



UNIVERSIDAD DE JAÉN
Escuela Politécnica Superior de Jaén

Trabajo Fin de Grado

Escriba: Plataforma web social para la creación y divulgación de textos literarios

Alumno: Marco Antonio Carrión Soriano

Tutor: Prof. D. José Ramón Balsas Almagro
Dpto: Informática

Junio, 2023

Tabla de contenidos

Tabla de Ilustraciones.....	4
Relación de Tablas.....	8
1. Introducción.....	9
1.1. Motivación.....	9
1.2. Objetivos.....	10
1.3. Metodología.....	11
1.4. Estructura del documento.....	12
2. Análisis.....	14
2.1. Análisis preliminar.....	14
2.1.1. Aplicaciones similares.....	15
2.1.1. Wattpad.....	15
2.1.2. Google docs.....	17
2.1.2. Funcionalidades y necesidades.....	18
2.1.3. Estudio de tecnologías.....	19
2.1.3.1. Tecnología front-end: React.....	20
2.1.3.2. Tecnología back-end: Spring boot.....	23
2.1.3.2. API generadora de imágenes: Stable Diffusion.....	26
2.2. Propuesta de solución.....	27
2.4. Historias de Usuario.....	28
2.5. Planificación inicial de tareas.....	45
2.5.1 Definición de los Sprints.....	49
2.5.1.1 Sprint 1.....	50
2.5.1.2 Sprint 2.....	52
2.5.1.3 Sprint 3.....	54
2.5.1.4 Sprint 4.....	57
2.5.1.4 Sprint 5.....	59
2.5.2. Evolución de la planificación inicial.....	60
2.6. Estudio de viabilidad.....	62
2.6.1- Costes Software.....	63
2.6.2- Costes Hardware.....	65
2.6.3- Salarios.....	66
2.6.4- Costes totales del proyecto.....	67
2.6.4- Costes de explotación del sistema.....	68
2.6.4- Modelos de negocio.....	69
2.7. Modelo de dominio.....	71
3. Diseño.....	77

3.1. Diagrama Entidad-Relación.....	77
3.2. Diagrama de Clases.....	81
3.2.1- Diagrama de clases del Servidor.....	82
3.2.2- Diagrama de clases del Cliente.....	87
3.3. Diagramas de secuencia.....	94
3.3.1. Inicio de sesión.....	94
3.3.2. Edición del texto de un documento.....	97
3.3.3. Creación de un comentario.....	102
3.3.4. Búsqueda de documentos y usuarios.....	105
3.4. Diseño de la Interfaz.....	108
3.4.1. Diseño de la API REST.....	108
3.4.2. Diseño Interfaz del cliente.....	117
3.4. Plan de pruebas.....	128
4. Implementación.....	131
4.1. Arquitectura.....	131
4.2. Detalles sobre implementación.....	134
4.2.1. Detalles sobre la implementación del servidor.....	134
4.2.1.1. Creación del proyecto.....	134
4.2.1.2. Estructura del proyecto.....	135
4.2.1.3. Definición de las entidades.....	137
4.2.1.4. Repositorios.....	141
4.2.1.5. Servicios.....	142
4.2.1.6. Controladores REST.....	144
4.2.1.7. Definición de los DTOs.....	147
4.2.1.8. Autenticación y control de acceso.....	148
4.2.1.9. Generación y consulta de documentación API REST.....	157
4.2.2. Implementación del cliente.....	158
4.2.2.1. Creación del proyecto.....	158
4.2.2.2. Estructura del proyecto.....	158
4.2.2.3. Definición de componentes.....	161
4.2.2.4. Estilos y responsividad.....	168
4.2.2.5. Servicios y peticiones al servidor.....	172
4.2.2.5. Interceptor e identificación de peticiones.....	174
4.2.2.6. Guardianes.....	177
4.2.2.7. Validaciones en el cliente.....	179
4.2.2.8. Despliegue de la API de generación de imágenes.....	183
5. Pruebas.....	187
5.1. Pruebas al Back-end.....	187

5.2. Pruebas de front-end.....	197
6. Conclusiones.....	204
6.1. Mejoras y trabajos futuros.....	208
7. Bibliografía.....	210
Apéndice I. Detalles diseño y funcionamiento controladores REST.....	213
Apéndice II. Manual de Usuario.....	221
Apéndice III. Descripción de contenidos suministrados.....	244
Apéndice IV. Manual de Instalación del sistema.....	246

Tabla de Ilustraciones

Ilustración 1. Ejemplo de funcionalidades del editor Wattpad.....	16
Ilustración 2. Funcionalidades del editor de texto Google docs.....	18
Ilustración 3. Gráfico burn-down comparación de planificaciones.....	61
Ilustración 4. Diagrama de modelo de dominio.....	73
Ilustración 5. Diagrama Entidad-Relación.....	80
Ilustración 6. Funcionamiento arquitectura cliente - servidor.....	81
Ilustración 7. Diagrama de clases del servidor.....	84
Ilustración 8. Diagrama de clases del cliente.....	90
Ilustración 9. Diagrama de secuencia inicio de sesión.....	96
Ilustración 10. Diagrama de secuencia edición de documento.....	101
Ilustración 11. Diagrama de secuencia creación de comentario.....	104
Ilustración 12. Diagrama de secuencia búsqueda documento.....	107
Ilustración 13. Fragmento de documentación swagger.....	109
Ilustración 14. Información de método desplegado en Swagger.....	110
Ilustración 15. Método con parámetro de objeto en Swagger.....	111
Ilustración 16. Modelo de la API en Swagger.....	112
Ilustración 17. Controlador documento en documentación Swagger.....	114
Ilustración 18. Método PATCH Controlador documentos.....	115
Ilustración 19. Método GET /document/genres del controlador documento.....	117
Ilustración 20. Diagrama jerárquico de la interfaz del cliente.....	119
Ilustración 21. Wireframe vista Profile en PC.....	122
Ilustración 22. Wireframe vista Profile en móvil.....	123
Ilustración 23. Wireframe vista Home en PC.....	124
Ilustración 24. Wireframe vista Home en dispositivo móvil.....	125
Ilustración 25. Wireframe vista Editor en PC.....	126
Ilustración 26. Wireframe vista Editor en dispositivo móvil.....	127
Ilustración 27. Interfaz de Postman.....	129
Ilustración 28. Diagrama arquitectónico.....	131
Ilustración 29. Ejemplo de la interfaz de spring inicializr.....	135
Ilustración 30. Jerarquía de carpetas del servidor.....	136
Ilustración 31. Diagrama de paquetes del servidor.....	137
Ilustración 32. Ejemplo creación de una entidad.....	138
Ilustración 33. Ejemplo de implementación de atributos.....	139
Ilustración 34. Ejemplo de implementación de relaciones.....	140

Ilustración 35. Ejemplo de implementación de Repositorio.....	141
Ilustración 36. Ejemplo de queries en repositorio.....	142
Ilustración 37. Definición de servicio e inyección de dependencias.....	143
Ilustración 38. Ejemplo lógica de negocio.....	144
Ilustración 39. Ejemplo anotaciones controlador REST.....	145
Ilustración 40. Ejemplo implementación end-point.....	146
Ilustración 41. Ejemplo manejo de excepciones del controlador.....	147
Ilustración 42. Ejemplo DTO.....	148
Ilustración 43. Controlador de autenticación.....	150
Ilustración 44. Servicio de detalles de usuario.....	151
Ilustración 45. Implementación detalles de usuario.....	152
Ilustración 46. Punto de entrada JWT.....	153
Ilustración 47. Proveedor JWT.....	154
Ilustración 48. Filtro JWT.....	155
Ilustración 49. Filtro JWT.....	156
Ilustración 50. Autorización métodos HTTP.....	157
Ilustración 51. Dependencias Swagger.....	157
Ilustración 52. Anotaciones de Swagger.....	158
Ilustración 53. Estructura de las carpetas del cliente.....	159
Ilustración 54. Ejemplo organización de los componentes.....	160
Ilustración 55. Diagrama de paquetes del cliente.....	161
Ilustración 56. Ejemplo definición de propiedades.....	162
Ilustración 57. Ejemplo definición de propiedades.....	163
Ilustración 58. Ejemplo definición de propiedades.....	164
Ilustración 59. Ejemplo definición Use State.....	164
Ilustración 60. Ejemplo de Use Effect.....	165
Ilustración 61. Ejemplo 1 de Helper Functions.....	166
Ilustración 62. Ejemplo 2 de Helper Functions.....	166
Ilustración 63. Ejemplo de código JSX.....	167
Ilustración 64. Ejemplo de código JSX.....	168
Ilustración 65. Ejemplo de código CSS.....	170
Ilustración 66. Ejemplo de Media Query CSS.....	171
Ilustración 67. Ejemplo funciones servicio.....	173
Ilustración 68. Importar funciones de un servicio.....	174
Ilustración 69. Ejemplo llamada a un servicio desde un componente.....	174
Ilustración 70. Métodos del interceptor.....	175
Ilustración 71. Métodos de servicio de tokens.....	177
Ilustración 72. Enrutamiento de la aplicación cliente.....	178

Ilustración 73. Ejemplo llamada a un servicio desde un componente.....	179
Ilustración 74. Ejemplo funciones para la gestión de eventos.....	181
Ilustración 75. Validación y envío de datos.....	182
Ilustración 76. Ejemplo de validación asíncrona.....	183
Ilustración 77. Despliegue de la API generadora de imágenes.....	185
Ilustración 78. Servicio generación de imágenes.....	186
Ilustración 79. Intento de registro con usuario en uso.....	189
Ilustración 80. Intento de registro sin cumplir restricciones.....	190
Ilustración 81. Registro exitoso.....	191
Ilustración 82. Inicio de sesión fallido.....	192
Ilustración 83. Inicio de sesión exitoso.....	193
Ilustración 84. Uso del token JWT.....	194
Ilustración 85. Petición a documento privado.....	194
Ilustración 86. Petición de documento válido.....	195
Ilustración 87. Creación de documento con género inexistente.....	196
Ilustración 88. Petición de documento con género existente.....	197
Ilustración 89. Formulario de registro con datos vacíos.....	199
Ilustración 90. Formulario de registro con datos erróneos.....	200
Ilustración 91. Intento de acceso a URL protegida.....	201
Ilustración 92. Acceso válido a un documento.....	201
Ilustración 93: Vista perfil en versión PC.....	202
Ilustración 94. Vista perfil en versión móvil.....	203
Ilustración 95: Métodos del controlador User-Controller.....	213
Ilustración 96. Método obtener usuarios por fragmento de nombre de usuario del controlador usuario.....	214
Ilustración 97. Controlador lectura en documentación Swagger.....	215
Ilustración 98. Controlador documento en documentación Swagger.....	216
Ilustración 99. Método PATCH Controlador documentos.....	217
Ilustración 100. Método GET del controlador documento.....	218
Ilustración 101. Controlador Comentario en interfaz de Swagger.....	219
Ilustración 102. Controlador Comentario en interfaz de Swagger.....	220
Ilustración 103. Controlador Actividad en interfaz de Swagger.....	221
Ilustración 104. Vista de registro.....	222
Ilustración 105. Vista de inicio de sesión.....	223
Ilustración 106. Vista principal.....	224
Ilustración 107. Información de notificación.....	225
Ilustración 108. Vista de perfil.....	226
Ilustración 109. Seguidores de un usuario.....	226

Ilustración 110. Formulario edición de perfil.....	227
Ilustración 111. Resultados edición perfil.....	227
Ilustración 112: Vista explorar.....	228
Ilustración 113. Ejemplo de búsqueda con barra.....	229
Ilustración 114: Búsqueda por géneros.....	230
Ilustración 115. Resultado búsqueda por géneros.....	231
Ilustración 116. Información básica de documentos.....	231
Ilustración 117: Información detallada de documentos.....	232
Ilustración 118. Vista lecturas.....	233
Ilustración 119: Vista escrituras.....	234
Ilustración 120. Primera parte formulario creación de documentos.....	235
Ilustración 121. Segunda parte formulario creación de documentos.....	235
Ilustración 122. Campo portada en formulario de creación de documentos.....	235
Ilustración 123. Segunda parte formulario creación de documentos.....	235
Ilustración 124: Vista editor de texto.....	236
Ilustración 125. Ejemplo generación de ilustración.....	237
Ilustración 126. Resultado generación de ilustración.....	237
Ilustración 127. Acceso a documentos creados.....	238
Ilustración 128. Confirmación eliminar documento.....	239
Ilustración 129. Confirmación publicar documento.....	239
Ilustración 130. Vista lectura.....	240
Ilustración 131: Diccionario.....	240
Ilustración 132. Ventana comentarios.....	241
Ilustración 133. Campo creación de reseñas.....	242
Ilustración 134. Respuestas a una reseña.....	243
Ilustración 135. Contenido del directorio principal de la entrega.....	244
Ilustración 136. Contenido del directorio Proyectos.....	244
Ilustración 137. Contenido del directorio Ejecutables.....	245
Ilustración 138. Contenido del directorio Diagramas.....	245
Ilustración 139. Orden ejecución del servidor.....	246
Ilustración 140. Servidor desplegado completamente.....	246
Ilustración 141. Ejecución de la API de generación de imágenes.....	247
Ilustración 142. Ejemplo de API Generadora de imágenes desplegada.....	247
Ilustración 143. Ejemplo de API Generadora de imágenes desplegada.....	248
Ilustración 144. Sustitución URL generación de imágenes.....	248

Relación de Tablas

Tabla 1. Información sobre historias de usuario y puntos de esfuerzo.....	48
Tabla 2. Costes software.....	64
Tabla 3: Costes hardware.....	66
Tabla 4. Costes salarios.....	67
Tabla 5: Costes totales.....	67
Tabla 6. Costes servicios.....	69
Tabla 7. Resumen contenidos de los controladores.....	113

1. Introducción

Para comenzar el presente trabajo hablaremos en primer lugar del contexto de aplicación sobre el que se desarrollará, comentaremos qué nos ha motivado a la creación del sistema propuesto, cuáles son los objetivos que esperamos alcanzar, la metodología que utilizaremos para el desarrollo y la estructura de sus contenidos.

Nuestro proyecto consistirá en el desarrollo de una aplicación web, que estará diseñada para que, mediante el esquema de red social, los usuarios puedan escribir, leer y compartir documentos, con la finalidad de facilitar el acceso a documentos de calidad para la población.

1.1. Motivación

La decisión detrás del desarrollo de esta aplicación se respalda en la creciente necesidad de herramientas en línea que permitan a los usuarios crear, leer y compartir documentos, además de un interés personal en lo que respecta a la escritura y lectura de libros.

La motivación principal para el desarrollo de la aplicación es conseguir que las nuevas generaciones mantengan un hábito de lectura, ya que con otro tipo de medios de información, como son la televisión o los directos o vídeos, el medio de información escrito no ha sabido adaptarse a las nuevas generaciones y necesidades. Para ello se intentarán adaptar los documentos a modelos de comunicación más actuales.

Aunque existen otras plataformas de creación de documentos pero carecen de ese componente social, por lo que se busca permitir a los usuarios interactuar entre ellos. Esto ayudará a que los usuarios que creen un documento puedan observar las

opiniones de otros usuarios y conocer qué deben mejorar o mantener, lo que ayudará a que poco a poco los documentos crezcan en calidad. También ayudará a dar a conocer a los documentos, puesto que al tener interacción, los documentos de más calidad serán recomendados y podrán acceder a un público mayor.

Otro aspecto que motiva la creación de la aplicación es la necesidad de un sistema de creación de ilustraciones sencillo para los documentos, ya que para poder conseguir que los documentos de usuarios independientes estén al nivel de documentos creados profesionalmente, es imprescindible conseguir que las ilustraciones de los documentos tengan buena calidad independientemente del nivel en habilidades gráficas de los usuarios. Esto se ha intentado en otro tipo de aplicaciones que incluyen editores gráficos, pero que siguen requiriendo cierto nivel en diseño por parte de los usuarios. Sin embargo, gracias a los nuevos avances tecnológicos en ámbitos como la inteligencia artificial, ahora la creación de ilustraciones y portadas es mucho más sencilla y automática.

En general, la motivación es la de crear una aplicación para fomentar la lectura y la escritura a usuarios de a pie, entregándoles las mayores facilidades posibles, tanto como para la creación de documentos con sus portadas e ilustraciones, como para la lectura y la publicación de documentos a grandes masas.

1.2. Objetivos

A continuación, enumeramos los distintos objetivos que deberemos cumplir al finalizar el proyecto:

- Realizar un estudio de necesidades y soluciones existentes.
- Diseñar un sistema informático especialmente orientado a la utilización de arquitecturas, principios de diseño y metodologías de desarrollo web.

- Seleccionar un entorno de desarrollo basado en tecnologías web e implementar un prototipo de aplicación web que satisfaga las necesidades que se hayan determinado.

1.3. Metodología

En este apartado hablaremos de la metodología que se utilizará para desarrollar el proyecto. La metodología que he escogido es una selección de técnicas derivadas de Scrum, adaptada a las necesidades del proyecto y a las limitaciones de contar con solo un miembro como equipo de trabajo. Scrum [1] es una metodología ágil que suele utilizarse para gestionar equipos de trabajo de forma adaptable y transparente, pero tiene cualidades que pueden ser extraídas y adaptadas a un entorno de trabajo individual.

Esta metodología ha sido elegida debido principalmente a su sistema de iteraciones, las cuales permiten la creación de versiones funcionales del producto y la posibilidad de adaptar dicho producto a cambios. Esta flexibilidad y organización que otorga Scrum puede ser muy útil incluso en trabajos individuales.

Al extrapolar Scrum al trabajo individual, los roles dejan de tener sentido, puesto que solo existe un único miembro. Sin embargo, las funciones que realizan estos roles deben estar presentes, y seré yo mismo quien se encargue de todas ellas. Me aseguraré de seguir los principios y valores de Scrum, función que realizaría el Scrum Master, además de definir y asignar prioridades a las historias de usuario del *Product Backlog*, tarea que realizaría el *Product Owner*.

Otra función que hay que adaptar serían los eventos. La Daily Scrum meeting es una reunión diaria que realizan los miembros de un equipo que aplica Scrum. En mi caso este evento se traducirá en una revisión diaria de mi trabajo, revisando qué he

realizado durante el día y qué obstáculos he tenido, pudiendo consultarlos con el tutor para saber qué medidas tomar. Además se preparará el trabajo para el día siguiente.

La función principal de la metodología ágil y que se mantiene prácticamente intacta son las iteraciones. Se le asignará a cada iteración un conjunto de funcionalidades y tareas a desarrollar además de unos límites temporales, al final de los cuales se tendrá un producto funcional con las funcionalidades asignadas a la iteración. Además al final de cada iteración/Sprint realizaré una revisión del mismo para evaluar el trabajo realizado y si hacen falta cambios en los sprints siguientes. Para dicha evaluación del trabajo usaré criterios de aceptación, asignados a sus respectivas historias de usuario. Estos criterios son condiciones que deben cumplirse para que una historia de usuario se considere completa, además de servir para que todas las partes del desarrollo (clientes, desarrolladores, usuarios, etc) sepan qué esperar de cada historia de usuario.

En resumen, como metodología de trabajo utilizaré una metodología inspirada en Scrum, no Scrum en su sentido estricto puesto que existe la necesidad de adaptarlo a un equipo de una sola persona. Aún así, muchas de sus prácticas pueden ser de gran utilidad para las necesidades de este proyecto.

1.4. Estructura del documento

El presente documento se ha organizado de la siguiente manera: En el punto 2, realizaremos un análisis preliminar, donde se estudiarán aplicaciones similares y las funcionalidades y requerimientos que se desean cumplir. Además realizaremos un estudio sobre las tecnologías que se utilizarán durante el desarrollo para el back-end y front-end y se presentará una propuesta de solución. Se detallarán también la planificación a seguir, las historias de usuario a desarrollar, realizaremos un estudio de

viabilidad y comentaremos el modelo de dominio, mostrando las entidades y relaciones entre ellas.

El punto 3 contiene los detalles del diseño de la aplicación, incluyendo distintos diagramas, entre ellos diagramas entidad-relación, diagramas de clases y diagramas de secuencia. También se incluyen detalles del diseño de la interfaz, tanto del cliente como de la API REST y una planificación de las pruebas a realizar.

En el apartado número 4 hablaremos sobre la arquitectura del sistema, proporcionando detalles sobre la implementación del cliente y el servidor. Se detallarán aspectos como la estructura de ambos proyectos, las clases existentes y las responsabilidades asignadas a dichas clases, además de algunos patrones de diseño aplicados en dicha implementación.

En la sección 5 nos centraremos en las pruebas, en concreto en cómo se ha llevado a cabo la planificación creada en el apartado número 3, comentando el número de pruebas realizadas y los resultados obtenidos, destacando algunos problemas detectados y comentando cómo fueron solucionados.

En el apartado número 6 realizaremos una síntesis de lo mostrado en el documento, comentando el nivel de cumplimiento de los objetivos que se plantearon inicialmente, incluyendo también una opinión personal sobre los conocimientos adquiridos e impresiones durante el desarrollo del proyecto. Por último añadiremos el estado del proyecto, adjuntando algunas mejoras posibles para el futuro.

Por último se mostrarán distintos apéndices con información adicional, principalmente un manual de usuario para mostrar las funcionalidades de la aplicación y cómo utilizarlas, un manual de instalación para poder instalar la aplicación en nuestro sistema y una descripción de los contenidos suministrados.

2. Análisis

2.1. Análisis preliminar

En el análisis preliminar hablaremos sobre el sistema del que partimos, la problemática que se nos presenta, soluciones similares y necesidades sin cubrir.

El sistema a desarrollar se propone ser una forma sencilla de creación, lectura, edición y divulgación de documentos. Su objetivo principal es fomentar tanto la escritura como la lectura mediante la inclusión de herramientas que permitan a las personas de a pie conseguir documentos de calidad.

Visto el objetivo del sistema, hablaremos ahora de por qué es necesario el sistema a desarrollar. Cada vez más la lectura por parte de las nuevas generaciones se va reduciendo [2], lo que hace que tanto la comprensión lectora como el vocabulario de la población se reduzca. Esto se debe, principalmente, a que actualmente los sistemas que más usuarios contienen son las redes sociales, y las redes sociales centradas en lecturas se han quedado atrás respecto al resto de sistemas similares como veremos a continuación.

Como hemos hablado, esta aplicación aspira a facilitar tareas que para la mayoría de usuarios son complejas, principalmente la ilustración de los mismos. Muchas veces para crear un documento de calidad, como por ejemplo una novela o un cuento para niños, un elemento clave es la inclusión de imágenes que acompañan a los textos. Sin embargo, no todo el mundo tiene la destreza para realizar esas ilustraciones por sí mismos o la posibilidad de comprar licencias de imágenes ya creadas.

2.1.1. Aplicaciones similares

Cuando hablamos de sistemas similares al que desarrollaremos, debemos hablar sobre qué criterio seguir para identificar aplicaciones de este estilo. Principalmente una aplicación similar será una aplicación que permite la escritura, lectura y el poder compartir documentos, a continuación analizamos algunas de ellas, destacando sus aspectos positivos que podríamos adoptar y los aspectos negativos que debemos subsanar.

2.1.1. Wattpad

Wattpad es quizás la aplicación más similar en cuanto a concepto a nuestra aplicación a desarrollar. Es una red social donde usuarios escriben documentos para que otros usuarios puedan leer esos mismos documentos. Elementos a tener en cuenta de esta aplicación son la capacidad de poder escribir y leer documentos de usuarios, las funcionalidades típicas de una red social, además de la interacción entre usuarios con comentarios.

Pero entonces, ¿Por qué es necesaria otra aplicación de este estilo? Hablaremos a continuación de los aspectos deficientes de WattPad y que debemos evitar

El primero de los aspectos que quiero destacar para subsanar en nuestra aplicación es el editor de texto de la aplicación:

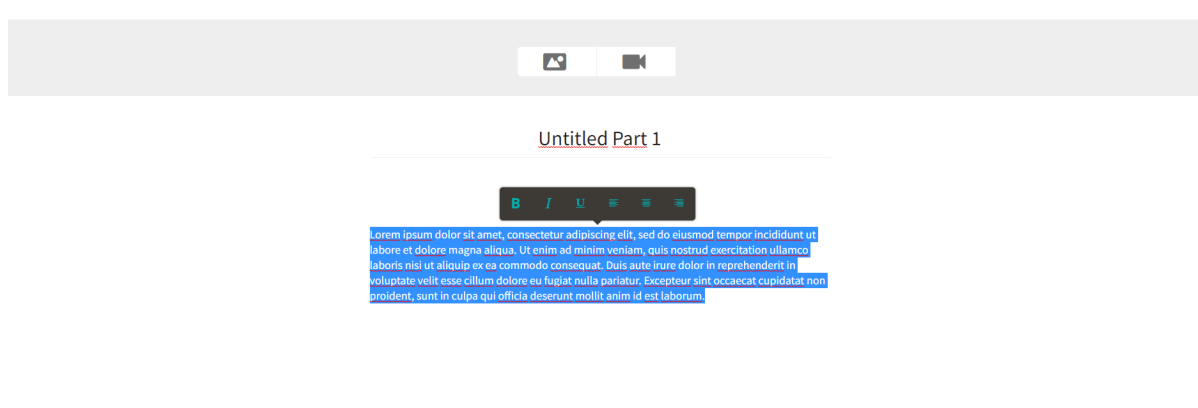


Ilustración 1. Ejemplo de funcionalidades del editor Wattpad.

Como se observa en la imagen superior, el editor de texto tiene únicamente las funcionalidades básicas, inserción de imágenes y video, subrayado, cursiva y negrita, y por último las alineaciones de texto. Vemos que no añade por ejemplo cambios al tamaño del texto, diferentes títulos y encabezados, etc. Esto se debe a que principalmente Wattpad está pensado para historias, novelas o relatos, que no necesitan de elementos muy complejos en el texto. Nuestra aplicación está pensada también para otros tipos de documentos, como por ejemplo documentos sobre ciencia o tecnología, que requieran tal vez de más herramientas como por ejemplo la capacidad para insertar fragmentos de código o tablas.

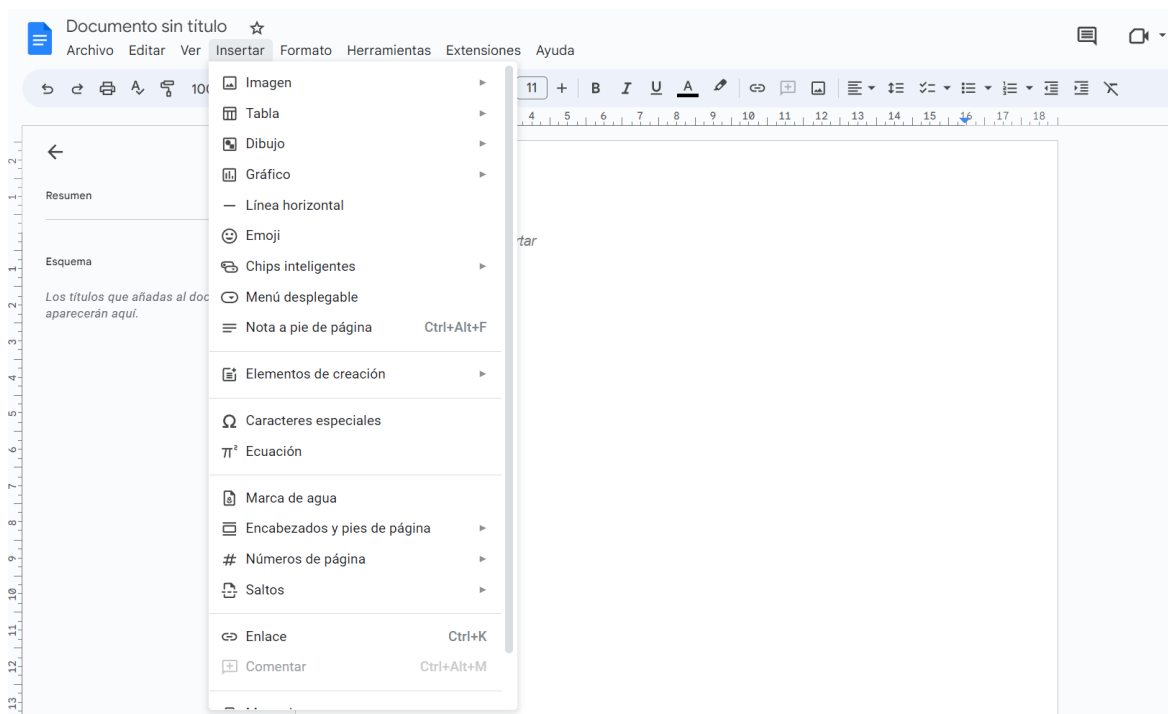
Además, como hablamos anteriormente, por ejemplo la generación de portadas se debe realizar con Canva¹, la cual es una plataforma de diseño gráfico que permite a los usuarios crear ilustraciones y contenido visual y que está pensada para el público estándar, pero que sigue requiriendo de cierta habilidad para la creación de imágenes, por lo que nuestra aplicación se centrará en facilitar aún más la inclusión de ilustraciones.

¹ https://www.canva.com/es_es/

2.1.2. Google docs

Cuando hablamos sobre Google docs², hablamos del caso contrario a la aplicación anterior. Google docs es un editor de texto implementado por google, que permite la creación de documentos y su posterior lectura por otros usuarios mediante el enlace del mismo.

Los puntos positivos de google docs son el alto número de herramientas (Ilustración 2) que contiene para la edición de documentos, además de la posibilidad de poder colaborar en la edición de los mismos. Aunque esta funcionalidad es bastante útil, nuestra aplicación no la implementará debido a las limitaciones de tiempo y medios. Sin embargo si se intentará la inclusión del mayor número de herramientas posibles para la creación y edición de los textos.



² <https://docs.google.com/>

Ilustración 2. Funcionalidades del editor de texto Google docs.

Uno de los puntos negativos de Google docs para la problemática central es su deficiente sistema de interacción entre lectores y documentos, ya que no incluye ningún sistema de búsqueda de documentos y si un usuario desea compartir su documento debe hacerlo con el enlace en aplicaciones externas, al igual que si desea valorar algún documento de otro usuario.

Además, volviendo a la funcionalidad principal del sistema a desarrollar, Google docs tampoco incluye un sistema para la creación sencilla de ilustraciones o portadas, lo que remarca de nuevo esa necesidad existente en el nicho.

2.1.2. Funcionalidades y necesidades

Como hemos visto en las aplicaciones similares, el hecho de no tener un sistema sencillo de creación de imágenes es una problemática que se repite en los dos principales sistemas del nicho en cuestión, por lo tanto nuestra aplicación deberá incluirlo. Además deberá incluir las funcionalidades positivas de las aplicaciones, como el gran número de funcionalidades de Google docs o el sistema de red social de Wattpad.

La aplicación además deberá funcionar correctamente tanto en ordenadores como en dispositivos móviles. Esto se debe a que cuando pensamos en escribir, los dispositivos más cómodos y utilizados son los ordenadores, mientras que cuando pensamos en leer los dispositivos por excelencia para ello son los dispositivos móviles, así que la aplicación debe ser funcional en ambos, siendo por defecto para ordenadores, pero siendo funcional en móvil, sobre todo en las funcionalidades de red social y de lectura. Se deberá poder comentar y puntuar los documentos, leerlos, manteniendo la aplicación misma la página de lectura, crear y editar los documentos,

poder seguir a otros usuarios, ver su actividad y sus libros, generar imágenes para la portada y para ilustrar el propio libro, entre otras.

2.1.3. Estudio de tecnologías

Para el desarrollo de nuestra red social basada en documentos, nos hemos decantado por una aplicación web sobre aplicaciones nativas o CMS debido a que las aplicaciones web ofrecen mayor accesibilidad, lo que permitirá a los usuarios poder leer documentos donde quieran, no requieren instalación y son independientes del sistema operativo, por lo que una sola aplicación nos servirá para todos los dispositivos, manteniendo una sola base de datos para todas las instancias de la aplicación web, teniendo los datos centralizados.

Estos beneficios superan a las desventajas, las cuales son la necesidad de conexión a internet, por lo que la aplicación requerirá estar conectada para funcionar, un peor rendimiento que aplicaciones nativas, ya que estas se adaptan al sistema operativo en el que se ejecutan, por lo que están optimizadas para dichos sistemas, menor seguridad, ya que estas aplicaciones pueden sufrir ataques como inyecciones SQL. Aunque las desventajas son numerosas, también son subsanables, la necesidad de conexión no es un problema puesto que la mayoría de usuarios disponen de datos móviles allá donde vayan, sobre el rendimiento, aunque sea menor que en aplicaciones nativas, sigue siendo un rendimiento más que válido y la seguridad se puede reforzar con diferentes librerías y sistemas.

En concreto, nuestra aplicación web está basada en cliente, lo que implica que gran parte de la lógica y el procesamiento se encuentre en el front-end. Para la implementación utilizaremos una arquitectura de tipo SPA o aplicación de una sola página, la se basa en hacer que la aplicación web reescriba dinámicamente la página actual en lugar de cargar páginas web completas. Nuestro cliente interacciona con un

back-end basado en spring boot que se encargará de proporcionar la información que el cliente requiera.

2.1.3.1. Tecnología front-end: React

Para el desarrollo de la parte Front de la aplicación hemos decidido usar React [3], la cual es una librería de JavaScript desarrollada por Facebook y que se utiliza para crear interfaces gráficas interactivas. Otras opciones fueron Angular, la cual es un framework completo, creado por Google y Vue JS, un framework para el desarrollo de aplicaciones web en JS y que hemos utilizado durante la carrera, el cual destaca por su simplicidad.

Como hemos comentado anteriormente, React es la librería que hemos decidido utilizar frente a las mencionadas, eso se debe a las siguientes características [4]:

1. Desarrollo orientado a componentes

Como muchas otras librerías y frameworks, React ofrece un sistema de componentes, el cual se podría comparar con bloques de construcción, cada componente representa una parte de la interfaz, como una barra de navegación, un botón, una modal, una tarjeta, etc. Cada componente es reutilizable, es decir, se puede utilizar en distintas partes de la aplicación, lo cual ayuda a simplificar el código al no tener que repetirlo constantemente.

Los componentes de React se caracterizan por mantener su propio estado, es decir, que si un dato cambia durante la ejecución de la aplicación, el componente puede actualizar su estado para mostrar los cambios en pantalla.

Tanto React como Angular son herramientas basadas en componentes, sin embargo React es más sencilla en este aspecto, dando posibilidad al uso de enfoques más

funcionales y el uso de JSX para combinar JavaScript y HTML. Por otro lado Angular está más orientado a objetos y es más rígido debido a su orientación a objetos y estructura definida.

2. Virtual DOM

React dispone de un Document Object Model virtual, lo cual le permite a React el evitar tener que actualizar el DOM real cada vez que cambia el estado de un componente. Esto se utiliza más tarde en el proceso de reconciliación, el cual consiste en comparar ambos DOMs (Real y Virtual) y encontrar la manera más óptima de pasar del DOM virtual al real. Esto hace que React sea una buena herramienta para obtener aplicaciones web optimizadas. Esta funcionalidad no está presente en Angular, lo que repercute en el rendimiento en aplicaciones de gran tamaño.

3. Comunidad

React se ha vuelto una librería muy popular desde su lanzamiento en 2013, esto ha conseguido que exista un gran número de documentación, tutoriales y guías que ayuden con el desarrollo de aplicaciones con esta herramienta. A esto se le suma una gran cantidad de módulos y librerías que se pueden utilizar en ella para facilitar la creación de muchas de las funcionalidades necesarias. Este ha sido uno de los puntos decisivos para escoger esta tecnología, ya que nos asegura que tendrá soporte durante mucho tiempo debido a su gran popularidad y nos facilitará mucho el desarrollo debido a lo comentado anteriormente.

Tanto Angular como React tienen una amplia comunidad que los respalda, sin embargo la comunidad de React es más amplia, por lo que en este punto es superior a la hora de conseguir documentación, proyectos y ejemplos.

4. Flexibilidad

A diferencia de Angular, React no es un framework completo, lo que se traduce en que solo se encarga de gestionar la capa de la vista, dejando por ejemplo la gestión de los estados o la enrutación a la elección del usuario, proporcionando soluciones complejas para aplicaciones que lo necesiten, como por ejemplo Redux, o permitiendo gestionar los estados a nivel de componentes para aplicaciones que así lo requieran.

Esta flexibilidad es una ventaja, puesto que permite la optimización de recursos y la personalización de funcionalidades pero también es una desventaja puesto que requiere más trabajo y tiempo.

5. Lenguaje de programación

React permite distintos lenguajes de programación, los cuales son bastante conocidos y utilizados, con mucha documentación y códigos de ejemplo. Principalmente la librería utiliza JavaScript, lenguaje de tipado dinámico por todos conocido y que incluye elementos como las funciones de flecha y las promesas. También permite JSX, que es una sintaxis de JavaScript que se asemeja al HTML, haciéndolo mucho más legible para aplicaciones web. Además mediante JavaScript y JSX se permiten los componentes funcionales o functional components, los cuales son una forma de creación de componentes en funciones de JavaScript que facilitan la implementación y legibilidad, además de tener mejor rendimiento en algunas ocasiones. Junto a todo esto, JavaScript también permite la gestión de eventos, lo que ayuda muchísimo a gestionar las interacciones con el usuario en la interfaz.

Angular por su parte utiliza TypeScript, un subconjunto de JavaScript que añade tipos estáticos y que también puede ser usado en React si se requiere, por lo que en React tenemos más diversidad de lenguajes a utilizar según las necesidades.

6. Conclusión

En conclusión en lenguaje elegido ha sido React, en contraposición a Angular debido a los siguientes motivos:

- Preferencia a la flexibilidad de React a la hora de poder escoger bibliotecas frente a la solución integral de Angular.
- Mejor eficiencia a la hora de manejar el DOM por parte de React debido a la inclusión del Virtual DOM, funcionalidad que Angular no tiene.
- Enfoque más funcional a la hora de realizar componentes por parte de React, mientras que Angular es más rígido.
- React tiene una comunidad más amplia con mayor cantidad de contenido y soluciones a problemas comunes.
- Preferencia del lenguaje JSX y JavaScript frente a TypeScript y HTML de Angular.

2.1.3.2. Tecnología back-end: Spring boot

Para el back-end hemos barajado distintas opciones, entre ellas Spring boot [5], Express y Django. El motivo principal para esta elección es el alto conocimiento en Spring boot debido a que fue impartido en clases a un alto nivel, por lo que aprovecharemos esos conocimientos para desarrollar la api Rest de esta aplicación. Además de ese punto, hay algunos otros que nos hicieron inclinarnos hacia esta tecnología:

1. Compatibilidad

Spring boot es compatible con muchas tecnologías, como Hibernate o Maven. Además su “auto-configuración”, que configura la aplicación basándose en las dependencias del proyecto facilita mucho la inserción de dependencias y módulos. Sumado a esto, Spring boot también se caracteriza por tener un servidor embebido (Normalmente Tomcat), lo que permite desarrollar ejecutar y probar aplicaciones sin la necesidad de instalar un servidor, ya que al ejecutarlo Spring boot iniciará con su propio servidor listo para recibir y responder peticiones HTTP.

Además, comparando Spring boot, la opción elegida y Django, observamos que al utilizar Java, Spring boot es compatible con cualquier plataforma con posibilidad de ejecutar Java, como gran variedad de sistemas operativos y servidores, mientras que Django usa python, que sigue siendo muy compatible con muchos sistemas pero no al mismo nivel que Java.

2. Seguridad

Como vimos anteriormente uno de los puntos débiles de las aplicaciones web es la seguridad. Spring boot soluciona este problema mediante Spring boot Security, el cual permite distintas tecnologías para garantizar una aplicación web segura, como por ejemplo JWT o OAuth2. Como hemos hablado en el punto anterior, esta dependencia se añade simplemente incluyendo en el archivo Pom.xml, nos otorga un sistema de identificación y de autorización con mucha documentación y diferentes métodos.

Respecto a la comparación entre los candidatos Django y Spring boot, ambos incorporan herramientas de seguridad bastante completas, por lo que este no es un campo decisivo a la hora de elegir la tecnología back-end a utilizar.

3. Comunidad

Al igual que en el caso de React, Spring boot tiene una amplia comunidad, lo cual asegura soporte a largo plazo, además de infinidad de tutoriales y material con el que aprender y guiarse. Además debido a sus muchos años de uso, la mayoría de problemas e impedimentos que encontremos en el desarrollo ya tendrá por parte de otros usuarios en foros y proyectos.

Comparando de nuevo las opciones que barajamos, vemos que tanto Spring boot como Django contienen una amplia comunidad, sin embargo debido al largo tiempo que Java lleva siendo utilizado, es bastante probable que el material y los recursos de este lenguaje, y por lo tanto de Spring boot, sea superior.

4. Lenguaje de programación

Como hemos comentado Spring boot utiliza Java como lenguaje de programación. Es un lenguaje de programación que ya conocemos y con el que tenemos un gran manejo y destaca por versatilidad, robustez y su ecosistema de bibliotecas. Hemos elegido programar en Java sobre otros lenguajes de programación como Python (Django) debido principalmente a la experiencia que tenemos con él y la amplia documentación y proyectos.

5. Conclusión

Resumiendo los puntos anteriores, la elección final de Spring boot sobre Django, se ha basado en los siguientes puntos:

- Preferencia del lenguaje Java debido a la experiencia que tengo con él, además de tipado fuerte, que me resulta más cómodo para aplicaciones back-end.

- Grandes características de seguridad mediante librerías como Spring Security.
- Amplia compatibilidad con muchos sistemas y servidores actuales.
- Gran comunidad para el apoyo en caso de problemas o necesidad de recursos externos.

2.1.3.2. API generadora de imágenes: Stable Diffusion

Como hemos comentado en la problemática, nuestra aplicación solventará el problema de dificultad de la creación de ilustraciones, para ello utilizaremos una inteligencia artificial del tipo Text2Img (Texto a imágenes), la cual mediante una entrada de texto generará una imagen que devolverá al front de la aplicación tras la posterior petición de la misma. Entre las distintas opciones están *Stable Diffusion*³, *MidJourney*⁴ y *DALLE-2*⁵. Finalmente nos decidimos por Stable Diffusion por los siguientes motivos [6]:

1. Open-Source

Stable Diffusion es open source, lo que significa que es posible la utilización de su código sin necesidad de pago, lo cual lo hace perfecto para el desarrollo del prototipo que nos concierne, puesto que, aunque otras opciones generan imágenes de mayor calidad, haremos que sea posible la sustitución de la API por otra en producción, mientras que para el prototipo y demostración *Stable Diffusion* hace un excelente trabajo.

2. Comunidad

³ <https://stablediffusionweb.com/>

⁴ <https://www.midjourney.com/>

⁵ <https://openai.com/product/dall-e-2>

Al ser Open Source, hay una gran comunidad de usuarios que mejoran el modelo continuamente, crean proyectos que se pueden probar e incluso APIs REST que podremos desplegar y usar en nuestro proyecto, ahorrándonos así el trabajo de tener que crear una API nosotros mismos.

2.2. Propuesta de solución

Tras el análisis, haremos aquí una propuesta de solución que resuma los apartados anteriores.

Para solucionar las necesidades y problemáticas existentes en el nicho de nuestra aplicación, nuestro sistema permitirá a un usuario registrarse, crear, editar y leer libros, acceder fácilmente a los libros que están leyendo, manteniendo la página por la que el usuario se encontraba, además de poder guardar documentos para una lectura posterior, además de comentar en ellos dejando una valoración. Sumado a esto el usuario podrá seguir a otros usuarios para ver su actividad. También permitirá generar imágenes para la ilustración de los mismos.

La aplicación intenta fomentar la lectura añadiendo tecnologías actuales, como el esquema de red social, y tecnologías novedosas como la generación de imágenes con IA, lo que ayudará a publicitar los documentos y generar documentos de calidad respectivamente.

Para desarrollar la aplicación utilizaremos React para la creación del proyecto front-end, creando nuestros propios componentes responsive sin el uso de librerías de componentes externas y Quill JS como editor textos embebidos.

Para el desarrollo de la parte back-end utilizaremos Spring boot con H2 como gestor de la base de datos y tomcat como servidor embebido, ambos incluyéndose en sus dependencias y de fácil instalación.

2.4. Historias de Usuario

Must have

1. Creación de una cuenta

Como usuario quiero poder crear una cuenta en la aplicación, para tener acceso a un perfil propio y control sobre las funcionalidades de la aplicación.

Criterios de aceptación:

- El usuario tiene acceso a un formulario de registro.
- El nombre de usuario que se coloca en el formulario debe ser único.
- El correo electrónico que se coloca en el formulario debe ser válido y único.
- Se puede agregar una imagen en el registro de manera opcional. Si no se agrega una imagen se coloca una por defecto.
- La contraseña debe tener como mínimo 8 caracteres y cumplir los requisitos de complejidad.
- Si alguno de los campos no cumple con las condiciones se le muestra al usuario un mensaje indicando el error y no se le permite completar el registro.

Descomposición en tareas (17h):

1. Diseñar el componente de formulario de registro. (2h)
2. Implementar el componente de formulario de registro. (2h)
3. Implementar validaciones de campos del formulario. (3h)
4. Crear la entidad usuario en la base de datos. (3h)

5. Implementar lógica de registro en el servidor, incluyendo validaciones correspondientes (5h)
6. Crear los DTOs necesarios para el envío de información. (2h)

2. Acceso a la cuenta

Como usuario quiero poder acceder a mi cuenta con las credenciales que indiqué en la creación de la cuenta.

Criterios de aceptación:

- El usuario tiene acceso a un formulario de inicio de sesión
- El usuario puede ingresar sus credenciales, nombre de usuario y contraseña en el formulario
- Si las credenciales no son correctas se le notifica al usuario en un mensaje.
- Si algún campo está vacío se resalta el campo en concreto y se notifica en un mensaje.

Descomposición en tareas (14h):

1. Diseñar el componente de formulario de inicio de sesión. (2h)
2. Implementar el componente de formulario de inicio de sesión. (2h)
3. Implementar la validación de credenciales en el cliente. (3h)
4. Implementar lógica de inicio de sesión en el servidor. (5h)
5. Crear DTOs necesarios para el inicio de sesión. (2h)

3. Modificación de cuenta

Como usuario quiero poder modificar tanto las credenciales de acceso como la información de mi perfil.

Criterios de aceptación:

- El usuario tiene acceso a un formulario de cambio de información y credenciales
- El nuevo usuario debe ser único.
- La nueva contraseña debe cumplir los requisitos de complejidad.
- El usuario puede subir una imagen para cambiar su imagen de perfil.
- El usuario puede agregar un texto en forma de descripción para su perfil.

Descomposición en tareas (16h):

1. Diseñar el componente formulario de modificación de cuenta. (2h)
2. Implementar el componente formulario de modificación de cuenta. (2h)
3. Implementar validaciones en los campos del cliente. (3h)
4. Implementar sistema de modificación de usuarios en el servidor. (7h)
5. Implementar validaciones de los datos de modificación en el servidor. (2h)

4. Creación de documentos

Como usuario quiero poder crear documentos de texto, pudiendo elegir su título, contenido, portada, etc.

Criterios de aceptación:

- El usuario tiene acceso a un formulario para la creación del documento
- El usuario puede introducir un título para el documento. En caso de no introducirlo se le colocará un título por defecto siguiendo el esquema “documento X”, siendo X el número de documentos con títulos por defecto que ya tiene el usuario.

- El usuario puede colocar una imagen de portada. En caso de no colocarla se le colocará una por defecto al documento.
- El usuario puede agregar una sinopsis al documento, en caso de no colocarla la sinopsis quedará vacía.
- El usuario puede seleccionar el género del libro entre una lista de géneros predeterminados. En caso de no agregar ningún género el documento quedará sin género.
- El usuario puede añadir etiquetas al documento sin ningún tipo de restricción.
- Una vez creado el documento este se encuentra en visibilidad “privada”.

Descomposición en tareas (21h):

1. Diseño del componente formulario para la creación de documentos. (3h)
2. Implementación del componente formulario para la creación de documentos. (3h)
3. Implementar validaciones necesarias al formulario. (3h)
4. Crear la entidad documento en el back-end. (5h)
5. Crear las funcionalidades necesarias para la creación de documentos en el servidor (5h)
6. Crear los DTOs necesarios para el envío y recibo de documentos. (2h)

5. Contenido de documentos

Como usuario quiero que el contenido de los documentos a crear pueda contener textos, imágenes, videos.

Criterios de aceptación:

- El usuario tiene acceso a una barra de herramientas con posibilidad de insertar texto, imagen y video.

- El usuario puede escribir texto en el documento y este queda guardado.
- El usuario puede insertar una imagen que se encuentre guardada en su dispositivo.
- El usuario puede insertar la Url de un video para que este quede embebido en el texto.
- Todos los tipos de multimedia se muestran correctamente en el documento.

Descomposición en tareas (21h):

1. Diseñar la vista donde se colocará el editor. (3h)
2. Implementar la vista donde se colocará el editor. (4h)
3. Añadir funcionalidades de edición y contenido multimedia. (7h)
4. Implementar el envío y recuperación del contenido de los documentos. (7h)

6. Herramientas de edición

Como usuario quiero poder aplicar los formatos básicos al contenido de los documentos.

Criterios de aceptación:

- El usuario tiene acceso a una barra de herramientas en el documento.
- El usuario puede cambiar el color al texto.
- El usuario puede subrayar, poner en cursiva y en negrita una parte del texto.
- El usuario puede cambiar el tamaño del texto.
- El usuario puede aplicar estilos al texto.
- El usuario puede cambiar la fuente del texto.
- El usuario puede alinear el texto.
- El usuario puede redimensionar las imágenes, manteniendo el “aspect ratio”.

Descomposición en tareas (23h):

1. Investigar distintas opciones para el editor de texto e implementar funcionalidades necesarias a la barra de herramientas. (7h)
2. Ajustar las funcionalidades de edición de textos. (8h)
3. Implementar lógica de guardado de los documentos en el servidor. (8h)

7. Visibilidad de documento

Como usuario quiero poder decidir cuándo un documento pasa de ser privado (solo visible por el usuario creador) a público (visible para el resto de usuarios).

Criterios de aceptación:

- El usuario puede cambiar el estado del documento de privado a público.
- Al cambiar el estado el documento pasa a ser visible para otros usuarios.
- Al cambiar el estado el usuario debe rellenar en un formulario los detalles del documento que falten, entre ellos el título, la portada, la sinopsis y los géneros. En caso de que falte un campo por rellenar, no se le permitirá publicar el documento.
- El usuario puede poner en privado un documento público de su autoría y este ya no se muestra al resto de usuarios.

Descomposición en tareas (14h):

1. Implementar funcionalidad para cambiar la visibilidad de un documento en el cliente. (7h)
2. Implementar el cambio de visibilidad en el servidor (7h).

8. Lectura de documentos

Como usuario identificado quiero poder leer un documento.

Criterios de aceptación:

- El usuario puede acceder a los documentos públicos de otros usuarios.
- El usuario no tiene acceso a la edición de estos documentos.
- El usuario ve el documento en el formato en el que el usuario creador lo publicó.

Descomposición en tareas (8h):

1. Diseñar componentes para poder acceder a documentos públicos de otros usuarios. (2h)
2. Implementar componentes para poder acceder a documentos públicos de otros usuarios. (2h)
3. Implementar lógica en el servidor para el acceso a documentos públicos (3h)

9. Edición de documentos

Como usuario quiero poder editar los documentos ya creados.

Criterios de aceptación:

- El usuario tiene acceso únicamente a la edición de sus documentos.
- El usuario puede eliminar y agregar texto a un documento ya creado y guardado.
- El usuario sólo puede editar textos privados, en caso de querer editar un documento público debe cambiar su visibilidad.
- Los cambios en el documento quedan guardados correctamente.

Descomposición en tareas (13h):

1. Diseñar la interfaz para acceder a la edición de documentos. (3h)
2. Implementar la interfaz para acceder a la edición de documentos. (5h)
3. Implementar la lógica en el servidor para la edición de documentos. (5h)

10. Borrado de documentos

Como usuario quiero poder eliminar un documento.

Criterios de aceptación:

- El usuario tiene acceso al borrado de los documentos de su autoría.
- Se le pregunta al usuario si está seguro de que quiere borrar el documento.
- Si el usuario acepta el documento queda borrado de la aplicación, y el creador ya no tiene acceso a él.
- Si el documento es público, el documento ya no aparecerá para el resto de usuarios.

Descomposición en tareas (9h):

7. Diseñar la interfaz para la eliminación de documentos propios. (2h)
8. Implementar la interfaz para la eliminación de documentos propios. (3h)
9. Implementar en el servidor la posibilidad de eliminar un documento. (4h)

Should have

11. Generación de ilustraciones

Como usuario quiero poder crear imágenes mediante entradas de texto, para insertarlas en el contenido y portada del documento.

Criterios de aceptación:

- El usuario tiene acceso a un formulario de creación de imágenes.
- El usuario puede introducir una descripción de la imagen que quiere generar, en caso de que no se introduzca descripción el formulario mostrará un error.
- El usuario puede introducir un ancho y un largo para la imagen, ambos entre 512 y 1024 píxeles. Si no se introduce tamaño la imagen por defecto será de 512x512 píxeles.
- Mientras se genera la imagen se muestra en el editor de texto una imagen del mismo tamaño que la imagen a generar, y que muestra un símbolo de carga.
- Cuando la imagen se ha generado la imagen de carga se sustituye por la imagen final.
- Si se le realizan cambios de tamaño o de formato a la imagen de carga, la imagen final debe mantener esos cambios.
- El usuario puede volver a generar otra imagen que sustituya a la imagen generada.
- El usuario tiene acceso a un campo de generación de imagen en la creación y publicación de documentos.
- El usuario puede introducir una descripción de la imagen a generar
- La imagen generada es del tamaño de la portada y cumple con las especificaciones del usuario.

Descomposición en tareas (15h):

1. Diseñar la interfaz con la que hacer peticiones de imágenes a partir de texto.
(2h)
2. Desarrollar la interfaz con la que hacer peticiones de imágenes a partir de texto.
(4h)
3. Buscar APIs generadoras de imágenes. (3h)

4. Adaptar la API a nuestro sistema y desplegarlas, haciendo su acceso público mediante URL. (6h)

12. Seguidores de un usuario

Como usuario quiero poder seguir a otros usuarios para estar al tanto de su actividad en la plataforma.

Criterios de aceptación:

- El usuario tiene acceso a un botón de seguir en el perfil del usuario a seguir.
- Al pulsar el botón el botón indica que se ha seguido al usuario.
- En la ventana principal se le notifica al usuario las lecturas y escrituras nuevas del usuario seguido.

Descomposición en tareas (11h):

1. Diseño de la interfaz para seguir a otros usuarios. (3h)
2. Desarrollo de la interfaz para seguir a otros usuarios. (3h)
3. Implementar funcionalidades de seguimiento de usuarios en el back-end. (5h)

13. Dejar de seguir usuario

Como usuario quiero poder dejar de seguir a los usuarios a los que sigo.

Criterios de aceptación:

- El usuario tiene acceso a un botón para dejar de seguir en el perfil del usuario seguido.
- Al pulsar el botón el botón indica que se ha dejado de seguir.
- En la ventana principal ya no aparecen notificaciones del usuario no seguido.

Descomposición en tareas (16h):

1. Diseño de la interfaz para dejar de seguir usuarios. (3h)
2. Desarrollo de la interfaz para dejar de seguir usuarios. (5h)
3. Implementar funcionalidades necesarias para eliminar seguidor/seguimiento en el back-end. (8h)

14. Perfil de usuario

Como usuario quiero tener un perfil público para que otros usuarios puedan ver mis seguidores, seguidos y documentos creados

Criterios de aceptación:

- El usuario puede acceder a su perfil para ver sus datos.
- Cualquier usuario puede acceder al perfil de un usuario y ver los documentos que ha escrito, los usuarios que sigue y los que le siguen.

Descomposición en tareas (15h):

1. Diseñar el perfil de usuario con la información requerida en el front-end. (3h)
2. Implementar el perfil de usuario con la información requerida en el front-end. (5h)
3. Implementar los end-points necesarios para las peticiones al servidor que realizará la vista. (7h)

15. Búsqueda de documentos

Como usuario quiero poder buscar un documento en función de su género.

Criterios de aceptación:

- El usuario tiene acceso a una barra de búsqueda donde puede introducir el título de un documento para buscarlo.
- El usuario tiene acceso a una selección de géneros por defecto que puede seleccionar para buscar documentos de ese género.
- Al realizar la búsqueda aparecen documentos que son de los géneros seleccionados o contienen en su título palabras de la búsqueda.

Descomposición en tareas (20h):

1. Diseñar vista exploración para la búsqueda de documentos públicos (3h)
2. Implementar vista exploratoria para la búsqueda de documentos públicos. (5h)
3. Implementar la interfaz de búsqueda de documentos mediante género. (6h)
4. Implementar en el back-end las funcionalidades y end-points necesarios para el funcionamiento de la búsqueda de documentos filtrados. (6h)

16. Creación de reseñas

Como usuario quiero poder dejar una reseña en un documento público.

Criterios de aceptación:

- El usuario puede escribir una reseña en un documento público.
- El usuario debe colocar una puntuación entre 0 y 5.
- El usuario puede no colocar ningún texto en la reseña.
- Una vez enviada la reseña, el resto de usuarios pueden verla.
- La media de las reseñas es visible en el documento para el resto de usuarios.

Descomposición en tareas (22h):

1. Diseñar el componente Reseña. (2h)
2. Desarrollar el componente Reseña. (4h)

3. Diseñar la interfaz para la creación de las reseñas (3h)
4. Implementar la interfaz para la creación de las reseñas (3h)
5. Implementar la entidad reseña en el back-end y las funcionalidades necesarias para su utilización (8h)
6. Implementar DTOs de reseña para las comunicaciones necesarias. (2h)

17. Respuesta a reseñas

Como usuario quiero poder contestar a las reseñas de otros usuarios.

Criterios de aceptación:

- El usuario puede responder a una reseña.
- El usuario no puede colocar la puntuación de una reseña en una respuesta.
- Una vez la respuesta se ha creado, el resto de usuarios pueden ver el comentario como respuesta a una reseña en específico.

Descomposición en tareas (22h):

1. Diseñar el componente Respuesta. (2h)
2. Crear el componente Respuesta. (4h)
3. Diseñar la interfaz para la creación de respuestas vinculadas a una reseña. (2h)
4. Implementar la interfaz para la creación de respuestas vinculadas a una reseña. (3h)
5. Implementar la entidad respuesta y sus funcionalidades y validaciones. (8h)
6. Crear los DTOs de respuesta para comunicación entre end-points. (3h)

Could have

18. Creación de comentario de línea

Como usuario quiero poder comentar una línea de un texto en específico.

Criterios de aceptación:

- El usuario puede seleccionar una parte del texto y crear un comentario de línea.
- El comentario de línea aparece en el margen exterior del documento como un icono.
- Al pulsar el icono del comentario el comentario se despliega dejando ver el comentario y resaltando en el documento el texto del que se hace el comentario.

Descomposición en tareas (11h):

1. Diseñar el componente comentario de línea. (3h)
2. Implementar el componente comentario de línea. (2h)
3. Diseñar la interfaz para vincular un fragmento de texto a un comentario. (2h)
4. Implementar la interfaz para vincular un fragmento de texto a un comentario. (6h)
5. Implementar la entidad comentario de línea. (5h)
6. Implementar DTOs de comentarios de línea. (3h)

19. Marcapáginas

Como usuario quiero que el sistema mantenga la página por la que leo un documento en concreto.

Criterios de aceptación:

- Al volver a entrar en un documento que ha estado leyendo, el usuario se encuentra con la misma parte del documento en la que salió de él.

Descomposición en tareas (11h):

1. Implementar funcionalidad de manejo del Scroll en la vista de lectura. (3h)
2. Implementar entidad lectura en el back-end para el almacenamiento del scroll/página de la lectura de un libro por un usuario concreto. (6h)
3. Implementar DTOs para los envíos y recibimientos de lecturas. (2h)

20. Diccionario

Como usuario quiero que el sistema muestre el significado de una palabra de un documento que estoy leyendo en caso de pedirlo.

Criterios de aceptación:

- El usuario debe poder pulsar sobre una palabra del texto en específico.
- Al pulsar sobre una palabra se muestra la definición de la palabra al usuario.
- El usuario puede cerrar la ventana con la definición en cualquier momento y continuar con la lectura.

Descomposición en tareas (12h):

1. Búsqueda y comparación de una APIs externas estilo diccionario. (2h)
2. Implementar llamadas a la API y procesamiento de los datos. (5h)
3. Implementar interfaz que muestre las definiciones obtenidas. (5h)

21. Actualización de documento

Como usuario quiero que se pueda seguir trabajando sobre un documento publicado y que los nuevos cambios no se muestren hasta publicarlo.

Criterios de aceptación:

- El usuario debe poder editar un documento público.
- Las ediciones que se le hagan al documento desde que se hizo público hasta ahora no deben ser públicas.
- El usuario debe poder volver a publicar el documento para hacer públicos los nuevos cambios.
- Una vez se hayan hecho públicos los nuevos cambios todos los usuarios pueden verlos.

Descomposición en tareas (13h):

1. Desarrollo e implementación del sistema de fragmentos de texto públicos y privados en el back-end. (7h)
2. Diseño de la interfaz para modificar los parámetros de visibilidad de partes del documento. (3h)
3. Creación de la interfaz para modificar los parámetros de visibilidad de partes del documento. (3h).

22. Autocompletados

Como usuario quiero que el sistema ayude en la escritura con autocompletados.

Criterios de aceptación:

- Al usuario se le muestran palabras que predicen la palabra que está escribiendo.
- Si el usuario quiere completar lo que escribe con la palabra mostrada, puede pulsar un botón y esta se completa.

- Si el usuario no autocompleta la palabra y esta deja de coincidir con la predicción, se le muestra una nueva palabra que encaje con lo que escribe el usuario.
- Si el usuario termina de escribir la palabra desaparece la ventana de autocompletado.

Descomposición en tareas (15h):

1. Implementación de un sistema de almacenamiento de palabras. (8h)
2. Creación del sistema de predicción de palabras según su uso en la lengua utilizada. (7h)

23. Visibilidad comentario de línea

Como usuario quiero poder decidir si un comentario de línea será público o privado

Criterios de aceptación:

- El usuario puede decidir si un comentario de línea es público o privado.
- Si el comentario es público se le mostrará a algunos de los usuarios lectores siguiendo algún criterio.
- Si el comentario es privado sólo es visible por el usuario, actuando como marcador.

Descomposición en tareas (8h):

1. Diseño de interfaz para cambiar la visibilidad de los comentarios de línea. (3h)
2. Implementación de interfaz para cambiar la visibilidad de los comentarios de línea. (2h)
3. Implementación de cambios en la entidad comentario de línea para admitir las distintas opciones de visibilidad. (3h)

24. Desplazamiento automático

Como usuario quiero que durante la lectura el documento se desplace a medida que leo para no tener que desplazar la ventana por mí mismo.

Criterios de aceptación:

- El usuario puede detener el desplazamiento y reanudarlo pulsando un botón.
- El usuario puede modificar la velocidad de desplazamiento automático en cualquier momento.
- Si se pulsan otras funcionalidades como el diccionario, el desplazamiento se detiene hasta que se cierre esa otra funcionalidad.

Descomposición en tareas (18h):

1. Diseñar la interfaz para permitir el scroll automático y ajustar la velocidad de este (2h).
2. Desarrollar la interfaz para permitir el scroll automático y ajustar la velocidad de este (2h).
3. Implementar la funcionalidad que permita que la ventana se desplace hacia abajo sin necesidad de interacción del usuario. (7h)
4. Implementar compatibilidad con el guardado del scroll (Marcapáginas) y con el diccionario. (7h)

2.5. Planificación inicial de tareas

A continuación hablaremos sobre la planificación de tareas. Como hemos visto en la definición de las historias de usuario, cada una de ellas tiene un apartado de descomposición en tareas. Este apartado muestra distintas tareas que se deben

realizar para completar esa historia de usuario, y que en su conjunto forman la pila de producto.

Ahora pasaremos a la definición de los Sprints, los cuales tendrán una duración fija y se compondrá de un conjunto de tareas seleccionadas de la pila de producto anteriormente mencionada. Para cada tarea, identificamos qué roles son necesarios para su resolución. Por ejemplo, un componente de React sería asignada a un desarrollador front-end, una tarea de desarrollo de la API será asignada a un desarrollador back-end, y el desarrollo de la documentación del producto y el diseño de la interfaz de la aplicación serían tareas asignadas al analista y diseñador respectivamente.

Además de asignar tareas a cada rol, se deberán asignar puntos de esfuerzo a cada tarea, los cuales nos ayudarán a tener una idea de cuánto trabajo será necesario realizar en cada Sprint. Para asignar estos valores, utilizaremos el Método MoSCoW [7], el cuál hemos explicado y utilizado en el punto anterior.

Para asignar los valores de puntos de esfuerzo debemos tener en cuenta la complejidad de cada una de las historias de usuario, la cantidad de trabajo que se le debe dedicar y el riesgo que lleva asociado. Los valores que asignaremos seguirán la secuencia de Fibonacci [8] (1,2,3,5,8,13...), que se usa frecuentemente en metodologías ágiles como Scrum. A las historias de usuario categorizadas como “Must have” se les asignará un valor alto, porque aunque en la mayoría de ocasiones no son excesivamente complicadas, si son de vital importancia puesto que el resto de historias de usuario dependen de ellas. De la misma forma las historias de usuario categorizadas como “*Should have*” también tendrán un valor alto, porque aunque no son tan importantes a nivel de dependencias y funcionalidades en la aplicación, sí que suelen ser más novedosas y de mayor dificultad a la hora de ser implementadas. Por

lo contrario, las historias de usuario “*Could have*” con seguridad tendrán un valor de puntos de esfuerzo bajo, puesto que no son vitales para la aplicación y suelen ser *features* que no requieren de demasiado esfuerzo para ser implementadas.

A continuación, muestro la asignación de puntos de esfuerzo a cada historia de usuario que se definió anteriormente (Tabla 1):

Nº de historia de usuario	Ámbito de la historia	Puntos de esfuerzo
1	Creación de la cuenta	13
2	Acceso a la cuenta	13
3	Modificación de la cuenta	5
4	Creación de documentos	8
5	Contenido de documentos	13
6	Herramientas de edición	13
7	Visibilidad de documento	3
8	Lectura de documentos	5
9	Edición de documentos	5
10	Borrado de documentos	5
11	Generación de ilustraciones	13
12	Seguidores de un usuario	8

13	Dejar de seguir usuario	5
14	Perfil de usuario	8
15	Búsqueda de documentos	8
16	Creación de reseñas	13
17	Respuesta a reseñas	8
18	Creación comentario de línea	13
19	Marcapáginas	5
20	Diccionario	8
21	Actualización visibilidad documento	8
22	Autocompletados	8
23	Visibilidad comentario de línea	5
24	Desplazamiento automático	8
Total		256

Tabla 1. Información sobre historias de usuario y puntos de esfuerzo

Como vemos, las historias de usuario más complejas tienen un valor mayor en puntos de esfuerzo, siendo la generación de ilustraciones y las herramientas de edición a las

que más puntos de esfuerzo se han asignado. Los puntos de historia nos servirán para saber cuánto hemos avanzado en cada iteración.

Los roles que he definido son:

Analista: Se encargará de documentar todo el proceso que se ha realizado en el proyecto, por lo que no tendrá tareas asignadas como tal, puesto que su tarea es la de dejar constancia de todos los pasos y procedimientos que se han llevado a cabo durante el proyecto. Además se dedicará a investigar sobre APIs para encontrar la mejor opción para las funciones que requieran de ellas.

Diseñador UX/UI: Sus tareas son las relacionadas con el diseño, principalmente de interfaces y componentes, en el ámbito del front-end, es decir, en los elementos visuales.

Desarrollador Front-end: Se encargará del diseño de la estructura del proyecto, determinará qué componentes conformarán el front-end, cómo se relacionan entre ellos y qué patrones de diseño aplicará. Además implementará estos mismos componentes y patrones.

Desarrollador Back-end: Su trabajo se llevará a cabo en la API Rest y el servidor, determinará qué patrones, entidades, servicios, repositorios y controladores utilizará, además de implementarlos.

2.5.1 Definición de los Sprints

A continuación hablaremos sobre los Sprints. Como hemos visto anteriormente, hemos asignado a cada tarea un valor de esfuerzo en horas, sumando entre todas ellas 374 horas.

A estas horas, se les debe sumar la tarea de documentar el proceso, tarea que el analista realiza en un plazo de 7h por iteración. Por lo tanto el resultado final será de 409h o 20 semanas.

El ritmo de trabajo será de 4h al día, descansando los fines de semana, es decir, trabajando cinco días a la semana.

A la hora de definir los Sprints debemos aclarar cuánto durarán los Sprints y cuántos serán. Al finalizar cada Sprint, se revisarán los criterios de aceptación, y se verá que historias de usuario han sido completadas y cuáles no. Las historias que no se hayan completado serán trasladadas al sprint siguiente si su prioridad es alta, o se descartan en caso de que su prioridad sea *“Should have”*. Además se les añadirá puntos de historia, que son la suma de los puntos de esfuerzo de las historias de usuario que se completen en ese Sprint.

En mi caso, he definido 5 Sprints de entre 95 y 75 horas, lo que corresponde a entre cuatro y cinco semanas de trabajo por Sprint:

2.5.1.1 Sprint 1

En el primer Sprint nos centraremos en realizar las tareas de historias de usuario *“Must Have”*, ya que éstas son las más importantes y las claves en el funcionamiento de la aplicación. De esta forma, al acabar el sprint tendremos una versión de la aplicación con las funcionalidades básicas, más importantes y de las que dependen funcionalidades e historias de usuario posteriores. Nos centraremos en lo relativo a las clases usuario y documento, ya que estas son las que nos darán las funcionalidades principales.

Este primer Sprint tendrá una duración de 75 horas, es decir, que con el ritmo de trabajo acordado serán aproximadamente cuatro semanas. Las historias de usuario que se completarán serán las historias de usuario comprendidas entre la historia 1 y la

historia 4, lo que en puntos de esfuerzo se traduce en un total de 39. Las tareas que se realizarán y los roles a las que serán asignadas son los siguientes:

- **Diseñador: 9h de trabajo asignado**

- 1.1 - Diseñar el componente de formulario de registro. (2h)
- 2.1 - Diseñar el componente de formulario de inicio de sesión. (2h)
- 3.1 - Diseñar el componente de formulario de modificación de cuenta (2h)
- 4.1 - Diseñar el componente de formulario para la creación de documento (3h)

- **Desarrollador front-end: 21h de trabajo asignado**

- 1.2 - Implementar el componente de formulario de registro. (2h)
- 1.3 - Implementar validaciones de campos del formulario. (3h)
- 2.2 - Implementar el componente de formulario de inicio de sesión. (2h)
- 2.3 - Implementar la validación de credenciales en el cliente. (3h)
- 3.2 - Implementar el componente formulario de modificación de cuenta. (2h)
- 3.3 - Implementar validaciones en los campos del cliente. (3h)
- 4.2 - Implementación del componente formulario para la creación de documentos. (3h)
- 4.3 - Implementar validaciones necesarias al formulario. (3h)

- **Desarrollador back-end: 38h de trabajo asignado**

- 1.4 - Crear la entidad usuario en la base de datos. (3h)
- 1.5 - Implementar lógica de registro en el servidor, incluyendo validaciones correspondientes (5h)
- 1.6 - Crear los DTOs necesarios para el envío de información. (2h)

- 2.4 - Implementar lógica de inicio de sesión en el servidor. (5h)
- 2.5 - Crear DTOs necesarios para el inicio de sesión. (2h)
- 3.4 - Implementar sistema de modificación de usuarios en el servidor. (7h)
- 3.5 - Implementar validaciones de los datos de modificación en el servidor. (2h)
- 4.4 - Crear la entidad documento en el back-end. (5h)
- 4.5 - Crear las funcionalidades necesarias para la creación de documentos en el servidor (5h)
- 4.6 - Crear los DTOs necesarios para el envío y recibo de documentos. (2h)

- **Analista: 7h de trabajo asignado**

- Documentar el desarrollo de la aplicación front-end durante el sprint (3h)
- Documentar el desarrollo de la aplicación back-end durante el sprint (4h)

Al final de este Sprint, deberíamos tener un prototipo funcional de una aplicación que permita el registro e identificación de usuarios, además de posibilidad de crear y leer un documento, lo cual es la funcionalidad principal de la aplicación. Aún así, siguen faltando funcionalidades clave como el poder escribir en esos documentos, funcionalidades que incluiremos en el siguiente Sprint.

2.5.1.2 Sprint 2

Partiendo de los avances del Sprint anterior, en este Sprint continuaremos con las tareas relativas a las historias de usuario de categoría “*Must have*”, diseñando e implementando

las operaciones básicas de borrado y edición que no teníamos anteriormente, además de otras funcionalidades. Este Sprint tendrá una duración de 94h, que en semanas de

trabajo se traduce en un total de cinco semanas con el ritmo de trabajo acordado. Las historias de usuario que se implementarán serán las comprendidas entre las historias 5 y 10, por lo cual el Sprint tiene un peso en puntos de esfuerzo de 44.

- **Diseñador: 10h de trabajo asignado**

- 5.1 - Diseñar la vista donde se colocará el editor. (3h)
- 8.1 - Diseñar componentes para poder acceder a documentos públicos de otros usuarios. (2h)
- 9.1 - Diseñar la interfaz para acceder a la edición de documentos. (3h)
- 10.1 - Diseñar la interfaz para la eliminación de documentos propios. (2h)

- **Desarrollador front-end: 36h de trabajo asignadas**

- 5.2 - Implementar la vista donde se colocará el editor. (4h)
- 5.3 - Añadir funcionalidades de edición y contenido multimedia. (7h)
- 6.2 - Ajustar las funcionalidades de edición de textos. (8h)
- 7.1 - Implementar funcionalidad para cambiar la visibilidad de un documento en el cliente. (7h)
- 8.2 - Implementar componentes para poder acceder a documentos públicos de otros usuarios. (2h)
- 9.2 - Implementar la interfaz para acceder a la edición de documentos. (5h)
- 10.2 - Implementar la interfaz para la eliminación de documentos propios. (3h)

- **Desarrollador back-end: 34h de trabajo asignadas**

- 5.4 - Implementar el envío y recuperación del contenido de los documentos. (7h)

- 6.3 - Implementar lógica de guardado de los documentos en el servidor. (8h)
 - 7.2 - Implementar el cambio de visibilidad en el servidor (7h).
 - 8.3 - Implementar lógica en el servidor para el acceso a documentos públicos (3h)
 - 9.3 - Implementar la lógica en el servidor para la edición de documentos. (5h)
 - 10.3 - Implementar en el servidor la posibilidad de eliminar un documento. (4h)
- **Analista: 14h de trabajo asignadas**
 - 6.1 - Investigar distintas opciones para el editor de texto e implementar funcionalidades necesarias a la barra de herramientas. (7h)
 - Documentar el desarrollo de la aplicación front-end durante el sprint (3h)
 - Documentar el desarrollo de la aplicación back-end durante el sprint (4h)

Al terminar este Sprint, habríamos terminado con las historias de usuario que deben estar en la aplicación. Ahora tendríamos un prototipo en el que es posible la edición de los documentos anteriormente creados, además de la posibilidad de eliminarlos. Además implementamos la posibilidad de cambiar la visibilidad de los documentos, cosa que consideramos esencial para las siguientes historias de usuario. En el siguiente Sprint se comenzaría con las funcionalidades que “debería” tener la aplicación.

2.5.1.3 Sprint 3

Como hemos mencionado en el punto anterior, en este Sprint se comenzará con la realización de las tareas pertenecientes a las historias de usuario de tipo “*Should have*”. Se implementarán las historias de usuario referentes a la generación de

imágenes, sistema de seguimiento entre usuarios, y comenzaremos con el sistema de reseñas. Este Sprint tendrá una duración aproximada de 84h o cuatro semanas de trabajo. Las historias de usuario que se completarán van desde la historia 11 a la 15, lo que nos deja con un balance de 42 puntos de historia.

- **Diseñador: 14h de trabajo asignado**

- 11.1 - Diseñar la interfaz con la que hacer peticiones de imágenes a partir de texto. (2h)
- 12.1 - Diseño de la interfaz para seguir a otros usuarios. (3h)
- 13.1 - Diseño de la interfaz para dejar de seguir usuarios. (3h)
- 14.1 - Diseñar el perfil de usuario con la información requerida en el front-end (3h)
- 15.1 - Diseñar vista exploración para la búsqueda de documentos públicos (3h)

- **Desarrollador front-end: 34h de trabajo asignadas**

- 11.2 - Desarrollar la interfaz con la que hacer peticiones de imágenes a partir de texto. (4h)
- 11.4 - Adaptar la API a nuestro sistema y desplegarlas, haciendo su acceso público mediante URL. (6h)
- 12.2 - Desarrollo de la interfaz para seguir a otros usuarios. (3h)
- 13.2 - Desarrollo de la interfaz para dejar de seguir usuarios. (5h)
- 14.2 - Implementar el perfil de usuario con la información requerida en el front-end. (5h)
- 15.2 - Implementar vista exploratoria para la búsqueda de documentos públicos. (5h)
- 15.3 - Implementar la interfaz de búsqueda de documentos mediante tags y género. (6h)

- **Desarrollador back-end: 26h de trabajo asignadas**

- 12.3 - Implementar funcionalidades de seguimiento de usuarios en el back-end. (5h)
- 13.3 - Implementar funcionalidades necesarias para eliminar seguidor/seguimiento en el back-end. (8h)
- 14.3 - Implementar los end-points necesarios para las peticiones al servidor que realizará la vista. (7h)
- 15.5 - Implementa en el back-end las funcionalidades y end-points necesarios para el funcionamiento de la búsqueda de documentos filtrados. (6h)

- **Analista: 10h de trabajo asignadas**

- 11.3 - Buscar APIs generadoras de imágenes. (3h)
- Documentar el desarrollo de la aplicación front-end durante el sprint (3h)
- Documentar el desarrollo de la aplicación back-end durante el sprint (4h)

Como hemos anunciado al comienzo del punto, en esta iteración hemos realizado la mayoría de las historias de usuario que deberían estar en la aplicación. Cuando se termine este Sprint, el prototipo funcional resultante permitirá el seguimiento de usuarios, buscar documentos mediante género y generar imágenes con inteligencia artificial. Como vemos esto es ya un prototipo bastante completo, aunque aún nos quedan algunas funcionalidades de red social como los comentarios y reseñas, que se trabajarán en la siguiente iteración. En la siguiente iteración continuaremos con el sistema de reseñas y respuestas y el sistema de búsqueda de documentos, además de comenzar con las historias de usuario que podrían implementarse “*Could have*”.

2.5.1.4 Sprint 4

En este Sprint seguiremos asignando las tareas a cada Rol del proyecto. Terminaremos las historias de usuario catalogadas como “Should have” y empezaremos con algunas de las historias de usuario de la categoría “Could have”. La duración total del Sprint será de 83h, lo que son aproximadamente 5 semanas de trabajo. Como en este Sprint incluiremos las historias de usuario entendidas entre la 16 y la 19, la suma de los puntos de historia del Sprint será de 39 puntos.

La asignación de tareas será la siguiente:

- **Diseñador: 14h de trabajo asignado**

- 16.1 - Diseñar el componente Reseña. (2h)
- 16.3 - Diseñar la interfaz para la creación de las reseñas (3h)
- 17.1 - Diseñar el componente Respuesta. (2h)
- 17.3 - Diseñar la interfaz para la creación de respuestas vinculadas a una reseña. (2h)
- 18.1 - Diseñar el componente comentario de línea. (3h)
- 18.3 - Diseñar la interfaz para vincular un fragmento de texto a un comentario. (2h)

- **Desarrollador front-end: 25h de trabajo asignadas**

- 16.2 - Desarrollar el componente Reseña. (4h)
- 16.4 - Implementar la interfaz para la creación de las reseñas. (3h)
- 17.2 - Crear el componente Respuesta. (4h)
- 17.4 - Implementar la interfaz para la creación de respuestas vinculadas a una reseña. (3h)
- 18.2 - Implementar el componente comentario de línea. (2h)

- 18.4 - Implementar la interfaz para vincular un fragmento de texto a un comentario. (6h)
- 19.1 - Implementar funcionalidad de manejo del Scroll en la vista de lectura. (3h)
- **Desarrollador back-end: 37h de trabajo asignadas**
 - 16.5 - Implementar la entidad reseña en el back-end y las funcionalidades necesarias para su utilización (8h)
 - 16.6 - Implementar DTOs de reseña para las comunicaciones necesarias. (2h)
 - 17.5 - Implementar la entidad respuesta y sus funcionalidades y validaciones. (8h)
 - 17.6 - Crear los DTOs de respuesta para comunicación entre end-points. (3h)
 - 18.5 - Implementar la entidad comentario de línea. (5h)
 - 18.6 - Implementar DTOs de comentarios de línea. (3h)
 - 19.2 - Implementar entidad lectura en el back-end para el almacenamiento del scroll/página de la lectura de un libro por un usuario concreto. (6h)
 - 19.3 - Implementar DTOs para los envíos y recibimientos de lecturas. (2h)
- **Analista: 7h de trabajo asignadas**
 - Documentar el desarrollo de la aplicación front-end durante el sprint (3h)
 - Documentar el desarrollo de la aplicación back-end durante el sprint (4h)

Como vemos la aplicación ya tiene un gran número de funcionalidades típicas de una red social. Al terminar este Sprint, ya se podrán realizar comentarios y reseñas,

además de poder responder a los mismos. Además se ha incluido también el sistema de marcapáginas, por lo que el prototipo resultante de este Sprint también será capaz de recordar la página por la que un usuario estaba leyendo un libro. En el siguiente Sprint, terminaremos con la asignación de las tareas.

2.5.1.4 Sprint 5

Entrando en el último Sprint, seguiremos asignando las tareas a cada Rol del proyecto hasta terminarlas. Al ser este el último Sprint, al acabarlo se deberá tener la versión final del producto. Este Sprint tendrá una extensión de 73h, lo que serían aproximadamente cuatro semanas de trabajo. El último Sprint comprenderá las historias de usuario desde la número 20 hasta la última, la número 24. Esto nos deja con un balance de puntos de historia de 37 asignados al Sprint final.

La asignación de tareas será la siguiente:

- **Diseñador: 8h de trabajo asignado**
 - 21.2 - Diseño de la interfaz para modificar los parámetros de visibilidad de partes del documento. (3h)
 - 23.1 - Diseño de interfaz para cambiar la visibilidad de los comentarios de línea. (3h)
 - 24.1 - Diseñar la interfaz para permitir el scroll automático y ajustar la velocidad de este (2h).
- **Desarrollador front-end: 31h de trabajo asignadas**
 - 20.2 - Implementar llamadas a la API y procesamiento de los datos. (5h)
 - 20.3 - Implementar interfaz que muestre las definiciones obtenidas. (5h)
 - 21.3 - Creación de la interfaz para modificar los parámetros de visibilidad de partes del documento. (3h).

- 23.2 - Implementación de interfaz para cambiar la visibilidad de los comentarios de línea. (2h)
- 24.2 - Desarrollar la interfaz para permitir el scroll automático y ajustar la velocidad de este (2h).
- 24.3 - Implementar la funcionalidad que permita que la ventana se desplace hacia abajo sin necesidad de interacción del usuario. (7h)
- 24.4 - Implementar compatibilidad con el guardado del scroll (Marcapáginas) y con el diccionario. (7h)
- **Desarrollador back-end: 25h de trabajo asignadas**
 - 21.1 - Desarrollo e implementación del sistema de fragmentos de texto públicos y privados en el back-end. (7h)
 - 23.3 - Implementación de cambios en la entidad comentario de línea para admitir las distintas opciones de visibilidad. (3h)
 - 22.1 - Implementación de un sistema de almacenamiento de palabras. (8h)
 - 22.2 - Creación del sistema de predicción de palabras según su uso en la lengua utilizada. (7h)
- **Analista: 9h de trabajo asignadas**
 - 20.1 - Búsqueda y comparación de una APIs externas estilo diccionario. (2h)
 - Documentar el desarrollo de la aplicación front-end durante el sprint (3h)
 - Documentar el desarrollo de la aplicación back-end durante el sprint (4h)

2.5.2. Evolución de la planificación inicial

Habiendo explicado en el punto anterior la planificación del proyecto, ahora podemos hacer una comparativa de la planificación teórica con la aplicación de dicha

planificación. A continuación veremos un gráfico burn-down que las compara y del que podremos obtener bastante información sobre el desarrollo de la aplicación durante las iteraciones.

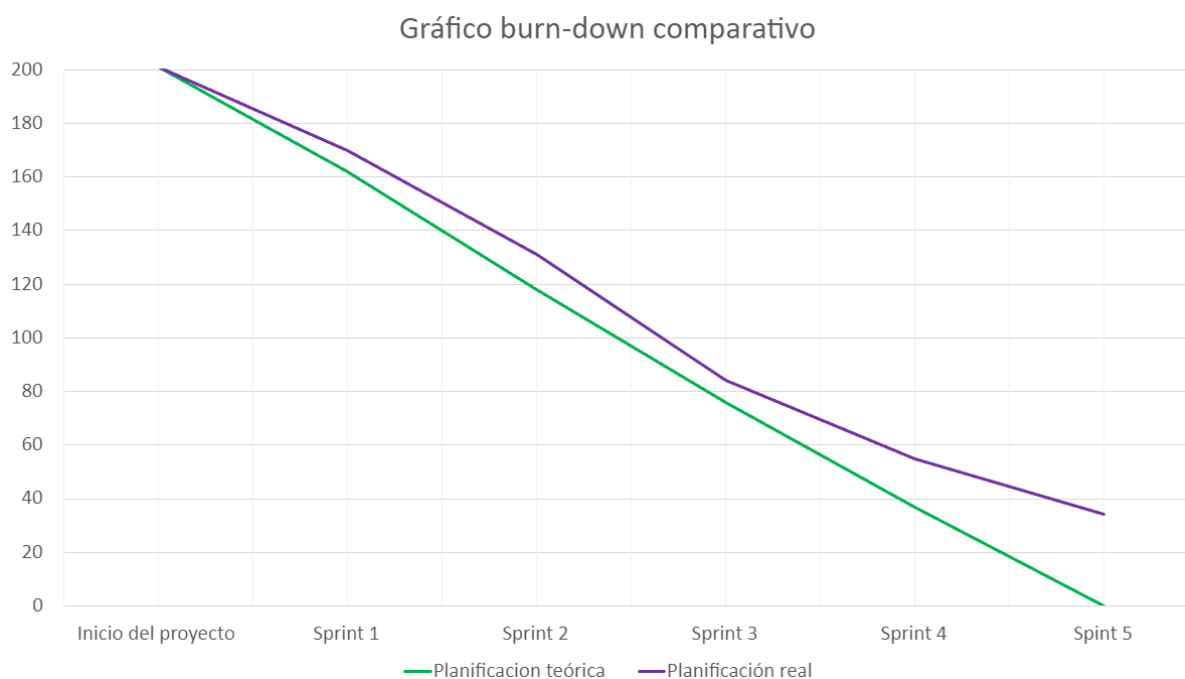


Ilustración 3. Gráfico burn-down comparación de planificaciones

Como se puede ver en la gráfica (Ilustración 3), ha habido retrasos en la planificación real respecto a la teórica, lo que ha resultado en una cantidad de puntos de esfuerzo que no se han conseguido finalizar, lo que es sinónimo de que algunas historias de usuario no han podido ser implementadas y de que varias de las historias de usuario han supuesto más esfuerzo del estimado en un inicio. Al centrarnos en los primeros Sprints, observamos que ya desde el comienzo existe un retraso, resultado de errores de estimación en el trabajo requerido para algunas de las operaciones básicas de la aplicación. Aún así, el desfase en los puntos de esfuerzo no es demasiado. Este retraso resultó en que las historias de usuario que no fueron implementadas en el

primer Sprint, en concreto la historia de usuario número 4, tuviese que ser implementada en el Sprint 2, lo que provocó el retraso que refleja la gráfica. Aún así, al finalizar el Sprint 3, es notable una recuperación significativa en el retraso de la planificación. Esto se debe a que ese Sprint fue más fructífero de lo esperado, pudiendo realizar las principales historias de usuario con mayor valor de puntos de esfuerzos a tiempo. No fue el caso de los Sprints 4 y 5, en los cuales la estimación errónea provocó el no finalizarlos con los puntos de esfuerzo esperados, en el caso del Sprint 5, hizo que tuviésemos que decidir qué historias realizar y cuales dejar a un lado en este prototipo final.

Las historias de usuario que tuvimos que descartar fueron las historias 18. Creación de comentario de línea, 21. Actualización de documento, 22. Autocompletados y 23. Visibilidad de comentarios de línea. Todas estas historias fueron categorizadas como *“Could have”*, por lo que eran aspectos más secundarios de la aplicación.

2.6. Estudio de viabilidad

En este apartado analizaremos los costes de desarrollo y explotación del proyecto, además de proponer algunos modelos de negocio que podría seguir el cliente para recuperar la inversión u obtener beneficios. Un buen estudio de los costes nos ayuda a conocer si un proyecto es viable desde el punto de vista financiero, a la toma correcta de decisiones, a la gestión de riesgos y nos ayuda a determinar qué modelo de negocio usar y qué precios adoptar.

El coste de desarrollo de una aplicación de estas características requiere de una cantidad considerable de gastos iniciales. Aquí se incluyen sueldos de los empleados mencionados anteriormente, siendo Desarrolladores, analista y diseñador, la compra del software y las licencias necesarias, además de los equipos hardware que se requieran.

A continuación analizaremos los costes de desarrollo en profundidad.

2.6.1- Costes Software

Los costes de desarrollo a nivel de Software incluyen todas las licencias de programas software, que suelen ir desde entornos de desarrollo a servicios externos necesarios. En la tabla 2 se hace un desglose de los costes software.

Software	Uso	Gastos durante el proyecto (5.5meses)	Gasto por suscripción
IntelliJ Idea Ultimate	Entorno de desarrollo para la API Rest	274,5€	599€/año
Visual Studio Code	Entorno de desarrollo para front-end	0€	0€
Notion	Tablero organizador de tareas	0€	0€
Google colab	Entorno de desarrollo API Generadora de imágenes	0€	0€
Google docs	Editor de textos	0€	0€

	para la creación de la documentación		
Visual Paradigm Enterprise	Herramienta de modelado para la creación de los diagramas UML	489,5€	89€/mes
GitHub	Sistema de control de versiones y alojamiento de código para el proyecto	0€	0€
Photoshop	Software de edición de imágenes para wireframes y diseños de la interfaz	137 €	24.99€/mes
Gastos totales	Gastos necesarios en aspectos software para el desarrollo de la aplicación	901€	

Tabla 2. Costes software

Aunque el gasto total de desarrollo en aspectos de software accede a un total de 901€, en el caso de nuestro desarrollo los costes en aplicaciones como Visual

Paradigm o IntelliJ Idea Ultimate se ven reducidos a cero puesto que la universidad nos otorga estas licencias gratuitamente. En algunos casos, como es el caso de IntelliJ, la licencia se obtiene por una duración mayor a la duración del proyecto, por lo que sería necesario pagar el precio completo de un año de suscripción a pesar de que la suscripción no se usará al completo.

2.6.2- Costes Hardware

En lo que a los costes Hardware se refiere, los empleados recibirán cada uno un portátil para que puedan utilizar durante el desarrollo del proyecto. Recursos externos como internet o luz no se tendrán en cuenta puesto que el trabajo será remoto. Se ha elegido un dispositivo portátil ya que el desarrollo de la aplicación no requerirá demasiada potencia, y es más cómodo para el trabajo remoto al poder portarse a cualquier lugar que se necesite, además de incluir todos los recursos necesarios, como pantalla, teclado, etc. Se entregará también un ratón para hacer el trabajo más cómodo a los empleados y unos cascos para facilitar las reuniones entre ellos.

A continuación desglosamos los gastos del desarrollo en cuestión de hardware para un único empleado, tomando en cuenta el precio por unidad y la vida útil de cada producto para calcular el gasto que representa en el proyecto (Tabla 3).

Hardware	Precio relativo al proyecto (5,5 meses)	Precio por unidad	Vida útil del producto
Portatil HP Pavilion	68,6€	599€	4 años
Logitech lift ratón	9,4€	62€	3 años

Logitech G Pro Auriculares	8€	88€	5 años
Total	86€	749€	

Tabla 3: Costes hardware

Como vemos en el desglose de los costes, los recursos hardware en lo que respecta al tiempo de desarrollo del proyecto ascienden a 86€.

2.6.3- Salarios

En este apartado comentaremos los costes que respecta a los salarios de los empleados. Para calcular los costes, nos guiaremos por las horas de trabajo asignadas a cada rol y el salario que recibirá por hora. Para conocer qué salario se ajusta a la realidad, utilizaremos la página web indeed [9], en la que se pueden consultar salarios según puesto de trabajo y lugar.

A continuación (Tabla 4), un desglose de los costes en salarios, indicando el pago por hora y el pago total.

Puesto	Salario / hora	Número de horas	Salario total
Analista	16€	47	752€
Desarrollador back-end	15,65€	147	2300€
Desarrollador front-end	16,67€	160	2677€

Diseñador	11,16€	55	613,8€
Total			6342€

Tabla 4. Costes salarios

Como vemos el resultado total sería de 6342€ de costes en salarios. Al ser ambos desarrolladores los trabajan un mayor número de horas debido a la carga de trabajo que se les impuso en la asignación de tareas, estos serán los más costosos en cuestión de salarios durante el desarrollo, a la par que los más fundamentales.

2.6.4- Costes totales del proyecto

Una vez que hemos realizado el cálculo de los costes en función de su categoría, podremos calcular el coste total del desarrollo del proyecto. Para ello, sumaremos los costes calculados anteriormente y posteriormente haremos una valoración de los costes obtenidos

Coste calculado	Valor del coste
Hardware	86€
Software	344€
Salarios	6342€
Total	6772€

Tabla 5: Costes totales

Como vemos en el resultado de los cálculos, el coste total para el desarrollo de la aplicación, será aproximadamente de 7597€, siendo la mayor parte de estos costes destinados a salarios. Este valor es importante puesto que ayudará tanto a la empresa

desarrolladora como al cliente a compararlo con los beneficios que espera y calcular el ROI⁶, además de ayudar a plantearse modelos de negocio que puedan suplir estos gastos en el tiempo esperado, modelos de negocio de los que hablaremos en el punto siguiente.

2.6.4- Costes de explotación del sistema

Una vez que hemos hablado de los costes de desarrollo, pasamos a comentar los costes de explotación. Estos costes son derivados de la necesidad de mantener la aplicación activa, como servidores, mensualidades de APIs utilizadas, etc.

A continuación hablaremos de los costes de instalaciones y servicios. En nuestro caso, al realizarse el trabajo de forma remota, no existen gastos relativos a oficinas ni entornos físicos.

Realizaremos un desglose de los gastos que se deberán emplear en servicios para el correcto funcionamiento de la aplicación. Tendremos en cuenta que necesitaremos un servidor, un dominio web y una API para la generación de imágenes una vez se haya desarrollado al completo.

Servicio	Precio al mes
Servidor AWS	277€
Base de datos Amazon RDS	138€
Dominio web	4,17€
MidJourney API	48€

⁶ Retorno por inversión

Total	467,17€
-------	---------

Tabla 6. Costes servicios

Al haber terminado con los cálculos de los gastos mensuales para el funcionamiento de la aplicación, podemos observar que los gastos ascienden a 467€ en servicios, todo esto en la etapa más temprana de funcionamiento de la aplicación.

Para subsanar estos gastos mensuales, tendremos que plantear modelos de negocio que nos den beneficios mayores a estos gastos al mes.

2.6.4- Modelos de negocio

En este apartado sobre el modelo de negocio, hablaremos del método que utilizaremos para generar ingresos. Comentaremos dos métodos como posibles sistemas de creación de beneficios, comentando sus desventajas y ventajas.

1. **Inclusión de anuncios:** Un método posible para generar ingresos sería la inclusión de anuncios en la aplicación web en forma de *banners* o videos flotantes. A partir de estos anuncios, la aplicación generará dinero por cada visita realizada. La métrica más utilizada para contabilizar cuánto se genera por visita es el CPM o coste por mil visitas. Este valor varía en cada País y página web. Para hacer nuestros cálculos, asumiremos que se recibirán 5€ por cada mil visitas a la página con anuncios. Con este CPM, necesitaríamos 93.434 visitas mensuales para cubrir los gastos mensuales de la aplicación. El principal beneficio de este sistema es que los usuarios podrán utilizar la aplicación de forma gratuita, lo que aumenta el número de usuarios que la visitan mensualmente. Los aspectos negativos son la invasividad de los anuncios, además de la necesidad de un gran número de usuarios activos. Otro aspecto positivo podría ser la posibilidad de pagar a los creadores de documentos un

porcentaje de los beneficios de los anuncios, lo que generaría más interés por crear documentos, lo que atraería a más usuarios.

2. **Suscripción mensual:** Otro método de negocio que podría implementar la aplicación sería la suscripción mensual. Con este método, se crea una suscripción que los usuarios puedan pagar con el fin de obtener beneficios que no se puedan conseguir de forma gratuita o para eliminar los anuncios de la aplicación y poder tener una interfaz más limpia y menos intrusiva. El precio de la suscripción depende de muchos factores, principalmente de los beneficios que otorga, incluso pueden existir varios tipos de suscripciones. En nuestro caso, pondremos el ejemplo de una suscripción que nos da acceso al uso de la generación de imágenes y publicita nuestros documentos en la aplicación de forma que llegue a más público y beneficie al autor. En este caso, supongamos que el precio de la suscripción será de 10€. Recordando que los gastos mensuales son de 476€, la aplicación necesitaría 500 usuarios suscritos al servicio para suplir los gastos mensuales en servicios externos. Los beneficios de este sistema son que si ofrecemos buenos servicios en la suscripción, no es necesario una cantidad tan grande de usuarios, además de que mantiene la interfaz limpia y sin anuncios. El punto negativo es que se limitan las funcionalidades de usuarios no suscritos.

Hemos comentado dos modelos de negocio de los más utilizados. En nuestro caso, se optará por utilización de ambos métodos, incluyendo publicidad en futuras versiones, lo cual nos otorgaría beneficios de la reproducción de anuncios, además de la inclusión de un sistema de suscripción que elimina estos anuncios para el usuario suscrito y otorga beneficios a la hora de recomendar su documento y usar ciertas funcionalidades.

2.7. Modelo de dominio

A continuación hablaremos sobre el modelo de dominio. El modelo de dominio es un modelo conceptual que representa las entidades de la problemática a tratar y las relaciones entre ellas. A lo largo del punto explicaremos con detenimiento las entidades y cómo son sus relaciones, adjuntando un diagrama para clarificar los conceptos explicados.

Comenzaremos con la definición de las entidades.

- **Usuario:** Esta entidad almacena los datos de un usuario que se registra en la aplicación. Podrá crear y leer documentos, por lo que se relaciona con documentos y con lectura. Además puede publicar comentarios, de los que derivan reseña, respuesta y comentario de línea. Estas acciones quedan guardadas en forma de instancias de la entidad Actividad. Los usuarios además siguen y tienen seguidores, por lo que la entidad usuario también se relaciona consigo misma.
- **Documento:** Contiene la información relativa a los documentos. Contienen reseñas y comentarios de línea. Además tienen un usuario creador, además de un valor asociado de puntuación en función de las reseñas del documento. Contienen también las lecturas de las que forman parte.
- **Comentario:** De esta entidad derivan las entidades Reseña, Respuesta y Comentario de línea. Contiene información del texto del comentario, su creador y el documento al que comentan, además de la fecha en la que se creó.

- **Reseña:** Es una especialización de la entidad comentario. Incluye además información como la puntuación otorgada en la reseña y las respuestas asociadas. Las respuestas sólo se pueden efectuar sobre una reseña. Sólo se permite una reseña por usuario y documento.
- **Respuesta:** Es una especialización de la entidad Comentario. Esta entidad se relaciona con Reseña, ya que su función es la de almacenar la información de respuestas de usuarios a una reseña en concreto. Un único usuario puede crear varias respuestas por documento y reseña.
- **Comentario de línea:** Es otra especialización de la entidad Comentario. Además de incluir la información que comentario contiene, incluye también información sobre qué parte del documento es la comentada.
- **Lectura:** Es una entidad que almacena información sobre una lectura de un documento por parte de un usuario. Esta información es la página por la que el usuario dejó su lectura la última vez que leyó el libro.
- **Actividad:** Esta entidad se relaciona con alguna de las otras entidades. Su función es la de almacenar la actividad de los usuarios, por ejemplo, si un usuario publica un documento, quedará registrado en una de estas entidades. Se almacenará también la fecha en la que se ha realizado la acción.

A continuación, mostramos un diagrama de modelo de dominio que muestra las relaciones entre dichas entidades. (Ilustración 4).

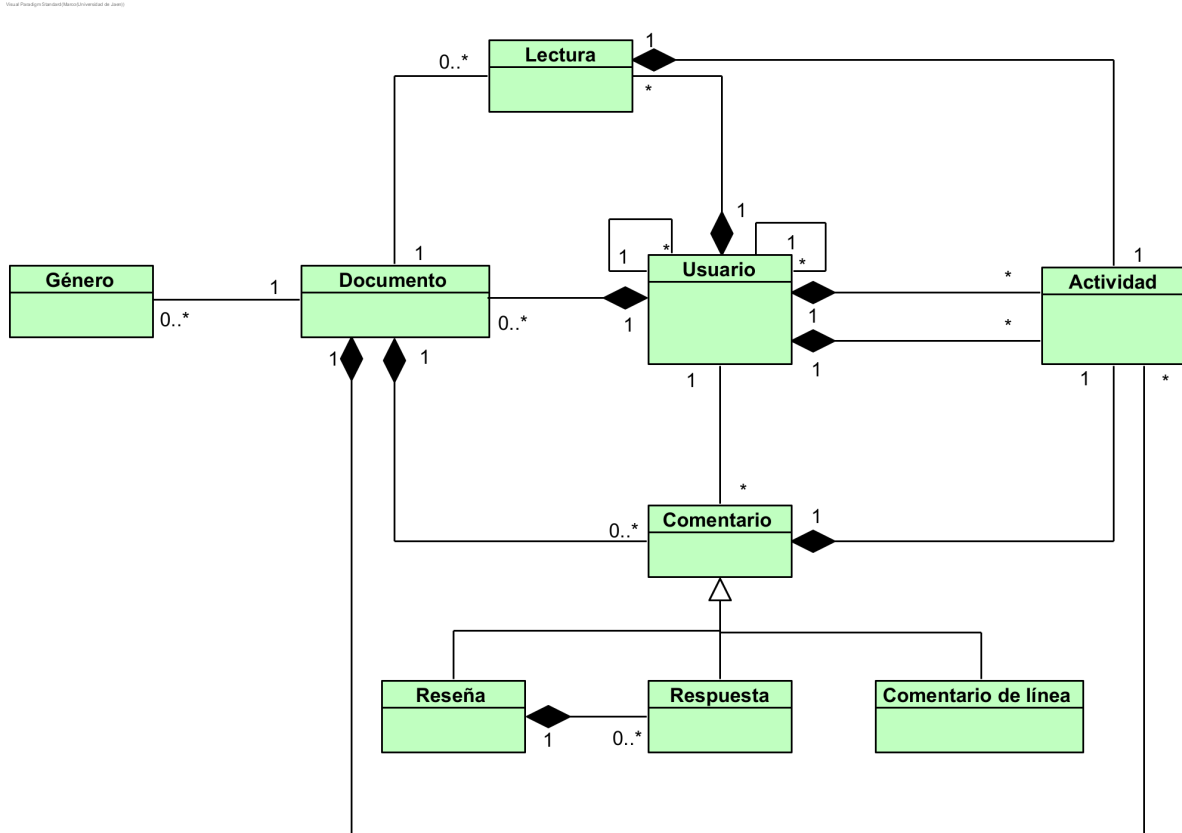


Ilustración 4. Diagrama de modelo de dominio

Una vez tenemos una vista general de las relaciones en el diagrama, las comentaremos una a una.

- Documento - Usuario: Es una relación de composición, ya que el documento depende del usuario para su existencia. Un documento se relaciona con el usuario que lo ha creado. Un usuario puede crear tantos documentos como quiera, mientras que un documento sólo es creado por un usuario, por lo tanto obtenemos la cardinalidad que observamos en la imagen.
- Usuario - Usuario: Es la relación de asociación de seguimiento entre usuarios. Un usuario puede seguir a muchos usuarios, y muchos usuarios pueden seguir a un usuario, de ahí la cardinalidad vista en ambas relaciones Usuario - Usuario

- Usuario - Comentario: Relación de asociación. Un usuario puede ser creador de varios comentarios de distintos tipos. Un comentario sólo puede ser escrito por un usuario.
- Documento - Reseña: Un documento tiene un número indeterminado de reseñas asociadas, existiendo la posibilidad de que no tenga ninguna reseña asociada. Es una relación de composición, si se elimina el documento las reseñas quedarían eliminadas.
- Reseña - Respuesta: Una respuesta es un comentario que responde a una reseña. Esta relación es un caso concreto de la relación entre documento y comentario. Varios usuarios pueden responder a una reseña, incluso el mismo usuario puede responder varias veces a la misma reseña, por lo tanto una reseña admite entre ninguna y muchas respuestas. Es una relación de composición, puesto que si la reseña se eliminase, las respuestas asociadas a esa reseña serían eliminadas.
- Documento - Comentario de línea: Es un caso concreto de la relación entre documentos y comentarios. Un documento puede relacionarse con un número indeterminado de comentarios de línea entre 0 y muchos, ya que un usuario puede crear estos tipos de comentarios varias veces en un mismo documento o varios usuarios pueden hacerlo. De la misma manera que el resto de los descendientes de comentario, esta relación sería una composición.
- Documento - Género: Un documento puede tener géneros asociados. La relación entre ambas entidades es de asociados como hemos comentado

anteriormente, ya que los documentos y los géneros existen de manera independiente.

- Documento - Lectura: Un documento puede aparecer en varias lecturas, mientras que una lectura sólo contempla un documento. Es importante aclarar que aunque un documento puede aparecer en varias lecturas, sólo puede aparecer una vez junto a un usuario en una lectura, es decir, los pares documento, usuario sí son únicos.
- Usuario - Lectura: Un usuario puede leer varios documentos, y por lo tanto aparece en varias lecturas, sin embargo una lectura sólo puede contemplar un usuario. Como hemos recalcado en la relación anterior, los pares usuario, documento sí son únicos. En caso de dejar de existir el usuario o el documento, la lectura quedaría eliminada, por lo que estaríamos ante una composición.
- Actividad - Usuario: Existen dos relaciones entre Actividad y Usuario, ambas serían una composición, debido a que Actividad depende del usuario para su existencia. La primera de ella indica la autoría de la actividad. Un usuario hace una acción y esta acción queda registrada bajo su autoría. Un usuario puede realizar varias acciones mientras que una acción es realizada por un usuario, de ahí la relación que se muestra en el diagrama. La segunda de estas relaciones indica sobre lo que se realiza la acción. Si un usuario A sigue a un usuario B, A será el sujeto de la primera relación, mientras que B será el sujeto de esta relación que estamos destacando ahora mismo.
- Actividad - Comentario, Actividad - Lectura: Estas relaciones siguen el mismo fundamento que la segunda relación del punto anterior. Actividad se relaciona con un sólo comentario, ya que esta relación almacena la acción de un usuario

de crear un comentario. En el sentido contrario un comentario sólo aparece en una actividad, puesto que sólo se crea una actividad en la creación. En caso de editar o borrar un comentario y crear otro, la actividad anterior se eliminaría, creando otra nueva y manteniendo la unicidad de la relación. Lo mismo ocurre con la Lectura, una actividad sólo contiene una lectura y una lectura sólo puede aparecer en una actividad, puesto que sólo se genera una lectura en el momento que un usuario concreto, comienza a leer un documento concreto, y la entidad será permanente en el tiempo para mantener los datos que se requieran en la base de datos indefinidamente. Ambas relaciones son de composición, ya que el comentario es parte esencial de la actividad, de la misma forma que lo es la lectura.

- **Actividad - Documento:** Una actividad sólo puede contener un documento, mientras que un documento puede aparecer en varias actividades, ya que distintos usuarios pueden comenzar a leer un mismo documento, lo que crearía actividades indicando esa acción por parte de los usuarios, guardando ese mismo documento en cada una de las actividades. Es una relación de composición puesto que la eliminación del documento resulta en el borrado de la actividad.

Como hemos visto en el diagrama de la ilustración 4, aunque no contamos con un gran número de entidades, las relaciones entre ellas son variadas y complejas, lo que más adelante provocará que necesitemos pulir estas relaciones y las dependencias que generen, tanto en el apartado de diseño como en el de implementación, para conseguir que las acciones sobre una de ellas, tengan el efecto deseado en las que dependen de ellas, sobre todo en las relaciones de composición, que son las más fuertes de todas las relaciones.

3. Diseño

El diseño es una etapa del desarrollo de un sistema software que determina cómo el sistema funcionará para cumplir con las historias de usuario que se han decretado en el apartado de análisis. Esto implica tomar las necesidades del usuario y trasladarlas a un diseño funcional. Este proceso incluirá la creación de diagramas Entidad-Relación, de clases y de secuencia. Esto ayudará a tener un mapa detallado para los desarrolladores del sistema.

Este punto es fundamental puesto que establece las bases que la implementación necesita. Si el diseño no es correcto el sistema puede no cumplir las necesidades de los usuarios o incluso ser imposible de implementar. Además se deben considerar posibles cambios futuros del sistema, ya que un buen diseño no solo refleja las necesidades actuales del sistema si no que también facilita los cambios y actualizaciones futuras.

Como hemos comentado anteriormente, usaremos distintos tipos de diagramas para reflejar el diseño del sistema de forma gráfica. Cada diagrama tiene una función específica que comentaremos en cada punto siguiente, pero también tienen características comunes. Por ejemplo, tanto los diagramas de clases como los de secuencia están orientados a objetos, y todos son eficaces en funciones como detectar problemas o comunicación universal, pudiendo ser interpretados por cualquier desarrollador o diseñador que conozca su funcionamiento.

3.1. Diagrama Entidad-Relación

Los diagramas Entidad-Relación son una herramienta gráfica que se utiliza para modelar los datos en un sistema y las relaciones entre estos mismos datos. Fue

desarrollado por Peter Chen en los años 70 y es muy utilizado en el diseño de bases de datos. Un diagrama entidad-relación puede representar tanto la estructura lógica como la estructura física de la base de datos.

Los diagramas Entidad-Relación constan de los siguientes componentes:

- Entidades: Son objetos con relevancia en el sistema que se está modelando. Estos son objetos del mundo real y que son distinguibles de otros objetos. Las entidades pueden ser fuertes, si su existencia no depende de la existencia de otras entidades, o débiles, en el caso contrario.
- Atributos: Son las propiedades o características que describen a las entidades, es decir, son un detalle que describe a una entidad en concreto. Se destaca el tipo de atributo “clave” el cuál es un atributo cuyo valor es único para cada instancia de un tipo de entidad.
- Relaciones: Son las conexiones o asociaciones entre entidades. Estas describen cómo las entidades interactúan entre sí. Existen distintos tipos de relaciones, y se dividen según su cardinalidad, es decir, cuántas instancias de una entidad están relacionadas con cuántas instancias de otra entidad.

Este tipo de diagramas se utilizan principalmente para crear un modelo visual de la base de datos que ayude a los diseñadores a entender el sistema y hacer que satisfaga las necesidades de la aplicación, además de servir como documentación para futuros desarrolladores y administradores de la base de datos.

Se utilizará un Mapeador de Objetos Relacionales, en concreto JPA. Este mapeador facilita el modelo de implementación de la base de datos, ya que nos permitirá crear

clases que se correspondan con las entidades del diagrama Entidad-Relación. Esto se hace mediante clases las cuales se completan con la información sobre las relaciones entre entidades y sus atributos. Luego JPA se encarga de generar automáticamente las tablas y relaciones de la base de datos en base a estas clases de entidad, por lo que ya no sería necesario crear y mantener manualmente los esquemas de la base de datos SQL. Todo este proceso se realiza mediante una implementación en Java y anotaciones específicas de JPA.

Otro proceso que realiza JPA es la normalización del modelo a tercera forma normal, lo que significa que cada entidad será dependiente únicamente de su atributo clave o identificador. Convertir el diagrama Entidad-Relación a tercera forma normal nos ayuda a reducir la duplicidad de los datos y evitar confusiones y problemas que se pueden producir cuando los datos no están almacenados de forma eficiente. Es importante recordar que esta tercera forma normal se consigue tras aplicar a las tablas la primera forma normal⁷ y segunda forma normal⁸

A continuación, el diagrama Entidad-Relación de nuestra base de datos (Ilustración 5)

⁷ La primera forma normal se consigue al asignarle a cada tabla una clave primaria y hacer que cada columna tenga un valor único

⁸ La segunda forma normal se consigue al hacer que cada columna de una tabla dependa completamente de su clave primaria

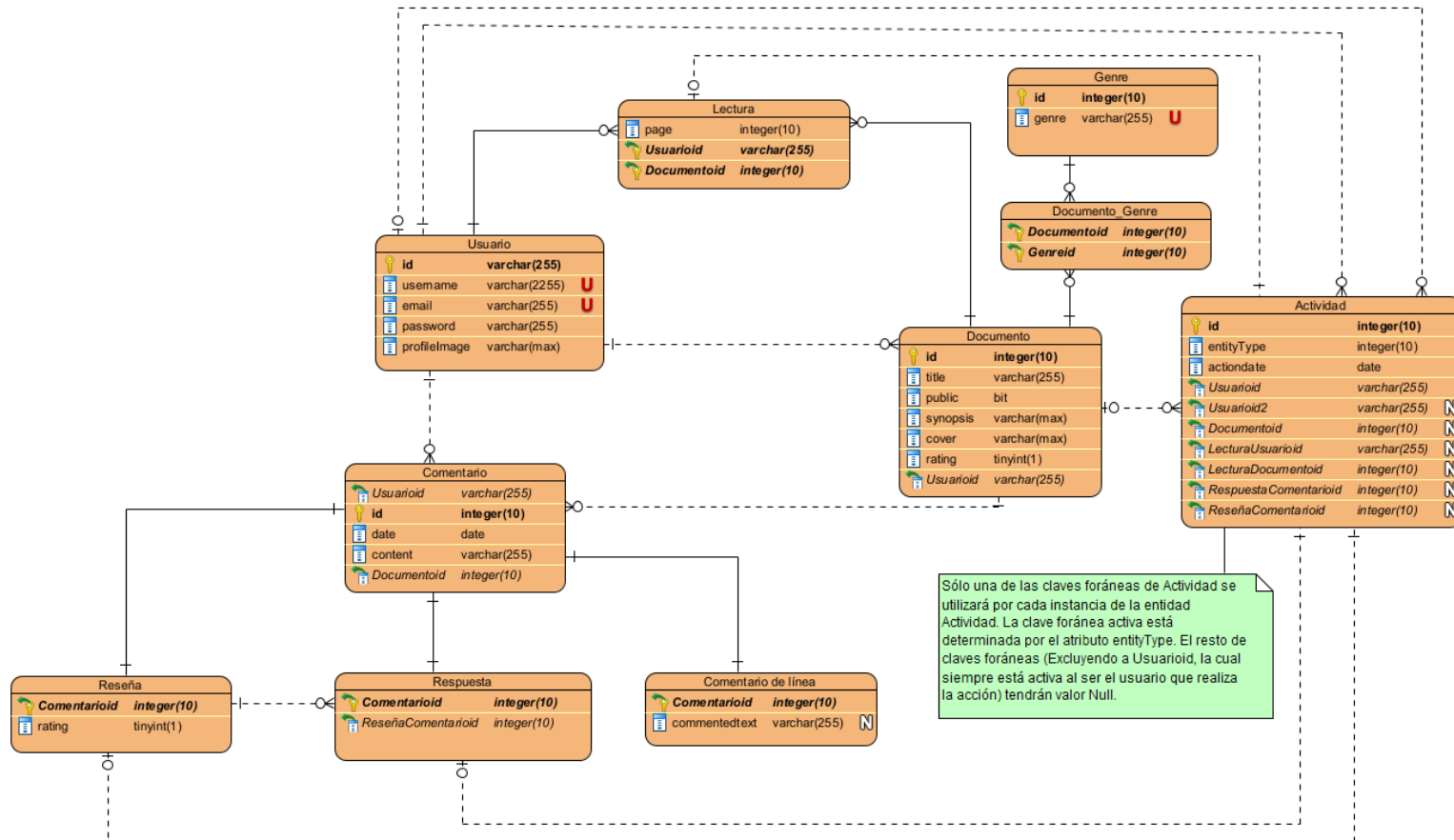


Ilustración 5. Diagrama Entidad-Relación

3.2. Diagrama de Clases

Tras haber mostrado y comentado el DER, nos centraremos en los diagramas de clases. Los diagramas de clase son uno de los diagramas UML más utilizados, ya que representan la estructura del sistema, junto a sus clases, atributos, métodos y relaciones.

Nuestra arquitectura es una arquitectura cliente-servidor. Este tipo de arquitectura es un modelo de diseño en el que los servicios se dividen en proveedores de servicios, también conocidos como servidores y solicitantes de servicios, también llamados clientes. Este es un modelo muy utilizado en aplicaciones web porque proporciona una forma eficaz de compartir servicios y recursos de una forma centralizada. En esta arquitectura el servidor se dedica a atender y satisfacer las peticiones que realizan los clientes.

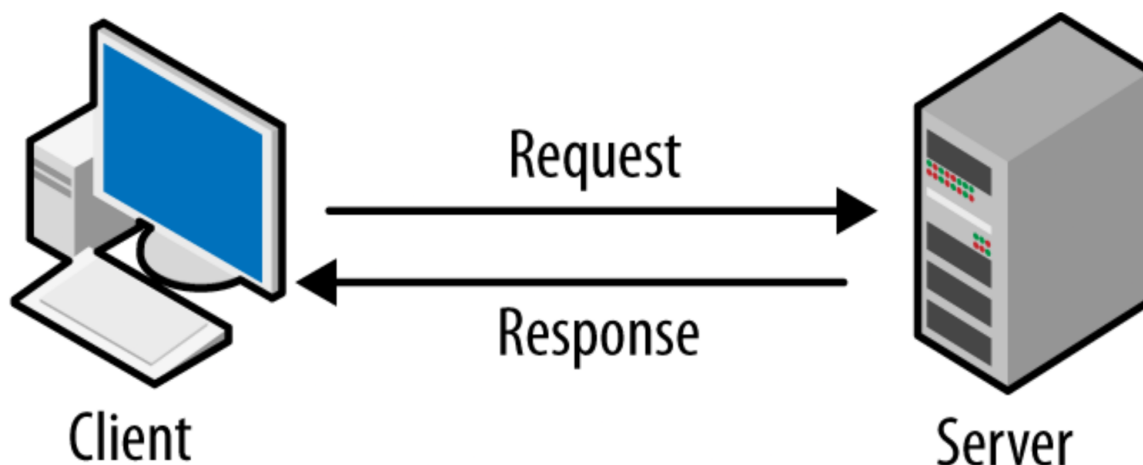


Ilustración 6. Funcionamiento arquitectura cliente - servidor

(https://madooei.github.io/cs421_sp20_homepage/assets/client-server-1.png)

Al utilizar este tipo de arquitectura, será necesario crear un diagrama de clases que especifique y documente las estructuras de ambos, cliente y servidor.

3.2.1- Diagrama de clases del Servidor

Como hemos introducido en el punto anterior, comentaremos la estructura del servidor además de ilustrarla con un diagrama de clases. El servidor se encarga de gestionar los recursos realizando las operaciones que se requieren por parte del cliente. Para cumplir con esta funcionalidad, implementaremos el patrón arquitectura REST.

El patrón REST [10] se utiliza para diseñar e implementar sistemas escalables, de alto rendimiento y robustos. REST no es un estándar ni un protocolo, es más bien un conjunto de principios que se pueden seguir para crear aplicaciones con las cualidades comentadas anteriormente. Los principios clave de las APIs REST son los siguientes:

- Protocolo Cliente/Servidor sin estado: El servidor no almacena información de las peticiones, por lo tanto cada petición debe incluir toda la información necesaria para que el servidor pueda entenderla, ya que tampoco guarda información entre distintas peticiones.
- Sistema de capas: El cliente no conoce si está conectado al servidor final, ya que puede estar conectado a un servidor intermedio. Esto lo que consigue es una mayor independencia entre el cliente y el servidor, lo que mejora la escalabilidad y flexibilidad del sistema.
- Interfaz uniforme: Las APIs REST implementan una interfaz uniforme, lo que se traduce en una simplificación del sistema y en una simplificación. Esta interfaz

incluye la identificación de recursos mediante URL, El uso de representaciones de datos en las peticiones HTTP, como por ejemplo JSON o los mensajes autodescriptivos, lo que refiere a que cada mensaje contiene información sobre cómo debe ser procesado.

- **Códigos de estado HTTP:** Este tipo de APIs utiliza códigos de estado HTTP para indicar el resultado de una solicitud de un cliente. por ejemplo un código HTTP “200 - OK”, indicaría que las operaciones producto de una petición han sido exitosas.
- **Métodos HTTP:** Este tipo de patrón utiliza métodos HTTP para indicar qué operación se debe realizar sobre el conjunto de recursos. Los métodos más comunes son GET (Obtener un recurso), POST (Crear un recurso), PUT (Actualizar un recurso), DELETE (Borrar un recurso) y PATCH (Actualización parcial de un recurso)

A continuación mostraremos el diagrama de clases del servidor (Ilustración 7) y procederemos a comentar los detalles más relevantes.

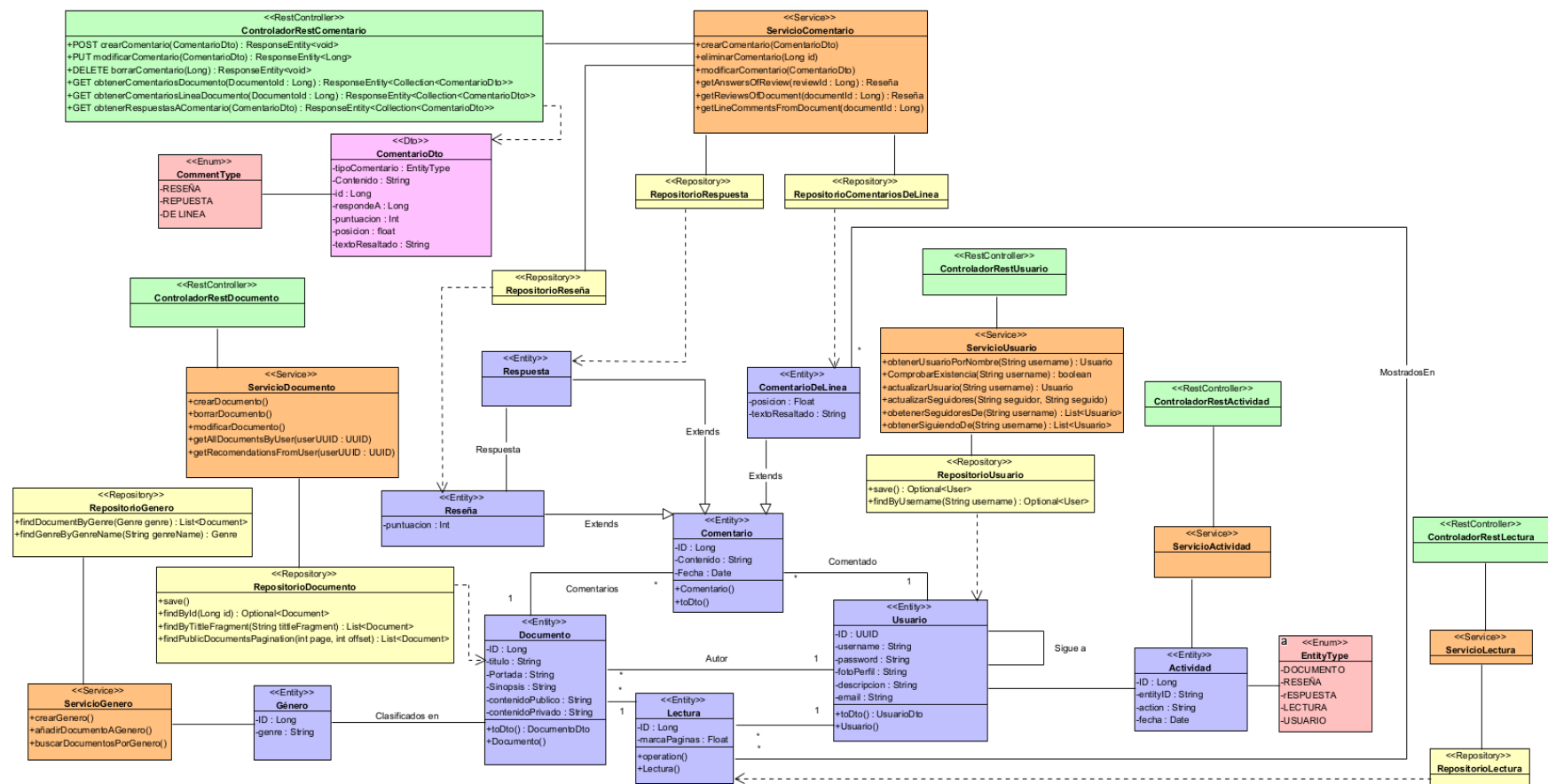


ilustración 7. Diagrama de clases del servidor

Vamos a comentar la estructura que utiliza el servidor para implementar el patrón REST. Como vemos en el diagrama UML, Las clases tienen un color y están etiquetadas según su función en el servidor y se clasifican en:

- Entidades: Las entidades se representan en el diagrama en color morado y bajo la etiqueta “*Entity*”. Las entidades son clases que representan los conceptos de dominio y se mapean a una tabla en la base de datos. Son los recursos con los que opera el servidor. Se observan en cada entidad los atributos principales que contiene, los cuales son la información y características de cada entidad.
- Repositorios: Los repositorios se muestran en el diagrama en color amarillo y etiquetados como “*Repository*”. Los repositorios son interfaces que proporcionan métodos para operar con la base de datos. Las operaciones básicas no han sido incluidas en el diagrama puesto que son numerosas y todos los repositorios las incluyen. Operaciones como guardar, buscar mediante atributos, borrar o actualizar son operaciones que todos los repositorios del diagrama incluyen pero por falta de espacio no han sido añadidas. Se han incluido en el diagrama las operaciones más importantes ya que el espacio es limitado.
- Servicios: Aparecen en el diagrama etiquetados como “*Service*” y con color naranja. Los servicios contienen la lógica de negocio en la aplicación y utilizan repositorios y otros servicios para leer y escribir en los datos. Son responsables de funcionalidades como el lanzamiento de excepciones y la realización de operaciones complejas mediante el uso de las operaciones simples de los repositorios. Normalmente hay un servicio por entidad, excepto en el caso del servicio Comentario, el cual gestiona las operaciones de los repositorios

comentario, respuesta, reseña y de línea, implementando el patrón estrategia, el cual decide qué comentario es el que se está pidiendo o creando en tiempo de ejecución y utiliza una estrategia u otra para realizar la operación.

- Controladores: Etiquetados como “*RestController*” y en color verde, manejan las solicitudes y respuestas HTTP. Utiliza rutas específicas para recibir dichas peticiones, que pueden ser de los distintos tipos de los que hemos hablado anteriormente y se muestra en el controlador Rest Comentario, el cual ha sido el único en el que se muestran las operaciones debido al poco espacio disponible. Normalmente cada controlador comunica la petición a un servicio, que se encarga de realizar las operaciones necesarias y llamar a los repositorios necesarios para devolver un valor al mismo controlador, el cual envía una respuesta. En el caso del Controlador Comentario vemos que es el Servicio Comentario quien realiza esta función. En estos controladores también se implementará el manejo de las excepciones. El único caso especial es el caso de la entidad “*Genre*”, la cual no cuenta con un controlador Rest puesto que estas entidades se crean en el despliegue de la aplicación y son entidades constantes, ya que existirán unos géneros determinados que no pueden ser añadidos ni eliminados por usuarios externos.
- DTO: Los DTO⁹ son objetos que se utilizan para encapsular datos, principalmente para ser enviados al cliente, definiendo así el formato de los objetos en las solicitudes y en las respuestas. Eso es de gran utilidad puesto que nos evita el enviar o recibir información innecesaria y evita el envío de las entidades que se relacionan con la entidad enviada, que a su vez se relacionan con otras entidades que también se enviarán, lo que provocaría envíos muy pesados que pueden incluso provocar errores en las aplicaciones.

⁹ Data Transfer Objects

Prácticamente todas las entidades del diagrama tienen asociado un DTO, del cual se hace uso en el controlador REST. En el diagrama aparecen en color rosado bajo la etiqueta “*Dto*”. En el caso del DTO de comentario, sólo existe uno para todos los tipos de comentario y mediante la enumeración “*commentType*” el servicio es capaz de discernir y realizar la estrategia necesaria como hemos comentado en el punto del servicio.

3.2.2- Diagrama de clases del Cliente

Ahora pasaremos a comentar la estructura del cliente en lo que a sus clases respecta. Comentaremos los patrones que hemos decidido utilizar y por qué hemos decidido dividir las clases de la manera en la que lo hemos hecho.

Para empezar, es importante aclarar que React es una biblioteca para la creación de interfaces, y por lo tanto no impone ningún patrón arquitectónico específico. Sin embargo hay una variedad de patrones [11] que se utilizan habitualmente y que se consideran buenas prácticas. A continuación comentaremos los patrones que hemos utilizado para definir la estructura de nuestra aplicación cliente.

- Patrón *composite*/composición sobre herencia [12]: Este patrón defiende la utilización de objetos simples para formar objetos más complejos. En nuestro caso hemos aplicado este patrón a los componentes. Un componente se ha creado en caso de ser reutilizable o ser una parte definida de un componente más grande, lo que provoca que los componentes grandes sean más simples de tratar y de mantener.
- Patrón de flujo unidireccional de los datos: Este patrón está implementado en React por defecto, y provoca que los datos fluyan en un solo sentido, desde los

componentes superiores a los inferiores, a través de *props*¹⁰ o propiedades. Esto ayuda a tener un flujo de datos más predecible y entendible.

- Patrón Observer [13]: Este patrón se implementa en React mediante los *hooks*¹¹. Los hooks son elementos que se utilizan principalmente en componentes funcionales¹² y que nos ayudan a gestionar el estado de un componente en React. Aunque estos hooks no son una implementación completa del patrón observador, si que se pueden ver como parte del ecosistema del patrón. Dos de los hooks que mejor ilustran esta similitud son los hook *use State* y *use Effect*.
- *Use State*: Es una función que te permite agregar estado local a un componente. Al cambiar el valor de un *use State*, el componente se renderiza, por lo que *use State* sería el elemento “observado” en el patrón observador.
- *Use Effect*: Este hook es una función a la que se le proporciona ciertas dependencias. Cuando una de estas dependencias cambia, ejecuta un conjunto de funciones o instrucciones que se le indiquen. En el patrón observador sería el elemento que se suscribe a las dependencias y observa los cambios en ellas.

Además de estos patrones, en la implementación del diagrama se ha propuesto seguir una arquitectura similar al patrón MVC [14]. Aunque React se centra principalmente en la V de MVC, es decir, en la vista, es posible hacer una analogía de cómo se implementaría el patrón MVC.

¹⁰ Objetos JavaScript que contienen información específica y se transmiten a un componente

¹¹ Función especial que permite agregar estados a componentes funcionales en React

¹² Funciones de React que hacen la labor de componentes, devolviendo código JSX y manteniendo su propio estado interno.

- **Modelo:** Son los datos de la aplicación, que se pueden encontrar principalmente en el estado de un componente, en forma de *use States* o *props*. También pueden encontrarse en el contexto de la aplicación en caso de que se necesite.
- **Vista:** Estos son los componentes de React, su función es la de describir la interfaz de usuario basándose en el modelo de datos. Cuando un modelo cambia, el componente se actualiza actualizando la vista.
- **Controlador:** Este es el más complicado de definir, ya que como React funciona con el flujo unidireccional de datos, como hemos explicado anteriormente, este controlador difiere del tipo de controlador del MVC tradicional. En nuestro caso el modelo se podría percibir como las funciones de eventos (*onClick*, *onChange*, etc) o los *use Effect*, es decir, los elementos que hacen que la vista se actualice cuando es necesario.

A continuación el diagrama de clases del cliente (Ilustración 8)

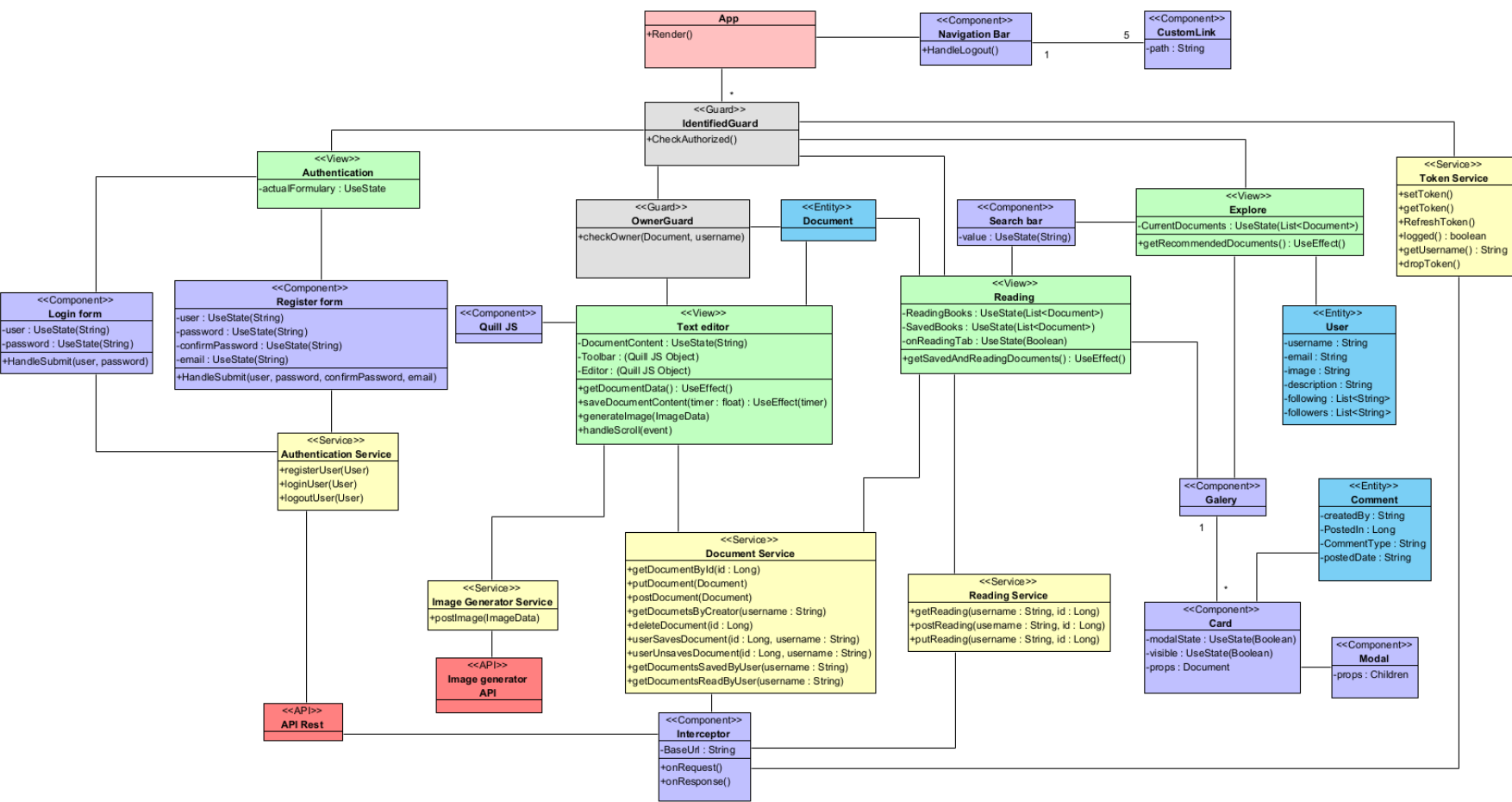


Ilustración 8. Diagrama de clases del cliente

En el diagrama de clases superior (Ilustración 8) se observan distintas clases con diferentes funcionalidades, empezaremos a comentarlas una a una y luego procederemos a un análisis general del diagrama.

- **Componentes:** Los componentes aparecen en morado bajo la etiqueta “*Component*”. Estos componentes son clases que se encargan de gestionar la vista y actualizarla mediante el controlador. Tienen funcionalidades definidas que los separan del resto de componentes y gestionan su propio estado. Muchas veces contienen a otros componentes o son contenidos por otros componentes, como se puede ver en el caso de “*navigation bar*” y “*Custom links*” o “*Galery*”, “*Card*” y “*Modal*”. Un ejemplo de estos componentes sería la barra de navegación, que tiene la función definida de permitir al usuario desplazarse entre las diferentes vistas. Tiene su propia interfaz visual y puede ser reutilizado cuantas veces sea necesario sin repetir código. Algunos componentes que no aparecen en el diagrama podrían ser “*Notification*”, el cual contiene la interfaz y lógica de una actividad realizada por un usuario, o “*Comment*”, el cual se encarga de mostrar los comentarios, incluyendo la lógica del tipo de usuario y otros factores.
- **Vistas:** Las vistas aparecen en verde bajo la etiqueta “*View*”. No confundir con la vista del patrón MVC. Estas vistas a efectos prácticos son componentes como cualquier otro, la única diferencia es que estos componentes no son componentes incluidos por otros componentes, si no que ellos incluyen componentes. Suelen tener una URL de la aplicación asignada. Un ejemplo que aparece en el diagrama podría ser la vista “*Text editor*”, la cual es la vista que incluye el editor de texto y otros componentes. Otras vistas que no aparecen en el diagrama podrían ser la vista Home o la vista Writting, las cuales se han omitido por el poco espacio y la similitud al resto de vistas.

- **Modelo:** El modelo aparece en azul bajo la etiqueta “*Entity*”. Estas clases contienen la información del modelo y de los datos que utilizarán los componentes para mostrar información y actualizar sus estados. Es posible que no sean definidas como una clase como tal a nivel de implementación ya que los props de los componentes no requieren de un tipado, por lo que no es necesario definir los elementos de un prop para que sea procesado. Aún así es importante recalcar en el diagrama qué componentes utilizan qué entidades. Algunas entidades que no aparecen en el diagrama podrían ser “*Actividad*”, que se relacionaría con “*Notification*” y “*Home*”, y que contendría información sobre la acción que realiza un usuario.
- **Servicios:** Los servicios aparecen bajo la etiqueta “*Service*” y en color amarillo. Los servicios contienen la funcionalidad necesaria para hacer peticiones al back-end para obtener información. Se relacionan con los componentes, los cuales les piden que hagan una o varias peticiones en específico. Una vez que reciben una respuesta con los datos que se requieren estos datos son transferidos a los componentes que actualizarán su estado o realizarán las funciones necesarias. Esto nos ayuda a separar responsabilidades y a mejorar la reutilización del código.
- **Guardianes:** Los guardianes aparecen en color gris bajo la etiqueta “*Guard*”. Los guardianes son componentes cuya función es la de comprobar que un usuario tiene acceso a una URL o vista en concreto. En el caso del diagrama se observan dos Guardianes, el que comprueba si un usuario está identificado y el que comprueba si un usuario es el dueño de un documento, el cual determinará si le da acceso a la edición en caso de que sí sea el dueño, o le rechaza el acceso en caso contrario. Estos Guardianes se relacionan con un

componente que muestra el error en caso de que se deniegue el acceso, pero que no aparecen en el diagrama por falta de espacio. Otros guardianes que no aparecen en el diagrama son los que comprueban que un documento es público para permitir su lectura, o el que comprueba que la URL existe y en caso contrario te redirige a un componente que te advierte de que la URL a la que intentas acceder no existe.

- **Interceptor:** El interceptor es un componente. Su función es la de interceptar todas las peticiones que realizan los servicios e insertar en ellas una identificación que el servidor necesita para autorizar las peticiones. Además se encarga de recoger esta identificación y enviarla a donde corresponda para ser almacenada. Nótese que este interceptor sólo se utiliza en las peticiones que se realizan a la API REST de la aplicación, en caso de hacer peticiones a APIs externas como se muestra en el diagrama en el caso del servicio generación de imagen, sus peticiones no pasarán por el interceptor. Las peticiones de registro e inicio de sesión que gestiona el servicio de autenticación tampoco pasarán por el interceptor puesto que con esas peticiones se obtendrá el identificador para la autenticación y antes de esas peticiones no se tiene almacenado ninguna clave de autorización.

Resumiendo, en el diagrama se podrían diferenciar tres partes principales, el Modelo siendo las clases etiquetadas como entidades, la Vista refiriéndose a los Componentes y el Controlador refiriéndonos a los servicios. Aunque ya sabemos que los componentes también incluyen elementos del controlador y del modelo, por lo que no sería un MVC puro. Otros aspectos a comentar del diagrama en general es que hay componentes cuyo uso y reutilización es muy alto, como podría ser el ejemplo de la modal, que en el diagrama aparece relacionada con otro único componente, pero que si tuviésemos acceso a un diagrama que incluyese todos los componentes y

vistas veríamos que la modal implementará y será implementada por muchos otros componentes.

3.3. Diagramas de secuencia

En este punto mostraremos y comentaremos algunos de los diagramas de secuencia más importantes para comprender el funcionamiento de ciertos aspectos de la aplicación. Los diagramas de secuencia son diagramas que nos ayudan a entender cómo los componentes de un sistema interactúan entre sí a lo largo del tiempo para completar una función concreta.

Decidir qué diagramas de secuencia son los más importantes o adecuados para mostrar es una decisión compleja, pero nos hemos decantado por mostrar los diagramas más representativos y de ámbitos diferentes, ya que los diagramas de un mismo ámbito suelen ser bastante similares.

En estos diagramas existirán dos partes diferenciadas, los elementos pertenecientes al cliente y los elementos pertenecientes al servidor. Los elementos del cliente se mostrarán en morado, mientras que los elementos del servidor se mostrarán en rojo.

Los procesos que se mostrarán en los diagramas de secuencia serán, el inicio de sesión, la edición del texto de un documento, la creación de un comentario de tipo respuesta y una búsqueda de documentos y usuarios.

3.3.1. Inicio de sesión

Para el inicio de sesión, optamos por diseñar e implementar un sistema basado en JWT, el cual es un sistema que se utiliza para transmitir información entre partes de forma segura, en este caso entre el cliente y el servidor. Su funcionamiento se basa en el envío de un token único y con una vida definida, el cual ayuda a identificar a un usuario. Este token se genera en el servidor, y es utilizado por este para saber si una operación está autenticada o no. Además el token se almacena en el cliente para

poder insertarse en las peticiones, además de para identificar al usuario en la parte del cliente, puesto que el token contiene información sobre el usuario identificado.

En el diagrama a continuación se mostrará cómo el usuario inicia sesión desde el cliente, mediante la vista de inicio de sesión. Esta vista tiene un componente asociado, el formulario de inicio de sesión, el cual se encarga de almacenar los datos del usuario y enviarlos al servicio de autenticación. Este servicio envía los datos en una petición POST al controlador Rest del servidor. Estos datos se envían y reciben en forma de un DTO con el nombre de usuario y contraseña del usuario.

Una vez la petición ha llegado al servidor, este comprueba que las credenciales sean válidas. Esto lo hará mediante el gestor de autenticación, el cual mediante el servicio de detalles de usuario y el repositorio usuario comprobará que las credenciales introducidas estén vinculadas a un usuario. En caso de que esto sea así, se creará un token que será devuelto hasta el mismo controlador Rest, que enviará una respuesta HTTP al cliente que incluirá el token en su cabecera. Este token se almacenará en el cliente en el Servicio de Token, el cual almacena el token en el Local Storage. Acto seguido se dará acceso al usuario a la vista Home.

En caso de que el gestor de autenticación no consiga validar los datos del usuario, no se generará ningún token, la respuesta HTTP será de tipo Unauthorized, es decir, no autorizada, y se mostrará en la vista de inicio de sesión un mensaje comunicando que la validación del inicio no ha sido correcta. A continuación, el diagrama que ilustra todo el funcionamiento comentado (Ilustración 9):

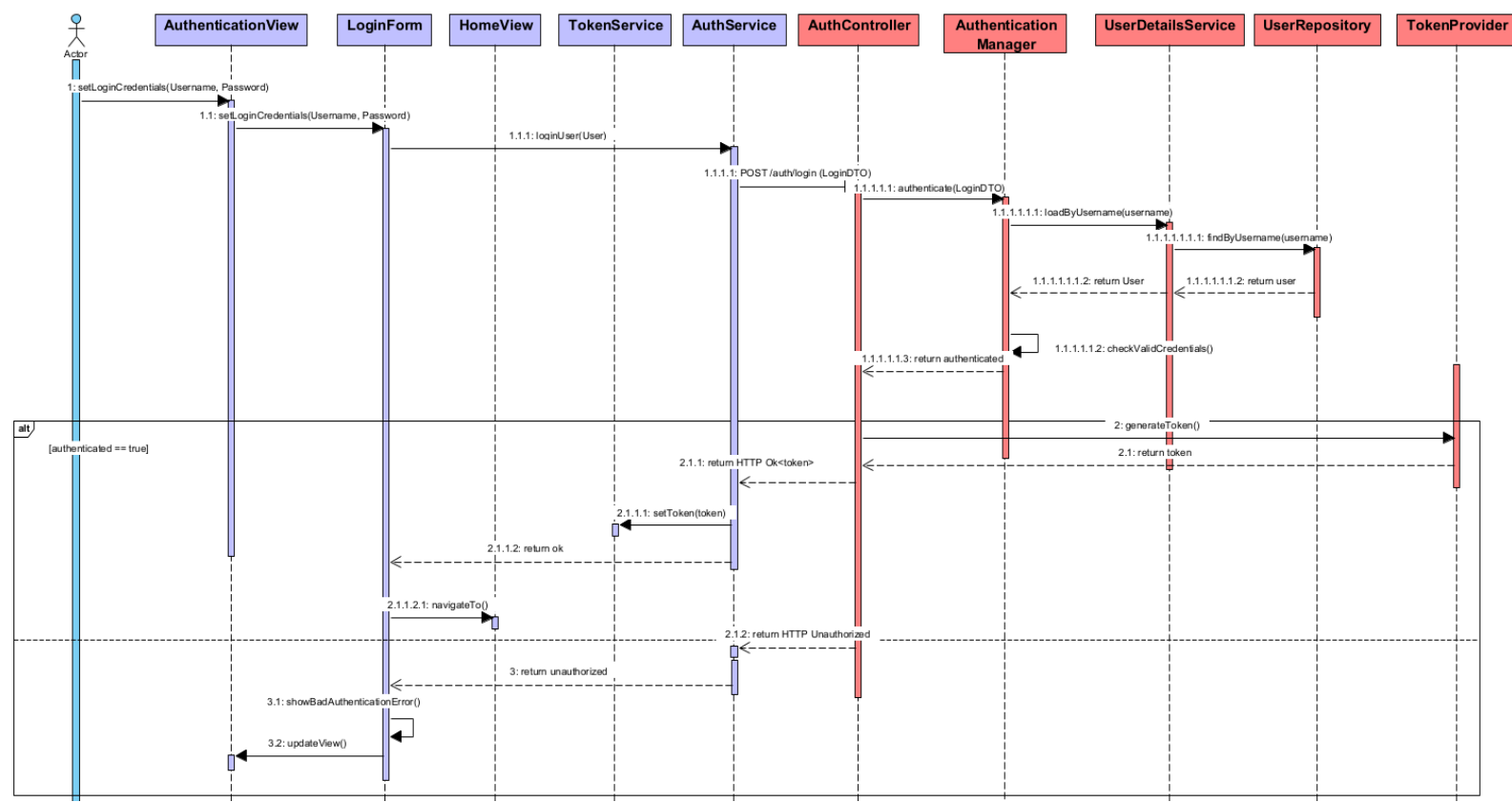


Ilustración 9. Diagrama de secuencia inicio de sesión

3.3.2. Edición del texto de un documento

El siguiente diagrama del que hablaremos mostrará la interacción entre distintos elementos de la aplicación a la hora de modificar un texto relativo a un documento. Es importante recalcar que esta edición no se refiere a la edición de parámetros como el título, sinopsis y portada de un documento, ya que esa funcionalidad se implementará de forma diferente a esta. Este diagrama se centra en la edición del texto del documento mediante el editor de texto que la aplicación incluye.

En primer lugar el usuario selecciona que quiere editar el texto de un documento de su creación en la interfaz. Esta opción sólo aparecerá en los documentos que él ha creado.

Acto seguido “*document card*”, que es la previsualización del documento donde aparece la opción redirigirá al usuario a un guardián. Este guardián comprobará si el usuario es el creador del documento. Como hemos comentado antes, la opción de edición del documento sólo aparecerá si eres el creador del mismo, por lo que podría parecer que este Guard es innecesario, pero su función principal es la de limitar el acceso mediante la URL, ya que cualquier usuario podría colocar la URL en su buscador e intentar acceder al documento.

Este guardián se encarga de realizar una petición en la que se comprobará que el usuario que ha iniciado sesión en el servidor y que está asociado al token, es decir, que hace la petición, es el autor del documento a editar.

Esta petición llega hasta el servidor, el cual devolverá un booleano indicando si el libro es de la autoría del usuario logueado o no. Se ha decidido devolver un booleano y no el documento completo, ya que devolver el documento completo sin saber si ese usuario tiene acceso a él o no, podría provocar que información sensible pueda ser

obtenida manipulando el cliente. Por lo cual, primero se devuelve el bool, y si él es autor del documento, se pedirá el documento.

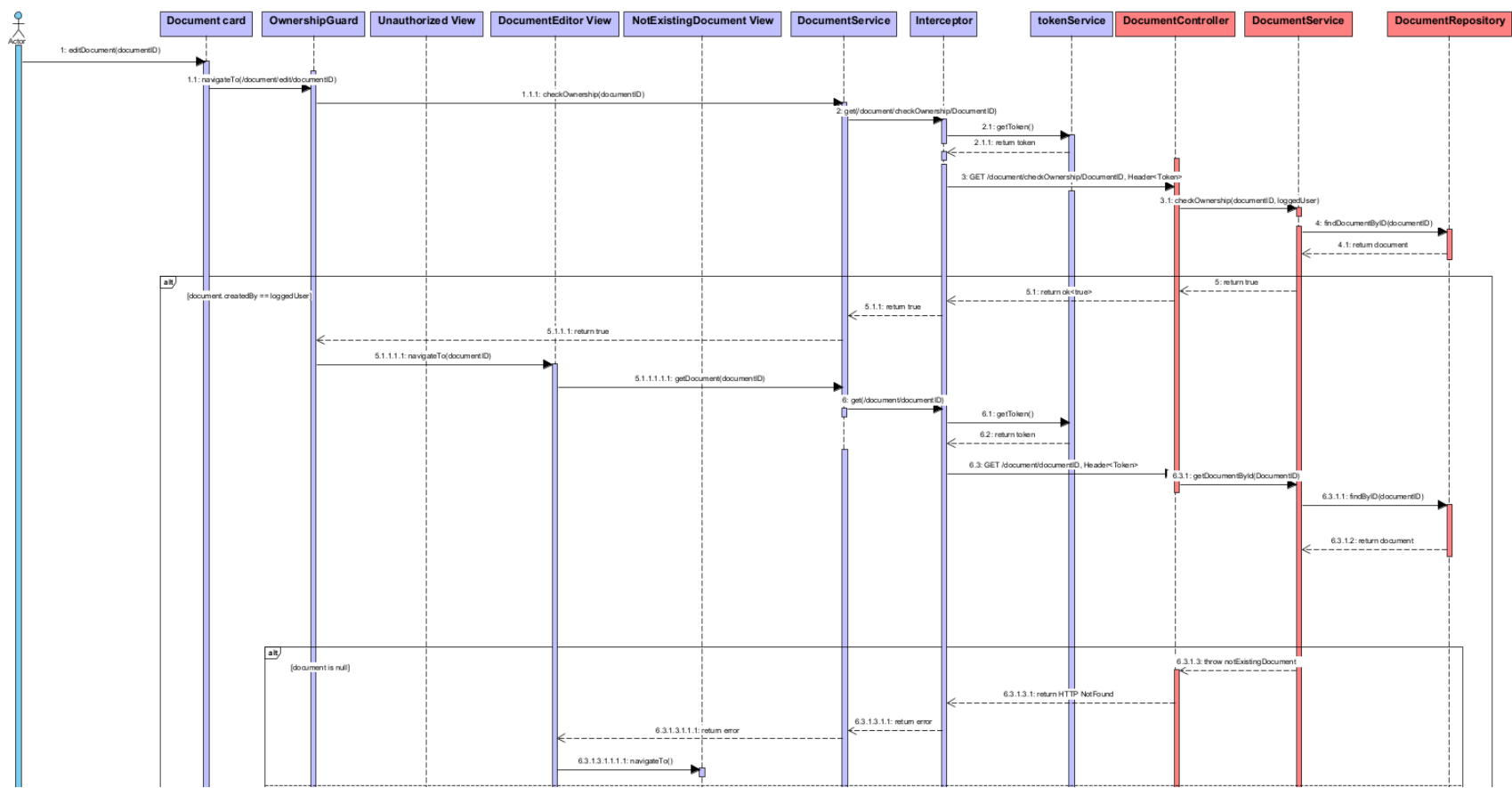
Una vez que se ha recibido respuesta, si la respuesta es negativa, es decir, el usuario no es creador del documento, el guardián redirigirá al usuario a una vista que le indica que no está autorizado a editar ese documento.

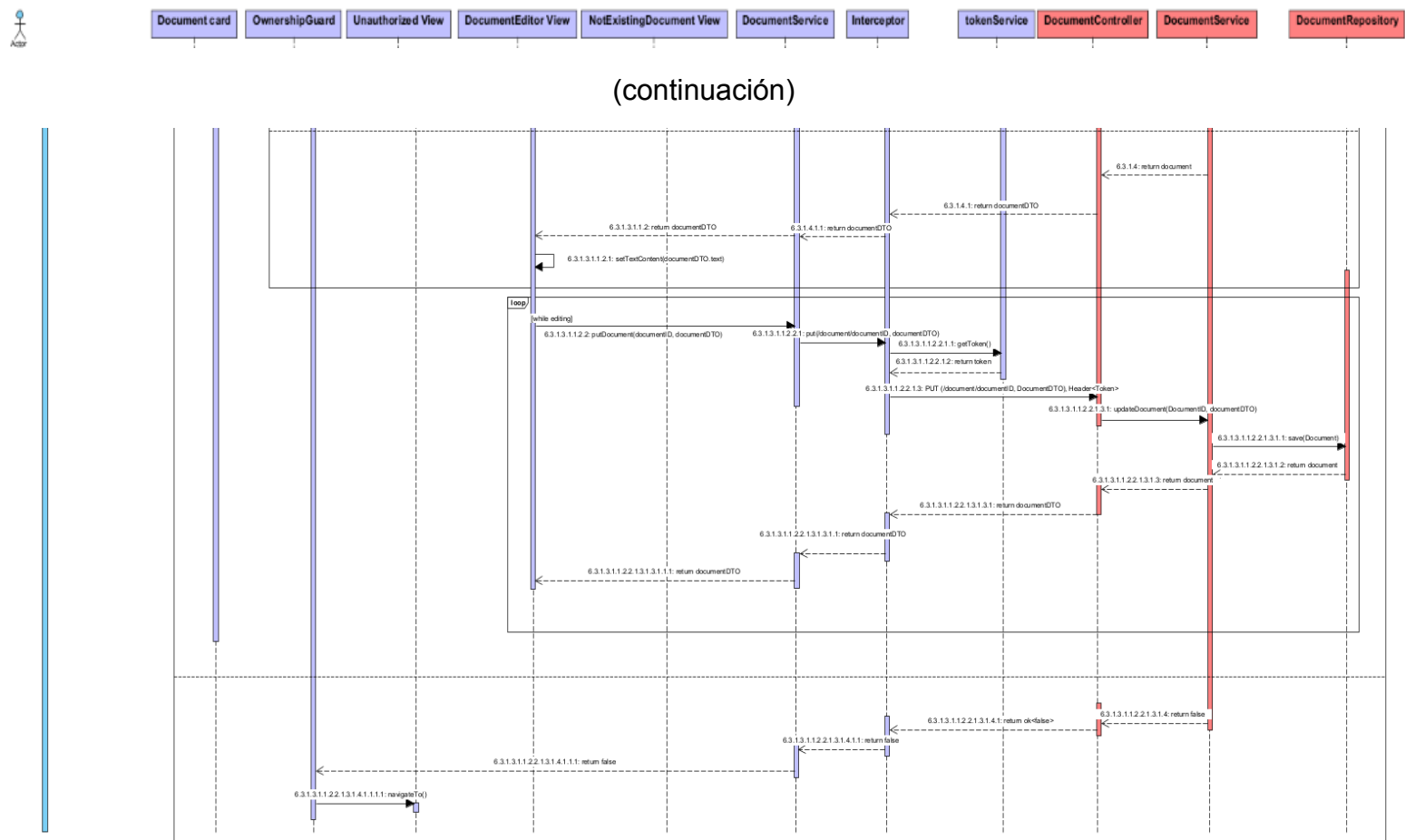
En caso de que la respuesta sea afirmativa, se redirigirá a la vista de edición y se pedirá al servidor los datos actuales del documento, para así rellenar el editor con el texto actual. En caso de no encontrar el documento, se devolverá una respuesta “*Not Found*” por parte del servidor, y se redirigirá a una vista que indica que el documento no existe. Esta comprobación de existencia se realiza en todas las operaciones que realiza el servidor, sólo aparece aquí para ahorrar espacio en los diagramas.

Si el documento existe se reciben los datos desde el servidor en forma de DTO y se coloca el contenido del documento en el editor de texto. Luego, cada cierto tiempo, mientras el usuario está editando, se envía al servidor una petición PUT con los cambios realizados. Esta petición devolverá un DTO que se utilizará para comprobar en el cliente que los cambios se han realizado correctamente en el back-end.

Como hemos comentado anteriormente, todas las peticiones al servidor en relación a los documentos, pasan por el servicio documento, el cual es responsable de realizar las peticiones al servidor en cuestión de documentos. Todas estas peticiones son enviadas a través del interceptor, el cual pide al servicio de tokens el token perteneciente al usuario logueado y lo inserta en la cabecera de la petición. Es importante recalcar dos casuísticas del interceptor que no aparecen en los diagramas:

- **Refresco del token:** Se puede dar el caso de que un token válido haya caducado. En ese caso el servidor responderá con código HTTP “No autorizado”, y no realizará la petición. En este caso, el interceptor es el encargado de pedir un nuevo token mediante el controlador REST “Auth”, en la url “/refresh”. Una vez el servidor devuelva un nuevo token, el interceptor lo insertará en la petición anteriormente rechazada para que ésta se efectúe.
- **Token no válido:** En caso de que el servidor responda con código HTTP “No autorizado” a una petición, se intente refrescar el token y este devuelve un error, el cliente interpreta que el token que está utilizando no es válido y por lo tanto redirige al usuario a la vista de inicio de sesión.





3.3.3. Creación de un comentario

Este diagrama mostrará cómo un usuario crea un comentario de tipo reseña, y como interaccionan los elementos involucrados en esta creación.

Para la creación de una reseña es importante recalcar que sólo puede existir una reseña en un documento de parte de un mismo usuario. Por lo tanto el servidor deberá controlar el caso en el que un usuario intente crear una reseña en un documento en el que ya hizo una reseña. La interfaz de la aplicación también se asegura de que esto no ocurra, ya que, si existe ya una reseña del usuario esta reseña sustituirá al componente que crea las reseñas, por lo que el usuario no tendrá acceso a él si ya ha creado una reseña, a menos que la elimine o edite, caso que no es el que estamos comentando puesto que este diagrama muestra exclusivamente la creación de una reseña.

Como vemos en el diagrama, el usuario indica que ha creado un comentario, de tipo reseña y con cierta puntuación en el componente *“Comment Posting”*. Este componente comunica al servicio comentario que debe hacer una petición al servidor para la creación del comentario. Como ya sabemos, los servicios, y en este caso el servicio comentario, se encarga de hacer las llamadas a la API REST en el ámbito de los comentarios. Haciendo uso del interceptor como en los casos anteriores, se identifica la cabecera de la petición con el token para que sea aceptada por el gestor de autorizaciones.

Una vez en el servidor, el controlador REST recibe la petición y llamará al servicio comentario. Este servicio creará el comentario en función del campo *“Comment Type”* del comentario DTO. Antes de la creación, al ser el comentario de tipo reseña, el servicio sabe que este tipo de comentarios es único por usuario y documento, por lo

que hace una búsqueda para saber si ya existe una reseña en el documento seleccionado y por el usuario que lo crea. En caso de que ya exista, el servidor lanzará una excepción y el controlador devolverá una respuesta HTTP Conflict. El cliente al recibir esta respuesta desactiva el componente de creación de comentarios, puesto que no debería de haber estado desde un primer momento, y si está ahí es por algún fallo no controlado. Este caso no debería ocurrir nunca, aún así existe este plan de contingencia en caso de que exista un fallo no detectado.

En caso de que no exista la reseña, esta se crea y es devuelta tanto por el servicio como el controlador, y en el cliente se sustituye el componente de creación de comentarios por la nueva reseña, permitiendo que el usuario vea su reseña en el lugar donde acaba de escribir y enviar la reseña y previniendo que pueda volver a crear otra reseña.

A continuación, el diagrama que contiene toda esta información (Ilustración 11).

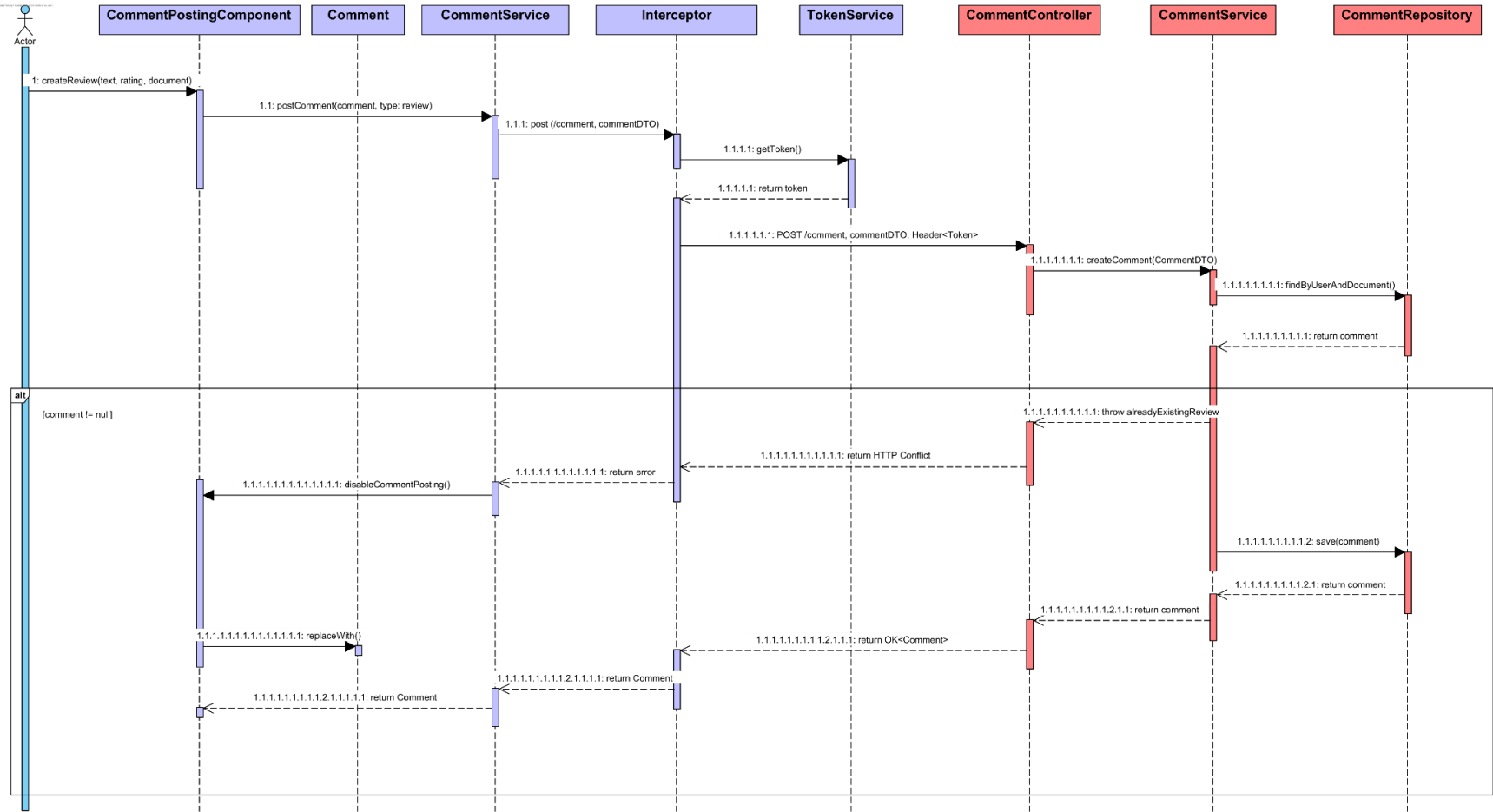


Ilustración 11. Diagrama de secuencia creación de comentario.

3.3.4. Búsqueda de documentos y usuarios

En este diagrama vemos como un usuario puede hacer uso de la vista de exploración para buscar un documento por géneros y por título, o buscar un usuario.

Existirán dos componentes para la búsqueda, el selector de géneros, el cual nos permitirá seleccionar los distintos géneros que deben tener los documentos que busquemos, y una barra de búsqueda que nos permite buscar documentos que incluyan el texto de la barra en su título o los usuarios que contengan el texto en su nombre.

La interacción entre géneros será de tipo “OR”, es decir, si se selecciona ciencia ficción y fantasía como géneros, se buscarán documentos de ciencia ficción, de fantasía o de ciencia ficción y fantasía. En el caso de la barra de búsqueda, su relación con los géneros es de tipo “AND”, por lo que en caso de incluir texto en la barra de búsqueda y seleccionar géneros, los documentos resultantes de la búsqueda serán los que complan con la condición de los géneros explicada anteriormente pero que tengan en su título parte del texto introducido en la barra de búsqueda.

Además se debe recalcar que se implementará paginado en cuestión de los documentos que se piden al servidor, por lo que las peticiones deberán incluir la página y el número de documentos que se están pidiendo. Esta petición se realizará cuando el usuario haya llegado al final de la ventana, es decir, ya no quedan más documentos que mostrar en el cliente.

En el diagrama de secuencia sólo se mostrarán las interacciones de exploración relativas a los documentos, puesto que si añadimos las interacciones para la búsqueda de usuarios el diagrama sería muy extenso.

A continuación se muestran las peticiones necesarias para que todo funcione correctamente. En caso de que el usuario llegue al final del scroll, es decir, a lo más bajo de la ventana, se volverá a realizar una petición del mismo tipo al servidor pero con un número de página incrementado en uno. Este comportamiento no se aprecia en el diagrama puesto que se carece de espacio para incluirlo, pero las peticiones que realizaría serían las mismas que se muestran en el diagrama cuando un usuario realiza la búsqueda. Al realizar una búsqueda, la página siempre comienza con valor cero.

Como se observa en el diagrama (Ilustración 12), para buscar los documentos de cierto género utilizamos el repositorio de los géneros en lugar del repositorio documentos, ya que al contener los géneros una relación con todos los documentos de ese género, la búsqueda de los documentos en ese repositorio es mucho más rápida y eficiente.

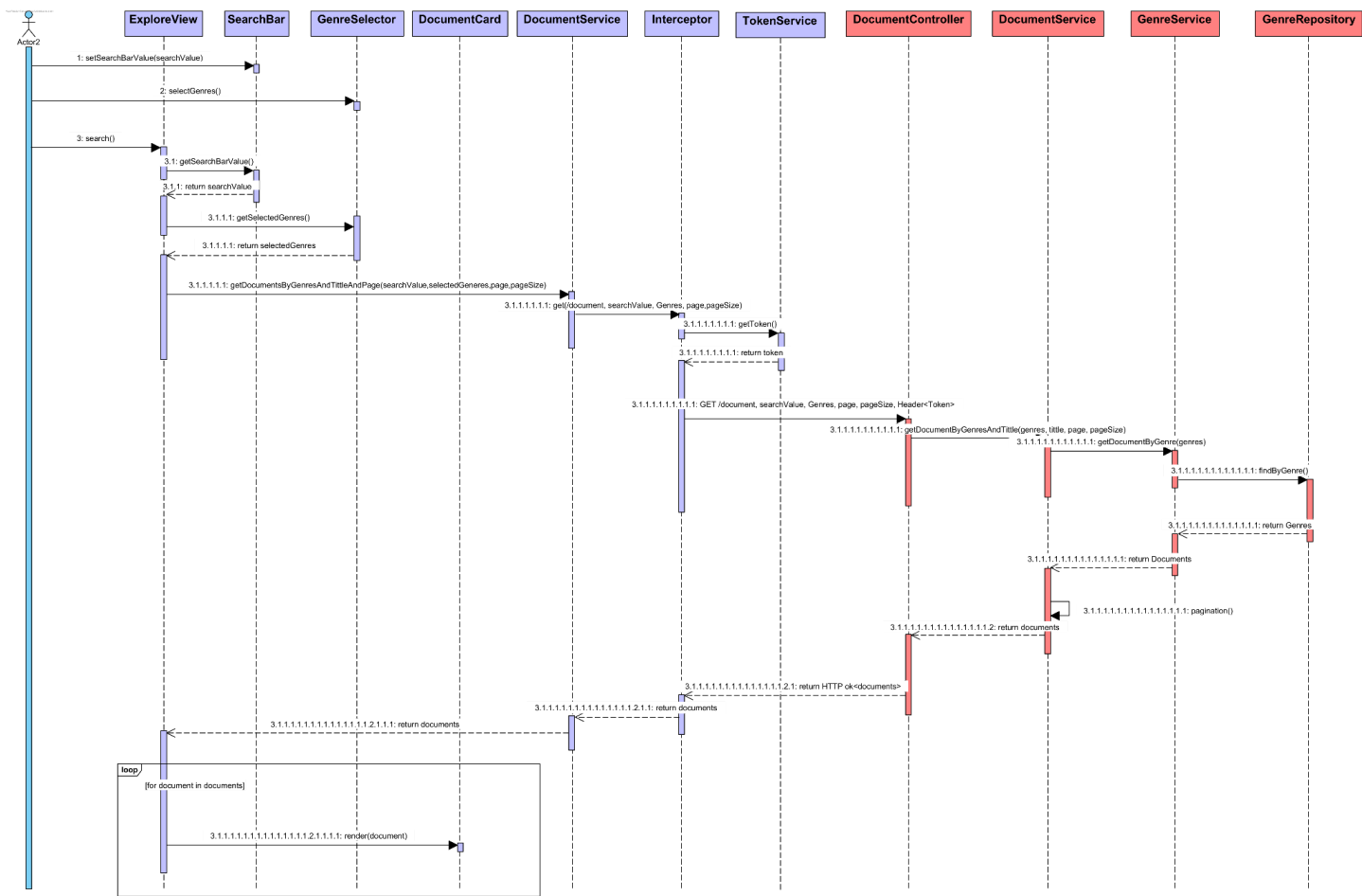


Ilustración 12. Diagrama de secuencia búsqueda documento.

3.4. Diseño de la Interfaz

La interfaz de la aplicación está formada por aquellos elementos con los que interaccionan agentes externos a la aplicación.

3.4.1. Diseño de la API REST

Para facilitar la comprensión del diseño de la API REST, se ha decidido utilizar una herramienta conocida como *Swagger*, basada en la especificación *OpenAPI*. Esta es una herramienta de código abierto que se utiliza para diseñar, construir y documentar APIs RESTful. En nuestro caso, la utilizaremos para documentar la API y realizar pruebas sobre ella.

Centrándonos en el ámbito de la documentación, este tipo de herramienta nos muestra información sobre:

- Rutas de acceso a la API o “*endpoints*”
- Operaciones disponibles, como GET, POST, PUT, etc.
- Parámetros de entrada y salida y su formato.
- Formato de los datos.
- Códigos de respuesta y de error.

Como hemos adelantado anteriormente, la característica con más utilidad de Swagger es la posibilidad de que los usuarios puedan visualizar y probar la API sin necesidad de escribir líneas de código.

Esta documentación en formato yaml se genera en una URL concreta. En nuestro caso, es accesible desde “<http://localhost:8080/swagger-ui/index.html#/>”, siendo localhost:8080 la raíz de la api.

Una vez hemos accedido a esta URL con la API REST desplegada, tendremos acceso al documento, en el cual se muestra una lista con los endpoints de la aplicación, agrupados por controlador, como se muestra en la ilustración 13.

reading-controller			^
PUT	/readings	Actualiza lectura	▼
POST	/readings	Crear lectura	▼
GET	/readings/get/{username}/{documentID}	Obtener lectura	▼
GET	/readings/check/{username}/{documentID}	Comprobar existencia lectura	▼
document-controller			^
GET	/documents/{id}	Obtener documento mediante ID	▼
PUT	/documents/{id}	Actualizar documento	▼
POST	/documents	Crea un documento	▼
GET	/documents/saved/{username}	Obtener documentos guardados	▼
POST	/documents/saved/{username}	Guarda un documento para un usuario	▼
DELETE	/documents/saved/{username}	Desmarcar documento guardado	▼
PATCH	/documents/{documentId}/visibility	Cambiar visibilidad	▼
GET	/documents/recommendation	Obtener recomendación global	▼
GET	/documents/read/{username}	Obtener documentos en lectura	▼
GET	/documents/public/{username}	Obtener documentos publicos de un usuario	▼
GET	/documents/private/{username}	Obtener documentos privados de un usuario	▼
GET	/documents/genres	Obtener documentos por género y página	▼
GET	/documents/checkPublic/{documentId}	Verificar si un documento es público	▼
GET	/documents/checkOwner/{documentId}	Verificar propietario del documento	▼
DELETE	/documents/{documentId}	Eliminar documento	▼
comment-controller			^
PUT	/comments/{commentID}	Actualizar comentario	▼

Ilustración 13. Fragmento de documentación swagger.

Como vemos en la ilustración 13, las rutas de acceso se muestran junto al tipo de método HTTP asociado y junto a una breve cita que explica la operación que se realiza de forma resumida.

Si decidimos desplegar uno de estos endpoints, tendremos acceso a una descripción más detallada de la operación que realiza el método, además de información sobre qué parámetros requiere y sobre los códigos de respuesta HTTP que devuelve, además del tipo de dato de retorno. Todo esto es visible en la ilustración 14, donde se observa el método que comprueba la existencia de un nombre de usuario en el servidor. Este devolverá un booleano indicando si el nombre de usuario existe o no en el servidor y requiere como parámetro el nombre a comprobar

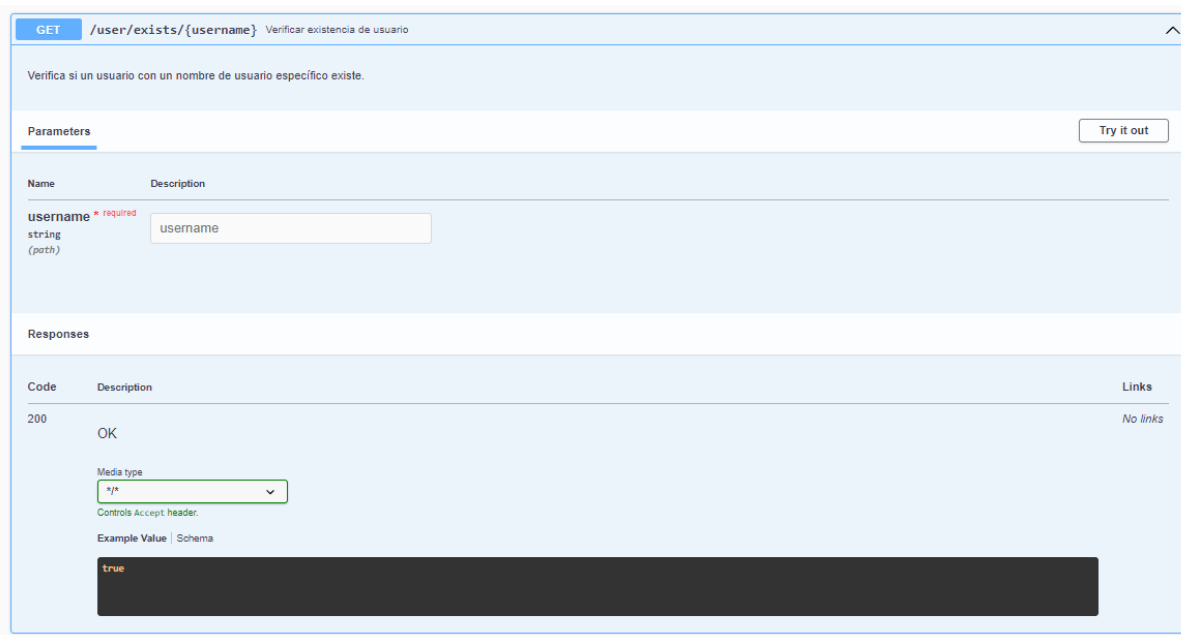


Ilustración 14. Información de método desplegado en Swagger

En caso de que se requiere como parámetro un objeto, este se mostrará en la interfaz en formato JSON, como se muestra en la ilustración siguiente (Ilustración 15), donde se observa la información referente al método de registro del controlador de autenticación, método el cual requiere como parámetro un objeto de tipo registro DTO.

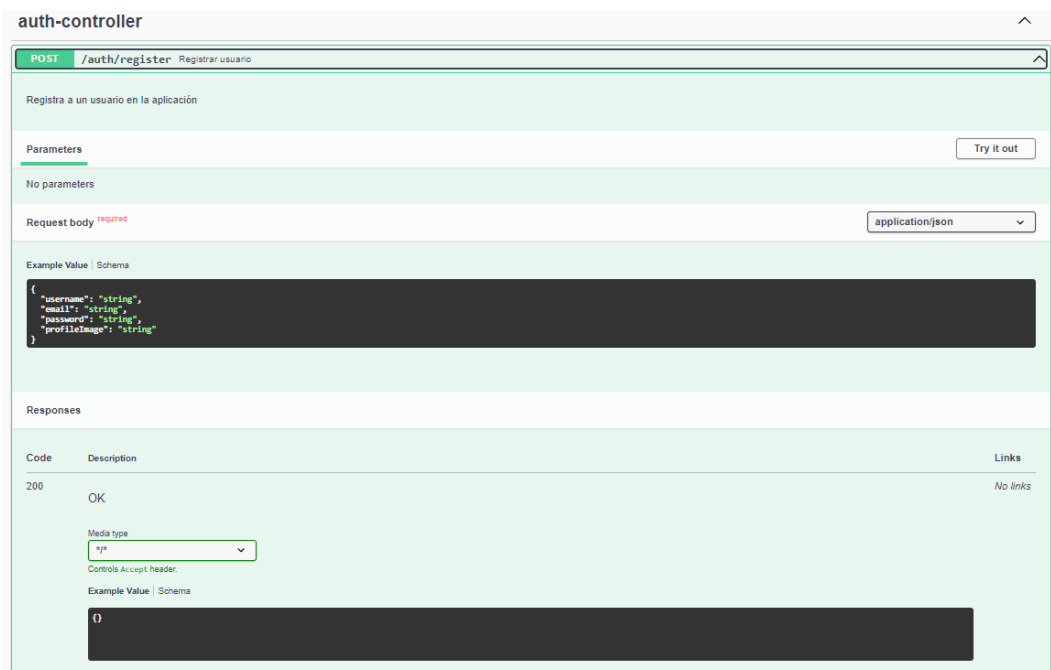


Ilustración 15. Método con parámetro de objeto en Swagger

Como vemos, cada método tiene asociada una URL o endpoint. Este entry point debe ser único para cada tipo de método HTTP, es decir, para una URL concreta sólo puede haber un método GET, POST, PUT, etc.

Los parámetros de estos métodos pueden estar incluidos en la URL, como es el caso del método mostrado en la ilustración 14 o pueden ser incluidos también en el body, o cuerpo de la petición, como es el caso de la petición en la ilustración 15. Además también pueden incluirse como parámetros de la consulta.

Además de los métodos HTTP, Swagger también incluye información sobre las entidades y DTOs que conforman el modelo de la API REST. En la ilustración 16 se observan algunas de estas clases. Al desplegarlas podemos observar sus atributos y los tipos de estos atributos.



Ilustración 16. Modelo de la API en Swagger

A continuación, una tabla (Tabla 7) que muestra información sobre los controladores implementados, en la que se mostrará información sobre qué controladores existen, una descripción sobre su funcionamiento, las entidades con la que trabaja y cuántos métodos HTTP contiene.

Controlador	Descripción	Entidades	Métodos HTTP
<i>User Controller</i>	Gestiona las peticiones relativas a los usuarios, como edición o peticiones de usuarios concretos.	Usuario	9
<i>Document Controller</i>	Gestiona las peticiones a documentos, incluyendo las	Documento, Género	15

	operaciones básicas y multitud de peticiones más complejas		
<i>Comment Controller</i>	Gestiona las peticiones sobre comentarios y sus distintos tipos.	Comentario, Reseña, Respuesta	8
<i>Reading Controller</i>	Se encarga de gestionar las peticiones de operaciones sobre las lecturas	Lectura	4
<i>Activity Controller</i>	Contiene un único método que devuelve la actividad de los usuarios incluidos en la petición.	Evento de Actividad	1
<i>Auth Controller</i>	Se encarga de gestionar las peticiones de autenticación como el registro, login o refresco del token JWT	Usuario	3

Tabla 7. Resumen contenidos de los controladores

Para comprender la filosofía de diseño seguida para cada controlador presentado, haremos una pequeña explicación basándonos en el controlador de documentos. Para una explicación más detallada sobre el funcionamiento de otros controladores se puede consultar el Apéndice I.

El controlador de documentos es el controlador más amplio debido a que los documentos son el eje central de la aplicación que estamos desarrollando. El controlador consta de numerosos métodos GET, un Método POST que crea el documento, y otro método POST que crea una relación entre documento y usuario,

siendo esta relación la relación de guardado. Existen de la misma manera dos métodos DELETE, uno que borra un documento y otro que elimina esta relación de guardado comentada anteriormente. El controlador también contiene un método PATCH. Este tipo de método ha sido utilizado para referenciar actualizaciones parciales de una entidad, en este caso de la entidad documento. La actualización que se realiza en este método es la actualización de la visibilidad del documento. Respecto a los métodos GET todos, a excepción del GET de obtención mediante géneros, siguen el mismo esquema anterior, se indica mediante un parámetro de ruta y la ruta específica qué documento se debe buscar y en función de qué criterio. El controlador se observa en la Ilustración 17.

document-controller			^
GET	/documents/{id}	Obtener documento mediante ID	▼
PUT	/documents/{id}	Actualizar documento	▼
POST	/documents	Crea un documento	▼
GET	/documents/saved/{username}	Obtener documentos guardados	▼
POST	/documents/saved/{username}	Guarda un documento para un usuario	▼
DELETE	/documents/saved/{username}	Desmarcar documento guardado	▼
PATCH	/documents/{documentId}/visibility	Cambiar visibilidad	▼
GET	/documents/recommendation	Obtener recomendación global	▼
GET	/documents/read/{username}	Obtener documentos en lectura	▼
GET	/documents/public/{username}	Obtener documentos publicos de un usuario	▼
GET	/documents/private/{username}	Obtener documentos privados de un usuario	▼
GET	/documents/genres	Obtener documentos por género y página	▼
GET	/documents/checkPublic/{documentId}	Verificar si un documento es público	▼
GET	/documents/checkOwner/{documentId}	Verificar propietario del documento	▼
DELETE	/documents/{documentId}	Eliminar documento	▼

Ilustración 17. Controlador documento en documentación Swagger

Los métodos que comentaremos en este controlador serán el método PATCH y el método GET a “/documents/genres”.

Como vemos en la ilustración 18, el método PATCH recibe por parámetro de tipo ruta un ID de tipo entero, el cual se corresponde con el documento del que se quiere cambiar la visibilidad. Una vez la visibilidad haya sido cambiada, se devuelve un DTO con la información del documento en cuestión y con la visibilidad de este actualizada. La respuesta HTTP será de tipo OK. En caso de que el documento no sea encontrado, se lanzará una excepción que resultará en una respuesta de tipo NOT_FOUND.

PATCH /documents/{documentId}/visibility Cambiar visibilidad

El propietario del documento puede cambiar su visibilidad.

Parameters

Name	Description
documentId * required integer(int64) (path)	documentId

Responses

Code	Description	Links
200	Visibilidad cambiada correctamente	No links
404	Documento no encontrado	No links

Example Value

```
{
  "id": 0,
  "title": "string",
  "cover": "string",
  "text": "string",
  "creatorUsername": "string",
  "synopsis": "string",
  "genres": [
    "string"
  ],
  "savedBy": [
    "string"
  ],
  "readings": [
    {
      "username": "string",
      "document": 0,
      "readingSpot": 0
    }
  ],
  "rating": 0,
  "public": true
}
```

Ilustración 18. Método PATCH Controlador documentos

En la siguiente ilustración (Ilustración 19), se muestra el método GET adjunto a la dirección `/documents/genres`. En la ilustración se muestra la descripción del método y los parámetros a introducir. He decidido recalcar este método debido a que los parámetros que se deben introducir son de tipo Query. Estos parámetros en la interfaz

de Swagger se introducen como si se trataran de un parámetro de ruta, sin embargo a nivel de petición desde el cliente la forma de enviar el parámetro difiere de la de los parámetros de ruta. Esta petición, como la mayoría de las peticiones devolverá el código HTTP OK en caso de que todo funcione según lo esperado y el código NOT_FOUND en caso de que uno de los géneros de los que se buscan documentos no exista. La respuesta de la petición será una lista de los documentos que cumplan las condiciones que se indican. Además, en este caso se incluyen parámetros como la página o el tamaño de la página puesto que esta petición incluye paginado y devolverá un conjunto distinto de documentos según la página y el tamaño de página que se indique. Dado un conjunto total de documentos y un tamaño de página concreto, la petición con un número de página específico siempre devolverá los mismos documentos.

GET /documents/genres Obtener documentos por género y página

Obtiene los documentos públicos basados en los géneros proporcionados y el fragmento del título.

Parameters

Name	Description
genres * required array[string] (query)	string string string Add string item
page * required integer(\$int32) (query)	page
pageSize * required integer(\$int32) (query)	pageSize
titleFragment * required string (query)	titleFragment

Execute

Responses

Code	Description	Links
200	Documentos devueltos correctamente	No links
404	Genero no encontrado	No links

Media type:

Controls Accept header:

Example Value | Schema

```
{
  "id": 0,
  "title": "string",
  "cover": "string",
  "text": "string",
  "creatorUsername": "string",
  "synopsis": "string",
  "genres": [
    "string"
  ],
  "savedBy": [
    "string"
  ],
  "readings": [
    {
      "username": "string",
      "document": 0,
      "readingSpot": 0
    }
  ],
  "rating": 0,
  "public": true
}
```

Ilustración 19. Método GET /document/genres del controlador documento

3.4.2. Diseño Interfaz del cliente

En el ámbito del diseño de la interfaz del cliente, primero comentaremos del diseño general, mostrando un diagrama jerárquico entre las vistas para indicar cómo llegar a cualquier pantalla desde la vista de inicio. Luego, entraremos en detalle sobre qué

contiene cada vista y cual es su uso, además de mostrar el diseño concreto de algunas de las vistas más importantes. Anteriormente, comentamos que nuestra aplicación será *responsive*¹³, por lo tanto para muchas de las vistas habrá que realizar dos diseños, uno que se aplicará en pantallas con resolución alta, principalmente ordenadores, y otra que se utilizará en dispositivos de menor tamaño, como móviles o tabletas [15].

El hacer las vistas *responsive* tiene también sus desventajas, ya que si se sobrecarga de elementos la vista de computador, estos elementos no tendrán espacio en la vista móvil debido a la reducción de tamaño, por lo tanto habrá que organizar los elementos para que estén bien distribuidos e incluso eliminar algunos elementos secundarios en vistas móviles.

A continuación se muestra el diagrama jerárquico (Ilustración 20). Este tipo de diagrama se utiliza para mostrar relaciones de orden del tipo arriba/abajo o superior/inferior. Este tipo de diagramas contienen dos tipos de elementos principales, los nodos y las relaciones. En nuestro caso los nodos serán las vistas o componentes que tienen una funcionalidad concreta y suficientemente separada del resto para considerarse un elemento a parte. Las relaciones se representan mediante una flecha, la cual indica que el elemento al que apunta la flecha es accesible desde el elemento del que sale la flecha, lo que nos ayuda a poder llegar a cualquier forma de la aplicación de forma guiada observando el diagrama.

¹³ Aplicación responsive: Aplicación que ha sido diseñada para adaptarse a las diferentes pantallas en las que se utiliza.

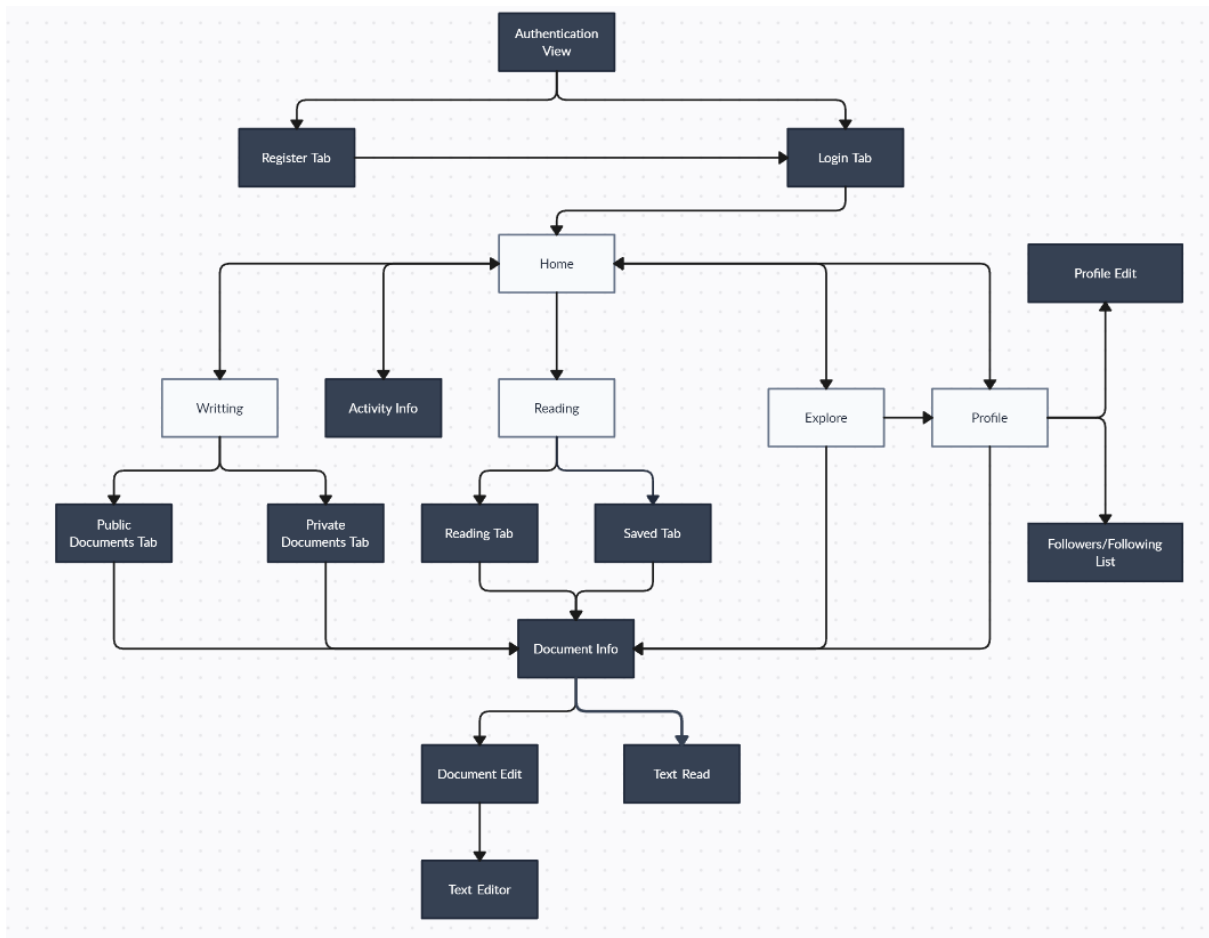


Ilustración 20. Diagrama jerárquico de la interfaz del cliente

En el diagrama anterior (Ilustración 20) se muestran los diferentes nodos y relaciones comentadas anteriormente. El color de los nodos de color blanco indica que son accesibles desde cualquier punto de la aplicación, puesto que es probable que estén en la barra de navegación. Comentaremos ahora cada nodo dando una pequeña descripción de cuáles son sus funcionalidades y por qué son un elemento diferenciado.

- **Authentication view:** Es la vista de la autenticación, contiene dos pestañas, una para el registro y otra para el inicio de sesión. En caso de utilizar el registro, se

redirigirá al inicio de sesión y una vez que el usuario inicie sesión será enviado a la vista principal o Home.

- *Home*: Es la vista principal, aquí se mostrarán las distintas notificaciones de usuarios a los que sigues. En caso de querer más información sobre una notificación, podrás pulsar sobre una de ellas, lo que abrirá una modal con la información de la notificación, lo que se representa en el diagrama como “*Activity Info*”. Todas las modales pueden cerrarse para volver a la vista en la que fueron abiertas.
- *Writing*: Esta es la vista de escritura, aquí se mostrarán los documentos que has escrito. Incluye dos pestañas, una de documentos privados, para acceder a los documentos con visibilidad privada y otra de documentos públicos, para acceder a tus documentos públicos de tu creación.
- *Reading*: Esta vista contiene los documentos que estás leyendo o quieres leer. Se divide en dos pestañas, una pestaña “*Reading tab*”, en la cual se muestran los documentos que estás leyendo y “*Saved tab*”, en la que se muestran los documentos que has guardado para leer más tarde.
- *Profile*: Esta vista es el perfil de un usuario. Aquí se puede acceder a la modal de edición de perfil, en caso de que sea tu perfil o se puede acceder a la lista de seguidores y seguidos, esta opción en cualquier perfil, tuyo o de otro usuario. También se muestran los documentos públicos que haya creado el usuario dueño del perfil.
- *Explore*: Aquí se permitirá al usuario buscar documentos y usuarios.

- *Document Info*: Todas las vistas anteriormente mencionadas, a excepción de la vista de registro e inicio de sesión, muestran documentos de alguna forma. Estos documentos pueden ser pulsados para mostrar una modal, la cual contendrá información sobre el documento y dará acceso a funcionalidades como la de lectura, o edición del documento. Esta modal es la modal de información del documento o “document info”.
- *Document edit*: Una modal que será accesible desde la información del documento siempre y cuando el usuario sea creador del documento en cuestión. Permitirá editar atributos del documento y dará acceso a la edición del contenido del mismo.
- *Text editor*: Es el editor de texto para editar el contenido del documento.
- *Text read*: Es el contenido del documento sin herramientas de edición para que los usuarios puedan leerlo. Siempre se podrá acceder si el documento es público y es accedido desde la información del documento.

A continuación mostraremos algunos wireframes de algunas de las vistas y elementos principales. Un wireframe, como se explica en [16] es una ilustración en colores grises que representa la estructura de una vista en una página web.

Por cada una de los elementos seleccionados se crearán dos wireframes, uno para dispositivos de sobremesa y otro para dispositivos móviles. Las vistas elegidas han sido la vista perfil, la vista principal y la vista de edición de documento.

Comenzaremos por la vista Perfil, en concreto en su versión de PC. Como hemos comentado esta vista contendrá información sobre los usuarios. Observando la

siguiente imagen (Ilustración 21), se pueden distinguir en la vista dos columnas, la columna izquierda contiene información sobre el usuario, como la imagen de perfil, nombre, correo electrónico, etc. Además contiene información sobre los seguidores y los seguidos, donde al pulsar se abrirá una modal que muestra la lista de usuarios que le siguen o que sigue. Junto a estos números tenemos el número de documentos públicos creados y debajo de estas estadísticas tenemos la descripción que el usuario puede escribir a su gusto. En la columna derecha tenemos los documentos públicos escritos por el usuario.

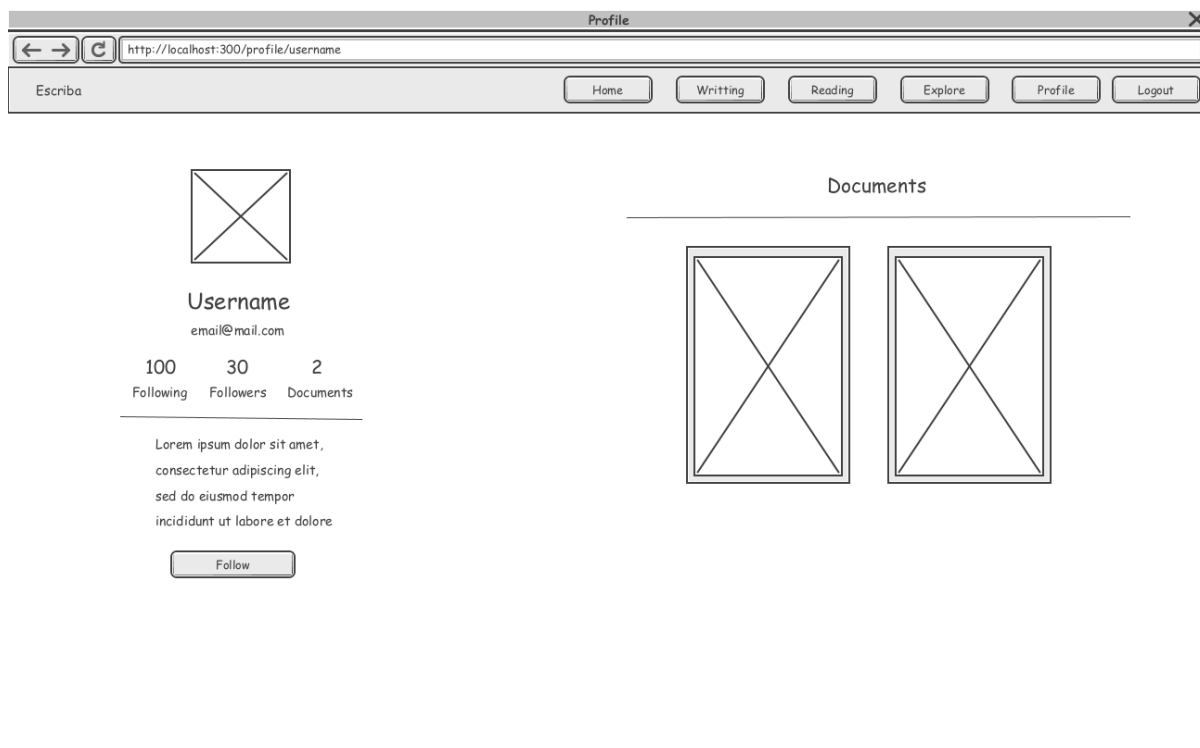


Ilustración 21. Wireframe vista Profile en PC.

Si nos centramos ahora en la vista de dispositivos móviles (Ilustración 22), se puede observar una redistribución de los elementos. Lo que antes era una distribución izquierda/derecha ahora es una distribución arriba/abajo, lo que encaja mucho mejor con los controles de un dispositivo móvil. Además, se ha incrementado el tamaño de la tarjeta de documentos para que tanto la portada como los datos sean más visibles.

Otro detalle a destacar es que la barra de navegación se ha reducido y convertido en un desplegable, esto ocurre siempre que la vista está en formato móvil.

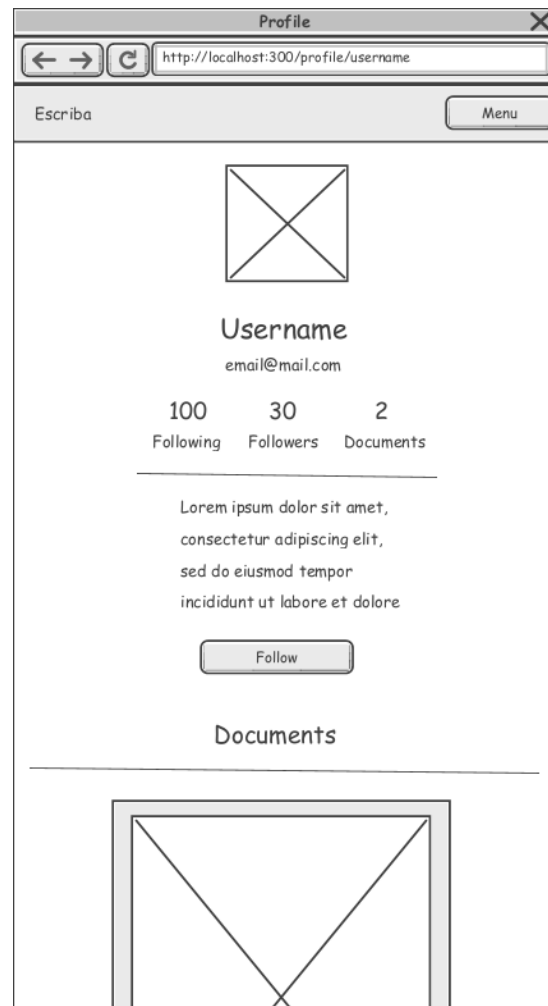


Ilustración 22. Wireframe vista Profile en móvil.

Comentaremos ahora la vista Home. Como vemos en la ilustración a continuación (Ilustración 23), se observa la vista Home para dispositivos de pantalla amplia. En la columna central se ven las distintas notificaciones que ha recibido el usuario. Se dividen en una cabecera, en la que se muestra qué usuario ha realizado qué acción. En el cuerpo de la notificación se muestra la acción realizada, por ejemplo, como es el caso de la primera notificación de la imagen, si un usuario publica un documento en el

cuerpo de la notificación aparecerá el documento publicado. A la izquierda aparece una recomendación de un documento y a la derecha aparece una lista de usuarios que la aplicación recomienda seguir. Estos dos elementos quedarán fijos en la pantalla aunque se realice un desplazamiento o Scroll.

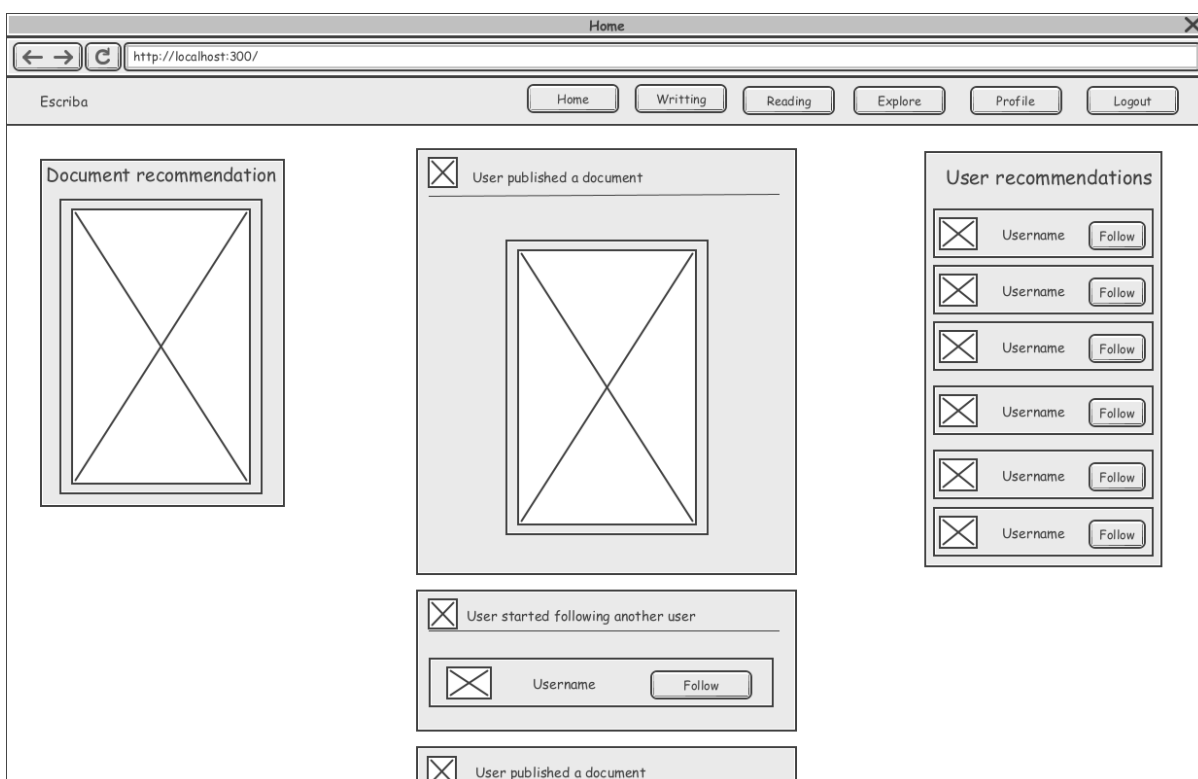


Ilustración 23. Wireframe vista Home en PC.

Después de comentar la vista en su versión de PC comentaremos ahora la vista en versión móvil, la cual se observa en la Ilustración 24. En esta versión se han eliminado los elementos laterales, haciendo que estos elementos aparezcan en forma reducida en la misma columna de las notificaciones, permitiendo así tener una sola columna que facilita el desplazamiento en dispositivos de pantalla reducida.

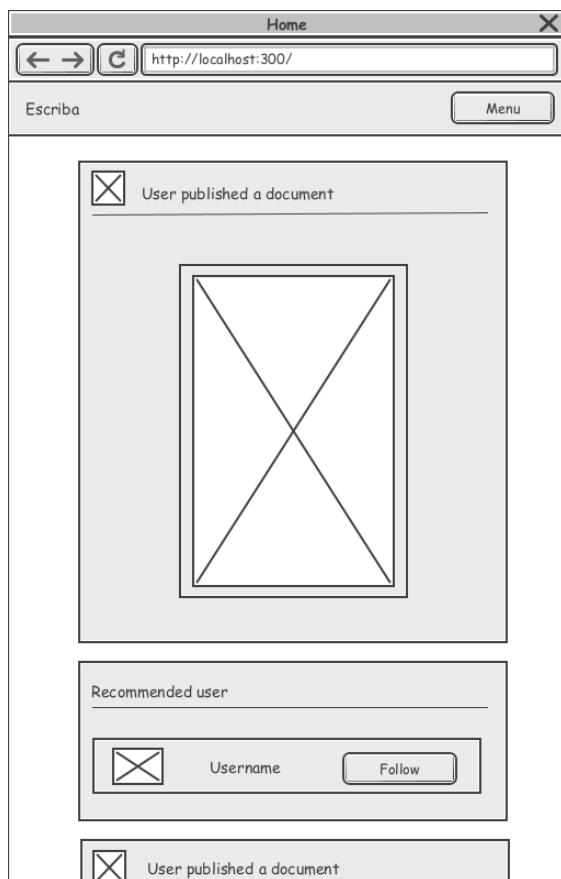


Ilustración 24. Wireframe vista Home en dispositivo móvil.

La siguiente vista a comentar es la vista de edición de texto. En su formato de pantallas de resolución amplia (Ilustración 25), observamos en la parte superior, bajo la barra de navegación la barra de herramientas. En la parte central se encuentra la hoja donde está el contenido del documento y donde este puede ser editado. A la derecha, se encuentra el generador de imágenes, el cual permite introducir una descripción de la imagen a generar y el ancho y alto de la misma. Este elemento se mantiene en ese lugar de la pantalla a pesar de que esta se desplace hacia abajo.

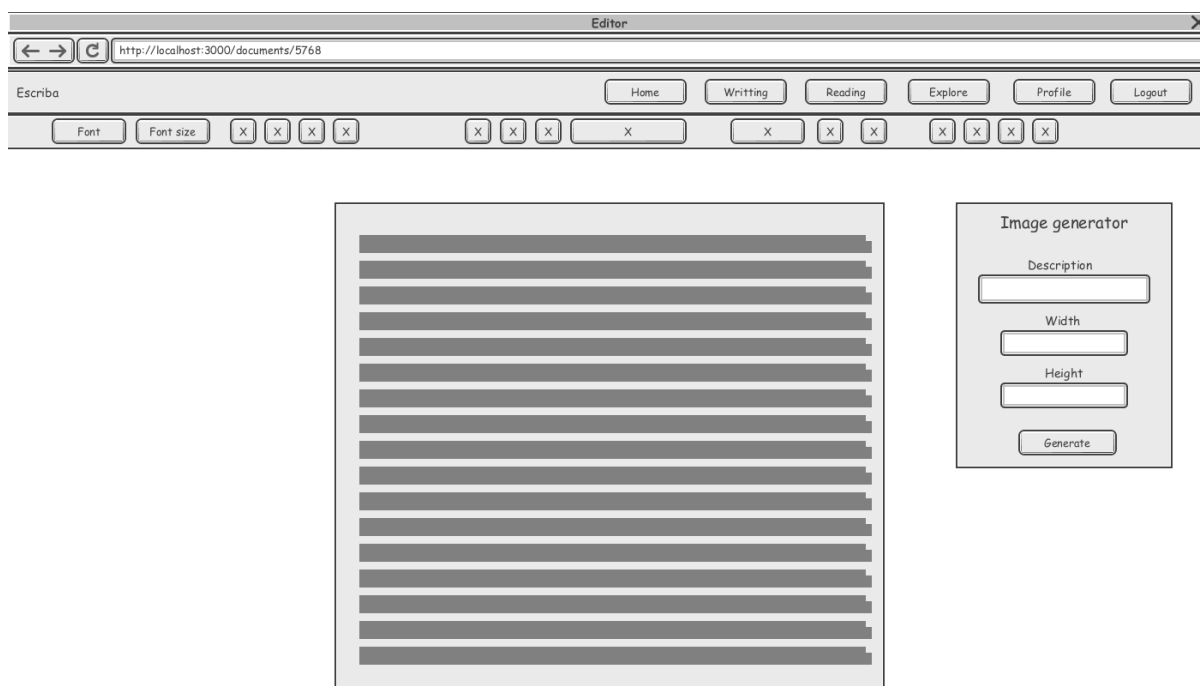


Ilustración 25. Wireframe vista Editor en PC.

En la misma vista, pero en formato móvil (Ilustración 26), la barra de herramientas se ha reducido a una versión más simple, ya que hemos considerado que la edición del texto desde el dispositivo móvil será mayormente usada para hacer retoques o arreglos a un documento, por lo tanto esta barra reducida tendrá sólo las funcionalidades más básicas. En cuanto al editor de texto, se ha cambiado su tamaño para aprovechar todo el ancho de la pantalla, lo que permite mantener lo máximo posible la longitud de las líneas y no crear líneas de texto demasiado estrechas. El generador de imágenes se ha minimizado en un botón en la parte inferior de la pantalla. Al pulsar el botón el generador de imágenes aparece en el centro de la pantalla para introducir datos, y al generar la imagen o pulsar el botón de nuevo este volverá a minimizarse.

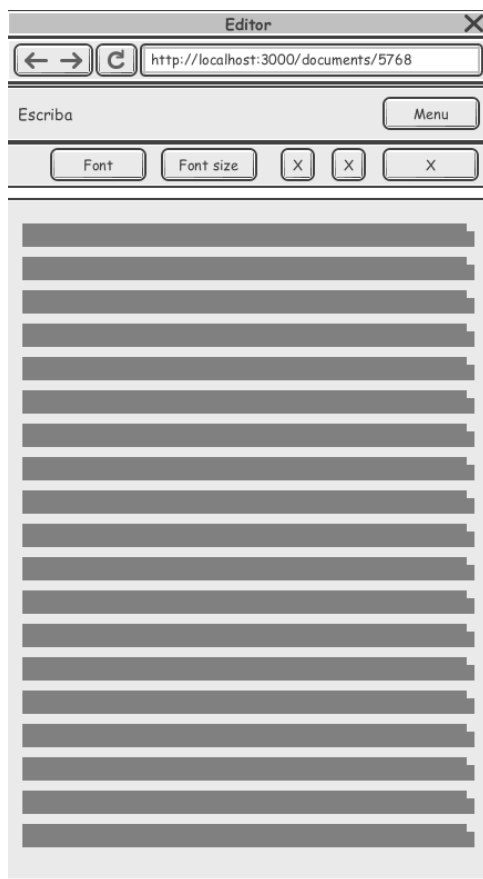


Ilustración 26. Wireframe vista Editor en dispositivo móvil.

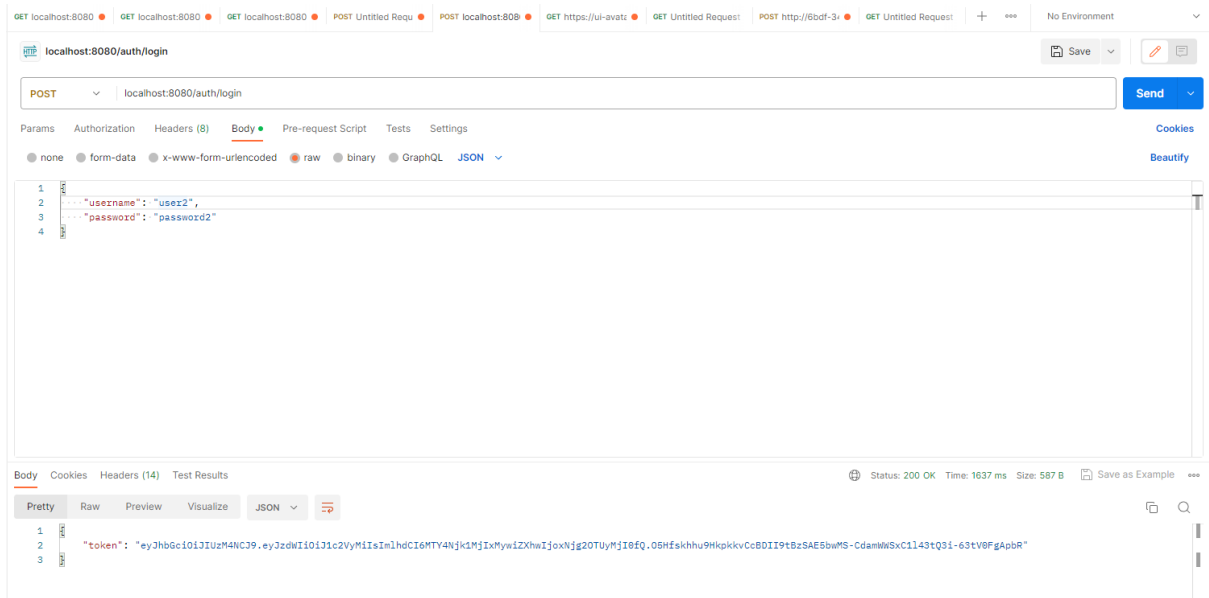
En general la mayoría de los elementos se ajustarán al tamaño de la pantalla de forma similar a los wireframes aquí mostrados. Otros funcionamientos de componentes a recalcar en dispositivos móviles son los de componentes como las modales, las cuales ocupan todo el espacio de la ventana, ya que en este tipo de dispositivos ventanas de reducido tamaño no son cómodas de manejar. Mientras tanto, en pantallas más grandes las modales se mostrarán como ventanas emergentes centradas que dejarán ver la vista que las ha abierto de forma borrosa detrás de ellas.

3.4. Plan de pruebas

En este punto sobre el plan de pruebas comentaremos qué tipo de pruebas aplicaremos al front-end y back-end para comprobar que la aplicación es robusta, segura y funciona de la forma esperada.

En lo que respecta al back-end, las pruebas que se realizarán serán pruebas de integración centradas en las pruebas de la API. Las pruebas de integración nos ayudan a comprobar que la interacción entre los diferentes elementos de nuestra aplicación, como servicios y componentes, es correcta, además de sacar a la luz distintos problemas que puedan surgir en dicha interacción. Al realizar estas pruebas sobre el back-end, nos aseguramos de que pueda ser correctamente usado por el front-end u otros elementos externos que hagan uso de él.

Para realizar estas pruebas, utilizaremos dos herramientas. Swagger, de la cual hemos hablado con anterioridad, y Postman, la cual es una herramienta que nos permite enviar solicitudes HTTP y recibir la respuesta a dicha solicitud para que pueda ser analizada. Gracias a esto, podremos probar diferentes casos de peticiones y comprobar cómo el comportamiento varía y si este está dentro de lo esperado. Se comprobará que cada endpoint devuelva los resultados correctos, responda con los códigos esperados, controle las excepciones de manera correcta y mantenga las peticiones seguras, permitiendo el acceso únicamente a usuarios que puedan tener acceso a ellas. En la ilustración 27 se muestra la interfaz de la aplicación Postman.

*Ilustración 27. Interfaz de Postman*

Como hemos mencionado, a parte de Postman utilizaremos Swagger. Esta interfaz, además de proporcionar documentación como se ha comentado en puntos anteriores, también permite el envío de peticiones de manera similar a Postman, con el objetivo de probar la aplicación junto a la documentación que Swagger aporta.

En lo que respecta al front-end, las pruebas que realizaremos serán pruebas de aceptación. Este tipo de pruebas nos permite saber si las funcionalidades que se deben entregar al usuario final han sido satisfechas o no. Para ello utilizaremos los criterios de aceptación definidos anteriormente, crearemos una plantilla en la que se incluyen las historias de usuario con sus criterios de aceptación, y recorreremos la aplicación anotado qué criterios se han cumplido y cuales no, comprobando también casos concretos como entradas inválidas para comprobar cómo reacciona el sistema.

También sería conveniente la implementación de pruebas unitarias o de componentes en el front-end, lo que ayudaría a futuras implementaciones, ya que si esas

implementaciones afectasen a código anterior negativamente, el error quedaría en evidencia al ejecutar los test, sin necesidad de volver a implementarlos, por lo que ayudaría a la posibilidad de expansión de la aplicación. En caso de que finalmente sean implementados, se implementarán test para las principales vistas, ya que estas vistas engloban al resto de componentes.

4. Implementación

4.1. Arquitectura

En este punto, comentaremos cuál es la arquitectura de la aplicación. Para ello, utilizaremos un diagrama arquitectónico, que mostrará los componentes más destacables a nivel de arquitectura y las relaciones que existen entre estos elementos.

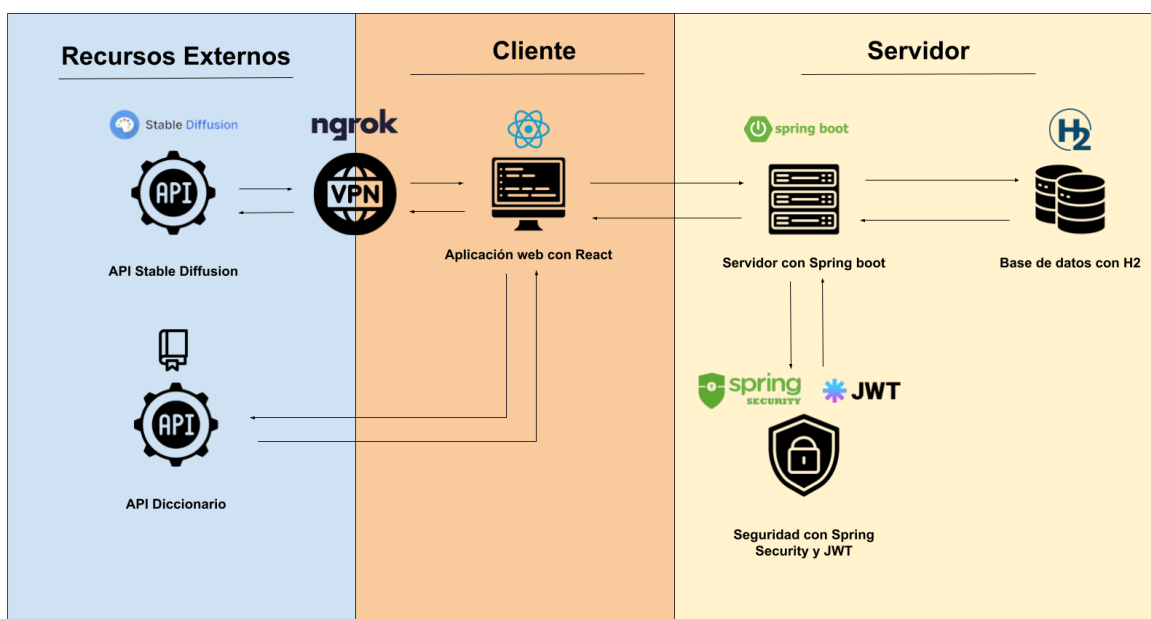


Ilustración 28. Diagrama arquitectónico

El diagrama se muestra en la ilustración 28. A primera vista vemos como el diagrama se divide en dos partes o secciones. La primera de ellas, a la izquierda, es la sección del cliente o capa de presentación. En ella se incluye la aplicación web o cliente REST, es decir, la parte de la aplicación que interactúa con los usuarios.

La aplicación web en React es la interfaz de usuario. En nuestra arquitectura decidimos utilizar React por su enfoque basado en componentes, lo que nos permite

dividir la interfaz en elementos más pequeños y reutilizables. La función principal de este elemento de la arquitectura es la de mostrar la información al usuario y recoger las interacciones que este mismo haga con la interfaz de usuario. Como sabemos por lo comentado en puntos anteriores, React emplea como lenguajes de programación JavaScript, JSX y CSS. Haciendo uso de esos lenguajes, se creará una librería de componentes propia, lo que trae como beneficio una mayor fidelidad de los resultados respecto a los diseños de interfaz, puesto que si se utilizan librerías de componentes como Bootstrap o Tailwind CSS, deberíamos adaptarnos a los componentes que nos ofrecen. Al implementar nuestros propios componentes podremos personalizar mejor los componentes, además de adquirir un mayor conocimiento sobre cómo funcionan los componentes en los niveles más internos.

En la sección de recursos externos encontramos algunas APIs. Las APIs externas son servicios o sistemas externos a nuestra aplicación que nos permiten interactuar y conseguir información de fuentes externas. En nuestro caso, las principales APIs externas que utilizaremos serán “*Stable Diffusion*” como API generadora de imágenes, a la cual se le enviará una petición desde el cliente, el cual recibirá una respuesta con la imagen generada según los parámetros del envío y procederá a utilizarla como se requiera. Esta API requiere el uso de un servicio externo llamado “*Ngrok*”, el cual es una herramienta que permite exponer una aplicación que se ejecuta en el servidor local a internet, ya que la API se está ejecutando un servidor local distinto al servidor local del cliente y servidor REST, lo que hace imposible la comunicación API - Aplicación REST sin esta herramienta.

Otra de las APIs que utilizaremos será la API de diccionario, que nos servirá para obtener las definiciones de una palabra en concreto, funcionalidad que se utilizará cuando un usuario esté leyendo un documento y necesite de una definición. Otra API

menos clave que se utilizará será una que cree imágenes de perfil por defecto a partir de las iniciales de los usuarios.

Entrando en la sección del servidor o capa de back-end, observamos que se distinguen tres elementos. El primero de ellos es el servidor, el cual se comunica con el cliente. Este servidor será implementado en Spring boot y se encarga de recibir, procesar y responder a las peticiones que realiza el cliente, o lo que es lo mismo, manejar la lógica de negocio, operar con los datos y devolver resultados de esas operaciones.

Para que el servidor pueda operar con esos datos, los datos deben estar almacenados en una base de datos. En el diagrama se observa una relación bidireccional entre el servidor y la base de datos, que representa las peticiones y respuestas de datos. Esta base de datos se implementa en el mismo Spring boot mediante una dependencia con H2, el cual es un sistema de gestión de bases de datos relacionales. En resumen, la función de la base de datos es la de almacenar y gestionar los datos. Por ejemplo, se encarga de la persistencia de la información además de la recuperación de esta misma. Al ser H2 una base de datos relacional la información se almacenará en tablas, donde cada columna es un atributo de la entidad almacenada y cada fila una instancia de una instancia de esa entidad.

Además de con la base de datos, el servidor se relaciona en el diagrama con un módulo de seguridad, en concreto con Spring Security. Este módulo facilita la creación e implementación de medidas de control de la seguridad, con funcionalidades como la autenticación, autorización o protección contra ataques. En concreto, nosotros realizaremos una implementación basada en JWT, ya que Spring Security permite la autenticación basada en tokens. El funcionamiento de JWT es sencillo de entender, cuando un usuario inicia sesión en la aplicación, este recibe un token que lo identifica como usuario y contiene información de sus permisos y roles. Esta implementación

que combina Spring boot y JWT nos deja con una capa de seguridad robusta y difícil de penetrar. Además al utilizar JWT se elimina la necesidad de almacenar información sobre la sesión activa en el servidor, lo que lo convierte en un modelo escalable y adecuado para APIs RESTful.

4.2. Detalles sobre implementación

Una vez hemos comentado la arquitectura del proyecto, nos centraremos ahora en los detalles sobre la implementación. Hablaremos sobre los elementos clave del servidor y el cliente, ilustrando fragmentos de código representativos de cada paso a seguir para implementar un prototipo funcional.

4.2.1. Detalles sobre la implementación del servidor

En este punto nos centraremos en los pasos seguidos para implementar el servidor. Estos pasos incluyen desde la creación del proyecto hasta la seguridad, pasando por la creación de entidades, servicios, repositorios y controladores.

4.2.1.1. Creación del proyecto

El primero de los pasos fue la generación del proyecto. Para conseguir un proyecto con las dependencias básicas utilizamos un servicio conocido como “*Spring Initializr*”¹⁴ (Ilustración 29), el cual nos permite generar un proyecto de spring boot con las dependencias que decidamos incluir en él. En nuestro caso, las dependencias iniciales con las que generamos el proyecto serán “*Spring data JPA*”, la cual incluye todo lo necesario para poder utilizar JPA, “*H2 database*”, que nos permite utilizar H2 como base de datos, a través de la API JDBC¹⁵, “*Lombok*”, que nos permite mediante anotaciones el acceso a los datos de las entidades de forma más sencilla y ayuda a reducir el código, “*Spring Security*” [17], que nos permitirá añadir seguridad a nuestra

¹⁴ <https://start.spring.io/>

¹⁵ Java Database Connectivity

aplicación, y por último, “*Validation*”, que nos permitirá validar los distintos Beans de nuestro servidor. En el futuro se añadirán más dependencias, pero estas son las dependencias con las que comenzamos inicialmente.

The screenshot shows the Spring Initializr web application interface. At the top, there is a green banner with the text "Meet the Spring team this August at SpringOne". Below the banner, the "spring initializr" logo is visible. The interface is divided into several sections:

- Project:** Includes radio buttons for "Gradle - Groovy", "Gradle - Kotlin", and "Maven" (selected).
- Language:** Includes radio buttons for "Java" (selected), "Kotlin", and "Groovy".
- Spring Boot:** Includes radio buttons for "3.1.1 (SNAPSHOT)", "3.1.0" (selected), "3.0.8 (SNAPSHOT)", "3.0.7", "2.7.13 (SNAPSHOT)", and "2.7.12".
- Project Metadata:** Includes input fields for "Group" (com.example), "Artifact" (demo), "Name" (demo), "Description" (Demo project for Spring Boot), and "Package name" (com.example.demo). It also has radio buttons for "Packaging" (Jar selected, War) and "Java" (20, 17 selected, 11, 8).
- Dependencies:** Includes a button "ADD DEPENDENCIES... CTRL + B" and a list of dependencies: "Spring Data JPA" (SQL), "H2 Database" (SQL), "Lombok" (DEVELOPER TOOLS), and "Spring Security" (SECURITY). Each dependency has a brief description.

At the bottom, there are three buttons: "GENERATE CTRL + G", "EXPLORE CTRL + SPACE", and "SHARE...".

Ilustración 29. Ejemplo de la interfaz de spring initializr.

Una vez que tenemos el proyecto creado y hemos accedido a él mediante el entorno de desarrollo que seleccionamos, será momento de comenzar la implementación de los elementos principales.

4.2.1.2. Estructura del proyecto

A la hora de decidir cómo estructurar el proyecto, decidimos crear una carpeta o paquete para tipo de clase. En concreto, creamos una carpeta para los controladores, otra para los servicios, otra para las entidades, etc. Todas estos Beans serán explicados en los siguientes puntos. También optamos por crear una carpeta que

encapsule la seguridad, y otra que contenga a las excepciones. En la ilustración 30 se observa la jerarquía de carpetas.

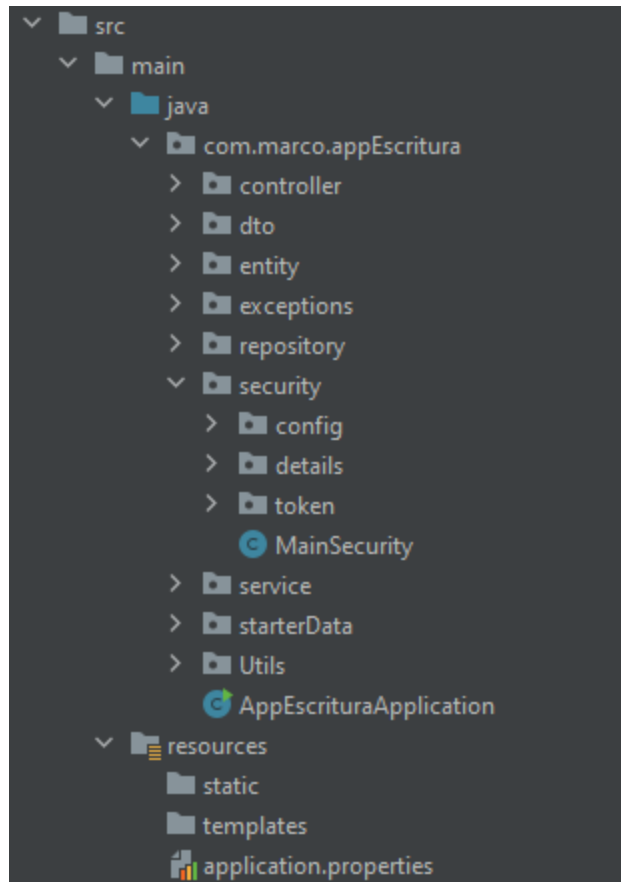


Ilustración 30. Jerarquía de carpetas del servidor.

La forma de organizar las carpetas nos ayuda a facilitar la comprensión del código y mejora la legibilidad y mantenibilidad. Incluso nos hace reforzar la idea de que cada clase debe tener una responsabilidad única, como por ejemplo los servicios encargarse de la lógica de negocio o los controladores de manejar las peticiones HTTP.

A continuación, en la ilustración 31, se muestra el diagrama de paquetes. Un diagrama de paquetes representa mediante paquetes la organización de la aplicación

y las relaciones entre estos elementos de organización. En nuestro caso, como se ve en la ilustración, hemos organizado los elementos por responsabilidades.

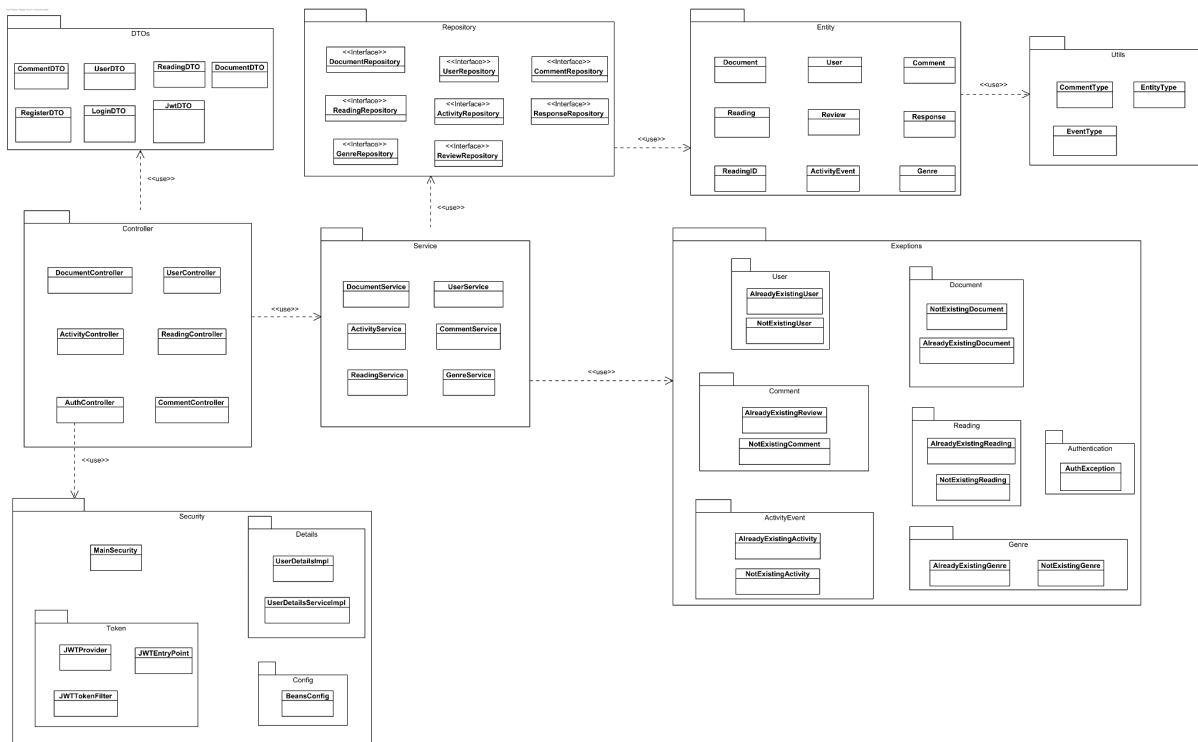
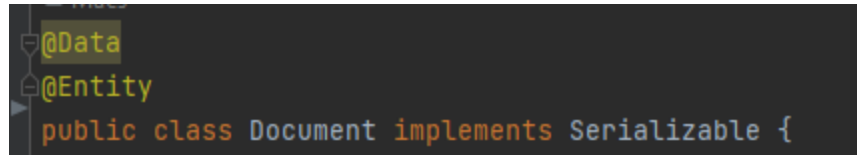


Ilustración 31. Diagrama de paquetes del servidor

4.2.1.3. Definición de las entidades

Las primeras clases a crear serán las entidades, para ello crearemos una clase Java con el nombre de la entidad. Una vez se haya creado la clase, el primer paso será declarar que esta clase será un Bean de tipo entidad. Los Beans, como se explica en [18], son objetos que se crean y se administran por un contenedor de Spring. En concreto una entidad es un tipo concreto de bean que se encarga de representar una tabla en una base de datos. Para convertir la clase en un *Entity Bean* debemos agregar la anotación “@Entity” como se observa en la siguiente ilustración, la ilustración 32.

A screenshot of a code editor showing Java code. The code includes the annotations `@Data` and `@Entity` on separate lines, followed by the declaration `public class Document implements Serializable {`. The annotations are highlighted in yellow, and the class name `Document` is in blue. The background is dark with light-colored text.

```
@Data
@Entity
public class Document implements Serializable {
```

Ilustración 32. Ejemplo creación de una entidad.

Como vemos en la imagen, se puede observar la notación que hemos contado para la definición de la entidad. Sobre ella se aprecia la notación “`@Data`”, la cual proviene del módulo comentado anteriormente “*Lombok*”, y que nos permite con esa simple notación tener acceso a los getters y setters de todos los atributos y relaciones que implementemos posteriormente. Es importante recalcar que esta notación afecta negativamente al uso de memoria y al rendimiento cuando se utiliza en entidades de JPA como es el caso, aunque como nuestra aplicación es un prototipo la utilizaremos para facilitar la implementación y la lectura del código.

Tras esto, se ha de incluir los atributos necesarios y relaciones con otras entidades. Para la definición de atributos, se implementarán como se muestra en la ilustración 33, donde se observan diferentes atributos, incluyendo los tipos de los atributos, los nombres y validaciones mediante el módulo *Validation* mencionado anteriormente.

```
@Id
@GeneratedValue(strategy = GenerationType.AUTO)
private long id;
2 usages
@Lob
@Column(columnDefinition = "TEXT")
String text;
1 usage
@NotBlank(message = "El título no puede estar vacío")
String tittle;
2 usages
boolean isPublic;
2 usages
@Lob
@Column(columnDefinition = "TEXT")
String synopsis;

2 usages
@Lob
@Column(columnDefinition = "TEXT")
String cover;

2 usages
@Min(value = 0, message = "La calificación no puede ser negativa")
@Max(value = 5, message = "La calificación no puede ser mayor a 5")
int rating;
```

Ilustración 33. Ejemplo de implementación de atributos.

Como vemos en la ilustración, existe un atributo llamado ID. Este atributo es obligatorio para las entidades, ya que representa la clave única que se utilizará en la tabla que estas entidades representan. Vemos además otros tipos de notaciones sobre los atributos, algunas de validación como “*NotBlank*” que no permite que el campo esté vacío o “*Max/Min*” que colocan límites al valor de un entero. Otras notaciones como “*Lob*¹⁶” indican que el atributo es de gran tamaño, en nuestro caso los textos son bastante extensos.

¹⁶ Lob es acrónimo de Large Object

Después de implementar los atributos de una entidad, implementaremos las relaciones con otras entidades. En nuestro caso la mayoría de relaciones son bidireccionales, lo que implica que ambas entidades de la relación tienen constancia de la relación, es decir, contienen un atributo que representa esa relación. Para crear una relación entre entidades bastará con identificar la cardinalidad de la misma y crear un atributo del tipo de la entidad con la que se relaciona. En caso de que la cardinalidad pueda ser mayor que uno, la relación se representará mediante una colección de objetos, en nuestro caso una lista. Además, para indicar la relación, se necesita incluir una anotación que indique la cardinalidad de la relación, tal y como se muestra en la ilustración 34.

```
4 usages
@ManyToMany(fetch = FetchType.EAGER, cascade = CascadeType.ALL)
List<Genre> genres;
2 usages
@ManyToMany(mappedBy = "savedDocuments")
List<User> savedBy;
1 usage
@OneToMany(mappedBy = "postedIn", cascade = CascadeType.ALL, orphanRemoval = true)
@OnDelete(action = OnDeleteAction.CASCADE)
List<Comment> reviews;
2 usages
@NotNull(message = "El creador no puede ser nulo")
@ManyToOne
User creator;
2 usages
@OneToMany(mappedBy = "beingRead", cascade = CascadeType.ALL, orphanRemoval = true)
@OnDelete(action = OnDeleteAction.CASCADE)
List<Reading> beingRead;
```

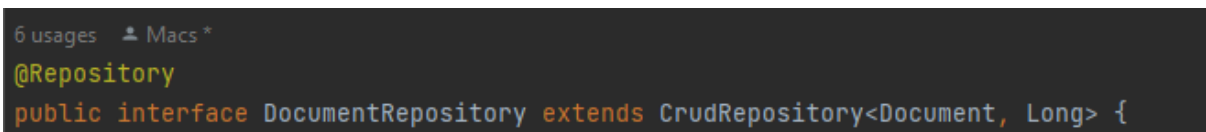
Ilustración 34. Ejemplo de implementación de relaciones.

Como vemos, se utilizan diferentes notaciones para indicar la cardinalidad, como “OneToMany”, “ManyToMany” o “ManyToOne” entre otras. En estas relaciones también se debe indicar cuál es propietario de la relación, lo que se traduce en que los cambios sólo se reflejarán si se realizan desde el lado que es propietario de la relación. Junto a estas notaciones, se incluyen diferentes parámetros que influyen en

cómo actúa un lado de la relación cuando se realizan inserciones o eliminaciones en el otro lado, como es el caso de “*cascade*” u “*orphanRemoval*”

4.2.1.4. Repositorios

Una vez que hemos creado las entidades, es turno de crear los repositorios. Los repositorios son interfaces, las cuales representan a la capa de almacenamiento de datos. En nuestro caso utilizaremos la interfaz “*CrudRepository*”, que nos permite utilizar funcionalidades como la búsqueda, creación eliminación o actualización sin tener que implementar nada. En la ilustración 35 mostramos cómo sería la creación del repositorio. Es importante recalcar que utilizamos la anotación “*@Repository*” para indicar a Spring boot que estamos tratando con un Bean de tipo repositorio.

A screenshot of a code editor with a dark background. At the top left, it says "6 usages" and "Mac's". The code is written in a light-colored font. It starts with the annotation "@Repository" in green. Below it, the line "public interface DocumentRepository extends CrudRepository<Document, Long> {" is shown in orange and black. The opening curly brace is also visible.

```
6 usages  Mac's *  
@Repository  
public interface DocumentRepository extends CrudRepository<Document, Long> {
```

Ilustración 35. Ejemplo de implementación de Repositorio.

En caso de que no sea suficiente con las operaciones que nos ofrece la interfaz Crud, podremos utilizar nuestras propias operaciones en el repositorio, principalmente las operaciones de búsqueda. En el ejemplo siguiente, ilustración 36, se observan diferentes búsquedas, donde mediante “*Queries SQL*” podremos especificar qué se debe buscar en la base de datos.

```

no usages  ↗ Macs
@Query("SELECT d FROM Document d WHERE d.creator.username = ?1")
Optional<List<Document>> findByCreatorUsername(String username);
no usages  ↗ Macs
@Query("SELECT DISTINCT d FROM Document d WHERE d.isPublic = false ORDER BY d.rating DESC, d.title ASC")
List<Document> findAllOrderByRatingAndTitle();
1 usage  ↗ Macs
@Query(value = "SELECT * FROM Document ORDER BY rating, title LIMIT ?1 OFFSET ?2",
      nativeQuery = true)
Iterable<Document> getPageDocuments(int pageSize, int offset);
1 usage  ↗ Macs
@Query("SELECT d FROM Document d WHERE d.isPublic = true AND (d.title = '' OR d.title LIKE %?1%)")
List<Document> findPublicDocumentsByTitleFragment(String fragment);
1 usage  ↗ Macs
@Query("SELECT d FROM Document d WHERE d.isPublic = true AND d.rating = (SELECT MAX(d.rating) FROM Document d) ORDER BY RAND() LIMIT 1 ")
Optional<Document> findRandomMostLikedPost();

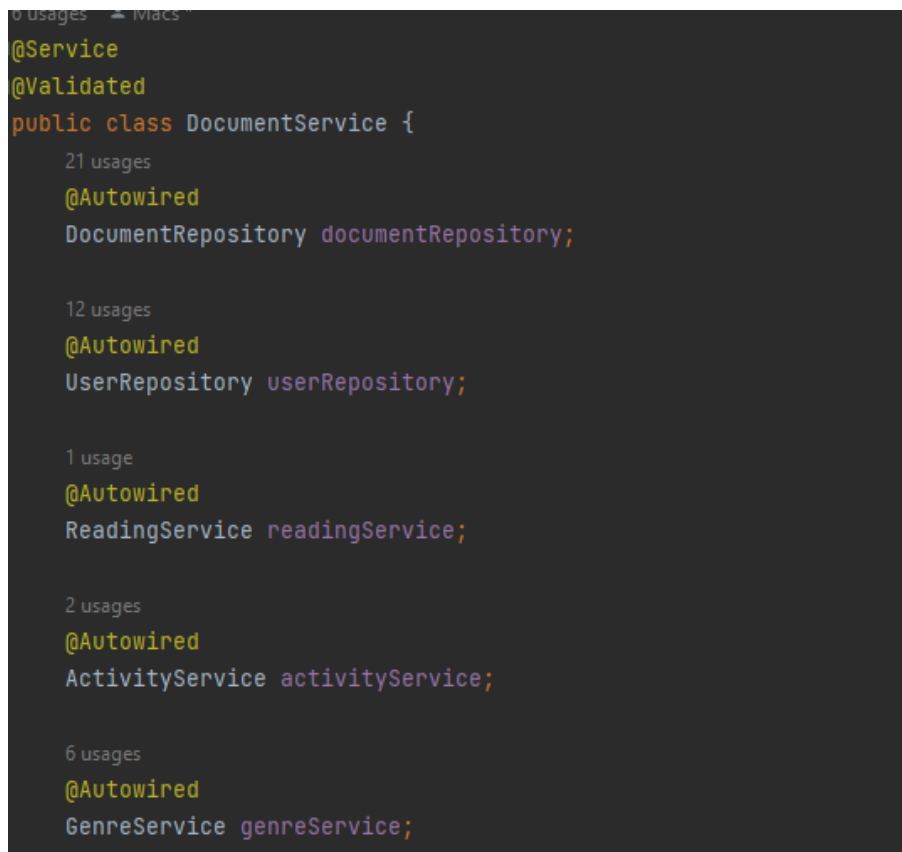
```

Ilustración 36. Ejemplo de queries en repositorio.

Como se puede ver en la ilustración anterior (Ilustración 36), las búsquedas que se realizan con consultas SQL son complejas y sería tedioso hacerlas a partir de operaciones derivadas de la interfaz CRUD.

4.2.1.5. Servicios

Tras haber implementado los repositorios, es el turno de los servicios. Los servicios son la capa donde se implementa la lógica de negocio. Es la capa intermedia entre los repositorios, que hemos visto anteriormente y los controladores, los cuales veremos a continuación. La lógica de negocio incluye operaciones con los datos y las comprobaciones necesarias para corroborar que todo funciona según lo previsto. No es necesario recalcar que los servicios también son Beans, y se identifican como tal mediante la anotación “@Service”. En la ilustración 37, se muestran las notaciones de los servicios y la inyección de dependencias.



```
0 usages  → Macs
@Service
@Validated
public class DocumentService {
    21 usages
    @Autowired
    DocumentRepository documentRepository;

    12 usages
    @Autowired
    UserRepository userRepository;

    1 usage
    @Autowired
    ReadingService readingService;

    2 usages
    @Autowired
    ActivityService activityService;

    6 usages
    @Autowired
    GenreService genreService;
```

Ilustración 37. Definición de servicio e inyección de dependencias

Como se ha mostrado en la ilustración 37, se incluyen la anotación de servicio para identificar a la clase como tal, y la anotación “*@Validated*”, la cual indica que se debe comprobar que los datos que se introducen son válidos.

La inyección de dependencias indica qué Beans serán utilizados por este servicio y se implementan mediante la definición de un objeto del tipo del Bean a inyectar junto con la notación “*@Autowired*”. Como vemos en la imagen, el servicio documento utiliza varios repositorios y varios servicios para realizar operaciones con los datos. Los estándar es que un servicio tenga una inyección de dependencias a su repositorio concreto, pero es común también el necesitar de otros repositorios o de otros servicios.

```
usage 1 Macs
public void userSavesDocument(String username, Long documentId) {
    Optional<User> userOptional = userRepository.findOneByUsername(username);
    Optional<Document> documentOptional = documentRepository.findById(documentId);

    if(!userOptional.isPresent()){
        throw new NotExistingUser("User with username: " + username + " does not exist");
    }

    if(!documentOptional.isPresent()){
        throw new NotExistingDocument("Document with id: " + documentId + " does not exist");
    }
}
```

Ilustración 38. Ejemplo lógica de negocio

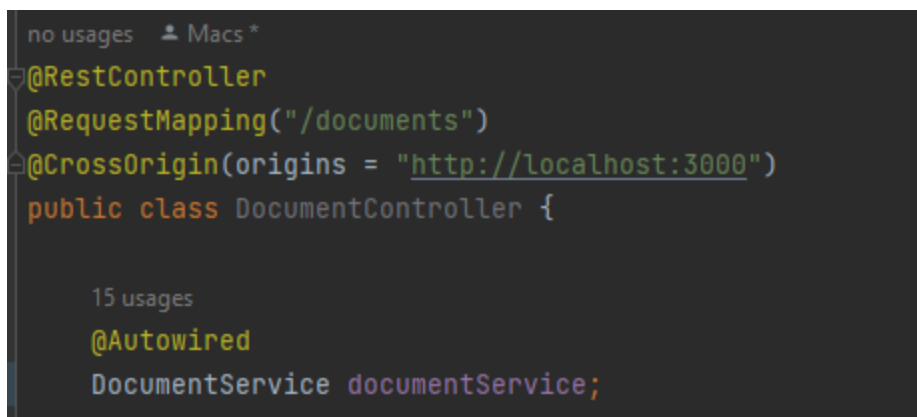
En la ilustración 38 observamos un fragmento de una función del servicio. En ella se realizan dos búsquedas y se comprueba que esas búsquedas han tenido un resultado, puesto que en caso de no comprobarse, la situación en la que la búsqueda no ha sido fructífera produciría una excepción no controlada que podría llevar al desplome y paralización de la aplicación completa. Por lo tanto, en caso de que las búsquedas no encuentren resultado, se lanzarán excepciones acordes al problema en cuestión, las cuales serán capturadas por el controlador, como veremos más adelante.

4.2.1.6. Controladores REST

Ahora es el turno de los controladores. Los controladores son los Beans que manejan las solicitudes HTTP entrantes y responden a estas solicitudes con respuestas HTTP. Para su implementación, crearemos una clase de Java, la cual anotaremos con “*@RestController*”. Además, en nuestro caso incluiremos la anotación “*@RequestMapping*” junto a una URL para organizar todos los métodos del controlador bajo una misma URL. Además será necesario otra anotación “*@CrossOrigin*”, la cual permitirá el acceso a las CORS¹⁷ desde la dirección especificada. Los controladores también requieren una inyección de dependencia, es

¹⁷ Las CORS son una especificación que permite a un servidor restringir los dominios que pueden acceder a sus recursos

nuestro caso estas inyecciones serán de uno o varios servicios. Todas estas características se observan en la ilustración 39.



```
no usages  Macs *
@RestController
@RequestMapping("/documents")
@CrossOrigin(origins = "http://localhost:3000")
public class DocumentController {

    15 usages
    @Autowired
    DocumentService documentService;
```

Ilustración 39. Ejemplo anotaciones controlador REST

Una vez hemos configurado la clase para que se comporte como un controlador y hemos inyectado las dependencias necesarias, se debe comenzar a implementar los *end-points*.

Para implementar los endpoints, será necesario incluir qué tipo de operación realiza el método HTTP y a qué URL se debe realizar la petición. Para ello utilizaremos la notación “@XMapping(“/URL”)”, donde la X será el tipo de petición (POST, GET, PUT, etc) y URL será la dirección concreta a la que se realiza la petición. Es importante recalcar que para cada URL sólo puede existir una operación de cada tipo. En la ilustración 40 se observa uno de estos *end-points*.

```
@GetMapping("/public/{username}")
public ResponseEntity<Iterable<DocumentDTO>> getPublicDocumentsBy(@PathVariable String username) {
    Iterable<Document> documents = documentService.getPublicDocumentsCreatedBy(username);
    Collection<Document> documentCollection = new ArrayList<>();
    documents.forEach(documentCollection::add);
    return ResponseEntity.ok(documentCollection.stream() Stream<Document>
        .map(document -> document.toDto()) Stream<DocumentDTO>
        .collect(Collectors.toList()));
}
```

Ilustración 40. Ejemplo implementación end-point

En la ilustración 40 se observa la anotación que hemos comentado anteriormente. Este en concreto es un método HTTP de tipo GET a la URL “/documents/public/username”, siendo username una parámetro que se incluye en la URL como indica el método. Todos estos métodos devuelven una respuesta HTTP, en concreto devuelven OK como cabecera de la respuesta en caso de que las operaciones se hayan completado correctamente, junto con la información requerida en el cuerpo de la respuesta.

En caso de que se haya producido una excepción durante las operaciones de la petición, el mismo controlador es el encargado de recoger esa excepción y procesarla. Esto se realiza mediante la implementación de un método concreto, al que se le añade la anotación “@ExceptionHandler”, que indica el tipo de excepción a procesar y la notación “@ResponseStatus”, que indica el código respuesta que se debe enviar al cliente. Todo esto se aprecia en los métodos de ejemplo de la ilustración 41.

```
no usages  👤 Macs
@ExceptionHandler({ConstraintViolationException.class})
@ResponseStatus(HttpStatus.BAD_REQUEST)
public ResponseEntity<String> handlerRestrictions(ConstraintViolationException e) {
    return ResponseEntity.badRequest().body(e.getMessage());
}

no usages  👤 Macs
@ExceptionHandler({NotExistingDocument.class})
@ResponseStatus(HttpStatus.NOT_FOUND)
public ResponseEntity<String> NotExistingExceptionHandler(RuntimeException e) {
    return ResponseEntity.badRequest().body(e.getMessage());
}

no usages  👤 Macs
@ExceptionHandler({AlreadyExistingDocument.class})
@ResponseStatus(HttpStatus.CONFLICT)
public ResponseEntity<String> AlreadyExistingExceptionHandler(RuntimeException e) {
    return ResponseEntity.badRequest().body(e.getMessage());
}
```

Ilustración 41. Ejemplo manejo de excepciones del controlador

4.2.1.7. Definición de los DTOs

Como hemos mencionado estos controladores reciben peticiones con unos parámetros y devuelven respuestas con información. En ocasiones, esta información que se incluye en los parámetros y en las respuestas llega a ser muy pesada si se trata de entidades completas, por lo tanto se han implementado unos DTOs que incluyan la información relevante de estas entidades pero que sean más ligeros, mostrando las relaciones entre entidades de una forma más sencilla, como por ejemplo incluyendo únicamente la clave de esa entidad relacionada y no la entidad al completo. Eso se hace porque las relaciones son uno de los elementos que más influyen en el tamaño de los objetos. Los DTOs se implementarán como una clase normal en Java, incluyendo atributos similares a los de la entidad que representan y simplificando las relaciones. Se utilizarán notaciones de “*Lombok*” como “*@Data*” o

“@NoArgsConstructor”, las cuales crean *getters* y *setters*, y constructores vacíos respectivamente. En la ilustración 42, vemos ejemplo de cómo se adaptan los atributos en un DTO.



```
23 usages  ⤴ Macs *
@Data
public class DocumentDTO {
    2 usages
    long id;
    1 usage
    String tittle;
    1 usage
    String cover;
    3 usages
    String text;
    1 usage
    boolean isPublic;
    1 usage
    String creatorUsername;
    1 usage
    String synopsis;
    2 usages
    List<String> genres;
    1 usage
    List<String> savedBy;
    1 usage
    List<ReadingDTO> readings;
    2 usages
    int rating;
```

Ilustración 42. Ejemplo DTO

Como se observa en la ilustración 42, relaciones como la del creador, pasan a ser un string con el nombre de usuario del creador. Otras como las lecturas, pasan de ser una lista de lecturas a una lista de lecturas DTO.

4.2.1.8. Autenticación y control de acceso

Hablaremos a continuación de la seguridad. La seguridad es un aspecto clave de la aplicación, ya que nos asegura que los datos sólo puedan ser accedidos por personas que tienen la autorización de acceder a ellos. La seguridad se ha llevado a cabo con diferentes clases que se relacionan entre sí. En nuestro caso, hemos utilizado Spring

Security junto con una implementación JWT. Esta implementación implica la generación de un token que se le entrega a cada usuario identificado y que este mismo usuario debe utilizar para que sus peticiones sean aceptadas. A continuación mencionaremos las clases y mostraremos las partes principales de cada clase

- *Auth Controller*: Es un controlador que se encarga de las peticiones HTTP relacionadas con la autenticación. Los métodos que incluye son el de registro, que se encarga de crear un usuario en la base de datos, el de inicio de sesión, que valida las credenciales y entrega un token JWT al usuario que ha iniciado sesión correctamente y el método de refresco, que refresca un token válido caducado mediante la entrega de otro token nuevo. En la siguiente ilustración, la ilustración 43 se muestra el controlador.

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

    no usages
    @Autowired
    PasswordEncoder passwordEncoder;

    1 usage
    @Autowired
    AuthenticationManager authenticationManager;

    2 usages
    @Autowired
    UserService userService;

    2 usages
    @Autowired
    JwtProvider jwtProvider;

    no usages 1 Macs *
    @Operation(summary = "Registrar usuario", description = "Registra a un usuario en la aplicación")
    @PostMapping("/register")
    public ResponseEntity<?> register(@Valid @RequestBody RegisterDto newUser){

        if(userService.existsByUsername(newUser.getUsername()))
            return new ResponseEntity(HttpStatus.BAD_REQUEST);

        User user = new User(newUser.getUsername(),
                             newUser.getPassword(),newUser.getEmail());
        user.setImage(newUser.getProfileImage());

        userService.save(user);
        return new ResponseEntity(HttpStatus.CREATED);
    }

    no usages 1 Macs
    @Operation(summary = "Iniciar sesión", description = "Inicia la sesión de un usuario en la aplicación")
    @PostMapping("/login")
    public ResponseEntity<?> login(@Valid @RequestBody LoginDto loggedUser){

        // ...
    }
}
```

Ilustración 43. Controlador de autenticación

Como vemos en la ilustración anterior, el controlador tiene un esquema similar a otros controladores, tiene sus propias inyecciones de dependencias y sus métodos HTTP con sus rutas propias.

- *User Details Service Impl*: Esta es una implementación de una interfaz que proporciona Spring Security, el cual la utiliza para obtener información del usuario en el momento de la autenticación. El método principal que contiene es “*LoadByUsername()*”, que se utiliza para buscar el usuario que inicia sesión en la base de datos de usuarios. Si lo encuentra devolverá el usuario en forma de “*UserDetails*” o detalles de usuario (Ilustración 44).

A screenshot of a code editor showing the implementation of the UserDetailsServiceImpl class. The code is in Java and implements the UserDetailsService interface. It includes an @Service annotation, an @Autowired UserRepository, and an @Override method loadUserByUsername. The method uses the repository to find a user by username and returns a UserDetails object built from the found user.

```
4 pages 1 Macs
@Service
public class UserDetailsServiceImpl implements UserDetailsService {

    1 usage
    @Autowired
    UserRepository userRepository;

    1 usage 1 Macs
    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        User user = userRepository.findOneByUsername(username).get();
        return UserDetailsImpl.build(user);
    }
}
```

Ilustración 44. Servicio de detalles de usuario

- *User Details Impl*: Implementa otra interfaz, en este caso “*UserDetails*”. Es como Spring Security entiende a un usuario. Esta interfaz contiene métodos para la autenticación y autorización. Sus atributos son el nombre de usuario y contraseña de un usuario, y sus métodos son *getters* y *setters* de acceso a esos datos. En la siguiente Ilustración (Ilustración 45), se observan algunos de estos atributos y métodos.

```
public class UserDetailsImpl implements UserDetails {  
    2 usages  
    private String username;  
    2 usages  
    private String password;  
  
    1 usage  ↗ Macs  
    public UserDetailsImpl(String username, String password) {  
        this.username = username;  
        this.password = password;  
    }  
  
    ↗ Macs  
    public static UserDetailsImpl build(User user){  
        return new UserDetailsImpl(user.getUsername(), user.getPassword());  
    }  
  
    1 usage  ↗ Macs  
    @Override  
    public Collection<? extends GrantedAuthority> getAuthorities() { return Collections.emptyList(); }  
  
    ↗ Macs  
    @Override  
    public String getPassword() { return password; }  
  
    ↗ Macs  
    @Override  
    public String getUsername() { return username; }  
  
    no usages  ↗ Macs  
    @Override  
    public boolean isAccountNonExpired() { return true; }  
  
    no usages  ↗ Macs  
    @Override  
    public boolean isAccountNonLocked() { return true; }  
  
    no usages  ↗ Macs  
    @Override  
    public boolean isCredentialsNonExpired() { return true; }
```

Ilustración 45. Implementación detalles de usuario

- **JWT Entry Point:** Esta clase se encarga de gestionar la casuística en la que un usuario no identificado intenta acceder al sistema. En nuestro caso ha sido implementada con el método “*commence*” el cual requiere de tres parámetros. El primero de ellos es la solicitud que resultó en excepción, el segundo es la respuesta HTTP y el tercero es la excepción que se ha lanzado. En la ilustración 46 se observa dicho método.

A screenshot of an IDE showing the implementation of the JwtEntryPoint class. The code is in Java and implements the AuthenticationEntryPoint interface. It includes a @Component annotation and an @Override method named commence. The commence method takes HttpServletRequest req, HttpServletResponse res, and AuthenticationException e as parameters and throws IOException and ServletException. The method body calls res.sendError(HttpServletResponse.SC_UNAUTHORIZED, "Unauthorized");. The IDE interface shows '2 usages' and '1 Macs' for the class, and 'no usages' and '1 Macs' for the method.

```
2 usages 1 Macs
@Component
public class JwtEntryPoint implements AuthenticationEntryPoint {

    no usages 1 Macs
    @Override
    public void commence(HttpServletRequest req, HttpServletResponse res, AuthenticationException e) throws IOException, ServletException {
        res.sendError(HttpServletResponse.SC_UNAUTHORIZED, "Unauthorized");
    }
}
```

Ilustración 46. Punto de entrada JWT

- **JWT Provider:** Se encarga de crear y validar los tokens JWT. Para ello firma los tokens, aplica la fecha de caducidad e incluye el nombre de usuario. A la hora de la comprobación se asegura de que la firma de un token es válida y además de que no ha expirado. En la siguiente Ilustración 47, se muestran algunas partes de esta clase.

```
4 usages  ↗ Macs *
@Component
public class JwtProvider {

    5 usages
    private final static Logger logger = LoggerFactory.getLogger(JwtProvider.class);

    5 usages
    @Value("${jwt.secret}")
    private String secret;

    2 usages
    @Value("${jwt.expiration}")
    private int expiration;

    1 usage  ↗ Macs
    public String generateToken(Authentication authentication) {
        UserDetailsImpl userDetailsImpl = (UserDetailsImpl) authentication.getPrincipal();
        return Jwts.builder()
            .setSubject(userDetailsImpl.getUsername())
            .setIssuedAt(new Date())
            .setExpiration(new Date(new Date().getTime() + expiration * 1800))
            .signWith(getSecret(secret))
            .compact();
    }

    1 usage  ↗ Macs *
    public String getUsernameFromToken(String token) {
        return Jwts.parserBuilder().setSigningKey(getSecret(secret)).build().parseClaimsJws(token).getBody().getSubject();
    }

    1 usage  ↗ Macs
    public boolean validateToken(String token) {
        try {
            Jwts.parserBuilder().setSigningKey(getSecret(secret)).build().parseClaimsJws(token);
            return true;
        }
    }
}
```

Ilustración 47. Proveedor JWT.

- **JWT Token Filter:** Esta clase es un filtro, se encarga de obtener el token de la cabecera de la solicitud, validarlo, y en caso de ser válido, crear un objeto de autenticación en el contexto de seguridad de Spring. La clase se muestra en la ilustración 48.

```
3 usages 1 Macs
@Component
public class JwtTokenFilter extends OncePerRequestFilter {

    1 usage
    private final static Logger logger = LoggerFactory.getLogger(JwtTokenFilter.class);

    2 usages
    @Autowired
    JwtProvider jwtProvider;

    1 usage
    @Autowired
    UserDetailsServiceImpl userDetailsService;

    no usages 1 Macs
    @Override
    protected void doFilterInternal(HttpServletRequest req, HttpServletResponse res, FilterChain filterChain) throws ServletException, IOException {
        try {
            String token = getToken(req);
            if(token != null && jwtProvider.validateToken(token)){
                String username = jwtProvider.getUsernameFromToken(token);
                UserDetails userDetails = userDetailsService.loadUserByUsername(username);

                UsernamePasswordAuthenticationToken auth =
                    new UsernamePasswordAuthenticationToken(userDetails, credentials: null, userDetails.getAuthorities());
                SecurityContextHolder.getContext().setAuthentication(auth);
            }
        } catch (Exception e){
            logger.error("fail en el método doFilter " + e.getMessage());
        }
        filterChain.doFilter(req, res);
    }

    1 usage 1 Macs
    private String getToken(HttpServletRequest request){
        String header = request.getHeader("Authorization");
    }
}
```

Ilustración 48. Filtro JWT.

- **Main Security:** Esta es la clase de seguridad principal. En ella se definen qué rutas son accesibles para el público sin identificar y qué rutas no, además de definir qué filtros y clases utilizará Spring Security y cómo se manejan las excepciones en seguridad. En nuestro caso, todas las rutas están protegidas, a excepción de las rutas del controlador de autorización, ya que no tiene sentido que un usuario deba estar autorizado para autorizarse. Además se dejan libres rutas como la de documentación de Swagger, de la que hablaremos más adelante. En la siguiente ilustración 49 se muestran algunas inyecciones de dependencias y el método principal de definición de los permisos de URLs.


```

1 usage
@Autowired
UserDetailsServiceImpl userDetailsServiceImpl;

1 usage
@Autowired
JwtEntryPoint jwtEntryPoint;

1 usage
@Autowired
PasswordEncoder passwordEncoder;

1 usage
@Autowired
JwtTokenFilter jwtTokenFilter;

2 usages
AuthenticationManager authenticationManager;

no usages  ⚙️ Macs *
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    AuthenticationManagerBuilder builder = http.getSharedObject(AuthenticationManagerBuilder.class);
    builder.userDetailsService(userDetailsServiceImpl).passwordEncoder(passwordEncoder);
    authenticationManager = builder.build();
    http.authenticationManager(authenticationManager);

    http.csrf().disable();
    http.cors();
    http.headers().frameOptions().disable();
    http.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS);

    http.authorizeHttpRequests().requestMatchers(...patterns: "/auth/**").permitAll()
        .requestMatchers(...patterns: "/v3/api-docs/**", "/swagger-ui/**", "/swagger-ui.html").permitAll()
        .requestMatchers(...patterns: "/user/exists/**").permitAll()
        .requestMatchers(...patterns: "/h2-db/**").permitAll()
        .anyRequest().authenticated();
    http.exceptionHandling().authenticationEntryPoint(jwtEntryPoint);
    http.addFilterBefore(jwtTokenFilter, UsernamePasswordAuthenticationFilter.class);
    return http.build();
}

```

Ilustración 49. Filtro JWT.

- Anotaciones de autorización: Una vez hemos implementado todas las clases, ya tenemos acceso al contexto del usuario que tiene la sesión iniciada en la petición. Por lo tanto, podemos definir autenticaciones extras a métodos HTTP. En la ilustración siguiente se observa el método “*getDocument()*”. A este método sólo se puede acceder el creador del documento en caso de que el documento sea privado, o puede acceder cualquier usuario si el documento es público. Por lo tanto, con la anotación “*@PreAuthorize*”, podemos indicar este

condicional para mantener seguro el método. En la ilustración 50 se observa el método con la anotación y el condicional.

```
@PreAuthorize("@documentService.checkOwner(authentication.principal.getUsername(), #id) || @documentService.checkPublic(#id)")
public ResponseEntity<DocumentDTO> getDocument(@PathVariable long id) {
    DocumentDTO document = documentService.getDocument(id).toDto();
    return ResponseEntity.ok(document);
}
```

Ilustración 50. Autorización métodos HTTP.

4.2.1.9. Generación y consulta de documentación API REST

Como sabemos, nuestra aplicación está documentada con Swagger. Para implementar esta documentación los pasos son muy simples. Primero, deberemos agregar las dependencias necesarias al archivo “*pom.xml*”. Las dependencias necesarias se muestran en la ilustración 51.

```
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
    <version>2.0.2</version>
</dependency>
```

Ilustración 51. Dependencias Swagger.

Una vez añadimos estas dependencias, ya encontraremos la documentación necesaria en la URL “*/swagger-ui/index.html*”. Sin embargo, esta documentación carece de descripciones y algunos datos que queremos incluir en la documentación. Para añadir descripciones, tipo de respuestas HTTP según el caso y tipo de dato devuelto a la documentación, será necesario utilizar distintas anotaciones. En nuestro caso utilizamos “*@Operation*” para mostrar un pequeño resumen de la utilidad del método y una descripción más detallada. También utilizamos “*@ApiResponse*s” para

mostrar qué respuesta HTTP devuelve el método y qué tipo de dato devuelve en función de si la petición se ha realizado correctamente o si ha habido una excepción o impedimento durante la operación. En la ilustración 52 se observa un método anotado con todas estas posibilidades.

```

@Operation(summary = "Obtener documentos por género y página", description = "Obtiene los documentos públicos basados en los géneros proporcionados y el fragmento del título.")
@ApiResponses(value = {
    @ApiResponse(responseCode = "200", description = "Documentos devueltos correctamente", content = @Content(array = @ArraySchema(schema = @Schema(implementation = DocumentDTO.class))),
    @ApiResponse(responseCode = "404", description = "Género no encontrado", content = @Content(schema = @Schema(implementation = Void.class)))
})
@GetMapping("/{genres}")
public ResponseEntity<List<DocumentDTO>> getDocumentsByGenreAndPage(

```

Ilustración 52. Anotaciones de Swagger.

4.2.2. Implementación del cliente

En este punto comentaremos los pasos a seguir para la implementación del cliente. Definiremos cómo hemos creado los componentes paso a paso, cómo hemos implementado la seguridad y cómo hemos aplicado el diseño responsive de los componentes, entre otros.

4.2.2.1. Creación del proyecto

La creación de un proyecto en React es muy sencilla. El único requisito necesario es tener instalado Node JS en tu sistema. Una vez que lo tengamos instalado, nos dirigiremos al directorio donde queramos crear el proyecto y ejecutaremos la orden “*npx create-react-app {nombre del proyecto}*”. Una vez que se termine la creación del proyecto, accederemos al directorio y ejecutaremos la orden “*npm start*” para ejecutar la aplicación y desplegarla en el servidor local, todo esto se explica con más profundidad en [19].

4.2.2.2. Estructura del proyecto

La estructura del proyecto se ha dividido en diferentes secciones. En concreto, hemos dividido las clases implementadas en componentes, servicios, guardianes y vistas, tal

y como se observa en la ilustración 53. Las carpetas “*files*” y “*utils*” contienen archivos de prueba y variables globales respectivamente.

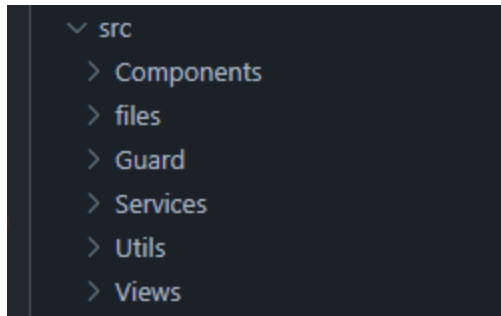


Ilustración 53. Estructura de las carpetas del cliente.

Dentro de la carpeta de componentes y vistas, se encuentran carpetas que dividen los componentes entre sí. Dentro de cada carpeta de componente, se encuentra el archivo JavaScript y el archivo CSS (Ilustración 54) en caso de que el componente lo necesite. También es posible que en un mismo componente existan varios archivos JS que son casos específicos de ese componente, como es el caso de modal, que contiene dentro de la carpeta distintas Modales con diferente contenido.

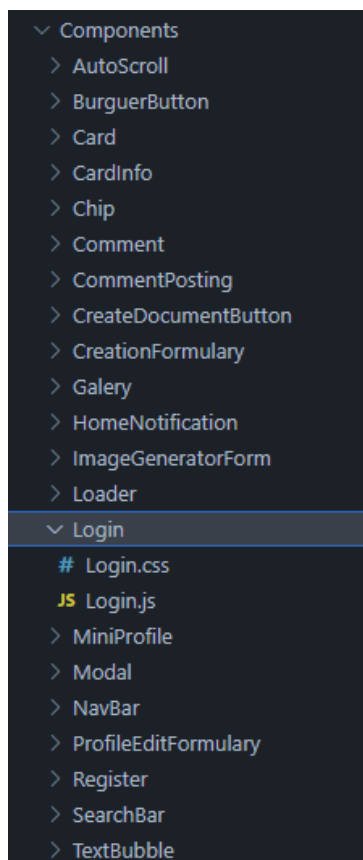


Ilustración 54. Ejemplo organización de los componentes.

A continuación (Ilustración 55) un diagrama de paquetes, donde se observa de forma más general cómo se organizan los elementos. En dicho diagrama se han mostrado los componentes más destacables. Los paquetes se han dividido en cuatro directorios según sus características. Los directorios en cuestión son “Vistas”, donde hemos incluido los componentes de nivel superior, “Componentes” donde se encuentran los componentes que no son vistas, y que a menudo están incluidos en grupo en otros directorios como se puede observar en la ilustración mencionada. También se han dividido en directorios los “Guardianes” y los “Servicios”, que como sabemos se encargan de proteger rutas y contactar con el servidor respectivamente.

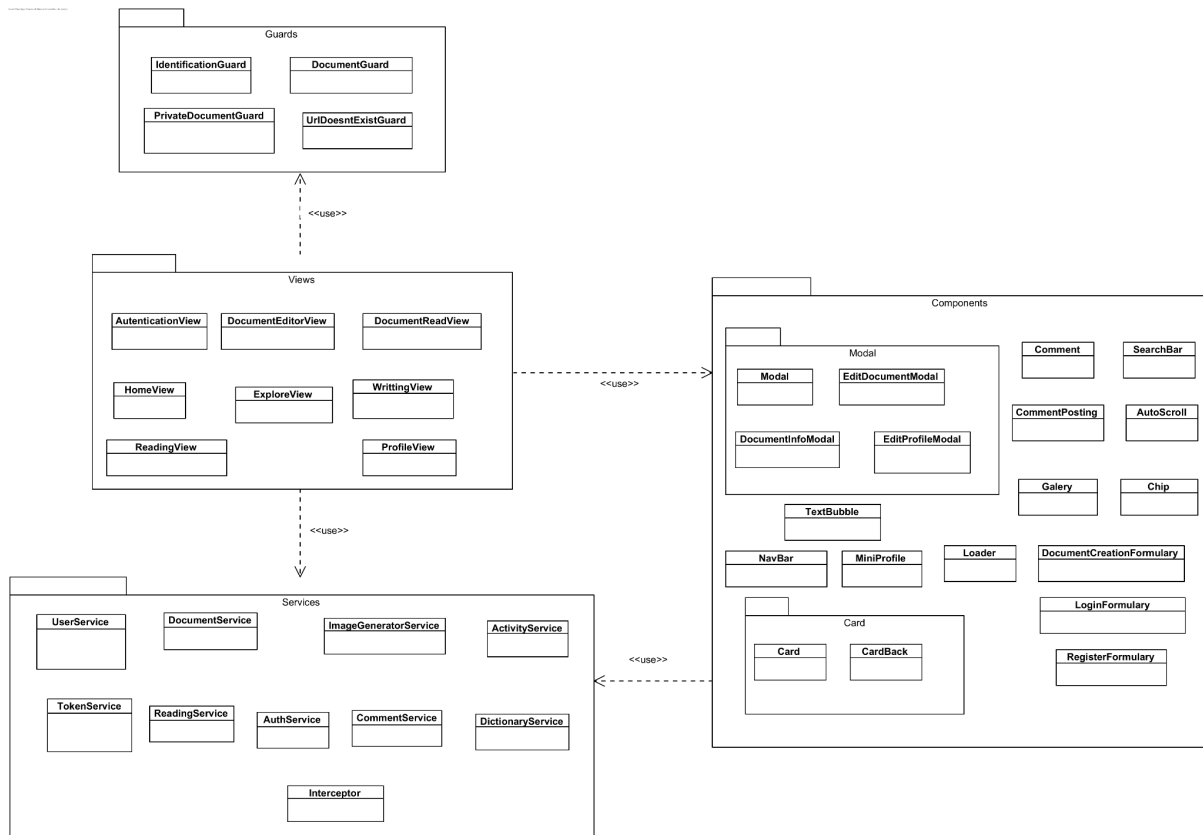


Ilustración 55. Diagrama de paquetes del cliente.

4.2.2.3. Definición de componentes

Para la implementación de un componente en React existen dos enfoques [20]. El primero de ellos es el enfoque de clase, el cual se caracteriza por dotar al componente de un ciclo de vida y utilizar acciones como el montaje, actualización y desmontaje. Además requieren del método render para devolver JSX. El segundo de estos enfoques es el de los componentes funcionales. Este tipo de componentes se implementan como una función de JavaScript y manejan su propio estado. En nuestro caso, hemos elegido el segundo enfoque, el de los componentes funcionales, ya que son más simples, suelen tener mejor rendimiento y dan acceso a los Hooks de estado, que son una herramienta muy útil y flexible.

Un componente funcional se puede dividir en distintas partes:

- Importaciones: Primero se deben importar las dependencias que el componente necesita. Estas dependencias se incluyen en la parte superior del código y comprenden otros componentes, hojas de estilo, funciones de servicios, dependencias de librerías, etc. Algunos ejemplos de código se encuentran en la ilustración 56, donde se han importado archivos de estilos en este caso “*TextBubble.css*”, Hooks de React como “*Use State*” o “*Use Effect*”, funciones de servicios como “*get definition*” del servicio “*Dictionary Service*”, otros componentes como “*Loader*” o iconos mediante la librería “*Material-UI*”.

```
import React, { useState, useEffect } from 'react';
import { getDefinition } from '../Services/DictionaryService';
import Loader from '../Loader/Loader';
import './TextBubble.css';
import CloseIcon from '@mui/icons-material/Close';
import ArrowBackIosIcon from '@mui/icons-material/ArrowBackIos';
import ArrowForwardIosIcon from '@mui/icons-material/ArrowForwardIos';
```

Ilustración 56. Ejemplo definición de propiedades

- Propiedades: Estas propiedades, como se explica en [21], son equiparables a parámetros que el componente padre pasa al componente en cuestión. Pueden ser objetos, cadenas, enteros, etc. Se encuentran en la cabecera del componente y se pasan desde componentes superiores como parámetro, siendo parámetros de sólo lectura. En la ilustración 57 se muestran algunas propiedades, en este caso del componente modal. El componente Modal es una ventana emergente que puede tener cualquier código JSX como contenido. Se ha definido con varias propiedades, que reflejan bastante bien el uso general de esta funcionalidad:
 - *Children*: Es de las propiedades más importantes y es especial, puesto que es una palabra reservada en React. Esta propiedad no se pasa

como parámetro como tal, si no que se refiere al código JSX que se incluye entre las etiquetas del componente.

- *Modal State*: Esta propiedad es un estado que indica si la modal es visible o no.
- *set Modal State*: Esta función cambia el valor del estado de la modal. Se utiliza para deshabilitar la Modal al pulsar sobre el botón de cerrar. Como vemos, las propiedades también pueden ser funciones del componente padre que pueden ser usadas por el componente hijo.
- *Title*: Es el título de la Modal. Esta propiedad nos ayuda a que la modal sea reutilizable, ya que podemos aplicar diferentes títulos al mismo componente.
- *full Screen*: Este prop indica si la Modal deberá ocupar toda la pantalla en caso de estar empleado un dispositivo móvil.

```
const Modal = ({ children, modalState, setModalState, tittle, fullscreen })
```

Ilustración 57. Ejemplo definición de propiedades

A continuación, mostraremos también cómo se llama a este componente desde el componente padre y se introducen los parámetros correspondientes para asignar valor a las propiedades. En la Ilustración 58 se observa este proceso. Como vemos, en la etiqueta de la Modal introducimos los valores de cada propiedad comentada anteriormente. Notamos que falta por asignar la propiedad Children, pero se debe recalcar que esta propiedad hace referencia a los elementos que sean hijos o estén dentro de la etiqueta “<Modal></Modal>”, por lo tanto esa propiedad sería el código JSX en el interior de las etiquetas.


```

return (<
  <Modal modalState={editModalState} setModalState={setEditModalState} title={"Edit document"} fullscreen={true}>
    <div className="creation-formulary-modal">
      <CreationFormulary card={card} title={title} cover={cover} synopsis={synopsis} setCover={setCover} setTitle={setTitle} setSynopsis={setSynopsis} selectedGenres={genres} />
    </div>
    <div className="center">
      <div className="buttons-line">
        <button className="button" onClick={() => setEditModalState(false)}>Cancel</button>
        <button className="button" onClick={updateDocument}>Save</button>
        <button className="button" onClick={moveToEdit}>Content</button>
      </div>
    </div>
  </Modal>
</>)

```

Ilustración 58. Ejemplo definición de propiedades

- Estado: Con el Hook “*Use State*” controlamos el estado de los componentes. Se colocan dentro de la función del componente, normalmente acto seguido de la cabecera. En la ilustración 59 se muestran dos ejemplos de “*Use State*” de un componente simple. El componente se utiliza para subir una reseña y los estados que se muestran son el texto que contendrá la reseña cuando se publique y la puntuación que tendrá. Como vemos, se le puede asignar un valor inicial a estos estados, en nuestro caso le colocamos valores que vienen desde las propiedades. Incluimos un operador “*OR*” que se utiliza para indicar que si el valor por defecto que se pasa por las propiedades es nulo, se deberá colocar como valor por defecto el otro lado del operador, en el primer caso si el texto del comentario es nulo, entonces el texto se colocaría como vacío. Esto se debe a que este componente se utiliza tanto para editar como crear comentarios, por lo que si estás editando un comentario el valor por defecto del texto sería el comentario sin editar, mientras que en caso de no estar editando si no creando, el valor por defecto será vacío.

```

const CommentPosting = ({ documentId, reloadContent, setEditing, editing, comment, isReply, respondingTo }) => {
  const [text, setText] = useState(comment?.text || '');
  const [review, setReview] = useState(comment?.rating || 0);

```

Ilustración 59. Ejemplo definición Use State

- Efectos: Los “*Use Effect*” se utilizan para ejecutar código ante el cambio de valor de una variable o estado. El parámetro observado se coloca entre llaves

al final del efector. Cuando el valor del parámetro cambia se ejecuta el código. En la ilustración 60 se observa un ejemplo de efector en el que al cambiar el valor de “*card.creatorUsername*”, es decir, al cambiar el valor del nombre de usuario que ha creado el documento, se llama a una función que pedirá los datos del nuevo usuario a través de ese nombre de usuario y se guardará la imagen del usuario en un estado para actualizar el componente renderizando dicha imagen.

```
useEffect(() => {  
  if (card?.creatorUsername) {  
    getUser(card?.creatorUsername).then(data => {  
      setCreatorPicture(data?.image);  
    });  
  }  
}, [card?.creatorUsername]);
```

Ilustración 60. Ejemplo de Use Effect

- **Funciones de ayuda:** Las funciones de ayuda son funciones que nos ayudan a realizar operaciones o manejar eventos y que se implementan en el interior del componente. En la ilustración 61 se observan funciones de este tipo. En concreto, estas funciones se encargan de gestionar un evento y guardar información sobre el estado del componente. Nos referimos por evento a una interacción entre el usuario y la interfaz, como pulsar un botón de la interfaz, pulsar alguna tecla del ratón o teclado o escribir texto en algún campo específico.

```
const onUserChange = (value) => {  
  setUser(value);  
  setLoginError(false);  
}  
  
const onPasswordChange = (value) => {  
  setPassword(value);  
  setLoginError(false);  
}
```

Ilustración 61. Ejemplo 1 de Helper Functions

Otros ejemplos son las siguientes funciones (Ilustración 62), las cuales se encargan de operar con el índice de definiciones y cambiar el estado de la aplicación para mostrar una u otra definición de una palabra

```
const handleNextDefinition = () => {  
  if (currentDefinitionIndex < definitions.length - 1) {  
    setCurrentDefinitionIndex(currentDefinitionIndex + 1);  
  }  
};  
  
const handlePreviousDefinition = () => {  
  if (currentDefinitionIndex > 0) {  
    setCurrentDefinitionIndex(currentDefinitionIndex - 1);  
  }  
};
```

Ilustración 62. Ejemplo 2 de Helper Functions

- **Return con JSX:** Esta parte contiene el código que luego será renderizado. También se puede incluir lógica con JavaScript para decidir qué se debería renderizar en qué casos. En la ilustración 63 se observa un fragmento de código JSX del componente comentario con distintos condicionales y algunos bucles. Para poder utilizar las expresiones JSX en React para aplicar lógica de JavaScript al código, deberemos usar llaves. En la primera parte de este

ejemplo vemos un atributo de tipo booleano llamado “*isReply*”, que indica si el comentario es una respuesta o no. En caso de no ser una respuesta, se mostrará el código JSX que se encuentra junto al operador “AND”. En caso contrario, ese código no se ejecutará ni aparecerá en el renderizado. Vemos también que dentro de ese fragmento se incluye un bucle que renderiza tantas estrellas rellenas como puntuación de la reseña y el resto las renderiza como estrellas vacías. El último fragmento de código contiene una comprobación que asegura que los iconos de edición y borrado sólo aparezcan sobre el comentario si el creador del comentario (“*comment.postedBy*”) es igual al usuario que está usando la interfaz (*getUsername* de *tokenService*).

```
    </div>
    {!isReply && <div className='comment-stars'>
      {(() => {
        const stars = [];
        for (let i = 1; i <= 5; i++) {
          if (i <= comment?.rating) {
            stars.push(<StarIcon className='star-icon' key={i} />);
          } else {
            stars.push(<StarOutlineIcon className='star-icon' key={i} />);
          }
        }
        return stars;
      })()}
    </div>}
    {comment?.postedBy === getUsername() && <div className='center'>
      <DeleteIcon onClick={() => setDeleteModal(true)}></DeleteIcon>
      <EditIcon onClick={() => setEditing(true)}></EditIcon>
    </div>}
```

Ilustración 63. Ejemplo de código JSX.

Ahora se mostrará en la ilustración 64 un componente más simple en el que se pueda ver el return completo a simple vista y en el que se tienen los mismos elementos de control. También se observa el uso de eventos mediante el parámetro “*onChange*” u “*onClick*”. Si añadimos una función que maneje este

evento, podremos obtener toda la información sobre los cambios producidos en el elemento JSX correspondiente a esa etiqueta.

```
return (<>
  <div className='comment-posting'>
    <div className="center">
      <input onChange={(event) => setText(event.target.value)} defaultValue={text} className='comment-text-area' />
    </div>
    {!isReply && <div className='row center posting-stars'>
      {[...Array(5)].map((_, i) => {
        <div
          key={i}
          onClick={() => handleRatingClick(i + 1)}
        >
          {i + 1 <= review ? <StarIcon /> : <StarOutlineIcon />}
        </div>
      )]}
    </div>
    {!editing ? (
      <div className="center">
        <button className="button" onClick={postingComment}>Post comment</button>
      </div>) : (
      <div className="center">
        <div className="center gap">
          <button className="button" onClick={postingComment}>Save changes</button>
          <button className="button" onClick={() => setEditing(false)}>Cancel</button>
        </div>
      </div>
    )}
  </div>
</>)
```

Ilustración 64. Ejemplo de código JSX.

4.2.2.4. Estilos y responsividad

En puntos anteriores sobre el código JSX, se ha podido observar el uso de la propiedad “*ClassName*” en algunos fragmentos. Esta propiedad se utiliza para aplicar una clase CSS a un elemento JSX, sería lo equivalente a “*class*” en HTML. Para poder crear una clase de estilos es necesario crear un archivo CSS. Recordamos que CSS es un lenguaje de hoja de estilos que se utiliza para describir la apariencia de elementos, los cuales pueden estar definidos en diferentes lenguajes. Algunas de las características que nos permite aplicar son fuentes de texto, colores y fondos, diseños de página y efectos y animaciones.

Para saber cuándo crear un archivo CSS hemos seguido un criterio concreto, el cual se fundamenta en la utilización del CSS global de la aplicación “*App.css*” para crear clases que se utilicen por toda la aplicación, como podría ser botones que mantienen

el mismo estilo aunque estén en distintas vistas, fuentes de texto que se mantienen constantes independientemente el componente, o clases que posicionan elementos en lugar recurrentes en muchas de las vistas de la aplicación, como podría ser un contenedor centrado. De esta forma, nos ahorramos el tener que repetir el código CSS en cada componente.

Para los componentes más específicos, se han creado sus propios archivos CSS, como podría ser el caso de la tarjeta de documento que muestra información sobre los documentos. Otro caso en el que se han creado archivos CSS es para un conjunto de componentes, como podrían ser la vista de edición de texto y la vista de lectura, que al compartir gran número de sub componentes, pueden compartir el archivo que les aplica estilo.

Para definir una clase se sigue el proceso que se muestra en la ilustración 65, en el que se nombra la clase y mediante llaves se incluye en su interior las propiedades que se quieren modificar y el valor de las mismas. En el ejemplo se muestra la clase de estilos principales de la barra de navegación. Se observa que se aplican valores a propiedades como el ancho, el alto, el color de fondo, o a propiedades no tan básicas como la posición con el valor sticky, que mantiene la barra de navegación en el mismo punto de la pantalla a pesar del desplazamiento de la ventana o la propiedad de justificación de contenido, que alinea el contenido al valor especificado.

```
.nav {  
  width: auto;  
  background-color: #333;  
  color: #F3F3F3;  
  display: flex;  
  justify-content: space-between;  
  align-items: center;  
  gap: 2rem;  
  padding: 0 1rem;  
  position: sticky;  
  top: 0px;  
  z-index: 2;  
  height: 3.25rem;  
}
```

Ilustración 65. Ejemplo de código CSS.

Una funcionalidad clave de los estilos es que nos permiten hacer la aplicación “responsive”, es decir, adaptarse al tamaño de la pantalla y mostrar diferentes estilos según el espacio disponible. Para ello utilizaremos tres técnicas que se explican en [22]:

- Layout flexible: Esto refiere a utilizar unidades relativas como porcentajes o root em¹⁸, en lugar de píxeles cuando sea necesario, lo que ayuda a que el tamaño de los elementos se ajuste al tamaño del espacio disponible.
- Grid y Flexbox: Estas son dos técnicas de diseño que facilitan el diseño adaptable. Grid ayuda a crear elementos en forma de malla o matriz, en el que se distinguen filas y columnas, lo que lo hace útil para diseños en dos dimensiones, mientras que flexbox es útil en diseños de una dimensión.

¹⁸Una unidad “root em” tiene el valor en píxeles del texto base del documento y es una unidad que se considera relativa.

- Media-Queries: Esta es quizás la técnica más importante. Utilizando media-queries se puede aplicar distintos estilos a un mismo elemento en función del tamaño de pantalla del dispositivo, lo que lo hace perfecto para el objetivo que buscamos. También se puede utilizar por ejemplo para aplicar estilos específicos al documento en la ventana de impresión, funcionalidad que utilizaremos para poder imprimir los documentos de forma correcta. Un ejemplo de uso de esta técnica se observa en el código de la Ilustración 66, donde se observa una *query* que se aplica cuando el tamaño de la pantalla es menor a 1500 píxeles. Como se observa en la imagen, además de ajustar las propiedades de la clase “image-generator”, se hace visible el elemento “generator-toggle-button”, que como se observa fuera del elemento media, en pantallas grandes no es visible.

```
6 .generator-toggle-button {  
7   position: fixed;  
8   bottom: 1rem;  
9   right: 1rem;  
10  color: #333;  
11  cursor: pointer;  
12  visibility: hidden;  
13 }  
14  
15 @media (max-width: 1500px) {  
16   .image-generator {  
17     position: fixed;  
18     top: 50%;  
19     left: 50%;  
20     transform: translate(-50%, -50%);  
21     width: 20rem;  
22     background-color: #F3F3F3;  
23     box-shadow: 0 0 5px 0 rgba(0, 0, 0, 0.5);  
24   }  
25  
26   .generator-toggle-button {  
27     visibility: visible;  
28   }  
29 }
```

Ilustración 66. Ejemplo de Media Query CSS.

4.2.2.5. Servicios y peticiones al servidor

Una vez hemos comentado cómo se implementa un componente típico, es momento de pasar a otro tipo de módulos más específicos: Los servicios. Los servicios son módulos de código que se utilizan para separar la lógica de peticiones al servidor de la lógica de los componentes, es decir, son un conjunto de funciones y herramientas para la comunicación con el servidor.

Aunque existen varias formas de crear estos servicios, nosotros hemos optado por implementarlos como un conjunto de funciones, las cuales tienen una característica especial, y es que son asíncronas. Esto significa que en lugar de devolver un objeto o un dato, devuelve una promesa, lo que se traduce en que se deberá pausar la ejecución hasta que la promesa se cumpla o sea rechazada. Para ello, al llamar al servidor se hará uso de la palabra clave *“await”*.

Además de estas funciones, se incluirá una constante con la URL a la que se deben hacer las peticiones. En el caso de que las peticiones se realicen a la API REST que hemos implementado, estas deberán pasar por el interceptor, del cual hablaremos más adelante. En caso de que las peticiones se realicen a una API externa no será necesario el paso por el interceptor. Para las peticiones al servidor, se ha utilizado la biblioteca de React *“Axios”* [23], la cual es una alternativa a la función por defecto *“fetch”* a la hora de realizar peticiones al servidor ya que ofrece diversas ventajas y felicitades, como más simplicidad, seguridad y funcionalidades adicionales.

En la ilustración 67 se muestra un fragmento de las funciones de un servicio, en concreto el servicio documento. Se muestran dos funciones del servicio, la primera de ellas es una petición al servidor de tipo GET, en la que se pide un documento por su identificador y la segunda es una petición de tipo POST donde se envía una petición

para crear un documento en el servidor. Como observamos en la ilustración, se pueden ver las características ya comentadas anteriormente, como la palabra clave “*async*” que indica que el método es asíncrono o la palabra clave “*export*” que nos permite incluir la función en otros componentes mediante un *include*. La petición se realiza mediante el *interceptor*, al que le indicamos el tipo de petición que vamos a realizar y como parámetros le pasaremos la URL y el *body* de la petición en formato JSON.

```
const url = "documents";

export async function getDocumentById(id) {
  try {
    const response = await Interceptor.get(`${url}/${encodeURIComponent(id)}`);
    return response.data;
  } catch (error) {
    if (error.response && error.response.status === 401) {
      console.log("El token expiró. Actualizando token...");
    } else {
      console.log("Ocurrió un error al obtener el documento:", error);
    }
  }
}

export async function postDocument(document) {
  try {
    const response = await Interceptor.post(url, JSON.stringify(document));
    return response.data;
  } catch (error) {
    if (error.response && error.response.status === 401) {
      console.log("El token expiró. Actualizando token...");
    } else {
      console.log("Ocurrió un error al obtener el documento:", error);
    }
  }
}
```

Ilustración 67. Ejemplo funciones servicio.

Para realizar las peticiones a estos servicios desde componentes, se deben llamar de una forma en específico. Lo primero será incluir las funciones del servicio que queremos usar mediante un *include* como hemos mencionado anteriormente y se observa en la siguiente ilustración 68. Como vemos, simplemente se indica qué función queremos incluir en el componente y la ruta en la que se encuentra el servicio.

```
import { getDocumentById } from "../../Services/DocumentService";
import { putReading, postReading, getReading, checkReading } from "../../Services/ReadingService";
```

Ilustración 68. Importar funciones de un servicio.

Una vez tengamos el servicio incluido en el componente, el siguiente paso será hacer las llamadas a la función donde se necesite. Al ser una función que devuelve una promesa es necesario incluir la cláusula “*then*”. El bloque de código dentro de este método se ejecutará cuando la promesa haya sido resuelta y se haya obtenido un resultado. En el caso del ejemplo de la ilustración 69, se llama a la función “*getDocumentById*”. Dentro del método “*then*” se incluye el nombre de la variable que contendrá el resultado de la promesa, es decir, el resultado de la petición al servidor, que en nuestro caso sabemos que es un documento.

Una vez se ha completado la promesa se entra dentro del bloque de código del método y se utiliza la variable *data* para asignar los valores que correspondan y actualizar el estado del componente con la nueva información.

```
getDocumentById(documentId).then(data => {  
  quill.root.innerHTML = data.text;  
  setDocuData(data);  
});
```

Ilustración 69. Ejemplo llamada a un servicio desde un componente.

4.2.2.5. Interceptor e identificación de peticiones

En el punto anterior hemos comentado que los servicios hacen uso del interceptor para hacer las peticiones al servidor, así que nos centraremos ahora en explicar cómo se ha implementado el interceptor y cual es su funcionamiento.

El interceptor se encarga de, como su nombre indica, interceptar todas las peticiones que se realizan al servidor para insertar un token de identificación que permitirá al servidor reconocer al usuario en la petición. También se encarga de recibir la respuesta del servidor y decidir si esta es válida o el token debe ser renovado.

Mostraremos primero la implementación del interceptor en la Ilustración 70 y luego procederemos a una explicación detallada.

```
import axios from 'axios';
import { getToken, setToken, refreshToken } from './TokenService';
import { logoutUser } from './AuthService';

const Interceptor = axios.create({
  baseURL: 'http://localhost:8080/',
});

// Añado el token a la cabecera de todas las peticiones.
Interceptor.interceptors.request.use(
  (config) => {
    const token = getToken();
    if (token) {
      config.headers['Authorization'] = `Bearer ${token}`;
      config.headers['Content-Type'] = 'application/json';
    }
    return config;
  },
  (error) => {
    return Promise.reject(error);
  }
);

// Si el token está caducado se pide otro y se envía la petición de nuevo
Interceptor.interceptors.response.use(
  (response) => response,
  async (error) => {
    if (error.response?.status === 401 && error.config._retry) {
      error.config._retry = true;
      const newToken = await refreshToken();
      setToken(newToken.data.token);
      error.config.headers['Authorization'] = `Bearer ${newToken.data.token}`;
      return axios(error.config);
    } else if (error.config._retry) {
      logoutUser();
    }
    throw error;
  }
);
```

Ilustración 70. Métodos del interceptor

Comenzando con la explicación, se puede observar en la ilustración 70 como el interceptor hace uso de la biblioteca “Axios” mencionada anteriormente y de métodos del servicio “Token Service”. También vemos que el interceptor se crea con la raíz de la URL del servidor, de esta forma no es necesario repetir este fragmento de ruta en cada servicio.

A la hora de hablar sobre los métodos que se aplican al interceptor, se observan dos de ellos:

- Método de peticiones: Este método obtiene el token del servicio de los tokens y lo inserta en la cabecera siempre que se hace una petición mediante el interceptor.
- Método de respuestas: Este método intercepta la respuesta del servidor. En caso de que la respuesta sea de error 401, se llamará a la función de refresco de token del servidor. Si la petición devuelve un nuevo token, este token se guardará en el servicio del token, mientras que si vuelve a devolver error 401, se asumirá que el token no es válido y se cerrará la sesión para el usuario.

Mostraremos también los métodos del servicio del token y los comentaremos, en la ilustración 71 se observan sus métodos principales.

```
import jwt_decode from 'jwt-decode';
import axios from 'axios';

export function setToken(token) {
  localStorage.setItem("token", token);
}

export function getToken() {
  return localStorage.getItem("token");
}

export async function refreshToken() {
  const token = getToken();
  return axios.post('http://localhost:8080/auth/refresh', { "token": token });
}

export function logged() {
  if (getToken()) return true;
  else return false;
}

export function getUsername() {
  if (!logged()) return null;
  try {
    const decodedToken = jwt_decode(getToken());
    return decodedToken.sub;
  } catch {
    return "";
  }
}

export function dropToken() {
  localStorage.removeItem('token');
}
```

Ilustración 71. Métodos de servicio de tokens.

Como se observa en la ilustración 71, hacemos uso de la librería “*jwt_decode*” para obtener los datos necesarios del token, principalmente el nombre del usuario que ha iniciado sesión y está vinculado al token. Otra cosa a destacar es que almacenamos el token en el almacenamiento local, el cual es un objeto de JavaScript que nos proporciona una forma sencilla de almacenar datos en la web de forma persistente. Eso significa que, aunque el usuario cierre el navegador, no requerirá de volver a iniciar la sesión ya que esta se almacena en el navegador. Para almacenar información en este objeto se debe guardar un par (clave, valor), en nuestro caso la clave del token es “*token*” y el valor del mismo es la cadena que recibimos desde el servidor.

4.2.2.6. Guardianes

Para acabar con la implementación del cliente, hablaremos sobre los guardianes. Los guardianes, como comentamos anteriormente en el diseño, son componentes que se encargan de proteger las rutas a las que no todos los usuarios pueden acceder, ya sea porque un documento es privado, porque el usuario no está logueado o porque intenta acceder a una ruta que no existe.

Para los guardianes que protegen vistas, como podría ser el caso de un guardián que controla el acceso a un documento. En la ilustración 72 se observa parte del enrutamiento de la aplicación, donde se puede ver cómo para algunas direcciones, las etiquetas de las vistas están contenidas dentro de las etiquetas de un Guardián o incluso de varios.

```
<Route
  path="/explore"
  element={<Guard>
    <Explore />
  </Guard>}
/>
<Route
  path="/write"
  element={<Guard>
    <Writing />
  </Guard>}
/>
<Route
  path="/documents/:id"
  element={<Guard>
    <DocumentGuard>
      <DocumentEdit />
    </DocumentGuard>
  </Guard>}
/>
<Route
  path="/documents/read/:id"
  element={<Guard>
    <PrivateDocumentGuard>
      <DocumentRead />
    </PrivateDocumentGuard>
  </Guard>}
/>
```

Ilustración 72. Enrutamiento de la aplicación cliente.

El hecho de que las vistas se aniden dentro de los guardianes se deben a que este tipo de guardianes contienen la propiedad “Children” comentada en el punto anterior sobre la implementación de los componentes, más específicamente en el punto de propiedades.

Al acceder a esa URL, el guardián realizará diferentes comprobaciones según el tipo de guardián que sea. En el caso de que el usuario esté autorizado, el guardián dará acceso a su componente hijo. En caso de que el usuario no esté autorizado, el guardián redirigirá al usuario a otro componente en el que se le indique información sobre este hecho.

En la ilustración 73 se observa el código de un guardián, en concreto el guardián que comprueba si un usuario puede editar un documento, es decir, si el usuario es el creador del documento. El primer paso que realiza el guardián es obtener la ID del documento a partir de la URL. Una vez ha conseguido el identificador del documento,

envía una petición al servidor a través del servicio documento. En caso de que la respuesta del servidor confirme que el usuario es el autor del documento, se le dará acceso a la vista, devolviendo “Children”, que sabemos que es la vista que se coloca dentro del guardián en el enrutamiento. En el caso contrario, se devolverá un componente “GuardView” junto con un mensaje que indica el motivo por el que se deniega el acceso, además de acompañarlo con un icono específico según el motivo de la redirección. Mientras que el guardián hace las comprobaciones necesarias, se mostrará en pantalla un componente cuya función es meramente estética e informa al usuario de que la aplicación está cargando.

```
const DocumentGuard = ({ children }) => {
  const { id } = useParams();
  const [isDocumentMine, setIsDocumentMine] = useState(null);

  useEffect(() => {
    checkOwner(id).then((result) => {
      setIsDocumentMine(result);
      console.log("GUARD DOCUMENTO RECIBE " + result);
    });
  }, [id]);

  if (isDocumentMine === null) {
    return <div className='center'><Loader /></div>;
  }

  if (!isDocumentMine) {
    return <GuardView message={"Only the creator of the document is able to edit it"} icon={"lock"}></GuardView>
  }

  return children;
};

export default DocumentGuard;
```

Ilustración 73. Ejemplo llamada a un servicio desde un componente.

4.2.2.7. Validaciones en el cliente

Una de las funcionalidades clave a la hora de implementar formularios son las validaciones. Las validaciones son comprobaciones que el cliente realiza sobre los datos de entrada, sobre todo en formularios, y con las que se asegura de que los datos introducidos son válidos.

Existen dos tipos de validaciones, las sincrónicas, las cuales se realizan en el cliente sin necesidad de hacer peticiones al servidor y las asíncronas, validaciones las cuales necesitan información del servidor para comprobar si los datos introducidos son válidos o no.

Un ejemplo de validación síncrona podría ser el comprobar que una contraseña contiene más de ocho caracteres. Esta comprobación se realizaría en el cliente mediante el uso de eventos y comprobando la longitud de los datos de entrada. Por el contrario, a la hora de saber si un nombre de usuario no existe y puede ser usado, será necesario realizar una petición al servidor para obtener la información sobre ese nombre de usuario.

En nuestro caso, las validaciones síncronas, que son las que más utilizamos, se han realizado mediante el uso de eventos y comprobaciones previas al envío de la petición. En la figura 74 se observan varios métodos de validación. Mediante la gestión de los eventos, cuando un usuario escribe por ejemplo en el campo del correo electrónico, se llama a su respectiva comprobación. En caso de que no se cumplan con los requisitos necesarios, una variable de estado que muestra el error del campo pasará de estar vacía a mostrar un mensaje de error al usuario.

```
const onEmailChange = (value) => {
  const emailRegex = /^[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,}$/i;
  const isValidEmail = emailRegex.test(value);
  setEmail(value);
  if (isValidEmail) {
    setEmailError("");
  } else {
    setEmailError("Use a valid email");
  }
}

const onPasswordChange = (value) => {
  setPassword(value);
  if (value.length < 5) {
    setPasswordError("The password must have at least 5 characters");
  } else {
    setPasswordError("");
  }
}

const onConfirmPasswordChange = (value) => {
  setConfirmPassword(value);
  if (value !== password) {
    setConfirmPasswordError("The passwords do not match");
  } else {
    setConfirmPasswordError("");
  }
}
```

Ilustración 74. Ejemplo funciones para la gestión de eventos.

A la hora de realizar la petición, esta sólo se realizará si ninguno de los campos muestra error, es decir, si todas las variables de estado relacionadas con los errores están vacías. Esto se implementará en la función que gestiona el evento de pulsación del botón de registro. Tal y como se muestra en la ilustración 75, primero se comprueba que ningún campo de error ni esté vacío, luego se crea el objeto de transferencia de datos con el que se enviará la información del usuario registrado y finalmente se envía mediante un método del servicio de autorización.

```
if (!blankField && userError === "" && emailError === "" && passwordError === "" && confirmPasswordError === "") {  
  // Convierto la imagen a encode64 para su envío  
  imageToBase64(  
    `https://ui-avatars.com/api/?name=${name}&background=333&color=f3f3f3&size=512`  
  )  
  .then((base64String) => {  
    const userObject = {  
      username: name,  
      password: password,  
      profileImage: base64String,  
    };  
  
    // Registro el usuario  
    registerUser(userObject)  
      .then((data) => {  
        console.log(data);  
        props.setInRegister(false);  
      })  
  })  
}  
};
```

Ilustración 75. Validación y envío de datos.

En lo que respecta a las validaciones asíncronas, se observa en la ilustración 76 un método que comprueba si el nombre de usuario que se ha introducido ya existe en la base de datos o no. Primero se comprueba que el nombre de usuario tenga al menos una longitud de cuatro, lo que sería una validación síncrona. En caso de que este sea el caso, entonces se pedirá al servidor mediante el servicio usuario que compruebe si el nombre de usuario está libre o no. En caso de que esté en uso, se muestra un mensaje de error indicando al usuario y se restringe el envío del formulario hasta que la validación se cumpla. En caso de que el usuario pase a escribir un nombre de usuario libre, el error desaparecerá y se permitirá el envío de la petición de registro.

```
const onUsernameChange = (value) => {  
  setName(value);  
  if (value.length < 4) {  
    setUserError("The username must have at least 4 characters");  
  } else {  
    checkUser(value).then((result) => {  
      if (!result) {  
        setUserError("");  
      } else {  
        setUserError("This username is already in use");  
      }  
    })  
  }  
}
```

Ilustración 76. Ejemplo de validación asíncrona.

4.2.2.8. Despliegue de la API de generación de imágenes.

Para la API de generación de imágenes decidimos desplegar alguna de las APIs de código abierto que existen actualmente. En concreto, optamos por desplegar una API que utiliza “*Stable Diffusion*” llamada “*Diffuzzers*”. Esta librería incluye código implementado en Python que se puede utilizar para desplegar una API que genera imágenes al recibir peticiones HTTP sin necesidad de programar, ya que la biblioteca incluye todo lo necesario para su ejecución inmediata.

A la hora de desplegar esta API, nos hemos encontrado con varios problemas. El primero de ellos es un equipo donde desplegarla, ya que este tipo de inteligencias artificiales requieren de una gran cantidad de recursos de los que no disponemos, principalmente requieren una GPU de alto rendimiento. Para solucionar este problema, optamos por utilizar “*Google Colab*”, una herramienta que proporciona Google para programar en la nube y que nos otorga una máquina de forma remota con prestaciones suficientes para ejecutar la API en cuestión.

Tras descargar la librería e incluir el código necesario para su ejecución, nos encontramos con un segundo problema. Al estar la API desplegada en una máquina remota, no podemos acceder a ella por medio de “*LocalHost*” o el servidor local, ya que nuestra máquina y la máquina de ejecución de la API son distintas. Para solucionar esta problemática, utilizamos “*Ngrok*”, la cual es una herramienta de código abierto que permite la creación de túneles seguros desde una red pública a un servidor local. En nuestro caso, utilizamos la página web del servicio para generar un token que, mediante código, nos permitió utilizar la herramienta para crear un túnel que nos da acceso al uso de la API.

El único inconveniente es que la URL a la API es dinámica, por lo que cada vez que la API se detiene y se vuelve a iniciar, la URL a la que se deben realizar las peticiones cambia. Aún así, las peticiones funcionan correctamente y al ser un prototipo este problema se puede solucionar en futuras versiones.

A continuación, se muestra el código implementado “*Google Colab*”. Como se observa en la ilustración 77, primero se instala la librería “*diffuzzers*” mediante “*pip install diffuzzers*”. Acto seguido, se instala “*Ngrok*” y se inicializan las “*environment variables*” o variables de entorno.

Tras la inicialización, se le inserta a “*Ngrok*” el token de acceso generado en la página oficial y se le encarga abrir un túnel en el puerto 10000, aunque sería válido cualquier puerto libre. Una vez configurado, ya sólo queda desplegar la API en el puerto con el túnel, en nuestro caso el 10000, y se podrán realizar peticiones a la API sin mayor problema. Al ejecutar todo el código, se nos otorgará una URL que será válida hasta el reinicio de la API donde podemos enviar las peticiones que queramos.

La librería de la API incluye además documentación con Swagger para conocer qué peticiones permite y qué tipo de datos requieren, además de especificar qué tipos de respuestas ofrece. La documentación es accesible a través de “`{URL}/docs`”, siendo URL la ruta generada en la ejecución de la API en el cuaderno de Google Colab.

```
[12] !pip install diffusers -qqq
```

```
[13] #Environment Variable
```

```
%env XZIMG_MODEL=stabilityai/stable-diffusion-2-1
```

```
env: XZIMG_MODEL=stabilityai/stable-diffusion-2-1
```

```
[15] %env DEVICE=cuda
```

```
env: DEVICE=cuda
```

```
[16] !pip install pyngrok
```

Looking in indexes: <https://pypi.org/simple>, <https://us-python.pkg.dev/colab-wheels/public/simple/>

Requirement already satisfied: pyngrok in /usr/local/lib/python3.10/dist-packages (5.2.1)

Requirement already satisfied: PyYAML in /usr/local/lib/python3.10/dist-packages (from pyngrok) (6.0)

```
[17] from pyngrok import ngrok
```

```
!wget https://bin.equinox.io/c/4VmDzA7iaHb/ngrok-stable-linux-amd64.zip
```

```
--2023-06-21 16:43:43-- https://bin.equinox.io/c/4VmDzA7iaHb/ngrok-stable-linux-amd64.zip  
Resolving bin.equinox.io (bin.equinox.io)... 54.237.133.81, 18.205.222.128, 54.161.241.46, ...  
Connecting to bin.equinox.io (bin.equinox.io)[54.237.133.81]:443... connected.  
HTTP request sent, awaiting response... 200 OK  
Length: 13921656 (13M) [application/octet-stream]  
Saving to: ‘ngrok-stable-linux-amd64.zip.1’  
  
ngrok-stable-linux- 100%[=====>] 13.28M 12.8MB/s in 1.0s  
  
2023-06-21 16:43:44 (12.8 MB/s) - ‘ngrok-stable-linux-amd64.zip.1’ saved [13921656/13921656]
```

```
[19] !ngrok authtoken 2f5c0e0adefebdd2da200ff7fa3afce74e6cf4cc510d45bfbb23a99890914000
```

Authtoken saved to configuration file: /root/.ngrok2/ngrok.yml

```
[20] ngrok_tunnel = ngrok.connect(10000)  
print('Public URL:', ngrok_tunnel.public_url)
```

Public URL: <http://0b60-35-204-57-156.ngrok-free.app>

```
!!diffusers api --port 10000
```

Ilustración 77. Despliegue de la API generadora de imágenes.

En el cliente de React, creamos un servicio para la generación de imágenes. Se comporta como cualquier otro servicio, simplemente envía peticiones de tipo POST a la URL que nos otorga la API. En la ilustración 78 se observa como existe una variable de entorno donde insertar la URL actual y cuando un componente requiere una imagen, simplemente llama al servicio con los parámetros requeridos para generar la imagen en forma de objeto. Se puede observar también que el servicio no utiliza el

interceptor, ya que esta API carece de seguridad y no tiene la misma implementación que la API REST.

```
import axios from 'axios';

const url = process.env.REACT_APP_IMAGE_GENERATOR_API_URL;

export async function generateImage(postData) {
  const options = {
    headers: {
      'Content-Type': 'application/json',
    },
  };
  try {
    const response = await axios.post(`${url}/text2img`, postData, options);
    return response.data;
  } catch (error) {
    return error;
  }
}
```

Ilustración 78. Servicio generación de imágenes.

5. Pruebas

En este punto, hablaremos sobre el desarrollo de las pruebas, qué criterios hemos seguido para aplicarlas y cómo nos hemos asegurado de que las funcionalidades están libres de fallos.

Dividiremos los resultados en las pruebas al cliente y pruebas al servidor, puesto que como comentamos anteriormente en la planificación de las pruebas, cada parte de la aplicación REST tiene un enfoque diferente a la hora de la realización de pruebas.

Es importante recalcar que estas pruebas no se centran sólo en encontrar errores, sino también en revisar que las funcionalidades que se esperan en las historias de usuario han sido cubiertas. De tal forma, las pruebas no sólo serán el resultado final de la aplicación, si no también el probar funcionalidades de la API REST y el cliente que la aplicación final puede no utilizar en este prototipo pero que deben funcionar correctamente.

5.1. Pruebas al Back-end

Para la realización de las pruebas a la API REST, se han realizado las pruebas comentadas en la planificación. Las pruebas que se definieron fueron pruebas de caja negra o de comportamiento, es decir, pruebas que se realizan mediante la entrada y la salida de datos, sin conocer el código, lo que se traduce como probar “lo que hace” y no “cómo lo hace”. Se han usado tanto la herramienta “*Swagger*” como la herramienta “*Postman*”. Comprobaremos cada punto de acceso de la aplicación en varias ocasiones, cada una de ellas variando la entrada y comprobando que la salida se corresponde con lo esperado. Por ejemplo, se probará a introducir datos no válidos, se comprobará que los datos devueltos son los correctos, etc. Estas pruebas se han

realizado simultáneamente a la implementación del back-end, principalmente al finalizar cada Sprint o Iteración del desarrollo.

Al tener nuestra API REST 40 endpoints, asumiendo que haremos de media 4 pruebas por método HTTP, esto nos dejaría aproximadamente con un total de 160 tipos de pruebas diferentes realizadas.

A continuación, iremos mostrando algunas de las más representativas que se le han realizado a algunos de estos endpoints.

La primera prueba que comentaremos es la petición de registro. Para el registro es necesario incluir un nombre de usuario, un correo electrónico y una contraseña. Las restricciones sobre el registro son que el nombre de usuario no puede ser repetido y debe tener al menos 3 caracteres de longitud, la contraseña debe tener al menos 5 caracteres de longitud y el correo electrónico debe ser un correo válido. Ninguno de estos campos debe estar vacío.

En la ilustración 79, se observa una petición de registro en la que se intenta registrar un usuario con un nombre de usuario que ya está en uso.

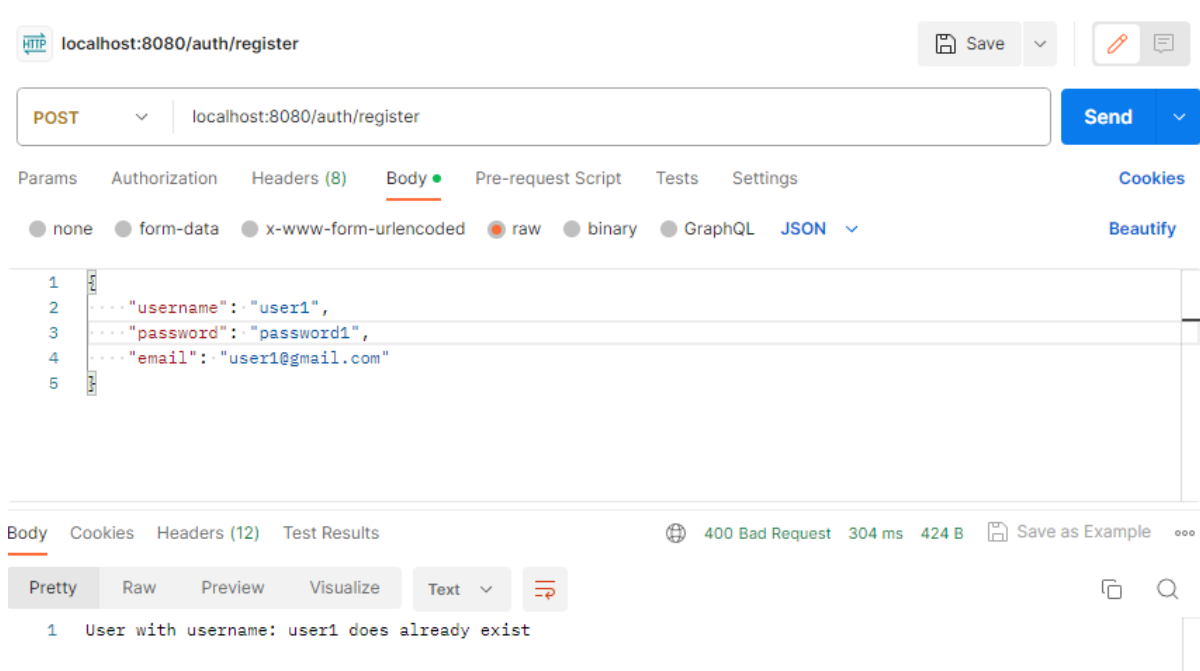


Ilustración 79. Intento de registro con usuario en uso.

Como se observa en la ilustración 79, los pasos para probar un endpoint serán introducir la URL a la que se quiere realizar la petición, en este caso “localhost:8080/auth/register”, seleccionar el tipo de petición, el cual en esta prueba es tipo POST e incluir los datos necesarios, que para esta petición en concreto es un objeto JSON en el body. Como vemos, al intentar registrar un usuario con un nombre que ya está en uso, se devuelve una respuesta HTTP con código 400 o “Bad Request” y se incluye un mensaje que informa de lo sucedido, en este caso el mensaje dice “El usuario con nombre: user1 ya existe”.

A continuación, en la ilustración 80, se observa una petición en la que el usuario está vacío y el correo no está en formato correcto. Como vemos, se informa de todas las restricciones que no se han cumplido, siendo en este caso que el usuario no puede estar vacío, el usuario debe tener al menos tres caracteres de extensión y el correo

debe ser un correo electrónico válido. Todos estos mensajes de información se separan por punto y coma en caso de ser una lista.

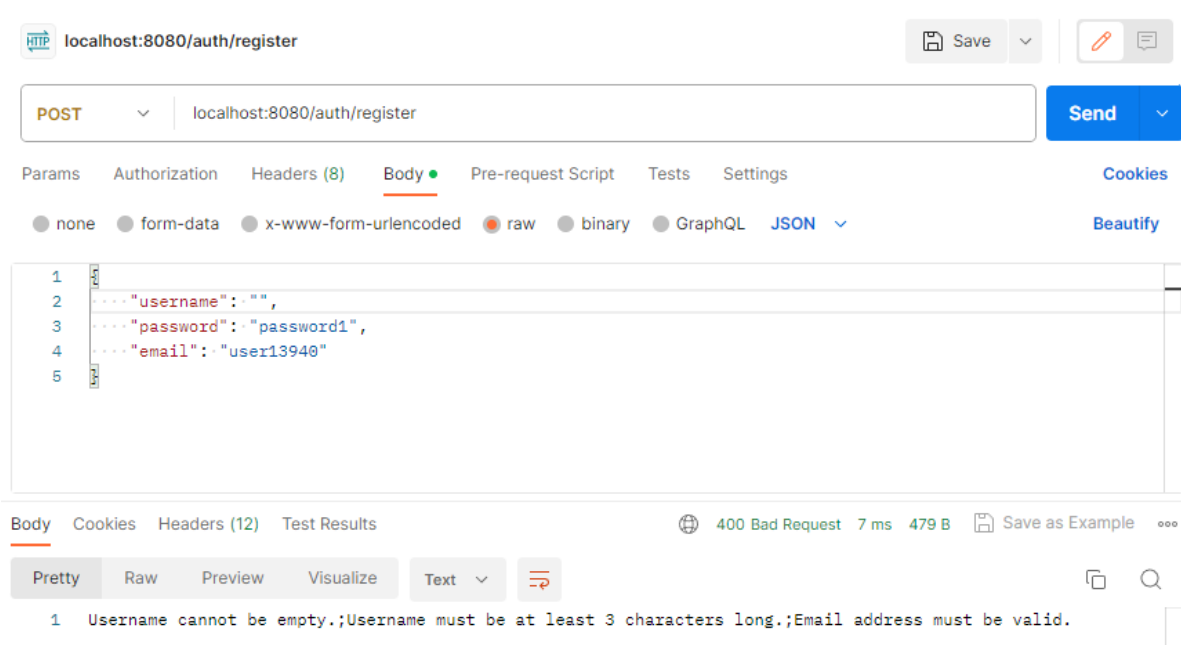


Ilustración 80. Intento de registro sin cumplir restricciones.

A continuación veremos el caso en el que el registro se produce correctamente. Se puede observar en la ilustración 81, como al realizar un registro exitoso se devuelve una respuesta HTTP con código 201 o “*CREATED*”, y se incluye un mensaje que informa que el registro ha sido exitoso. En este caso el método no devuelve ninguna información en concreto porque no es necesario.

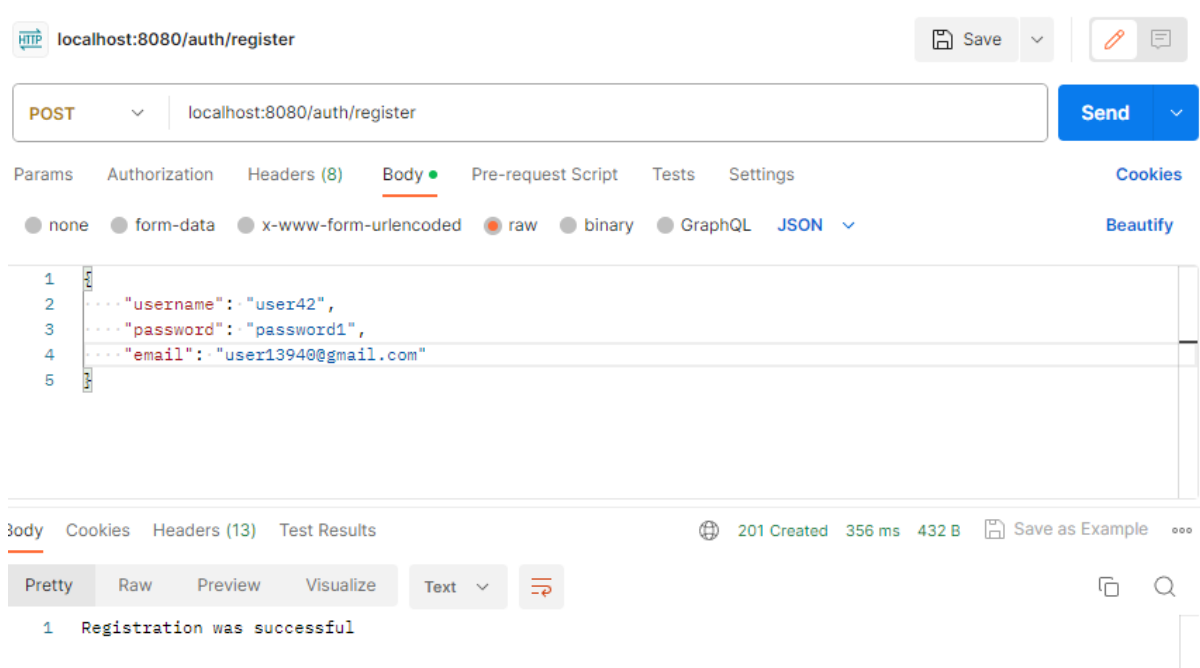


Ilustración 81. Registro exitoso.

Una vez hemos visto cómo se realiza la petición de registro y algunas de las casuísticas que se pueden producir a la hora de realizar el registro, comentaremos las pruebas del proceso de inicio de sesión.

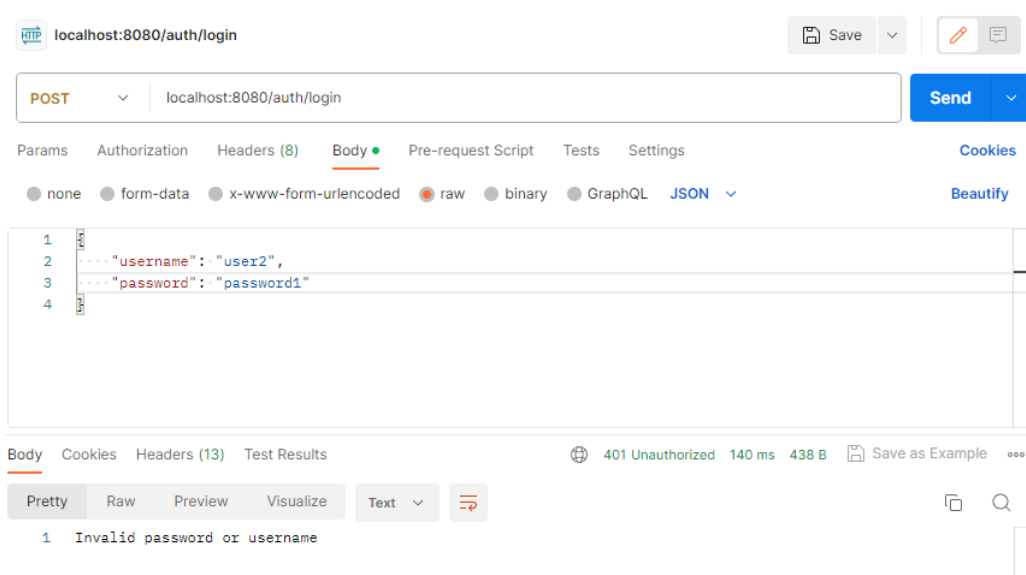


Ilustración 82. Inicio de sesión fallido.

En la ilustración 82, se observa un intento de inicio de sesión en el que la contraseña y el usuario no se corresponden, por lo que el servidor devuelve una respuesta HTTP con código 401 o de tipo “no autorizado”.

En el caso contrario, el inicio de sesión devolverá una respuesta con código 200 u “OK”, además de incluir el cuerpo de la respuesta el token que nos servirá para indicar en las peticiones que somos un usuario autorizado. Esto se muestra en la ilustración a continuación (Ilustración 83).

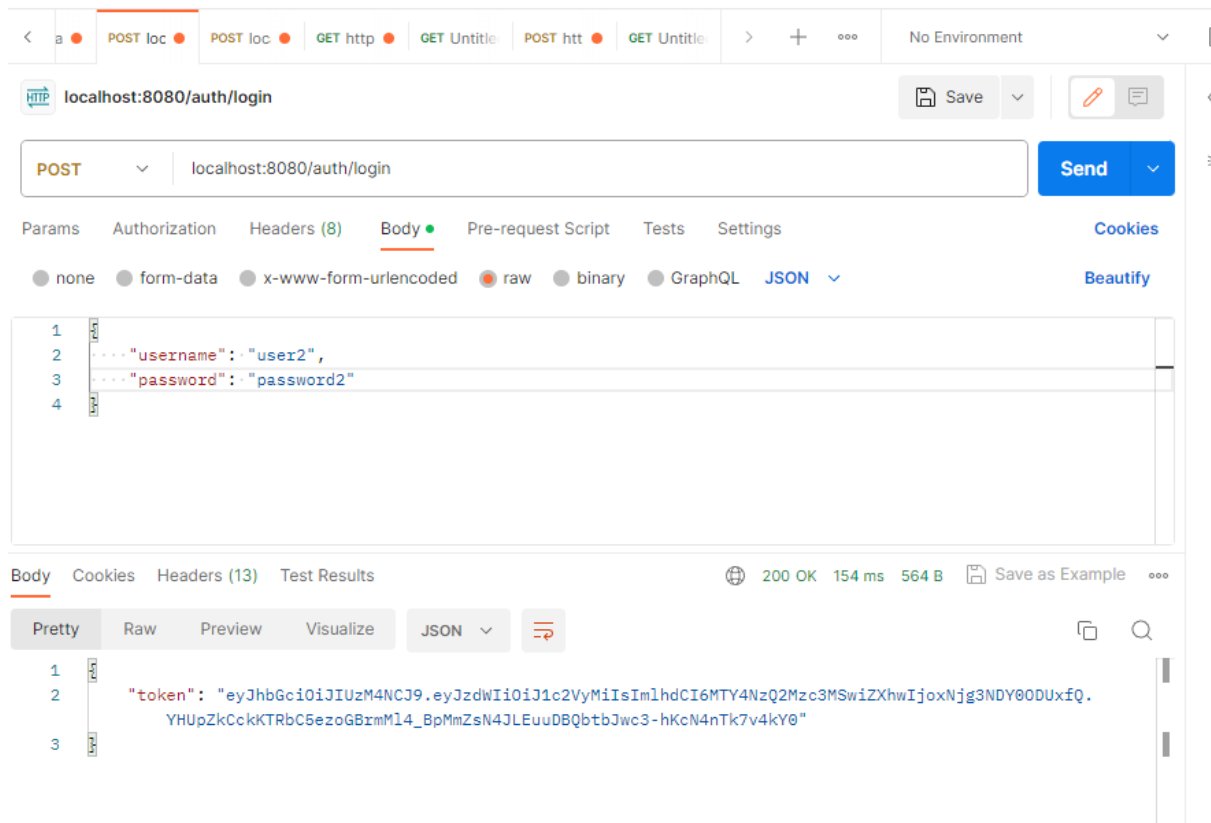


Ilustración 83. Inicio de sesión exitoso.

Una vez que hemos comentado algunas de las validaciones básicas, pasaremos a comentar otro tipo de comprobaciones.

La primera de ellas son las pre-autorizaciones, es decir, comprobaciones que se realizan a la hora de hacer una petición a un método HTTP. Un ejemplo sería el método que devuelve un documento, que sólo puede devolver un documento si el que hace la petición es el creador, en caso de que el documento sea privado o a cualquier usuario si el documento es público.

Para realizar esta prueba, hay que introducir en la cabecera de la petición el token, que como hemos visto nos identifica como “user2”, tal y como se observa en la ilustración 84.

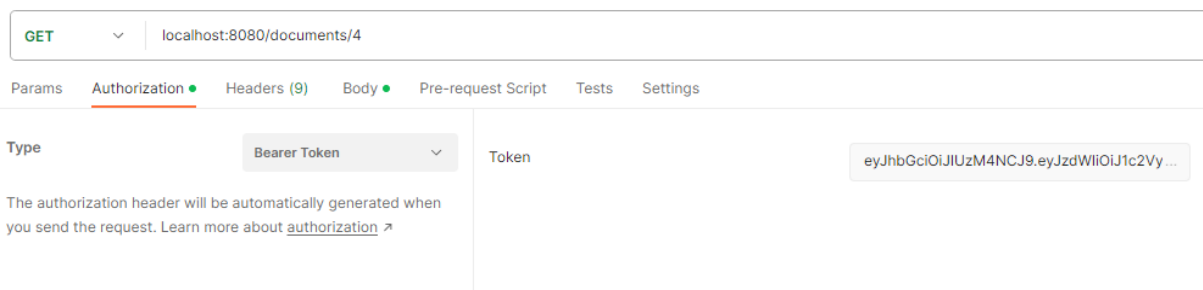


Ilustración 84. Uso del token JWT.

Una vez introducimos el token, haremos una petición a “/localhost:8080/documents/4”, donde el número indica la ID del documento a pedir.

En la ilustración 85 se observa lo que ocurre cuando la petición va dirigida a conseguir la información de un documento privado y del que “user2” no es autor: Se obtiene una respuesta HTTP de tipo no autorizado, con código 401.

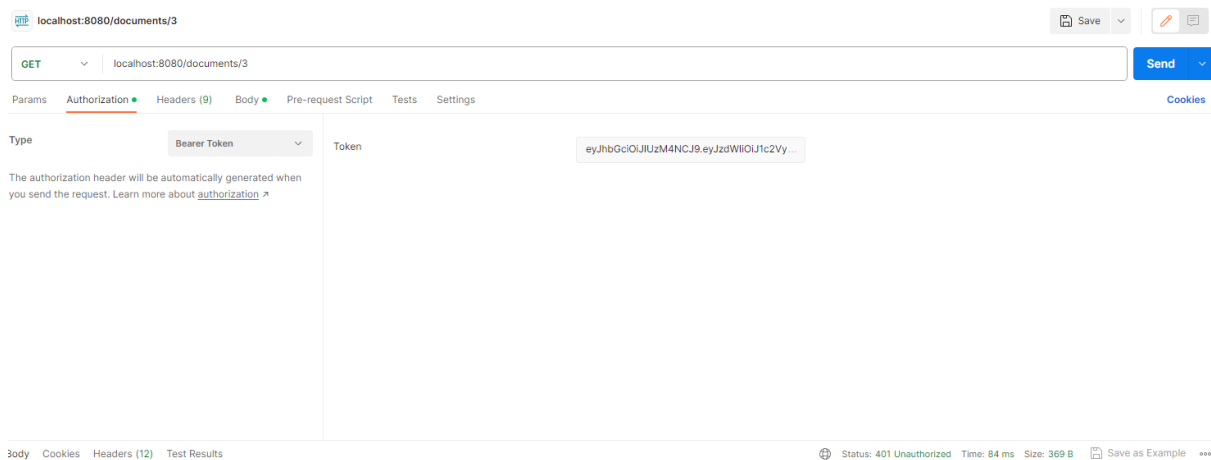


Ilustración 85. Petición a documento privado.

Sin embargo, en el caso de intentar acceder a un documento público, se obtienen los datos del documento en formato JSON, incluyendo información como su ID, título, portada en formato “*encode64*”, etc. Un ejemplo se observa en la ilustración 86.

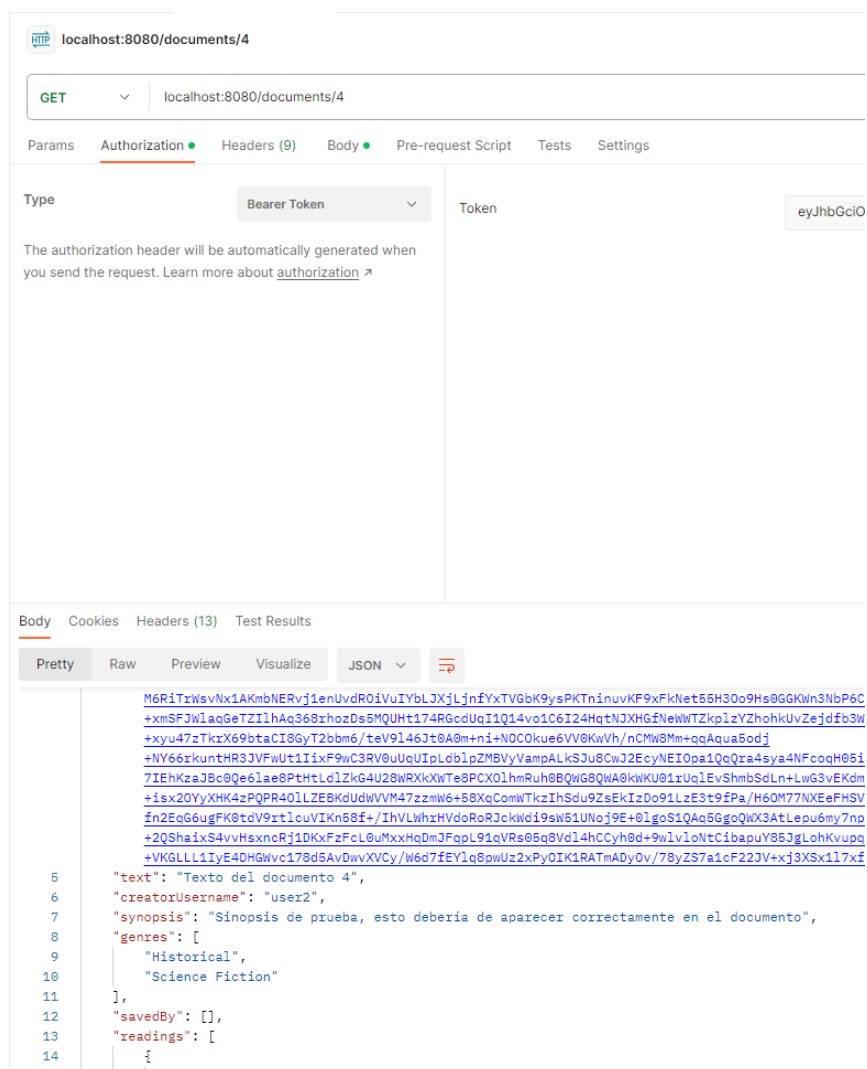


Ilustración 86. Petición de documento válido.

Tras comentar el caso de la preautorización, ahora comentaremos qué ocurre cuando se lanza una excepción en el caso de que un elemento que se intenta crear ya exista o un elemento que se intenta buscar no exista. En ese caso, el sistema está

preparado para manejar la excepción e informar con un mensaje a quien realiza la petición.

El caso que mostraremos de ejemplo será un caso en el que se intenta guardar un documento con un género que no se encuentra en la base de datos. En la ilustración 87, se observa como se intenta crear un documento con un género no existente y el servidor responde con un código de error HTTP 400 y con un mensaje que indica que el género “género no existente” no existe.

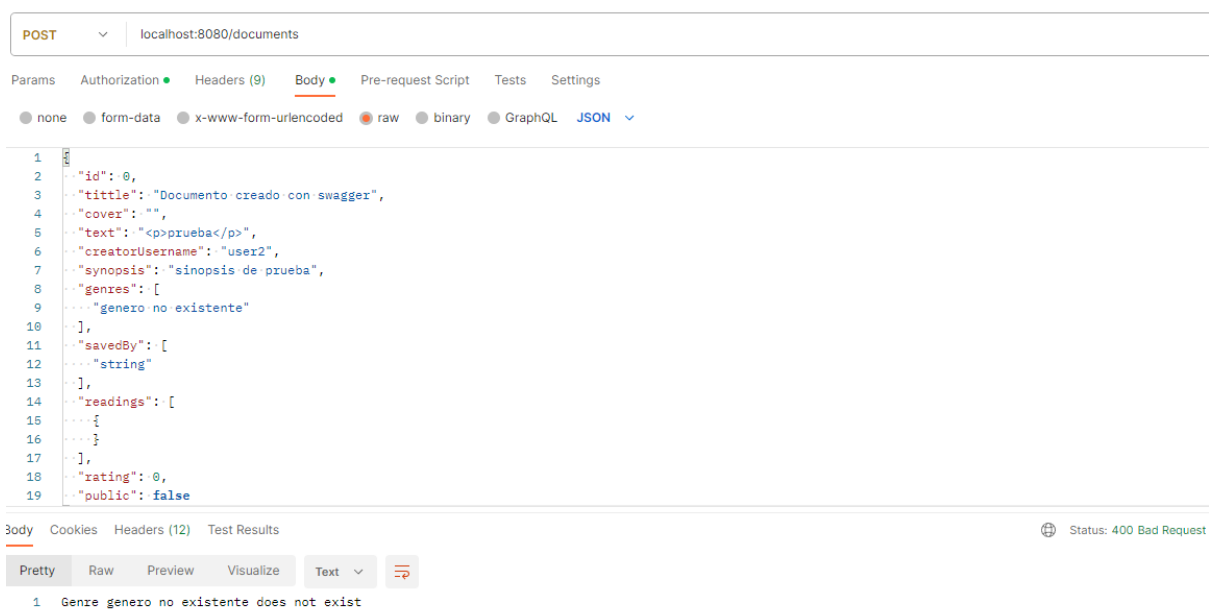


Ilustración 87. Creación de documento con género inexistente.

En caso de que utilicemos un género válido, como ciencia-ficción, el servidor responde con código 200, indicando que todos los procesos han concluido correctamente e incluyendo en el cuerpo de la respuesta el ID asignado al nuevo documento creado. Esta petición se muestra en la ilustración 88.

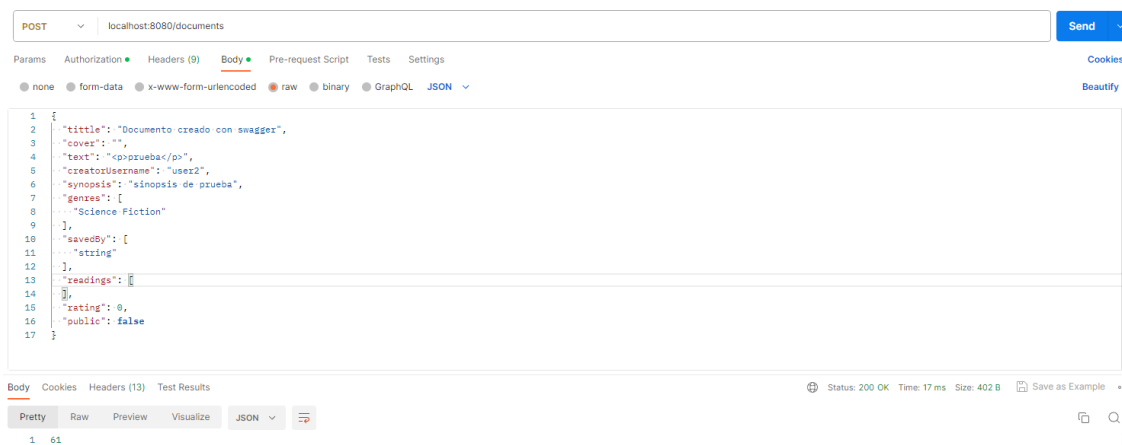


Ilustración 88. Petición de documento con género existente.

Al realizar estas pruebas se detectaron diferentes errores, entre ellos la posibilidad de enviar datos incorrectos por falta de validación en algunos elementos, lo cual fue implementado en cuanto se detectó el problema.

Otro de los errores que se descubrió mediante pruebas a los endpoints fue la existencia de excepciones no controladas, lo cual es un problema grave que puede hacer que el back-end colapse y detenga su ejecución. En cuanto se descubrió el problema, se implementaron los mecanismos necesarios para manejar esas excepciones.

5.2. Pruebas de front-end

Es el turno de revisar las pruebas que realizamos sobre el front-end, que como comentamos en la planificación de las pruebas, se basará en comprobar que se cumplen los criterios de aceptación y que la seguridad de la aplicación funciona correctamente.

En lo que respecta a los formularios, existen aproximadamente cinco formularios con tres campos de media, lo que nos deja con un total de quince campos a probar. Se

comprobará también que no se permite el envío de los datos si los campos no son correctos.

En la cuestión de los guardianes, existen cuatro guardianes que se deben probar: el que controla el acceso a un documento privado, el que controla el acceso de edición sólo al creador, el que comprueba si la URL existe y el que comprueba que el usuario está logueado.

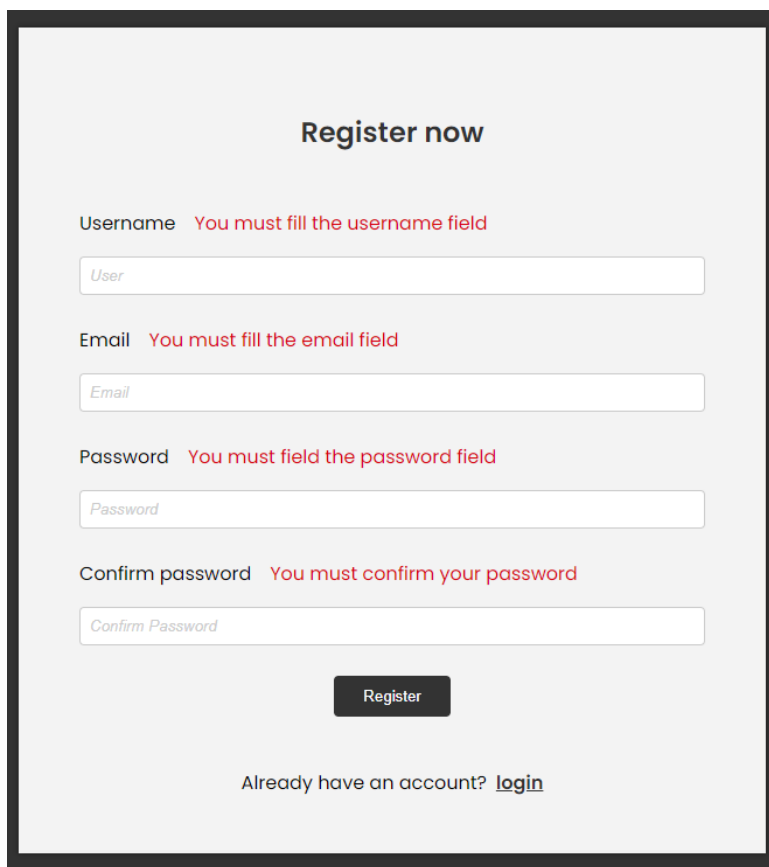
Conociendo que cada guardián permite de media dos tipos de entradas posibles (Usuario o con acceso o usuario sin acceso), se deberán hacer un total de 8 pruebas a los guardianes.

Para comprobar que las vistas se ajustan correctamente en todo tipo de pantallas, se realizará un recorrido por todas las vistas en dos tipos de pantalla, una pantalla estándar de PC (*1280x1024 píxeles*) y una pantalla con resolución estándar de móvil (*390x844 píxeles*). Sabiendo que existen un total de 8 vistas, habrá que revisar 16 vistas en total.

Por supuesto, también se comprobaron los criterios de aceptación de las historias de usuario, que como definimos al principio del documento, son un aproximado de 96 criterios de aceptación, con una media de cuatro por cada historia de usuario.

A modo de ejemplo, explicaremos la metodología seguida comentando algunas de las pruebas realizadas. La primera prueba que ejecutaremos será un intento de registro, y mostraremos las distintas posibilidades que existen a la hora de rellenar el formulario.

En la ilustración 89 se observa lo que ocurre cuando todos los campos se dejan vacíos y se intenta hacer el registro. Como vemos, aparece un error en los campos que están vacíos indicando que el campo del formulario no puede estar vacío.



The image shows a registration form titled "Register now". It contains four input fields: "Username", "Email", "Password", and "Confirm password". Each field has a red error message above it: "You must fill the username field", "You must fill the email field", "You must field the password field", and "You must confirm your password". Below the fields is a dark "Register" button. At the bottom, there is a link "login" preceded by the text "Already have an account?".

Register now

Username You must fill the username field

User

Email You must fill the email field

Email

Password You must field the password field

Password

Confirm password You must confirm your password

Confirm Password

Register

Already have an account? [login](#)

Ilustración 89. Formulario de registro con datos vacíos.

A continuación, en la ilustración 90 se muestra el caso en el que el usuario ya está en uso, el email no es válido y la contraseña no está confirmada correctamente. Como se observa, de nuevo se incluyen mensajes que indican al usuario qué campos no están rellenados correctamente y cuál es el motivo de que no se permita el registro.

Register now

Username This username is already in use

Email Use a valid email

Password

Confirm password The passwords do not match

Already have an account? [login](#)

Ilustración 90. Formulario de registro con datos erróneos.

Una vez probamos todas las posibilidades de los formularios, otra de las pruebas que quisimos realizar fue la seguridad de las rutas, ya que lo consideramos un elemento de suma importancia para mantener la información segura.

En la ilustración 91, se observa lo que ocurre cuando un usuario intenta acceder a un documento privado de otro usuario, lo que se traduce en que no tiene acceso a él. Como sabemos, en este caso el guardián hace efecto, redirigiendo al usuario a una vista que le informa de que sólo el creador del documento puede editarlo y obligándolo a utilizar la barra de navegación para volver a una vista a la que tenga acceso.

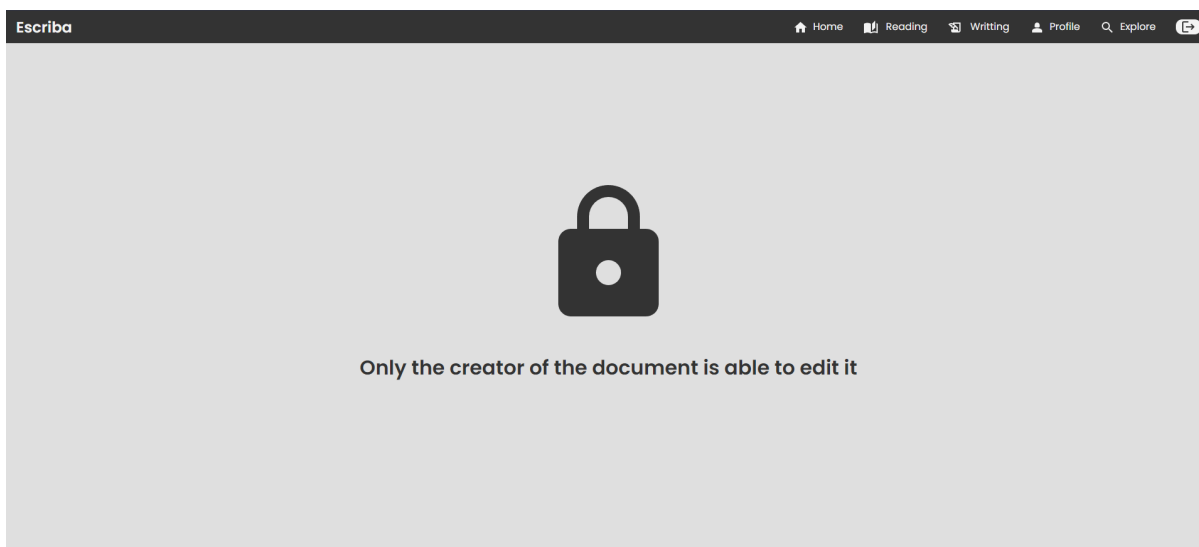


Ilustración 91. Intento de acceso a URL protegida.

En el caso contrario, si nos identificamos como el usuario que tiene acceso al documento e intentamos acceder mediante URL, se observa en la ilustración 92 que se permite el acceso sin mayor inconveniente.

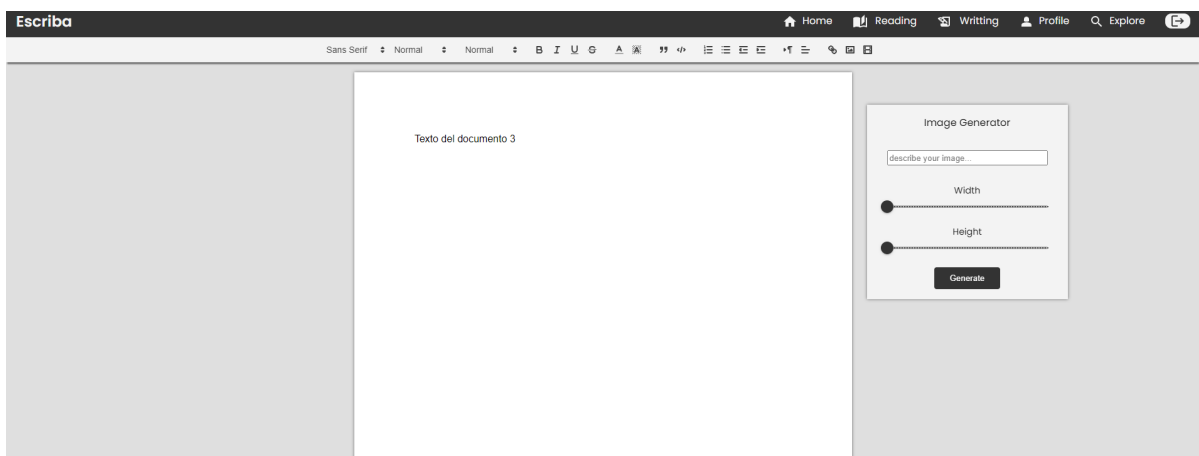


Ilustración 92. Acceso válido a un documento.

Por último, he querido mostrar también las pruebas realizadas en el ámbito de la responsividad de la aplicación. Se ha recorrido la aplicación tanto en formato PC como en formato móvil y se ha comprobado que la interfaz funciona correctamente y se

ajusta al tamaño de la pantalla. En la ilustración 93 se observa la vista perfil en formato PC. Como se ve su interfaz se muestra correctamente y es similar a la diseñada.

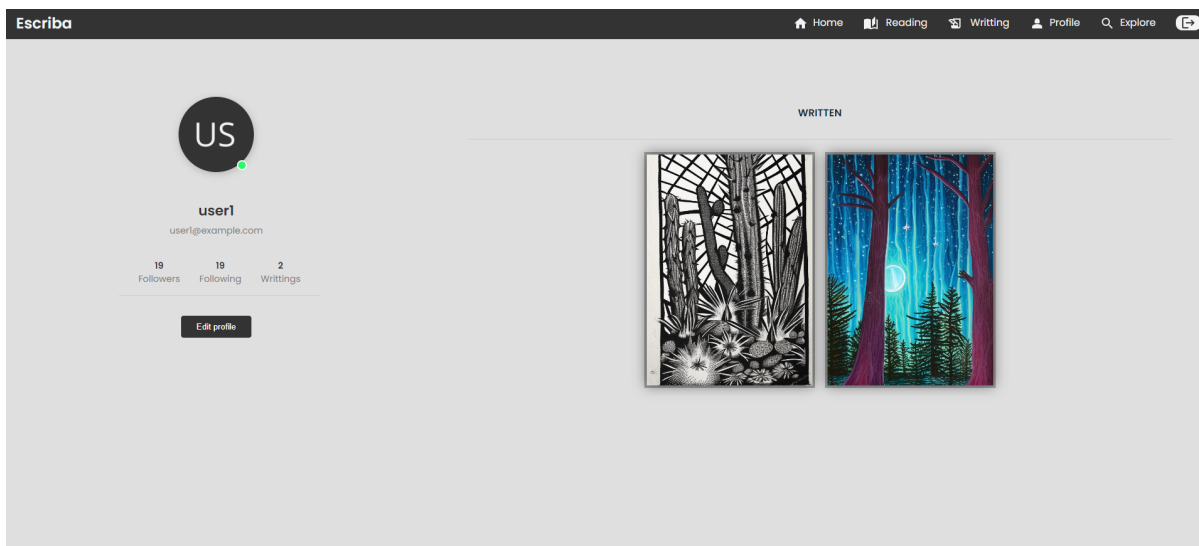


Ilustración 93: Vista perfil en versión PC.

De la misma manera, en la versión móvil (Ilustración 94) se observa que la interfaz se muestra correctamente y se corresponde con el diseño.

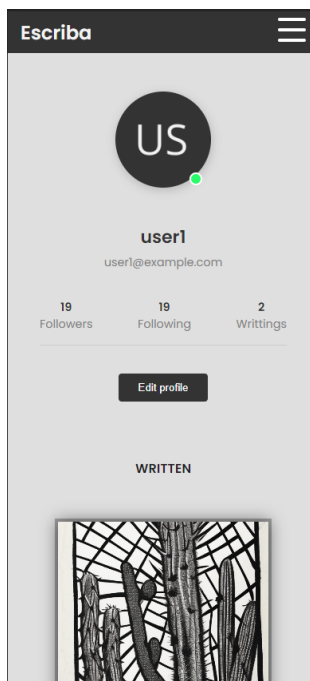


Ilustración 94. Vista perfil en versión móvil.

Este tipo de pruebas nos sirvió para detectar errores varios. Los más destacables fueron errores en la adaptación de la vista a cada tipo de pantalla, donde algunos elementos no se colocaban de la forma correcta.

Otro tipo de errores detectados fueron la posibilidad de acceder a la lectura de documentos privados por un fallo en el guardián responsable de esa ruta. El haberlo detectado mediante la exploración de la aplicación y criterios de aceptación nos hizo poder solucionarlos rápidamente.

6. Conclusiones

A lo largo del documento, hemos comentado todos los puntos relevantes y a tener en cuenta sobre el desarrollo del proyecto. Tras ver todos estos aspectos en detalle, resumimos la información proporcionada a modo de conclusión.

En lo que respecta al cumplimiento de los objetivos, podemos afirmar que los objetivos iniciales se han cumplido al completo. El primero de los objetivos era la realización de un estudio de las necesidades y las soluciones. Este objetivo se ha cumplido, ya que si observamos el apartado de análisis, más concretamente el punto 2 y sus subapartados, donde hemos estudiado las necesidades a cubrir paso a paso. En el punto 2.1 hemos analizado aplicaciones similares, obteniendo información útil sobre qué funcionalidades de esas aplicaciones podemos implementar y qué funcionalidades se pueden mejorar. En el siguiente apartado comentamos las necesidades y funcionalidades necesarias que deberá incluir la aplicación para subsanar esas necesidades. También realizamos un estudio de tecnologías, que nos permitió decidir qué tecnologías eran las más adecuadas para desarrollar nuestro sistema. Finalmente, hemos realizado una propuesta de solución, en la que hemos expuesto cuáles han sido los resultados obtenidos del estudio de los puntos anteriores. Por último, hemos realizado una planificación del trabajo a realizar mediante el uso de historias de usuario, a las cuales se les ha asignado un valor que se utilizó para priorizar en la planificación las historias de usuario más importantes. Junto con esta planificación se realizó un estudio de viabilidad, que nos ayudó a cerciorarnos de que el sistema era viable, dotándonos de información sobre los gastos que supone el desarrollo del proyecto, además de proponer algunos modelos de negocio para subsanar esos gastos. El segundo de los objetivos se centra principalmente en el diseño. El diseño ha sido un punto central en el desarrollo de la

aplicación y en los contenidos de esta documentación. Hemos hablado de él en el punto 3 y sus apartados, basándonos en una arquitectura cliente servidor, por lo que hemos realizado diferentes estudios del diseño tanto para la parte front-end como para el back-end. Este diseño se ha acompañado de distintos diagramas para cada fase del diseño, desde diagramas de clase hasta diagramas de secuencia. También se ha realizado un diseño de la interfaz para ambos componentes de la arquitectura, la interfaz gráfica del cliente mediante distintos wireframes y la API del servidor mediante una herramienta de documentación como es Swagger.

El último de los objetivos que se debían alcanzar era seleccionar un entorno de desarrollo e implementar un prototipo de aplicación web que satisfaga las necesidades anteriormente mencionadas. Este objetivo ha sido cumplido, tal y como se observa en el apartado 3, en el cual se ha documentado el proceso de implementación, donde se han detallado todas las características de la arquitectura y los pasos que se han seguido para la implementación del prototipo.

El enfoque propuesto para este proyecto se fundamentó en el uso de tecnologías modernas y actuales, React para la capa de la interfaz de usuario y Spring Boot para el back-end, siendo todo esto desarrollado con una metodología basada en Scrum.

La selección de las tecnologías ha sido, desde mi punto de vista, una forma de asentar conocimientos adquiridos en el grado, como es el caso del uso de Spring boot, un framework que fue impartido en la carrera a un alto nivel. Por otro lado, el seleccionar React como biblioteca para el desarrollo del front-end ha sido todo un desafío, puesto que no partía de conocimientos previos en frameworks de este estilo. A pesar de esto, se ha realizado un proyecto robusto y funcional, gracias en parte a la metodología ágil seleccionada, la cual nos permitió revisar el trabajo ya realizado a medida que nuestros conocimientos iban evolucionando. Esto demuestra que no sólo

los conocimientos del grado son útiles en cuanto a las tecnologías impartidas durante el mismo, si no que lo impartido en distintas asignaturas es de gran ayuda a la hora de aprender una nueva tecnología, lenguaje o framework, ayudando así a mantener los conocimientos actualizados y a poder ampliar estos mismos conocimientos.

En lo que respecta al mantenimiento del sistema y escalabilidad, se han utilizado tecnologías actuales y con mucho respaldo por parte de empresas y usuarios, lo que facilita el mantenimiento y la ampliación del sistema. Además, la estructura de los proyectos y la asignación de responsabilidades entre clases, ayudan a que el código sea legible para futuros desarrolladores, además de escalable. El uso de patrones de diseño reconocidos desempeña la misma función, asignando responsabilidades correctamente y permitiendo a desarrolladores comprender el código más fácilmente.

El principal desafío al que nos hemos enfrentado, aparte de la necesidad de aprender una nueva tecnología, era el tiempo asignado al proyecto, que junto a una desviación en la planificación provocó que algunas de las historias de usuario no llegasen a ser implementadas. Sin embargo, como se especifica en metodologías ágiles, se priorizaron las historias con más valor o más fundamentales, lo que cristalizó en una aplicación que, aunque no contiene todas las funcionalidades esperadas en la planificación, si contiene todas las funcionalidades básicas y clave, junto con gran parte de las funcionalidades más secundarias. Otro de los desafíos a los que nos enfrentamos fue la utilización de tecnologías muy novedosas como puede ser el caso de Stable Diffusion, una inteligencia artificial para la generación de imágenes que aún no tiene la suficiente documentación ni proyectos de ejemplo asociados, por lo que tuvimos que investigar diversas fuentes y diversos proyectos, además de analizar diferentes métodos para desplegar la API necesaria. Todo esto resultó en un gran consumo de tiempo en investigación que no pudo ser empleado en otros aspectos del proyecto.

Bajo mi punto de vista, los aspectos más relevantes del proyecto y que más interesantes me han resultado han sido el aprendizaje de React, del cual he obtenido bastante conocimiento, y que puede ser muy útil para el futuro. A esto se le suma el diseño e implementación de los componentes a su nivel más básico, sin la utilización de librerías externas, lo que me permitió entender en profundidad el funcionamiento de un componente a nivel de estilos y operaciones. Otro aspecto muy interesante ha sido el diseño e implementación de una aplicación con esquema de red social, el cual es un tipo de aplicación que utilizamos a diario, ya que este tipo de aplicaciones sociales aglutinan a gran número de usuarios. El conocer cómo se diseñan e implementan este tipo de aplicaciones, aunque sea a nivel básico, ha sido bastante educativo e interesante. Por último, la utilización de inteligencia artificial ha sido un aspecto clave del proyecto y que me ha agradado mucho implementar, ya que he descubierto bastante información sobre este tema que me ha ayudado a implementar este tipo de tecnologías en la aplicación, obteniendo un resultado que debido al poco tiempo que estas IAs llevan entre nosotros, impresiona bastante.

Por último, me gustaría recalcar algo que he mencionado anteriormente, y es que durante el proyecto se han aplicado gran parte de los conocimientos adquiridos durante el grado, desde gestión de proyectos, pasando por diseño software hasta conocimientos en implementación como diseño de algoritmos o estructura de datos, si obviar los conocimientos más fundamentales para este proyecto, los obtenidos en desarrollo de aplicaciones web y en desarrollo de aplicaciones empresariales, las cuales se centran en tecnologías front-end y back-end respectivamente. No sólo el grado nos ha dotado de información sobre todos estos ámbitos, si no que también nos ha proporcionado las herramientas para ampliar nuestro conocimiento, ya que lo impartido durante el grado es aplicable no sólo a las tecnologías usadas en el mismo, si no también a otras tecnologías asentadas e incluso novedosas.

6.1. Mejoras y trabajos futuros

Comentaremos a continuación distintas mejoras y ampliaciones que se le podrían aplicar al proyecto en futuras actualizaciones. Parte de estas mejoras incluirán las historias de usuario no implementadas, pero también algunas mejoras sobre lo implementado.

- Añadir comentarios de línea, funcionalidad que se planeaba desarrollar (Historia de usuario nº 18) pero que se abandonó por falta de tiempo. Esto ayudaría a los usuarios a destacar partes de los documentos que leen, para ellos o para otros usuarios. Esta funcionalidad sería sencilla de implementar debido a que el sistema ya está orientado para incluir comentarios de este tipo, pero como comentamos no se incluyeron por falta de tiempo.
- Implementar una API de generación de imágenes propia o contratación de un servicio externo, ya que como se ha explicado durante la documentación, nuestra API hace la función de prototipo, para mostrar las distintas posibilidades que ofrece este tipo de servicios. Sin embargo, al estar desplegada en un cuaderno de Google Colab, esto no lo hace apto para ser utilizado en una aplicación en funcionamiento. Para ello lo óptimo sería contratar un servicio externo, lo cual sería fácil de incluir puesto que sólo habría que hacer cambios menores a las clases encargadas de gestionar la interacción con la API.
- La escritura colaborativa, es decir, permitir a varios usuarios escribir sobre un documento de forma simultánea se podría implementar en nuestra aplicación para permitir esta funcionalidad. Para ello, habría que implementar conexiones vía “*websockets*” y realizar algunos cambios a algunas de las clases tanto del

front-end como del back-end, ya que la aplicación no está pensada para incluir esta función, aunque sí sería posible incluirla con los cambios necesarios.

- Mejoras en el sistema de recomendación de documentos, es decir, incluir un sistema mejorado que muestre documentos a los usuarios en función de sus gustos y lecturas, manteniendo información sobre las reseñas y lecturas de los usuarios. Esto sería sencillo de implementar en nuestra aplicación, puesto que ya implementa un sistema de búsqueda por género y título, por lo que únicamente habría que seleccionar los géneros que un usuario prefiere y ofrecerle documentos que cumplan con sus preferencias.

Como vemos, aunque el sistema incluye todas las funcionalidades básicas, aún permite algunas mejoras para iteraciones futuras. Esto se conocía durante el desarrollo del proyecto, y por eso la base del proyecto se ha construido pensando en facilitar la ampliación y el desarrollo sobre lo ya implementado, priorizando la división en módulos y la legibilidad del código. Todo esto ha sido posible gracias a la metodología empleada, que nos ha facilitado la implementación por módulos para poder incluir funcionalidades nuevas en cada iteración. A su vez, la documentación del sistema informa de los elementos clave, lo que ayuda a comprender el funcionamiento del prototipo y qué partes son las que deben ser modificadas para permitir nuevas funcionalidades.

7. Bibliografía

- [1] K. Schwaber, "SCRUM Development Process". Advanced Development Methods. Acceso: Feb, 2023.
- [2] eldiario.es. (Marzo, 2021). "La comprensión lectora baja de forma significativa en Primaria". [Online]. Disponible en: https://www.eldiario.es/sociedad/baja-forma-significativa-comprension-lectora-primaria_1_10207332.html. Acceso: Feb, 2023.
- [3] Deymar A. (Jun 15, 2023). "¿Qué es React? Una guía completa para principiantes". Hostinger. [Online]. Disponible en: <https://www.hostinger.es/tutoriales/que-es-react/>. Acceso: Feb, 2023.
- [4] F. G. de Zúñiga, "Las 5 características más destacadas de React.js". Arsys Blog. [Online]. Disponible en: <https://www.arsys.es/blog/caracteristicas-reactjs>. Acceso: Feb, 2023.
- [5] "Spring Framework". Spring. [Online]. Disponible en: <https://spring.io/projects/spring-framework>. Acceso: Feb, 2023.
- [6] Í. C. Bengi, "Midjourney vs Stable Diffusion". Hashdork. [Online]. Disponible en: <https://hashdork.com/es/midjourney-vs-stable-diffusion/>. Acceso: Feb, 2023.
- [7] C. Simões. "MoSCoW: Qué es y cómo priorizar en el desarrollo de tu aplicación". itdo.com. [Online]. Disponible en: <https://www.itdo.com/blog/moscow-que-es-y-como-priorizar-en-el-desarrollo-de-tu-aplicacion/>. Acceso: Marzo, 2023.
- [8] A. Prats. "Scrum Poker: Evaluando el esfuerzo de las tareas". Piensa en Software, Desarrolla en Colores. [Online]. Disponible en: [https://blogs.encamina.com/piensa-en-software-desarrolla-en-colores/scrum-poker-evaluando-esfuerzo-tareas/#:~:text=El%20sistema%20utiliza%20un%20rango,%2C%20%E2%88%9E%2C%20\(coffee\)](https://blogs.encamina.com/piensa-en-software-desarrolla-en-colores/scrum-poker-evaluando-esfuerzo-tareas/#:~:text=El%20sistema%20utiliza%20un%20rango,%2C%20%E2%88%9E%2C%20(coffee).). Acceso: Marzo, 2023.

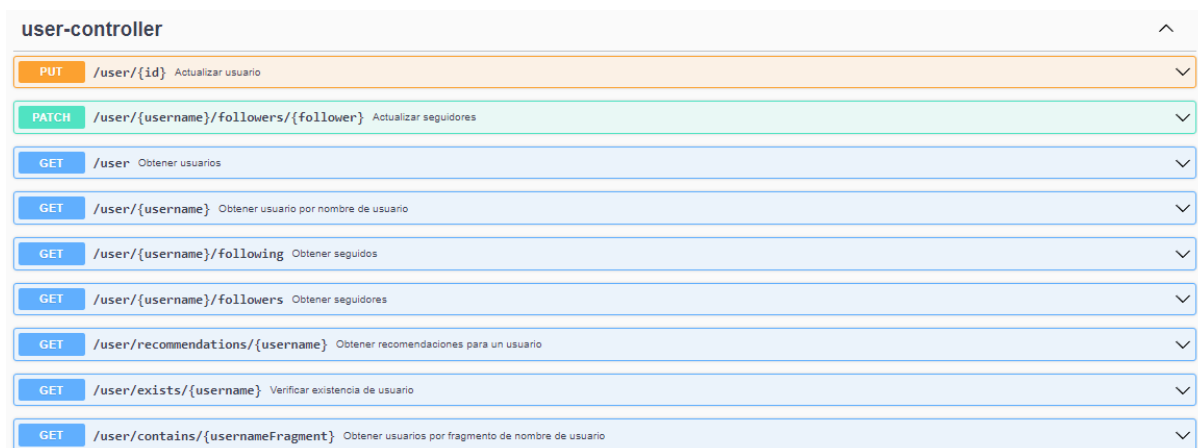
- [9] Indeed. [Online]. Disponible en: <https://es.indeed.com/>. Acceso: Marzo, 2023.
- [10] "¿Qué es una API RESTful?". Amazon Web Services. [Online]. Disponible en: <https://aws.amazon.com/es/what-is/restful-api/>. Acceso: Marzo, 2023.
- [11] "Patrones de diseño: Composición en React". CloudAppi. [Online]. Disponible en: <https://cloudappi.net/patrones-de-diseno-composicion-en-react/>. Acceso: Marzo, 2023.
- [12] "Patrón Composite: estructura de árbol flexible para objetos complejos". IONOS Digital Guide. [Online]. Disponible en: <https://www.ionos.es/digitalguide/paginas-web/desarrollo-web/patron-composite/>. Acceso: Marzo, 2023.
- [13] "Patrón Observer: cómo notificar cambios y eventos en el desarrollo web". IONOS Digital Guide. [Online]. Disponible en: <https://www.ionos.es/digitalguide/paginas-web/desarrollo-web/que-es-el-patron-observer/>. Acceso: Marzo, 2023.
- [14] "Model–view–controller". Wikipedia. [Online]. Disponible en: <https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller>. Acceso: Abril, 2023.
- [15] F. Scipion. "¿Qué es una web responsive?". Lifestyle al cuadrado. [Online]. Disponible en: <https://www.lifestylealcuadrado.com/que-es-una-web-responsive/>. Acceso: Abril, 2023.
- [16] "¿Qué es un wireframe para un sitio web?". Lucidchart. [Online]. Disponible en: <https://www.lucidchart.com/pages/es/que-es-un-wireframe-para-un-sitio-web>. Acceso: Abril, 2023.
- [17] "Spring Security". Spring. [Online]. Disponible en: <https://spring.io/projects/spring-security>. Acceso: Abril, 2023.

- [18] "Spring Bean Annotations: @Component, @Service, @Repository, and @Controller". Baeldung. [Online]. Disponible en:
<https://www.baeldung.com/spring-bean>. Acceso: Abril, 2023.
- [19] "Create a New React App". React. [Online]. Disponible en:
<https://legacy.reactjs.org/docs/create-a-new-react-app.html>. Acceso: Mayo, 2023.
- [20] "Functional vs Class Components in React". Medium. [Online]. Disponible en:
<https://djoech.medium.com/functional-vs-class-components-in-react-231e3fbd7108>. Acceso: Mayo, 2023.
- [21] "React Functional Components: State and Props". Ordinary Coders. [Online]. Disponible en:
<https://ordinarycoders.com/blog/article/react-functional-components-state-props>. Acceso: Mayo, 2023.
- [22] P. Gutiérrez. "Responsive Design: una estructura adaptable". Genbeta. [Online]. Disponible en:
<https://www.genbeta.com/desarrollo/responsive-design-estructura-adaptable>. Acceso: Mayo, 2023.
- [23] Barger, R. (Fecha de publicación). "Cómo usar Axios con React". freeCodeCamp. [Online]. Disponible en:
<https://www.freecodecamp.org/espanol/news/como-usar-axios-con-react/>. Acceso: Mayo, 2023.

Apéndice I. Detalles diseño y funcionamiento controladores REST

En este apéndice comentaremos de forma más detallada el diseño y el funcionamiento de los controladores REST de nuestra aplicación.

Comenzaremos hablando sobre el “User-Controller”. Como se observa en la ilustración 95, el controlador usuario contiene varios métodos GET, un método PUT y un método PATCH. Es notoria la ausencia de los métodos POST y DELETE. La ausencia del método POST se debe a que el encargado de la creación de los usuarios es el controlador de autorización, como veremos más adelante. La ausencia del método DELETE se debe a que no se consideró necesaria la eliminación de usuarios puesto que las historias de usuario no lo requerían.



user-controller	
PUT	/user/{id} Actualizar usuario
PATCH	/user/{username}/followers/{follower} Actualizar seguidores
GET	/user Obtener usuarios
GET	/user/{username} Obtener usuario por nombre de usuario
GET	/user/{username}/following Obtener seguidos
GET	/user/{username}/followers Obtener seguidores
GET	/user/recommendations/{username} Obtener recomendaciones para un usuario
GET	/user/exists/{username} Verificar existencia de usuario
GET	/user/contains/{usernameFragment} Obtener usuarios por fragmento de nombre de usuario

Ilustración 95: Métodos del controlador User-Controller

Para mostrar un método representativo del controlador mostraremos el método GET que obtiene usuarios que contengan en su nombre de usuario la cadena de texto que se pasa como parámetro. En este caso la petición se realiza a “/user/contains/{usernameFragment}”, siendo “usernameFragment” un parámetro de

tipo ruta. Al hacer la petición, el valor a devolver será una lista que contenga los DTOs de los usuarios que cumplan con la condición que se ha mencionado anteriormente. Todos los detalles del método se observan en la siguiente ilustración (Ilustración 96)

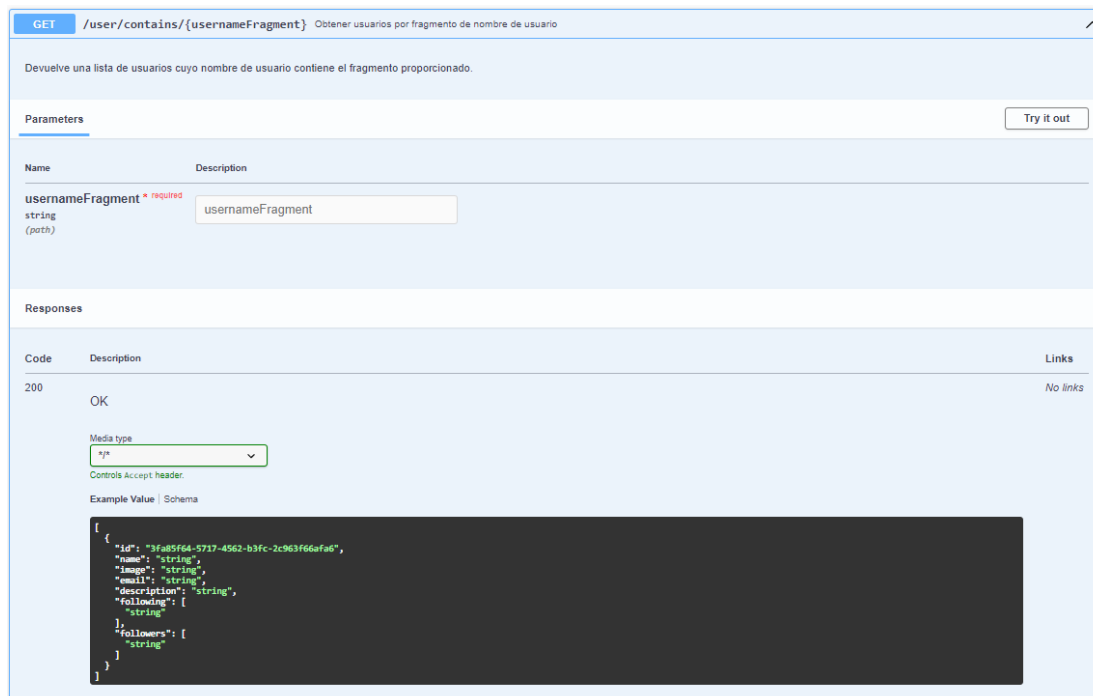
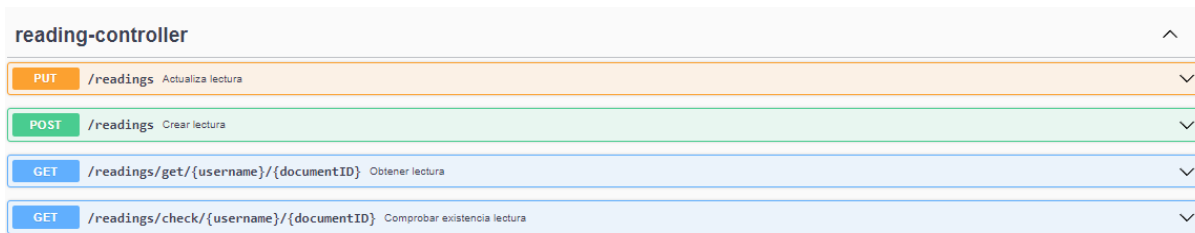


Ilustración 96. Método obtener usuarios por fragmento de nombre de usuario del controlador usuario

Continuamos ahora con el controlador de lecturas, donde se observa que este controlador es más simple, debido a que las lecturas sólo son usadas en puntos concretos de la aplicación. Observamos que contiene un método POST con el que se crean las lecturas, el cual se produce en el endpoint “/readings”. Este método devolverá únicamente una respuesta HTTP de tipo OK en caso de que el guardado haya sido correcto. No se devolverá ningún DTO puesto que no es necesario para nuestra aplicación. Lo mismo ocurre con el método de creación lectura, el cual crea una lectura. Sólo necesitaremos saber si la lectura se ha creado correctamente o no para manejar los distintos errores. En cuanto a los métodos GET, observamos que existen dos, uno que recupera la información de una lectura en función del documento que está siendo leído y el usuario que lo está leyendo y otro que comprueba si una

lectura existe, en función de los mismos parámetros. Estos métodos se observan en la ilustración a continuación (Ilustración 97).



reading-controller	
PUT	/readings Actualiza lectura
POST	/readings Crear lectura
GET	/readings/get/{username}/{documentID} Obtener lectura
GET	/readings/check/{username}/{documentID} Comprobar existencia lectura

Ilustración 97. Controlador lectura en documentación Swagger

Es el turno del controlador de documentos, el cual es el controlador más amplio debido a que los documentos son el eje central de la aplicación que estamos desarrollando. El controlador consta de numerosos métodos GET, un Método POST que crea el documento, y otro método POST que crea una relación entre documento y usuario, siendo esta relación la relación de guardado. Existen de la misma manera dos métodos DELETE, uno que borra un documento y otro que elimina esta relación de guardado comentada anteriormente. El controlador también contiene un método PATCH. Este tipo de método ha sido utilizado para referenciar actualizaciones parciales de una entidad, en este caso de la entidad documento. La actualización que se realiza en este método es la actualización de la visibilidad del documento. Respecto a los métodos GET todos, a excepción del GET de obtención mediante géneros, siguen el mismo esquema anterior, se indica mediante un parámetro de ruta y la ruta específica qué documento se debe buscar y en función de qué criterio. El controlador se observa en la Ilustración 98.

document-controller			^
GET	/documents/{id}	Obtener documento mediante ID	▼
PUT	/documents/{id}	Actualizar documento	▼
POST	/documents	Crear un documento	▼
GET	/documents/saved/{username}	Obtener documentos guardados	▼
POST	/documents/saved/{username}	Guardar un documento para un usuario	▼
DELETE	/documents/saved/{username}	Desmarcar documento guardado	▼
PATCH	/documents/{documentId}/visibility	Cambiar visibilidad	▼
GET	/documents/recommendation	Obtener recomendación global	▼
GET	/documents/read/{username}	Obtener documentos en lectura	▼
GET	/documents/public/{username}	Obtener documentos públicos de un usuario	▼
GET	/documents/private/{username}	Obtener documentos privados de un usuario	▼
GET	/documents/genres	Obtener documentos por género y página	▼
GET	/documents/checkPublic/{documentId}	Verificar si un documento es público	▼
GET	/documents/checkOwner/{documentId}	Verificar propietario del documento	▼
DELETE	/documents/{documentId}	Eliminar documento	▼

Ilustración 98. Controlador documento en documentación Swagger

Los métodos que comentaremos en este controlador serán el método PATCH y el método GET a `"/documents/genres"`.

Como vemos en la ilustración 99, el método PATCH recibe por parámetro de tipo ruta un ID de tipo entero, el cual se corresponde con el documento del que se quiere cambiar la visibilidad. Una vez la visibilidad haya sido cambiada, se devuelve un DTO con la información del documento en cuestión y con la visibilidad de este actualizada. La respuesta HTTP será de tipo OK. En caso de que el documento no sea encontrado, se lanzará una excepción que resultará en una respuesta de tipo NOT_FOUND.

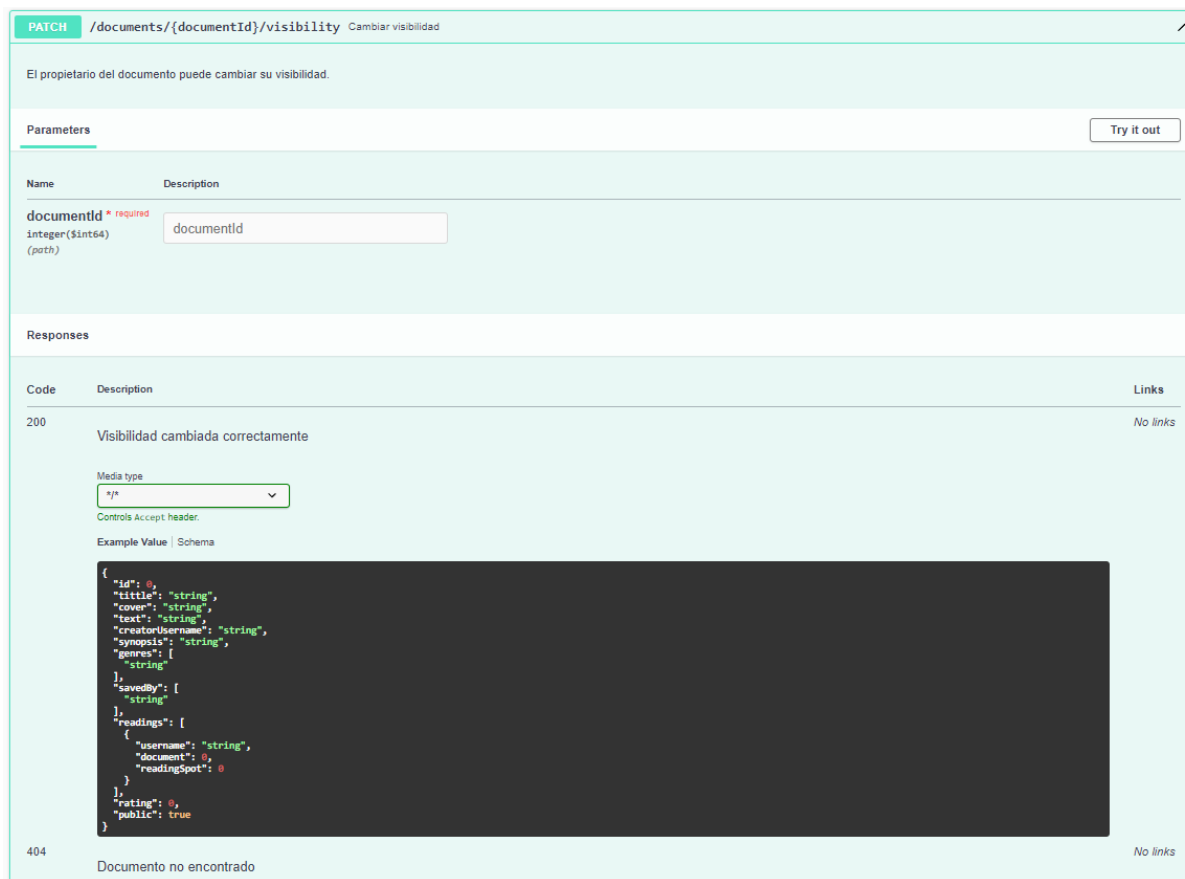


Ilustración 99. Método PATCH Controlador documentos

En la siguiente ilustración (Ilustración 100), se muestra el método GET adjunto a la dirección `/documents/genres`. En la ilustración se muestra la descripción del método y los parámetros a introducir. He decidido recalcar este método debido a que los parámetros que se deben introducir son de tipo Query. Estos parámetros en la interfaz de Swagger se introducen como si se trataran de un parámetro de ruta, sin embargo a nivel de petición desde el cliente la forma de enviar el parámetro difiere de la de los parámetros de ruta. Esta petición, como la mayoría de las peticiones devolverá el código HTTP OK en caso de que todo funcione según lo esperado y el código NOT_FOUND en caso de que uno de los géneros de los que se buscan documentos no exista. La respuesta de la petición será una lista de los documentos que cumplan las condiciones que se indican. Además, en este caso se incluyen parámetros como la

página o el tamaño de la página puesto que esta petición incluye paginado y devolverá un conjunto distinto de documentos según la página y el tamaño de página que se indique. Dado un conjunto total de documentos y un tamaño de página concreto, la petición con un número de página específico siempre devolverá los mismos documentos.

GET /documents/genres Obtener documentos por género y página

Obtiene los documentos públicos basados en los géneros proporcionados y el fragmento del título.

Parameters

Cancel

Name	Description
genres * required array[string] (query)	string string string Add string item
page * required integer(\$int32) (query)	page
pageSize * required integer(\$int32) (query)	pageSize
titleFragment * required string (query)	titleFragment

Execute

Responses

Code	Description	Links
200	Documentos devueltos correctamente	No links
404	Genero no encontrado	No links

Media type:

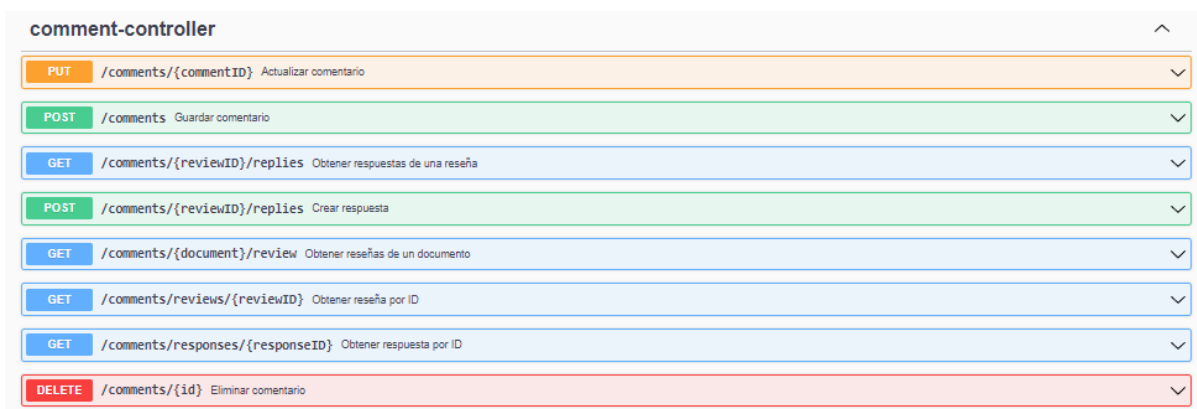
Controls: Accept header.

Example Value | Schema

```
{
  "id": 0,
  "title": "string",
  "cover": "string",
  "text": "string",
  "creatorUsername": "string",
  "synopsis": "string",
  "genres": [
    "string",
    "string"
  ],
  "savedBy": [
    "string"
  ],
  "readings": [
    {
      "username": "string",
      "document": 0,
      "readingSpot": 0
    }
  ],
  "rating": 0,
  "public": true
}
```

Ilustración 100. Método GET del controlador documento

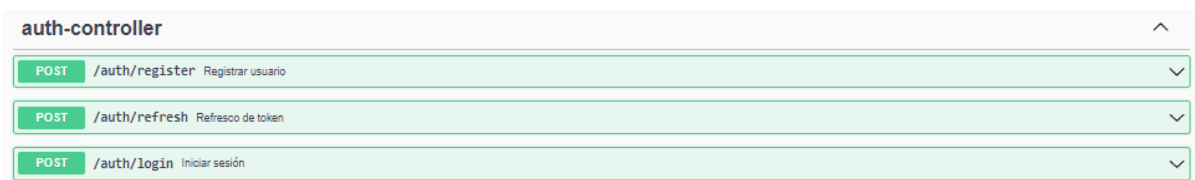
El siguiente controlador que comentaremos será el controlador de comentarios (Ilustración 101). Este controlador es bastante estándar, contiene métodos para la creación, edición y borrado de comentarios, además de métodos para la obtención de los comentarios, algunos de ellos para tipos de comentarios específicos, como las reseñas (Reviews) o respuestas (Responses). No comentaremos ningún método en específico, ya que siguen un esquema bastante básico. En caso de cualquier duda se puede revisar la documentación en la URL comentada anteriormente.



comment-controller		
PUT	/comments/{commentID}	Actualizar comentario
POST	/comments	Guardar comentario
GET	/comments/{reviewID}/replies	Obtener respuestas de una reseña
POST	/comments/{reviewID}/replies	Crear respuesta
GET	/comments/{document}/review	Obtener reseñas de un documento
GET	/comments/reviews/{reviewID}	Obtener reseña por ID
GET	/comments/responses/{responseID}	Obtener respuesta por ID
DELETE	/comments/{id}	Eliminar comentario

Ilustración 101. Controlador Comentario en interfaz de Swagger

Ahora hablaremos sobre el controlador Rest de autenticación (Ilustración 102). En él se encuentran los métodos relativos a la autenticación del usuario, siendo estos métodos el registro, el inicio de sesión y el refresco del token. Como se puede observar en la Ilustración 102 todos los métodos son de tipo POST. Todos ellos además devuelven el código HTTP 401 Unauthorized en caso de fallar en la operación. Tanto la función de inicio de sesión como la de refresco de Token devuelven un Token JWT, mientras que la de registro sólo notifica que la entidad ha sido creada mediante una respuesta HTTP CREATED.



auth-controller		^
POST	/auth/register	Registrar usuario
POST	/auth/refresh	Refresco de token
POST	/auth/login	Iniciar sesión

Ilustración 102. Controlador Comentario en interfaz de Swagger

Por último hablaremos sobre el controlador REST de actividad. Este es quizás el controlador más particular de todos. En su caso, contiene únicamente un endpoint de tipo GET. Se observa a simple vista que operaciones básicas, como por ejemplo la creación de las actividades mediante POST no es posible desde este controlador. Esto se debe a cómo funcionan las actividades. Las actividades registran las acciones de los usuarios, por lo tanto las acciones se crean cuando un usuario realiza una petición al servidor que se considera una actividad. Por ejemplo, si un usuario comienza a leer un documento, es decir, se crea una nueva entidad lectura mediante el uso de POST en el controlador lectura, esta petición no solo creará la lectura de la petición, si no que también creará un registro de esa actividad en el repositorio correspondiente de actividad. Por lo tanto, al crearse y eliminarse las actividades como consecuencia de otras peticiones, no se añade método POST a este controlador, puesto que además de innecesario, no tendrían uso alguno. El único método GET implementado, devuelve las actividades que han realizado los usuarios cuyos nombres de usuario se pasan como parámetro, tal y como se observa en la Ilustración 103. Además este método también incluye paginación, por lo que también requiere los parámetros de número de página y tamaño de página como hemos comentado en otros métodos con paginación.

activity-controller

GET /activity/recent Actividad reciente

Devuelve la información de las actividades que han realizado los usuarios adjuntos a la petición

Parameters [Try it out](#)

Name	Description
pageSize * required integer(\$int32) (query)	pageSize
pageNumber * required integer(\$int32) (query)	pageNumber
usernames * required array[string] (query)	

Responses

Code	Description	Links
200	OK	No links
	Media type: */ Controls: Accept header. Example Value Schema <pre>[{ "id": 0, "username": "string", "action": "string", "entityType": "USER", "entityId": "string", "timestamp": "2023-06-14T12:43:10.000Z" }]</pre>	
400	Bad Request	No links
	Media type: */ Example Value Schema <pre>string</pre>	

Ilustración 103. Controlador Actividad en interfaz de Swagger

Apéndice II. Manual de Usuario

A continuación mostraremos las diferentes funcionalidades que nuestro sistema permite.

En primer lugar, al iniciar la aplicación lo primero que veremos será la vista de identificación. En ella podremos registrarnos, en caso de queramos crear una nueva cuenta o iniciar sesión si queremos utilizar una cuenta ya existente. En nuestro caso, recomendamos la utilización de uno de los usuarios de prueba generados al inicio de la ejecución. Para utilizar uno de estos usuarios basta con introducir “*userX*” como usuario y “*passwordX*” como contraseña, siendo X un número entre 1 y 20. En la ilustración 104 se observa la vista de registro y en la ilustración 105 la vista de inicio de sesión.

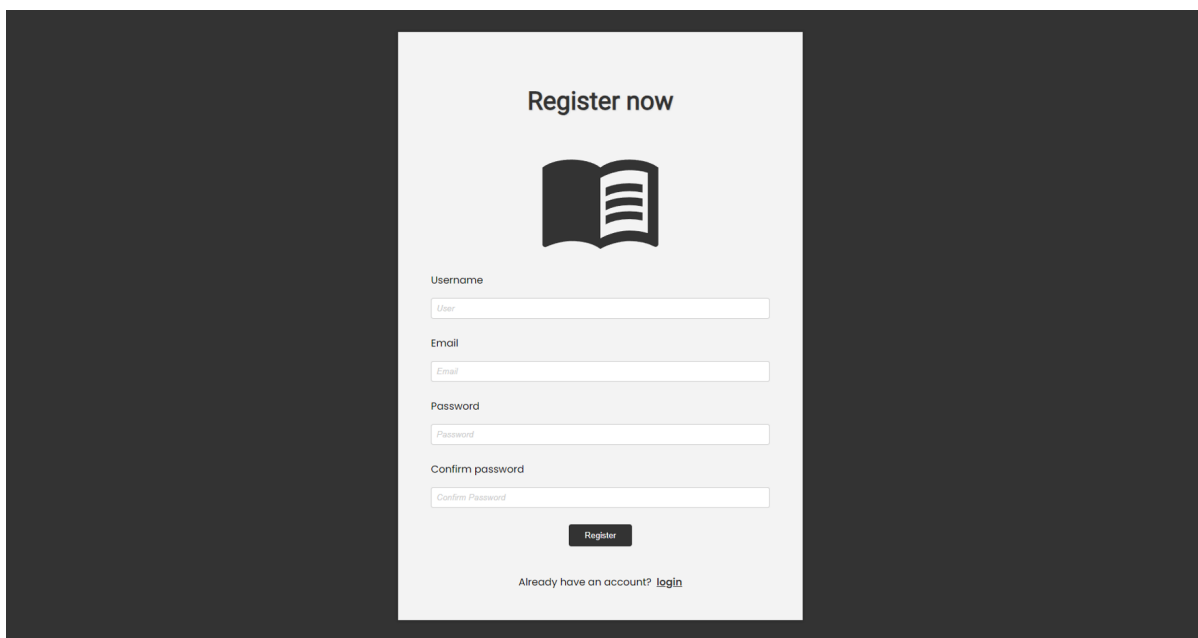


Ilustración 104. Vista de registro.

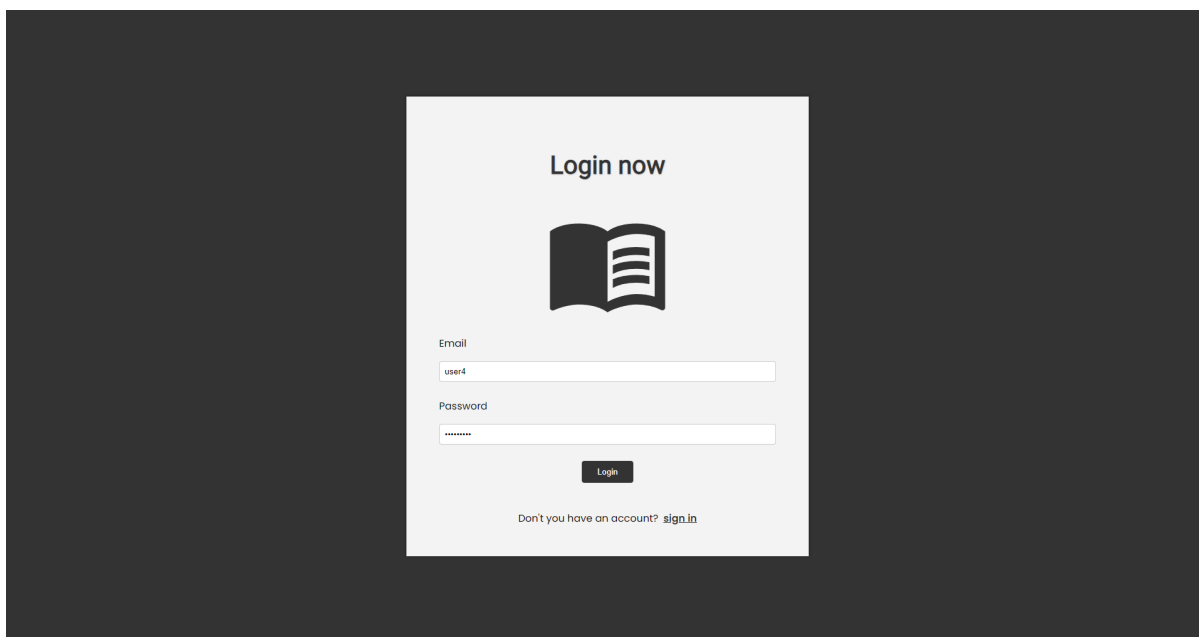


Ilustración 105. Vista de inicio de sesión.

Bajo los formularios de las ilustraciones 104 y 105, se observa un enlace que nos permite cambiar entre ambos formularios.

Una vez que hemos iniciado sesión, aparecemos en la vista principal, donde en caso de que utilicemos uno de los usuarios ya creados, podremos observar notificaciones que nos informan de las acciones realizadas por usuarios a los que seguimos, tal y como se observa en la Ilustración 106. Además, a la izquierda podremos observar un documento que la aplicación nos recomienda leer en función de la puntuación del documento y a la derecha se muestran usuarios que la aplicación nos recomienda seguir.

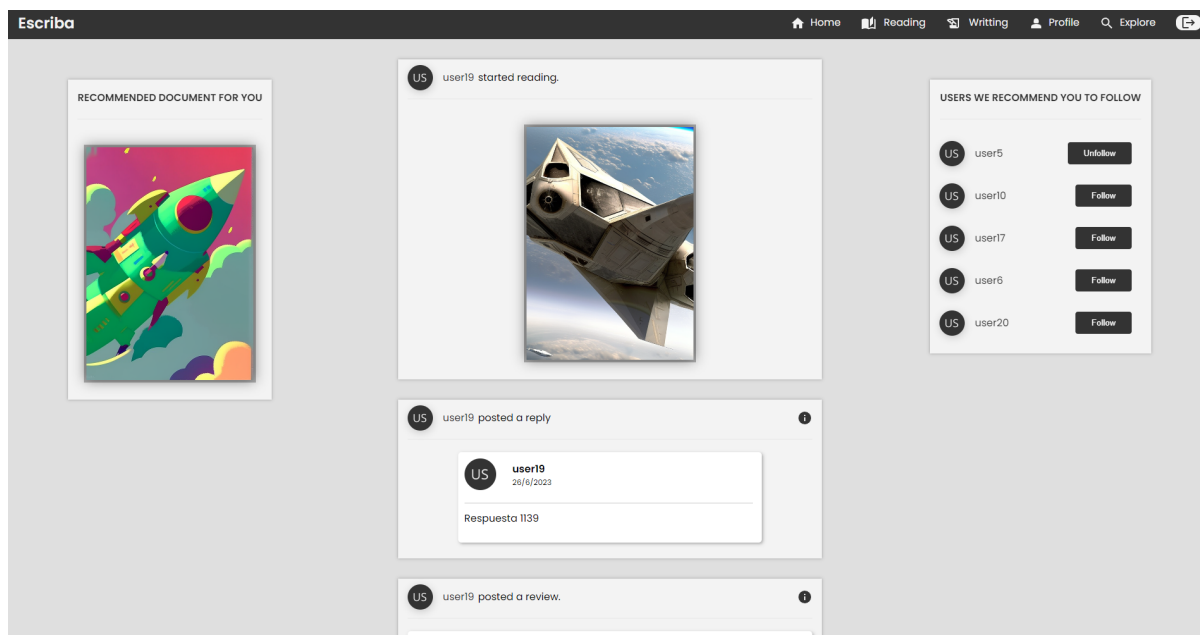


Ilustración 106. Vista principal.

Una vez en esta vista, podemos obtener más información sobre las notificaciones. Por ejemplo, si tenemos una notificación en la que un usuario ha respondido a una reseña de otro usuario en un documento, podremos acceder a la reseña a la que ha contestado y el documento donde se realizó pulsando sobre el botón de información en la notificación en concreto, tal y como se muestra en la ilustración 107.

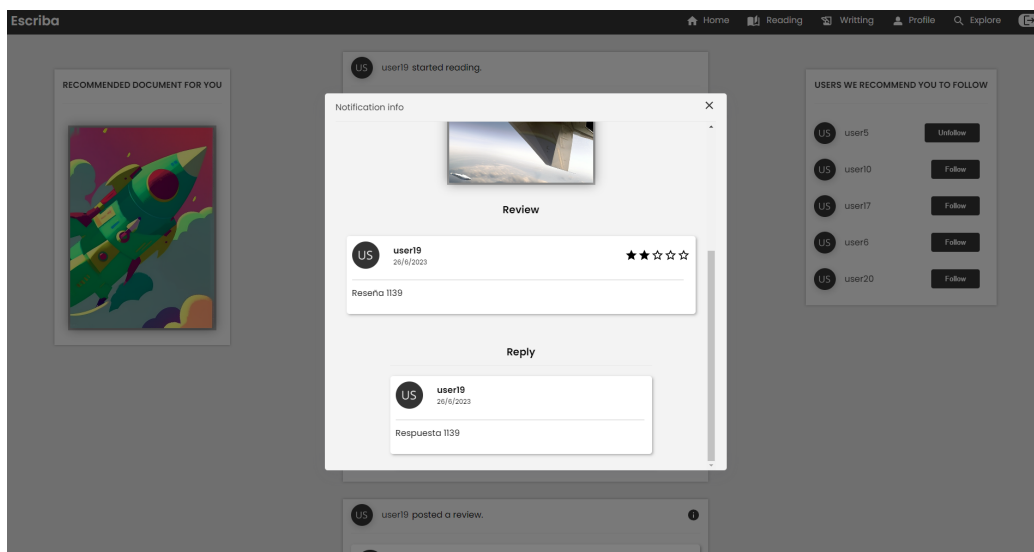


Ilustración 107. Información de notificación.

Utilizando la barra de navegación podremos desplazarnos a otras vistas. A continuación, nos desplazamos a la vista de perfil, donde tendremos acceso a los datos de nuestro propio usuario. Allí, tal y como se observa en la ilustración 108, observamos los documentos públicos que hemos creado. En nuestro caso, si utilizamos un usuario generado, tendremos dos documentos públicos.

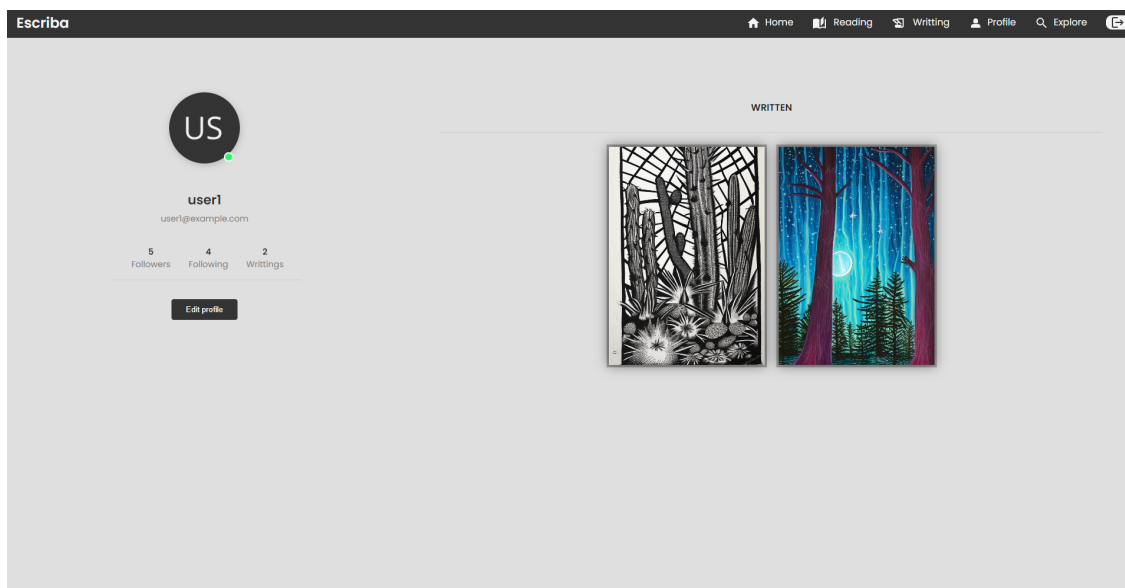


Ilustración 108. Vista de perfil.

Desde el perfil podemos observar el número de seguidores que tenemos y el número de usuarios a los que seguimos. Pulsando sobre estos números, se abre una ventana que nos permite ver los nombres e imágenes de perfil de estos usuarios. Además, pulsando en sus nombres, podremos acceder a su perfil, como veremos más tarde. En la ilustración 109 se observa un listado de los usuarios que siguen a nuestro usuario, tras pulsar sobre el botón que da acceso a ello.

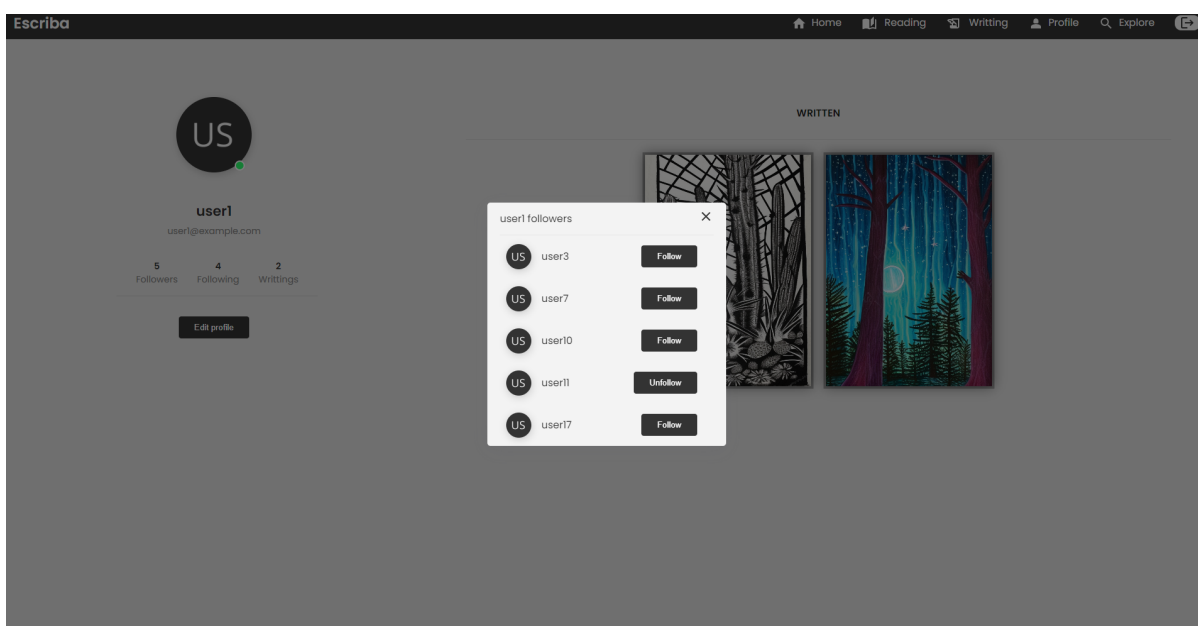


Ilustración 109. Seguidores de un usuario.

Además, en el perfil tendremos acceso a un editor de perfil que nos permite cambiar nuestro correo electrónico, descripción e imagen de perfil. Para ello, deberemos pulsar en el botón “*Edit profile*”, lo cual abrirá una ventana con el formulario en concreto, tal y como se observa en la ilustración 110. Para subir una imagen, simplemente deberemos pulsar sobre la imagen actual.

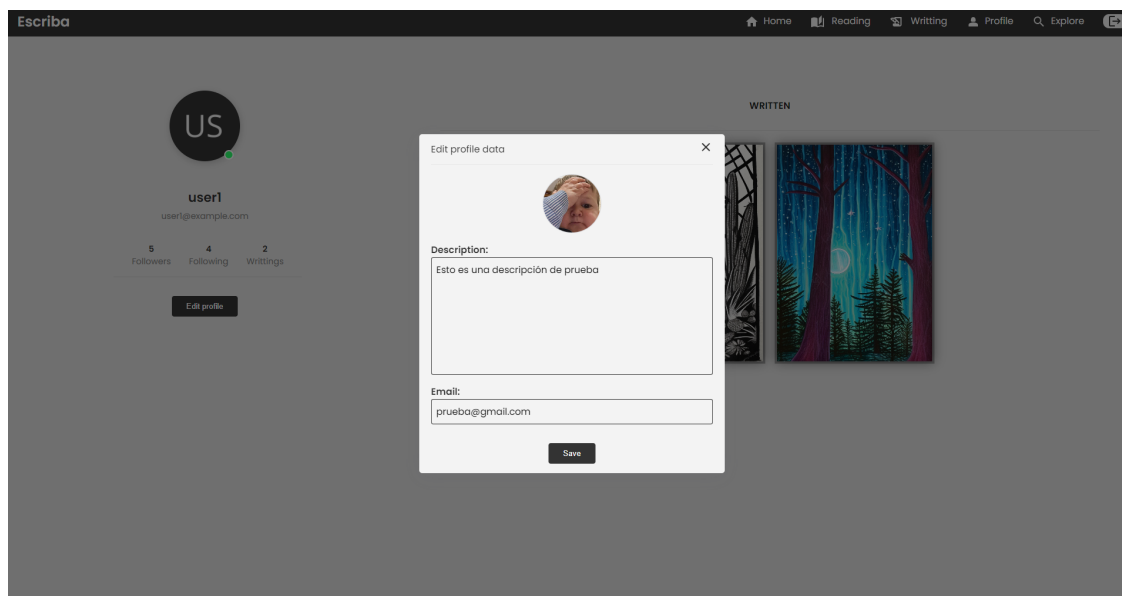


Ilustración 110. Formulario edición de perfil.

Una vez que pulsamos sobre “Save”, los datos se aplican al perfil, tal y como se muestra en la ilustración 111.

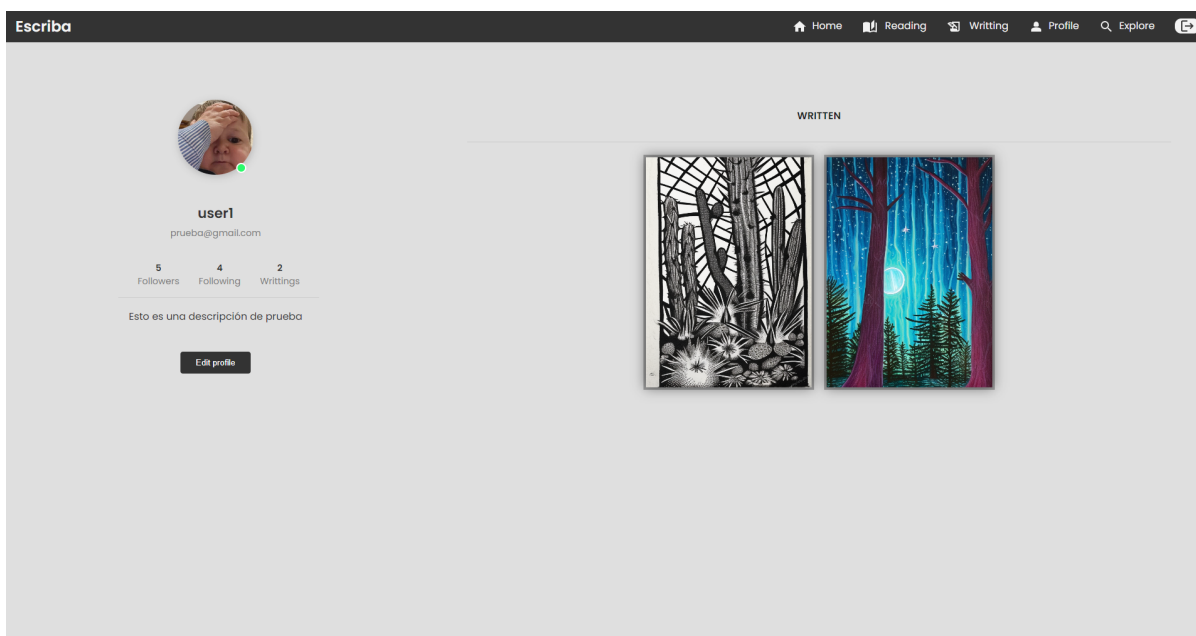


Ilustración 111. Resultados edición perfil.

Una vez hemos visto todas las funcionalidades de la vista perfil, pasaremos a la vista “Explore”. En esta vista los usuarios podrán buscar documentos en función de los géneros y título. En la Ilustración 112 se observa la vista al acceder a ella.

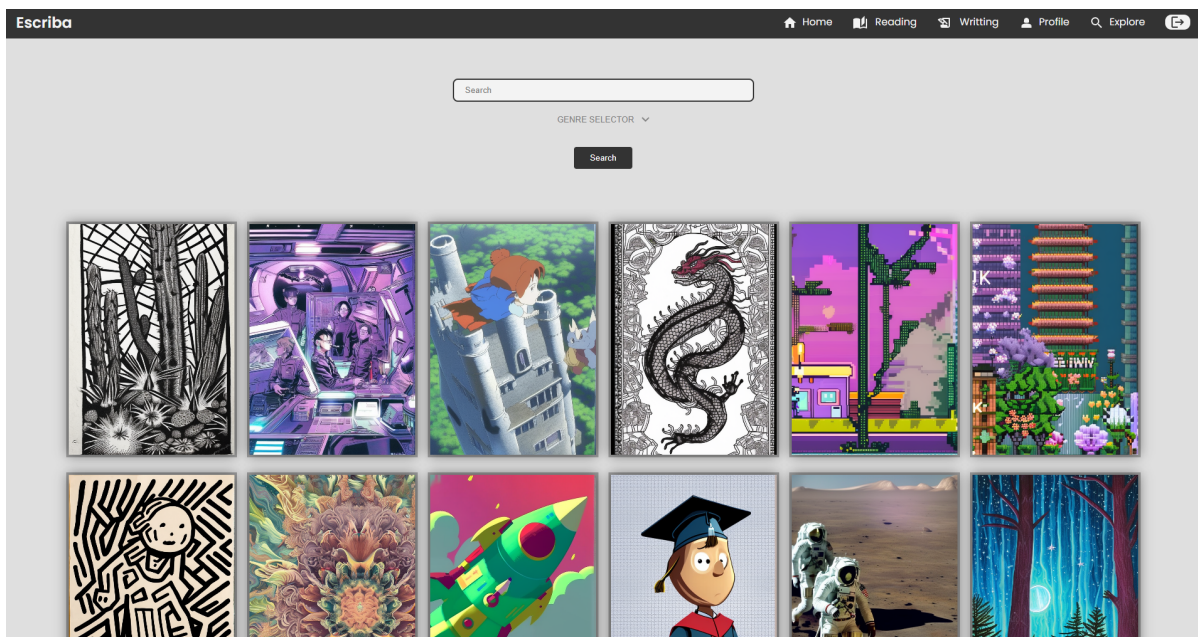


Ilustración 112: Vista explorar.

La vista contiene una barra de búsqueda, en la que se puede introducir texto con el cual se devolverán documentos y usuarios que coinciden con la búsqueda, tal y como se muestra en la ilustración 113. Al escribir “1” en la barra de búsqueda y ejecutar la búsqueda, se mostrarán todos los documentos con “1” en su título y algunos usuarios con “1” en su nombre. En caso de querer ver todos los usuarios que cumplan la condición podemos pulsar sobre “Explore users”, donde se abrirá una ventana con todos los usuarios que coinciden con la búsqueda.

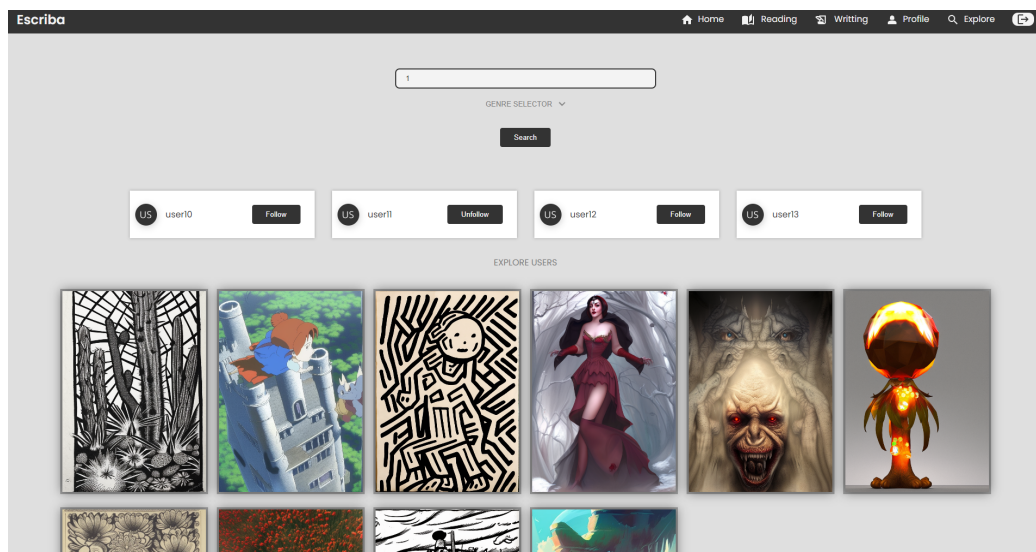


Ilustración 113. Ejemplo de búsqueda con barra.

En caso de abrir el desplegable de los géneros, veremos distintos géneros literarios que se pueden seleccionar para realizar una búsqueda de documentos de algunos de estos géneros. En la ilustración 114 se observa el desplegable y géneros seleccionados.

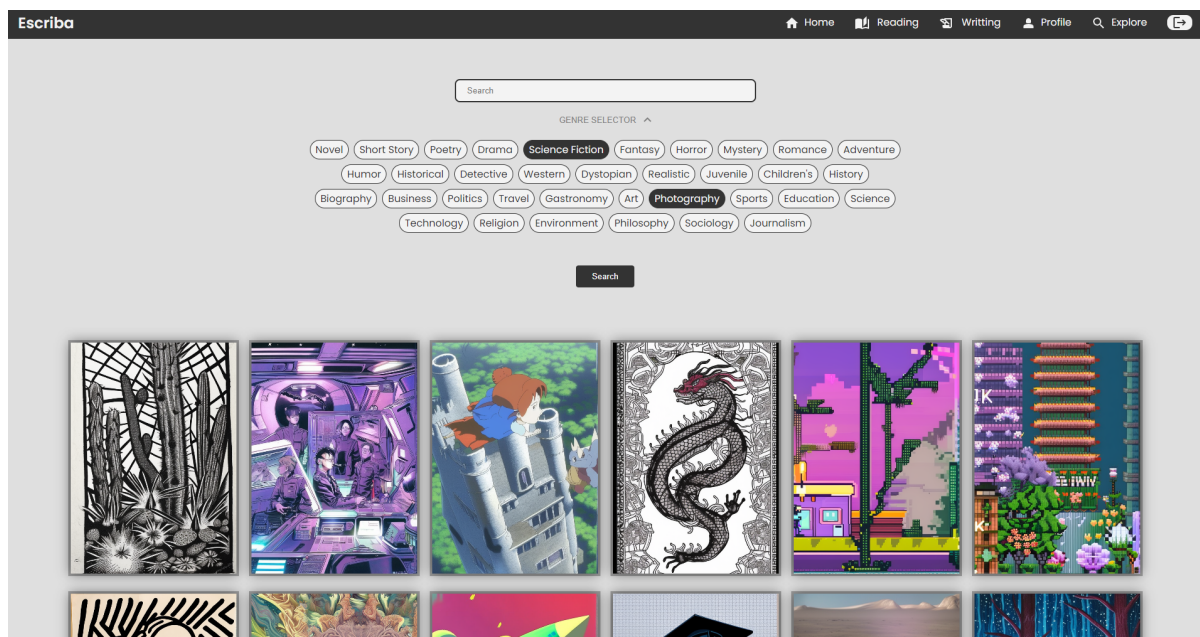


Ilustración 114: Búsqueda por géneros

Una vez seleccionamos los géneros y buscamos, se nos mostrarán los documentos que se clasifiquen dentro de al menos uno de esos géneros. En la ilustración 115 vemos el resultado de una de estas búsquedas.

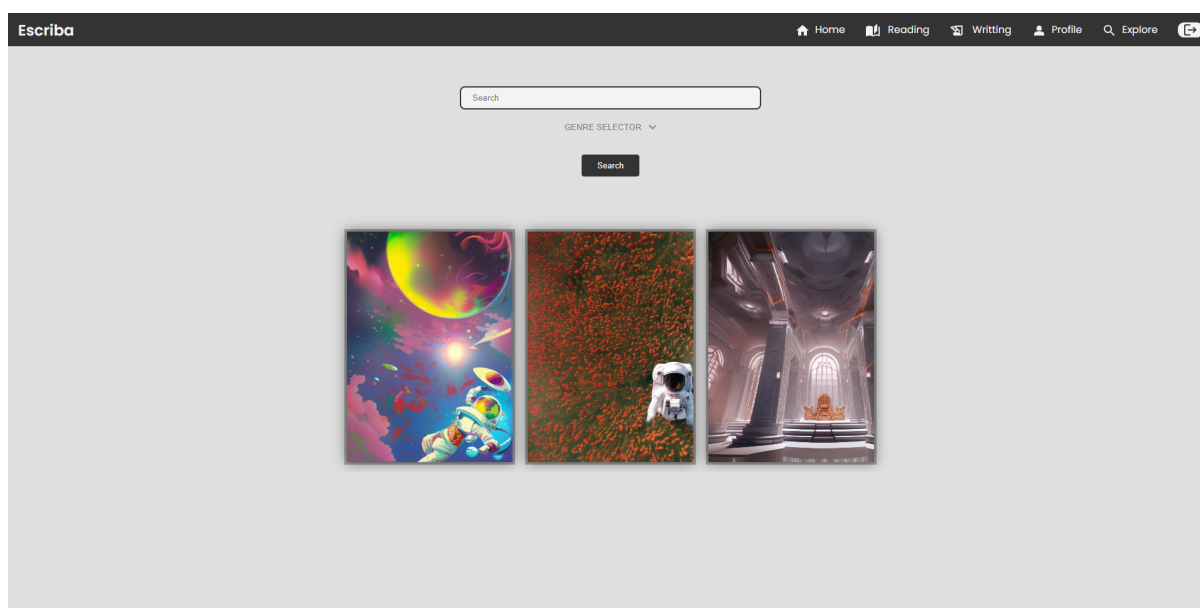
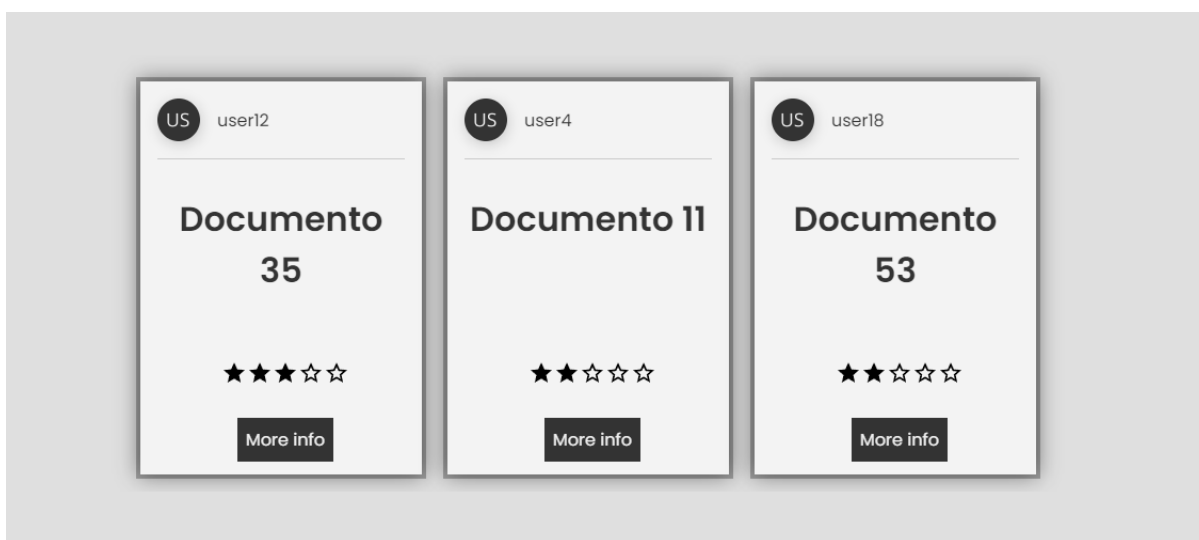


Ilustración 115. Resultado búsqueda por géneros

Al pulsar sobre uno de los documentos, tendremos acceso a información básica de los mismos, como se muestra en la ilustración 116, a simple vista veremos el creador del documento, el cual al pulsar en su nombre nos redirigirá a su perfil, la puntuación del documento, en la cual si pulsamos tendremos acceso a las reseñas y respuestas del mismo y el título del documento. Además tendremos acceso a un botón que nos dará más información del documento y más opciones.

*Ilustración 116. Información básica de documentos*

Al pulsar sobre más información, tendremos acceso a los géneros del documento, además de a su sinopsis, como se muestra en la ilustración 117.

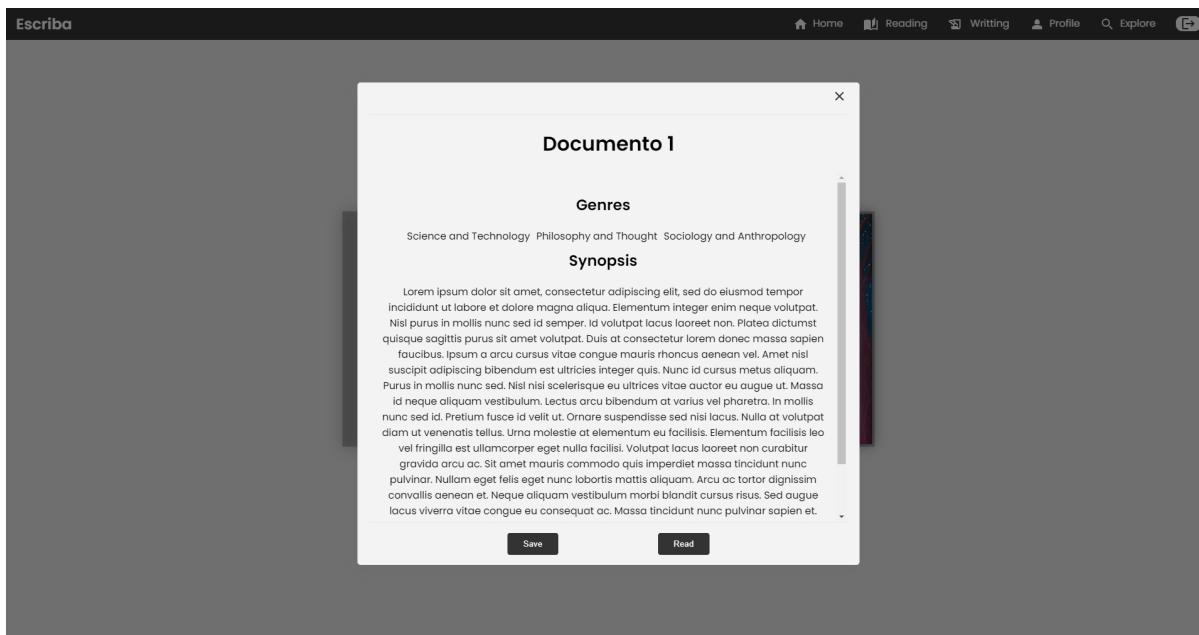


Ilustración 117: Información detallada de documentos

A continuación comentaremos la vista “*Reading*”, donde el usuario tendrá acceso a los documentos que está leyendo y a los documentos que ha guardado para leer más tarde. En la ilustración 118 se observa la vista lecturas de un usuario generado. La vista contiene una barra de filtrado para filtrar los documentos por título. Para que un documento se agregue a la pestaña de lecturas basta con que el usuario lo haya leído en una ocasión.

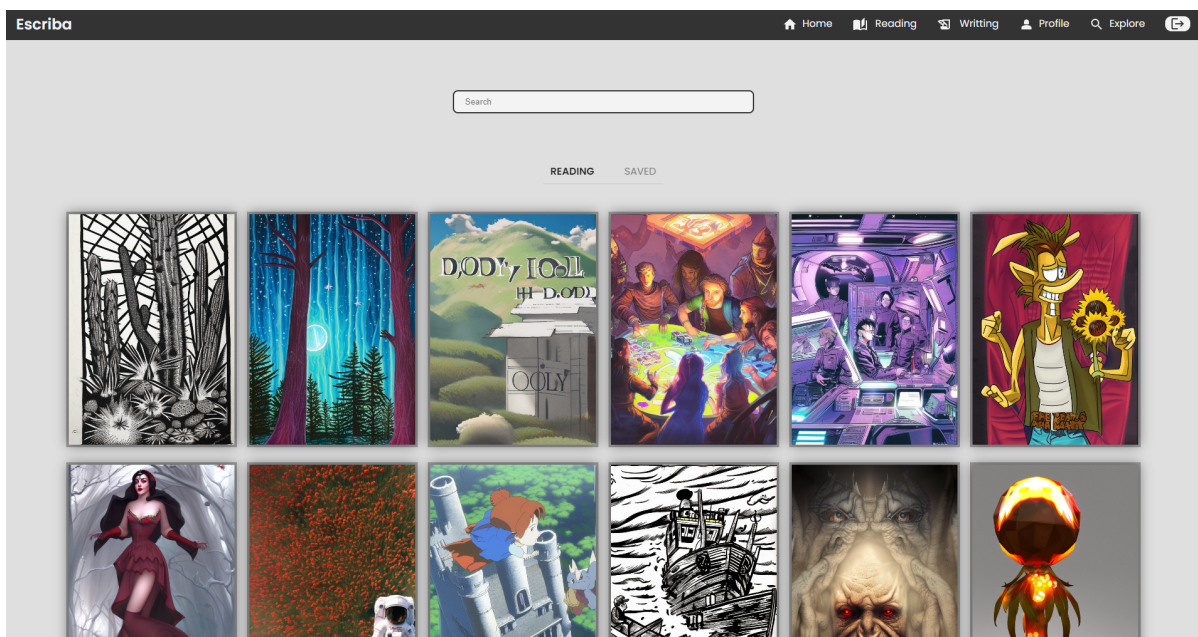


Ilustración 118. Vista lecturas

En la pestaña de guardados se incluirán los documentos que se hayan guardado. Para guardar un documento basta con pulsar el botón “Save” en la ventana de información detallada de un documento.

La vista “*Writting*” es una vista similar, también incluye una barra de filtrado y dos pestañas, una para tus documentos creados que son públicos y otra para los documentos privados. Además la vista incluye un botón para la creación de documentos, como se muestra en la ilustración 119.

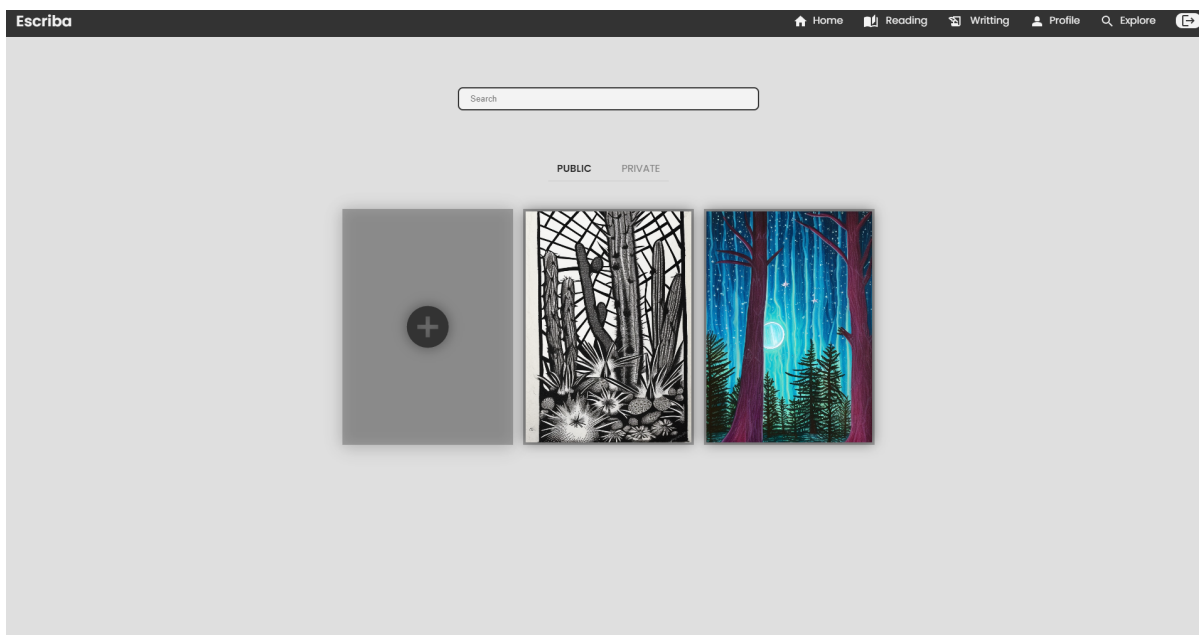


Ilustración 119: Vista escrituras

Al pulsar el botón de creación de documento se nos abre un formulario que se muestra en la ilustración 120 y 121. En el formulario hay distintos campos para aplicar un título, una sinopsis y unos géneros.

Ilustración 120. Primera parte formulario creación de documentos

Ilustración 121. Segunda parte formulario creación de documentos

El campo más destacable del formulario es el generador de portadas. Incluiremos el texto que describa la imagen que queramos generar y pulsamos el botón, lo cual comenzará la generación de la imagen, tal y como se muestra en la ilustración 122. Tras unos segundos, obtendremos como resultado una portada, como la que se muestra en la ilustración 123.

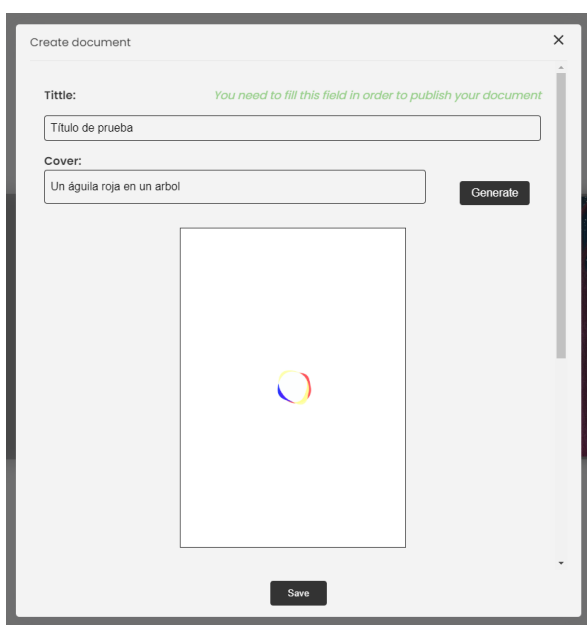


Ilustración 122. Campo portada en formulario de creación de documentos.

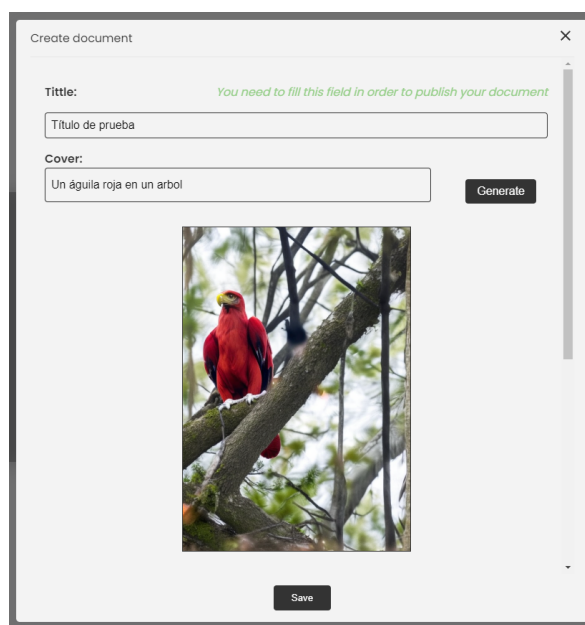


Ilustración 123. Segunda parte formulario creación de documentos

Una vez que estemos contentos con el resultado, podemos guardar el documento. Esto nos redirigirá a la vista del editor de documentos, donde podremos escribir el contenido del documento aplicando distintos formatos al texto. En la ilustración 124 se muestra la vista junto a texto con formato. El editor de texto permite además incluir imágenes que el usuario tenga en su sistema. El contenido del documento se guarda cada 3 segundos, por lo que no es necesario guardar el contenido manualmente.

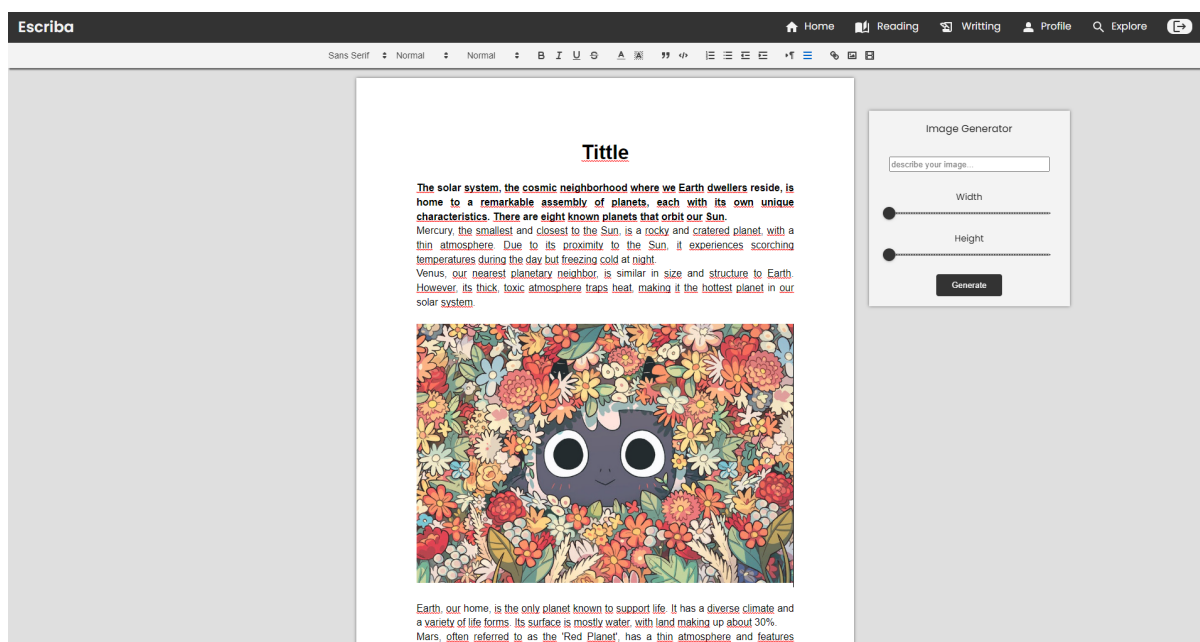


Ilustración 124: Vista editor de texto.

El editor de texto permite también incluir ilustraciones generadas por IA. Para ello podemos utilizar el generador de imágenes que se encuentra a la derecha del editor. Los “sliders” nos permiten ajustar el tamaño de la imagen y en el campo de texto debemos introducir una descripción de la imagen a generar. La imagen se creará en la última selección del usuario. En la ilustración 125 se observa un ejemplo de los campos de generación y la ilustración 126 el resultado de esa generación en concreto.

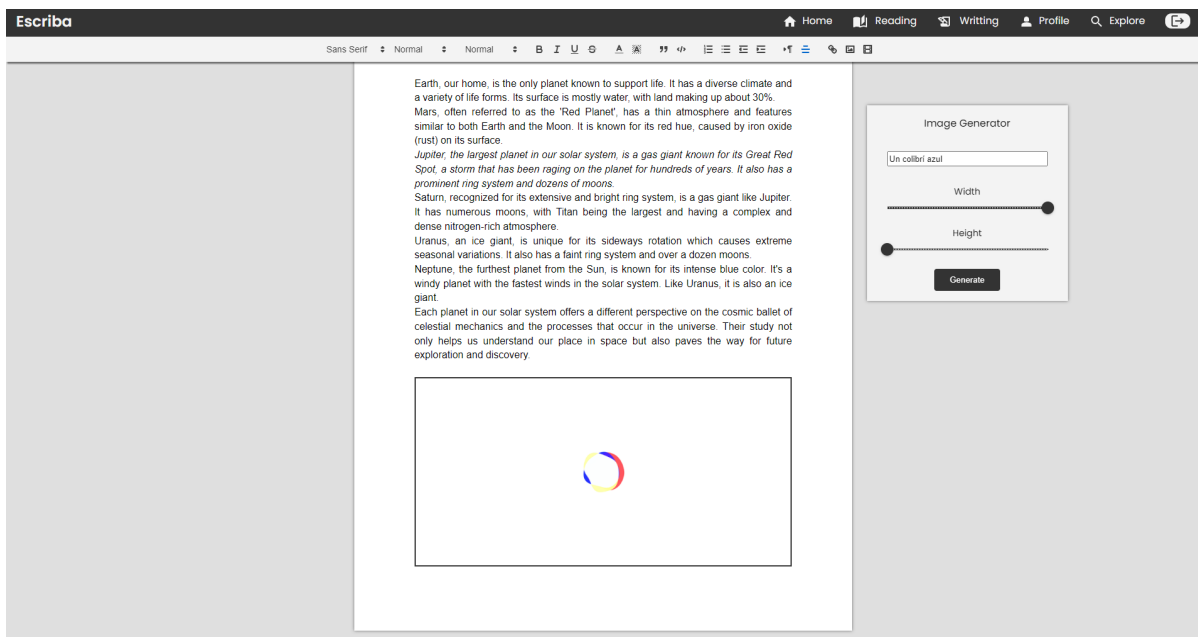


Ilustración 125. Ejemplo generación de ilustración.

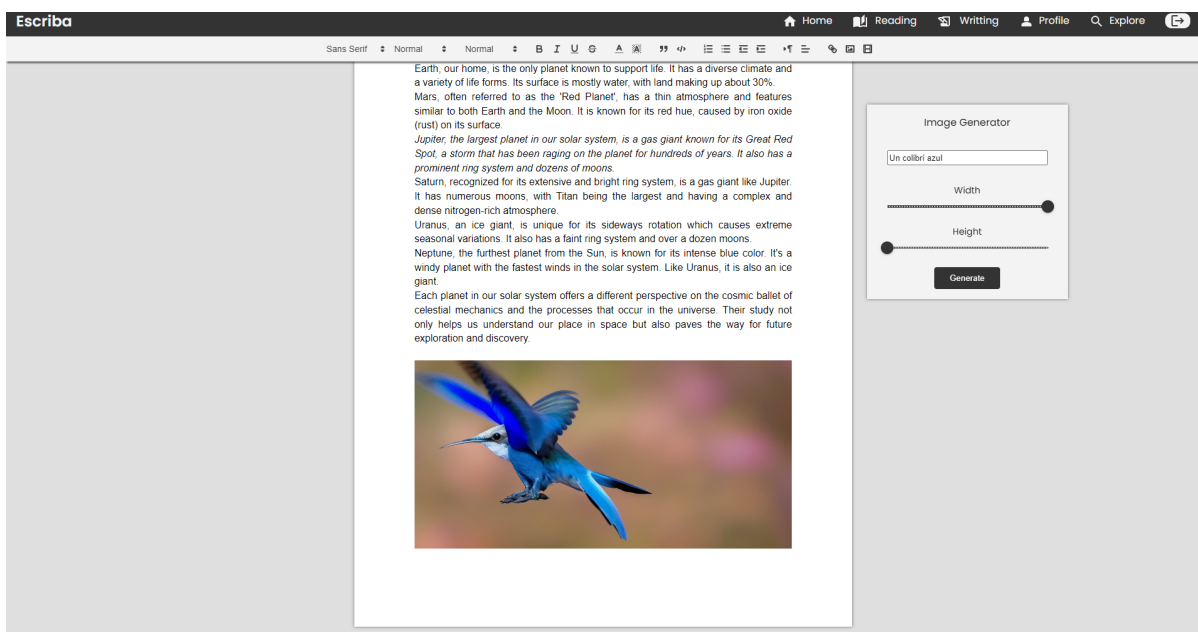


Ilustración 126. Resultado generación de ilustración.

Si salimos y volvemos a la vista de escritura, en la pestaña de documentos privados, podremos observar que ahí se encuentra el documento que hemos creado, tal y como se muestra en la ilustración 127.

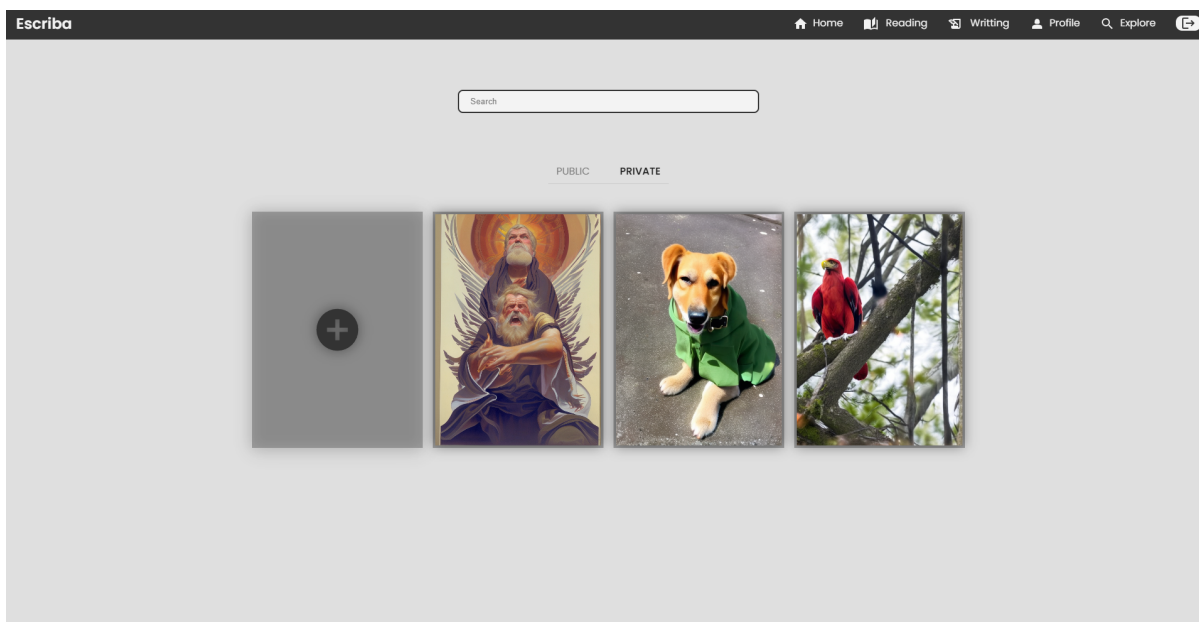


Ilustración 127. Acceso a documentos creados.

En caso de querer publicar el documento podremos hacerlo desde el reverso de la tarjeta, donde existen las opciones de eliminar, editar y publicar documentos. En las ilustraciones 128 y 129 se muestran las opciones de eliminar y publicar documentos respectivamente. Si tu documento cumple las restricciones para poder ser publicado, la ventana te permitirá publicarlo. Una vez el documento se publique será visible para todos los usuarios.

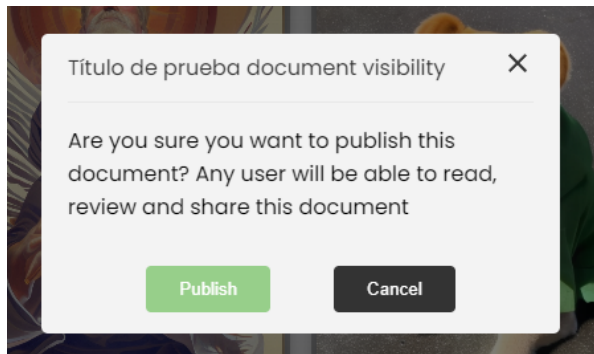
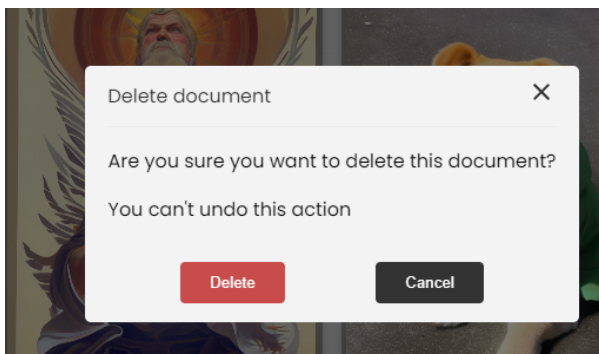


Ilustración 128. Confirmación eliminar documento Ilustración 129. Confirmación publicar documento

A continuación comentaremos cómo leer un documento. Para leer un documento simplemente debemos acceder a los detalles de un documento en el reverso de su tarjeta y allí observaremos un botón que nos permitirá acceder a la vista de lectura. La vista de lectura se muestra en la ilustración 130, y consta simplemente una vista con el texto y un “slider” que controla el desplazamiento automático. Cuanto más valor tenga, más velocidad tendrá el desplazamiento. Bajo él se encuentra un botón para detener o comenzar el desplazamiento automático.

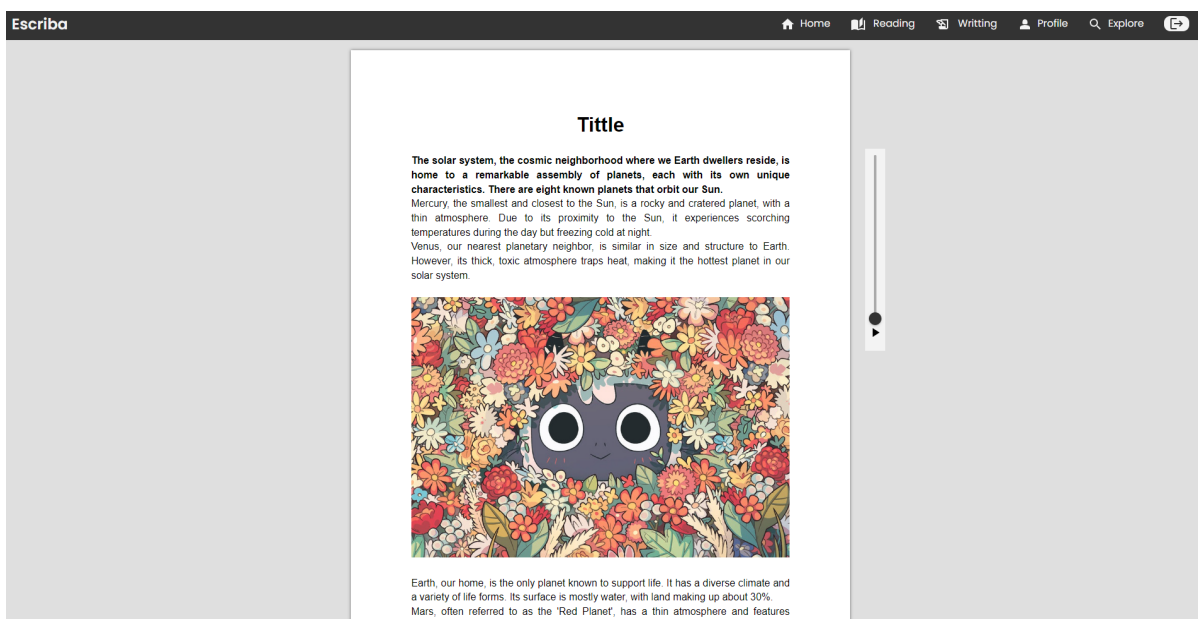
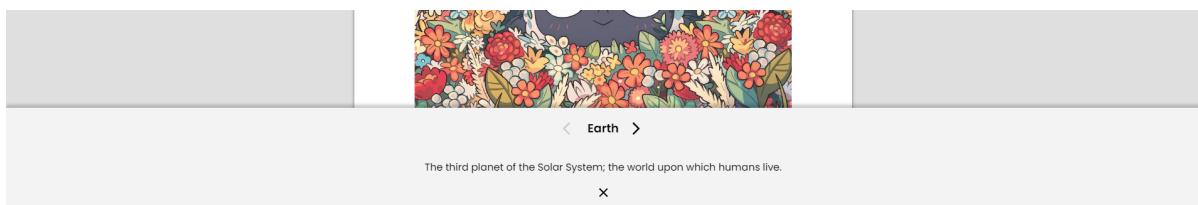


Ilustración 130. Vista lectura

Otra funcionalidad del documento es el diccionario. Al pulsar sobre una palabra se abrirá una ventana que mostrará todas las definiciones de una palabra, tal y como se observa en la ilustración 131. Es importante recalcar que esta funcionalidad sólo está disponible para palabras en inglés y no encontrará definiciones de palabras en español. Podremos desplazarnos entre las distintas definiciones pulsando en las flechas que se encuentran junto a la palabra. Para cerrar el diccionario simplemente pulsamos sobre la cruz en el inferior del diccionario. El diccionario no siempre aparece en el inferior de la pantalla, si pulsamos en una palabra que se encuentre en la parte inferior de la pantalla el diccionario aparecerá en la parte superior para no entorpecer la lectura.

*Ilustración 131: Diccionario*

Por último, explicaremos cómo crear una reseña y una respuesta. Para ello deberemos pulsar sobre la valoración de estrellas de un documento, en el reverso de su tarjeta. Una vez pulsemos ahí, se nos abrirá una ventana con los comentarios de ese documento, tal y como se muestra en la ilustración 132.

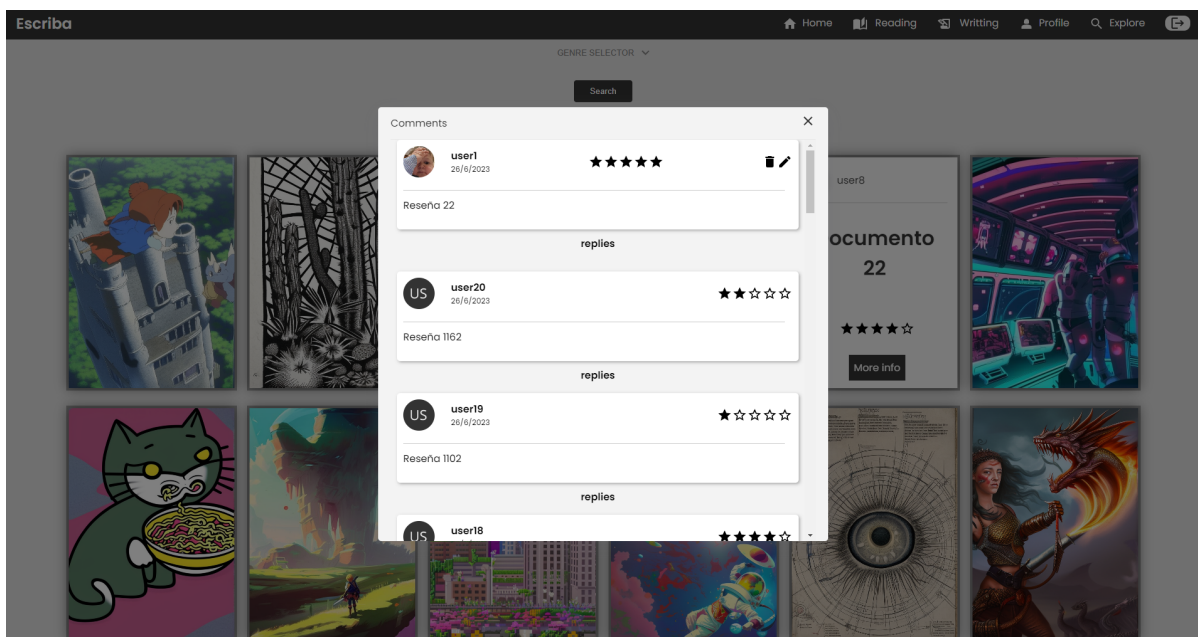


Ilustración 132. Ventana comentarios

Como sabemos sólo se puede crear una reseña por documento, y en la ilustración anterior (Ilustración 132) se observa como ya existe una reseña de nuestra parte. En caso de querer escribir otra, podemos eliminar la reseña o editarla. En ambos casos se abrirá un campo para la creación de la reseña donde podremos introducir el texto y las estrellas que deseemos.

En caso de eliminar la reseña, obtendremos la siguiente interfaz (Ilustración 133).

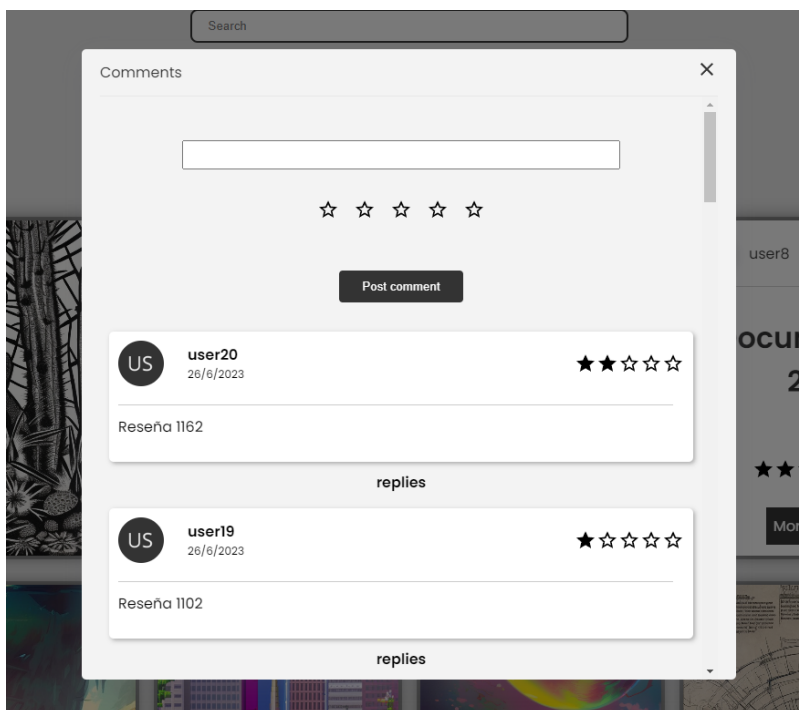


Ilustración 133. Campo creación de reseñas

Para responder a una reseña, simplemente pulsamos sobre el desplegable “replies” de la reseña en concreto, lo cual mostrará las respuestas de la reseña y un campo para escribir tu propia respuesta como se muestra en la ilustración 134.

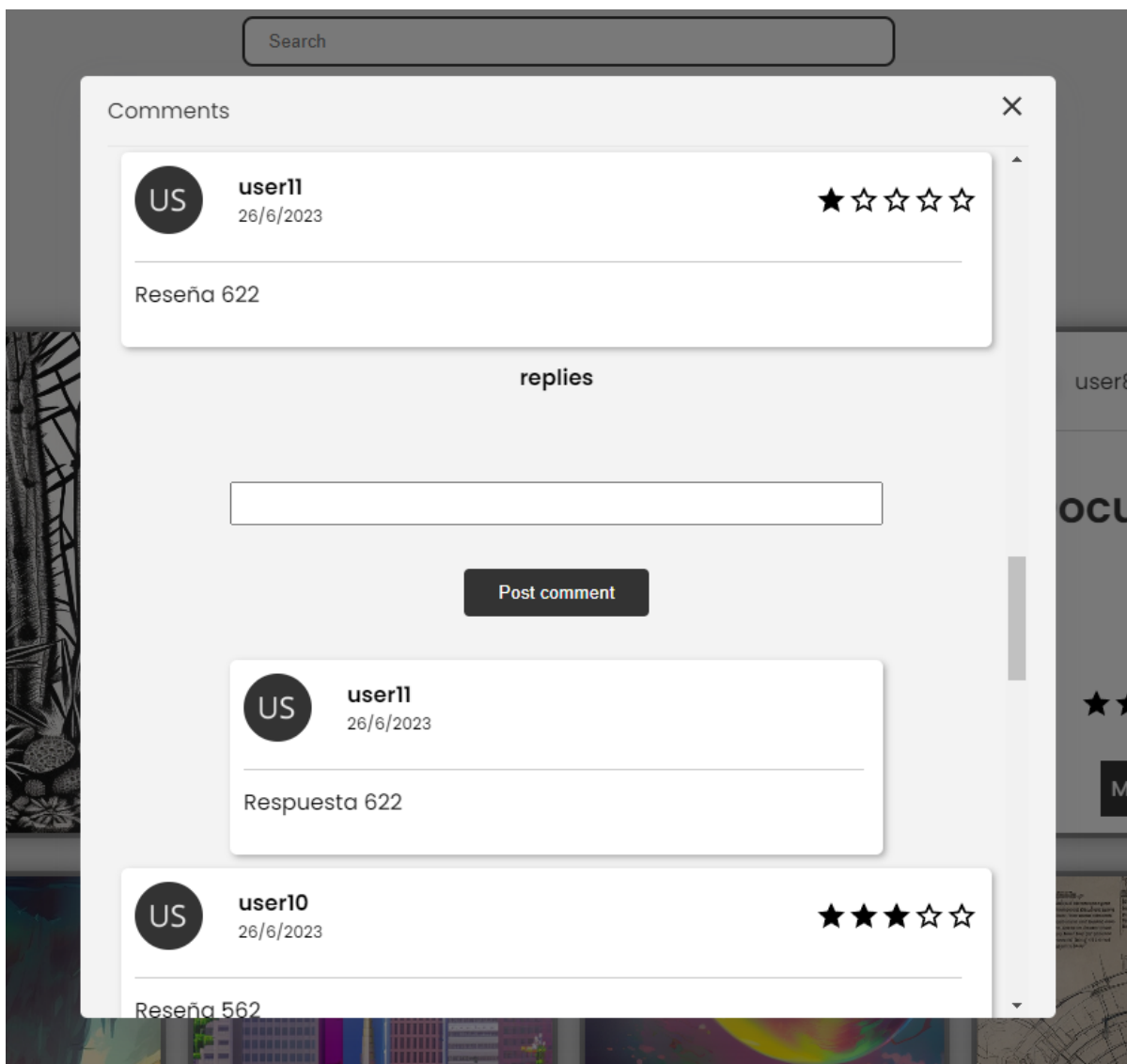


Ilustración 134. Respuestas a una reseña

Como podemos ver la aplicación contiene gran número de funcionalidades, pero la mayoría de ellas son intuitivas, por lo que no es necesario profundizar demasiado en la mayoría de ellas.

Apéndice III. Descripción de contenidos adicionales suministrados

En el fichero MaterialesComplementarios.zip que acompaña a esta memoria se suministran contenidos adicionales organizados en los siguientes directorios, tal y como se muestra en la ilustración 135. Estos directorios son el directorio “Proyectos”, el directorio “Ejecutables” y el directorio “Diagramas”.

Nombre	Fecha de modificación	Tipo	Tamaño
 Diagramas	27/06/2023 3:25	Carpeta de archivos	
 Ejecutables	27/06/2023 4:42	Carpeta de archivos	
 Material complementario	27/06/2023 21:23	Carpeta de archivos	
 Proyectos	27/06/2023 0:48	Carpeta de archivos	

Ilustración 135. Contenido del directorio principal de la entrega

Dentro del directorio “Proyectos”, encontramos las carpetas de los respectivos proyectos cliente y servidor, tal y como se muestra en la ilustración 136.

Nombre	Fecha de modificación	Tipo	Tamaño
 Escriba-Client	27/06/2023 2:21	Carpeta de archivos	
 Escriba-Server	27/06/2023 0:51	Carpeta de archivos	

Ilustración 136. Contenido del directorio Proyectos

En el apartado de “Ejecutables”, se encuentran los ejecutables del cliente y servidor, además de el cuaderno de Google Colab en formato archivo y en un enlace a un cuaderno ya abierto que es accesible mediante una cuenta UJA. En la ilustración 137 mostramos el contenido de la carpeta “Ejecutables”

Nombre	Fecha de modificación	Tipo	Tamaño
 Escriba-Client	27/06/2023 4:36	Carpeta de archivos	
 Escriba-Diffuzzers-API	27/06/2023 4:43	Carpeta de archivos	
 Escriba-Server	27/06/2023 2:24	Carpeta de archivos	

Ilustración 137. Contenido del directorio Ejecutables

Por último, en el directorio “Diagramas” encontramos lo que se muestra en la ilustración 138, los diferentes diagramas en formato imagen y el proyecto completo de Visual Paradigm.

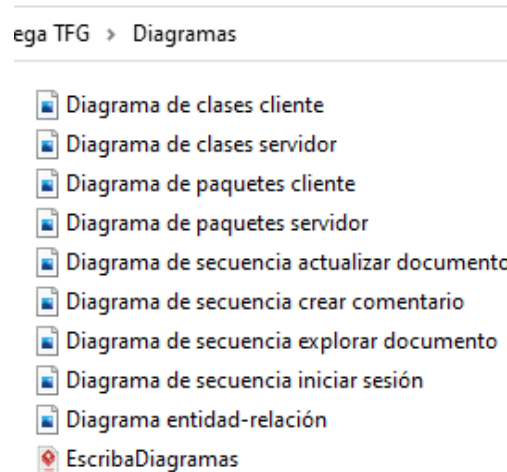


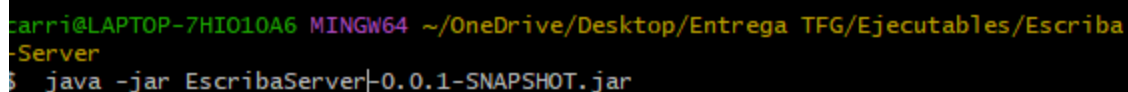
Ilustración 138. Contenido del directorio Diagramas

La carpeta de material complementario contiene el archivo yaml generado por swagger, que puede utilizarse para acceder a la documentación o como generador de API REST a partir de la especificación.

Apéndice IV. Manual de Instalación del sistema

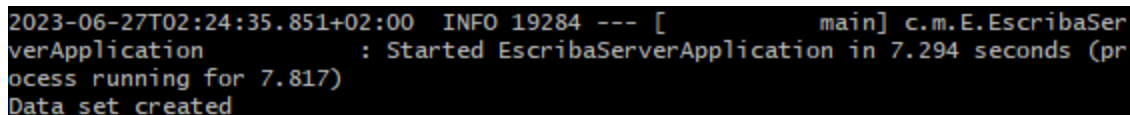
Una vez que hemos comentado los contenidos del material entregado podemos explicar cómo realizar una instalación y despliegue de ambas aplicaciones entregadas.

Para realizar un despliegue del servidor, debemos dirigirnos al directorio “Ejecutables\Escriba-Server”. Una vez en el repositorio, mediante el uso de una consola o CMD ejecutaremos la orden “`java -jar EscribaServer-0.0.1-SNAPSHOT.jar`”. La aplicación comenzará a ejecutarse y sabremos cuando está lista porque la consola devolverá el mensaje “*Data set created*” (El proceso de generación de los datos tarda unos segundos), lo que nos indica que el conjunto de datos de prueba ya ha sido generado. En la ilustración 139 se muestra la orden a ejecutar, mientras que en la ilustración 140 se observa la consola cuando la aplicación se ha ejecutado al completo.



```
carri@LAPTOP-7HI010A6 MINGW64 ~/OneDrive/Desktop/Entrega TFG/Ejecutables/Escriba-Server
$ java -jar EscribaServer-0.0.1-SNAPSHOT.jar
```

Ilustración 139. Orden ejecución del servidor.



```
2023-06-27T02:24:35.851+02:00 INFO 19284 --- [          main] c.m.E.EscribaServerApplication : Started EscribaServerApplication in 7.294 seconds (process running for 7.817)
Data set created
```

Ilustración 140. Servidor desplegado completamente.

A continuación es el turno de desplegar la API generadora de imágenes. Para ello, simplemente nos dirigimos al enlace al cuaderno de google o abrimos el archivo en nuestro propio cuadro, ambos se encuentran en el directorio “\Ejecutables\Escriba-Diffuzzers-API”. Una vez tengamos el archivo en nuestro cuaderno, seleccionamos la opción “Ejecutar todas las celdas” en la barra de herramientas, tal y como se muestra en la siguiente ilustración 141.

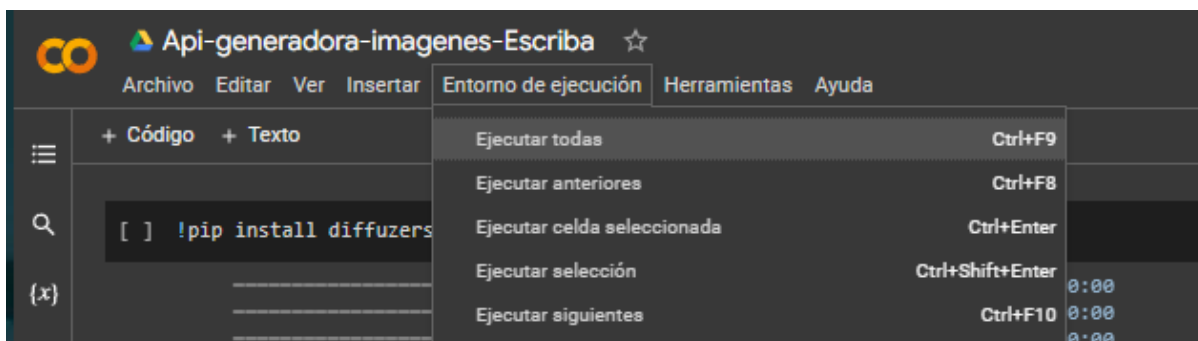


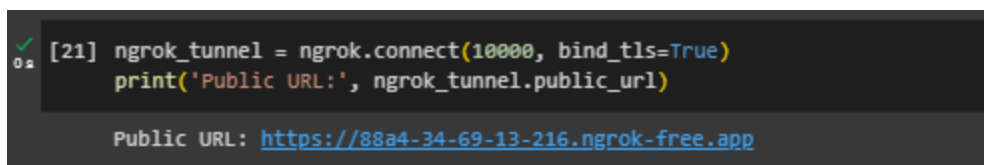
Ilustración 141. Ejecución de la API de generación de imágenes.

Una vez seleccionada esa opción, las casillas comenzarán a ejecutarse. Sabremos que la API se ha desplegado correctamente y está lista para ser usada porque la última casilla mostrará la salida que se muestra en la ilustración 142.

```
!diffuzzers api --port 10000
...
2023-06-27 01:58:28.292654: I tensorflow/core/platform/cpu_feature_guard.cc:182] This TensorFlow binary is optimized to use available CPU instructions in performance-critical operations.
To enable the following instructions: AVX2 FMA, in other operations, rebuild TensorFlow with the appropriate compiler flags.
2023-06-27 01:58:29.597374: W tensorflow/compiler/tf2tensorrt/utils/py_utils.cc:38] TF-TRT Warning: Could not find TensorRT
INFO: Started server process [2900]
2023-06-27 01:58:32.533 INFO uvicorn.error: Started server process [2900]
INFO: Waiting for application startup.
2023-06-27 01:58:32.533 INFO uvicorn.error: Waiting for application startup.
> INFO 00000 Starting Diffuzzers API 00000
> INFO TextImage Model: stabilityai/stable-diffusion-2-1
> INFO TextImage Pipeline: Vanilla
> INFO Inpainting Model: None
> INFO Device: cuda
> INFO Output Path: None
> INFO Token Identifier:
> INFO Token Embeddings URL:
> INFO Loading x2img model...
You have disabled the safety checker for <class 'diffusers.pipelines.stable_diffusion.pipeline_stable_diffusion_img2img.StableDiffusionImg2ImgPipeline'> by passing 'safety_checker=None'. Ensure that you
You have disabled the safety checker for <class 'diffusers.pipelines.stable_diffusion.pipeline_stable_diffusion_instruct_pix2pix.StableDiffusionInstructPix2PixPipeline'> by passing 'safety_checker=None'. Ensure that you
> INFO Loading inpainting model...
> INFO API is ready to use!
INFO: Application startup complete.
2023-06-27 01:59:07.736 INFO uvicorn.error: Application startup complete.
INFO: Uvicorn running on http://127.0.0.1:10000 (Press CTRL+C to quit)
2023-06-27 01:59:07.737 INFO uvicorn.error: Uvicorn running on http://127.0.0.1:10000 (Press CTRL+C to quit)
```

Ilustración 142. Ejemplo de API Generadora de imágenes desplegada.

Una vez que la API esté desplegada, debemos copiar el enlace de generación de imágenes para incluirlo en el cliente y que la comunicación se establezca correctamente. La URL se genera en una de las casillas del cuaderno, en concreto la que se muestra en la ilustración 143.



```
[21] ngrok_tunnel = ngrok.connect(10000, bind_tls=True)
      print('Public URL:', ngrok_tunnel.public_url)

Public URL: https://88a4-34-69-13-216.ngrok-free.app
```

Ilustración 143. Ejemplo de API Generadora de imágenes desplegada.

Una vez que tenemos el enlace y la API se ha desplegado, debemos acceder al archivo que se encuentra en el ejecutable del cliente, en concreto al archivo que maneja las variables de entorno, accedemos al directorio “\Ejecutables\Escriba-Client\env” y lo abrimos con cualquier editor de texto. Ahí observaremos un campo para incluir la URL que hemos copiado, sustituiremos la URL del archivo por el enlace que hemos creado en el cuadro, tal y como se muestra en la ilustración 144. Una vez hemos incluido la URL, guardamos el archivo.

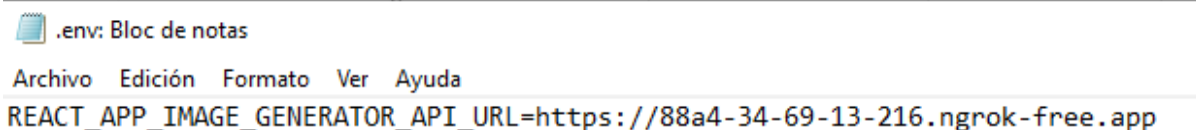


Ilustración 144. Sustitución URL generación de imágenes.

Tras esto, podremos desplegar la aplicación del cliente. Para ellos nos desplazaremos al directorio “/Ejecutables/Escriba-Client” y ejecutaremos la orden “npm install”, la cual descargará todas las dependencias que nuestro proyecto necesita y que se encuentran indicadas en el archivo “package.json”. Cuando la instalación termine,

ejecutaremos el comando “*npm start*”, el cual abrirá el navegador con una instancia de la aplicación cliente.