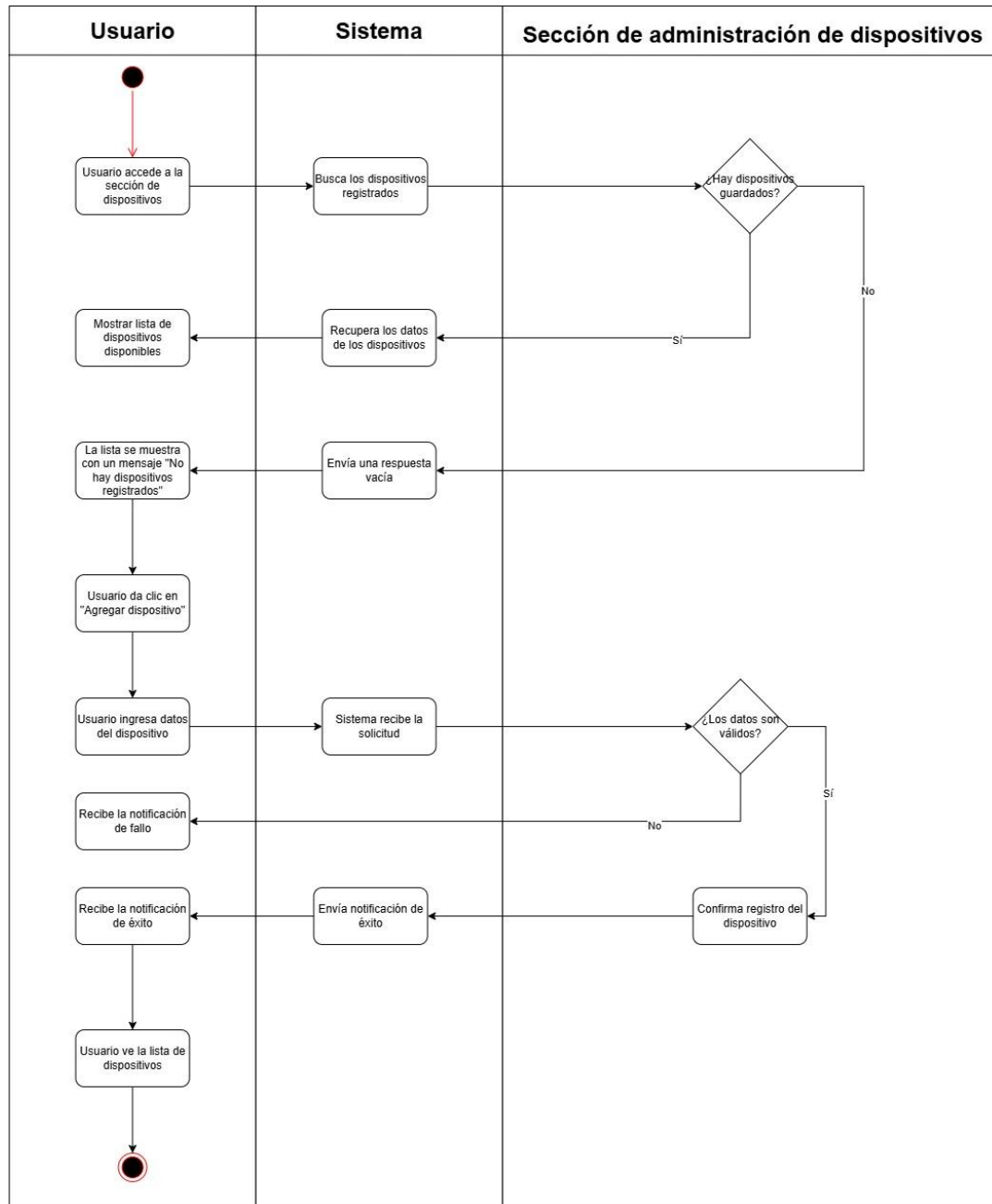
Vista de procesos para eliminar a un usuario del sistema.



Ahora, en la sección de administración de dispositivos, la Figura 37 ilustra cómo que el usuario y el sistema agregan un nuevo dispositivo:

Figura 37

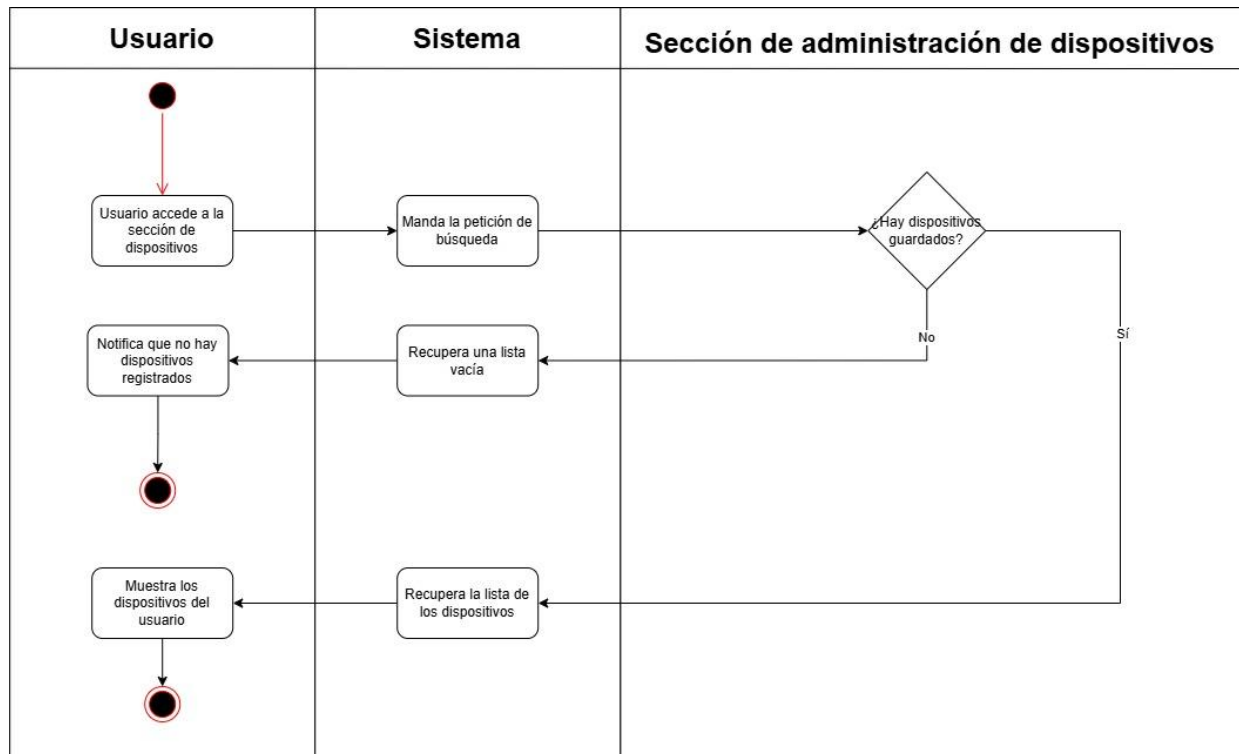
Vista de procesos para añadir un nuevo dispositivo al sistema.



La Figura 38 muestra cómo un usuario puede visualizar el listado de sus dispositivos almacenados en el sistema. Si existen dispositivos, se mostrarán en un listado; de lo contrario, el sistema notificará al usuario que no hay dispositivos, lo que a nivel de *backend* se representa como una lista vacía:

Figura 38

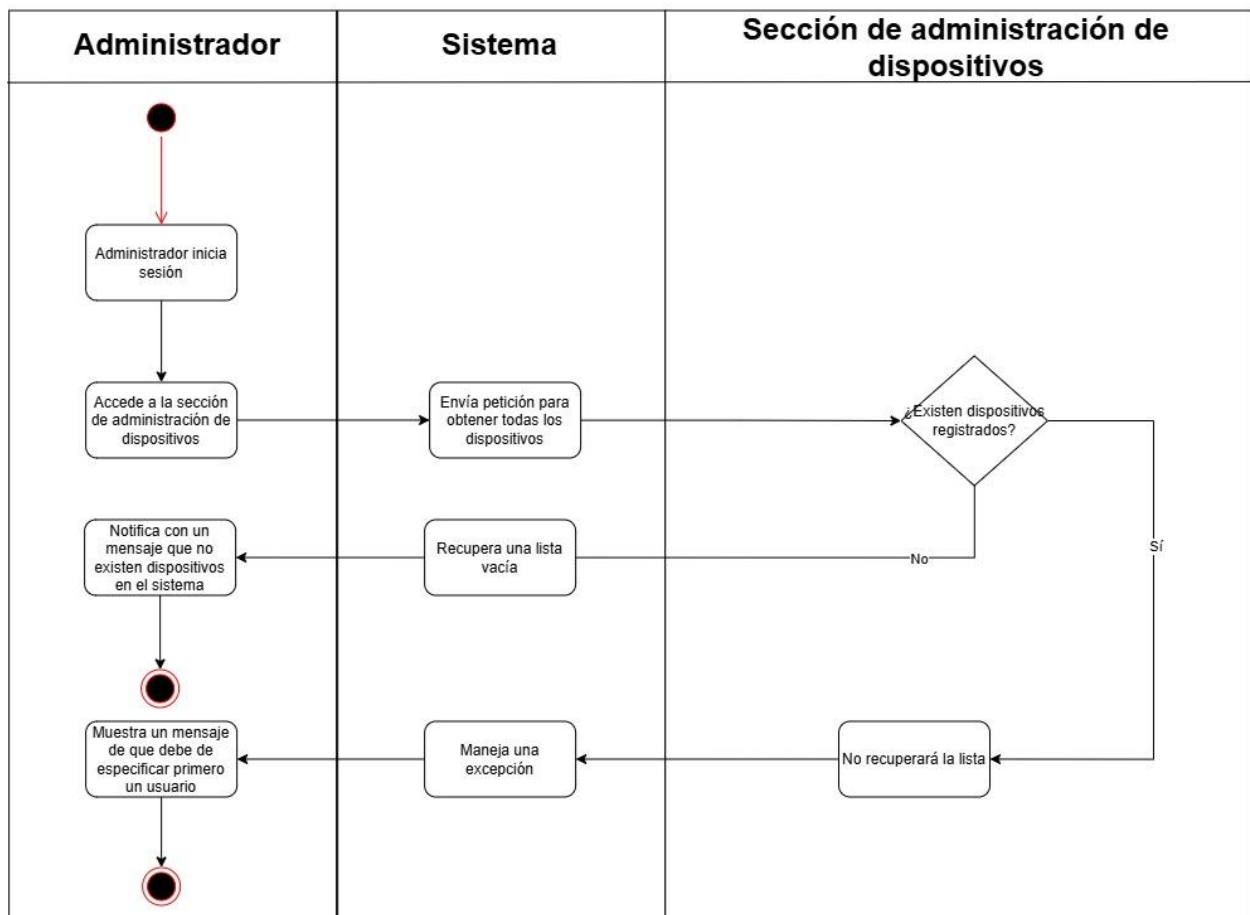
Vista de procesos en el que se visualizan los dispositivos de un solo usuario.



Posteriormente, se presenta la obtención de todos los dispositivos. Cabe destacar que esta funcionalidad se diseñó de acuerdo a los requerimientos funcionales para no permitir la recuperación directa de todos los dispositivos guardados en el sistema. Esta decisión permite facilitar la creación de procesos específicos, como la búsqueda de un dispositivo por su ID o la visualización de dispositivos por usuario. Además, ni el administrador ni los usuarios tendrán acceso para recuperar el listado completo de dispositivos previamente almacenados. El proceso para obtener todos los dispositivos se presenta en la Figura 39:

Figura 39

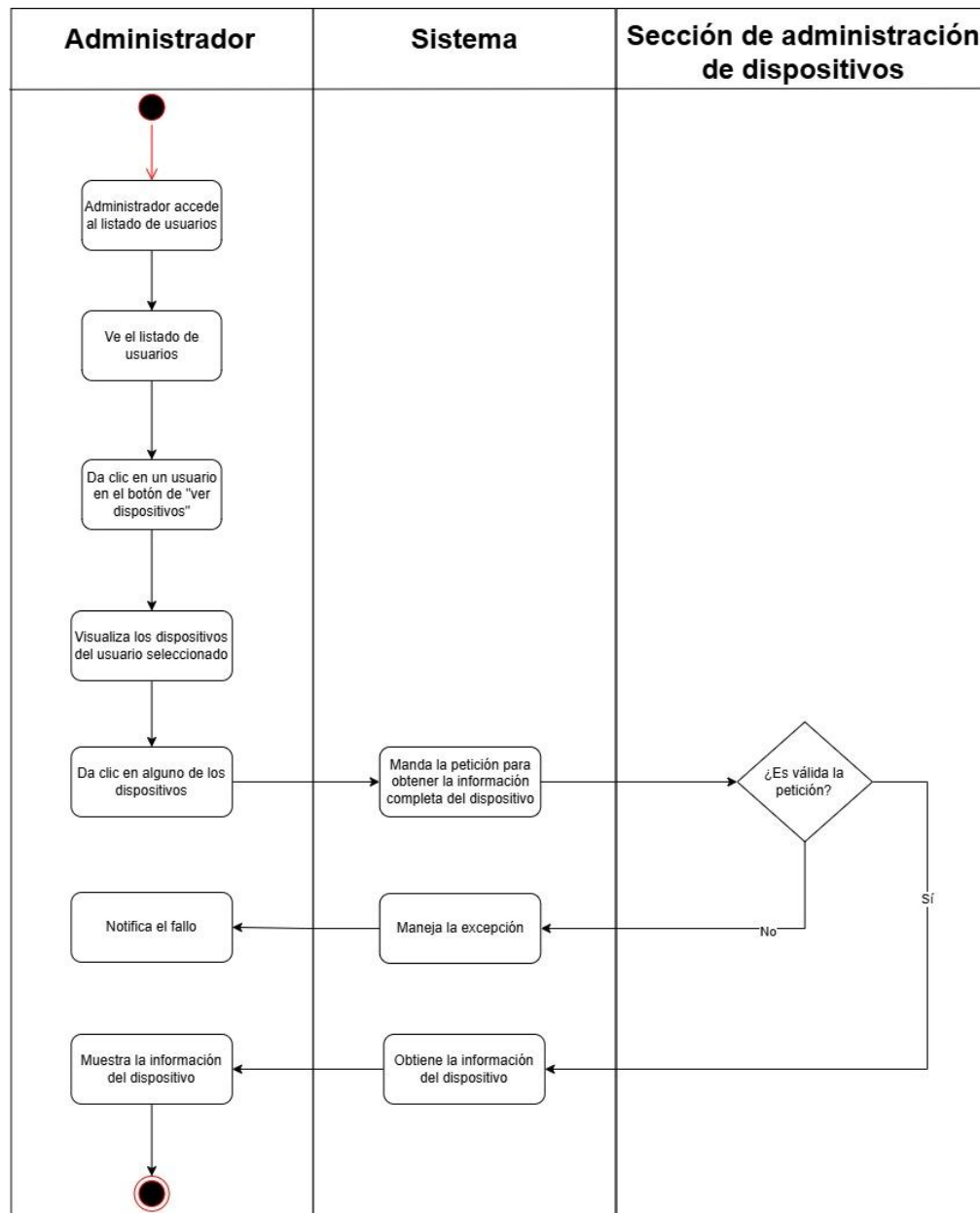
Vista de procesos para obtener todos los dispositivos.



La búsqueda de un dispositivo por su ID se ilustra en la Figura 40 que representa la obtención de un dispositivo por ID, que realiza la búsqueda de dispositivos para visualizar su información. Si el dispositivo no es encontrado, se notificará al usuario:

Figura 40

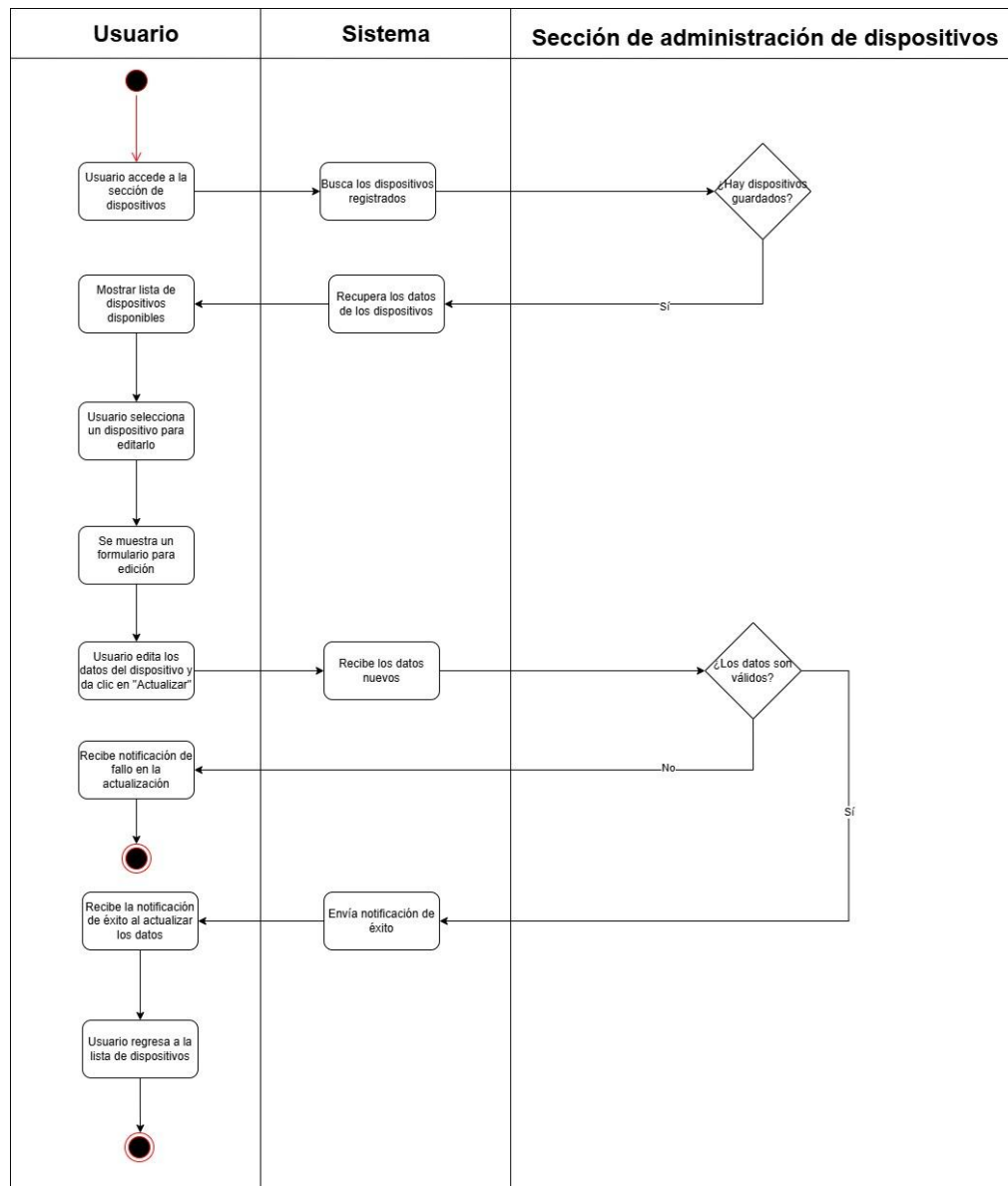
Vista de procesos que muestra la búsqueda de un dispositivo por ID.



La Figura 41 muestra cómo el usuario puede editar los detalles de un dispositivo. Primero, el sistema busca si existen dispositivos. Si al actualizar los datos del dispositivo, estos son correctos, la operación se realizará con éxito. En caso contrario, se notificará al usuario el fallo:

Figura 41

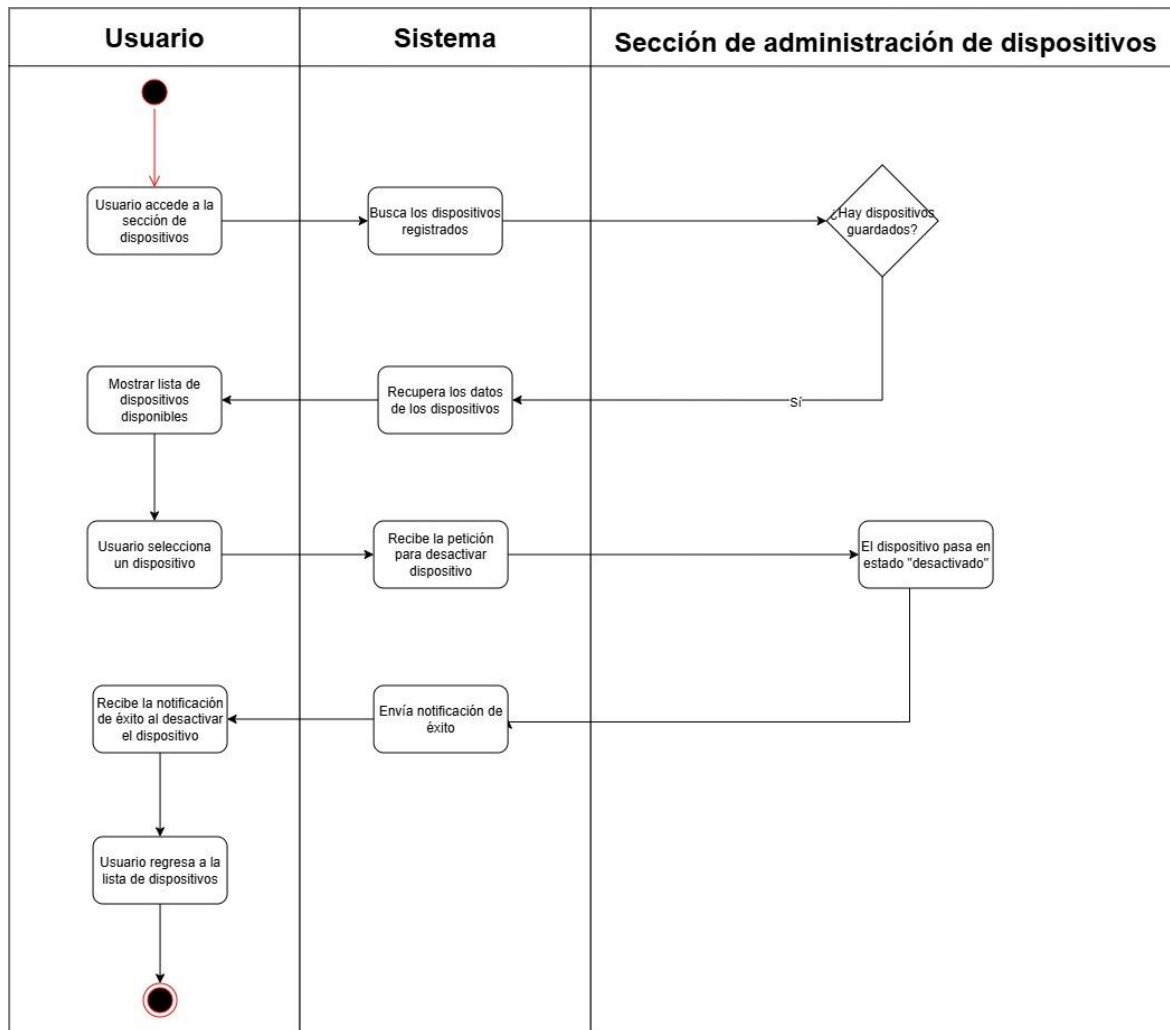
Vista de procesos en el que se actualizan detalles de un dispositivo.



La Figura 42 detalla cómo un usuario desactiva su dispositivo en el sistema. Primero, el sistema verifica que el dispositivo exista. Una vez encontrado y seleccionado, al realizar la petición para desactivarlo, internamente se llevará a cabo la desactivación. Este mismo proceso se aplica para volver a activar el dispositivo. Esta opción es una alternativa para no eliminar el dispositivo de forma permanente:

Figura 42

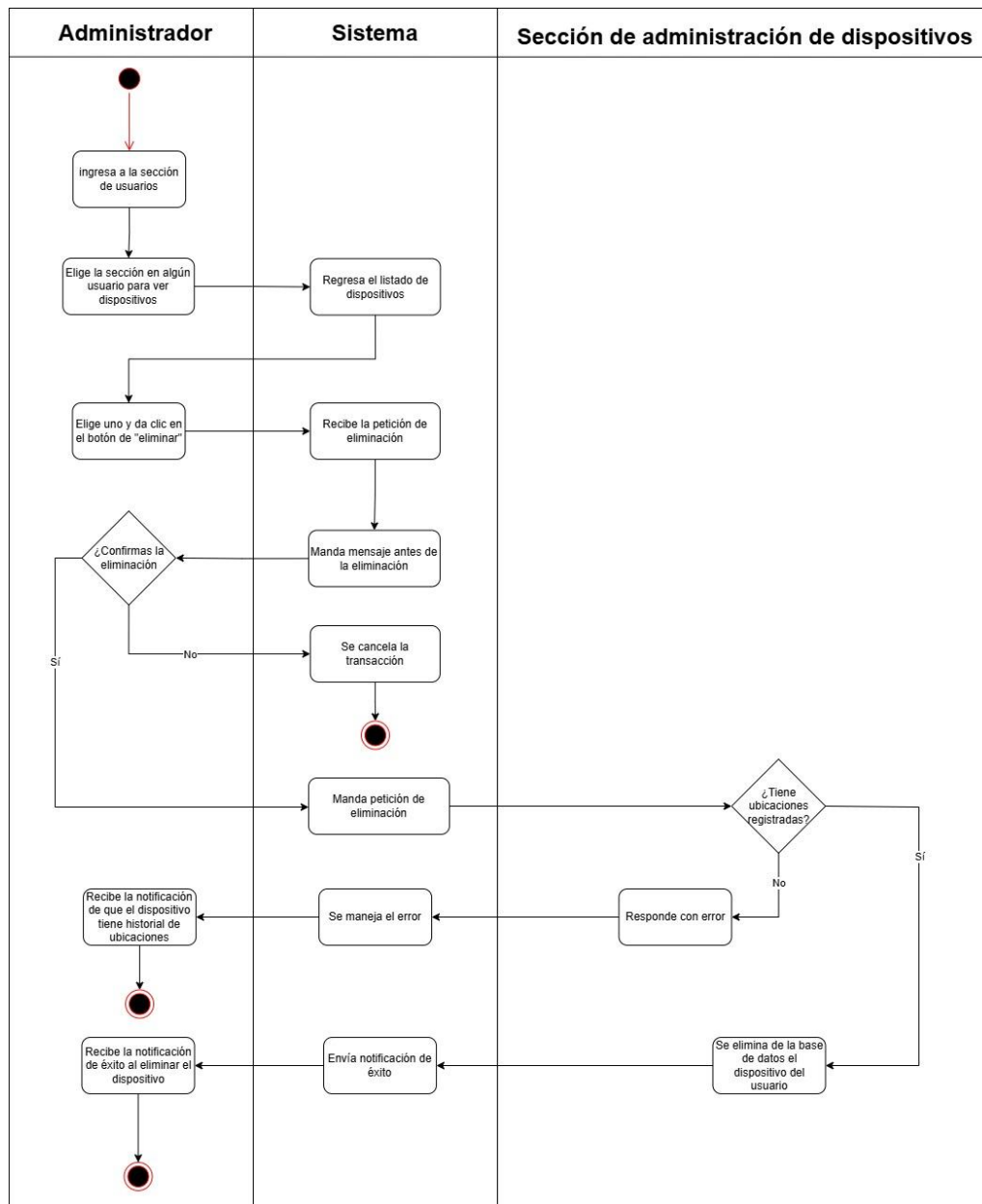
Vista de procesos que muestra el flujo de desactivar el estado de un dispositivo.



En la Figura 43 se visualiza el último diagrama para la administración de dispositivos, se contempla la eliminación de dispositivos. Por el momento, solo el administrador tendrá la facultad de eliminar los dispositivos:

Figura 43

Vista de procesos que plasma la eliminación de un dispositivo.

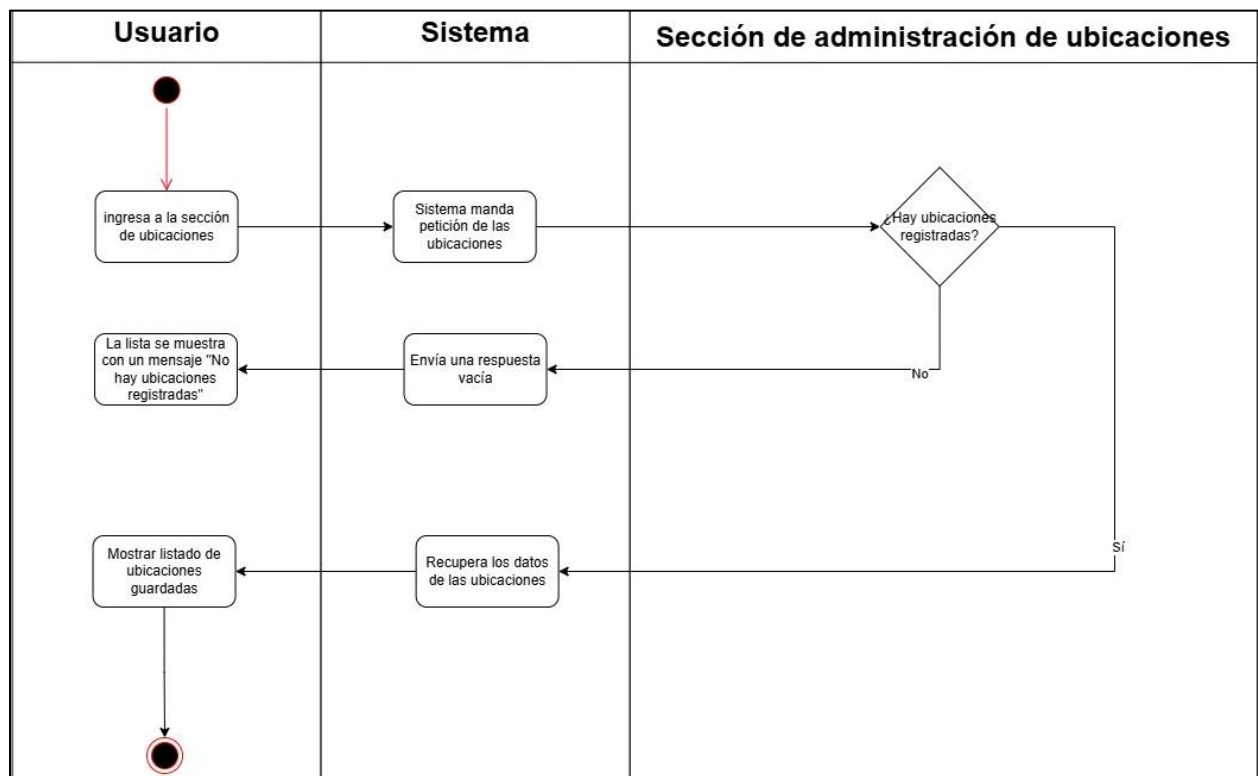


En los siguientes diagramas de actividades abordan la administración de ubicaciones, presentando tres procesos principales: la visualización de todas las ubicaciones de un dispositivo, el almacenamiento de una ubicación y el filtrado de fechas para mostrar las posiciones dentro de rangos establecidos.

El primer diagrama, se enfoca en obtener todas las posiciones que han sido almacenadas en un dispositivo. Cuando el usuario acceda a la sección de ubicaciones, si no hay ubicaciones almacenadas, se mostrará un mensaje indicándolo. Por el contrario, si existen ubicaciones, se devolverá una lista con todas las posiciones que se almacenaron en el dispositivo. La consulta de ubicaciones se presenta en la Figura 44:

Figura 44

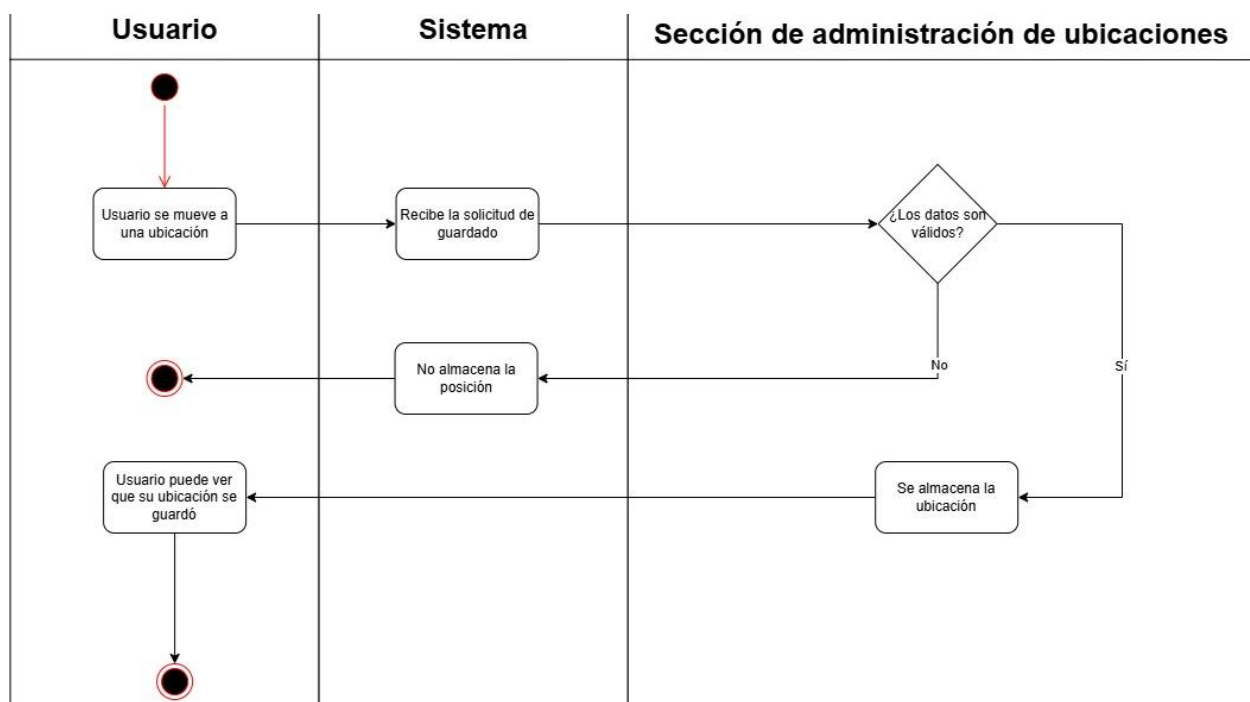
Vista de procesos que muestra todas las ubicaciones de un dispositivo.



El segundo diagrama, describe cómo el sistema almacena las ubicaciones de un usuario. Cuando un usuario se desplaza (con su dispositivo), el sistema debe recibir y verificar la corrección de los datos capturados. Si los datos no son válidos, la posición no se almacenará. Sin embargo, si la información es correcta, la posición se almacenará. Aunque no se notificará directamente al usuario, este podrá ver las posiciones guardadas en la sección de ubicaciones. El almacenamiento de una nueva posición se representa en la Figura 45:

Figura 45

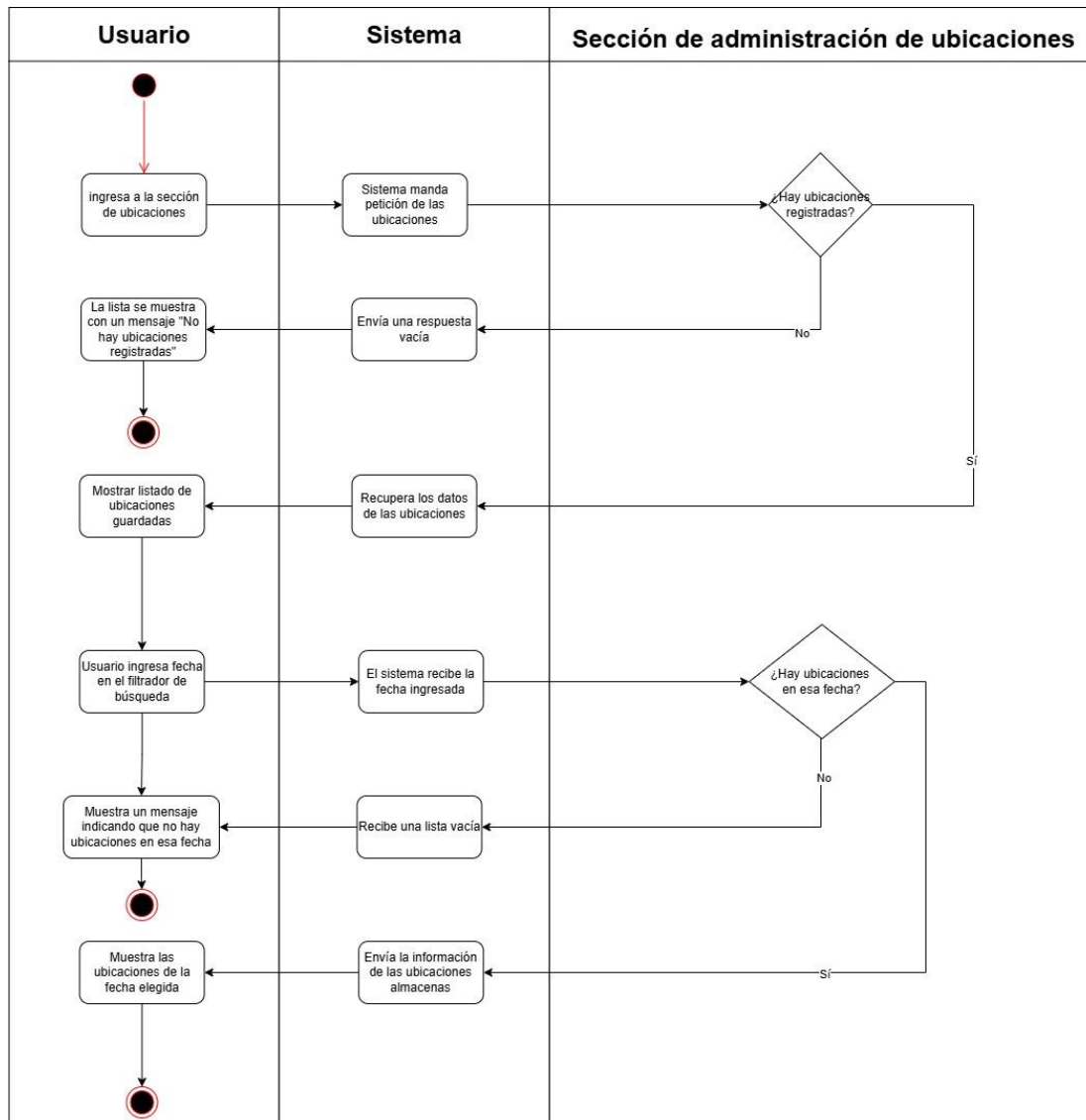
Vista de procesos que muestra como el sistema almacena una nueva posición.



El tercer diagrama, refleja el proceso de filtrado de una ubicación por fechas. Si existen ubicaciones almacenadas dentro del rango especificado, se devolverá una lista de ellas. En caso contrario, se devolverá una lista vacía, que se notificará al usuario que no hay posiciones guardadas. El filtrado de ubicaciones se ilustra en la Figura 46:

Figura 46

Vista de procesos que muestra el filtrado de ubicaciones.

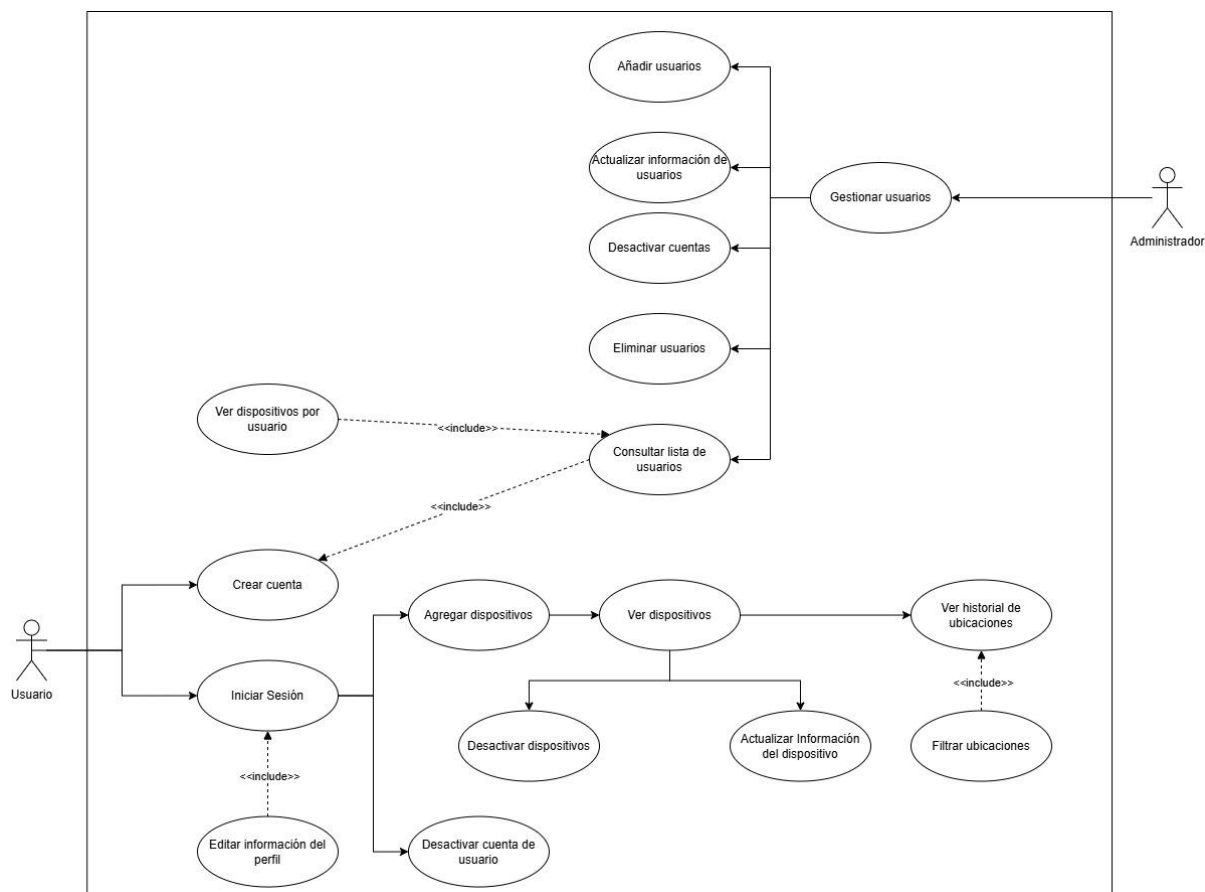


La quinta vista, es la “Vista de escenarios”, que se aplica mediante un diagrama de casos de uso. En esta perspectiva se consideran a dos actores principales: el administrador y el usuario.

El administrador tiene la capacidad de gestionar a los usuarios, lo que implica visualizar a los usuarios registrados en el sistema, además de realizar acciones como: añadir usuarios, actualizar su información, desactivar cuentas y eliminar usuarios. Además, puede visualizar los dispositivos que cada usuario posee. El usuario tiene la capacidad de crear su cuenta; se pretende que en una fase posterior se implemente la autenticación. El usuario podrá realizar acciones como: editar su información personal, desactivar su cuenta, agregar sus dispositivos, desactivarlos, actualizar la información de los dispositivos, y ver por cada dispositivo su historial de ubicaciones por cada dispositivo, así como filtrarlas por fecha. Los escenarios de interacción se representan en la Figura 47:

Figura 47

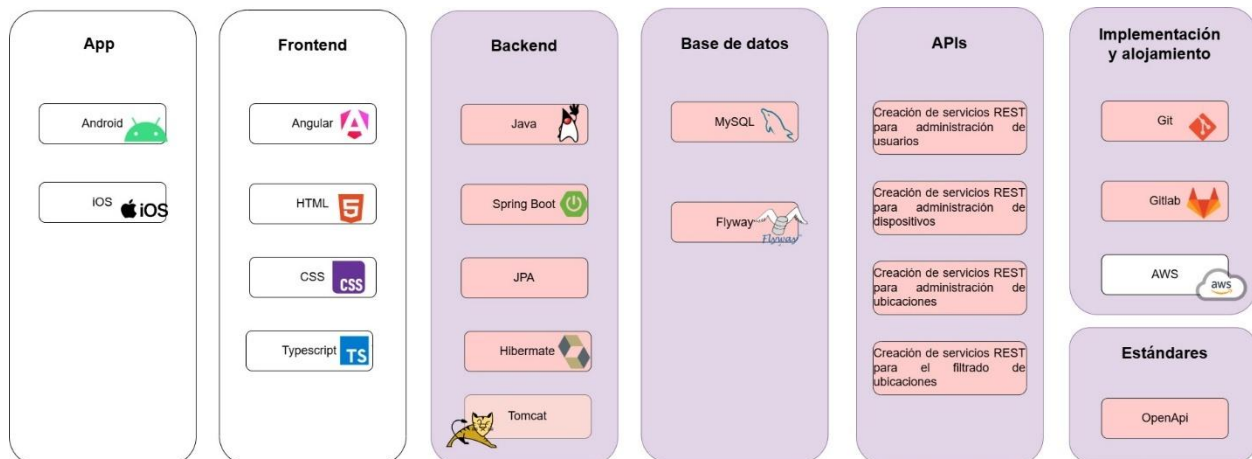
Vista de escenarios mediante diagrama de casos de uso.



Finalmente, se presenta el desarrollo de un *stack tecnológico*, que detalla las tecnologías que se emplearán en la construcción del proyecto. En esta etapa, el *backend* se compone del lenguaje Java, framework Spring Boot y tecnologías como JPA, Hibernate y Tomcat para la ejecución del componente. La base de datos utilizará MySQL como gestor; Flyway para gestionar las migraciones en la construcción de la base de datos. Las APIs se crearán en esta fase, y para la implementación y alojamiento, se usarán Git y GitLab para el control de versiones del proyecto (con la excepción de AWS, que se abordará en una fase posterior). Finalmente, se utilizará el estándar OpenAPI para la documentación final dirigida a los programadores, facilitando la comprensión de los *endpoints* creados. Las tecnologías empleadas se resumen en la Figura 48:

Figura 48

Stack Tecnológico del proyecto.

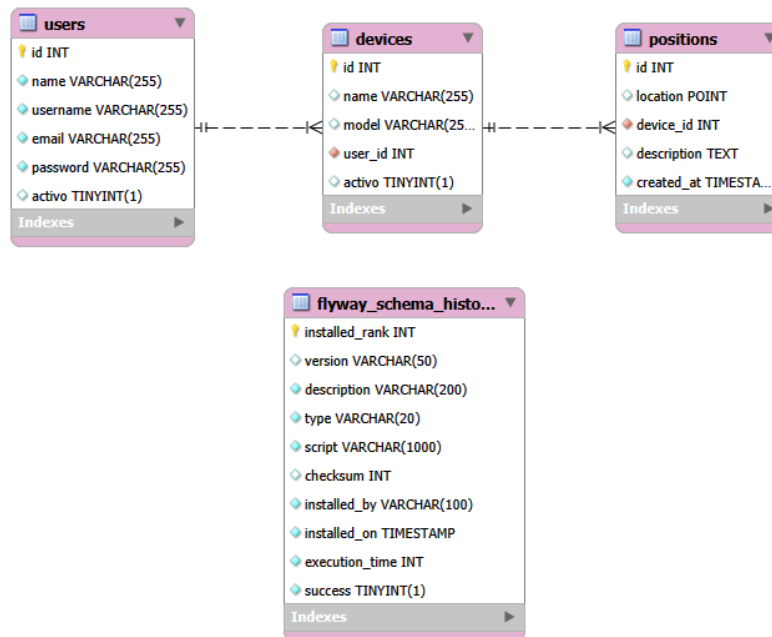


3.1.8. Esquema de base de datos

El uso de un esquema relacional de base de datos posibilita la distribución de la información del sistema, permitiendo que sea escalable con el tiempo. A continuación se presenta la estructura general de la base de datos que se muestra en la Figura 49:

Figura 49

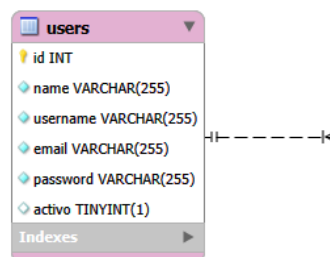
Esquema relacional de la base de datos.



En primer lugar, la tabla “users”, cuenta con seis campos, que almacenarán a los usuarios que creen una cuenta o sean registrados en el sistema. Su estructura se presenta en la Figura 50:

Figura 50

Tabla “users”.



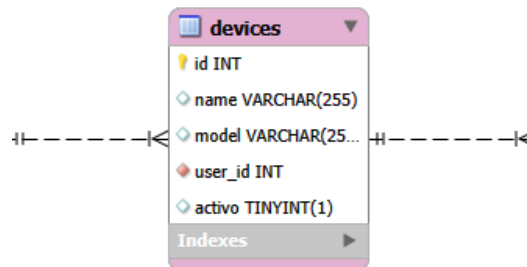
Los atributos que contiene la tabla son los siguientes:

- Id_users: representa a llave primaria, la cual en implementación será de tipo incremental.
- Username: es el nombre de usuario que tendrá la cuenta del usuario,
- Name: representa el nombre real o cotidiano del usuario.
- Email: es el correo electrónico con el que podrá iniciar sesión el usuario.
- Password: es la contraseña con la que accede al sistema.
- Activo: permite guardar valores de tipo booleano para indicar para activar la cuenta.

En segundo lugar, la tabla “devices”, representa los dispositivos asociados a uno o varios usuarios. La estructura de la tabla 'devices' puede observarse en la Figura 51:

Figura 51

Tabla "devices".



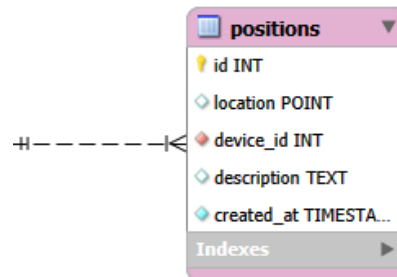
Esta tabla está conformada por los siguientes campos:

- Id_devices: es la llave primaria que tendrá la tabla, la cual es de tipo incremental.
- User_id: llave foránea a la tabla “users” en la que se hará la relación o unión entre tablas.
- Name: es el nombre del dispositivo que registre el usuario.
- Model: es el modelo o marca del dispositivo.
- Activo: indica si el dispositivo está activado o desactivado en el sistema, el cual recibe valores de tipo booleano.

Por último, en la tabla “positions”, representa todas las ubicaciones que se almacenan en los dispositivos. La Figura 52 muestra la definición de la tabla 'positions' utilizada para almacenar ubicaciones:

Figura 52

Tabla "positions".



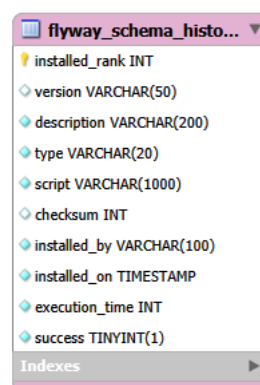
Se conforma por las columnas:

- id_positions: es la llave primaria de la tabla positions, es auto incremental.
- location: es de tipo POINT (declarado en el modelo en Spring Boot) y almacena la ubicación en coordenadas de latitud y longitud como "x" y "y".
- device_id: es la llave foránea que conecta a la tabla devices.
- description: guarda el nombre del lugar en el que se guarda la posición.
- created_at: guarda la fecha y hora en la que la posición fue capturada.

El uso de migraciones proporcionado por Flyway, crea una tabla automáticamente, que maneja las migraciones que se hagan mediante Spring Boot, mostrada en la Figura 53:

Figura 53

Tabla generada por Flyway.



3.1.9. Estructura del proyecto

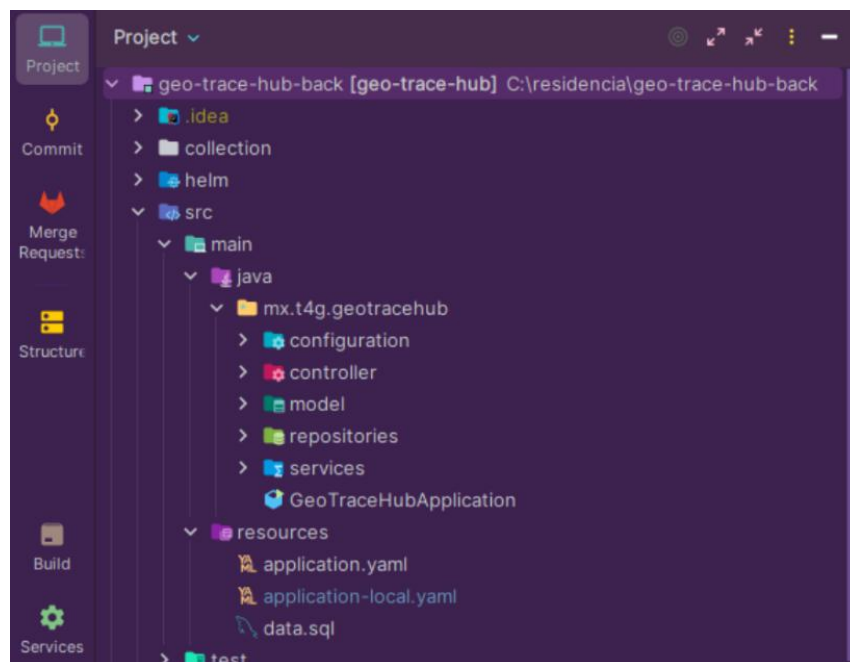
El proyecto se organiza con su directorio principal `src` en paquetes fundamentales:

- configuration,
- controller,
- model,
- repositories
- y services.

Estos paquetes son cruciales para mantener un código limpio en el proyecto. Además, se incluye un paquete llamado `resources`, donde se debe crear un archivo `.properties` o `.yaml` para establecer la conexión con la base de datos. También existe un paquete `test`, diseñado para la ejecución de pruebas del código. Los paquetes principales del proyecto se organizan como se observa en la Figura 54:

Figura 54

Paquetes y archivos cruciales del proyecto.



El archivo de conexión a la base de datos es de tipo `.yaml` se define para conectarse al manejador de base de datos el cual es MySQL Workbench. Primero se establece el nombre de la aplicación, la cual al ejecutarse tiene el nombre de “sandi”. La configuración del nombre de la aplicación en el archivo `.yaml` se muestra en la Figura 55:

Figura 55

Declaración del nombre de la aplicación en el archivo `.yaml`.

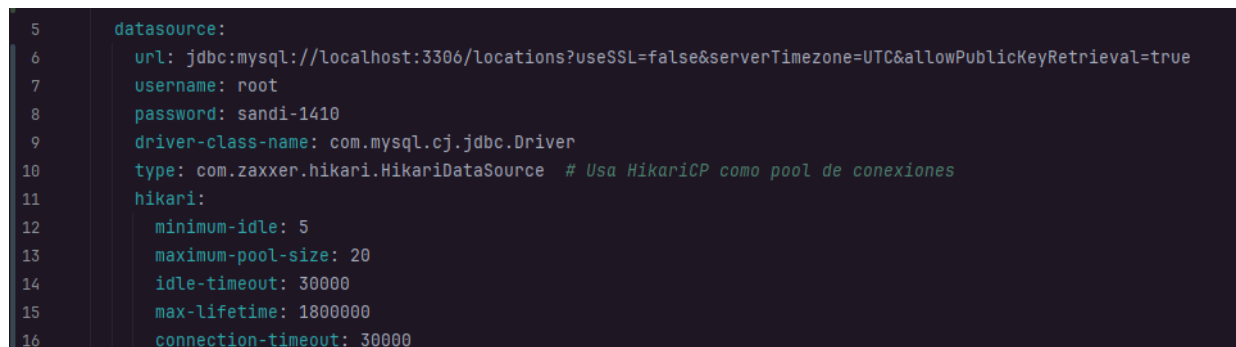
A screenshot of a code editor showing a file named 'application-local.yaml'. The file contains three lines of YAML configuration: line 1 is 'spring:', line 2 is ' application:', and line 3 is ' name: sandi'. The editor has a dark theme with a purple header bar.

```
1  spring:
2    application:
3      name: sandi
```

En esta sección se configura la conexión a la base de datos MySQL, se incluye la URL de conexión a la base de datos, las credenciales de acceso a la base de datos, el controlador JDBC para MySQL, se configura la agrupación de conexiones (connection pooling) con HikariCP. La conexión a MySQL se indica en la Figura 56:

Figura 56

Configuración de la conexión a la base de datos en el archivo `.yaml`.

A screenshot of a code editor showing a file named 'application-local.yaml'. The file contains configuration for a data source and HikariCP settings. Lines 5-11 define the 'datasource' with URL, username, password, driver class, and type. Lines 12-16 define the 'hikari' settings including minimum idle, maximum pool size, idle timeout, max lifetime, and connection timeout. The editor has a dark theme with a purple header bar.

```
5  datasource:
6    url: jdbc:mysql://localhost:3306/locations?useSSL=false&serverTimezone=UTC&allowPublicKeyRetrieval=true
7    username: root
8    password: sandi-1410
9    driver-class-name: com.mysql.cj.jdbc.Driver
10   type: com.zaxxer.hikari.HikariDataSource # Usa HikariCP como pool de conexiones
11   hikari:
12     minimum-idle: 5
13     maximum-pool-size: 20
14     idle-timeout: 30000
15     max-lifetime: 1800000
16     connection-timeout: 30000
```

Posteriormente, se configura JPA, manteniendo la sesión de Hibernate abierta durante la renderización de la vista y estableciendo el dialecto de Hibernate para MySQL. La configuración de Hibernate retrasa la inicialización de la fuente de datos hasta que sea necesaria. Se establece la estrategia de creación de tablas en la base de datos como "none", lo que significa que no se crearán ni actualizarán tablas automáticamente. También se activa la visualización de las

consultas SQL ejecutadas por Hibernate, formateando dichas consultas para hacerlas más legibles. La Figura 57 muestra la forma en que se integran JPA e Hibernate:

Figura 57

Configuración de JPA e Hibernate en el archivo .yaml.

```
18   jpa:
19     open-in-view: false
20     database-platform: org.hibernate.dialect.MySQLDialect
21     hibernate:
22       ddl-auto: create # Opciones: update, create, create-drop, none
23       show-sql: true
24     properties:
25       hibernate:
26         format_sql: true
27         #dialect: org.hibernate.dialect.MySQL8Dialect
28         transaction_ita_platform: org.hibernate.engine.transaction.ita.platform.internal.NoItaPlatform
```

Por último, se configuran los mensajes de los logs, en el que se mostrarán mensajes en la depuración. La activación del modo debug se visualiza en la Figura 58:

Figura 58

Activación de la depuración en el archivo .yaml.

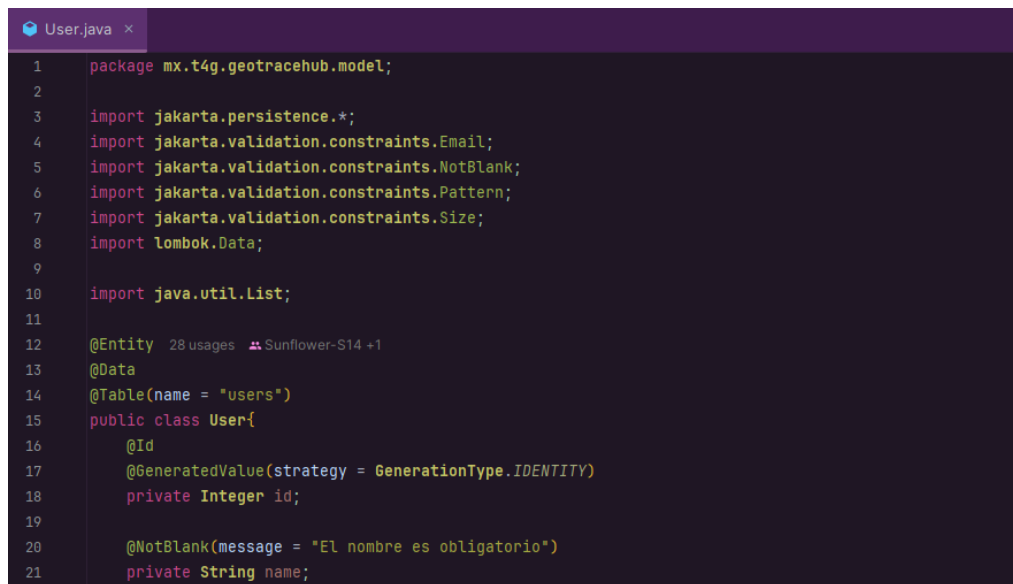
```
logging:
  level:
    root: DEBUG
    org.hibernate.SQL: DEBUG
    org.hibernate.type.descriptor.sql.BasicBinder: TRACE
```

3.1.10. Administración de usuarios

Modelo o entidad usuario: en la carpeta de models se crea un nuevo archivo, anotado con `@Entity` a la clase “User”, que representa a un usuario en la base de datos en forma de tabla y se mapean las columnas. Se implementan validaciones con la librería *Validation* para usar garantizar código limpio. El modelo User y sus atributos se representan en la Figura 59:

Figura 59

Modelo User.

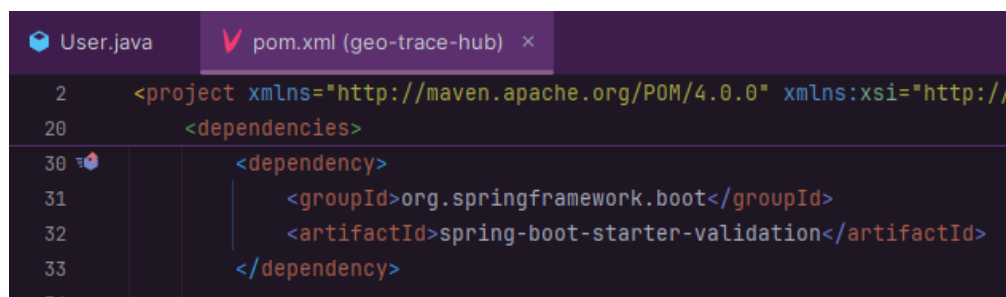


```
1 package mx.t4g.geotracehub.model;
2
3 import jakarta.persistence.*;
4 import jakarta.validation.constraints.Email;
5 import jakarta.validation.constraints.NotBlank;
6 import jakarta.validation.constraints.Pattern;
7 import jakarta.validation.constraints.Size;
8 import lombok.Data;
9
10 import java.util.List;
11
12 @Entity 28 usages  Sunflower-S14 +1
13 @Data
14 @Table(name = "users")
15 public class User{
16     @Id
17     @GeneratedValue(strategy = GenerationType.IDENTITY)
18     private Integer id;
19
20     @NotBlank(message = "El nombre es obligatorio")
21     private String name;
```

Se agrega en la Figura 60 una dependencia para el uso de las validaciones con `@Valid`:

Figura 60

Dependencia validation en el pom.xml.

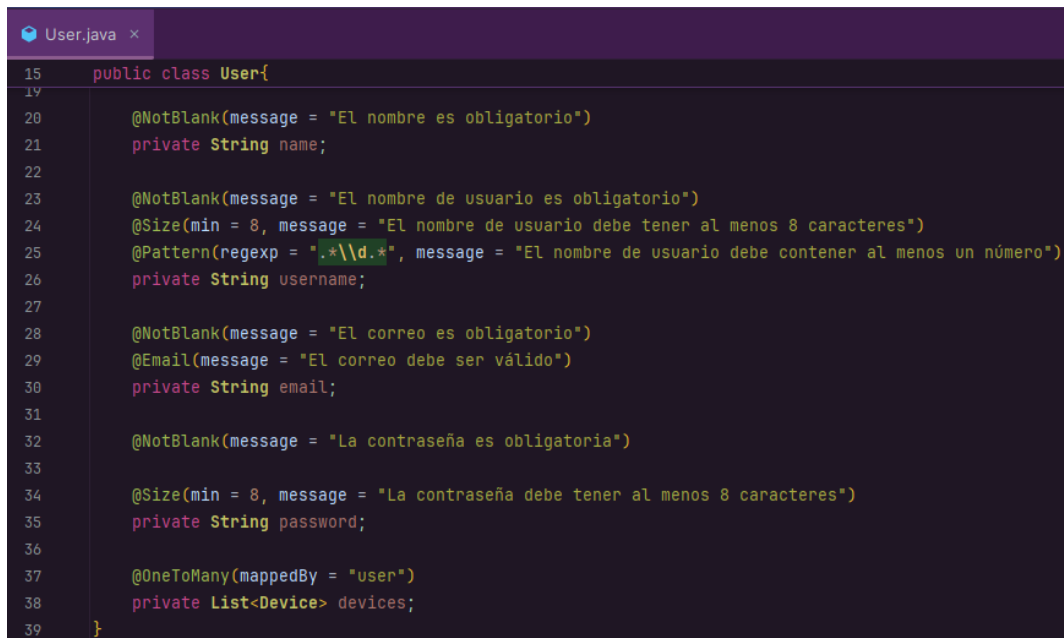


```
2 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
20 <dependencies>
30 <dependency>
31 <groupId>org.springframework.boot</groupId>
32 <artifactId>spring-boot-starter-validation</artifactId>
33 </dependency>
```

La entidad “user” se inicia declarando el ID de los registros como auto incrementales, únicos para cada usuario y las validaciones implementadas son el uso de `@NotBlank` permiten establecer a los campos como obligatorios. La anotación `@Size` declara la longitud mínima de un campo y validaciones de caracteres especiales usando `@Pattern`. El uso de `@Email` es para validación del correo electrónico. Por último se establece una llave foránea de tipo `OneToMany` que se enlazaré con el modelo `devices`. La Figura 61 detalla los atributos del modelo `User` mapeados a la base de datos:

Figura 61

Declaración de los atributos para la tabla `users` en el modelo `User`.



```
15 public class User{
16
17     @NotBlank(message = "El nombre es obligatorio")
18     private String name;
19
20     @NotBlank(message = "El nombre de usuario es obligatorio")
21     @Size(min = 8, message = "El nombre de usuario debe tener al menos 8 caracteres")
22     @Pattern(regexp = ".*\\d.*", message = "El nombre de usuario debe contener al menos un número")
23     private String username;
24
25     @NotBlank(message = "El correo es obligatorio")
26     @Email(message = "El correo debe ser válido")
27     private String email;
28
29     @NotBlank(message = "La contraseña es obligatoria")
30
31     @Size(min = 8, message = "La contraseña debe tener al menos 8 caracteres")
32     private String password;
33
34     @OneToMany(mappedBy = "user")
35     private List<Device> devices;
36
37 }
```

Repository o Repositorio: define el repositorio “`UserRepository`” utilizado para interactuar con la base de datos y realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) sobre la entidad “`User`”.

Se establecen métodos booleanos para verificar si los usuarios ya existían mediante su nombre de usuario o por su email, también para recuperar una lista de usuarios que estén activos. La interfaz `UserRepository` se muestra en la Figura 62:

Figura 62

Interfaz UserRepository.

```
UserRepository.java x
1 package mx.t4g.geotracehub.repositories;
2
3 import mx.t4g.geotracehub.model.User;
4 import org.springframework.data.jpa.repository.JpaRepository;
5 import org.springframework.stereotype.Repository;
6
7 import java.util.List;
8
9 @Repository 4 usages 🚩 Sunflower-S14 +1
10 public interface UserRepository extends JpaRepository<User, Integer> {
11
12     boolean existsByUsername(String username); 2 usages 🚩 Sunflower-S14
13     boolean existsByEmail(String email); 2 usages 🚩 Sunflower-S14
14
15     List<User> findByActivoTrue(); 2 usages 🚩 Sunflower-S14
16
17 }
```

Service o servicio: define la interfaz “UserService”, la cual se empleará para encapsular la lógica del negocio relacionada con la gestión de usuarios. Es decir, define los métodos que se utilizarán para interactuar con las operaciones sobre los usuarios, como listar los usuarios registrados, obtener un usuario por su Id, guardar un usuario, actualizar datos del usuario y eliminar a un usuario, o activar o desactivar el estado de una cuenta de un usuario. La Figura 63 contiene la definición de la interfaz del servicio de usuario:

Figura 63

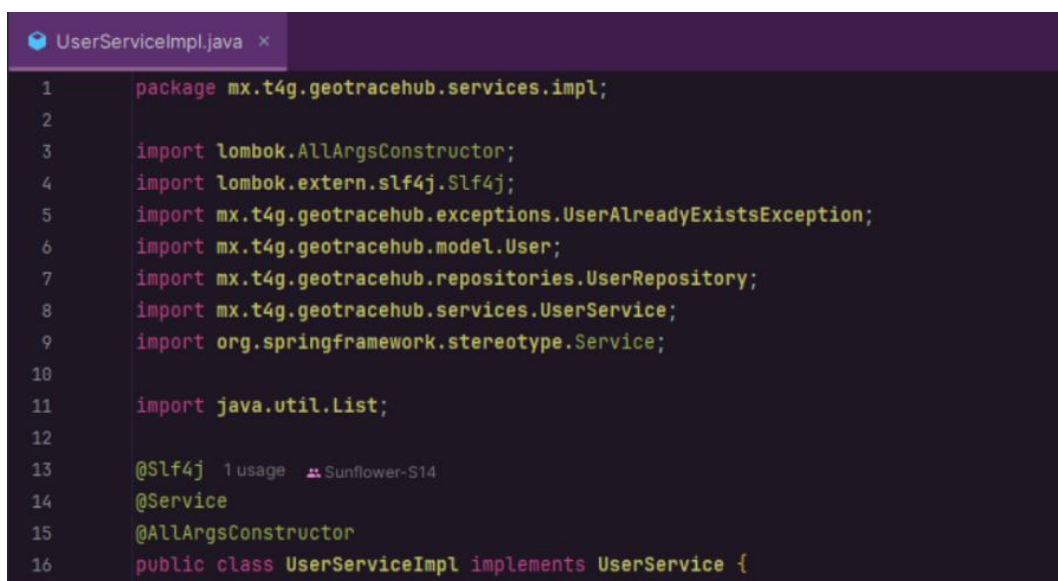
Interfaz UserService.

```
UserService.java x
1 package mx.t4g.geotracehub.services;
2
3 import mx.t4g.geotracehub.dto.UpdateUserDTO;
4 import mx.t4g.geotracehub.model.User;
5 import java.util.List;
6
7 public interface UserService { 7 usages 1 implementation 🚩 Sunflower-S14
8     List<User> getAllUsers(); 4 usages 1 implementation 🚩 Sunflower-S14
9     User getUserById(Integer userId); 3 usages 1 implementation 🚩 Sunflower-S14
10    User saveUser(User user); 3 usages 1 implementation 🚩 Sunflower-S14
11    User updateUser(Integer userId, UpdateUserDTO updatedUser); 3 usages 1 implementation 🚩 Sunflower-S14
12    void deleteUser(Integer userId); 3 usages 1 implementation 🚩 Sunflower-S14
13
14    void userStatus(Integer userId, boolean status); 2 usages 1 implementation 🚩 Sunflower-S14
15
16 }
```

Servicio implementado o Service Implements: define la clase “UserServiceImpl” que se anota con `@Service` indicando que es un servicio, utilizado para encapsular la lógica de negocio e implementa la interfaz “UserService”. Esta clase define las operaciones específicas para cada método, también utiliza el repositorio “UserRepository” para interactuar con la base de datos y realizar operaciones CRUD sobre los usuarios. La implementación del servicio de usuario se encuentra en la Figura 64:

Figura 64

Clase UserServiceImpl.



```
1 package mx.t4g.geotracehub.services.impl;
2
3 import lombok.AllArgsConstructor;
4 import lombok.extern.slf4j.Slf4j;
5 import mx.t4g.geotracehub.exceptions.UserAlreadyExistsException;
6 import mx.t4g.geotracehub.model.User;
7 import mx.t4g.geotracehub.repositories.UserRepository;
8 import mx.t4g.geotracehub.services.UserService;
9 import org.springframework.stereotype.Service;
10
11 import java.util.List;
12
13 @Slf4j 1 usage Sunflower-S14
14 @Service
15 @AllArgsConstructor
16 public class UserServiceImpl implements UserService {
```

El primer método, llamará al repositorio para retornar todos los usuarios que existan. La Figura 65 muestra la lógica para obtener todos los usuarios:

Figura 65

Método getAllUsers.



```
16 public class UserServiceImpl implements UserService {
17
18     private final UserRepository userRepository;
19
20     @Override 2 usages Sunflower-S14
21     public List<User> getAllUsers() {
22         log.info("Fetching all users");
23         return userRepository.findAll();
24     }
25 }
```


El segundo método, “userStatus”, permite modificar el estado de la cuenta de un usuario, el cual gestiona mediante valores booleanos (true y false). Inicialmente, se invoca al repositorio para buscar a un usuario por su ID. Si el usuario no se encuentra, se lanzará una excepción. Si se localiza, se cambiará el estado del usuario al valor proporcionado y se volverá a llamar al repositorio para guardar los cambios. La activación o desactivación del estado del usuario se realiza como en la Figura 66:

Figura 66

Método userStatus.

```
26  public void userStatus(Integer userId, boolean status) { 2 usages  Sunflower-S14 *
27     log.info("Changing user status for ID: {}", userId);
28     User user = userRepository.findById(userId)
29         .orElseThrow(() -> new RuntimeException("User not found with ID: " + userId));
30     user.setActivo(status);
31     userRepository.save(user);
32 }
```

El tercer método, es “getUserById”, el cual invoca al repositorio para buscar al usuario por su ID y lo retorna si lo encuentra. En caso de no hallarlo, lanzará una excepción genérica, notificando al usuario que el ID no existe. La Figura 67 presenta el método utilizado para obtener un usuario por ID:

Figura 67

Método getUserById.

```
34  @Override 3 usages  Sunflower-S14
35  public User getUserById(Integer userId) {
36     log.info("Fetching user with ID: {}", userId);
37     return userRepository.findById(userId)
38         .orElseThrow(() -> new RuntimeException("User not found with ID: " + userId));
39 }
```

El cuarto método, “saveUser” permite guardar un registro de un nuevo usuario, llamará al repositorio para guardarlo en la base de datos. El guardado de nuevos usuarios se representa en la Figura 68:

Figura 68

Método saveUser.

```
41      @Override 3 usages Sunflower-S14 +1
42      public User saveUser(User user) {
43
44          if (userRepository.existsByUsername(user.getUsername())){
45              throw new ValidateException("Nombre de usuario", "El nombre de usuario ya está registrado");
46          }
47          if (userRepository.existsByEmail(user.getEmail())){
48              throw new ValidateException("Email", "El correo ya está registrado");
49          }
50          log.info("Saving user: {}", user);
51          return userRepository.save(user);
52      }
53  }
```

El quinto método, “updateUser” invoca al repositorio para actualizar el registro de un usuario. Primero, busca el ID del usuario para mostrar su información. Si el usuario existe, verifica que los atributos no se hayan modificado para evitar la eliminación de datos existentes; si algún dato se modificó, se actualiza la información y se retorna el usuario actualizado. Si no se encuentra al usuario con el ID proporcionado, se lanzará una excepción genérica, La actualización de los datos de usuario se realiza según la lógica mostrada en la Figura 69:

Figura 69

Método updateUser.

```
55      @Override no usages Sunflower-S14
56      public User updateUser(Integer userId, User updatedUser) {
57          log.info("Updating user with ID: {}", userId);
58          return userRepository.findById(userId).map( User existingUser -> {
59              existingUser.setName(updatedUser.getName());
60              existingUser.setEmail(updatedUser.getEmail());
61              existingUser.setUsername(updatedUser.getUsername());
62              return userRepository.save(existingUser);
63          }).orElseThrow(() -> new RuntimeException("User not found with ID: " + userId));
64      }
```

El último método, “deleteUser” opera bajo una condición que busca la existencia del ID de usuario proporcionado. Si el ID no existe, se lanzará una excepción indicando que no se encontró el usuario. En caso contrario, si el usuario es encontrado, se procederá con su eliminación. La eliminación de usuarios está ilustrada en la Figura 70:

Figura 70

Método deleteUser.

```
66      @Override 3 usages Sunflower-S14
67      public void deleteUser(Integer userId) {
68          log.info("Deleting user with ID: {}", userId);
69          if (!userRepository.existsById(userId)) {
70              throw new RuntimeException("User not found with ID: " + userId);
71          }
72          userRepository.deleteById(userId);

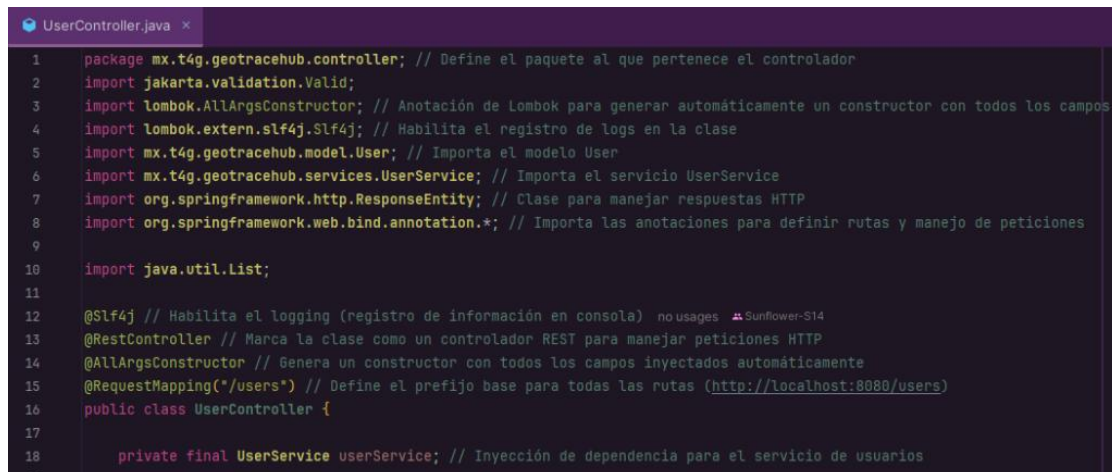
```

En cada uno de estos métodos en “UserServicesImpl” se declara un *log.info*, que permite mostrar en pruebas unitarias o cuando se ejecute el código mensajes en la terminal.

Controlador REST o Controller REST: se crea la clase “UserController” para manejar las solicitudes HTTP relacionadas con la gestión de usuarios, la cual está anotada con `@RestController` indica que la clase es un controlador *REST* que maneja solicitudes y devuelve respuestas HTTP. Se definen los *endpoints* para la ejecución de las operaciones CRUD de usuarios. Declarado con la anotación `@RequestMapping` la ruta base para gestión de usuarios que es `/users`. Esta clase utiliza el servicio `UserService` inyectado por Spring para realizar operaciones CRUD sobre los usuarios en los que se definen los métodos anotados con `@GetMapping`, `@PostMapping`, `@PutMapping` y `@DeleteMapping` para indicar el tipo de solicitud HTTP que manejan. La Figura 71 presenta el controlador principal de la entidad User:

Figura 71

Clase UserController.

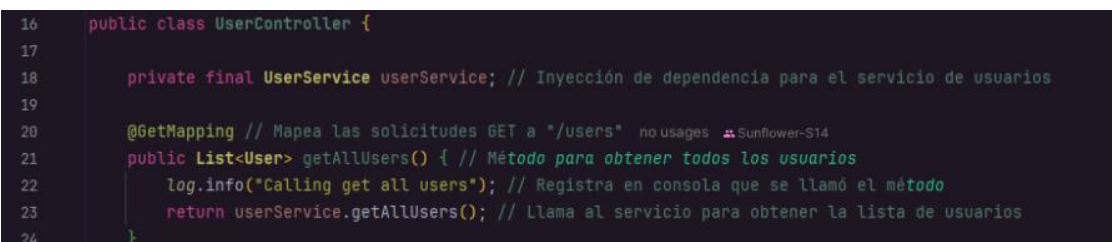


```
1 package mx.t4g.geotrachub.controller; // Define el paquete al que pertenece el controlador
2 import jakarta.validation.Valid;
3 import lombok.AllArgsConstructor; // Anotación de Lombok para generar automáticamente un constructor con todos los campos
4 import lombok.extern.slf4j.Slf4j; // Habilita el registro de logs en la clase
5 import mx.t4g.geotrachub.model.User; // Importa el modelo User
6 import mx.t4g.geotrachub.services.UserService; // Importa el servicio UserService
7 import org.springframework.http.ResponseEntity; // Clase para manejar respuestas HTTP
8 import org.springframework.web.bind.annotation.*; // Importa las anotaciones para definir rutas y manejo de peticiones
9
10 import java.util.List;
11
12 @Slf4j // Habilita el logging (registro de información en consola) no usages Sunflower-S14
13 @RestController // Marca la clase como un controlador REST para manejar peticiones HTTP
14 @AllArgsConstructor // Genera un constructor con todos los campos inyectados automáticamente
15 @RequestMapping("/users") // Define el prefijo base para todas las rutas (http://localhost:8080/users)
16 public class UserController {
17
18     private final UserService userService; // Inyección de dependencia para el servicio de usuarios
19
20     // ... (other methods would follow) ...
21 }
```

El primer endpoint, se configura la ruta base con `@GetMapping` en el método “`getAllUsers`” que al llamar a `/users`, se invoca al servicio que obtiene todos los usuarios que estén registrados. La Figura 72 muestra cómo se gestionan las solicitudes para listar usuarios:

Figura 72

Método `getAllUsers` del controlador.

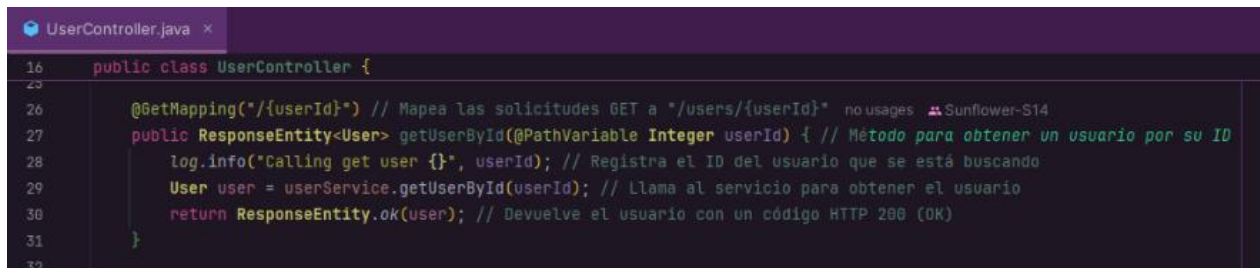


```
16 public class UserController {
17
18     private final UserService userService; // Inyección de dependencia para el servicio de usuarios
19
20     @GetMapping // Mapea las solicitudes GET a "/users" no usages Sunflower-S14
21     public List<User> getAllUsers() { // Método para obtener todos los usuarios
22         log.info("Calling get all users"); // Registra en consola que se llamó el método
23         return userService.getAllUsers(); // Llama al servicio para obtener la lista de usuarios
24     }
25 }
```

El segundo método, “getUserById” el endpoint declarado con *@GetMapping* es *users/userId*. Llamará al servicio para buscar el usuario mediante su ID y si es encontrado retornará una respuesta HTTP “200 OK” indicando que la petición fue realizada de manera correcta. La lógica para obtener usuarios por ID está representada en la Figura 73:

Figura 73

Método getUserById del controlador.



```
16 public class UserController {
27     @GetMapping("/{userId}") // Mapea las solicitudes GET a "/users/{userId}" no usages Sunflower-S14
28     public ResponseEntity<User> getUserById(@PathVariable Integer userId) { // Método para obtener un usuario por su ID
29         log.info("Calling get user {}", userId); // Registra el ID del usuario que se está buscando
30         User user = userService.getUserById(userId); // Llama al servicio para obtener el usuario
31         return ResponseEntity.ok(user); // Devuelve el usuario con un código HTTP 200 (OK)
32     }
33 }
```

El tercer método, “saveUser” implementa la anotación *@Valid* al cuerpo de la respuesta que recibe la información de un nuevo usuario que permite implementar las validaciones declaradas en el modelo. El endpoint declarado con *@PostMapping* es *users/register*, llamará al servicio para guardar al usuario y luego mostrar un código “200 OK” indicando que el usuario fue registrado correctamente. La creación de un nuevo usuario desde el controlador se representa en la Figura 74:

Figura 74

Método saveUser del controlador.



```
33 //Guarda un usuario en el sistema
34 @PostMapping("/register") no usages Sunflower-S14
35 public ResponseEntity<Object> saveUser(@Valid @RequestBody User user) {
36     log.info("Calling save user with this information {}", user);
37
38     userService.saveUser(user);
39     return ResponseEntity.ok( body: "Usuario registrado correctamente");
40 }
```

El cuarto método, es “updateUser” en el que con *@PutMapping* la ruta final es *users/userId* llama al servicio para actualizar la información del usuario en el que después de guardar la información

retornará una respuesta "200 OK". La Figura 75 ilustra la actualización de usuarios desde el controlador:

Figura 75

Método updateUser del controlador.

```
41
42     @PutMapping("/{userId}") // Mapea las solicitudes PUT a "/users/{userId}" no usages Sunflower-S14
43     public ResponseEntity<User> updateUser(@PathVariable Integer userId, @RequestBody User updatedUser) { // Método para actualizar un usuario
44         log.info("Calling update user {} with data {}", userId, updatedUser); // Registra el ID y los nuevos datos del usuario
45         User user = userService.updateUser(userId, updatedUser); // Llama al servicio para actualizar el usuario
46         return ResponseEntity.ok(user); // Devuelve el usuario actualizado con código HTTP 200 (OK)
47     }
```

El quinto método, "userStatus", se asocia a la ruta `users/userId/status` mediante `@PatchMapping`. Esta ruta permite que desde el endpoint se envíe el estado en un valor booleano para activar o desactivar la cuenta de un usuario. Se invocará al servicio, pasando el ID del usuario como el estado (false o true). Si la operación es exitosa, se retornará una respuesta "200 OK". El cambio de estado del usuario es manejado como se ve en la Figura 76:

Figura 76

Método userStatus del controlador.

```
49     @PatchMapping("/{userId}/status") no usages new *
50     public ResponseEntity<String> userStatus(@PathVariable Integer userId, @RequestParam boolean activo) {
51         log.info("Toggling user status for ID: {}", userId);
52         userService.userStatus(userId, activo);
53         return ResponseEntity.ok( body: "User status updated successfully");
54     }
```

Y por último, en "deleteUser", el endpoint final configurado con `@DeleteMapping` será `/users/deviceId`. Este endpoint invocará al servicio para buscar el ID del usuario proporcionado, y tras la eliminación exitosa, retornará una respuesta "200 OK". La eliminación desde el controlador se representa en la Figura 77:

Figura 77

Método deleteUser del controlador.

```
57     @DeleteMapping("/{userId}") // Mapea las solicitudes DELETE a "/users/{userId}" no usages new *
58     public ResponseEntity<Void> deleteUser(@PathVariable Integer userId) { // Método para eliminar un usuario
59         log.info("Calling delete user {}", userId); // Registra el ID del usuario que se va a eliminar
60         userService.deleteUser(userId); // Llama al servicio para eliminar el usuario
61         return ResponseEntity.noContent().build(); // Devuelve una respuesta HTTP 204 (No Content) para indicar éxito
62     }
63 }
```

3.1.11. Configuración de Flyway:

Para utilizar *Flyway*, es necesario importar las siguientes dependencias en el archivo pom.xml. La integración de Flyway mediante dependencias se muestra en la Figura 78:

Figura 78

Dependencias de Flyway en el archivo pom.xml.

```
<dependency>
  <groupId>org.flywaydb</groupId>
  <artifactId>flyway-core</artifactId>
  <version>10.20.1</version>
</dependency>
<dependency>
  <groupId>org.flywaydb</groupId>
  <artifactId>flyway-mysql</artifactId>
  <version>10.20.1</version> <!-- O una versión más reciente -->
</dependency>
```

En el archivo de conexión a la base de datos, que en este caso es application-local.yaml, se añade la configuración necesaria para que la aplicación pueda utilizar *Flyway*. La configuración de Flyway en el archivo .yaml se visualiza en la Figura 79:

Figura 79

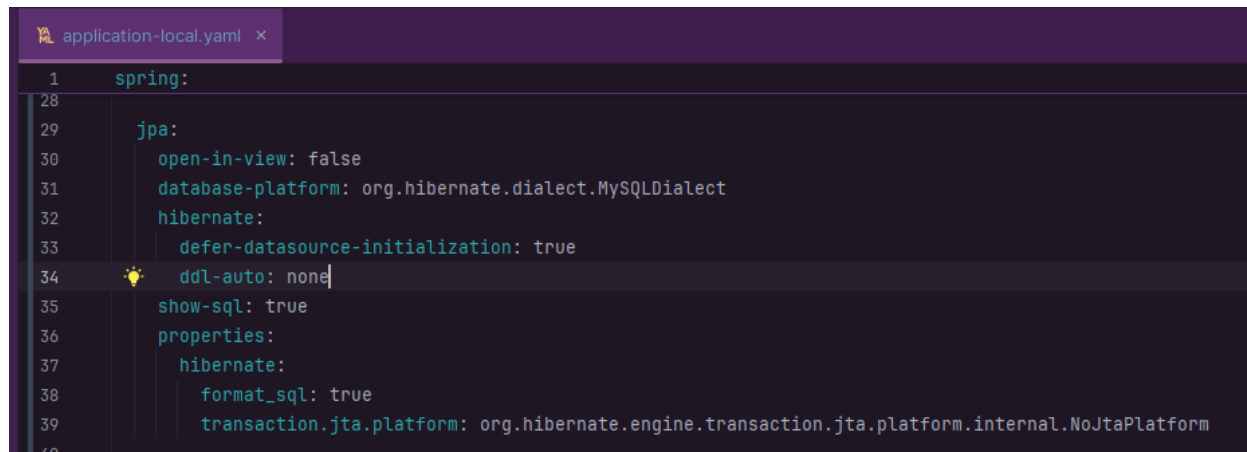
Configuración de Flyway en el archivo .yaml.

```
application-local.yaml x
1  spring:
16  flyway:
17    url: jdbc:mysql://localhost:3306/locations?useSSL=false&serverTimezone=UTC&allowPublicKeyRetrieval=true
18    user: root
19    password: sandi-1410
20    enabled: true
21    baseline-on-migrate: true
22    locations: classpath:db/migration
23    placeholders:
24      mysql_version: "8.0"
25      migration-scripts-table: flyway_schema_history
26      sql-migration-prefix: V
27      sql-migration-suffixes: .sql
```


Para evitar conflictos y un bucle infinito entre JPA y Flyway, dado que ambos se inicializarán simultáneamente, es necesario realizar un ajuste en la sección de configuración de JPA. Se debe establecer la propiedad `ddl-auto` en "none", lo que previene colisiones entre JPA y Flyway. La Figura 80 expone el conflicto de configuración entre JPA y Flyway:

Figura 80

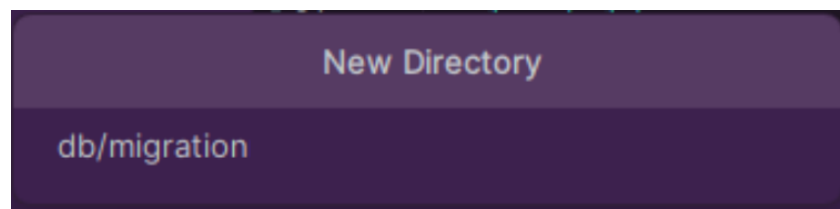
Configuración del conflicto entre JPA y Flyway.



Posteriormente, se creará el siguiente directorio dentro de `main/resources` para almacenar las migraciones que utilizará Flyway. El directorio donde se ubican las migraciones SQL se muestra en la Figura 81:

Figura 81

Directorio para las migraciones con Flyway.



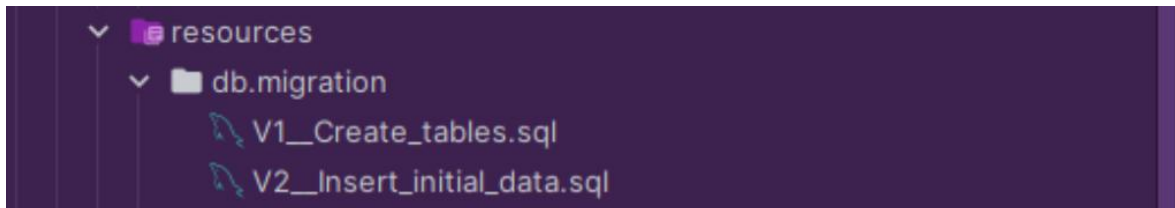
Como resultado se obtiene el siguiente directorio en el proyecto: `src/main/resources/db.migration`.

Dentro del directorio, se crean archivos `.sql` para crear las migraciones que utilizará Flyway, las cuales se crean manualmente. Para probar que Flyway tome las migraciones se crean los

siguientes archivos con la nomenclatura “V{Numero}__nombre_migracion.sql”. Las migraciones generadas para la base de datos se presentan en la Figura 82:

Figura 82

Migraciones creadas dentro del directorio.



En el primer archivo de migración, se establece un script para crear la base de datos, la cual contiene las tres tablas del esquema de base de datos, el primero es el de “users” en donde se escribe en lenguaje SQL la construcción de la tabla declarando su llave primaria, atributos de la tabla y restricciones que tendrán los atributos en sus validaciones. La creación de la tabla users se representa en la Figura 83:

Figura 83

Estructura de la creación de la tabla users.

```
V1__Create_tables.sql x
1  CREATE TABLE if not exists locations.users (
2      id INT AUTO_INCREMENT PRIMARY KEY,
3      name VARCHAR(255) NOT NULL,
4      username VARCHAR(255) NOT NULL,
5      email VARCHAR(255) NOT NULL,
6      password VARCHAR(255) NOT NULL,
7      CONSTRAINT chk_username_length CHECK (CHAR_LENGTH(username) >= 8),
8      CONSTRAINT chk_username_number CHECK (username REGEXP '[0-9]'),
9      CONSTRAINT chk_password_length CHECK (CHAR_LENGTH(password) >= 8),
10     CONSTRAINT unique_username UNIQUE (username),
11     CONSTRAINT unique_email UNIQUE (email)
12 );
```

La segunda tabla, es “devices”, se declaró su llave primaria, atributos y su llave foránea a la tabla “positions”, la cual es la tercera tabla en la que de la misma manera se declaró su llave primaria,

atributos y su llave foránea a la tabla “devices”. La definición de las tablas devices y positions se ilustra en la Figura 84:

Figura 84

Estructura de la creación de la tabla devices y positions.

```
V1__Create_tables.sql x
9      CONSTRAINT chk_password_length CHECK (CHAR_LENGTH(password) >= 8),
10      CONSTRAINT unique_username UNIQUE (username),
11      CONSTRAINT unique_email UNIQUE (email)
12  );
13
14  create table if not exists locations.devices
15  (
16      id INT AUTO_INCREMENT PRIMARY KEY,
17      name VARCHAR(255),
18      model VARCHAR(255),
19      user_id INT NOT NULL,
20      constraint fk_device_user
21      foreign key (user_id) references locations.users (id)
22  );
23
24  CREATE TABLE if not exists locations.positions (
25      id INT AUTO_INCREMENT PRIMARY KEY,
26      location POINT,
27      device_id INT NOT NULL,
28      description TEXT,
29      CONSTRAINT fk_position_device FOREIGN KEY (device_id) REFERENCES devices(id)
30  );
```

En la segunda migración se establece el script para insertar datos en las tablas. La Figura 85 muestra cómo se insertan registros entre entidades relacionadas:

Figura 85

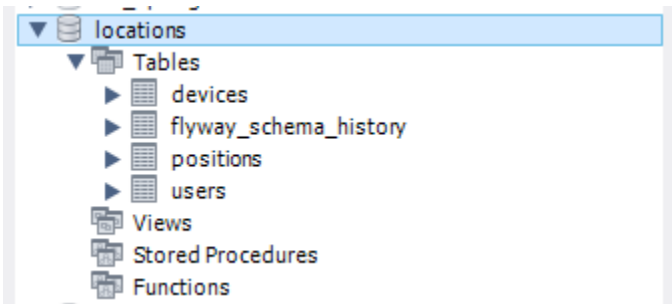
Estructura de intercesión de datos en SQL.

```
V2__Insert_initial_data.sql x
1
2  INSERT INTO users (name, username, email, password)
3  VALUES ('Juan Pérez', 'juanperez123', 'juan.perez@example.com', 'password123');
4
5
6  INSERT INTO devices (name, model, user_id)
7  VALUES ('Smartphone XYZ', 'Model 123', 1);
8
9
10 INSERT INTO positions (location, device_id, description)
11 VALUES (ST_GeomFromText('POINT(19.432608 -99.133209)'), 1, 'Ubicación en Ciudad de México');
```

Después de ejecutar la aplicación sin errores, en Workbench se visualizará una tabla adicional en la base de datos, denominada “Flyway_schema_history”. La estructura completa de la base de datos se muestra en la Figura 86:

Figura 86

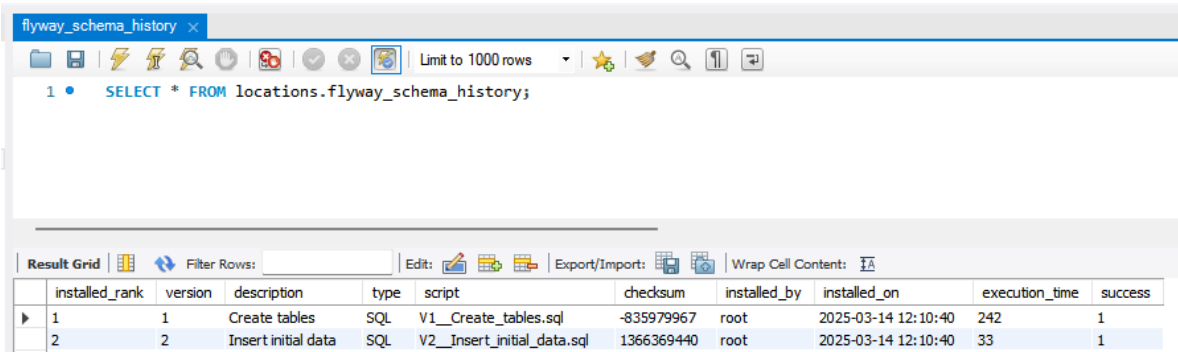
Base de datos en Workbench con sus tablas.



Las migraciones aplicadas se almacenan como se visualiza en la Figura 87:

Figura 87

Migraciones almacenadas en la tabla Flyway_schema_history.

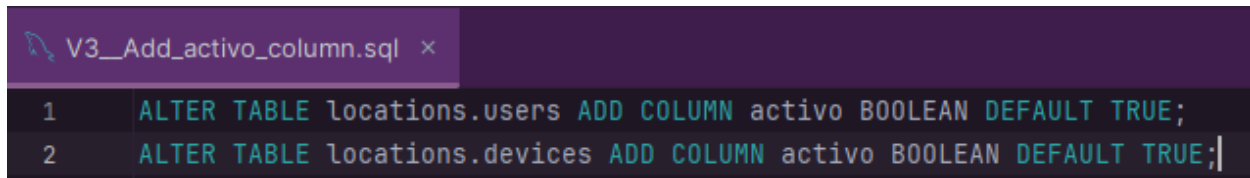


installed_rank	version	description	type	script	checksum	installed_by	installed_on	execution_time	success
1	1	Create tables	SQL	V1__Create_tables.sql	-835979967	root	2025-03-14 12:10:40	242	1
2	2	Insert initial data	SQL	V2__Insert_initial_data.sql	1366369440	root	2025-03-14 12:10:40	33	1

Además, se pueden crear múltiples scripts, por ejemplo, para añadir la columna "activo" a las tablas "users" y "devices" utilizando lenguaje SQL. El contenido de la tercera migración se detalla en la Figura 88:

Figura 88

Estructura de la tercera migración.

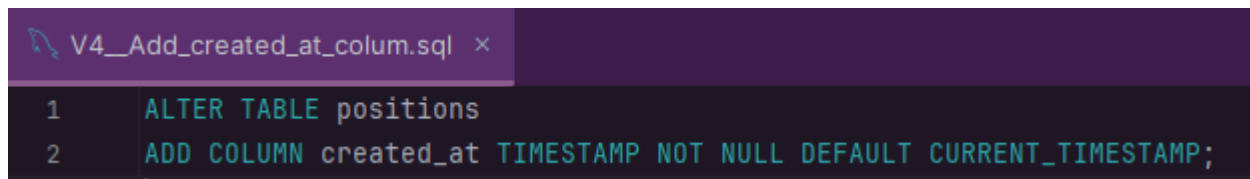


```
V3__Add_activo_column.sql x
1 ALTER TABLE locations.users ADD COLUMN activo BOOLEAN DEFAULT TRUE;
2 ALTER TABLE locations.devices ADD COLUMN activo BOOLEAN DEFAULT TRUE;
```

Por último, se agregó en la tabla “positions” el atributo “created_at” para guardar la fecha en tiempo real del guardado de ubicaciones. La Figura 89 contiene la estructura utilizada para la cuarta migración:

Figura 89

Estructura de la cuarta migración.



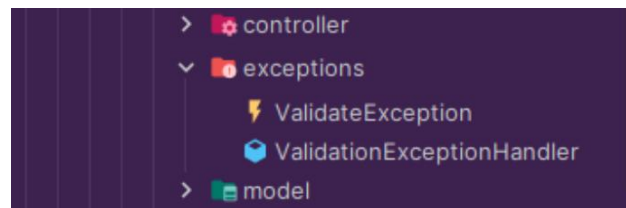
```
V4__Add_created_at_colum.sql x
1 ALTER TABLE positions
2 ADD COLUMN created_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP;
```

3.1.12. Manejo de control de errores para la administración de usuarios

Se crea un nuevo paquete denominado “exceptions” en el directorio principal del proyecto para almacenar las excepciones. Dentro de este paquete, se crearon dos clases: “ValidationExceptionHandler” y “ValidateException”. La creación del paquete para manejo de excepciones se presenta en la Figura 90:

Figura 90

Creación del paquete exceptions.



La clase “ValidationExceptionHandler” es un controlador *REST* para el manejo global de excepciones, anotado como `@RestControllerAdvice`. Su paquete base serán los controladores del proyecto. El manejo centralizado de errores se implementa como se ve en la Figura 91:

Figura 91

Clase ValidationExceptionHandler.

```
ValidationExceptionHandler.java x
1  package mx.t4g.geotracehub.exceptions;
2
3  import org.springframework.http.HttpStatus;
4  import org.springframework.validation.FieldError;
5  import org.springframework.web.bind.MethodArgumentNotValidException;
6  import org.springframework.web.bind.annotation.ExceptionHandler;
7  import org.springframework.web.bind.annotation.ResponseStatus;
8  import org.springframework.web.bind.annotation.RestControllerAdvice;
9
10 import java.util.Collections;
11 import java.util.HashMap;
12 import java.util.Map;
13
14 @RestControllerAdvice(basePackages = "mx.t4g.geotracehub.controller") no usages Sunflower-S14 *
15 public class ValidationExceptionHandler {
16
```

Se declararon dos métodos. El primero, “handleValidationExceptions”, permite capturar la excepción “MethodArgumentNotValidException” (incorporada en Java) para gestionar los errores durante el registro de usuarios. Esto permite mostrar al usuario, de manera clara, los fallos que

impiden guardar el registro por el incumplimiento de las validaciones. La lógica para manejar excepciones de validación se muestra en la Figura 92:

Figura 92

Método `handleValidationExceptions`.

```
18 // Manejo de validaciones de @Valid (400)
19 @ResponseStatus(HttpStatus.BAD_REQUEST) no usages Sunflower-S14
20 @ExceptionHandler(MethodArgumentNotValidException.class)
21 @
22 public Map<String, String> handleValidationExceptions(MethodArgumentNotValidException ex) {
23     Map<String, String> errors = new HashMap<>();
24     ex.getBindingResult().getAllErrors().forEach((ObjectError error) -> {
25         String fieldName = ((FieldError) error).getField();
26         String errorMessage = error.getDefaultMessage();
27         errors.put(fieldName, errorMessage);
28     });
29     return errors;
30 }
```

El segundo método, "handleUserAlreadyExistsException", utiliza la clase "ValidateException" para retornar un mensaje que indica la existencia de un usuario. Esto impide que se cree un nuevo registro si el usuario ya ha sido registrado previamente con el mismo nombre o correo electrónico. El tratamiento de errores por duplicados se encuentra en la Figura 93:

Figura 93

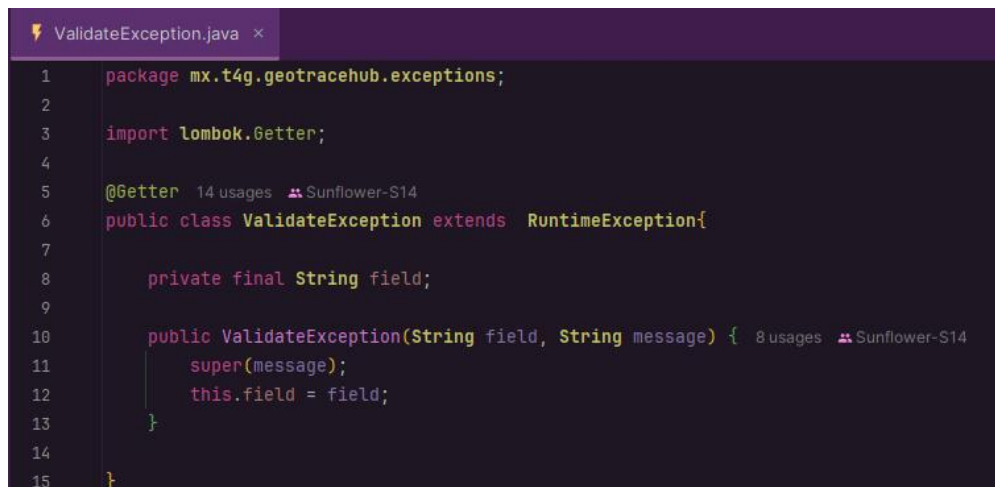
Método `handleUserAlreadyExistsException`.

```
30
31 // Manejo de existencia de usuario (400)
32 @ResponseStatus(HttpStatus.BAD_REQUEST) no usages Sunflower-S14
33 @ExceptionHandler(ValidateException.class)
34 @
35 public Map<String, String> handleUserAlreadyExistsException(ValidateException ex) {
36     return Collections.singletonMap(ex.getField(), ex.getMessage());
37 }
```

La segunda clase es "ValidateException", permite obtener el mensaje de error obtenido en las validaciones de cuentas existentes. La definición de una excepción personalizada se presenta en la Figura 94:

Figura 94

Clase ValidateException.



```
1 package mx.t4g.geotracehub.exceptions;
2
3 import lombok.Getter;
4
5 @Getter 14 usages Sunflower-S14
6 public class ValidateException extends RuntimeException{
7
8     private final String field;
9
10    public ValidateException(String field, String message) { 8 usages Sunflower-S14
11        super(message);
12        this.field = field;
13    }
14
15 }
```

Después se modificó la clase “UserServiceImpl” al método “saveUser”. Se implementaron condicionales que llaman al repositorio para verificar si el nombre de usuario es existente y este lance la excepción personalizada, de la misma forma para el correo electrónico, si ya previamente se registró, lanzará la excepción con los mensajes que visualizará el usuario. La Figura 95 muestra el punto donde se lanza la excepción ValidateException:

Figura 95

Lanzamiento de la excepción en el método saveUser del service implements.



```
16 public class UserServiceImpl implements UserService {
17
18     @Override 2 usages Sunflower-S14 +1
19     public User saveUser(User user) {
20
21         if (userRepository.existsByUsername(user.getUsername())){
22             throw new UserAlreadyExistsException("Nombre de usuario", "El nombre de usuario ya está registrado");
23         }
24
25         if (userRepository.existsByEmail(user.getEmail())){
26             throw new UserAlreadyExistsException("Email", "El correo ya está registrado");
27         }
28
29         log.info("Saving user: {}", user);
30         return userRepository.save(user);
31     }
32 }
```

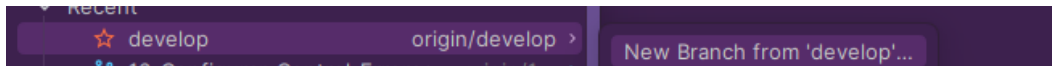
Posteriormente se crean más excepciones personalizadas dentro del paquete de exceptions.

3.1.13. Subir cambios por tickets

Primero se crea una rama sobre la que se está trabajando (develop). La gestión de ramas inicia como se ilustra en la Figura 96:

Figura 96

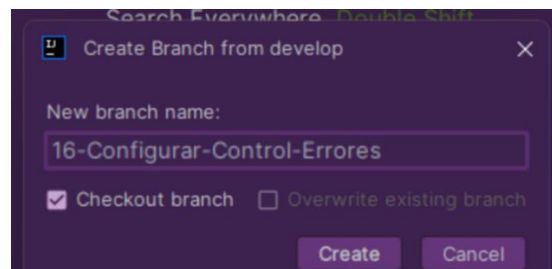
Creación de una rama.



Al momento de crear la rama se indica el número de ticket de la tarea asignada por *GitLab* y su nombre. Para finalizar se marca en la casilla “Checkout branch” para así cambiar a la rama creada. El nombramiento de nuevas ramas se realiza como se observa en la Figura 97:

Figura 97

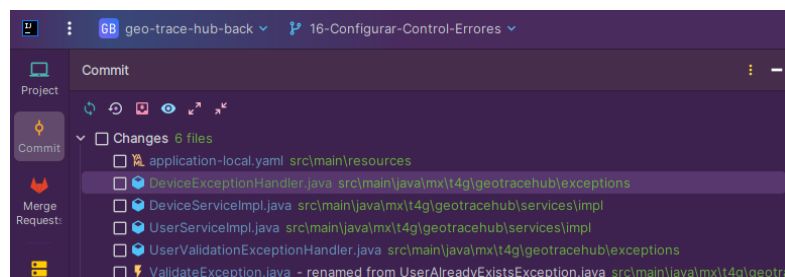
Asignación del nombre de la rama.



En la rama se modifican los archivos necesarios para completar la tarea. Posteriormente en la sección de “Commit” del editor, se visualizan los archivos que se han modificado. Los archivos modificados se visualizan en la Figura 98:

Figura 98

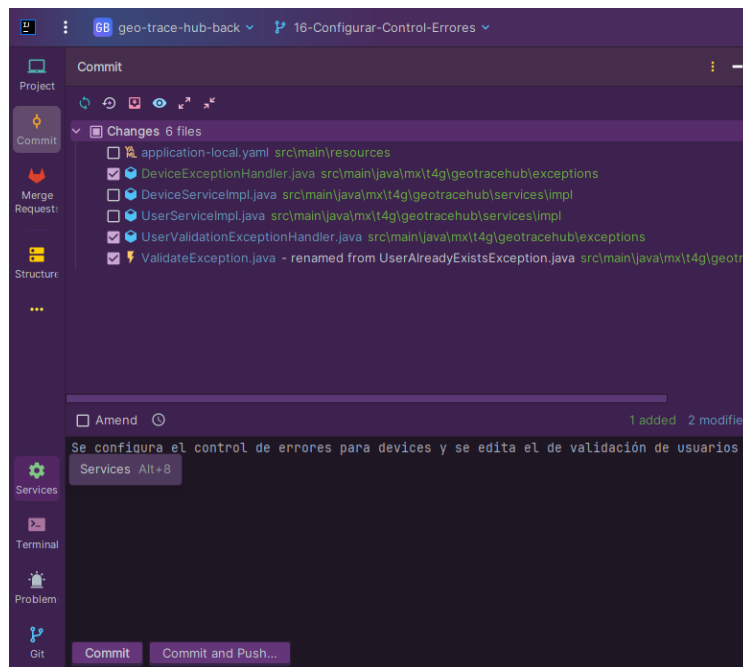
Archivos con los cambios realizados.



Se seleccionan los archivos que pertenecen a la tarea o ticket, permite escribir el mensaje del commit, finalizando se tiene que dar clic en el botón de “Commit and Push”. La selección de archivos antes del commit se ilustra en la Figura 99:

Figura 99

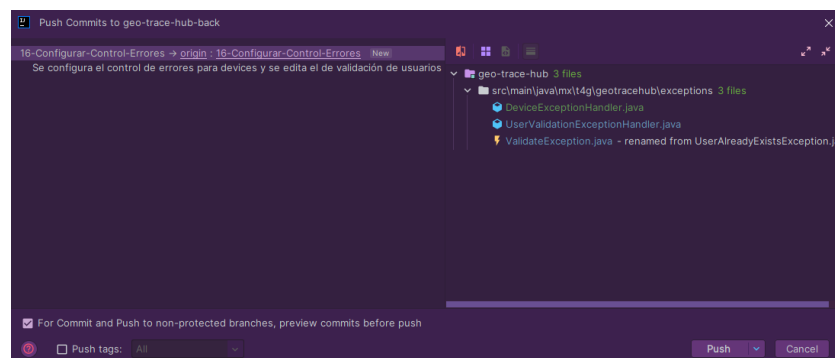
Selección de los archivos para crear un commit.



Se mostrará una vista previa de los archivos seleccionados para el commit y el nombre de la rama en la que se aplicará, para luego hacer clic en “Push”. La Figura 100 muestra la vista previa antes de confirmar un commit:

Figura 100

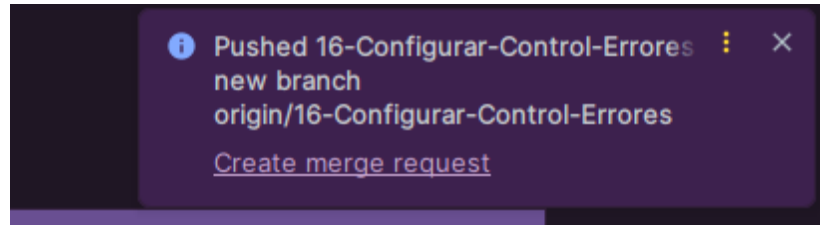
Vista previa de los archivos con commit.



Luego, aparecerá un mensaje en el editor donde se deberá hacer clic en “Create merge request” como se muestra en la Figura 101:

Figura 101

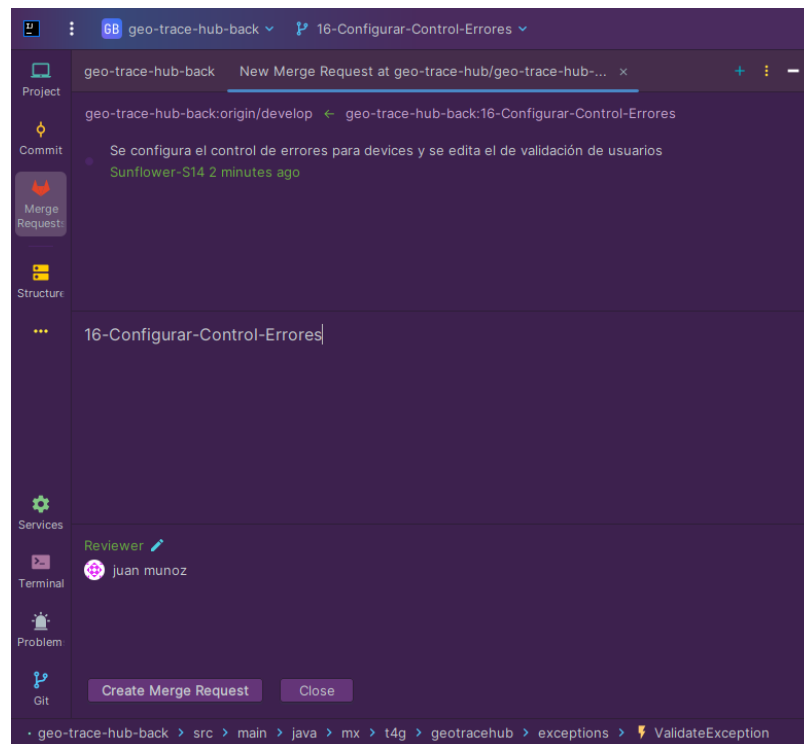
Mensaje para crear el merge request.



Después se muestra la siguiente ventana, en la que se agrega el nombre de la rama y se asigna al Scrum Master para que revise los cambios. Se hace clic en "Create Merge Request". "El proceso de creación de una solicitud de fusión se representa en la Figura 102:

Figura 102

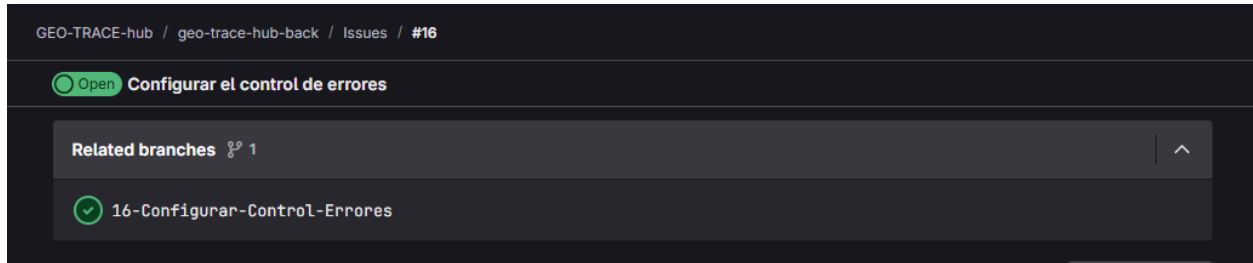
Creación del merge request.



Se tiene que esperar a la revisión de los cambios. En GitLab se enlaza la tarea del ticket con la rama que se creó La Figura 103 muestra el merge request creado:

Figura 103

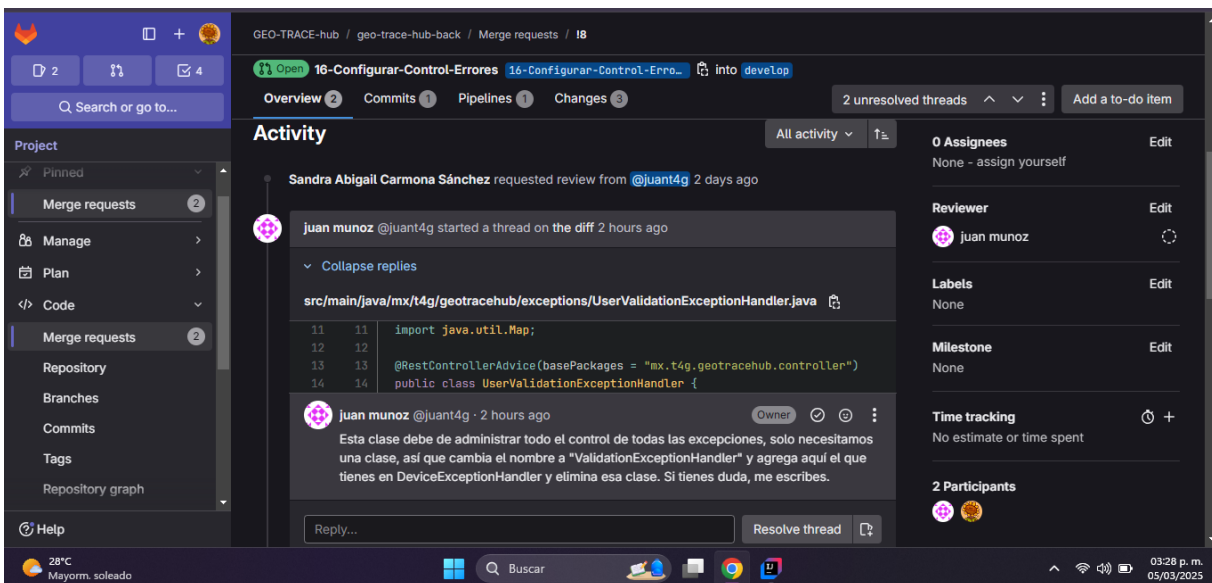
Merge request enlazado al ticket en GitLab.



Cuando se aprueban los cambios, se añaden al repositorio. En caso contrario, se deben realizar correcciones en la rama, basándose en los comentarios añadidos por el revisor, puede verse en la Figura 104:

Figura 104

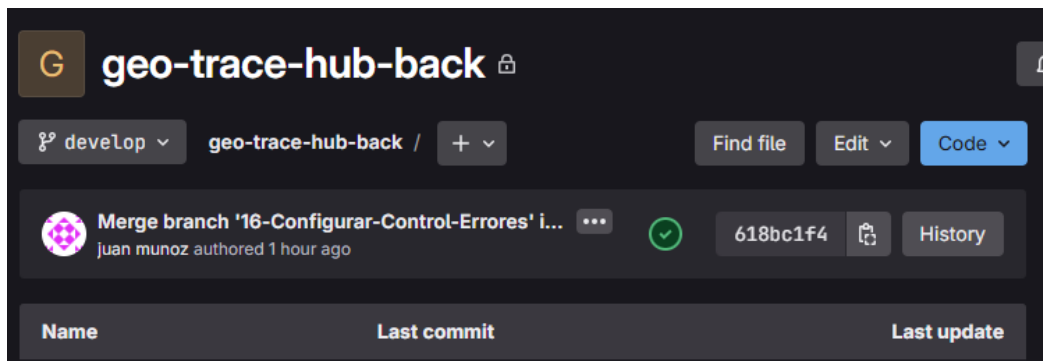
Ejemplo de observaciones a modificar.



Después de atender las observaciones, se crea un nuevo Commit & Push en la sección de "Commit" del IDE. Tras aprobar la fusión, los cambios se integran en la rama "develop", como se muestra en la Figura 105:

Figura 105

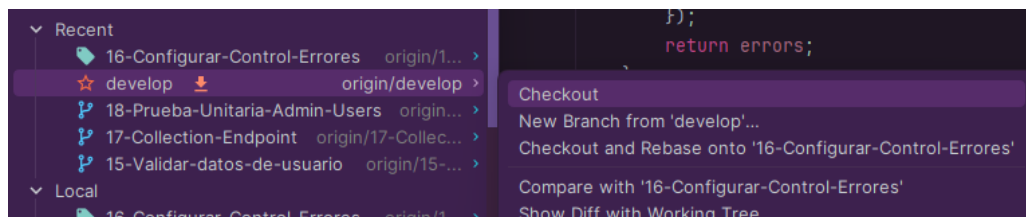
Fusión de las ramas en el repositorio.



Posteriormente, se pasa a la rama principal (que es develop) en la que se desarrolla el proyecto ilustrado en la Figura 106:

Figura 106

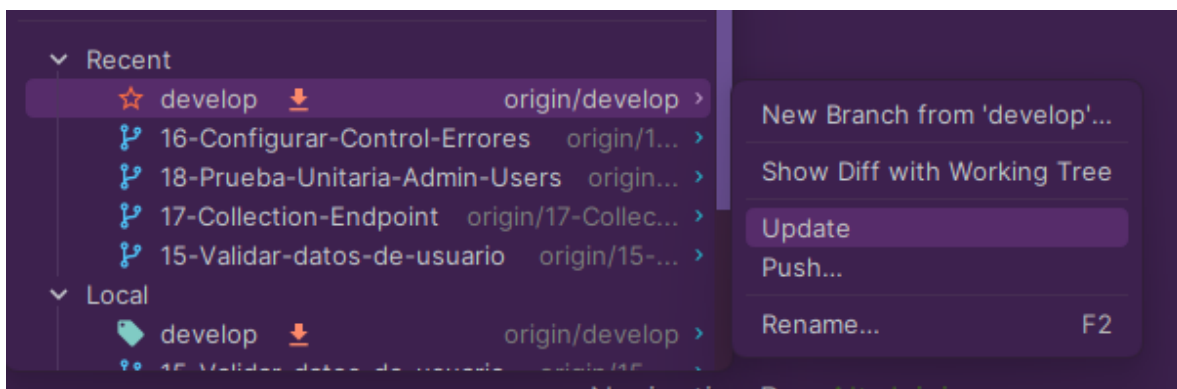
Cambio a la rama principal.



En la Figura 107 se actualiza la rama para descargar los cambios del repositorio:

Figura 107

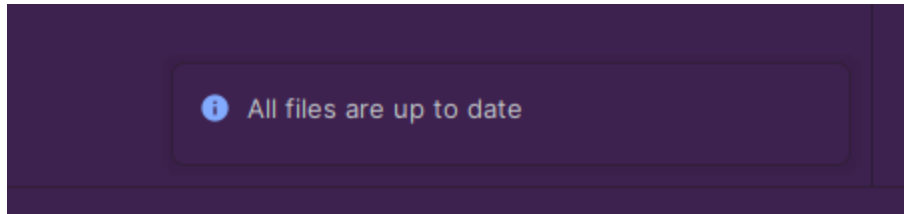
Actualización de la rama principal.



Aparecerá el mensaje de que todos los archivos han sido actualizados, mostrada en la Figura 108:

Figura 108

Mensaje de actualización de los cambios.



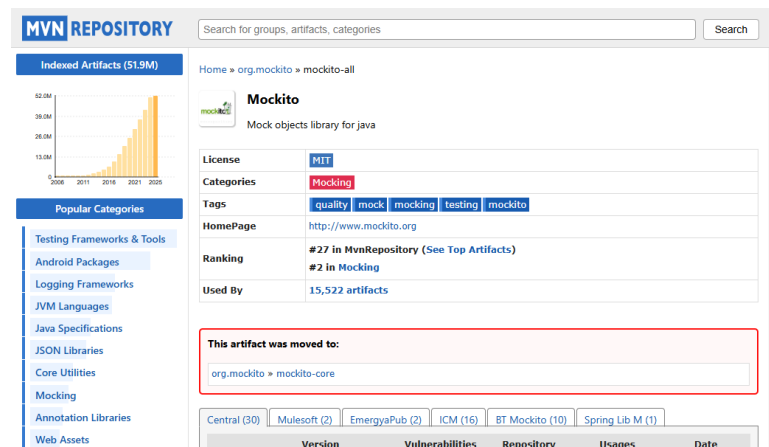
3.1.14. Pruebas unitarias para la administración de usuarios

Metodología: utilizando el framework JUnit y la biblioteca Mockito para la creación de objetos mock para aislar la lógica de negocio de la clase “UsersServiceImpl” y probar sus métodos de manera independiente, para probar la lógica de negocio de manera efectiva y garantizar la calidad del código.

Para utilizar la biblioteca de Mockito, se accede al repositorio de Maven alojado en el siguiente enlace <https://mvnrepository.com/artifact/org.mockito/mockito-all>, como se muestra en la Figura 109:

Figura 109

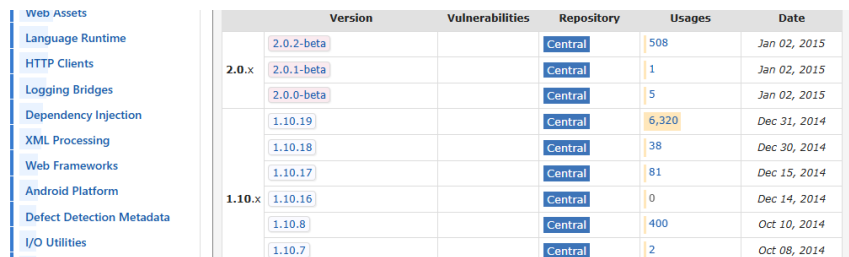
Página oficial de MVN REPOSITORY.



Al navegar, se encuentran sus diferentes versiones, incluyendo tanto las betas como las estables. La Figura 110 muestra la búsqueda de la dependencia Mockito en MVN Repository:

Figura 110

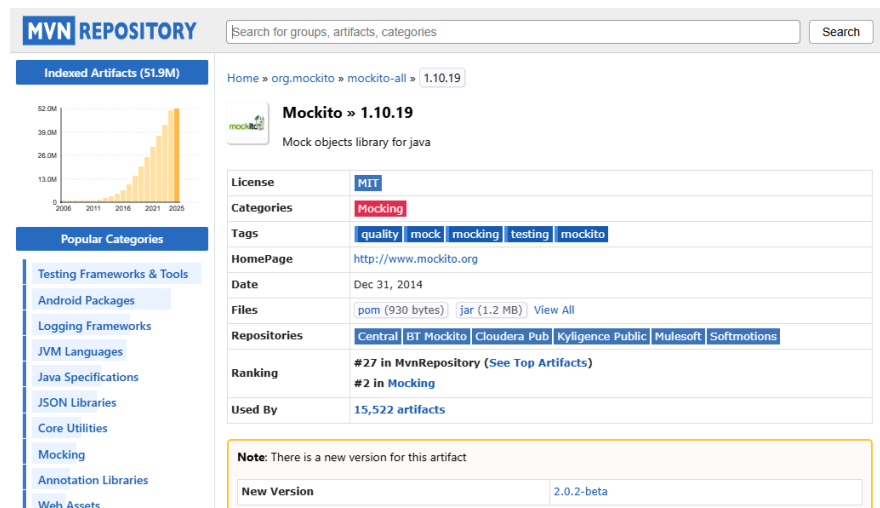
Versiones de Mockito.



Se selecciona la 1.10.19, la cual es la versión más usada, mostrada en la Figura 111:

Figura 111

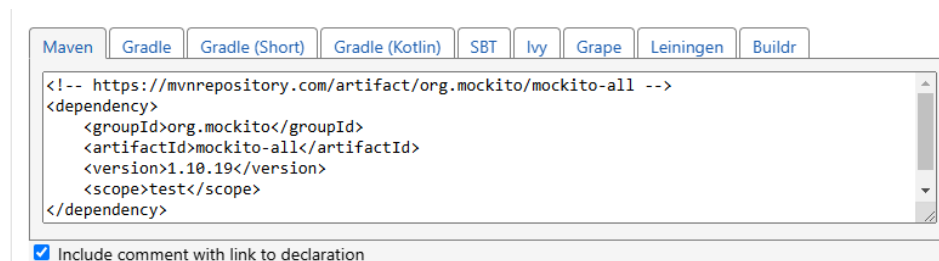
Versión de Mockito 1.10.19.



Se encuentra la dependencia para añadirla al proyecto de Spring Boot, tal y como se muestra en la Figura 112:

Figura 112

Dependencia de Mockito en Maven.



Se agrega la dependencia al archivo *pom.xml* del proyecto. La Figura 103 contiene la dependencia JUnit añadida para pruebas unitarias:

Figura 113

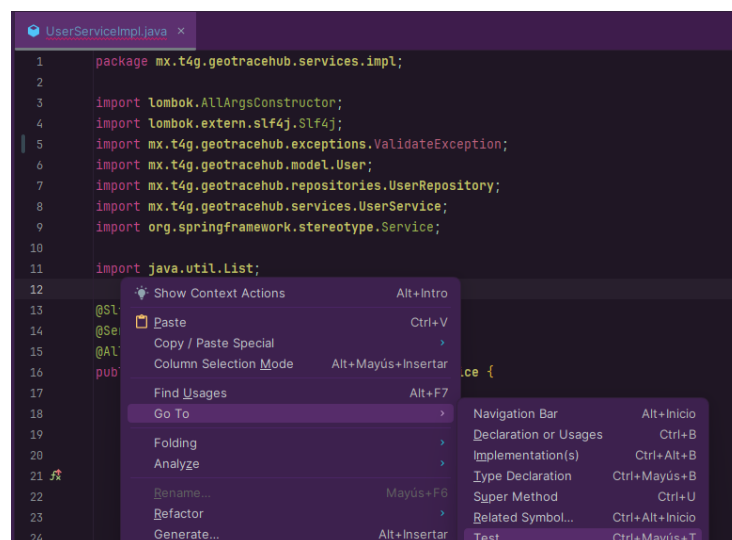
Dependencia de Mockito en el pom.xml.

```
72      <!-- https://mvnrepository.com/artifact/org.mockito/mockito-all -->
73      <dependency>
74          <groupId>org.mockito</groupId>
75          <artifactId>mockito-all</artifactId>
76          <version>1.10.19</version>
77          <scope>test</scope>
78      </dependency>
79  </dependencies>
```

Se sincronizan los cambios para que se descargue la dependencia. Para comenzar con las pruebas unitarias, se ingresará a “UserServiceImpl”. Gracias a IntelliJ, es posible realizar las pruebas a través de la opción “Test”. La clase encargada de validar los métodos del servicio de usuario se presenta en la Figura 114:

Figura 114

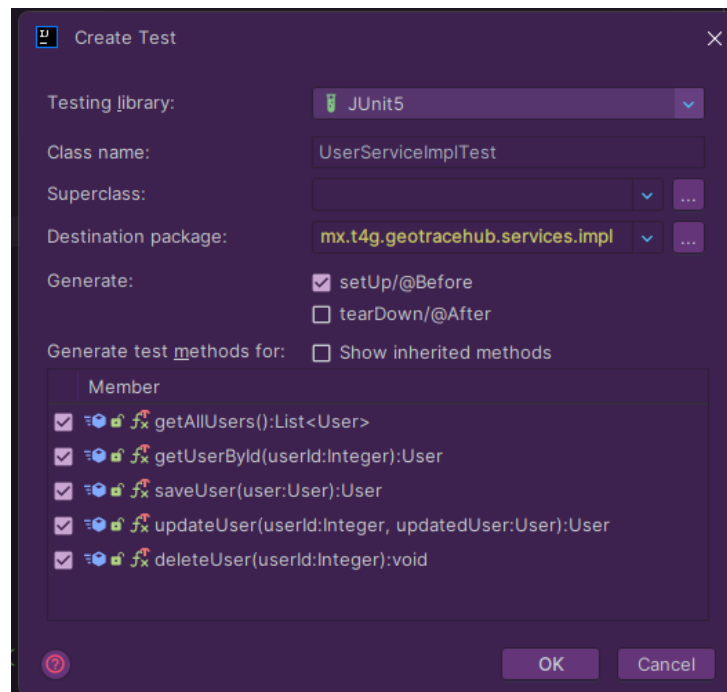
Clase UserServiceImpl para pruebas unitarias.



Se mostrarán los métodos creados en el "UserServiceImpl", en el que eligen todos los métodos y se marcará la casilla setUp/@Before, lo que permitirá inicializar la configuración de Mockito mostrada en la Figura 115:

Figura 115

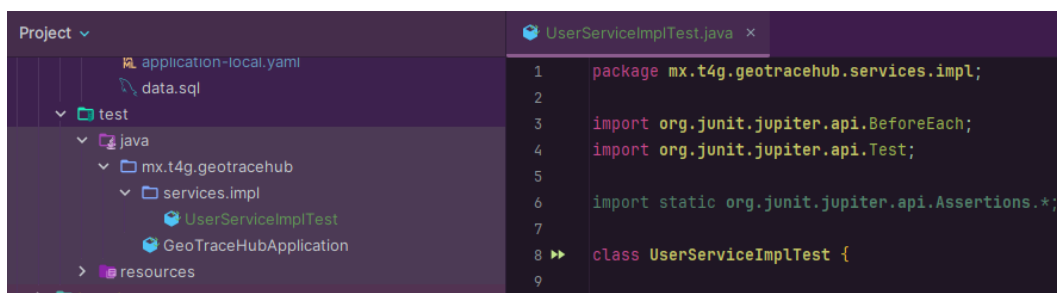
Selección de los métodos a ejecutar en pruebas unitarias.



Se creará una nueva clase, denominada "UserServiceImpl", que se almacenará en la carpeta "test" generada por el proyecto. Esta clase generará métodos para la prueba, en los que se deberá declarar la lógica correspondiente, ilustrada en la Figura 116:

Figura 116

Creación de la clase UserServiceImplTest.



Para utilizar Mockito en las pruebas, se declaran atributos como se observa en la Figura 117:

Figura 117

Declaración de los atributos.

```
class UserServiceImplTest { new *  
  
    @Mock 1 usage  
    private UserRepository userRepository;  
  
    @InjectMocks 1 usage  
    private UserServiceImpl userService;
```

Anotados con la anotación `@Mock` al atributo private “UserRepository userRepository” en el que se declara un objeto mock que simula el comportamiento del repositorio de usuarios, permitiendo aislar la dependencia del repositorio en la clase de servicio reflejado en la Figura 118:

Figura 118

Declaración del mock a usar.

```
@Mock 1 usage  
private UserRepository userRepository;
```

El siguiente atributo “UserServiceImpl userService” utiliza `@InjectMocks` para inyectar el objeto mock “UserRepository” (declarado previamente) dentro del objeto real que es “UserServiceImpl”, esto permite probar el comportamiento de “UserServiceImpl” sin necesidad de una conexión real a la base de datos ilustrada en la Figura 119:

Figura 119

Declaración de la inyección del mock.

```
@InjectMocks 1 usage  
private UserServiceImpl userService;
```

La configuración del entorno de prueba se muestra en la Figura 120, en la que se crea un objeto usuario que asegura a cada prueba unitaria comience con un estado conocido:

Figura 120

Configuración del método setUp.

```
private User user; 10 usages
@BeforeEach new *
void setUp() {
    MockitoAnnotations.initMocks( testClass: this);

    user = new User();
    user.setId(1);
    user.setUsername("carmona_san14");
    user.setName("Abigail");
    user.setEmail("abigailcarmona237@gmail.com");
    user.setPassword("contraseña_1234");
}
```

Es decir, se crea un nuevo objeto “User” y se establecen sus propiedades con valores de prueba que se utilizará en las pruebas unitarias para simular un usuario real, este método usa la anotación `@BeforeEach` que indica que el método “setUp()” se debe ejecutar antes de cada prueba unitaria para así tener datos para ejecutar con éxito las demás pruebas.

El siguiente método indicado en la Figura 121, se anotó con `@Test` permitirá realizar la prueba al primer método del “UserServiceImpl” en el que se obtienen todos los usuarios:

Figura 121

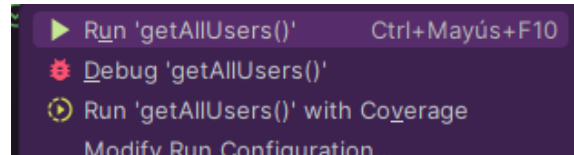
Método getAllUsers del test unitario.

```
@Test new *
void getAllUsers() {
    when(userRepository.findAll()).thenReturn(Arrays.asList(user));
    assertNotNull(userService.getAllUsers());
    assertEquals( expected: 1, userService.getAllUsers().size());
}
```

Ya que el método está configurado para probar la prueba, se da clic en el botón de “play” cerca del método como se aprecia en la Figura 122:

Figura 122

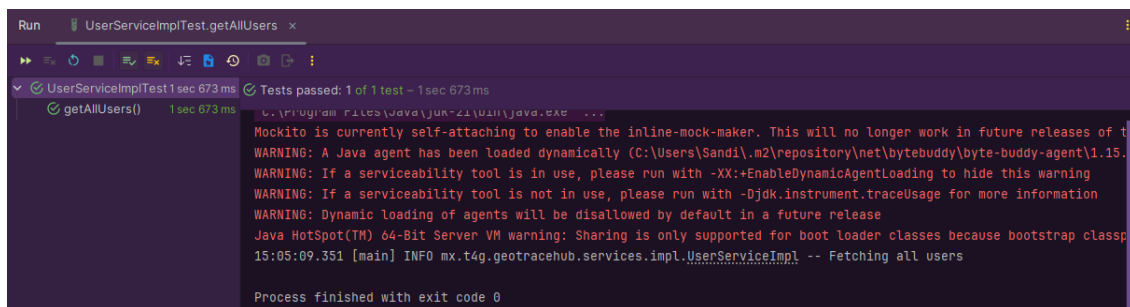
Acción de ejecución del método.



En la Figura 123 se evidencian algunos *warnings* sobre la interacción de las dependencias pero no son errores, esto no tiene que ver con la ejecución de la prueba unitaria porque la prueba se marca el color verde con palomitas e imprime “Test passed”, lo que indica que la prueba pasó correctamente y en los logs muestra el mensaje “Fetching all users”:

Figura 123

Ejecución del test del método getAllUsers.



Cuando una prueba se consideró exitosa, el IDE marcará al método con una palomita como se aprecia en la Figura 124:

Figura 124

Prueba aprobada del método getAllUsers.



Posteriormente, la Figura 125 ilustra el método “getUserById()”, se verifica que se puede buscar un usuario por su ID:

Figura 125

Método getUserById del test unitario.

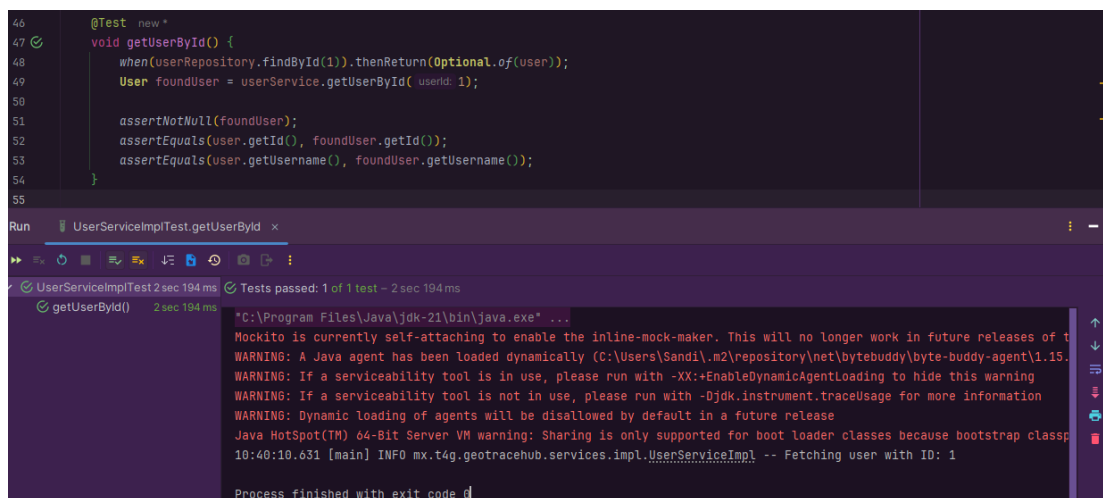
```
@Test new *
void getUserById() {
    when(userRepository.findById(1)).thenReturn(Optional.of(user));
    User foundUser = userService.getUserById(1);

    assertNotNull(foundUser);
    assertEquals(user.getId(), foundUser.getId());
    assertEquals(user.getUsername(), foundUser.getUsername());
}
```

Al ejecutar en la Figura 126 aparece el mensaje del log mostrando el mensaje “Fetching user with ID:1”. Se visualiza que marca que el método pasó la prueba:

Figura 126

Ejecución del test del método getUserById.



The screenshot displays an IDE with a Java unit test and its execution output. The test code is as follows:

```
46 @Test new *
47 void getUserById() {
48     when(userRepository.findById(1)).thenReturn(Optional.of(user));
49     User foundUser = userService.getUserById(1);
50
51     assertNotNull(foundUser);
52     assertEquals(user.getId(), foundUser.getId());
53     assertEquals(user.getUsername(), foundUser.getUsername());
54 }
55
```

Below the code, the 'Run' tab shows the test results:

- UserServiceImplTest.getUserById x
- UserServiceImplTest 2 sec 194 ms
- Tests passed: 1 of 1 test - 2 sec 194 ms
- getUserById() 2 sec 194 ms

The console output shows the following log messages:

```
"C:\Program Files\Java\jdk-21\bin\java.exe" ...
Mockito is currently self-attaching to enable the inline-mock-maker. This will no longer work in future releases of t
WARNING: A Java agent has been loaded dynamically (C:\Users\Sandi\.m2\repository\net\bytebuddy\byte-buddy-agent\1.15.
WARNING: If a serviceability tool is in use, please run with -XX:+EnableDynamicAgentLoading to hide this warning
WARNING: If a serviceability tool is not in use, please run with -Djdk.instrument.traceUsage for more information
WARNING: Dynamic loading of agents will be disallowed by default in a future release
Java HotSpot(TM) 64-Bit Server VM warning: Sharing is only supported for boot loader classes because bootstrap classp
10:40:10.631 [main] INFO mx.t4g.geotracehub.services.impl.UserServiceImpl -- Fetching user with ID: 1
Process finished with exit code 0
```

La configuración del método “saveUser” se muestra en la Figura 127 permite guardar a un usuario, usando mockito en el test permite inyectar los datos declarados:

Figura 127

Método saveUser del test unitario.

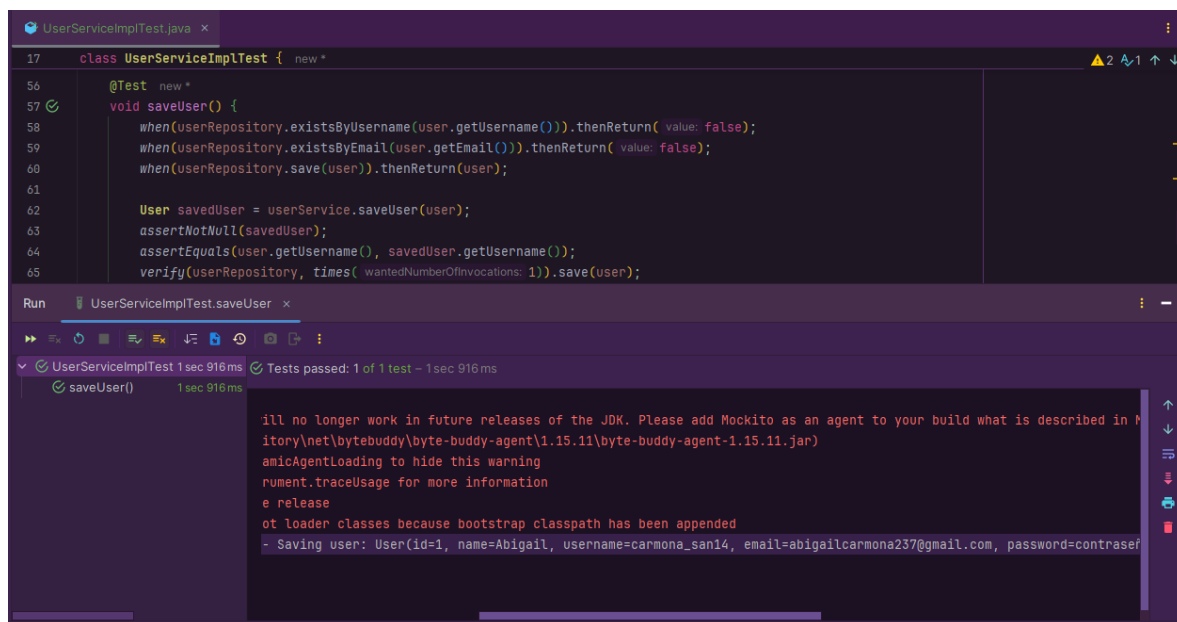
```
@Test new *
void saveUser() {
    when(userRepository.existsByUsername(user.getUsername())).thenReturn( value: false);
    when(userRepository.existsByEmail(user.getEmail())).thenReturn( value: false);
    when(userRepository.save(user)).thenReturn(user);

    User savedUser = userService.saveUser(user);
    assertNotNull(savedUser);
    assertEquals(user.getUsername(), savedUser.getUsername());
    verify(userRepository, times( wantedNumberOfInvocations: 1)).save(user);
}
```

En la Figura 128 se presenta el resultado de ejecución. Se visualiza que se aprobó la prueba y en el log se muestra el mensaje “Saving user: User(id=1, name=Abigail, username=carmona_san14, email=abigailcarmona237@gmail.com, password=contraseña_1234, devices=null)”:

Figura 128

Ejecución del test del método saveUser.



```
class UserServiceImplTest {
    @Test new *
    void saveUser() {
        when(userRepository.existsByUsername(user.getUsername())).thenReturn( value: false);
        when(userRepository.existsByEmail(user.getEmail())).thenReturn( value: false);
        when(userRepository.save(user)).thenReturn(user);

        User savedUser = userService.saveUser(user);
        assertNotNull(savedUser);
        assertEquals(user.getUsername(), savedUser.getUsername());
        verify(userRepository, times( wantedNumberOfInvocations: 1)).save(user);
    }
}

Run UserServiceImplTest.saveUser
Tests passed: 1 of 1 test - 1 sec 916 ms
saveUser() 1 sec 916 ms

Saving user: User(id=1, name=Abigail, username=carmona_san14, email=abigailcarmona237@gmail.com, password=contraseña_1234, devices=null)
```

El siguiente método de test es “updateUser()” se encarga de actualizar correctamente los datos de un usuario existente y retorna el objeto actualizado. Primero se crea un nuevo objeto “User” que utilice los datos que se declararon para la actualización. Se tiene que encontrar el registro por su ID y al guardarlo retornara al objeto “updatedUser” que verifica el objeto no sea nulo, también verifica que las propiedades del objeto *result* sean iguales a las propiedades del objeto “updatedUser”, esto se muestra en la Figura 129:

Figura 129

Método updateUser del test unitario.

```
68      @Test new *
69      void updateUser() {
70          User updatedUser = new User();
71          updatedUser.setName("Carmen");
72          updatedUser.setEmail("carmen@gmail.com");
73          updatedUser.setUsername("mcar_SH14");
74
75          when(userRepository.findById(1)).thenReturn(Optional.of(user));
76          when(userRepository.save(any(User.class))).thenReturn(updatedUser);
77
78          User result = userService.updateUser( userId: 1, updatedUser);
79
80          assertNotNull(result);
81          assertEquals(updatedUser.getName(), result.getName());
82          assertEquals(updatedUser.getEmail(), result.getEmail());
83          assertEquals(updatedUser.getUsername(), result.getUsername());
84      }
```

La información mostrada en la Figura 130 corresponde a la ejecución del método. Esta marca que se aprobó la prueba y como log imprimió correctamente: “Updating user with ID: 1”

Figura 130

Ejecución del test del método updateUser.



```
68      @Test new *
69      void updateUser() {
70          User updatedUser = new User();
71          updatedUser.setName("Carmen");
72          updatedUser.setEmail("carmen@gmail.com");
73          updatedUser.setUsername("mcar_SH14");
74
75          when(userRepository.findById(1)).thenReturn(Optional.of(user));
76          when(userRepository.save(any(User.class))).thenReturn(updatedUser);
77
78          User result = userService.updateUser( userId: 1, updatedUser);
79
80          assertNotNull(result);
81          assertEquals(updatedUser.getName(), result.getName());
82          assertEquals(updatedUser.getEmail(), result.getEmail());
83          assertEquals(updatedUser.getUsername(), result.getUsername());
84      }
```

Run UserServiceImplTest.updateUser

Tests passed: 1 of 1 test - 1 sec 663 ms

updateUser() 1 sec 663 ms

"C:\Program Files\Java\jdk-21\bin\java.exe" ...

Mockito is currently self-attaching to enable the inline mock-maker. This will no longer work in future releases of Mockito.

WARNING: A Java agent has been loaded dynamically (C:\Users\Sandil\m2\repository\net\bytebuddy\byte-buddy-agent\1.15.0\byte-buddy-agent-1.15.0.jar). If you are using Mockito via the Mockito Gradle plugin, you should use Mockito 1.10.0 or earlier. If you are using Mockito via the Mockito Maven plugin, you should use Mockito 1.10.0 or earlier. If you are using Mockito via the Mockito JAR, you should use Mockito 1.10.0 or earlier.

WARNING: If a serviceability tool is in use, please run with -XX:+EnableDynamicAgentLoading to hide this warning.

WARNING: If a serviceability tool is not in use, please run with -Djdk.instrument.traceUsage for more information.

WARNING: Dynamic loading of agents will be disallowed by default in a future release.

Java HotSpot(TM) 64-Bit Server VM warning: Sharing is only supported for boot loader classes because bootstrap classpath is not well formed.

11:06:06.974 [main] INFO mx.t4g.geotracehub.services.impl.UserServiceImpl -- Updating user with ID: 1

Process finished with exit code 0

Finalmente, puede observarse en la Figura 131 el método "deleteUser()", se verificará que no se lance ninguna excepción al eliminar un usuario y que al llamar al método "deleteById()", se elimine el usuario especificado, en este caso el usuario que se inicializó en el método "Setup":

Figura 131

Método deleteUser del test unitario.

```
86      @Test new *
87      void deleteUser() {
88          when(userRepository.existsById(1)).thenReturn( value: true);
89          doNothing().when(userRepository).deleteById(1);
90
91          assertDoesNotThrow(() -> userService.deleteUser( userid: 1));
92          verify(userRepository, times( wantedNumberOfInvocations: 1)).deleteById(1);
93      }
```

Al ejecutar la prueba marca que la prueba pasó y en el log se mostrará el mensaje “Deleting user with ID: 1”, mostrada de manera clara en la Figura 132:

Figura 132

Ejecución del test del método deleteUser.



```
86      @Test new *
87      void deleteUser() {
88          when(userRepository.existsById(1)).thenReturn( value: true);
89          doNothing().when(userRepository).deleteById(1);
90
91          assertDoesNotThrow(() -> userService.deleteUser( userid: 1));
92          verify(userRepository, times( wantedNumberOfInvocations: 1)).deleteById(1);
93      }
```

Run UserServiceImplTest.deleteUser x

UserServiceImplTest 1 sec 554 ms Tests passed: 1 of 1 test - 1 sec 554 ms

deleteUser() 1 sec 554 ms

"C:\Program Files\Java\jdk-21\bin\java.exe" ...

Mockito is currently self-attaching to enable the inline-mock-maker. This will no longer work in future releases of t

WARNING: A Java agent has been loaded dynamically (C:\Users\Sand1\.m2\repository\net\bytebuddy\byte-buddy-agent\1.15.

WARNING: If a serviceability tool is in use, please run with -XX:+EnableDynamicAgentLoading to hide this warning

WARNING: If a serviceability tool is not in use, please run with -Djdk.instrument.traceUsage for more information

WARNING: Dynamic loading of agents will be disallowed by default in a future release

Java HotSpot(TM) 64-Bit Server VM warning: Sharing is only supported for boot loader classes because bootstrap classp

11:06:50.813 [main] INFO mx.t4g.geotracehub.services.impl.UserServiceImpl -- Deleting user with ID: 1

Process finished with exit code 0

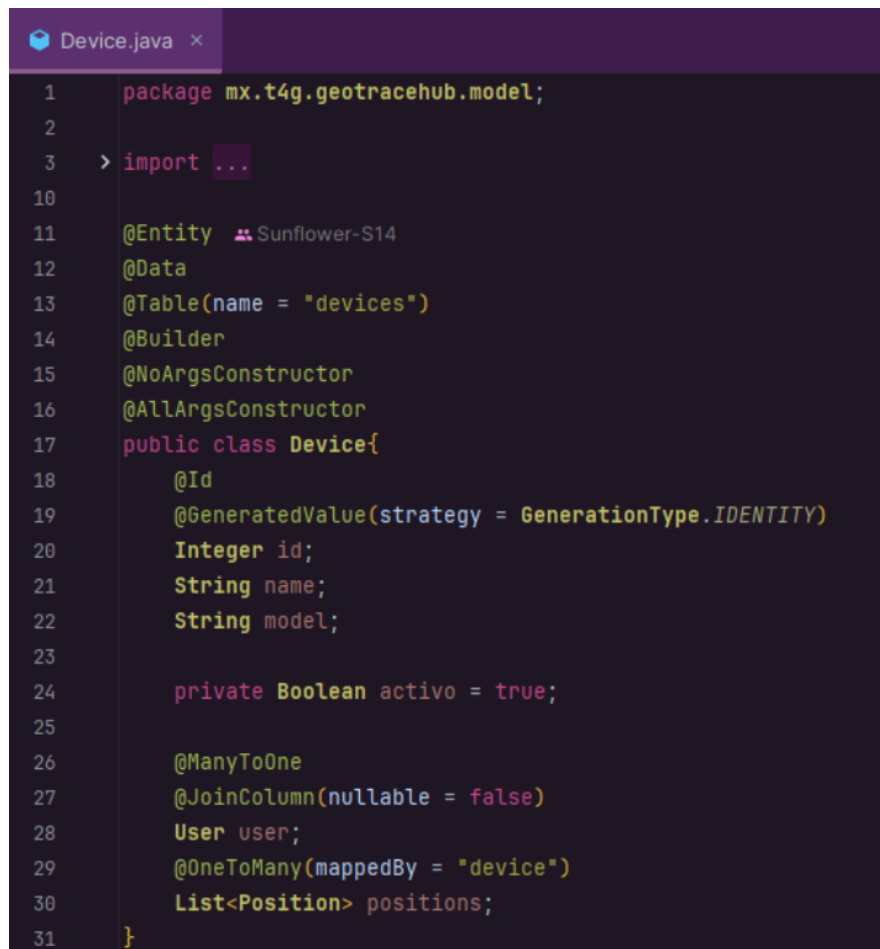
Cuando una prueba unitaria a un método no pasa la prueba, ya sea porque no se declaró de manera adecuada o no funciona el método original, el IDE lo marcará en rojo con el icono de una “X”.

3.1.15. Administración de dispositivos

Modelo o entidad dispositivo: en la carpeta de model se creó un nuevo modelo, el cual se llamó “Device” en el que Hibernate convertirá este modelo en una tabla. La Figura 133 muestra la declaración detallada de los atributos como es el ID, el nombre del dispositivo, el modelo del dispositivo, un campo de activo para que puedan activar y desactivar los dispositivos y en el que se declaran las llaves foráneas a la tabla “users” con *@ManyToMany* (muchos a muchos) como a “positions” con *@OneToMany* (uno a muchos) para que Hibernate cree las relaciones en las tablas:

Figura 133

Clase Device.



```
1 package mx.t4g.geotracehub.model;
2
3 > import ...
10
11 @Entity
12 @Data
13 @Table(name = "devices")
14 @Builder
15 @NoArgsConstructor
16 @AllArgsConstructor
17 public class Device{
18     @Id
19     @GeneratedValue(strategy = GenerationType.IDENTITY)
20     Integer id;
21     String name;
22     String model;
23
24     private Boolean activo = true;
25
26     @ManyToOne
27     @JoinColumn(nullable = false)
28     User user;
29     @OneToMany(mappedBy = "device")
30     List<Position> positions;
31 }
```

Repository o Repositorio: se define la clase “DispositiveRepository” en el que se implementa JPA para las operaciones CRUD en la base de datos, aquí se definen dos consultas SQL en el que se declararon dos; el primero “findByActivoTrue” permite mostrar los dispositivos con estado de activo y el segundo “findByUserId” obtiene el ID del usuario, ambas declaraciones devuelven una lista de dispositivos. La funcionalidad descrita se ejemplifica en la Figura 134:

Figura 134

Interfaz DispositiveRepository.

```
1 package mx.t4g.geotracehub.repositories;
2
3 import mx.t4g.geotracehub.model.Device;
4 import org.springframework.data.jpa.repository.JpaRepository;
5 import org.springframework.stereotype.Repository;
6
7 import java.util.List;
8
9 @Repository 8 usages  Sunflower-S14
10 public interface DispositiveRepository extends JpaRepository<Device, Integer>
11     List<Device> findByActivoTrue(); 3 usages  Sunflower-S14
12     List<Device> findByUserId(Integer userId); 1 usage  Sunflower-S14
13 }
```

Service o servicio: se crea la interfaz “DeviceService” invoca a los métodos que contienen la lógica del negocio o en simples palabra, el service define qué acciones realizará con los datos, por ejemplo para “getAllDevices” permite obtener todos los dispositivos.

Con “getDeviceById” obtiene el dispositivo por su ID. En “saveDevice” permite guardar un dispositivo. Para “updateDevice” se actualiza la información del dispositivo. En “deviceStatus” actualiza el estado de un dispositivo. Con “deleteDevice” elimina un dispositivo y en “getDevicesByUserId” permite obtener todos los dispositivos de un usuario ilustrada en la Figura 135:

Figura 135

Interfaz DeviceService.

```
DeviceService.java x
1 package mx.t4g.geotracehub.services;
2
3 import mx.t4g.geotracehub.model.Device;
4 import java.util.List;
5
6 public interface DeviceService { 7 usages 1 implementation 🐞 Sunflower-S14
7     List<Device> getAllDevices(); 2 usages 1 implementation 🐞 Sunflower-S14
8     Device getDeviceById(Integer deviceId); 3 usages 1 implementation 🐞 Sunflower-S14
9     Device saveDevice(Device device); 3 usages 1 implementation 🐞 Sunflower-S14
10    Device updateDevice(Integer deviceId, Device updatedDevice); 3 usages 1 implementation 🐞 Sunflower-S14
11    void deviceStatus(Integer deviceId, boolean status); 2 usages 1 implementation 🐞 Sunflower-S14
12    void deleteDevice(Integer deviceId); 3 usages 1 implementation 🐞 Sunflower-S14
13    List<Device> getDevicesByUserId(Integer userId); 1 usage 1 implementation 🐞 Sunflower-S14
14 }
```

Servicio implementado o Service Implements: se creó la clase “DeviceServiceImpl” que implementa la interfaz “DeviceService” en el que se escribe la lógica de negocio de la gestión de dispositivos en el que se llama al repositorio para la interacción con el mismo en donde se ejecutan las operaciones SQL.

Figura 136

Métodos getAllDevices y deviceStatus en DeviceServiceImpl.

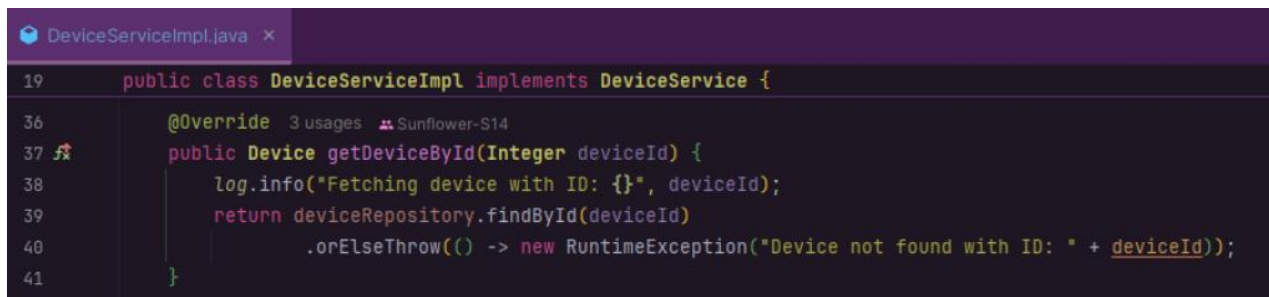
```
19 public class DeviceServiceImpl implements DeviceService {
20
21     private final DispositiveRepository deviceRepository;
22
23     @Override 2 usages 🐞 Sunflower-S14
24     public List<Device> getAllDevices() {
25         log.info("Fetching all devices");
26         return deviceRepository.findByActivoTrue();
27     }
28
29     public void deviceStatus(Integer deviceId, boolean status) { 2 usages 🐞 Sunflower-S14 *
30         log.info("Changing device status for ID: {}", deviceId);
31         Device device = deviceRepository.findById(deviceId)
32             .orElseThrow(() -> new RuntimeException("Device not found with ID: " + deviceId));
33         deviceRepository.save(device);
34     }
```

De ejemplo en el método “getAllDevices” llama al “deviceRepository” para que encuentre y regrese todos los dispositivos que estén activos y con “deviceStatus” busca primero un dispositivo por su ID, sino lo encuentra imprimirá una excepción y si es encontrado el dispositivo guardará el nuevo estado del dispositivo (para activarlo o desactivarlo), su estructura puede consultarse en la Figura 136.

El uso de “getDeviceById” se presenta en la Figura 137 que funciona para buscar el dispositivo por su ID en el que si es encontrado retorna el dispositivo, sino es encontrado devuelve una excepción genérica:

Figura 137

Método getDeviceById.

A screenshot of an IDE showing the implementation of the `getDeviceById` method in `DeviceServiceImpl.java`. The code is as follows:

```
19 public class DeviceServiceImpl implements DeviceService {  
36     @Override 3 usages Sunflower-S14  
37     public Device getDeviceById(Integer deviceId) {  
38         log.info("Fetching device with ID: {}", deviceId);  
39         return deviceRepository.findById(deviceId)  
40             .orElseThrow(() -> new RuntimeException("Device not found with ID: " + deviceId));  
41     }  
}
```

En la Figura 138 se muestra el método “saveDevice”, se realiza la operación de guardado de un dispositivo en el que llamará al repositorio para poder guardar el dispositivo a la base de datos:

Figura 138

Método saveDevice.

A screenshot of an IDE showing the implementation of the `saveDevice` method in `DeviceServiceImpl.java`. The code is as follows:

```
43     @Override 3 usages Sunflower-S14  
44     public Device saveDevice(Device device) {  
45         log.info("Saving device: {}", device);  
46         return deviceRepository.save(device);  
47     }  
}
```

De acuerdo con la Figura 139, se observa la estructura de “updateDevice” que usa el ID del dispositivo que se actualizará. Primero buscará el dispositivo por ID si lo encuentra si no se

modificó el nombre, descripción o el usuario no marcará como nulo el campo si no se realizó alguna modificación y si se modificó algún atributo tomará la información correctamente y permitirá guardar el dispositivo, sino encuentra el ID del dispositivo por ser inexistente en la base de datos mostrará una excepción:

Figura 139

Método updateDevice.

```
49      @Override 3 usages Sunflower-S14
50      public Device updateDevice(Integer deviceId, Device updatedDevice) {
51          log.info("Updating device with ID: {}", deviceId);
52          return deviceRepository.findById(deviceId).map( Device existingDevice -> {
53              existingDevice.setName(updatedDevice.getName());
54              existingDevice.setDescription(updatedDevice.getDescription());
55              existingDevice.setUser(updatedDevice.getUser());
56              return deviceRepository.save(existingDevice);
57          }).orElseThrow(() -> new RuntimeException("Device not found with ID: " + deviceId));
58      }
```

La funcionalidad del método “deleteDevice” se ejemplifica en la Figura 140 que verifica la existencia de un dispositivo por su ID, si lo encuentra procederá a eliminarse el dispositivo, pero sino lo encuentra mostrará una excepción:

Figura 140

Método deleteDevice.

```
@Override 3 usages Sunflower-S14
public void deleteDevice(Integer deviceId) {
    log.info("Deleting device with ID: {}", deviceId);
    if (!deviceRepository.existsById(deviceId)) {
        throw new RuntimeException("Device not found with ID: " + deviceId);
    }
    deviceRepository.deleteById(deviceId);
}
```

Para terminar, la Figura 141 muestra el método “getDeviceByUserId”, buscará mediante condiciones si el usuario del ID es nulo o menor de cero, lanzará una excepción personalizada, si el ID no cumple estas condiciones significará que podrá devolver una lista de dispositivos asociados al usuario:

Figura 141

Método `getDevicesById`.

```
public List<Device> getDevicesById(Integer userId) { 1 usage 🚩 Sunflower-S14 *
    log.info("Buscando dispositivos para el usuario ID: {}", userId);

    if (userId == null || userId <= 0) {
        throw new ValidateException("userId", "El ID de usuario debe ser válido");
    }

    List<Device> devices = deviceRepository.findById(userId);

    return devices;
}
```

En cada uno de estos métodos tienen declarado el *log.info* que permite mostrar en las pruebas unitarias y de integración si estos métodos están funcionando, también estos mensajes aparecen cuando se ejecute el código y se llamen a los *endpoints*.

Controlador REST o Controller REST: en la Figura 142 se creó la clase “DeviceController” para manejar los *endpoints* relacionados con la gestión de dispositivos. Se declara la ruta base que es */devices* y se llamará al “DeviceService” para llamar a la lógica del negocio cuando se llamen las solicitudes HTTP.

Figura 142

Clase `DeviceController`.

```
DeviceController.java x
1 package mx.t4g.geotracehub.controller;
2 > import ...
11
12 @Slf4j // Habilita el logging para esta clase 3 usages 🚩 Sunflower-S14
13 @RestController // Indica que esta clase es un controlador REST
14 @AllArgsConstructor // Genera un constructor con todos los argumentos automáticamente
15 @RequestMapping("/devices") // Define la ruta base para este controlador
16 public class DeviceController {
17
18     private final DeviceService deviceService;
```

El primer endpoint, es el de la ruta base declarado en el método “getAllDevices” que al llamar a todos los dispositivos mediante el método *GET*, intencionalmente lanzará una excepción personalizada, la cual previamente en la administración de usuarios se creó y se puede reutilizar el método para que maneje el error con un mensaje indicando que debe de especificar un usuario para obtener el dispositivo en específico, se evidencia en la Figura 143:

Figura 143

Método getAllDevices del controlador.

```
16 public class DeviceController {
17
18     private final DeviceService deviceService;
19
20     // Endpoint para obtener todos los dispositivos
21     @GetMapping no usages  Sunflower-S14 +1
22     public List<Device> getAllDevices() {
23         log.info("Calling get all devices"); // Mensaje de log para depuración
24         if(true){
25             throw new ValidateException("Error", "Debe de especificar un usuario");
26         }
27         return deviceService.getAllDevices(); // Llama al servicio para obtener la lista de dispositivos
28     }
```

El segundo endpoint, se define en la Figura 144, se declara la ruta base es *devices/{deviceId}* mediante el uso del *@GetMapping*, es declarado en el método “getDeviceById” en el que mostrará la información de un dispositivo por su ID al llamar al service lo buscará pasándolo como parámetro en la URL en el que si lo encuentra mostrará un estado HTTP 200 indicando que la petición fue realizada de manera correcta:

Figura 144

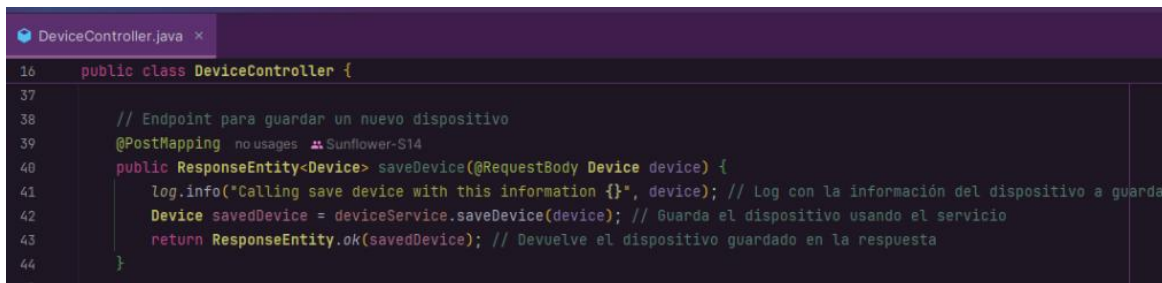
Método getDeviceById del controlador.

```
29
30     // Endpoint para obtener un dispositivo por su ID
31     @GetMapping("/{deviceId}") no usages  Sunflower-S14
32     public ResponseEntity<Device> getDeviceById(@PathVariable Integer deviceId) {
33         log.info("Calling get device {}", deviceId); // Mensaje de log con el ID solicitado
34         Device device = deviceService.getDeviceById(deviceId); // Obtiene el dispositivo del servicio
35         return ResponseEntity.ok(device); // Devuelve el dispositivo en la respuesta
36     }
```


La Figura 145 se representa en el tercer endpoint, está declarado en el método “saveDevice” la ruta será `/devices` mediante la petición `@PostMapping`, en el que si la información del dispositivo es completa llamará al servicio para que guarde el dispositivo en la base de datos retornando un estado `200 OK`.

Figura 145

Método saveDevice del controlador.

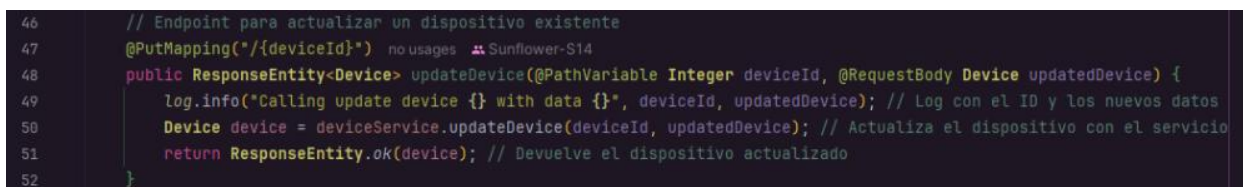
A screenshot of an IDE showing the `saveDevice` method in `DeviceController.java`. The code is as follows:

```
16 public class DeviceController {
37
38     // Endpoint para guardar un nuevo dispositivo
39     @PostMapping no usages Sunflower-S14
40     public ResponseEntity<Device> saveDevice(@RequestBody Device device) {
41         log.info("Calling save device with this information {}", device); // Log con la información del dispositivo a guardar
42         Device savedDevice = deviceService.saveDevice(device); // Guarda el dispositivo usando el servicio
43         return ResponseEntity.ok(savedDevice); // Devuelve el dispositivo guardado en la respuesta
44     }
45 }
```

El cuarto endpoint, está en el método “updateDevice” usando `@PutMapping` para su ruta completa que es `devices/deviceId`. Permite invocar al servicio para realizar la actualización de la información de un dispositivo retornando una respuesta `200 OK` cuando el dispositivo se actualice correctamente, se ilustra en la Figura 146:

Figura 146

Método updateDevice del controlador.

A screenshot of an IDE showing the `updateDevice` method in `DeviceController.java`. The code is as follows:

```
46 // Endpoint para actualizar un dispositivo existente
47 @PutMapping("/{deviceId}") no usages Sunflower-S14
48 public ResponseEntity<Device> updateDevice(@PathVariable Integer deviceId, @RequestBody Device updatedDevice) {
49     log.info("Calling update device {} with data {}", deviceId, updatedDevice); // Log con el ID y los nuevos datos
50     Device device = deviceService.updateDevice(deviceId, updatedDevice); // Actualiza el dispositivo con el servicio
51     return ResponseEntity.ok(device); // Devuelve el dispositivo actualizado
52 }
```

La Figura 147 muestra el quinto endpoint, se declara en el método “deviceStatus” donde su ruta se establece con `@PatchMapping`, en el que se muestra el endpoint `devices/deviceId/status`. Permite pasar el estado en la URL con valores booleanos (`true/false`) para activar o desactivar el dispositivo, en el que si todo sale bien muestra una respuesta `200 OK`:

Figura 147

Método deviceStatus del controlador.

```
54      @PatchMapping("/{deviceId}/status") no usages  Sunflower-S14
55      public ResponseEntity<String> deviceStatus(@PathVariable Integer deviceId, @RequestParam boolean activo) {
56          log.info("Toggling device status for ID: {}", deviceId);
57          deviceService.deviceStatus(deviceId, activo);
58          return ResponseEntity.ok( body: "Device status updated successfully");
59      }
```

El sexto endpoint, declarado en el método “deleteDevice” con ruta *devices/deviceId* declarado con *@DeleteMapping* eliminará un dispositivo por su ID, que mostrará una respuesta *204 Not Content*, se ilustra en la Figura 148:

Figura 148

Método deleteDevice del controlador.

```
// Endpoint para eliminar un dispositivo por su ID
@DeleteMapping("/{deviceId}") no usages  Sunflower-S14
public ResponseEntity<Void> deleteDevice(@PathVariable Integer deviceId) {
    log.info("Calling delete device {}", deviceId); // Log con el ID del dispositivo a eliminar
    deviceService.deleteDevice(deviceId); // Llama al servicio para eliminar el dispositivo
    return ResponseEntity.noContent().build(); // Devuelve una respuesta sin contenido indicando éxito
}
```

Por último, se visualiza en la Figura 149 el método “getDevicesByUser” con ruta final *devices/user/userId* mediante *@GetMapping* permitirá obtener todos los dispositivos que un usuario tenga registrado en el sistema, en el que se mostrará mediante una lista de dispositivos que buscará en el servicio primero el ID del usuario al que se requiera visualizar todos los dispositivos que se registraron al sistema que al consultar este endpoint retornará una respuesta *200 OK*:

Figura 149

Método getDevicesByUser del controlador.

```
@GetMapping("/user/{userId}") no usages  Sunflower-S14
public ResponseEntity<List<Device>> getDevicesByUser(@PathVariable Integer userId) {
    log.info("Solicitando dispositivos para el usuario ID: {}", userId);
    List<Device> devices = deviceService.getDevicesByUserId(userId);
    return ResponseEntity.ok(devices);
}
```

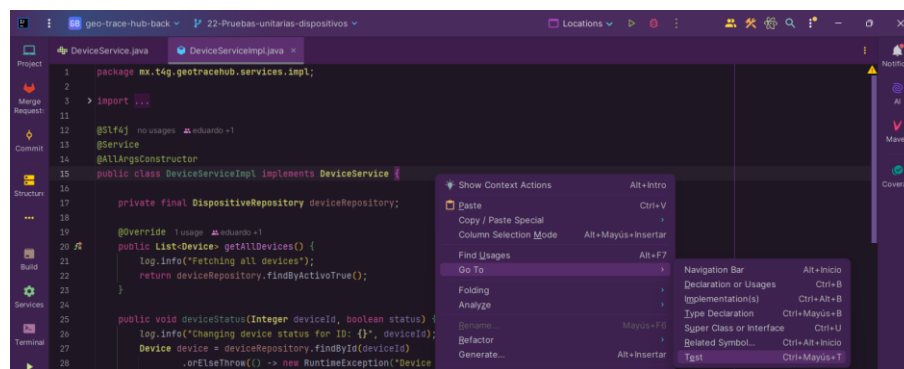
3.1.16. Pruebas unitarias en la administración de dispositivos

Metodología: usando Mockito para aislar la lógica de negocio de la clase “DevicesServiceImpl” para probar la lógica de negocio de manera efectiva que garantizará la calidad del código del servicio.

La Figura 150 muestra en la clase “devicesServicesImpl” que al dar clic derecho se despliega un menú de opciones, posteriormente se da clic en **Go To** y se visualiza la opción de “Test”:

Figura 150

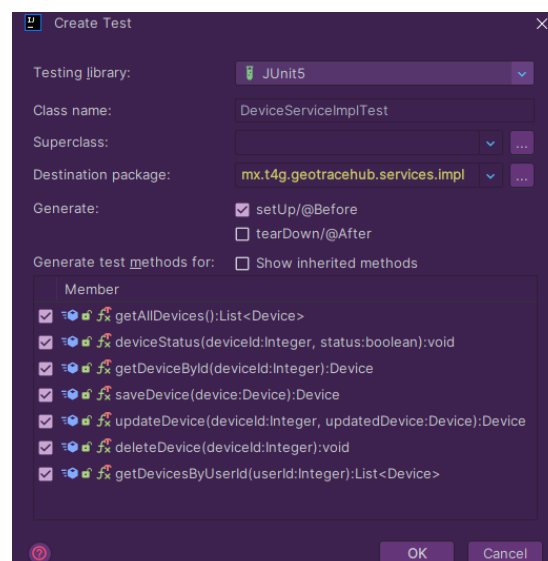
Clase devicesServicesImpl para crear la prueba unitaria.



En la Figura 151 se incluye la anotación **@Before** y se eligen todos los métodos de la clase:

Figura 151

Selección de los métodos de la clase DeviceServiceImpl.



En el archivo del test, se declara el mock el cual será el repository y la inyección del mock que es la clase DeviceServiceImpl para la simulación de datos de dispositivos que es inicializada con el `@BeforeEach`. Se añaden los datos para la prueba en donde se crea un usuario y a este usuario se le agrega un dispositivo, se observa en la Figura 152:

Figura 152

Declaración de los atributos en la clase DeviceServiceImplTest.

```
18
19 ▶▶ class DeviceServiceImplTest { new *
20
21     @Mock 13 usages
22     private DispositivoRepository deviceRepository;
23
24     @InjectMocks 5 usages
25     private DeviceServiceImpl deviceService;
26
27     private Device device; 9 usages
28
29     @BeforeEach new *
30     void setUp() {
31         User user = new User();
32         user.setId(1);
33         user.setName("Sandi");
34
35         device = Device.builder()
36             .id(1)
37             .name("Device1")
38             .model("ModelX")
39             .activo(true)
40             .user(user)
41             .build();
42     }
```

En la Figura 153 se inicializa el método "getAllDevices" que obtiene todos los dispositivos:

Figura 153

Método getAllDevices del test unitario.

```
19 class DeviceServiceImplTest { new *
43
44     @Test new *
45 ▶ void getAllDevices() {
46     when(deviceRepository.findByActivoTrue()).thenReturn(Arrays.asList(device));
47
48     List<Device> devices = deviceService.getAllDevices();
49
50     assertNotNull(devices);
51     assertEquals( expected: 1, devices.size());
52     verify(deviceRepository, times( wantedNumberOfInvocations: 1)).findByActivoTrue();
53 }
```

La Figura 154 presenta la ejecución de la prueba que imprime el mensaje “Fetching all devices” de los logs que se declararon en cada método de la clase deviceserviceimpl:

Figura 154

Ejecución del test del método getAllDevices.



```
DeviceServiceImplTest.java x
20 class DeviceServiceImplTest {
46     @Test
47     void getAllDevices() {
48         when(deviceRepository.findById(1)).thenReturn(Arrays.asList(device));
49
50         List<Device> devices = deviceService.getAllDevices();
51
52         assertNotNull(devices);
53         assertEquals("expected: 1 devices size()", devices.size());
54     }
55 }

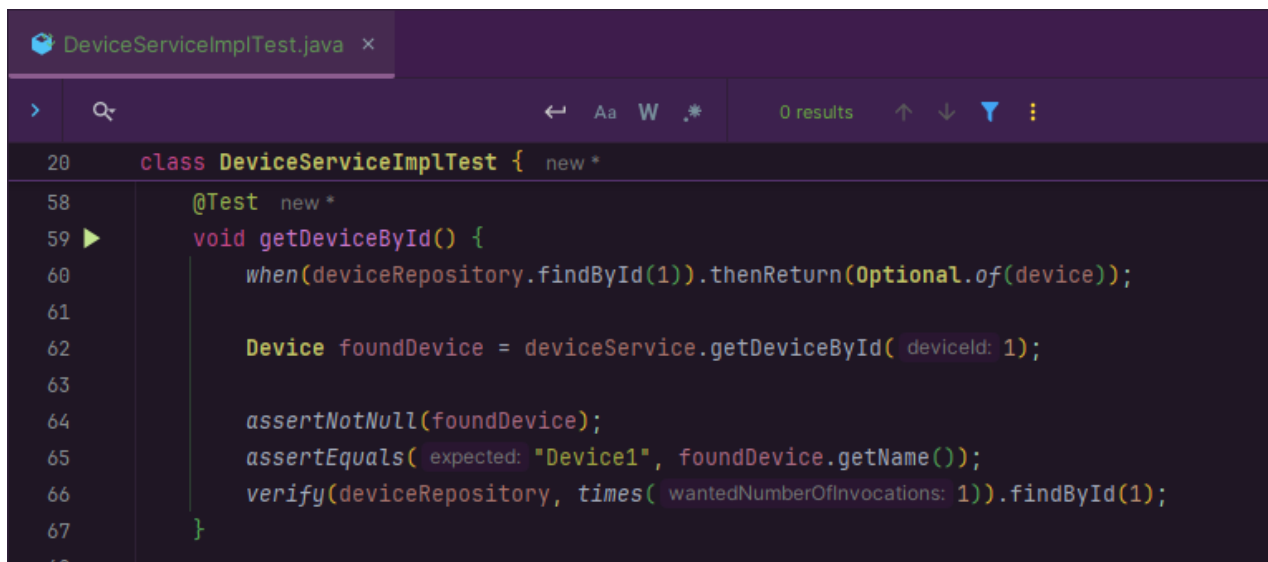
Run DeviceServiceImplTest.getAllDevices x
Tests passed: 1 of 1 test - 1sec 444ms
getAllDevices() 1sec 444ms
"C:\Program Files\Java\jdk-21\bin\java.exe" ...
Java HotSpot(TM) 64-Bit Server VM warning: Sharing is only supported for boot loader classes because bootstrap classpath
13:31:42.524 [main] INFO mx.t4g.geotracehub.services.impl.DeviceServiceImpl -- Fetching all devices

Process finished with exit code 0
```

En la Figura 155 se crea la prueba para “getDeviceById()” que buscará un dispositivo por su ID:

Figura 155

Método getDeviceById del test unitario.



```
DeviceServiceImplTest.java x
20 class DeviceServiceImplTest {
58     @Test
59     void getDeviceById() {
60         when(deviceRepository.findById(1)).thenReturn(Optional.of(device));
61
62         Device foundDevice = deviceService.getDeviceById(1);
63
64         assertNotNull(foundDevice);
65         assertEquals("Device1", foundDevice.getName());
66         verify(deviceRepository, times(1)).findById(1);
67     }
68 }
```

Al ejecutar la prueba unitaria, lanza el mensaje “Fetching device with ID: 1” el cual indica que la prueba se ejecutó correctamente mostrada en la Figura 156:

Figura 156

Ejecución del test del método `getDeviceById`.



```
DeviceServiceImplTest.java x
20 class DeviceServiceImplTest {
58   @Test
59   void getDeviceById() {
60       when(deviceRepository.findById(1)).thenReturn(Optional.of(device));
61
62       Device foundDevice = deviceService.getDeviceById(1);
63
64       assertNotNull(foundDevice);
65       assertEquals("Device1", foundDevice.getName());
66   }
67 }

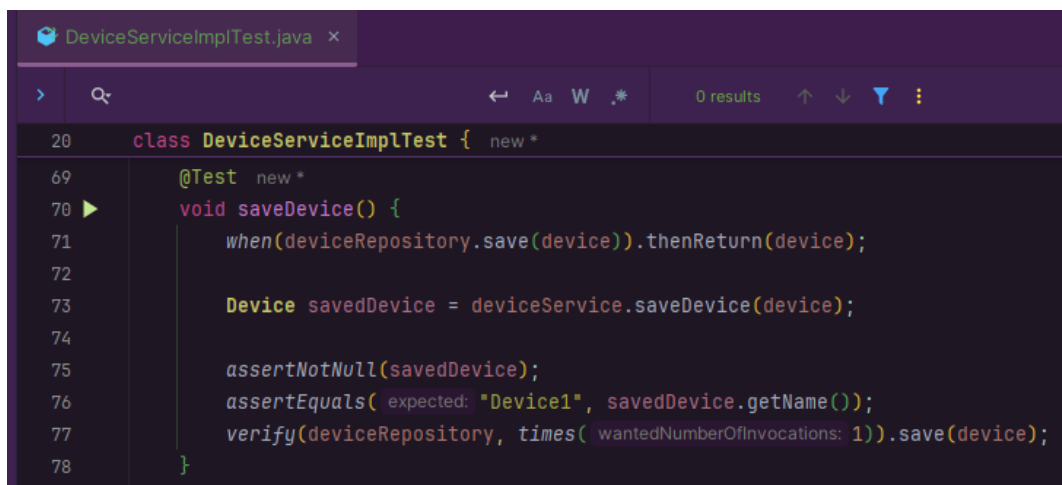
Run DeviceServiceImplTest.getDeviceById
DeviceServiceImpl 1sec 326ms Tests passed: 1 of 1 test - 1sec 326ms
getDeviceById() 1sec 326ms
"C:\Program Files\Java\jdk-21\bin\java.exe" ...
Java HotSpot(TM) 64-Bit Server VM warning: Sharing is only supported for boot loader classes because bootstrap classpath is not well formed
13:32:18.346 [main] INFO mx.t4g.geotracehub.services.impl.DeviceServiceImpl -- Fetching device with ID: 1

Process finished with exit code 0
```

La Figura 157 representa al método “`saveDevice()`” que simula cuando se guarda un nuevo dispositivo en el sistema:

Figura 157

Método `saveDevice` del test unitario.

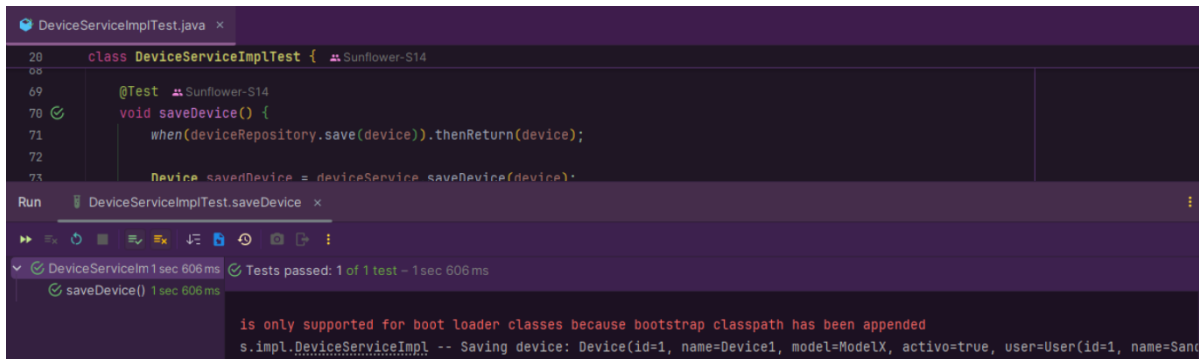


```
DeviceServiceImplTest.java x
20 class DeviceServiceImplTest {
69   @Test
70   void saveDevice() {
71       when(deviceRepository.save(device)).thenReturn(device);
72
73       Device savedDevice = deviceService.saveDevice(device);
74
75       assertNotNull(savedDevice);
76       assertEquals("Device1", savedDevice.getName());
77       verify(deviceRepository, times(1)).save(device);
78   }
79 }
```

En el que al ejecutar la prueba se encuentra en la Figura 158 imprime: “Saving device: Device(id=1, name=Device1, model=ModelX, activo=true, user=User(id=1, name=Sandi, username=null, email=null, password=null, activo=true, devices=null), positions=null)”:

Figura 158

Ejecución del test del método saveDevice.

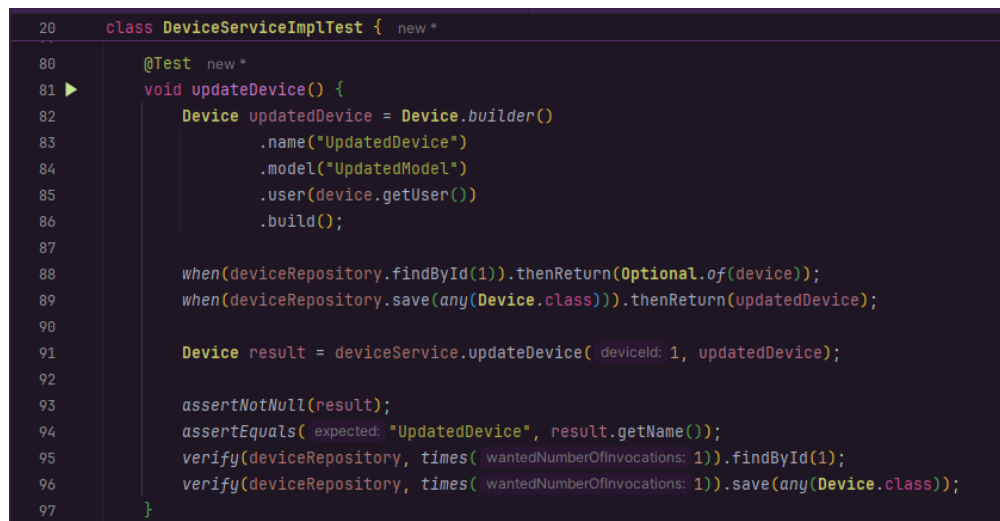


```
20 class DeviceServiceImplTest {
69 @Test
70 void saveDevice() {
71     when(deviceRepository.save(device)).thenReturn(device);
72
73     Device savedDevice = deviceService.saveDevice(device);
74 }
75 }
76
77 Run DeviceServiceImplTest.saveDevice
78
79 DeviceServiceImpl 1 sec 606 ms Tests passed: 1 of 1 test - 1 sec 606 ms
80
81 saveDevice() 1 sec 606 ms
82
83 is only supported for boot loader classes because bootstrap classpath has been appended
84 s.impl.DeviceServiceImpl -- Saving device: Device(id=1, name=Device1, model=ModelX, activo=true, user=User(id=1, name=San...
```

La Figura 159 presenta el test para el método “updateDevice()” que simula la actualización de datos del dispositivo. Primero busca el dispositivo por su ID, al encontrarlo podrá actualizar el dispositivo con los datos nuevos:

Figura 159

Método updatedDevice del test unitario.



```
20 class DeviceServiceImplTest {
80 @Test
81 void updateDevice() {
82     Device updatedDevice = Device.builder()
83         .name("UpdatedDevice")
84         .model("UpdatedModel")
85         .user(device.getUser())
86         .build();
87
88     when(deviceRepository.findById(1)).thenReturn(Optional.of(device));
89     when(deviceRepository.save(any(Device.class))).thenReturn(updatedDevice);
90
91     Device result = deviceService.updateDevice(1, updatedDevice);
92
93     assertNotNull(result);
94     assertEquals("UpdatedDevice", result.getName());
95     verify(deviceRepository, times(1)).findById(1);
96     verify(deviceRepository, times(1)).save(any(Device.class));
97 }
```

Al ejecutar la prueba se imprime el mensaje “Updating device with ID: 1”, imprimiendo el mensaje del log en el que indica un mensaje de que la actualización se aplicó al dispositivo con ID 1, este comportamiento se observa en la Figura 160:

Figura 160

Ejecución del test del método updateDevice.



```
20 class DeviceServiceImpTest {
81     void updateDevice() {
82         Device updatedDevice = Device.builder()
83             .name("UpdatedDevice")
84             .model("UpdatedModel")
85             .user(device.getUser())
86             .build();
87     }
}
```

Run DeviceServiceImpTest.updateDevice

DeviceServiceImpTest: 1 sec 331 ms Tests passed: 1 of 1 test - 1 sec 331 ms

updateDevice: 1 sec 331 ms

"C:\Program Files\Java\jdk-21\bin\java.exe" ...
Java HotSpot(TM) 64-Bit Server VM warning: Sharing is only supported for boot loader classes because bootstrap classpath
13:36:42.946 [main] INFO mx.t4g.geotracehub.services.impl.DeviceServiceImpl -- Updating device with ID: 1
Process finished with exit code 0

La Figura 161 representa el método “deleteDevice()” que permite simular la eliminación del dispositivo por su ID en el que primero buscará si el dispositivo existe y al ser verdad procede a eliminar el dispositivo:

Figura 161

Método deleteDevice del test unitario.



```
20 class DeviceServiceImpTest {
98
99     @Test
100     void deleteDevice() {
101         when(deviceRepository.existsById(1)).thenReturn(true);
102         doNothing().when(deviceRepository).deleteById(1);
103
104         assertDoesNotThrow(() -> deviceService.deleteDevice(1));
105         verify(deviceRepository, times(1)).deleteById(1);
106     }
}
```

La Figura 162 representa la ejecución de la prueba unitaria. Se imprime el mensaje “Deleting device with ID: 1” en el que indica que se eliminó el dispositivo con ID 1:

Figura 162

Ejecución del test del método deleteDevice.



```
DeviceServiceImplTest.java x
20 class DeviceServiceImplTest {
99 @Test
100 void deleteDevice() {
101     when(deviceRepository.existsById(1)).thenReturn(true);
102     doNothing().when(deviceRepository).deleteById(1);
103
104     assertDoesNotThrow(() -> deviceService.deleteDevice(1));
105     verify(deviceRepository, times(1)).deleteById(1);
106 }

Run DeviceServiceImplTest.deleteDevice x
DeviceServiceImplTest.deleteDevice 1 sec 665 ms Tests passed: 1 of 1 test - 1 sec 665 ms
deleteDevice 1 sec 665 ms
"C:\Program Files\Java\jdk-21\bin\java.exe" ...
Java HotSpot(TM) 64-Bit Server VM warning: Sharing is only supported for boot loader classes because bootstrap classpath is not well formed
13:37:35.089 [main] INFO mx.t4g.geotracehub.services.impl.DeviceServiceImpl -- Deleting device with ID: 1
Process finished with exit code 0
```

Finalmente, en el método “getDevicesByUserId()” permite verificar que funciona el servicio al llamarlo, en donde se ingresa un ID de un usuario y la lista de dispositivos ya asignada para que se pueda realizar la llamada adecuada al servicio; se encuentra en la Figura 163:

Figura 163

Método getDevicesByUserId del test unitario.

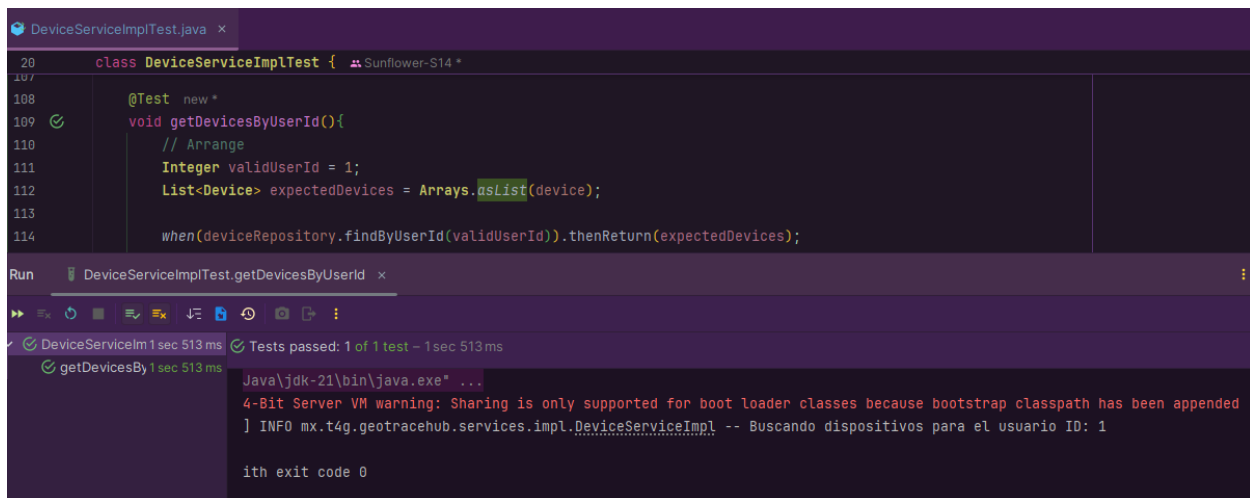


```
108 @Test new *
109 void getDevicesByUserId(){
110     // Arrange
111     Integer validUserId = 1;
112     List<Device> expectedDevices = Arrays.asList(device);
113
114     when(deviceRepository.findById(validUserId)).thenReturn(expectedDevices);
115
116     // Act
117     List<Device> result = deviceService.getDevicesByUserId(validUserId);
118
119     // Assert
120     assertNotNull(result);
121     assertEquals(expectedDevices.size(), result.size());
122     assertEquals(expectedDevices.get(0).getName(), result.get(0).getName());
123     verify(deviceRepository, times(1)).findById(validUserId);
124 }
125 }
```


La Figura 164 ilustra la ejecución del test. Primero se marcará de color verde, indicando que el test pasó de manera correcta y en la terminal se visualiza que se imprimieron los mensajes de los logs:

Figura 164

Ejecución del test del método `getDevicesByUserId`.



```
class DeviceServiceImplTest {
    @Test new *
    void getDevicesByUserId(){
        // Arrange
        Integer validUserId = 1;
        List<Device> expectedDevices = Arrays.asList(device);

        when(deviceRepository.findById(validUserId)).thenReturn(expectedDevices);
    }
}
```

Run DeviceServiceImplTest.getDevicesByUserId

DeviceServiceImpl 1 sec 513 ms Tests passed: 1 of 1 test - 1 sec 513 ms

getDevicesBy 1 sec 513 ms

Java\jdk-21\bin\java.exe" ...
4-Bit Server VM warning: Sharing is only supported for boot loader classes because bootstrap classpath has been appended
] INFO mx.t4g.geotracehub.services.impl.DeviceServiceImpl -- Buscando dispositivos para el usuario ID: 1

ith exit code 0

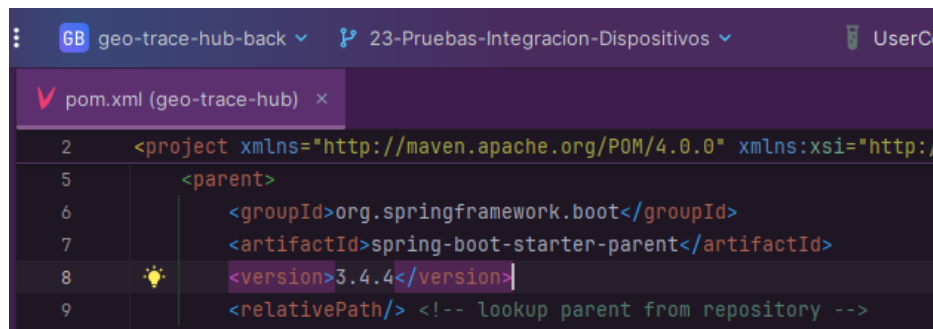
3.1.17. Pruebas de integración de la administración de dispositivos

Metodología: configurar un entorno de prueba con Spring Boot Test, Mockito y MockMvc, usando el mock del DeviceService, configurando MockMvc para DeviceController. Con el servicio configurado se simulan solicitudes HTTP que verifican las respuestas de manera aislada y controlada, lo que facilita la detección de errores y problemas de integración en una etapa temprana del desarrollo.

La Figura 165 muestra el archivo pom.xml. En este se actualiza el *starter parent* a la versión 3.4.4 que permite tener a mockito por default con una versión más actualizada de Spring Boot:

Figura 165

Actualización de la dependencia starter-parent

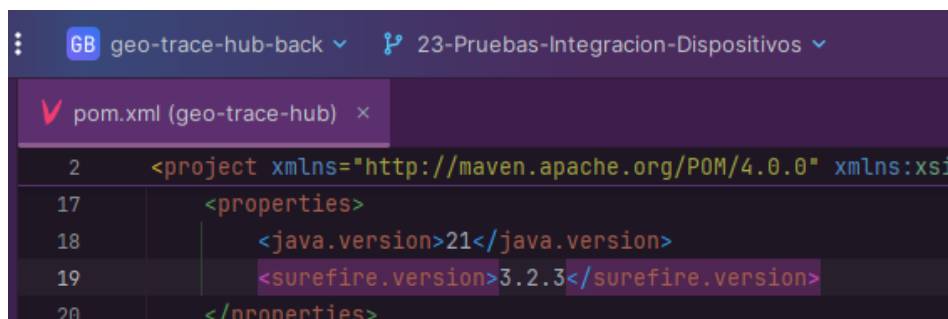


```
2 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
5 <parent>
6 <groupId>org.springframework.boot</groupId>
7 <artifactId>spring-boot-starter-parent</artifactId>
8 <version>3.4.4</version>
9 <relativePath/> <!-- lookup parent from repository -->
```

La Figura 166 contiene el uso de la versión de java surefire.version en 3.2.3:

Figura 166

Adición del surefire.version en el properties.

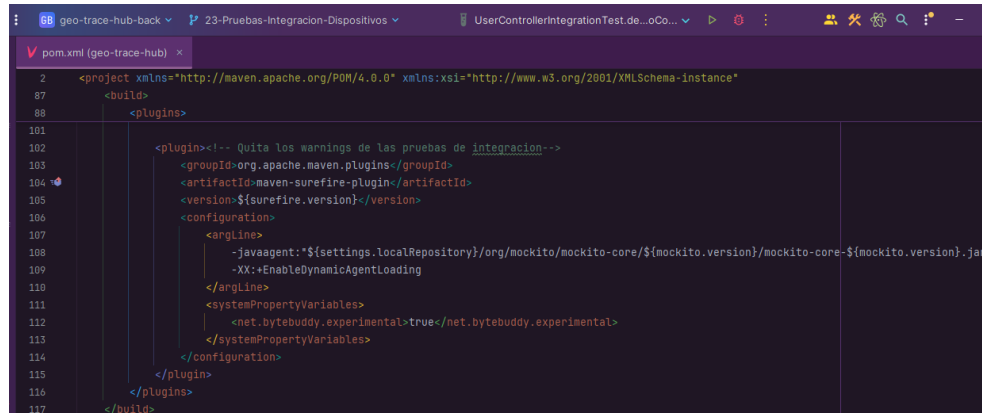


```
2 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="
17 <properties>
18 <java.version>21</java.version>
19 <surefire.version>3.2.3</surefire.version>
20 </properties>
```

Se agregó en la sección de plugins la configuración para eliminar los warnings en la ejecución de las pruebas, como se muestra en la Figura 167:

Figura 167

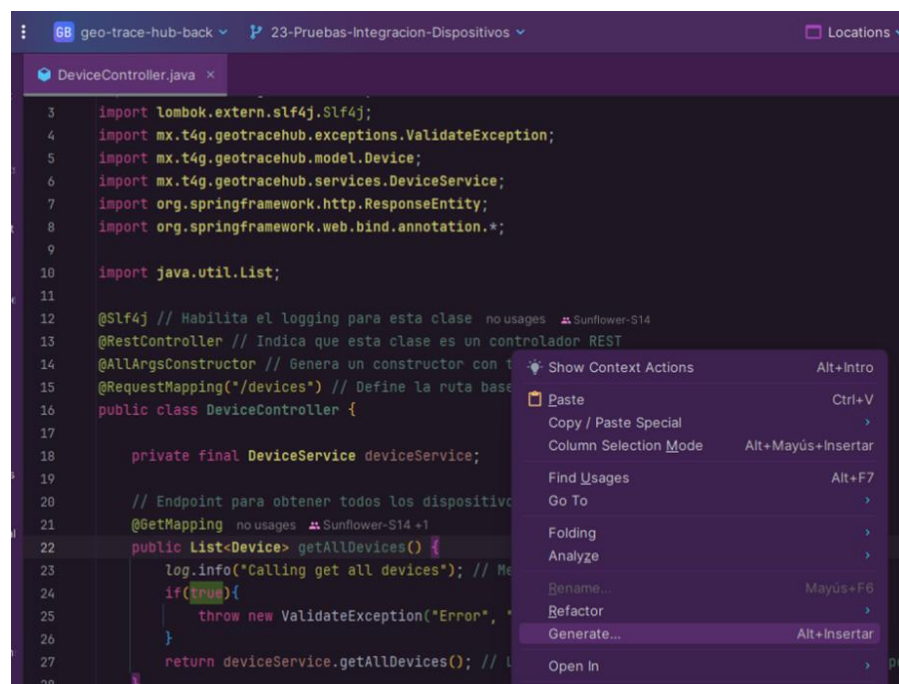
Plugin Maven-surefire-plugin para eliminar los warnings.



Para las pruebas de integración se usa la clase `DeviceController`. Con clic derecho se despliega una ventana con diferentes opciones, y se da clic en “generate”, ilustrado en la Figura 168:

Figura 168

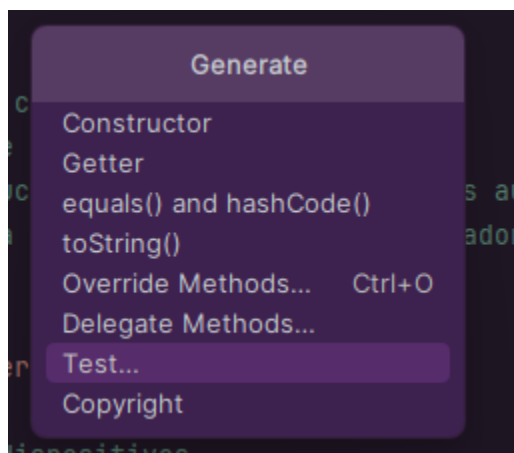
Uso de DeviceController para las pruebas de integración.



Se muestra en la Figura 169 las diferentes opciones en la que se elige la opción de “Test”:

Figura 169

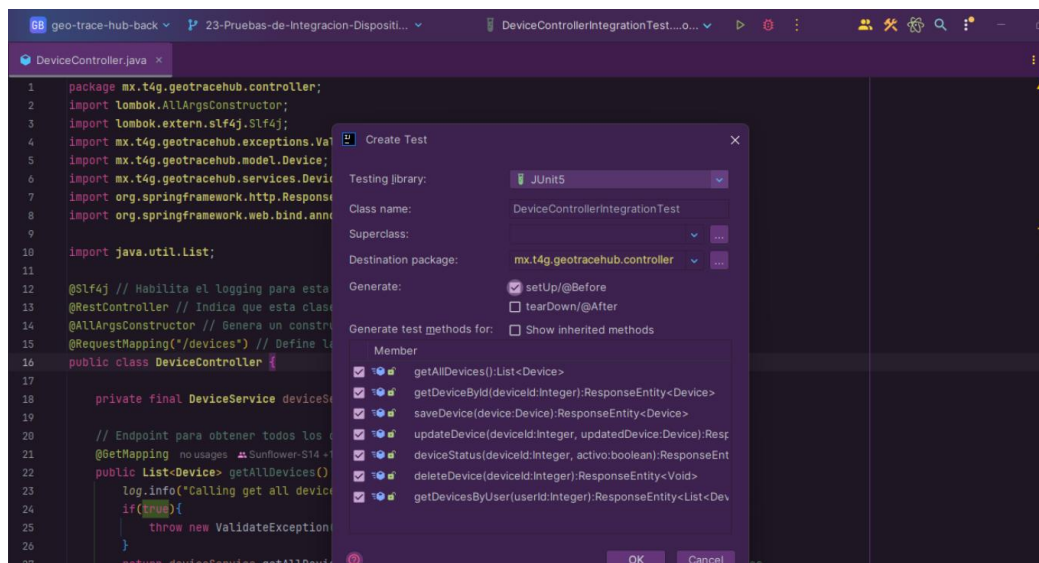
Menú de opciones “Generate”.



Se declara el nombre de la clase de Test como “DeviceControllerIntegrationTest”, se marca el `@Before` para tener el método inicializado dentro de la clase de test y se selecciona todos los métodos del controlador; se representa en la Figura 170:

Figura 170

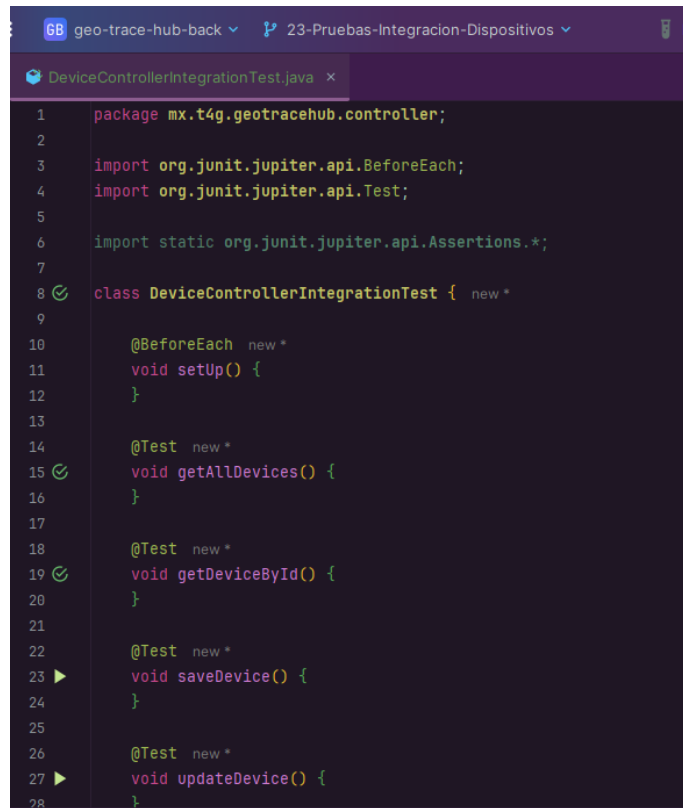
Selección de todos los métodos del controlador de dispositivos para el test de integración.



Al darle en “Ok” se crea la clase de test, con los métodos vacíos, puede verse en la Figura 171:

Figura 171

Clase DeviceControllerIntegrationTest para test de integración de dispositivos.



```
1 package mx.t4g.geotracehub.controller;
2
3 import org.junit.jupiter.api.BeforeEach;
4 import org.junit.jupiter.api.Test;
5
6 import static org.junit.jupiter.api.Assertions.*;
7
8 class DeviceControllerIntegrationTest { new *
9
10     @BeforeEach new *
11     void setUp() {
12     }
13
14     @Test new *
15     void getAllDevices() {
16     }
17
18     @Test new *
19     void getDeviceById() {
20     }
21
22     @Test new *
23     void saveDevice() {
24     }
25
26     @Test new *
27     void updateDevice() {
28     }
```

A la clase se importan las siguientes anotaciones:

- `@ExtendWith(SpringExtension.class)` que indica que se utilizará el framework de pruebas de Spring.
- `@WebMvcTest(DeviceController.class)`: se declara el controlador `DeviceController` que permite realizar las pruebas de integración al controlador.
- `@AutoConfigureMockMvc`: configura automáticamente el objeto `MockMvc` para realizar pruebas de integración.

Se visualiza en la Figura 172 el uso de las anotaciones en la clase:

Figura 172

Uso de las anotaciones `@ExtendWith`, `@WebMvcTest` y `@AutoConfigureMockMvc` en la clase `DeviceControllerIntegrationTest`.



```
1 package mx.t4g.geotracehub.controller;
2
3 > import ...
22
23 @ExtendWith(SpringExtension.class) new *
24 @WebMvcTest(DeviceController.class)
25 @AutoConfigureMockMvc
26 public class DeviceControllerIntegrationTest {
27
```

En las variables de configuración se usa `MockMvc` que permite realizar solicitudes HTTP a la aplicación, un `objectMapper` que convertirá objetos a JSON y viceversa. Se llama a “deviceService” que se mockea para simular el comportamiento del servicio de los dispositivos puede observarse en la Figura 173:

Figura 173

Declaración de las variables con `MockMvc`.



```
28 @Autowired 7 usages
29 private MockMvc mockMvc;
30
31 @Autowired 2 usages
32 private ObjectMapper objectMapper;
33
34 private DeviceService deviceService; 7 usages
35
```

En la Figura 174 se inicializa el `setUp` con el `@BeforeEach` se mockea el servicio de dispositivos en el que configura el controlador de dispositivos con el mock y se configura el objeto `MockMvc`

para realizar solicitudes HTTP al controlador juntamente se incluye los controladores de excepciones globales para manejar las excepciones que existan en las peticiones:

Figura 174

Configuración del método setUp en la prueba de integración de dispositivos.

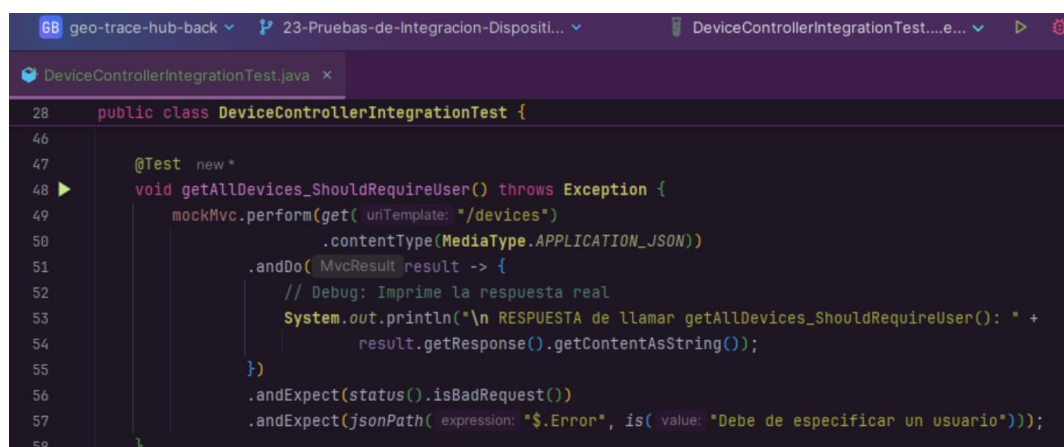


```
28 public class DeviceControllerIntegrationTest {
37
38     @BeforeEach new *
39     void setUp() {
40         deviceService = Mockito.mock(DeviceService.class);
41         DeviceController controller = new DeviceController(deviceService);
42         mockMvc = MockMvcBuilders.standaloneSetup(controller)
43             .setControllerAdvice(new ValidationExceptionHandler())
44             .build();
45     }
```

La Figura 175 muestra el primer método, “getAllDevices_ShouldRequireUser()” permite que al llamar a la petición /devices lanzará una respuesta *BadRequest* en el que le indicará al usuario que no podrá mostrar todos los dispositivos si es que no se especifica al usuario que le corresponde:

Figura 175

Método getAllDevices_ShouldRequireUser del test de integración.

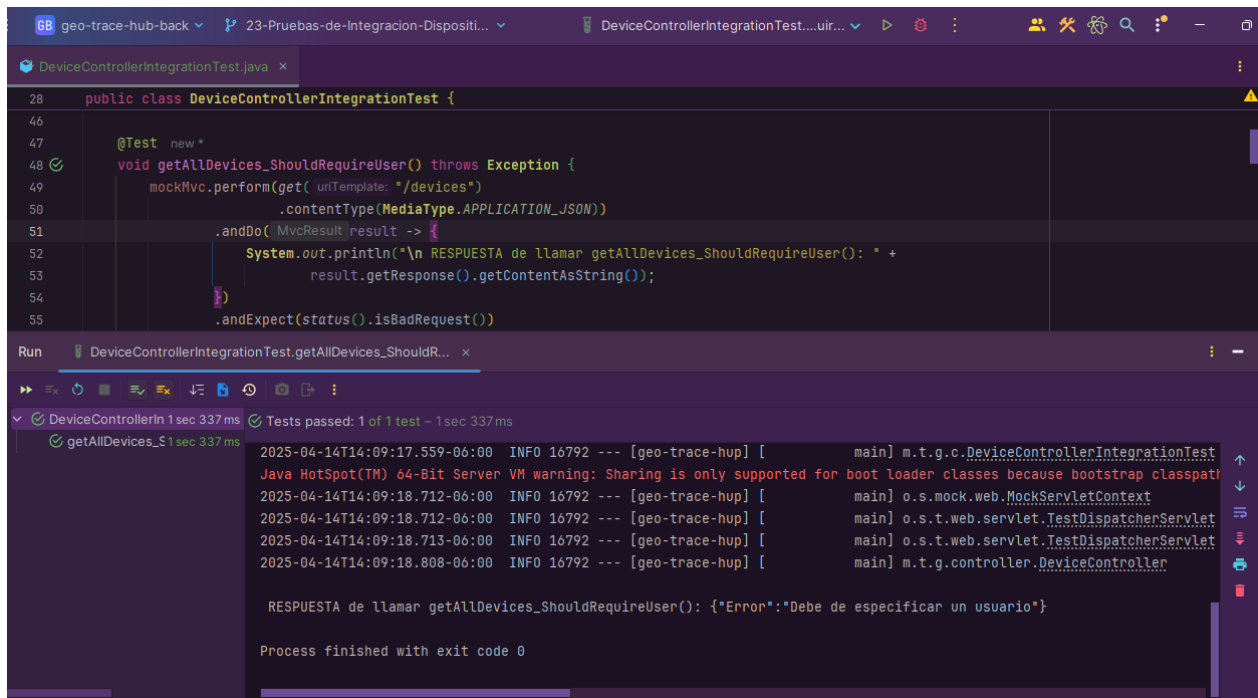


```
28 public class DeviceControllerIntegrationTest {
46
47     @Test new *
48     void getAllDevices_ShouldRequireUser() throws Exception {
49         mockMvc.perform(get(uriTemplate: "/devices")
50             .contentType(MediaType.APPLICATION_JSON))
51             .andExpect(status().isBadRequest());
52             // Debug: Imprime la respuesta real
53             System.out.println("\n RESPUESTA de llamar getAllDevices_ShouldRequireUser(): " +
54                 result.getResponse().getContentAsString());
55         })
56         .andExpect(status().isBadRequest())
57         .andExpect(jsonPath(expression: "$.Error", is(value: "Debe de especificar un usuario")));
58     }
```

El resultado de esta prueba en el que en la terminal se imprime el mensaje: “RESPUESTA de llamar getAllDevices_ShouldRequireUser(): {"Error":"Debe de especificar un usuario"}” y el método como se ilustra en la Figura 176 se marca con una palomita de color verde indicando que la prueba pasó correctamente:

Figura 176

Resultado del método getAllDevices_ShouldRequireUser.



```
28 public class DeviceControllerIntegrationTest {
46
47     @Test new *
48     void getAllDevices_ShouldRequireUser() throws Exception {
49         mockMvc.perform(get(uriTemplate: "/devices")
50             .contentType(MediaType.APPLICATION_JSON))
51             .andExpect(result -> {
52                 System.out.println("\n RESPUESTA de llamar getAllDevices_ShouldRequireUser(): " +
53                     result.getResponse().getContentAsString());
54             })
55             .andExpect(status().isBadRequest())

```

Run DeviceControllerIntegrationTest.getAllDevices_ShouldRequireUser() x

Tests passed: 1 of 1 test - 1 sec 337 ms

getAllDevices_ShouldRequireUser() 1 sec 337 ms

```
2025-04-14T14:09:17.559-06:00 INFO 16792 --- [geo-trace-hup] [main] m.t.g.c.DeviceControllerIntegrationTest
Java HotSpot(TM) 64-Bit Server VM warning: Sharing is only supported for boot loader classes because bootstrap classpath
2025-04-14T14:09:18.712-06:00 INFO 16792 --- [geo-trace-hup] [main] o.s.mock.web.MockServletContext
2025-04-14T14:09:18.712-06:00 INFO 16792 --- [geo-trace-hup] [main] o.s.t.web.servlet.TestDispatcherServlet
2025-04-14T14:09:18.713-06:00 INFO 16792 --- [geo-trace-hup] [main] o.s.t.web.servlet.TestDispatcherServlet
2025-04-14T14:09:18.808-06:00 INFO 16792 --- [geo-trace-hup] [main] m.t.g.controller.DeviceController

RESPUESTA de llamar getAllDevices_ShouldRequireUser(): {"Error":"Debe de especificar un usuario"}

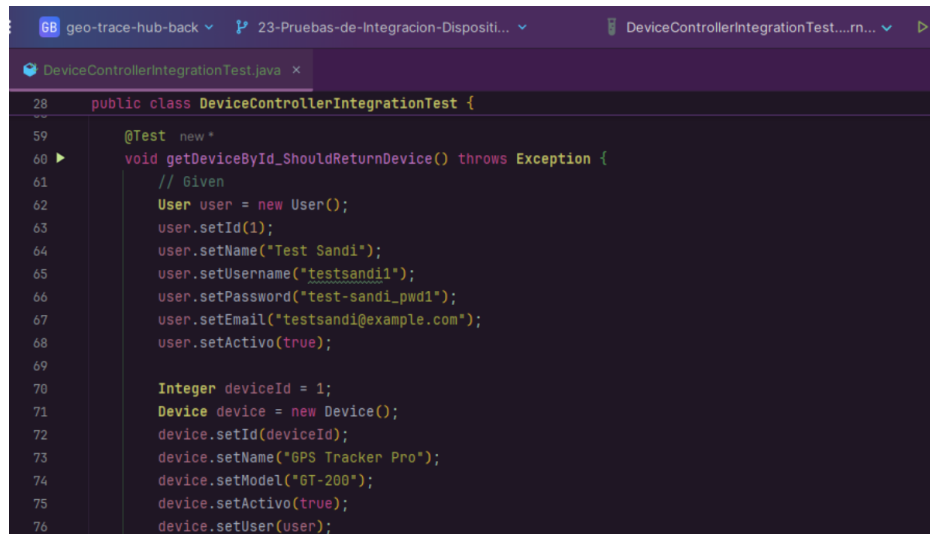
Process finished with exit code 0

```

La Figura 177 detalla el segundo método “getDeviceById_ShouldReturnDevice()” que permitirá obtener un dispositivo por ID, para esto se crea un objeto “User” con atributos del modelo, luego se define el “deviceId” que se va a buscar en la solicitud. Se crea un objeto “Device” con atributos como ID, nombre, modelo y estado activo que es asociado con el usuario creado anteriormente:

Figura 177

Método `getDeviceById_ShouldReturnDevice` del test de integración.

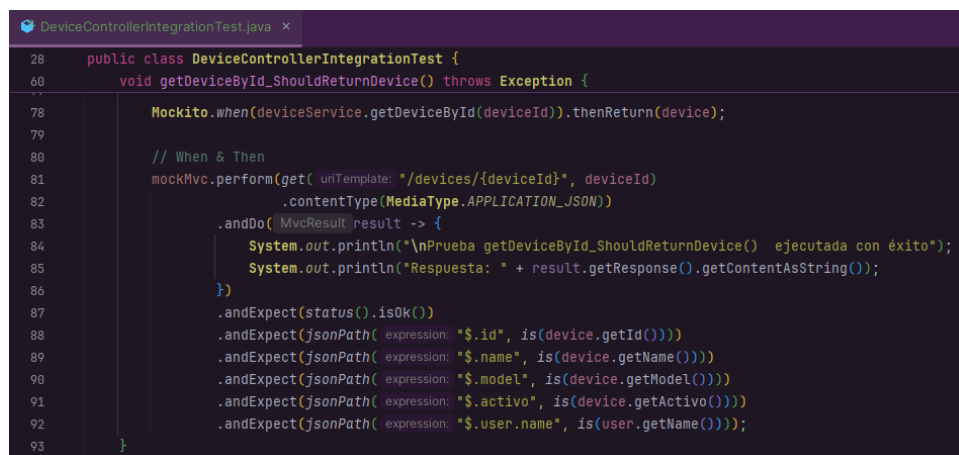


```
28 public class DeviceControllerIntegrationTest {
59     @Test new *
60     void getDeviceById_ShouldReturnDevice() throws Exception {
61         // Given
62         User user = new User();
63         user.setId(1);
64         user.setName("Test Sandi");
65         user.setUsername("testsandi1");
66         user.setPassword("test-sandi_pwd1");
67         user.setEmail("testsandi@example.com");
68         user.setActivo(true);
69
70         Integer deviceId = 1;
71         Device device = new Device();
72         device.setId(deviceId);
73         device.setName("GPS Tracker Pro");
74         device.setModel("GT-200");
75         device.setActivo(true);
76         device.setUser(user);
77     }
78 }
```

Después se utiliza Mockito, para simular el comportamiento del servicio “deviceService” cuando se llama al método “getDeviceById” con deviceId, en el que se configura para que devuelva el objeto device creado anteriormente. Posteriormente con *mockMvc* mediante la solicitud *GET* al endpoint `/devices/{deviceId}` busca el dispositivo y si lo encuentra imprime mensajes de éxito en la consola recuperando los datos completos, se representa en la Figura 178:

Figura 178

Configuración de la solicitud en el método `getDeviceById_ShouldReturnDevice`.



```
28 public class DeviceControllerIntegrationTest {
60     void getDeviceById_ShouldReturnDevice() throws Exception {
61         // Given
62         User user = new User();
63         user.setId(1);
64         user.setName("Test Sandi");
65         user.setUsername("testsandi1");
66         user.setPassword("test-sandi_pwd1");
67         user.setEmail("testsandi@example.com");
68         user.setActivo(true);
69
70         Integer deviceId = 1;
71         Device device = new Device();
72         device.setId(deviceId);
73         device.setName("GPS Tracker Pro");
74         device.setModel("GT-200");
75         device.setActivo(true);
76         device.setUser(user);
77
78         Mockito.when(deviceService.getDeviceById(deviceId)).thenReturn(device);
79
80         // When & Then
81         mockMvc.perform(get("/devices/{deviceId}", deviceId)
82             .contentType(MediaType.APPLICATION_JSON))
83             .andExpect(status().isOk())
84             .andExpect(jsonPath("$.id", is(device.getId())))
85             .andExpect(jsonPath("$.name", is(device.getName())))
86             .andExpect(jsonPath("$.model", is(device.getModel())))
87             .andExpect(jsonPath("$.activo", is(device.getActivo())))
88             .andExpect(jsonPath("$.user.name", is(user.getName())));
89     }
90 }
91
92
93 }
```

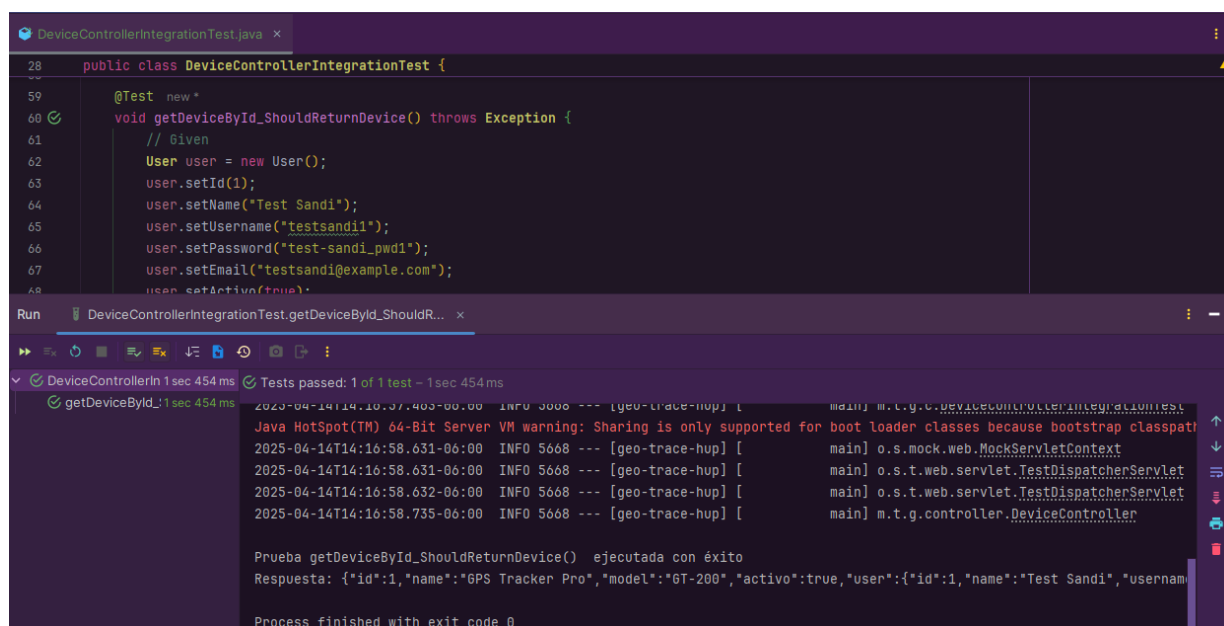
El resultado del test se muestra en la Figura 179 que marcó de color verde indicando que ejecutó adecuadamente y en la termina se imprimió lo siguiente:

“Prueba getDeviceById_ShouldReturnDevice() ejecutada con éxito”

“Respuesta: {\"id\":1,\"name\":\"GPS Tracker Pro\", \"model\":\"GT-200\", \"activo\":true, \"user\":{\"id\":1, \"name\":\"TestSandi\", \"username\":\"testsandi1\", \"email\":\"testsandi@example.com\", \"password\":\"test-sandi_pwd1\", \"activo\":true}}”:

Figura 179

Ejecución del test del método getDeviceById_ShouldReturnDevice.



The screenshot shows an IDE with two panels. The top panel displays the source code of `DeviceControllerIntegrationTest.java`. The bottom panel shows the output of the test execution, which is successful.

```
28 public class DeviceControllerIntegrationTest {
59
60     @Test new *
61     void getDeviceById_ShouldReturnDevice() throws Exception {
62         // Given
63         User user = new User();
64         user.setId(1);
65         user.setName("Test Sandi");
66         user.setUsername("testsandi1");
67         user.setPassword("test-sandi_pwd1");
68         user.setEmail("testsandi@example.com");
69         user.setActivo(true);
70     }
71 }
```

Run: DeviceControllerIntegrationTest.getDeviceById_ShouldReturnDevice() x

DeviceControllerIn 1 sec 454 ms Tests passed: 1 of 1 test - 1 sec 454 ms

getDeviceById_1 sec 454 ms

2025-04-14T14:16:58.631-06:00 INFO 5668 --- [geo-trace-hup] [main] m.t.g.c.DeviceControllerIntegrationTest: Prueba getDeviceById_ShouldReturnDevice() ejecutada con éxito

2025-04-14T14:16:58.631-06:00 INFO 5668 --- [geo-trace-hup] [main] o.s.mock.web.MockServletContext: Respuesta: {\"id\":1,\"name\":\"GPS Tracker Pro\", \"model\":\"GT-200\", \"activo\":true, \"user\":{\"id\":1, \"name\":\"Test Sandi\", \"username\":\"testsandi1\", \"email\":\"testsandi@example.com\", \"password\":\"test-sandi_pwd1\", \"activo\":true}}

2025-04-14T14:16:58.632-06:00 INFO 5668 --- [geo-trace-hup] [main] o.s.t.web.servlet.TestDispatcherServlet: Process finished with exit code 0

2025-04-14T14:16:58.735-06:00 INFO 5668 --- [geo-trace-hup] [main] m.t.g.controller.DeviceController:

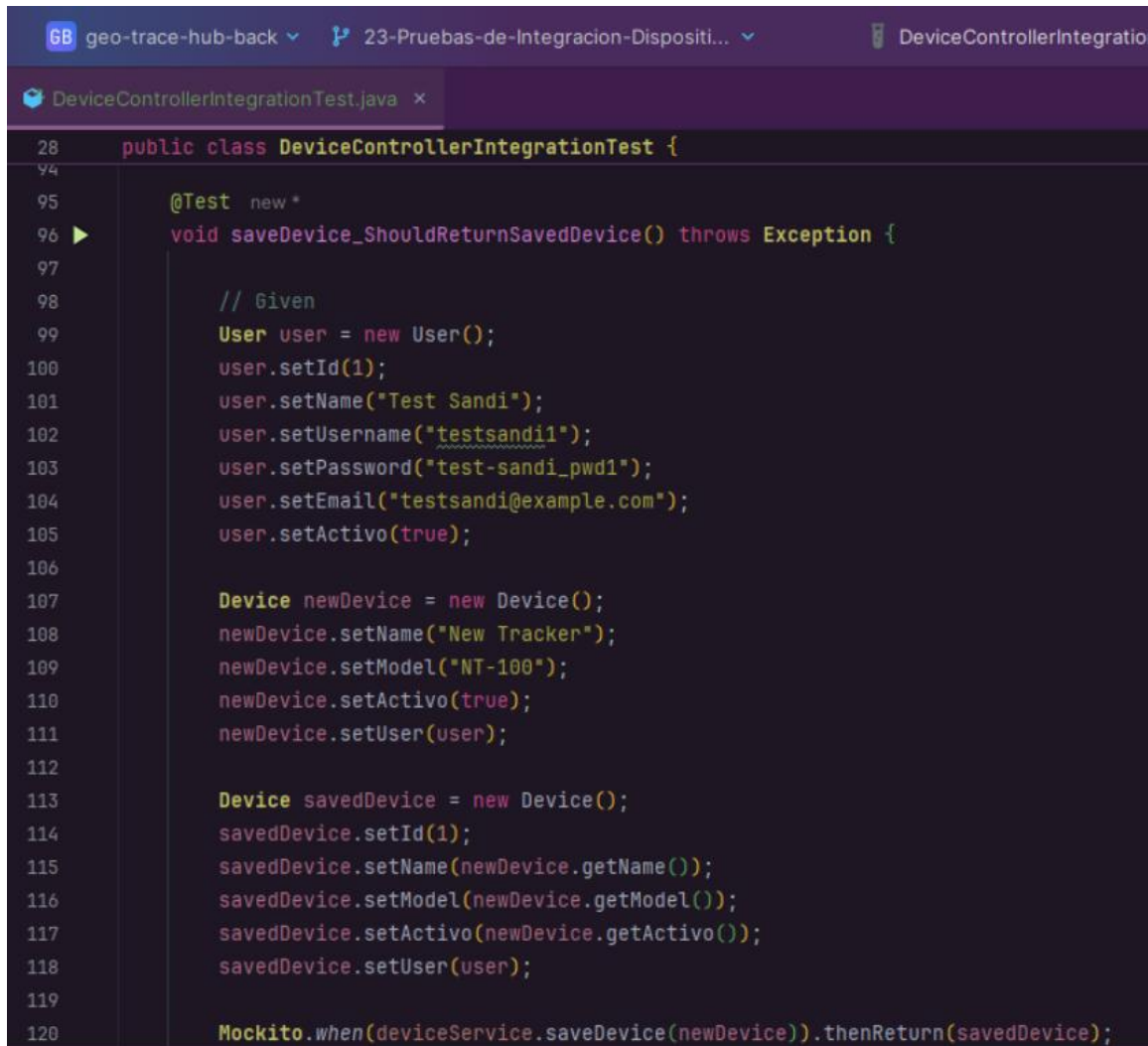
La Figura 180 plasma el tercer test “saveDevice_ShouldReturnSavedDevice()” permite verificar si guarda un dispositivo y devuelva el dispositivo guardado. Se crea primero un objeto “User” para crear un objeto “Device” llamado “newDevice” con atributos como nombre, modelo y el estado activo del dispositivo, este dispositivo se asocia con el usuario creado anteriormente. Luego se crea otro objeto nombrado “savedDevice” que simula ser el dispositivo guardado en la base de datos después de llamar al método “saveDevice”.

Este objeto tiene un ID asignado y los mismos atributos que “newDevice”, luego se usa Mockito para simular el comportamiento del servicio deviceService cuando se llama al método

“saveDevice” con “newDevice” como parámetro, se configura para que devuelva el objeto “savedDevice”:

Figura 180

Método saveDevice_ShouldReturnSavedDevice.



```
28 public class DeviceControllerIntegrationTest {
94
95     @Test new *
96     void saveDevice_ShouldReturnSavedDevice() throws Exception {
97
98         // Given
99         User user = new User();
100         user.setId(1);
101         user.setName("Test Sandi");
102         user.setUsername("testsandi1");
103         user.setPassword("test-sandi_pwd1");
104         user.setEmail("testsandi@example.com");
105         user.setActivo(true);
106
107         Device newDevice = new Device();
108         newDevice.setName("New Tracker");
109         newDevice.setModel("NT-100");
110         newDevice.setActivo(true);
111         newDevice.setUser(user);
112
113         Device savedDevice = new Device();
114         savedDevice.setId(1);
115         savedDevice.setName(newDevice.getName());
116         savedDevice.setModel(newDevice.getModel());
117         savedDevice.setActivo(newDevice.getActivo());
118         savedDevice.setUser(user);
119
120         Mockito.when(deviceService.saveDevice(newDevice)).thenReturn(savedDevice);
```

La prueba utiliza *mockMvc* para simular una solicitud *POST* al endpoint */devices*. El cuerpo de la solicitud es el objeto *newDevice* serializado mediante JSON mediante *objectMapper.writeValueAsString(newDevice)* permitiendo que en el resultado se impriman los mensajes de éxito de la ejecución y esperando la respuesta de status *200 OK* y los datos del dispositivo guardado, se observa en la Figura 181:

Figura 181

Configuración de la solicitud para saveDevice_ShouldReturnSavedDevice.

```
DeviceControllerIntegrationTest.java x
28 public class DeviceControllerIntegrationTest {
96 void saveDevice_ShouldReturnSavedDevice() throws Exception {
122 // When & Then
123 mockMvc.perform(post( uriTemplate: "/devices")
124 .contentType(MediaType.APPLICATION_JSON)
125 .content(objectMapper.writeValueAsString(newDevice)))
126 .andExpect(status().isOk())
127 .andExpect(jsonPath(expression: "$.id", is(savedDevice.getId())))
128 .andExpect(jsonPath(expression: "$.name", is(savedDevice.getName())))
129 .andExpect(jsonPath(expression: "$.model", is(savedDevice.getModel())))
130 .andExpect(jsonPath(expression: "$.activo", is(savedDevice.getActivo())))
131 .andExpect(jsonPath(expression: "$.user.name", is(user.getName())));
132 }
133 }
134 }
135 }
136 }
```

La Figura 182 presenta el resultado del test fue el siguiente mensaje: “Prueba saveDevice_ShouldReturnSavedDevice ejecutada con éxito Respuesta: {“id”:1, “name”:“New Tracker”, “model”:“NT-100”,“activo”:true, “user”:{“id”:1, “name”:“Test Sandi”, “username”:“testsandi1”, “email”: “testsandi@example.com”, “password”:“test-sandi_pwd1”,“activo”:true}}”. Y se marcó el test en verde:

Figura 182

Ejecución del test del método saveDevice_ShouldReturnSavedDevice.

```
Run DeviceControllerIntegrationTest.saveDevice_ShouldRet... x
DeviceControllerIntegrationTest.saveDevice_ShouldReturnSavedDevice() Tests passed: 1 of 1 test - 1sec 461ms
saveDevice_ShouldReturnSavedDevice() 1sec 461ms
Java HotSpot(TM) 64-Bit Server VM warning: Sharing is only supported for boot loader classes because bootstrap classpath is not well populated
2025-04-14T14:29:23.361-06:00 INFO 3552 --- [geo-trace-hup] [ main] o.s.mock.web.MockServletContext
2025-04-14T14:29:23.362-06:00 INFO 3552 --- [geo-trace-hup] [ main] o.s.t.web.servlet.TestDispatcherServlet
2025-04-14T14:29:23.363-06:00 INFO 3552 --- [geo-trace-hup] [ main] o.s.t.web.servlet.TestDispatcherServlet
2025-04-14T14:29:23.625-06:00 INFO 3552 --- [geo-trace-hup] [ main] m.t.g.controller.DeviceController

Prueba saveDevice_ShouldReturnSavedDevice ejecutada con éxito
Respuesta: {"id":1,"name":"New Tracker","model":"NT-100","activo":true,"user":{"id":1,"name":"Test Sandi","username":"testsandi1","email":"testsandi@example.com","password":"test-sandi_pwd1","activo":true}}
Process finished with exit code 0
```

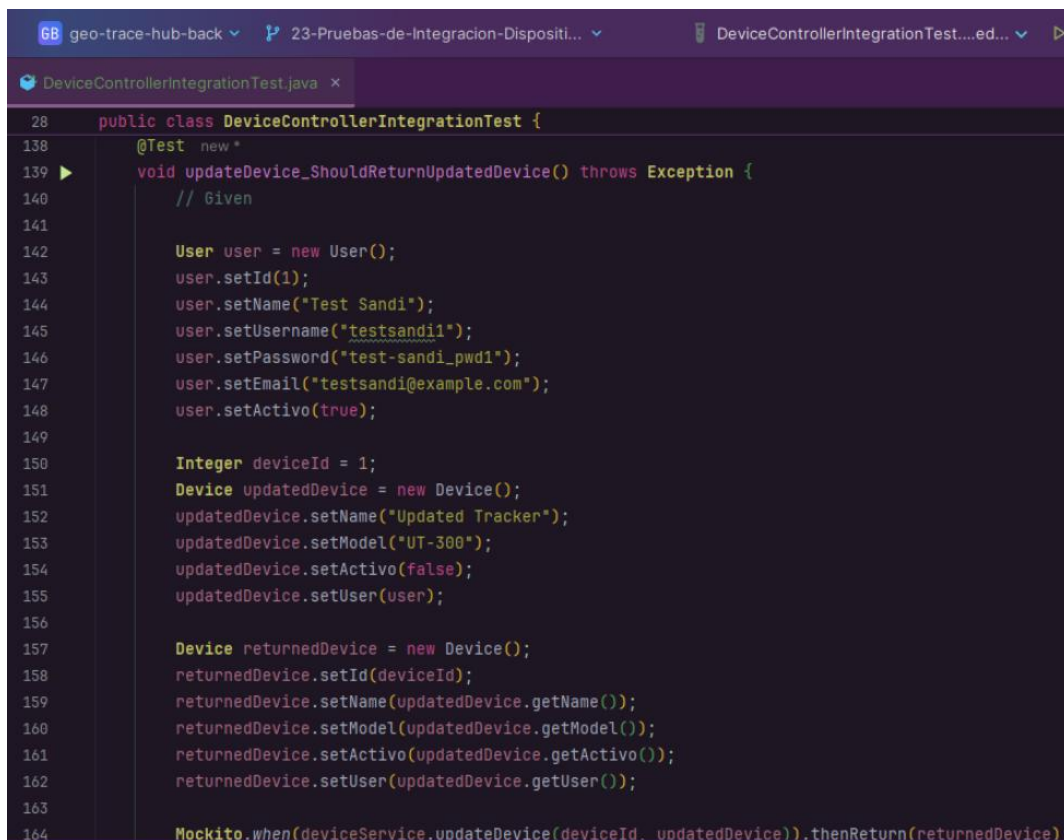
El cuarto método de test es “updateDevice_ShouldReturnUpdatedDevice()” que permitirá probar la actualización de la información de un dispositivo. Como se indica en la Figura 183 se inicia con la configuración inicial de crear un objeto “User” para asociarlo con el objeto “Device” en el que se define solamente el deviceId del dispositivo que se va a actualizar, en este caso, con valor 1

que posteriormente se crea un objeto nombrado “updatedDevice” con los nuevos atributos que se quieren asignar al dispositivo existente, para así crear otro objeto “Device” de nombre “returnedDevice” permite simular el dispositivo actualizado después de llamar al método “updateDevice”.

Posteriormente se usa *Mockito.when* que simulará el comportamiento del servicio “deviceService” cuando se llama al método “updateDevice” así como a “updatedDevice”, para que devuelva el objeto “returnedDevice”:

Figura 183

Método updateDevice_ShouldReturnUpdatedDevice



```
28 public class DeviceControllerIntegrationTest {
138     @Test new *
139     void updateDevice_ShouldReturnUpdatedDevice() throws Exception {
140         // Given
141
142         User user = new User();
143         user.setId(1);
144         user.setName("Test Sandi");
145         user.setUsername("testsandi1");
146         user.setPassword("test-sandi_pwd1");
147         user.setEmail("testsandi@example.com");
148         user.setActivo(true);
149
150         Integer deviceId = 1;
151         Device updatedDevice = new Device();
152         updatedDevice.setName("Updated Tracker");
153         updatedDevice.setModel("UT-300");
154         updatedDevice.setActivo(false);
155         updatedDevice.setUser(user);
156
157         Device returnedDevice = new Device();
158         returnedDevice.setId(deviceId);
159         returnedDevice.setName(updatedDevice.getName());
160         returnedDevice.setModel(updatedDevice.getModel());
161         returnedDevice.setActivo(updatedDevice.getActivo());
162         returnedDevice.setUser(updatedDevice.getUser());
163
164         Mockito.when(deviceService.updateDevice(deviceId, updatedDevice)).thenReturn(returnedDevice);
```

Ahora en la Figura 184 usa *mockMvc* que simula una solicitud HTTP *PUT* al endpoint */devices/{deviceId}*, donde {deviceId} se reemplaza con el valor de deviceId, para que imprimen los valores del dispositivo actualizado y esperando una respuesta *200 OK*:

Figura 184

Configuración de la solicitud para updateDevice_ShouldReturnUpdatedDevice.

```
DeviceControllerIntegrationTest.java x
28 public class DeviceControllerIntegrationTest {
139 void updateDevice_ShouldReturnUpdatedDevice() throws Exception {
163
164     Mockito.when(deviceService.updateDevice(deviceId, updatedDevice)).thenReturn(returnedDevice);
165
166     // When & Then
167     mockMvc.perform(put( uriTemplate: "/devices/{deviceId}", deviceId)
168         .contentType(MediaType.APPLICATION_JSON)
169         .content(objectMapper.writeValueAsString(updatedDevice)))
170         .andExpect( MvcResult result -> {
171             System.out.println("\nPrueba updateDevice_ShouldReturnUpdatedDevice() ejecutada con éxito");
172             System.out.println("Respuesta: " + result.getResponse().getContentAsString());
173         })
174         .andExpect(status().isOk())
175         .andExpect(jsonPath( expression: "$.id", is(deviceId)))
176         .andExpect(jsonPath( expression: "$.name", is(updatedDevice.getName())))
177         .andExpect(jsonPath( expression: "$.model", is(updatedDevice.getModel())))
178         .andExpect(jsonPath( expression: "$.activo", is(updatedDevice.getActivo())))
179         .andExpect(jsonPath( expression: "$.user.name", is(user.getName())));
180 }
```

La Figura 185 muestra que sí se aprobó el test de integración marcándolo de color verde y en la terminal se imprimió lo siguiente:

“Prueba updateDevice_ShouldReturnUpdatedDevice() ejecutada con éxito. Respuesta: {"id":1, "name":"Updated Tracker", "model":"UT 300", "activo":false, "user": {"id":1, "name":"Test Sandi", "username":"testsandi1", "email":"testsandi@example.com", "password":"test-sandi_pwd1", "activo":true}}”:

Figura 185

Ejecución del test del método updateDevice_ShouldReturnUpdatedDevice.

```
DeviceControllerIntegrationTest.java x
28 public class DeviceControllerIntegrationTest {
139 void updateDevice_ShouldReturnUpdatedDevice() throws Exception {
140     // Given
141
142     User user = new User();
143     user.setId(1);
144     user.setName("Test Sandi");
145     user.setUsername("testsandi1");
146     user.setPassword("test-sandi_pwd1");
147     user.setEmail("testsandi@example.com");
148     user.setActivo(true);
149 }

Run DeviceControllerIntegrationTest.updateDevice_ShouldReturnUpdatedDevice() x
Tests passed: 1 of 1 test - 1sec 451ms
updateDevice_ShouldReturnUpdatedDevice() 1sec 451ms

Java HotSpot(TM) 64-Bit Server VM warning: Sharing is only supported for boot loader classes because bootstrap classpath is not well initialized
2025-04-14T14:53:18.648-06:00 INFO 16352 --- [geo-trace-hup] [main] o.s.mock.web.MockServletContext
2025-04-14T14:53:18.649-06:00 INFO 16352 --- [geo-trace-hup] [main] o.s.t.web.servlet.TestDispatcherServlet
2025-04-14T14:53:18.650-06:00 INFO 16352 --- [geo-trace-hup] [main] o.s.t.web.servlet.TestDispatcherServlet
2025-04-14T14:53:18.904-06:00 INFO 16352 --- [geo-trace-hup] [main] m.t.g.controller.DeviceController

Prueba updateDevice_ShouldReturnUpdatedDevice() ejecutada con éxito
Respuesta: {"id":1,"name":"Updated Tracker","model":"UT-300","activo":false,"user":{"id":1,"name":"Test Sandi","username":"testsandi1","email":"testsandi@example.com","password":"test-sandi_pwd1","activo":true}}

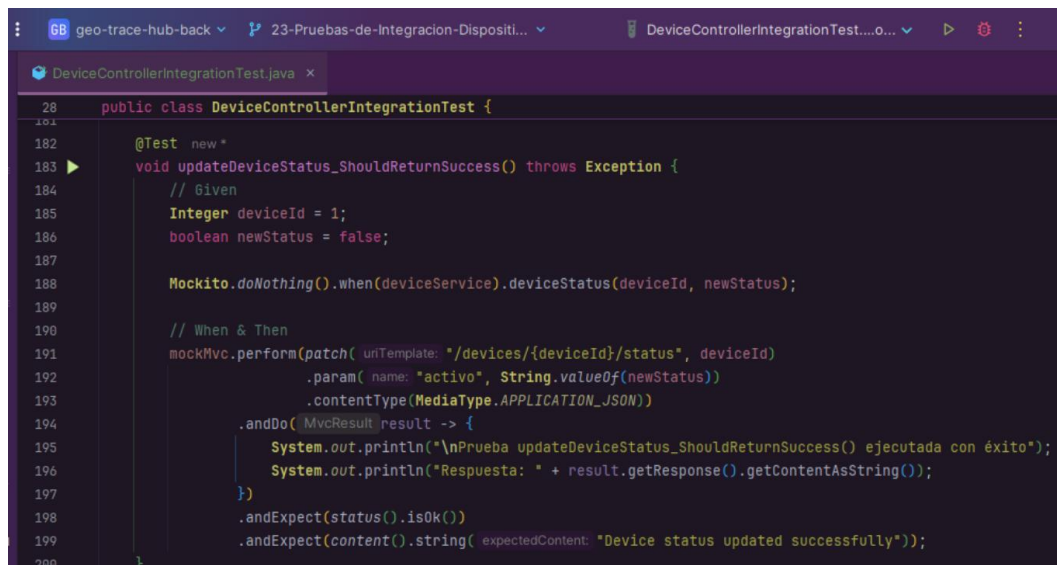
Process finished with exit code 0
```

El quinto método del test verifica el funcionamiento del endpoint para actualizar el estado (activo o inactivo) de un dispositivo (Device) existente. Se define el “deviceId” que se va a actualizar y se define el nuevo estado que se asignará al dispositivo, en este caso con false (inactivo). Luego con Mockito simulará el comportamiento del servicio deviceService cuando se llama al método deviceStatus para que no realice ninguna acción.

Posteriormente con *mockMvc* simula una solicitud *PATCH* al endpoint */devices/{deviceId}/status* que verificará el envío de una solicitud para actualizar el estado de un dispositivo existente que tendrá el mensaje esperado, se visualiza en la Figura 186:

Figura 186

Método updateDeviceStatus_ShouldReturnSuccess.



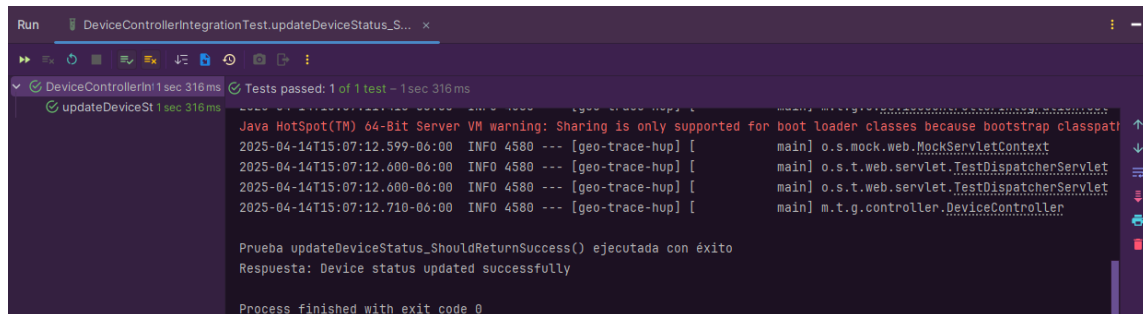
```
28 public class DeviceControllerIntegrationTest {
181
182     @Test new *
183     void updateDeviceStatus_ShouldReturnSuccess() throws Exception {
184         // Given
185         Integer deviceId = 1;
186         boolean newStatus = false;
187
188         Mockito.doNothing().when(deviceService).deviceStatus(deviceId, newStatus);
189
190         // When & Then
191         mockMvc.perform(patch( uriTemplate: "/devices/{deviceId}/status", deviceId)
192             .param( name: "activo", String.valueOf(newStatus))
193             .contentType(MediaType.APPLICATION_JSON))
194             .andDo( MvcResult result -> {
195                 System.out.println("\nPrueba updateDeviceStatus_ShouldReturnSuccess() ejecutada con éxito");
196                 System.out.println("Respuesta: " + result.getResponse().getContentAsString());
197             })
198             .andExpect(status().isOk())
199             .andExpect(content().string( expectedContent: "Device status updated successfully"));
200     }
```

La Figura 187 detalla el comportamiento de la ejecución del test, el cual se marcó de color verde y el resultado del test en la terminal es el siguiente:

“Prueba updateDeviceStatus_ShouldReturnSuccess() ejecutada con éxito. Respuesta: Device status updated successfully”.

Figura 187

Ejecución del test del método `updateDeviceStatus_ShouldReturnSuccess`.



```
Run DeviceControllerIntegrationTest.updateDeviceStatus_S... x
Tests passed: 1 of 1 test - 1 sec 316 ms
updateDeviceSt 1 sec 316 ms
Java HotSpot(TM) 64-Bit Server VM warning: Sharing is only supported for boot loader classes because bootstrap classpath
2025-04-14T15:07:12.599-06:00 INFO 4580 --- [geo-trace-hup] [main] o.s.mock.web.MockServletContext
2025-04-14T15:07:12.600-06:00 INFO 4580 --- [geo-trace-hup] [main] o.s.t.web.servlet.TestDispatcherServlet
2025-04-14T15:07:12.600-06:00 INFO 4580 --- [geo-trace-hup] [main] o.s.t.web.servlet.TestDispatcherServlet
2025-04-14T15:07:12.710-06:00 INFO 4580 --- [geo-trace-hup] [main] m.t.g.controller.DeviceController

Prueba updateDeviceStatus_ShouldReturnSuccess() ejecutada con éxito
Respuesta: Device status updated successfully

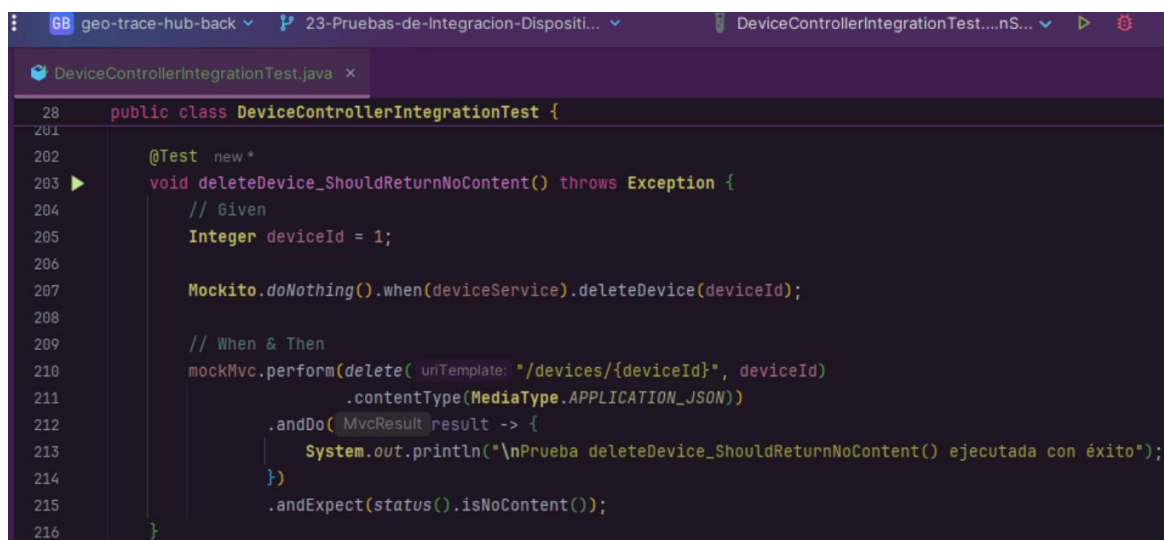
Process finished with exit code 0
```

La Figura 188 contiene el sexto método, permite verificar la eliminación de un dispositivo. Se define el `deviceId` que se va a eliminar, en este caso, con valor 1, luego se utiliza Mockito para simular el comportamiento del servicio “deviceService” cuando se llama al método `deleteDevice` con `deviceId` como parámetro.

En este caso, se configura para que no realice ninguna acción, simulando que la eliminación fue exitosa, luego con `mockMvc` para simular una solicitud `DELETE` al endpoint `/devices/{deviceId}`, donde `{deviceId}` se reemplaza con el valor de `deviceId`, en el que la respuesta tiene el código de estado esperado (`204 No Content`), simulando la interacción con el servicio mediante Mockito.

Figura 188

Método `deleteDevice_ShouldReturnNoContent`.



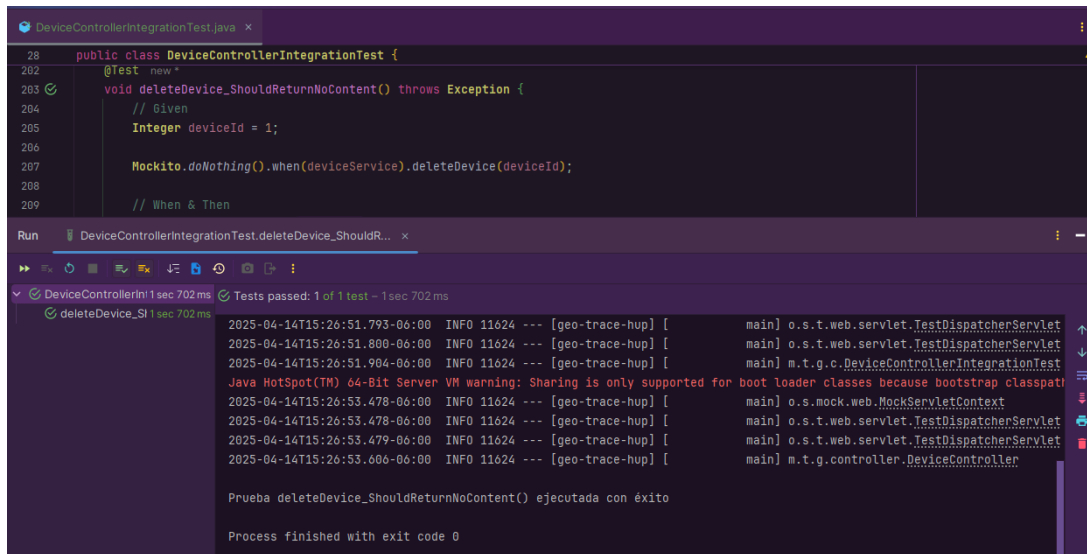
```
68 geo-trace-hub-back 23-Pruebas-de-Integracion-Dispositi... DeviceControllerIntegrationTest...ns...
DeviceControllerIntegrationTest.java x
28 public class DeviceControllerIntegrationTest {
201
202     @Test new *
203     void deleteDevice_ShouldReturnNoContent() throws Exception {
204         // Given
205         Integer deviceId = 1;
206
207         Mockito.doNothing().when(deviceService).deleteDevice(deviceId);
208
209         // When & Then
210         mockMvc.perform(delete( uriTemplate: */devices/{deviceId}*, deviceId)
211             .contentType(MediaType.APPLICATION_JSON))
212             .andExpect(status().isNoContent());
213         System.out.println("\nPrueba deleteDevice_ShouldReturnNoContent() ejecutada con éxito");
214     }
215
216 }
```


La Figura 189 muestra el resultado del test en la terminal con mensajes como:

“Prueba deleteDevice_ShouldReturnNoContent() ejecutada con éxito” y este test se marcó en verde:

Figura 189

Ejecución del test del método deleteDevice_ShouldReturnNoContent.

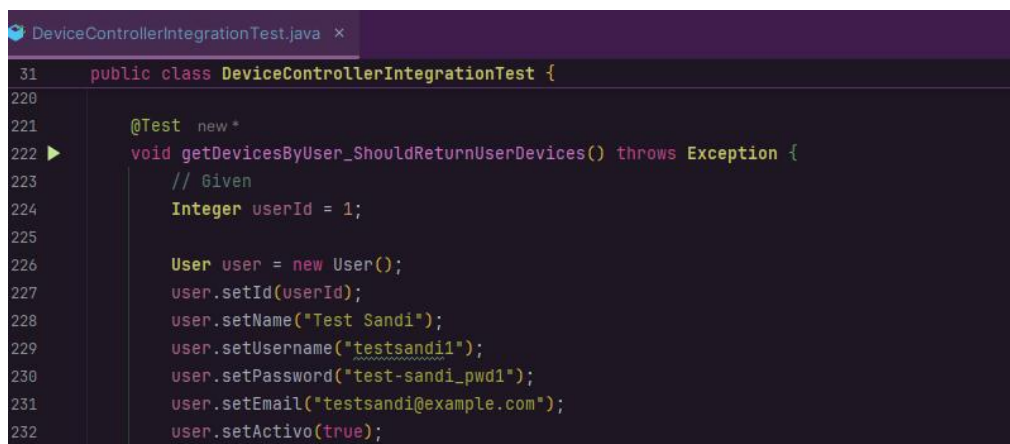


```
28 public class DeviceControllerIntegrationTest {
29     @Test new *
30     void deleteDevice_ShouldReturnNoContent() throws Exception {
31         // Given
32         Integer deviceId = 1;
33
34         Mockito.doNothing().when(deviceService).deleteDevice(deviceId);
35
36         // When & Then
37     }
38 }
39
Run DeviceControllerIntegrationTest.deleteDevice_ShouldR...
DeviceControllerIntegrationTest.deleteDevice_ShouldReturnNoContent() Tests passed: 1 of 1 test - 1sec 702ms
deleteDevice_ShouldReturnNoContent() 1sec 702ms
2025-04-14T15:26:51.793-06:00 INFO 11624 --- [geo-trace-hup] [main] o.s.t.web.servlet.TestDispatcherServlet
2025-04-14T15:26:51.800-06:00 INFO 11624 --- [geo-trace-hup] [main] o.s.t.web.servlet.TestDispatcherServlet
2025-04-14T15:26:51.904-06:00 INFO 11624 --- [geo-trace-hup] [main] m.t.g.c.DeviceControllerIntegrationTest
Java HotSpot(TM) 64-Bit Server VM warning: Sharing is only supported for boot loader classes because bootstrap classpath
2025-04-14T15:26:53.478-06:00 INFO 11624 --- [geo-trace-hup] [main] o.s.mock.web.MockServletContext
2025-04-14T15:26:53.478-06:00 INFO 11624 --- [geo-trace-hup] [main] o.s.t.web.servlet.TestDispatcherServlet
2025-04-14T15:26:53.479-06:00 INFO 11624 --- [geo-trace-hup] [main] o.s.t.web.servlet.TestDispatcherServlet
2025-04-14T15:26:53.606-06:00 INFO 11624 --- [geo-trace-hup] [main] m.t.g.controller.DeviceController
Prueba deleteDevice_ShouldReturnNoContent() ejecutada con éxito
Process finished with exit code 0
```

Por último, en la Figura 190 muestra al método “getDevicesByUser_ShouldReturnUserDevices” que retorna los dispositivos de un solo usuario. Primero se creará un usuario al que se le asociarán dos dispositivos:

Figura 190

Método getDevicesByUser_ShouldReturnUserDevices.



```
31 public class DeviceControllerIntegrationTest {
32
33     @Test new *
34     void getDevicesByUser_ShouldReturnUserDevices() throws Exception {
35         // Given
36         Integer userId = 1;
37
38         User user = new User();
39         user.setId(userId);
40         user.setName("Test Sandi");
41         user.setUsername("testsandi1");
42         user.setPassword("test-sandi_pwd1");
43         user.setEmail("testsandi@example.com");
44         user.setActivo(true);
45     }
46 }
```

Después en la Figura 191 se detallan dos dispositivos que se agregaron a una lista o array de dispositivos, en el que con Mockito llamará al servicio que busca el ID del usuario creado para mostrar los dispositivos de ese usuario:

Figura 191

Configuración en `getDevicesByUser_ShouldReturnUserDevices`.

```
DeviceControllerIntegrationTest.java x
31 public class DeviceControllerIntegrationTest {
222 void getDevicesByUser_ShouldReturnUserDevices() throws Exception {
233
234     List<Device> devices = new ArrayList<>();
235     Device device1 = new Device();
236     device1.setId(1);
237     device1.setName("GPS Tracker Pro");
238     device1.setModel("6T-200");
239     device1.setActivo(true);
240     device1.setUser(user);
241
242     Device device2 = new Device();
243     device2.setId(2);
244     device2.setName("GPS Tracker Basic");
245     device2.setModel("6T-100");
246     device2.setActivo(true);
247     device2.setUser(user);
248
249     devices.add(device1);
250     devices.add(device2);
251
252     Mockito.when(deviceService.getDevicesById(userId)).thenReturn(devices);
253 }
```

En la Figura 192 se usa *mockMvc* que simula la petición del endpoint teniendo como resultado la búsqueda de todos los dispositivos con su información perteneciente al usuario:

Figura 192

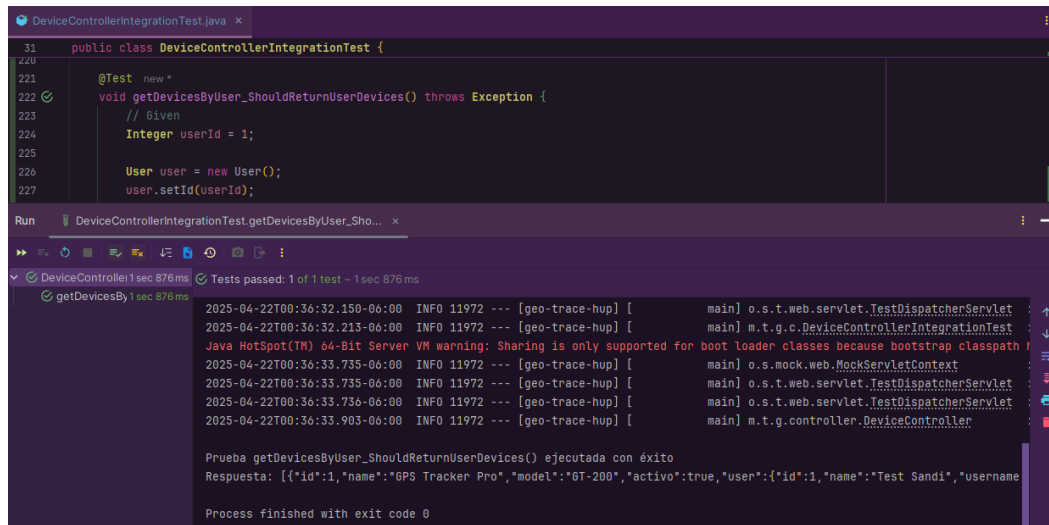
Configuración de la solicitud en `getDevicesByUser_ShouldReturnUserDevices`.

```
DeviceControllerIntegrationTest.java x
31 public class DeviceControllerIntegrationTest {
222 void getDevicesByUser_ShouldReturnUserDevices() throws Exception {
231
232     Mockito.when(deviceService.getDevicesById(userId)).thenReturn(devices);
233
234     // When & Then
235     mockMvc.perform(get( uriTemplate: "/devices/user/{userId}", userId)
236                     .contentType(MediaType.APPLICATION_JSON))
237         .andExpect(status().isOk())
238         .andExpect(jsonPath("expression: "$", hasSize(2)))
239         .andExpect(jsonPath("expression: "$[0].id", is(device1.getId())))
240         .andExpect(jsonPath("expression: "$[0].name", is(device1.getName())))
241         .andExpect(jsonPath("expression: "$[0].model", is(device1.getModel())))
242         .andExpect(jsonPath("expression: "$[0].activo", is(device1.getActivo())))
243         .andExpect(jsonPath("expression: "$[0].user.name", is(user.getName())))
244         .andExpect(jsonPath("expression: "$[1].id", is(device2.getId())))
245         .andExpect(jsonPath("expression: "$[1].name", is(device2.getName())))
246         .andExpect(jsonPath("expression: "$[1].model", is(device2.getModel())))
247         .andExpect(jsonPath("expression: "$[1].activo", is(device2.getActivo())))
248         .andExpect(jsonPath("expression: "$[1].user.name", is(user.getName())));
249
250 }
```

Al ejecutar el test, se marcó en verde con una palomita, indicando que el test sí pasó la prueba, ilustrada en la Figura 193:

Figura 193

Ejecución del test del método `getDevicesByUser_ShouldReturnUserDevices`.



```
31 public class DeviceControllerIntegrationTest {
220
221     @Test new *
222     void getDevicesByUser_ShouldReturnUserDevices() throws Exception {
223         // Given
224         Integer userId = 1;
225
226         User user = new User();
227         user.setId(userId);
228
229     }
230 }
231
232 Run DeviceControllerIntegrationTest.getDevicesByUser_Sho... x
233
234 Tests passed: 1 of 1 test - 1 sec 876 ms
235
236 getDevicesByUser_ShouldReturnUserDevices() ejecutada con éxito
237 Respuesta: [{"id":1,"name":"GPS Tracker Pro","model":"GT-200","activo":true,"user":{"id":1,"name":"Test Sandi","username
238 Process finished with exit code 0
```

En la terminal se imprimieron los siguientes mensajes en la depuración:

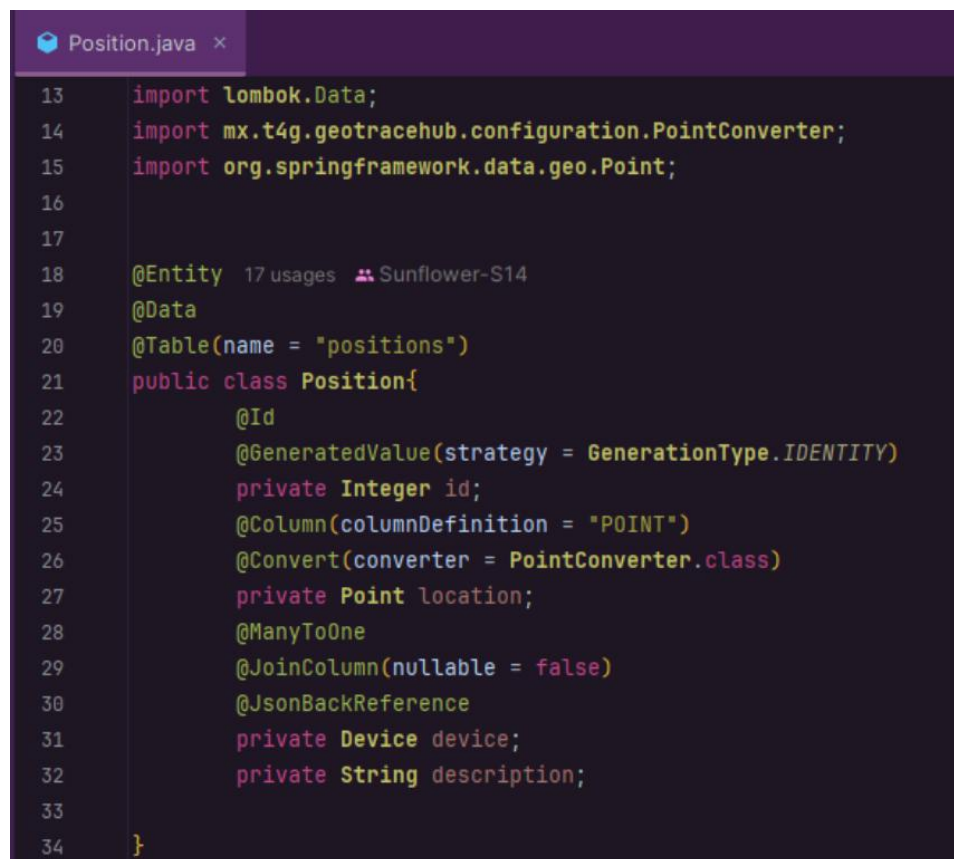
- Prueba `getDevicesByUser_ShouldReturnUserDevices()` ejecutada con éxito
- Respuesta: [{"id":1,"name":"GPS Tracker Pro","model":"GT-200","activo":true,"user":{"id":1,"name":"Test Sandi","username":"testsandi1","email":"testsandi@example.com","password":"test-sandi_pwd1","activo":true}},{"id":2,"name":"GPS Tracker Basic","model":"GT-100","activo":true,"user":{"id":1,"name":"Test Sandi","username":"testsandi1","email":"testsandi@example.com","password":"test-sandi_pwd1","activo":true}}]

3.1.18. Administración de ubicaciones

Modelo o entidad: se crea el modelo position que permitirá en una nueva tabla declarada con el nombre de “positions” almacenar las ubicaciones geográficas de los dispositivos. Sus atributos son “location” de tipo *point* que permite el almacenamiento en la base de datos de la posición en coordenadas, también se declara el atributo “device” que representa al dispositivo al que pertenece la posición, relacionado mediante una relación muchos a uno, y el atributo “description” es la descripción de la posición que en sí puede ser opcional el llenado de este atributo, ejemplificada en la Figura 194:

Figura 194

Clase Position.

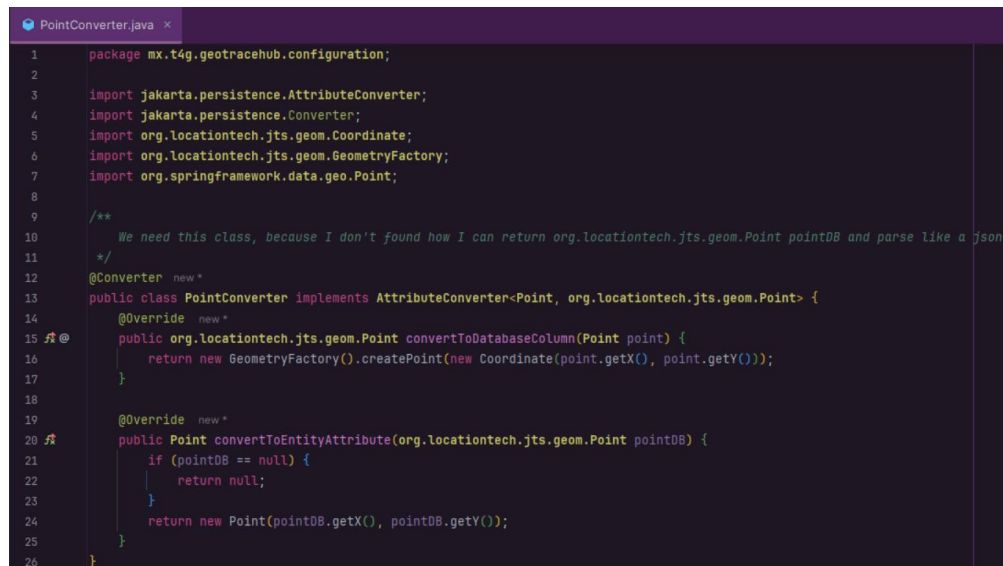


```
13 import lombok.Data;
14 import mx.t4g.geotracehub.configuration.PointConverter;
15 import org.springframework.data.geo.Point;
16
17
18 @Entity 17 usages 🚩 Sunflower-S14
19 @Data
20 @Table(name = "positions")
21 public class Position{
22     @Id
23     @GeneratedValue(strategy = GenerationType.IDENTITY)
24     private Integer id;
25     @Column(columnDefinition = "POINT")
26     @Convert(converter = PointConverter.class)
27     private Point location;
28     @ManyToOne
29     @JoinColumn(nullable = false)
30     @JsonBackReference
31     private Device device;
32     private String description;
33
34 }
```

En la Figura 195 permite visualizar la conversión así como el almacenamiento de las posiciones se crea dentro de la carpeta configuration la clase “PointConverter” que actúa como un intermediario entre el modelo Position y la base de datos, permitiendo que se almacenen como recuperar puntos geográficos de manera correcta y transparente para la solicitud:

Figura 195

Clase PointConverter.



```
1 package mx.t4g.geotracehub.configuration;
2
3 import jakarta.persistence.AttributeConverter;
4 import jakarta.persistence.Converter;
5 import org.locationtech.jts.geom.Coordinate;
6 import org.locationtech.jts.geom.GeometryFactory;
7 import org.springframework.data.geo.Point;
8
9 /**
10  * We need this class, because I don't found how I can return org.locationtech.jts.geom.Point pointDB and parse like a json
11  */
12 @Converter(new *)
13 public class PointConverter implements AttributeConverter<Point, org.locationtech.jts.geom.Point> {
14     @Override(new *)
15     public org.locationtech.jts.geom.Point convertToDatabaseColumn(Point point) {
16         return new GeometryFactory().createPoint(new Coordinate(point.getX(), point.getY()));
17     }
18
19     @Override(new *)
20     public Point convertToEntityAttribute(org.locationtech.jts.geom.Point pointDB) {
21         if (pointDB == null) {
22             return null;
23         }
24         return new Point(pointDB.getX(), pointDB.getY());
25     }
26 }
```

Repositorio o repository: se crea la interfaz “PositionRepository” apreciada en la Figura 196 que interactúa con la base de datos realizando consultas SQL con el método “getAllByDevice(Device device)”, este devuelve una lista de posiciones asociadas a un dispositivo específico:

Figura 196

Interfaz PositionsRepository.



```
1 package mx.t4g.geotracehub.repositories;
2
3 import mx.t4g.geotracehub.model.Device;
4 import mx.t4g.geotracehub.model.Position;
5 import org.springframework.data.jpa.repository.JpaRepository;
6 import org.springframework.stereotype.Repository;
7
8 import java.util.List;
9
10 @Repository(2 usages, Sunflower-S14)
11 public interface PositionsRepository extends JpaRepository<Position, Integer> {
12     List<Position> getAllByDevice(Device device); 1 usage, Sunflower-S14
13 }
```

Service o servicio: define la interfaz “PositionService” que llamará a los métodos que contienen la lógica de negocio, en este caso solo se tienen creados dos, “getPointByDevice” que permitirá obtener todas las posiciones de un dispositivo y “storePosition” que sirve para almacenar posiciones, ilustrada en la Figura 197:

Figura 197

Interfaz PositionsService.



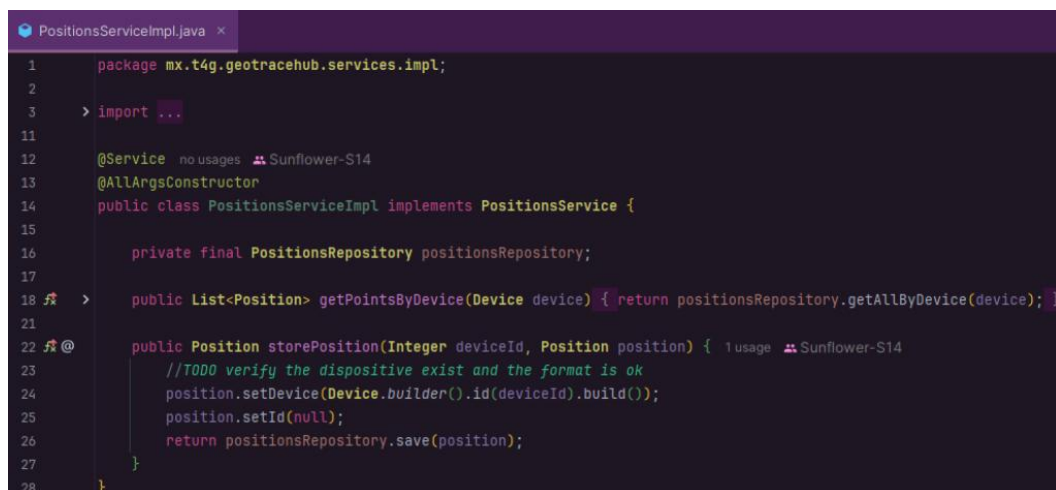
```
1 package mx.t4g.geotracehub.services;
2
3 > import ...
4
5
6
7
8 public interface PositionsService { 4 usages 1 implementation 🚩 Sunflower-S14
9
10     List<Position> getPointsByDevice(Device device); 1 usage 1 implementation 🚩 Sunflower-S14
11     Position storePosition(Integer deviceId, Position position); 1 usage 1 implementation 🚩 Sunflower-S14
12 }
```

Servicio implementado o Service Implements: se creó la clase “PositionsServiceImpl” mostrado en la Figura 198. Esta clase contiene la lógica del negocio, en el que se tienen dos métodos:

1. getPointsByDevice(Device device): devuelve una lista de posiciones asociadas a un dispositivo específico.
2. storePosition(Integer deviceId, Position position): este método almacena una nueva posición en la base de datos.

Figura 198

Métodos getPointsByDevice y storePosition.



```
1 package mx.t4g.geotracehub.services.impl;
2
3 > import ...
4
5
6
7
8
9
10
11
12 @Service no usages 🚩 Sunflower-S14
13 @AllArgsConstructor
14 public class PositionsServiceImpl implements PositionsService {
15
16     private final PositionsRepository positionsRepository;
17
18     public List<Position> getPointsByDevice(Device device) { return positionsRepository.getAllByDevice(device); }
19
20
21
22     public Position storePosition(Integer deviceId, Position position) { 1 usage 🚩 Sunflower-S14
23         //TODO verify the dispositive exist and the format is ok
24         position.setDevice(Device.builder().id(deviceId).build());
25         position.setId(null);
26         return positionsRepository.save(position);
27     }
28 }
```

Controlador *REST* o Controller *REST*: se define la clase “PositionsController” que permite establecer los *endpoints* raíz *device/{deviceId}/positions* y los *endpoints* para las solicitudes HTTP se declararon dos que se puede analizar en la Figura 199:

Figura 199

Clase PositionsController.

```
PositionsController.java x
1 package mx.t4g.geotracehub.controller; // Define el paquete donde se encuentra esta clase
2
3 > import ...
17
18 @Slf4j // Anotación de Lombok para habilitar logging en la clase no usages Sunflower-S14
19 @RestController // Define esta clase como un controlador REST
20 @AllArgsConstructor // Genera un constructor con todos los argumentos automáticamente
21 @RequestMapping("/device/{deviceId}/positions") // Define la ruta base para este controlador
22 public class PositionsController {
```

Se creó el método “getPositions” que permite manejar las solicitudes *GET* para obtener todas las posiciones de un dispositivo, en el que el endpoint asignado con *@GetMapping* el cual en este caso indicará que se basará en la ruta raíz del controlador de posiciones representado en la Figura 200:

Figura 200

Método getPositions del controlador.

```
21 @RequestMapping("/device/{deviceId}/positions") // Define la ruta base para este controlador
22 public class PositionsController {
23
24     private final PositionsService positionsService; // Servicio para manejar las posiciones
25
26     @GetMapping // Maneja solicitudes GET a la ruta base no usages Sunflower-S14
27     public List<Position> getPositions(@PathVariable Integer deviceId) { // Método para obtener posiciones de un dispositivo
28         log.info("Calling get positions from device {}", deviceId); // Registra una entrada en el log
29         return positionsService.getPointsByDevice(Device.builder().id(deviceId).build()); // Llama al servicio para obtener posiciones y devuelve
30     }
```

Asimismo se creó el método “savePosition” que maneja solicitudes *POST* para guardar una nueva posición de un dispositivo. Su endpoint también se basará en la ruta raíz pero con *@PostMapping* para almacenar las posiciones, puede verse en la Figura 201:

Figura 201

Método savePosition del controlador.

```
32  @PostMapping // Maneja solicitudes POST a la ruta base no usages Sunflower-S14
33  public ResponseEntity<Position> savePosition(@PathVariable Integer deviceId, @RequestBody Position position) { // Metodo para guardar una posi
34      log.info("Calling save position from device {} with this information {}", deviceId, position ); // Registra una entrada en el log
35      Position positionSaved = positionsService.storePosition(deviceId, position); // Llama al servicio para almacenar la posición
36      return ResponseEntity.ok(positionSaved); // Devuelve la posición guardada con una respuesta HTTP 200 OK
37  }
38  }
```

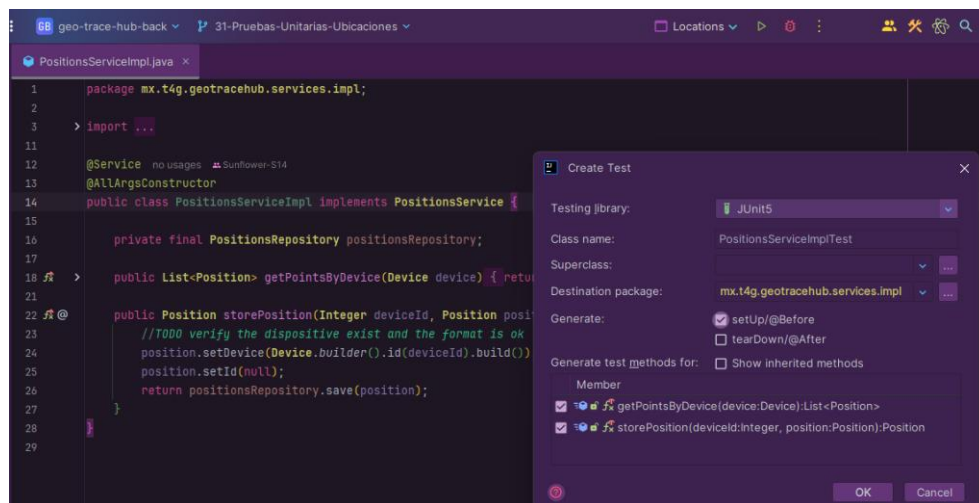

3.1.19. Pruebas unitarias de ubicaciones

Metodología: basado en Mockito para aislar la lógica de negocio de la clase “PositionsServiceImpl” para probar sus métodos de manera independiente de manera efectiva que garantizará la calidad del código en la lógica del negocio.

La clase “PositionsServiceImpl” evidenciada en la Figura 202 se crea la clase para las pruebas unitarias (previamente hecha en los demás servicios *REST*), en el que se selecciona la anotación *@Before* y los dos métodos de la clase:

Figura 202

Selección de los métodos en la clase PositionsServiceImpl.



En el que se crea la siguiente clase, se definió el mock al repositorio de las posiciones y el “injectmock” que usa a la clase “positionsServiceImpl” representado en la Figura 203:

Figura 203

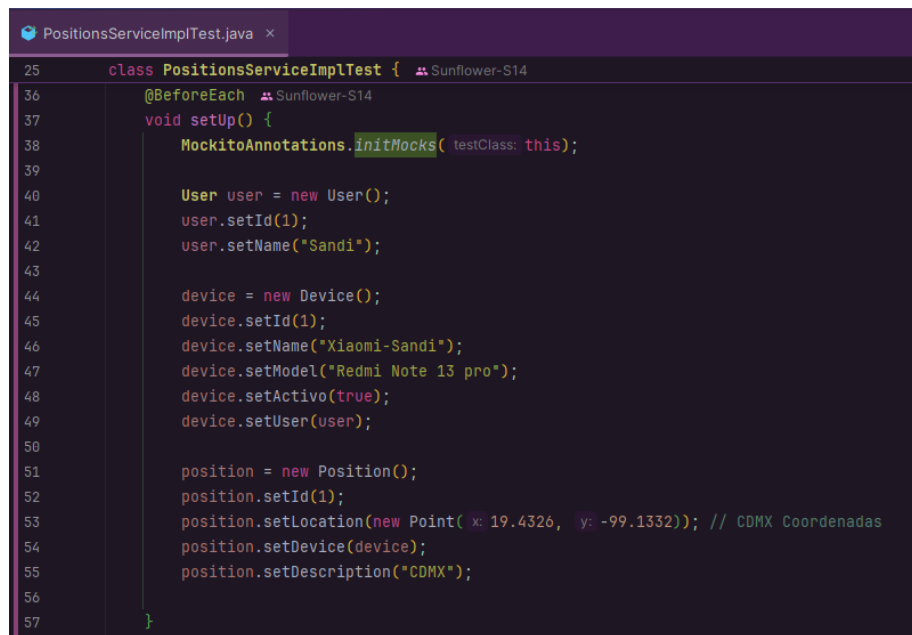
Creación de la clase PositionsServiceImplTest para las pruebas unitarias.

```
25 class PositionsServiceImplTest {
26
27     @Mock
28     private PositionsRepository positionsRepository;
29
30     @InjectMocks
31     private PositionsServiceImpl positionsServiceImpl;
32
33     private Device device;
34     private Position position;
```

En la Figura 204 se inicializa el *BeforeEach* con los datos de prueba; se crea un usuario de prueba con su dispositivo, se le agrega una posición, de ejemplo sería la CDMX con sus coordenadas:

Figura 204

Configuración del método setUp en la prueba unitarias de ubicaciones.



```
25 class PositionsServiceImplTest {
36     @BeforeEach
37     void setUp() {
38         MockitoAnnotations.initMocks(this);
39
40         User user = new User();
41         user.setId(1);
42         user.setName("Sandi");
43
44         Device device = new Device();
45         device.setId(1);
46         device.setName("Xiaomi-Sandi");
47         device.setModel("Redmi Note 13 pro");
48         device.setActivo(true);
49         device.setUser(user);
50
51         Position position = new Position();
52         position.setId(1);
53         position.setLocation(new Point(19.4326, -99.1332)); // CDMX Coordenadas
54         position.setDevice(device);
55         position.setDescription("CDMX");
56     }
57 }
```

El método "getPointsByDevice()" de la Figura 205 permite obtener todas las ubicaciones por dispositivo:

Figura 205

Método getPointsByDevice del test unitario.



```
25 class PositionsServiceImplTest {
58     @Test
59     void getPointsByDevice() {
60         // Simulamos que el repositorio devuelve una lista de posiciones
61         when(positionsRepository.getAllByDevice(device)).thenReturn(Arrays.asList(position));
62
63         List<Position> result = positionsServiceImpl.getPointsByDevice(device);
64
65         assertNotNull(result);
66         assertEquals(1, result.size());
67         assertEquals("CDMX", result.get(0).getDescription());
68         verify(positionsRepository, times(1)).getAllByDevice(device);
69     }
}
```

Al método “getPointsByDevice” se le agregó un *log* para que en la ejecución de la prueba unitaria aparezca un mensaje “Fetching all positions by dispositive”, se visualiza en la Figura 206:

Figura 206

Al método getPointsByDevice de la clase positionsServiceImpl se le agrega el log.info.

```
public List<Position> getPointsByDevice(Device device)
{
    log.info("Fetching all positions by dispositive");
    return positionsRepository.getAllByDevice(device);
}
```

La Figura 207 corresponde al resultado de ejecutar el test, que imprime el mensaje del *log* al mismo tiempo el test se marcó en color verde indicando que pasó la prueba:

Figura 207

Ejecución del test del método getPointsByDevice.



Al método “storePosition()” simula que se agregó una posición a un dispositivo. Primero se asegura de que el dispositivo exista y que la posición a almacenar exista, posteriormente puede guardar la ubicación; su definición está contenida en la Figura 208:

Figura 208

Método storePosition del test unitario.

```
@Test new *
void storePosition() {
    // Verificar que 'device' no sea null
    assertNotNull(device, message: "device debería estar inicializado en setUp()");
    assertNotNull(position, message: "position debería estar inicializado en setUp()");

    // Simulamos que el repositorio guarda la posición
    when(positionsRepository.save(any(Position.class))).thenReturn(position);

    Position result = positionsServiceImpl.storePosition(device.getId(), position);

    assertNotNull(result);
    assertEquals("expected: 'CDMX'", result.getDescription());
    assertEquals(device.getId(), result.getDevice().getId());
    verify(positionsRepository, times(wantedNumberOfInvocations: 1)).save(any(Position.class));
}
```

Al método “storePosition” de la clase PositionsServiceImpl se agrega un *log* para que se imprima un mensaje en la ejecución del test, el cual es descrito en la Figura 209:

Figura 209

Al método storePosition se le agrega el log.info.

```
public Position storePosition(Integer deviceId, Position position) {
    //TODO verify the dispositive exist and the format is ok
    log.info("Save position in dispositive {}");
    position.setDevice(Device.builder().id(deviceId).build());
    position.setId(null);
    return positionsRepository.save(position);
}
```

En la Figura 210 se visualiza que al ejecutar el test se marca de color verde y en la terminal se imprimen los siguientes mensajes:

```
-- Validating position before saving: Position(id=1, location=Point [x=19.432600, y=-99.133200],  
device=Device(id=1, name=Xiaomi-Sandi, model=Redmi Note 13 pro, activo=true,  
user=User(id=1, name=Sandi)
```

```
-- Saving position with description: CDMX, device ID: 1
```

Figura 210

Ejecución del test del método storePosition.



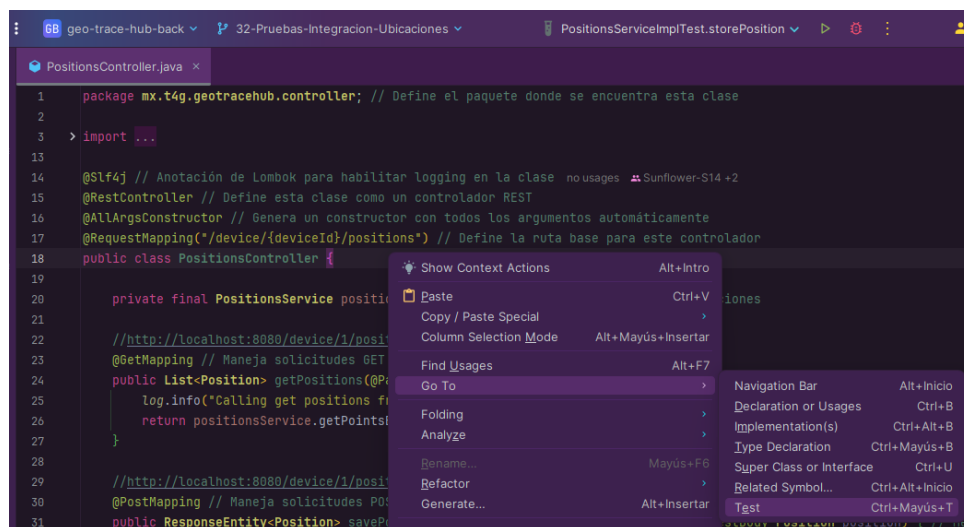
3.1.20. Pruebas de integración de ubicaciones

Metodología: se centra en verificar la correcta interacción entre el controlador “PositionsController” y la capa de servicio “PositionsService”, simulando solicitudes HTTP reales a través de *MockMvc*.

La Figura 211 corresponde a los pasos de creación del test de integración:

Figura 211

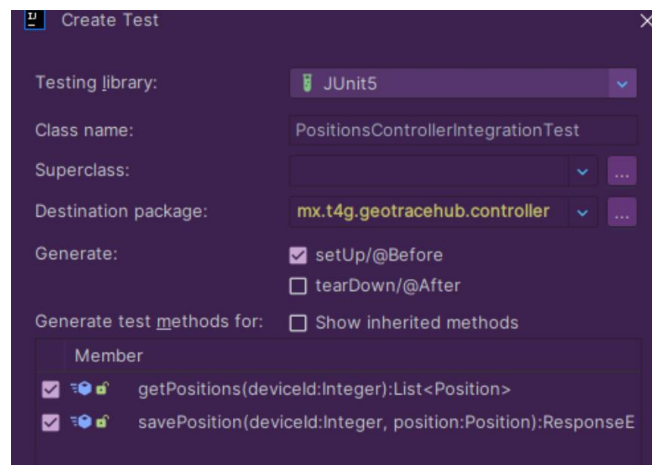
Clase PositionsController para la prueba de integración de ubicaciones.



En la Figura 212 aparece la siguiente ventana, para generar la clase de prueba:

Figura 212

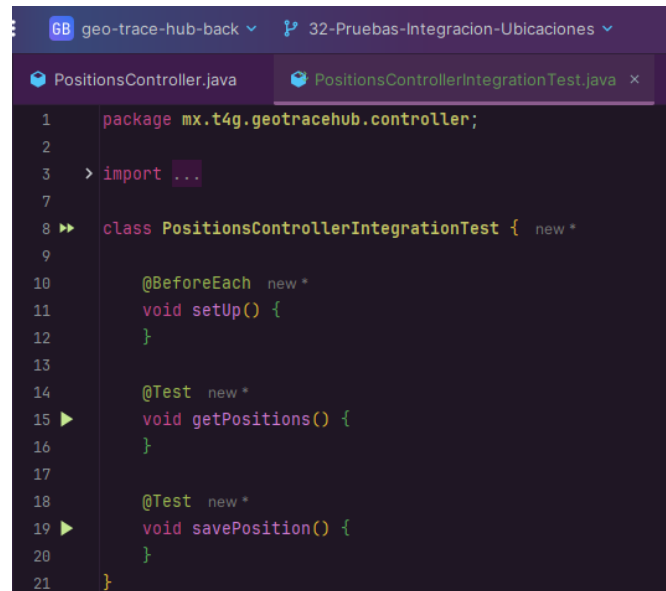
Selección de los métodos para la prueba de integración.



En la Figura 213 se crea la clase “PositionsControllerIntegrationTest” con los métodos generados para realizar las pruebas de integración:

Figura 213

Clase PositionsControllerIntegrationTest.

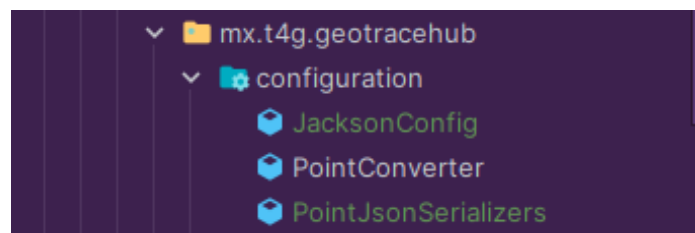


```
1 package mx.t4g.geotracehub.controller;
2
3 > import ...
4
5
6
7
8 class PositionsControllerIntegrationTest { new *
9
10     @BeforeEach new *
11     void setUp() {
12     }
13
14     @Test new *
15     void getPositions() {
16     }
17
18     @Test new *
19     void savePosition() {
20     }
21 }
```

La Figura 214 muestra que los test de integración puedan reconocer la conversión del objeto tipo POINT en la base de datos de la prueba, se crean 2 clases “JacksonConfig” y “PointJsonSerializers” dentro de la carpeta “configuration”:

Figura 214

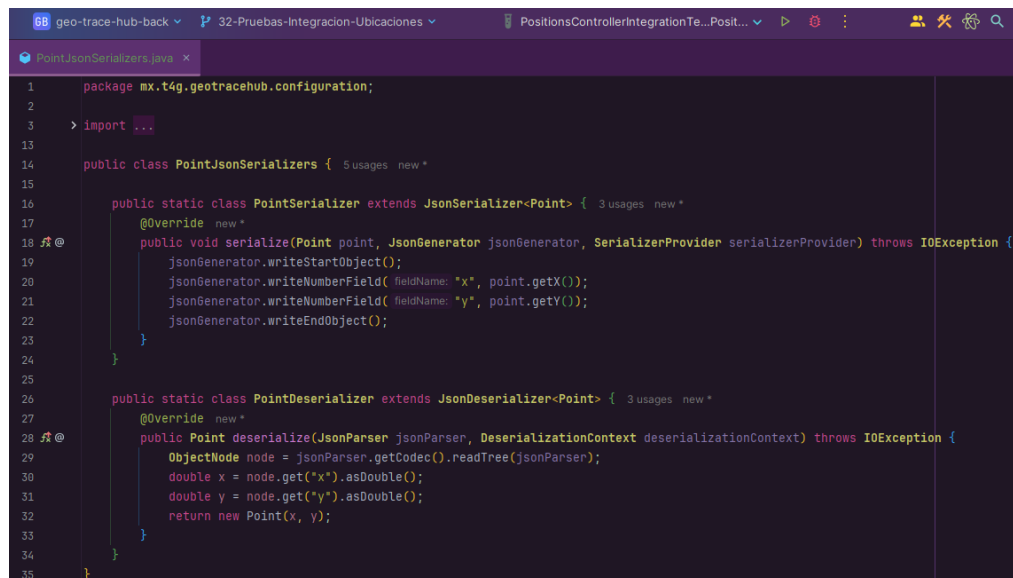
Creación de las clases JacksonConfig y PointJsonSerializers.



La clase “PointJsonSerializers” se crean dos clases estáticas las cuales van a permitir la serialización y deserialización de la ubicación de tipo POINT porque en las pruebas de integración Spring no sabe cómo convertir un objeto Point a una respuesta JSON, que es la salida que tendrán las pruebas de integración, el cual es descrito en la Figura 215:

Figura 215

Clase PointJsonSerializers.

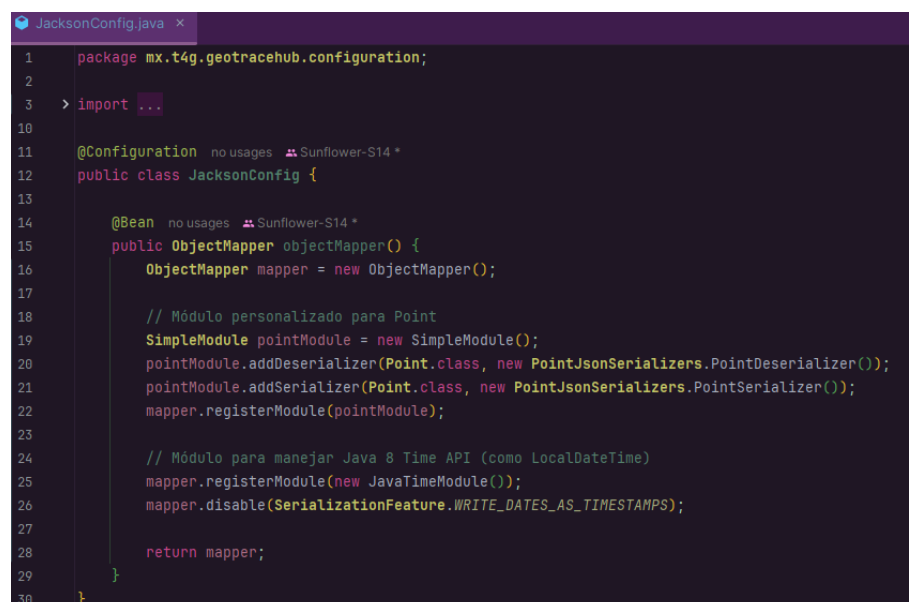


```
1 package mx.t4g.geotracehub.configuration;
2
3 > import ...
13
14 public class PointJsonSerializers { 5 usages new *
15
16     public static class PointSerializer extends JsonSerializer<Point> { 3 usages new *
17         @Override new *
18         public void serialize(Point point, JsonGenerator jsonGenerator, SerializerProvider serializerProvider) throws IOException {
19             jsonGenerator.writeStartObject();
20             jsonGenerator.writeNumberField("x", point.getX());
21             jsonGenerator.writeNumberField("y", point.getY());
22             jsonGenerator.writeEndObject();
23         }
24     }
25
26     public static class PointDeserializer extends JsonDeserializer<Point> { 3 usages new *
27         @Override new *
28         public Point deserialize(JsonParser jsonParser, DeserializationContext deserializationContext) throws IOException {
29             ObjectNode node = jsonParser.getCodec().readTree(jsonParser);
30             double x = node.get("x").asDouble();
31             double y = node.get("y").asDouble();
32             return new Point(x, y);
33         }
34     }
35 }
```

Luego en la clase “JacksonConfig” sirve para indicar el objeto Point tendrá un manejo especial para la serialización/deserialización JSON, ya que no es una clase estándar de Java que Jackson sepa cómo manejar automáticamente. Esta lógica puede revisarse en la Figura 216:

Figura 216

Clase JacksonConfig.

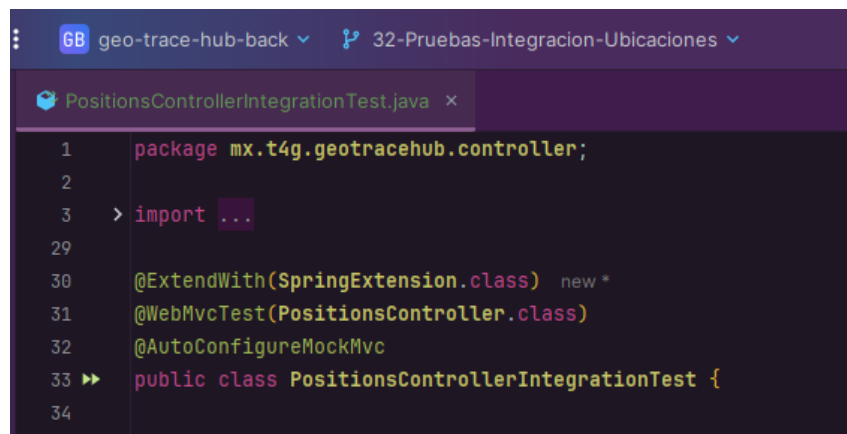


```
1 package mx.t4g.geotracehub.configuration;
2
3 > import ...
10
11 @Configuration no usages Sunflower-S14 *
12 public class JacksonConfig {
13
14     @Bean no usages Sunflower-S14 *
15     public ObjectMapper objectMapper() {
16         ObjectMapper mapper = new ObjectMapper();
17
18         // Módulo personalizado para Point
19         SimpleModule pointModule = new SimpleModule();
20         pointModule.addDeserializer(Point.class, new PointJsonSerializers.PointDeserializer());
21         pointModule.addSerializer(Point.class, new PointJsonSerializers.PointSerializer());
22         mapper.registerModule(pointModule);
23
24         // Módulo para manejar Java 8 Time API (como LocalDateTime)
25         mapper.registerModule(new JavaTimeModule());
26         mapper.disable(SerializationFeature.WRITE_DATES_AS_TIMESTAMPS);
27
28         return mapper;
29     }
30 }
```


La Figura 217 corresponde a la estructura del test de integración que utilizan las anotaciones `@ExtendWith` indica a la clase del test que se usará el framework de pruebas de Spring (`SpringExtension.class`). Luego con la anotación `@WebMvcTest` le indica al test que la clase sobre la cual se hará la prueba de integración en este caso es con “`PositionsController`”. Y la anotación `@AutoConfigureMockMvc` permite que se configure automáticamente el objeto `MockMvc` para la prueba de integración:

Figura 217

Anotaciones `@ExtendWith`, `@WebMvcTest` y `@AutoConfigureMockMvc` en la clase `PositionsControllerIntegrationTest`.

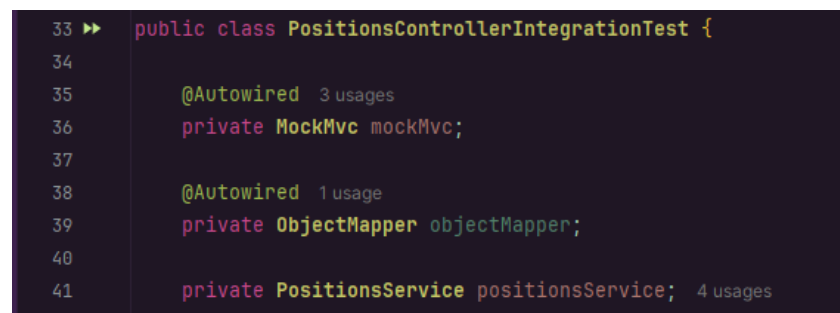


```
1 package mx.t4g.geotracehub.controller;
2
3 > import ...
29
30 @ExtendWith(SpringExtension.class) new *
31 @WebMvcTest(PositionsController.class)
32 @AutoConfigureMockMvc
33 >> public class PositionsControllerIntegrationTest {
34
```

Luego en la clase se declaran las variables de configuración con `MockMvc` que se usará para realizar solicitudes HTTP del controlador. Un `objectMapper` que convierte objetos a JSON. Se llama a “`positionsService`” que se mockea para simular el comportamiento del service de las ubicaciones, representado en la Figura 218:

Figura 218

Declaración de las variables con `MockMvc` en la clase del test.



```
33 >> public class PositionsControllerIntegrationTest {
34
35     @Autowired 3 usages
36     private MockMvc mockMvc;
37
38     @Autowired 1 usage
39     private ObjectMapper objectMapper;
40
41     private PositionsService positionsService; 4 usages
42
```

Se configura el método `setUp` en la Figura 219 en donde se mockea el servicio de dispositivos y se usa un objeto `MockMvc` para realizar solicitudes HTTP al controlador juntamente se incluye los controladores de excepciones globales para manejar las excepciones existentes en las peticiones y evitar errores en las respuestas:

Figura 219

Configuración del método `setUp` en la prueba de integración de ubicaciones.

```
33 public class PositionsControllerIntegrationTest {
43     @BeforeEach new *
44     void setUp() {
45         positionsService = Mockito.mock(PositionsService.class);
46         PositionsController controller = new PositionsController(positionsService);
47         mockMvc = MockMvcBuilders.standaloneSetup(controller)
48             .setControllerAdvice(new ValidationExceptionHandler())
49             .build();
50     }
}
```

El primer método “`getPositions_ShouldReturnPositionsList()`” consiste en obtener todas las ubicaciones de un dispositivo, en el que primero se simula la obtención de un dispositivo de ID con valor de 1. Luego se instancia un objeto “device” para preparar el dispositivo creado, luego se asocian dos ubicaciones al dispositivo guardándolos en un listado de posiciones. Se utiliza Mockito para simular la respuesta del servicio “PositionsService” cuando se llama al método “`getPointsByDevice()`”, de modo que devuelve la lista de posiciones previamente definida, se ilustra claramente en la Figura 220:

Figura 220

Método `getPositions_ShouldReturnPositionsList` del test de integración.

```
public class PositionsControllerIntegrationTest {
    @Test new *
    void getPositions_ShouldReturnPositionsList() throws Exception {
        // Given
        Integer deviceId = 1;

        Device device = Device.builder().id(deviceId).build();

        Position position1 = new Position();
        position1.setId(1);
        position1.setLocation(new Point( x: -99.1332, y: 19.4326)); // x=longitude, y=latitude
        position1.setDevice(device);
        position1.setDescription("Posición inicial");

        Position position2 = new Position();
        position2.setId(2);
        position2.setLocation(new Point( x: -99.1331, y: 19.4327));
        position2.setDevice(device);
        position2.setDescription("Posición secundaria");

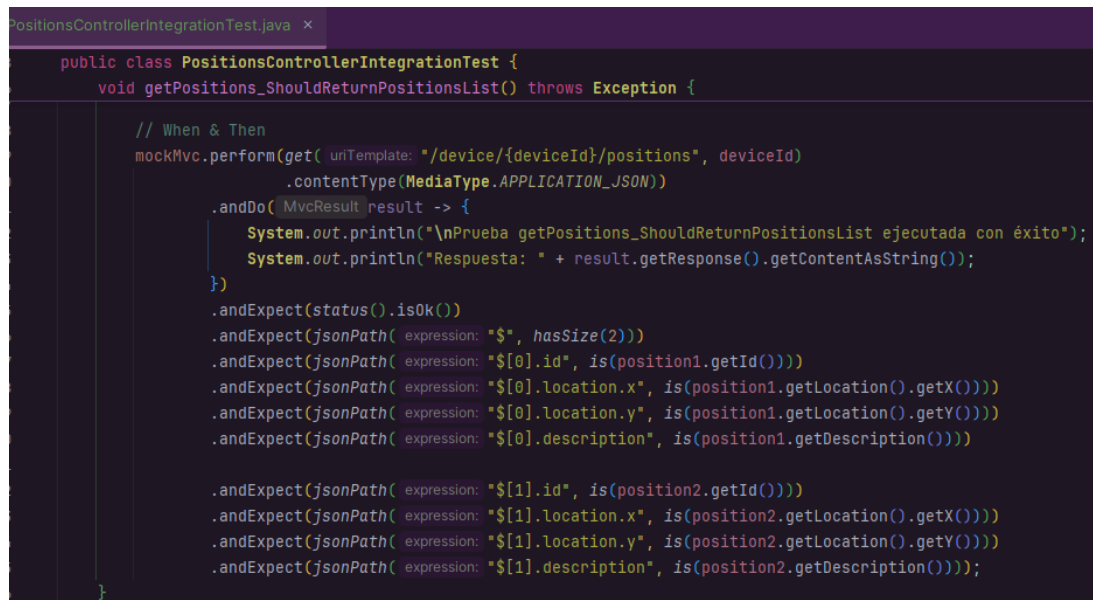
        List<Position> positions = Arrays.asList(position1, position2);

        Mockito.when(positionsService.getPointsByDevice(Mockito.any(Device.class))).thenReturn(positions);
    }
}
```

A continuación, se simula una solicitud *GET* al *endpoint* */device/{deviceId}/positions* utilizando *MockMvc* pasando el ID del dispositivo. Esto permite probar la funcionalidad del controlador “*PositionsController*”. Una vez que se ha simulado la solicitud, se verifica que la respuesta del controlador sea correcta con estado HTTP *200 OK* lo que indica que la solicitud se realizó correctamente, luego se verifica que la respuesta JSON contenga una lista de 2 elementos, que es el número de posiciones que se definieron en el escenario, mostrado en la Figura 221:

Figura 221

Configuración de la solicitud para `getPositions_ShouldReturnPositionsList`.



```
PositionsControllerIntegrationTest.java x
public class PositionsControllerIntegrationTest {
    void getPositions_ShouldReturnPositionsList() throws Exception {

        // When & Then
        mockMvc.perform(get(uriTemplate: "/device/{deviceId}/positions", deviceId)
            .contentType(MediaType.APPLICATION_JSON))
            .andExpect(status().isOk())
            .andExpect(jsonPath(expression: "$", hasSize(2)))
            .andExpect(jsonPath(expression: "$[0].id", is(position1.getId())))
            .andExpect(jsonPath(expression: "$[0].location.x", is(position1.getLocation().getX())))
            .andExpect(jsonPath(expression: "$[0].location.y", is(position1.getLocation().getY())))
            .andExpect(jsonPath(expression: "$[0].description", is(position1.getDescription())))

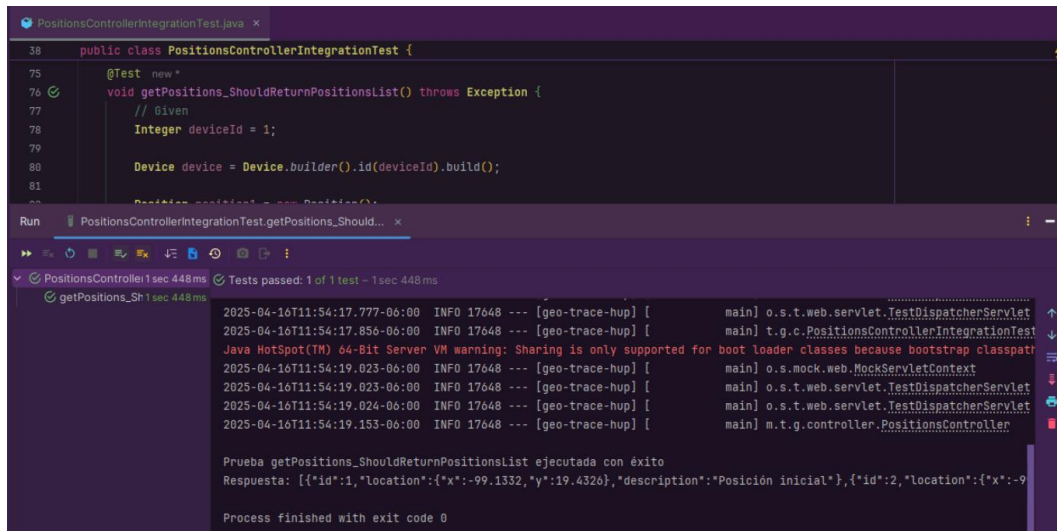
            .andExpect(jsonPath(expression: "$[1].id", is(position2.getId())))
            .andExpect(jsonPath(expression: "$[1].location.x", is(position2.getLocation().getX())))
            .andExpect(jsonPath(expression: "$[1].location.y", is(position2.getLocation().getY())))
            .andExpect(jsonPath(expression: "$[1].description", is(position2.getDescription())));
    }
}
```

En la Figura 222 se muestra el resultado de la ejecución se muestra en la figura en la que se muestra que el test se marcó de color verde y se imprimen los siguientes mensajes:

“Prueba `getPositions_ShouldReturnPositionsList` ejecutada con éxito.” “Respuesta: [{“id”:1, “location”:{“x”:-99.1332,“y”:19.4326}, “description”:“Posición inicial”}, {“id”:2, “location”:{“x”:-99.1331, “y”:19.4327}, “description”:“Posición secundaria”}]”

Figura 222

Ejecución del test del método `getPositions_ShouldReturnPositionsList`.



```
38 public class PositionsControllerIntegrationTest {
75
76     @Test new *
77     void getPositions_ShouldReturnPositionsList() throws Exception {
78         // Given
79         Integer deviceId = 1;
80
81         Device device = Device.builder().id(deviceId).build();
82
83         Position newPosition = new Position();
84         newPosition.setLocation(new Point(-99.1332, 19.4326));
85         newPosition.setDescription("Nueva posición");
86
87         // No asignamos el dispositivo aquí, ya que se manejará en el servicio
88
89         Position savedPosition = new Position();
90         savedPosition.setId(3);
91         savedPosition.setLocation(newPosition.getLocation());
92         savedPosition.setDevice(device);
93         savedPosition.setDescription(newPosition.getDescription());
94     }
95 }
```

Run PositionsControllerIntegrationTest.getPositions_Should... x

PositionsController 1 sec 448ms Tests passed: 1 of 1 test - 1 sec 448ms

getPositions_ShouldReturnPositionsList 1 sec 448ms

2025-04-16T11:54:17.777-06:00 INFO 17648 --- [geo-trace-hup] [main] o.s.t.web.servlet.TestDispatcherServlet

2025-04-16T11:54:17.856-06:00 INFO 17648 --- [geo-trace-hup] [main] t.g.c.PositionsControllerIntegrationTest

Java HotSpot(TM) 64-Bit Server VM warning: Sharing is only supported for boot loader classes because bootstrap classpath

2025-04-16T11:54:19.023-06:00 INFO 17648 --- [geo-trace-hup] [main] o.s.mock.web.MockServletContext

2025-04-16T11:54:19.023-06:00 INFO 17648 --- [geo-trace-hup] [main] o.s.t.web.servlet.TestDispatcherServlet

2025-04-16T11:54:19.024-06:00 INFO 17648 --- [geo-trace-hup] [main] o.s.t.web.servlet.TestDispatcherServlet

2025-04-16T11:54:19.153-06:00 INFO 17648 --- [geo-trace-hup] [main] m.t.g.controller.PositionsController

Prueba getPositions_ShouldReturnPositionsList ejecutada con éxito

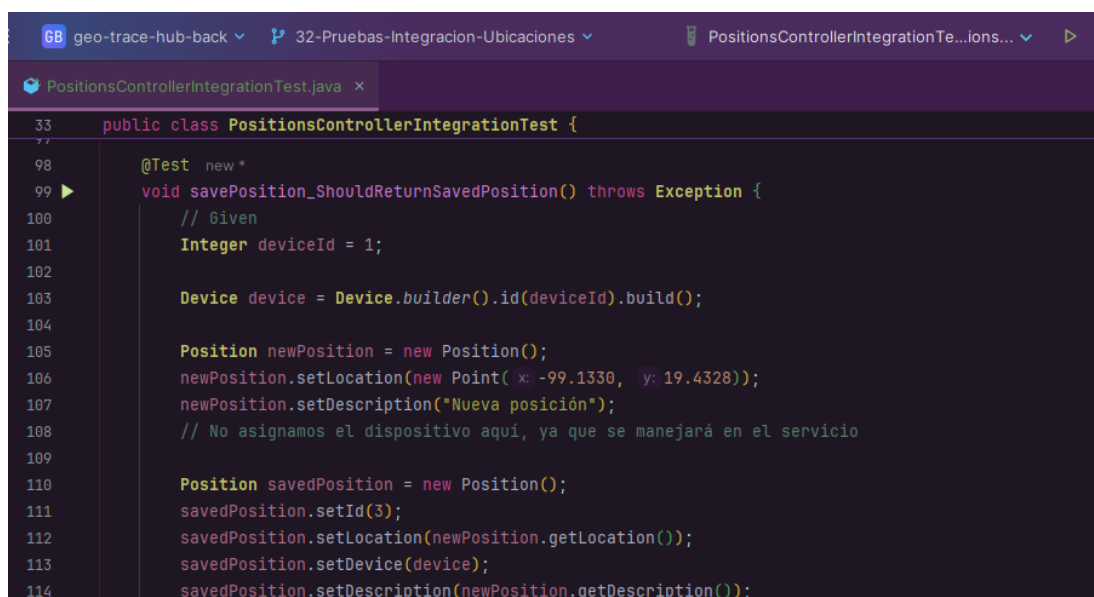
Respuesta: [{"id":1,"location":{"x":-99.1332,"y":19.4326},"description":"Posición inicial"}, {"id":2,"location":{"x":-99.1332,"y":19.4326},"description":"Posición inicial"}]

Process finished with exit code 0

En el segundo test “`savePosition_ShouldReturnSavedPosition()`” funciona de manera similar al anterior. Se prepara el escenario de prueba creando un dispositivo con un ID determinado y una nueva posición que se va a guardar, luego se utiliza Mockito para simular la respuesta del servicio “`PositionsService`”, se encuentra representado en la Figura 223:

Figura 223

Método `savePosition_ShouldReturnSavedPosition` del test de integración.



```
33 public class PositionsControllerIntegrationTest {
77
98     @Test new *
99     void savePosition_ShouldReturnSavedPosition() throws Exception {
100         // Given
101         Integer deviceId = 1;
102
103         Device device = Device.builder().id(deviceId).build();
104
105         Position newPosition = new Position();
106         newPosition.setLocation(new Point(-99.1332, 19.4326));
107         newPosition.setDescription("Nueva posición");
108         // No asignamos el dispositivo aquí, ya que se manejará en el servicio
109
110         Position savedPosition = new Position();
111         savedPosition.setId(3);
112         savedPosition.setLocation(newPosition.getLocation());
113         savedPosition.setDevice(device);
114         savedPosition.setDescription(newPosition.getDescription());
115     }
116 }
```

Cuando se llama al método “storePosition()”, de modo que devolverá la posición guardada con un ID asignado, se pasará a simular la solicitud POST utilizando MockMvc, pasando el ID del dispositivo y la nueva posición como cuerpo de la solicitud en formato JSON, esto permite probar la funcionalidad del controlador “PositionsController” para guardar nuevas posiciones, Una vez que se ha simulado la solicitud, se verifica que la respuesta sea correcta en términos de código de estado HTTP y contenido JSON, queda evidenciado en la Figura 224:

Figura 224

Configuración de la solicitud para savePosition_ShouldReturnSavedPosition.

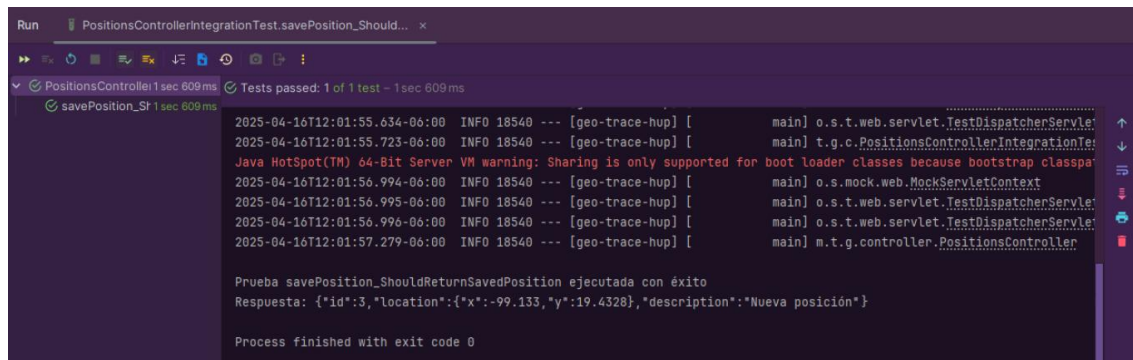
```
PositionsControllerIntegrationTest.java x
33  public class PositionsControllerIntegrationTest {
99  void savePosition_ShouldReturnSavedPosition() throws Exception {
116
117      Mockito.when(positionsService.storePosition(Mockito.eq(deviceId), Mockito.any(Position.class)))
118          .thenReturn(savedPosition);
119
120      // When & Then
121      mockMvc.perform(post( uriTemplate: "/device/{deviceId}/positions", deviceId)
122          .contentType(MediaType.APPLICATION_JSON)
123          .content(objectMapper.writeValueAsString(newPosition)))
124          .andExpect( MvcResult result -> {
125              System.out.println("\nPrueba savePosition_ShouldReturnSavedPosition ejecutada con éxito");
126              System.out.println("Respuesta: " + result.getResponse().getContentAsString());
127          })
128          .andExpect(status().isOk())
129          .andExpect(jsonPath( expression: "$.id", is(savedPosition.getId())))
130          .andExpect(jsonPath( expression: "$.location.x", is(savedPosition.getLocation().getX())))
131          .andExpect(jsonPath( expression: "$.location.y", is(savedPosition.getLocation().getY())))
132          .andExpect(jsonPath( expression: "$.description", is(savedPosition.getDescription())))
133  }
```

Al ejecutar el test, en la imagen se muestra el resultado del test como aprobatorio al marcarse color verde y una palomita del mismo color y se imprimieron los siguientes mensajes en la terminal, uno indicando que se ejecutó la prueba de éxito y el otro mostrando la respuesta en JSON, dicha operación se puede analizar en la Figura 225:

- “Prueba savePosition_ShouldReturnSavedPosition ejecutada con éxito”.
- “Respuesta: {\"id\":3,\"location\":{\"x\":-99.133,\"y\":19.4328},\"description\":\"Nueva posición\"}”

Figura 225

Ejecución del test del método savePosition_ShouldReturnSavedPosition.



```
Run PositionsControllerIntegrationTest.savePosition_Should... x
Tests passed: 1 of 1 test - 1 sec 609 ms
savePosition_ShouldReturnSavedPosition 1 sec 609 ms
2025-04-16T12:01:55.634-06:00 INFO 18540 --- [geo-trace-hup] [main] o.s.t.web.servlet.TestDispatcherServlet
2025-04-16T12:01:55.723-06:00 INFO 18540 --- [geo-trace-hup] [main] t.g.c.PositionsControllerIntegrationTest
Java HotSpot(TM) 64-Bit Server VM warning: Sharing is only supported for boot loader classes because bootstrap classpa
2025-04-16T12:01:56.994-06:00 INFO 18540 --- [geo-trace-hup] [main] o.s.mock.web.MockServletContext
2025-04-16T12:01:56.995-06:00 INFO 18540 --- [geo-trace-hup] [main] o.s.t.web.servlet.TestDispatcherServlet
2025-04-16T12:01:56.996-06:00 INFO 18540 --- [geo-trace-hup] [main] o.s.t.web.servlet.TestDispatcherServlet
2025-04-16T12:01:57.279-06:00 INFO 18540 --- [geo-trace-hup] [main] m.t.g.controller.PositionsController

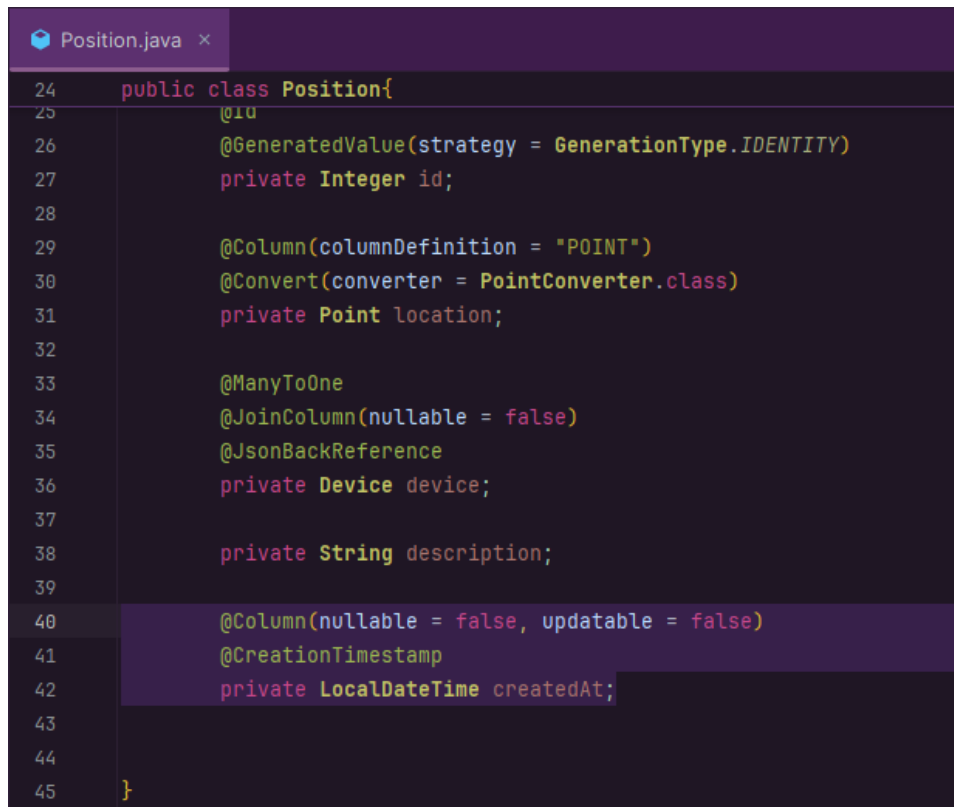
Prueba savePosition_ShouldReturnSavedPosition ejecutada con éxito
Respuesta: {"id":3,"location":{"x":-99.133,"y":19.4328},"description":"Nueva posición"}
Process finished with exit code 0
```

3.1.21. Filtrado de ubicaciones

Modelo o entidad: para el filtrado de ubicaciones se usará el modelo “Position” previamente creado se le agrega un nuevo atributo a la clase en el que la tabla positions tendría la tupla “createdAt” que no podrá ser nula; no se podrá actualizar este valor porque se va a guardar con la anotación `@CreationTimestamp` la fecha y la hora en la que se guarde en automático una posición. El atributo “createdAt” es de tipo `LocalDateTime`, lo cual se ejemplifica en código en la Figura 226:

Figura 226

Atributo createdAt en la clase Position.



```
24 public class Position{
25     @Id
26     @GeneratedValue(strategy = GenerationType.IDENTITY)
27     private Integer id;
28
29     @Column(columnDefinition = "POINT")
30     @Convert(converter = PointConverter.class)
31     private Point location;
32
33     @ManyToOne
34     @JoinColumn(nullable = false)
35     @JsonBackReference
36     private Device device;
37
38     private String description;
39
40     @Column(nullable = false, updatable = false)
41     @CreationTimestamp
42     private LocalDateTime createdAt;
43
44
45 }
```

Repository o Repositorio: como se muestra en la Figura 227 se utiliza el repositorio “PositionsRepository” en el que se creará una lista de posiciones con el nombre de “findByDeviceAndCreatedAtBetween” que permitirá buscar por dispositivo juntamente con la

hora en tiempo local de inicio así como la del fin en el que se crean la consulta SQL en el que buscará la position entre una fecha específica en un dispositivo:

Figura 227

Método `findByDeviceAndCreatedAtBetween` en la clase `PositionsRepository`.

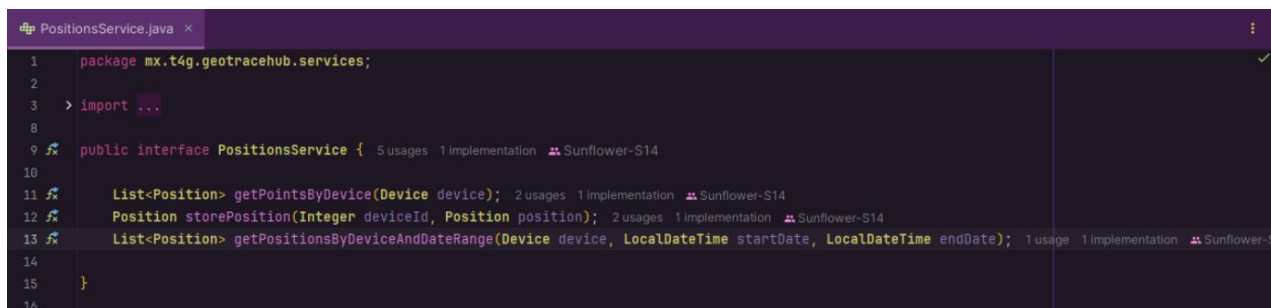


```
1 package mx.t4g.geotracehub.repositories;
2
3 > import ...
10
11 @Repository 4 usages Sunflower-S14
12 public interface PositionsRepository extends JpaRepository<Position, Integer> {
13
14     List<Position> getAllByDevice(Device device); 3 usages Sunflower-S14
15     List<Position> findByDeviceAndCreatedAtBetween(Device device, LocalDateTime startDate, LocalDateTime endDate); 1 usage Sunflower-S14
16
17 }
18
```

Service o servicio: se creó una lista de posiciones de las cuales con “getPositionsByDeviceAndDateRange” llama a la lógica del negocio creada en el service implements para realizar el filtrado, puede visualizarse en la Figura 228:

Figura 228

Método `getPositionsByDeviceAndDateRange` en la clase `PositionsService`.

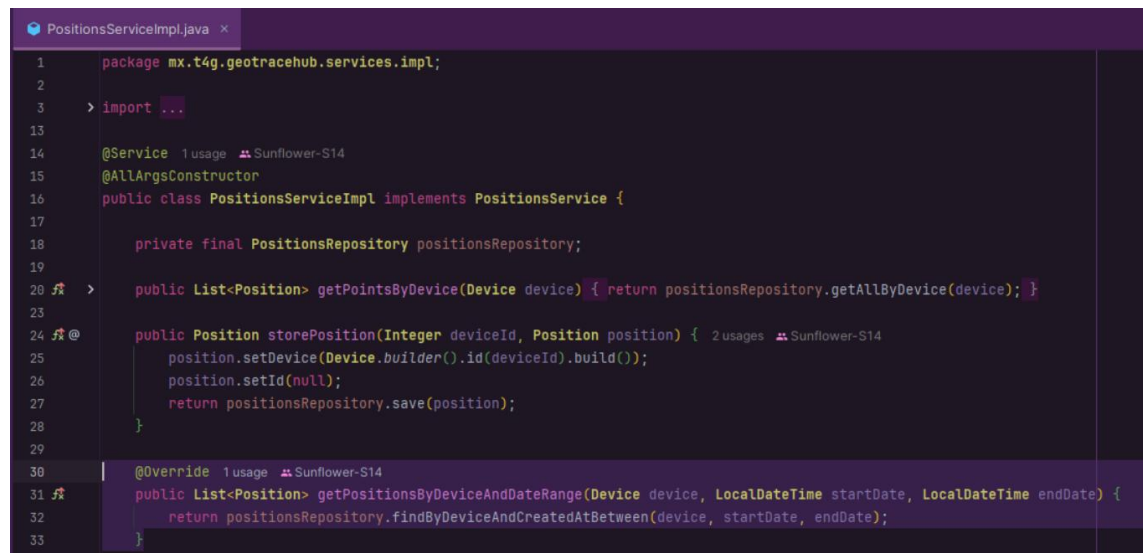


```
1 package mx.t4g.geotracehub.services;
2
3 > import ...
8
9 public interface PositionsService { 5 usages 1 implementation Sunflower-S14
10
11     List<Position> getPointsByDevice(Device device); 2 usages 1 implementation Sunflower-S14
12     Position storePosition(Integer deviceId, Position position); 2 usages 1 implementation Sunflower-S14
13     List<Position> getPositionsByDeviceAndDateRange(Device device, LocalDateTime startDate, LocalDateTime endDate); 1 usage 1 implementation Sunflower-S14
14
15 }
16
```

Servicio implementado o Service Implements: a continuación en la Figura 229 se aplica la lógica del negocio en la clase PositionsServiceImpl, en la que llama al repositorio para buscar la posición de un dispositivo con la fecha de inicio y fin:

Figura 229

Método `getPositionsByDeviceAndDateRange` en `PositionsServiceImpl`.

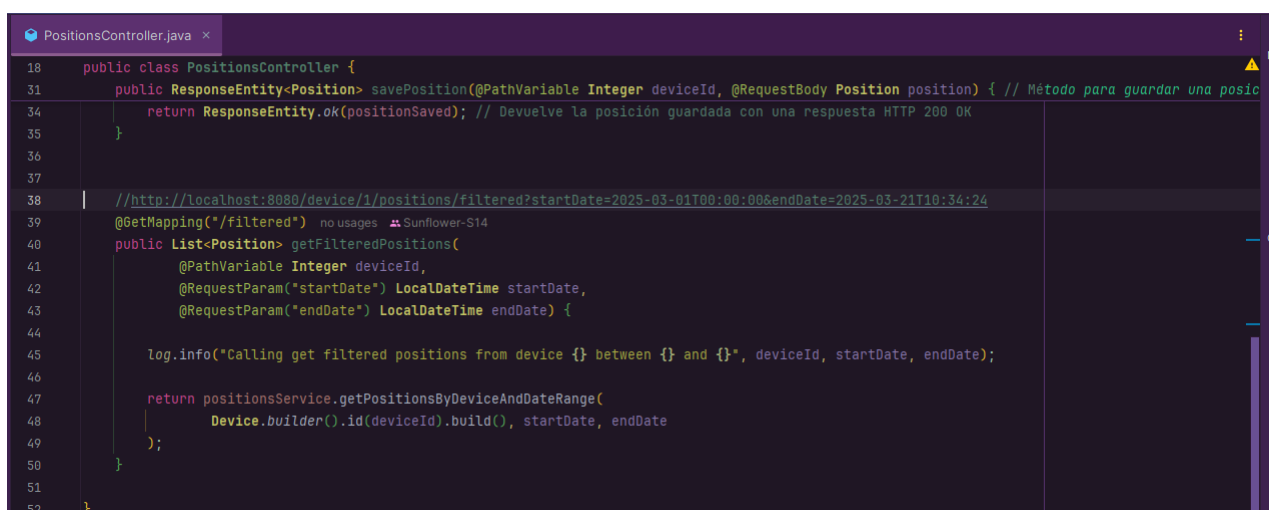


```
1 package mx.t4g.geotracehub.services.impl;
2
3 > import ...
13
14 @Service 1 usage Sunflower-S14
15 @AllArgsConstructor
16 public class PositionsServiceImpl implements PositionsService {
17
18     private final PositionsRepository positionsRepository;
19
20     > public List<Position> getPointsByDevice(Device device) { return positionsRepository.getAllByDevice(device); }
23
24     > public Position storePosition(Integer deviceId, Position position) { 2 usages Sunflower-S14
25         position.setDevice(Device.builder().id(deviceId).build());
26         position.setId(null);
27         return positionsRepository.save(position);
28     }
29
30 @Override 1 usage Sunflower-S14
31 > public List<Position> getPositionsByDeviceAndDateRange(Device device, LocalDateTime startDate, LocalDateTime endDate) {
32     return positionsRepository.findByDeviceAndCreatedAtBetween(device, startDate, endDate);
33 }
```

Controlador REST o Controller REST: se crea en “PositionsController” el endpoint que servirá para llamar al filtrado de ubicaciones con el método “getFilteredPositions” que establece con `@GetMapping` el endpoint “/filtered”, el cual recibirá el ID del dispositivo, la fecha de inicio y fecha de fin puede verse en la Figura 230:

Figura 230

Se agrega el método `getFilterPositions` en la clase `PositionsController`.



```
18 public class PositionsController {
31     public ResponseEntity<Position> savePosition(@PathVariable Integer deviceId, @RequestBody Position position) { // Método para guardar una posic
34         return ResponseEntity.ok(positionSaved); // Devuelve la posición guardada con una respuesta HTTP 200 OK
35     }
36
37
38 //http://localhost:8080/device/1/positions/filtered?startDate=2025-03-01T00:00:00&endDate=2025-03-21T10:34:24
39 @GetMapping("/filtered") no usages Sunflower-S14
40 public List<Position> getFilteredPositions(
41     @PathVariable Integer deviceId,
42     @RequestParam("startDate") LocalDateTime startDate,
43     @RequestParam("endDate") LocalDateTime endDate) {
44
45     log.info("Calling get filtered positions from device {} between {} and {}", deviceId, startDate, endDate);
46
47     return positionsService.getPositionsByDeviceAndDateRange(
48         Device.builder().id(deviceId).build(), startDate, endDate
49     );
50 }
51
52 }
```

3.1.22. Pruebas y correcciones

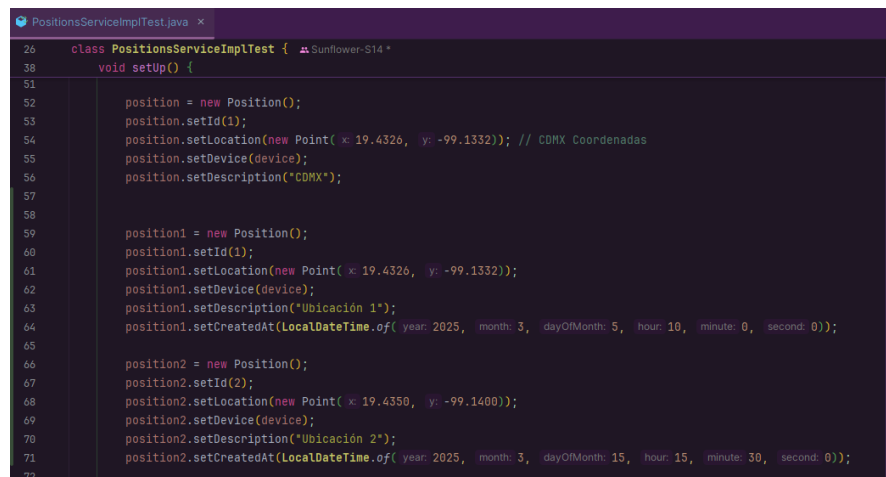
Pruebas unitarias del filtrado.

Metodología: usar Mockito y Junit en la clase “PositionsServiceImplTest” para realizar la prueba unitaria del método de filtrado de ubicaciones.

En el método setUp de la clase PositionsServiceImplTest se crean dos dispositivos para la prueba unitaria:

Figura 231

Creación de dos dispositivos para la PositionsServiceImplTest

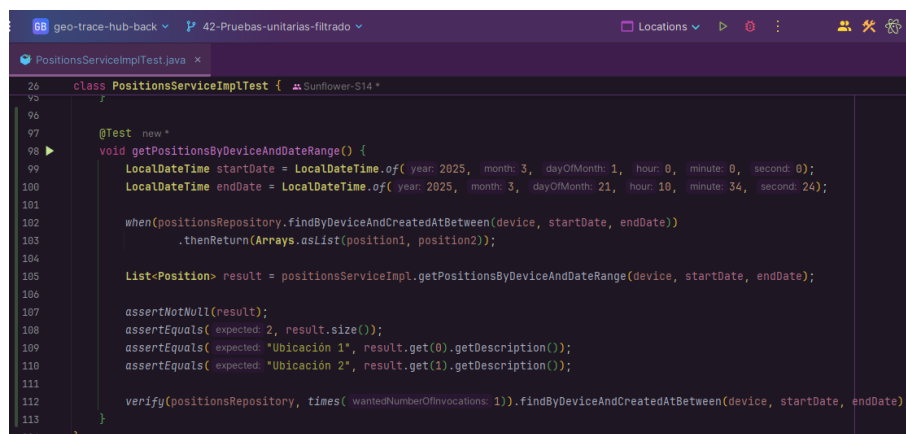


```
26 class PositionsServiceImplTest {
38     void setUp() {
51         position = new Position();
52         position.setId(1);
53         position.setLocation(new Point(19.4326, -99.1332)); // CDMX Coordenadas
54         position.setDevice(device);
55         position.setDescription("CDMX");
56
57
58
59         position1 = new Position();
60         position1.setId(1);
61         position1.setLocation(new Point(19.4326, -99.1332));
62         position1.setDevice(device);
63         position1.setDescription("Ubicación 1");
64         position1.setCreatedAt(LocalDate.of(2025, 3, 5, 10, 0, 0));
65
66         position2 = new Position();
67         position2.setId(2);
68         position2.setLocation(new Point(19.4350, -99.1400));
69         position2.setDevice(device);
70         position2.setDescription("Ubicación 2");
71         position2.setCreatedAt(LocalDate.of(2025, 3, 15, 15, 30, 0));
72     }
}
```

En la Figura 232 se crea el método “getPositionsByDeviceAndDateRange:

Figura 232

Método getPositionsByDeviceAndDateRange del test unitario.



```
26 class PositionsServiceImplTest {
96
97     @Test
98     void getPositionsByDeviceAndDateRange() {
99         LocalDateTime startDate = LocalDate.of(2025, 3, 1, 0, 0, 0);
100         LocalDateTime endDate = LocalDate.of(2025, 3, 21, 10, 34, 24);
101
102         when(positionRepository.findByDeviceAndCreatedAtBetween(device, startDate, endDate))
103             .thenReturn(Arrays.asList(position1, position2));
104
105         List<Position> result = positionsServiceImpl.getPositionsByDeviceAndDateRange(device, startDate, endDate);
106
107         assertNotNull(result);
108         assertEquals(2, result.size());
109         assertEquals("Ubicación 1", result.get(0).getDescription());
110         assertEquals("Ubicación 2", result.get(1).getDescription());
111
112         verify(positionRepository, times(1)).findByDeviceAndCreatedAtBetween(device, startDate, endDate);
113     }
114 }
```

Antes de la ejecución de la prueba, se modifica el método agregando un *log* que nos permite visualizar un mensaje en las pruebas unitarias, queda reflejado en la Figura 233:

Figura 233

Método `getPositionsByDeviceAndDateRange` se añade un `info.log`.

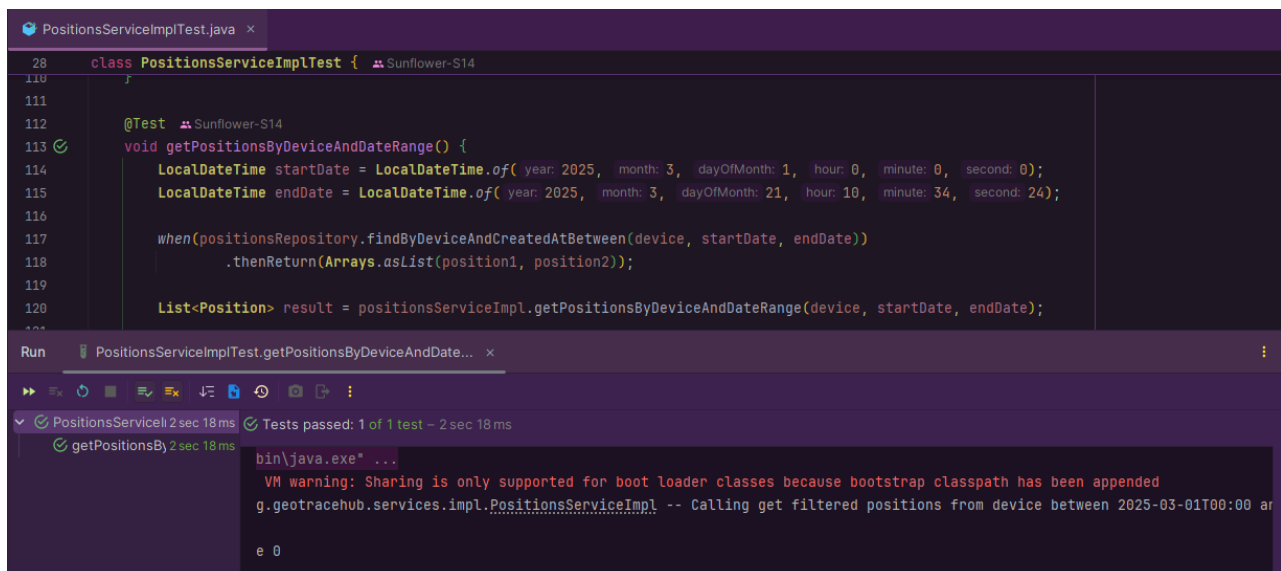
```
@Override 2 usages Sunflower-S14 *
public List<Position> getPositionsByDeviceAndDateRange(Device device, LocalDateTime startDate, LocalDateTime endDate) {
    log.info("Calling get filtered positions from device between {} and {}", startDate, endDate);
    return positionsRepository.findByDeviceAndCreatedAtBetween(device, startDate, endDate);
}
```

Al ejecutar la prueba unitaria se marcó en verde con un a palomita e imprimió el log declarado en el Service:

“Calling get filtered positions from device between 2025-03-01T00:00 and 2025-03-21T10:34:24” indicando que se ejecutó el método adecuadamente, evidenciado en la Figura 234:

Figura 234

Ejecución del test del método `getPositionsByDeviceAndDateRange`.



```
PositionsServiceImplTest.java x
28 class PositionsServiceImplTest { Sunflower-S14
110
111
112 @Test Sunflower-S14
113 void getPositionsByDeviceAndDateRange() {
114     LocalDateTime startDate = LocalDateTime.of( year: 2025, month: 3, dayOfMonth: 1, hour: 0, minute: 0, second: 0);
115     LocalDateTime endDate = LocalDateTime.of( year: 2025, month: 3, dayOfMonth: 21, hour: 10, minute: 34, second: 24);
116
117     when(positionsRepository.findByDeviceAndCreatedAtBetween(device, startDate, endDate))
118         .thenReturn(Arrays.asList(position1, position2));
119
120     List<Position> result = positionsServiceImpl.getPositionsByDeviceAndDateRange(device, startDate, endDate);
121
Run PositionsServiceImplTest.getPositionsByDeviceAndDate... x
Tests passed: 1 of 1 test - 2 sec 18 ms
getPositionsByDeviceAndDateRange 2 sec 18 ms
bin\java.exe ...
VM warning: Sharing is only supported for boot loader classes because bootstrap classpath has been appended
g.geotracehub.services.impl.PositionsServiceImpl -- Calling get filtered positions from device between 2025-03-01T00:00 ar
e 0
```

Pruebas de integración del filtrado.

Metodología: uso de Mockito y MockMvc junto con la dependencia jackson-datatype-jsr310 para manejar datos LocalDateTime en la clase “PositionControllerIntegrationTest”.

En la Figura 235 se muestra al pom.xml que se añade una dependencia que permite manejar la fecha y la hora como lo es un dato de tipo LocalDateTime para la prueba de integración:

Figura 235

Dependencia jackson.datatype-jsr310 en el pom.xml.

```
85 <dependency>
86   <groupId>com.fasterxml.jackson.datatype</groupId>
87   <artifactId>jackson-datatype-jsr310</artifactId>
88   <version>2.15.3</version>
89 </dependency>
```

El método setUp() se modificó para registrar el módulo *JavaTimeModule* para que Jackson pueda serializar y deserializar correctamente objetos LocalDateTime, se visualiza en la Figura 236:

Figura 236

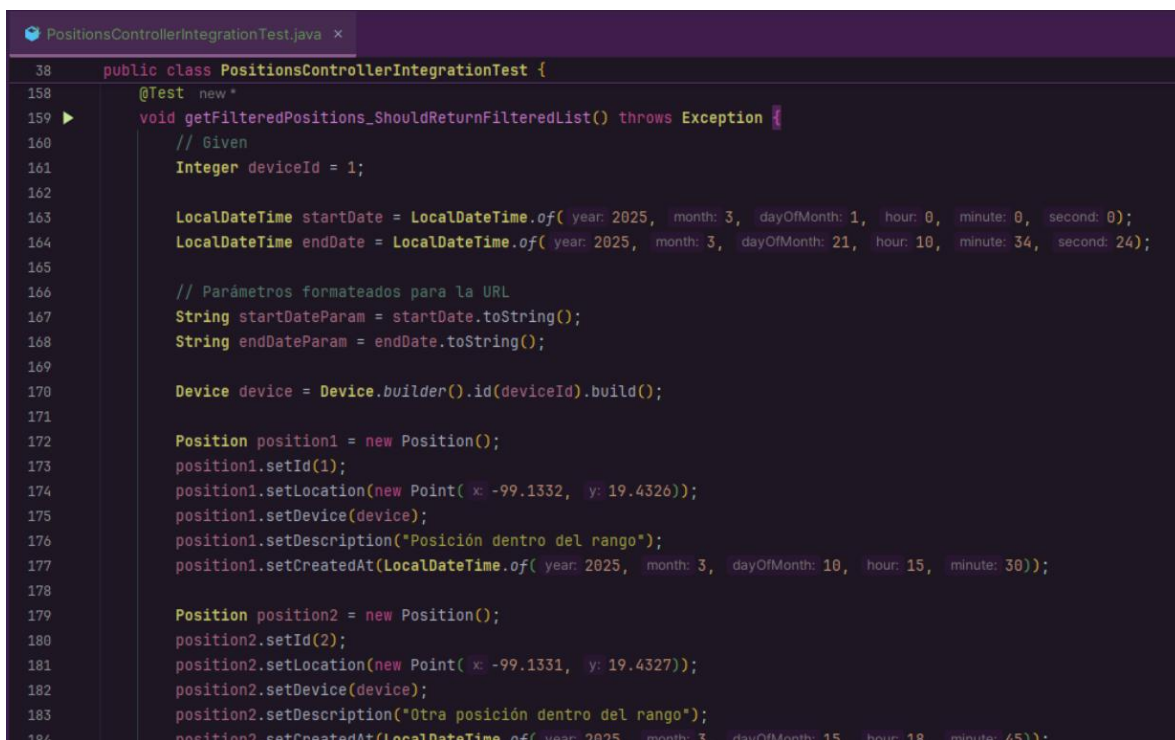
Configuración del método setUp en la clase PositionControllerIntegrationTest.

```
38 public class PositionControllerIntegrationTest {
39
40     @BeforeEach
41     void setUp() {
42         positionsService = Mockito.mock(PositionsService.class);
43         PositionsController controller = new PositionsController(positionsService);
44
45         ObjectMapper customObjectMapper = new ObjectMapper();
46
47         // Registrar módulo para LocalDateTime
48         customObjectMapper.registerModule(new JavaTimeModule());
49
50         // Configurar la serialización de fechas
51         customObjectMapper.configure(SerializationFeature.WRITE_DATES_AS_TIMESTAMPS, false);
52
53         // Registrar el módulo para Point
54         SimpleModule pointModule = new SimpleModule();
55         pointModule.addSerializer(Point.class, new PointJsonSerializers.PointSerializer());
56         pointModule.addDeserializer(Point.class, new PointJsonSerializers.PointDeserializer());
57         customObjectMapper.registerModule(pointModule);
58
59         mockMvc = MockMvcBuilders.standaloneSetup(controller)
60             .setControllerAdvice(new ValidationExceptionHandler())
61             .setMessageConverters(new MappingJackson2HttpMessageConverter(customObjectMapper))
62             .build();
63
64         this.objectMapper = customObjectMapper;
65     }
66 }
```

En la Figura 237 se muestra la creación del test “getFilteredPositions_ShouldReturnFilteredList()” que permitirá verificar que devuelva una lista filtrada de posiciones para un dispositivo determinado dentro de un rango de fechas específico. Primero se define un dispositivo con un ID y se establece un rango de fechas (startDate y endDate) para filtrar las posiciones para posteriormente crear dos posiciones (position1 y position2) que se encuentran dentro del rango de fechas definido:

Figura 237

Método getFilteredPositions_ShouldReturnFilteredList del test de integración.



```
38 public class PositionsControllerIntegrationTest {
158     @Test new *
159     void getFilteredPositions_ShouldReturnFilteredList() throws Exception {
160         // Given
161         Integer deviceId = 1;
162
163         LocalDateTime startDate = LocalDateTime.of( year: 2025, month: 3, dayOfMonth: 1, hour: 0, minute: 0, second: 0);
164         LocalDateTime endDate = LocalDateTime.of( year: 2025, month: 3, dayOfMonth: 21, hour: 10, minute: 34, second: 24);
165
166         // Parámetros formateados para la URL
167         String startDateParam = startDate.toString();
168         String endDateParam = endDate.toString();
169
170         Device device = Device.builder().id(deviceId).build();
171
172         Position position1 = new Position();
173         position1.setId(1);
174         position1.setLocation(new Point( x: -99.1332, y: 19.4326));
175         position1.setDevice(device);
176         position1.setDescription("Posición dentro del rango");
177         position1.setCreatedAt(LocalDateTime.of( year: 2025, month: 3, dayOfMonth: 10, hour: 15, minute: 30));
178
179         Position position2 = new Position();
180         position2.setId(2);
181         position2.setLocation(new Point( x: -99.1331, y: 19.4327));
182         position2.setDevice(device);
183         position2.setDescription("Otra posición dentro del rango");
184         position2.setCreatedAt(LocalDateTime.of( year: 2025, month: 3, dayOfMonth: 15, hour: 18, minute: 45));
```

Las posiciones se guardarán en una lista y con Mockito simula la respuesta del servicio “PositionsService”. Cuando se llama al método “getPositionsByDeviceAndDateRange()”, devuelve la lista de posiciones filtradas, puede analizarse en la Figura 238 la estructura codificada del guardo de ubicaciones:

Figura 238

Se configura el guardado de las ubicaciones.

```
PositionsControllerIntegrationTest.java x
38 public class PositionsControllerIntegrationTest {
159 void getFilteredPositions_ShouldReturnFilteredList() throws Exception {
185
186     List<Position> filteredPositions = Arrays.asList(position1, position2);
187
188     Mockito.when(positionsService.getPositionsByDeviceAndDateRange(
189         Mockito.any(Device.class),
190         Mockito.eq(startDate),
191         Mockito.eq(endDate)
192     )).thenReturn(filteredPositions);
```

Posteriormente con MockMvc se simula una solicitud GET a `/device/{deviceId}/positions/filtered` pasando el ID del dispositivo los parámetros `startDate` y `endDate` como parámetros de consulta para devolver que la respuesta sea correcta. Este flujo puede observarse en la Figura 239:

Figura 239

Configuración de la solicitud en getFilteredPositions_ShouldReturnFilteredList.

```
PositionsControllerIntegrationTest.java x
38 public class PositionsControllerIntegrationTest {
159 void getFilteredPositions_ShouldReturnFilteredList() throws Exception {
193
194     // When & Then
195     mockMvc.perform(get( uriTemplate: "/device/{deviceId}/positions/filtered", deviceId)
196         .param( name: "startDate", startDateParam)
197         .param( name: "endDate", endDateParam)
198         .contentType(MediaType.APPLICATION_JSON))
199         .andExpect(status().isOk())
200         .andExpect(jsonPath( expression: "$", hasSize(2)))
201         .andExpect(jsonPath( expression: "$[0].id", is(position1.getId())))
202         .andExpect(jsonPath( expression: "$[0].location.x", is(position1.getLocation().getX())))
203         .andExpect(jsonPath( expression: "$[0].location.y", is(position1.getLocation().getY())))
204         .andExpect(jsonPath( expression: "$[0].description", is(position1.getDescription())))
205         .andExpect(jsonPath( expression: "$[0].createdAt", exists()))
206         .andExpect(jsonPath( expression: "$[1].id", is(position2.getId())))
207         .andExpect(jsonPath( expression: "$[1].location.x", is(position2.getLocation().getX())))
208         .andExpect(jsonPath( expression: "$[1].location.y", is(position2.getLocation().getY())))
209         .andExpect(jsonPath( expression: "$[1].description", is(position2.getDescription())))
210         .andExpect(jsonPath( expression: "$[1].createdAt", exists()));
211 }
212 }
```

La Figura 240 muestra como devuelve en el resultado del test que se muestre de color verde y en la terminal imprime los siguientes mensajes, como respuesta JSON de las posiciones filtradas:

- “Prueba getFilteredPositions_ShouldReturnFilteredList ejecutada con éxito.”
- “Respuesta: [{\"id\":1, \"location\":{\"x\":-99.1332, \"y\":19.4326}, \"description\":\"Posición dentro del rango\", \"createdAt\":\"2025-03-10T15:30:00\"}, {\"id\":2,\"location\":{\"x\":-99.1331, \"y\":19.4327}, \"description\":\"Otra posición dentro del rango\", \"createdAt\":\"2025-03-15T18:45:00\"}]”

Figura 240

Ejecución del test del método getFilteredPositions_ShouldReturnFilteredList.

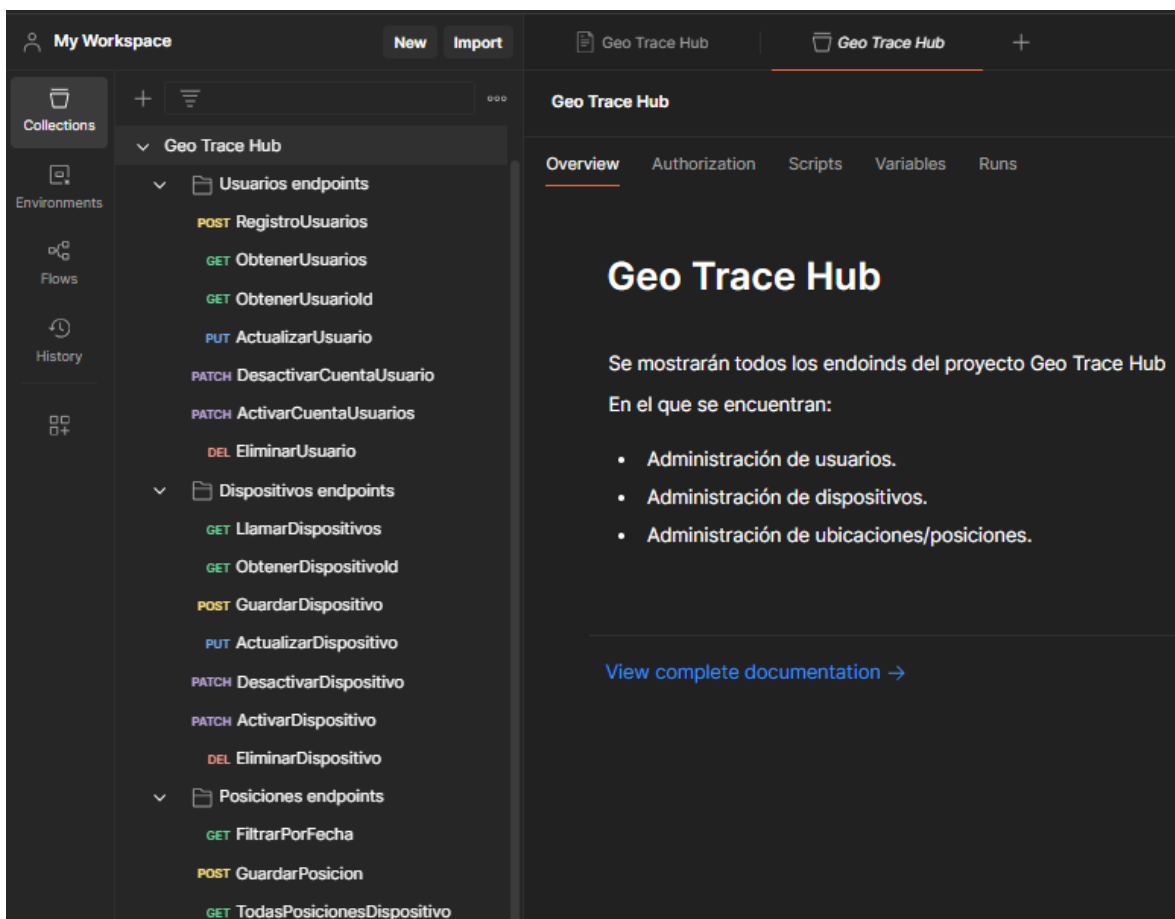
Pruebas de aceptación en el *backend*

Metodología: establecer criterios de aceptación que permitirán evaluar la respuesta de los *endpoints* para posteriormente probar la colección de *endpoints* mediante *POSTMAN* y si la respuesta no es la esperada aplicar correcciones para que cumplan los criterios de aceptación propuestos.

Para realizar las pruebas de aceptación en el *backend* se realizó una colección en *POSTMAN* de los *endpoints* creados en el proyecto como se muestran en la Figura 241:

Figura 241

Colección de POSTMAN de los endpoints creados.



Administración de usuarios

El primer endpoint es sobre mostrar todos los usuarios, representado de la Tabla 3:

Tabla 3

Estructura de la petición "Listar usuarios"

Tipo de petición	Endpoint a probar
GET	http://localhost:8080/users

Criterios de aceptación:

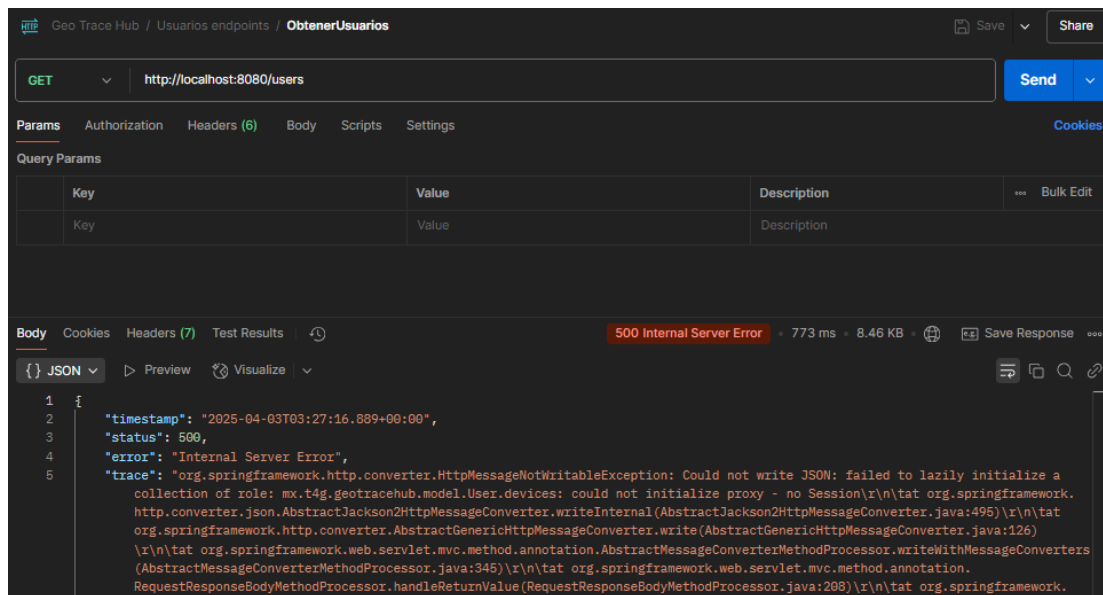
1. Se debe devolver una lista de todos los usuarios registrados.
 - 1.1. La respuesta debe incluir id, nombre, nombre de usuario, correo, contraseña, y el campo activo.
2. Si no hay usuarios registrados, debe devolver una lista vacía con *200 OK*.

En la Figura 242 se visualiza el comportamiento del criterio 1.1 en el endpoint, se muestra un mensaje de error el cual es: *"Could not write JSON: failed to lazily initialize a collection of role: mx.t4g.geotracehub.model.User.devices: could not initialize proxy - no Session"*.

Este mensaje significa que la referencia entre los modelos User y Device no se ha cargado completamente el objeto de devices en user y viceversa en Devices no se ha cargado el objeto user:

Figura 242

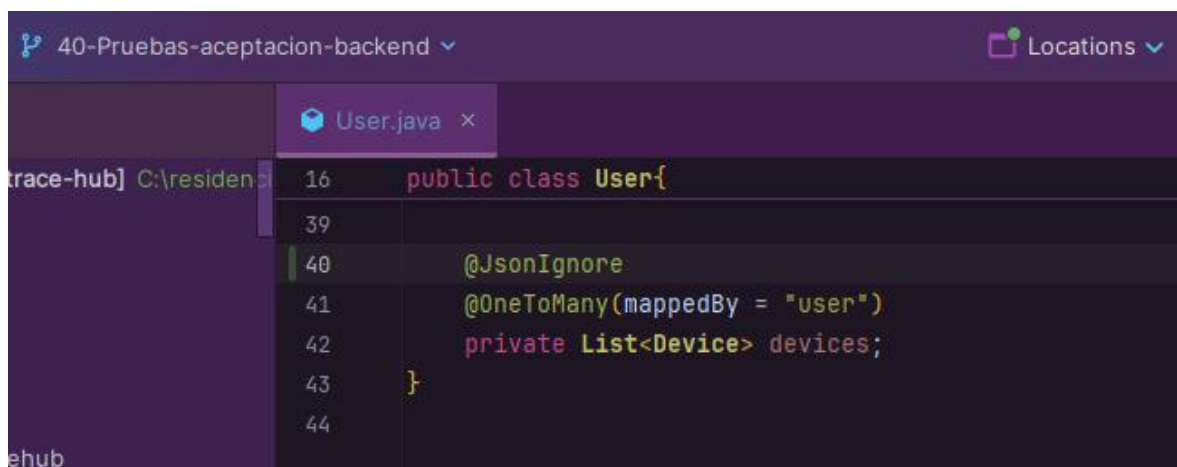
Respuesta errónea del criterio 1.1 de listar los dispositivos.



Para arreglar este error en la clase User se agregará la anotación `@JsonIgnore` que permitirá ignorar la propiedad devices al serializar el objeto "User", sin embargo, esto significa que no se podrá acceder a la lista de dispositivos desde *POSTMAN* en este endpoint, pero es innecesario ver los dispositivos, ya que solo se ven las cuentas de usuario creadas en el sistema, puede observarse en la Figura 243:

Figura 243

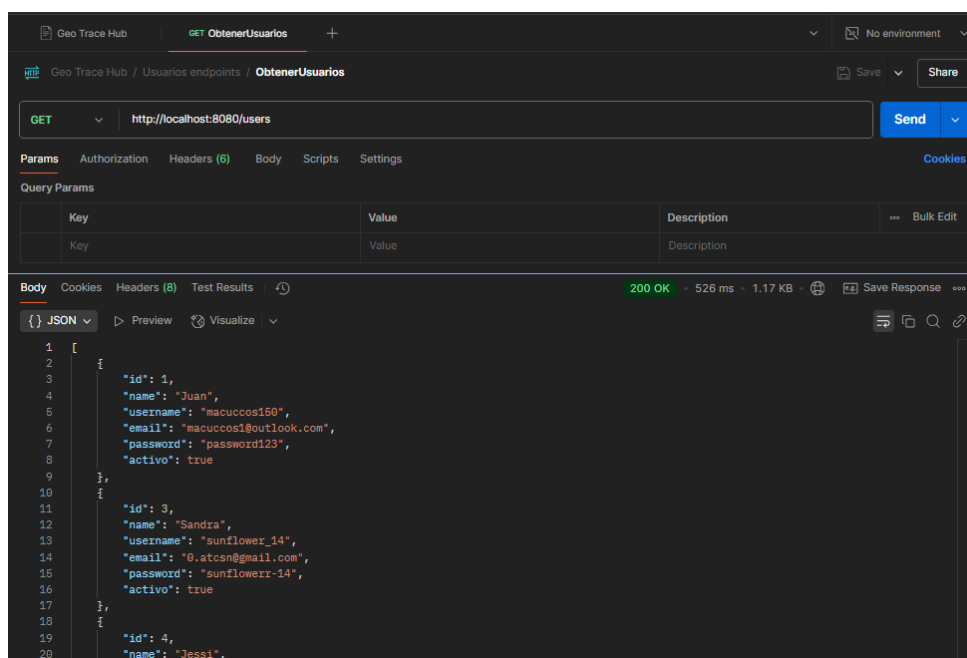
Se añade la anotación `@JsonIgnore` en la clase User.



En la Figura 244 se presenta que al ejecutar el endpoint nuevamente se muestra un JSON con todos los usuarios en el sistema así satisfaciendo al criterio 1.1:

Figura 244

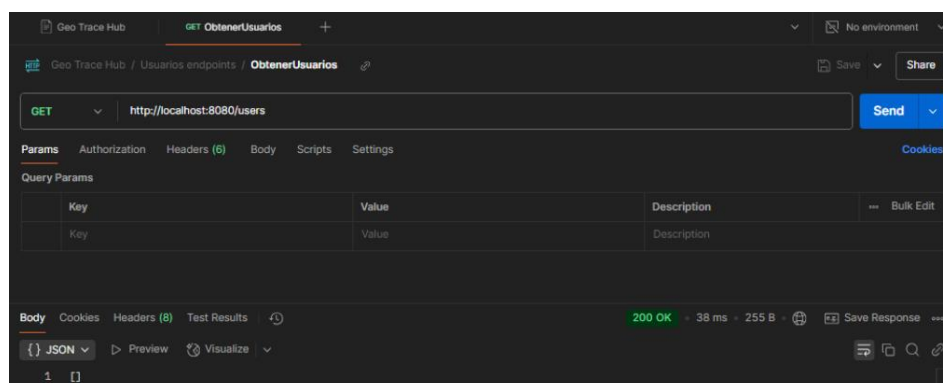
Respuesta 200 OK del criterio 1.1 de listar los dispositivos.



El resultado del criterio 2 se cumple de manera adecuada debido a que si no existen usuarios registrados devolverá una respuesta `200 OK` mostrando una lista vacía; el comportamiento esperado se confirma en la Figura 245:

Figura 245

Resultado 200 OK del criterio 2 de listar los dispositivos.



El segundo endpoint es obtener a un usuario por su ID, descrito en la Tabla 4:

Tabla 4

Estructura de la petición “ver usuario mediante su ID”.

Tipo de petición	Endpoint a probar
GET	http://localhost:8080/users/{id}

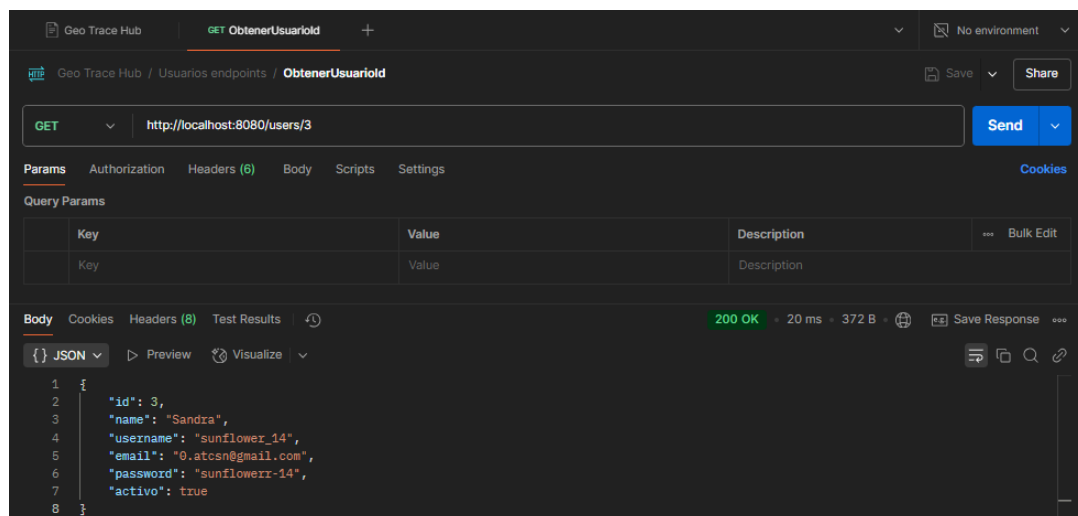
Criterios de aceptación:

1. Se debe devolver la información de un usuario específico.
 - 1.1. La respuesta debe incluir id, nombre, nombre de usuario, contraseña, correo, campo activo.
2. Si el usuario no existe, debe devolver un *404 Not Found* y un mensaje indicando que no existe el usuario.

La Figura 246 corresponde al resultado en *POSTMAN* del criterio 1.1, que cumple satisfactoriamente buscando a un usuario con ID = 3 y responde una respuesta JSON con los datos del usuario:

Figura 246

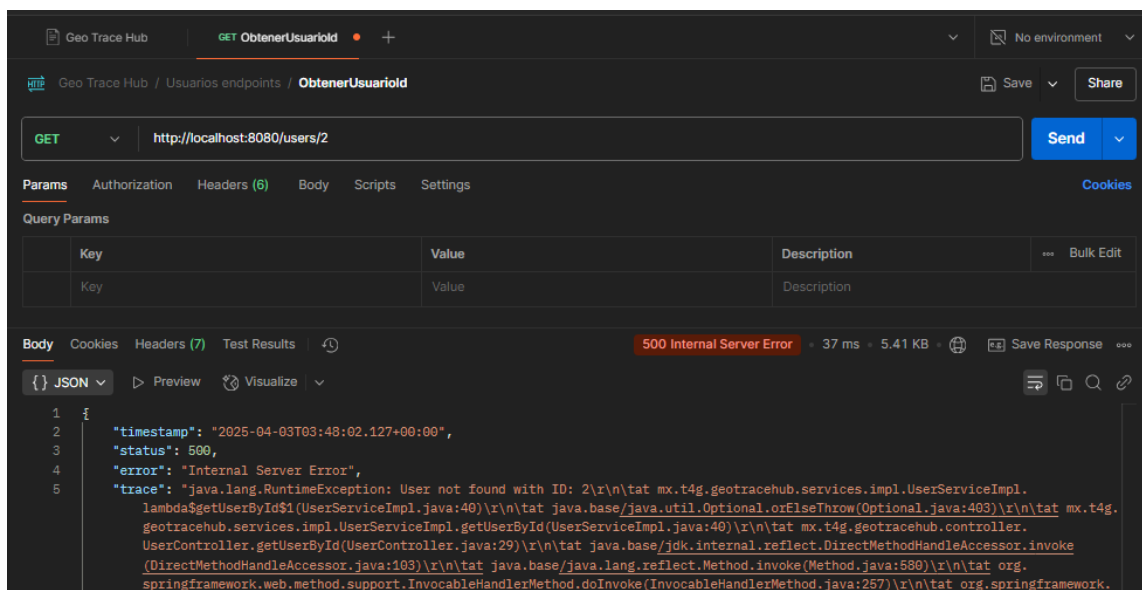
Resultado criterio 1.1 de ver usuario mediante su ID.



El resultado de probar el criterio de aceptación 2 se muestra en la Figura 247 mediante un error *500 Internal Server Error* y se esperaba no un error *404 Bad Request*:

Figura 247

Resultado erróneo del criterio 2 de ver usuario mediante su ID.



En la Figura 248 se crea una excepción personalizada que captura cuando no se encuentre al usuario indicado:

Figura 248

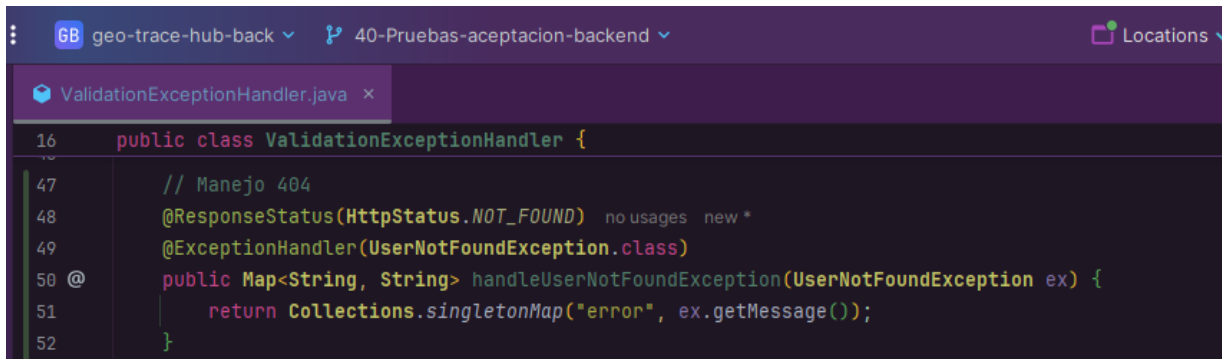
Creación de la clase UserNotFoundException.

```
geo-trace-hub-back 40-Pruebas-aceptacion-backend
UserNotFoundException.java x
1 package mx.t4g.geotracehub.exceptions;
2
3 import org.springframework.http.HttpStatus;
4 import org.springframework.web.bind.annotation.ResponseStatus;
5
6 @ResponseStatus(HttpStatus.NOT_FOUND) 4 usages new *
7 public class UserNotFoundException extends RuntimeException {
8     public UserNotFoundException(String message) { 1 usage new *
9         super(message);
10     }
11 }
```

Previamente ya se había construido un controlador global de excepciones que permitirá manejar la clase de la excepción *UserNotFoundException*, como se muestra en la Figura 249:

Figura 249

Método `handleUserNotFoundException` en `ValidationExceptionHandler`.

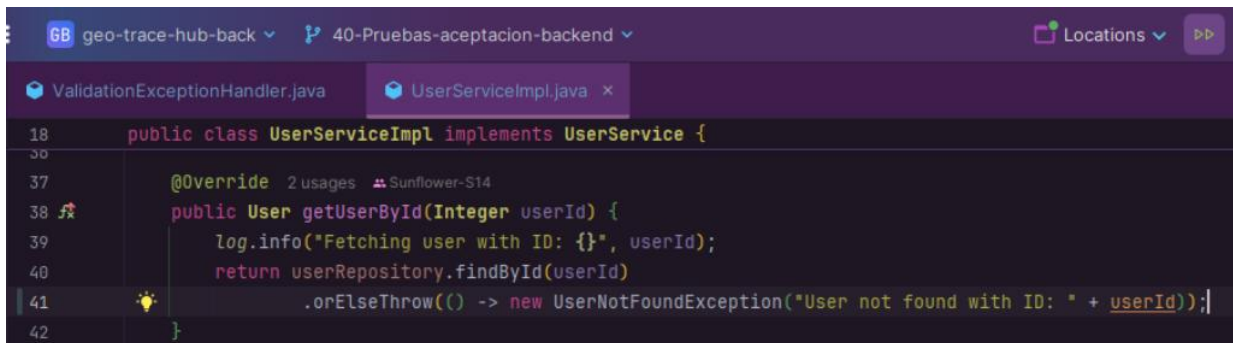


```
16 public class ValidationExceptionHandler {  
47     // Manejo 404  
48     @ResponseStatus(HttpStatus.NOT_FOUND) no usages new *  
49     @ExceptionHandler(UserNotFoundException.class)  
50     public Map<String, String> handleUserNotFoundException(UserNotFoundException ex) {  
51         return Collections.singletonMap("error", ex.getMessage());  
52     }  
}
```

En la Figura 250 se modifica la clase “`UserServiceImpl`” para invocar la excepción en el método:

Figura 250

Se llama la excepción en `getUserById` de la clase `UserServiceImpl`.

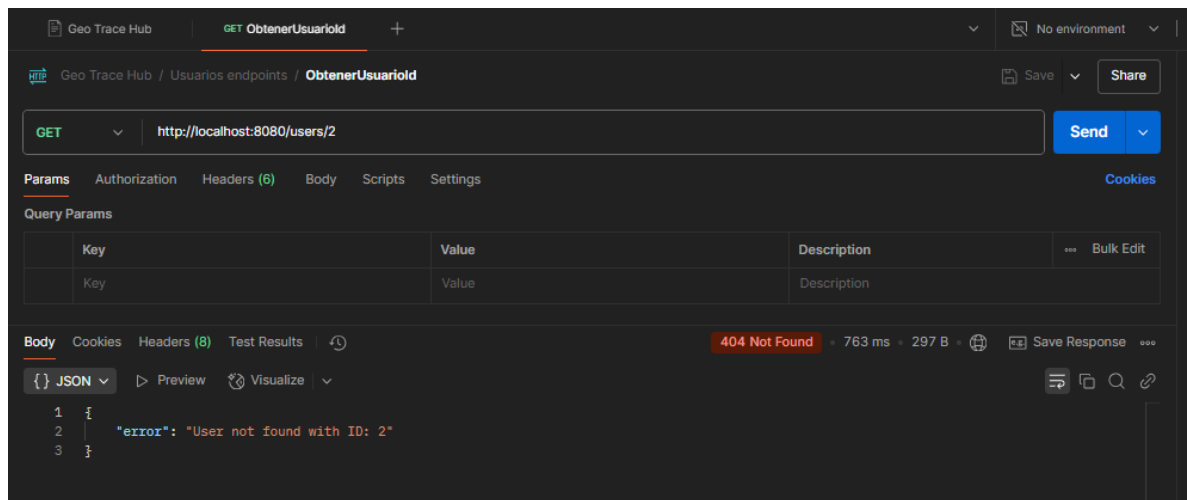


```
18 public class UserServiceImpl implements UserService {  
30  
37     @Override 2 usages Sunflower-S14  
38     public User getUserById(Integer userId) {  
39         log.info("Fetching user with ID: {}", userId);  
40         return userRepository.findById(userId)  
41             .orElseThrow(() -> new UserNotFoundException("User not found with ID: " + userId));  
42     }  
}
```

Al guardar y ejecutar se muestra el error 404 con un mensaje de error en donde indica que no encuentra al usuario. Así satisface el criterio de aceptación 2, cuyo resultado se documenta en la Figura 251:

Figura 251

Resultado del criterio 2 de ver usuario mediante su ID.



El tercer endpoint permite registrar a un usuario, ilustrado en la Tabla 5:

Tabla 5

Estructura de la petición "registro de usuario"

Tipo de petición	Endpoint a probar
POST	http://localhost:8080/users/register

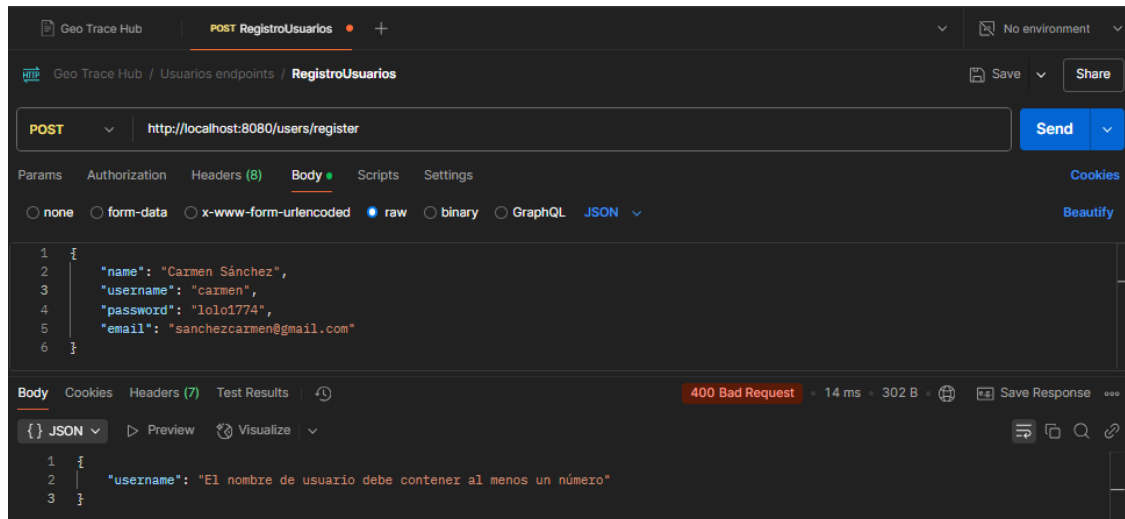
Criterios de aceptación:

1. Se debe poder registrar un usuario con nombre, nombre de usuario, correo, contraseña y campo activo.
 - 1.1. El nombre de usuario debe tener un numero al menos debe devolver *400 Bad Request*.
 - 1.2. El nombre de usuario mínimo tiene 8 caracteres debe devolver *400 Bad Request*.
 - 1.3. Si falta un campo obligatorio, debe devolver *400 Bad Request*.
 - 1.4. Si el registro es exitoso, debe devolver *200 OK* y los datos del usuario.
2. Si el correo ya está en uso, debe devolver *400 Bad Request* con un mensaje indicando que el correo ya está registrado.
3. Si el nombre de usuario ya está en uso, debe devolver *400 Bad Request* con un mensaje indicando que el nombre de usuario ya está registrado.

El resultado para el criterio 1.1 es el siguiente, el resultado en POSTMAN al introducir solo caracteres en el nombre de usuario, lanza un resultado *400 Bad Request* que indica en su mensaje que “el nombre de usuario debe de contener al menos un número”, como se muestra en la Figura 252:

Figura 252

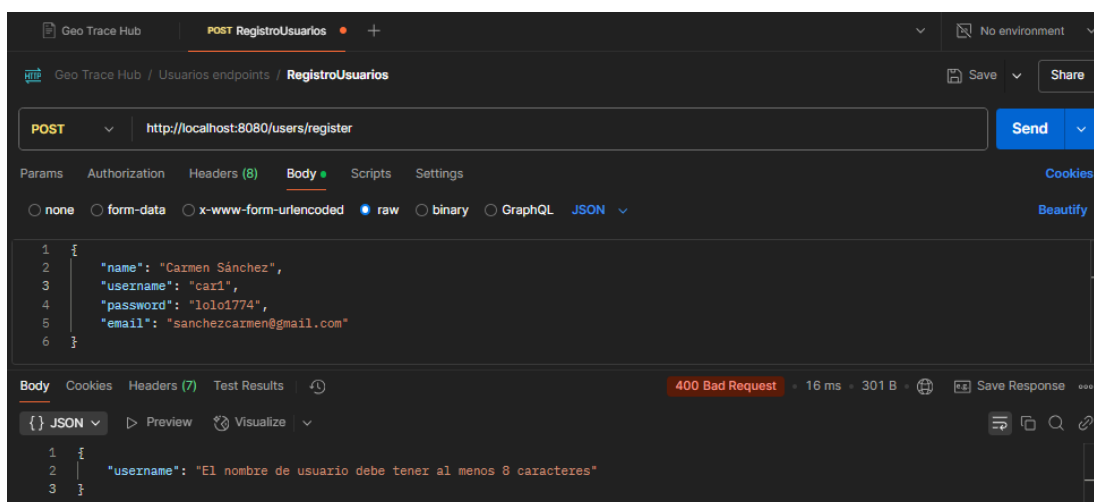
Resultado del criterio 1.1 del registro de usuario.



La Figura 253 se representa la ejecución del criterio 1.2 que respondió con un *400 Bad Request* con el mensaje “el nombre de usuario debe tener al menos 8 caracteres” en este ejemplo se le agregó un número al nombre de usuario pero como es menor a los caracteres que esperaba por eso mostró el mensaje de error en el nombre de usuario. Este criterio se cumplió adecuadamente:

Figura 253

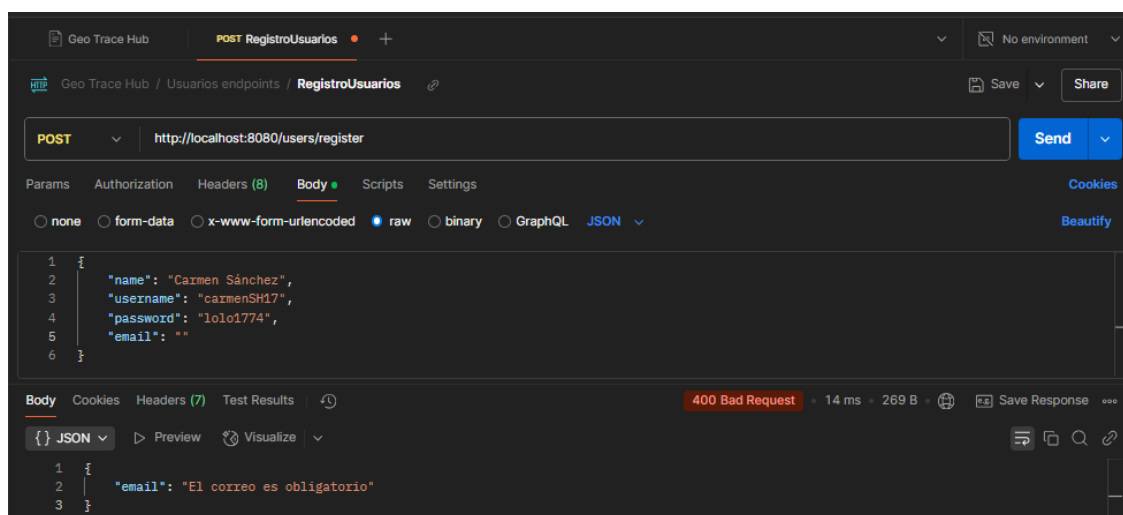
Resultado del criterio 1.2 del registro de usuario.



El resultado para el criterio 1.3 evidenciado en la Figura 254 consiste en que si falta por llenar un campo devolverá una respuesta *400 Bad Request* indicando en un mensaje que falta ese campo, en este ejemplo, no se agregó el email y al mandar la petición muestra el mensaje “El correo es obligatorio”:

Figura 254

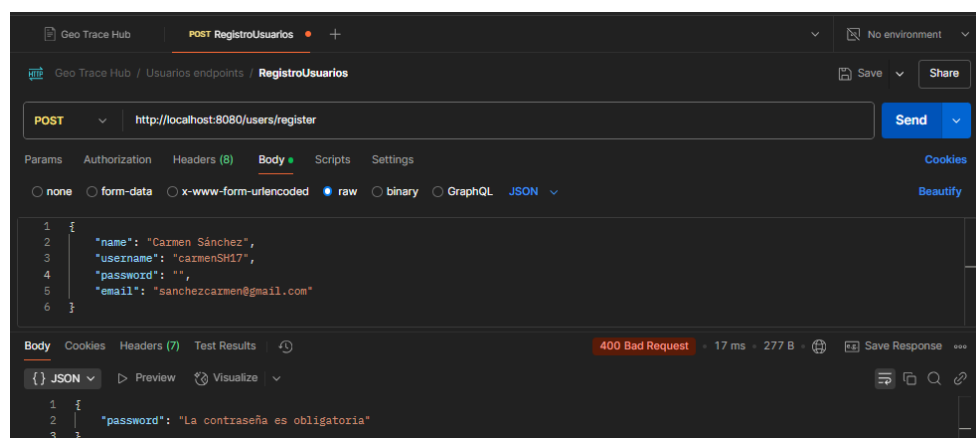
Resultado del criterio 1.3 del registro de usuario.



Usando este mismo criterio, en la Figura 255 se dejó vacío el campo de contraseña y mostró una respuesta 400 indicando el mensaje “La contraseña es obligatoria”:

Figura 255

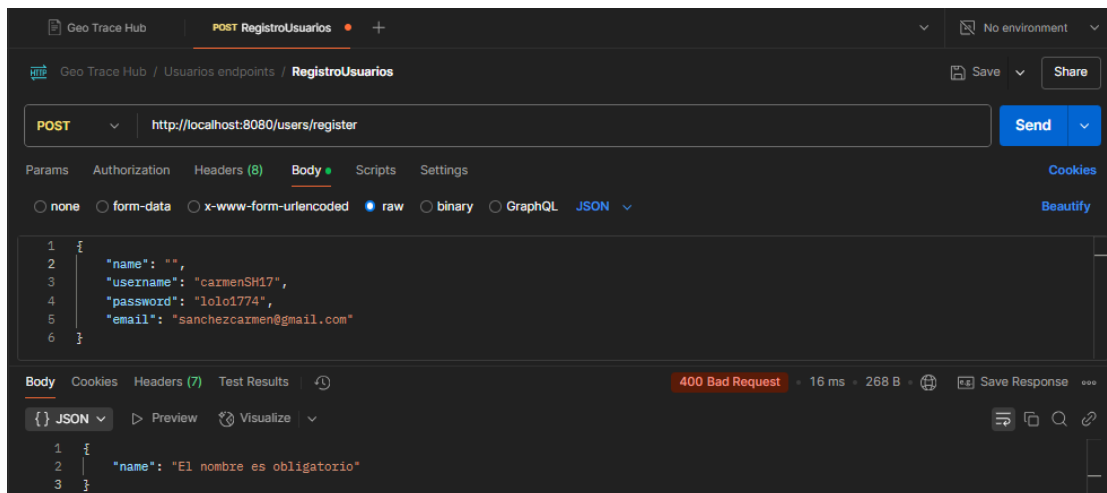
Ejecución nuevamente del criterio 1.3 del registro de usuario.



Por último, en la Figura 256 se mandó la petición sin el nombre y devolvió una respuesta 400 con el mensaje “El nombre es obligatorio”, indicando que los diferentes campos sin llenar si devuelven un mensaje:

Figura 256

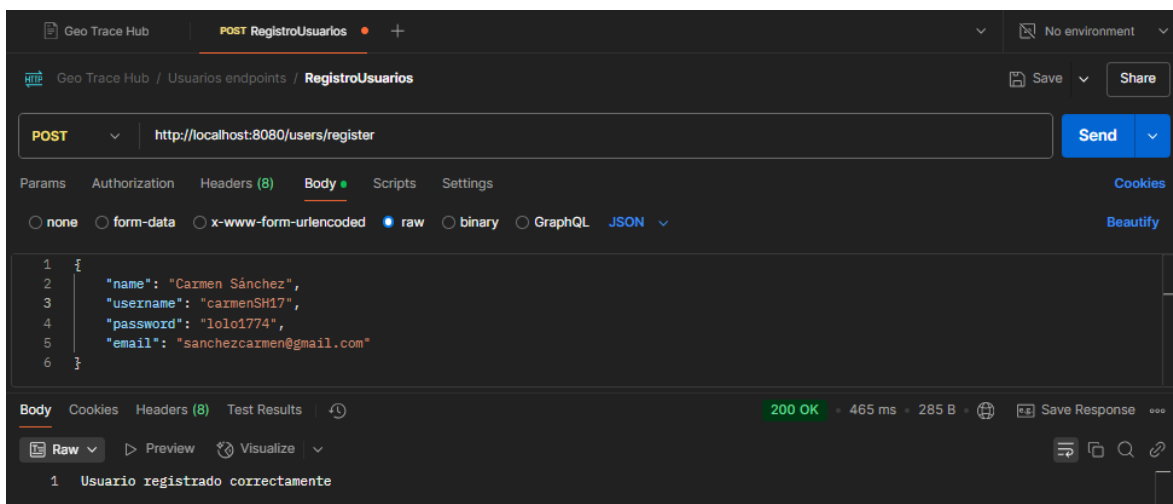
Ejecución del criterio 1.3 con otro parámetro en el registro de usuario.



El resultado para el criterio 1.4 es que *POSTMAN* guardó los datos del usuario devolviendo una respuesta `200 OK` y mostrando el mensaje “usuario registrado correctamente”, como se aprecia en la Figura 257:

Figura 257

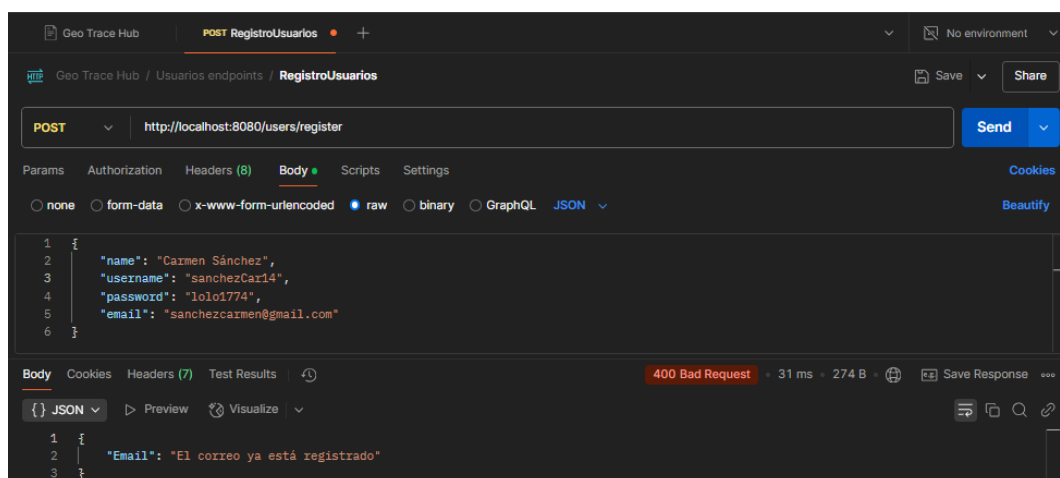
Resultado del criterio 1.4 del registro de usuario.



En la Figura 258 se intenta registrar los datos diferentes, a excepción del email, este comportamiento genera una respuesta de error *400 Bad Request* que muestra un mensaje: “el correo ya está registrado”. Mostrando que es una respuesta *400 Bad Request* cumpliendo así el criterio 2:

Figura 258

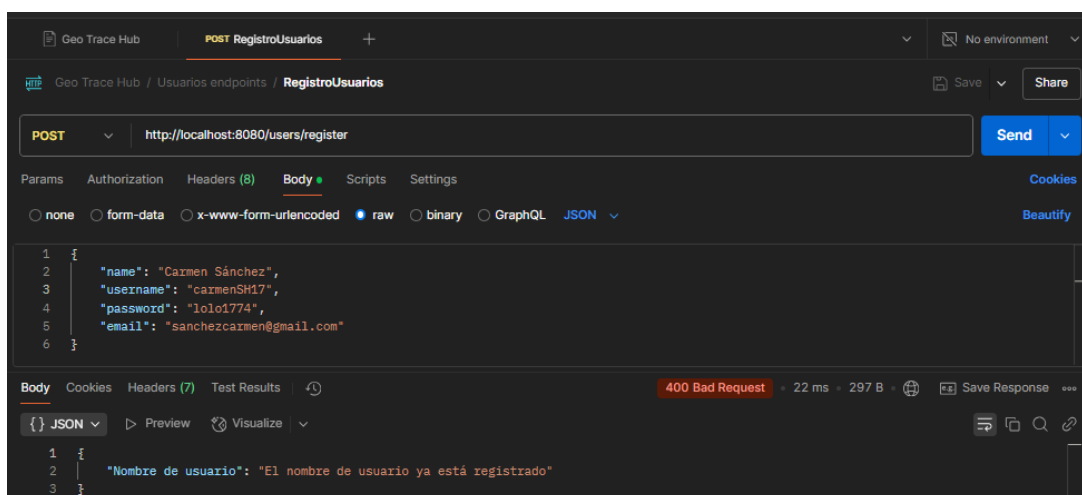
Resultado del criterio 2 del registro de usuario.



En la Figura 259 se cambian todos los datos a excepción del nombre de usuario, en el que se muestra el mensaje “El nombre de usuario ya está registrado” y así cumplirá el criterio 3:

Figura 259

Resultado del criterio 3 del registro de usuario.



El cuarto endpoint permite la edición de un usuario, como se muestra en la Tabla 6:

Tabla 6

Estructura de la petición "edición de usuarios"

Tipo de petición	Endpoint a probar
PUT	http://localhost:8080/users/{id}

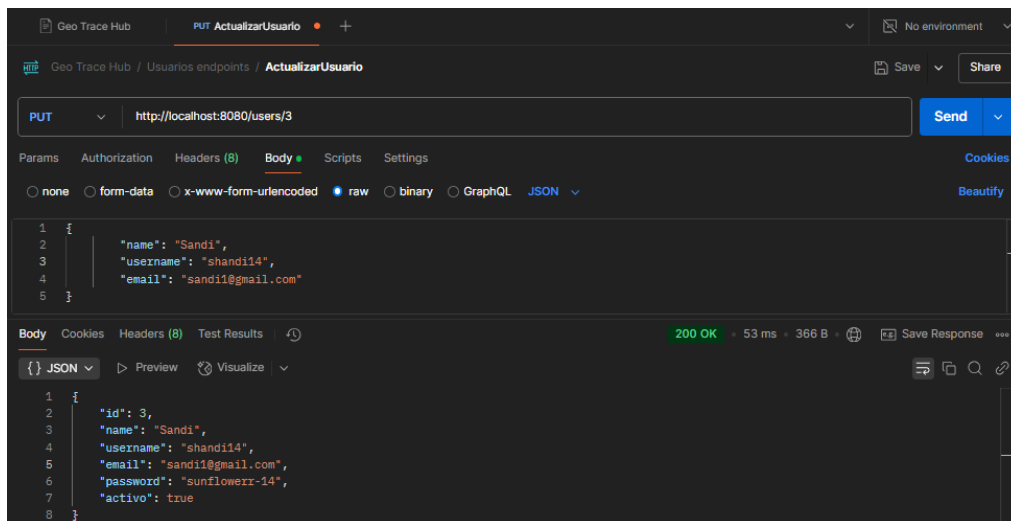
Criterios de aceptación:

1. Se debe poder actualizar los datos de un usuario (nombre, nombre de usuario, correo), debe devolver *200 OK*.
2. No se debe poder cambiar la contraseña en este endpoint.
3. Si el usuario no existe, debe devolver *404 Not Found*.
4. Si los datos son inválidos, debe devolver *400 Bad Request*.

En la Figura 260, se muestra el resultado del criterio 1, en el que se indicó en el endpoint el número de ID, se usaron los datos en formato JSON para actualizar en el registro los cuales son nombre, nombre de usuario y correo electrónico. Se devolvió una respuesta *200 OK* mostrando una respuesta JSON con todos los datos del usuario y se refleja que se aplicó la actualización cumpliendo el criterio de aceptación:

Figura 260

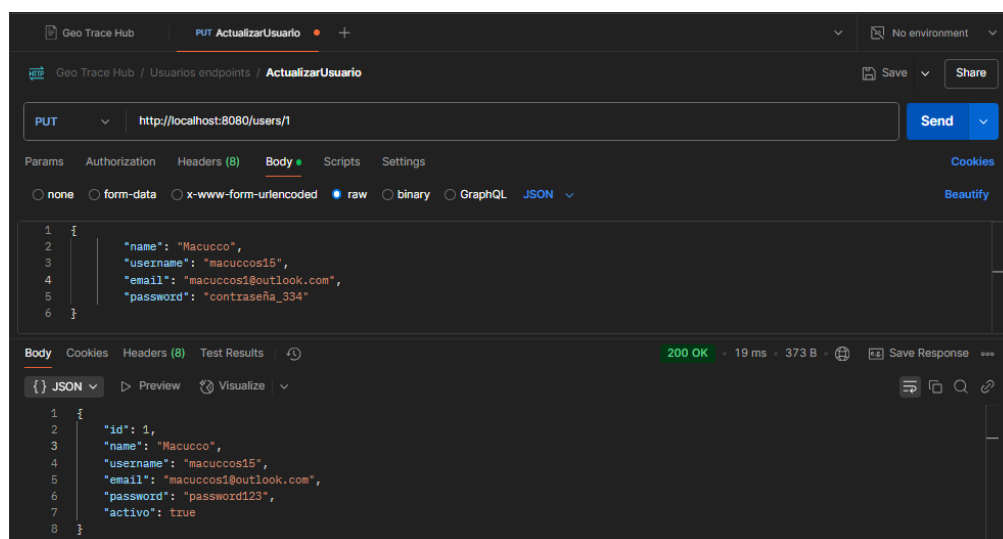
Resultado del criterio 1 de la edición de usuarios.



Para probar el criterio 2 se agrega en la respuesta JSON el campo de contraseña. Al mandar la solicitud, no aplica el cambio cumpliendo así con el criterio de aceptación. Este endpoint solo modifica nombre, nombre de usuario y correo, como se ilustra en la Figura 261. Para modificar la contraseña sería crear otro endpoint que permita actualizar la contraseña, no obstante para este proyecto está fuera del alcance:

Figura 261

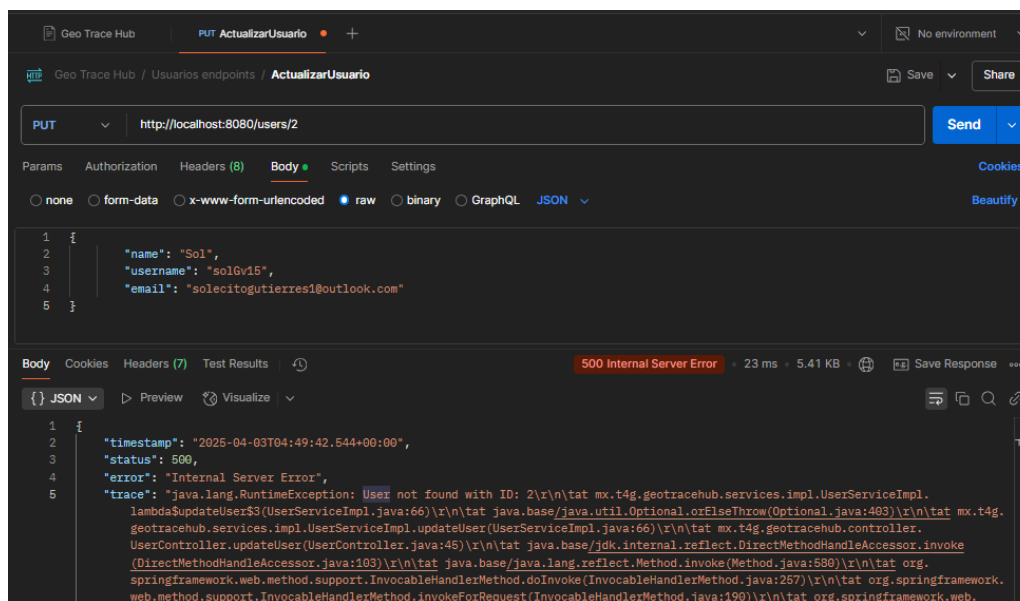
Resultado del criterio 2 de la edición de usuarios.



En la Figura 262 corresponde al resultado del criterio 3, *POSTMAN* mostró un error *500 Internal Server Error* en el que indica que el usuario no se encontró:

Figura 262

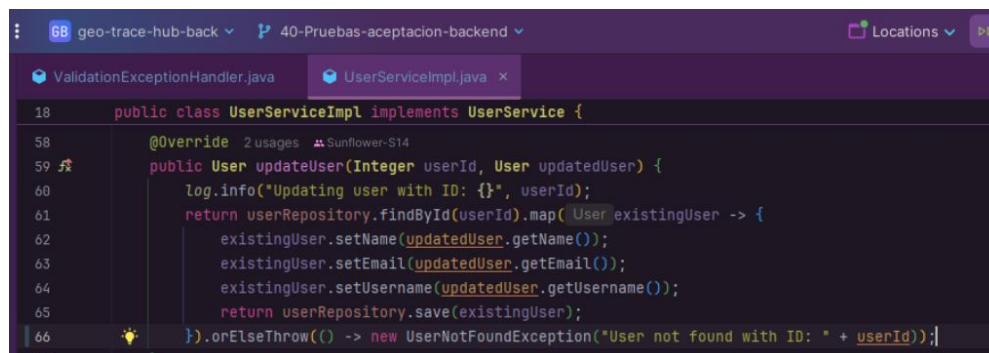
Resultado de error del criterio 3 de la edición de usuarios.



Para evitar este error y que devuelva el error *404 Not Found*, en la Figura 263 muestra la clase “*UserNotFoundException*” previamente creada y se llama en el método “*updateUser*” que pertenece a la clase “*UserServiceImpl*”:

Figura 263

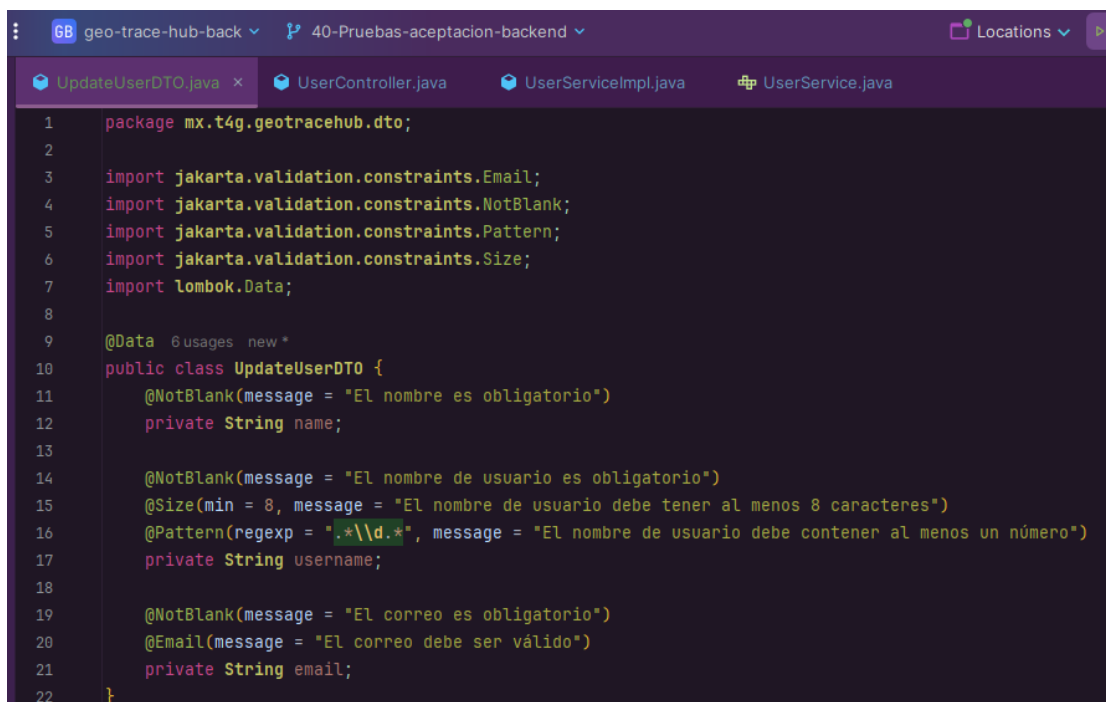
En el método `updateUser` se llama la excepción `UserNotFoundException`.



En la Figura 266 se crea una clase “*DTO*” que es como una capa intermedia entre el modelo y el servicio que permite aplicar validaciones a los atributos de nombre, nombre de usuario y correo para poder actualizar los datos del usuario:

Figura 266

Clase UpdateUserDTO .

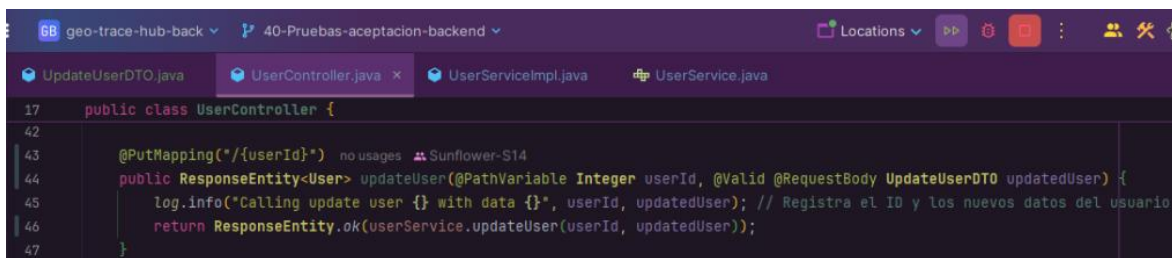


```
1 package mx.t4g.geotracehub.dto;
2
3 import jakarta.validation.constraints.Email;
4 import jakarta.validation.constraints.NotBlank;
5 import jakarta.validation.constraints.Pattern;
6 import jakarta.validation.constraints.Size;
7 import lombok.Data;
8
9 @Data 6 usages new *
10 public class UpdateUserDTO {
11     @NotBlank(message = "El nombre es obligatorio")
12     private String name;
13
14     @NotBlank(message = "El nombre de usuario es obligatorio")
15     @Size(min = 8, message = "El nombre de usuario debe tener al menos 8 caracteres")
16     @Pattern(regexp = ".*\\d.*", message = "El nombre de usuario debe contener al menos un número")
17     private String username;
18
19     @NotBlank(message = "El correo es obligatorio")
20     @Email(message = "El correo debe ser válido")
21     private String email;
22 }
```

En la Figura 267 se modifica el controlador “UserController” en el *RequestBody* de usuario se agrega la anotación *@Valid* y se llama a la clase “*DTO*” como parámetros que aplican las validaciones de los datos:

Figura 267

Método updateUser invoca la clase DTO.

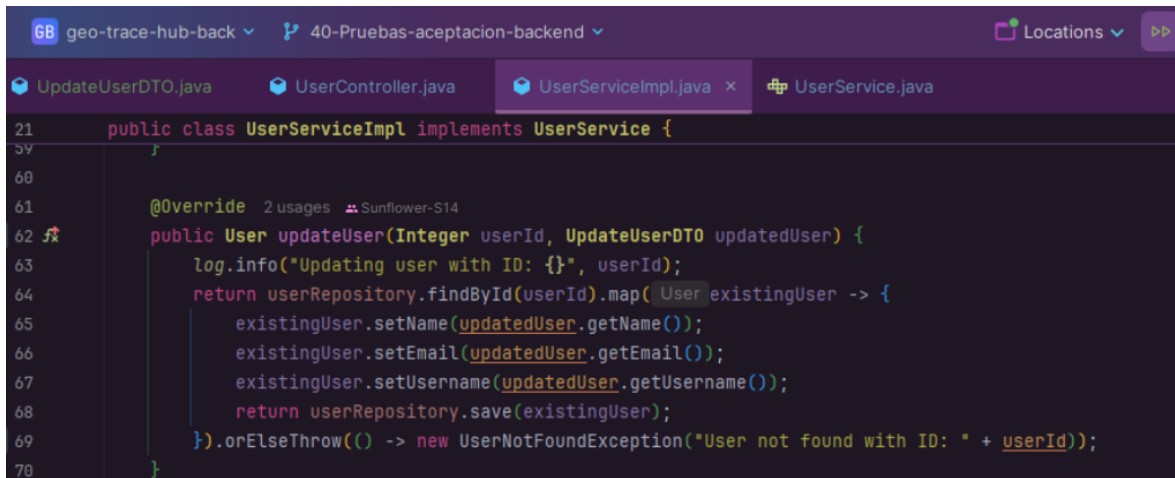


```
17 public class UserController {
42
43     @PutMapping("/{userId}") no usages Sunflower-S14
44     public ResponseEntity<User> updateUser(@PathVariable Integer userId, @Valid @RequestBody UpdateUserDTO updatedUser) {
45         log.info("Calling update user {} with data {}", userId, updatedUser); // Registra el ID y los nuevos datos del usuario
46         return ResponseEntity.ok(userService.updateUser(userId, updatedUser));
47     }
```

Se modifica las validaciones de datos que no estén en blanco o nulo en la clase “UserServiceImpl” que se presenta en la Figura 268:

Figura 268

La clase UserServiceImpl usa la clase DTO .

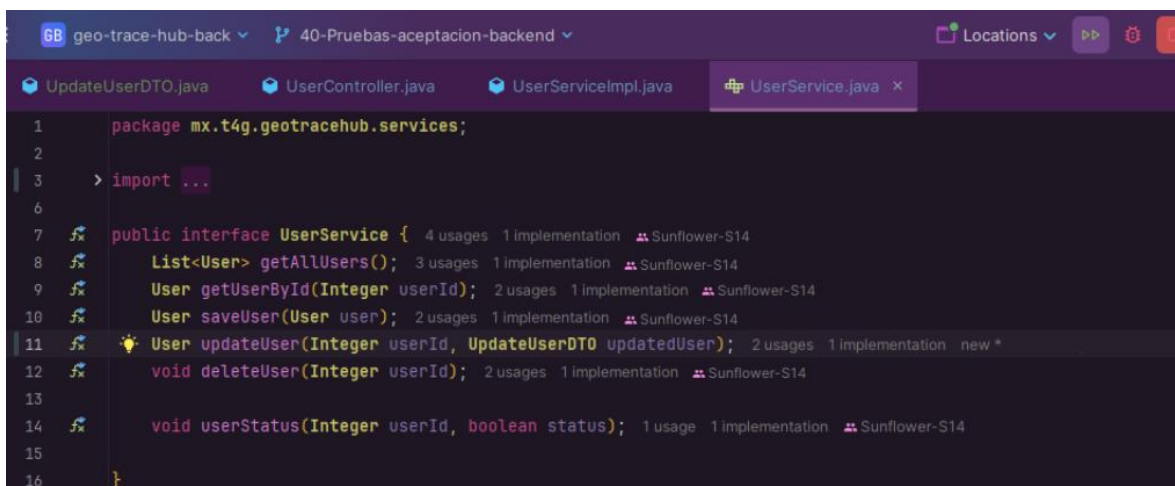


```
21 public class UserServiceImpl implements UserService {
59 }
60
61 @Override 2 usages Sunflower-S14
62 public User updateUser(Integer userId, UpdateUserDTO updatedUser) {
63     log.info("Updating user with ID: {}", userId);
64     return userRepository.findById(userId).map( User existingUser -> {
65         existingUser.setName(updatedUser.getName());
66         existingUser.setEmail(updatedUser.getEmail());
67         existingUser.setUsername(updatedUser.getUsername());
68         return userRepository.save(existingUser);
69     }).orElseThrow(() -> new UserNotFoundException("User not found with ID: " + userId));
70 }
```

En la Figura 269 se ilustra la clase “UserService” se aplican modificaciones para usar la clase *DTO* en el método “updateUser”:

Figura 269

El método updateUser usa la clase DTO en la clase UserService.

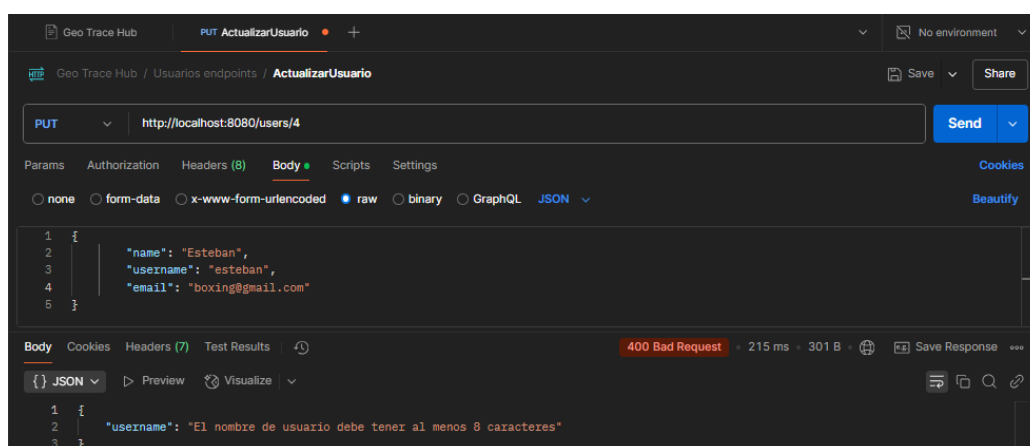


```
1 package mx.t4g.geotracehub.services;
2
3 > import ...
6
7 public interface UserService { 4 usages 1 implementation Sunflower-S14
8     List<User> getAllUsers(); 3 usages 1 implementation Sunflower-S14
9     User getUserById(Integer userId); 2 usages 1 implementation Sunflower-S14
10    User saveUser(User user); 2 usages 1 implementation Sunflower-S14
11    User updateUser(Integer userId, UpdateUserDTO updatedUser); 2 usages 1 implementation new *
12    void deleteUser(Integer userId); 2 usages 1 implementation Sunflower-S14
13
14    void userStatus(Integer userId, boolean status); 1 usage 1 implementation Sunflower-S14
15
16 }
```

Después de corregir y volver a ejecutar el *backend*, en la Figura 270 se muestra que en POSTMAN al mandar la petición mandará una respuesta *400 Bad Request* con mensajes de las validaciones:

Figura 270

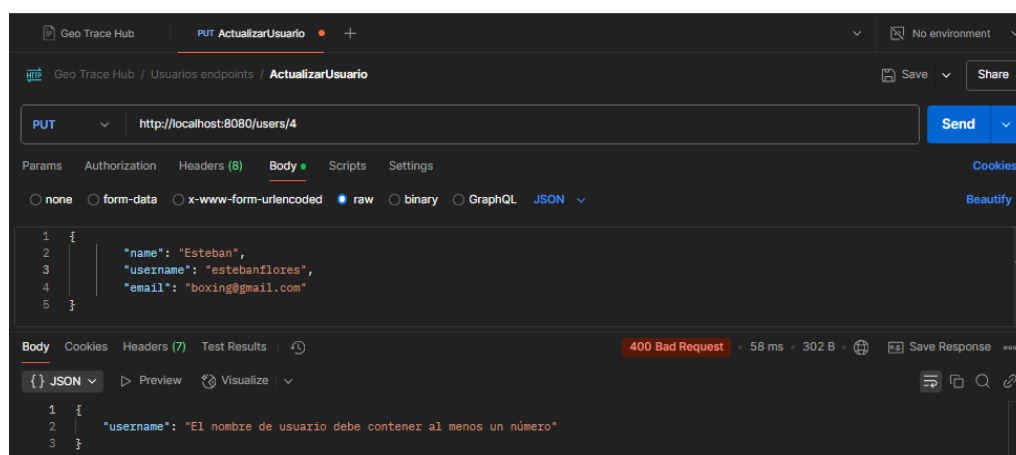
Resultado del criterio 4 de la edición de usuarios en el que el nombre de usuario no cumple con el criterio de longitud.



De ejemplo en la Figura 271 se muestra que no se cumple la validación de que el nombre de usuario debe tener al menos un número y así se cumple este criterio de aceptación:

Figura 271

Resultado del criterio 4 de la edición de usuarios en el que el nombre de usuario no cumple los criterios de al menos un número en el nombre de usuario.



El quinto endpoint permite desactivar o activar una cuenta de un usuario, presentado en la Tabla 7:

Tabla 7

Estructura de la petición "cambiar estado activo de la cuenta de usuario"

Tipo de petición	Endpoint a probar
PATCH	http://localhost:8080/users/1/status?activo=false
	http://localhost:8080/users/1/status?activo=true

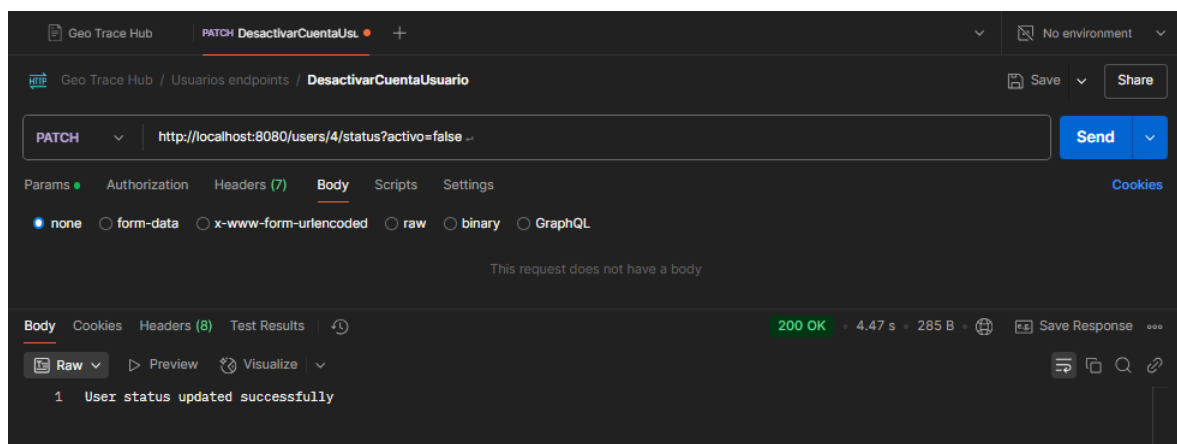
Criterios de aceptación:

1. Se debe poder activar o desactivar un usuario.
 - 1.1. Si la operación es exitosa, debe devolver *200 OK* con el nuevo estado.
2. Si el usuario no existe, debe devolverá un 422 Unprocessable Entity (que indica que hay error en la lógica del negocio al buscar un usuario que no existe para desactivarlo).

El resultado del criterio 1.1 corresponde a la Figura 272 en el que al cambiar el estado se recibe una respuesta *200 OK* con un mensaje “user status updated successfully”:

Figura 272

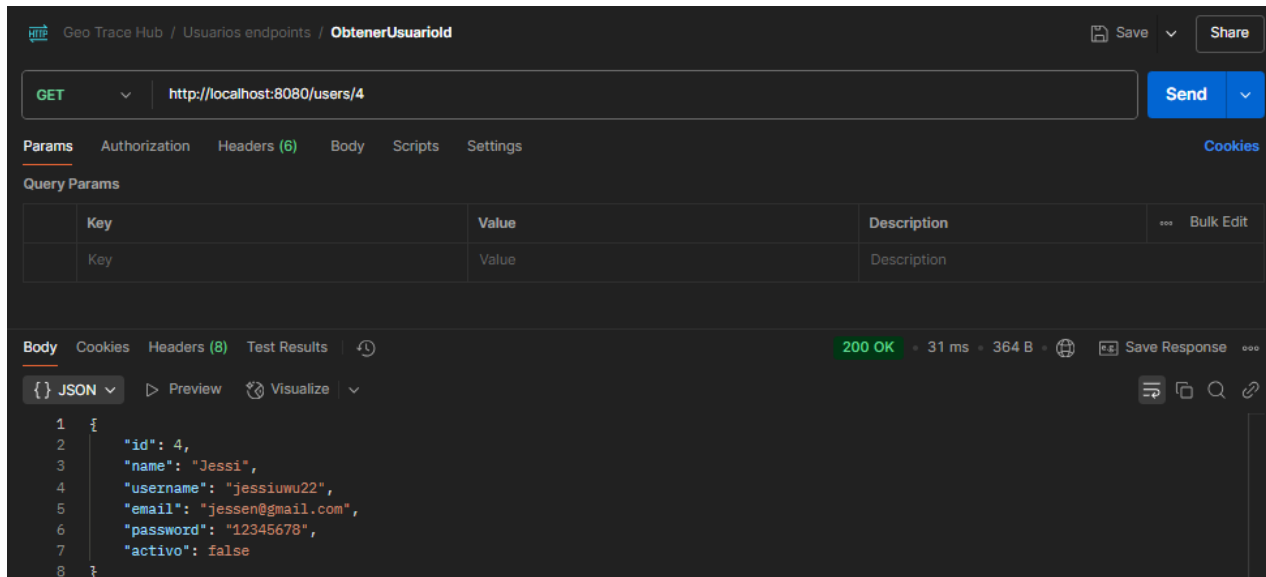
Resultado del criterio 1.1 para cambiar estado activo de la cuenta de usuario.



Para verificar que sí se actualiza en la Figura 273 se muestra que al ejecutar el endpoint de obtener un usuario por su ID se muestra al usuario con el campo “activo” en “false”:

Figura 273

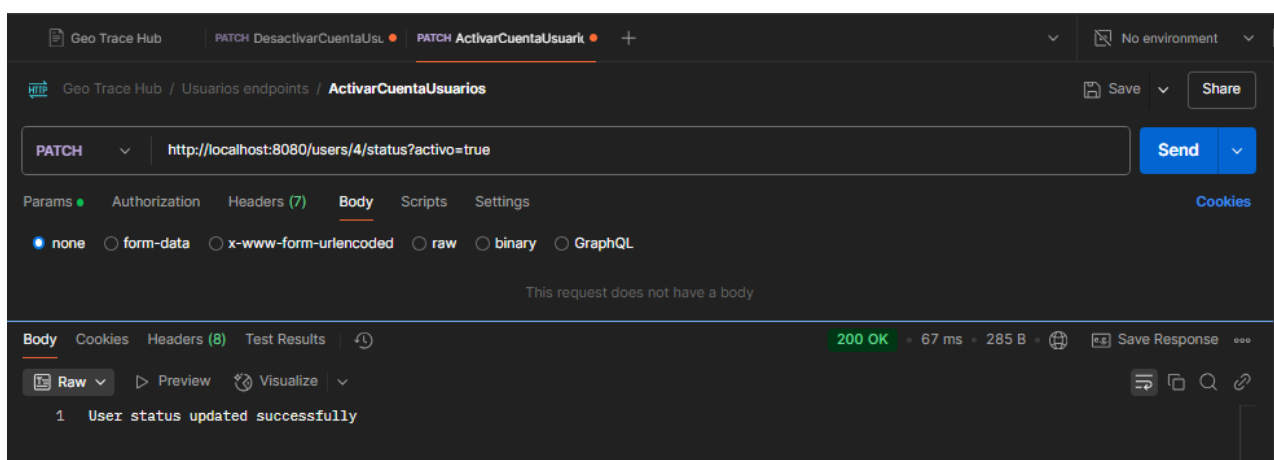
Cambio de estado en falso de la cuenta de usuario.



En la Figura 274 se muestra que para volver a activar la cuenta solo se cambia el estado a “true” que recibe como resultado una respuesta 200 OK con el mismo mensaje:

Figura 274

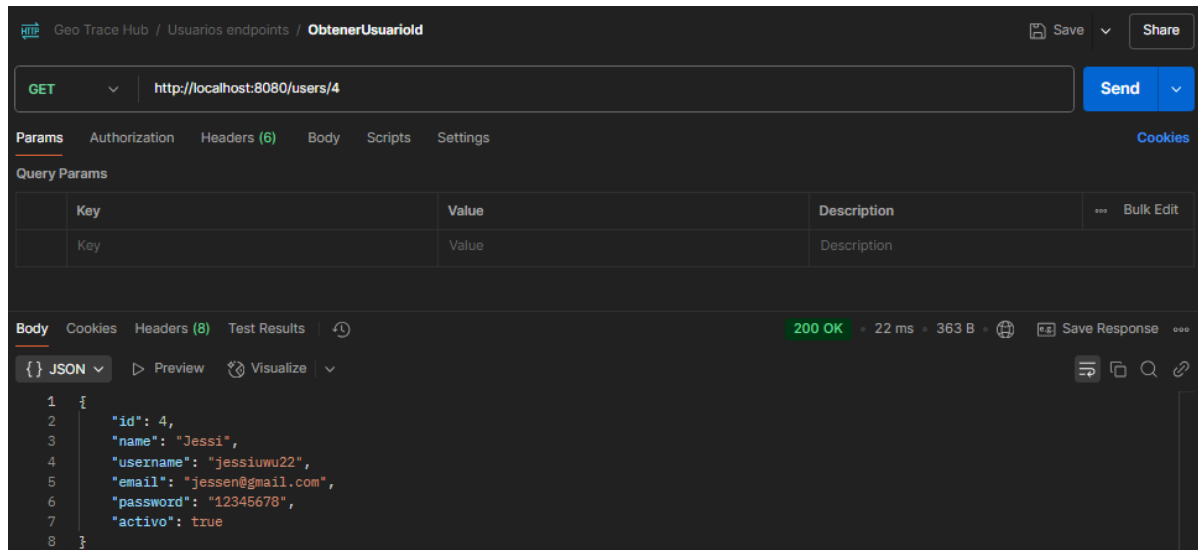
Resultado de cambiar el estado en activo.



Al buscar nuevamente el usuario, en la Figura 275 se confirma que el criterio 1.1 se han cumplido:

Figura 275

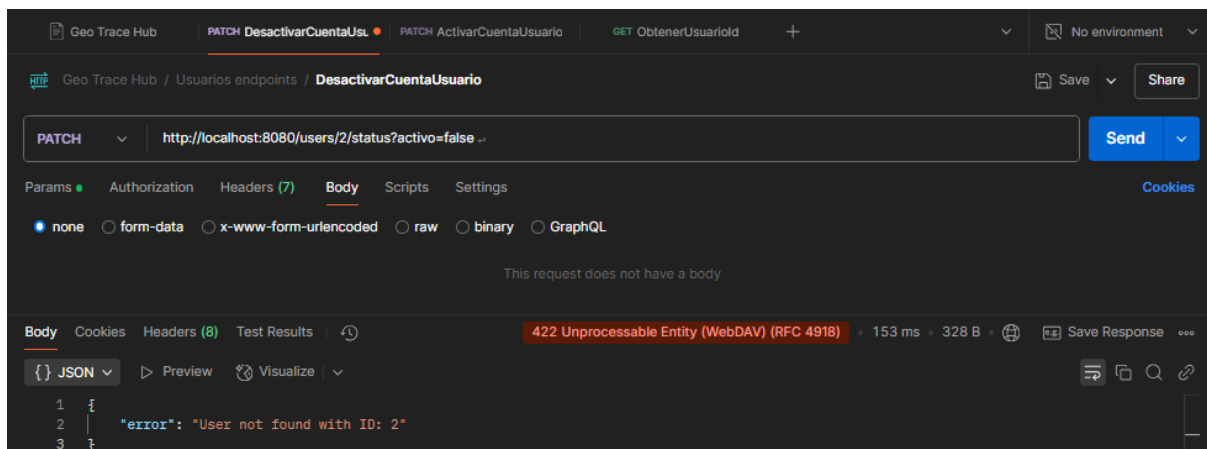
Cambio de estado en verdadero de la cuenta de usuario.



En la Figura 276 se muestra el resultado del criterio 2 en el que se indicó al endpoint que desactivara un usuario que no existe teniendo como respuesta **422 Unprocessable** que indica que la solicitud no se pudo procesar debido a errores en los datos proporcionados:

Figura 276

Resultado del criterio 2 cambiar estado activo de la cuenta de usuario.



El último endpoint, se comporta para la eliminación de un usuario a través de la Tabla 8:

Tabla 8

Estructura de la petición "eliminación de un usuario".

Tipo de petición	Endpoint a probar
DELETE	http://localhost:8080/users/{id}

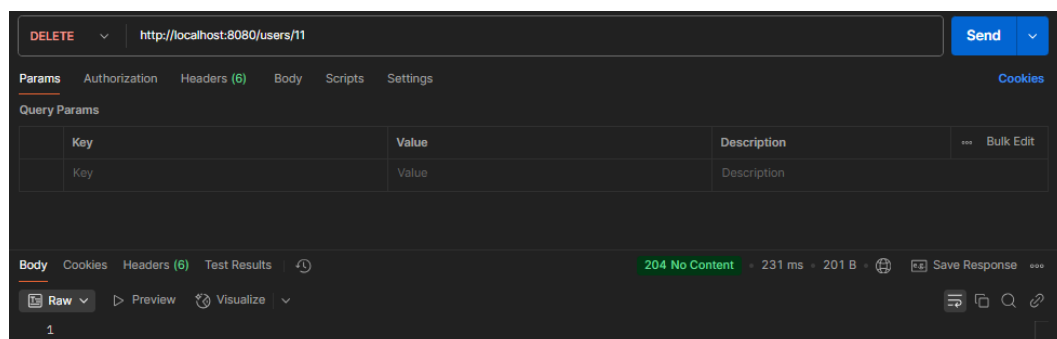
Criterios de aceptación:

1. Se debe poder eliminar un usuario sin dependencias activas.
 - 1.1. Si la eliminación es exitosa, debe devolver *204 No Content*.
2. Si el usuario no existe, debe devolver *404 Not Found*.
3. Si el usuario tiene registros relacionados que impiden su eliminación, debe devolver *409 Conflict*.

En el resultado del criterio 1.1 reflejado en la Figura 277 se eliminó un usuario que no tenía ni dispositivos ni ubicaciones registradas, lo cual permitió eliminar adecuadamente el usuario devolviendo la respuesta *204 No Content*:

Figura 277

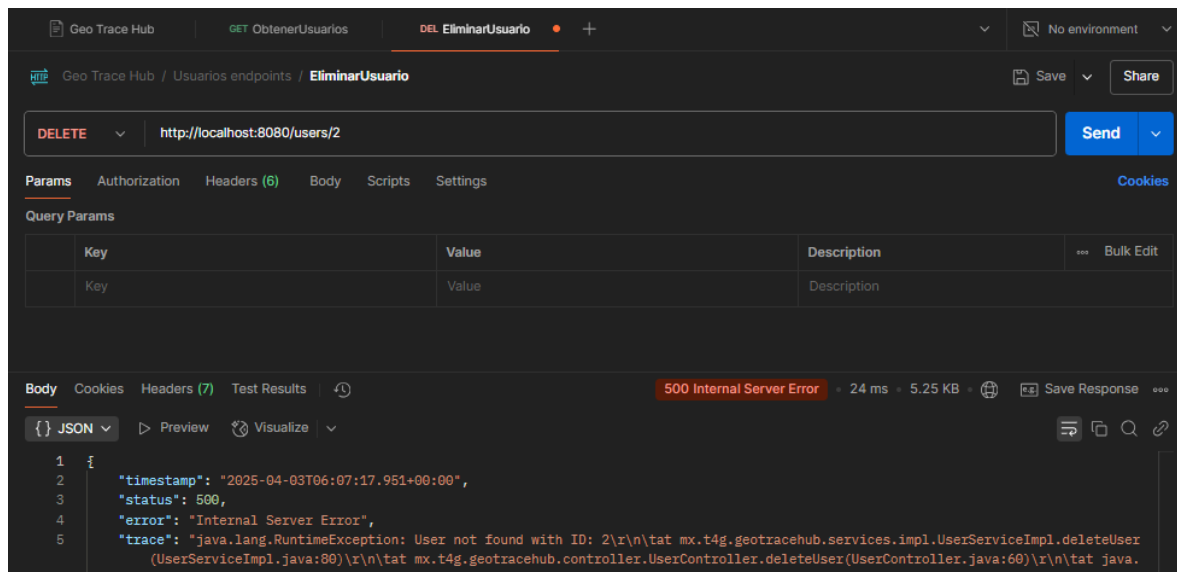
Resultado del criterio 1.1 de la eliminación de un usuario.



A través de la Figura 278, se realiza la prueba del criterio 2 en el que se intenta eliminar un usuario que no existe manda un error *500 Internal Server Error* el cual no era la respuesta esperada para el criterio 2:

Figura 278

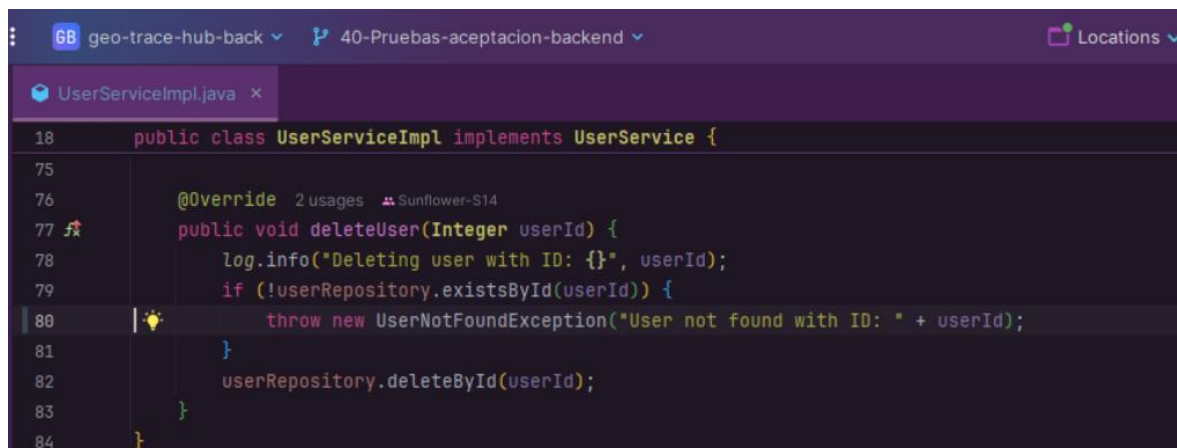
Resultado erróneo del criterio 2 de la eliminación de un usuario.



En la Figura 279 se muestra que se corrigió en la clase `UserServiceImpl` en el método `deleteUser` se llama a la excepción para mostrar limpiamente el error en respuesta *404 Not Found*:

Figura 279

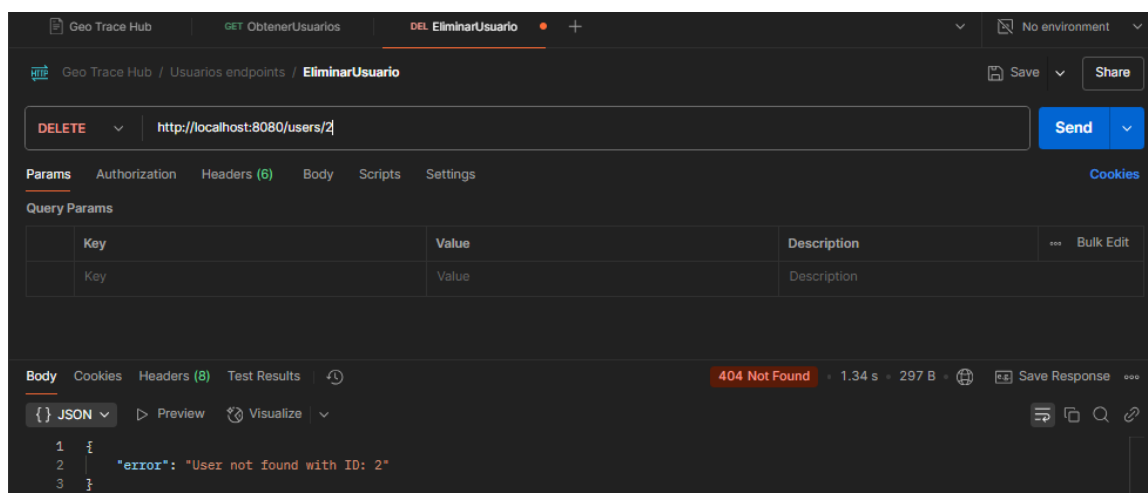
Se llama la excepción `UserNotFoundException` en el método `deleteUser`.



Después de aplicar cambios, al mandar la petición en la Figura 280 devuelve un mensaje indicando que no se encuentra el usuario especificado con una respuesta *404 Not Found* cumpliendo el criterio de aceptación:

Figura 280

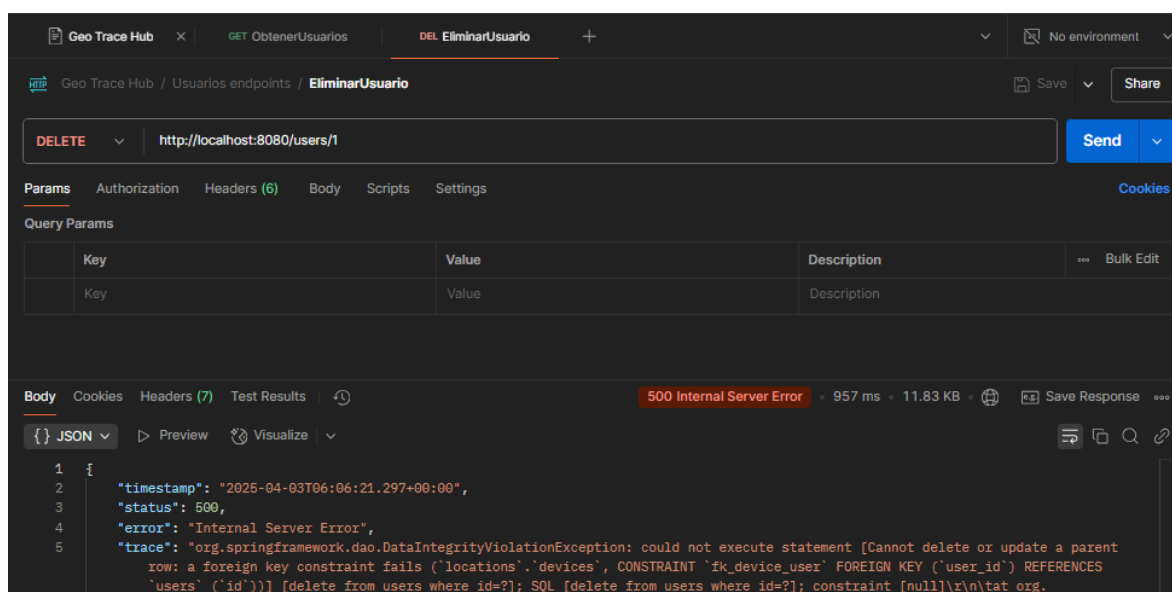
Resultado del criterio 2 de la eliminación de un usuario.



Al probar el criterio 3 en la Figura 281 se mostró un error *500* y no la respuesta *409 conflict* :

Figura 281

Resultado erróneo del criterio 3 de la eliminación de un usuario.



Se crea en la Figura 282 una excepción que indica que existen conflictos de datos:

Figura 282

Clase ConflictException.



```
1 package mx.t4g.geotracehub.exceptions;
2
3 import lombok.Getter;
4
5 @Getter 4 usages new *
6 public class ConflictException extends RuntimeException {
7     public ConflictException(String message) { 1 usage new *
8         super(message);
9     }
10 }
```

En el controlador de excepciones capturará el “ConflictException” y devolverá un *409 Conflict* ilustrada en la Figura 283 debido a que no se pueden eliminar los datos por las restricciones de las llaves foráneas:

Figura 283

Método handleConflictException en ValidationExceptionHandler.

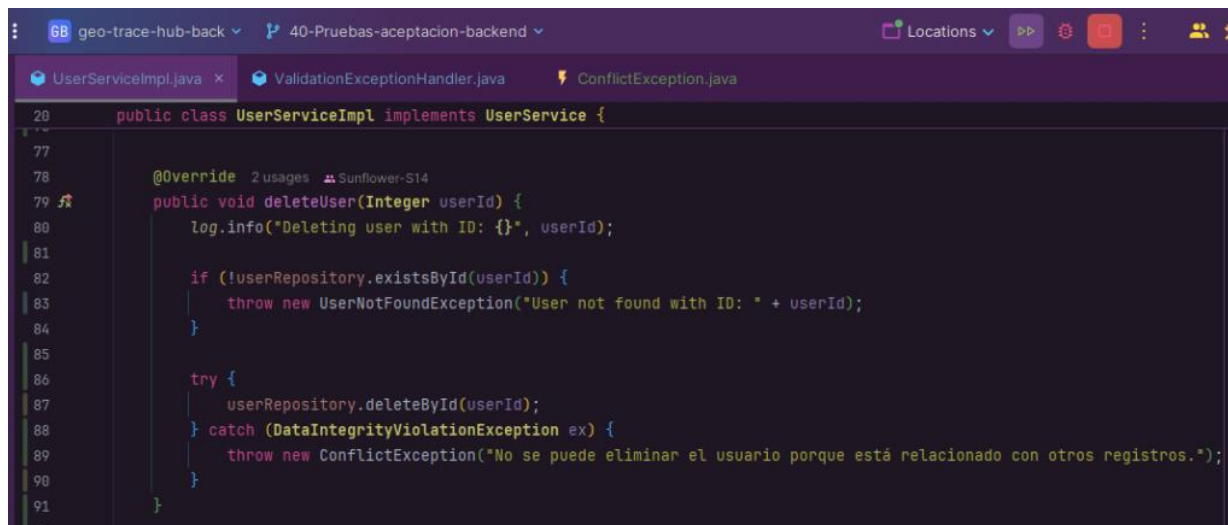


```
16 public class ValidationExceptionHandler {
17
18     // Manejo de conflictos(restricciones de clave foránea)
19     @ResponseStatus(HttpStatus.CONFLICT) no usages new *
20     @ExceptionHandler(ConflictException.class)
21     @
22     public Map<String, String> handleConflictException(ConflictException ex) {
23         return Collections.singletonMap("error", ex.getMessage());
24     }
25 }
```

En la Figura 284 se modifica en la clase UserServiceImpl el método deleteUser para capturar la excepción:

Figura 284

Uso de las excepciones UserNotFoundException y ConflictException en el método deleteUser.

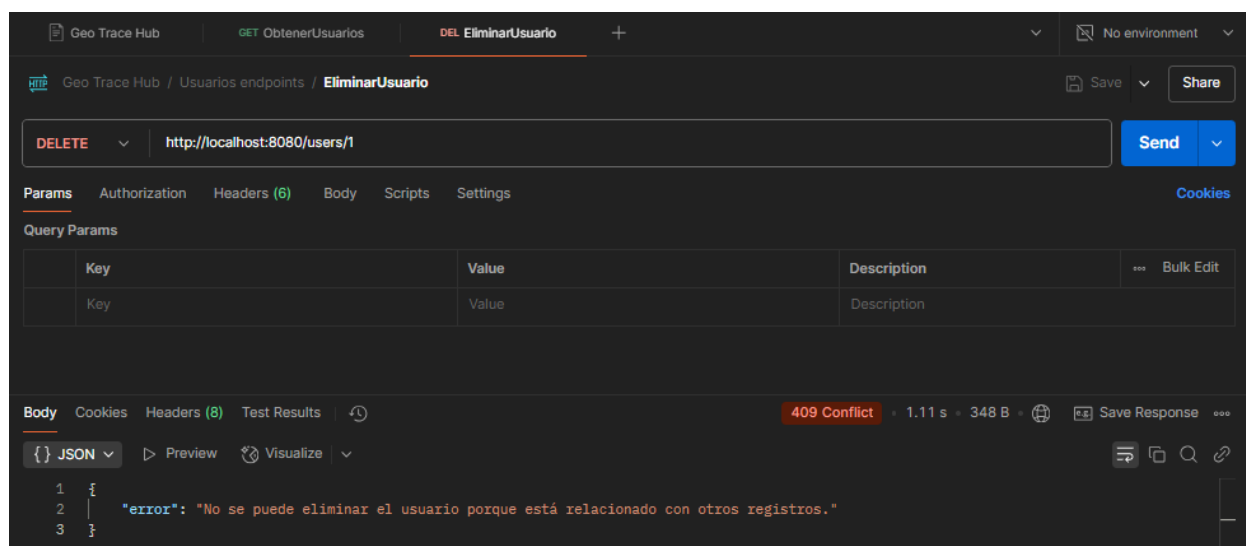


```
20 public class UserServiceImpl implements UserService {
77
78     @Override 2 usages Sunflower-S14
79     public void deleteUser(Integer userId) {
80         log.info("Deleting user with ID: {}", userId);
81
82         if (!userRepository.existsById(userId)) {
83             throw new UserNotFoundException("User not found with ID: " + userId);
84         }
85
86         try {
87             userRepository.deleteById(userId);
88         } catch (DataIntegrityViolationException ex) {
89             throw new ConflictException("No se puede eliminar el usuario porque está relacionado con otros registros.");
90         }
91     }
92 }
```

Al guardar y ejecutar nuevamente la petición, en la Figura 285 se aprecia que devuelve una respuesta **409 Conflict** y el mensaje “No se puede eliminar el usuario porque está relacionado con otros registros”:

Figura 285

Resultado del criterio 3 de la eliminación de un usuario.



Geo Trace Hub / Usuarios endpoints / EliminarUsuario

DELETE http://localhost:8080/users/1

Params Authorization Headers (6) Body Scripts Settings Cookies

Query Params

Key	Value	Description
Key	Value	Description

Body Cookies Headers (8) Test Results

409 Conflict 1.11 s 348 B

JSON Preview Visualize

```
1 {
2   "error": "No se puede eliminar el usuario porque está relacionado con otros registros."
3 }
```

Administración de dispositivos

El primer endpoint de esta sección es invocar a todos los dispositivos registrados, su estructura está en la Tabla 9:

Tabla 9

Estructura de la petición para “llamar a todos los dispositivos”.

Tipo de petición	Endpoint a probar
GET	http://localhost:8080/devices

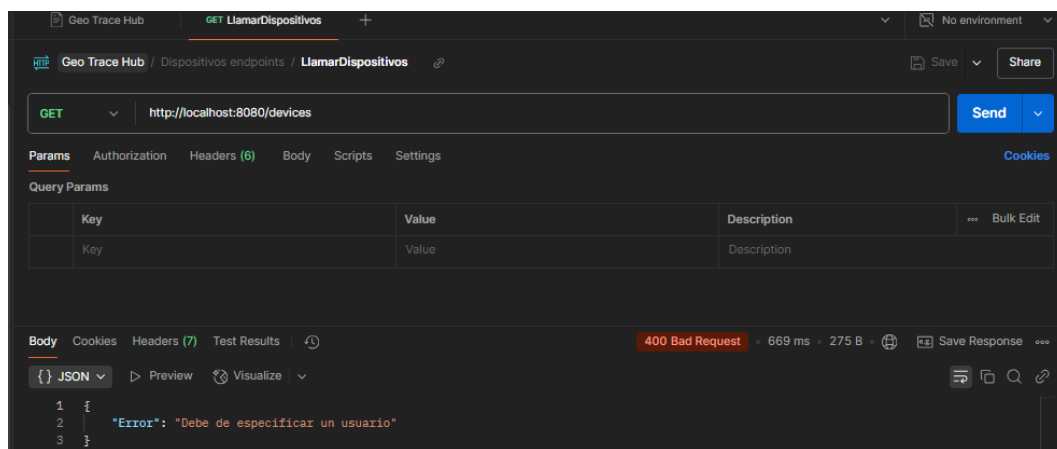
Criterios de aceptación:

1. Al llamar todos los dispositivos debe de mandar un mensaje de error indicando que se debe de especificar un usuario en específico, devolviendo un *400 Bad Request*.

La Figura 286 muestra el resultado del criterio 1 implica en que no se pueden obtener todos los dispositivos porque son de diferentes usuarios, por lo que en la prueba solo se muestra un mensaje indicando que debe de especificar el usuario, se muestra una respuesta *400 Bad Request*:

Figura 286

Resultado del criterio 1 para llamar a todos los dispositivos.



En la Tabla 10 se muestra el segundo endpoint permite buscar un dispositivo por su ID para visualizar su información:

Tabla 10

Estructura de la petición "listar un dispositivo mediante su ID".

Tipo de petición	Endpoint a probar
GET	http://localhost:8080/devices/{id}

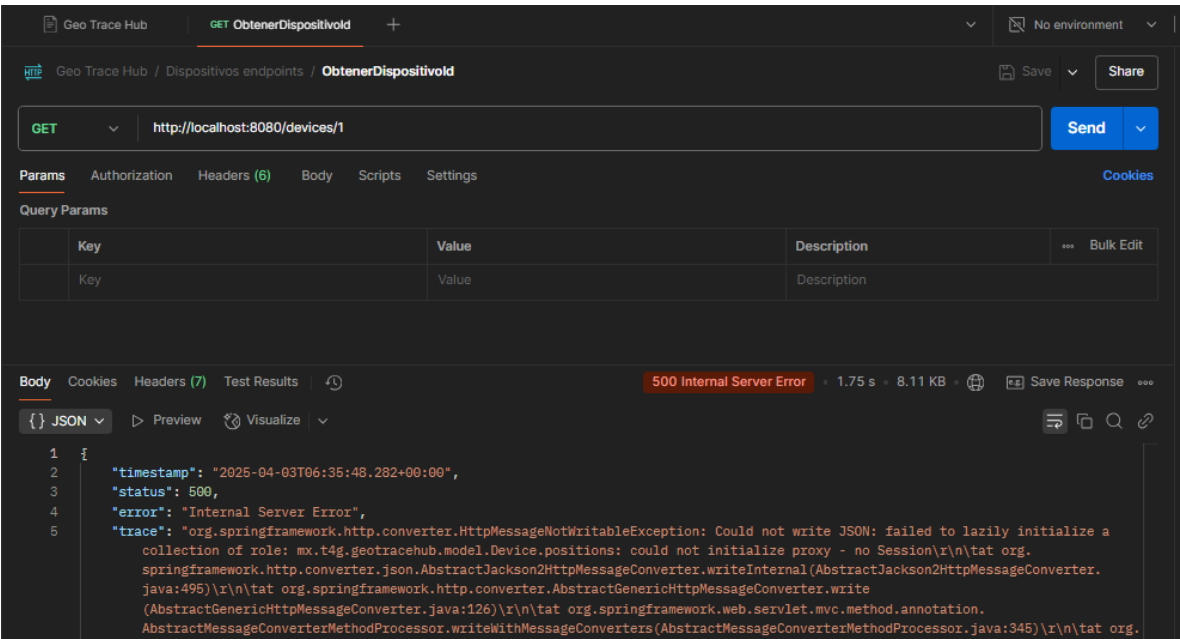
Criterios de aceptación:

- 1. Debe de devolver los datos del dispositivo juntamente del usuario a que le pertenecen.

En el resultado del criterio 1, se muestra en la Figura 287 un error que es similar a lo que le pasaba al endpoint de listar usuarios, en el que se inicializan los objetos pero no lo logran porque quieren serializarse al mismo tiempo:

Figura 287

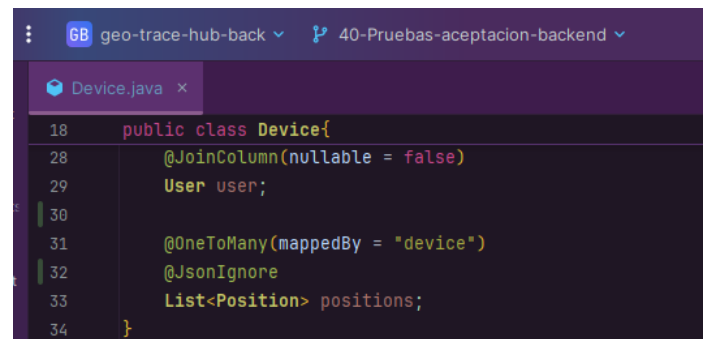
Resultado erróneo del criterio 1 para listar un dispositivo mediante su ID.



Para arreglar esto en la Figura 288 se agrega la anotación `@JsonIgnore` en el modelo “Device”, al atributo `position` el cual por el momento no se necesita en el endpoint la inicialización de las posiciones:

Figura 288

Se añade la anotación `@JsonIgnore` en la clase `Device`.

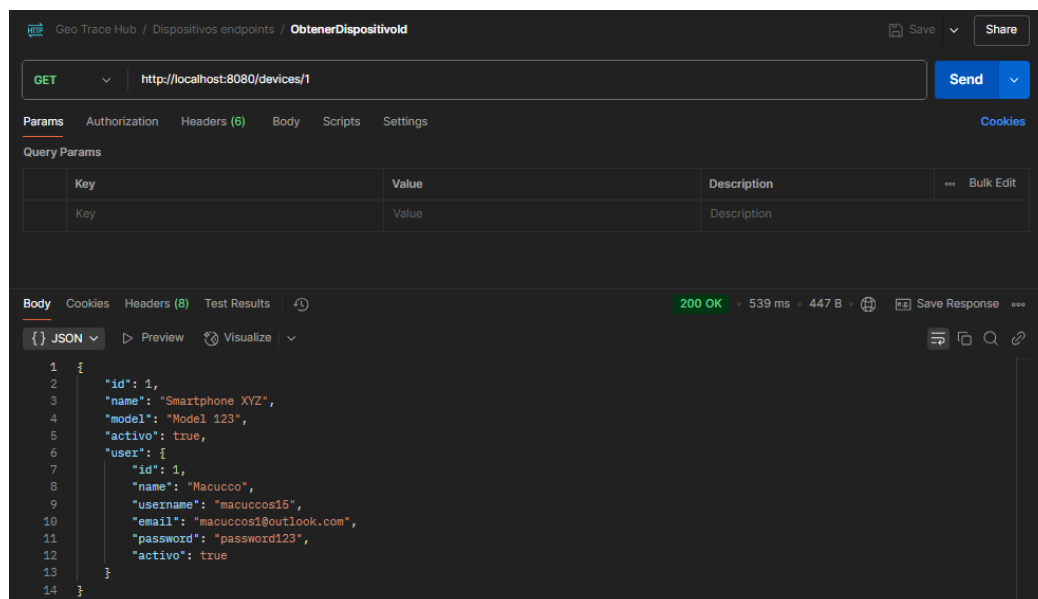


```
18 public class Device{
28     @JoinColumn(nullable = false)
29     User user;
30
31     @OneToMany(mappedBy = "device")
32     @JsonIgnore
33     List<Position> positions;
34 }
```

Al guardar y realizar la petición en la Figura 289 se muestra que en POSTMAN se obtiene una respuesta `200 OK` e imprime los datos del dispositivo juntamente con los del usuario al que le pertenece el dispositivo:

Figura 289

Resultado del criterio 1 para listar un dispositivo mediante su ID.



Geo Trace Hub / Dispositivos endpoints / ObtenerDispositivoId

GET http://localhost:8080/devices/1

Params Authorization Headers (6) Body Scripts Settings Cookies

Query Params

Key	Value	Description
Key	Value	Description

Body Cookies Headers (8) Test Results

200 OK 539 ms 447 B Save Response

JSON Preview Visualize

```
1 {
2   "id": 1,
3   "name": "Smartphone XYZ",
4   "model": "Model 123",
5   "active": true,
6   "user": {
7     "id": 1,
8     "name": "Macucco",
9     "username": "macuccos15",
10    "email": "macuccos1@outlook.com",
11    "password": "password123",
12    "active": true
13  }
14 }
```

El tercer endpoint, permite registrar un dispositivo asociándolo a un usuario indicado en la Tabla 11:

Tabla 11

Estructura de la petición "registro de un dispositivo".

Tipo de petición	Endpoint a probar
POST	http://localhost:8080/devices

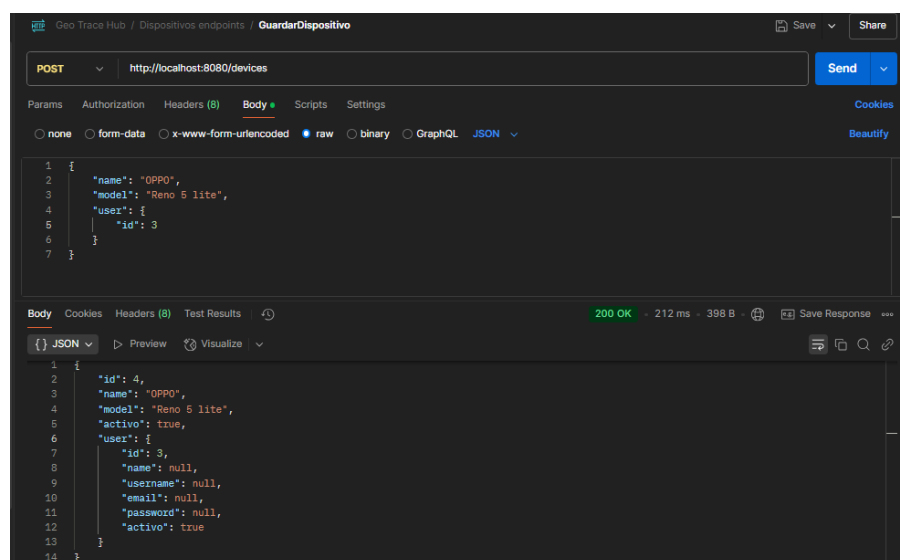
Criterios de aceptación:

1. El sistema debe permitir registrar un dispositivo con nombre del dispositivo y modelo del dispositivo.
 - 1.1. Si el registro es exitoso, debe devolver un código *200 OK* y los datos del dispositivo.
2. Si faltan datos obligatorios, debe devolver un error *400 Bad Request*.

El resultado del criterio 1.1 se guarda el dispositivo con todos los datos, como se aprecia en la Figura 290:

Figura 290

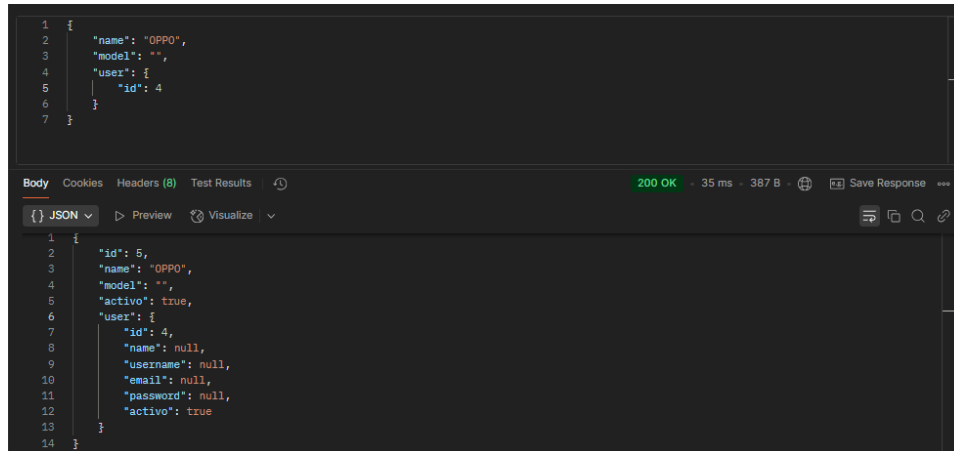
Resultado del criterio 1.1 del registro de un dispositivo.



En cuanto a la prueba del criterio 2, en la Figura 291 se muestra que no se está cumpliendo el criterio, debido a que se guardó un dispositivo sin haber agregado el modelo:

Figura 291

Resultado del criterio 2 del registro de un dispositivo en el que faltaron datos para el registro.



```
1 {
2   "name": "OPPO",
3   "model": "",
4   "user": {
5     "id": 4
6   }
7 }
```

Body Cookies Headers (8) Test Results 200 OK - 35 ms - 387 B Save Response

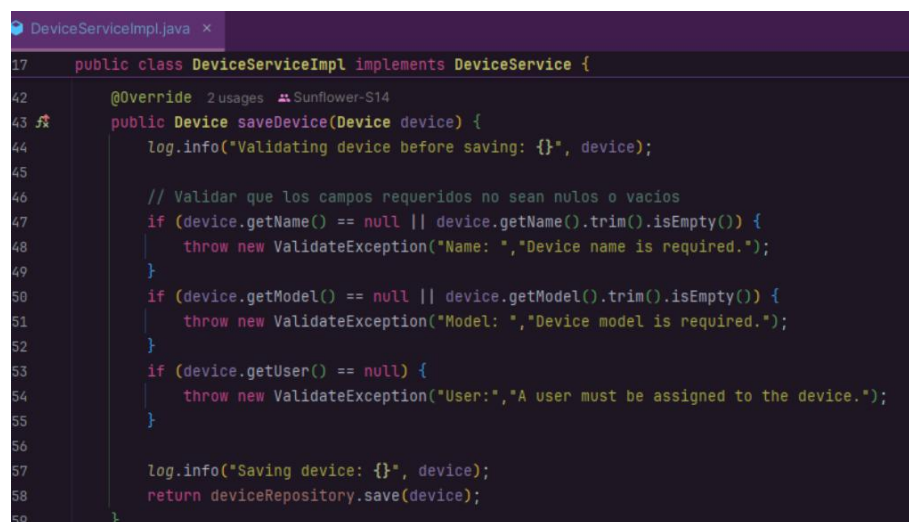
{ JSON Preview Visualize

```
1 {
2   "id": 5,
3   "name": "OPPO",
4   "model": "",
5   "activo": true,
6   "user": {
7     "id": 4,
8     "name": null,
9     "username": null,
10    "email": null,
11    "password": null,
12    "activo": true
13  }
14 }
```

En la Figura 292 se modifica la clase “DeviceServiceImpl” el método “saveDevice” que llama a la excepción “ValidateException” juntamente con condicionales que permitirán validar que los campos requeridos no sean nulos o vacíos:

Figura 292

Uso de la excepción ValidateException en el método saveDevice.

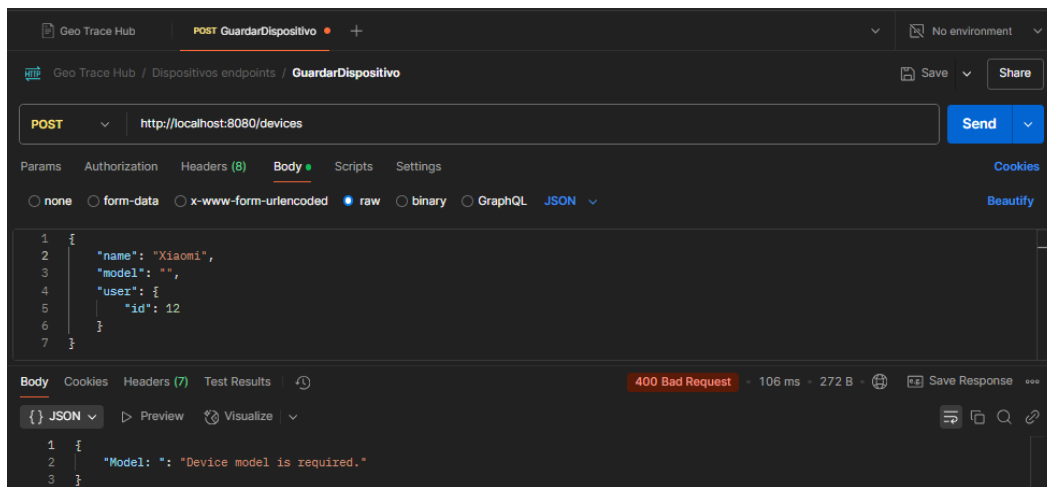


```
DeviceServiceImpl.java x
17 public class DeviceServiceImpl implements DeviceService {
42   @Override 2 usages Sunflower-S14
43   public Device saveDevice(Device device) {
44     log.info("Validating device before saving: {}", device);
45
46     // Validar que los campos requeridos no sean nulos o vacios
47     if (device.getName() == null || device.getName().trim().isEmpty()) {
48       throw new ValidateException("Name: ", "Device name is required.");
49     }
50     if (device.getModel() == null || device.getModel().trim().isEmpty()) {
51       throw new ValidateException("Model: ", "Device model is required.");
52     }
53     if (device.getUser() == null) {
54       throw new ValidateException("User:", "A user must be assigned to the device.");
55     }
56
57     log.info("Saving device: {}", device);
58     return deviceRepository.save(device);
59   }
}
```


Luego de aplicar los cambios e intentar guardar un dispositivo, en la Figura 293 se dejó intencionalmente el campo de modelo sin información, dando como respuesta *400 Bad Request* con un mensaje de que es necesario que se especifique el modelo del dispositivo:

Figura 293

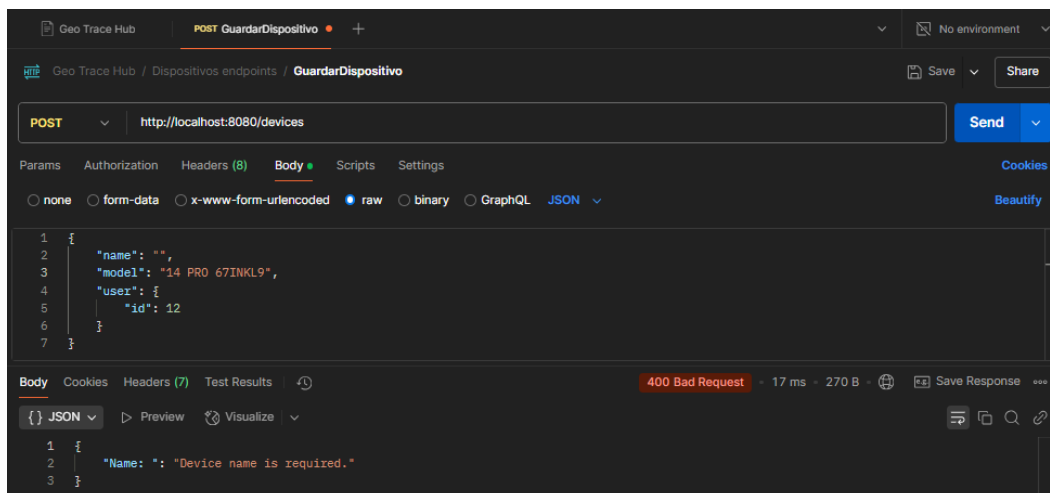
En el criterio 2 se omite el nombre del dispositivo y muestra una advertencia.



Ahora en la Figura 294 se dejó vacío el campo de nombre que responde la petición con una respuesta *400 Bad Request* y el mensaje que indicará que el nombre del dispositivo es requerido:

Figura 294

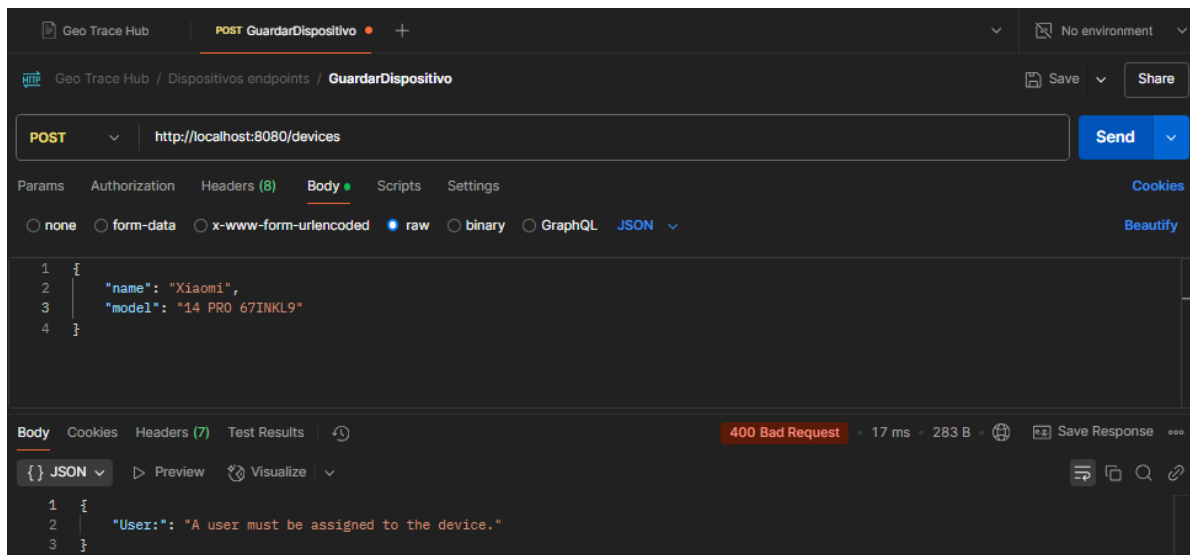
En el criterio 2 se omite el nombre del dispositivo y se muestra un mensaje.



Finalmente en la Figura 295 no se le asigna el dispositivo que se guardará a algún usuario esto permite que responda con *400 Bad Request* con un mensaje que indicará que el usuario se tiene que asignar al dispositivo, así se cumple el criterio 2:

Figura 295

Resultado del criterio 2 en el que se muestran los mensajes correspondientes.



En el cuarto endpoint, permite actualizar un dispositivo, se describe en la Tabla 12:

Tabla 12

Estructura de la petición "actualización de información de un dispositivo".

Tipo de petición	Endpoint a probar
PUT	http://localhost:8080/devices/{id}

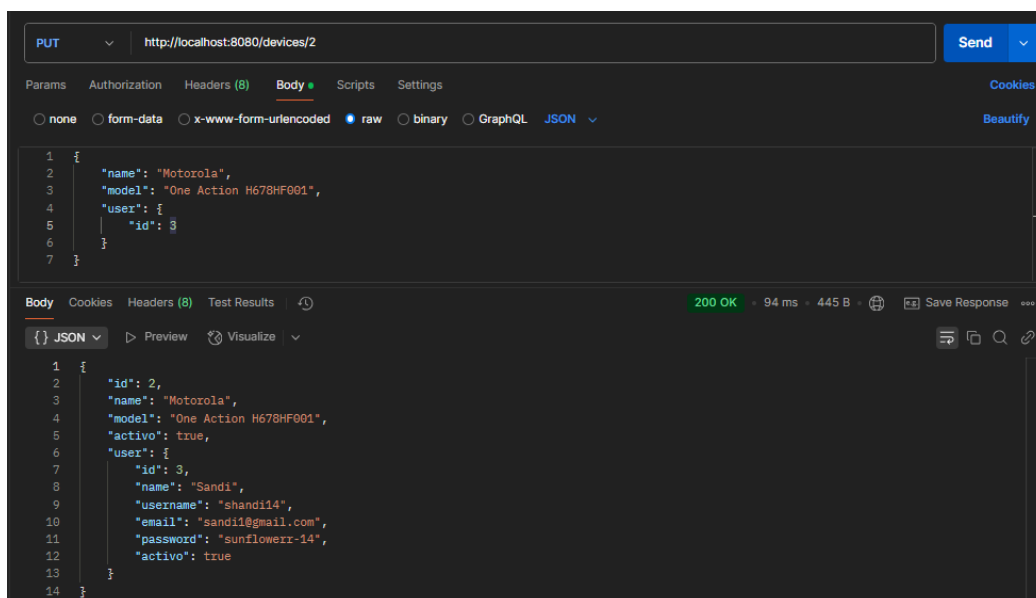
Criterios de aceptación:

1. Se debe poder actualizar un dispositivo existente con nombre del dispositivo, modelo del dispositivo y usuario que le pertenece el dispositivo.
 - 1.1. Si la actualización es exitosa, debe devolver un código *200 OK* y los datos actualizados.
2. Si el dispositivo no existe, debe devolver un error *404 Not Found*.
3. Si solo se actualiza un dato, los datos registrados deben persistir después de la actualización.

Al ejecutar el criterio 1.1 se actualiza un dispositivo de manera correcta devolviendo la respuesta *200 OK* y el cuerpo del JSON con la actualización, puede verse en la Figura 296:

Figura 296

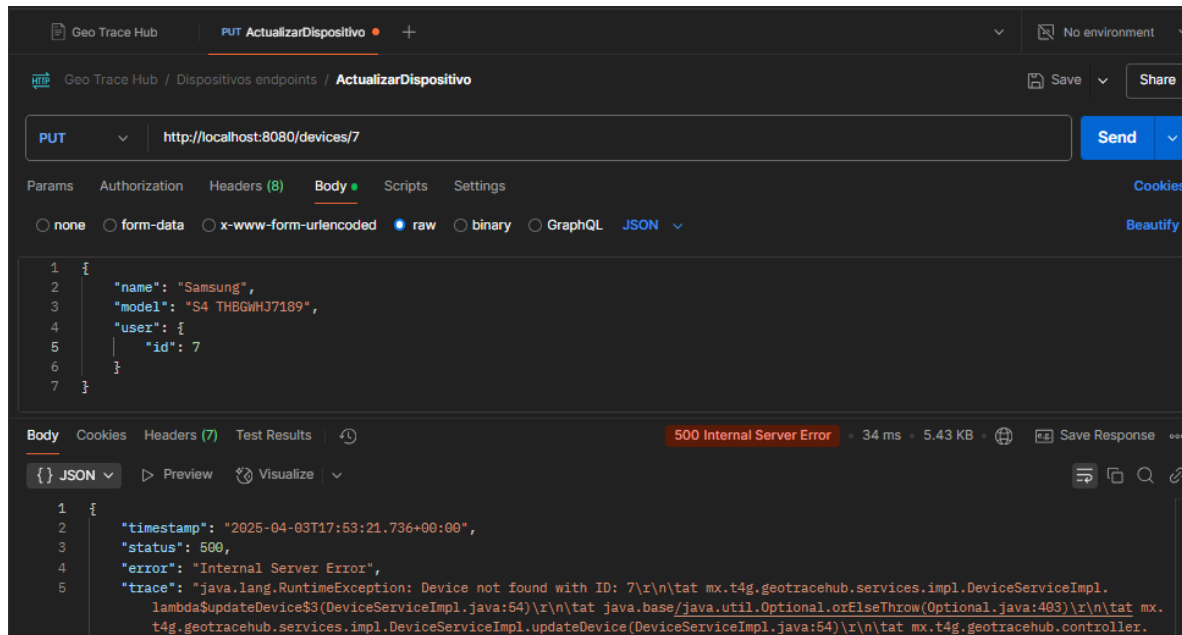
Resultado del criterio 1.1 de la actualización de información de un dispositivo.



En la Figura 297 se muestra que al probar el criterio 2 se obtiene como respuesta *500 Internal Server Error* y no con *404 Bad Request*:

Figura 297

Resultado erróneo del criterio 2 de la actualización de un dispositivo.



Para darle solución a esto, en la Figura 298 se muestra que se crea una nueva clase de excepción que capturará el error al no encontrar el dispositivo indicado:

Figura 298

Clase DeviceNotFoundException.

```
DeviceNotFoundException.java x
1 package mx.t4g.geotracehub.exceptions;
2
3 public class DeviceNotFoundException extends RuntimeException { 4 usages new *
4     public DeviceNotFoundException(String message) { 1 usage new *
5         super(message);
6     }
}
```

En la Figura 299 se maneja la excepción creada en el controlador global de excepciones:

Figura 299

Método `handleDeviceNotFoundException` en `ValidationExceptionHandler`.

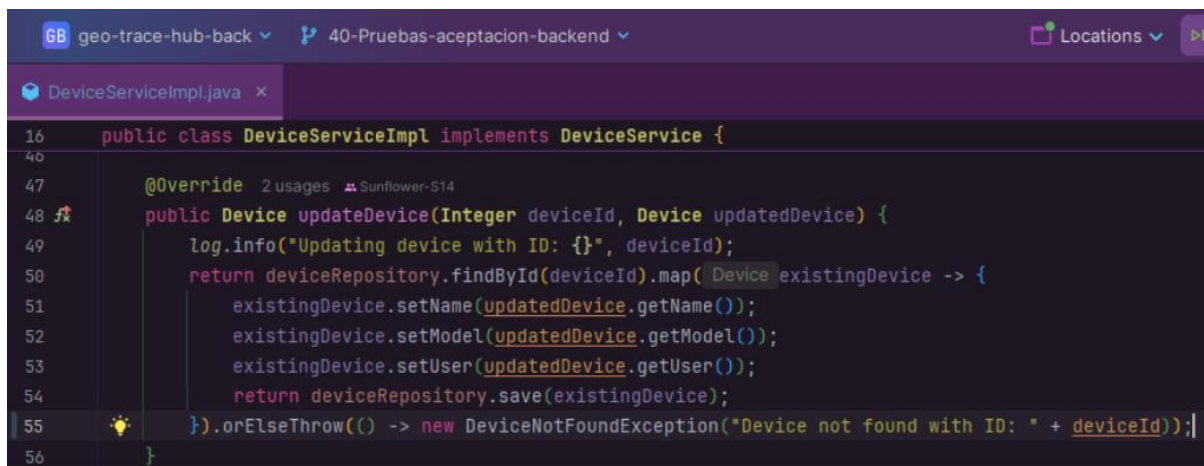


```
16 public class ValidationExceptionHandler {
59 }
60
61 // Manejo de dispositivo no encontrado (404)
62 @ResponseStatus(HttpStatus.NOT_FOUND) no usages new *
63 @ExceptionHandler(DeviceNotFoundException.class)
64 @
65 public Map<String, String> handleDeviceNotFoundException(DeviceNotFoundException ex) {
66     return Collections.singletonMap("error", ex.getMessage());
67 }
```

En la clase `DeviceServiceImpl` se muestra en la Figura 300 la modificación al método `updateDevice` para invocar la excepción:

Figura 300

Se llama la excepción en el método `updateDevice`.

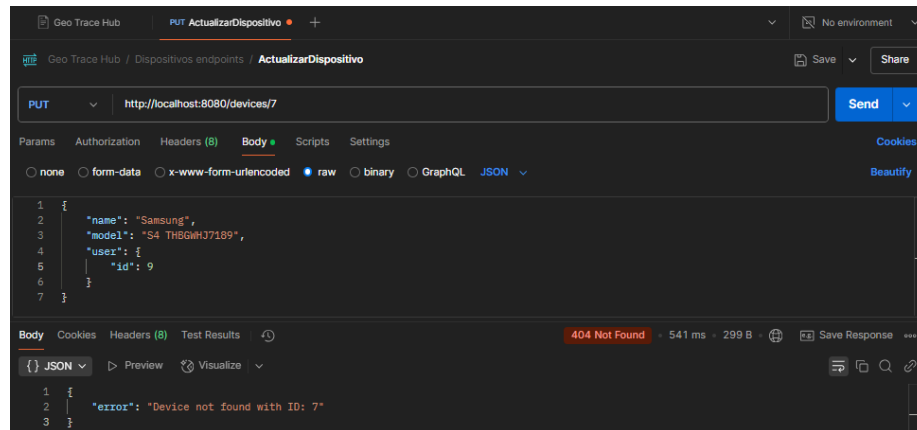


```
16 public class DeviceServiceImpl implements DeviceService {
46
47 @Override 2 usages Sunflower-S14
48 public Device updateDevice(Integer deviceId, Device updatedDevice) {
49     log.info("Updating device with ID: {}", deviceId);
50     return deviceRepository.findById(deviceId).map( Device existingDevice -> {
51         existingDevice.setName(updatedDevice.getName());
52         existingDevice.setModel(updatedDevice.getModel());
53         existingDevice.setUser(updatedDevice.getUser());
54         return deviceRepository.save(existingDevice);
55     }).orElseThrow(() -> new DeviceNotFoundException("Device not found with ID: " + deviceId));
56 }
```

Al volver a ejecutar, se cumple el criterio de aceptación en donde se muestra una respuesta 404 y un mensaje indicando que el dispositivo no existe, este resultado se muestra en la Figura 301:

Figura 301

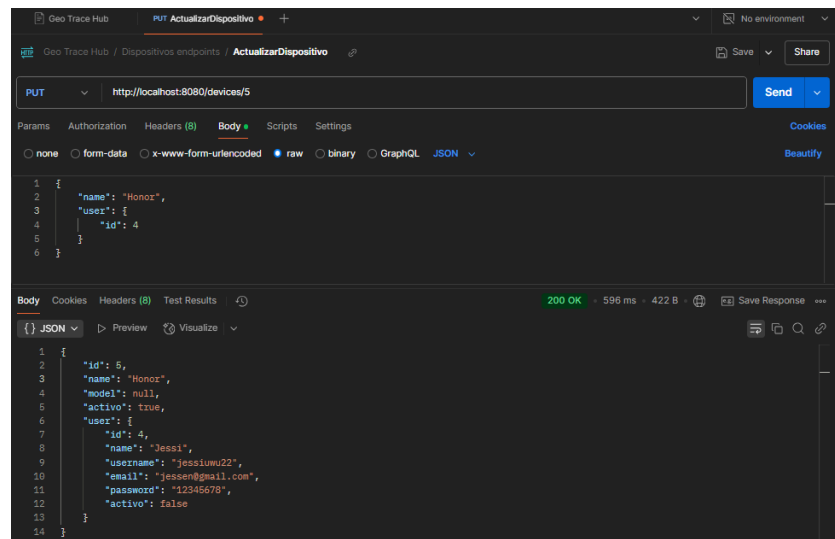
Resultado del criterio 2 de la actualización de información de un dispositivo.



En la Figura 302 se aprecia que el criterio 3 no se cumple debido a que solo se indicó el nombre del dispositivo, se omitió el modelo del dispositivo, pero toma por nulo el campo del modelo del dispositivo que previamente tenía datos y ahora son de tipo *null*:

Figura 302

Resultado no esperado del criterio 3 de la actualización de un dispositivo.



En la Figura 303 se resuelve mediante la modificación de la clase “DeviceServiceImpl” en el método “updateDevice” se agregan condiciones que permitan verificar si un campo viene de tipo *null* no lo actualiza y mantiene el valor anterior, permitiendo solo sobrescribir los valores que realmente se especificaron en el formato JSON:

Figura 303

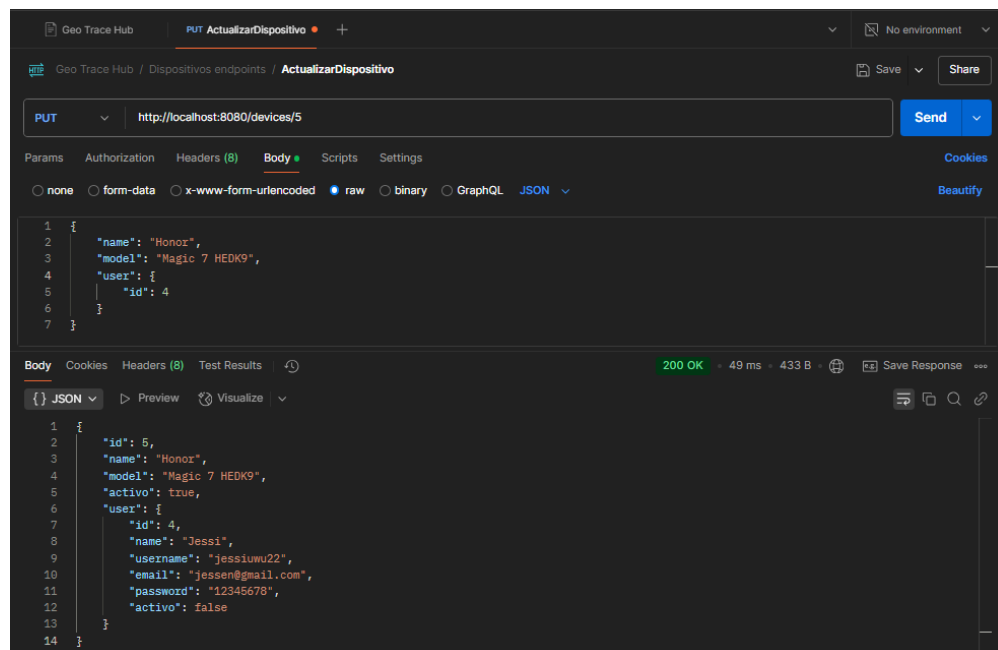
Se agregan validaciones en el método `updateDevice`.

```
DeviceServiceImpl.java x
16 public class DeviceServiceImpl implements DeviceService {
47     @Override 2 usages Sunflower-S14
48     public Device updateDevice(Integer deviceId, Device updatedDevice) {
49         log.info("Updating device with ID: {}", deviceId);
50
51         return deviceRepository.findById(deviceId).map(Device existingDevice -> {
52             // Solo actualiza si el nuevo valor no es null
53             if (updatedDevice.getName() != null) {
54                 existingDevice.setName(updatedDevice.getName());
55             }
56             if (updatedDevice.getModel() != null) {
57                 existingDevice.setModel(updatedDevice.getModel());
58             }
59             if (updatedDevice.getUser() != null) {
60                 existingDevice.setUser(updatedDevice.getUser());
61             }
62             return deviceRepository.save(existingDevice);
63         }).orElseThrow(() -> new DeviceNotFoundException("Device not found with ID: " + deviceId));
64     }
}
```

Antes de ejecutar la petición, en la Figura 304 primero se agregan los datos completos para mostrar el contraste, en donde se agregan todos los datos:

Figura 304

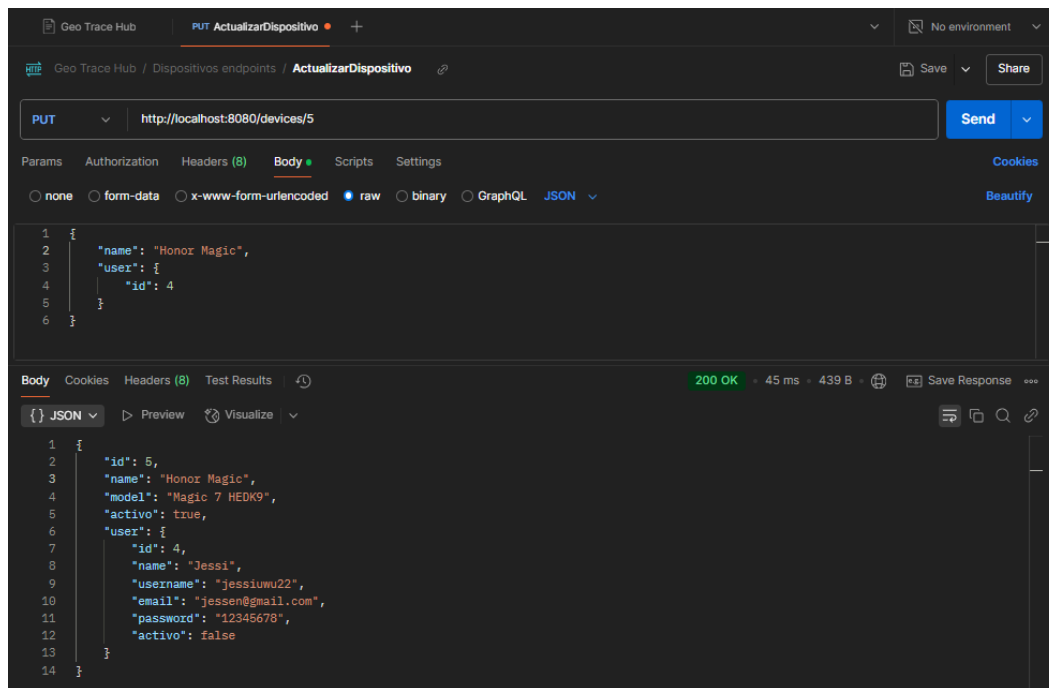
Se añaden todos los datos nuevos en la actualización.



En la Figura 305 se ilustra que solo se actualiza el “nombre del dispositivo”. Como resultado se obtiene de respuesta que se muestra en el campo de “modelo” en dónde no se especificó en la actualización su valor persistió y no se volvió nulo, así se cumplió el criterio 3:

Figura 305

Resultado del criterio 3 de la actualización de información de un dispositivo.



En la Tabla 13 se plasma el quinto endpoint, que sirve para desactivar o activar un dispositivo:

Tabla 13

Estructura de la petición "cambiar estado de un dispositivo".

Tipo de petición	Endpoint a probar
PATCH	http://localhost:8080/users/1/status?activo=false
	http://localhost:8080/users/1/status?activo=true

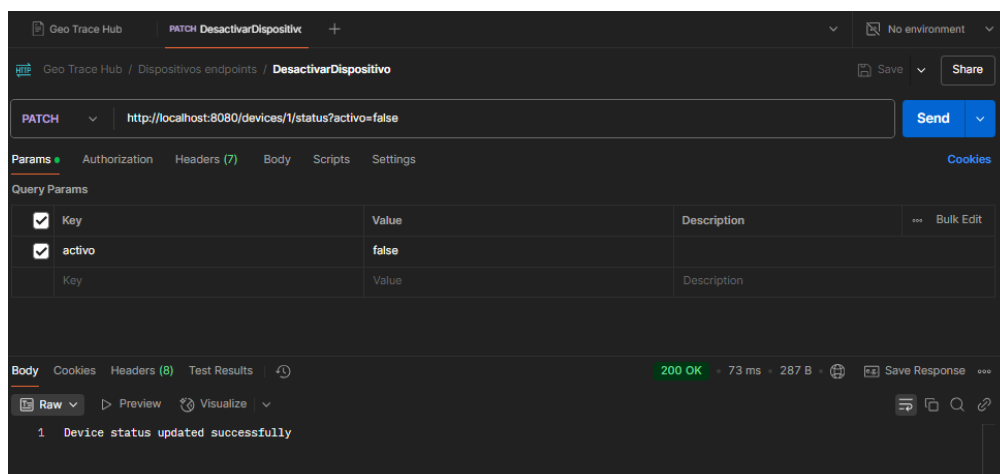
Criterios de aceptación:

1. El sistema debe permitir activar o desactivar un dispositivo existente.
 - 1.1. Si el estado se cambia correctamente, debe devolver *200 OK*.
2. Si el dispositivo no existe, debe devolver *404 Not Found*.

El criterio 1.1 se cumple adecuadamente en la Figura 306 mostrando una respuesta *200 OK* y con el mensaje de que se ha actualizado el estado del dispositivo:

Figura 306

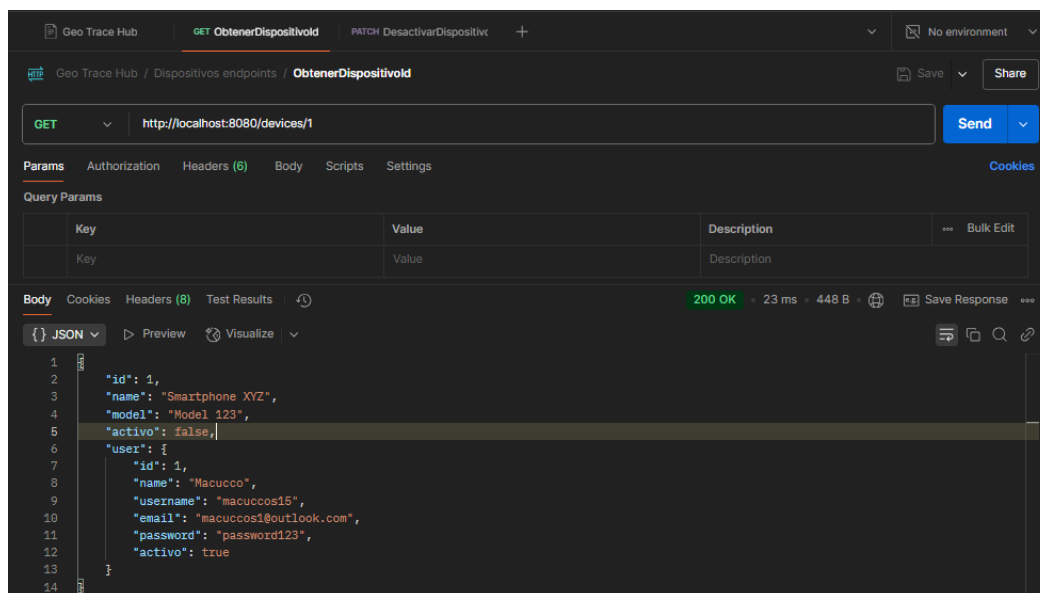
Resultado del criterio 1.1 para cambiar estado de un dispositivo



En la Figura 307 se visualiza el endpoint de listar dispositivo por ID que muestra el campo “activo” en “false”:

Figura 307

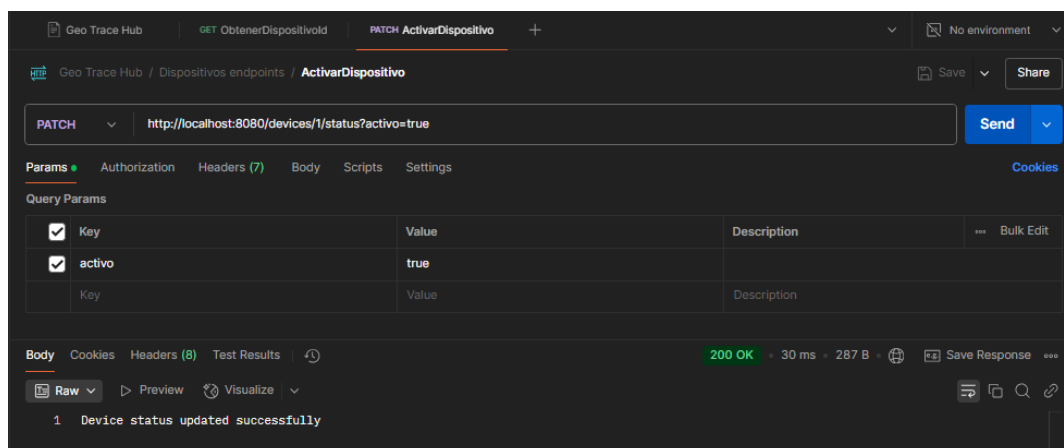
Listado de la información de un dispositivo con el estado en falso.



En la Figura 308 se activa nuevamente el dispositivo. Se pasa el parámetro a “true” para volver activar el dispositivo y al ejecutar responde con un **OK 200** indicando que el estado del dispositivo se actualizo correctamente:

Figura 308

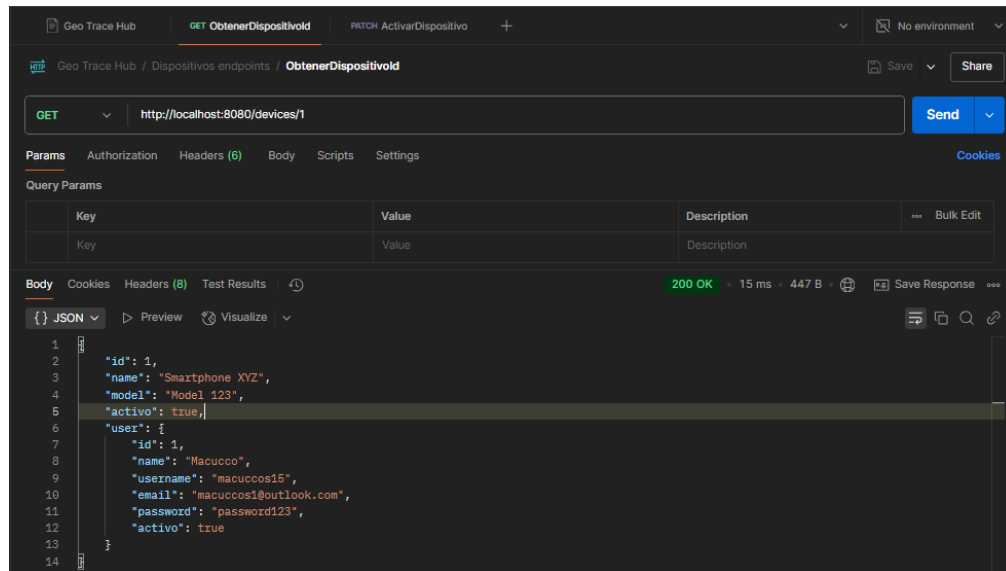
Activación de un dispositivo en estado verdadero.



Se puede mirar en la Figura 309 que al buscar el dispositivo el campo “activo” vuelve a ser “true”:

Figura 309

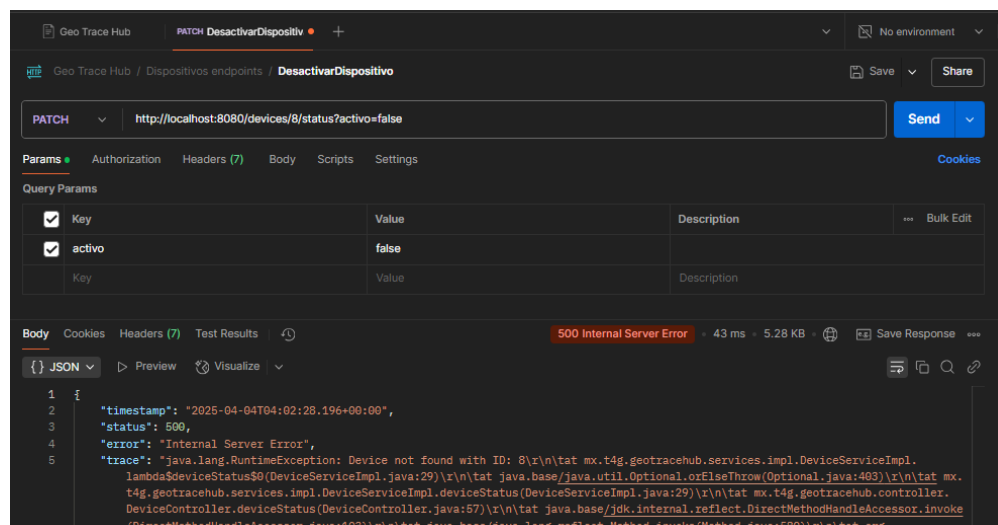
Listado de la información de un dispositivo con el estado en verdadero.



En la Figura 310 se prueba el endpoint con un dispositivo que no existe haciendo que el criterio 2 no se cumple debido a que se está lanzando un error `500` y no un error `404 Not Found`:

Figura 310

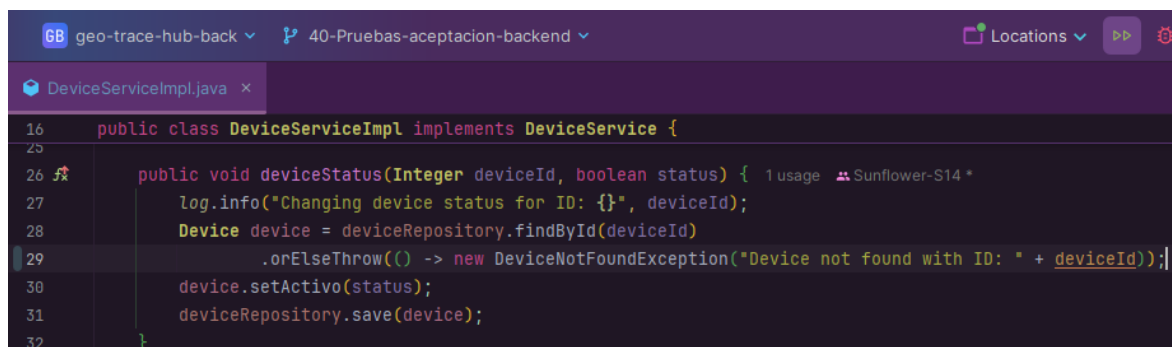
Resultado erróneo del criterio 2 de cambiar estado de un dispositivo.



En la Figura 311 de la clase “DeviceServiceImpl” se invoca la excepción “DeviceNotFoundException” previamente creada:

Figura 311

Uso de DeviceNotFound en deviceStatus de la clase DeviceServiceImpl.

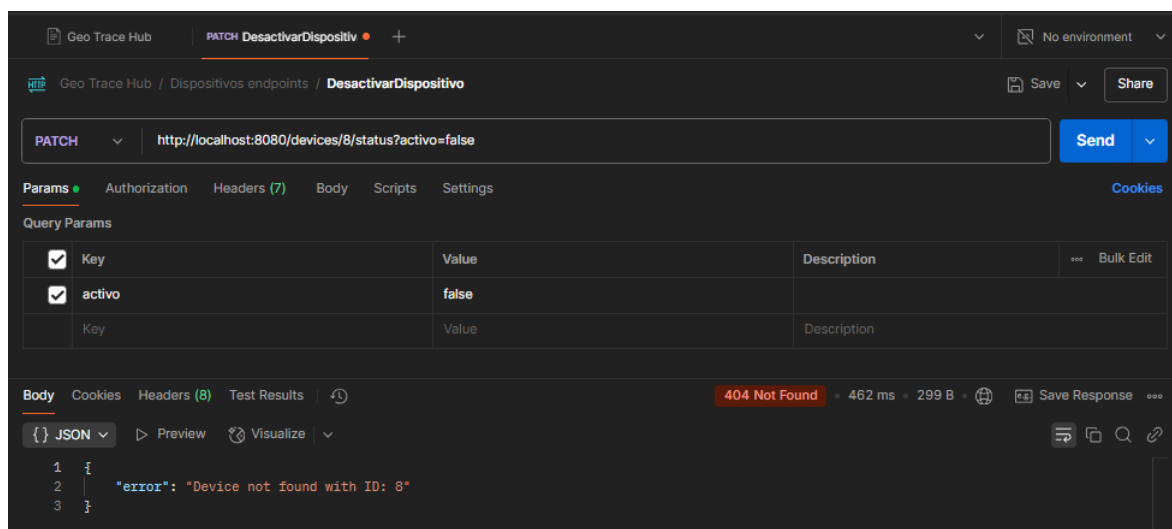


```
16 public class DeviceServiceImpl implements DeviceService {
25
26     public void deviceStatus(Integer deviceId, boolean status) { 1 usage Sunflower-S14 *
27         log.info("Changing device status for ID: {}", deviceId);
28         Device device = deviceRepository.findById(deviceId)
29             .orElseThrow(() -> new DeviceNotFoundException("Device not found with ID: " + deviceId));
30         device.setActivo(status);
31         deviceRepository.save(device);
32     }
```

En la Figura 312 se prueba el endpoint para desactivar el dispositivo en el que devuelve una respuesta *404 Not Found*, mandando un mensaje de error indicando que no existe el dispositivo, cumpliendo así con el criterio de aceptación 2:

Figura 312

Resultado del criterio 2 en cambiar estado de un dispositivo no existente.



Geo Trace Hub / PATCH DesactivarDispositivo

Geo Trace Hub / Dispositivos endpoints / DesactivarDispositivo

PATCH http://localhost:8080/devices/8/status?activo=false

Params

Key	Value	Description
activo	false	

Body

404 Not Found • 462 ms • 299 B

```
{
  "error": "Device not found with ID: 8"
}
```

El sexto endpoint, tiene la función para eliminar un dispositivo, mostrada en la Tabla 14:

Tabla 14

Estructura de la petición "eliminación de un dispositivo"

Tipo de petición	Endpoint a probar
DELETE	http://localhost:8080/devices/{id}

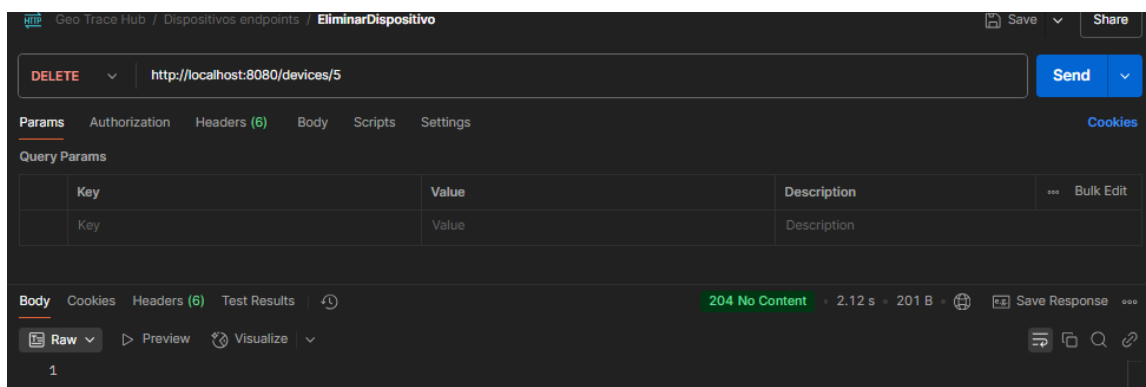
Criterios de aceptación:

1. Se debe poder eliminar un dispositivo existente.
 - 1.1. Si la eliminación es exitosa, debe devolver *204 No Content*.
2. Si el dispositivo no existe, debe devolver *404 Not Found*.
3. No debe permitir eliminar un dispositivo con ubicaciones registradas, en cuyo caso debe devolver *409 Conflict*.

Se visualiza en la Figura 313 que al probar el criterio 1.1, al eliminar un dispositivo se recibe la respuesta *204 No Content* que significa que el dispositivo de ejemplo, con ID 5 se eliminó correctamente:

Figura 313

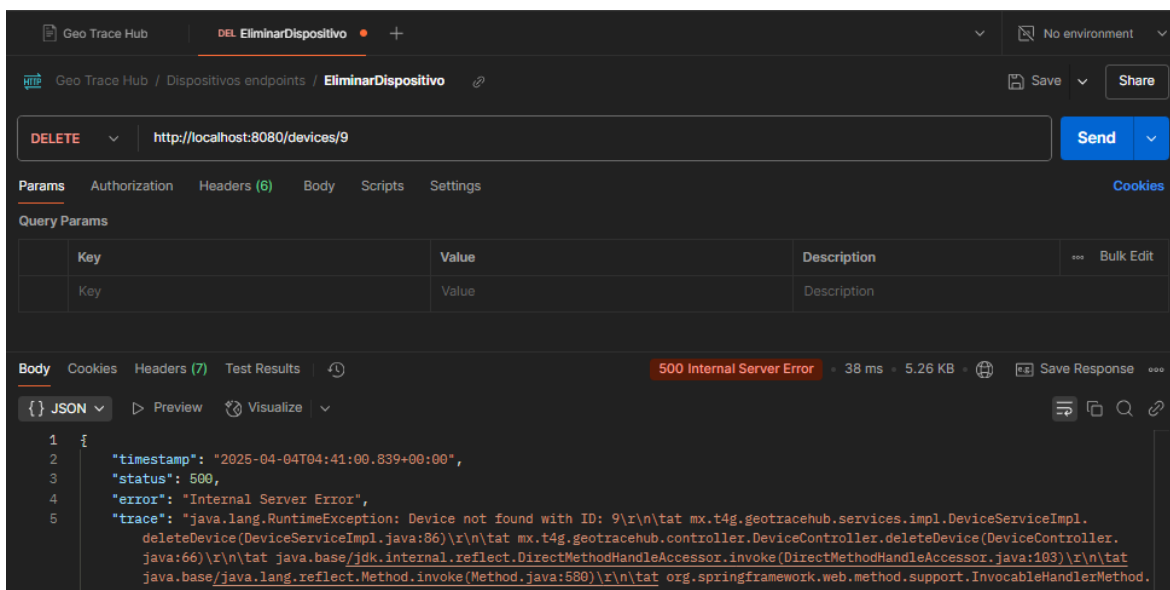
Resultado del criterio 1.1 de la eliminación de un dispositivo.



En el criterio 2 no se cumple ya que se está intentando eliminar un dispositivo que no existe y devuelve una respuesta *500 Internal Server Error* capturado en la Figura 314:

Figura 314

Resultado erróneo del criterio 2 de la eliminación de un dispositivo.



La solución se muestra en la Figura 315 que modifica la clase “`DeviceServiceImpl`” para llamar a la excepción “`DeviceNotFoundException`” en el método `deleteDevice`:

Figura 315

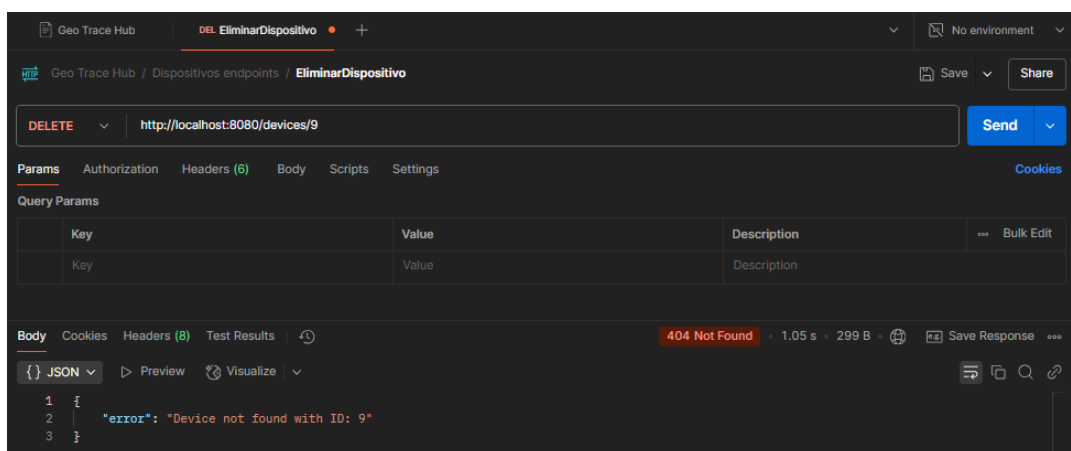
Uso de `DeviceNotFoundException` en el método `deleteDevice`.



Después de guardar así como de realizar la petición se recibe una respuesta *404 Not Found*, adicionalmente con un mensaje indicando que el dispositivo no se encontró, cumpliendo así con el criterio 2 representado de forma esquemática 316:

Figura 316

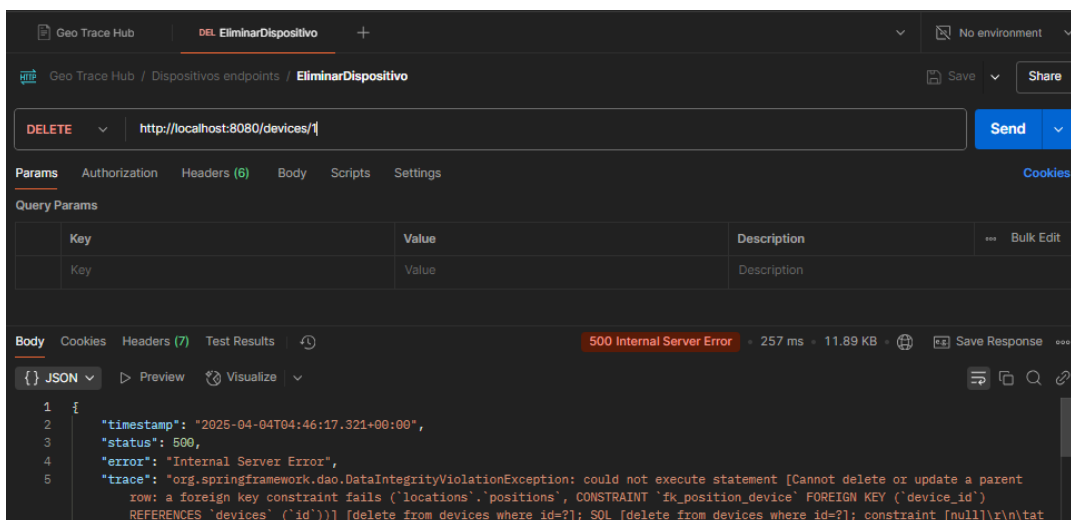
Resultado del criterio 2 de la eliminación de un dispositivo.



En cuanto al criterio 3, no se está cumpliendo y está devolviendo una respuesta *500 Internal Server Error* al intentar eliminar un dispositivo con ubicaciones guardadas ilustrada en la Figura 317:

Figura 317

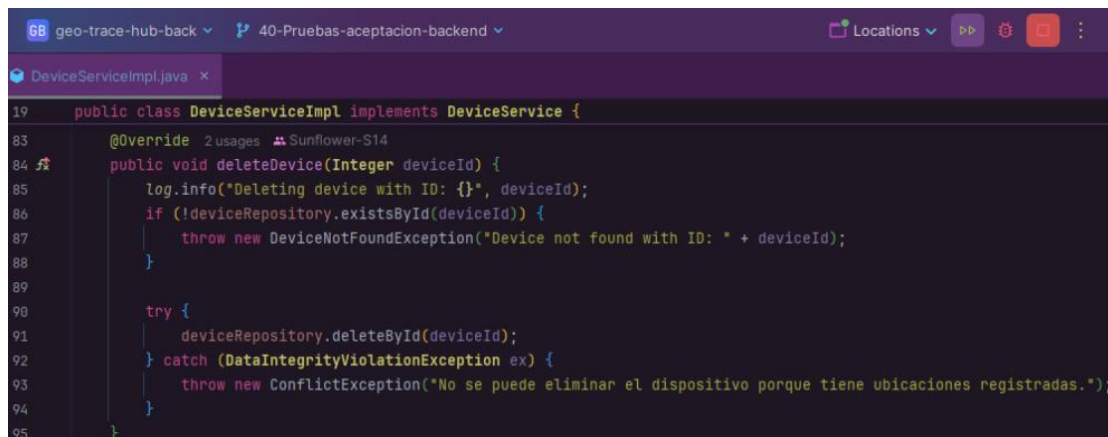
Resultado erróneo del criterio 3 en la eliminación de un dispositivo.



Este problema se resuelve de la Figura 318 en el cual se modifica en el método deleteDevice se usa un try-catch para atrapar la excepción “DataIntegrityViolationException” para manejar los errores relacionados con la eliminación en llaves foráneas:

Figura 318

Modificación del método deleteDevice de la clase DeviceServiceImpl.

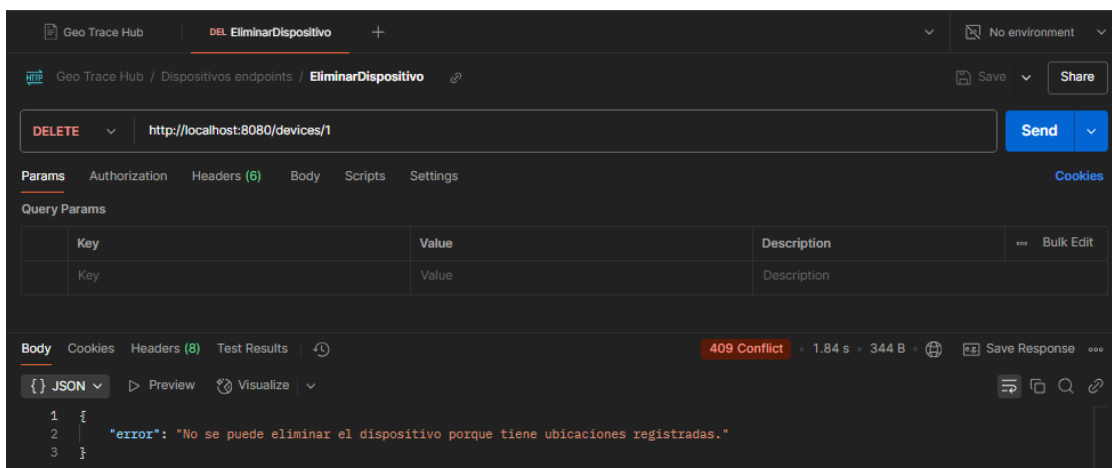


```
19 public class DeviceServiceImpl implements DeviceService {
83
84     @Override 2 usages Sunflower-S14
85     public void deleteDevice(Integer deviceId) {
86         log.info("Deleting device with ID: {}", deviceId);
87         if (!deviceRepository.existsById(deviceId)) {
88             throw new DeviceNotFoundException("Device not found with ID: " + deviceId);
89         }
90
91         try {
92             deviceRepository.deleteById(deviceId);
93         } catch (DataIntegrityViolationException ex) {
94             throw new ConflictException("No se puede eliminar el dispositivo porque tiene ubicaciones registradas.");
95         }
96     }
97 }
```

En la Figura 319 se ejecuta de nuevo la prueba que obtiene una respuesta *409 Conflict* acompañado de un mensaje que indica que no se puede eliminar el dispositivo porque tiene ubicaciones registradas, así cumpliendo con el criterio de aceptación 3:

Figura 319

Resultado del criterio 3 de la eliminación de un dispositivo.



Geo Trace Hub / Dispositivos endpoints / EliminarDispositivo

DELETE http://localhost:8080/devices/1

Params Authorization Headers (6) Body Scripts Settings Cookies

Query Params

Key	Value	Description
Key	Value	Description

Body Cookies Headers (8) Test Results

409 Conflict 1.84 s 344 B

JSON Preview Visualize

```
1 {
2   "error": "No se puede eliminar el dispositivo porque tiene ubicaciones registradas."
3 }
```


La Tabla 15 ofrece la estructura del último endpoint que permite invocar los dispositivos que tiene un usuario:

Tabla 15

Estructura de la petición “mostrar los dispositivos que tiene un usuario”.

Tipo de petición	Endpoint a probar
GET	http://localhost:8080/devices/user/{userId}

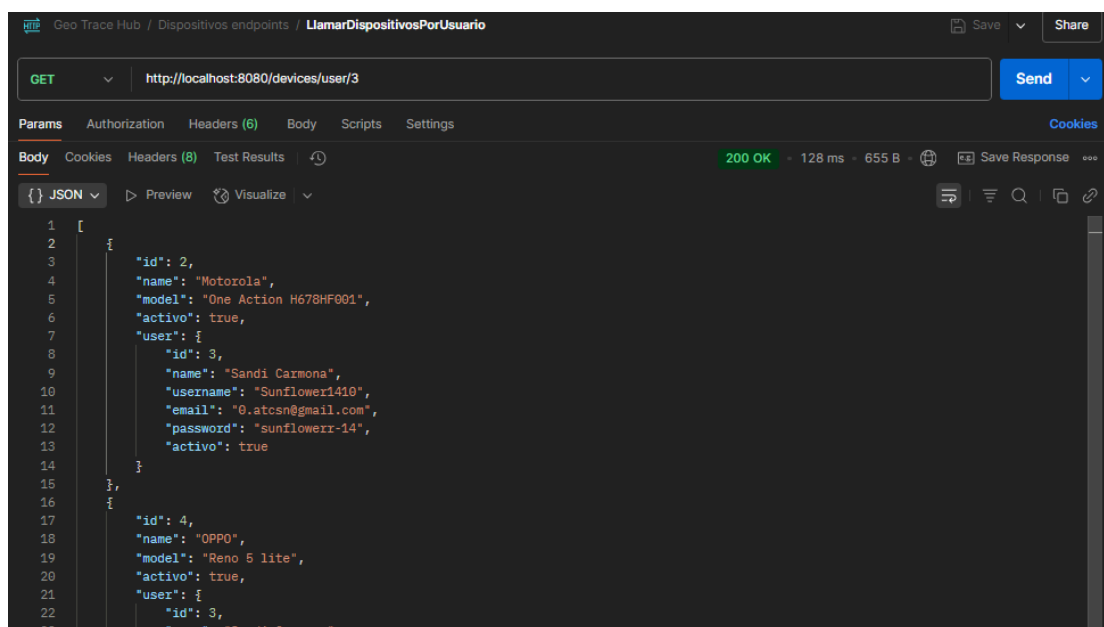
Criterios de aceptación:

1. Si el usuario con el ID especificado existe y tiene dispositivos asociados debe devolver código *200 OK* y una lista de dispositivos en formato JSON con sus atributos del dispositivo
2. Si el usuario existe pero no tiene dispositivos asignados, deberá devolver una respuesta *404 Not Found* con un mensaje indicando que no se encuentran dispositivos registrados.
3. Si el ID del usuario no corresponde a ningún usuario registrado, devolverá código *404 Not Found* con un mensaje de error: "Usuario no encontrado".
4. Si se proporciona un ID que no es un número entero positivo, mostrará código *400 Bad Request* con Mensaje: "ID de usuario debe ser válido".

El criterio 1 se cumple en la Figura 320 que al ejecutar el endpoint se especifica el ID del usuario y la respuesta que devuelve es una lista de dispositivos, en esta prueba solo muestra dos dispositivos que le corresponden al usuario:

Figura 320

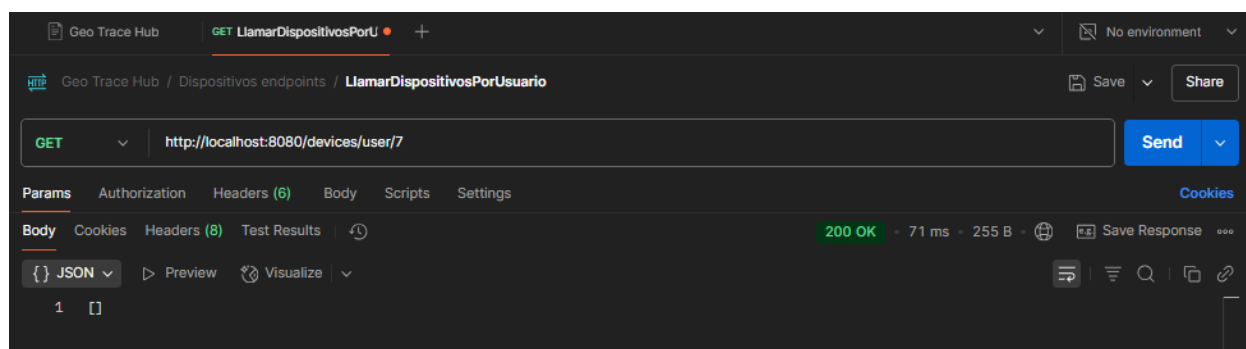
Resultado del criterio 1 en mostrar los dispositivos que tiene un usuario.



El criterio 2 no se cumple en la Figura 321 debido a que debería de retornar una respuesta 404 *Not Found* y no un 200 OK, aunque este enfoque es correcto, para cumplir con el criterio deberá de mostrar un mensaje de error:

Figura 321

Resultado inesperado del criterio 2 en la mostrar los dispositivos de un usuario.



En la Figura 322 se realiza el cambio en la clase “DeviceServiceImpl” se agrega una condición; si la lista es vacía lanzará la excepción personalizada en el que se especifica el mensaje de error:

Figura 322

Se llama a DeviceNotFoundException al método getDevicesById.

```
public List<Device> getDevicesById(Integer userId) { 1 usage Sunflower-S14
    log.info("Buscando dispositivos para el usuario ID: {}", userId);

    if (userId == null || userId <= 0) {
        throw new ValidateException("userId", "El ID de usuario debe ser válido");
    }

    List<Device> devices = deviceRepository.findById(userId);

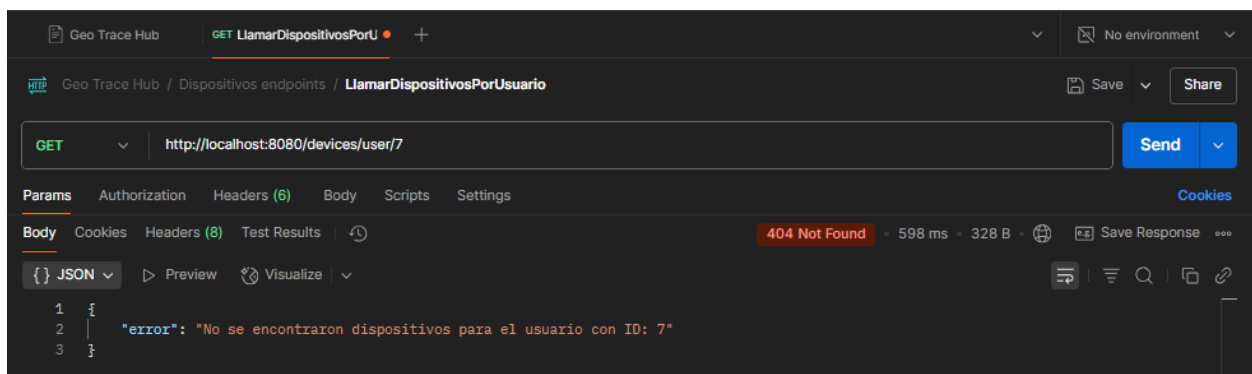
    if (devices.isEmpty()) {
        throw new DeviceNotFoundException("No se encontraron dispositivos para el usuario con ID: " + userId);
    }

    return devices;
}
```

Al volver a ejecutar se puede visualizar en la Figura 323 el mensaje de error que indica que ese usuario no tiene dispositivos, esto permite identificar rápidamente que usuarios tienen dispositivos y cuales no los tienen:

Figura 323

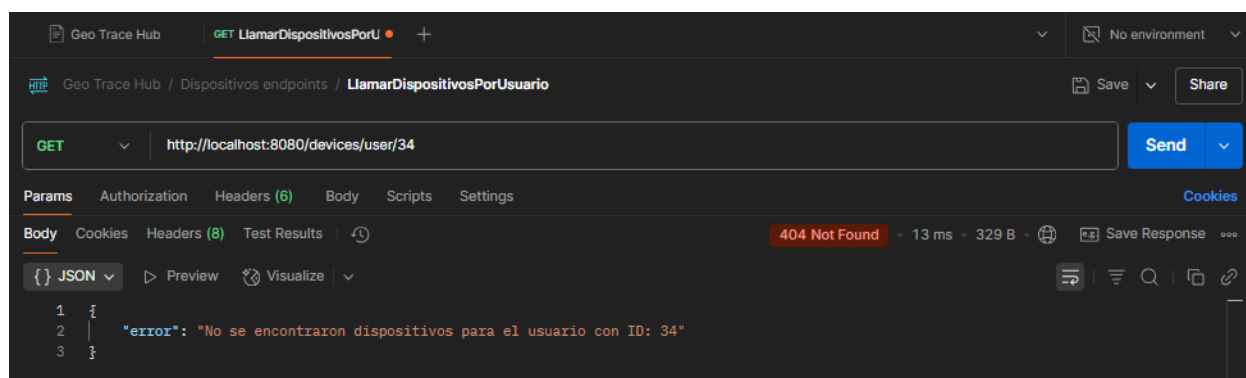
Resultado del criterio 2 en mostrar los dispositivos que tiene un usuario.



En el criterio 3 es plasmado en la Figura 324 que llama al endpoint con un ID de usuario que no está registrado, usa la excepción creada para el criterio 2 el cual permite que este criterio también se cumpla, debido a que muestra un mensaje de error claro:

Figura 324

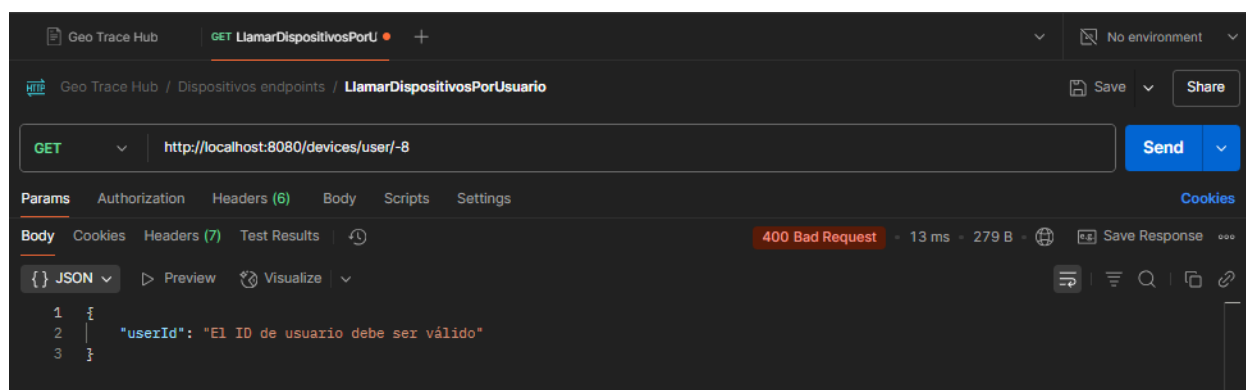
Resultado del criterio 3 para mostrar los dispositivos que tiene un usuario.



Por último, el criterio 4 se cumple en la Figura 325 al asignarle al endpoint un número negativo y muestra un mensaje indicando que el número no es válido:

Figura 325

Resultado del criterio 4 para mostrar los dispositivos que tiene un usuario.



Administración de ubicaciones

El primer endpoint, permite obtener todas las ubicaciones de un dispositivo, puede consultarse en la Tabla 16:

Tabla 16

Estructura de la petición "invocar todas las ubicaciones de un dispositivo".

Tipo de petición	Endpoint a probar
GET	http://localhost:8080/device/{id}/positions

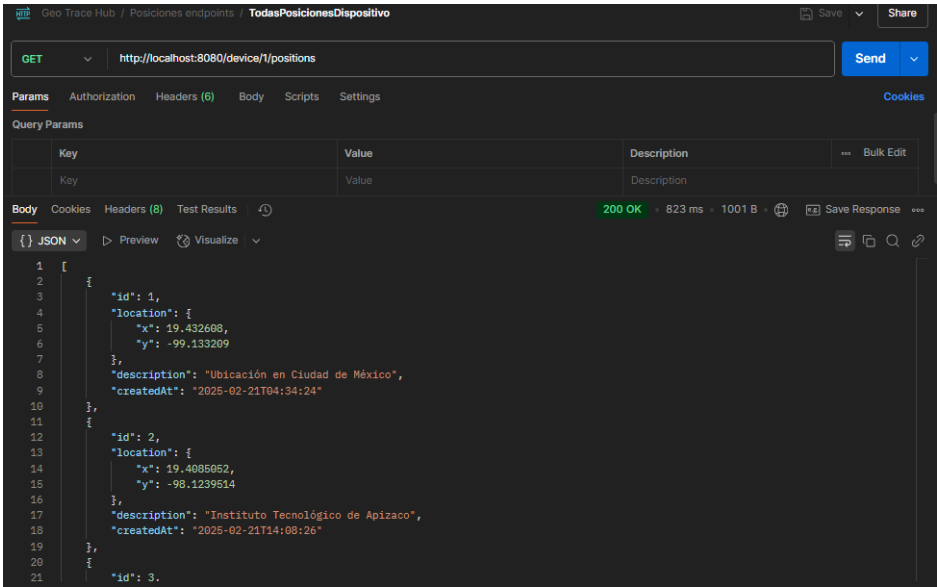
Criterios de aceptación:

1. Se deben listar todas las ubicaciones asociadas a un dispositivo con una respuesta *200 OK*.
 - 1.1. La respuesta debe incluir id, la ubicación en coordenadas "x" y "y", descripción de la ubicación y mostrará la fecha y hora en la que se guardó la posición devolviendo así una respuesta con *200 OK*.
2. Si el dispositivo no existe, debe devolver *404 Not Found*.
3. Si no hay ubicaciones registradas, debe devolver una lista vacía con *200 OK*.

El resultado del criterio 1.1 muestra en la Figura 326 un listado de las ubicaciones guardadas en un dispositivo:

Figura 326

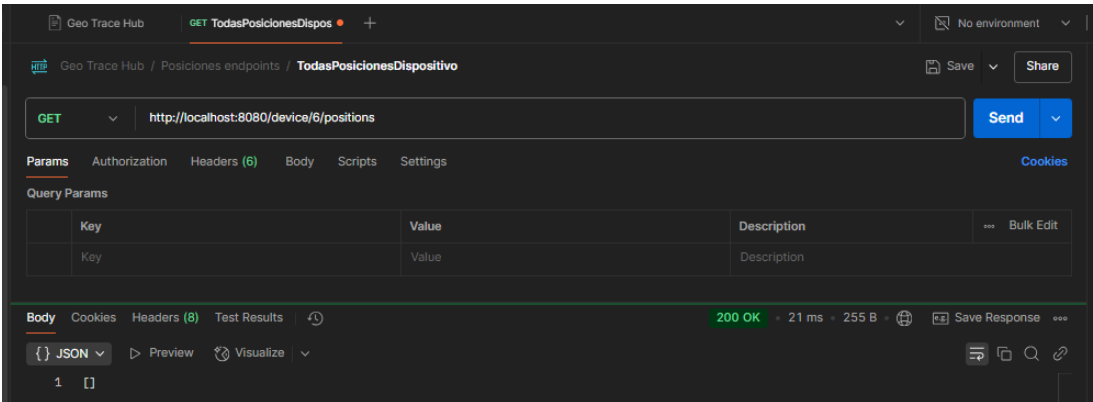
Resultado del criterio 1 para invocar todas las ubicaciones de un dispositivo.



El resultado del criterio 2 no se está cumpliendo en la Figura 327 debido a que lanza una respuesta `200 OK` que muestra una lista vacía y esto puede ser confuso debido a que se puede confundir con un dispositivo existente pero sin posiciones:

Figura 327

Resultado inesperado del criterio 2 para obtener todas las ubicaciones almacenadas en un dispositivo.



En la Figura 328 se ejemplifica que para corregirlo se llama a la excepción en la clase “PositionsServicesImpl” en el método “getPointsByDevice” para manejar cuando no se encuentra un dispositivo:

Figura 328

En getPointByDevice usa la excepción DeviceNotFoundException.

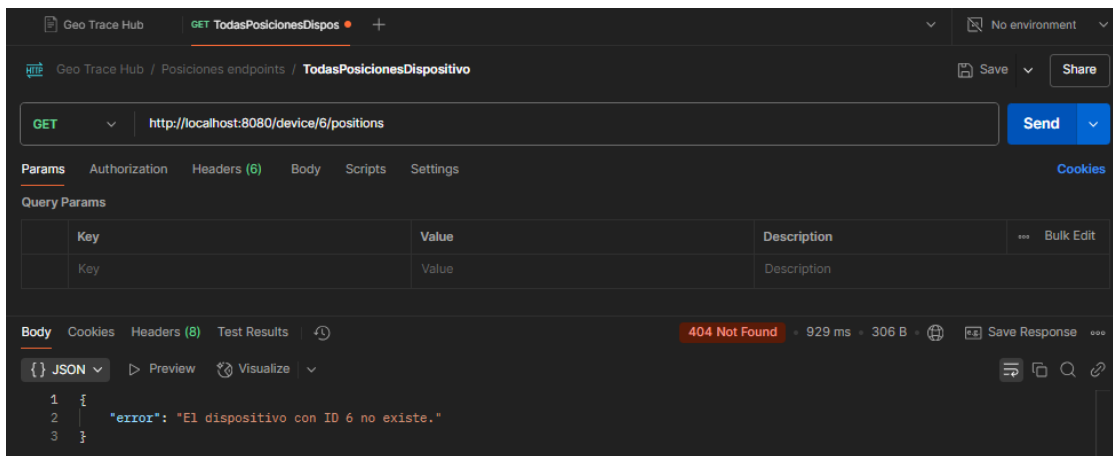


```
18 public class PositionsServiceImpl implements PositionsService {
22
23     @
24     public List<Position> getPointsByDevice(Device device) { 2 usages  Sunflower-S14
25         if (!dispositivoRepository.existsById(device.getId())) {
26             throw new DeviceNotFoundException("El dispositivo con ID " + device.getId() + " no existe.");
27         }
28
29         return positionsRepository.getAllByDevice(device);
30     }
```

Al volver a realizar la petición se cumple el criterio al mostrar una respuesta *404 Not Found* y el mensaje de que no se encontró el dispositivo porque no existe, representado en la Figura 329:

Figura 329

Resultado del criterio 2 para invocar todas las ubicaciones de un dispositivo.



Geo Trace Hub GET TodasPosicionesDispos + No environment

Geo Trace Hub / Posiciones endpoints / TodasPosicionesDispositivo Save Share

GET http://localhost:8080/device/6/positions Send

Params Authorization Headers (6) Body Scripts Settings Cookies

Query Params

Key	Value	Description	Bulk Edit
Key	Value	Description	

Body Cookies Headers (8) Test Results 404 Not Found 929 ms 306 B Save Response

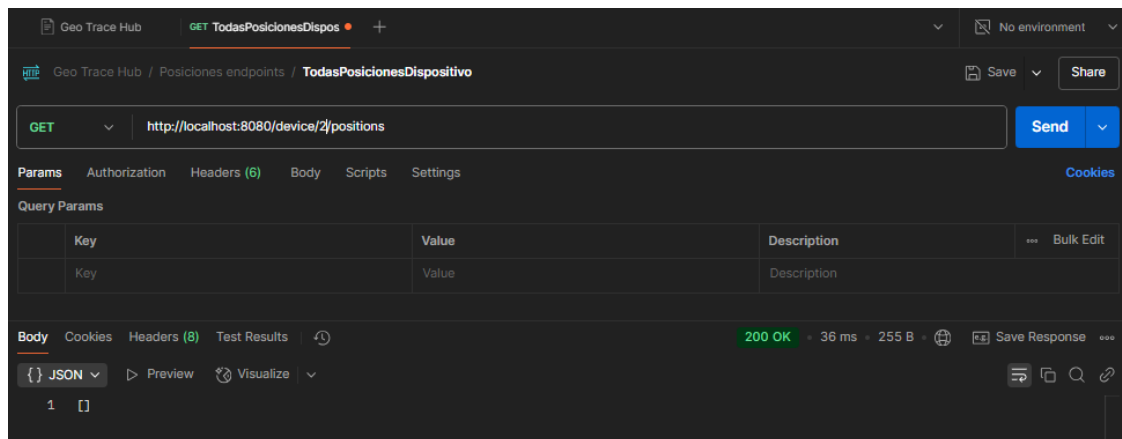
JSON Preview Visualize

```
1 {
2   "error": "El dispositivo con ID 6 no existe."
3 }
```

Por último, el resultado del criterio 3 se muestra en la Figura 330 que imprime una lista vacía cuando un dispositivo no tiene posiciones almacenadas, cumpliendo así con el criterio de aceptación:

Figura 330

Resultado del criterio 3 para invocar todas las ubicaciones de un dispositivo.



El segundo endpoint, tiene la funcionalidad de guardar una ubicación, reflejada en la Tabla 17:

Tabla 17

Estructura de la petición "almacenar ubicación".

Tipo de petición	Endpoint a probar
POST	http://localhost:8080/device/{id}/positions

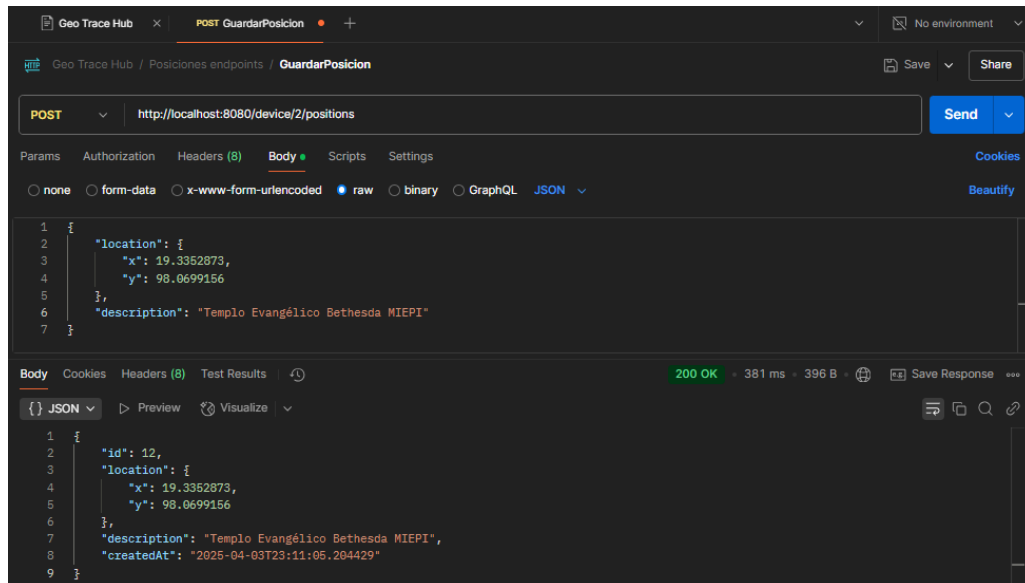
Criterios de aceptación:

1. Se debe poder registrar una ubicación con latitud, longitud, y el tiempo local lo almacenará directamente.
 - 1.1. Si el registro es exitoso, debe devolver *200 OK* con la ubicación guardada.
2. Si el dispositivo no existe, debe devolver *404 Not Found*.
3. Si falta algún dato obligatorio, debe devolver *400 Bad Request*.

En la Figura 331 se evidencia el resultado del criterio 1.1 sí se cumple al asignarle al endpoint el ID del dispositivo junto con el cuerpo JSON se agregan los datos de la ubicación y la descripción del lugar en el que responde con una respuesta *200 OK*. Muestra la posición junto con la fecha en la que se guardó la posición:

Figura 331

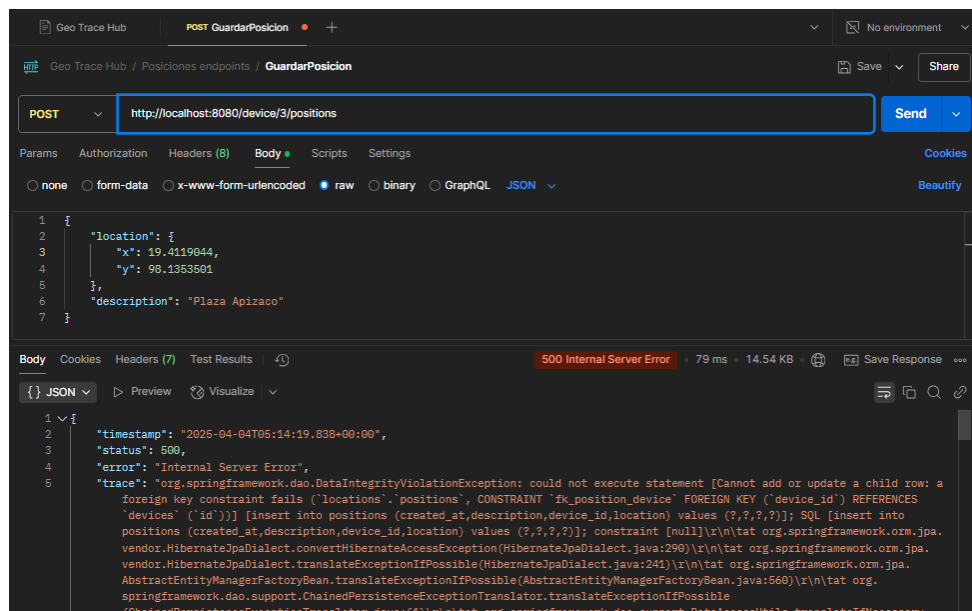
Resultado del criterio 1 para almacenar ubicaciones.



En la Figura 332 muestra que el criterio 2 no se cumple ya que si no encuentra un dispositivo, directamente manda un error *500 Internal Server Error*.

Figura 332

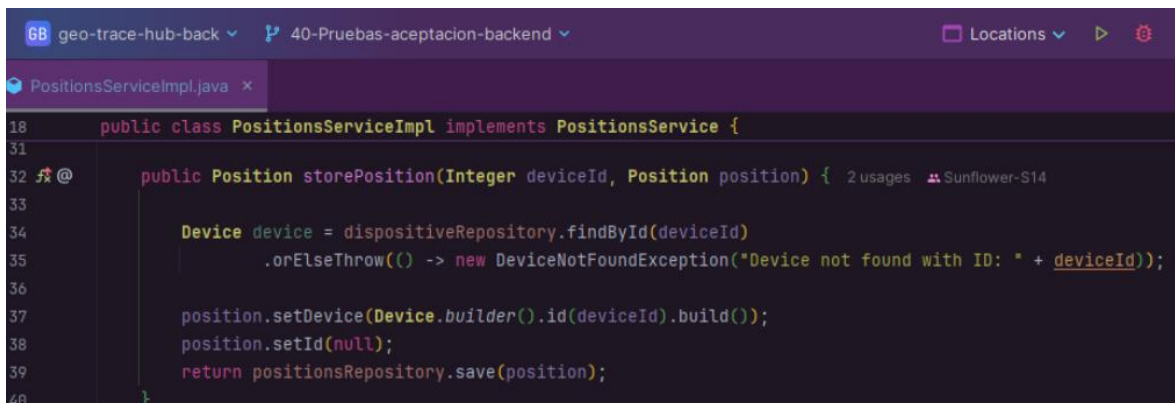
Resultado erróneo del criterio 2 para almacenar ubicaciones.



Para resolver esto en la Figura 333 se muestra a la clase “PositionsServiceImpl” en el método “storePosition” se modifica para que primero busque si existe el dispositivo mediante su id, sino existe se lanzará una excepción con la clase “DeviceNotFoundException”:

Figura 333

Se usa DeviceNotFoundException en el método storePosition.

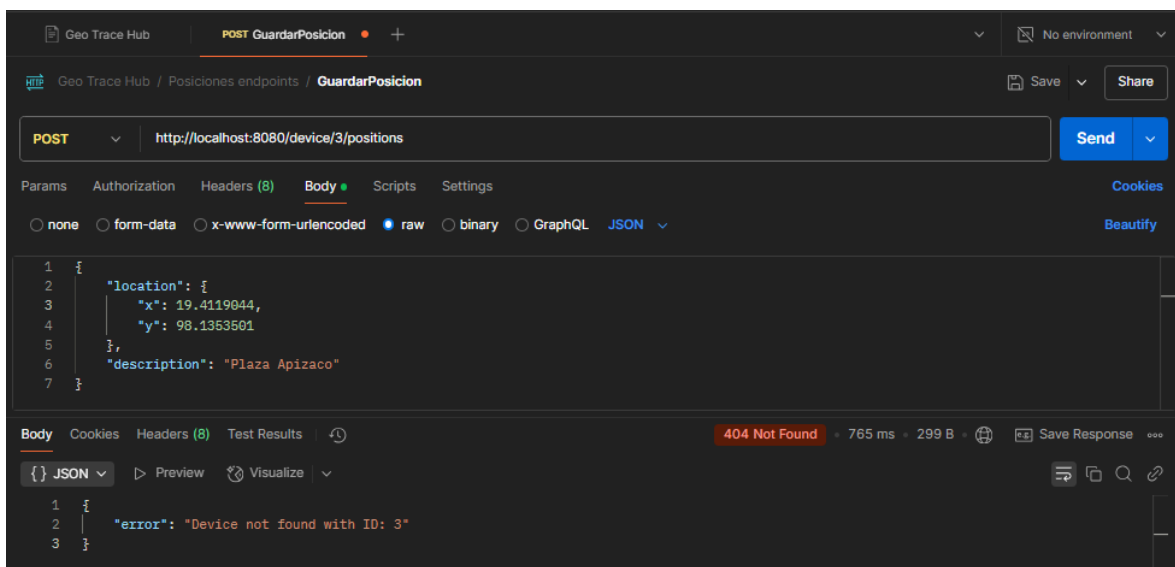


```
18 public class PositionsServiceImpl implements PositionsService {
31
32 @
33 public Position storePosition(Integer deviceId, Position position) {
34     Device device = dispositivoRepository.findById(deviceId)
35         .orElseThrow(() -> new DeviceNotFoundException("Device not found with ID: " + deviceId));
36
37     position.setDevice(Device.builder().id(deviceId).build());
38     position.setId(null);
39     return positionsRepository.save(position);
40 }
```

La Figura 334 presenta que luego de ejecutar y realizar la petición mostrará una respuesta 404 *Not Found* con el mensaje de que el dispositivo no se encontró, con esto se cumple el criterio 2:

Figura 334

Resultado del criterio 2 para almacenar ubicaciones.



Geo Trace Hub POST GuardarPosicion

Geo Trace Hub / Posiciones endpoints / GuardarPosicion

POST http://localhost:8080/device/3/positions

Params Authorization Headers (8) Body Scripts Settings

none form-data x-www-form-urlencoded raw binary GraphQL JSON

```
1 {
2   "location": {
3     "x": 19.4119944,
4     "y": 98.1353501
5   },
6   "description": "Plaza Apizaco"
7 }
```

Body Cookies Headers (8) Test Results

404 Not Found 765 ms 299 B

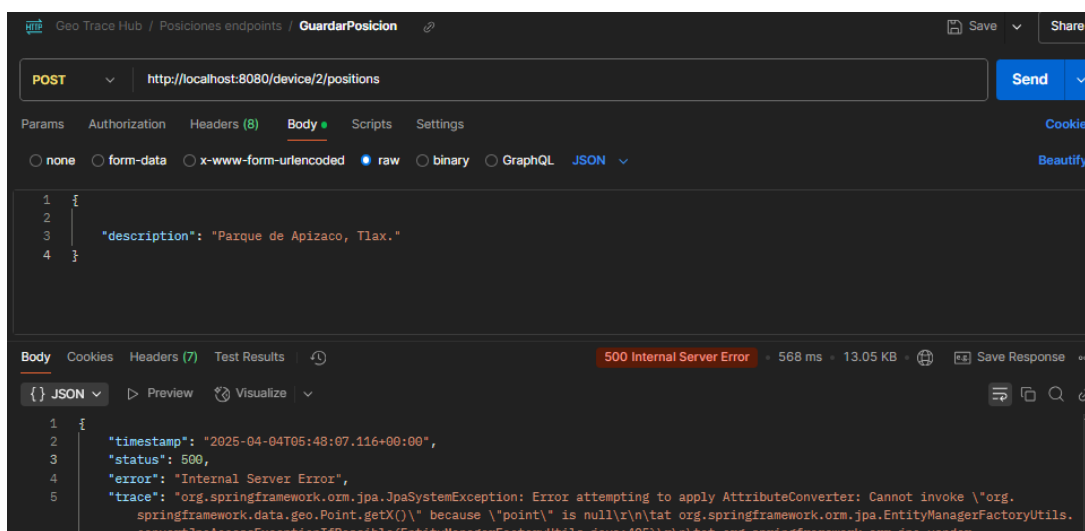
JSON Preview Visualize

```
1 {
2   "error": "Device not found with ID: 3"
3 }
```

Para el criterio 3 se tiene que indicar que los campos son obligatorios para guardar una posición, sin embargo no se agregaron las coordenadas y devuelve una respuesta *500 Internal Server Error*, como se confirma en la Figura 335:

Figura 335

Resultado erróneo del criterio 3 para almacenar ubicaciones.



En la Figura 336 se resuelve modificando la clase “`PositionsServiceImpl`”, en el que se agregan condicionales para validar campos obligatorios, si estos están asignados guardará la ubicación:

Figura 336

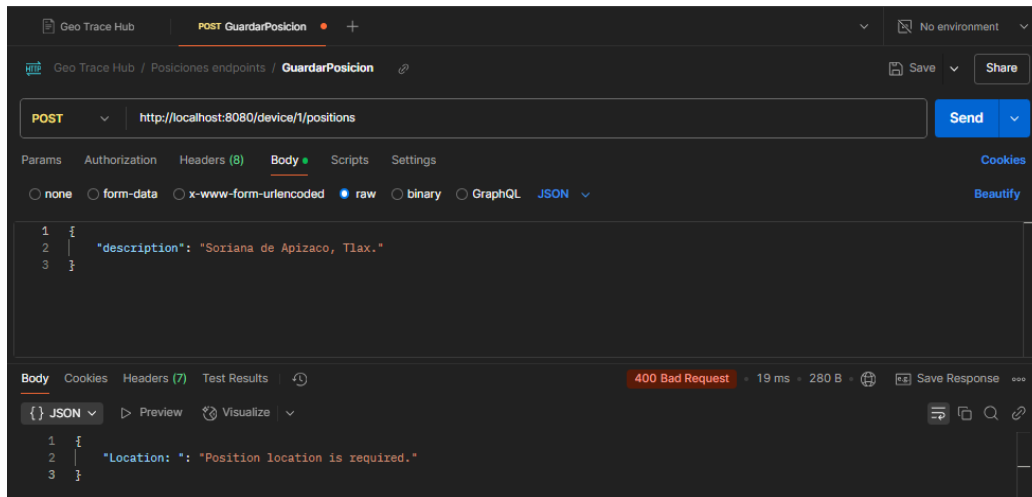
Validaciones en el método `storePosition` de la clase `PositionsServiceImpl`.

```
PositionsServiceImpl.java x
20 public class PositionsServiceImpl implements PositionsService {
34
35     @Override 2 usages Sunflower-S14
36     public Position storePosition(Integer deviceId, Position position) {
37         log.info("Validating position before saving: {}", position);
38
39         Device device = deviceRepository.findById(deviceId)
40             .orElseThrow(() -> new DeviceNotFoundException("Device not found with ID: " + deviceId));
41
42         if (position.getLocation() == null) {
43             throw new ValidateException("Location: ", "Position location is required.");
44         }
45
46         if (position.getDescription() == null || position.getDescription().trim().isEmpty()) {
47             throw new ValidateException("Description: ", "Position description is required.");
48         }
49
50         position.setDevice(device);
51         position.setId(null);
52
53         log.info("Saving position: {}", position);
54         return positionRepository.save(position);
55     }
56 }
57 }
```

La Figura 337 presenta la ejecución del endpoint que devuelve una respuesta *400 Bad Request* y el mensaje de que es obligatoria la ubicación:

Figura 337

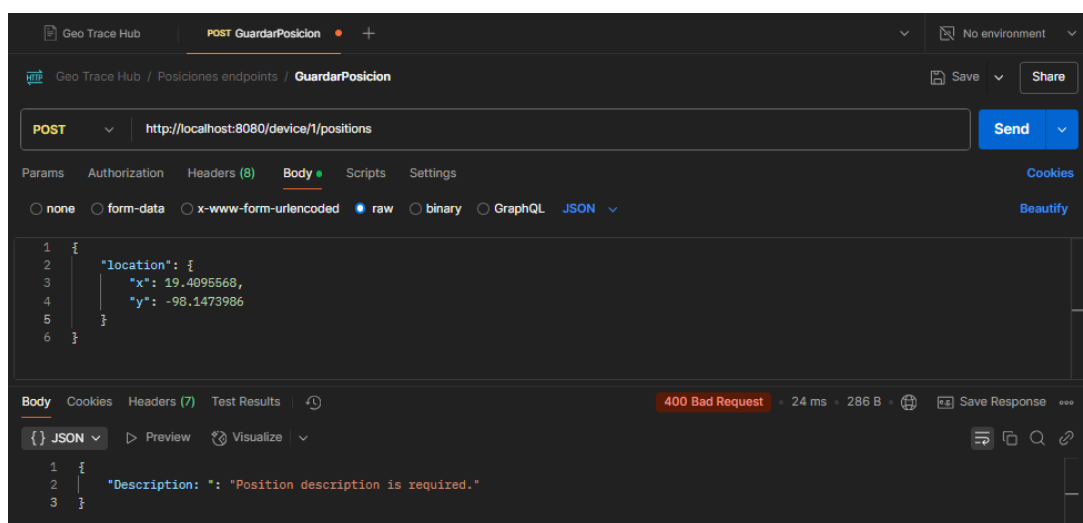
Ejecución del criterio 3 para el almacenado de ubicaciones.



De la misma forma con el campo “descripción” en la Figura 338 devuelve una respuesta *400 Bad Request* indicando que se necesita este dato, así se cumple el criterio de aceptación 3:

Figura 338

Resultado del criterio 3 para almacenar ubicaciones.



Por último, en la Tabla 18 muestra un endpoint que sirve para filtrar ubicaciones por rango de fecha en un dispositivo:

Tabla 18

Estructura de la petición "filtrado de ubicaciones".

Tipo de petición	Endpoint a probar
GET	http://localhost:8080/device/1/positions/filtered?startDate=2025-03-01T00:00:00&endDate=2025-03-21T14:34:24

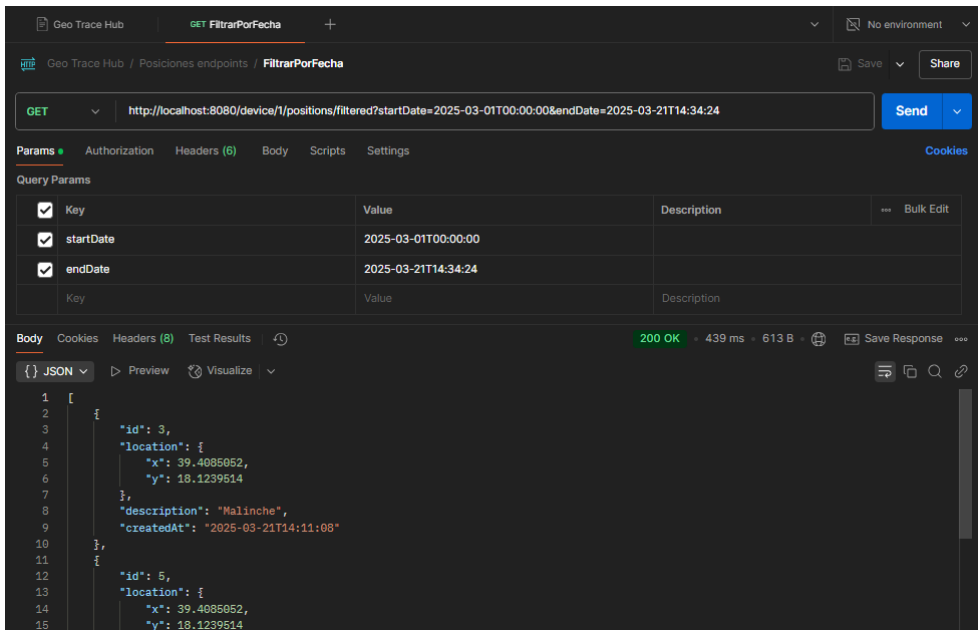
Criterios de aceptación:

- 1. Se debe poder filtrar las ubicaciones por rango de fechas.
- 2. Si no hay ubicaciones en ese rango, debe devolver una lista vacía con 200 OK.

El criterio 1 se cumple de manera satisfactoria debido a que el dispositivo establecido con los parámetros de fecha de inicio 2025-03-01T00:00:00 y de fin 2025-03-21T14:34:24 muestra la lista de posiciones guardadas en el dispositivo devolviendo una respuesta 200 OK:

Figura 339

Resultado del criterio 1 para el filtrado de ubicaciones.



Mostrando en la Figura 340 que la prueba imprime tres ubicaciones con el rango de tiempo especificado:

Figura 340

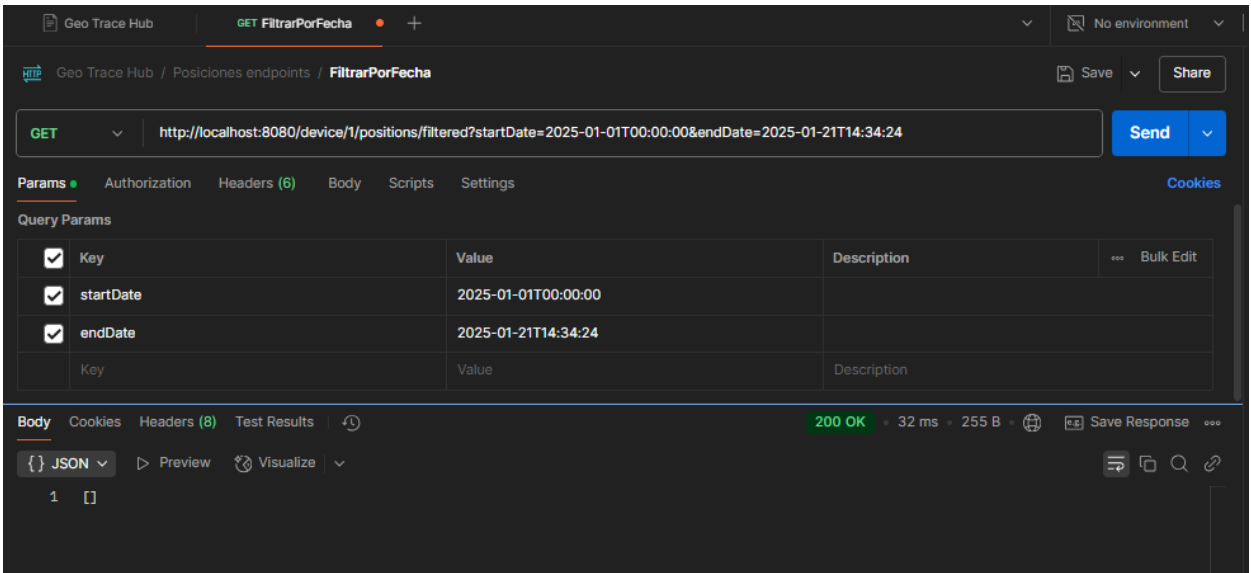
Listado de ubicaciones en un periodo establecido.



En la Figura 341 se visualiza el criterio 2, este se cumple debido a que en un dispositivo se filtra por fecha de inicio así como de fin, mostrando una lista vacía indicando que no hay posiciones registradas en ese rango de fechas. Se usó de fecha de inicio 2025-01-01T00:00:00 y fecha de fin 2025- 2025-01-21T14:34:24

Figura 341

Resultado del criterio 2 para el filtrado de ubicaciones.

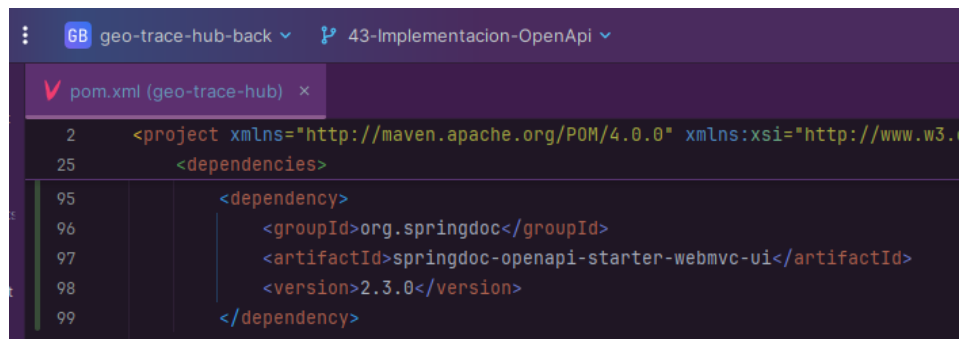


3.1.23. Implementar el estándar OpenAPI en los endpoints desarrollados

Se agrega en la Figura 342 la dependencia del estándar *OpenAPI*:

Figura 342

Dependencia OpenAPI en el pom.xml.

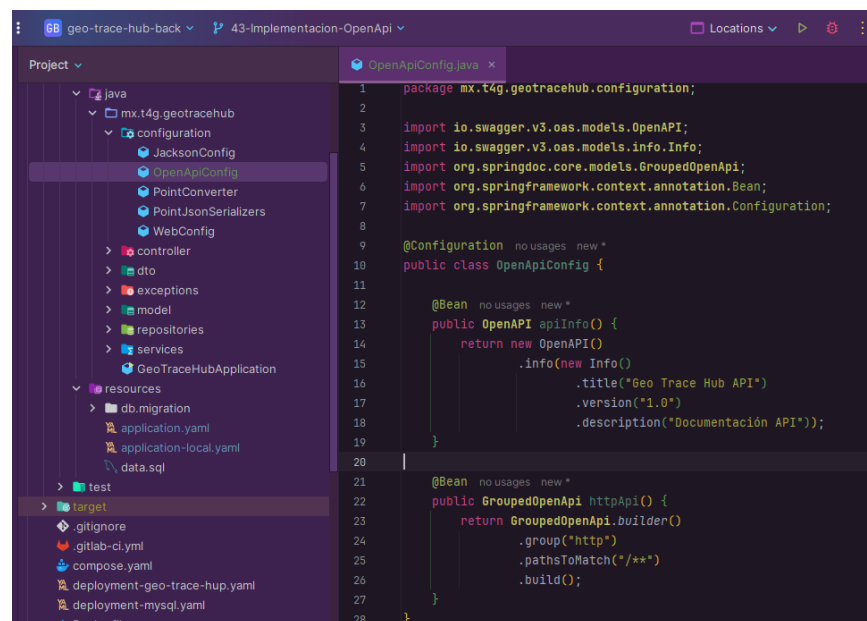


```
2 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
25 <dependencies>
95 <dependency>
96 <groupId>org.springdoc</groupId>
97 <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
98 <version>2.3.0</version>
99 </dependency>
```

En la Figura 343 muestra que en la carpeta “configuration” se crea una clase de nombre “OpenAPIConfig” en la que permite establecer la configuración mínima que se mostrará en la documentación:

Figura 343

Clase OpenAPIConfig.

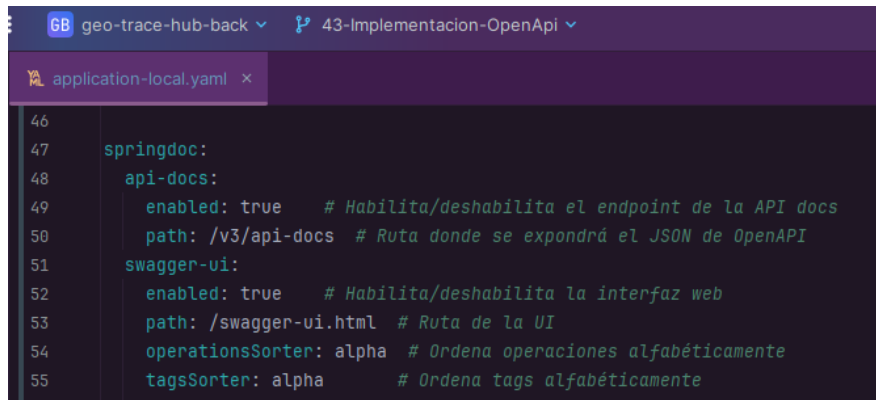


```
1 package mx.t4g.geotracehub.configuration;
2
3 import io.swagger.v3.oas.models.OpenAPI;
4 import io.swagger.v3.oas.models.info.Info;
5 import org.springdoc.core.models.GroupedOpenApi;
6 import org.springframework.context.annotation.Bean;
7 import org.springframework.context.annotation.Configuration;
8
9 @Configuration
10 public class OpenAPIConfig {
11
12     @Bean
13     public OpenAPI apiInfo() {
14         return new OpenAPI()
15             .info(new Info()
16                 .title("Geo Trace Hub API")
17                 .version("1.0")
18                 .description("Documentación API"));
19     }
20
21     @Bean
22     public GroupedOpenApi httpApi() {
23         return GroupedOpenApi.builder()
24             .group("http")
25             .pathsToMatch("/**")
26             .build();
27     }
28 }
```


En el archivo `.yaml` de la Figura 344 se agregan configuraciones para controlar el endpoint que genera el documento OpenAPI en formato JSON y configura la interfaz visual de *Swagger*:

Figura 344

Configuración de OpenAPI en el archivo `.yaml`.

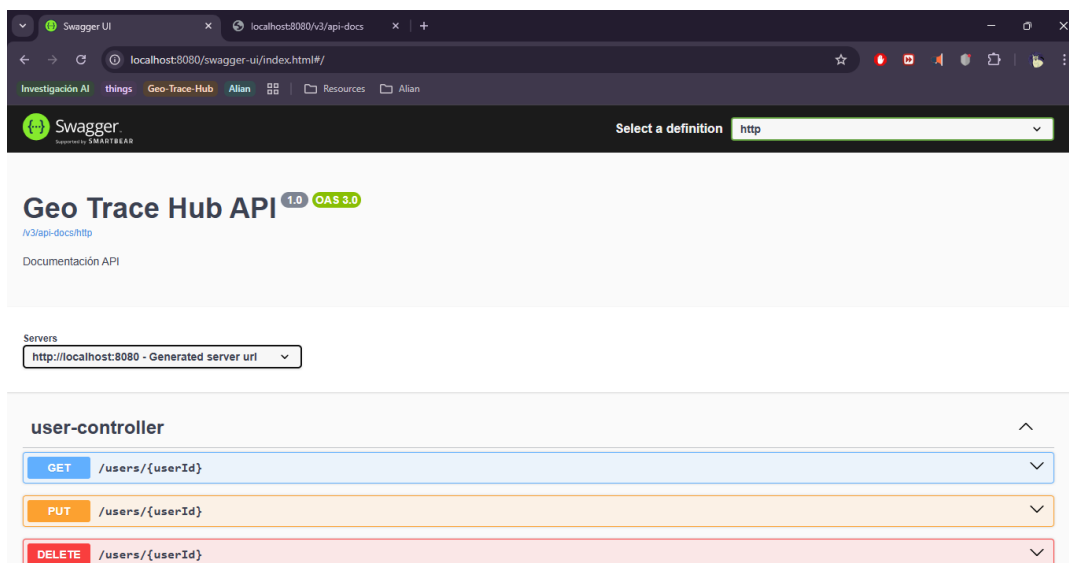


```
46
47 springdoc:
48   api-docs:
49     enabled: true # Habilita/deshabilita el endpoint de la API docs
50     path: /v3/api-docs # Ruta donde se expondrá el JSON de OpenAPI
51   swagger-ui:
52     enabled: true # Habilita/deshabilita la interfaz web
53     path: /swagger-ui.html # Ruta de la UI
54     operationsSorter: alpha # Ordena operaciones alfabéticamente
55     tagsSorter: alpha # Ordena tags alfabéticamente
```

Una vez modificado, se ejecuta el *backend* para iniciar la aplicación, en el que en automático, se tendrá acceso Swagger UI, la cual es documentación interactiva: <http://localhost:8080/swagger-ui.html> queda reflejado en la Figura 345:

Figura 345

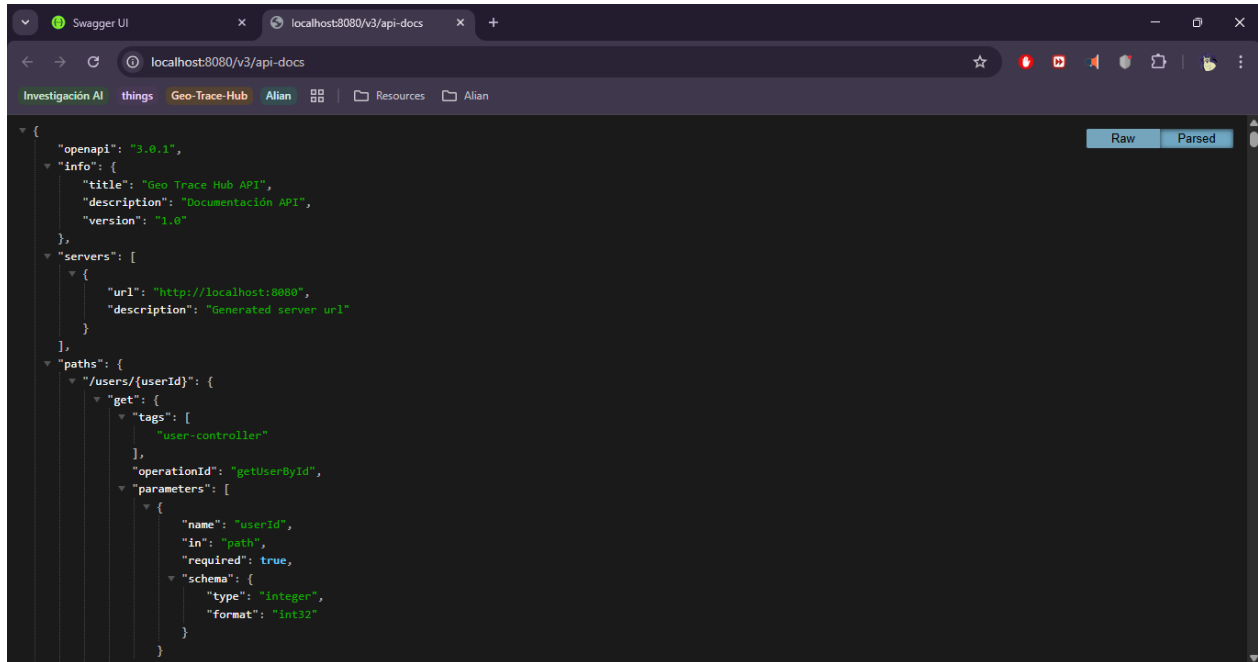
Interfaz de OpenAPI.



En la Figura 346 muestra la especificación OpenAPI en JSON mediante <http://localhost:8080/v3/api-docs>:

Figura 346

OpenAPI en formato JSON.



OpenAPI en automático genera los *endpoints* del *API REST* desarrollada con Spring Boot, para mostrar una documentación interactiva y completa de la API.

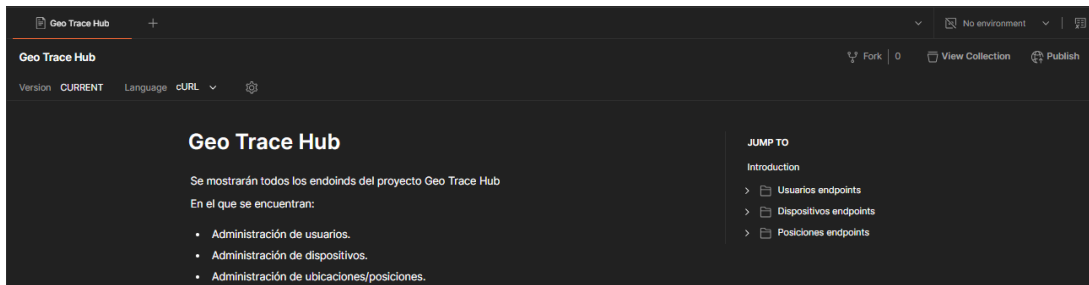
IV. Resultados

4.1. Colección POSTMAN de la suite de pruebas

Se creó una colección en POSTMAN que se utilizó para la ejecución de pruebas de aceptación, que a su vez, permite visualizar a los desarrolladores la estructura de los *endpoints*, para ejecutar su comportamiento en respuestas JSON. Se encuentran tres grupos de *endpoints* los cuales son la administración de usuarios, dispositivos y ubicaciones, como se aprecia en la Figura 347:

Figura 347

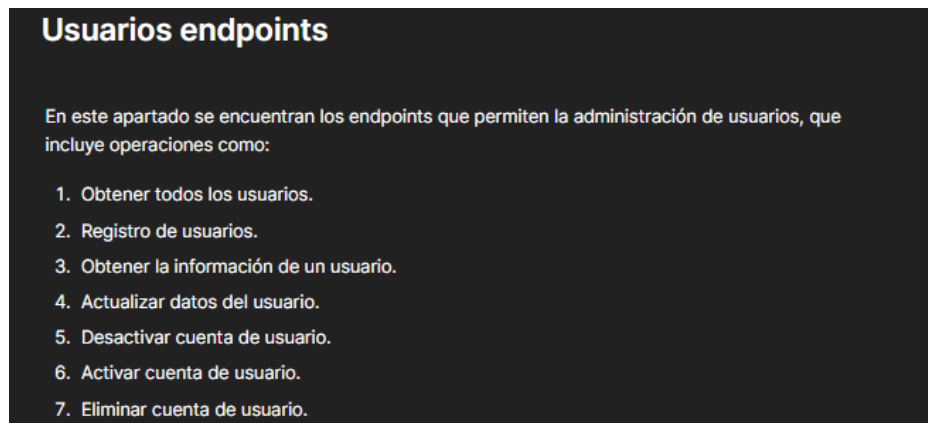
Introducción a la colección de POSTMAN.



Primero, en la Figura 348 se muestran los *endpoints* de la administración de usuarios, el que se describe en un listado los siete *endpoints* que lo conforman:

Figura 348

Colección de endpoints de usuarios.



Uno de los *endpoints* es de tipo POST está en la Figura 349 en el que permite registrar un nuevo usuario al sistema, en donde se visualiza la URL y el cuerpo en JSON el ejemplo de los campos que debe de ingresar con su respectiva información:

Figura 349

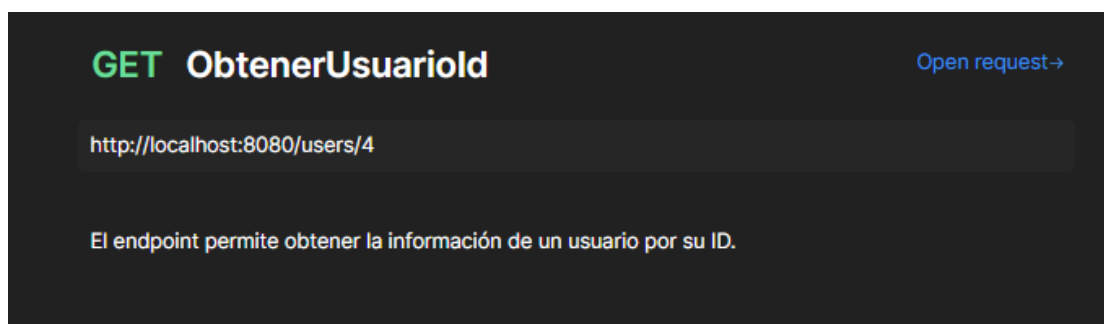
Endpoint registro de usuarios.



El siguiente endpoint es de tipo GET, reflejado en la Figura 350 permite buscar un usuario por su ID, el cual mostrará la información completa de su registro:

Figura 350

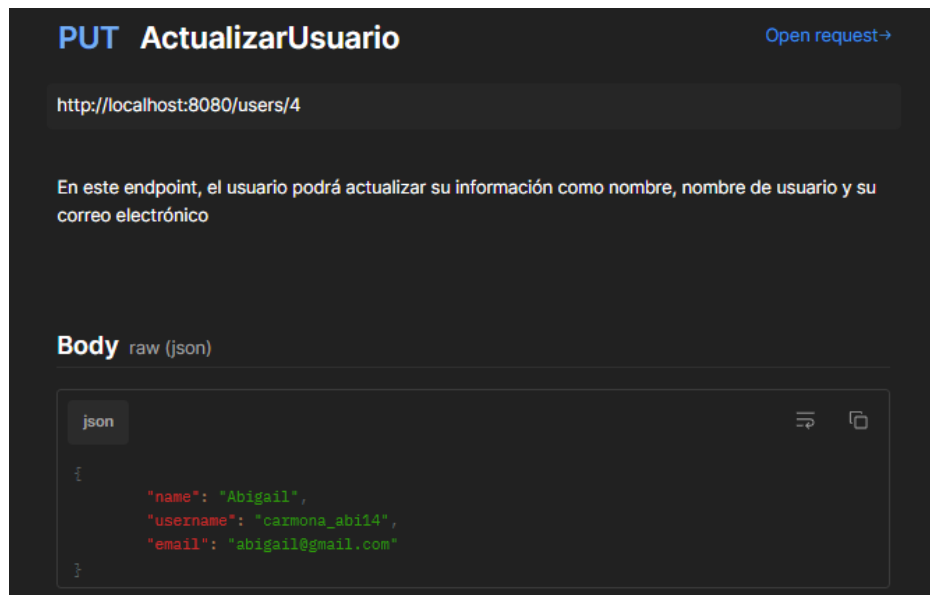
Endpoint obtener usuarios por su ID.



Luego se muestra un endpoint de tipo PUT, evidenciado en la Figura 351 que permite actualizar la información de un usuario en el que en la URL se debe de especificar el ID del usuario que es mediante JSON se editan los datos los cuales solo es el nombre, nombre de usuario y correo:

Figura 351

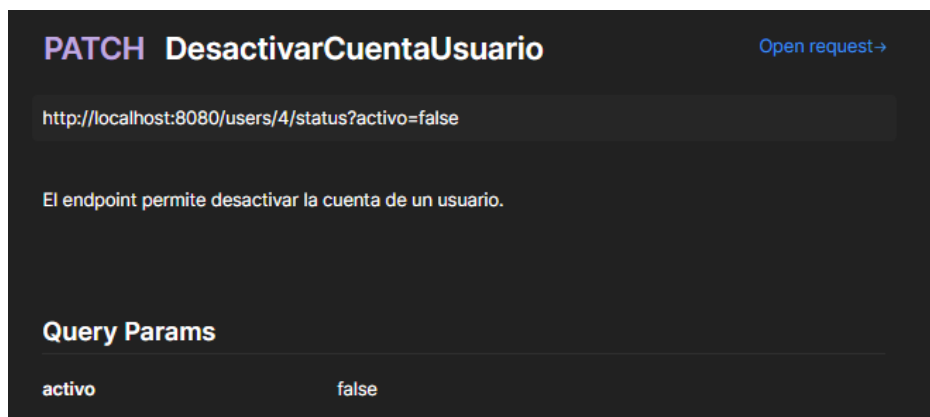
Endpoint actualizar a un usuario.



Otro endpoint importante es de tipo PATCH, se visualiza en la Figura 352 que desactiva las cuentas de los usuarios mediante la URL que usa valores booleanos para desactivar la cuenta:

Figura 352

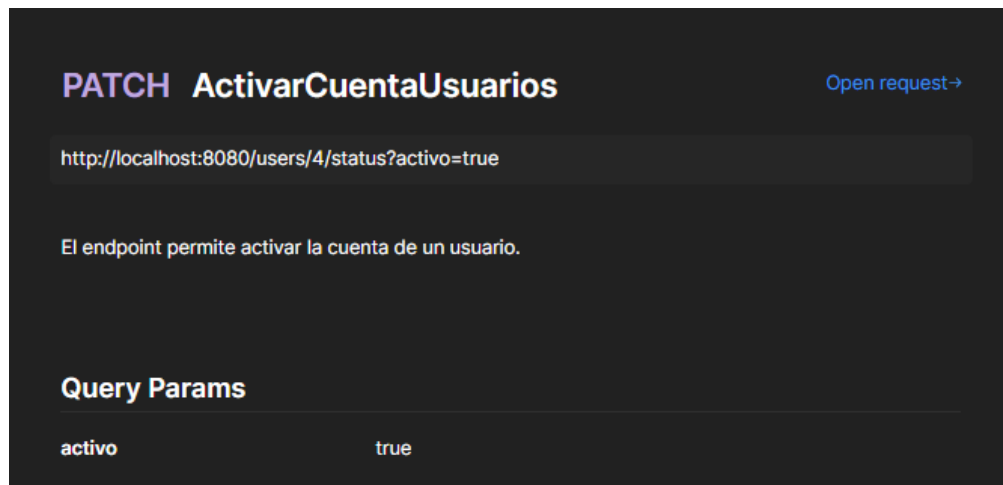
Endpoint desactivar cuenta de usuario.



También se tiene en el mismo endpoint de tipo PATCH, reflejado en la Figura 353 para actualizar el estado de la cuenta de usuario, pero en este se pasa como valor en verdadero para activar la cuenta:

Figura 353

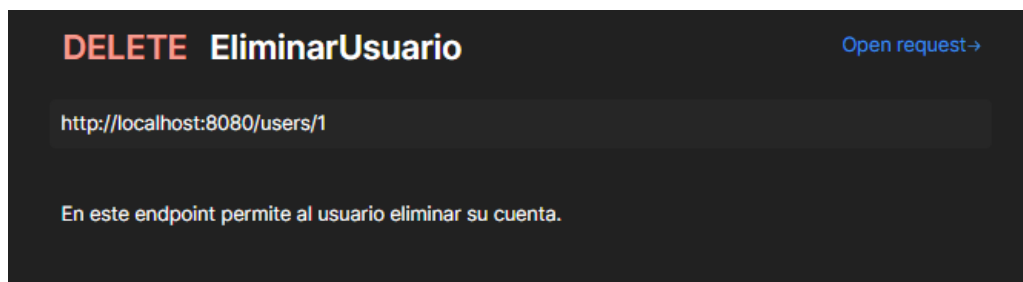
Endpoint activar cuenta de usuario.



Por último, se encuentra la petición de tipo DELETE documentada en la Figura 354 que permite eliminar un usuario a través de su ID:

Figura 354

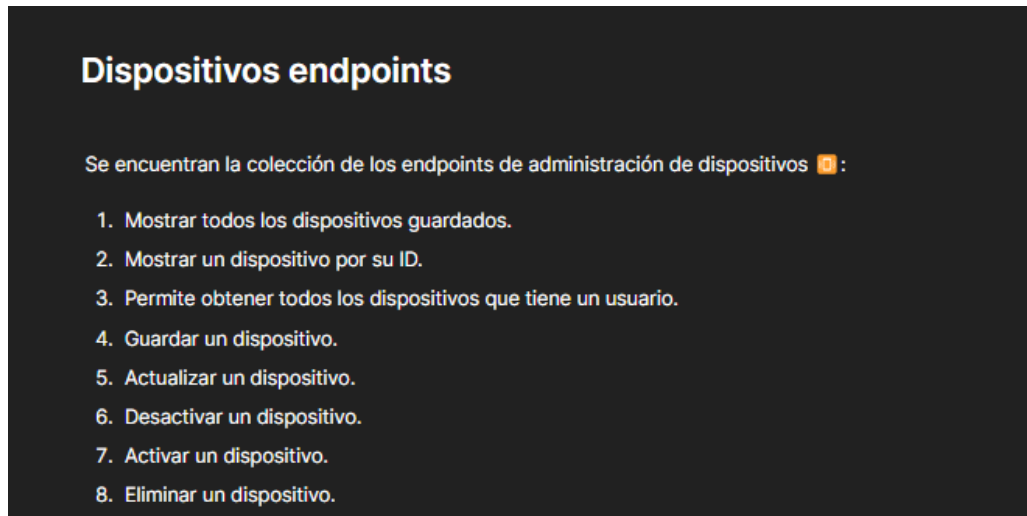
Endpoint eliminar usuario.



El siguiente grupo mostrado en la Figura 355 es la administración de dispositivos en los cuales se muestran ocho *endpoints*:

Figura 355

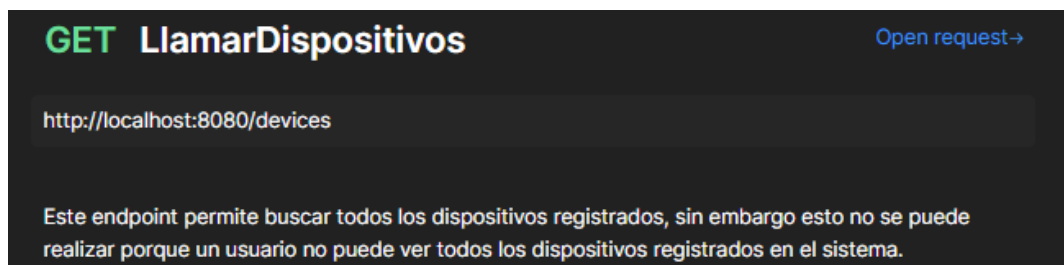
Colección de endpoints de dispositivos.



A continuación, se muestra el primer endpoint de este grupo es de tipo GET, reflejado en la Figura 356 que busca todos los dispositivos que hay en el sistema, sin embargo por los requerimientos no debe de mostrarse, solo un mensaje de que tiene que indicar al usuario primero:

Figura 356

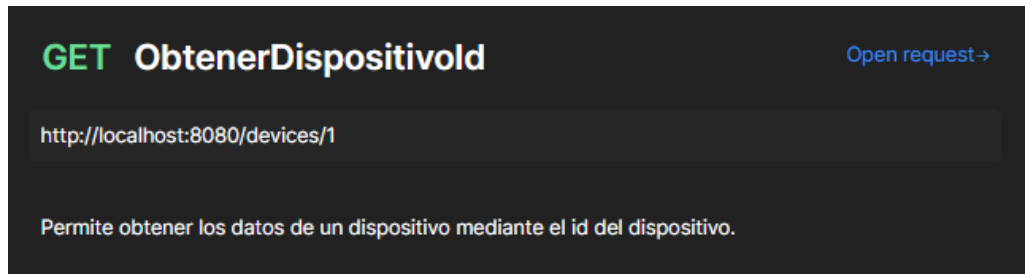
Endpoint llamar dispositivos.



En cuanto al siguiente endpoint de tipo GET, permite obtener la información de un dispositivo indicando su ID en la URL, mostrado en la Figura 357:

Figura 357

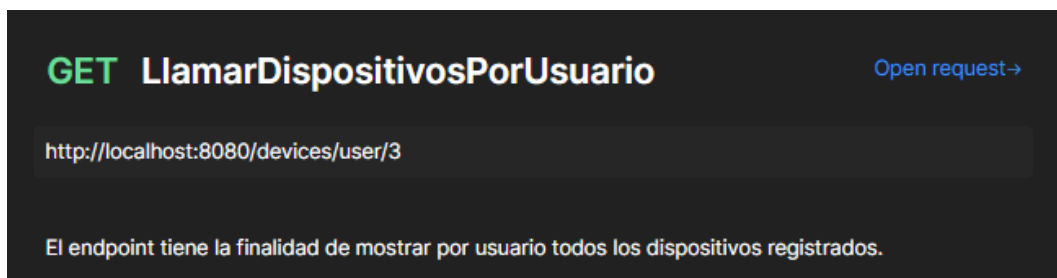
Endpoint obtener dispositivo por ID.



El próximo endpoint de tipo GET, documentado en la Figura 358 permite indicarle en la URL el ID del usuario que al ejecutarlo mostrará el listado de los dispositivos registrados por el usuario:

Figura 358

Endpoint llamar dispositivos por usuario.



También se creó un endpoint de tipo POST, que permite registrar la información de un dispositivo mediante JSON se puede visualizar en la Figura 359:

Figura 359

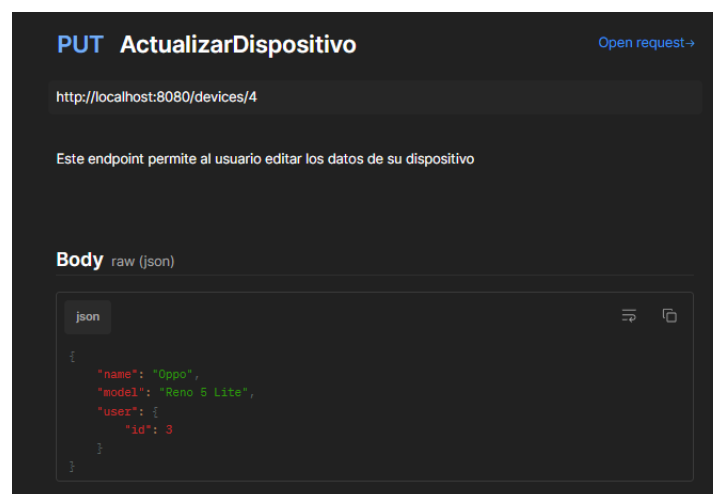
Endpoint guardar dispositivo.



Uno de los *endpoints* siguientes es de tipo PUT, su estructura está en la Figura 360 que permite actualizar los datos de un dispositivo mediante la búsqueda del dispositivo en la URL y en JSON se indicará los datos que se actualizarán:

Figura 360

Endpoint actualizar dispositivo.



Adicionalmente en la Figura 361 se contempla el endpoint de tipo PATCH, el cual permite desactivar el dispositivo:

Figura 361

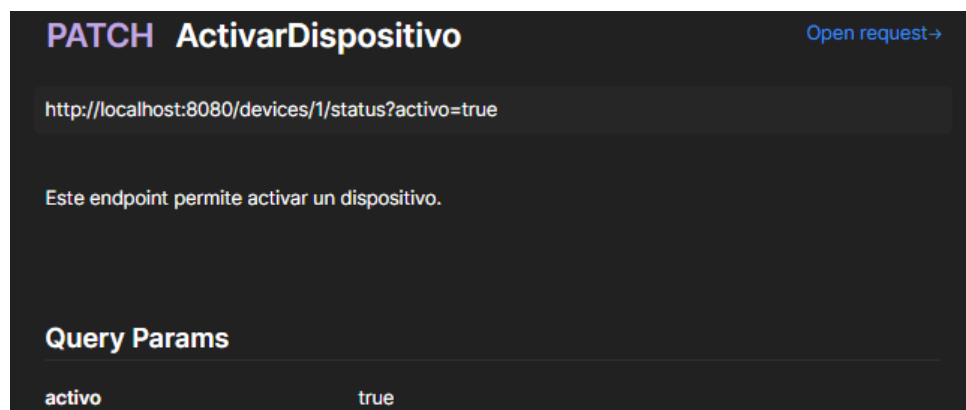
Endpoint desactivar dispositivo.



Así como el endpoint anterior, en la Figura 362 se muestra que cambia solo el parámetro del estado de activo, en este caso es en "true" en el cual le indicará al dispositivo que cambie a un estado activo:

Figura 362

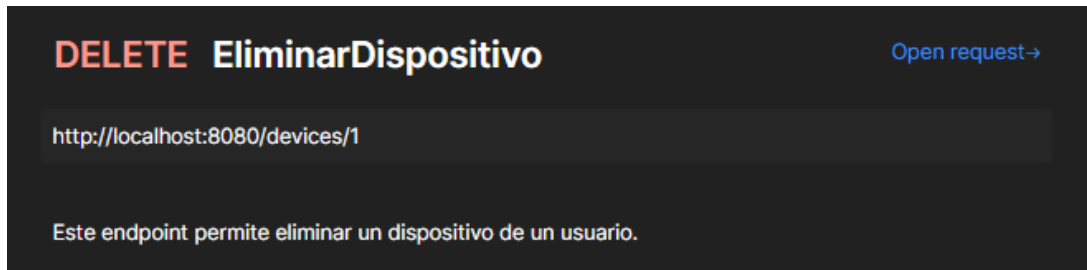
Endpoint activar dispositivo.



Otro endpoint destacado es de tipo DELETE, que elimina un dispositivo mediante su ID, descrito en la Figura 363:

Figura 363

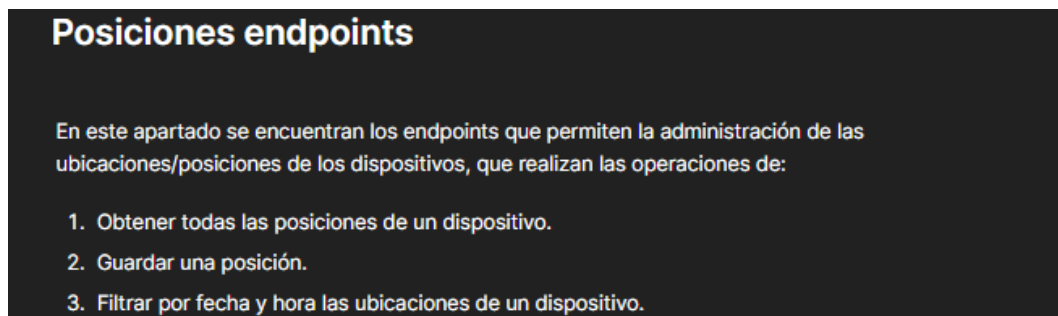
Endpoint eliminar dispositivo.



Para finalizar, en la Figura 364 se muestran los endpoints de ubicaciones los cuales son solamente tres:

Figura 364

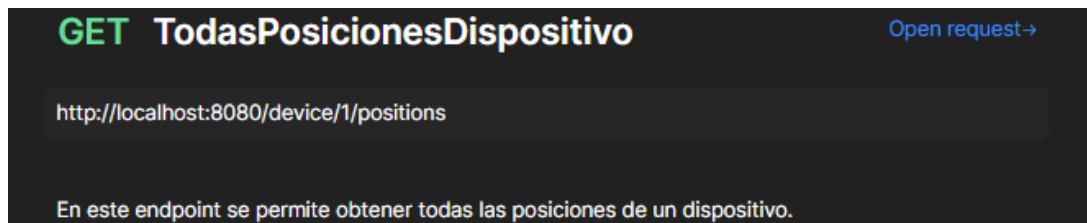
Colección de endpoints de posiciones.



El primer endpoint de tipo GET permite obtener todas las posiciones almacenadas de un dispositivo, en el que en la URL se especifica el ID del dispositivo, puede revisarse en la Figura 365:

Figura 365

Endpoint obtener todas las posiciones de un dispositivo.



El segundo endpoint de tipo POST permite guardar una ubicación en donde en la URL se especifica el ID del dispositivo y en formato JSON se mandan los datos de la ubicación en coordenadas “x” y “y”, juntamente con la descripción del lugar en donde se capturó la posición; su definición está contenida en la Figura 366:

Figura 366

Endpoint guardar posición.



En el tercer endpoint, de tipo GET permite filtrar las ubicaciones de un dispositivo por fecha, la cuales se tiene que indicar en la URL el ID del dispositivo, fecha de inicio así como la fecha de

fin para establecer el rango de las ubicaciones en una fecha y hora determinada, estas fechas debe de ser en formato YYYY-MM-DDTHH:MM:SS en el que se almacenaría como un tipo de dato DateTime. Esta lógica puede revisarse en la Figura 367:

Figura 367

Endpoint filtrar posición por fecha.



GET FiltrarPorFecha [Open request →](#)

`http://localhost:8080/device/1/positions/filtered?startDate=2025-03-01T00:00:00&endDate=2025-03-21T14:34:24`

Este endpoint permite especificar el dispositivo para filtrar por fecha en formato año-mes-día y tiempo en formato hora:minutos:segundos que retornará en una lista todas las ubicaciones guardadas, sino hay retorna una lista vacía.

Query Params

startDate	2025-03-01T00:00:00
endDate	2025-03-21T14:34:24

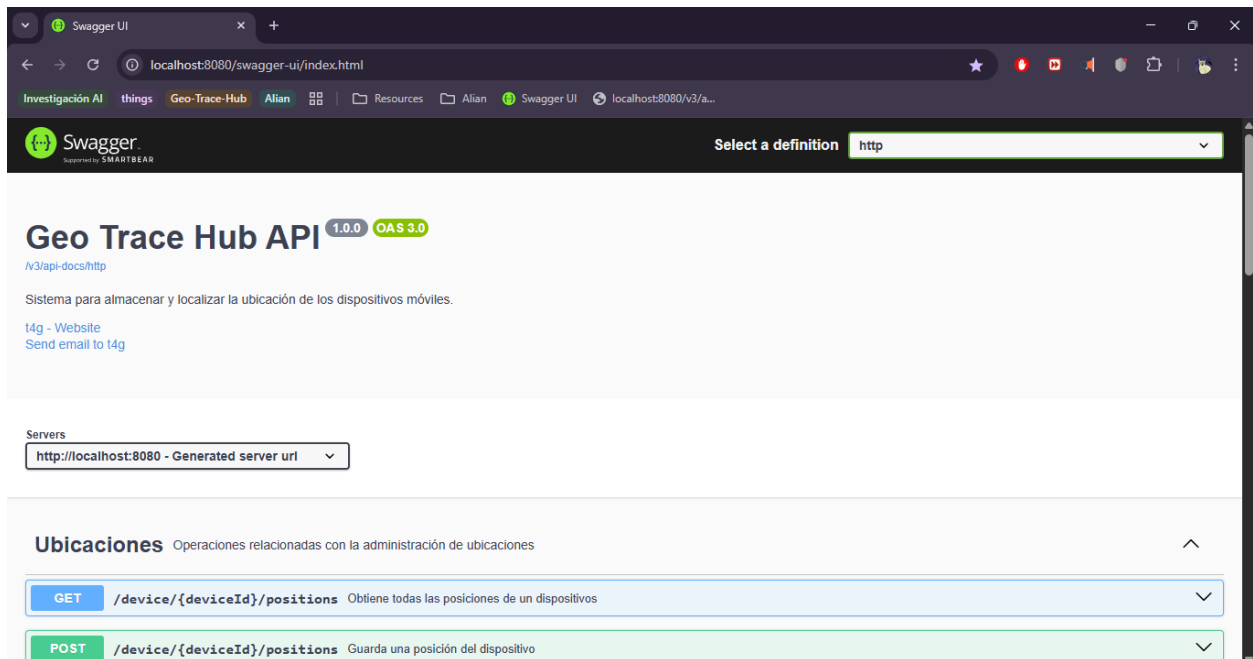
4.2. Manual creado mediante OpenAPI

Los programadores pueden revisar la documentación interactiva del *backend REST* desarrollado por Spring Boot mediante el siguiente enlace, que estará disponible en la ejecución del *backend*: <http://localhost:8080/swagger-ui/index.html>

En el que los programadores frontend y móvil verán la siguiente pantalla en la que podrán probar cada endpoint creado en el *backend REST*, cuya vista inicial se presenta en la Figura 368:

Figura 368

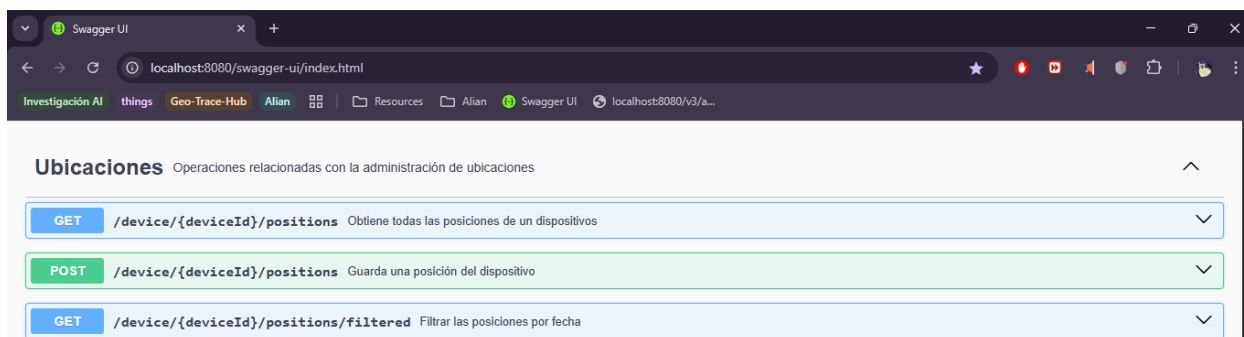
Documentación de OpenAPI.



En la Figura 369 se visualizan los *endpoints* de “Ubicaciones”, los cuales son tres *endpoints* desarrollados:

Figura 369

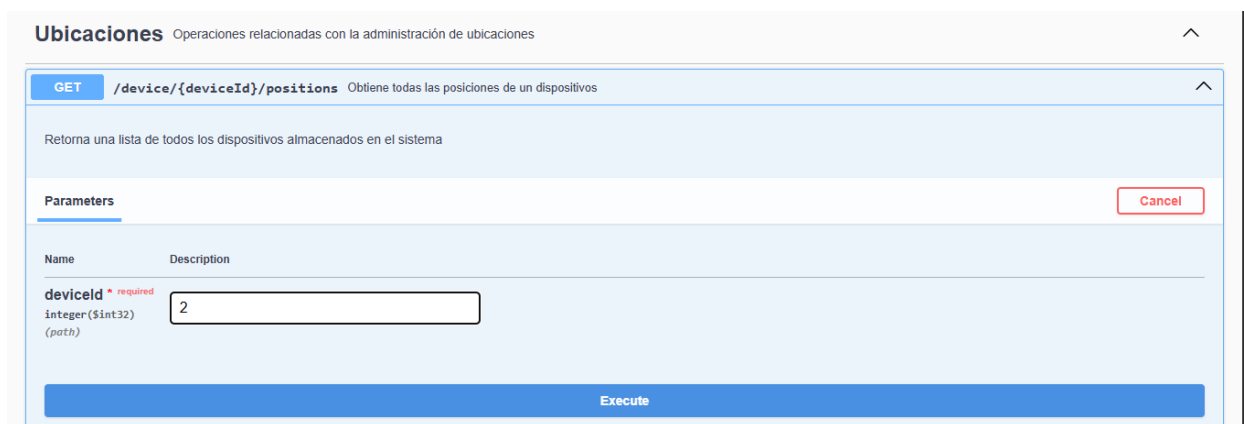
Endpoints de ubicaciones en OpenAPI.



El primer endpoint permite obtener todas las posiciones guardadas de un dispositivo en el que puede ingresar directamente desde esta interfaz el ID del dispositivo que consultarán sus ubicaciones, representado gráficamente en la Figura 370:

Figura 370

Primer endpoint de ubicaciones.

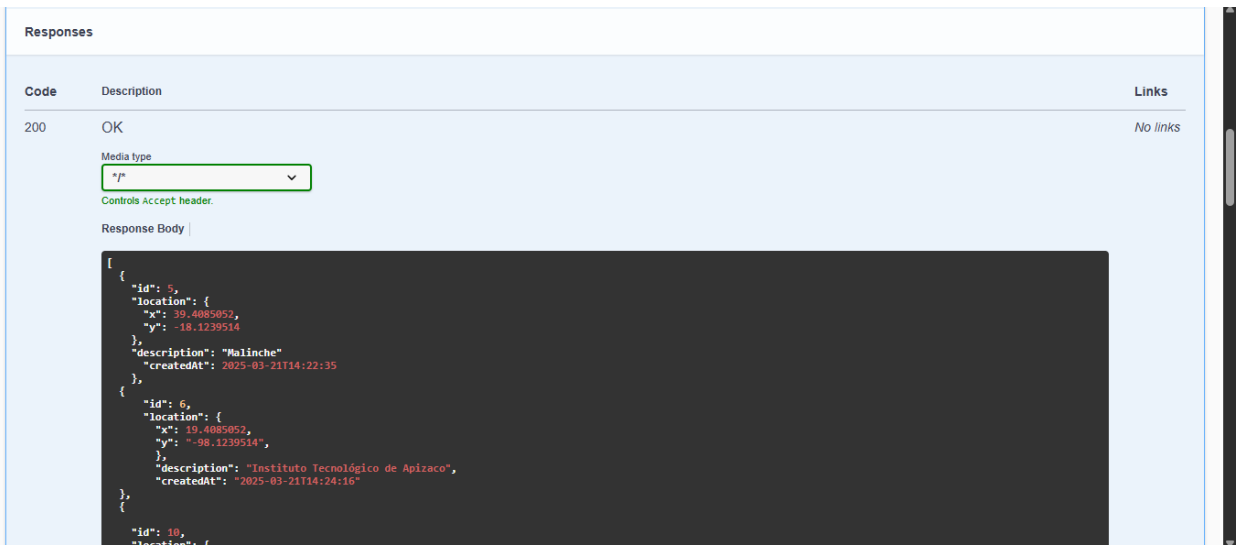


Al ejecutar el endpoint, mostrará en la sección de respuestas el código de la petición HTTP realizada la cual es 200, indicando que se ejecutó de manera correcta, en el cuerpo de la respuesta aparecerá un listado de posiciones guardas en el dispositivo que se especificó, mostrando el ID de la posición guardada (el cual es generado en automático). La posición

guardada por coordenadas “x” y “y”, la descripción de la posición en el que se almacena el nombre del lugar visitado y la fecha que en automático se captura al almacenar la posición lo que nos servirá posteriormente para filtrar por fechas las posiciones de un dispositivo, como se muestra en la Figura 371:

Figura 371

Resultado del primer endpoint de ubicaciones.



En cuanto al segundo endpoint permite guardar una ubicación, en la que en esta interfaz se especifica directamente el ID del dispositivo al que se va a guardar la posición (para asociarlo) y se debe de especificar en formato JSON la información a guardar, lo que es la ubicación en coordenadas, la descripción del lugar, lo que es el ID de la posición (generada en automático) así como la fecha en la que se guardó eso se hace en automático, puede analizarse en la Figura 372:

Figura 372

Segundo endpoint de ubicaciones.

POST /device/{deviceId}/positions Guarda una posición del dispositivo

Permite almacenar la posición del dispositivo asociandolo al usuario y su dispositivo

Parameters

Name	Description
deviceId * required integer(\$int32) (path)	15

Request body required application/json

```
{  "location": {    "x": 19.4076236,    "y": -98.1691303  },  "description": "La maquina, Apizaco."}
```

En la Figura 373 se evidencia que después de ejecutar el endpoint, se obtienen las respuestas, el formato *curl* de la petición realizada, la URL que se solicitó, la respuesta del servidor se ve el código de respuesta el cual es el 200 y en el cuerpo de respuesta se visualiza los datos en estructura JSON de lo que se guardó:

Figura 373

Resultado del segundo endpoint de ubicaciones.

Responses

Curl

```
curl -X 'POST' \
  'http://localhost:8080/device/15/positions' \
  -H 'accept: */*' \
  -H 'Content-Type: application/json' \
  -d '{
    "location": {
      "x": 19.4076236,
      "y": -98.1691303
    },
    "description": "La maquina, Apizaco."
  }'
```

Request URL

```
http://localhost:8080/device/15/positions
```

Server response

Code	Details
200	Response body <pre>{ "location": { "x": 19.4076236, "y": -98.1691303 }, "description": "La maquina, Apizaco.", "createdAt": "2025-05-24T13:56:21"}</pre>

En el tercer endpoint de la Figura 374 permite realizar el filtrado de ubicaciones de un dispositivo, en el que se indica el ID del dispositivo, la fecha de inicio y la fecha de fin en formato YYYY-MM-DDTHH:MM:SS:

Figura 374

Tercer endpoint de ubicaciones.

GET /device/{deviceId}/positions/filtered Filtrar las posiciones por fecha

Permite mediante el ID del dispositivo filtrar por fecha de inicio y fecha de fin las ubicaciones almacenadas

Parameters

Name	Description
deviceId * required integer(\$int32) (path)	3
startDate * required string(\$date-time) (query)	2025-05-21T14:20:00
endDate * required string(\$date-time) (query)	2025-05-21T18:00:00

Execute

Luego de ejecutar el endpoint, en las repuestas se obtiene el código 200 y en el cuerpo de respuesta, se muestran un listado de posiciones en formato JSON en el que se mostrarán las posiciones entre el rango especificado, se describe en la Figura 375:

Figura 375

Resultado del tercer endpoint de ubicaciones.

Responses

Code	Description	Links
200	OK	No links

Media type: */*
Controls Accept header.

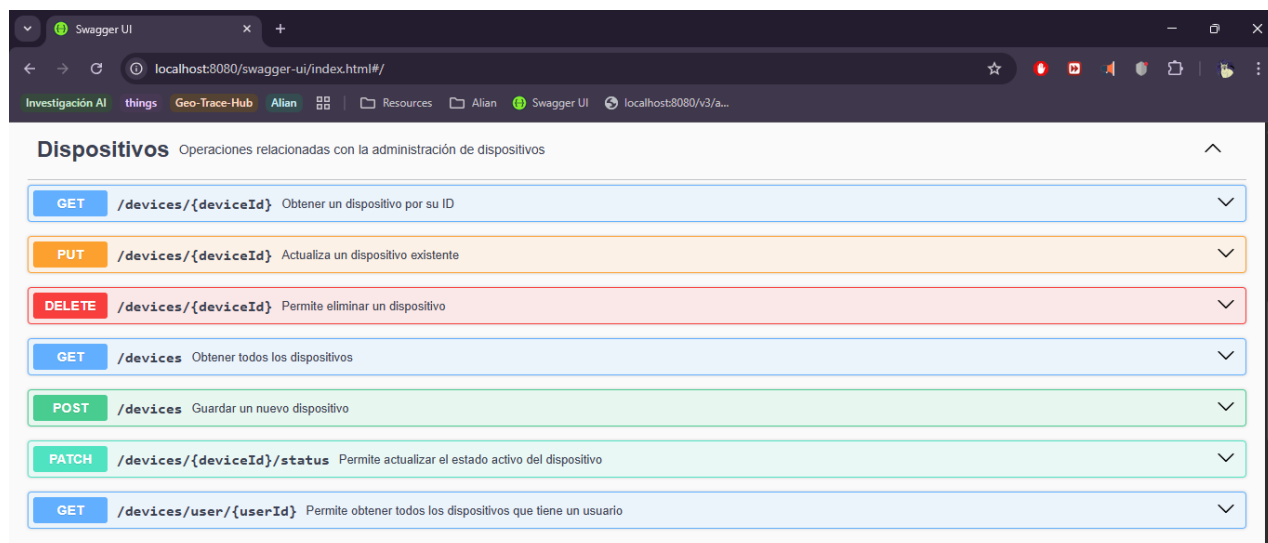
Response Body

```
[
  {
    "id": 20,
    "location": {
      "x": 19.339039,
      "y": -98.2028265
    },
    "description": "Gran Patio",
    "createdAt": 2025-05-24T17:30:00
  },
  {
    "id": 22,
    "location": {
      "x": 19.3165811,
      "y": -98.2569145
    },
    "description": "Tlaxcala de Xicoténcatl",
    "createdAt": 2025-05-24T14:24:16
  }
]
```

En la Figura 376 documenta la sección de *endpoints* de “Dispositivos”, se muestran los siete *endpoints* con su descripción:

Figura 376

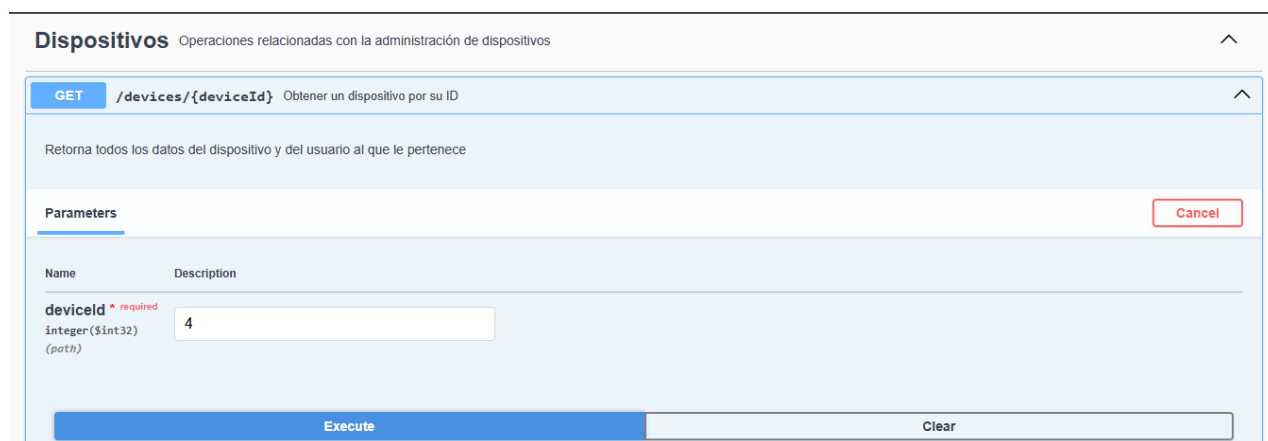
Endpoints de dispositivos en OpenAPI.



En el primer endpoint de esta sección en la Figura 377 se plasma un método de tipo GET que permite buscar un dispositivo mediante su ID:

Figura 377

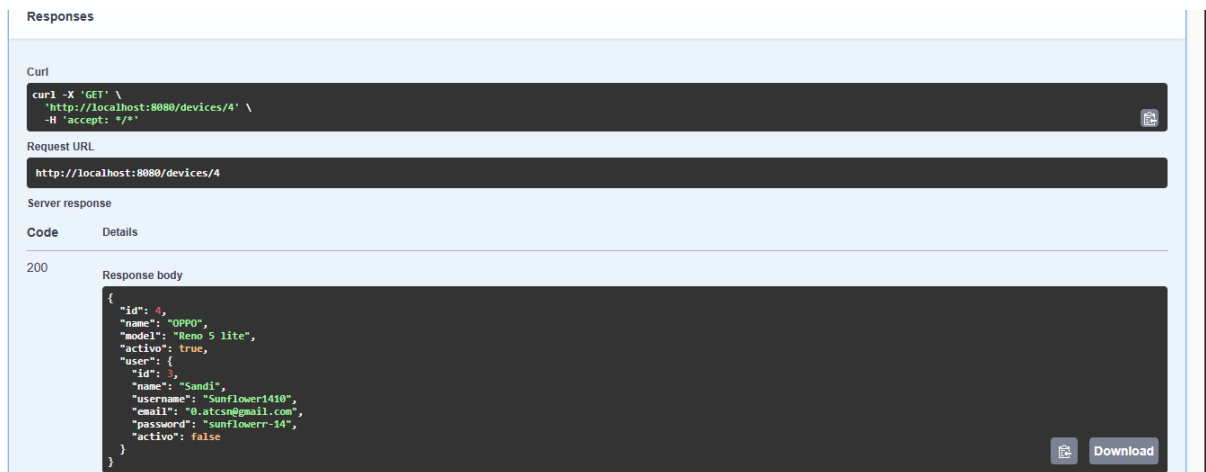
Primer endpoint de dispositivos.



Al ejecutar el endpoint muestra la sección de respuestas en formato *curl*, la URL de la petición, la respuesta del servidor mostrando código 200, el cuerpo de la respuesta en JSON con la información del dispositivo; puede observarse en la Figura 378:

Figura 378

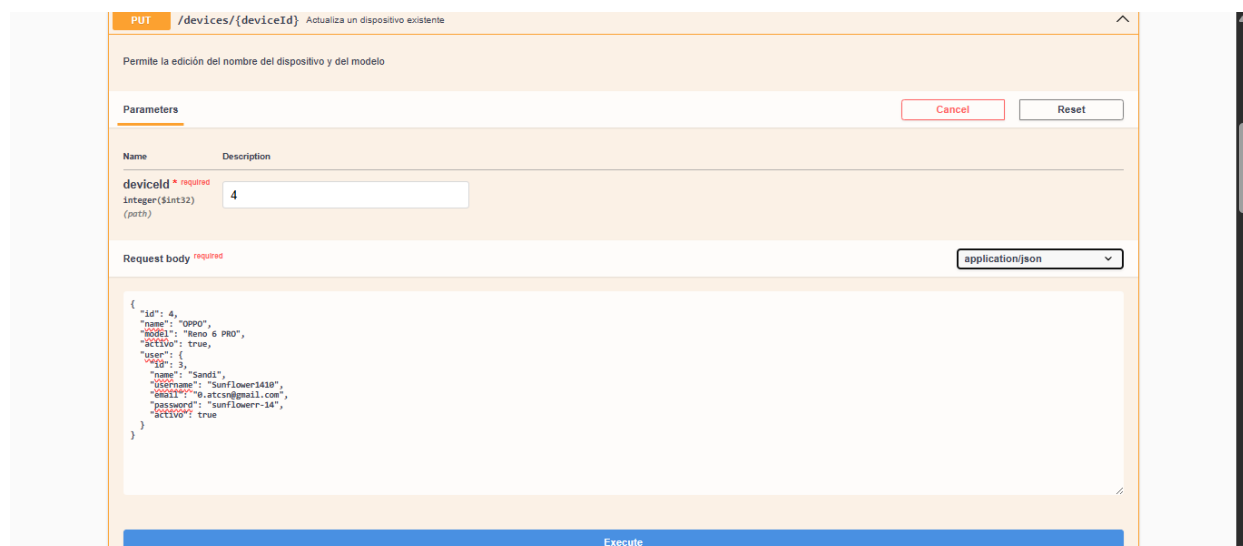
Resultado del primer endpoint de dispositivos.



En el segundo endpoint permite actualizar un dispositivo, primero busca el ID, mediante JSON se actualizan los datos, en este ejemplo ya se agregaron los datos a actualizar, se presenta en la Figura 379:

Figura 379

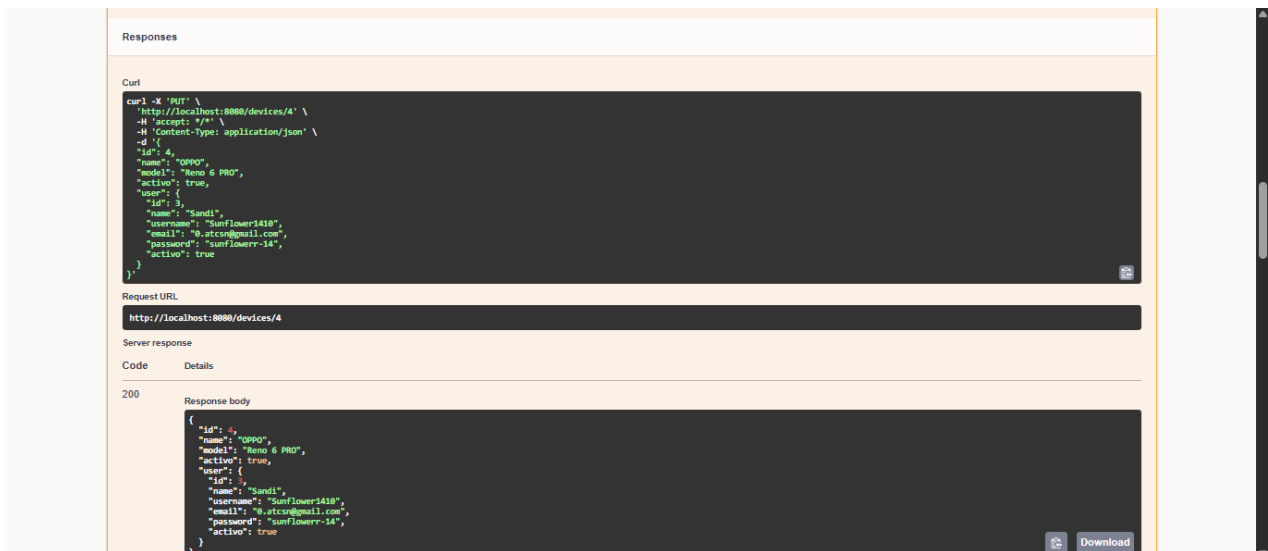
Segundo endpoint de dispositivos.



Después de ejecutar el endpoint se muestra en la Figura 380 las respuestas como el formato *curl*, la URL de la petición, la respuesta del servidor mostrando código 200, el cuerpo de la respuesta en JSON con la información actualizada del dispositivo:

Figura 380

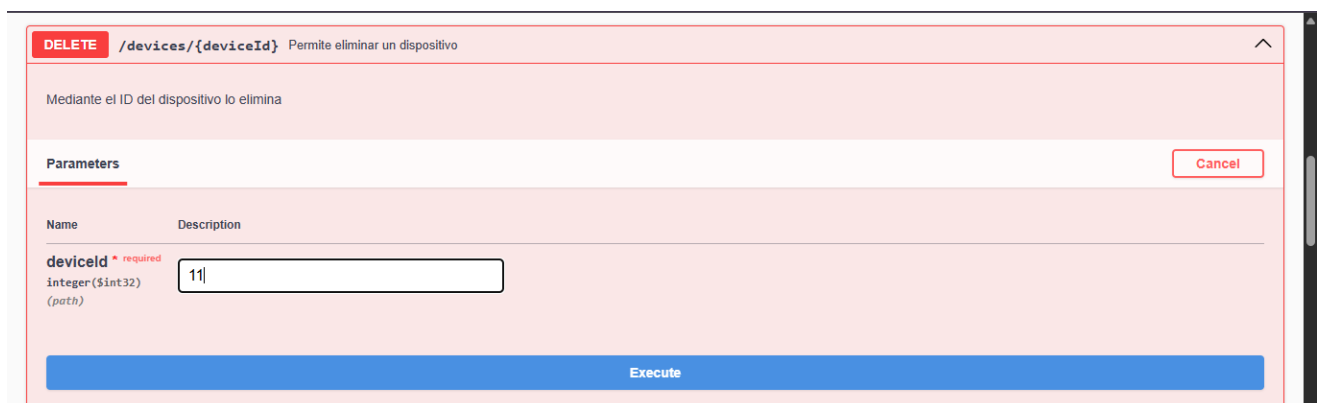
Resultado del segundo endpoint de dispositivos.



El tercer endpoint de la Figura 381 permite eliminar un dispositivo mediante la búsqueda de su ID:

Figura 381

Tercer endpoint de dispositivos.



Como se aprecia en la Figura 382 el resultado de ejecutar el endpoint muestra el formato *curl* de la petición, la URL de la petición, la respuesta del servidor mostrando código 204, en este caso solo aparecen los encabezados de las respuestas:

Figura 382

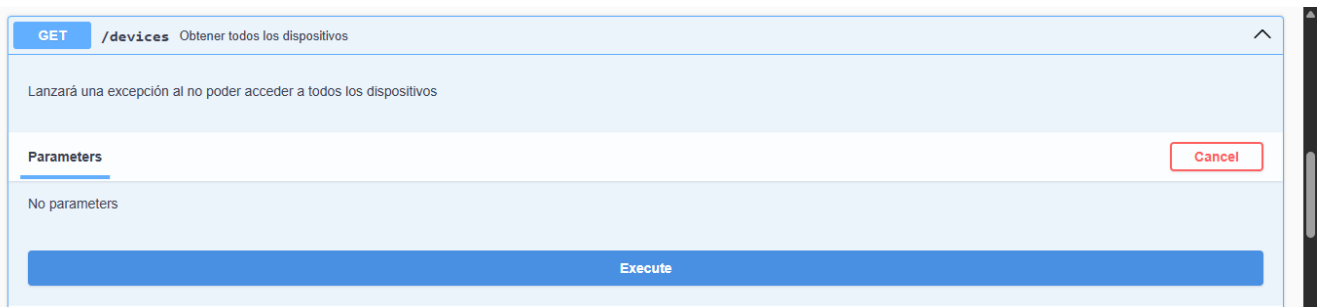
Resultado del tercer endpoint de dispositivos.



La Figura 383 muestra el cuarto endpoint permite que si se intentan ver todos los dispositivos registrados, muestre un mensaje de error:

Figura 383

Cuarto endpoint de dispositivos.



Luego de ejecutar el endpoint se muestra en respuestas el formato *curl* de la petición realizada, la URL de la petición ejecutada, la respuesta del servidor mostrando código 400, como es un error intencional, se muestra un mensaje de error indicando que primero se debe de especificar un usuario, se describe en la Figura 384:

Figura 384

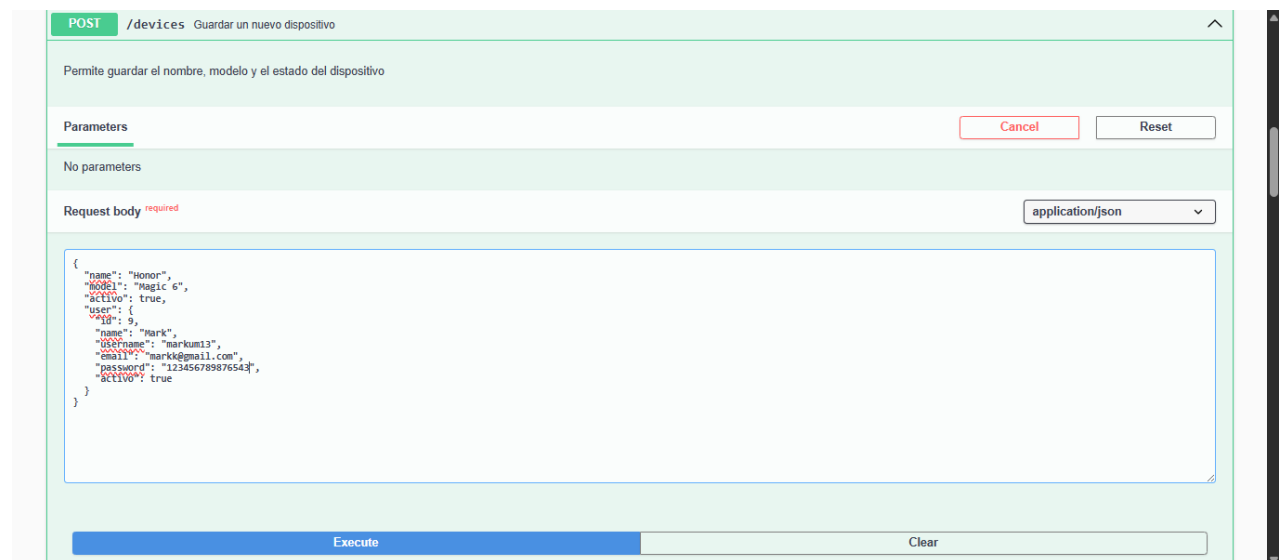
Resultado del cuarto endpoint de dispositivos.



En la Figura 385 se ilustra el quinto endpoint, permite guardar un dispositivo nuevo en formato JSON en el que sin especificar el ID lo guardará en el sistema e igual manera no es necesario especificar todos los datos de un usuario, sino que con solo asociar el ID del usuario es suficiente:

Figura 385

Quinto endpoint de dispositivos.



Como resultado del endpoint en la Figura 386 plasma el formato *curl* de la petición realizada, la URL de la petición mandada, la respuesta del servidor mostrando código 200 con el cuerpo de la respuesta en formato JSON del dispositivo nuevo juntamente con la información del usuario asociado:

Figura 386

Resultado del quinto endpoint de dispositivos.



La Figura 387 muestra el sexto endpoint que actualiza el estado activo del dispositivo mediante el ID del dispositivo en el que se cambiará por el estado nuevo después de la actualización:

Figura 387

Sexto endpoint de dispositivos.

PATCH /devices/{deviceId}/status Permite actualizar el estado activo del dispositivo

Con valores de verdadero y falso

Parameters Cancel

Name	Description
deviceId * required integer(\$int32) (path)	1
activo * required boolean (query)	false

Execute

Como respuesta de la ejecución del endpoint se visualiza el formato *curl* de la petición, la URL de la petición realizada, la respuesta del servidor mostrando código 200, así como un mensaje de la actualización del estado del dispositivo, representado gráficamente en la Figura 388:

Figura 388

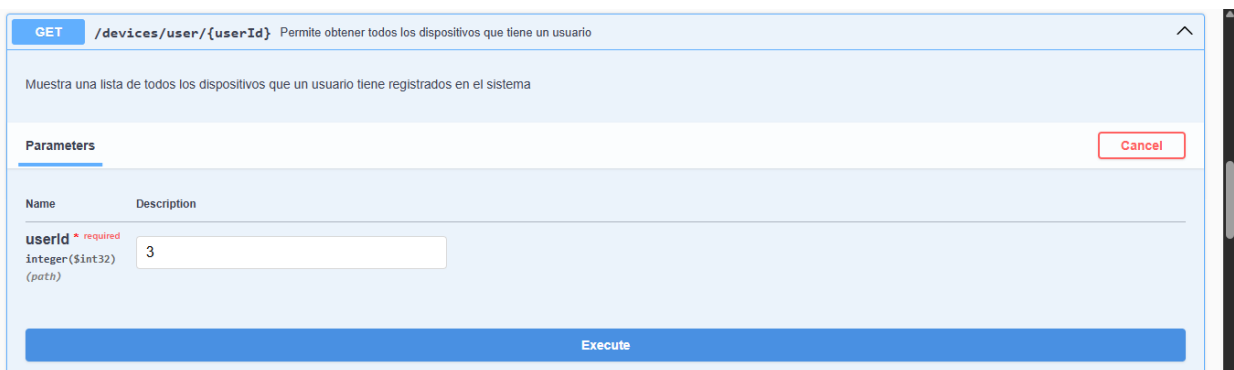
Resultado del sexto endpoint de dispositivos.



La Figura 389 corresponde al último endpoint, permite obtener todos los dispositivos que tiene un usuario mediante su ID:

Figura 389

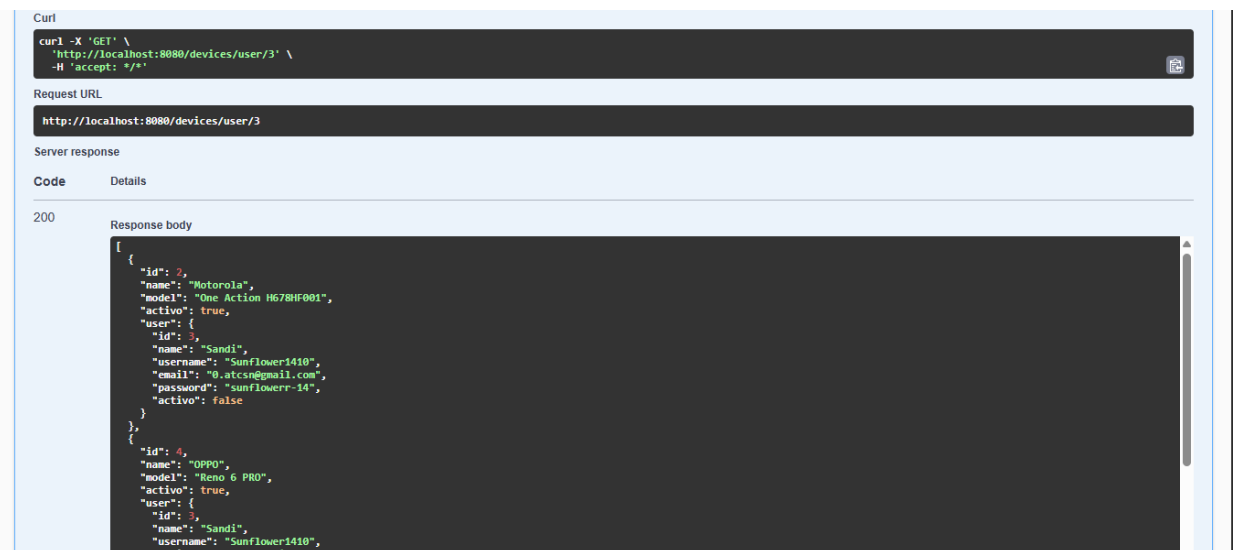
Séptimo endpoint de dispositivos.



El resultado está en la Figura 390 que devuelve en la sección de respuestas objetos JSON de los dispositivos junto con la petición *curl* y la URL de la petición:

Figura 390

Resultado del séptimo endpoint de dispositivos.



En la Figura 391 se aprecia la sección de *endpoints* para “Usuarios” conformado por seis *endpoints* con su descripción:

Figura 391

Colección de endpoints de usuarios.

Usuarios Operaciones relacionadas con la administración de usuarios			
GET	/users/{userId}	Buscar usuario por ID	▼
PUT	/users/{userId}	Permite actualizar la información de un usuario	▼
DELETE	/users/{userId}	Eliminar la cuenta de un usuario	▼
POST	/users/register	Guarda un usuario en el sistema	▼
PATCH	/users/{userId}/status	Permite actualizar el estado activo de la cuenta del usuario	▼
GET	/users	Obtener todos los usuarios	▼

En la Figura 392 se evidencia el primer endpoint que obtiene la información de un usuario mediante su ID:

Figura 392

Primer endpoint de usuarios.

Usuarios Operaciones relacionadas con la administración de usuarios

GET /users/{userId} Buscar usuario por ID

Retorna la información del usuario correspondiente al ID proporcionado

Parameters Cancel

Name	Description
userId * required integer(\$int32) (path)	12

Execute

En el que al ejecutarlo se visualiza en la Figura 393 devuelve el formato *curl* de la petición, la URL de la petición realizada, la respuesta del servidor mostrando código 200, el cuerpo de la respuesta en formato JSON que muestra: ID, nombre, nombre de usuario, email, contraseña y el estado activo de la cuenta:

Figura 393

Resultado del primer endpoint de usuarios.

Responses

Curl

```
curl -X 'GET' \
  'http://localhost:8080/users/12' \
  -H 'accept: */*'
```

Request URL

```
http://localhost:8080/users/12
```

Server response

Code	Details
200	Response body <pre>{ "id": 12, "name": "Carmen", "username": "carwenShi7", "email": "sanchezcarmen@gmail.com", "password": "lolol774", "activo": true }</pre> Download

El segundo endpoint permite actualizar los datos de la cuenta de usuario como, el nombre, nombre de usuario, el correo electrónico, en el que se buscará por el ID del usuario y en formato JSON se actualizan los datos, se encuentra documentada visualmente en la Figura 394:

Figura 394

Segundo endpoint de usuarios.

The screenshot shows a REST client interface for a PUT request to the endpoint `/users/{userId}`. The description states: "Permite actualizar la información de un usuario" and "Permite la edición del nombre, nombre de usuario y del email". The "Parameters" section shows a required parameter `userId` of type `integer($int32)` with a value of `12`. The "Request body" section is set to `application/json` and contains a JSON object:

```
{  "name": "Ma. Carmen",  "username": "CarSH_17",  "email": "carsh@gmail.com"}
```

. There are "Cancel" and "Reset" buttons in the top right.

La Figura 395 corresponde al resultado en formato *curl* de la petición que incluye el tipo de petición realizada, la URL de la petición junto con la respuesta del servidor mostrando código 200 así exponiendo la respuesta del cuerpo en JSON de los datos que se actualizaron:

Figura 395

Resultado del segundo endpoint de usuarios

The screenshot shows the "Responses" section of a REST client. It displays the *curl* command used for the request:

```
curl -X 'PUT' \  'http://localhost:8080/users/12' \  -H 'accept: */*' \  -H 'Content-Type: application/json' \  -d '{  "name": "Ma. Carmen",  "username": "CarSH_17",  "email": "carsh@gmail.com"  }'
```

. The "Request URL" is `http://localhost:8080/users/12`. The "Server response" section shows a status code of `200`. The "Response body" is a JSON object:

```
{  "id": 12,  "name": "Ma. Carmen",  "username": "CarSH_17",  "email": "carsh@gmail.com",  "password": "loio1774",  "activo": true}
```

. There is a "Download" button next to the response body.

En el tercer endpoint permite eliminar la cuenta de un usuario, el cual se realiza mediante la búsqueda del ID del usuario a borrar, se encuentra representado gráficamente en la Figura 396:

Figura 396

Tercer endpoint de usuarios.

The screenshot shows a REST client interface for the endpoint `DELETE /users/{userId}` with the description "Eliminar la cuenta de un usuario". Below the endpoint, there is a section titled "Parameters" with a "Cancel" button. A table lists the parameters:

Name	Description
<code>userId</code> * required <code>integer(\$int32)</code> (path)	7

At the bottom, there is a blue "Execute" button.

Luego de ejecutar la petición, en la Figura 397 se ilustra las respuestas del servidor mostrando código 204 indicando que la solicitud fue exitosa, pero no hay contenido para devolver en la respuesta:

Figura 397

Resultado del tercer endpoint de usuarios.

The screenshot shows the "Responses" section of a REST client. It displays the curl command used for the request:

```
curl -X 'DELETE' \
  'http://localhost:8080/users/7' \
  -H 'accept: */*'
```

Below this, it shows the "Request URL" as `http://localhost:8080/users/7`. The "Server response" section shows a "Code" of 204 and "Details" for the response headers:

```
connection: keep-alive
date: Sat, 17 May 2025 23:56:26 GMT
keep-alive: timeout=60
vary: Origin,Access-Control-Request-Method,Access-Control-Request-Headers
```

En el cuarto endpoint permite crear la cuenta de un nuevo usuario en el sistema en formato JSON ingresando, nombre, nombre de usuario, correo y contraseña, se describe en la Figura 398:

Figura 398

Cuarto endpoint de usuarios.

The screenshot shows a REST client interface with a green header bar. The method is **POST** and the endpoint is **/users/register**. Below the header, there is a description: "Guarda un usuario en el sistema". A sub-header indicates: "Permite guardar el nombre, nombre de usuario, email y contraseña". The **Parameters** section is empty, with "No parameters" displayed. The **Request body** is marked as "required" and the media type is set to **application/json**. The request body contains a JSON object:

```
{  "name": "Quintin",  "username": "carmona08_Q",  "email": "carmonamejorada890@gmail.com",  "password": "quintin_0372"}
```

La Figura 399 muestra que al ejecutar la petición en la sección de repuestas se mostrará la petición *curl*, la URL de la petición, la respuesta del servidor mostrando código 200 y un mensaje indicando que se guardó el usuario:

Figura 399

Resultado del tercer endpoint de usuarios.

The screenshot shows the **Responses** section of a REST client. It displays the **Curl** command used for the request, the **Request URL** as **http://localhost:8080/users/register**, and the **Server response**. The response status is **Code 200**. The **Response body** contains the message: "Usuario registrado correctamente". There are icons for copying and downloading the response body.

En el quinto endpoint documentado en la Figura 400 permite actualizar el estado de la cuenta en el que se ingresa el ID del usuario para activar o desactivar el nuevo estado que tendrá, mediante valores booleanos:

Figura 400

Quinto endpoint de usuarios.

PATCH /users/{userId}/status Permite actualizar el estado activo de la cuenta del usuario

Con valores de verdadero y falso

Parameters Cancel

Name	Description
userId * required integer(\$int32) (path)	3
activo * required boolean (query)	true

Execute

En la Figura 401 se muestra que el resultado de la ejecución la petición en formato *curl*, la URL de la petición y la respuesta del servidor mostrando código 200 y un mensaje indicando que se actualizó el estado de la cuenta:

Figura 401

Resultado del quinto endpoint de usuarios.

Responses

Curl

```
curl -X 'PATCH' \
  'http://localhost:8080/users/3/status?activo=true' \
  -H 'accept: */*'
```

Request URL

```
http://localhost:8080/users/3/status?activo=true
```

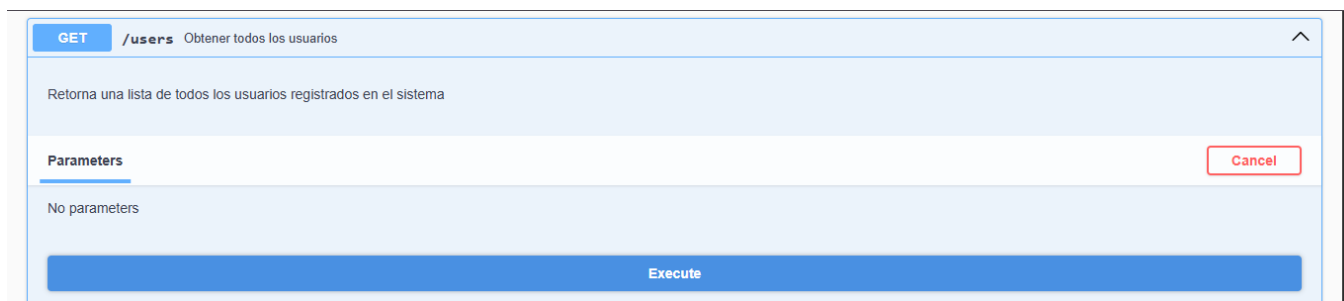
Server response

Code	Details
200	Response body <pre>User status updated successfully</pre> Download

En la Figura 402 se visualiza el último endpoint que obtiene todos los usuarios registrados en el sistema, en la cual no se ingresan parámetros para ejecutar la consulta:

Figura 402

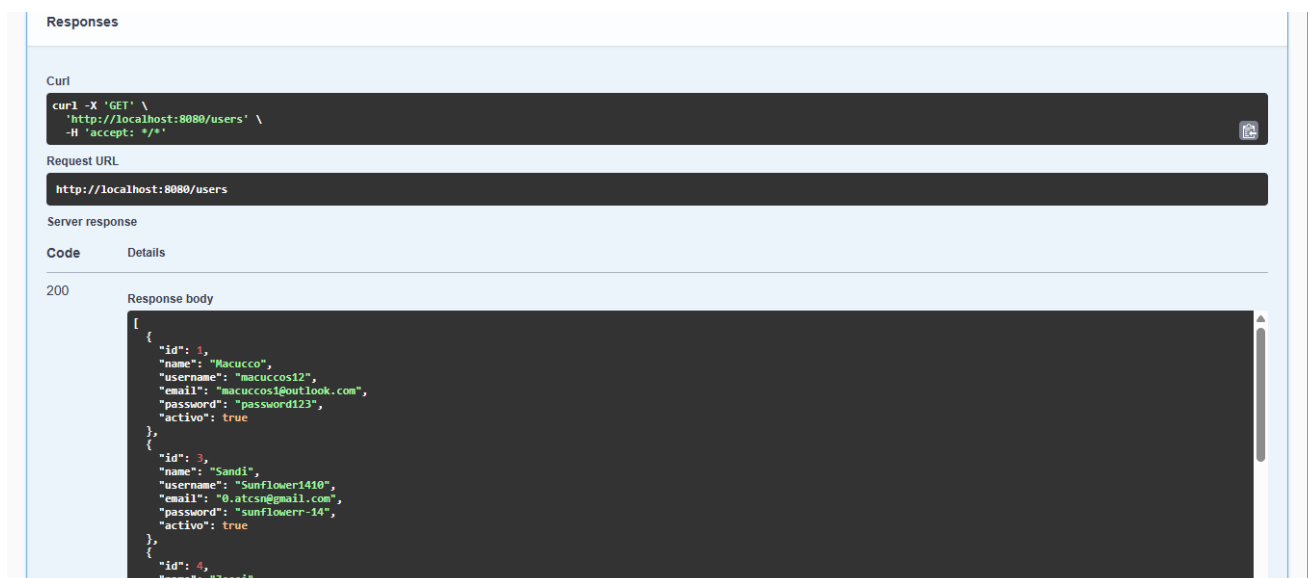
Sexto endpoint de usuarios.



Al ejecutar el endpoint devuelve en la sección de repuestas la petición *curl*, la URL de la petición, mostrando la respuesta del servidor en mostrando código 200, el listado de todas las cuentas almacenadas en la base de datos se describe en la Figura 403:

Figura 403

Resultado del sexto endpoint de usuarios.



Por último, en la Figura 404 se aprecia que en el manual se encuentran los cinco modelos creados:

Figura 404

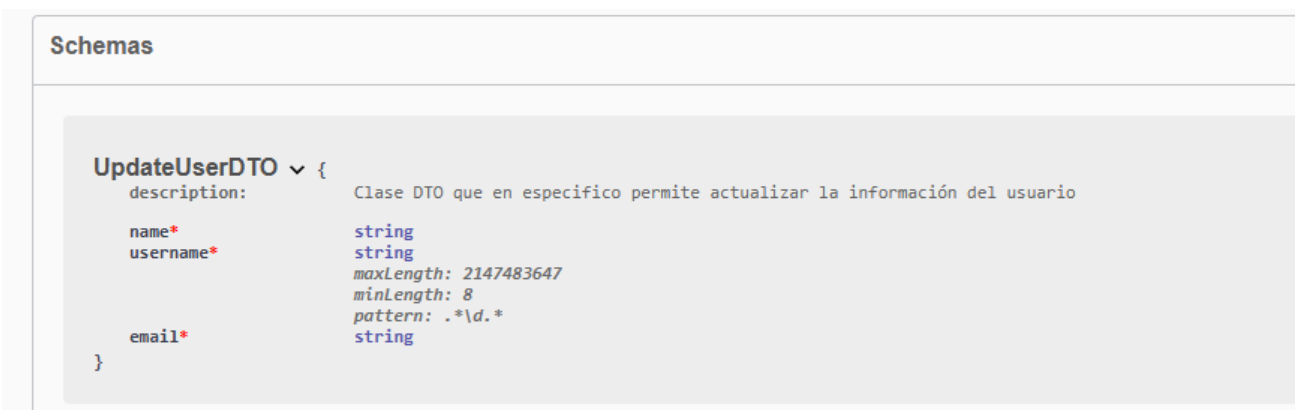
Esquemas creados en el proyecto.



La Figura 405 presenta el primer esquema, correspondiente a una clase “*DTO*” diseñada para actualizar la información de un usuario. Esta clase permite visualizar las validaciones implementadas en la clase `UserServiceImpl`. En esta sección, se muestra el nombre de la clase junto con sus atributos obligatorios, indicados con asteriscos, así como el tipo de dato asociado a cada atributo:

Figura 405

Esquema UpdateUserDTO .



La Figura 406 describe la segunda clase, correspondiente al modelo “User”, que representa la estructura de la tabla ‘users’ en la base de datos. En esta figura se muestran los campos obligatorios y no obligatorios, junto con el tipo de dato asociado a cada uno:

Figura 406

Esquema User.

User ▾ {	
description:	Entidad que representa a un usuario en el sistema
id	integer(\$int32)
name*	string
username*	string
	maxLength: 2147483647
	minLength: 8
	pattern: .*\\d.*
email*	string
password*	string
	maxLength: 2147483647
	minLength: 8
activo	boolean
}	

La Figura 407 corresponde al esquema “Device”, que incluye sus atributos propios, uno de los cuales es una llave foránea que se conecta al modelo “User”. En esta figura también se pueden ver todos los atributos que conforman la clase “User”, lo que permite plasmar la relación de las clases en esta documentación:

Figura 407

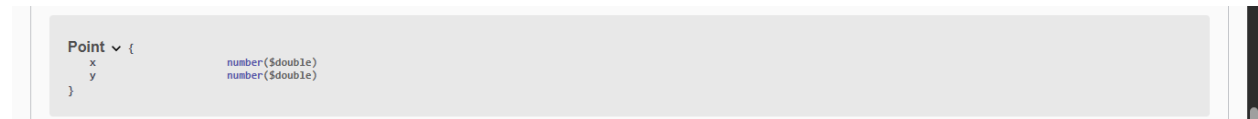
Esquema Device.

Device ▾ {	
description:	Entidad que representa a un dispositivo en el sistema
id	integer(\$int32)
name	string
model	string
activo	boolean
user	User ▾ {
	description:
	Entidad que representa a un usuario en el sistema
	id
	integer(\$int32)
	name*
	string
	username*
	string
	maxLength: 2147483647
	minLength: 8
	pattern: .*\\d.*
	email*
	string
	password*
	string
	maxLength: 2147483647
	minLength: 8
	activo
	boolean
	}
}	

La Figura 408 ilustra la clase “Point”, que representa la ubicación en coordenadas “x” y “y”, ambas de tipo double, permitiendo almacenar con precisión la longitud de cada coordenada:

Figura 408

Esquema Point.



Finalmente en la Figura 409 presenta el esquema “Position”, que incluye sus atributos y dos llaves foráneas: una que se conecta a la clase “Point” convertir la ubicación en coordenadas “x” y “y”, mientras que la otra que se relaciona con el esquema “Device”:

Figura 409

Esquema Position.



En la documentación se puede desplegar a detalle todos los atributos que conforman los esquemas que están relacionadas entre sí.

V. Conclusiones

5.1. Conclusiones del proyecto

El proyecto ha demostrado que la implementación del desarrollo de un backend REST en Spring Boot ha permitido alcanzar el objetivo de desarrollar un servicio REST capaz de recibir, almacenar, gestionar posiciones geográficas de dispositivos móviles en una base de datos MySQL, permitiendo su consulta y administración de manera eficiente.

Los resultados obtenidos sugieren que al desarrollar un backend REST en Spring Boot permite que las ubicaciones se almacenen en la base de datos relacional lo que permite conectar este backend al portal web que a su vez será utilizado por la aplicación móvil para realizar el almacenamiento de datos de la información de los usuarios, el cual la API creada permitirá recibir las solicitudes del frontend.

De esta manera, los datos se almacenan de forma escalable a largo plazo, lo que ofrece una alternativa de guardado de ubicaciones en un servidor externo, en lugar de almacenar solo el historial de ubicaciones en los dispositivos móviles como lo ha implementado Google en estos últimos años.

5.2. Recomendaciones

Algunas recomendaciones importantes es primero revisar la documentación en OpenAPI para tener mejor comprensión del backend REST creado, el cual permitirá a los desarrolladores tener de manera tangible la estructura de los *endpoints* realizados permitiendo probar el funcionamiento de cada uno, teniendo como resultado mejor comprensión.

La segunda recomendación es visualizar la estructura del *backend* hecho en lenguaje Java mediante el framework Spring Boot, en el que si los programadores *frontend* tienen conocimiento sobre Java con Spring Boot revisen la estructura del código y realicen el archivo de comunicación para que el backend y frontend interactúen.

Como tercera recomendación es si la documentación en OpenAPI no es suficiente tienen la opción de acceder a la colección hecha en POSTMAN, que tiene guardada en cada petición la estructura completa de la URL.

Estas recomendaciones facilitarán el desarrollo de las siguientes fases del proyecto como en el desarrollo de la interfaz de la aplicación móvil, el portal web así mismo como la implementación de la seguridad, que se podrán conectar a esta API REST para así gestionar los datos en una base de datos relacional.

5.3. Experiencia profesional adquirida

A través de mi experiencia personal de este proyecto, he adquirido habilidades valiosas en el área de desarrollo de software, específicamente en el backend que me ha permitido implementar tecnologías que permiten realizar de manera rápida, escalable y robusta construir backend para que pueda seguir evolucionando el proyecto.

También he aprendido la importancia de tener el código organizado usando técnicas de código limpio y separar las responsabilidades en el código para no generar deuda técnica. Esto me ha permitido crecer profesionalmente y mejorar mi capacidad para el desarrollo de backend.

Asimismo, el uso de técnicas de testing me permitió explorar esta área importante del desarrollo de software, garantizando que lo que realicé funciona de la manera esperada. El backend considero que es una de las áreas de desarrollo más importante, que me permitió desenvolverme para aplicar lo que previamente aprendí, complementándolo con este proyecto.

VI. Competencias desarrolladas

El desarrollo del proyecto permitió la adquisición de conocimientos técnicos, metodológicos y profesionales que contribuyen de forma significativa al perfil del desarrollador de software, en específico en el backend. A nivel técnico, se fortalecieron las habilidades en la construcción de API REST utilizando el framework Spring Boot, partiendo de conocimientos previos básicos. La implementación de persistencia de datos mediante JPA e Hibernate facilitó la gestión eficiente de bases de datos relacionales como MySQL. Asimismo, se logró familiarización con herramientas esenciales como Maven, para la automatización del ciclo de vida del proyecto; OpenAPI para la documentación de los endpoints desarrollados en el backend.

En cuanto a las habilidades metodológicas, al incorporar principios de Clean Code para asegurar un desarrollo claro así como mantenible en el que me obligó a salir de la zona de confort, para que los próximos programadores entiendan este backend de manera fácil. Por otra parte, al trabajar bajo metodologías ágiles como Scrum, adquirí la noción de cómo se trabaja en el ámbito laboral para la organización del equipo. De igual forma, el uso de Kanban para la planificación y seguimiento de tareas mediante GitLab, herramienta en la cual no había trabajado, sino con su variante GitHub.

Algo nuevo fue el desarrollo de pruebas de software, en donde implementé pruebas unitarias, de integración y de aceptación para asegurar que el backend esté preparado para las solicitudes que se realicen en el frontend. Para finalizar, a nivel profesional, desarrollé competencias clave como la autonomía en el aprendizaje, la resolución de problemas técnicos, la comunicación efectiva de los avances, la capacidad de adaptación a nuevos desafíos con tecnologías de las cuales no tenía mucha práctica. Además, adquirí un compromiso con la calidad del producto entregado.

VII. Fuentes de información

- Álvarez, C. (11 de julio de 2024). *Maven: ¿Que es? .¿Como funciona?*
<https://www.arquitecturajava.com/que-es-Maven/>
- Baker, J. (17 de septiembre de 2019). *Spring y Spring Boot Frameworks: una breve historia.*
<https://dzone.com/articles/history-of-spring-framework-spring-boot-framework>
- Balazs, R. (17 de octubre de 2023). *Java Hibernate vs JPA: What is the difference?*
<https://bluebirdinternational.com/java-Hibernate-vs-JPA/>
<https://bluebirdinternational.com/java-Hibernate-vs-JPA/>
- Bass, L., Clements, P., & Kazman, R. (2003). *Software Architecture in Practice*. Addison-Wesley.
https://doi.org/https://www.google.com.mx/books/edition/Software_Architecture_in_Practice/mdilu8Kk1WMC?hl=es-419&gbpv=0
- Bernal, D. (16 de agosto de 2021). ✨📖 *Comenzando con Java: Presentación, orígenes e historia del lenguaje de programación, Hola Mundo....*
<https://medium.com/p/f1e8acf1d0af#abdb>
- Canle, E. (18 de marzo de 2025). *¿Qué es Spring Boot y para qué sirve?*
<https://www.tokioschool.com/noticias/spring-boot/>
- Choudhary, B. (29 de abril de 2025). *9 Clean Code Java Principles For A High Performance Application.* <https://www.finoit.com/articles/clean-code-java/>
- De Dios, M. (04 de septiembre de 2024). *Qué es Scrum y cómo funciona esta metodología de trabajo.* . <https://www.wearemarketing.com/es/blog/scrum-que-es-como-funciona.html#>
- Del Vayo, A. (18 de noviembre de 2024). *Adiós en Google Maps a tu historial de ubicaciones: esto es lo que tienes que hacer para no perderlo para siempre.*
https://www.elespanol.com/elandroidelibre/aplicaciones/20241118/adios-google-maps-historial-ubicaciones-hacer-no-perderlo-siempre/902159829_0.html
- García, M. (2025). *Qué es y para qué sirve Java Spring Boot.* <https://platzi.com/blog/que-es-spring-boot/>
- GeeksforGeeks. (13 de marzo de 2025). *Spring Data JPA - Find Records From MySQL.*
<https://www.geeksforgeeks.org/spring-data-JPA-find-records-from-MySQL/>

- Joyanes, L., & Zahonero, I. (2011). *PROGRAMACIÓN EN JAVA 6. Algoritmos, programación orientada a objetos e interfaz gráfica de usuario*. Mc Graw-Hill.
- Kruchten, P. (1995). Architectural Blueprints—The “4+1” View Model of Software Architecture. *IEEE Software*, 12(6), 42–50. <https://doi.org/https://doi.org/10.1109/52.469759>
- Laoyan, S. (08 de enero de 2025). *¿Qué es la Metodología Agile y cómo revoluciona la gestión de proyectos?* <https://asana.com/es/resources/agile-methodology>
- López, J. (10 de mayo de 2023). *Cómo aplicar técnicas de clean code en Java 18*. <https://openwebinars.net/blog/como-aplicar-tecnicas-de-clean-code-en-java-18/#:~:text=Clean%20Code%20es%20una%20metodolog%C3%ADa,Code%20de%20forma%20muy%20eficiente.>
- Lowe, D., Casal, J., Hoppen, M., Bartel, S., Carmichael, A., Bozheva, T., . . . Schwillens, W. (febrero de 2021). *LA GUÍA OFICIAL DEL MÉTODO KANBAN*. https://resources.Kanban.university/wp-content/uploads/2021/08/The-Official-Kanban-Guide_Spanish_A4.pdf
- Maldonado, R. (19 de septiembre de 2024). *POJO en Java: Aprende a crear código más simple*. <https://keepcoding.io/blog/que-es-un-pojo-en-java/>
- Martin, R. (2009). *Código limpio. Manual del estilo para el desarrollo ágil de software*. <https://doi.org/https://elhacker.info/manuales/Lenguajes%20de%20Programacion/Codigo%20limpio%20-%20Robert%20Cecil%20Martin.pdf>
- Martins, J. (19 de enero de 2025). *¿Qué es la metodología Kanban y cómo funciona?* <https://asana.com/es/resources/what-is-Kanban>
- Menzinsky, A., López, G., & Palacio, J. (25 de julio de 2016). *Guía de Scrum Manager*. https://www.scrummanager.com/files/sm_proyecto.pdf
- Moya, R. (31 de marzo de 2012). *Modelo “4+1” vistas de Kruchten (para Dummies)*. <https://jarroba.com/modelo-41-vistas-de-kruchten-para-dummies/>
- Nache, R. (22 de diciembre de 2022). *Metodologías ágiles de desarrollo de software más usadas*. <https://www.armadilloamarillo.com/blog/metodologias-agiles-de-desarrollo-software/>
- Nieva, G. (19 de febrero de 2019). *Versiones y ediciones de Java*. <https://dcodingames.com/versiones-y-ediciones-de-java/>

Souza, B. (02 de febrero de 2023). *¿Qué es Maven?* <https://www.aluracursos.com/blog/que-es-Maven>

Vergara, A. (07 de septiembre de 2016). *JPA vs Hibernate ¿Cual es la diferencia?* <https://www.blog.facilcloud.com/noticias/JPA-vs-Hibernate/>

Vyas, R. (23 de diciembre de 2024). *Marco de API de persistencia de Java (JPA) para el acceso a bases de datos con ejemplo.* <https://www.index.dev/blog/java-persistence-api-JPA-guide>

Константин. (27 de febrero de 2021). *Lo que necesita saber sobre Spring: historia, módulos clave, comparación con Java EE.* <https://javarush.com/es/groups/posts/es.3546.lo-que-necesita-saber-sobre-spring-historia-mdulos-clave-comparacin-con-java-ee>

VIII. Anexos

8.1. Glosario

API REST: interfaz de programación que permite la comunicación entre diferentes sistemas a través del protocolo HTTP, respetando principios de arquitectura REST. Facilita operaciones como creación, lectura, actualización y eliminación de datos.

Backend: parte del sistema que se encarga de la lógica de negocio, la gestión de datos y la comunicación con bases de datos o servicios externos. Es invisible para el usuario final.

Código 200 OK: código de estado HTTP que indica que la solicitud se ha procesado correctamente y se ha devuelto una respuesta exitosa al cliente.

Clean Code: conjunto de buenas prácticas en programación orientadas a escribir código legible, mantenible y fácil de entender por otros desarrolladores.

Código 204 No Content: indica que la solicitud fue exitosa, pero no se devuelve contenido en la respuesta. Se usa comúnmente para operaciones de eliminación.

Código 400 Bad Request: solicitud que indica al cliente como incorrecta o no puede ser procesada debido a errores en los datos.

Código 404 Not Found: se utiliza cuando el recurso solicitado no fue encontrado en el servidor.

Código 409 Conflict: indica que hubo un conflicto con el estado actual del recurso, como intentar crear un elemento que ya existe.

Código 500 Internal Server Error: error general que indica que ocurrió una condición inesperada en el servidor, impidiendo procesar la solicitud.

Curl: en el desarrollo de APIs es usado comúnmente para probar endpoints y enviar datos a servicios REST.

DTO (Data Transfer Object): objeto utilizado para transferir datos entre capas de una aplicación. Permite encapsular y estructurar la información que se envía o recibe en las solicitudes, sin exponer directamente las entidades del modelo de base de datos.

Endpoint: punto de acceso definido en una API donde se pueden ejecutar operaciones específicas mediante solicitudes HTTP (como GET, POST, PUT o DELETE).

Firebase: plataforma de desarrollo de aplicaciones proporcionada por Google, que ofrece servicios *backend* como bases de datos en tiempo real y autenticación. En el proyecto, se menciona como una alternativa no elegida.

Flyway: herramienta que permite gestionar versiones de esquemas en bases de datos. Automatiza la ejecución de scripts SQL para mantener la integridad del modelo de datos.

Frontend: parte visual de un sistema con la que el usuario interactúa directamente. Aunque no se desarrolló en este proyecto, el *backend* fue diseñado para integrarse con aplicaciones móviles o web.

GitLab: plataforma de control de versiones basada en Git que permite la gestión de proyectos mediante repositorios, así como el seguimiento de tareas usando metodologías ágiles.

Hibernate: herramienta de mapeo objeto-relacional utilizada en Java que permite interactuar con bases de datos relacionales sin escribir consultas SQL manuales.

JPA (Java Persistence API): API de Java utilizada para la gestión de la persistencia de datos. Define una forma estándar de mapear objetos Java a bases de datos relacionales.

Kanban: método de gestión visual de tareas que organiza el trabajo en columnas (como “Por hacer”, “En proceso” y “Terminado”) para optimizar el flujo de trabajo.

Maven: herramienta para la automatización del ciclo de vida de proyectos Java. Facilita la gestión de dependencias, compilación, pruebas y empaquetado del software.

MySQL: Sistema de gestión de bases de datos relacional ampliamente utilizado en entornos de desarrollo web y empresarial.

OpenAPI: especificación utilizada para documentar APIs REST, facilitando la visualización, prueba y comprensión de los *endpoints* disponibles.

Postman: herramienta utilizada para realizar pruebas a APIs REST, enviando solicitudes HTTP y visualizando las respuestas de forma estructurada.

Scrum: marco de trabajo ágil para la gestión de proyectos, basado en ciclos iterativos llamados sprints, roles definidos y reuniones de seguimiento frecuentes.

Spring Boot: framework de desarrollo en Java que simplifica la creación de aplicaciones robustas mediante configuraciones predeterminadas y una arquitectura modular.

8.2. Presentación mostrada en la empresa

Las siguientes Figuras corresponden a la presentación creada para exposición del proyecto:

Figura 410

Portada de la presentación del proyecto.



Figura 411

Sección de antecedentes en la presentación a la empresa.



Figura 412

Sección del objetivo general del proyecto en la presentación a la empresa.



Figura 413

Sección de objetivos específicos en la presentación a la empresa.

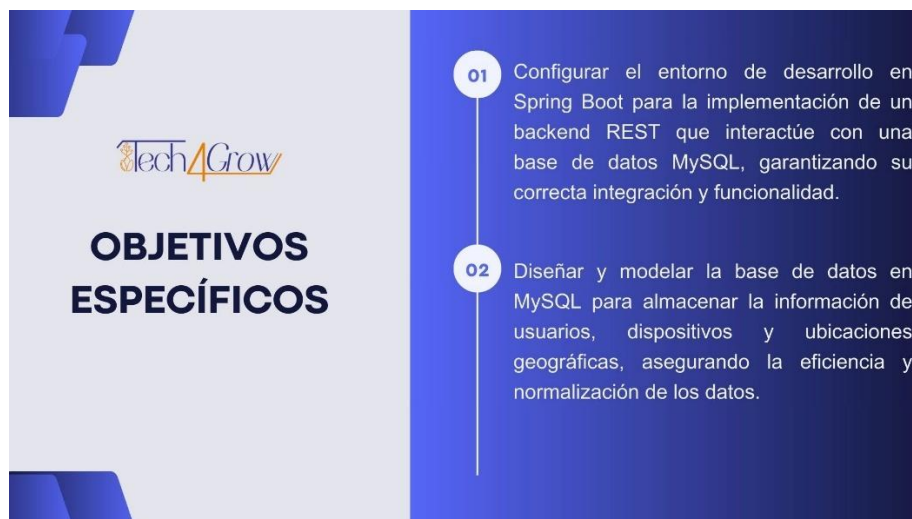
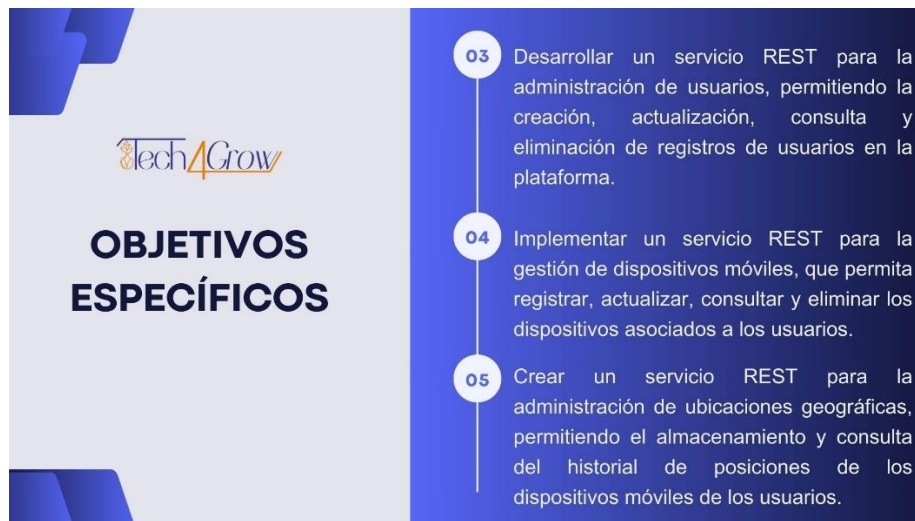


Figura 414

Continuación de los objetivos específicos en la presentación a la empresa.



OBJETIVOS ESPECÍFICOS

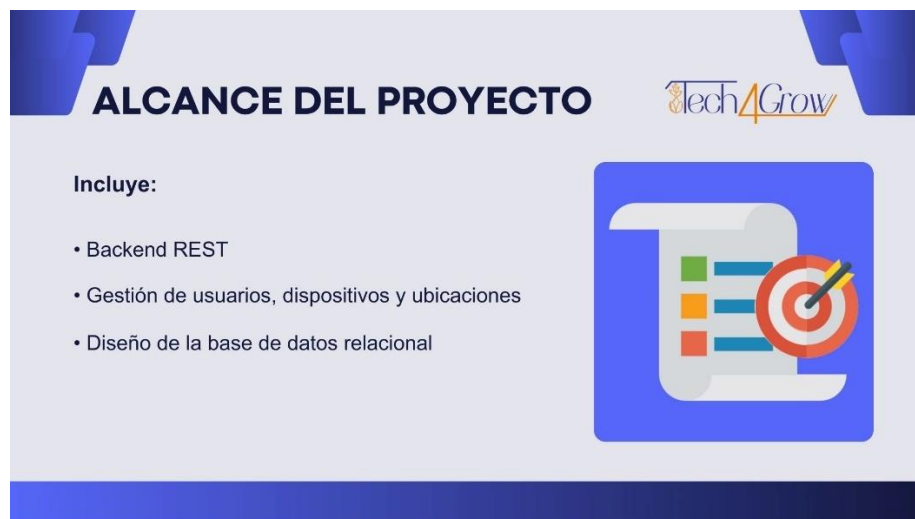
03 Desarrollar un servicio REST para la administración de usuarios, permitiendo la creación, actualización, consulta y eliminación de registros de usuarios en la plataforma.

04 Implementar un servicio REST para la gestión de dispositivos móviles, que permita registrar, actualizar, consultar y eliminar los dispositivos asociados a los usuarios.

05 Crear un servicio REST para la administración de ubicaciones geográficas, permitiendo el almacenamiento y consulta del historial de posiciones de los dispositivos móviles de los usuarios.

Figura 415

Sección de alcances del proyecto en la presentación a la empresa.



ALCANCE DEL PROYECTO

Incluye:

- Backend REST
- Gestión de usuarios, dispositivos y ubicaciones
- Diseño de la base de datos relacional



Figura 416

Sección de fuera de alcance en la presentación a la empresa.



Figura 417

Sección de metodología de trabajo en la presentación a la empresa.



Figura 418

Sección de tecnologías utilizadas en la presentación a la empresa.

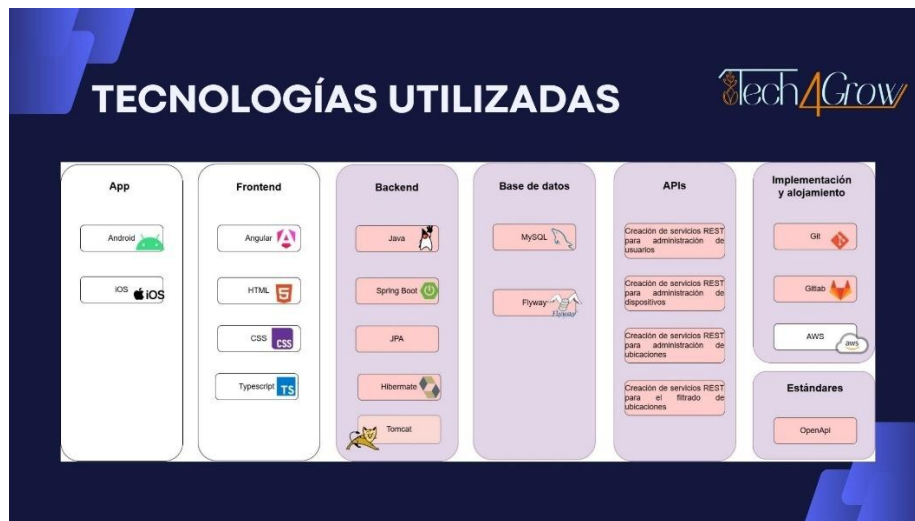


Figura 419

Sección de resultados en la presentación a la empresa.

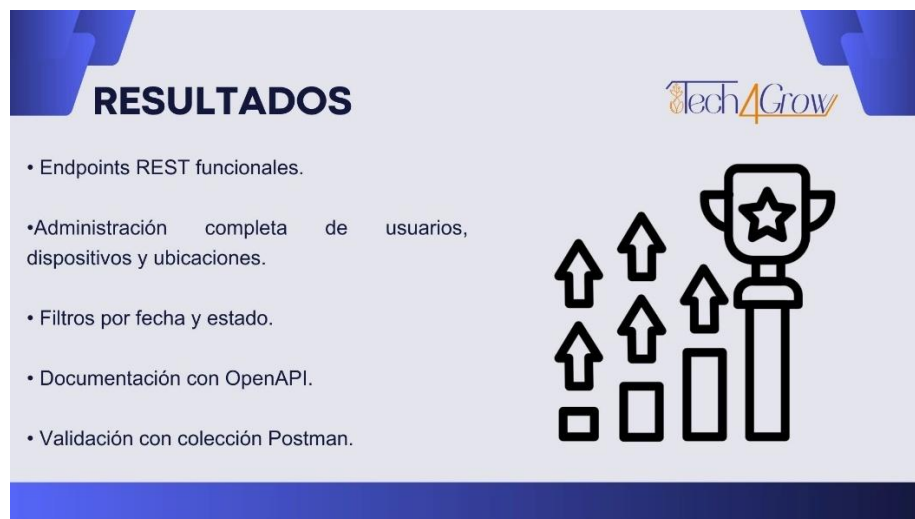


Figura 420

Sección de conclusiones en la presentación a la empresa.

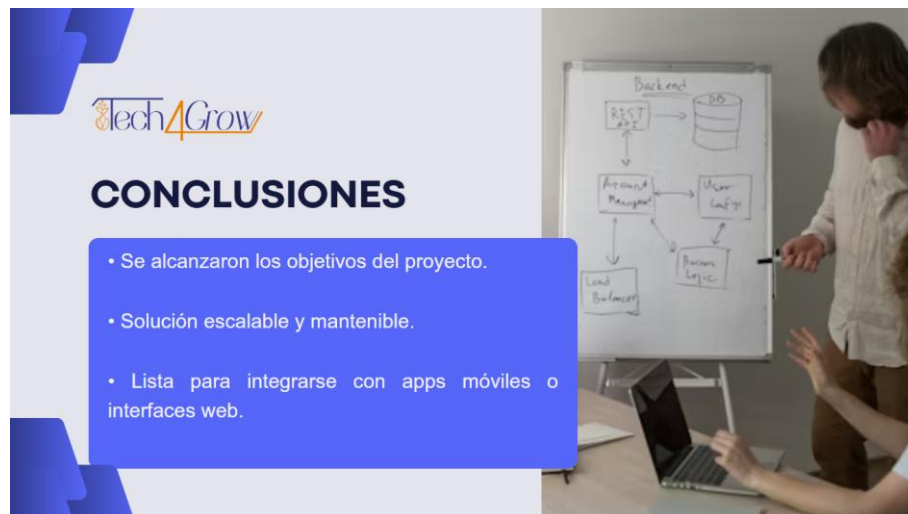


Figura 421

Sección de agradecimientos en la presentación a la empresa.

