Universidad de Cádiz
Escuela Superior de Ingeniería

TRABAJO DE FIN DE GRADO

GRADO EN INGENIERÍA INFORMÁTICA

MADRID EN VIVO: UNA APLICACIÓN GUÍA DE EVENTOS DE LA CIUDAD DE MADRID PARA DISPOSITIVOS MÓVILES ANDROID, QUE GENERA NOTIFICACIONES GRACIAS AL PROCESAMIENTO DE DATOS EN TIEMPO REAL

AUTOR: ALBERTO ARAGÓN TORRALBA

Puerto Real, junio 2025

Universidad de Cádiz
Escuela Superior de Ingeniería

TRABAJO DE FIN DE GRADO

GRADO EN INGENIERÍA INFORMÁTICA

MADRID EN VIVO: UNA APLICACIÓN GUÍA DE EVENTOS DE LA CIUDAD DE MADRID PARA DISPOSITIVOS MÓVILES ANDROID, QUE GENERA NOTIFICACIONES GRACIAS AL PROCESAMIENTO DE DATOS EN TIEMPO REAL

DIRECTORA: GUADALUPE ORTIZ BELLOT
AUTOR: ALBERTO ARAGÓN TORRALBA

Puerto Real, junio 2025

Agradecimientos

A mis padres, por educarme en la constancia y en el saber posponer la recompensa. En especial a mi madre, por ser mi apoyo incondicional y por enseñarme que la diferencia está en hacer las cosas bien. Eres todo un ejemplo a seguir mamá. A mi pareja, por compartir conmigo cada pequeña frustración y hacerme andar cuando ni siquiera yo sabía si sería capaz de levantarme. Roberto, te quiero. A todas esas personas que me tendieron la mano desde el principio aunque no se lo pidiese. Javi, eres ejemplo de compañerismo. Gracias Andrea, por devolverme la ilusión por seguir desarrollando el proyecto cada vez que sentía que me rendía. A mi compañera de estudios incondicional. Gracias Laura por abrirme las puertas de tu casa para conseguir animarme a seguir con el proyecto, y por esos descansos de risas que sabían a gloria. A todos mis amigos, de los que algunos incluso me ayudaron con las pruebas del proyecto en la distancia. Gracias a Natalia, Alberto, Rebeca, Clara, Paco, Laura Noe y Laura Roldán. Se os quiere chicos. Y por último, agradecer a mi tutora Lupe, por su disposición cuando le he pedido ayuda y por su gran amabilidad, animándome siempre a alcanzar una mejor versión de este proyecto.

Índice general

Índice de figuras viii

Índice de tablas xii

I Prolegómeno 1

1. Introducción 3
1.1. Motivación . . . 3
1.2. Alcance y objetivos . . . 3
1.3. Glosario de términos . . . 4
1.4. Estandarización del documento . . . 4
1.5. Organización del documento . . . 5

2. Contexto 7
2.1. Contexto . . . 7
2.2. Tecnologías . . . 7
2.2.1. Anypoint Studio . . . 7
2.2.2. Esper . . . 7
2.2.3. Java . . . 8
2.2.4. RabbitMQ . . . 8
2.2.5. MariaDB Server . . . 8
2.2.6. HeidiSQL . . . 8
2.2.7. nITROGEN . . . 9
2.2.8. Android Studio - Virtual Device Manager . . . 9
2.2.9. IntelliJ IDEA . . . 9
2.2.10. Expo Go . . . 9
2.2.11. Visual Studio Code . . . 9
2.2.11.1. Extensión LaTeX Workshop para Visual Studio Code . 9
2.2.12. React Native . . . 10
2.2.13. Node.js . . . 10
2.2.14. Express . . . 10
2.2.15. Figma . . . 10
2.2.16. Draw.io . . . 10
2.2.17. Excalidraw . . . 11
2.2.18. Crossenv . . . 11
2.2.19. Jest . . . 11
2.2.20. SupesTest . . . 11
2.2.21. Scrcpy . . . 11

3. Plan de gestión del proyecto 13
3.1. Metodología de desarrollo . . . 13

3.2. Planificación del proyecto - fases del modelo de gestión incremental . . 13
3.2.1. Búsqueda de datos y consolidación de ideas 13
3.2.2. Diseño de esquemas y flujos Mule para recoger y transformar datos 14
3.2.3. Estudio inicial de React Native y Node.js 14
3.2.4. Desarrollo de aplicación móvil (front-end y back-end) 14
3.2.5. Diseño patrones Esper EPL 14
3.2.6. Pruebas 15
3.2.7. Mejoras 15
3.3. Diagrama de Gantt 15
3.4. Riesgos 16

II Desarrollo 17

4. Requisitos del sistema 19
4.1. Requisitos funcionales 19
4.2. Requisitos no funcionales 20
4.3. Requisitos de información 20

5. Análisis del sistema 23
5.1. Modelo de casos de uso 23
5.1.1. Caso de uso “Iniciar sesión” 23
5.1.2. Caso de uso “Registrarse” 25
5.1.3. Caso de uso “Ver perfil” 26
5.1.4. Caso de uso “Editar perfil” 26
5.1.5. Caso de uso “Cerrar sesión” 28
5.1.6. Caso de uso “Seleccionar una categoría y explorar su lista de eventos” 28
5.1.7. Caso de uso “Marcar/desmarcar “me gusta” de un evento” . . . 29
5.1.8. Caso de uso “Explorar lista de “me gusta” ” 30
5.1.9. Caso de uso “Desmarcar “me gusta” de la lista” 31
5.1.10. Caso de uso “Ver a quién más le gusta un evento” 32
5.1.11. Caso de uso “Crear chat individual” 32
5.1.12. Caso de uso “Crear chat grupal” 33
5.1.13. Caso de uso “Eliminar un chat individual/grupal” 35
5.1.14. Caso de uso “Acceder a un chat individual/grupal” 35
5.1.15. Caso de uso “Enviar mensaje a un chat individual/grupal” . . . 36
5.1.16. Caso de uso “Ver perfil de otro usuario” 37
5.1.17. Caso de uso “Ver grupo” 38
5.1.18. Caso de uso “Modificar grupo” 39
5.1.19. Caso de uso “Consultar en el mapa próximos eventos y aforo de parking” 41
5.2. Modelo de interfaz de usuario 42
5.3. Modelo de comportamiento 42

6. Diseño del sistema 45
6.1. Arquitectura del sistema 45
6.2. Esquema de la base de datos 48

6.3. Diseño de la interfaz de usuario de la aplicación móvil 49

7. Implementación **53**
7.1. Aplicación Mule 53
7.1.1. Definición de esquemas 53
7.1.2. Definición de patrones 54
7.1.2.1. Implementación de la clase Java Haversine 65
7.1.3. Despliegue de esquemas y patrones 66
7.1.4. Obtención de datos y envío de eventos simples 67
7.1.4.1. Eventos de Madrid - obtención y estructura de los datos 67
7.1.4.2. Información de parking - obtención y estructura de los datos 77
7.1.5. Envío de eventos simples al motor CEP 83
7.1.6. Procesamiento de eventos complejos y notificaciones 84
7.1.6.1. Clase Java "SendEmail" 84
7.1.6.2. Clase Java "SendExpoNotification" 87
7.1.7. Recuperación de eventos simples frente a apagados o reinicios . 89
7.2. Aplicación Android 92
7.2.1. Distribución de ficheros 92
7.2.2. Front-end 92
7.2.2.1. Navegación 92
7.2.2.2. Autenticación: contextos, validaciones y notificaciones push 96
7.2.2.3. Explorador de eventos 99
7.2.2.4. Lista de "me gusta" 103
7.2.2.5. Lista de usuarios a los que también les gusta un evento 104
7.2.2.6. Mapa: implementación con expo-location y react-native-maps 105
7.2.2.7. Chats: navegación desde la lista de chats, creación y edición de Grupos 108
7.2.2.8. Chats: agregar al chat a nuevos usuarios mediante grupos compartidos 119
7.2.2.9. Chats: intercambio de mensajes mediante uso de web sockets 120
7.2.3. Back-end 122
7.2.3.1. Autenticación: bcrypt y JSON Web Token 123
7.2.3.2. Obtención de usuarios con un "me gusta" en común . 127
7.2.3.3. Chats: comprobación de la existencia de un chat individual entre dos usuarios 128
7.2.3.4. Chats: creación, modificación y eliminación de Chats . 130
7.2.3.5. Chats: intercambio de mensajes mediante uso de web sockets 131
7.2.3.6. Chats: notificaciones de mensajes enviados 134
7.2.3.7. Subida de imágenes de perfil y de grupo usando express-fileupload 134
7.2.3.8. Modelo de eventos en el back-end 136

8. Pruebas del sistema **139**

8.1. Pruebas unitarias . . . 139
8.1.1. Patrones de eventos con Esper EPL Online . . . 139
8.1.1.1. Aclaración sobre horario local de España y horario GMT/UTC usado por la herramienta . . . 139
8.1.1.2. Probando los patrones de eventos . . . 139
8.2. Pruebas de integración . . . 150
8.2.1. Pruebas de integración de patrones Esper EPL, Mule ESB y RabbitMQ usando nITROGEN . . . 150
8.2.2. Pruebas de integración del back-end con la base de datos usando Jest Y SuperTest . . . 173
8.2.2.1. authSesions.test.js . . . 174
8.2.2.2. usersInteresesEventos.test.js . . . 177
8.2.2.3. chats.test.js . . . 177
8.2.2.4. parking.test.js . . . 182
8.3. Pruebas de aceptación . . . 182
8.3.1. Usabilidad . . . 185
8.3.1.1. Resultados de la encuesta . . . 187
8.3.2. Utilidad . . . 190
8.3.2.1. Resultados de la encuesta . . . 190
8.4. Pruebas de rendimiento de la aplicación Mule . . . 193
8.4.1. Análisis de consumo de CPU y memoria . . . 193
8.4.1.1. Primer estudio . . . 193
8.4.1.2. Segundo estudio . . . 195
8.4.1.3. Visualización de resultados del programa Oshi.jar . . . 197
8.4.2. Análisis de tiempo de respuesta . . . 197
8.5. Verificación de la calidad . . . 199

III Epílogo **201**

9. Conclusiones **203**
9.1. Objetivos alcanzados y valoración de la funcionalidad del proyecto . . . 203
9.2. Lecciones aprendidas y valoración personal del proyecto . . . 204
9.3. Trabajo futuro . . . 204

Bibliografía **209**

IV Anexos **212**

A. Manual de desarrollador **214**
A.1. Introducción . . . 214
A.2. nITROGEN . . . 214
A.3. Anypoint Studio . . . 214
A.3.1. Importar aplicación Mule en Anypoint Studio . . . 215
A.4. RabbitMQ . . . 215
A.4.1. Erlang . . . 215
A.4.2. Servidor RabbitMQ . . . 215

A.4.3. Administrador de RabbitMQ 216
A.4.4. Panel de control gráfico de RabbitMQ 216
A.5. Base de datos 216
A.6. Expo Go 219
A.7. Fichero .env del directorio `MadridTFG\client` 219

B. Manual de usuario **221**
B.1. Inicio de sesión y registro 221
B.2. Explorador de eventos 222
B.3. Mapa 224
B.4. Me gusta 228
B.5. Chats 233
B.6. Perfil 243

Índice de figuras

3.1. Diagrama de Gantt 15

5.1. Diagrama de casos de uso 23
5.2. Diagrama de secuencia 43

6.1. Arquitectura lógica del sistema 45
6.2. Solicitudes a API externa y almacenamiento persistente de los datos recogidos 46
6.3. Generación de datos y eventos simples mediante el uso de la aplicación móvil 47
6.4. Envío de eventos simples desde la aplicación Mule 47
6.5. Generación de eventos complejos y notificaciones 48
6.6. Modelo conceptual de clases 49
6.7. Mockup 52

7.1. Distribución de ficheros para el front-end 93
7.2. Distribución de ficheros para el back-end 94
7.3. Botón cerrar sesión de la pantalla de perfil de usuario 99

8.1. Conversión de fecha horario local español al valor EPOCH de su equivalente fecha GMT/UTC usando el conversor https://espanol.epochconverter.com/ 140
8.2. Conversión de fecha GMT/UTC a EPOCH usando el conversor https://espanol.epochconverter.com/ 140
8.3. Prueba unitaria patrón "DetectaInicioSesionFallido" usando Esper EPL Online 140
8.4. Prueba unitaria patrón "DetectaFuerzaBruta" usando Esper EPL Online 141
8.5. Prueba unitaria patrón "UbicacionSospechosa" usando Esper EPL Online 141
8.6. Prueba unitaria patrón "MenosDe5Dias" usando Esper EPL Online . . 142
8.7. Prueba unitaria patrón "MenosDe24h" usando Esper EPL Online . . . 142
8.8. Prueba unitaria patrón "MenosDe2h" usando Esper EPL Online 143
8.9. Prueba unitaria patrón "MeGustaMenosDe5Dias" usando Esper EPL Online 144
8.10. Prueba unitaria patrón "MeGustaMenosDe24h" usando Esper EPL Online 144
8.11. Prueba unitaria patrón "MeGustaMenosDe2h" usando Esper EPL Online 145
8.12. Prueba unitaria patrón "AforoLimitado" usando Esper EPL Online . . 145
8.13. Prueba unitaria patrón "AforoLimitado_MeGustaMenosDe24h" usando Esper EPL Online 146
8.14. Prueba unitaria patrón "AforoLimitado_MeGustaMenosDe2h" usando Esper EPL Online 146
8.15. Prueba unitaria patrón "ParkingCompleto" usando Esper EPL Online . 147
8.16. Prueba unitaria patrón "Lunes" usando Esper EPL Online 148
8.17. Prueba unitaria patrón "EventoEnCurso" usando Esper EPL Online . . 148

8.18. Prueba unitaria patrón “MeGustaEventoEnCurso” usando Esper EPL Online . . . 149
8.19. Prueba unitaria patrón “HoyNoSeCelebra” usando Esper EPL Online . 149
8.20. Prueba de integración patrón “InicioSesionFallido” desde Mule Anypoint 150
8.21. Aviso sobre el evento complejo “InicioSesionFallido” desde Gmail . . . 151
8.22. Configuración global de nITROGEN para la prueba de integración del patrón “DetectaFuerzaBruta” . . . 151
8.23. Prueba de integración patrón “DetectaFuerzaBruta” desde Mule Anypoint 152
8.24. Aviso sobre el evento complejo “DetectaFuerzaBruta” desde Gmail . . 152
8.25. Aviso sobre el evento complejo “InicioSesionFallido” también cuando se detecta “DetectaFuerzaBruta” desde Gmail . . . 153
8.26. Envío de evento simple “Ubicación” desde la consola de Virtual Studio Code . . . 153
8.27. Generación de evento simple “Ubicación” con otro valor en el campo “country” desde nITROGEN . . . 153
8.28. Prueba de integración patrón “UbicacionSospechosa” desde Mule Anypoint . . . 154
8.29. Aviso sobre el evento complejo “UbicacionSospechosa” desde Gmail . . 154
8.30. Prueba de integración patrón “MenosDe5Dias” desde Mule Anypoint . 155
8.31. Prueba de integración patrón “MeGustaMenosDe5Dias” desde Mule Anypoint . . . 155
8.32. Prueba de integración de notificación push del patrón “MeGustaMenosDe5Dias” desde dispositivo móvil . . . 156
8.33. Prueba de integración patrón “AforoLimitado” desde Mule Anypoint . 156
8.34. Prueba de integración patrón “MenosDe24h” desde Mule Anypoint . . 157
8.35. Prueba de integración patrón “MeGustaMenosDe24h” desde Mule Anypoint . . . 157
8.36. Prueba de integración de notificación push del patrón “MeGustaMenosDe24h” desde dispositivo móvil . . . 159
8.37. Prueba de integración patrón “MenosDe2h” desde Mule Anypoint . . . 160
8.38. Prueba de integración patrón “MeGustaMenosDe2h” desde Mule Anypoint 160
8.39. Prueba de integración patrón “AforoLimitado_MeGustaMenosde2h” desde Mule Anypoint . . . 161
8.40. Prueba de integración de notificación push del patrón “AforoLimitado_MeGustaMenosde2h” desde dispositivo móvil . . . 162
8.41. Evento simple de tipo “Parking” generado desde nITROGEN con localización correspondiente al Parking Saba Plaza Mostenses(Madrid) . . . 163
8.42. Prueba de integración patrón “ParkingPlazasLibresCerca_MeGustaMenosDe2h” desde Mule Anypoint . . . 163
8.43. Prueba de integración de notificación push del patrón “ParkingPlazasLibresCerca_MeGustaMenosDe2h” desde dispositivo móvil . . . 164
8.44. Prueba de integración patrón “ParkingCompleto” desde Mule Anypoint 165
8.45. Prueba de integración de notificación push del patrón “ParkingCompleto” desde dispositivo móvil . . . 166
8.46. Evento simple de tipo “EventosMadrid” generado desde nITROGEN con localización en la localidad de San Fernando(Cádiz) . . . 167
8.47. Prueba de integración patrón “MenosDe2hCerca” desde Mule Anypoint 167

8.48. Prueba de integración de notificación push del patrón “MenosDe2hCerca” desde dispositivo móvil 168
8.49. Prueba de integración del patrón “Martes” desde Mule Anypoint 169
8.50. Evento en curso en la base de datos vista desde la aplicación Android . 170
8.51. Prueba de integración del patrón “EventoEnCurso” desde Mule Anypoint 170
8.52. Me gusta a evento en curso en la base de datos desde la aplicación Android 171
8.53. Prueba de integración del patrón “MeGustaEventoEnCurso” desde Mule Anypoint 171
8.54. Inserción en la base de datos de evento en curso simulado 172
8.55. Prueba de integración del patrón “HoyNoSeCelebra” desde Mule Anypoint 172
8.56. Notificación del patrón “HoyNoSeCelebra” desde la aplicación Android 173
8.57. Tabla de distribución normal proporcionada por la Universidad de Murcia 185
8.58. Interpretación Puntuación SUS 187
8.59. Pregunta número 1 del cuestionario - pruebas de aceptación 187
8.60. Pregunta número 2 del cuestionario - pruebas de aceptación 187
8.61. Pregunta número 3 del cuestionario - pruebas de aceptación 188
8.62. Pregunta número 4 del cuestionario - pruebas de aceptación 188
8.63. Pregunta número 5 del cuestionario - pruebas de aceptación 188
8.64. Pregunta número 6 del cuestionario - pruebas de aceptación 189
8.65. Pregunta número 7 del cuestionario - pruebas de aceptación 189
8.66. Pregunta número 8 del cuestionario - pruebas de aceptación 189
8.67. Pregunta número 9 del cuestionario - pruebas de aceptación 190
8.68. Pregunta número 10 del cuestionario - pruebas de aceptación 190
8.69. Pregunta número 11 del cuestionario - pruebas de aceptación 191
8.70. Pregunta número 12 del cuestionario - pruebas de aceptación 191
8.71. Pregunta número 13 del cuestionario - pruebas de aceptación 191
8.72. Pregunta número 14 del cuestionario - pruebas de aceptación 192
8.73. Pregunta número 15 del cuestionario - pruebas de aceptación 192
8.74. Pregunta número 16 del cuestionario - pruebas de aceptación 192
8.75. Pregunta número 17 del cuestionario - pruebas de aceptación 193
8.76. Pregunta número 18 del cuestionario - pruebas de aceptación 193
8.77. Porcentaje de consumo de CPU de la aplicación Mule 194
8.78. Consumo de memoria en MiB de la aplicación Mule 195
8.79. Porcentaje de consumo de CPU de la aplicación Mule durante 10 minutos 196
8.80. Consumo de memoria en MiB de la aplicación Mule durante 10 minutos 196
8.81. Tiempo de respuesta con eventos dummy desde Mule Anypoint 199

B.1. Pantalla de inicio de sesión 221
B.2. Pantalla de registro 222
B.3. Pantalla del explorador de eventos 223
B.4. Pantalla de lista de eventos de la categoría seleccionada 224
B.5. Pantalla del mapa 225
B.6. Pantalla del mapa - información de uso 226
B.7. Pantalla del mapa - información de plazas libres 227
B.8. Pantalla del mapa - información de evento seleccionado 228
B.9. Pantalla de me gusta 229
B.10. Pantalla de me gusta en caso de no haber dejado ningún “me gusta” . . 230

B.11.Lista de “me gusta” en común vacía 231
B.12.Lista de usuarios con un “me gusta” en común 232
B.13.Consulta de perfil de otro usuario 233
B.14.Lista de chats vacía 234
B.15.Lista de chats 235
B.16.Intercambio de mensajes en un chat 236
B.17.Creación de un chat grupal 237
B.18.Alerta al intentar crear un chat grupal sin añadir usuarios 238
B.19.Intercambio de mensajes en un chat grupal 239
B.20.Pantalla de consulta de grupo 240
B.21.Pantalla de edición de grupo 241
B.22.Alerta al intentar dejar el grupo con un solo miembro 242
B.23.Mensaje al intentar eliminar chat 243
B.24.Pantalla de perfil de usuario 244
B.25.Pantalla de perfil de usuario sin “me gusta” 244
B.26.Pantalla de edición de perfil de usuario 245

Índice de tablas

3.1. Riesgos 16
3.2. Posibles soluciones ante riesgos 16

8.1. Análisis de consumo de recursos 194

Parte I

Prolegómeno

1. Introducción

A continuación, se describe la motivación del presente proyecto y su alcance. También se incluye un glosario de términos y la organización del resto de la presente documentación.

1.1. Motivación

La ciudad de Madrid se ha convertido en un referente para quienes buscan nuevas oportunidades laborales. En esta búsqueda, son muchas las personas que optan por dejar atrás su hogar y alejarse de sus seres queridos. Adaptarse a nuevos lugares o personas y no encontrarse sólo puede ser un reto difícil.

No solamente la ciudad se distingue por las oportunidades que ofrece: Madrid se envuelve de uno de los entornos culturales y de ocio más ricos del país.

Este TFG tiene como propósito desarrollar un producto software que contribuya a hacer de esos nuevos comienzos una experiencia de adaptación más sencilla, y en general, que ayude tanto a nuevos como a viejos residentes a explorar y disfrutar de ese ocio que la ciudad les ofrece permitiéndoles conocer personas con gustos afines.

1.2. Alcance y objetivos

El objetivo de este TFG es desarrollar una aplicación móvil para dispositivos Android que sirva de guía de eventos de la ciudad de Madrid, y que a su vez ofrezca una interacción entre los usuarios en base a sus gustos en cuanto a tipos de eventos se refiere a través de un sistema de mensajería.

La aplicación será capaz de notificar a los usuarios de diversas situaciones de interés gracias a la tecnología CEP, que procesará datos proporcionados por el Ayuntamiento de Madrid a través de su web municipal y datos provenientes de la interacción de los usuarios con la aplicación Android.

Podemos desglosar el objetivo principal en varios subobjetivos:

1. Desarrollar el front-end de la aplicación Android en React Native.
2. Desarrollar el back-end de la aplicación con Nodejs y Express, haciendo la autenticación más segura gracias al uso de Java Web Token (JWT).
3. Diseño de una base de datos para dar persistencia a los datos que usa la aplicación.
4. Desarrollar una arquitectura software basada en CEP: implementado usando el bus de servicios empresariales (ESB) de Mule a través de la interfaz gráfica de AnyPoint y el motor CEP de Esper, así como el bróker de mensajería RabbitMQ.

5. Definir los patrones y esquemas de eventos que el motor CEP será capaz de detectar.
6. Dotar a la aplicación de un sistema de notificaciones gracias a Expo.
7. Desarrollo de sistema de mensajería gracias a la tecnología de Socket.io.

1.3. Glosario de términos

BD Base de datos

TFG Trabajo de Fin de Grado

CEP Complex Event Processing

EPL Event Processing Language

ESB Enterprise Service Bus

AMQP Advanced Message Queuing Protocol

HTTP HyperText Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

CRUD Create, Read, Update and Delete

API Application Programming Interface

IoT Internet of Things

IDE Entorno de desarrollo integrado

AVD Dispositivo Virtual de Android

MVC Modelo-Vista-Controlador

GMT/UTC Greenwich Mean Time/Tiempo Universal Coordinado

1.4. Estandarización del documento

Para el desarrollo de la documentación se ha seguido el estándar ISO/IEC/IEEE 16326 sobre la gestión de proyectos de ingeniería software y de sistemas, adaptándo la estructura del documento a la seguida comúnmente por un trabajo de fin de grado y desechando aquellos apartados más enfocados a proyectos de ingeniería software de organizaciones.

1.5. Organización del documento

Detallamos el contenido de la memoria. Para comenzar tenemos el índice general, el índice de figuras y el índice de tablas.

En el capítulo Introducción hablaremos sobre la motivación del proyecto de la cual surgió la idea del mismo, además de los objetivos planteados para desarrollarlo.

En el capítulo Contexto se detallan las alternativas existentes dentro del entorno tecnológico y se detallan las diferentes tecnologías y herramientas utilizadas para el desarrollo del proyecto.

En el capítulo Plan de gestión del proyecto se describen todos los aspectos relativos a la gestión del proyecto: metodología, organización, costes, planificación, riesgos y aseguramiento de la calidad.

En el capítulo Requisitos del sistema se presentan los objetivos y requisitos del proyecto.

El capítulo Análisis del sistema cubre el análisis del sistema de información a desarrollar, haciendo uso del lenguaje de modelado UML para representar las distintas funcionalidades del sistema mediante un diagrama de casos de uso.

En el capítulo Diseño del sistema se recoge el diseño de la arquitectura del sistema informático, el diseño físico de datos y el diseño detallado de la interfaz de usuario.

El capítulo Implementación trata sobre todos los aspectos relacionados con la implementación del sistema en código, haciendo uso de un determinado entorno tecnológico.

En el capítulo Pruebas del sistema se detallan diferentes tipos de pruebas realizadas, como unitarias, de integración o de aceptación.

En el capítulo Conclusiones se detallan las lecciones aprendidas tras el desarrollo del presente proyecto y se identifican las posibles oportunidades de mejora sobre el software desarrollado.

Para terminar, tenemos el apartado donde detallamos toda la bibliografía, y el Anexo, donde encontramos un apartado dedicado al Manual de desarrollador y otro al Manual de usuario.

2. Contexto

En este capítulo se describen las alternativas existentes dentro del entorno tecnológico y se detallan las diferentes tecnologías y herramientas utilizadas para el desarrollo del proyecto.

2.1. Contexto

Ya vimos anteriormente en el apartado dedicado a ello la Motivación de este proyecto, ahora centrémonos en el contexto en que se va a implementar el proyecto. Dentro del entorno tecnológico existe la web del Ayuntamiento de Madrid que ofrece información sobre los eventos que se celebran en la ciudad, y cuya funcionalidad sirvió de inspiración para consolidar las principales ideas sobre a qué problemas se quería dar solución con este proyecto. A diferencia de la web, para este proyecto se pretende implementar funcionalidades que permitan a los usuarios compartir intereses y comunicarse entre sí. Además permite conocer información relevante en tiempo real sobre los eventos de Madrid por medio de notificaciones. Se va a implementar una funcionalidad que permite consultar la localización en el mapa de los eventos de Madrid así como de los parking de la ciudad con información sobre sus plazas libres. Esta idea surgió consultando los datos que proporcionaba el Ayuntamiento de Madrid por medio de su API. Existe una aplicación móvil de la Empresa Municipal de Transportes de Madrid (EMT Madrid) que tiene dicha funcionalidad: Parking Madrid. Sin embargo se han leído algunas reseñas de distintos usuarios y parece ser que la aplicación lleva años sin actualizar la información. Para este proyecto, la información se recibe cada pocos minutos, y no suele estar desactualizada mucho más de 10 minutos.

2.2. Tecnologías

2.2.1. Anypoint Studio

"Anypoint Studio es el entorno de desarrollo de integración basado en Eclipse de MuleSoft para diseñar y probar aplicaciones Mule" (MuleSoft, 2024). Para el desarrollo de este proyecto se ha usado con la finalidad de obtener recurrentemente y almacenar de manera persistente en la Base de Datos los datos de los endpoints correspondientes, y de formatearlos para su posterior procesamiento por parte del motor CEP.

2.2.2. Esper

Esper es un motor de procesamiento de eventos complejos CEP que detecta situaciones de interés detectando ciertas condiciones sobre los datos que procesa, definidas

por medio de patrones en lenguaje EPL(Esper Processing Language), que implementa y extiende el estándar SQL y permite expresiones enriquecidas sobre eventos y tiempo (EsperTech, 2024).

2.2.3. Java

Java es uno de los lenguajes de programación más usados (Corporation, 2024), y en el caso de este proyecto ha sido utilizado para escribir clases Java incluidas en los flujos de trabajo de la Aplicación Mule, para el formateo de los datos entrantes obtenidos de las distintas APIs y que serán procesados por el motor CEP. También se han implementado clases que se ocupan del almacenamiento de esos datos, tanto en colas del broker de mensajería RabbitMQ (del que damos más detalles en la sección 2.2.4) como en la base de datos.

2.2.4. RabbitMQ

"RabbitMQ es un broker de mensajería y transmisión confiable y maduro, que es fácil de implementar en entornos de nube, locales y en su máquina local"(RabbitMQ, 2024). Para el desarrollo del proyecto se usa como receptor y emisor tanto de eventos simples como complejos para su procesamiento.

2.2.5. MariaDB Server

MariaDB Server es un sistema de gestión de bases de datos relacionales de código abierto. Es uno de los servidores de bases de datos más populares del mundo. MariaDB Server sigue manteniendo altos niveles de compatibilidad con MySQL, y la mayoría de las aplicaciones populares que utilizan MySQL funcionarán sin problemas con MariaDB (MariaDB Corporation, 2024b).

Debido a la naturaleza del proyecto y las existentes dependencias y relaciones entre entidades, era lo más coherente escoger un sistema de gestión de bases de datos de tipo relacional.

2.2.6. HeidiSQL

HeidiSQL es un software gratuito de código abierto para trabajar de manera intuitiva con bases de datos y permite conectarse a una variedad de bases de datos, como MariaDB, MySQL, Microsoft SQL, PostgreSQL, SQLite, Interbase y Firebird (Becker, 2024).

Su interfaz gráfica facilita el trabajo de desarrollo, por ejemplo a la hora de crear o modificar pequeños detalles de las tablas de la base de datos sin necesidad de tener que escribir las consultas enteras.

2.2.7. nITROGEN

Un generador de datos para arquitecturas de procesamiento de datos de IoT (Prado, 2024). En este proyecto se utiliza para generar datos en formato JSON y realizar pruebas con ellos, desplegando los patrones mencionados en la sección 2.2.2 y comprobando su efectividad sobre distintos datos simulados.

2.2.8. Android Studio - Virtual Device Manager

Android Studio es un entorno de desarrollo para el desarrollo de aplicaciones Android (Google, 2024a). Para facilitar algunas pruebas y comprobaciones de manera más directa para el proyecto, se ha usado una herramienta particular que ofrece este entorno de desarrollo llamada Virtual Device Manager, que permite crear una configuración para lo que se conoce como AVD (Dispositivo Virtual de Android), que nos permite emular un dispositivo Android (Google, 2024b).

2.2.9. IntelliJ IDEA

IntelliJ IDEA es un entorno de desarrollo usado para el desarrollo de código en Java (jetbrains, 2025). En ese proyecto se ha utilizado durante la realización de las Pruebas de rendimiento de la aplicación Mule, para acceder al código fuente del programa empleado para el análisis del consumo de recursos.

2.2.10. Expo Go

Expo Go es un entorno sandbox que permite realizar cambios en el código fuente de la aplicación y visualizar los cambios directamente en un emulador o en un dispositivo móvil (Expo, 2024). Debido a ello y a que para el proyecto se escogió React Native como tecnología front-end, se descartó trabajar con Android Studio para edición de código, ya que soporta lenguajes como Java o Kotlin pero no React Native, y usar solo su herramienta Virtual Device Manager mencionada en la sección 2.2.8.

2.2.11. Visual Studio Code

Gracias a la compatibilidad con varios lenguajes de programación, Visual Studio Code ha sido el entorno de desarrollo que se ha utilizado para desarrollar el código fuente del front-end en React Native, y el back-end con Nodejs y Express (Microsoft, 2024).

2.2.11.1. Extensión LaTeX Workshop para Visual Studio Code

Esta extensión permite compilar, crear y editar documentos en LaTex en el propio entorno de desarrollo. La decisión de utilizarla para el desarrollo de la documentación se

dió en una fase avanzada del desarrollo del proyecto, habiéndo hecho uso en un principio de Overleaf para editar el documento y sustituyéndolo por la extensión mencionada. El documento se extendió lo suficiente como para superar el plan gratuito que Overleaf ofrecía.

2.2.12. React Native

React Native es un framework basado en el lenguaje JavaScript usado para el desarrollo de Aplicaciones para IOS y Android (concretamente para dispositivos Android en este proyecto). Su popularidad en el mercado y su potencial han sido de los principales motivos para su utilización y aplicación en el proyecto (Meta Platforms, 2024c).

2.2.13. Node.js

Node.js es un entorno de ejecución de JavaScript que permite a los desarrolladores crear servidores, aplicaciones web, herramientas de línea de comandos y scripts. En este proyecto se usa para crear y arrancar el servidor que actúa de back-end desarrollado en JavaScript. Este servidor se encarga de gestionar las solicitudes HTTP y conexiones con la base de datos, así como de servir de intermediario entre el front-end y los servicios de procesamiento de eventos en tiempo real: Algunas interacciones del usuario con la Aplicación se procesan en este servidor antes de ser recibidas y procesadas por la Aplicación Mule (Foundation, 2024).

2.2.14. Express

Express es un framework de Node.js que permite definir rutas para manejar las solicitudes de los usuarios en el el servidor back-end de este proyecto. También simplifica el proceso de inicialización y ejecución del servidor en Node.js (Team, 2024d).

2.2.15. Figma

Figma es una herramienta de diseño online cuyo propósito para este proyecto ha sido servir para poder desarrollar el mockup de la aplicación Android (obsérvese el resultado en la figura 6.7 del apartado Diseño de la interfaz de usuario de la aplicación móvil).

2.2.16. Draw.io

Draw.io es una herramienta UML online que ha permitido para este proyecto desarrollar el diagrama de casos de uso (obsérvese el resultado en la figura 5.1 del apartado Modelo de casos de uso).

2.2.17. Excalidraw

Excalidraw es una herramienta online que ha permitido desarrollar para este proyecto el diagrama que representa la arquitectura lógica del sistema (obsérvese el resultado en la figura 6.1 del apartado Arquitectura del sistema).

2.2.18. Crossenv

Especificar el valor de la variable de entorno NODE_ENV permite adaptar el comportamiento del código, adecuandolo a la fase del ciclo de vida en que se encuentre(desarrollo, producción o pruebas). En Sistemas Operativos Windows la definición de variables de entorno pueden dar lugar a error y no ser persistentes cuando se definen dentro de los comandos o scripts en el package.json. Una solución es instalar la dependencia cross-env (Dodds, 2024).

2.2.19. Jest

Este framework de JavaScript es usado para realizar pruebas unitarias o de integración, usándose para el segundo caso junto con SupesTest en este proyecto para realizar validaciones sobre diferentes endpoints de la aplicación mediante la creación de tests (Jest, 2025b) (ver apartado Pruebas de integración del back-end con la base de datos usando Jest Y SuperTest).

2.2.20. SupesTest

Biblioteca que permite probar los diferentes endpoints del backend desarrollado con Express (ladjs, 2025) (ver apartado Pruebas de integración del back-end con la base de datos usando Jest Y SuperTest).

2.2.21. Scrcpy

Herramienta que permite compartir la pantalla de un dispositivo móvil Android conectado mediante USB (Genymobile, 2025).

3. Plan de gestión del proyecto

En esta sección se describen todos los aspectos relativos a la gestión del proyecto: metodología, organización, planificación, riesgos.

3.1. Metodología de desarrollo

La metodología usada para el desarrollo del proyecto ha sido el modelo de gestión incremental, que consiste en que cada fase evolutiva del proyecto suponga la implementación de una o varias nuevas funcionalidades hasta llegar al estado esperado del producto. Las fases que propone este método están recogidas en la sección 3.2.

3.2. Planificación del proyecto - fases del modelo de gestión incremental

En este apartado se detallan cada una de las fases de la Metodología de desarrollo llevada a cabo. En el apartado siguiente podemos observar en el Diagrama de Gantt cómo se distribuyen en el tiempo todas estas fases.

3.2.1. Búsqueda de datos y consolidación de ideas

Para decidir el tema del proyecto, el principal desafío era pensar un caso de estudio que puediera resolver un problema interesante en base a datos disponibles proporcionados por alguna API, y por supuesto, encontrar una fuente apropiada de datos. Se tenía claro desde un principio que este proyecto estaría basado en el procesamiento de eventos complejos (CEP), desarrollando una aplicación móvil que fuera capaz de, habiéndose detectado situaciones de interés para los usuarios, les mandara notificaciones sobre las mismas. Entre los posibles temas pensados se encontraban el procesamiento de datos de transporte público (trenes de Renfe o autobuses), eventos celebrados en la localidad de San Fernando(Cádiz) o en la propia capital de la provincia, o incluso datos sobre las depuradoras de esta última.

Finalmente, se decantó por la segunda idea pero trasladándola a la ciudad de Madrid por falta de datos en las localidades pensadas en un inicio. El proyecto se encargaría entonces de procesar datos referentes a eventos celebrados en la ciudad de Madrid, además de datos sobre plazas libres en varios de sus parking. Además la aplicación sería capaz de permitir a los usuarios interactuar entre ellos mediante un chat.

3.2.2. Diseño de esquemas y flujos Mule para recoger y transformar datos

Una vez decidido el tema del proyecto planteado, se llevó a cabo la Definición de esquemas. Tras ello, se estudió la estructura del JSON proporcionado por cada uno de los dos endpoints (uno para los datos de eventos celebrados en Madrid y otro para las plazas de parking). Se implementaron en Mule flujos para poder transformar esos datos, recogiendo la información que interesaba para la realización del proyecto y dándole el formato adecuado. Para ello se implementaron clases java que procesaran los datos entrantes. En esta etapa también se creó la base de datos y sus respectivas tablas, donde se guardarían los datos pertinentes.

3.2.3. Estudio inicial de React Native y Node.js

Se decidieron las tecnologías con las que se desarrollaría el proyecto. Se decidió desarrollar el front-end de la aplicación con React Native con el fin de aprender sobre esta tecnología debido a su uso dentro del mercado laboral. Por otro lado ocurría lo mismo con node.js, que se utilizaría en este caso para desarrollar el back-end. Se tenía algo de experiencia usando node.js dado que se impartió en la asignatura "Programación en Internet", sin embargo era necesario repasar parte de su semántica y sintaxis. No ocurriría lo mismo con React Native, pues se tuvieron que estudiar sus conceptos base desde cero.

3.2.4. Desarrollo de aplicación móvil (front-end y back-end)

Conforme se iban estudiando conceptos de las tecnologías utilizadas, se iba poniendo en práctica lo aprendido aplicándolo en el desarrollo del proyecto. Esta fase es la que más tiempo llevó con diferencia, debido a la falta de experiencia con React Native sobre todo. Aparte, se pretendía aparte de desarrollar una aplicación funcional, darle un aspecto actual. El desarrollo de la funcionalidad de chat es una de las partes dentro de esta fase que más tiempo llevó, tanto en su código dentro del back-end como en el front-end.

3.2.5. Diseño patrones Esper EPL

Se empezaron a plasmar las ideas en forma de patrones que detectarían situaciones de interés para los usuarios de la aplicación. La Definición de patrones se alarga hasta casi el final del desarrollo del proyecto, pues se van corrigiendo pequeños errores (no por ello menos importantes) que se van detectando mientras se lleva a cabo de forma paralela la fase de Pruebas. Aprender a manejar y comparar fechas en formato EPOCH fue alguno de los retos durante esta fase de desarrollo.

3.2.6. Pruebas

La pruebas realizadas durante esta fase no deben confundirse con las pruebas realizadas durante todo el desarrollo, pues tras cada nueva implementación y, siguiendo el modelo de gestión incremental, se realizan pruebas para verificar su correcto funcionamiento individual y una correcta integración con todo el sistema.

Consideremos esta fase como la fase de pruebas realizada tras haberse desarrollado el sistema satisfaciendo los requisitos funcionales. Permitió la detección de pequeños fallos en patrones, además de plantear posibles Mejoras que llevarían a cabo en la fase dedicada a las mismas. Todo el proceso queda recogido en el apartado dedicado a las Pruebas del sistema de este documento.

3.2.7. Mejoras

Las Pruebas de aceptación dieron lugar a algunas nuevas ideas de funcionalidades para la aplicación además de servir para detectar algún error (como pasó por ejemplo con el chat). Se mejoró la accesibilidad dentro de la aplicación, haciendo las pantallas más intuitivas para el usuario (uso de componente Modal, textos informativos y leyendas en el mapa). Se añadió la funcionalidad para acceder a los perfiles de usuario a través de la lista de usuarios con "me gusta" compartido: antes sólo se podía hacer a través del chat con usuarios que tuvieras agregados al mismo, lo cual no tenía demasiado sentido, puesto que lo lógico sería poder saber sobre el usuario antes de decidir agregarlo al chat. Aparte de estas nuevas mejoras, se implementaron algunos patrones que añadirían más valor al producto final.

3.3. Diagrama de Gantt

En esta sección se proporciona el diagrama de Gantt (ver figura 3.1), en el que se puede observar el tiempo en el que se ha trabajado en las diferentes tareas que, en su conjunto, han permitido la realización del proyecto, entendiendo por tiempo de realización de una tarea el que transcurre desde su fecha de inicio a su fecha de finalización.

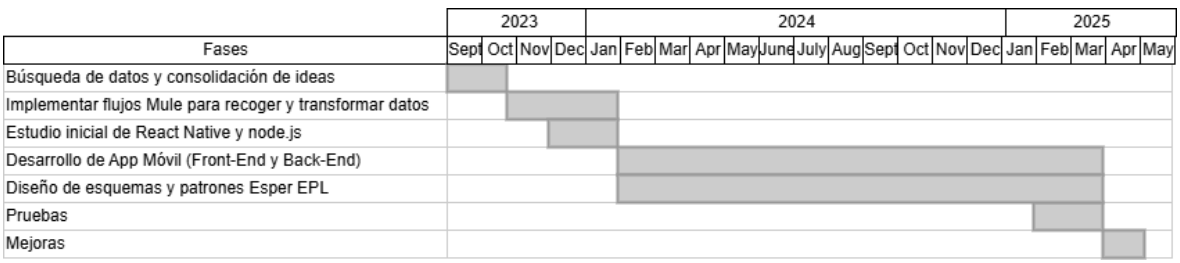


Figura 3.1: Diagrama de Gantt

3.4. Riesgos

En este apartado detallamos los posibles riesgos del proyecto, indicando su grado de impacto (efecto que la ocurrencia del citado riesgo tendría en el desarrollo del proyecto) y la probabilidad de ocurrencia (ver tabla 3.1). A su vez, se plantean posibles soluciones para reducir los efectos del riesgo una vez se haya materializado o disminuir que este ocurra (ver tabla 3.2).

Tabla 3.1

Riesgos

Riesgos			
Riesgo	**Detalles**	**Probabilidad**	**impacto**
R1	Obsolescencia de la versión del ESB de Mule empleada	Alta	Alto
R2	Pérdida o fallas del equipo donde se almacena localmente el proyecto	Media	Alto
R3	Obsolescencia o falta de mantenimiento de una de las API utilizadas	Media	Alto

Tabla 3.2

Posibles soluciones ante riesgos

Posibles soluciones ante riesgos		
Riesgo	**Solución**	**Detalles**
R1	S1	Grabación de vídeos de demostración de apoyo de cara a la defensa ante el tribunal de TFG
R2	S2	Copias de seguridad de los ficheros del proyecto en un repositorio como Git (copia privada)
R3	S3	Simulación de datos con nITROGEN, inserción de datos manualmente en la base de datos
R3	S4	Procesamiento de datos leídos de un fichero en vez de la base de datos

Parte II

Desarrollo

4. Requisitos del sistema

En este capítulo se presentan los objetivos y requisitos del proyecto. Dado que se ha seguido como Metodología de desarrollo un modelo de gestión incremental, al inicio de cada nueva iteración se realizará un análisis de requisitos. Esto supone que tras cada fase evolutiva del proyecto se incluyen nuevos requisitos (o se realizan modificaciones de los anteriores) dentro del análisis original.

4.1. Requisitos funcionales

Descripción completa de la funcionalidad que ofrece el sistema.

- **Generación y obtención de datos de fuentes externas**: El sistema de detección de eventos complejos (Aplicación Mule) debe recuperar de manera periódica tanto los datos ofrecidos por el ayuntamiento de Madrid como los datos generados a partir de la interacción del usuario con la aplicación y su uso, tal como se detalla en la sección 4.3.
- **Formateo de datos para su procesamiento**: El sistema de detección de eventos complejos (Aplicación Mule) debe ser el encargado de dar formato a los datos provenientes de las fuentes detalladas en la sección 4.3, y de enviarlos como eventos simples al motor CEP.
- **Detección de situaciones de interés**: Una vez los datos sean recibidos por el motor CEP, este debe detectar situaciones de interés (eventos complejos) para el usuario. Los patrones, escritos en lenguaje EPL (Esper Processing Language), deberán de detectar de manera correcta cada una de dichas situaciones.
- **Envío de Notificaciones push**: El sistema debe ser capaz de notificar al usuario de las situaciones de interés detectadas enviando una notificación push al dispositivo donde tenga su sesión activa.
- **Sistema de inicio de sesión y autenticación**: La aplicación móvil debe ofrecer un sistema de inicio de sesión permitiendo a los usuarios registrarse y, como su propio nombre indica, iniciar sesión con su correo electrónico y contraseña.
- **Personalización de Perfil**: La aplicación móvil debe permitir editar información del perfil de usuario, tales como datos personales o la imagen de perfil.
- **Sistema de Navegación**: Se debe ofrecer al usuario un sistema de navegación intuitivo por el que navegar de una pantalla a otra de la aplicación.
- **Interacción del usuario con la aplicación y otros usuarios**: El usuario debe poder dar "me gusta" a los eventos y añadir al chat a otros usuarios que también hayan indicado que les gusta el mismo evento, o que pertenezcan a un mismo chat grupal.

- **Chat**: El usuario debe poder intercambiar mensajes con aquellos usuarios a los que haya agregado previamente.
- **Personalización del perfil**: El usuario debe poder modificar datos de su perfil desde la aplicación móvil.
- **Visualización de la ubicación de eventos y parkings en un mapa**: La aplicación móvil debe ofrecer un mapa en el que se visualizan los eventos para el día que seleccione el usuario pulsando uno de los cinco botones que aparecen arriba del mapa, que marcan el día actual y los cuatro días posteriores. También se muestran las distintas ubicaciones de parkings de Madrid, así como información acerca del número de plazas libres.

4.2. Requisitos no funcionales

En este apartado se incluirán las actividades y tareas relacionadas con el aseguramiento de calidad a realizar durante el desarrollo del software.

- **Disponibilidad**: El sistema debe estar preparado para funcionar diariamente con el menor número de interrupciones para ser capaz de poder notificar a tiempo al usuario de las distintas situaciones de interés, y para que el usuario cuente en todo momento con datos actualizados.
- **Fiabilidad**: El sistema debe cumplir con los requisitos del usuario y funcionar de manera que efectúe sus funciones de manera correcta, manejando errores de manera efectiva.
- **Usabilidad**: La interfaz de la aplicación móvil debe ser intuitiva y fácil de usar por parte del usuario.
- **Mantenibilidad**: El código debe seguir una organización modular con el fin de mejorar la legibilidad y hacer del sistema uno más mantenible.
- **Eficiencia**: Tanto el desarrollo de código como el diseño de patrones deben llevarse a cabo teniendo en cuenta detalles que favorezcan el rendimiento del sistema y el aprovechamiento eficiente de recursos sin afectar negativamente a la experiencia del usuario.

4.3. Requisitos de información

El sistema cumple los siguientes requisitos de información:

- El sistema debe recuperar, almacenar en la base de datos y actualizar de forma periódica la información referente a los Eventos que tendrán lugar en Madrid. Los datos deben ser obtenidos en formato JSON a través de la API del ayuntamiento de Madrid mediante este endpoint.

- El sistema debe recuperar periódicamente la información referente a los parkings de Madrid y sus plazas libres ofrecida por el ayuntamiento de Madrid, que es actualizada en tiempo real y está disponible a través del siguiente endpoint.
- El sistema debe almacenar en la base de datos información de la interacción de los usuarios con la aplicación móvil, tales como: sesiones de inicio de sesión (procesándose los intentos de inicio de sesión fallidos), eventos marcados como 'me gusta' y personas añadidas al chat incluyendo los propios chats con sus respectivos mensajes.
- El sistema debe obtener datos de ubicación del usuario que está usando la aplicación móvil, aunque esta información no debería almacenarse mucho más tiempo del que tarda en ser procesada por el sistema, ya que se solo debe almacenarse de manera temporal en una cola del broker de mensajería que se empleará, del que hablamos en la sección 2.2.4.

5. Análisis del sistema

Este capítulo cubre el análisis del sistema de información a desarrollar, haciendo uso del lenguaje de modelado UML.

5.1. Modelo de casos de uso

A partir de los requisitos funcionales descritos anteriormente, se emplearán los casos de uso como mecanismo para representar las interacciones entre los actores y el sistema bajo estudio. La figura 5.1 representa el diagrama de casos de uso del sistema. Para cada caso de uso se indican los actores implicados, las precondiciones y post-condiciones, los pasos que conforman el escenario principal y el conjunto de posibles escenarios alternativos.

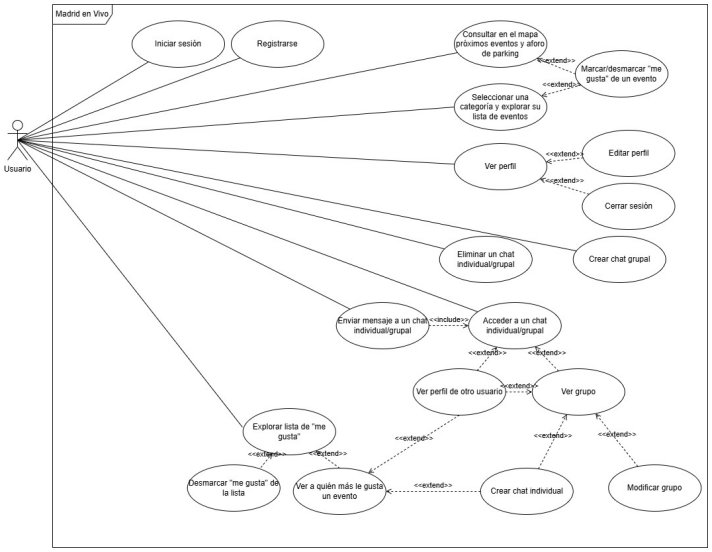


Figura 5.1: Diagrama de casos de uso

5.1.1. Caso de uso “Iniciar sesión”

- **Descripción**
  - El usuario introduce el E-mail y contraseña con los que debe haberse registrado previamente para autenticarse y poder utilizar la aplicación.
- **Actores**
  - Primario: Usuario
- **Escenario Principal**

1. El usuario abre la aplicación.
2. El sistema muestra la pantalla de inicio de sesión, que contiene y muestra los campos para que el usuario ingrese e-mail y contraseña.
3. El usuario modifica el contenido de los campos.
4. El sistema valida el valor introducido por el usuario en cada campo.
5. El usuario presiona el botón "ACCEDER".
6. El sistema redirige al usuario a la pantalla principal de la aplicación.

- **Escenario Alternativo 4.a**
  1. El sistema valida el valor introducido por el usuario en el campo e-mail y detecta un error de formato, mostrando el mensaje "Introduce un E-mail correcto".
  2. Se vuelve al paso 3 del escenario principal.
- **Escenario Alternativo 4.b**
  1. El sistema detecta que el campo e-mail se encuentra vacío y muestra el mensaje "E-mail requerido".
  2. Se vuelve al paso 3 del escenario principal.
- **Escenario Alternativo 4.c**
  1. El sistema detecta que el campo contraseña se encuentra vacío y muestra el mensaje "Contraseña requerida".
  2. Se vuelve al paso 3 del escenario principal.
- **Escenario Alternativo 5.a**:
  1. El sistema detecta que e-mail introducido no ha sido registrado previamente, por lo que se muestra el mensaje "Usuario no encontrado".
  2. Se vuelve al paso 3 del escenario principal.
- **Escenario Alternativo 5.b**:
  1. El sistema detecta que la contraseña introducida no es la correspondiente al e-mail introducido, por lo que se muestra el mensaje "Contraseña Incorrecta".
  2. Se vuelve al paso 3 del escenario principal.
- **Escenario Alternativo 5.c**:
  1. El sistema detecta que ya existe una sesión activa para el usuario cuyos datos han sido introducidos en otro dispositivo, mostrando el mensaje "Error al iniciar sesión. Ya hay una sesión activa para este usuario en otro dispositivo.".
  2. Se vuelve al paso 3 del escenario principal.

- **Postcondiciones**
  - Se establece una sesión de usuario activa en el dispositivo con el que se ha iniciado sesión.

5.1.2. Caso de uso "Registrarse"

- **Descripción**
  - El sistema solicita al usuario que introduzca sus datos para registrarse en la aplicación.
- **Actores**
  - Primario: Usuario
- **Escenario Principal**
  1. El usuario abre la aplicación.
  2. El sistema muestra la pantalla de inicio de sesión.
  3. El usuario pulsa el texto "Regístrate", que sirve de enlace para ser dirigido a la pantalla de registro de usuario.
  4. El sistema muestra la pantalla de registro de usuario, en la que se encuentran los campos para introducir los datos del nuevo usuario como lo son su nombre, apellidos, e-mail y contraseña.
  5. El usuario modifica el contenido de los campos.
  6. El sistema valida el valor introducido por el usuario en cada campo.
  7. El usuario presiona el botón "REGISTRARSE".
  8. El sistema redirige al usuario a la pantalla de inicio de sesión.
- **Escenario Alternativo 6.a**
  1. El sistema valida el valor introducido por el usuario en el campo e-mail y detecta un error de formato, mostrando el mensaje "Introduce un E-mail correcto".
  2. Se vuelve al paso 5 del escenario principal.
- **Escenario Alternativo 6.b**
  1. El sistema detecta qué campo se encuentra vacío y lo muestra por mensaje.
  2. Se vuelve al paso 5 del escenario principal.
- **Escenario Alternativo 7.a**
  1. El sistema detecta que ya existe otro usuario registrado con el e-mail o el nombre de usuario introducidos, y muestra una alerta con el mensaje "El e-mail o el nickname ya está registrado".

2. Se vuelve al paso 5 del escenario principal.

- **Postcondiciones**
  - Se registra un nuevo usuario con los datos introducidos.

5.1.3. Caso de uso "Ver perfil"

- **Descripción**
  - El sistema pone a disposición del usuario sus datos de perfil.
- **Actores**
  - Primario: Usuario
- **Precondiciones**
  - El usuario ha iniciado sesión.
- **Escenario Principal**
  1. El usuario pulsa la opción "Perfil" en la barra de navegación.
  2. El sistema muestra los datos del usuario, como su imagen de perfil, nickname, Descripción, nombre y apellidos. Además también muestra una lista con el título "Eventos Que Me Gustan" de las distintas categorías de evento junto al número de eventos a los que ha dado "me gusta" de cada una, ordenados de mayor a menor. También se muestran un botón para editar la foto de perfil, otro para editar los datos de perfil con el título "Editar Perfil" y otro botón "Cerrar Sesión".
  3.
- **Escenario Alternativo 2.a**
  1. El sistema detecta que el usuario no ha dejado ningún "me gusta" a ningún evento, y muestra en lugar de la lista el mensaje "Da 'me gusta' a los eventos que más te interesen y recibe información relevante sobre ellos".
- **Postcondiciones**
  - El usuario visualiza los datos de su perfil.

5.1.4. Caso de uso "Editar perfil"

- **Descripción**
  - El usuario modifica sus datos de perfil.
- **Actores**
  - Primario: Usuario
- **Precondiciones**

- El usuario ha iniciado sesión.

- **Casos de uso a los que extiende**
  - Caso de uso "Ver perfil"

- **Escenario Principal**
  1. El usuario pulsa la opción "Perfil" en la barra de navegación.
  2. El sistema muestra los datos del usuario, como su imagen de perfil, nickname, Descripción, nombre y apellidos. Además también muestra una lista con el título "Eventos Que Me Gustan" de las distintas categorías de evento junto al número de eventos a los que ha dado "me gusta" de cada una, ordenados de mayor a menor. También se muestran un botón para editar la foto de perfil, otro para editar los datos de perfil con el título "Editar Perfil" y otro botón "Cerrar Sesión".
  3. El usuario pulsa el botón "Editar Perfil".
  4. El sistema muestra campos donde el usuario puede modificar su Descripción de perfil, nombre o apellidos. También muestra un botón "Confirmar" y un botón "Cancelar".
  5. El usuario modifica el contenido de los campos.
  6. El usuario pulsa el botón "Confirmar" o "Cancelar".

- **Escenario Alternativo 3.a**
  1. El usuario pulsa el botón para modificar su foto de perfil.
  2. El sistema abre la galería del dispositivo del usuario.
  3. El usuario selecciona una nueva imagen de perfil.

- **Escenario Alternativo 4.a**
  1. El campo está vacío y se muestra por mensaje que se añada un nuevo valor para el campo.
  2. Se vuelve al paso 4 del escenario principal.

- **Escenario Alternativo 5.a**
  1. El usuario borra el contenido de un campo.
  2. El campo está vacío y se muestra por mensaje que se añada un nuevo valor para el campo.
  3. Se vuelve al paso 4 del escenario principal.

- **Postcondiciones**
  - Si el usuario pulsa el botón "Confirmar", el usuario modifica los datos de su perfil.
  - Si el usuario pulsa el botón "Cancelar", el usuario no modifica los datos de su perfil.

5.1.5. Caso de uso "Cerrar sesión"

- **Descripción**
  - El sistema solicita al usuario que introduzca sus datos para registrarse en la aplicación.
- **Actores**
  - Primario: Usuario
- **Precondiciones**
  - El usuario ha iniciado sesión.
- **Casos de uso a los que extiende**
  - Caso de uso "Ver perfil"
- **Escenario Principal**
  1. El usuario pulsa la opción "Perfil" en la barra de navegación.
  2. El sistema muestra los datos del usuario, como su imagen de perfil, nickname, Descripción, nombre y apellidos. Además también muestra una lista con el título "Eventos Que Me Gustan" de las distintas categorías de evento junto al número de eventos a los que ha dado "me gusta" de cada una, ordenados de mayor a menor. También se muestran un botón para editar la foto de perfil, otro para editar los datos de perfil con el título "Editar Perfil" y otro botón "Cerrar Sesión".
  3. El usuario pulsa el botón "Cerrar Sesión".
  4. El sistema cierra la sesión del usuario y le lleva a la página de inicio de sesión.
- **Escenario Alternativo 2.a**
  1. El sistema detecta que el usuario no ha dejado ningún "me gusta" a ningún evento, y muestra en lugar de la lista el mensaje "Da 'me gusta' a los eventos que más te interesen y recibe información relevante sobre ellos".
- **Postcondiciones**
  - El usuario cierra su sesión.

5.1.6. Caso de uso "Seleccionar una categoría y explorar su lista de eventos"

- **Descripción**
  - El sistema muestra una lista con las diferentes categorías de eventos. Mostrará una lista de los eventos filtrados por la categoría seleccionada por el usuario.

- **Actores**
  - Primario: Usuario
- **Precondiciones**
  - El usuario ha iniciado sesión.
- **Escenario Principal**
  1. El sistema redirige al usuario a la pantalla "Explorador de Eventos", que muestra una lista con las diferentes categorías de eventos.
  2. El usuario pulsa una categoría de eventos.
  3. El sistema redirige al usuario a la pantalla "Lista de Eventos". Cada evento tiene, aparte de desplegables con información, el botón "Me gusta" o "Ya No me Gusta".
  4. El usuario pulsa opcionalmente los botones desplegables para leer información sobre el evento.
- **Escenario Alternativo 2.a**
  1. El usuario no selecciona ninguna categoría de eventos.
  2. Se sale del caso de uso.
- **Postcondiciones**
  - El usuario visualiza los eventos de la categoría seleccionada, así como información sobre los mismos.

5.1.7. Caso de uso "Marcar/desmarcar "me gusta" de un evento"

- **Descripción**
  - El sistema muestra una lista con las diferentes categorías de eventos. Mostrará una lista de los eventos filtrados por la categoría seleccionada por el usuario.
- **Actores**
  - Primario: Usuario
- **Casos de uso a los que extiende**
  - Caso de uso "Seleccionar una categoría y explorar su lista de eventos"
  - Caso de uso "Consultar en el mapa próximos eventos y aforo de parking"
- **Precondiciones**
  - El usuario ha iniciado sesión.
- **Escenario Principal**

1. El sistema muestra la pantalla “Lista de Eventos” en caso de haberse realizado el caso de uso Caso de uso “Seleccionar una categoría y explorar su lista de eventos”, o muestra información sobre un único evento al seleccionarlo en el mapa al realizar el caso de uso Caso de uso “Consultar en el mapa próximos eventos y aforo de parking”. Cada evento tiene, aparte de desplegables con información, el botón “Me gusta” o “Ya No me Gusta”, dependiendo de si el usuario dejó o no un “me gusta” al evento.
2. El usuario pulsa opcionalmente los botones desplegables para leer información sobre el evento.
3. El sistema muestra información del evento.
4. El usuario marca o desmarca un “me gusta” pulsando el botón “Me gusta” o “Ya No me Gusta” respectivamente.

- **Escenario Alternativo 2.a**
  1. El usuario no pulsa ningún desplegable de información.
  2. se salta al paso 6 del escenario principal.
- **Escenario Alternativo 4.a**
  1. El usuario no pulsa ningún botón.
  2. se sale del caso de uso.
- **Postcondiciones**
  - El usuario visualiza los eventos de la categoría seleccionada, así como información sobre los mismos.
  - El usuario marca o desmarca un “me gusta” para alguno de los eventos de la lista.

5.1.8. Caso de uso “Explorar lista de “me gusta” ”

- **Descripción**
  - El sistema muestra una lista con los eventos a los que el usuario ha dado “me gusta”.
- **Actores**
  - Primario: Usuario
- **Precondiciones**
  - El usuario ha iniciado sesión.
- **Escenario Principal**
  1. El usuario pulsa la opción “Me Gusta” en la barra de navegación.

2. El sistema muestra una lista con los eventos a los que el usuario ha dado "me gusta". Cada evento tiene, aparte de desplegables con información, los botones "Ver a quién más le gusta" y "Ya No me Gusta".
3. El usuario pulsa opcionalmente los botones desplegables para leer información sobre el evento.
4. El sistema muestra información del evento.

- **Escenario Alternativo 2.a**
  1. El sistema detecta que el usuario no ha dejado ningún "me gusta" a ningún evento, y muestra en lugar de la lista el mensaje "Da 'me gusta' a los eventos que más te interesen y recibe información relevante sobre ellos".
- **Escenario Alternativo 3.a**
  1. El usuario no pulsa ningún desplegable de información.
- **Postcondiciones**
  - El usuario visualiza los eventos a los que ha dado "me gusta", así como información sobre los mismos.

5.1.9. Caso de uso "Desmarcar "me gusta" de la lista"

- **Descripción**
  - El usuario indica que ya no le gusta uno de los eventos de la lista de eventos a los que hadado "me gusta".
- **Actores**
  - Primario: Usuario
- **Precondiciones**
  - El usuario ha iniciado sesión.
- **Casos de uso a los que extiende**
  - Caso de uso "Explorar lista de "me gusta" "
- **Escenario Principal**
  1. El usuario pulsa la opción "Me Gusta" en la barra de navegación.
  2. El sistema muestra una lista con los eventos a los que el usuario ha dado "me gusta". Cada evento tiene, aparte de desplegables con información, los botones "Ver a quién más le gusta" y "No me interesa".
  3. El usuario pulsa el botón "No me Interesa" de uno de los eventos de la lista.
  4. El sistema elimina de la lista el evento en cuestión.
- **Postcondiciones**

- El usuario elimina de su lista de eventos a los que ha dado “me gusta” el evento seleccionado.

5.1.10. Caso de uso “Ver a quién más le gusta un evento”

- **Descripción**
  - El usuario accede a una lista de usuarios que también ha dado “me gusta” a un evento seleccionado.
- **Actores**
  - Primario: Usuario
- **Precondiciones**
  - El usuario ha iniciado sesión.
- **Casos de uso a los que extiende**
  - Caso de uso “Explorar lista de “me gusta” ”
- **Escenario Principal**
  1. El usuario pulsa la opción “Me Gusta” en la barra de navegación.
  2. El sistema muestra una lista con los eventos a los que el usuario ha dado “me gusta”. Cada evento tiene, aparte de desplegables con información, los botones “Ver a quién más le gusta” y “No me interesa”.
  3. El usuario pulsa el botón “Mira a quién más le gusta este evento” de uno de los eventos de la lista.
  4. El sistema redirige al usuario a una nueva pantalla que muestra una lista con los usuarios que también han indicado que le gusta el evento.
- **Escenario Alternativo 4.a**
  1. El sistema detecta que no hay ningún otro usuario que haya dejado un “me gusta” para el evento seleccionado, por lo que mostrará un mensaje indicándolo en vez de la lista de usuarios.
- **Postcondiciones**
  - El sistema muestra la lista de usuarios que han dejado un “me gusta” al evento seleccionado en caso de haberlos.

5.1.11. Caso de uso “Crear chat individual”

- **Descripción**
  - El usuario elige a uno de los usuarios de la lista para agregarlo a sus chats.
- **Actores**

- Primario: Usuario

- **Precondiciones**
  - El usuario ha iniciado sesión.
  - El usuario ha consultado la lista de usuarios a los que les gusta un evento (Caso de uso “Explorar lista de “me gusta” ”) o ha consultado la lista de miembros de un grupo (Caso de uso “Ver grupo”).
- **Casos de uso a los que extiende**
  - Caso de uso “Explorar lista de “me gusta” ”
  - Caso de uso “Ver grupo”
- **Escenario Principal**
  1. El sistema muestra la lista de usuarios, y junto a los que el usuario aún no ha agregado a sus chats un botón con el texto “Agregar a mis Chats” para tal propósito.
  2. El usuario selecciona un usuario para agregarlo a sus chats.
  3. Se vuelve al paso 1 del escenario principal hasta que el usuario no salga de la pantalla.
- **Escenario Alternativo 1.a**
  1. El sistema muestra la lista de usuarios a los que les gusta un evento (Caso de uso “Explorar lista de “me gusta” ”) pero ningún otro usuario ha dado “me gusta” a ese evento. La lista se muestra vacía.
- **Escenario Alternativo 1.b**
  1. El sistema muestra la lista de usuarios a los que les gusta un evento (Caso de uso “Explorar lista de “me gusta” ”) o ha consultado la lista de miembros de un grupo (Caso de uso “Ver grupo”), pero no existe en la lista ningún usuario que no tenga agregado a sus chats.
  2. Se sale del caso de uso.
- **Postcondiciones**
  - El usuario agrega a sus chats a otros usuarios de manera opcional.

5.1.12. Caso de uso “Crear chat grupal”

- **Descripción**
  - El usuario crea un chat grupal con uno o más usuarios.
- **Actores**
  - Primario: Usuario
- **Precondiciones**

- El usuario ha iniciado sesión.
- El usuario debe de haber accedido a su lista de chats pulsando "Mis Chats" en la barra de navegación.

- **Escenario Principal**

  1. El sistema muestra la pantalla para crear el grupo en la que se da la posibilidad de establecer y visualizar una imagen de grupo, rellenar un input para dar un nombre al grupo y Añadir a los usuarios que compondrán el grupo de la lista de usuarios que tiene agregados a su chat. También se muestra al usuario un botón "Crear Grupo".
  2. El usuario establece una imagen de grupo, rellena el input para dar un nombre al grupo y añade a los usuarios que compondrán el grupo de la lista de usuarios que tiene agregados a su chat. Tras ello, el usuario crea el grupo pulsando el botón "Crear Grupo".
  3. El sistema redirige al usuario a la pantalla de chats.

- **Escenario Alternativo 1.a**

  1. El sistema detecta que el usuario no ha agregado a ningún otro usuario a sus chats y muestra la lista vacía.
  2. El usuario pulsa el botón "Crear Grupo".
  3. El sistema muestra una alerta con el mensaje "Debes seleccionar al menos un usuario para poder crear grupo".

- **Escenario Alternativo 1.b**

  1. El usuario no establece una imagen de grupo, pero rellena el input para dar un nombre al grupo y añade a los usuarios que compondrán el grupo de la lista de usuarios que tiene agregados a su chat. Tras ello, el usuario crea el grupo pulsando el botón "Crear Grupo".
  2. Se vuelve al paso 3 del escenario principal.

- **Escenario Alternativo 1.c**

  1. El usuario no rellena el input para dar un nombre al grupo, pero establece una imagen de grupo y añade a los usuarios que compondrán el grupo de la lista de usuarios que tiene agregados a su chat. Tras ello, el usuario crea el grupo pulsando el botón "Crear Grupo".
  2. Se vuelve al paso 3 del escenario principal.

- **Escenario Alternativo 1.d**

  1. El usuario no establece una imagen de grupo ni rellena el input para dar un nombre al grupo, pero añade a los usuarios que compondrán el grupo de la lista de usuarios que tiene agregados a su chat. Tras ello, el usuario crea el grupo pulsando el botón "Crear Grupo".
  2. Se vuelve al paso 3 del escenario principal.

- **Postcondiciones**
  - El usuario crea un chat grupal con los usuarios seleccionados.

5.1.13. Caso de uso "Eliminar un chat individual/grupal"

- **Descripción**
  - El usuario elimina un chat.
- **Actores**
  - Primario: Usuario
- **Precondiciones**
  - El usuario ha iniciado sesión.
  - El usuario tiene, al menos, un chat en su lista de chats.
  - El usuario debe de haber accedido a su lista de chats pulsando "Mis Chats" en la barra de navegación.
- **Escenario Principal**
  1. El sistema muestra la pantalla de lista de chats al usuario.
  2. El usuario mantiene pulsado un chat de la lista.
  3. El sistema muestra un mensaje por pantalla, dando la opción al usuario de eliminar el chat o cancelar la acción.
  4. El usuario selecciona la opción de eliminar el chat.
- **Escenario Alternativo 1.a**
  1. El sistema detecta que el usuario no ha agregado a ningún otro usuario a sus chats y muestra la lista vacía.
- **Escenario Alternativo 4.a**
  1. El usuario cancela la opción de eliminar el chat.
- **Postcondiciones**
  - El usuario elimina el chat seleccionado de su lista de chats.

5.1.14. Caso de uso "Acceder a un chat individual/grupal"

- **Descripción**
  - El usuario accede a un chat.
- **Actores**
  - Primario: Usuario

- **Precondiciones**
  - El usuario ha iniciado sesión.
  - El usuario tiene, al menos, un chat en su lista de chats.
  - El usuario debe de haber accedido a su lista de chats pulsando "Mis Chats" en la barra de navegación.
- **Escenario Principal**
  1. El sistema muestra la pantalla de lista de chats al usuario.
  2. El usuario pulsa un chat de la lista.
  3. El sistema redirige al usuario a una pantalla que muestra en la cabecera la foto y nombre del chat. En la pantalla se muestran los mensajes que han sido enviados y recibidos dentro del chat, así como un input en la parte inferior cuya finalidad es que el usuario escriba y envíe un nuevo mensaje.
- **Postcondiciones**
  - El usuario accede al chat seleccionado.

5.1.15. Caso de uso "Enviar mensaje a un chat individual/-grupal"

- **Descripción**
  - El usuario envía un mensaje a un chat.
- **Actores**
  - Primario: Usuario
- **Precondiciones**
  - El usuario ha iniciado sesión.
  - El usuario tiene, al menos, un chat en su lista de chats.
  - El usuario debe de haber accedido a su lista de chats pulsando "Mis Chats" en la barra de navegación.
  - El usuario realiza el Caso de uso "Acceder a un chat individual/grupal".
- **Casos de uso que incluye**
  - Caso de uso "Acceder a un chat individual/grupal"
- **Escenario Principal**
  1. El usuario modifica el valor del input para escribir un nuevo mensaje y pulsa el botón "Enviar".
  2. El sistema añade el mensaje enviado por el usuario al conjunto de mensajes enviados al chat y lo muestra por pantalla junto a ellos.

- **Escenario Alternativo 1.a**
  1. El sistema detecta que el mensaje es vacío y no hace nada al pulsar el botón de envío de mensaje.
  2. Se vuelve al paso 1 del escenario principal.
- **Postcondiciones**
  - El usuario envía un mensaje al chat.

5.1.16. Caso de uso "Ver perfil de otro usuario"

- **Descripción**
  - El sistema pone a disposición del usuario los datos de perfil de otro usuario.
- **Actores**
  - Primario: Usuario
- **Precondiciones**
  - El usuario ha iniciado sesión.
  - El usuario tiene, al menos, un chat en su lista de chats.
  - El usuario debe de haber accedido a su lista de chats pulsando "Mis Chats" en la barra de navegación.
  - El usuario realiza el Caso de uso "Acceder a un chat individual/grupal", por lo que debe tener almenos a un usuario agregado a su chat. O bien el usuario realiza el caso de uso Caso de uso "Ver a quién más le gusta un evento".
- **Casos de uso a los que extiende**
  - Caso de uso "Acceder a un chat individual/grupal"
  - Caso de uso "Ver grupo"
  - Caso de uso "Ver a quién más le gusta un evento"
- **Escenario Principal**
  1. El sistema muestra el nickname de usuario y su imagen en la cabecera del usuario del chat (cuando se extiende al Caso de uso "Acceder a un chat individual/grupal"), o bien una lista de usuarios miembros del grupo con sus respectivos nickname e imagen de perfil (cuando se extiende al Caso de uso "Ver grupo"). En caso de haber realizado el caso de uso Caso de uso "Ver a quién más le gusta un evento", se mostrará también la lista de usuarios como la anterior, pero esta vez de los usuarios con quien comparte un "me gusta".
  2. El usuario pulsa el nickname o la imagen del usuario del que quiere consultar su perfil.

3. El sistema muestra los datos del usuario, como su imagen de perfil, nickname, Descripción, nombre y apellidos. Además también muestra una lista con el título "Eventos Que Me Gustan" de las distintas categorías de evento junto al número de eventos a los que ha dado "me gusta" de cada una, ordenados de mayor a menor.

- **Escenario Alternativo 3.a**

1. El sistema detecta que el usuario seleccionado en ese momento ha quitado cualquier "me gusta" que pudiese haber dejado a algún evento, y muestra en lugar de la lista el mensaje "UPS, parece que nickname de usuario aún no ha interactuado con la aplicación".

- **Postcondiciones**

  - El usuario visualiza los datos de perfil del usuario seleccionado.

5.1.17. Caso de uso "Ver grupo"

- **Descripción**

  - El sistema pone a disposición del usuario los datos del chat grupal.

- **Actores**

  - Primario: Usuario

- **Precondiciones**

  - El usuario ha iniciado sesión.
  - El usuario tiene, al menos, un chat de grupo en su lista de chats.
  - El usuario debe de haber accedido a su lista de chats pulsando "Mis Chats" en la barra de navegación.
  - El usuario realiza el Caso de uso "Acceder a un chat individual/grupal" y accede a un chat de grupo.

- **Casos de uso a los que extiende**

  - Caso de uso "Acceder a un chat individual/grupal"

- **Escenario Principal**

1. El sistema muestra el nombre de grupo y su imagen en la cabecera del chat (cuando se extiende al Caso de uso "Acceder a un chat individual/grupal"), o bien una lista de usuarios miembros del grupo con sus respectivos nickname e imagen de perfil (cuando se extiende al Caso de uso "Ver grupo").
2. El usuario pulsa el nombre de grupo o su imagen.
3. El sistema muestra los datos del grupo, como su imagen y nombre. Además también muestra una lista con el título "Miembros del grupo", que permitirá para cada uno de ellos:

- Consultar su perfil pulsando su nickname o imagen (se realiza Caso de uso "Ver perfil de otro usuario").
- Eliminar al usuario del grupo pulsando el botón "Eliminar".
- Agregar al usuario a la lista de chats pulsando el botón "Agregar a mis Chats".

El usuario que usa la aplicación también se incluye en dicha lista, y podrá acceder a su propio perfil pulsando su nickname o imagen (se realiza Caso de uso "Ver perfil") o salir del grupo pulsando el botón "Salir del Grupo". Además, se muestra el botón "Editar Grupo" implementado para tal fin.

- **Escenario Alternativo 3.a**

1. El sistema detecta que el usuario seleccionado en ese momento ha quitado cualquier "me gusta" que pudiese haber dejado a algún evento, y muestra en lugar de la lista el mensaje "UPS, parece que nickname de usuario aún no ha interactuado con la aplicación".

- **Postcondiciones**
  - El usuario visualiza los datos del grupo.

5.1.18. Caso de uso "Modificar grupo"

- **Descripción**
  - El usuario modifica los datos de un chat grupal.
- **Actores**
  - Primario: Usuario
- **Precondiciones**
  - El usuario ha iniciado sesión.
  - El usuario tiene, al menos, un chat de grupo en su lista de chats.
  - El usuario debe de haber accedido a su lista de chats pulsando "Mis Chats" en la barra de navegación.
  - El usuario realiza el Caso de uso "Acceder a un chat individual/grupal" y accede a un chat de grupo.
- **Casos de uso a los que extiende**
  - Caso de uso "Ver grupo"
- **Escenario Principal**

1. El usuario pulsa el botón "Editar Grupo".
2. El sistema muestra junto a la imagen de grupo, un botón de edición.
3. El usuario pulsa el botón de edición de imagen de grupo.

4. El sistema abre la galería del usuario y muestra las imágenes almacenadas en el dispositivo móvil.
5. El usuario selecciona una nueva imagen de grupo.
6. El sistema muestra un input contenedor del nombre de grupo.
7. El usuario modifica el contenido del input de nombre de grupo.
8. El sistema muestra una lista de nombre "Añadir al Grupo Usuarios Agregados a Mis Chats". Esta muestra los usuarios que el usuario que usa la aplicación ha agregado a sus chats, junto a un botón para añadirlo o eliminarlo del grupo.
9. El usuario elimina o añade al grupo a uno de los usuarios de la lista.
10. El sistema muestra un botón "Editar Grupo" que al ser pulsado aplicará los cambios realizados.
11. El usuario pulsa el botón "Editar Grupo", confirmando y aplicando los cambios realizados.

- **Escenario Alternativo 2.a**
  1. El usuario no pulsa el botón "Editar Grupo", sino que pulsa el botón "Eliminar" y saca del grupo a alguno de los miembros, o bien sale él mismo del grupo pulsando el botón "Salir del Grupo"
  2. Se sale del caso de uso
- **Escenario Alternativo 3.a**
  1. El usuario no pulsa el botón de edición de imagen de grupo.
  2. Se pasa al paso 6 del escenario principal.
- **Escenario Alternativo 7.a**
  1. El usuario no modifica el contenido del input de nombre de grupo.
  2. Se pasa al paso 8 del escenario principal.
- **Escenario Alternativo 9.a**
  1. El usuario no modifica los miembros del grupo, pues no añade ni elimina a ningún usuario de la lista de agregados a sus chats.
  2. Se pasa al paso 10 del escenario principal.
- **Escenario Alternativo 11.a**
  1. El usuario sale de la pantalla sin pulsar el botón "Editar Grupo", por lo que los cambios realizados no se aplican.
  2. Se sale del caso de uso.
- **Postcondiciones**
  - El usuario modifica opcionalmente los datos del grupo.

5.1.19. Caso de uso "Consultar en el mapa próximos eventos y aforo de parking"

- **Descripción**
  - El usuario visualiza en un mapa la ubicación de los distintos eventos del día que seleccione, junto con la ubicación de algunos parking de Madrid con infromación sobre sus plazas libres.
- **Actores**
  - Primario: usuario
- **Precondiciones**
  - La API del Ayuntamiento de Madrid está disponible y su endpoint es accesible.
  - El sistema accede y recoge los datos, procesándolos y almacenándolos en la base de datos.
- **Escenario Principal**
  1. El sistema muestra una pantalla con un **mapa**, donde se visualiza la localización de los diferentes **eventos** que se dan el día actual. También muestra unos **botones** para seleccionar el día del que el usuario quiere ver qué eventos tendrán lugar durante el mismo en el mapa. Además se visualizan algunos de los **parking** de Madrid en el mapa, con información como su nombre y sus plazas libres, cambiando el icono de color dependiendo del número de plazas:
     - El color será azul si no hay datos sobre las plazas libres para el parking.
     - El color será rojo si el número de plazas libres es nulo.
     - El color será amarillo si el número de plazas libres es menor o igual a 20.
     - El color será verde si el número de plazas libres es mayor a 20.
  2. El usuario pulsa uno de los botones y selecciona uno de los días.
  3. El sistema muestra en el mapa los eventos que tendrán lugar durante el día seleccionado por el usuario.
- **Escenario Alternativo 3.a**
  1. El usuario no selecciona ningún día.
  2. El sistema sigue mostrando los eventos para el día actual.
- **Postcondiciones**
  - El usuario visualiza los eventos para el día actual en el mapa o para el día seleccionado, así como la ubicación de los parking con su número de plazas libres.

- **Descripción**
  - La API del Ayuntamiento de Madrid publica nuevos datos de plazas libres de parking en su endpoint.
- **Actores**
  - Primario: API del Ayuntamiento de Madrid
- **Precondiciones**
  - La API del Ayuntamiento de Madrid está disponible y su endpoint es accesible
- **Escenario Principal**
  1. La API del Ayuntamiento de Madrid publica nuevos datos en su endpoint.
  2. El sistema recoge los datos, los procesa y los almacena.
- **Escenario Alternativo 2.a**
  1. Se produce un error y no se almacenan los nuevos datos publicados.
  2. Se sale del caso de uso.
- **Postcondiciones**
  - El sistema almacena los nuevos datos de plazas de parking publicados por la API del Ayuntamiento de Madrid.

5.2. Modelo de interfaz de usuario

En esta sección se deberá incluir un prototipo de baja fidelidad o mockup de la interfaz de usuario del sistema. Además, es preciso elaborar un diagrama de navegación, reflejando la secuencia de pantallas a las que tienen acceso los diferentes roles de usuario y la conexión entre éstas.

5.3. Modelo de comportamiento

En esta sección se incluye el modelo de comportamiento desarrollado a partir de los casos de uso definidos. Para ello, se adjunta el diagrama de secuencia del sistema (ver imagen 5.2), modelando así el comportamiento del mismo y la interacción entre cada uno de los objetos que lo componen, representadas mediante líneas de vida o mensajes.

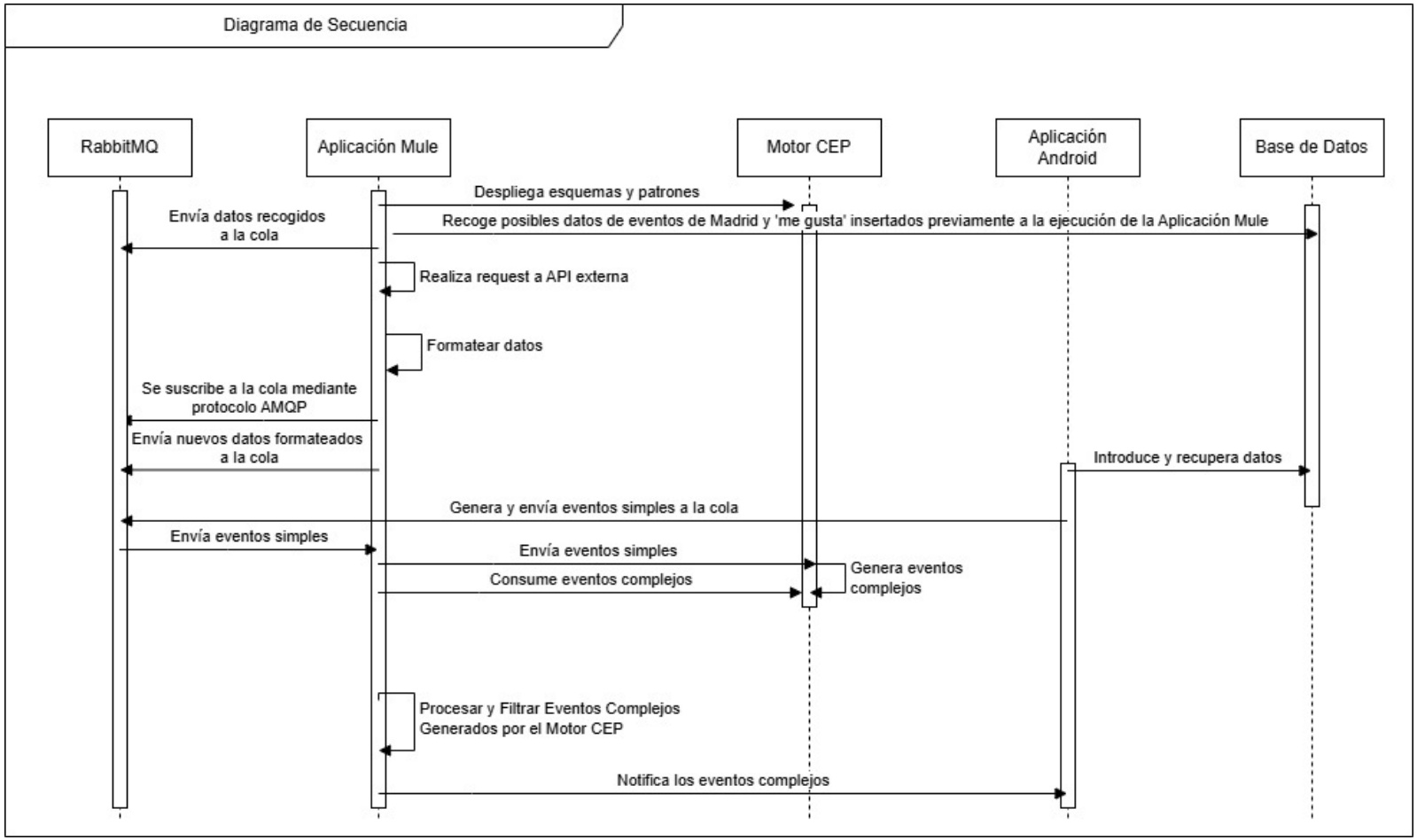


Figura 5.2: Diagrama de secuencia

6. Diseño del sistema

En este capítulo se recoge el diseño de la arquitectura del sistema informático, el diseño físico de datos y el diseño detallado de la interfaz de usuario.

6.1. Arquitectura del sistema

En esta sección se define la arquitectura general del sistema, especificando la infraestructura tecnológica necesaria para dar soporte al software y la estructura de los componentes que lo forman.

Observemos en la imagen 6.1 la arquitectura del sistema. Vamos a detallar las relaciones y comunicación entre los distintos componentes de la misma.

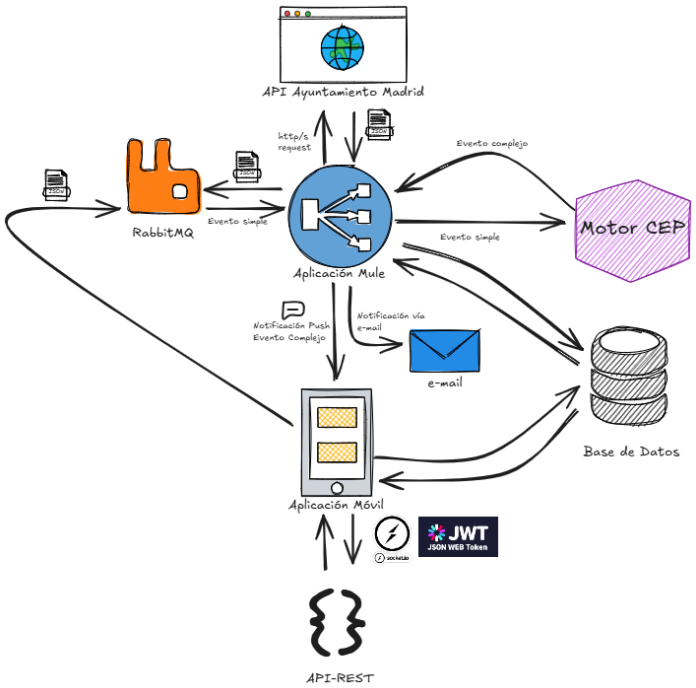


Figura 6.1: Arquitectura lógica del sistema

- **Solicitudes a API externa y almacenamiento persistente de los datos recogidos**: la aplicación Mule realiza solicitudes a la API externa del ayuntamiento de Madrid, y recibe los datos de la misma mediante un json. Estos datos son formateados y almacenados en la base de datos (ver imagen 6.2).

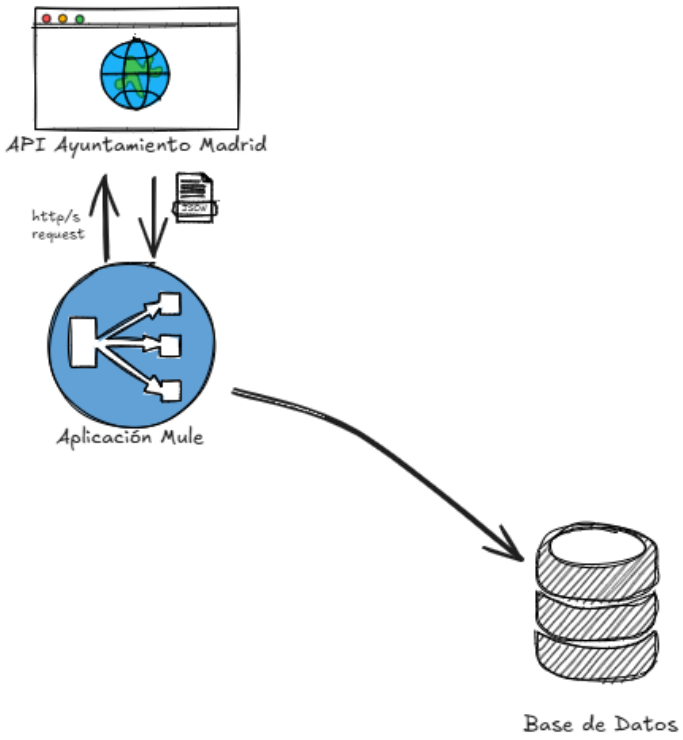


Figura 6.2: Solicitudes a API externa y almacenamiento persistente de los datos recogidos

- **Generación de datos y eventos simples mediante el uso de la aplicación móvil**: La aplicación móvil (front-end) manda solicitudes a la API REST (back-end), cuya implementación hace uso por un lado de Json Web Token (JWT) para hacer el sistema de autenticación más seguro, y por otro lado de web sockets (más concretamente de la librería socket.io de JavaScript) para el desarrollo de la funcionalidad del chat.

  Algunos de los endpoints del back-end permiten recuperar datos de la base de datos o insertarlos. La interacción del usuario con la aplicación genera nuevos datos que son insertados en la base de datos, y nuevos eventos simples que son enviados mediante un json a la correspondiente cola de RabbitMQ (ver imagen 6.3).

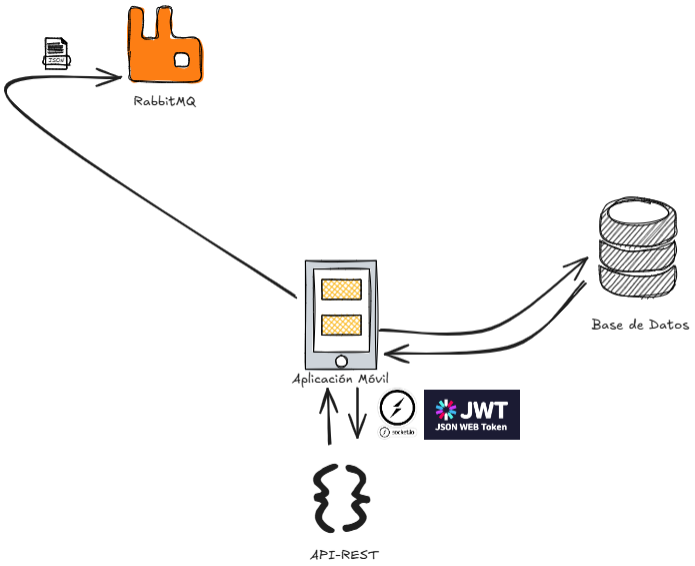


Figura 6.3: Generación de datos y eventos simples mediante el uso de la aplicación móvil

- **Envío de eventos simples desde la aplicación Mule a RabbitMQ**: Los datos recogidos y formateados de la API externa son enviados y almacenados como eventos simples en la respectiva cola RabbitMQ, una para cada tipo de evento. Se ha implementado una medida frente a apagados o reinicios de la aplicación Mule debido a que la naturaleza del proyecto es la de un trabajo de fin de grado. Cuando se ejecuta la aplicación Mule se recogen datos que han sido recogidos y almacenados previamente en la base de datos, y envía los correspondientes eventos simples a la cola de RabbitMQ para que puedan ser procesados por el motor CEP (ver imagen 6.4).

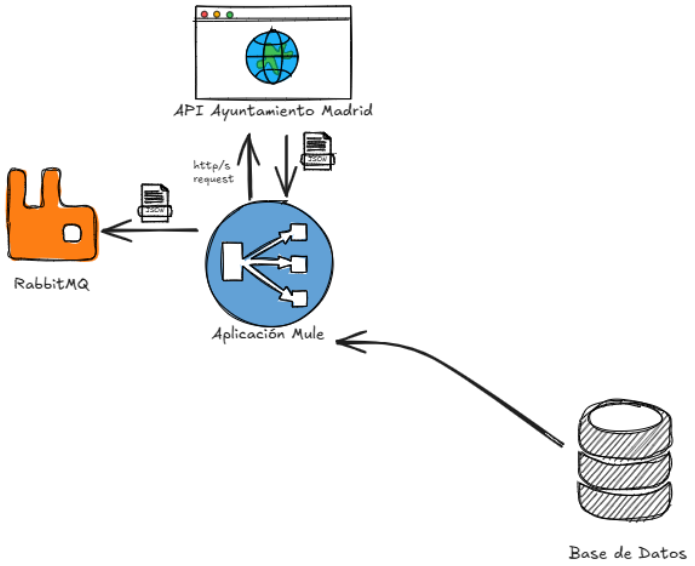


Figura 6.4: Envío de eventos simples desde la aplicación Mule

- **Generación de eventos complejos y notificaciones**: La aplicación Mule recoge los eventos simples desde RabbitMQ y son enviados al motor CEP. Todos los eventos simples serán procesados por el motor CEP, y los patrones desplegados detectarán eventos complejos. Varios tipos de eventos complejos serán insertados en el flujo de eventos, por lo que servirán de nuevo al motor CEP para que detecte otros tipos de eventos complejos. Los eventos complejos serán notificados al usuario vía e-mail o bien mediante notificaciones al dispositivo móvil (ver imagen 6.5).

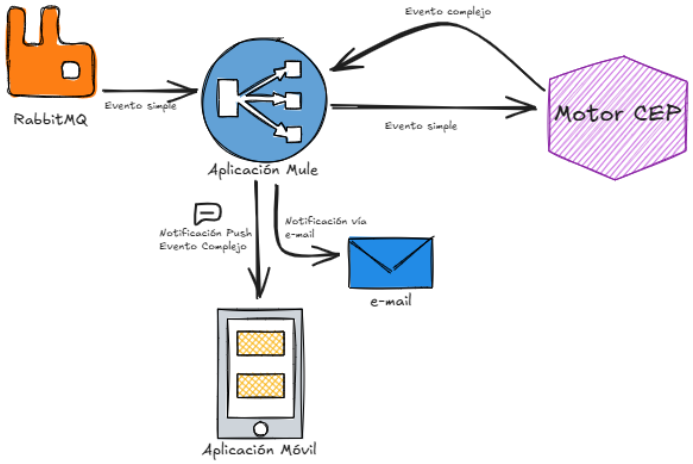


Figura 6.5: Generación de eventos complejos y notificaciones

6.2. Esquema de la base de datos

En esta sección se define la estructura de datos que utilizará el sistema, a partir del modelo de conceptual de clases (ver imagen 6.6), de manera que teniendo presente los requisitos establecidos para el sistema de información y las particularidades del entorno tecnológico, se consiga un acceso eficiente de los datos.

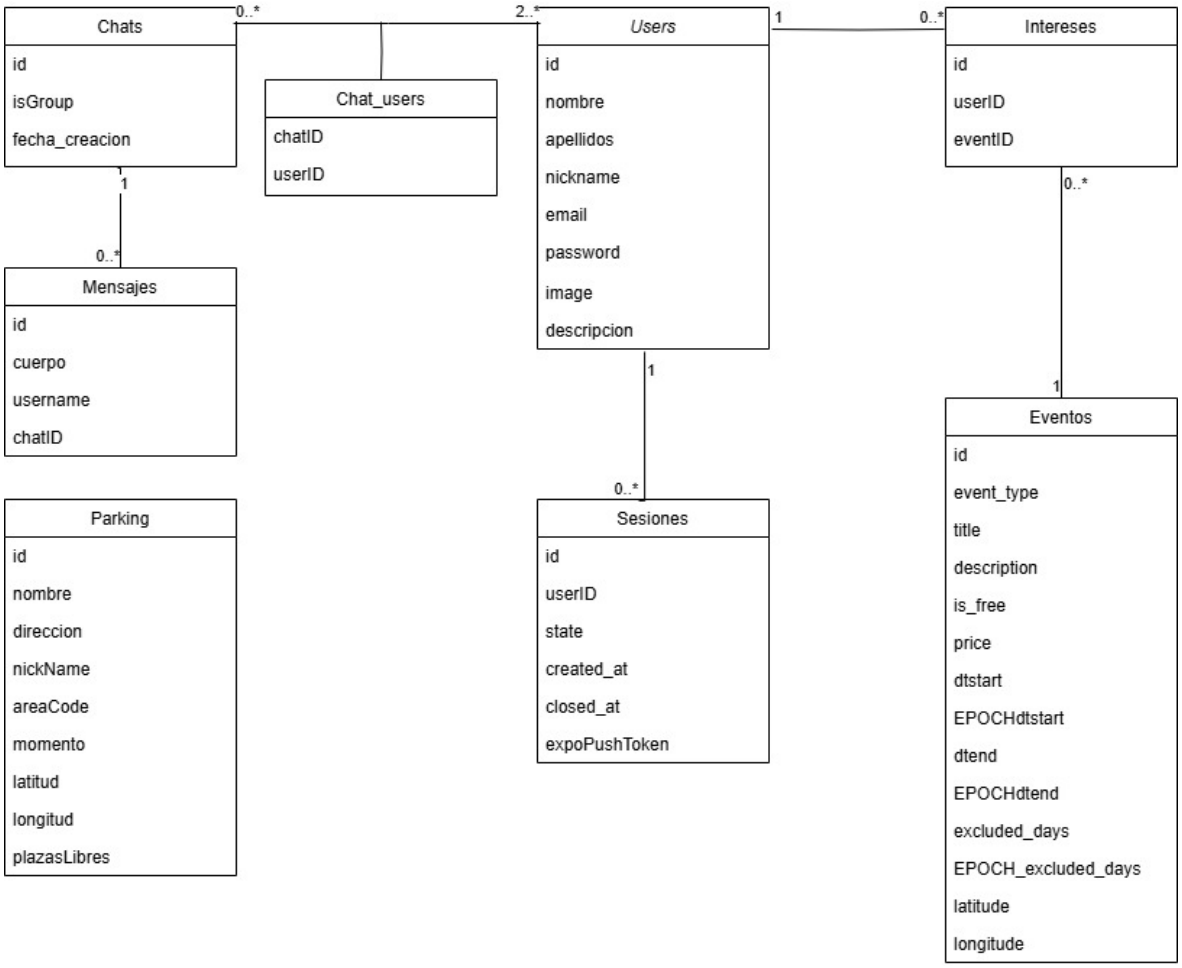


Figura 6.6: Modelo conceptual de clases

6.3. Diseño de la interfaz de usuario de la aplicación móvil

En esta sección se detallarán los principios a tener en cuenta a la hora de implementar la interfaz de usuario del sistema, incluyendo el conjunto de estilos (colores, tipografías, iconografía, etc.) y el conjunto de componentes visuales que se utilizarán. Se incluirá, además, un prototipo de alta fidelidad con el diseño de algunas de las pantallas del sistema.

El diseño de la interfaz de usuario previo a su implementación es esencial si queremos tener una clara idea de hacia dónde dirigimos el trabajo durante el propio proceso de desarrollo. Ayuda a tener una idea clara de las pantallas con las que contará la aplicación final, qué elementos queremos mostrar y de qué manera podrá visualizarlos el usuario. Se ha realizado dicho diseño realizando un mockup de alta fidelidad, es decir, la aplicación se ha desarrollado de manera que cada pantalla siga de la manera más estricta posible su diseño. La paleta de colores muestra los colores que la interfaz de usuario utiliza, siendo uno de los más representativos de la aplicación el color “teal”, el primero de los colores de la paleta. El resto de colores serán utilizados de manera más

secundaria a excepción del último elemento de la paleta, que representa una fusión de colores que será utilizada en algunas de las pantallas de la aplicación.

Hablamos brevemente sobre algunas de las pantallas:

- **Login**: Pantalla que permitirá al usuario autenticarse introduciendo su dirección de correo electrónico y contraseña, o registrarse.
- **Explorador de Eventos**: Pantalla a la que se accede desde el menú de navegación. Muestra botones con las diferentes categorías de eventos que al ser pulsados, mostrarán los eventos de su correspondiente categoría al usuario.
- **Categoría Seleccionada**: Pantalla "Lista de Eventos" a la que se navega pulsando uno de los botones de la anterior pantalla. Muestra los eventos disponibles de la categoría seleccionada. Cada evento contiene desplegables que muestran información sobre el evento, como su precio, fecha y hora de inicio o ubicación donde tendrá lugar el mismo. También cuenta con un botón que permite al usuario indicar que el evento le gusta. Para el caso contrario, si ya se había marcado el evento con un "me gusta", se muestra otro botón para indicar que no le gusta el evento.
- **Me Gusta**: Pantalla "Eventos que te gustan" a la que se accede desde el menú de navegación. Muestra los eventos que han sido marcados con un "me gusta" por el usuario. Igual que ocurría en la pantalla anterior "Lista de Eventos", para cada evento se muestran desplegabes con información sobre el evento. Muestra un botón con el que el usuario podrá indicar que el evento ya no le gusta y otro botón con el texto "Mira a quién más le gusta este evento", que navegará a una nueva pantalla. Esta muestra una lista de usuarios que ya han dado me gusta a ese evento, ofreciendo al usuario la posibilidad de añadirlos al chat.
- **Lista de Chats**: Esta pantalla muestra una lista de usuarios que hayan sido agregados al chat mediante la funcionalidad del botón "Mira a quién más le gusta este evento" de la pantalla "Me Gusta". Además mostrará los grupos creados a los que pertenezca el usuario. Si se deja pulsado un chat, la aplicación dará la opción de borrarlo.
- **Chat**: Esta pantalla muestra un cuadro de texto para enviar un mensaje al usuario o usuarios correspondientes.
- **Perfil**: A esta pantalla se accede o bien desde el menú de navegación para mostrar información del propio perfil, o bien desde la pantalla "Chat". En el primero de los casos permite añadir o modificar la descripción y foto de perfil. Muestra la situación en que el usuario aún no ha dado "me gusta" a ningún evento.
- **Perfil con Descripción y "Me Gusta"**: No es sino la pantalla "Perfil" en el caso en que se haya dado algún "me gusta" a cualquier evento. Muestra las distintas categorías de Eventos junto al número de eventos de dicha categoría que han sido marcados con un "me gusta" por el usuario.
- **Consultar Grupo**: A esta pantalla se accede desde la pantalla "Chat" en el caso de que el chat corresponda a un grupo. Muestra la foto del grupo junto a su nombre, y una lista de los miembros del grupo. También un botón que permite

editar ciertos aspectos del grupo y que navega a la pantalla “Editar Grupo”.

- **Editar Grupo**: A esta pantalla se accede desde la pantalla “Consultar Grupo” pulsando su botón “Editar Grupo”. Permite modificar la imagen de grupo y nombre de grupo, además de eliminar a algún integrante del grupo. Mostrará también a los usuarios que se tengan añadidos al chat(lo que variará de un integrante del grupo a otro) para poder añadirlos también al grupo.

- **Mapa**: La pantalla “Próximos Eventos” muestra una serie de botones para seleccionar para qué día nos interesa visualizar los eventos que están por celebrarse. Además de los eventos, se mostrarán también algunos de los parking distribuidos por Madrid. Los parking presentarán distinto color dependiendo del número de plazas libres disponibles.

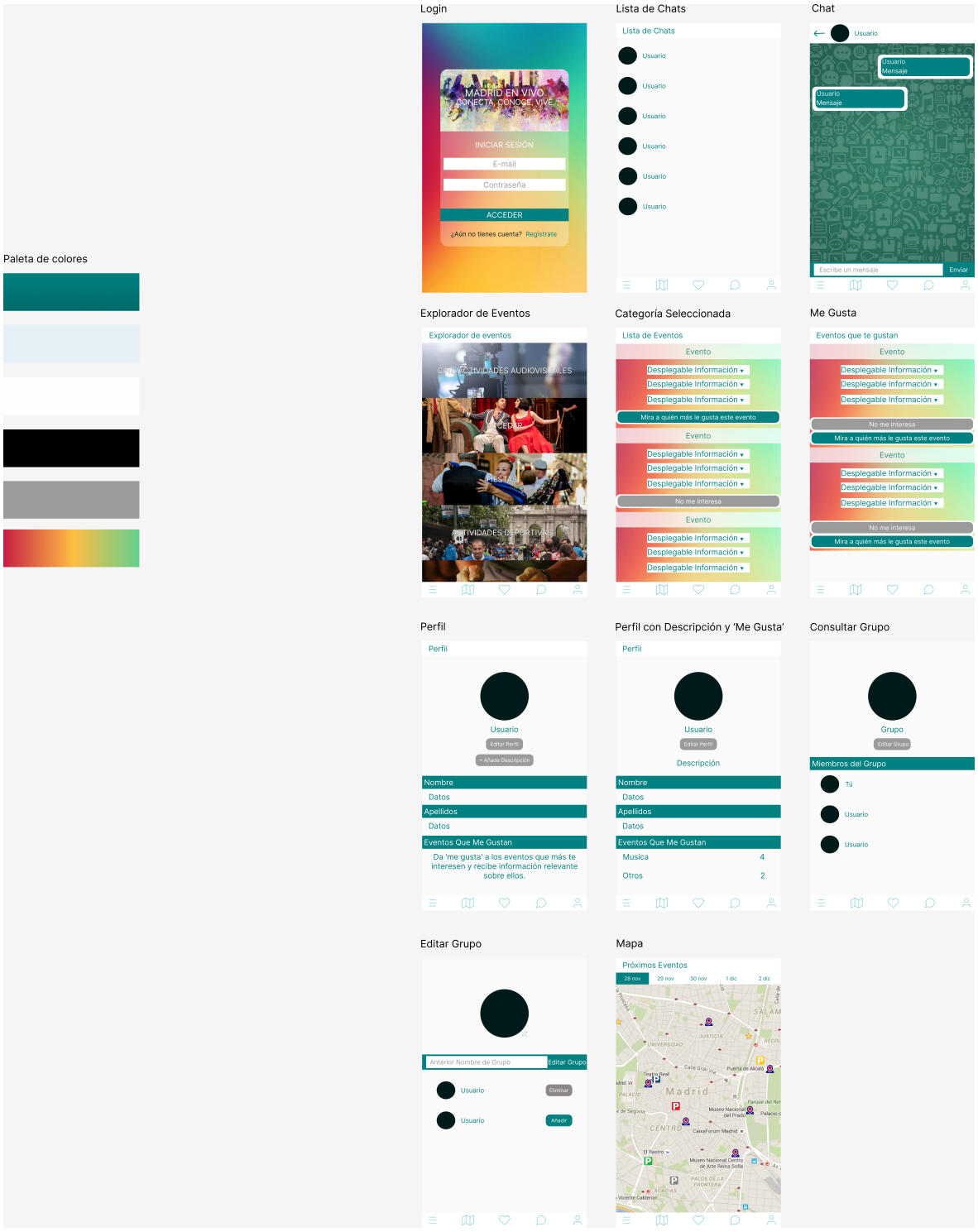


Figura 6.7: Mockup

7. Implementación

Este capítulo trata sobre todos los aspectos relacionados con la implementación del sistema en código, haciendo uso de un determinado entorno tecnológico.

7.1. Aplicación Mule

7.1.1. Definición de esquemas

Se definen en el extracto de código 7.1 los esquemas que representarán los diferentes tipos de eventos simples que detectará el motor CEP.

Extracto de código 7.1: Definición de esquemas

```
@public @buseventtype create json schema EventosMadrid as
    (eventTypeName String, id Integer, event_type String,
    title String, description String, price String,
   latitude Double, longitude Double, dtstart String,
   EPOCHdtstart Long, dtend String, EPOCHdtend Long,
   daysOfWeek String,  EPOCH_excluded_days long[]);

@public @buseventtype create json schema
   InicioSesionFallido as (eventTypeName String, email
   String);

@public @buseventtype create json schema MeGusta as (
   eventTypeName string, id Integer, userID Integer,
   eventID Integer);

@public @buseventtype create json schema Ubicacion as (
   eventTypeName string, userID Integer, email string,
   latitude Double, longitude Double, city String, region
    String, country String);

@public @buseventtype create json schema Parking as (
   eventTypeName String, id Integer, latitude Double,
   longitude Double, address String, nickName String,
   name String, moment String, freeParkingSpaces Integer,
    areaCode String  );

@public @buseventtype create json schema dummy as (
   eventTypeName String, id Integer, EPOCH_now Long, now
   String);
```

7.1.2. Definición de patrones

Vemos en este apartado cómo hemos implementado en Esper los diferentes patrones, que una vez desplegados permitirán al motor CEP detectar situaciones de interés para posteriormente notificar al usuario.

- **DetectaInicioSesionFallido**: Como su propio nombre indica, este patrón detectará cada evento simple de tipo "InicioSesionFallido".

Extracto de código 7.2: Definición del patrón "DetectaInicioSesionFallido"

```
@public @Name('DetectaInicioSesionFallido')
select 'DetectaInicioSesionFallido' as
   eventTypeName, i.email as email, count(*) as
   count
from InicioSesionFallido.win:time(1 min) i
group by i.email
having count(*)=3;
```

- **DetectaFuerzaBruta**: Como su propio nombre indica, este patrón detectará cada evento simple de tipo "DetectaFuerzaBruta". Se interpretará como un intento de ataque de fuerza bruta a la sesión del usuario la situación en que, para la cuenta dada por el correo del usuario se captan en un tiempo de un minuto como mínimo 100 intentos de sesión fallidos.

Extracto de código 7.3: Definición del patrón "DetectaFuerzaBruta"

```
@public @Name('DetectaFuerzaBruta')
select 'DetectaFuerzaBruta' as eventTypeName, i.
   email as email, count(*) as count
from InicioSesionFallido.win:time(1 min) i
group by i.email
having count(*)=100;
```

- **UbicacionSospechosa**: Como su propio nombre indica, este patrón detectará cada cambio de país en la ubicación para un usuario concreto. Se podría pensar que entonces, si un usuario inicia sesión en otro país y envía datos de ubicación de manera simultánea a la vez que el usuario legítimo de la cuenta, este patrón estaría detectando eventos complejos de manera repetitiva. Sin embargo, la implementación del sistema no permite dos inicios de sesión en dispositivos distintos, por lo que no se corre ese riesgo. El patrón está hecho de manera que para cada ubicación que llegue, se compruebe con solo la primera de las siguientes ubicaciones de cada usuario que sean recibidas.

Extracto de código 7.4: Definición del patrón "UbicacionSospechosa"

```
@public @Name('UbicacionSospechosa')
```

```
select 'UbicacionSospechosa' as eventTypeName, u2
   .userID as userID, u2.city as city, u2.region
   as region, u2.country as country, u2.email as
   email
from pattern [
(every u1=Ubicacion ->  every-distinct (u2.userID
   ) u2=Ubicacion )
    ]
WHERE u1.userID=u2.userID AND u1.country!=u2.
   country;
```

- **MenosDe5Dias**: Este patrón detecta el caso en que quedan menos de 5 días para el comienzo de un evento de Madrid. Para su implementación se ha hecho uso de un temporizador **timer:interval**, gracias al que realiza una comprobación cada minuto para cada evento de Madrid concreto hasta que se cumpla la condición.

  Teniendo en cuenta el consumo de recursos, en ese marco de 5 días se podría incrementar bastante más el tiempo entre intervalos, dado que un retraso incluso de varias horas en el aviso tampoco sería muy significativo para el usuario, y sin embargo ahorra al sistema hacer comprobaciones constantemente en intervalos de menor tiempo hasta que se cumpla la condición. Sin embargo debido a que el proyecto realizado es un trabajo de fin de grado, de cara a la exposición se ha decidido que el tiempo entre intervalos sea de un minuto para facilitar la defensa.

  La condición a evaluar es que el Timestamp del momento en que se realiza la comprobación del patrón tendrá que estar entre los siguientes valores EPOCH:

  - Momento en que quedan 5 días para el comienzo del evento:

    $$e.EPOCHdtstart - 432000000$$

  - Momento en que quedan 24 horas para el comienzo del evento:

    $$e.EPOCHdtstart - 86400000$$

  El evento detectado se inserta en el flujo de eventos para que pueda ser detectado por otros patrones. Una vez alcanzado el momento en que queden 24 horas para la fecha de inicio del evento, será tarea del **patrón "MenosDe24h"** detectar su respectivo motivo de alerta.

Extracto de código 7.5: Definición del patrón "MenosDe5Dias"

```
@public @Name('MenosDe5Dias')
INSERT INTO MenosDe5Dias
select 'MenosDe5Dias' as eventTypeName, e.id as
   id, e.title as title, e.dtstart as dtstart,
   current_timestamp() as currentTime
from pattern [every-distinct (e.id) e=
   EventosMadrid ->  every timer:interval(1 min)
   ]
```

```
where current_timestamp() >= (e.EPOCHdtstart -
   432000000) AND current_timestamp() < (e.
   EPOCHdtstart - 86400000)
GROUP BY e.id
HAVING COUNT(*) = 1;
```

- **MeGustaMenosDe5Dias**: Detecta cada vez que un usuario distinto le da "me gusta" a un evento detectado por el **patrón "MenosDe5Dias"**, para el que queda menos de cinco días. Este patrón ha sido implementado de forma que el evento complejo sea detectado por el motor CEP independientemente del orden en que se inserten en el flujo de eventos el evento complejo **"MenosDe5Dias"** y el evento simple **"me gusta"**, teniendo en cuenta las tres posibles situaciones:

  1. Un usuario da "me gusta" al evento antes de que queden menos de 5 días para su comienzo y, por tanto, antes de que el motor CEP detecte el evento complejo **"MenosDe5Dias"**.

  2. Quedan menos de 5 días para el comienzo del evento, pero al **patrón "MenosDe5Dias"** le queda aún una comprobación pendiente para ese evento debido al temporizador usado para su implementación, gracias al que realiza una comprobación cada minuto para cada evento de Madrid concreto hasta que se cumpla la condición.

  3. Quedan menos de 5 días para el comienzo del evento, se detecta el evento complejo **"MenosDe5Dias"** y tras ello un usuario da "me gusta" al evento.

Extracto de código 7.6: Definición del patrón "MeGustaMenosDe5Dias"

```
@public @Name('MeGustaMenosDe5Dias')
select 'MeGustaMenosDe5Dias' as eventTypeName, m.
   id as eventID, m.title as title, mg.userID as
   userID
from pattern [
(every m=MenosDe5Dias -> every-distinct (mg.id)
   mg=MeGusta)
OR
(every-distinct (mg.id) mg=MeGusta -> every m=
   MenosDe5Dias)
]
where mg.eventID = m.id;
```

- **MenosDe24h**: Este patrón detecta el caso en que quedan menos de 24 horas para el comienzo de un evento de Madrid. Para su implementación se ha hecho uso de un temporizador **timer:interval**, gracias al que realiza una comprobación cada minuto para cada evento de Madrid concreto hasta que se cumpla la condición.

  Teniendo en cuenta el consumo de recursos, en ese marco de 24 horas se podría incrementar bastante más el tiempo entre intervalos, dado que un retraso de varios minutos o hasta incluso de una hora en el aviso tampoco sería muy significativo para el usuario, y sin embargo ahorra al sistema hacer comprobaciones constan-

temente en intervalos de menor tiempo hasta que se cumpla la condición. Sin embargo debido a que el proyecto realizado es un trabajo de fin de grado, de cara a la exposición se ha decidido que el tiempo entre intervalos sea de un minuto para facilitar la defensa.

La condición a evaluar es que el Timestamp del momento en que se realiza la comprobación del patrón tendrá que estar entre los siguientes valores EPOCH:

- Momento en que quedan 24 horas para el comienzo del evento:

$$e.EPOCHdtstart - 86400000$$

- Momento en que quedan 2 horas para el comienzo del evento:

$$e.EPOCHdtstart - 7200000$$

El evento detectado se inserta en el flujo de eventos para que pueda ser detectado por otros patrones. Una vez alcanzado el momento en que queden 2 horas para la fecha de inicio del evento, será tarea del **patrón "MenosDe2h"** detectar su respectivo motivo de alerta.

Extracto de código 7.7: Definición del patrón "MenosDe24h"

```
@public @Name('MenosDe24h')
INSERT INTO MenosDe24h
select 'MenosDe24h' as eventTypeName, e.id as id,
    e.title as title, e.dtstart as dtstart,
   current_timestamp() as currentTime
FROM pattern [
every-distinct (e.id) e=EventosMadrid ->  every
   timer:interval(1 min)
]
where current_timestamp() >= (e.EPOCHdtstart -
   86400000) AND current_timestamp() < (e.
   EPOCHdtstart - 7200000)
GROUP BY e.id
HAVING COUNT(*) = 1;
```

- **MeGustaMenosDe24h**: Detecta cada vez que un usuario distinto le da "me gusta" a un evento detectado por el **patrón "MenosDe24h"**, para el que queda menos de 24 horas. Este patrón ha sido implementado de forma que el evento complejo sea detectado por el motor CEP independientemente del orden en que se inserten en el flujo de eventos el evento complejo **"MenosDe24h"** y el evento simple **"me gusta"**, teniendo en cuenta las tres posibles situaciones:

  1. Un usuario da "me gusta" al evento antes de que queden menos de 24 horas para su comienzo y, por tanto, antes de que el motor CEP detecte el evento complejo **"MenosDe24h"**.

2. Quedan menos de 24 horas para el comienzo del evento, pero al **patrón "MenosDe24h"** le queda aún una comprobación pendiente para ese evento debido al temporizador usado para su implementación, gracias al que realiza una comprobación cada minuto para cada evento de Madrid concreto hasta que se cumpla la condición.

3. Quedan menos de 24 horas para el comienzo del evento, se detecta el evento complejo **"MenosDe24h"** y tras ello un usuario da "me gusta" al evento.

Extracto de código 7.8: Definición del patrón "MeGustaMenosDe24h"

```
@public @Name('MeGustaMenosDe24h')
INSERT INTO MeGustaMenosDe24h
select 'MeGustaMenosDe24h' as eventTypeName, m.id
    as eventID, m.title as title, mg.userID as
   userID
from pattern [
(every m=MenosDe24h -> every-distinct (mg.id) mg=
   MeGusta)
OR
(every-distinct (mg.id) mg=MeGusta -> every m=
   MenosDe24h)
]
where mg.eventID = m.id;
```

- **MenosDe2h**: Este patrón detecta el caso en que quedan menos de 2 horas para el comienzo de un evento de Madrid. Para su implementación se ha hecho uso de un temporizador **timer:interval**, gracias al que realiza una comprobación cada minuto para cada evento de Madrid concreto hasta que se cumpla la condición.

  Teniendo en cuenta el consumo de recursos, en ese marco de 2 horas se podría incluso incrementar el tiempo entre intervalos unos minutos más, dado que un pequeño retraso en el aviso con unos minutos de margen tampoco sería muy significativo para el usuario, y sin embargo ahorra al sistema hacer comprobaciones constantemente en intervalos de menor tiempo hasta que se cumpla la condición. Sin embargo debido a que el proyecto realizado es un trabajo de fin de grado, de cara a la exposición se ha decidido que el tiempo entre intervalos sea de un minuto para facilitar la defensa.

  La condición a evaluar es que el Timestamp del momento en que se realiza la comprobación del patrón tendrá que estar entre los siguientes valores EPOCH:

  - Momento en que quedan 2 horas para el comienzo del evento:

    $$e.EPOCHdtstart - 7200000$$

  - Momento para el que está programado el inicio del evento de Madrid:

    $$e.EPOCHdtstart$$

El evento detectado se inserta en el flujo de eventos para que pueda ser detectado por otros patrones.

Extracto de código 7.9: Definición del patrón "MenosDe2h"

```
@public @Name('MenosDe2h')
INSERT INTO MenosDe2h
select 'MenosDe2h' as eventTypeName, e.id as id,
   e.title as title, e.latitude as latitude, e.
   longitude as longitude, e.dtstart as dtstart,
   current_timestamp() as currentTime
from  pattern [every-distinct (e.id) e=
   EventosMadrid ->  every timer:interval(1 min)
   ]
where current_timestamp() >= (e.EPOCHdtstart -
   7200000) AND current_timestamp() < e.
   EPOCHdtstart
GROUP BY e.id
HAVING COUNT(*) = 1;
```

- **MeGustaMenosDe2h**: Detecta cada vez que un usuario distinto le da me gusta a un evento detectado por el **patrón "MenosDe2h"**, para el que queda menos de 2 horas. Este patrón ha sido implementado de forma que el evento complejo sea detectado por el motor CEP independientemente del orden en que se inserten en el flujo de eventos el evento complejo **"MenosDe2h"** y el evento simple **"me gusta"**, teniendo en cuenta las tres posibles situaciones:

  1. Un usuario da "me gusta" al evento antes de que queden menos de 2 horas para su comienzo y, por tanto, antes de que el motor CEP detecte el evento complejo **"MenosDe2h"**.

  2. Quedan menos de 2 horas para el comienzo del evento, pero al **patrón "MenosDe2h"** le queda aún una comprobación pendiente para ese evento debido al temporizador usado para su implementación, gracias al que realiza una comprobación cada minuto para cada evento de Madrid concreto hasta que se cumpla la condición.

  3. Quedan menos de 2 horas para el comienzo del evento, se detecta el evento complejo **"MenosDe2h"** y tras ello un usuario da "me gusta" al evento.

Extracto de código 7.10: Definición del patrón "MeGustaMenosDe2h"

```
@public @Name('MeGustaMenosDe2h')
INSERT INTO MeGustaMenosDe2h
select 'MeGustaMenosDe2h' as eventTypeName, m.id
   as eventID, m.title as title, mg.userID as
   userID, m.latitude as latitude, m.longitude as
    longitude
from pattern [
```

```
(every m=MenosDe2h -> every-distinct (mg.id) mg=
   MeGusta)
OR
(every-distinct (mg.id) mg=MeGusta -> every m=
   MenosDe2h)
]
where mg.eventID = m.id;
```

- **AforoLimitado**: Este patrón detecta cada evento de Madrid distinto (según su identificador) que llega y que contiene la palabra "aforo" bien en su atributo "price" o bien en su atributo "description" (destacar las diferentes comprobaciones teniendo en cuenta mayúsculas y minúsculas). El evento detectado se inserta en el flujo de eventos para que pueda ser detectado por otros patrones. La idea es detectar eventos que tengan límite de aforo.

Extracto de código 7.11: Definición del patrón "AforoLimitado"

```
@public @Name('AforoLimitado')
INSERT INTO AforoLimitado
select 'AforoLimitado' as eventTypeName, e.id as
   id
from pattern [every-distinct (e.id) e=
   EventosMadrid]
where
e.price like '%aforo%' OR
e.price like '%Aforo%' OR
e.price like '%AFORO%' OR
e.description like '%aforo%' OR
e.description like '%Aforo%' OR
e.description like '%AFORO%';
```

- **AforoLimitado_MeGustaMenosDe24h**: Detecta la situación en que un evento al que un usuario ha dado me gusta y para el que queden menos de 24 horas para su comienzo (evento complejo **"MeGustaMenosDe24h"**), también tenga limitaciones de aforo (evento complejo **"AforoLimitado"**). Esto se consigue detectando la situación en que uno de los dos eventos complejos mencionados llegue detrás de otro, en uno u otro sentido, pues puede ocurrir que los me gusta se recuperen antes que el propio evento al reiniciar la aplicación Mule (ver apartado 7.1.7).

Extracto de código 7.12: Definición del patrón "AforoLimitado_MeGustaMenosDe24h"

```
@public @Name('AforoLimitado_MeGustaMenosDe24h')
select 'AforoLimitado_MeGustaMenosDe24h' as
   eventTypeName, m.eventID as eventID, m.title
   as title, m.userID as userID
```

```
from pattern[
(a=AforoLimitado -> every m=MeGustaMenosDe24h)
OR
(every m=MeGustaMenosDe24h -> a=AforoLimitado)
]
WHERE a.id = m.eventID;
```

- **AforoLimitado_MeGustaMenosDe2h**: Detecta la situación en que un evento al que un usuario ha dado me gusta y para el que queden menos de 2 horas para su comienzo (evento complejo **"MeGustaMenosDe2h"**), también tenga limitaciones de aforo (evento complejo **"AforoLimitado"**). Esto se consigue detectando la situación en que uno de los dos eventos complejos mencionados llegue detrás de otro, en uno u otro sentido, pues puede ocurrir que los me gusta se recuperen antes que el propio evento al reiniciar la aplicación Mule (ver apartado 7.1.7).

Extracto de código 7.13: Definición del patrón "AforoLimitado_MeGustaMenosDe2h"

```
@public @Name('AforoLimitado_MeGustaMenosDe2h')
select 'AforoLimitado_MeGustaMenosDe2h' as
   eventTypeName, m.eventID as eventID, m.title
   as title, m.userID as userID
from pattern[
(every a=AforoLimitado -> every m=
   MeGustaMenosDe2h)
OR
(every m=MeGustaMenosDe2h -> every a=
   AforoLimitado)
]
WHERE a.id = m.eventID;
```

- **MenosDe2hCerca**: Detecta la situación en que un evento (sin necesidad de que el usuario le haya dado me gusta) y para el que queden menos de 2 horas para su comienzo (evento complejo **"MenosDe2h"**), se encuentre a una distancia inferior o igual a 1 kilómetro. Esto se consigue gracias a los datos de ubicación (que para cada usuario estarán llegando de manera constante) representados por medio del tipo de evento simple **"Ubicacion"**, que deberán seguir al tipo de evento **"MenosDe2h"**. Para calcular la distancia que hay desde la ubicación del usuario en un momento dado, se ha implementado la clase java **"Haversine"** (ver apartado 7.1.2.1) a cuyo método deberemos pasar como parámetros los datos de ubicación del usuario (campos "latitude" y "longitude" del evento "Ubicacion") y del evento en cuestión (campos "latitude" y "longitude" del evento "MenosDe2h") al realizar la llamada. El resultado devuelto por este método será esa distancia en kilómetros, realizandose la comprobación de que sea menor a 1 kilómetro.

Extracto de código 7.14: Definición del patrón "MenosDe2hCerca"

```
@public @Name('MenosDe2hCerca')
```

```
select 'MenosDe2hCerca' as eventTypeName,m.title
   as title, u.userID as userID
from pattern [
every-distinct (m.id) m=MenosDe2h -> every-
   distinct (u.userID) u=Ubicacion
 ]
WHERE esper.Haversine.main(u.latitude, u.
   longitude, m.latitude, m.longitude) <= 1;
```

- **ParkingPlazasLibresCerca_MeGustaMenosDe2h**: Detecta las situaciones en las que haya un parking con almenos 1 plaza libre, y que se encuentre a menos de 1 kilómetro de distancia de un evento al que un usuario ha dado me gusta (usando de nuevo la clase java **Haversine** detallada en el apartado 7.1.2.1) y empiece en menos de 2 horas. Esto se consigue gracias a la llegada de eventos simples de tipo **"Parking"**, que deberán seguir a un evento de tipo **"MeGusta-MenosDe2h"**. Los eventos de parking llegan de manera constante cada minuto para que los usuarios cuenten con información actualizada. Para cada identificador de parking, se realiza la comprobación para nuevos datos o eventos de tipo "Parking" cada media hora (Desde el momento en que quedan 2 horas para el evento), detectándose sólo un evento complejo para ese identificador por cada intervalo de tiempo. Se inserta en el flujo de eventos cada evento detectado.

Extracto de código 7.15: Definición del patrón "ParkingPlazasLibresCerca_MeGustaMenosDe2h"

```
@public @Name('
   ParkingPlazasLibresCerca_MeGustaMenosDe2h')
INSERT INTO
   ParkingPlazasLibresCerca_MeGustaMenosDe2h
select
    'ParkingPlazasLibresCerca_MeGustaMenosDe2h'
       as eventTypeName,
    p.id as id,
    p.name as name,
    p.freeParkingSpaces as freeParkingSpaces,
    esper.Haversine.main(m.latitude, m.longitude,
        p.latitude, p.longitude) as distance,
    m.userID as userID,
    m.title as title,
    m.eventID as eventID

from pattern [
every m=MeGustaMenosDe2h -> every p=Parking
]
WHERE esper.Haversine.main(m.latitude, m.
   longitude, p.latitude, p.longitude) <= 1 AND p
   .freeParkingSpaces>=1
group by p.id,m.userID
```

```
        output first every 30 minutes;
```

- **ParkingCompleto**: Detecta la situación en que, habiéndose detectado un parking con plazas libres que estuviera cerca de un eventos que comienza en menos de 2 horas y al que un usuario ha indicado que le gusta (evento complejo de tipo **"ParkingPlazasLibresCerca_MeGustaMenosDe2h"**), le lleguen datos sobre ese mismo parking (evento simple de tipo **"Parking"**) que representen la ausencia de plazas disponibles. La idea es avisar de que, aunque se detectara que existían plazas libres para un parking, este se ha quedado sin plazas disponibles.

Extracto de código 7.16: Definición del patrón "ParkingCompleto"

```
        @public @Name('ParkingCompleto')
        select
            'ParkingCompleto' as eventTypeName,
            pc.name as name,
            pc.freeParkingSpaces as freeParkingSpaces,
            pc.distance as distance,
            pc.userID as userID,
                pc.eventID as eventID

        from pattern [
        every pc=
           ParkingPlazasLibresCerca_MeGustaMenosDe2h -> p
           =Parking
        ]
        WHERE p.freeParkingSpaces=0 AND pc.id=p.id;
```

- **Detectar día de la semana**: Se ha diseñado un patrón para cada día de la semana. Cada patrón detecta su día de la semana cuando se alcanza la hora 00:00 de su respectivo día. El día Domingo es el 0, Lunes es 1, Martes es 2, y así sucesivamente. Vemos como ejemplo el patrón "Martes".

Extracto de código 7.17: Definición del patrón "Martes"

```
        @public @Name('Martes')
        INSERT INTO DayOfWeek
        select "TU" as DayOfWeek
        FROM pattern [
        every timer:at(0, 0, *, *, 2)
        ];
```

- **EventoEnCurso**: Se ha diseñado un patrón que detecta si un evento está en curso, es decir, si un evento ha comenzado pero no ha terminado aún.

Extracto de código 7.18: Definición del patrón "EventoEnCurso"

```
@public @Name('EventoEnCurso')
INSERT INTO EventoEnCurso
select 'EventoEnCurso' as eventTypeName, e.id as
   id, e.title as title, e.dtstart as dtstart, e.
   daysOfWeek as daysOfWeek,
e.EPOCH_excluded_days as EPOCH_excluded_days,
current_timestamp() as currentTime
from pattern [every-distinct (e.id) e=
   EventosMadrid ->  every timer:interval(1 min)
   ]
where e.EPOCHdtstart <= current_timestamp() and e
   .EPOCHdtend > current_timestamp()
GROUP BY e.id
HAVING COUNT(*) = 1;
```

- **MeGustaEventoEnCurso**: Se ha diseñado un patrón que detecta cuando, estando un evento en curso, un usuario le da "me gusta". También detecta la situación en que, un evento al que algún usuario ha dejado un "me gusta", comienza o se detecta que está en curso.

Extracto de código 7.19: Definición del patrón "MeGustaEventoEnCurso"

```
@public @Name('MeGustaEventoEnCurso')
INSERT INTO MeGustaEventoEnCurso
select 'MeGustaEventoEnCurso' as eventTypeName, m.id
   as eventID, m.title as title, mg.userID as userID,
    m.daysOfWeek as daysOfWeek,
m.EPOCH_excluded_days as EPOCH_excluded_days
from pattern [
(every m=EventoEnCurso -> every-distinct (mg.id) mg=
   MeGusta)
OR
(every-distinct (mg.id) mg=MeGusta -> every m=
   EventoEnCurso)
 ]
where mg.eventID = m.id;
```

- **HoyNoSeCelebra**: Se ha diseñado un patrón que detecta cuando, estando un evento en curso y un usuario le da "me gusta", se detecta que el día actual es un día de la semana para el cual ese evento no está programado (el día actual no se encuentra dentro del atributo "daysOfWeek"). Se realiza la comprobación en ambas direcciones por si se detecta antes el día de la semana actual antes que el evento está en curso.

Extracto de código 7.20: Definición del patrón "HoyNoSeCelebra"

```
@public @Name('HoyNoSeCelebra')
```

```
INSERT INTO HoyNoSeCelebra
select 'HoyNoSeCelebra' as eventTypeName, m.eventID
   as eventID, m.title as title, m.userID as userID,
   d.DayOfWeek as DayOfWeek
from pattern
[
(every m=MeGustaEventoEnCurso -> every d=DayOfWeek)
OR
(every d=DayOfWeek-> every m=MeGustaEventoEnCurso)
]
where
m.daysOfWeek IS NOT NULL
AND
NOT m.daysOfWeek LIKE '%' || d.DayOfWeek || '%';
```

Se han implementado tres patrones adicionales con el fin de realizar las Pruebas de rendimiento de la aplicación Mule y que se detallan en el apartado dedicado al Análisis de tiempo de respuesta.

7.1.2.1. Implementación de la clase Java Haversine

La fórmula de **Haversine** (contributors, 2024), o también conocida como fórmula del semiverseno, es la fórmula utilizada para implementar la clase java "Haversine" en nuestra aplicación Mule. Esta fórmula calcula la distancia entre dos puntos de una esfera, dada la longitud y latitud de cada uno de los dos puntos. Para este proyecto se ha implementado en lenguaje Java (Ordoñez, 2024).

Extracto de código 7.21: Implementación de la clase java "Haversine"

```
public class Haversine
{
        @SuppressWarnings("unchecked")
        public static synchronized double main(double
           latitudP1, double longitudP1, double latitudP2,
           double longitudP2)
        {
                final double RADIO_TIERRA = 6371.01;

                latitudP1 = Math.toRadians(latitudP1);
        longitudP1 = Math.toRadians(longitudP1);
        latitudP2 = Math.toRadians(latitudP2);
        longitudP2 = Math.toRadians(longitudP2);


        double distancia = RADIO_TIERRA * Math.acos(Math.sin(
           latitudP1) * Math.sin(latitudP2)
```

```
                + Math.cos(latitudP1) * Math.cos(latitudP2) *
                    Math.cos(longitudP1 - longitudP2));

        return distancia;

        }//fin_main
}//fin_class
```

7.1.3. Despliegue de esquemas y patrones

Para desplegar los diferentes esquemas y patrones se ha hecho uso de la clase java **flujo "AddEventPatternToEsper"** proporcionada por el profesorado en la asignatura impartida en el grado "Ingeniería de Sistemas de Información" , en el que gracias al componente "file" y al componente "File to String" se obtiene el contenido de los ficheros del directorio **C:\AnypointStudioWorkspace\pruebasplitter\src\main\resources\PatronesSinProcesar** y los mueve al directorio **C:\AnypointStudioWorkspace\pruebasplitter\src\main\resources\PatronesProcesados**. No obstante, se ha hecho una pequeña **modificación** sobre cómo se procesan los datos para que pueda procesar todos los patrones escritos en un mismo fichero, y no tengan que estar escritos en un fichero diferente cada uno. En el código original, a la variable "pattern" se le asignaba solo el primero de los elementos del array que representaba el payload entrante. Se ha modificado para que esta variable el array entero, y vaya procesando uno a uno cada elemento que representa a cada patrón (ver código 7.22). Ese contenido en formato String le llega a la **clase java "AddEventPatternToEsperComponent"**, que se ocupa de desplegar patrones y esquemas para el motor CEP(ver extracto de código 7.22).

Extracto de código 7.22: Implementación de la clase Java "AddEventPatternToEsperComponent"

```
public class AddEventPatternToEsperComponent implements
   Callable {

        @Override
        public Object onCall(final MuleEventContext
           eventContext) throws Exception {

                new EsperUtils();

                final Object payload = eventContext.
                   getMessage().getPayload();

                System.out.println("*** Adding new EPL " +
                   payload.toString());
            final  EPStatement[] pattern = EsperUtils.
               deployNewEventPattern(payload.toString().
```

```
            toString()).getStatements();

        for(int i=0; i<pattern.length; i++)
        {
            pattern[i].addListener((newComplexEvents,
               oldComplexEvents, detectedEventPattern,
               epRuntime) -> {
                if (newComplexEvents != null) {
                    String eventPatternName =
                       detectedEventPattern.getName();
                    Object underlying = newComplexEvents
                       [0].getUnderlying();
                    System.out.println("** Complex event
                       '" + eventPatternName + "'
                       detected: " + underlying);

                    //...


                    }//end_if
            });
        }//end_for
            return payload.toString();
    }
}
```

7.1.4. Obtención de datos y envío de eventos simples

7.1.4.1. Eventos de Madrid - obtención y estructura de los datos

Se define el flujo **"GET_EventosMadrid"**. En él, un componente "Poll" se ocupa de recuperar los datos realizando una petición de método GET mediante el protocolo HTTP al correspondiente endpoint en el puerto 80 :

Extracto de código 7.23: Implementación del flujo "GET_EventosMadrid"

```
    <flow name="GET_EventosMadrid">
            <poll doc:name="Poll">
                    <fixed-frequency-scheduler frequency
                       ="300" timeUnit="SECONDS"
                       startDelay="60"/>
                    <http:request config-ref="
                       HTTP_Request_Configuration"
        path="/portal/site/egob/menuitem.
           ac61933d6ee3c31cae77ae7784f1a5a0/?vgnextoid
           =00149033f2201410VgnVCM100000171f5a0aRCRD&amp;
```

```
            format=json&amp;file=0&amp;filename=300107-0-
            agenda-actividades-eventos&amp;mgmtid=57
            be24206a91b510VgnVCM2000001f4a900aRCRD&amp;
            preview=full"
        method="GET" doc:name="HTTP"/>
            </poll>
            <byte-array-to-string-transformer doc:name="
               Byte Array to String"/>
            <component class="esper.FormatoEvento" doc:
               name="FormatoEvento"/>

            <json:json-to-object-transformer returnClass
               ="java.util.List" doc:name="JSON to Object
               /Array"/>


            <foreach collection="#[payload]" doc:name="
               For Each Event" rootMessageVariableName="
               event">
                  <json:object-to-json-transformer doc:
                     name="Object to JSON (Replace '=' 
                     with ':')"/>
                  <component class="esper.InsertDB" doc
                     :name="Inserta Evento Madrid Base
                     de Datos y cola AMQP"/>

            </foreach>
    </flow>
```

La petición se realiza un minuto después de que se ejecute la aplicación Mule (observar configuración del componente Poll), con el fin de que se recojan previamente y se consuman como eventos simples, los eventos culturales que ya se encontraban almacenados en la base de datos gracias al flujo **flujo "InicializacionEventosDB-> Cola"**. El JSON entrante es procesado por la clase java **"FormatoEvento"** (ver extracto de código 7.24). El JSON obtenido está estructurado de tal manera que un subobjeto llamado **"@context"** contendrá información acerca de la estructuración de los datos, mientras que el otro subobjeto **"@graph"** contendrá la información en sí misma. Es este último el que nos interesa, por lo que la clase java se encargará de discriminar el primer objeto y que solo se procese en el flujo el subobjeto "@graph". Además, se encarga de que el caracter "
" no cause problemas no deseados a la hora de procesar los datos.

Extracto de código 7.24: Implementación de la clase java "FormatoEvento"

```
public class FormatoEvento {

    @SuppressWarnings("unchecked")
```

```
        public synchronized String FormatoEvento(@Payload
           String eventReceived)
        {
        try {

        String newEventReceived = eventReceived.replace("\\",
            "\\\\");
                newEventReceived = newEventReceived.replace
                   ("'", "");

        JSONObject myJSONMessage = new JSONObject(
           newEventReceived);
        JSONArray graphArray = myJSONMessage.getJSONArray("
           @graph");

        return graphArray.toString();
        }catch(JSONException e) {
            System.out.println("Error desde FormatoEvento.
               java en el JSON: " + e.getMessage());
            return new String();
        }
        }
    }
```

El JSON resultante se transforma, mediante un componente "json : json-to-object-transformer" en un Array donde cada elemento puede ser iterado mediante un componente "foreach". Para cada elemento ha sido necesario volver a transformarlos de nuevo a su formato JSON original mediante el componente "json : object-to-json-transformer". Finalmente, cada uno de los elementos es procesado por la clase java **"InsertDB"** y enviado a su respectiva cola AMQP e insertado en la respectiva tabla en la base de datos.

En el extracto de código 7.25 podemos observar la implementación de dicha clase java. Esta clase se encarga de dar el formato oportuno a cada uno de los campos del JSON de entrada. Además se encarga de manejar la excepción de tipo "SQLIntegrityConstraintViolationException", en concreto aquella cuyo mensaje de error es "Duplicate entry '[valor de clave primaria]' for key 'PRIMARY'INFO". De este modo, nos aseguramos de que la conexión que se abre para ese dato de entrada se cierre correctamente. Solo se insertarán aquellos valores no repetidos en la base de datos.

Extracto de código 7.25: Implementación de la clase java "InsertDB"

```
    public class InsertDB {
        private Connection connection;
        private String host = //...;
        private final String DB = //...;
        private String user = //...;
        private String password = //...;
```

```
private final String QUEUE_NAME = "EventosMadrid";
private final String EXCHANGE_NAME = "";


@SuppressWarnings("unchecked")
public synchronized void insert(@Payload String
   eventReceived)
{
        JSONObject myJSONMessage = new JSONObject(
           eventReceived);

        try {
                Class.forName("com.mysql.cj.jdbc.
                   Driver");
                connection = DriverManager.
                   getConnection("jdbc:mysql://"+
                   this.host + "/" + this.DB + "?user
                   =" + this.user + "&password=" +
                   this.password);


                //TYPE
                String type ;
                if(myJSONMessage.has("@type")){
                        type = myJSONMessage.
                           getString("@type");
                        String[] CamposURL = type.
                           split("/");
                        type = CamposURL[CamposURL.
                           length - 1];


                }else{
                        type = null;
                }


                //TITLE
                String title = myJSONMessage.
                   getString("title");

                //ID
                int id = myJSONMessage.getInt("id");

                //DESCRIPTION
                String description = myJSONMessage.
```

```
                getString("description") ;

            if(description.isEmpty())
            {
                    description = null;
            }else {
                    description = description.
                       replaceAll("'", "");
                    description = description.
                       replaceAll("\"", "");

                    description = description.
                       replaceAll("`", "");
                    description = description.
                       replaceAll("  ", "");
            }


            //IS_FREE
            int freeInt = myJSONMessage.getInt("
               free");
            boolean is_free;
            if(freeInt == 0)
                    is_free = false;
            else
                    is_free = true;


            //PRICE
            String price = myJSONMessage.
               getString("price");

            if(price.isEmpty())
            {
                    price = null;
            }



            //DTSTART Y DTEND (FORMATOS DATE Y
               EPOCH)
            String Cadenadtstart = myJSONMessage.
               getString("dtstart");
    Timestamp Datedtstart = Timestamp.valueOf(
       Cadenadtstart);

    Long EPOCHdtstart = Datedtstart.getTime();
```

```
String Cadenadtend = myJSONMessage.getString("
   dtend");
Timestamp Datedtend = Timestamp.valueOf(
   Cadenadtend);
Long EPOCHdtend = Datedtend.getTime();


//EXCLUDED DAYS
SimpleDateFormat dateFormatExcludedDay = new
   SimpleDateFormat("dd-MM-yyyy");

String originalExcluded_days = myJSONMessage.
   getString("excluded-days");
String[] array_excluded_days;

if (originalExcluded_days == null ||
   originalExcluded_days.isEmpty())
     array_excluded_days = new String[0];
else
{
    //Hacer un array con cada split(";")
        array_excluded_days =
           originalExcluded_days.split(";");
}


JsonArray excluded_days = new JsonArray();
JsonArray EPOCH_excluded_days = new JsonArray();


for(int j=0; j<array_excluded_days.length; j++)
{
    String StringExcludedDay =
       array_excluded_days[j].replace("/","-");
    excluded_days.add( StringExcludedDay );
    EPOCH_excluded_days.add(
       dateFormatExcludedDay.parse(
       StringExcludedDay).getTime() );
}



//RECURRENCE

//Solo me quedo con el que cambia: days (dias de
```

```
   la semana que se celebra)
String days;
JsonArray json_days;
if(myJSONMessage.has("recurrence")){
    days = myJSONMessage.getJSONObject("
       recurrence").getString("days");

        String[] array_days = days.split(",");
        json_days = new JsonArray();

        for(int i=0; i<array_days.length; i++)
        {
            json_days.add( array_days[i] );
        }

}else {
    json_days = null;
}


//LATITUDE Y LONGITUDE
//Recupero la Localizacion del evento(coordenadas
   ) si es que tiene

Double latitude = null;
Double longitude = null;

if(myJSONMessage.has("location"))
{
    JSONObject location = myJSONMessage.
       optJSONObject("location");


    if (location != null)
    {
            latitude = location.getDouble("
               latitude");
                longitude = location.getDouble("
                   longitude");
    }

}


//INSERCION BASE DE DATOS
Statement statement = connection.createStatement
   ();
```

```
String query = "INSERT INTO prueba(id, event_type
   , title, description, is_free, price, dtstart,
    EPOCHdtstart, dtend, EPOCHdtend,
   excluded_days, EPOCH_excluded_days, days,
   latitude, longitude) VALUES('"+id+"', '"+type
   +"', '"+title+"', '"+description+"', "+freeInt
   +", '"+price+"', '"+Datedtstart+"', "+
   EPOCHdtstart+", '"+Datedtend+"', "+EPOCHdtend
   +", '"+excluded_days+"', '"+
   EPOCH_excluded_days+"', '"+json_days+"', "+
   latitude+", "+longitude+")" ;
statement.executeUpdate(query);
connection.close();



//ENVIO COLA AMQP
JSONObject jsonMessage = new JSONObject();
jsonMessage.put("eventTypeName", QUEUE_NAME);
jsonMessage.put("id", id);
jsonMessage.put("event_type", type);
jsonMessage.put("title", title);
jsonMessage.put("description", description);
//jsonMessage.put("is_free", is_free);
//jsonMessage.put("price", price);
jsonMessage.put("dtstart", Datedtstart);
jsonMessage.put("EPOCHdtstart", EPOCHdtstart);
jsonMessage.put("dtend", Datedtend);
jsonMessage.put("EPOCHdtend", EPOCHdtend);
/*
jsonMessage.put("excluded_days",
   json_excluded_days);
jsonMessage.put("days", json_days);
jsonMessage.put("latitude", latitude);
jsonMessage.put("longitude", longitude);
*/

String message = jsonMessage.toString();




ConnectionFactory factory = new ConnectionFactory
   ();
factory.setHost("localhost");

com.rabbitmq.client.Connection connection =
```

```
            factory.newConnection();
        Channel channel = connection.createChannel();

        channel.queueDeclare(QUEUE_NAME, true, false,
            false, null);



        channel.basicPublish(EXCHANGE_NAME, QUEUE_NAME,
            null, message.getBytes());



        channel.close();
        connection.close();




            }catch(
               SQLIntegrityConstraintViolationException
               e){

        try
        {
            connection.close();
        }catch(SQLException error) {
            System.out.println("Error al cerrar la
               conexion: " + error.getMessage());
        }

            }
            catch(Exception  e){

                    System.out.print("Error= "+ e.
                       getMessage());
                    try
        {
            connection.close();
        }catch(SQLException error) {
            System.out.println("Error al cerrar la
               conexion: " + error.getMessage());
        }

            }
    }
```

Destacamos algunos de los campos del JSON de entrada:

- **type**: originalmente tiene un formato de URL como el del siguiente, en el caso por ejemplo de los eventos de tipo "DanzaBaile" :

  https://datos.madrid.es/egob/kos/actividades/DanzaBaile

  Puesto que la información que interesa es el último campo del enlace, crearemos gracias al método **"split"** de la clase String un array, donde cada elemento corresponderá a uno de los campos del enlace, quedándonos con el último de ellos y almacenándolo en la varibale de tipo String "type".

- **description**: Este atributo puede contener caracteres problemáticos que pueden provocar error en el procesamiento de los datos. La clase se encarga de eliminarlos gracias al método **"replaceAll"** de la clase String.

- **dtstart y dtend**: La fecha la extraemos en formato String y sacamos su valor en formato Timestamp gracias al método **"valueOf"** de la clase Timestamp. Para obtener su valor en milisegundos (formato EPOCH), debemos hacer uso del método **"getTime"**, también de la clase Timestamp. El valor en formato EPOCH nos resultará de utilidad para realizar comparaciones sobre fechas en los patrones definidos (ver apartado 7.1.2).

- **excluded-days**: Los datos del JSON original dados por el atributo tienen el formato que sigue el siguiente ejemplo: "excluded-days": "15/10/2024;19/11/2024;". Gracias al método **"split"** de la clase String construimos un array con cada fecha separada por el caracter ";" del String original (Creamos un array vacío en caso de no existir datos de días excluídos). Creamos dos JsonArray llamados "excluded_days" y "EPOCH_excluded_days" con el fin de ser los datos que insertaremos en nuestrta base de datos y enviaremos a la cola.

  Para que sigan el formato "dd-MM-yyy", se reemplazan los caracteres "/" por "-" de cada elemento del array original, para insertarlo en el primero de los nuevos JsonArray. En el caso del segundo nuevo JsonArray, cada elemento String del array original será parseado a Timestamp según el formato mencionado para obtener su valor en milisegundos o formato EPOCH, y almacenarlo en este segundo nuevo JsonArray. La razón de que se haya escogido ese formato de fecha es que es el formato seguido por el sistema de gestión de bases de datos utilizado **MariaDB** para el tipo **Datetime** (MariaDB Corporation, 2024a).

- **recurrence**: El formato de este atributo es el seguido por el siguiente ejemplo:

```
"recurrence": {
    "days": "TU,WE,TH,FR,SA,SU",
    "frequency": "WEEKLY",
    "interval": 1
}
```

  Es trivial concluir que la frecuencia del evento dados uno o varios días de la semana, es semanal. Por tanto nos quedaremos con el campo del subobjeto que nos interesa, en este caso el atributo "days". Extreaemos la cadena del payload, y gracias al método **"split"** de la clase String obtenemos un array de tipo String.

Finalmente, definimos un JsonArray e iremos añadiendo elemento a elemento cada uno de los días de la semana excluídos, contenidos en el array de tipo String.

- **location**: Este atributo del JSON original contiene la latitud y la longitud de la ubicación en que tendrá lugar el evento. El formato que sigue es el que presenta el siguiente ejemplo:

```
        "location": {
            "latitude": 40.413896825498995,
            "longitude": -3.7054692169323364
                }
```

Se obtiene el JSONObject por medio del método **"optJSONObject"** de la clase JSONObject, y se almacenan sus dos atributos en las variables de tipo Double "latitude" y "longitude".

7.1.4.2. Información de parking - obtención y estructura de los datos

Se define el flujo **"GET_DatosParking"**. En él, un componente "Poll" se ocupa de recuperar los datos realizando una petición de método GET mediante el protocolo HTTPS al correspondiente endpoint en el puerto 443 :

Extracto de código 7.26: Implementación del flujo "GET_DatosParking"

```
<flow name="GET_DatosParking">
            <poll doc:name="Poll">
                    <fixed-frequency-scheduler frequency
                       ="60" timeUnit="SECONDS"
                       startDelay="60"/>
                    <http:request config-ref="
                       HTTPS_Request_Configuration_Parking
                       "
        path="/MTPAR_RSINFO/restInfoParking/listParking?
           language=ES"
        method="GET" doc:name="HTTPS"
        />
            </poll>
            <byte-array-to-string-transformer doc:name="
               Byte Array to String"/>
            <json:json-to-object-transformer returnClass
               ="java.util.List" doc:name="JSON to Object
               /Array"/>



            <foreach collection="#[payload]" doc:name="
               For Each Event" rootMessageVariableName="
```

```
                parking">
                    <json:object-to-json-transformer doc:
                        name="Object to JSON (Replace '=' 
                        with ':')"/>
                    <logger message="#[payload]" level="
                        INFO" doc:name="Logger"/>
                    <component class="esper.
                        JsonToParkingQueue" doc:name="
                        JsonToParkingQueueAndDB"/>

            </foreach>


    </flow>
```

Se realiza la petición cada 60 segundos, es decir, cada minuto. Hay datos cuyos valores tardan como máximo no más de 10 minutos en ser actualizados y estar disponibles desde el recurso. El JSON entrante se transforma, mediante un componente "json : json-to-object-transformer" en un Array donde cada elemento puede ser iterado mediante un componente "foreach". Para cada elemento ha sido necesario volver a transformarlos de nuevo a su formato JSON original mediante el componente "json : object-to-json-transformer". Finalmente, cada uno de los elementos es procesado por la clase java **"JsonToParkingQueue"** y enviado a su respectiva cola AMQP e insertado en la respectiva tabla en la base de datos.

En el extracto de código 7.27 podemos observar la implementación de dicha clase java. Observar cómo se eliminan del JSON original algunos atributos que no nos resultan de utilidad gracias al método "remove" de JSONObject. Mandamos a cola solo aquellos datos de parking que sí cuentan con información sobre las plazas disponibles en su atributo **"occupations"**, pues son los datos que interesan y que serán enviados al motor CEP como eventos simples. De igual manera dicho atributo será eliminado y reemplazado por un nuevo atributo **"freeParkingSpaces"** que contendrá únicamente la información necesaria. El formato tomado por el atributo **"occupations"** es el que sigue el ejemplo:

```
    "occupations":[{"code":"001","free":271,"moment
       ":"2024-11-27T14:08:54.000+01:00","name":"Total","
       renewalIndex":"0.00"}]
```

Del subobjeto "occupations" también querremos extraer el campo "moment", cuya información viene dada siguiendo el formato del siguiente ejemplo : "moment":"2024-11-26T15:31:53.000+01:00".

Lo que hacemos es extraer en formato String el valor de ese atributo "moment", y parsearlo a un tipo "Date" con el formato "yyyy-MM-dd'T'HH:mm:ss.SSSXXX". A continuación, cambiaremos su formato a otro nuevo: "yyyy-MM-dd HH:mm:ss.S". El resultado será almacenado en la variable de tipo String "stringFormattedMoment", que se insertará directamente en el JSON como valor de un nuevo atributo **"moment"**

independiente del subobjeto “occupations”, como ocurre con el nuevo atributo “freeParkingSpaces” introducido anteriormente. No obstante, necesitamos que la fecha se encuentre en formato Date para su posterior inserción en la base de datos, y para ello hacemos uso del método “valueOf” de “Timestamp”, al que pasamos como parámetro la variable String “stringFormattedMoment”, y cuyo resultado almacenamos en la variable de tipo Date “dateFormattedMoment”.

Explicamos en detalle la inserción de mensajes en una cola AMQP en el apartado 7.1.7 en lo referente al extracto de código 7.33. Para la inserción en la base de datos debemos tener en cuenta que las inserciones se realizan haciendo uso de la cláusula **ON DUPLICATE KEY UPDATE** en la consulta, que permitirá actualizar un registro existente sin provocar errores por tener una clave duplicada.

Extracto de código 7.27: Implementación de la clase java “JsonToParkingQueue”

```
public class JsonToParkingQueue {

        //declaracion variables privadas conexion DB
    //...

        private final String QUEUE_NAME = "Parking";
        private final String EXCHANGE_NAME = "";


        @SuppressWarnings("unchecked")
        public synchronized void insert(@Payload String
           eventReceived)
        {
        //Declaro varibales fuera del try para referenciarlas
            //desde el bloque cath

        //Para meter en la base de datos aun siendo null
        Integer freeParkingSpaces = null ;
        Timestamp dateFormattedMoment = null;
        String stringFormattedMoment = null;


                JSONObject myJSONMessage = new JSONObject(
                   eventReceived);
                String query;

                Statement statement = null;

                try {


        myJSONMessage.put("eventTypeName", QUEUE_NAME);
```

```
double latitude = Double.parseDouble( myJSONMessage.
   getString("latitude") ) ;
double longitude = Double.parseDouble( myJSONMessage.
   getString("longitude") ) ;

myJSONMessage.put("latitude", latitude);
myJSONMessage.put("longitude", longitude);


myJSONMessage.remove("family");
myJSONMessage.remove("category");
myJSONMessage.remove("state");
myJSONMessage.remove("type");
myJSONMessage.remove("administrativeArea");
myJSONMessage.remove("familyCode");
myJSONMessage.remove("town");
myJSONMessage.remove("country");




//Solo mando a la cola aquellos donde haya datos //
   sobre plazas libres

if(myJSONMessage.has("occupations")) {


    JSONObject occupation = myJSONMessage.
       getJSONArray("occupations").getJSONObject(0);


    freeParkingSpaces = occupation.getInt("free");

    myJSONMessage.put("freeParkingSpaces",
       freeParkingSpaces);



    SimpleDateFormat originalDateFormat = new
       SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ss.SSSXXX
       ");
    SimpleDateFormat dateFormat = new
       SimpleDateFormat("yyyy-MM-dd HH:mm:ss.S");


    String stringOriginalMoment = occupation.
```

```
   getString("moment");


Date dateOriginalMoment = originalDateFormat.
   parse(stringOriginalMoment);

stringFormattedMoment = dateFormat.format(
   dateOriginalMoment);

//Para insertar en la Base de Datos
dateFormattedMoment = Timestamp.valueOf(
   stringFormattedMoment);

myJSONMessage.put("moment", stringFormattedMoment
   );

myJSONMessage.remove("occupations");

String message = myJSONMessage.toString();


    //ENVIO COLA AMQP

ConnectionFactory factory = new ConnectionFactory
   ();
factory.setHost("localhost");

com.rabbitmq.client.Connection connection =
   factory.newConnection();
Channel channel = connection.createChannel();

//Parametros -> channel.queueDeclare(QUEUE_NAME,
   Durable, Exclusive, AutoDelete, Arguments);
channel.queueDeclare(QUEUE_NAME, true, false,
   false, null);



channel.basicPublish(EXCHANGE_NAME, QUEUE_NAME,
   null, message.getBytes());



channel.close();
connection.close();
```

```
}//end_if


   //   INSERCION/ACTUALIZACION BASE DE DATOS

Class.forName("com.mysql.cj.jdbc.Driver");
        connection = DriverManager.getConnection("
           jdbc:mysql://"+ this.host + "/" + this.DB
           + "?user=" + this.user + "&password=" +
           this.password);



        statement = connection.createStatement();

        if(dateFormattedMoment != null)
                query = "INSERT INTO parking(
    //...
    , latitud, longitud, plazasLibres) VALUES(
    //...
    , '"+dateFormattedMoment+"', "+latitude+", "+
       longitude+", "+freeParkingSpaces+") ON
       DUPLICATE KEY UPDATE momento = '"+
       dateFormattedMoment+"', plazasLibres = "+
       freeParkingSpaces+" " ;
        else
                query = "INSERT INTO parking(
    //...
    , momento, latitud, longitud, plazasLibres)
       VALUES(
    //...
    , "+null+", "+latitude+", "+longitude+", "+
       freeParkingSpaces+") ON DUPLICATE KEY UPDATE
       momento = "+null+", plazasLibres =  "+
       freeParkingSpaces+" " ;



        statement.executeUpdate(query);



  connection.close();



        }//end_try
```

```
        //...


        }
}
```

7.1.5. Envío de eventos simples al motor CEP

El envío de eventos simples al motor CEP se realiza gracias a la implementación de la clase java **"SendEventToEsperComponent"**, cuya implementación puede observarse en el extracto de código 7.28.

Extracto de código 7.28: Implementación del método "sendEventToEsper" de la clase java de mismo nombre

```
public class SendEventToEsperComponent {
        @SuppressWarnings("unchecked")
        public synchronized void sendEventToEsper(@Payload
           String eventReceived) {
        JSONObject myJSONMessage = new JSONObject(
           eventReceived);
        if (myJSONMessage.has("eventTypeName")){
            String eventTypeName = myJSONMessage.getString("
               eventTypeName");
            if (eventTypeName!=null) {
                System.out.println ("TIPO: "+ eventTypeName);
                //En la version original de los profesores se
                    enviaba eventReceived
                EsperUtils.sendEventTyped(eventReceived,
                   eventTypeName);
            }
        }
        }
}
```

Se define un flujo para cada tipo de evento simple definido en el apartado dedicado a la Definición de esquemas. Poniendo de ejemplo el código XML de uno de estos flujos (ver extracto de código 7.29), vemos que la aplicación mule obtiene la información de la respectiva cola de mensajes de RabbitMQ, transforma los datos al tipo String y hace uso de la clase mencionada anteriormente **"SendEventToEsperComponent"** para hacer llegar los eventos simples al motor CEP.

Extracto de código 7.29: Implementación del flujo "EventosSimples-EventosMadrid"

```
<flow name="EventosSimples-EventosMadrid">
```

```
    <amqp:inbound-endpoint exchangeName="default" queueName="
       EventosMadrid" responseTimeout="10000"  connector-ref
       ="AMQP_0_9_Connector" doc:name="AMQP-0-9" exchangeType
       ="fanout"/>
    <byte-array-to-string-transformer doc:name="Byte Array to
        String"/>
    <component class="esper.SendEventToEsperComponent" doc:
       name="SendEventToEsperComponent"/>
</flow>
```

7.1.6. Procesamiento de eventos complejos y notificaciones

Se implementa mediante el **flujo "CEP"**, que mediante un componente **choice** es capaz de filtrar los eventos complejos salientes del componente **vm** según el tipo de evento complejo que se ha detectado. Con el fin de notificar al usuario, se han desarrollado dos clases java.

7.1.6.1. Clase Java "SendEmail"

Explicamos en este apartado la implementación de esta clase java. Formateamos el payload recibido y nos quedamos con el JSON interno, eliminando las llaves que lo contienen. Creamos un objeto JSONObject con dicho valor resultante tras haberlo dejado en un formato apropiado para un JSON (por ello reemplazamos los "=" por ":"). En la variable "emailTo" almacenaremos el valor del atributo "email" del JSON, que será la dirección de correo a la que se enviará el mensaje. En la variable "emailFrom" almacenamos la dirección de correo desde la que se envía el mensaje. En este caso y al ser la naturaleza del proyecto la de un trabajo de fin de grado, se ha usado la dirección de correo electrónica propia de la universidad. La variable "passwordFrom" almacenará una contraseña de aplicación generada desde la cuenta de Google que permitirá posteriormente la autenticación para el envío del correo.

Si vamos al apartado Definición de patrones podemos ver que se implementan tres patrones cuya finalidad es detectar situaciones de interés que se notifican vía email al usuario correspondiente:

- **DetectaInicioSesionFallido**
- **DetectaFuerzaBruta**
- **UbicacionSospechosa**

Filtramos el JSON del evento complejo recibido según su atributo "eventTypeName", y dependiendo de su valor le daremos a las variables "subject" y "content" (almacenarán el valor del asunto y el mensaje respectivamente) un valor u otro.

Observamos el extracto de código 7.30. Para configurar el protocolo smtp (simple mail transfer protocol) definimos las variables "mProperties" de tipo Properties,

"mSession" de tipo Session (de la librería javax.mail) y "mCorreo" de tipo MimeMessage (de la librería javax.mail.internet). La configuración vendrá dada por las propiedades que añadimos a la varibale "mProperties".

Damos valor a nuestra variable de sesión "mSession" creando una instancia de sesión con las propiedades contenidas en la variable "mProperties". Definimos el cuerpo del correo con la variable "mCorreo", dándole el valor de un objeto de tipo "MimeMessage" inicializado con el valor de nuestra sesión "mSession". Tras ello, utilizamos las variables definidas al principio del código para definir dirección de correo que emite el mensaje con el método "setFrom", la destinataria con "setRecipient", el asunto con "setSubject" y el cuerpo del mensaje con "setText".

Por último, para que se produzca el envío del correo definimos una variable "mTransport" de tipo Transport, que se encargará dada nuestra sesión, se enviar el correo según el protocolo especificado (como hemos dicho anteriormente smtp). Establecemos la conexión con el método "connect" pasando como parámetro las credenciales definidas de la dirección de correo desde la que se enviará el mensaje. Enviamos el mensaje con el método "send", al que pasamos como parámetro el correo que definimos en nuestra variable "mCorreo", y la dirección destinataria recogida de la misma variable por medio del método "getRecipients". Hecho esto, cerramos la conexión.

Extracto de código 7.30: Configuración protocolo smtp de la clase java "SendEmail"

```
public class SendEmail
{

    @SuppressWarnings("unchecked")
    public synchronized void main(@Payload String
       eventReceived)
    {

       //...
        Properties mProperties = new Properties();
        System.out.println("emailTo: " + emailTo);

        mProperties.put("mail.smtp.host", "smtp.gmail.com
           ");
        mProperties.put("mail.smtp.ssl.trust", "smtp.
           gmail.com");
        mProperties.setProperty("mail.smtp.starttls.
           enable", "true");
        mProperties.setProperty("mail.smtp.port", "587");
        mProperties.setProperty("mail.smtp.user",
           emailFrom);
        mProperties.setProperty("mail.smtp.ssl.protocols
           ", "TLSv1.2");
        mProperties.setProperty("mail.smtp.auth", "true")
           ;
```

```
        Session mSession = Session.getDefaultInstance(
           mProperties,
                new javax.mail.Authenticator(){
                    protected PasswordAuthentication
                       getPasswordAuthentication() {
                        return new PasswordAuthentication
                           (
                                emailFrom, passwordFrom);
                    }
                }
        );


        MimeMessage mCorreo = new MimeMessage(mSession);
        try {
            mCorreo.setFrom(new InternetAddress(emailFrom
               ));
            mCorreo.setRecipient(Message.RecipientType.TO
               , new InternetAddress(emailTo));
            mCorreo.setSubject(subject);
            mCorreo.setText(content, "ISO-8859-1", "html
               ");


        } catch (AddressException e) {
            e.printStackTrace();
        } catch (MessagingException e) {
            e.printStackTrace();
        }


        try {
            Transport mTransport = mSession.getTransport
               ("smtp");
            mTransport.connect(emailFrom, passwordFrom);
            mTransport.send(mCorreo, mCorreo.
               getRecipients(Message.RecipientType.TO));
            mTransport.close();

        } catch (NoSuchProviderException e) {
            e.printStackTrace();
        } catch (MessagingException e) {
            e.printStackTrace();
        }

    }//fin_main
```

```
    }
```

7.1.6.2. Clase Java "SendExpoNotification"

El código extrae del payload que llega como parámetro el userID del usuario al que se va a notificar, y se ejecuta una consulta que devolverá las sesiones activas almacenadas para ese usuario en la base de datos en orden descendiente respecto a la fecha de creación de la sesión. Haremos que la variable "expoPushToken" almacene el token de la última de las sesiones activas del usuario.

Definimos el **cuerpo de la solicitud** para enviar la notificación push en la variable **"PostParams"**, al que le añadimos unos parámetros **"title"** y **"body"** con unos valores u otros dependiendo del tipo de evento complejo que se haya procesado, es decir, filtrando los eventos según su atributo **"eventTypeName"** mediante un **switch**.

Para cada uno de sus casos, se comprobará si en el momento de ejecución del código existe un "me gusta" en la base de datos para el evento sobre el que se va a notificar mediante una consulta al endpoint definido en el back-end **`http://localhost:8080/intereses/checkIfExistsInteres/userID/eventID`**. Dicha consulta se lleva a cabo mediante la implementación del método **"fetchCheckIfExistsInteres"**. Se guardará el resultado booleano de la consulta que devuelve el método en una variable **"notificar"**. En base a su valor, se llevará a cabo la ejecución de un return para acabar con la ejecución del método principal de la clase o bien del código que seguimos explicando a continuación. Esto se hace con la intención de evitar notificar sobre eventos que alguna vez recibieron un "me gusta" por parte de un usuario que, posteriormente, eliminó ese "me gusta".

Otros parámetros de la solicitud post a considerar son:

- **"to"**: se le asigna el expoPushToken del dispositivo al que debe enviarse la notificación.
- **"sound"**: establece el sonido de la notificación. En nuestro caso hemos asignado el valor por defecto.
- **"data"**: se le asigna información adicional, aunque en nuestro caso no se ha llevado a cabo, poniendo únicamente valores de ejemplo.

Podemos ver parte de la implementación de esta clase java en el extracto de código 7.31, donde se realiza la solicitud de tipo Post. En caso de que la consula retorne un valor true, se abre una conexión que devuelve un objeto de tipo **HttpURLConnection**, mediante cuyos métodos **"setRequestMethod"** y **"setRequestProperty"** podremos asignar el método de la solicitud como POST y sus propiedades(Content-Type y Accept) respectivamente. Para esa conexción abrimos un flujo de salida gracias al método **"getOutputStream()"**, y se envía en bytes el cuerpo de la solicitud para que esta se lleve a cabo.

Extracto de código 7.31: Post request en la implementación de la clase java "SendExpoNotification"

```
public class SendExpoNotification
{
    //...

        @SuppressWarnings("unchecked")
        public synchronized void main(@Payload String
           eventReceived)
        {

                //...

            URL url = new URL("https://exp.host/--/api/v2/
               push/send");
            HttpURLConnection conn = (HttpURLConnection) url.
               openConnection();

            //Configuracion de la conexion
            conn.setRequestMethod("POST");
            conn.setRequestProperty("Content-Type", "
               application/json");
            conn.setRequestProperty("Accept", "application/
               json");
            conn.setDoOutput(true);


            String stringPostParams = PostParams.toString();

            System.out.println("Parametros Peticion POST: " +
                stringPostParams);


            // Escribir el cuerpo en la solicitud
            try (OutputStream os = conn.getOutputStream()) {
                byte[] input = stringPostParams.getBytes(
                   StandardCharsets.UTF_8);
                os.write(input, 0, input.length);
            }

            int responseCode = conn.getResponseCode();
            System.out.println("Response Code: " +
               responseCode);

            // Manejar la respuesta...
        } catch (Exception e) {
```

```
            e.printStackTrace();
        }


        }//fin_main

}//fin_class
```

7.1.7. Recuperación de eventos simples frente a apagados o reinicios

Al ser ejecutada la Aplicación Mule, y al ser los eventos simples procesados por el motor CEP, estos desaparecen de la cola. Se ha implementado por ello el **flujo "InicializacionEventosDB-> Cola"** con la idea de poder recuperar los eventos simples que ya han sido procesados por el motor CEP de la base de datos, y poder volver a procesarlos cuando la Aplicación Mule se apague y vuelva a ejecutarse en otro momento. Esto se hace ya que los eventos simples que representan la celebración de un evento y los que representan los me gusta que los usuarios dan a los eventos deben de persistir de alguna manera para poder detectar situaciones de interés para los usuarios mediante el motor CEP. En el extracto de código 7.32, se muestra el código XML con la implementación del flujo, que utiliza un **componente quartz** que iniciará la ejecución del flujo un segundo tras la ejecución de la aplicación Mule (se ha asignado el valor 0 a la propiedad "repeatCount" para que el flujo se ejecute una única vez).

Extracto de código 7.32: Implementación del flujo "InicializacionEventosDB-Cola"

```
    <flow name="InicializacionEventosDB-Cola">
                <quartz:inbound-endpoint jobName="
                   singleExecutionJob"
                             repeatInterval="1000"
                             repeatCount="0"
                             responseTimeout="10000"
                             doc:name="Quartz" connector-ref
                                ="Quartz" startDelay="30000">
                          <quartz:event-generator-job/>
        </quartz:inbound-endpoint>
                <component class="esper.SelectDB" doc:name="
                   Select Eventos Guardados Previamente en la
                    Base de Datos"/>
                <component class="esper.SelectMeGustaDB" doc:
                   name="Select Me Gusta Guardados
                   Previamente en la Base de Datos"/>

        </flow>
```

En el extracto de código 7.33 vemos cómo implementamos una de las dos **clases java**, en este caso **"SelectDB"**, para poder recuperar los eventos de Madrid de la base de datos y mandarlos a su respectiva cola AMQP. Creamos una conexión con la base de datos y ejecutamos la consulta definida en la variable "query", y para cada registro vamos a crear un JSONObject en el que iremos metiendo todos los campos recuperados del registro.
Tras ello, crearemos una conexión **com.rabbitmq.client.Connection** creando un objeto **ConnectionFactory** y ejecutando su método **"newConnection()"**, y un canal Channel resultado de ejecutar el método **"createChannel()"** de la conexción creada. Mediante este canal podremos enviar cada evento como mensaje a la cola, la aplicación Mule podrá recuperarlos y el motor CEP los volverá a procesar como eventos simples.

Extracto de código 7.33: Implementación de la clase java"SelectDB"

```
public class SelectDB {

        private Connection connection;
        private String host = //... ;
        private final String DB = //... ;
        private String user = //... ;
        private String password = //... ;

        private final String QUEUE_NAME = "EventosMadrid";
        private final String EXCHANGE_NAME = "";


        @SuppressWarnings("unchecked")
        public synchronized void main() {

                try {

                        Class.forName("com.mysql.cj.jdbc.
                           Driver");
                        connection = DriverManager.
                           getConnection("jdbc:mysql://"+
                           this.host + "/" + this.DB + "?user
                           =" + this.user + "&password=" +
                           this.password);


                        Statement statement = connection.
                           createStatement();
                        String query = "SELECT * FROM eventos
                           " ;

                        ResultSet rs = statement.executeQuery
                           (query);
```

```
                while (rs.next())
                {
                JSONObject jsonMessage = new
                   JSONObject();
            jsonMessage.put("eventTypeName",
               QUEUE_NAME);
            jsonMessage.put("id", rs.getInt("id"));
            jsonMessage.put("event_type", rs.
               getString("event_type"));
            jsonMessage.put("title", rs.getString("
               title"));
            jsonMessage.put("description", rs.
               getString("description"));


            jsonMessage.put("dtstart", rs.
               getTimestamp("dtstart"));
            jsonMessage.put("EPOCHdtstart", ((Long)rs
               .getLong("EPOCHdtstart")));


            //...



            String message = jsonMessage.toString();


            ConnectionFactory factory = new
               ConnectionFactory();
            factory.setHost("localhost");

            com.rabbitmq.client.Connection connection
                = factory.newConnection();
            Channel channel = connection.
               createChannel();

        channel.queueDeclare(QUEUE_NAME, Durable,
           Exclusive, AutoDelete, Arguments);
            channel.queueDeclare(QUEUE_NAME, true,
               false, false, null);

            channel.basicPublish(EXCHANGE_NAME,
               QUEUE_NAME, null, message.getBytes());

            channel.close();
            connection.close();
```

```
                    }//Fin_While
                      connection.close();
                }
        catch(Exception  e){
                System.out.print("Error= "+ e.getMessage());
                        try
            {
                connection.close();
            }catch(SQLException error) {
                System.out.println("Error al cerrar la
                   conexion: " + error.getMessage());
            }

        }

        }//fin_main


}
```

7.2. Aplicación Android

7.2.1. Distribución de ficheros

Para el desarrollo de la Aplicación se ha desarrollado por un lado el front-end, que es la parte visual e interactiva de la aplicación; y por otro lado el back-end, cuyo propósito es implementar y ofrecer servicios de conectividad y procesamiento de datos.

En las figuras 7.1 y 7.2 vemos como los ficheros se distribuyen en dos directorios principales: un directorio para el front-end y otro para el backend respectivamente.

7.2.2. Front-end

Explicamos en este apartado algunos de los detalles más relevantes de cada una de las pantallas de

7.2.2.1. Navegación

La funcionalidad que permite al usuario navegar entre pantallas dentro de la Aplicación viene dada por la implementación del componente Navigation en el fichero Navigation.jsx.

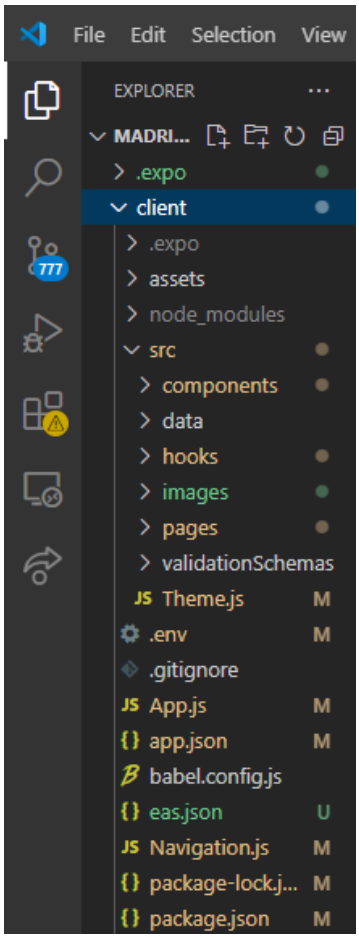


Figura 7.1: Distribución de ficheros para el front-end

La navegación principal de la Aplicación toma forma gracias al uso de "Bottom Tab Navigator"(Meta Platforms, 2024a) y de "Stack Navigator"(Meta Platforms, 2024d). Con ellos creamos distintos componentes de navegación.

El componente "Navigation" utiliza un renderizado condicional(ver extracto de código 7.34) para devolver un componente de navegación u otro según si el usuario ha iniciado sesión o no (damos más detalles sobre la autenticación del usuario en el apartado 7.2.2.2). Estos componentes de navegación que se renderizarán son "MyTabScreens" y "MyStackScreens". Este último implementa la navegación entre las pantallas de inicio de sesión y registro, mientras que el otro componente de navegación se ocupa de implementar la navegación entre el resto de pantallas de la aplicación, tomando la forma de una barra que permite al usuario visualizar las principales pantallas a las que puede navegar.

Extracto de código 7.34: Renderizado condicional en el componente "Navigation"

```
const Navigation = () => {

    const {user} = useContext(AuthenticatedUserContext)

    return (
```

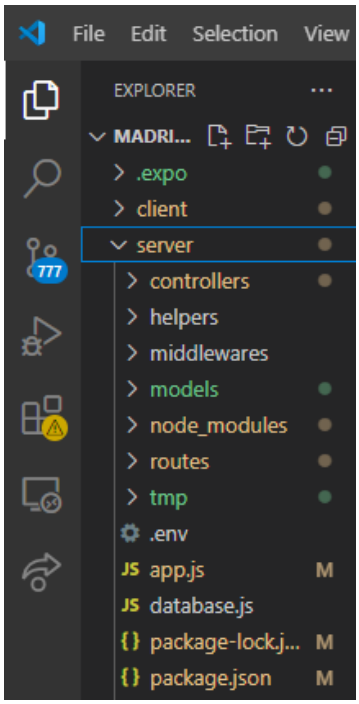


Figura 7.2: Distribución de ficheros para el back-end

```
        <NavigationContainer>
            {user ? (
                    <>
                    <MyTabScreens/>
                    </>
                    ):  <MyStackScreens/> }
        </NavigationContainer>
    )
}
```

Cabe decir que profundizando en la estructura de la implementación del sistema de navegación de la Aplicación, esta es más compleja ya que se hace uso de varios componentes de tipo "Stack Navigator" dentro del propio componente "Bottom Tab Navigator" para lograr dar la funcionalidad que la aplicación requiere y poder navegar desde las pantallas principales a las que se puede acceder desde este este a otras pantallas. En los extractos de código 7.35 y 7.36 vemos cómo uno de los componentes entre los que se navega dentro de "MyTabScreens" es un componente de tipo "Stack Navigator".

Extracto de código 7.35: Componente "MyTabsScreens"

```
const MyTabScreens = () => {
    return
    (
        <Tab.Navigator
            initialRouteName='Main'
```

```
            screenOptions={{
                headerShown:false,
                tabBarActiveTintColor:theme.colors.
                   textSecondary,
                tabBarActiveBackgroundColor:theme.colors.
                   secondary
            }}
        >
            <Tab.Screen name='Main'
             component={EventosStackScreens}
             options={{
                tabBarIcon : ({color, size}) =>
                (
                    <FontAwesome name="list" size={24} color
                       ={theme.colors.textSecondary} />
                )
             }}
            />

            //...
        </Tab.Navigator>
    )
}//end_MyTabScreens
```

Extracto de código 7.36: Componente "EventosStackScreens"

```
const EventosStackScreens = () => {
    return (
        <Stack.Navigator initialRouteName="MainEventos">

            <Stack.Screen name="MainEventos" component={Main}
                 options={{
            title: 'Explorador de Eventos',
            headerTitleStyle: styles.header,
            }} />

            <Stack.Screen name="ListaEventos" component={
               ListaEventos}
            options={({ navigation }) => ({
            title: 'Lista de Eventos',
            headerLeft: () => (
                <TouchableOpacity onPress={() => navigation.
                   goBack()} style={{ paddingLeft: 10 }}>
                    <FontAwesome name="arrow-left" size={24}
                       color={theme.colors.textSecondary} />
                </TouchableOpacity>
            ),
```

```
            headerTitleStyle: styles.header,

            })}
            />
        </Stack.Navigator>
    )
}
```

Para la parte visual del componente de navegación de tipo "Bottom Tab Navigator", se hace uso de iconos de la librería "FontAwesome" del paquete "@expo/vector-icons".

7.2.2.2. Autenticación: contextos, validaciones y notificaciones push

Para la implementación del formulario de inicio de sesión y de registro se ha usado la biblioteca "Formik" (Palmer, 2024), con la que se evita tener que usar para cada campo una variable de estado creando el objeto "initialValues". Creamos un componente "FormikInputValue" con el que nos ahorramos trabajar con handleChange y con values, y que podemos usar en este y cualquier otro formulario. Este componente recibe el nombre del input así como el resto de props. Se hace uso del hook "useField" de Formik, que recibe ese nombre como parámetro y devuelve los siguientes elementos:

- field: Contiene el valor del input. Accedemos al mismo mediante field.value.
- meta: Contiene el error de la validación, si es que existe. Accedemos al mismo mediante meta.error.
- helpers: Con el que por medio de la función "setValue", actualizamos el valor del input.

Más adelante veremos que se hace una petición POST al endpoint para iniciar sesión. Se ha implementado en este componente un condicional que, en caso de existir un error de validación, se mostrará el mensaje bajo el input.

Se gestiona mediante "validationSchema" la validación de los campos, implementada en el componente "LogInValidationSchema" del fichero "LogInValidationSchema.jsx" situado en el directorio "validationSchemas" mediante el uso de la biblioteca de validación de esquemas "Yup" (Quense, 2024). En el extracto de código 7.37 vemos como mediante la función "shape" se definen las propiedades del esquema de validación y las condiciones que cada campo del formulario debe cumplir para el caso del formulario de inicio de sesión.

Extracto de código 7.37: Uso de "Yup" para definir propiedades del esquema de validación del formulario de inicio de sesión en el fichero "LogInValidationSchema.jsx"

```
import * as yup from 'yup';

export const LogInValidationSchema = yup.object().shape({
    email: yup
```

```
    .string()
    .email("Introduce un E-mail correcto")
    .required(),

    password : yup
    .string()
    .required()
})
```

Para implementar el sistema de autenticación de la aplicación móvil se ha hecho uso de la función "createContext" de React para definir el contexto "AuthenticatedUserContext", cuyo propósito es permitir recuperar desde cualquier parte de la aplicación definida dentro del provider del contexto creado información acerca del usuario.

El componente "AuthenticatedUserProvider" renderiza el provider del contexto creado, y permite definir los elementos que serán accesibles mediante el uso del provider para todo componente definido dentro del mismo. De esta manera, envolviendo dentro de él al componente "Navigation" del que hablamos en el apartado 7.2.2.1, permitimos a todo componente de la aplicación para el que se haya implementado sistema de navegación acceder a los datos de usuario.

En el extracto de código 7.38 vemos cómo "LogIn", componente para el que se ha implementado navegación en el componente "Navigation", recupera datos del contexto. En especial se da acceso al dato "expoPushToken", que se envía en el cuerpo de la petición POST a uno de los endpoints que implementamos en el back-end. Observando la implementación de la función "handleLogin" vemos cómo se actualiza el valor para "user", un estado de React Native donde se almacenan los datos de usuario y que también se comparte entre los componentes definidos dentro del provider. Este valor se volverá null en el momento en que el usuario cierre sesión, pulsando el botón de la pantalla "Perfil" para tal propósito (ver figura 7.3) que ejecutará la función "handleLogout" del componente Perfil definido en el fichero "Perfil.jsx", donde se realiza el proceso inverso a la función de inicio de sesión. Observamos también la variable de estado "invalidLogIn", en la que se almacenará el error devuelto en caso de un fallo en la petición POST y cuyo valor se usa para mostrarse justo antes del botón "ACCEDER".

Extracto de código 7.38: Petición POST en la función "HandleLogin" en el fichero "LogIn.jsx"

```
export default function LogIn({navigation})
{
    const [invalidLogIn, setInvalidLogIn] = useState("")
    const {setUser, expoPushToken} = useContext(
       AuthenticatedUserContext)

    const handleLogin = async (values, setInvalidLogIn) => {


        try{
```

```
            const url = `http://${process.env.IP_SERVER
               }:8080/auth/login`
            console.log(url)


            const response = await globalThis.fetch (url,
            {
                method:"POST",
                headers: {
                    "Content-Type" : "application/json"
                },
                body:JSON.stringify({
                    ...values,
                    expoPushToken:expoPushToken
                })

            })
            const user = await response.json()

            if(response.ok)
            {

                await AsyncStorage.setItem(
                    'loggedUser', JSON.stringify(user)
                )
                setUser(user)
            }
            else
            {
                setInvalidLogIn(user.error)
            }

        }catch(error){
            console.error("Error al iniciar sesion:", error);
        }
    }

    // ...
}
```

Con esta petición se intenta insertar una nueva sesión para el usuario que intenta iniciar sesión, pero la implementación del sistema de autenticación en el back-end (de la que se habla más en detalle en el apartado 7.2.3.1) no permite que un usuario mantenga dos o más sesiones activas en dispositivos distintos simultáneamente(cada dispositivo móvil genera un "expoPushToken" único). La aplicación está diseñada para que el usuario haga uso de ella y reciba notificaciones en el dispositivo móvil que envía datos de ubicación, entre otros, al back-end.

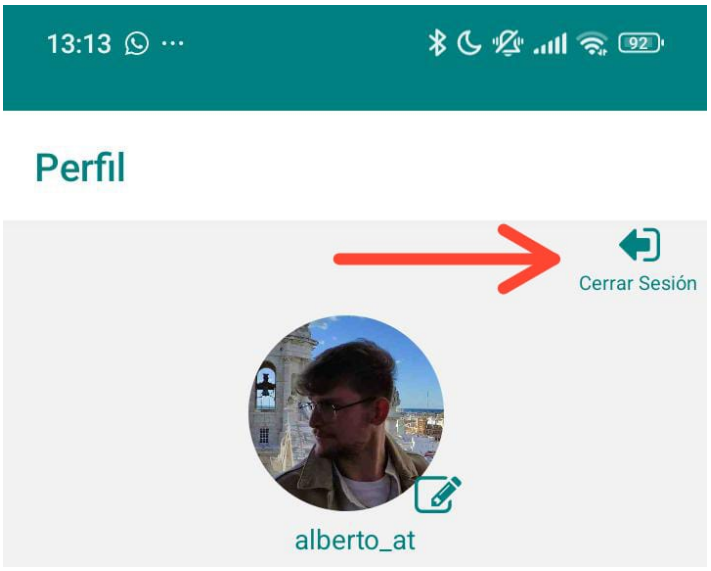


Figura 7.3: Botón cerrar sesión de la pantalla de perfil de usuario

La generación del "expoPushToken" se lleva a cabo al renderizarse el componente "AuthenticatedUserProvider" gracias a la definición de useEffect dentro del mismo, donde se utiliza la función "registerForPushNotificationsAsync". Esta función se define en el mismo fichero y configura un canal de notificaciones al que se le ha llamado "default" para después generar el token haciendo uso del "projectID" en la función "getExpoPushTokenAsync" del módulo "expo-notifications" (si el usuario ha otorgado permisos de notificaciones). Destacar que las notificaciones push y la generación del token solo se llevarán a cabo en dispositivos físicos. Tanto este código, como el referente a la configuración de los listener y el manejador de notificaciones han sido recogidos de la documentación oficial de la biblioteca "Expo Notifications" (Team, 2024c).

Volviendo al fichero "Perfil.jsx", vemos que dentro del componente "Perfil" se llama al hook "useDatosPerfil" para recuperar los datos de perfil. Dentro del hook se realiza una petición al correspondiente endpoint definido en el back-end, enviando el token que se genera al iniciar sesión (también desde el back-end) en la cabecera "Authorization". Hablamos de ello en detalle en el apartado 7.2.3.1.

7.2.2.3. Explorador de eventos

Implementada haciendo uso de un "ScrollView" donde cada componente dentro de la lista es un "TouchableOpacity", que al ser pulsado navegará por medio de la función "navigation.navigate" al componente "ListaEventos" definido en el fichero "ListaEventos.jsx", pasando como parámetro de la función una cadena de texto almacenada en la constante "event_type" que representa el tipo de eventos que se quieren visualizar.

El componente "ListaEventos" recoge ese parámetro gracias al uso del hook "useRoute" (Meta Platforms, 2024g) de la librería "@react-navigation/native", que devuelve un objeto que contiene información sobre los parámetros que se han pasado al navegar a la pantalla donde se ejecuta. Este objeto lo almacenamos en la constante "route", y

desestructuramos el parámetro de nombre “event_type” de la propiedad “params” de dicha constante.

La finalidad de pasar como parámetro a la pantalla la constante “event_type” cobra sentido cuando se ejecuta la función “fetchEventos”, la cual definimos para realizar una petición al back-end para obtener todos los eventos guardados en la base de datos cuyo tipo corresponda con el que la constante representa, es decir, con el tipo seleccionado por el usuario. Esta función se ejecuta cada vez que el componente se vuelve activo en la navegación gracias al uso del hook “useFocusEffect”(Meta Platforms, 2024f) de la librería “@react-navigation/native” y el hook “useCallback”(Meta Platforms, 2024e) de React, que permite almacenar la definición de funciones entre los conocidos “re-renders”. Esto quiere decir, se ejecuta cuando el componente se encuentra en la pantalla que se visualiza en ese momento. Podemos apreciar el uso de ambos hooks para la implementación del componente en el extracto de código 7.39.

Extracto de código 7.39: Uso de los hooks “useRoute”, “useFocusEffect” y “useCallback” en la implementación del componente “ListaEvento”

```
const ListaEventos = () => {
//...
    const route = useRoute();
    const { event_type } = route.params;

    const fetchEventos = async () => {
            try{
                const url = `http://${process.env.IP_SERVER
                   }:8080/eventos/${event_type}`
                console.log(url)

                const response = await globalThis.fetch (url)
                const json = await response.json()
                setEventos(json)
            }catch(error){
                console.error("Error peticion Get de todos
                   los eventos: ", error)
            }
    }

    useFocusEffect(
        React.useCallback(()=>{
            fetchEventos()
            return ()=>{}
        },[])
    )

//...
}
```

Una vez ejecutada la función “fetchEventos”, el array de eventos obtenido es almacenado en la variable de estado “eventos”, que será el valor asignado a la propiedad “data” del “FlatList”(Meta Platforms, 2024b) utilizado para mostrar la lista de eventos del tipo seleccionado, siendo cada elemento del array un elemento de la lista. También Se ha implementado y asignado el componente “RepositoryItem”, definido en el fichero “RepositoryItem.jsx”, a la propiedad “renderItem”. El componente asignado a esa propiedad es el encargado de definir cómo debe renderizarse cada elemento de la lista. Cada elemento de la propiedad “data” será pasado como “prop” al componente asignado a la propiedad “renderItem”.Se definen dentro del archivo “RepositoryItem.jsx” dos componentes: el propio “RepositoryItem” y otro componente “RepositoryItemHeaders” que el primero renderiza como una cabecera para mostrar el atributo que representa el título o nombre del evento.

También renderiza otro componente “RepositoryStats” definido en el fichero “RepositoryStats.jsx”. En el extracto de código 7.40 observamos cómo este componente hace uso de variables de estado para cada atributo pasado mediante las props que se quiera visualizar, y son inicializadas al valor booleano “false”. También vemos que se renderiza para cada atributo un “TouchableOpacity” que al ser pulsado convertirá el valor de la correspondiente variable de estado al valor opuesto. A modo de desplegable se ha implementado un renderizado condicional ternario de modo que, al ser el valor de la variable de estado “true”, se renderiza un icono u otro de la biblioteca “FontAwesome” del paquete “@expo/vector-icons”. De igual manera se implementa un renderizado condicional para renderizar un “View” (“ScrollView” en el caso del atributo “description”) contenedor del valor del atributo en base al valor de la variable de estado. Vemos en el extracto de código todo lo explicado para el caso de la variable de estado “daysOfWeek”, en cuyo caso el valor de la variable de estado se modifica realizando una llamada a la función implementada en el mismo fichero “DiasCastellano”, que como se puede intuir por su propio nombre, transforma al castellano las acortaciones de los días de la semana.

Extracto de código 7.40: Pequeña parte de la implementación del componente “RepositoryStats” haciendo uso de renderizados condicionales

```
const RepositoryStats = props => {
    const [description, setDescription] = useState(false)
    const [dtstart, setDtstart] = useState(false)
    const [dtend, setDtend] = useState(false)
    const [daysOfWeek, setDaysOfWeek] = useState(false)
    const [price, setPrice] = useState(false)


    return(
        <View style = {{flexDirection:'column',  padding
           :20}}>
        //...
        {props.daysOfWeek && props.daysOfWeek!=='null' &&(
            <View style={{marginBottom:10}}>
                <TouchableOpacity onPress={()=>setDaysOfWeek
```

```
                    (!daysOfWeek)} style={[styles.
                    TouchableAttribute, daysOfWeek && {
                    backgroundColor:theme.colors.transparente
                    }]}>
                     <StyledText fontSize='subheading' style={
                        styles.attribute}>Dias de la Semana</
                        StyledText>
                     {daysOfWeek ?(
                         <FontAwesome name="caret-up" size
                            ={12} color={theme.colors.
                            textSecondary} />
                     ):(
                         <FontAwesome name="caret-down" size
                            ={12} color={theme.colors.
                            textSecondary} />
                     )}
                </TouchableOpacity>

                {daysOfWeek && (
                     <View style={styles.
                        containerDataAttribute}>
                      <StyledText style={styles.
                         dataAttribute}>{DiasCastellano(
                         props.daysOfWeek)}</StyledText>
                     </View>
                )}
            </View>
        )}
        </View>
        //...
    )//fin_return
}
```

Una vez detallada la implementación dl componente “RepositoryStats”, debemos explicar que el componente “RepositoryItem”, además de renderizar los componentes “RepositoryItemHeaders” y “RepositoryStats”, renderiza condicionalmente un botón u otro en base al valor de la variable de estado “existsIntereses”, que representa mediante valores booleanos si el usuario ha dejado un “me gusta” a ese evento o no. Esta variable de estado toma un valor booleano u otro al enfocarse el componente, ejecutando gracias a los hooks “useFocusEffect” y “useCallback” la función “FetchCheckIfExistsInteres”. Esta función realiza una petición al back-end para obtener el valor booleano requerido. El primero de los botones se renderiza en caso negativo, para permitir al usuario mediante la función “PostInteres” marcar que le gusta el evento realizando una petición de tipo POST al correspondiente endpoint del back-end. Se implementa el segundo botón análogamente, llamando al ser pulsado a la función “onPressDelete” pasada como prop desde el componente “ListaEventos”, que realiza en este caso una petición de tipo DELETE. En ambos casos, al realizar la correspondiente petición al back-end, se actualiza

el valor de la variable de estado llamando a la función “FetchCheckIfExistsInteres”.

7.2.2.4. Lista de “me gusta”

Se ha implementado una pantalla que muestra los eventos que el usuario ha seleccionado que le gustan por medio del componente “ListaMeGusta” del fichero “ListaMeGusta.jsx”. Para su implementación, y de forma muy similar a la del componente “ListaEventos” definido en el fichero “ListaEventos.jsx”, se hace uso de un “FlatList” que mostrará cada evento almacenado en la variable de estado “meGusta”. Estos eventos son recogidos como resultado de la petición que se realiza dentro de la función “fetchMeGusta” al endpoint correspondiente definido en el back-end. La visualización de nuevos “me gusta” es posible navegando a la pantalla del Explorador de eventos y marcando nuevos eventos como “me gusta”, por lo que se hace uso de “useFocusEffect” y “React.useCallback” para ejecutar la función que realiza la petición mencionada cada vez que el componente “ListaMeGusta” se enfoca (como explicamos en el apartado 7.2.2.3). Por el contrario si un evento de la lista es marcado como “ya no me interesa”, la variable de estado se actualiza y no es necesario volver a renderizar el componente para visualizar la lista actualizada, pues el Flatlist usa como data dicha variable de estado “meGusta”. Esto sucede al pulsar el botón correspondiente que ejecuta la función “DeleteMeGusta”, cuya implementación puede observarse en el extracto de código 7.41. El componente **“MeGustaRepositoryItem”** hará de “renderItem”, y su implementación es muy similar a la del componente “RepositoryItem” detallado en el apartado 7.2.2.3.

Extracto de código 7.41: Implementación de la función “DeleteMeGusta” del componente “ListaMeGusta”

```
const DeleteMeGusta = async(eventID) => {
    try{
        const url = `http://${process.env.IP_SERVER
           }:8080/intereses/deleteMeGusta/${user.id}/${
           eventID}`
        console.log(url)

        const response = await globalThis.fetch (url, {
            method:'DELETE',
            headers:{
                'Content-Type':'application/json'
            }
        })

        if(response.ok)
        {
            const meGustaActualizados = meGusta.filter(
               evento => evento.id !== eventID)
            setMeGusta(meGustaActualizados)
        }
```

```
            else{
                throw new Error('Error realizando la peticion
                   ');
            }

        }catch(error){
            console.error("Error Delete MeGusta: ", error)
        }
    }
```

7.2.2.5. Lista de usuarios a los que también les gusta un evento

Se ha implementado una pantalla que muestra una lista de usuarios que han dado "me gusta" a un evento para el que el usuario también lo ha hecho. Su implementación se lleva a cabo por medio del componente "**ListaUsuariosMeGusta**" del fichero "ListaUsuariosMeGusta.jsx". A esta pantalla se navega desde la Lista de "me gusta", pulsando el botón "Mira a quién más le gusta este evento" de uno de los eventos de la lista. Desde esa pantalla se pasa como parámetro de ruta el identificador de evento "eventID". Mediante el uso de un "useFocusEffect" se llama a la función "fetchUsersWhoLikedEvent" implementada en el fichero. Esta función se encarga de realizar un fetch al endpoint **`http://${process.env.IP_SERVER}:8080/users/getUsersWhoLikedEvent/${user.id}/${eventID}`**. Se asigna como valor la longitud de la respuesta a la variable de estado "numUsersWhoLikedEvent", utilizada para el condicional que hará mostrar un mensaje por medio de un View cuando su valor sea cero y, por tanto, no existan otros usuarios que hayan dado "me gusta" a ese evento. Se asigna la propia respuesta a la variable de estado "usersWhoLikedEvent", cuyo valor se asigna a su vez a la propiedad "data" del FlatList que se mostrará en caso de existir algún usuario que haya dejado un "me gusta" al evento.

El componente "**UsuariosMeGustaRepositoryItem**" definido en el fichero "UsuariosMeGustaRepositoryItem.jsx" hará de renderItem. De cada elemento de la variable de estado "usersWhoLikedEvent" se le pasa como prop el "nickname" y el identificador de usuario "id", que se usarán en dicho componente. Mediante el uso de "useFocusEffect" se llama a las funciones "FetchCheckIfExistsIndividualChat" y "fetchGroupUserImage".

- **FetchCheckIfExistsIndividualChat**: Recibe el identificador del usuario que usa la aplicación, y el identificador del usuario de la lista con cuyos datos se renderiza el componente. Se realiza un fetch a la ruta **`http://${process.env.IP_SERVER}:8080/chats/checkIfExistsIndividualChat/${user1ID}/${user2ID}`** y se almacena la respuesta en la variable de estado "existsIndividualChat". Esta variable de estado se emplea mostrar, en caso de no existir chat individual entre ambos usuarios, un botón "Agregar a mis chats" junto a la imagen y nickname del usuario con que se renderiza el componente que hace de renderItem "UsuariosMeGustaRepositoryItem". Al pulsar dicho botón, se llamará a las funciones "PostInsertNewChat" y "PostInsertNewChatUser" (esta segunda dos veces, una para el usuario que usa la aplicación y otra para el usuario que desea agregar), que hacen

fetch a las rutas **http://${process.env.IP_SERVER}:8080/chats/insertNewChat/** y **http://${process.env.IP_SERVER}:8080/chats/insertNewChatUser/** respectivamente. Para la segunda función se envía como parámetro el identificador del nuevo chat creado.

- **fetchGroupUserImage**: Realiza un fetch a la ruta **http://${process.env.IP_SERVER}:8080/users/getImageByUserID/${props.id}** y asigna el valor de la respuesta a la variable de estado "imageUri", que será el valor asignado a la propiedad "source" del componente "Image" que mostrará la foto de perfil del usuario junto a su nickname. En caso de tener valor nulo, se mostrará la imagen "defaultUserImage".

7.2.2.6. Mapa: implementación con expo-location y react-native-maps

Al renderizarse el componente, gracias a un "useEffect", se solicita al usuario permisos de ubicación mediante la función **"requestForegroundPermissionsAsync"** de **"expo-location"**(Team, 2024b).
Mediante el método **"Location.watchPositionAsync"** se obtiene la ubicación del usuario, pasándole como parámetros un objeto con las opciones de precisión y de intervalo de distancia, a los que se le asigna respectivamente el valor "Location.Accuracy.BestForNavigation" y 250. Se obtendrá la ubicación cada 250 metros con la vista puesta en el consumo de recursos. Teniendo en cuenta que un evento se puede detectar cercano en el rango de 1 kilómetro, detectar la ubicación del usuario cada cuarta parte de esa distancia se ha considerado suficiente. Aparte de ese objeto de opciones, se le pasa un callback con la ubicación obtenida.

La ubicación del usuario se va actualizando gracias a la variable de estado **"location"** y su setter **"setLocation"**. Los datos de ubicación se envían como JSON a la respectiva cola amqp "Ubicacion". Para ello se ha implementado la función **"PostUbicacion"**, que recibe como parámetro el identificador de usuario (obtenido gracias al uso de "useContext" y el contexto "AuthenticatedUserContext"), ,su latitud, longitud, nombre de la ciudad, región y país para realizar una petición POST al endpoint implementado en el back-end `http://${process.env.IP_SERVER}:8080/users/UbicacionRabbitamqp` enviando los parámetros en el cuerpo de la solicitud.

La implementación hace uso de dos hooks implementados en otros ficheros e importados **"useEventos"** y **"useParking"**,que recuperan los datos de los eventos de la base de datos y los datos de parking respectivamente. Se renderiza un componente "MapView", y para cada evento y parking un "marker" en el mapa, además de otro "marker" con la posición del usuario. Para los parkings se ha implementado un renderizado condicional, en el que dependiendo del número de plazas disponibles se mostrará un icono u otro (variando de color el icono), igual que pasa en el "Callout", que al pulsar el icono permite que se muestre el nombre del parking, junto a las plazas disponibles, donde se cambia el color de fondo en base también al número de plazas sin ocupar. Algunos parking no constan de dicha información, por lo que solo se muestra la información disponible gracias de nuevo a otro renderizado condicional.

Por otro lado, se ha implementado una serie de botones cuya funcionalidad es seleccionar entre el día actual(selecionado por defecto) o uno de los 4 días siguientes.

Prestemos atención al extracto de código 7.42. Se recorre un array de cinco elementos, cuyos valores discretos van incrementando de uno en uno desde el número cero. Con ello conseguimos retornar para cada elemento un "TouchableOpacity" en el que la fecha será la fecha actual, a la que se suma el número de días equivalente al valor del elemento del array (o lo que es igual a su índice, por ello el nombre "index" que se le da a cada elemento recorrido). Como dijimos antes, el día actual es el que se selecciona por defecto al renderizarse el componente mediante el uso de un "useEffect" y la función **"handleDateSelectionDay"** (explicamos su implementación ver la implementación de los botones, y se puede observar en el extracto de código 7.43), a la que se pasa la fecha actual como parámetro. El botón del día actual se marca como seleccionado por defecto también al renderizarse el componente, pues se inicializa la variable de estado **"SelectedButton"** con el valor 0, valor que representa al primero de los botones. El día seleccionado tendrá un estilo diferente al resto de botones, gracias a un renderizado condicional. Al pulsar el botón y gracias a la propiedad "onPress", la variable de estado tomará el valor del "index" correspondiente a dicho botón, y este pasará a renderizarse con el nuevo estilo, sustituyendo al botón pulsado anteriormente.

Extracto de código 7.42: Implementación de botones de selección de fecha en el componente "Map"

```
<View style={styles.buttonDayContainer}>
    {[0, 1, 2, 3, 4].map(index =>
    {
    const date = new Date();
    date.setDate(date.getDate() + index);

    const options = { day: 'numeric', month: 'short' };
    const formattedDate = date.toLocaleDateString('es-ES
       ', options);

    return (
        <TouchableOpacity
        style={SelectedButton == index ? styles.
           SelectedButtonDay : styles.buttonDay}
        key={index}
        onPress={() =>{setSelectedButton(index);
           handleDateSelectionDay(date)}}
        >

        <Text style={SelectedButton == index ? styles.
           SelectedButtonDayText : styles.buttonDayText
           }>{formattedDate}</Text>

        </TouchableOpacity>
    )
    })
    }
</View>
```

De igual manera, vemos que al pulsar el botón se llama también a la función **"handleDateSelectionDay"** pasando la fecha como parámetro. Prestemos atención al extracto de código 7.43. Por medio del setter **"setEventsSelectedDate"**, almacenamos en la variable de estado **"EventsSelectedDate"** los eventos disponibles para la fecha seleccionada **"selectedDate"** que recibe como parámetro la función. Esto se consigue filtrando los eventos mediante el método "filter", retornando aquellos eventos para los que la condición dada ("selectedDate" se encuentra entre la fecha de inicio y de finalización del evento) es true. Como vimos anteriormente, gracias a los marcadores de "MapView" se podrá representar en el mapa el contenido de la variable de estado **"EventsSelectedDate"**.

Extracto de código 7.43: Implementación de la función "handleDateSelectionDay" en el componente "Map"

```
        const handleDateSelectionDay = (selectedDate)=>{
            if (eventos)
            {
            const auxEventsSelectedDate =  eventos.filter(
               evento => {
                  const DateDtstart = new Date(evento.dtstart);
                  const DateDtend = new Date(evento.dtend);
                  return (selectedDate >= DateDtstart &&
                     selectedDate < DateDtend )

            })

            setEventsSelectedDate( auxEventsSelectedDate );
            }
        }
```

Como se mencionaba anteriormente, por defecto se llamará a esta función con el valor de la fecha del día actual, gracias a un useEffect que esperará a que se recuperen todos los eventos por medio del hook "useEventos()" y almacene el valor de todos los eventos en la constante "eventos".

Se ha implementado un View a modo de leyenda que explica el significado de cada icono de parking, con un reloj que indica la hora actual en tiempo real, dando a entender al usuario que la información sobre las plazas disponibles está lo más actualizada posible. Para que el reloj muestre la hora en tiempo real, se ha implementado un useEffect con un serInterval que cada 1000 milisegundos (o lo que es lo mismo, cada segundo) actualizará el valor de la variable de estado que representa la hora actual usado en el View asignando el valor devuelto por "new Date()".

Se ha implementado además un botón con un signo de interrogación que sirve como guía de uso de esta pantalla para el usuario. Se hace uso de un **Modal** y de la variable de estado **modalVisible**, la cual se inicializa con el valor false y cambiará de

valor al pulsar el botón mencionado, actualizándose gracias a su setter setModalVisible. El modal se mostrará cuando la variable de estado mencionada tenga el valor true.

También se ha implementado una funcionalidad para los iconos de localización de eventos en el mapa. Al pulsarlos, se mostrará un View que reutiliza el código del componente "RepositoryItem" ya utilizado en la implementación del Explorador de eventos. Esto es posible gracias los setter "setEventoSeleccionado(evento)" y "setModalEventoVisible(true)", que actualizan los valores de las variables de estado "EventoSeleccionado" y "modalEventoVisible" respectivamente. El nombre de la última se debe a que el View implementado se muestra muy similar a cómo lo haría un componente Modal. Sin embargo, para el correcto funcionamiento de la funcionalidad se tuvo que optar por un componente View, pues el Modal no permitía la interacción con los TouchableOpacity del componente reutilizado.

7.2.2.7. Chats: navegación desde la lista de chats, creación y edición de Grupos

Desde nuestro "TabNavigator" podemos navegar al componente de navegación "ChatStackScreens", cuya ruta inicial es el componente **"ListaChats"**.Al ser renderizado, este componente realiza una solicitud de tipo GET al back-end para obtener todos los chats(individuales y/o grupales) a los que pertenece el usuario ejecutando mediante un "useFocusEffect" la función "fetchChats" (Ver extracto de código 7.44). Para cada elemento del JSONArray recibido, se comprueba si es un chat individual o grupal. En el primer caso, se obtiene y recorre el campo "memberUsersIDs", y para cada usuario dinstindo al que usa el chat se realiza una solicitud al back-end para obtener los datos de cada usuario, siendo el "chatname" el "nickname" del usuario, a diferencia del segundo caso en que será "groupName"(en este segundo caso no se devuelve el identificador del otro usuario participante del grupo, pues hay varios). Todo será devuelto en forma de promesa y almacenado en "nuevoJSON".

Extracto de código 7.44: Función "fetchChats" del componente "ListaChats"

```
const fetchChats = async () => {
    try{
        const url = `http://${process.env.IP_SERVER}:8080/
           chats/getChatsFromUser/${user.id}`
        const response = await globalThis.fetch (url)
        const json = await response.json()

        const nuevoJSON = await Promise.all( json.map(async (
           item) =>
        {
            //Si no es grupo Nombre Usuario
            if(!item.isGroup)
            {

                const [nuevoItem] = await Promise.all( JSON.
                   parse(item.memberUserIDs).filter(id => id
```

```
                !== user.id).map(async(id)=>
            {

                const url = `http://${process.env.
                    IP_SERVER}:8080/users/getByID/${id}`
                console.log(url)
                const response = await globalThis.fetch (
                    url)
                const userData = await response.json()
                const chatName = userData.nickname
                const chatUserID = userData.id


                const nuevoItem = {...item, chatName,
                    chatUserID}
                return nuevoItem
            }))
            return nuevoItem
        }
        //Si es Grupo Nombre Grupo
        else{

            const chatName = item.groupName
            console.log("Nombre de grupo: "+chatName)

            const [nuevoItem] = await Promise.all( JSON.
                parse(item.memberUserIDs).filter(id => id
                !== user.id).map(async(id)=>
            {
                const nuevoItem = {...item, chatName}
                return nuevoItem

            }))

            return nuevoItem
        }

    }))

    console.log("json modificado:")
    console.log(nuevoJSON)

    setChats(nuevoJSON)
    setNumChats(nuevoJSON.length)

}catch(error){
    console.error("Error peticion Get de los Chats del
```

```
            Usuario: ", error)
    }
}
```

En caso de que la longitud de la respuesta sea nula, y por tanto también el valor de la variable de estado "numChats", se mostrará un View mostrando un mensaje aclaratorio sobre cómo agregar a otros usuarios al chat. En caso contrario, los chats recogidos son almacenados en una variable de estado y son mostrados por medio de un "FlatList", cuyo "renderItem" es el componente **"ChatRepositoryItem"** definido en el fichero con mismo nombre. A este componente se le pasa la variable de estado "fetchChats", cuya prop "chatID" se usa como propiedad "key" para renderizar la lista de chats. Al ser enfocado este componente, se realiza una solicitud al back-end para obtener la imagen del Usuario en caso de ser un chat individual, y la imagen de grupo del chat en caso de ser un grupo, gracias a la comprobación del valor booleano de la prop "isGroup".

Esta imagen obtenida, junto al nombre del chat dado por el valor de la prop "chatName", se renderizan gracias al componente **"ChatRepositoryStats"** definido en el mismo fichero. El contenedor de estos datos es un "TouchableOpacity" que tiene dos funcionalidades.

Al ser pulsado de forma prolongada y gracias a la propiedad "'onLongPress", se mostrará al usuario un Alert, que le dará la opción de eliminar el chat seleccionado. Este Alert puede cancelarse gracias a la propiedad cancelable a la que se le ha asignado el valor true, pero de escoger la opción "Eliminar", se realizará una llamada a las funciones definidas "fetchDeleteIndividualChat(chatID)" y "fetchChats()", que se encargaran de realizar las solicitudes oportunas al back-end para eliminar el chat en cuestión y para volver a realizar la petición de los chats existentes del usuario.

Por otro lado, al ser pulsado y gracias a la propiedad "onPress", se navegará al componente **"Chats"**, cuya implementación detallaremos en el apartado 7.2.2.9. Si vamos al fichero "Navigation.js" vemos cómo la pantalla de este componente renderiza como header al componente **"ChatCustomHeader"** definido en el fichero de su mismo nombre. Este componente muestra un botón en forma de flecha que permite volver a la anterior pantalla en la navegación haciendo uso de la función " navigation.goBack()", junto con la imagen de chat y el nombre de chat. Estos dos últimos elementos están envueltos en un "TouchableOpacity" que, en base de nuevo al valor de la variable isGroup (recibida a través de parámetros de rutas y obtenida haciendo uso del hook "useRoute()"), navegará a una pantalla u otra. En caso de ser un chat individual, navegará a la pantalla **"ConsultPerfil"**, y en caso de ser un grupo a la pantalla **"ConsultGrupo"**.

El componente **"ConsultPerfil"** es muy parecido al componente **"Perfil"** a excepción de la funcionalidad de modificación de datos, estando solo implementada para el segundo de los componentes. Se definen dos funciones que se ejecutan al renderizarse el componente gracias al uso de "useEffect". La primera de las funciones es "fetchDatosPerfil", encargada de realizar una solicitud al back-end para almacenar en una variable de estado los datos del perfil cuyo id se corresponde con el valor del recibido parámetro de ruta "chatUserID". La segunda de las funciones, "fetchIntereses", se encarga de la

misma forma que la anterior de almacenar los intereses de los usuarios. Se muestra entonces junto al nickname, descripción y datos personales del usuario su imagen de perfil, recibida por medio del parámetro de ruta "imageUri". De no recibirse ninguna imagen de usuario se mostrará la imagen por defecto ''defaultUserImage", almacenada en el directorio "src/images" del proyecto. Se muestra mediante un "FlatList" una lista de los intereses del usuario. El "renderItem" de la lista indica cómo se muestran los datos, dejando a un lado los nombres de los tipos de evento, y a otro lado el número de eventos que le han gustado al usuario de cada tipo.

El componente **"ConsultGrupo"** muestra junto al nombre de grupo la imagen de grupo recibida por medio del parámetro de ruta "imageUri". Muestra los miembros del grupo por medio de un "FlatList" y usa como propiedad "data" el valor del parámetro de ruta recibido "memberUserIDs".

La implementación del componente **"ConsultGroupRepositoryItem"** tiene como objetivo servir de "renderItem" de la lista. Se definen dos funciones que se ejecutan al ser enfocado el componente: "fetchGroupUserImage" y "fetchGroupUserNickname". Estas se encargan, como sus nombres indican, de realizar una solicitud al backend para recuperar la imagen y nickname del usuario miembro del grupo para el que se renderizó el componente respectivamente. Este componente renderiza a otro llamado "ConsultGroupRepositoryStats", encargado de mostrar los datos que recibe el primero envueltos por un "TouchableOpacity" que al ser pulsado hará que se navegue al componente **"Perfil"** si el identificador (atributo "id") del usuario seleccionado coincide con el del usuario que está utilizando la aplicación, o bien al componente **"ConsultPerfil"** en caso de seleccionar otro miembro del grupo.

Además se implementan dos botones. El primero es el botón "Agregar a mis Chats", del que hablaremos en el apartado Chats: agregar al chat a nuevos usuarios mediante grupos compartidos. El segundo es el botón dedicado a eliminar a algún miembro del grupo, cuya implementación se puede observar en el extracto de código 7.45. Su texto será "Salir del Grupo" o "Eliminar" gracias a un operador condicional ternario que evaluará el valor de la variable "groupUserID" y comprobará si el usuario para el que se renderiza el componente es el propio usuario que usa la aplicación o no.

Gracias a la variable "numMemberUsersIDs", se muestra un Alert con el mensaje "No puedes dejar el grupo sin otros miembros." cuando, en caso de existir solo un miembro aparte del usuario que utiliza la aplicación, se pulsa este botón para intentar eliminarlo del grupo.

En caso contrario, se procederá a la eliminación del miembro del grupo llamando a la función "DeleteGroupUser", a la que se pasa como parámetros el identificador del chat de grupo "chatID" y del usuario a eliminar "groupUserID", realizando una petición a la url `http://${process.env.IP_SERVER}:8080/chats/deleteChatUser/${chatID}/${userID}` con los valores recibidos. Tras ello, se redirige al usuario a la pantalla "ListaChats".

Extracto de código 7.45: Implementación del botón para eliminar miembros de un grupo en el componente "ConsultGroupRepositoryStats"

```
<TouchableOpacity style={styles.botonEliminar}
```

```
    onPress={ ()=>{

        if(numMemberUsersIDs>2)
        {
            DeleteGroupUser(chatID, groupUserID)
            navigation.navigate('ListaChats')
        }
        else
        {
            Alert.alert("No puedes dejar el grupo sin otros
               miembros.")
        }


    }}
    >
        <Text style={styles.textoBoton}>
            {(groupUserID === user.id)?"Salir del Grupo":"
               Eliminar"}
        </Text>
    </TouchableOpacity>
```

Volviendo la vista al componente **"ConsultGrupo"**, vemos que tiene un "TouchableOpacity" con el texto "Editar Grupo". Este botón permite navegar al componente **"CreateGroupScreen"**. Sin embargo, este componente es también accesible por medio de otro "TouchableOpacity" con el texto "Crear Nuevo Grupo" en el componente **"ListaChats"**, el componente principal de la navegación implementada para el chat.

Vamos a detallar entonces la implementación del componente **"CreateGroupScreen"** teniendo en cuenta los dos componentes desde los que se puede navegar hasta él, pero antes veremos qué es lo que el componente renderiza para ambos casos. Renderiza junto a un "TextInput" que actualiza una variable de estado con el nuevo nombre del grupo una imagen (que será recibida como parámetro de ruta sólo en el caso de editar grupo) y un botón para seleccionar imagen que ejecutará la función definida "seleccionarImagen". Esta implementación es igual en el componente **Perfil**. Esta función solicita al usuario permisos para acceder a la galería ejecutando el método "requestMediaLibraryPermissionsAsync" del módulo "expo-image-picker" (Team, 2024a). Si el usuario otorga los permisos necesarios, se ejecuta el método "launchImageLibraryAsync" del mismo módulo. Vemos la implementación de la función en el extracto de código 7.46 .

Extracto de código 7.46: Función "seleccionarImagen" del componente "CreateGroupScreen"

```
    const seleccionarImagen = async () => {

        const { status } = await ImagePicker.
```

```
   requestMediaLibraryPermissionsAsync();
if (status !== 'granted') {
Alert.alert('Se requieren permisos para acceder a la
   galeria');
return;
}

let result;
try {
    result = await ImagePicker.
       launchImageLibraryAsync({
        mediaTypes: ImagePicker.MediaTypeOptions.All,
        allowsEditing: true,
        aspect: [4, 3],
        quality: 0.25,
    });
} catch (error) {
    console.error('Error al seleccionar imagen:',
       error.message, error);
    Alert.alert('Error', 'Hubo un problema al
       seleccionar la imagen.');
    return;
}



if (!result.canceled)
{
    const ImagePickerUri = result.assets[0].uri; //
       Acceder a la URI
    setImagePickerUri(ImagePickerUri)
    const mimeType = result.assets[0].mimeType; //
       Acceder a la URI
    const fileName = result.assets[0].fileName; //
       Acceder a la URI



    // Crear un objeto FormData para enviar la imagen
        al backend
    let formData = new FormData();
    formData.append('image', {
    uri: ImagePickerUri,
    name: fileName,
    type: mimeType,
    });
```

```
            setFormDataSelectedImage(formData)
        }//end_if
    }//end_function
```

También se muestra un "FlatList" que mostrará los chats individuales del usuario que está utilizando la aplicación para que pueda añadirlos o eliminarlos del gruopo, y para ambos casos que usará como propiedad "data" el valor de la variable de estado "chats" que será obtenida como resultado de la ejecución de la función "fetchChats" ejecutada al tomar foco el componente. Esta función discrimina aquellos chats que son grupos gracias a la propiedad "isGroup" de los registros recibidos de la base de datos, quedándose solo con los chats individuales. El componente que sirve de "renderItem" es **"CreateGroupRepositoryItem"**. Es importante en esta implementación observar que se les pasa como props tanto la variable de estado "addedUsers" como el setter "setAddedUsers" Al tomar foco este último componente para cada uno de los usuarios, obtiene la imagen del mismo realizando la pertinente solicitud al back-end, y almacenando el valor de la "uri" en la variable de estado "imageUri". Es el componente **"CreateGroupRepositoryStats"** el que se encarga de mostrar la imagen y el nickname del usuario (en este caso dado por la prop "chatName"). También renderiza un botón que hará que los usuarios se añadan o eliminen del grupo, actualizando la variable de estado que le llega al componente **"CreateGroupRepositoryItem"** como prop "addedUsers" por medio del setter "setAddedUsers". Si Observamos el extracto de código 7.47, podemos ver cómo hace uso de un renderizado condicional ternario para renderizar un botón para añadir a aquellos usuarios que no pertenezcan al grupo, y otro botón para eliminar a los usuarios miembros del grupo.

Extracto de código 7.47: Cuerpo del componente "CreateGroupRepositoryStats"

```
const CreateGroupRepositoryStats = ({chatUserID, chatName,
   imageUri, addedUsers, setAddedUsers}) => {
    return(
    < >
        <View style={styles.container}>

            <Image
                source={imageUri ? {uri:imageUri} :
                   defaultUserImage}
                style={styles.image}
            />

            <Text style={{ color: theme.colors.textSecondary,
                fontWeight: 'bold' }}>{chatName}</Text>

            <View style={styles.contenedorBoton}>
            {addedUsers.includes(chatUserID) ?
                (
                <TouchableOpacity style={styles.botonEliminar
                   }
```

```
            onPress={()=>{
                console.log("addedUsers : "+addedUsers)

                //HAY QUE HACER UN FILTER CON EL ID
                   DISTINTO QUE ES LO QUE SE ASIGNA TODO
                   SIN ESE USER.
                setAddedUsers( (prevAddedUsers) =>
                {
                    console.log("prevAddedUsers"+
                       prevAddedUsers)
                    const users = Array.isArray(
                       prevAddedUsers) ? prevAddedUsers :
                        JSON.parse(prevAddedUsers);
                    return users.filter(id => id !==
                       chatUserID);
                })
            }}
            >
                <Text style={styles.textoBoton}>Eliminar
                   </Text>
            </TouchableOpacity>
            ):
            (

            <TouchableOpacity style={styles.botonAnadir}
            onPress={()=>
            {
                console.log("addedUsers : "+addedUsers)

                setAddedUsers( (prevAddedUsers) =>
                {
                    console.log("prevAddedUsers"+
                       prevAddedUsers)
                    const users = Array.isArray(
                       prevAddedUsers) ? prevAddedUsers :
                        JSON.parse(prevAddedUsers);
                    return [...users, chatUserID]
                })
            }}
            >

                <Text style={styles.textoBoton}>Anadir</
                   Text>

            </TouchableOpacity>
            )
        }
```

```
            </View>
        </View>
    </>
    )
}
```

Sin embargo, la acción no se llevará a cabo hasta que en el componente padre **"CreateGroupScreen"** no se pulse el botón para confirmar la operación. Es ahora donde entra en juego el valor del parámetro de ruta "editGroup" recibido por este componente, y dicho de otra forma, desde qué otro componente se navegó a él y qué funcionalidad debe ofrecer:

- **Navegación desde "ConsultGrupo"**: Su botón "Editar Grupo" permite navegar al componente pasando parámetros de ruta que serán necesarios para poder implementar las funcionalidades requeridas para modificar el grupo en cuestión, como vemos en el extracto de código 7.48 . En especial, se envía el parámetro "editGroup" con el valor booleano true al componente **"CreateGroupScreen"**. Esto permite identificar desde qué componente se navega gracias a un renderizado condicional.

  Como vemos en el extracto de código 7.49, una vez actualizado el valor de la variable de estado "addedUsers" y pulsado este botón se eliminarán de la base de datos aquellos usuarios del grupo que hasta ahora eran miembros del grupo y han sido seleccionados para ser eliminados desde los componentes **"CreateGroupRepositoryItem"** y **"CreateGroupRepositoryStats"**, desde donde se actualiza el valor de la variable de estado "addedUsers". Esto lo sabemos comprobando que el identificador de usuario se encontrara en el parámetro de ruta "memberUserIDsRouteParam" y no se encuentre en "addedUsers". Por otro lado, debemos añadir los nuevos usuarios al grupo realizando también la correspondiente solicitud al back-end, pero no sin antes comprobar previamente que, efectivamente, el usuario del array "addedUsers" que se va a añadir no existía previamente en la base de datos para así evitar insertar entradas duplicadas, lo que provocaría error.

  También se modifica el nombre de grupo en la base de datos, además de la imagen, que es seleccionada una vez pulsado en botón de modificación de imagen gracias a la **función "seleccionarImagen"**. Esta función actualizará el valor de la **variable de estado "formDataSelectedImage"** por medio del **setter "setFormDataSelectedImage"** con el valor del objeto "formData". Este objeto contendrá un atributo "image" con la información requerida por la imagen (los atributos "uri", "name" y "type"). Volviendo a la implementación de la funcionalidad del **botón "editar grupo"** del componente en el renderizado condicional ternario, además de modificar el nombre de grupo como decíamos, también modifica la imagen, añadiendo un atributo "chatID" con el valor del parámetro de ruta "chatIDRouteParam" a ese objeto "formDataSelectedImage": esto es posible gracias al uso de una copia auxiliar con el valor de la variable de estado, a la que se añade el atributo y cuyo valor resultado se usará para sobreescribir el valor de la variable de estado con el setter. Para llevar a cabo la modificación de la nueva imagen en la base de datos, se llama a la **función "fetchWithRetry"**

que realiza la correspondiente solicitud al back-end.

Extracto de código 7.48: Botón "Editar Grupo" del componente "ConsultGrupo"

```
    <TouchableOpacity
    style={styles.TouchableCrearGrupo}
    onPress=
    {()=>
        {
        navigation.navigate('CreateGroupScreen',
        {
            memberUserIDsRouteParam:memberUserIDs,
            editGroup:true,
            chatIDRouteParam:chatID,
            chatNameRouteParam:chatName,
            imageUriRouteParam:imageUri
        })
        }
    }>

        <Text
        style={styles.TextTouchableCrearGrupo}>
            Editar Grupo
        </Text>

    </TouchableOpacity>
```

Extracto de código 7.49: Renderizado condicional ternario con valor editGroup a true

```
<>
    <TouchableOpacity
    style={styles.botonConfirmar}
    onPress={async()=>
        {
        console.log("addedUsers desde CreateGroupScreen :
            "+ addedUsers)
        if(addedUsers.length>1 && memberUserIDsRouteParam
            && addedUsers!==undefined )
        {
            //Borrar aquellos usuarios que ya existian en
                el grupo, pero que se han eliminado
            JSON.parse(memberUserIDsRouteParam).map(async
               (id) => {
                if( !(addedUsers.includes(id)) )
                    await DeleteGroupUser(
                       chatIDRouteParam, id)
            });
```

```
            // Anadir nuevos usuarios al grupo
            if(Array.isArray(addedUsers))
            {
                addedUsers.map(async (id) => {
                    if (id !== user.id && await
                       checkIfExistsChatUser(
                       chatIDRouteParam, id)===false) {
                       return await
                          PostInsertNewGroupUser(
                          chatIDRouteParam, id);
                    }
                });
            }else
            {
                JSON.parse(addedUsers).map(async (id) =>
                   {
                    if (id !== user.id && await
                       checkIfExistsChatUser(
                       chatIDRouteParam, id)===false) {
                       return await
                          PostInsertNewGroupUser(
                          chatIDRouteParam, id);
                    }
                });
            }




            await PostUpdateGroupName(chatIDRouteParam,
               textInput)

            if(formDataSelectedImage)
            {
                let auxFormDataSelectedImage =
                   formDataSelectedImage
                auxFormDataSelectedImage.append('chatID',
                    chatIDRouteParam);
                setFormDataSelectedImage(
                   auxFormDataSelectedImage)
                await fetchWithRetry(ImagePickerUri,
                   formDataSelectedImage, 3);
            }

            navigation.navigate('ListaChats')
        }
        else{
```

```
            Alert.alert("Selecciona almenos a otro
               usuario para compartir grupo.")
        }

    }}
    >
        <Text style={styles.textoBotonConfirmar}>Editar
           Grupo</Text>
    </TouchableOpacity>
</>
```

- **Navegación desde "ListaChats"**: Su botón "Crear Nuevo Grupo" permite navegar al componente "CreateGroupScreen", y a diferencia del anterior no le pasa ningún parámetro de ruta al componente destino. Por tanto en el caso del renderizado ternario entrará en el segundo caso, donde la expresión "editGroup" se evalúe a falsa. Igual que explicamos en el caso anterior, la variable de estado "addedUsers" se actualizará dentro de los componentes a los que se pasa como prop: "CreateGroupRepositoryItem" y "CreateGroupRepositoryStats". Como primera diferencia con el caso anterior, observamos que previamente no existe ningún miembro en el grupo, por lo que solo se tienen en cuenta los miembros que se encuentren en la variable de estado : En el caso anterior, los miembros previos a la edición del grupo se encontraban representados por el parámetro de ruta "memberUserIDsRouteParam", que en este caso no se le pasa al componente "CreateGroupScreen" mediante la navegación. La segunda diferencia con el caso anterior es que se llama a la función definida "PostInsertNewGroupChat" y que junto al usuario que crea el grupo, inserta en la base de datos también al resto de usuarios que se encuentren en la variable de estado "addedUsers", comprobando que se haya seleccionado almenos a un usuario para añadir al grupo, y mostrando un "Alert" en caso contrario. La implementación para la insersión de la imagen de grupo y la inserción del nuevo nombre de grupo es igual a la del caso anterior. Al realizar las llamadas a las funciones pertinentes se navegará al componente anterior "ListaChats" de nuevo.

7.2.2.8. Chats: agregar al chat a nuevos usuarios mediante grupos compartidos

Esta funcionalidad no deja de estar implementada y de funcionar de manera idéntica a como se implementa y detalla en el apartado Lista de usuarios a los que también les gusta un evento, solo que aplicada esta vez a otra pantalla, concretamente a la pantalla de consulta de grupo. Para detallar cómo se consigue añadir esta funcionalidad a las ya mencionadas en el apartado Chats: navegación desde la lista de chats, creación y edición de Grupos para el componente "ConsultGroupRepositoryItem", mencionamos la implementación de otro useFocusEffect y de una nueva función "FetchCheckIfExistsIndividualChat", que será llamada desde este para realizar una petición get a la url **`http://${process.env.IP_SERVER}:8080/chats/checkIfExistsIndividualChat/${user1ID}/${user2ID}`** , siendo "`user1ID`" y "`user2ID`" los identificadores de los dos usuarios en-

tre los que se quiere comprobar si exite un chat individual y que recibe esta función como parámetros. La respuesta será almacenada en la variable de estado "existsIndividualChat". Para el usuario para el que se renderiza el componente, si el valor de "existsIndividualChat" es un valor booleano false, y dicho usuario no es el propio usuario que está usando la aplicación, se mostrará un botón con el texto "Agregar a mis Chats". Al pulsar dicho botón, se llamará a las funciones "PostInsertNewChat" y "PostInsertNewChatUser" (esta segunda dos veces, una para el usuario que usa la aplicación y otra para el usuario que desea agregar), que hacen fetch a las rutas **http://${process.env.IP_SERVER}:8080/chats/insertNewChat/** y **http://${process.env.IP_SERVER}:8080/chats/insertNewChatUser/** respectivamente. Para la segunda función se envía como parámetro el identificador del nuevo chat creado.

7.2.2.9. Chats: intercambio de mensajes mediante uso de web sockets

En este apartado nos centraremos en la implementación del componente **"Chats"** con la importación del cliente de **"socket.io"**. Al crear el socket estamos creando en el extracto de código 7.50 una conexión entre el front-end y el back-end, además de definir qué información se enviará con cada evento enviado. El atributo "serverOffset" nos servirá para representar el identificador del último mensaje almacenado en la base de datos.

Extracto de código 7.50: Definición de socket

```
    socket = io(`http://${process.env.IP_SERVER}:8080`, {
     auth: {
         username: user.nickname ,
         userID: user.id,
         serverOffset: 0, //por defecto
         chatID
     }
})
```

Se ha definido una variable de estado "ListaMensajes" que dará valor a la propiedad "data" del "FlatList" que mostrará los mensajes del chat sobre los que se podrá hacer scroll. Se ha definido un componente **"FlatListRenderItem"** que, como su nombre indica, hará de "renderItem" de la lista. Para cada mensaje mostrará el usuario que lo escribió junto al cuerpo del mensaje. Gracias al uso de un renderizado condicional, tendrá un estilo u otro dependiendo si el mensaje es el usuario que usa el chat o no. En el primer caso los mensajes se muestran a la derecha de la pantalla, en el segundo a la izquierda.

Se ha implementado un "TextInput" y la variable de estado "mensaje", a la que se le asigna mediante el setter "setMensaje" el valor que escribamos como input gracias a la propiedad "onChangeText". Para enviar el mensaje se ha implementado un botón que al ser pulsado ejecuta la función "socket.emit" que envía un evento de tipo "chat message" junto al valor de la variable de estado "mensaje". Esto solo se ejecuta en caso de que el mensaje no sea vacío. Para comprobarlo se ha hecho uso del método "trim",

que elimina todos espacios consecutivos de la variable que se encuentren en los extremos de la cadena, y para cuyo valor devuelto comprobamos si es vacío comprobando su longitud (ver extracto de código 7.51).

Extracto de código 7.51: Implementación del botón para enviar mensajes del componente “Chats”

```
<TouchableOpacity
style={style.botonEnviar}
onPress={()=>{
    if(mensaje.trim().length !== 0)
    {
        socket.emit('chat message', mensaje);
        setMensaje("");
    }
}}
>
```

Cuando el listener del back-end escucha el evento enviado, emite otro evento “chat message” de vuelta al front-end, para lo que se ha implementado en el mismo también un listener (ver extracto de código 7.52) para escuchar mensajes tanto que acaban de ser enviados por el usuario que utiliza el chat, como mensajes sacados de la base de datos (ver implementación de la parte del back-end en el apartado 7.2.3.5). Junto a los eventos enviados desde el back-end se reciben en el front-end también parámetros como el propio mensaje, el usuario que escribió el mensaje y el identificador del mensaje en la base de datos (serverOffset). Este último se utiliza para actualizar el valor del serverOffset de auth en el front-end con el valor que llega desde el back-end al recibir el último mensaje. Se añade la información de los mensajes recibidos a la variable de estado “ListaMensajes”. Vemos en el extracto de código mencionado como también se comprueba para cada mensaje que llega, si este ya ha sido recibido y se encuentra en la lista de mensajes mencionada. El valor booleano resultante de la comprobación será almacenado en la constante “mensajeExiste”, en base a cual se añadirá a la lista el mensaje recibido en caso de ser false, evitando duplicados. Esto ocurre ya que cada vez que un usuario se conecta al chat, se recogen los mensajes almacenados de la base de datos enviándose desde el backend al chat, por lo que para el usuario que ya los ha recibido no se deben volver a tener en cuenta los mensajes.

Extracto de código 7.52: Definición de listener para recibir mensajes en el front-end

```
socket.on('chat message', (msg, serverOffset, username)
   =>{
    console.log(msg)

        setListaMensajes(prevMessages => {
            const mensajeExiste = prevMessages.find(
              m => m.key === String(serverOffset) && m.
                 text === msg && m.username === username
            );
```

```
                if(mensajeExiste)
                    return prevMessages
                else
                {
                    socket.auth.serverOffset = serverOffset
                       //Actualizo para saber donde se ha
                       quedado el cliente (ultimo mensaje
                       anadido)
                    return [...prevMessages,{key: String(
                       serverOffset), text: msg, username}]
                }
            });
    })
```

7.2.3. Back-end

La implementación del back-end se ha llevado a cabo siguiendo un patrón de arquitectura software similar al modelo-vista-controlador(MVC), pero que delega la parte de la vista al Front-end. Esto quiere decir que el back-end ha sido implementado como una API RESTful, cuyos ficheros de código han sido implementados y distribuidos según su funcionalidad en tres directorios distintos:

- **routes**: Los ficheros de este directorio son los encargados de definir las rutas y endpoints de la API RESTful, mediante los que el front-end podrá enviar solicitudes.
- **controllers**: Los controladores hacen de intermediarios entre la ruta y el modelo puesto que reciben las solicitudes HTTP y gestionan cómo se enviará la respuesta al cliente después de que el modelo se encargue de realizar las operaciones oportundas sobre los datos. Se encargan también de manejar posibles errores y de gestionar la autenticación, en nuestro caso utilizando JSON Web Tokens (JWT) (Auth0, 2024). Entraremos más en detalle sobre el uso de JSON Web Tokens en el apartado 7.2.3.1
- **models**: Los modelos se encargan de realizar las consultas y operaciones CRUD sobre los datos: creación, actualización, lectura y eliminación.

Como ya mencionamos en la sección 2.2.14, se ha hecho uso de Express para, además de utilizar su funcionalidad "Router" para implementar los ficheros ubicados en el directorio "routes" descrito anteriormente, inicializar el servidor permitiendo que maneje las solicitudes entrantes. En el extracto de código 7.53 podemos ver cómo se lleva a cabo dicha inicialización. Además vemos que se hace uso tanto del middleware "Cors"(permitiendo que se reciban solicitudes de cualquier origen), como de "express.json"(para que se acepten solicitudes con cuerpo en formato JSON) y de "fileUpload" cuyo uso lo veremos en la sección dedicada a la Subida de imágenes de perfil y de grupo usando express-fileupload.

Extracto de código 7.53: Uso de “Express” para inicializar servidor

```
const app = express();

app.use(logger('dev'))
app.use(express.json());
app.use(corsMiddleware());
app.use(fileupload());

const server = createServer(app)

server.listen(8080, ()=> {
    console.log("Server running on port 8080");
});
```

7.2.3.1. Autenticación: bcrypt y JSON Web Token

La implementación de la autenticación en el lado del back-end destaca por tener su propio router y controller, pero desde este último delegar la tarea de operar sobre los datos a los modelos “usersModel” y “sesionesModel”.

También destaca por el uso de dos funciones que han sido definidas en el archivo “handleBcrypt.js” ubicado en el directorio “helpers”. Estas funciones, como el nombre del propio fichero indica, utilizan la librería “bcrypt”(Cousens, 2023) del paquete “bcryptjs”. En el extracto de código 7.54 vemos cómo la función definida “encrypt” hace uso de la función de la librería “bcrypt.hash” para encriptar las contraseñas, y cómo la función definida “compare” utiliza la función “bcrypt.compare” con el fin de comparar una contraseña introducida en texto plano con su versión encriptada.

Extracto de código 7.54: Uso de funciones de “bcrypt” para encriptación y comparación de contraseñas en el fichero “handleBcrypt.js”

```
import bcrypt from "bcryptjs"

const encrypt = async (plainText) => {
    const hashPassword = await bcrypt.hash(plainText, 10)
    return hashPassword
}

const compare = async (plainPassword, hashPassword) => {
    return await bcrypt.compare(plainPassword,hashPassword)
}

export { encrypt, compare };
```

Vamos a centrarnos en la función “login” del controlador para ver la utilidad de todo lo anterior. Esta función recibe del cuerpo de la solicitud el correo electrónico y la

contraseña del usuario. Ejecuta el método "getByEmail" de "usersModel", que busca y obtiene de la base de datos el registro del usuario(incluída contraseña encriptada) cuyo correo electrónico se ha recibido, si es que existe.

La función "bcrypt.compare" devuelve un booleano de valor true en caso de una comparación exitosa entre la contraseña encriptada de la base de datos y la contraseña introducida en el formulario de inicio de sesión. Se puede observar en el extracto de código 7.55 el caso en que la contraseña introducida desde el formulario no coincida con la contraseña encriptada de la base de datos, se mandará un mensaje a la cola amqp haciendo uso de la biblioteca "amqplib" de Node.js (más concretamente del archivo "amqplib/callback_api.js"). Esto será procesado como evento simple por el motor CEP, y será útil para detectar situaciones de varios intentos de sesión fallidos en la cuenta del usuario gracias al patrón "InicioSesionFallido" del que hablamos en el apartado 7.1.2.

Extracto de código 7.55: Método "login" del controlador "authController" haciendo uso de la biblioteca "amqplib"

```
static async login (req, res)
{
    const {email, password, expoPushToken} = req.body
    const [user] = await usersModel.getByEmail(email)

    if(!user)
    {
        res.status(404).send({error : "Usuario no encontrado
           "})
        return
    }

    const passwordIsCorrect = await compare(password, user.
       password)

    const userForToken = {
            id: user.id,
            nickname: user.nickname,
            email: user.email
        }

    const token = jwt.sign(userForToken, process.env.
       JWT_SECRET)

    if(passwordIsCorrect)
    {

        //Si hay sesion activa para el mismo telefono (y por
           tanto mismo expoPushToken) no dara error
        const sessions = await sesionesModel.
           ExistsAnActiveSessionInAnotherDevice(user.id,
```

```
        expoPushToken)
    console.log(sessions)

    if (sessions.length != 0)
    {
        console.error('Error al iniciar sesion. Ya hay
            una sesion activa para este usuario.');
        return res.status(409).send({ error: "Error al
            iniciar sesion. Ya hay una sesion activa para
            este usuario en otro dispositivo." });
    }
    else
    {
        try{
            await sesionesModel.create(user.id, true,
                expoPushToken);
        }catch (error){
            console.error('Error al crear la sesion:',
                error);
            return res.status(500).send({ error: "Error
                al iniciar sesion. Intentalo de nuevo mas
                tarde." });
        }


        res.status(200).send(
        {
            id: user.id,
            name: user.name,
            nickname: user.nickname,
            email: user.email,
            token
        })
    }
}
else
{
    const queue = 'InicioSesionFallido';

    amqplib.connect('amqp://localhost', (err, conn) => {
    if (err) throw err;

    conn.createChannel((err, ch1) => {
        if (err) throw err;

        ch1.assertQueue(queue);
```

```
            ch1.sendToQueue(queue, Buffer.from(JSON.stringify
               ({"eventTypeName":"InicioSesionFallido", "
               email":email})));

          });
        });


        res.status(409).send({
            error:"Contrasena Incorrecta"
        })
    }

}
```

Por otro lado, en caso de que la contraseña introducida fuera correcta, podemos observar que se define un payload llamado "userForToken" para introducir la información que interesa del usuario en un JSON Web Token. Se usa la función "sign" de la biblioteca "jsonwebtoken" para firmar el token con una clave que se encuentra definida en el archivo ".env" y asegurar que el token no se modifica. En caso de haberse introducido una contraseña correcta y no existir una sesión activa en otro dispositivo para ese usuario (se comprueba por medio del método "ExistsAnActiveSessionInAnotherDevice" del modelo "sesionesModel"), se enviará una respuesta que contendrá los datos del usuario que inicia sesión junto al JSON Web Token generado.

Vamos a ver cómo se hace uso del mencionado JSON Web Token que se genera al iniciar sesión en la función de "usersController", que llama justo al método mencionado anteriormente "getByEmail". En el inicio de sesión se ejecutaba directamente este método del modelo "usersModel" desde el controlador "authController", sin hacer de intermediario el controlador "usersController".

En el extracto de código 7.56 vemos cómo para poder llamar desde el controlador "usersController" al mencionado método "getByEmail" garantizando el permiso de acceso, es necesario recuperar de la cabecera "authorization" el token(Generado al inciarse la sesión), decodificarlo usando la clave del archivo ".env" y recuperar en este caso el correo electrónico del usuario.

Extracto de código 7.56: Método "getByEmail" del controlador "usersController" haciendo uso de JSON Web Token

```
static async getByEmail (req, res)
{
    const authorization = req.get('authorization')
    let token = null

    if(authorization && authorization.toLowerCase().
       startsWith('bearer'))
    {
```

```
        token = authorization.split(' ')[1]
    }

    let decodedToken = {}
    //Verificamos que con el secret podemos volver a abrir
       este token
    try{
        decodedToken = jwt.verify(token, process.env.
           JWT_SECRET)
    }catch(error){
        console.error('Token invalido:', error.message)
        return res.status(401).json({ error: 'Acceso no
           autorizado: token no valido' })
    }

    console.log("decodedToken : ")
    console.log(decodedToken)

    const {email} = decodedToken

    const user = await usersModel.getByEmail(email);
    console.log(user)

    res.status(200).send(user);
}
```

En el apartado dedicado a las Pruebas de integración del back-end con la base de datos usando Jest Y SuperTest se verifica el correcto funcionamiento de los endpoints implementados haciendo uso de Json Web Token. Las rutas protegidas mediante el uso de Json Web Token en sus respectivos controladores son:

- chatsRouter
  - `/getChatsFromUser/:userID/`
  - `/getOnlyIndividualChatsFromUser/:userID/`
  - `/checkIfExistsIndividualChat/:user1ID/:user2ID/`
- usersRouter
  - `/:email`

7.2.3.2. Obtención de usuarios con un "me gusta" en común

Siguiendo la organización de directorios y ficheros detallada en el apartado 7.2.3, la implementación de esta funcionalidad se ha llevado a cabo en tres ficheros de código: un fichero de ruta, un controlador y un modelo. Centrarnos en cómo se obtienen los

datos esperados de la base de datos es fundamental para detallar en esencia la implementación de esta funcionalidad, por ello se muestra el método "getUsersWhoLikedEvent" del modelo "usersModel", ubicado en el fichero "\server\models\users.js" (ver extracto de código 7.57).

En él podemos observar la consulta en la que utilizamos una cláusula JOIN ON para extraer los registros de la tabla "users" que han dejado un "me gusta" (por tanto hay un registro en la tabla "intereses" con el valor de su identificador asignado a la clave foránea "intereses.userID") a un evento concreto dado por el parámetro recibido "eventID". En el resultado buscamos obtener a todos los usuarios que cumplan dicha condición a excepción del usuario que utiliza en ese momento la aplicación, cuyo identificador será recibido por el método a través del parámetro "userID".

Extracto de código 7.57: Método "getUsersWhoLikedEvent" del modelo "usersModel"

```
    static async getUsersWhoLikedEvent(userID, eventID){
        const [row] = await pool.query(
            'SELECT users.* FROM users JOIN intereses ON
               users.id = intereses.userID  WHERE users.id !=
                ? AND intereses.eventID = ?;',
            [userID, eventID]
        )
        return row;
    }
```

7.2.3.3. Chats: comprobación de la existencia de un chat individual entre dos usuarios

Esta funcionalidad junto a la detallada en el apartado Chats: creación, modificación y eliminación de Chats son esenciales para el correcto funcionamiento de la parte front-end, concretamente para la funcionalidad de poder agregar usuarios al chat a través de la Lista de usuarios a los que también les gusta un evento y la pantalla "ConsultGrupo", de cuya implementación se habla en el apartado Chats: agregar al chat a nuevos usuarios mediante grupos compartidos. Siguiendo la organización de directorios y ficheros detallada en el apartado 7.2.3, la implementación de esta funcionalidad se ha llevado a cabo en tres ficheros de código: un fichero de ruta, un controlador y un modelo. Centrarnos en cómo se obtienen los datos de la base de datos y en cómo el controlador procesa esos datos es esencial para detallar la implementación de la funcionalidad.

Obsérvese el extracto de código 7.58 donde se muestra la implementación del método "getIndividualChatByUsersID" del modelo "chatsModel", ubicado en el fichero "\server\models\chats.js".

Extracto de código 7.58: Método "getIndividualChatByUsersID" del modelo "chatsModel"

```
    static async getIndividualChatByUsersID(user1ID, user2ID)
       {
```

```
        const [row] = await pool.query(
            `SELECT chatID
            FROM CHAT_USERS
            JOIN chats ON chat_users.chatID = chats.id
            WHERE chat_users.userID IN (?, ?) AND !chats.
               isGroup
            GROUP BY chat_users.chatID
                        HAVING COUNT(DISTINCT chat_users.
                           userID) = 2`,
            [user1ID, user2ID]
        )

        return row;
    }
```

Se recoge aquel identificador del chat que esté asignado como valor a la clave foránea "chatID" de la tabla "chat_users" y para el que, los usuarios que pertenezcan al chat tengan uno de los dos identificadores recibidos como parámetros (es decir, que el campo chat_users.userID contenga el valor "`user1ID`" o el valor "`user2ID`") y el chat no sea un grupo (se comprueba el campo chats.isGroup).

A continuación se explica algo muy relevante para la consulta: Aunque se compruebe que no es un grupo, debemos recoger solo aquellos chats para los que se cuentan exactamente dos identificadores de usuario diferentes gracias a
HAVING COUNT(DISTINCT chat_users.userID) = `2`. La clausula WHERE hace que solo se tengan en cuenta los valores "`user1ID`" y "`user2ID`" para chat_users.userID (un chat al que pertenece solamente uno de ellos tendría el valor `1` para HAVING COUNT(DISTINCT chat_users.userID)), por lo que es necesario añadir el operador HAVING COUNT mencionado para que devuelva los chats a los que estrictamente pertenecen los dos identificadores recibidos.

Si no añadimos esto a la consulta se recibirán, además del chat esperado, los chats individuales a los que pertenecen de manera independiente los dos usuarios cuyos identificadores recibe el método.

Obsérvese a continuación el extracto de código 7.59 donde se muestra la implementación del método "checkIfExistsIndividualChat" del controlador "chatsController", ubicado en el fichero "`\server\controllers\chats.js`".

Extracto de código 7.59: Método "checkIfExistsIndividualChat" del controlador "chatsController"

```
    static async checkIfExistsIndividualChat (req, res) {
        const {user1ID, user2ID} = req.params
        const [chatID] = await chatsModel.
           getIndividualChatByUsersID(user1ID, user2ID);
        const result = ( chatID != undefined )

        res.status(200).send(result);
```

```
}
```

Observamos que el resultado enviado será un valor booleano devuelto por el operador lógico empleado. Devolverá true en caso de recibir un valor definido, es decir, que en la consulta se recibiera un identificador de un chat que cumpliera las condiciones.

7.2.3.4. Chats: creación, modificación y eliminación de Chats

Vamos a detallar la implementación de las funcionalidades desde la perspectiva de la idea de chat, es decir, qué métodos del modelo nos permite crearlo, hacer cambios en él y eliminarlo. Siguiendo la organización de directorios y ficheros detallada en el apartado 7.2.3, la implementación de esta funcionalidad se ha llevado a cabo en tres ficheros de código: un fichero de ruta, un controlador y un modelo.

- **Creación**: Obsérvese el extracto de código 7.60 donde se muestra la implementación del método "insertNewChat" del modelo "chatsModel", ubicado en el fichero "`\server\models\chats.js`". Recibe el parámetro isGroup que servirá para identificar si el chat será individual o grupal.

Extracto de código 7.60: Método "insertNewChat" del modelo "chatsModel"

```
static async insertNewChat(isGroup){

const [result] = await pool.query(
    'INSERT INTO chats (isGroup) VALUES (?);',[
       isGroup]
)
return result
}
```

- **Modificación**: Obsérvese el extracto de código 7.61 donde se muestra la implementación del método "insertNewChatUser" del modelo "chatsModel", ubicado en el fichero "`\server\models\chats.js`". Aunque su finalidad es la de modificar los miembros de un chat(ya sea en el momento de su creación o posteriormente), realmente lo que se hace es crear nuevos registros en la tabla "chat_users" de la base de datos. Se recibe los valores a asignar a las claves foráneas del nuevo registro por medio de los parámetros de mismo nombre que las mismas: "chatID" que relaciona a esta tabla con "chats", y "userID" que la relaciona con la tabla "users".

Extracto de código 7.61: Método "insertNewChat" del modelo "chatsModel"

```
static async insertNewChatUser(chatID,userID){

const [result] = await pool.query(
    'INSERT INTO chat_users (chatID, userID)
       VALUES (?,?);',[chatID, userID]
```

```
        )
        return result
        }
```

Eliminar a un usuario de un grupo se lleva a cabo gracias al método "deleteChatUser" del modelo "chatsModel" (Ver extracto de código 7.62). Su función es modificar un chat, pero realmente se lleva a cabo una eliminación de uno de los registros de la tabla "chat_users".

Extracto de código 7.62: Método "deleteChatUser" del modelo "chatsModel"

```
        static async deleteChatUser(chatID, userID){
            const result = await pool.query(
                'DELETE FROM chat_users WHERE chatID = ?
                   AND userID = ?;',
                [chatID, userID]
            )
            return result
            }
```

La subida de imágenes se detalla en el apartado Subida de imágenes de perfil y de grupo usando express-fileupload. La implementación del método "editGroupName" implementado también en el mismo modelo es trivial.

- **Eliminación**: Para eliminar un chat se implementa el método "deleteIndividualChat" dentro del mismo modelo, al que se le pasa el identificador del chat a eliminar como parámetro y cuya implementación es trivial. Gracias a la restricción ON DELETE CASCADE aplicada a las claves foráneas "ChatID" de la tabla de relación "chat_users", y "chatID" de la tabla "mensajes", se eliminan de estas tablas los registros para los que estas claves foráneas tengan el valor del identificador del chat eliminado.

7.2.3.5. Chats: intercambio de mensajes mediante uso de web sockets

Inicializamos el servidor Socket.io pasándo como argumento "server", consiguiendo enlazar así ambos servidores y haciendo que utilicen mismo puerto y url. En el extracto de código 7.64 vemos cómo definimos un listener principal a la escucha de una conexión, y a partir de él el resto de listener. Vemos cómo se hace uso del método join del socket, al que pasamos el identificador del chat al que se conecta el usuario como parámetro. Esto permite estblecer a qué chat en concreto se envían los mensajes, evitando que los mensajes lleguen a chats que no corresponden.

Observemos también en el mismo extracto de código 7.64 el condicional: Utilizamos la propiedad **"socket.recovered"** (Socket.IO, 2024) para saber si el cliente ha tenido una perdida de conexión y la reconexión no ha sido exitosa, en cuyo caso la propiedad será falsa. En el caso de la primera conexión, la propiedad también será falsa y se entrará dentro del cuerpo del condicional. Se ejecuta entonces la función **"selectFromLastMsgReceived"** del modelo **"messagesModel"** (ver extracto

de código 7.63) para, como su nombre indica, seleccionar los mensajes de la base de datos a partir de un identificador de mensaje y de chat dados. Por cada registro resultado de la consulta, se enviará un evento "chat message" junto al cuerpo del mensaje, su identificador y el usuario que lo escribió al front-end.

Extracto de código 7.63: Método "selectFromLastMsgReceived" del modelo "messagesModel"

```
static async selectFromLastMsgReceived(serverOffset,
   chatID){
     const [results] = await pool.query(
         'SELECT id, cuerpo, username FROM mensajes WHERE
            chatID = ? AND id > ?',[chatID, serverOffset]
     )
     return results
 }
```

Nos centramos en el listener de eventos de tipo "chat message" (ver de nuevo el extracto de código 7.64). Cuando un mensaje es enviado desde el front-end, el back-end se encarga de escuchar el evento y de insertar el nuevo mensaje en la base de datos(ejecutando el método **"insert"** del modelo **"messagesModel"**). El listener recibe como parámetro el cuerpo del mensaje y por medio del **handshake** la información necesaria sobre el usuario que envió el mensaje. Envía de vuelta un evento del mismo tipo con el cuerpo del mensaje, el identificador del registro de la base de datos en que se ha almacenado el mensaje y el usuario que lo escribió.

Extracto de código 7.64: Inicialización de servidor Socket.IO y definición de listener en el back-end

```
const io = new Server(server)

io.on('connection', async (socket)=>
{

    socket.join(socket.handshake.auth.chatID);

    console.log(socket.handshake.auth.username,"se ha
       conectado al chat: ",socket.handshake.auth.chatID)
       ;

    socket.on('disconnect', ()=> {
        console.log("Un Usuario se ha Desconectado al
           chat");

    })


    socket.on('chat message', async (msg) => {
```

```
        console.log("Mensaje:", msg)
        console.log("chatID:", socket.handshake.auth.
            chatID)

        const username = socket.handshake.auth.username
            ?? "anonymous"

        try{
            const result = await messagesModel.insert(msg
                , username, socket.handshake.auth.userID,
                socket.handshake.auth.chatID)

            //id del ultimo mensaje que se ha guardado (
                serverOffset)
            io.to(socket.handshake.auth.chatID).emit('
                chat message', msg, (result.insertId).
                toString(), username)


        }catch(e){
            console.error(e)
            return
        }
    })




    //Para que le lleguen los mensajes al usuario cuando
        se pierde conexion y no se recupera de manera
        natural, o en la primera conexion
    //Se recuperan mensajes con id mayor al serverOffset
        dentro de auth (id del ultimo mensaje recibido)
    if(!socket.recovered){
        try {
            const results = await messagesModel.
                selectFromLastMsgReceived(socket.handshake
                .auth.serverOffset, socket.handshake.auth.
                chatID)

            console.log( "Mensajes sacados de la BD que
                aun no han sido recibidos por el cliente")
            console.log(results)

            results.forEach((row)=>{
                io.to(socket.handshake.auth.chatID).emit
```

```
                    ('chat message', row.cuerpo, (row.id).
                    toString(), row.username)

            })


            socket.recovered = true;

        } catch (e) {
            console.error(e)
        }
    }

})
```

7.2.3.6. Chats: notificaciones de mensajes enviados

Para conseguir que se notifiquen mensajes recién enviados e insertados en la base de datos, se implementa el método "insert(msg, username, userID, chatID)" de "messagesModel", al que le llegan como parámetros el propio cuerpo del mensaje, el username del autor del mismo junto a su identificador, y el identificador del chat al que se envía el mensaje. Dentro de este método, además de la inserción del mensaje en la base de datos, se realiza una llamada al método "chatsModel.getRestOfUsersFromChat(chatID,userID)", pasándo como parámetros el identificador de chat y del usuario que manda el mensaje. Es sencillo intuir que, como su nombre indica, devuelve los identificadores del resto de miembros del chat. Tras ello, recorre el array de identificadores mapeándolo, y para cada uno de ellos realiza una llamada al método "sesionesModel.getExpoPushToken" pasándole su valor como parámetro.

Sólo en el caso de que tenga una sesión activa, se enviará la notificación por medio de la llamada al método "messagesModel.sendPushNotification(msg, username, anotherUserExpoPushToken.expoPushToken)". Este último enviará la notificación al dispositivo correspondiente con el token recibido, que tendrá de título el nombre de usuario y de cuerpo el nombre de usuario y mensaje recibidos respectivamente.

7.2.3.7. Subida de imágenes de perfil y de grupo usando express-fileupload

La implementación de estas dos funcionalidades se lleva a cabo mediante el uso de **express-fileupload** (Girges, 2024), una extensión de express que permite al sevidor recibir archivos por parte del cliente para posteriormente poder almacenarlos de manera persistente en la base de datos. Siguiendo la organización de directorios y ficheros detallada en el apartado 7.2.3, se han implementado tres ficheros de código para cada una de las dos funcionalidades: un fichero de ruta, un controlador y un modelo.

Nos centramos en la implementación del controlador. El controlador se ha implementado de igual manera para ambas funcionalidades mediante los métodos **"Uploa-**

dImage” del controlador **“usersController”** y **“UploadGroupImage”** del controlador **“chatsController”** . En el extracto de código 7.65 vemos cómo se ha implementado el primero de los métodos mencionados para el controlador. Comprobamos que se haya enviado junto a la solicitud algún archivo (por medio del objeto req.files), y en caso de no ser así retorna un código de estado 400 indicando ausencia de imagen. Se le asigna la representación binaria de la imagen en un buffer a la constante “imagenBinaria”. La imagen será almacenada en su formato binario en la base de datos, en un campo de tipo “MEDIUMBLOB” como vemos en el apartado dedicado al Esquema de la base de datos. A partir de este punto es donde radica la diferencia entre ambas funcionalidades, en la implementación de un método del modelo y los parámetros que se le pasan:

- **Método “UploadImage” del modelo “usersModel”**: Recibe como parámetro junto a la imagen en formato binario el identificador de usuario “userID” para insertar la imagen en el campo “image” del registro correspondiente de la tabla “users”.
- **Método “UploadGroupImage” del modelo “chatsModel”**: Recibe como parámetro junto a la imagen en formato binario el identificador de chat “chatID” para insertar la imagen en el campo “groupImage” del registro correspondiente de la tabla “chats”.

Extracto de código 7.65: Método “UploadImage” del controlador “usersController”

```
static async UploadImage(req, res){

    if (!req.files) {
        return res.status(400).send("No se encontro ningun
           archivo.");
    }

    const userID = req.body.userID


    const image = req.files.image;
    const imagenBinaria = image.data;

    const result = await usersModel.UploadImage(imagenBinaria
       , userID);

    if (result.affectedRows > 0) {
        res.status(200).json({ message: 'Imagen insertada con
            exito' });
    } else {
        res.status(404).json({ message: 'Imagen no insertada
           con exito' });
    }
}
```

7.2.3.8. Modelo de eventos en el back-end

Es interesante prestar atención a cómo se recogen los datos de eventos de la base de datos cuando se solicitan desde la pantalla del Explorador de eventos. En este modelo se implementan los métodos que serán utilizados cuando se realicen las solicitudes desde el front-end a sus respectivos endpoints. No se recogen eventos que ya hayan alcanzado su fecha de finalización, por lo que todas las consultas a la base de datos de este modelo incluyen "`dtend >= NOW() ORDER BY dtend ASC`". De esta manera, los primeros eventos que el usuario visualice en el explorador serán en gran parte los más actuales, pues aunque unos eventos tienen una duración más prolongada, muchos de ellos empiezan y acaban el mismo día.

8. Pruebas del sistema

En este capítulo se presenta el plan de pruebas del sistema. Las pruebas, como ya se comentó en el apartado dedicado a las fase de Pruebas del desarrollo incremental, se han realizado conforme se iban implementando nuevas funcionalidades al sistema.

8.1. Pruebas unitarias

Las pruebas unitarias tienen por objetivo localizar errores en cada nuevo artefacto software desarrollado, antes que se produzca la integración con el resto de artefactos del sistema. A continuación detallamos las diferentes pruebas unitarias realizadas.

8.1.1. Patrones de eventos con Esper EPL Online

8.1.1.1. Aclaración sobre horario local de España y horario GMT/UTC usado por la herramienta

Antes de nada, debemos explicar cómo se interpretan las fechas y horas en Esper EPL Online. A diferencia de la aplicación Mule, que sí tendrá en cuenta la hora local del sistema, las fechas en Esper EPL Online siguen el horario GMT/UTC (Greenwich Mean Time/Tiempo Universal Coordinado).

Hora local en España respecto de la hora GMT/UTC:

- Horario local de España en invierno: GMT+01h
- Horario local de España en verano: GMT+02h

Ejemplo: Imagina que quieres simular que el momento de comienzo de un evento de Madrid es el dado por la fecha local española "2024-10-04 14:30:00.000". Obsérvese en la figura 8.1 que su equivalente horario GMT/UTC sería de dos horas menos (Teniendo en cuenta que el horario establecido para esa fecha sería el de verano). Por tanto, si escribimos en "dtstart" el valor de la fecha y hora local española, puede dar lugar a confusión puesto que el valor EPOCH realmente es el de su equivalente horario GMT/UTC. Por tanto, para evitar confusiones se van a utilizar directamente fechas GMT/UTC como valores tanto para los campos en formato EPOCH como para los campos en formato fecha. De esta manera el valor EPOCH correspondrá directamente a la fecha GMT/UTC del atributo "dtstart" (ver figura 8.2).

8.1.1.2. Probando los patrones de eventos

Probamos entonces los diferentes patrones de eventos implementados:

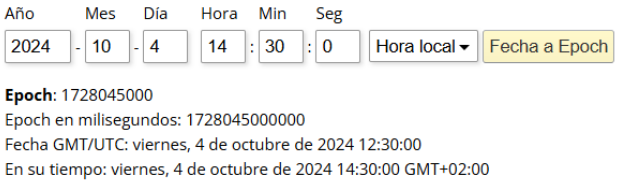


Figura 8.1: Conversión de fecha horario local español al valor EPOCH de su equivalente fecha GMT/UTC usando el conversor https://espanol.epochconverter.com/

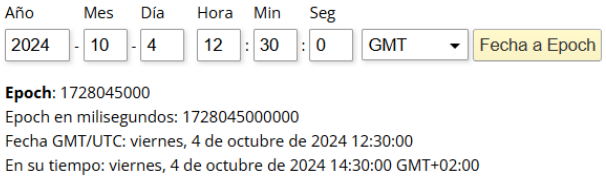


Figura 8.2: Conversión de fecha GMT/UTC a EPOCH usando el conversor https://espanol.epochconverter.com/

- **DetectaInicioSesionFallido**: En la figura 8.3 podemos observar cómo el patrón detecta para cada distinta dirección de correo electrónico cuando se dan 3 inicios de sesión fallidos en el intervalo de tiempo de 1 min.

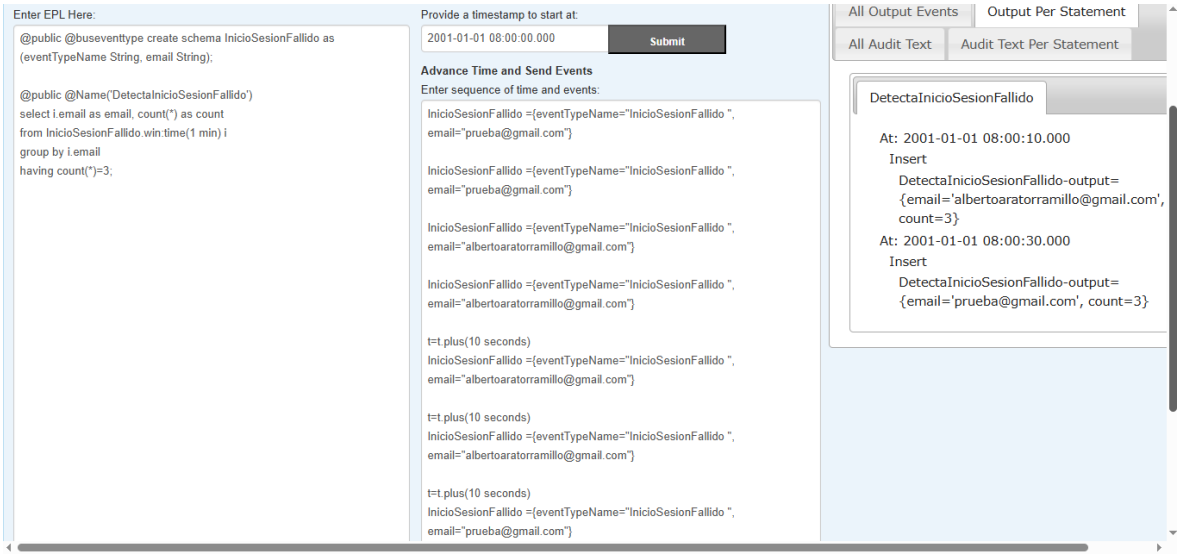


Figura 8.3: Prueba unitaria patrón "DetectaInicioSesionFallido" usando Esper EPL Online

- **DetectaFuerzaBruta**: En la figura 8.3 podemos observar cómo el patrón detecta para cada distinta dirección de correo electrónico cuando se dan 10 inicios de sesión fallidos en el intervalo de tiempo de 1 segundo. El patrón ha sido ligeramente modificado con el fin de facilitar las pruebas unitarias, pues el patrón original detecta 100 inicios de sesión fallidos en el intervalo de tiempo de 1 minuto.

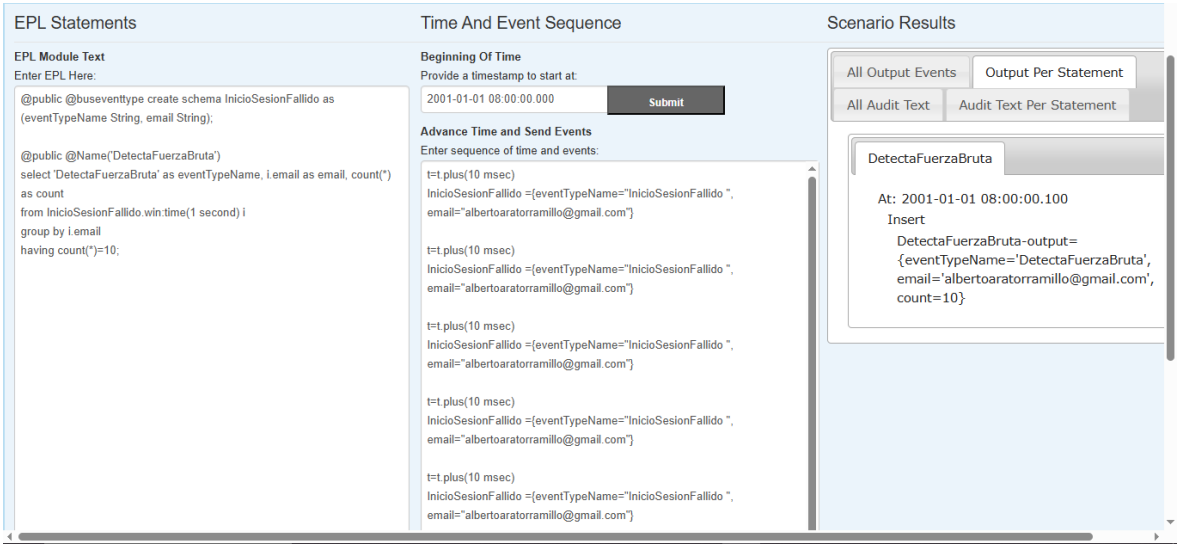


Figura 8.4: Prueba unitaria patrón "DetectaFuerzaBruta" usando Esper EPL Online

- **UbicacionSospechosa**: En la figura 8.5 podemos observar cómo, para cada usuario, el patrón detecta cada cambio del campo referente al País de cada ubicación recibida del mismo.

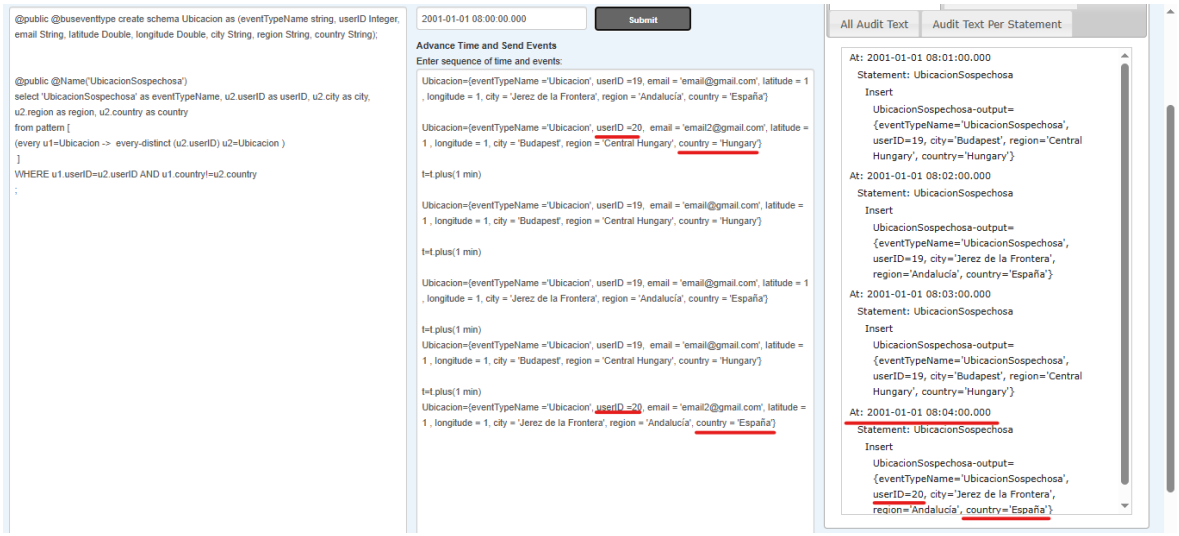


Figura 8.5: Prueba unitaria patrón "UbicacionSospechosa" usando Esper EPL Online

- **MenosDe5Dias**: En la figura 8.6 podemos observar cómo enviamos 3 eventos de Madrid como eventos simples, cada uno con una fecha de comienzo diferente. Vemos que es al segundo de los eventos al que efectivamente le quedan menos de 5 días para que comience desde la fecha de comienzo establecida "2024-10-01 12:30:00.000". Vemos que los patrones cumplen su función, pues aunque para el tercer evento queden menos de 5 días también, ya es tarea del **patrón "MenosDe2h"** detectarlo.

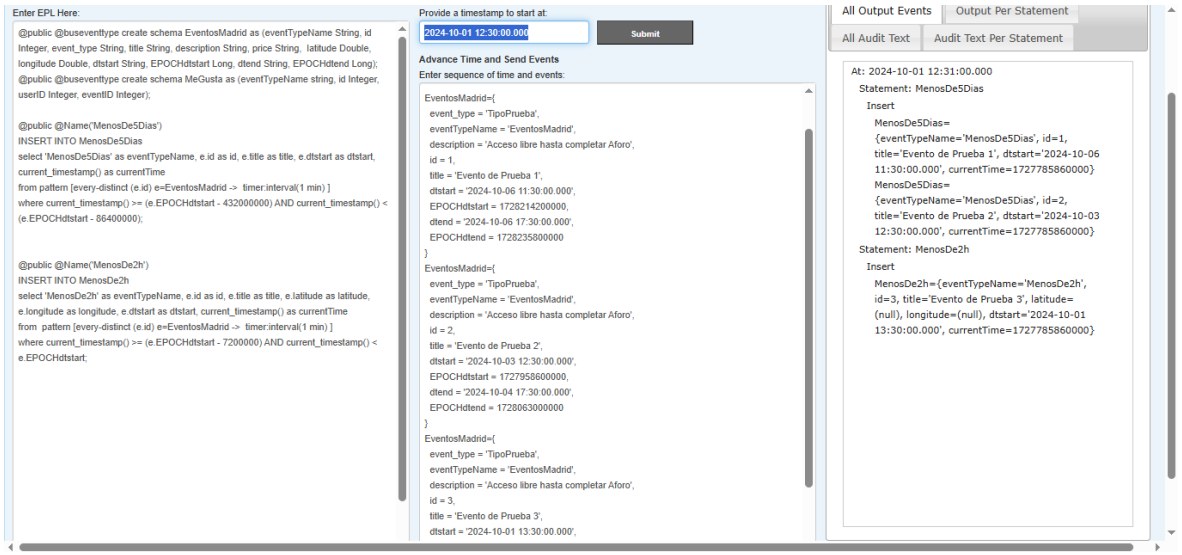


Figura 8.6: Prueba unitaria patrón “MenosDe5Dias” usando Esper EPL Online

- **MenosDe24h**: En la figura 8.7 podemos observar cómo hacemos que el momento de comienzo sea “2024-10-01 12:30:00.000”. Enviamos un evento “Evento de Prueba 2”. Vemos que, tras comprobar las condiciones del patrón después del primer intervalo de 1 minuto, en el momento “2024-10-01 12:31:00.000” se activa este patrón. Tiene un correcto funcionamiento, pues desde el momento establecido de comienzo hasta el momento “2024-10-02 10:30:00.000” en que empieza el evento de Madrid, quedarían unas 22 horas (concretamente 21 horas y 59 minutos), siendo menor de 24 horas. Además queda comprobado que, efectivamente, el patrón se comprueba cada minuto para cada evento de Madrid, y que una vez cumple la condición del patrón, deja de detectar eventos de Madrid de mismo identificador en cada intervalo de 1 minuto.

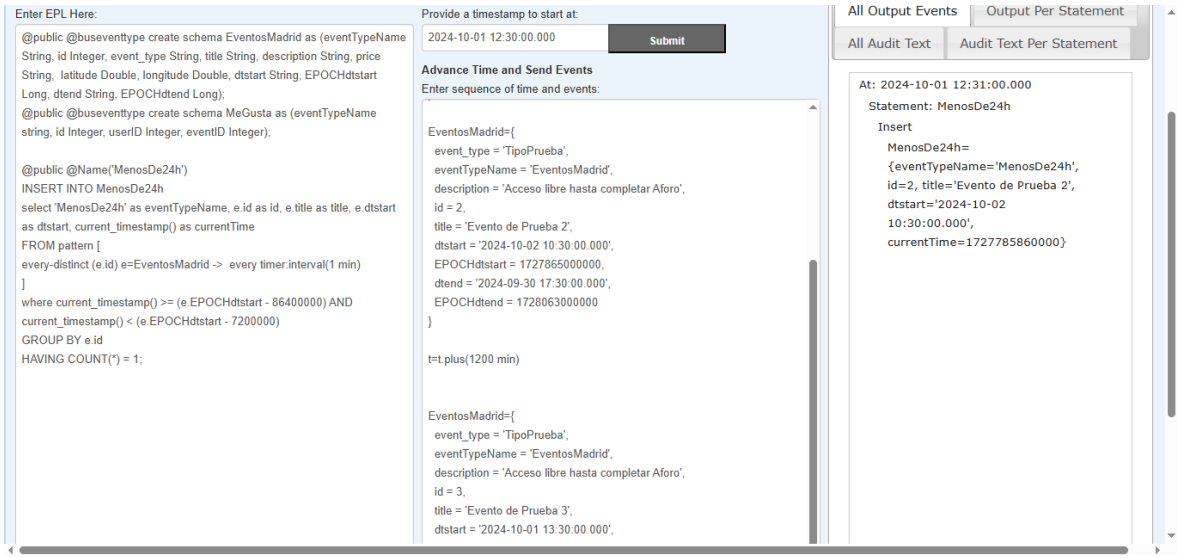


Figura 8.7: Prueba unitaria patrón “MenosDe24h” usando Esper EPL Online

- **MenosDe2h**: En la figura 8.8 podemos observar cómo hacemos que el momen-

to de comienzo sea “2024-10-01 12:30:00.000”. Enviamos un evento “Evento de Prueba 2”, que en principio vemos que es detectado por el **patrón MenosDe24h** como vimos anteriormente para la prueba de ese patrón. Vemos que, tras hacer que pasen 20 horas (1200 minutos), en el momento “2024-10-02 08:30:00.000” se activa este patrón, pues es a partir del momento en que quedan dos horas para el comienzo del evento, que tiene el valor “2024-10-02 10:30:00.000”. Comprobamos de esta forma que, efectivamente, el patrón se comprueba cada minuto para cada evento de Madrid, y que una vez cumple la condición del patrón, deja de detectar eventos de Madrid de mismo identificador en cada intervalo de 1 minuto.

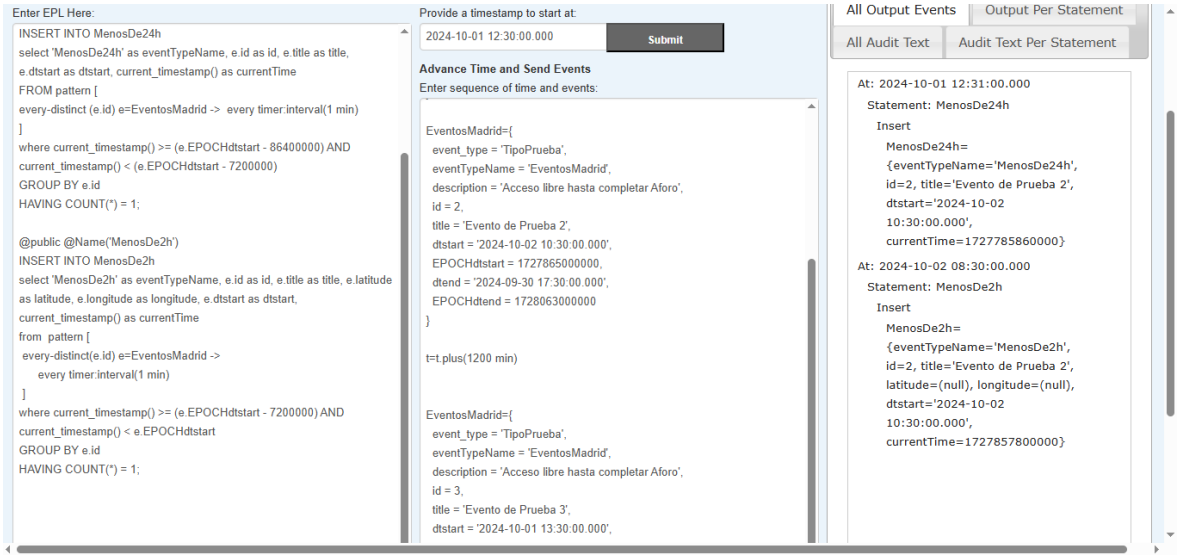


Figura 8.8: Prueba unitaria patrón “MenosDe2h” usando Esper EPL Online

- **MeGustaMenosDe5Dias**: En la figura 8.10 podemos observar el resultado de generar dos eventos simples de tipo “EventosMadrid ” y para cada uno de ellos un evento simple de tipo “MeGusta” de parte de un mismo usuario (mismo userID) cuando quedan menos de 5 días para el comienzo del evento.

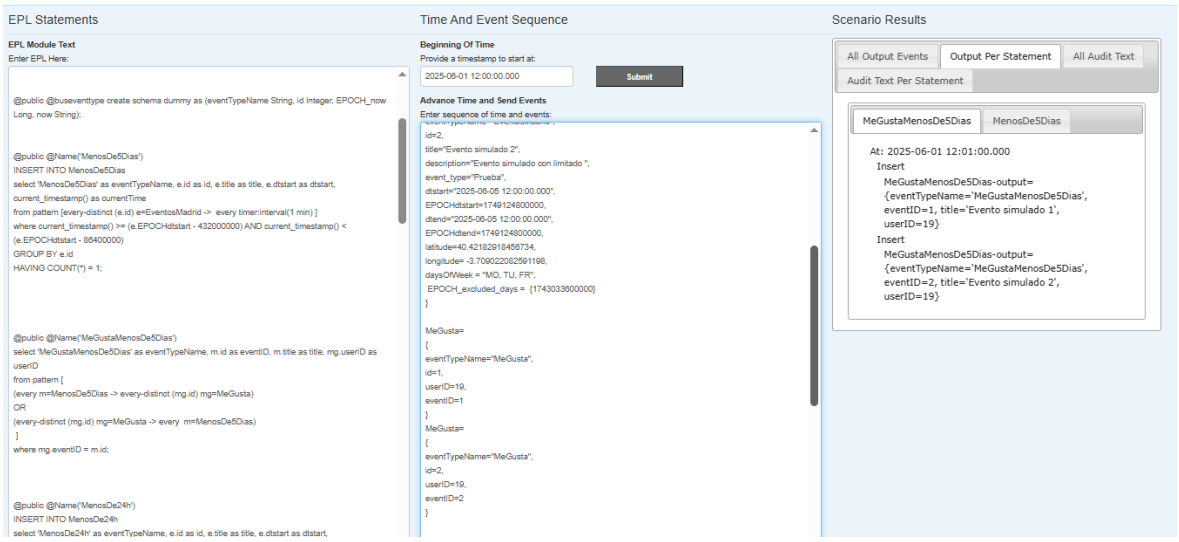


Figura 8.9: Prueba unitaria patrón "MeGustaMenosDe5Dias" usando Esper EPL Online

- **MeGustaMenosDe24h**: En la figura 8.10 podemos observar el resultado de generar dos eventos simples de tipo "EventosMadrid " y para cada uno de ellos un evento simple de tipo "MeGusta" de parte de un mismo usuario (mismo userID) cuando quedan menos de 24 días para el comienzo del evento.

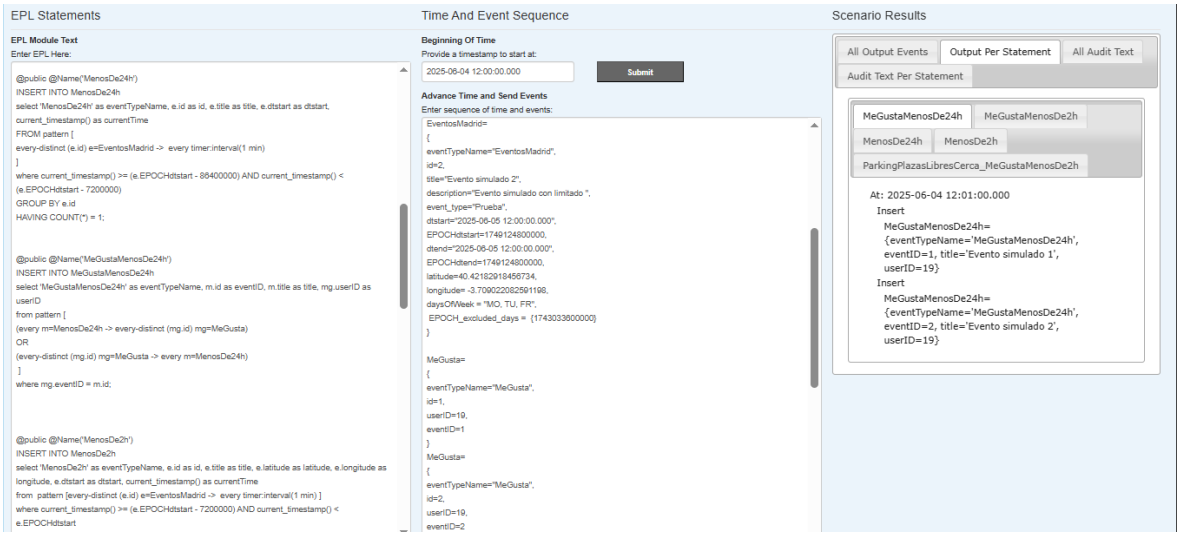


Figura 8.10: Prueba unitaria patrón "MeGustaMenosDe24h" usando Esper EPL Online

- **MeGustaMenosDe2h**: Vemos como para el mismo ejemplo de la prueba unitaria del patrón **MeGustaMenosDe2h** al pasar el tiempo, justo cuando quedan 2 horas para que el evento comience, se generan los dos eventos complejos de tipo MeGustaMenosDe2h (ver imagen 8.11).

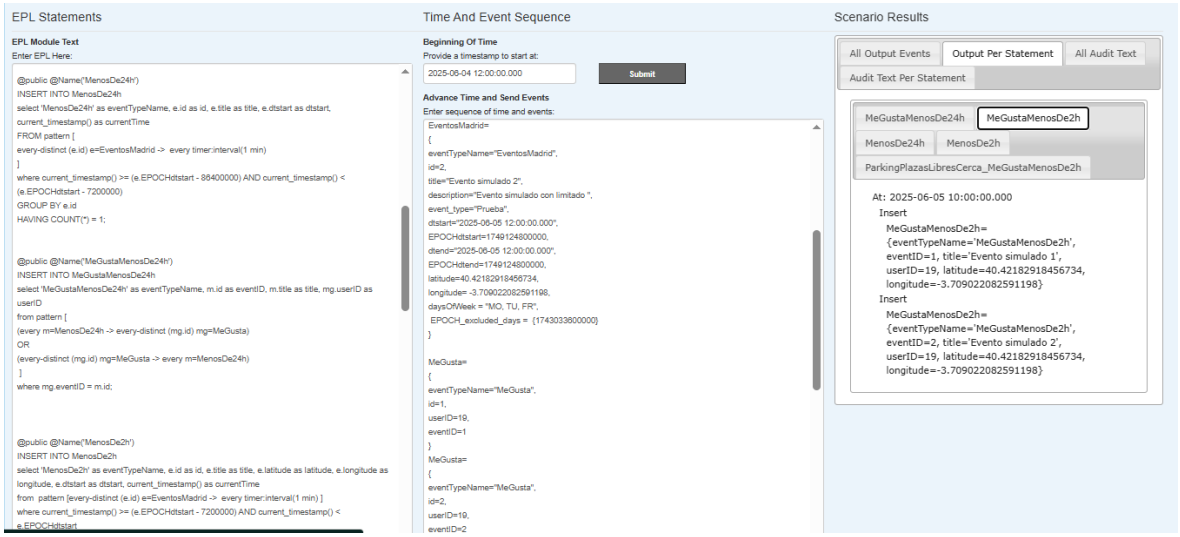


Figura 8.11: Prueba unitaria patrón "MeGustaMenosDe2h" usando Esper EPL Online

- **AforoLimitado**:En la figura 8.12 observamos cómo añadiendo este patrón se detecta aquel evento para el que hay aforo limitado, bien se indica en su precio mediante el campo "price" (por ejemplo "gratis hasta completar aforo") o bien en su campo "description" (por ejemplo "Acceso libre hasta completar Aforo").

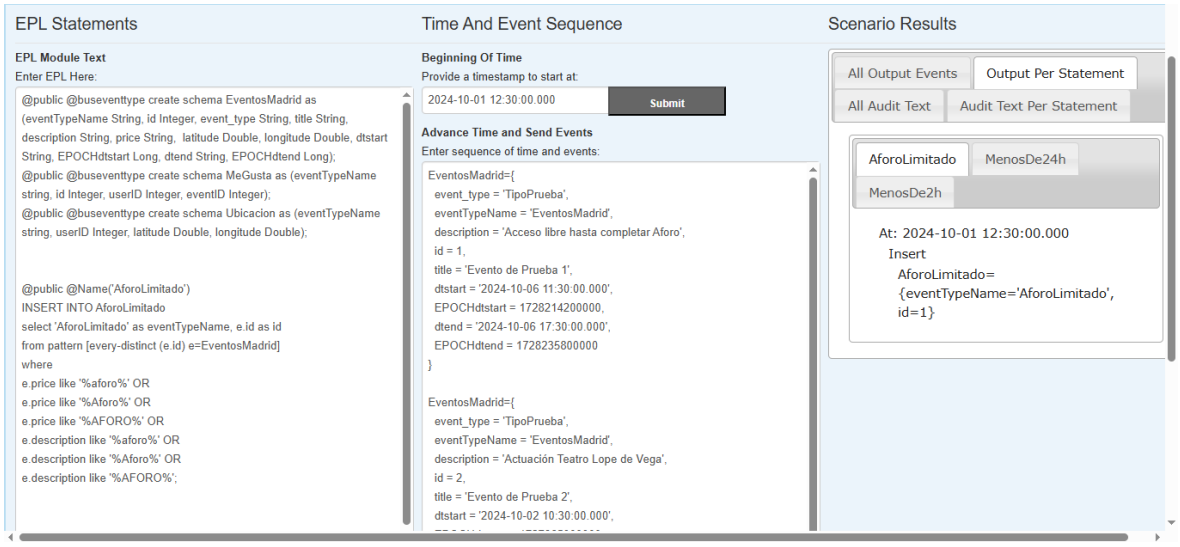


Figura 8.12: Prueba unitaria patrón "AforoLimitado" usando Esper EPL Online

- **AforoLimitado_MeGustaMenosDe24h**: En la figura 8.13 observamos cómo añadiendo este patrón se detecta la situación en que quedan menos de 24 horas para un evento al que un usuario concreto ha dado "me gusta", y para cuyo evento hay aforo limitado. Obsérvese que el evento de identificador con valor 2 no es detectado por este patrón, pues quedan menos de 2 horas para su comienzo como veremos en la prueba de el patrón **AforoLimitado_MeGustaMenosDe2h**.

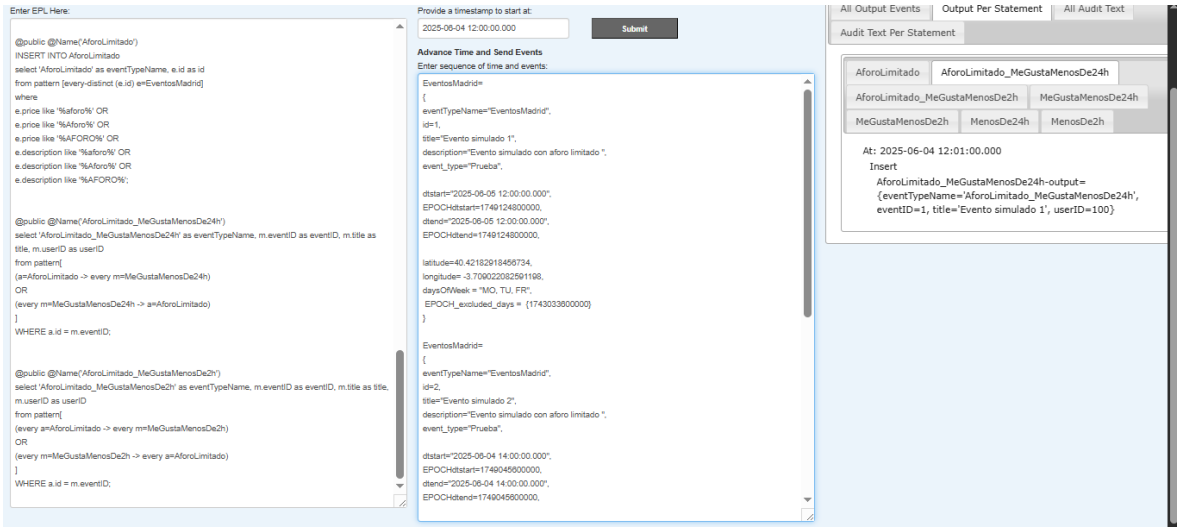


Figura 8.13: Prueba unitaria patrón "AforoLimitado_MeGustaMenosDe24h" usando Esper EPL Online

- **AforoLimitado_MeGustaMenosDe2h**: En la figura 8.14 observamos cómo añadiendo este patrón se detecta la situación en que quedan menos de dos horas para un evento al que un usuario concreto ha dado "me gusta", y para cuyo evento hay aforo limitado. Vemos que, aparte de detectarse el evento de identificador con valor 2 que no detectaba el patrón **AforoLimitado_MeGustaMenosDe24h**, al pasar 1 día detectará el evento de identificador con valor 1 que sí detectaba ese patrón, justo cuando quedan 2 horas para su comienzo.

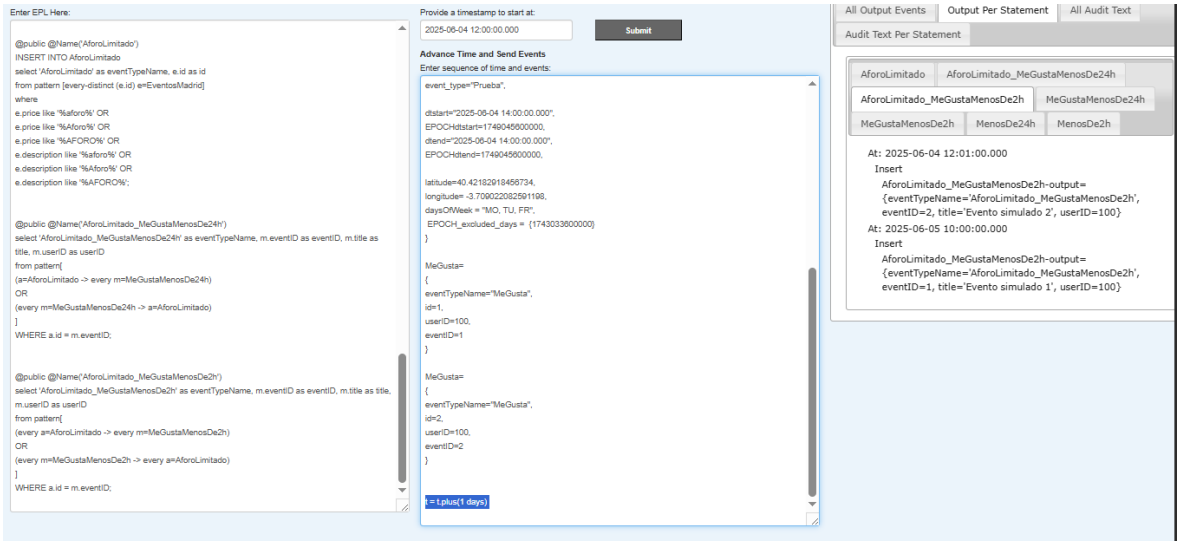


Figura 8.14: Prueba unitaria patrón "AforoLimitado_MeGustaMenosDe2h" usando Esper EPL Online

- **ParkingPlazasLibresCerca_MeGustaMenosDe2h**: Debido a las limitaciones de la herramienta empleada, probaremos el funcionamiento de este patrón en el apartado Pruebas de integración de patrones Esper EPL, Mule ESB y Rab-

bitMQ usando nITROGEN, ya que depende de la Implementación de la clase Java Haversine y no es posible definirla en esta herramienta.

- **ParkingCompleto**: Observemos la figura 8.15. Para la prueba de este patrón se ha simulado la detección de un evento complejo por parte del **patrón "ParkingPlazasLibresCerca_MeGustaMenosDe2h"** definiendo un esquema y enviando un evento simple con dicho nombre y los atributos que se seleccionarían en el patrón original. Vemos como, tras unos 30 minutos llegaría un evento simple "Parking" con su atributo "freeParkingSpaces " a 0. Es entonces cuando se este patrón se activa.

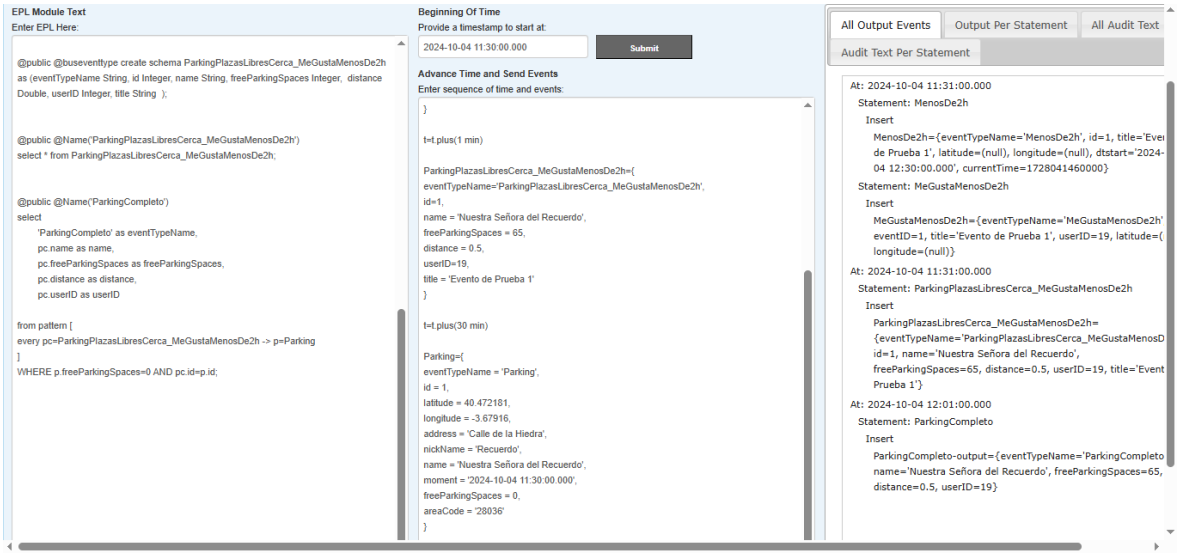


Figura 8.15: Prueba unitaria patrón "ParkingCompleto" usando Esper EPL Online

- **MenosDe2hCerca**: Debido a las limitaciones de la herramienta empleada, probaremos el funcionamiento de este patrón en el apartado Pruebas de integración de patrones Esper EPL, Mule ESB y RabbitMQ usando nITROGEN, ya que depende de la Implementación de la clase Java Haversine y no es posible definirla en esta herramienta.

- **Detecta día de la semana**: Observemos la figura 8.16. Para esta prueba se establece como hora de inicio las 23:57 del día 23-03-2025, cuyo día es un Domingo. Vemos como al pasar una hora, se detecta el patrón Lunes. Y vemos como al aumentar 14 días, se detectan los dos Lunes siguientes también.

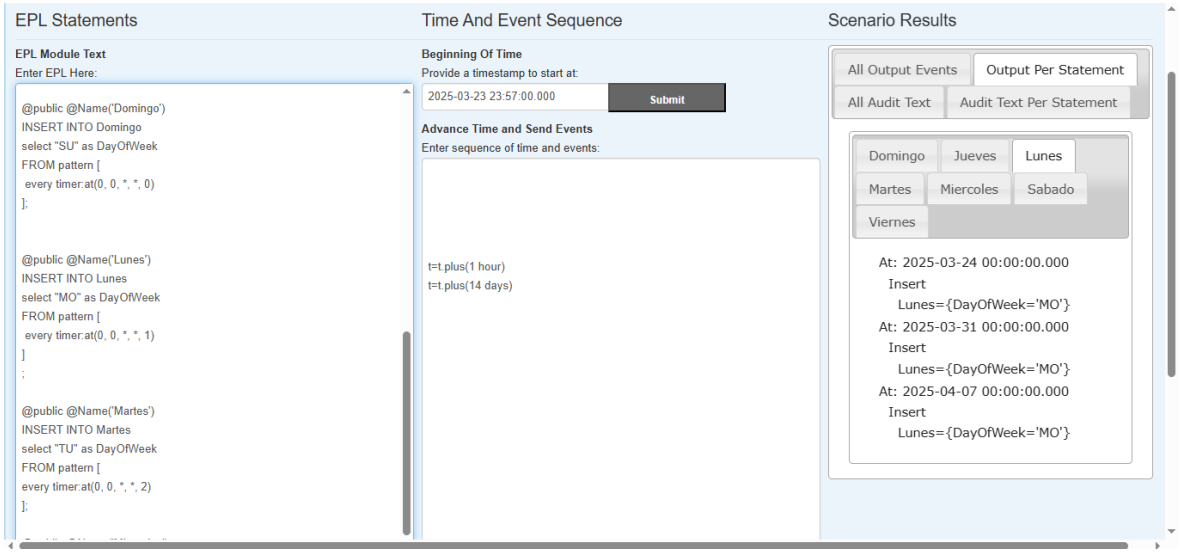


Figura 8.16: Prueba unitaria patrón "Lunes" usando Esper EPL Online

- **EventoEnCurso**: Observemos la figura 8.16. Vemos que se establece como momento de inicio el dado por la hora 23:59:00 del día 26-03-2025. El evento de identificador de valor 1 tiene por fecha de inicio "2025-03-27 00:00:00.000" y por fecha de fin "2025-03-30 10:00:00.000". Se detecta entonces al pasar un minuto desde el momento inicial que este patrón está en curso. Esto sucede ya que hemos realizado un incremento de tiempo con "t.plus()" para que se detecten los otros patrones. Si nos fijamos, también se detecta un segundo evento de Madrid con misma fecha de inicio y de fin, que se insertó en el flujo de eventos para probar que el patrón funcionara para cada evento de Madrid entrante.

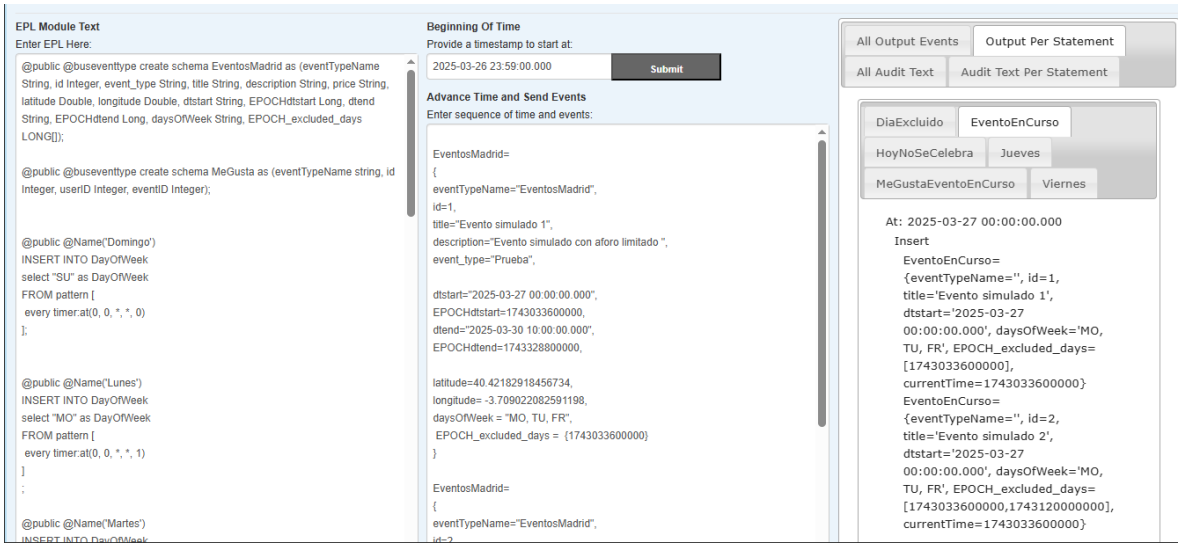


Figura 8.17: Prueba unitaria patrón "EventoEnCurso" usando Esper EPL Online

- **MeGustaEventoEnCurso**: Seguimos teniendo en cuenta los dos eventos de Madrid insertados para probar el patrón **EventoEnCurso**. Vemos como tras

enviar dos eventos simples de tipo **MeGusta**, uno para cada uno de ellos, se detecta este patrón.

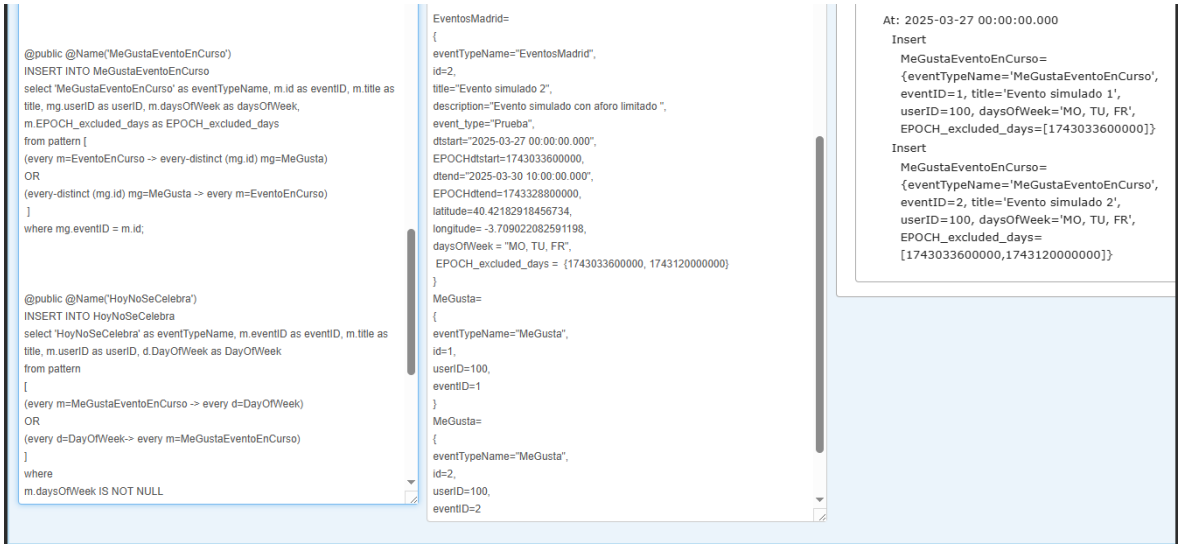


Figura 8.18: Prueba unitaria patrón "MeGustaEventoEnCurso" usando Esper EPL Online

- **HoyNoSeCelebra**: Siguiendo con los eventos simples y complejos insertados en el flujo de eventos para probar los patrones **EventoEnCurso** y **MeGustaEventoEnCurso**, vemos en la imagen 8.19 cómo al haberse detectado el evento complejo **Jueves** (se detecta también viernes porque incrementamos 2 días para probar también los patrones dedicados a detectar el día de la semana) y el evento complejo **MeGustaEventoEnCurso**, se detecta este patrón. Esto ocurre ya que los dos eventos de Madrid insertados tienen como valor de su campo "daysOfWeek" la cadena "MO, TU, FR". Esto representa los días de la semana que el evento se celebra. Vemos que el Jueves (TH) no está incluido.

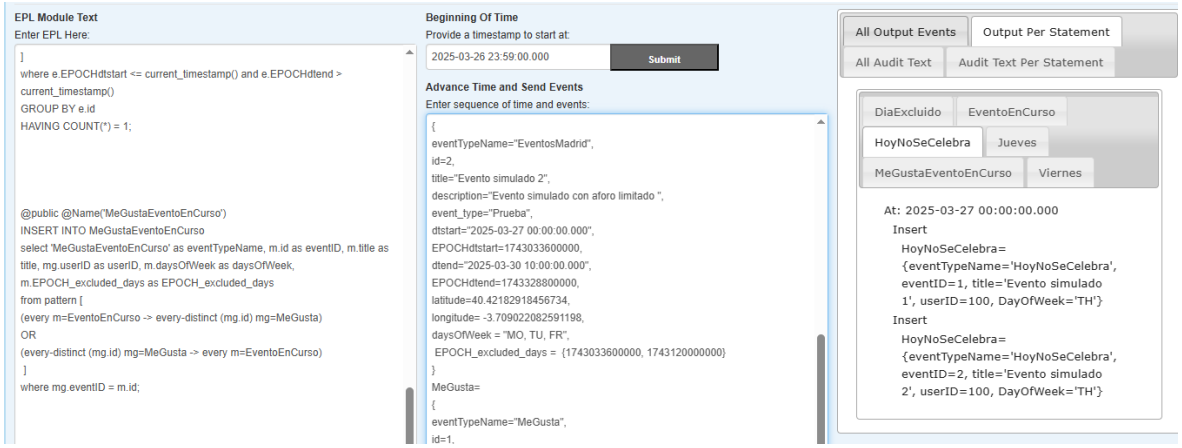


Figura 8.19: Prueba unitaria patrón "HoyNoSeCelebra" usando Esper EPL Online

8.2. Pruebas de integración

Probamos el funcionamiento de los componentes al trabajar de manera conjunta para verificar que satisfaga los Requisitos no funcionales del proyecto.

8.2.1. Pruebas de integración de patrones Esper EPL, Mule ESB y RabbitMQ usando nITROGEN

La implementación de la Clase Java "SendExpoNotification" comprobaba la existencia de un "me gusta" en la base de datos para notificar al usuario de la detección de aquellos patrones que comprobaban si un evento lo había recibido por parte de un usuario. Debido a ello, para estos patrones tanto el evento simple "EventosMadrid" como "MeGusta" no podrán simplemente simulados por nITROGEN, sino que deben de insertarse en la base de datos.

- **DetectaInicioSesionFallido**: Podemos observar en la imagen 8.20 cómo el patrón se activa justo cuando se recoge el evento simple **"InicioSesionFallido"** número 3 de la cola. Se han enviado con nITROGEN esos 3 eventos separados entre sí por apenas unos segundos, por lo que efectivamente este patrón se activa cuando detecta 3 eventos simples de tipo **"InicioSesionFallido"** seguidos en no más de un minuto.

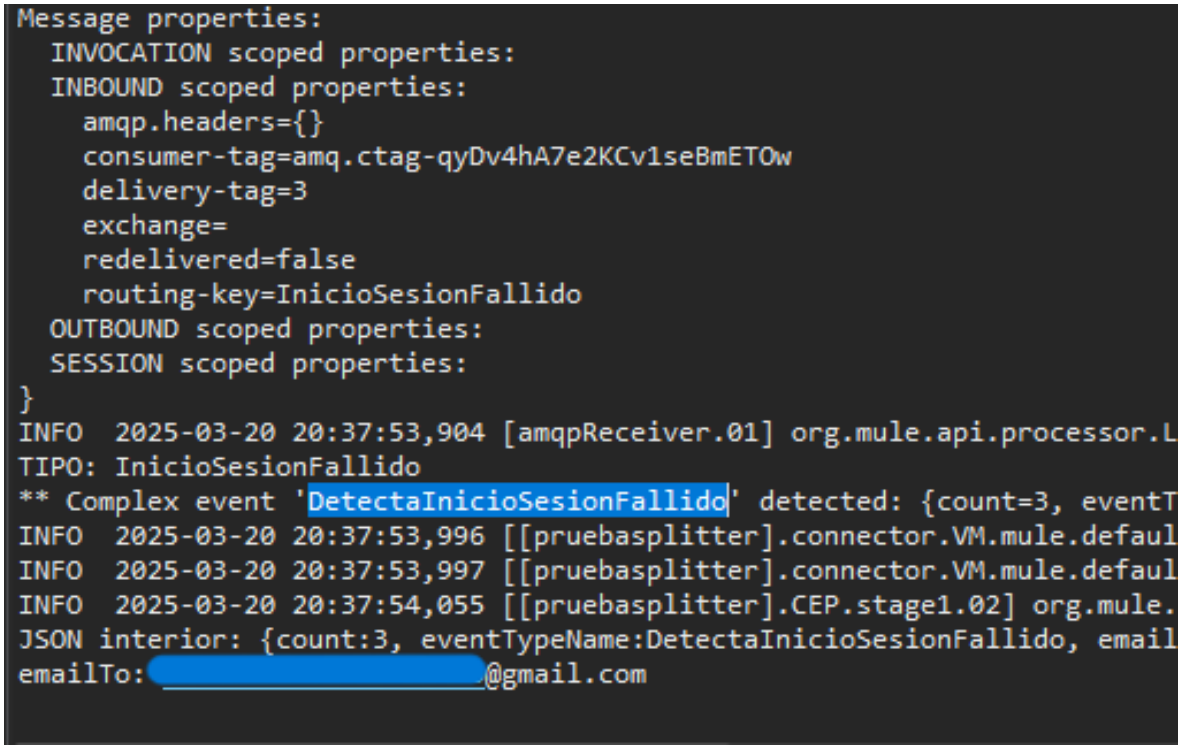


Figura 8.20: Prueba de integración patrón "InicioSesionFallido" desde Mule Anypoint

En la imagen 8.21 vemos como, efectivamente, en el momento "20:37:53" tras detectarse el evento complejo, se notifica por correo al usuario.

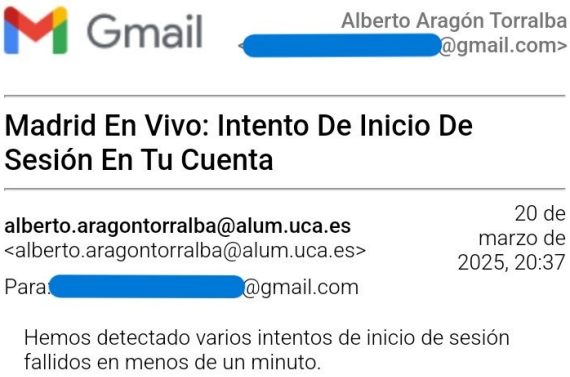


Figura 8.21: Aviso sobre el evento complejo "InicioSesionFallido" desde Gmail

- **DetectaFuerzaBruta**: Vemos en la imagen 8.22 la configuración global que se ha usado para nITROGEN, de manera que se mande un número considerable de eventos simples en un plazo de 15 segundos.

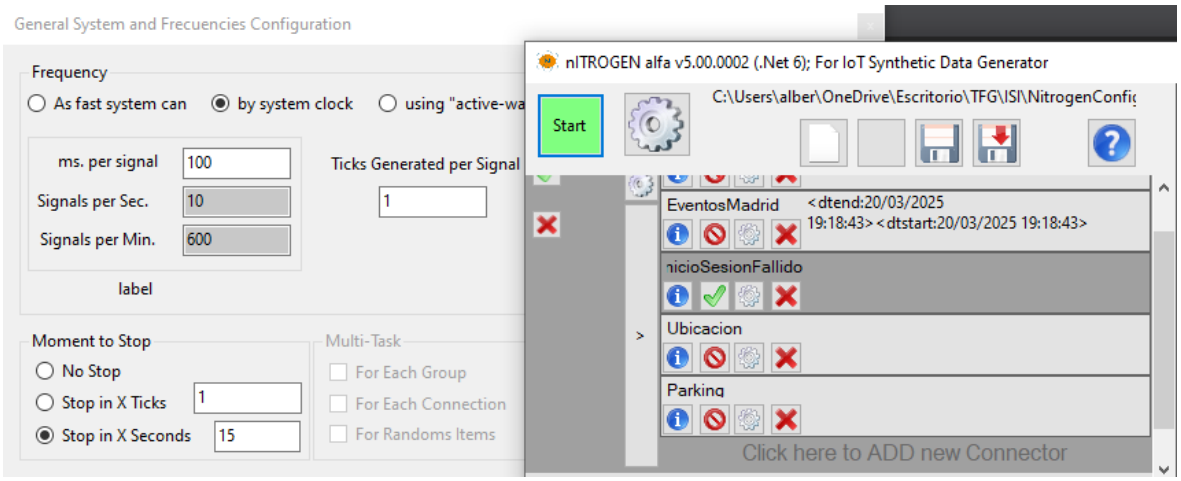


Figura 8.22: Configuración global de nITROGEN para la prueba de integración del patrón "DetectaFuerzaBruta"

Podemos observar en la imagen 8.23 cómo el patrón se activa justo cuando se recoge el evento simple **"InicioSesionFallido"** número 100 de la cola. Se demuestra con la imagen proporcionada anteriormente, que muestra la configuración global para nITROGEN, que se envían en un plazo de 15 segundos y por tanto de menos de un minuto.

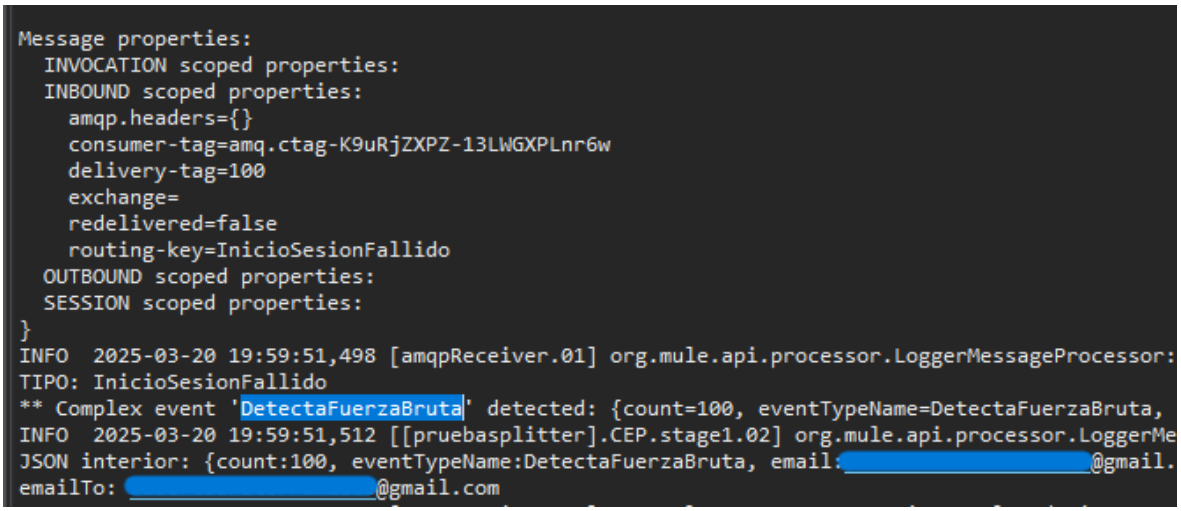


Figura 8.23: Prueba de integración patrón "DetectaFuerzaBruta" desde Mule Anypoint

En la imagen 8.24 vemos como, efectivamente, en el momento "19:59:51" tras detectarse el evento complejo, se notifica por correo al usuario.



Figura 8.24: Aviso sobre el evento complejo "DetectaFuerzaBruta" desde Gmail

Se puede observar en la imagen 8.25 que también es enviado en este caso otro correo notificando sobre la detección del evento complejo **"InicioSesionFallido"**, puesto que en menos de 1 minuto se han detectado 100 eventos simples, y por tanto, también este patrón ha detectado 3.

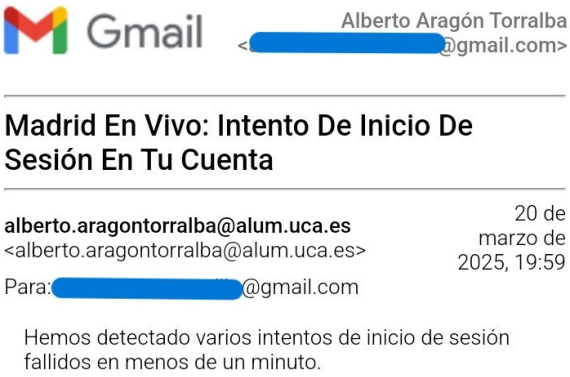


Figura 8.25: Aviso sobre el evento complejo "InicioSesionFallido" también cuando se detecta "DetectaFuerzaBruta" desde Gmail

- **UbicacionSospechosa**: Vemos en la imagen 8.26 la consola de Virtual Studio Code en el momento en que desde el front-end(consola a la derecha) se obtienen los datos de ubicación del dispositivo y se realiza una petición POST al backend(consola a la izquierda). Vemos que la localización enviada es desde España.

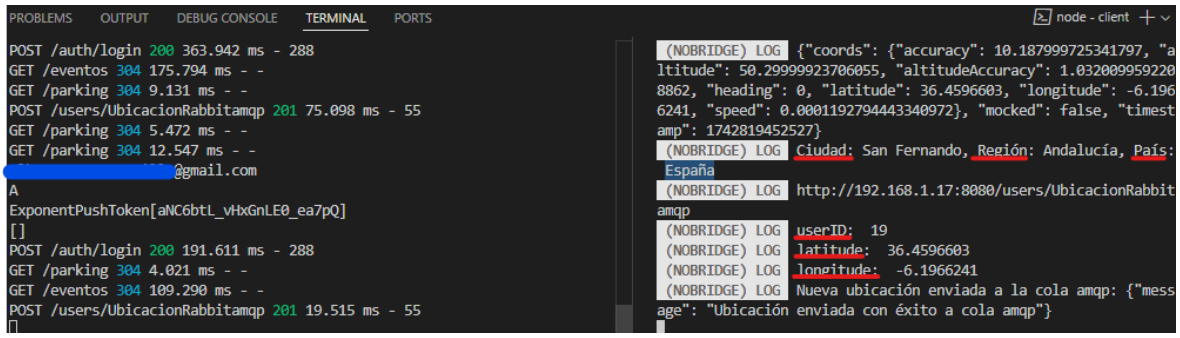


Figura 8.26: Envío de evento simple "Ubicación" desde la consola de Virtual Studio Code

A continuación generamos desde nITROGEN otro evento simple de tipo "Ubicacion" pero esta vez con otra ubicación, simulando que es enviada desde Budapest, en Hungría (ver imagen 8.26).

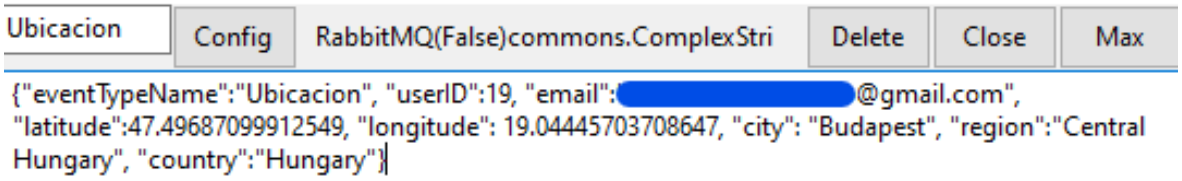


Figura 8.27: Generación de evento simple "Ubicación" con otro valor en el campo "country" desde nITROGEN

Vemos en la figura 8.28 que al enviarse el segundo evento simple de ubicación, este patrón detecta el evento complejo.

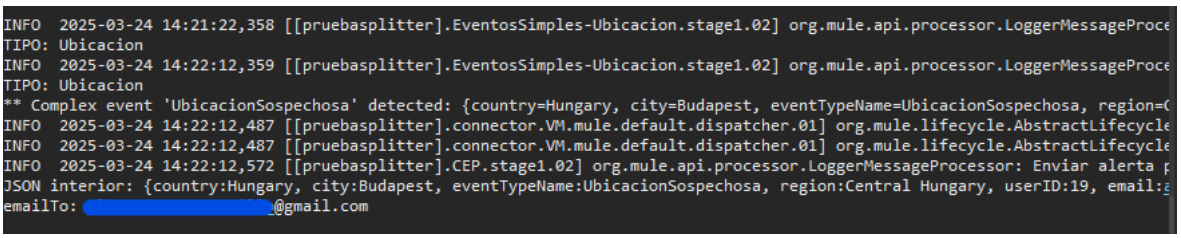


Figura 8.28: Prueba de integración patrón "UbicacionSospechosa" desde Mule Anypoint

En la imagen 8.29 vemos como, efectivamente, en el momento "14:22:12" tras detectarse el evento complejo, se notifica por correo al usuario.



Figura 8.29: Aviso sobre el evento complejo "UbicacionSospechosa" desde Gmail

- **MenosDe5Dias**: Se ha generado con nITROGEN el envío a la respectiva cola RabbitMQ un evento simple de tipo "EventosMadrid" con fecha de inicio a "21/03/2025 15:00:00", con valor 1742565600000 en formato EPOCH en milisegundos.

  Vemos que se recoge de la cola y se envía como Evento simple al motor CEP. En el momento "2025-03-17 12:27:03" el patrón comprueba si dicho evento simple cumples su condición. Comprobamos que, tras el primer intervalo de 1 minuto, en el momento "2025-03-18 19:14:12", se evalua la condición del patrón a verdadera y se detecta el evento complejo **MenosDe5Dias**.

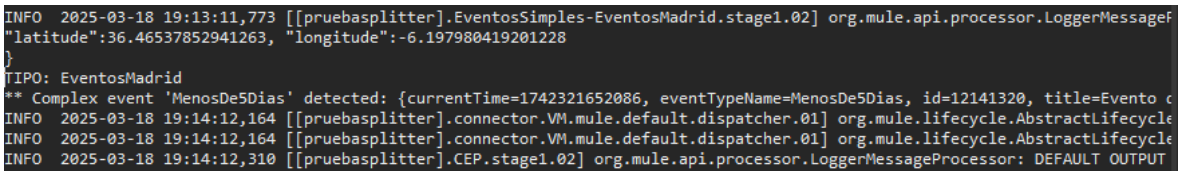


Figura 8.30: Prueba de integración patrón "MenosDe5Dias" desde Mule Anypoint

- **MeGustaMenosDe5Dias**: Se ha insertado en la base de datos un registro para la tabla "eventos", que al ejecutar la aplicación Mule y por tanto el **flujo "InicializacionEventosDB-> Cola"**, provoca el envío a la respectiva cola RabbitMQ un evento simple de tipo **"EventosMadrid"** con fecha de inicio a "21/03/2025 15:00:00", con valor 174256560000 en formato EPOCH en milisegundos.

  También se inserta un registro en la tabla "intereses" con su atributo "eventID" de mismo valor que el identificador del evento simple "EventosMadrid" enviado justo antes. Se genera también un evento simple **"MeGusta"** al ejecutar la aplicación Mule y por tanto el **flujo "InicializacionEventosDB-> Cola"**.

  Ambos se envían cada uno desde su respectiva cola como eventos simples al motor CEP.

  Comprobamos que, tras el primer intervalo de 1 minuto desde que llega el evento de Madrid, se evalua la condición del primer patrón a verdadera y se detecta el evento complejo **MenosDe5Dias**. Tras ello, este segundo patrón detecta el evento complejo **"MeGustaMenosDe5Dias"**.

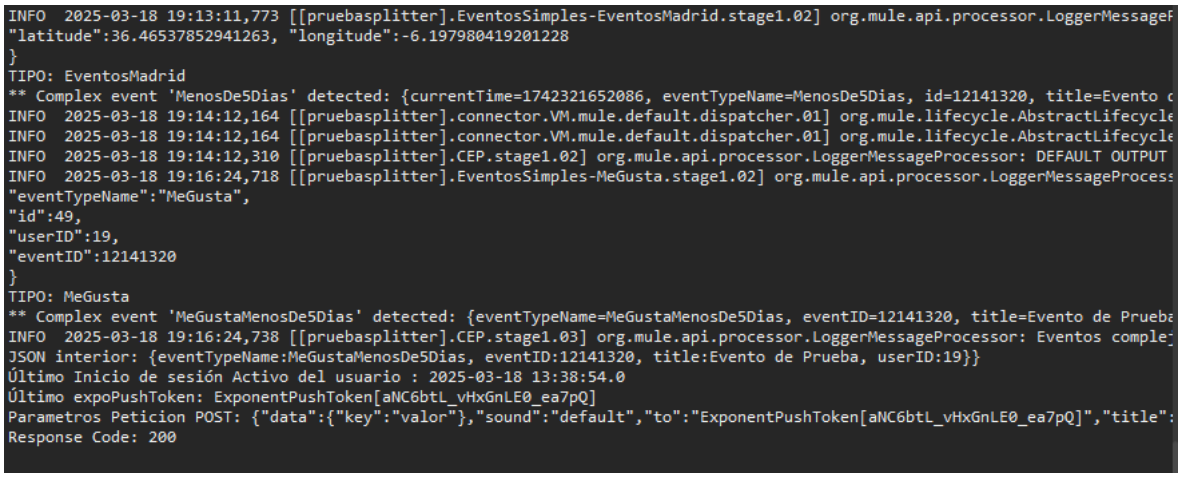


Figura 8.31: Prueba de integración patrón "MeGustaMenosDe5Dias" desde Mule Anypoint

Tras ello y como podemos observar en la imagen 8.36, este evento es procesado por la Clase Java "SendExpoNotification" y se envía la correspondiente notificación push al dispositivo móvil en que el usuario tiene una sesión activa.

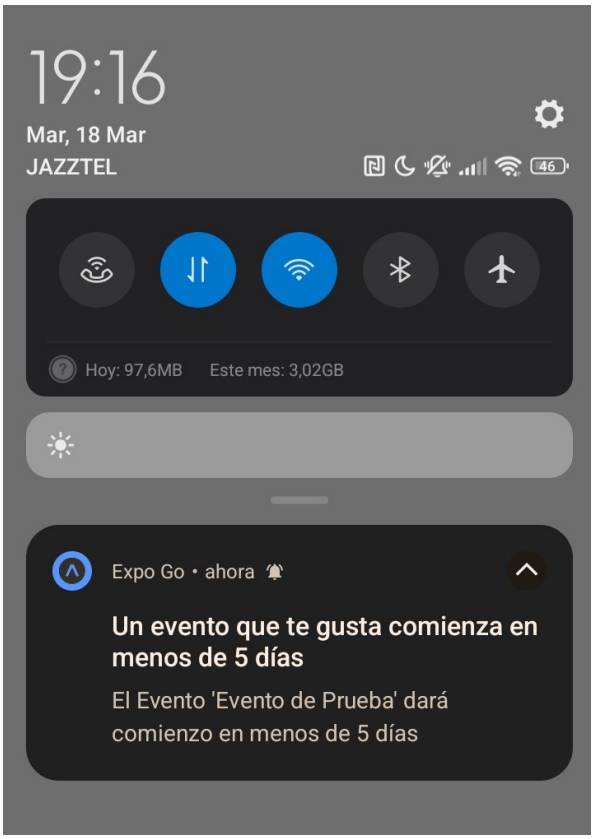


Figura 8.32: Prueba de integración de notificación push del patrón "MeGustaMenosDe5Dias" desde dispositivo móvil

- **AforoLimitado**: Se ha generado con nITROGEN el envío a la respectiva cola RabbitMQ un evento simple de tipo "EventosMadrid" con su campo "description" con el valor "Acceso libre hasta completar Aforo". Vemos en la imagen 8.33 como justo tras enviarse como evento simple al motor CEP, este patrón detecta que cumple su condición generando el Evento complejo **"AforoLimitado"**.

```
INFO  2025-03-17 17:49:00,319 [[pruebasplitter].EventosSimples-EventosMadrid.stage1.02] org.mule.api.processor.LoggerMessageP
TIPO: EventosMadrid
** Complex event 'AforoLimitado' detected: {eventTypeName=AforoLimitado, id=12141320}
INFO  2025-03-17 17:49:00,658 [[pruebasplitter].connector.VM.mule.default.dispatcher.01] org.mule.lifecycle.AbstractLifecycle
INFO  2025-03-17 17:49:00,658 [[pruebasplitter].connector.VM.mule.default.dispatcher.01] org.mule.lifecycle.AbstractLifecycle
INFO  2025-03-17 17:49:00,768 [[pruebasplitter].CEP.stage1.02] org.mule.api.processor.LoggerMessageProcessor: DEFAULT OUTPUT
```

Figura 8.33: Prueba de integración patrón "AforoLimitado" desde Mule Anypoint

- **MenosDe24h**: Se ha generado con nITROGEN el envío a la respectiva cola RabbitMQ un evento simple de tipo "EventosMadrid" con fecha de inicio a "18/03/2025 12:25:00", con valor 1742297100000 en formato EPOCH en milisegundos.

  Vemos que se recoge de la cola y se envía como Evento simple al motor CEP en el momento "2025-03-17 12:27:03". Comprobamos que, tras el primer intervalo de 1 minuto desde que llega el evento de Madrid, se evalua la condición del patrón a verdadera y se detecta el evento complejo **MenosDe24h**.

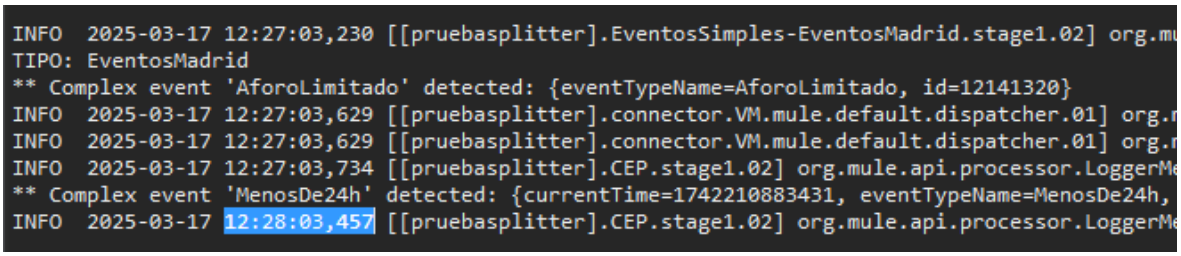


Figura 8.34: Prueba de integración patrón "MenosDe24h" desde Mule Anypoint

- **MeGustaMenosDe24h**: Se ha generado con nITROGEN el envío a la respectiva cola RabbitMQ un evento simple de tipo **"EventosMadrid"** con fecha de inicio a "18/03/2025 12:25:00", con valor 1742297100000 en formato EPOCH en milisegundos. También se envía un evento simple **"MeGusta"** desde nITROGEN, con su atributo "eventID" de mismo valor que el identificador del evento simple "EventosMadrid" enviado justo antes.

  Vemos que ambos se recogen de su respectiva cola y se envían como eventos simples al motor CEP.

  Comprobamos que, tras el primer intervalo de 1 minuto desde que llega el evento de Madrid, se evalua la condición del patrón a verdadera y se detecta el evento complejo **MenosDe24h**. Tras ello, este patrón detecta el evento complejo **"MeGustaMenosDe24h"** dado el envío del evento simple **"me gusta"** anteriormente.

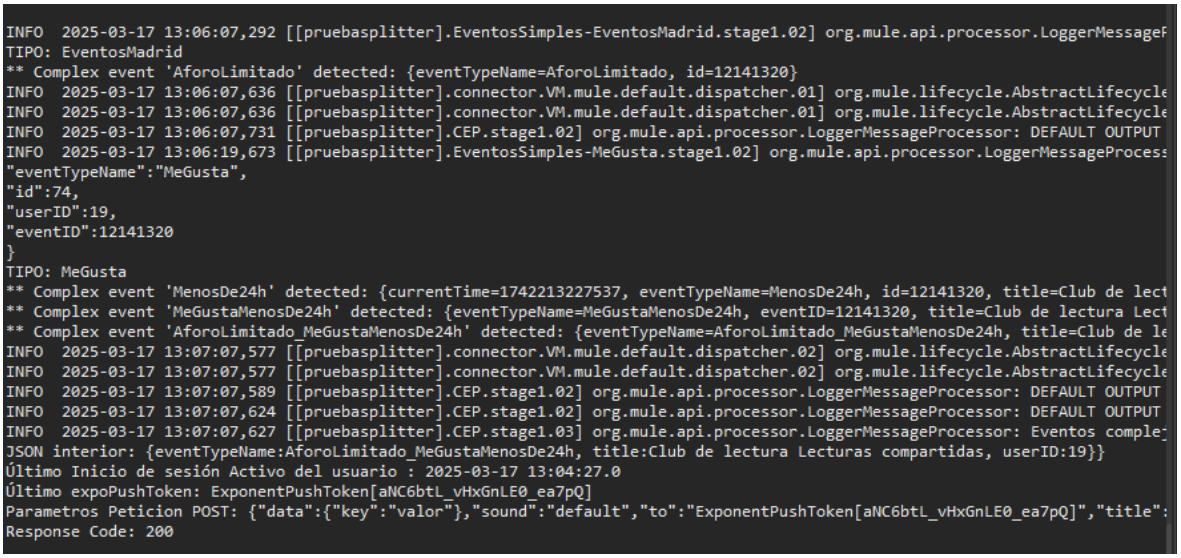


Figura 8.35: Prueba de integración patrón "MeGustaMenosDe24h" desde Mule Anypoint

- **AforoLimitado_MeGustaMenosDe24h**: Observamos la imagen 8.35 utilizada para mostrar el resultado de las pruebas de integración del **patrón "MeGustaMenosDe24h"**. Se ha insertado en la base de datos un registro para la tabla "eventos", que al ejecutar la aplicación Mule y por tanto el **flujo "InicializacionEventosDB-> Cola"**, provoca el envío a la respectiva cola RabbitMQ un

evento simple de tipo **“EventosMadrid”** con fecha de inicio a “18/03/2025 12:25:00”, con valor 1742297100000 en formato EPOCH en milisegundos. También se inserta un registro en la tabla “intereses” con su atributo “eventID” de mismo valor que el identificador del evento simple “EventosMadrid” enviado justo antes. Se genera también un evento simple **“MeGusta”** al ejecutar la aplicación Mule y por tanto el **flujo “InicializacionEventosDB-> Cola”**. Ambos se envían cada uno desde su respectiva cola como eventos simples al motor CEP. Justo tras la llegada del evento simple **“EventosMadrid”** se detecta el evento complejo **“AforoLimitado”**. Comprobamos que, tras el primer intervalo de 1 minuto desde que llega el evento simple **“EventosMadrid”**, se evalua la condición del patrón a verdadera y se detecta el evento complejo **MenosDe24h**. Tras ello, este patrón detecta el evento complejo **“MeGustaMenosDe24h”** dado el envío del evento simple **“me gusta”** anteriormente. Este patrón detecta entonces el evento complejo **“AforoLimitado_MeGustaMenosDe24h”**. Tras ello y como podemos observar en la imagen 8.36, este evento es procesado por la Clase Java “SendExpoNotification” y se envía la correspondiente notificación push al dispositivo móvil en que el usuario tiene una sesión activa.

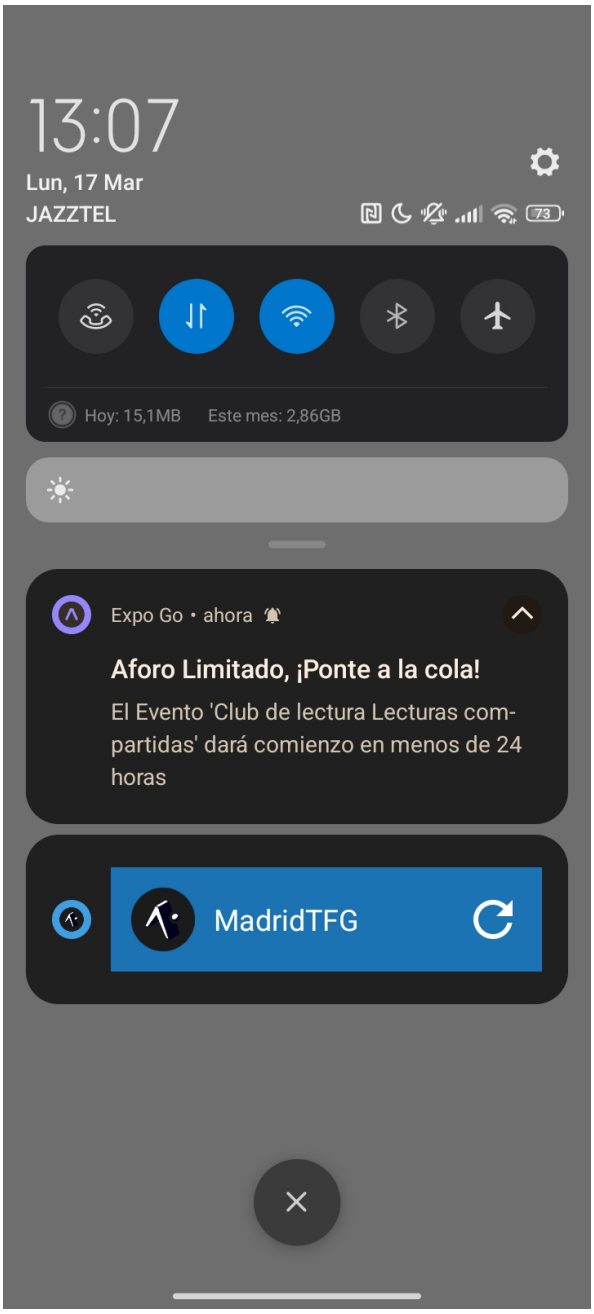


Figura 8.36: Prueba de integración de notificación push del patrón "MeGustaMenosDe24h" desde dispositivo móvil

- **MenosDe2h**: Se ha insertado en la base de datos un registro para la tabla "eventos", que al ejecutar la aplicación Mule y por tanto el **flujo "InicializacionEventosDB-> Cola"**, provoca el envío a la respectiva cola RabbitMQ un evento simple de tipo **"EventosMadrid"** con fecha de inicio a "17/03/2025 19:00:00", con valor 1742297100000 en formato EPOCH en milisegundos.

  Observémos la imagen 8.37. Vemos que se recoge de la cola y se envía como Evento simple al motor CEP. Vemos que justo tras enviarse al motor CEP el evento simple "EventosMadrid" se detecta el evento complejo **"AforoLimitado"**, pues el evento tenía su atributo "description" con el valor "Acceso libre hasta com-

pletar Aforo" Comprobamos que, tras el primer intervalo de 1 minuto desde que llega el evento de Madrid, se evalua la condición del patrón a verdadera y se detecta el evento complejo **"MenosDe2h"**, pues quedan efectivamente menos de 2 horas desde el momento "2025-03-17 18:01:50" en que se envía el evento simple al motor CEP hasta la fecha de inicio del evento ya mencionada.

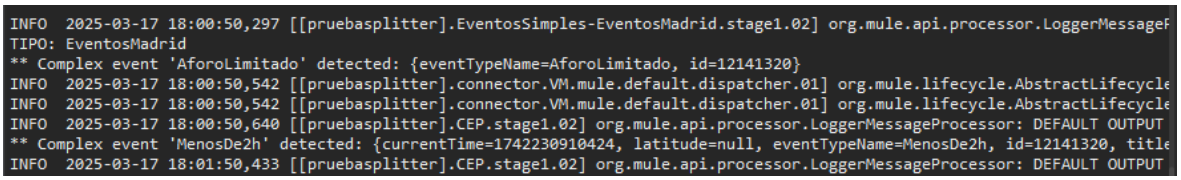


Figura 8.37: Prueba de integración patrón "MenosDe2h" desde Mule Anypoint

- **MeGustaMenosDe2h**: Observemos la imagen 8.38. Tras detectarse el evento complejo **"MenosDe2h"** de la prueba para el patrón de mismo nombre, También se inserta un registro en la tabla "intereses" con su atributo "eventID" de mismo valor que el identificador del evento simple "EventosMadrid" enviado justo antes. Para este caso en que la ejecución de la aplicación Mule no se reinicia y por tanto no se ejecutará el **flujo "InicializacionEventosDB-> Cola"**, se ha generado con nITROGEN el envío a la respectiva cola RabbitMQ de un evento simple de tipo "MeGusta" con sus atributos de mismo valor que los del registro que acabamos de insertar. Entonces este patrón de activa y detecta el evento complejo **"MeGustaMenosDe2h"**.

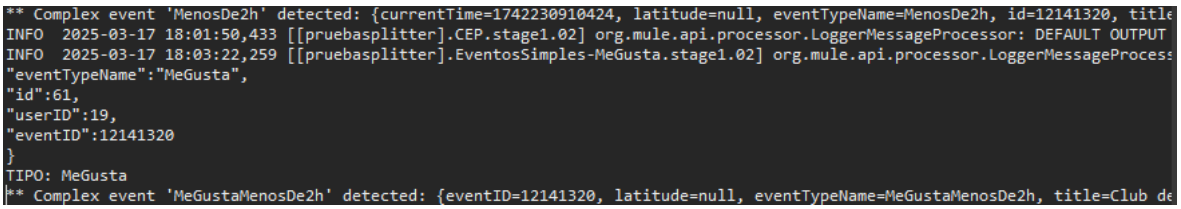


Figura 8.38: Prueba de integración patrón "MeGustaMenosDe2h" desde Mule Anypoint

- **AforoLimitado_MeGustaMenosDe2h**: Para esta prueba, seguimos con la misma ejecución que para las pruebas de los patrones **MenosDe2h** y **MeGustaMenosDe2h**. Si observamos la imagen 8.39, vemos que justo tras enviarse al motor CEP el evento simple "EventosMadrid" se detectaba el evento complejo **"AforoLimitado"**, pues el evento simple tenía su atributo "description" con el valor "Acceso libre hasta completar Aforo".

  Tras la detección del evento complejo **"MeGustaMenosDe2h"**, vemos cómo se detecta a su vez el evento complejo **"AforoLimitado_MeGustaMenosde2h"**.

  Como podemos observar en la imagen 8.40, este evento es procesado por la Clase Java "SendExpoNotification" y se envía la correspondiente notificación push al dispositivo móvil en que el usuario tiene una sesión activa.

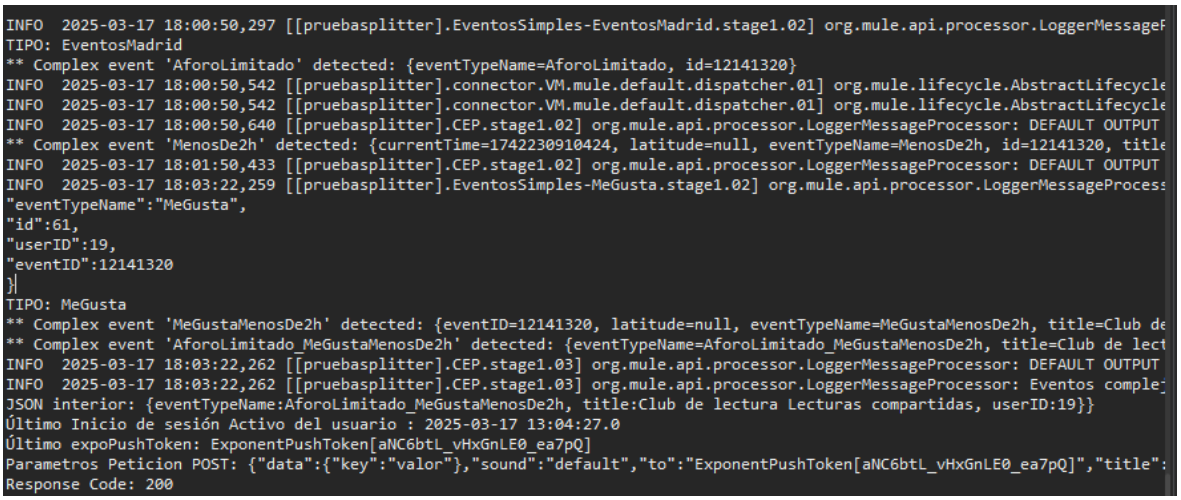


Figura 8.39: Prueba de integración patrón “AforoLimitado_MeGustaMenosde2h” desde Mule Anypoint

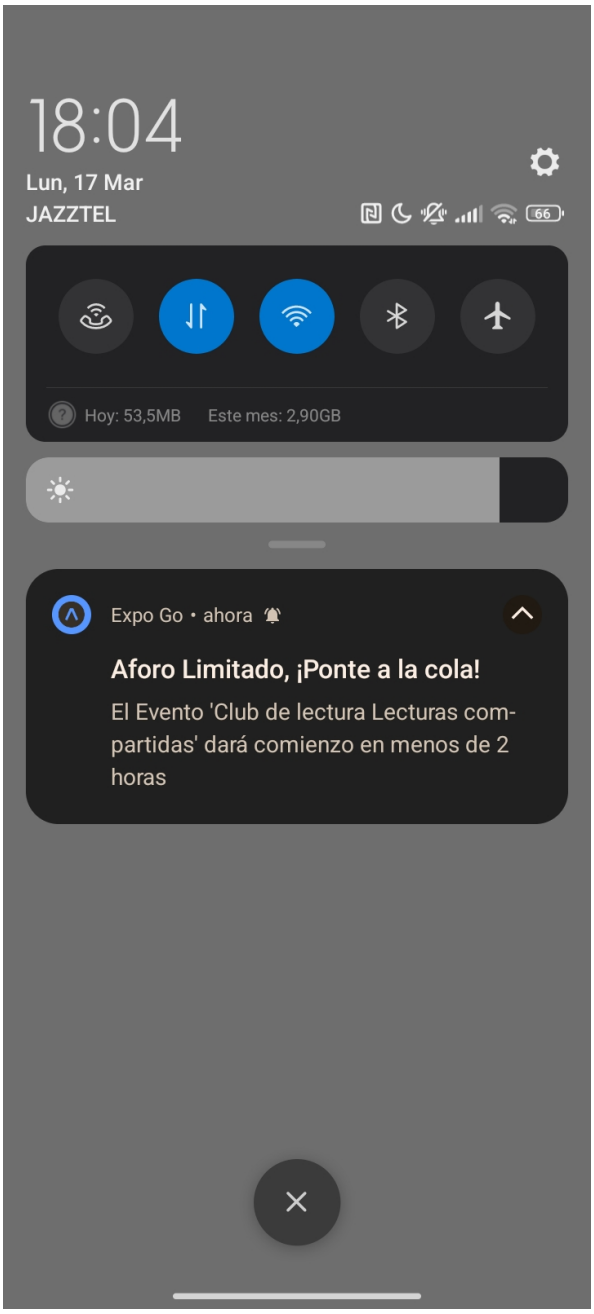


Figura 8.40: Prueba de integración de notificación push del patrón "AforoLimitado_MeGustaMenosde2h" desde dispositivo móvil

- **ParkingPlazasLibresCerca_MeGustaMenosDe2h**:

  Se ha insertado en la base de datos un registro para la tabla "eventos", que al ejecutar la aplicación Mule y por tanto el **flujo "InicializacionEventosDB-> Cola"**, provoca el envío a la respectiva cola RabbitMQ un evento simple de tipo **"EventosMadrid"** con fecha de inicio a "19/03/25 20:00:00", con valor 1742410800000 en formato EPOCH en milisegundos. Se ha establecido su localización como la del Teatro Lope de Vega en Madrid("latitude":40.42182918456734, "longitude": -3.709022082591198).

También se inserta un registro en la tabla "intereses" con su atributo "eventID" de mismo valor que el identificador del evento simple "EventosMadrid" enviado justo antes. Se genera también un evento simple **"MeGusta"** al ejecutar la aplicación Mule y por tanto el **flujo "InicializacionEventosDB-> Cola"**.

Ambos se envían cada uno desde su respectiva cola como eventos simples al motor CEP. Se detectan tras 1 minuto el evento complejo **"MenosDe2h"** y **"MeGustaMenosDe2h"**.

Por otro lado, en la imagen 8.41 observamos el cuerpo del evento simple simulado de tipo **"Parking"** que será enviado a su correspondiente cola y será procesado por el motor CEP, con una plaza libre y la localización del Parking Saba Plaza Mostenses, cerca de Gran Vía y del Teatro Lope de Vega.

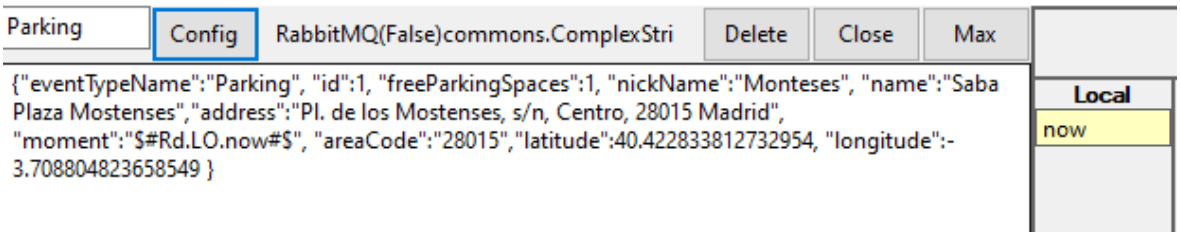


Figura 8.41: Evento simple de tipo "Parking" generado desde nITROGEN con localización correspondiente al Parking Saba Plaza Mostenses(Madrid)

Al haber seleccionado una ubicación céntrica para el evento, varios parking se encuentran a menos de un kilómetro de distancia de él. Bajo esa condición se detecta el evento complejo **"ParkingPlazasLibresCerca_MeGustaMenosDe2h"**, como se puede observar en la imagen 8.42 (el atributo "eventID" de valor 1 corresponde al evento de prueba insertado en la base de datos).

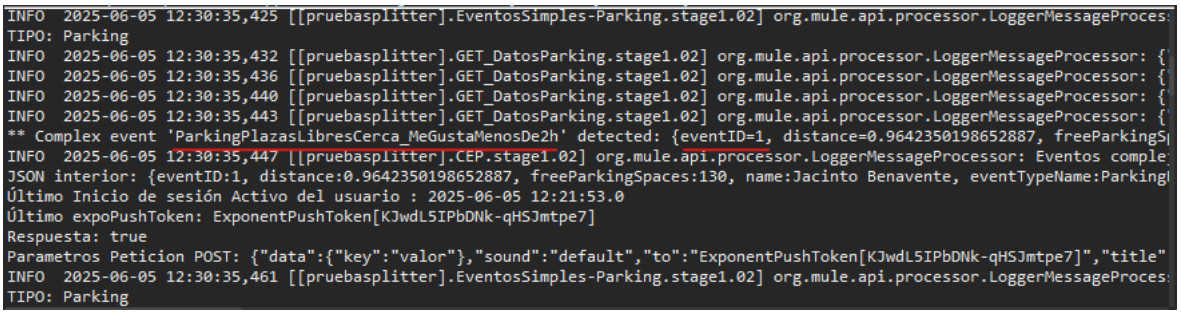


Figura 8.42: Prueba de integración patrón "ParkingPlazasLibresCerca_MeGustaMenosDe2h" desde Mule Anypoint

Como podemos observar en la imagen 8.43, los eventos complejos detectados de este tipo son procesados por la Clase Java "SendExpoNotification" y para cada uno se envía la correspondiente notificación push al dispositivo móvil en que el usuario tiene una sesión activa.

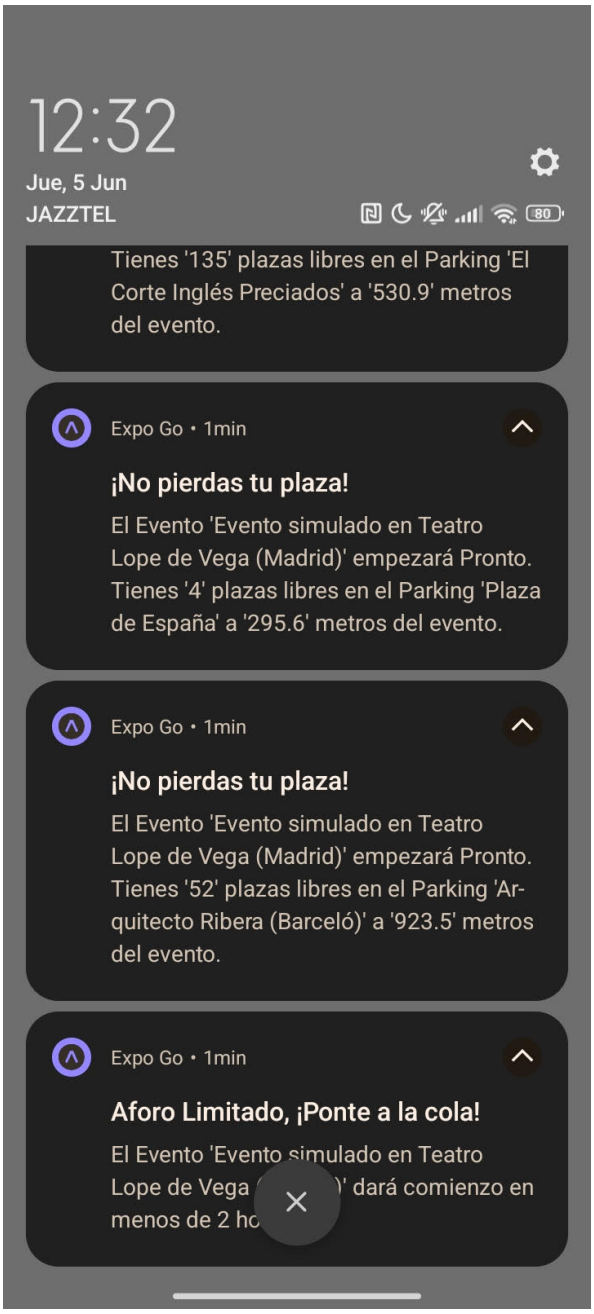


Figura 8.43: Prueba de integración de notificación push del patrón "ParkingPlazasLibresCerca_MeGustaMenosDe2h" desde dispositivo móvil

- **ParkingCompleto**: Obsérvese la imagen 8.44. Tras la detección del evento complejo **"ParkingPlazasLibresCerca_MeGustaMenosDe2h"**, enviamos otro evento simple de tipo **"Parking"** con identificador de mismo valor que el evento anterior, pero con plazas libres igual a cero. Si observamos la imagen 8.44 vemos que es entonces cuando se genera el evento complejo **"ParkingCompleto"**. Se ha probado enviando de nuevo un evento simple de tipo **"Parking"** con mismos valores y de nuevo con una plaza libre. Vuelve a avisarse al usuario con el evento complejo de tipo **"ParkingPlazasLibresCerca_MeGustaMenosDe2h"**.

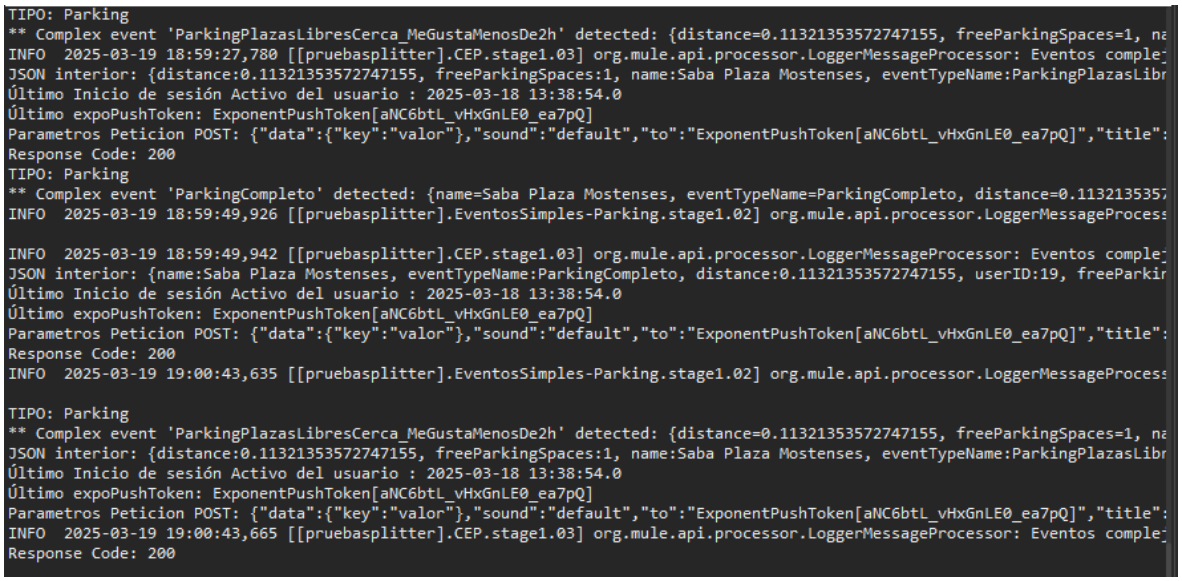


Figura 8.44: Prueba de integración patrón “ParkingCompleto” desde Mule Anypoint

Como podemos observar en la imagen 8.45, este evento es procesado por la Clase Java “SendExpoNotification” y se envía la correspondiente notificación push al dispositivo móvil en que el usuario tiene una sesión activa.

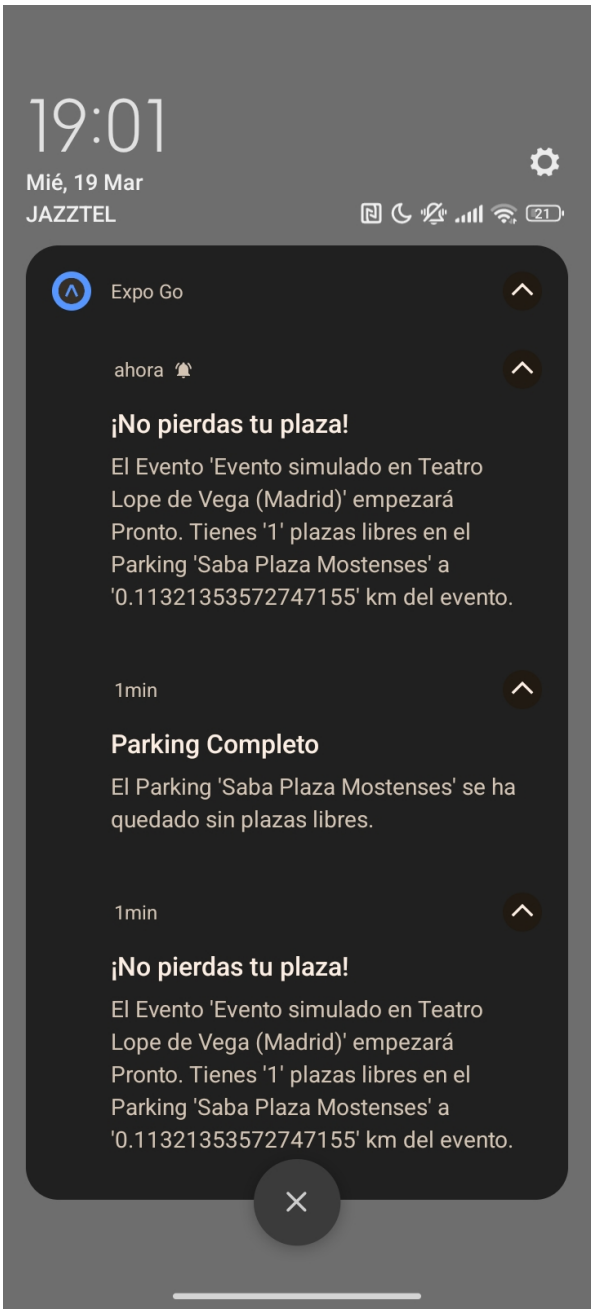


Figura 8.45: Prueba de integración de notificación push del patrón "ParkingCompleto" desde dispositivo móvil

- **MenosDe2hCerca**: Podemos observar en la imagen 8.46 el evento simple de tipo **"EventosMadrid"** simulado y enviado a su respectiva cola amqp. Para probar que realmente el sistema en su conjunto funciona y dado que no nos encontramos en Madrid a la hora de realizar esta prueba, se ha tomado la decisión de generar un evento con localización cercana a donde nos encontramos en el momento de realizar la prueba, en la localidad de San Fernando(Cádiz). Se podría haber simulado un evento de Madrid con localización en Madrid y haber simulado también un evento simple de tipo **"ubicación"** con localización también en Madrid, pero de esta manera no probaríamos realmente el sistema en su totalidad,

pues el envío por parte del back-end de la ubicación no se probaría. Observemos que se ha establecido como fecha de inicio el valor "18/03/2025 15:00:00", y el valor 1742297100000 como su respectivo formato EPOCH en milisegundos.

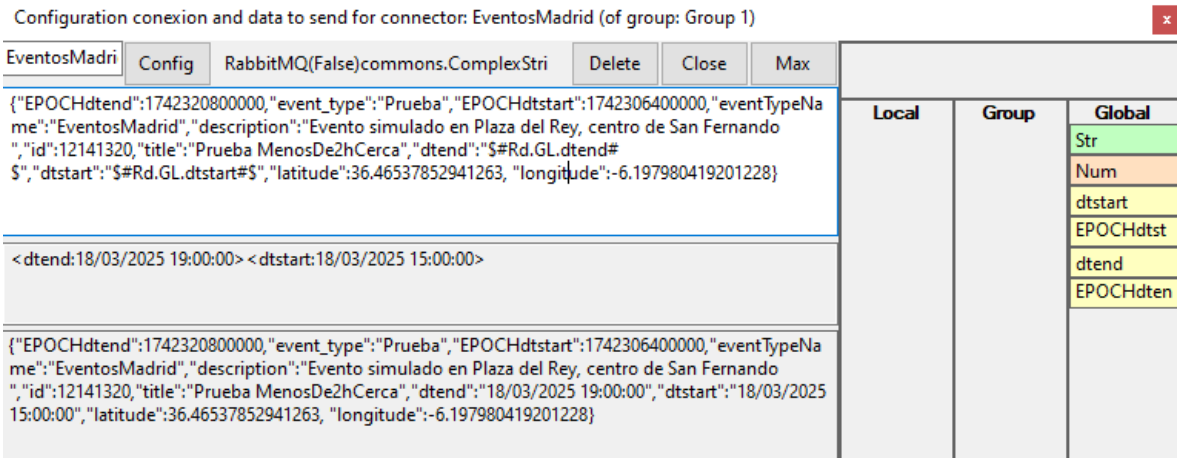


Figura 8.46: Evento simple de tipo "EventosMadrid" generado desde nITROGEN con localización en la localidad de San Fernando(Cádiz)

Observémos la imagen 8.47. Vemos que se recoge de la cola y se envía como Evento simple al motor CEP. Comprobamos que, tras el primer intervalo de 1 minuto desde que llega el evento de Madrid, se evalua la condición del patrón a verdadera y se detecta el evento complejo **"MenosDe2hCerca"**, pues quedan efectivamente menos de 2 horas desde el momento "13:37:22" en que la condición del patrón es evaluada hasta la fecha de inicio del evento ya mencionada.

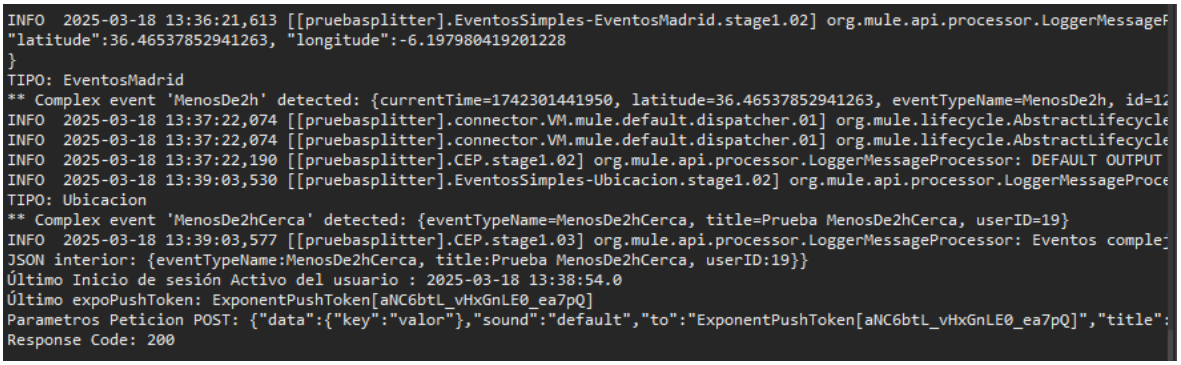


Figura 8.47: Prueba de integración patrón "MenosDe2hCerca" desde Mule Anypoint

Como podemos observar en la imagen 8.48, este evento es procesado por la Clase Java "SendExpoNotification" y se envía la correspondiente notificación push al dispositivo móvil en que el usuario tiene una sesión activa y desde el que se ha enviado la localización.

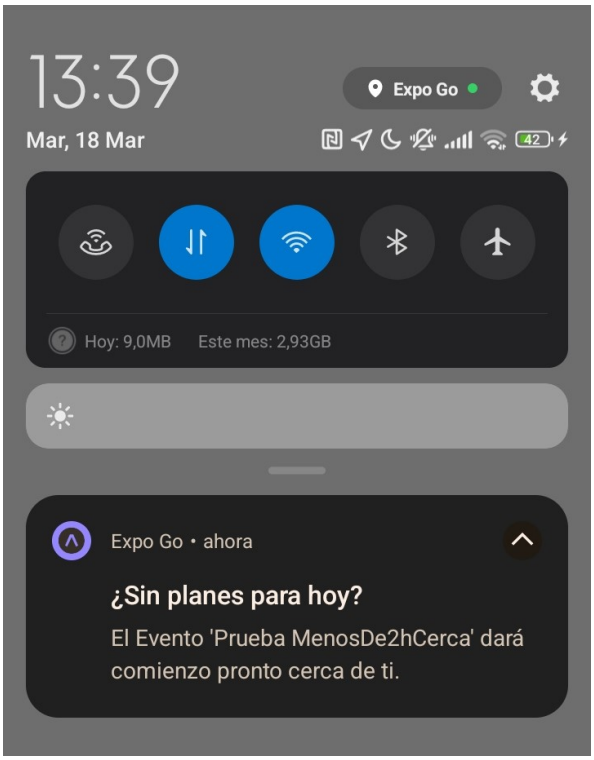


Figura 8.48: Prueba de integración de notificación push del patrón "MenosDe2hCerca" desde dispositivo móvil

- **Detecta día de la semana**: Para realizar esta prueba hemos modificado ligeramente el patrón "Jueves", para que en vez de detectarse a las 00:00 se detecte a las 20:39.

Extracto de código 8.1: Modificación del patrón "Jueves"

```
@public @Name('Jueves')
INSERT INTO DayOfWeek
select "TH" as DayOfWeek
FROM pattern [
every timer:at(39, 20, *, *, 4)
];
```

Vemos como, efectivamente, se detecta el patrón a dicha hora en la imagen 8.49.

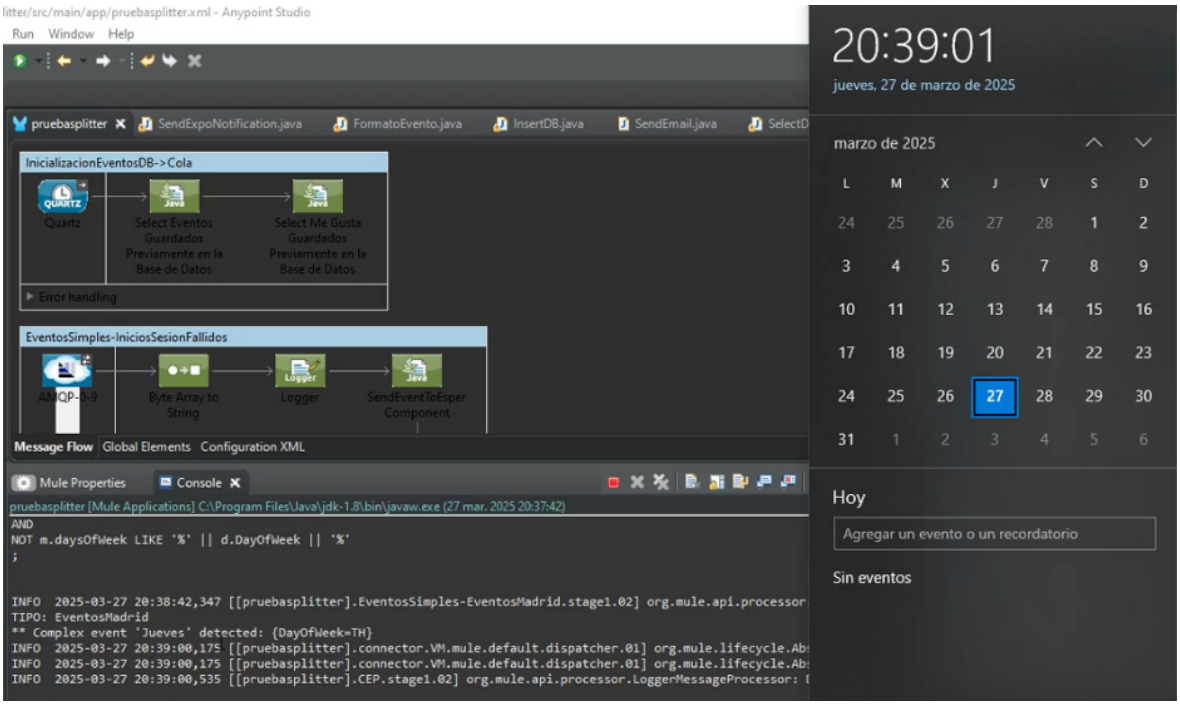


Figura 8.49: Prueba de integración del patrón “Martes” desde Mule Anypoint

- **EventoEnCurso**: Vemos como, tras un minuto, se detecta el patrón detectándo el evento que podemos observar en la imagen 8.50.



Figura 8.50: Evento en curso en la base de datos vista desde la aplicación Android

```
INFO  2025-03-27 20:38:42,347 [[pruebasplitter].EventosSimples-EventosMadrid.stage1.02] org.mule.api.processor.LoggerMessageF
TIPO: EventosMadrid
** Complex event 'Jueves' detected: {DayOfWeek=TH}
INFO  2025-03-27 20:39:00,175 [[pruebasplitter].connector.VM.mule.default.dispatcher.01] org.mule.lifecycle.AbstractLifecycle
INFO  2025-03-27 20:39:00,175 [[pruebasplitter].connector.VM.mule.default.dispatcher.01] org.mule.lifecycle.AbstractLifecycle
INFO  2025-03-27 20:39:00,535 [[pruebasplitter].CEP.stage1.02] org.mule.api.processor.LoggerMessageProcessor: DEFAULT OUTPUT
** Complex event 'EventoEnCurso' detected: {currentTime=1743104383353, eventTypeName=, id=1, title=Evento simulado ESI, daysC
INFO  2025-03-27 20:39:43,365 [[pruebasplitter].CEP.stage1.02] org.mule.api.processor.LoggerMessageProcessor: Evento En Cursc
```

Figura 8.51: Prueba de integración del patrón “EventoEnCurso” desde Mule Anypoint

- **MeGustaEventoEnCurso**: Vemos en la imagen 8.52 cómo desde la aplicación se deja un “me gusta” al evento en curso, y habiéndose detectado antes el patrón **EventoEnCurso**, se detecta este patrón.



Figura 8.52: Me gusta a evento en curso en la base de datos desde la aplicación Android

```
TIPO: MeGusta
** Complex event 'MeGustaEventoEnCurso' detected: {eventTypeName=MeGustaEventoEnCurso, eventID=1, title=Evento simulado ESI,
```

Figura 8.53: Prueba de integración del patrón “MeGustaEventoEnCurso” desde Mule Anypoint

- **HoyNoSeCelebra**: Como vemos en la imagen 8.54, el evento insertado para la detección del patrón **EventoEnCurso** no contiene en su campo “daysOfWeek” el valor “TH”.

```sql
INSERT INTO `eventos` (
  `id`, `event_type`, `title`, `description`, `is_free`, `price`,
  `dtstart`, `EPOCHdtstart`, `dtend`, `EPOCHdtend`,
  `excluded_days`, `EPOCH_excluded_days`, `daysOfWeek`, `latitude`, `longitude`
)
VALUES (
   1,
  'Prueba',
  'Evento Simulado ESI',
  'Evento simulado con aforo limitado',
  1,
  '15 euros',
  '2025-03-27 19:00:00',
  1743098400000,
  '2025-03-31 20:30:00',
  1743445800000,
  '["29-3-2025"]',
  '[1743238140000]',
  'MO,TU,WE,SA',
  36.53750309646749, -6.201304888474803
);
```

Figura 8.54: Inserción en la base de datos de evento en curso simulado

Vemos entonces en la imagen 8.55, como tras detectarse el patrón **MeGustaEventoEnCurso**, y tras haberse detectado anteriormente el patrón **Jueves**, se detecta el patrón **HoyNoSeCelebra**.

```
TIPO: MeGusta
** Complex event 'MeGustaEventoEnCurso' detected: {eventTypeName=MeGustaEventoEnCurso, eventID=1, title=Evento simulado ESI,
** Complex event 'HoyNoSeCelebra' detected: {eventTypeName=HoyNoSeCelebra, eventID=1, DayOfWeek=TH, title=Evento simulado ESI
INFO  2025-03-27 20:40:11,299 [[pruebasplitter].CEP.stage1.02] org.mule.api.processor.LoggerMessageProcessor: DEFAULT OUTPUT
INFO  2025-03-27 20:40:11,313 [[pruebasplitter].CEP.stage1.02] org.mule.api.processor.LoggerMessageProcessor: Eventos comple
JSON interior: {eventTypeName:HoyNoSeCelebra, eventID:1, DayOfWeek:TH, title:Evento simulado ESI, userID:19}}
Último Inicio de sesión Activo del usuario : 2025-03-27 20:18:26.0
Último expoPushToken: ExponentPushToken[aNC6btL_vHxGnLE0_ea7pQ]
Respuesta: true
Parametros Peticion POST: {"data":{"key":"valor"},"sound":"default","to":"ExponentPushToken[aNC6btL_vHxGnLE0_ea7pQ]","title":
Response Code: 200
```

Figura 8.55: Prueba de integración del patrón “HoyNoSeCelebra” desde Mule Anypoint

Vemos en la imagen 8.56 como efectivamente desde la aplicación Android se recibe la notificación.



Figura 8.56: Notificación del patrón "HoyNoSeCelebra" desde la aplicación Android

8.2.2. Pruebas de integración del back-end con la base de datos usando Jest Y SuperTest

Estas pruebas tienen como objetivo garantizar el correcto funcionamiento del subsistema back-end, analizando que la realización de cada una de sus operaciones tenga el comportamiento y resultado esperados en la base de datos. Para el testing se ha hecho uso de una base de datos auxiliar que no es sino una copia de la original, cuya finalidad es evitar inserciones, modificaciones o borrados no deseados sobre esta. Con ese fin se define en el archivo .env una variable de entorno llamada MYSQL_DATABASE_TEST. En el extracto de código 8.2 puede verse cómo queda el archivo :

Extracto de código 8.2: Condicional en el fichero "database.js" para el uso de una base de datos u otra según el entorno

```
import mysql from 'mysql2';
import dotenv from 'dotenv';

dotenv.config();

const pool = mysql.createPool({
    host : process.env.MYSQL_HOST,
    user : process.env.MYSQL_USER,
    password : process.env.MYSQL_PASSWORD,
```

```
        database : process.env.NODE_ENV==='test' ? process.
           env.MYSQL_DATABASE_TEST : process.env.
           MYSQL_DATABASE

    }).promise();


    export {pool}
```

Definimos los siguientes scripts en el fichero package.json con el fin de poder indicar de manera más ágil mediante comandos más simples en qué entorno ejecutaremos nuestro código (ver extracto de código 8.3). El uso de cross-env queda justificado en el subapartado 2.2.18, dentro del apartado dedicado a las Tecnologías.

Extracto de código 8.3: Scripts en el fichero "package.json"

```
    "scripts": {
        "dev": "cross-env NODE_ENV=development npx nodemon
           app.js",
        "start": "cross-env NODE_ENV=production node app.js",
        "test":"cross-env NODE_ENV=test node --experimental-
           vm-modules node_modules/jest/bin/jest.js --
           detectOpenHandles --forceExit",
        "test:watch": "npm run test -- --watch"
    }
```

Centrémonos en el script "test". La opción "node –experimental-vm-modules" proporciona compatibilidad con los "ECMAScript Modules" (Jest, 2025a). Se indica la ruta del ejecutable de Jest ‘‘node_modules/jest/bin/jest.js’’ para ejecutarlo. Las opciones "–detectOpenHandles" y "–forceExit" las usamos para evitar que alguna operación asíncrona quede abierta antes de que termine la ejecución de Jest y pueda provocar errores, cerrando entonces la ejecución de Jest.

Veremos a continuación los test que se han implementado, para los que se disponen los siguientes ficheros que veremos a continuación en el directorio "tests". Prestamos atención también a la presencia del uso de Json Web Token (JWT) en los test de los ficheros 8.2.2.3 y 8.2.2.1:

8.2.2.1. authSesions.test.js

Este fichero se ha implementado para el testing de operaciones definidas por medio de los endpoints localizados en los archivos ‘‘/server/routes/auth.js’’ y ‘‘/server/tests/users.js’’, integrando su funcionamiento a las tablas de la base de datos "users" y "sesiones" en el primer caso, y solamente "users" en el segundo.

Obsérvese en el extracto de código 8.4 el contenido del fichero mencionado, en especial los test que comprueban la respuesta producto de las peticiones a los endpoints protegidos por el uso de Json Web Token, que debe ser el código 401 en caso de no tener autorización por medio del correspondiente token.

Extracto de código 8.4: Uso de Jest y SuperTests en el fichero “authSesions.test.js”

```
import supertest from 'supertest'
import {app, server} from '../app'
import {pool} from '../database'
import {usersModel} from '../models/users'
import {sesionesModel} from '../models/sesiones'



const api = supertest(app)

let token
let userID

beforeAll(async () => {
    await usersModel.deleteAll()
    await sesionesModel.deleteAll()
})

//...

test('Se genera JWT(Java Web Token) al iniciar sesion', async
    ()=>{
    const response = await api
        .post(`/auth/login/`)
        .send(
            {
                "email": "test@gmail.com",
                "password": "A",
                "expoPushToken": "ExponentPushToken[
                   aNC6btL_vHxGnLE0_ea7pQ]"
            }
        )
        .expect(200)
        .expect('Content-Type', /application\/json/)

        token = response.body.token
        userID = response.body.id

        expect(token).toBeDefined()
})


test('Se recibe codigo 409 "Conflict" al intentar iniciar
   sesion para un usuario con una sesion activa en otro
   dispositivo (esta tiene distinto expoPushToken al enviado
   en el cuerpo de la solicitud).', async ()=>{
```

```
    await api
        .post(‘/auth/login/‘)
        .send(
            {
                "email": "test@gmail.com",
                "password": "A",
                "expoPushToken": ""
            }
        )
        .expect(409)
        .expect('Content-Type', /application\/json/)
})



test('Al hacer peticion para obtener los datos de usuario por
    medio de su e-mail, se recibe codigo 401 "Unauthorized"
   al no proporcionar JWT(Java Web Token).', async ()=>{
    await api
        .get(‘/users/test@gmail.com/‘)
        .expect(401)
        .expect('Content-Type', /application\/json/)
})


test('Se realiza correctamente la peticion para obtener los
   datos de usuario por medio de su e-mail al proporcionar
   JWT(Java Web Token)', async ()=>{
    await api
        .get(‘/users/albertoaratorramillo@gmail.com/‘)
        .set("Authorization", ‘Bearer ${token}‘)
        .expect(200)
        .expect('Content-Type', /application\/json/)
})


test('Se cierra la sesion de usuario correctamente', async ()
   =>{
    await api
        .post(‘/auth/logout/‘)
        .send(
            {
                "userID":userID
            }
        )
        .expect(200)
        .expect('Content-Type', /application\/json/)
```

```
})


afterAll(async()=>{
    server.close()
    await pool.end()
})
```

8.2.2.2. usersInteresesEventos.test.js

Este fichero se ha implementado para el testing de operaciones definidas por medio de los endpoints localizados en los archivos ‘‘/server/routes/users.js’’, ‘‘/server/routes/intereses.js’’ y ‘‘/server/tests/eventos.js’’, integrando su funcionamiento a las tablas de la base de datos “users”, “intereses” y “eventos”.

8.2.2.3. chats.test.js

Este fichero se ha implementado para el testing de operaciones definidas por medio de los endpoints localizados en el archivo ‘‘/server/routes/chats.js’’, integrando su funcionamiento a las tablas de la base de datos “chats” y “chat_users”.

Obsérvese en el extracto de código 8.5 el contenido del fichero mencionado, en especial los test que comprueban la respuesta producto de las peticiones a los endpoints protegidos por el uso de Json Web Token, que debe ser el código 401 en caso de no tener autorización por medio del correspondiente token.

Extracto de código 8.5: Uso de Jest y SuperTests en el fichero “chats.test.js”

```
    import supertest from 'supertest'
    import {app, server} from '../app'
    import {pool} from '../database'
    import {chatsModel} from '../models/chats'
    import {usersModel} from '../models/users'
    import jwt from 'jsonwebtoken';


    import path from 'path';
    import { fileURLToPath } from 'url';

    const __filename = fileURLToPath(import.meta.url);
    const __dirname = path.dirname(__filename);


    const api = supertest(app)
```

```
let chatID
let user1ID
let user2ID
let groupChatID
let token

beforeAll(async () => {
    await chatsModel.deleteAll()
    await usersModel.deleteAll()
})


test('Se realiza correctamente la insercion de chat en la
    base de datos. Se obtiene codigo 201 "Created"',
   async ()=>{
    const response = await api
        .post(`/chats/insertNewChat/`)
        .send(
            {
                "isGroup": 0
            }
        )
        .expect(201)
        .expect('Content-Type', /application\/json/)

        chatID = response.body.insertId

})


test('Se realiza correctamente la insercion de un primer
   usuario en la base de datos. Se obtiene codigo 201 "
   Created', async ()=>{

    const response = await api
        .post(`/users/create/`)
        .send(
            {
                "nombre":"user1",
                "apellidos":"test",
                "nickname":"user1",
                "email":"user1@gmail.com",
                "password":"A"
            }
        )
        .expect(201)
        .expect('Content-Type', /application\/json/)
```

```
        user1ID = response.body.insertId

        const userForToken = {
        id : user1ID ,
        nickname : "user1" ,
        email : "user1@gmail.com"
        }
        token = jwt.sign ( userForToken , process . env .
        JWT_SECRET )
})

test('Se realiza correctamente la insercion de un segundo
    usuario en la base de datos. Se obtiene codigo 201 "
   Created', async ()=>{
    const response = await api
        .post('/users/create/')
        .send(
            {
                "nombre":"user2",
                "apellidos":"test",
                "nickname":"user2",
                "email":"user2@gmail.com",
                "password":"A"
            }
        )
        .expect(201)
        .expect('Content-Type', /application\/json/)

        user2ID = response.body.insertId
})

test('Se realiza correctamente la insercion del primer
   usuario a un chat individual en la base de datos. Se
   obtiene codigo 201 "Created"', async ()=>{
    await api
        .post('/chats/insertNewChatUser/')
        .send(
            {
                "chatID" : chatID,
                "userID" : user1ID
            }
        )
        .expect(201)
        .expect('Content-Type', /application\/json/)
})
```

```
test('Se realiza correctamente la insercion del segundo
   usuario a un chat individual en la base de datos. Se
   obtiene codigo 201 "Created"', async ()=>{
    await api
        .post(`/chats/insertNewChatUser/`)
        .send(
            {
                "chatID" : chatID,
                "userID" : user2ID
            }
        )
        .expect(201)
        .expect('Content-Type', /application\/json/)
})

test('Se obtiene informacion sobre la existencia del
   primer usuario en el chat creado', async ()=>{
    await api
        .get(`/chats/checkIfExistsChatUser/${chatID}/${
          user1ID}`)
        .expect(200)
        .expect('Content-Type', /application\/json/)
})



test('Se obtiene informacion sobre la existencia de un
   chat individual entre dos usuarios concretos', async
   ()=>{
    await api
        .get(`/chats/checkIfExistsIndividualChat/${
          user1ID}/${user2ID}`)
        .set('Authorization', `Bearer ${token}`)
        .expect(200)
        .expect('Content-Type', /application\/json/)
})

test('Al realizar la peticion para obtener informacion
   sobre la existencia de un chat individual entre dos
   usuarios concretos, se recibe codigo 401 "Unauthorized
   " al no proporcionar JWT(Java Web Token).', async ()
   =>{
    await api
        .get(`/chats/checkIfExistsIndividualChat/${
          user1ID}/${user2ID}`)
        .expect(401)
        .expect('Content-Type', /application\/json/)
```

```
})


//...


test('Se obtiene informacion sobre los chats individuales
    a los que pertenece un usuario concreto', async ()=>{
    await api
        .get(`/chats/getOnlyIndividualChatsFromUser/${
           user1ID}`)
        .set('Authorization', `Bearer ${token}`)
        .expect(200)
        .expect('Content-Type', /application\/json/)
})


//...

test('Se obtiene informacion sobre los chats grupales o
   individuales a los que pertenece un usuario concreto',
    async ()=>{
await api
    .get(`/chats/getChatsFromUser/${user1ID}`)
    .set('Authorization', `Bearer ${token}`)
    .expect(200)
    .expect('Content-Type', /application\/json/)
})


test('Al realizar la peticion para obtener informacion
   sobre los chats grupales o individuales a los que
   pertenece un usuario concreto, se recibe codigo 401 "
   Unauthorized" al no proporcionar JWT(Java Web Token)
   .', async ()=>{
    await api
        .get(`/chats/getChatsFromUser/${user1ID}`)
        .expect(401)
        .expect('Content-Type', /application\/json/)
})


test('Se obtiene informacion sobre los chats individuales
    a los que pertenece un usuario concreto', async ()=>{
    await api
        .get(`/chats/getOnlyIndividualChatsFromUser/${
           user1ID}`)
```

```
            .set('Authorization', `Bearer ${token}`)
            .expect(200)
            .expect('Content-Type', /application\/json/)
    })


    test('Al realizar la peticion para obtener informacion
       sobre los chats individuales a los que pertenece un
       usuario concreto, se recibe codigo 401 "Unauthorized"
       al no proporcionar JWT(Java Web Token).', async ()=>{
        await api
            .get(`/chats/getOnlyIndividualChatsFromUser/${
               user1ID}`)
            .expect(401)
            .expect('Content-Type', /application\/json/)
    })


    afterAll(async()=>{
        server.close()
        await pool.end()
    })
```

8.2.2.4. parking.test.js

Este fichero se ha implementado para el testing de operaciones definidas por medio de los endpoints localizados en el fichero ``/server/routes/parking.js'' integrando su funcionamiento a la tabla de la base de datos "parking".

8.3. Pruebas de aceptación

En estas pruebas, la calidad y usabilidad del producto final es objeto de estudio mediante el uso del usuario. Uno de los cuestionarios de usabilidad estandarizados más utilizados para evaluar la percepción de usabilidad es la Escala de Usabilidad del Sistema(SUS). El cuestionario cuenta con diez preguntas para evaluar la **usabilidad** de la aplicación que se disponen de manera que el usuario lea una pregunta con tono positivo (o sobre una cualidad positiva del sistema) y la siguiente a esta tenga un tono negativo, así sucesivamente. El resto de preguntas tras la décima, estarán enfocadas en evaluar la **utilidad** de la aplicación.

1. Creo que me gustaría usar este sistema con frecuencia. (Elija del 1 al 5, siendo 1 totalmente en desacuerdo y 5 totalmente de acuerdo)
2. El sistema me pareció innecesariamente complejo. (Elija del 1 al 5, siendo 1 totalmente en desacuerdo y 5 totalmente de acuerdo)

3. Me pareció fácil de usar. (Elija del 1 al 5, siendo 1 totalmente en desacuerdo y 5 totalmente de acuerdo)

4. Creo que necesitaría la ayuda de un técnico para poder usarlo. (Elija del 1 al 5, siendo 1 totalmente en desacuerdo y 5 totalmente de acuerdo)

5. Las diversas funciones de este sistema me parecieron bien integradas. (Elija del 1 al 5, siendo 1 totalmente en desacuerdo y 5 totalmente de acuerdo)

6. Pensé que había demasiada inconsistencia en este sistema. (Elija del 1 al 5, siendo 1 totalmente en desacuerdo y 5 totalmente de acuerdo)

7. Imagino que la mayoría de la gente aprendería a usar este sistema muy rápidamente. (Elija del 1 al 5, siendo 1 totalmente en desacuerdo y 5 totalmente de acuerdo)

8. El sistema me pareció muy engorroso de usar. (Elija del 1 al 5, siendo 1 totalmente en desacuerdo y 5 totalmente de acuerdo)

9. Me sentí muy seguro al usarlo. (Elija del 1 al 5, siendo 1 totalmente en desacuerdo y 5 totalmente de acuerdo)

10. Necesité aprender muchas cosas antes de poder empezar a usar este sistema. (Elija del 1 al 5, siendo 1 totalmente en desacuerdo y 5 totalmente de acuerdo)

11. Las características más interesantes de Madrid en Vivo son:

    - El explorador de eventos.
    - El sistema de chat.
    - El sistema de notificaciones.
    - El mapa con información sobre los eventos y parking.

12. Algunas cuestiones que considerarías para posibles mejoras:

    - Envío de imágenes y vídeos al chat.
    - Nuevas situaciones de interés sobre los eventos a notificar.
    - Obtención de datos sobre transporte público.
    - Detección y notificación de cuentas con actividad sospechosa o mal uso de la aplicación.
    - Otra...

13. Tener información actualizada de los eventos que están por celebrarse en Madrid podría ayudarme a mejorar y enriquecer la calidad de mi tiempo de ocio durante mi estancia en la ciudad. (Elija del 1 al 5, siendo 1 totalmente en desacuerdo y 5 totalmente de acuerdo)

14. Recibir notificaciones sobre plazas de parking libres cerca de un evento con aforo limitado y que empieza pronto es útil para el usuario. (Elija del 1 al 5, siendo 1 totalmente en desacuerdo y 5 totalmente de acuerdo)

15. Tener información actualizada de plazas libres de parking en Madrid es útil para agilizar mi estacionamiento cerca del evento o buscar otro modo de transporte a tiempo si es necesario. (Elija del 1 al 5, siendo 1 totalmente en desacuerdo y 5 totalmente de acuerdo)

16. Tener información actualizada de los eventos que están por celebrarse en Madrid puede ayudarme a desarrollar interés por nuevos tipos de actividades. (Elija del 1 al 5, siendo 1 totalmente en desacuerdo y 5 totalmente de acuerdo)

17. ¿Piensas que el uso de la aplicación puede ayudar a crear nuevas relaciones sociales a gente en la ciudad de Madrid? (Elija del 1 al 5, siendo 1 totalmente en desacuerdo y 5 totalmente de acuerdo)

18. ¿Qué impacto positivo piensas que puede llegar a tener el uso de la aplicación para alguien que acaba de mudarse a la ciudad de Madrid? (Elija del 1 al 5, siendo 1 totalmente en desacuerdo y 5 totalmente de acuerdo)

Se ha estimado el Nivel de confianza **Z** siguiendo la fórmula:

$$n = \frac{Z^2 \cdot p \cdot (1-p)}{d^2} \Rightarrow Z = \sqrt{\frac{n \cdot d^2}{p \cdot (1-p)}}$$

- n: Tamaño mínimo de la muestra ⇒ En nuestro caso han participado 10 personas
- Z: Nivel de confianza
- p: Probabilidad de éxito o prevalencia esperada ⇒ 0.5 (Valor usado cuando no hay estudios previos ya que cubre mayor rango de posibilidades)
- d: Margen de error o precisión ⇒ 5 por ciento = 0.05

$$Z = \sqrt{\frac{10 \cdot 0{,}05^2}{0{,}5 \cdot (1-0{,}5)}} \approx 0{,}32$$

Para $Z = 0{,}32$, buscamos en la tabla de distribución normal el valor correspondiente de $P(Z \leq 0{,}32) = 0{,}6255$. (ver imagen 8.57). En dicha imagen vemos la tabla de distribución normal (proporcionada por la Universidad de Murcia a través del siguiente enlace)

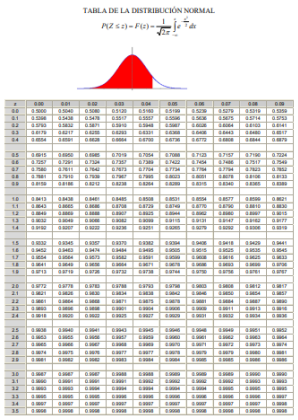


Figura 8.57: Tabla de distribución normal proporcionada por la Universidad de Murcia

Calculamos el nivel de confianza:

$$\text{Nivel de confianza} = 2 \cdot P(Z \leq 0{,}32) - 1 \Rightarrow \text{Nivel de confianza} = 2 \cdot 0{,}6255 - 1 = 0{,}251$$

Tenemos entonces un nivel de confianza de 25.1 %, un nivel bastante bajo, por lo que los resultados de los datos recogidos no representarán con mucha certeza a la mayoría de la población. Aún así, dado que la naturaleza del proyecto es la de un trabajo de fin de grado, justificamos el valor del nivel de confianza y la cantidad de participantes que han colaborado.

8.3.1. Usabilidad

Calculamos el porcentaje SUS. Las preguntas impares generan una respuesta positiva, pero las pares generan una respuesta negativa, por lo que tenemos que invertir sus puntuaciones.

1. Puntuamos los resultados del usuario para las 10 preguntas de usabilidad
   - Preguntas Impares:

     $$\text{Calificación del usuario} - 1$$

   - Preguntas Pares:

     $$5 - \text{Calificación del usuario}$$

2. Obtenemos la suma de puntos del usuario
3. Multiplicamos la suma de puntos por 2,5 para obtener su puntuación individual sobre 100: La suma de todos los puntos podría dar un máximo de cuarenta, de ahí la multiplicación por 2,5 para obtener la escala de 100
4. Repetimos los pasos anteriores para todos los usuarios y calculamos la media para obtener un porcentaje SUS

- Usuario 1:

$$2{,}5 \times (3+4+4+4+4+4+4+4+4+4) = 2{,}5 \times 39 = 97{,}5$$

- Usuario 2:

$$2{,}5 \times (4+4+4+4+4+4+4+4+4+4) = 2{,}5 \times 40 = 100{,}0$$

- Usuario 3:

$$2{,}5 \times (3+1+4+1+4+5+2+5+4+1) = 2{,}5 \times 30 = 75{,}0$$

- Usuario 4:

$$2{,}5 \times (3+2+4+1+4+1+3+1+4+2) = 2{,}5 \times 25 = 62{,}5$$

- Usuario 5:

$$2{,}5 \times (3+2+3+1+2+2+3+2+2+2) = 2{,}5 \times 22 = 55{,}0$$

- Usuario 6:

$$2{,}5 \times (3+1+4+1+3+2+3+1+3+1) = 2{,}5 \times 22 = 55{,}0$$

- Usuario 7:

$$2{,}5 \times (4+1+4+2+3+1+4+5+0+1) = 2{,}5 \times 25 = 62{,}5$$

- Usuario 8:

$$2{,}5 \times (4+1+4+1+4+1+4+1+4+1) = 2{,}5 \times 25 = 62{,}5$$

- Usuario 9:

$$2{,}5 \times (3+1+3+2+4+1+3+1+3+1) = 2{,}5 \times 22 = 55{,}0$$

- Usuario 10:

$$2{,}5 \times (4+1+2+1+4+1+4+1+4+1) = 2{,}5 \times 23 = 57{,}5$$

Ahora calculamos la media de los resultados para obtener la puntuación SUS:

$$\text{SUS} = (97{,}5+100{,}0+75{,}0+62{,}5+55{,}0+55{,}0+62{,}5+62{,}5+55{,}0+57{,}5) \div 10 = 68{,}25$$

En la imagen 8.58 podemos observar la interpretación de la puntuación SUS obtenida del siguiente enlace, viendo que el nivel de usabilidad de nuestra aplicación según el mismo se calificaría como buena, y por poco casi llegaría a excelente.

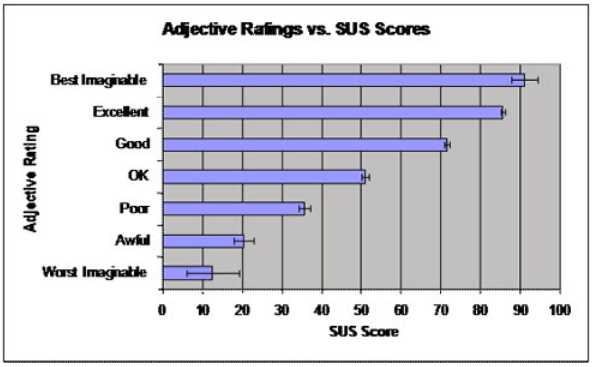


Figura 8.58: Interpretación Puntuación SUS

8.3.1.1. Resultados de la encuesta

- **Pregunta 1**: Vemos que a la mayoría de usuarios les gustaría usar el sistema con frecuencia, lo que significa que las funcionalidades que ofrece pueden resultarle útiles o fáciles de usar.

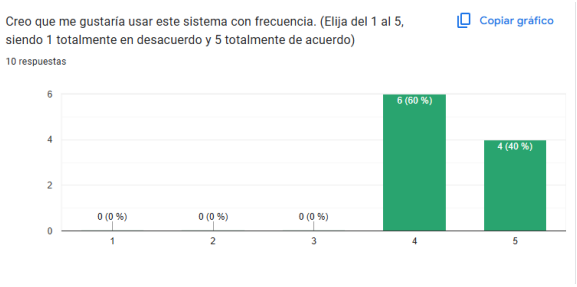


Figura 8.59: Pregunta número 1 del cuestionario - pruebas de aceptación

- **Pregunta 2**: Vemos que la mayoría de los participantes opina que el sistema no es muy complejo, lo que significa que la aplicación es fácil de usar.

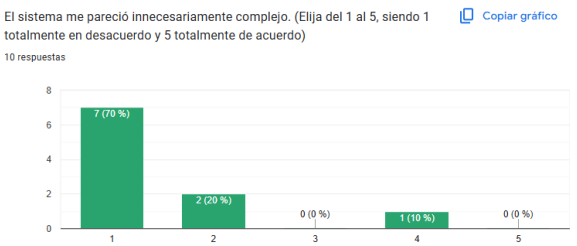


Figura 8.60: Pregunta número 2 del cuestionario - pruebas de aceptación

- **Pregunta 3**: Es la misma pregunta que la anterior pero formulada de otra manera, y obtenemos prácticamente los mismos resultados.

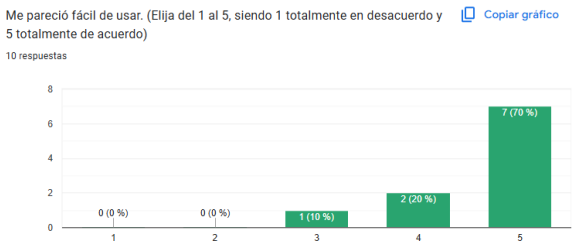


Figura 8.61: Pregunta número 3 del cuestionario - pruebas de aceptación

- **Pregunta 4**: Se pregunta al usuario si cree necesaria la ayuda de un técnico para usar el sistema. Vemos que, de nuevo, responden en base a cómo de fácil consideran que la aplicación es fácil de usar, y los resultados vuelven a ser los mismos.

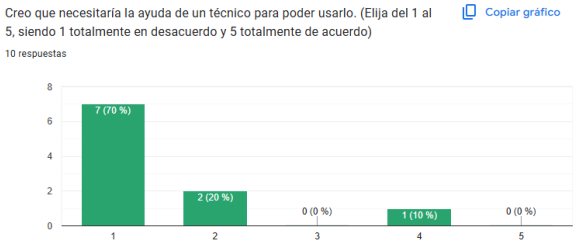


Figura 8.62: Pregunta número 4 del cuestionario - pruebas de aceptación

- **Pregunta 5**: Aquí se pregunta sobre la integración de las funciones del sistema. Esta vez la respuesta de mayor valor positivo ha sido la más contestada por los usuarios.

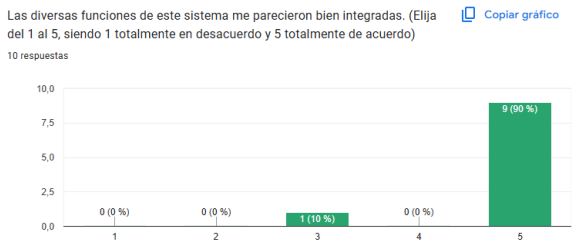


Figura 8.63: Pregunta número 5 del cuestionario - pruebas de aceptación

- **Pregunta 6**: Se pregunta sobre la inconsistencia en el sistema. Los votos siguen siendo muy similares a las contestaciones dadas en anteriores preguntas.

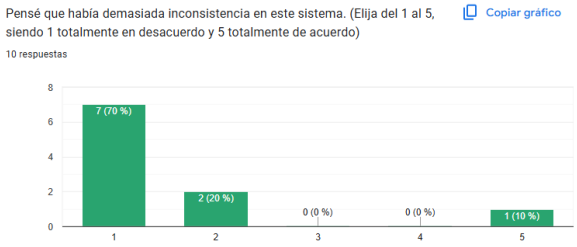


Figura 8.64: Pregunta número 6 del cuestionario - pruebas de aceptación

- **Pregunta 7**: Se pregunta al usuario si considera que el sistema es fácil de aprender a utilizar. Aunque el voto negativo de uno de los usuarios se sigue manteniendo, el resto de votos aparece para esta pregunta más dividido y equilibrado.

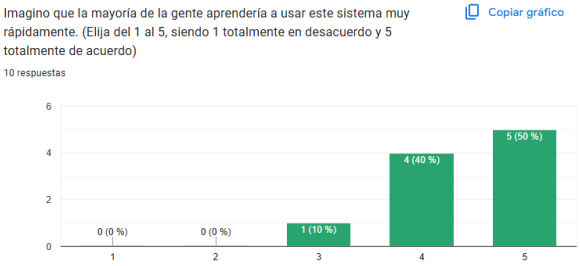


Figura 8.65: Pregunta número 7 del cuestionario - pruebas de aceptación

- **Pregunta 8**: Se vuelve a realizar una pregunta negativa sobre la usabilidad del sistema, y los votos siguen siendo prácticamente idénticos a los de la mayoría de preguntas.

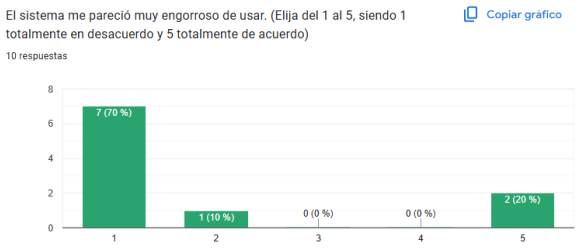


Figura 8.66: Pregunta número 8 del cuestionario - pruebas de aceptación

- **Pregunta 9**: Para esta pregunta, las respuestas varían un poco, dando otro de los usuarios una respuesta un poco más negativa.

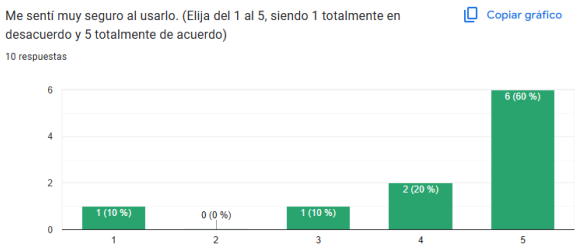


Figura 8.67: Pregunta número 9 del cuestionario - pruebas de aceptación

- **Pregunta 10**: Como vemos en la gráfica, los resultados son nuevamente idénticos a los de la mayoría de preguntas, considerando la mayoría que no necesitarían aprender muchas cosas antes de utilizar el sistema.

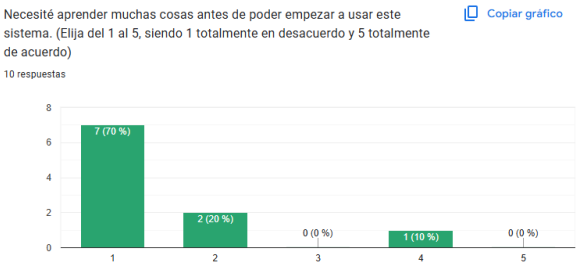


Figura 8.68: Pregunta número 10 del cuestionario - pruebas de aceptación

Observamos que las respuestas, aunque las preguntas variaran incluso de un tono negativo a uno positivo y viceversa, son prácticamente idénticas para la mayoría de preguntas, por lo que podemos considerar que el 90 % de los usuarios considera que el sistema tiene una buena usabilidad.

8.3.2. Utilidad

Observamos en este apartado las respuestas de los usuarios a las preguntas dedicadas a analizar la utilidad del sistema. Esto sirve para determinar las funcionalidades que más interesan o son más útiles a los usuarios finales.

8.3.2.1. Resultados de la encuesta

- **Pregunta 11**: La mayoría de usuarios, siendo este el 70 % de ellos, considera como la característica más importante del sistema la funcionalidad del mapa.



Figura 8.69: Pregunta número 11 del cuestionario - pruebas de aceptación

- **Pregunta 12**: La mayoría de usuarios, siendo este el 50 % de ellos, considera la obtención de datos de transporte público como mejor opción de entre las propuestas como posibles mejoras.



Figura 8.70: Pregunta número 12 del cuestionario - pruebas de aceptación

- **Pregunta 13**: Todos los usuarios participantes consideran que la información actualizada sobre los eventos de Madrid que ofrece el sistema puede ayudar a mejorar la calidad del tiempo de ocio en la ciudad.

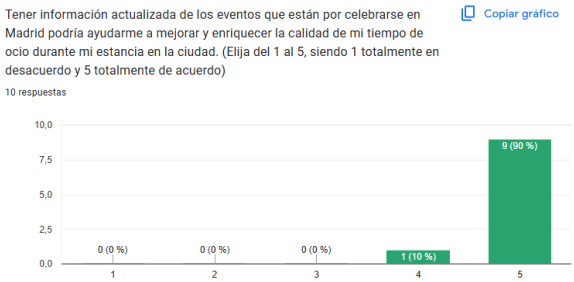


Figura 8.71: Pregunta número 13 del cuestionario - pruebas de aceptación

- **Pregunta 14**: Todos los usuarios participantes consideran de utilidad la funcionalidad de notificar al usuario sobre plazas de parking libres cerca de un evento con aforo limitado y que empieza pronto.

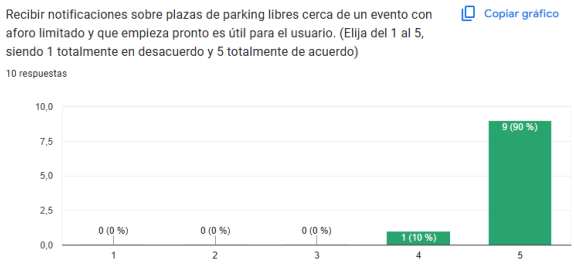


Figura 8.72: Pregunta número 14 del cuestionario - pruebas de aceptación

- **Pregunta 15**: Todos los participantes dan el voto de mayor valor positivo a la utilidad de la funcionalidad de recibir información actualizada sobre plazas de parking libres, desde el punto de vista de que permita aparcar más facilmente cerca de un evento o directamente decidir usar otro medio de transporte.

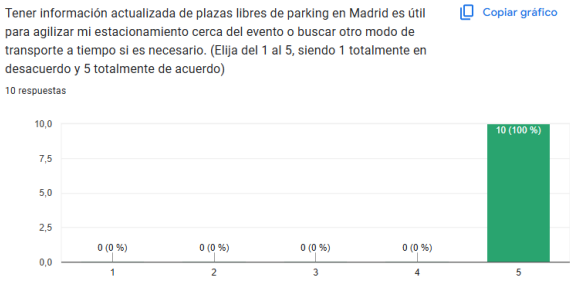


Figura 8.73: Pregunta número 15 del cuestionario - pruebas de aceptación

- **Pregunta 16**: Todos los usuarios participantes consideran de utilidad la funcionalidad de recibir información actualizada sobre los eventos que están por celebrarse en Madrid, desde el punto de vista de que pueda favorecer al desarrollo de nuevos intereses sobre nuevos tipos de eventos por parte del usuario.

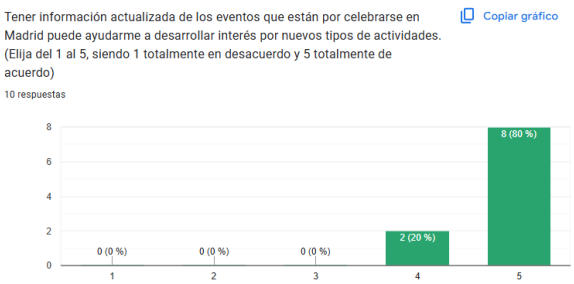


Figura 8.74: Pregunta número 16 del cuestionario - pruebas de aceptación

- **Pregunta 17**: Todos los usuarios participantes consideran que el uso del sistema puede ayudar a crear nuevas relaciones sociales a la gente en la ciudad de Madrid.

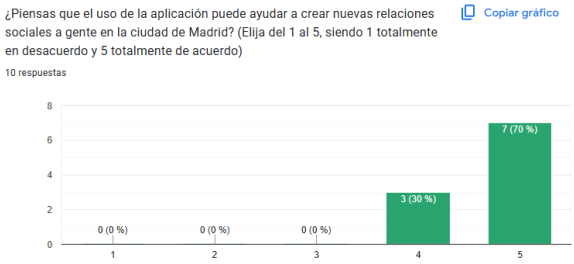


Figura 8.75: Pregunta número 17 del cuestionario - pruebas de aceptación

- **Pregunta 18**: Todos los usuarios participantes consideran que el uso del sistema puede tener un impacto positivo para los usuarios que recién acaban de mudarse a la ciudad de Madrid.

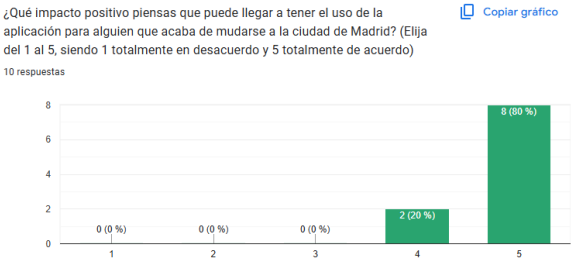


Figura 8.76: Pregunta número 18 del cuestionario - pruebas de aceptación

8.4. Pruebas de rendimiento de la aplicación Mule

En este apartado se realiza un análisis sobre el consumo de recursos de la aplicación Mule frente a una carga relevante de eventos y su procesamiento tras el despliegue de patrones, lo que también crea ventanas temporales. También se analiza bajo esas condiciones el tiempo de respuesta de la aplicación Mule, es decir, el tiempo que transcurre entre que un evento simple se consume y se genera el correspondiente evento complejo en caso de cumplir las condiciones de un patrón.

8.4.1. Análisis de consumo de CPU y memoria

8.4.1.1. Primer estudio

Para llevar a cabo este estudio se ha hecho uso de un programa java desarrollado por el grupo UCASE de Ingeniería del Software, que habitualmente lo usa para este fin. El programa en cuestión se llama Oshi.jar y permite obtener el porcentaje de uso de CPU y memoria de un proceso. Se debe ejecutar ejecutando el siguiente comando: java -cp Oshi.jar Main PID. El PID es el ID del proceso del cual se quieren obtener las estadísticas.

Debemos analizar los resultados y sacar conclusiones en base a las especificaciones del sistema en que se realizan las pruebas, donde se aloja la base de datos y desde el que se ejecutan tanto Mule como RabbitMQ y nITROGEN (ver tabla 8.1).

Tabla 8.1

Análisis de consumo de recursos

Análisis de consumo de recursos		
Tipo de Recurso	**Especificaciones del sistema**	**Pico de consumo**
CPU	Intel(R) Core(TM) i3-7020U 2.30GHz	68,75 %
RAM	8,00 GB	800,5 MiB

En las imágenes 8.77 y 8.78 se puede ver el porcentaje de consumo de CPU y la cantidad de MiB que se consumen de memoria respectivamente a lo largo de prácticamente un minuto. durante el cual se recogen diferentes eventos simples de la base de datos, siendo un total de 2096 eventos culturales de Madrid, 24 “me gusta” e información sobre 76 parking.

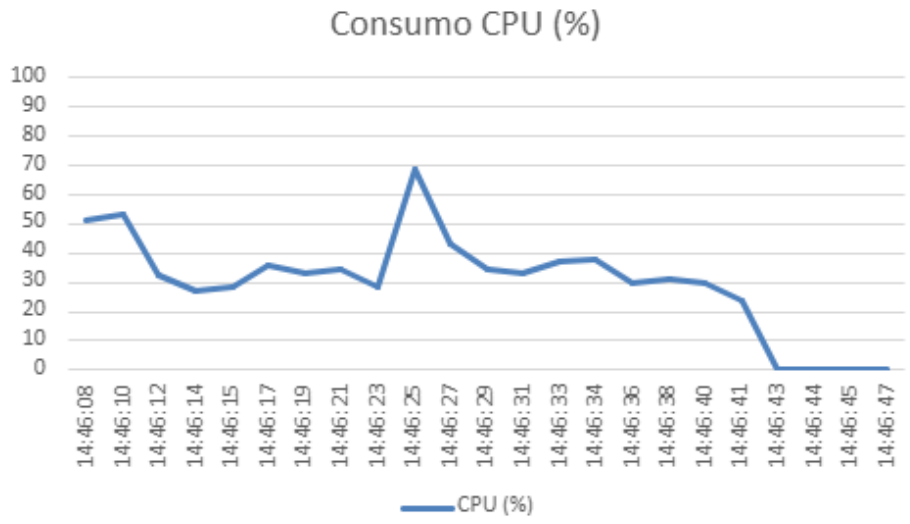


Figura 8.77: Porcentaje de consumo de CPU de la aplicación Mule

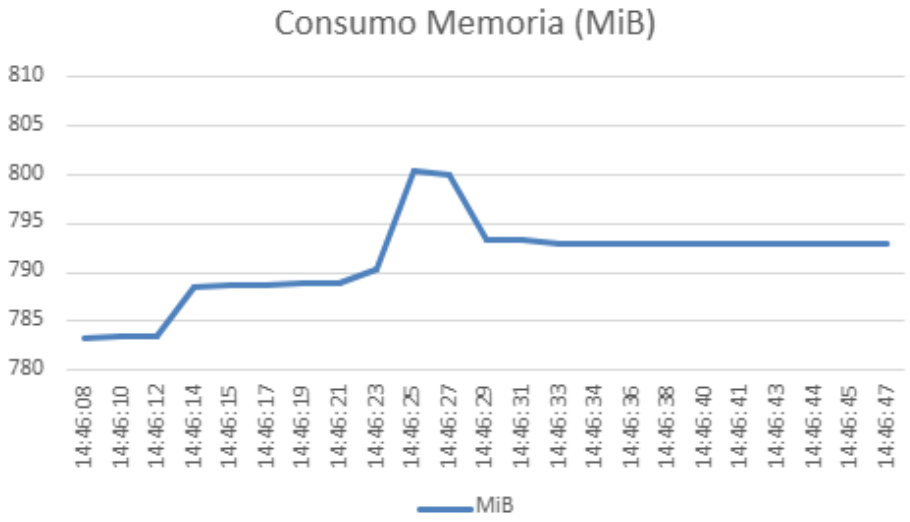


Figura 8.78: Consumo de memoria en MiB de la aplicación Mule

Podemos ver que el máximo consumo de CPU y memoria coinciden en el mismo segundo. El consumo máximo de memoria ha sido de 800,5 MiB , una fracción minúscula de aproximadamente un 10 % de la memoria RAM, de 8,00 GB (8192 MiB). Por tanto, podemos concluir que la aplicación Mule no consume mucha memoria. Por otro lado, el porcentaje de consumo máximo de CPU ha sido de un 68,75 %. Dado el procesador del dispositivo en que se realizan las pruebas esto no es desfavorable teniendo en cuenta también que este pico se alcanza con el incremento del consumo de CPU durante solo dos segundos, tras los cuales vuelve a disminuir. En este estudio la mayor parte del tiempo el porcentaje de consumo de CPU varía entre el 27,35 % y el 35,55 %.

8.4.1.2. Segundo estudio

Por otro lado hacemos una segunda prueba, generando eventos simples desde nITROGEN durante 10 minutos (600 segundos). La cadencia es aproximadamente de unos 123 eventos generados por segundo. Esta vez los eventos simples generados son de tipo “MeGusta”. Se ha escogido este tipo de evento porque la llegada y consumo de eventos simples de este tipo a este ritmo es más fiel a una situación que bien podría darse en la realidad (al usar la aplicación un gran número de usuarios) que si, por el contrario, se generaran eventos culturales con la misma frecuencia. Podemos comprobar los resultados en las graficas de las imágenes 8.79 y 8.80.

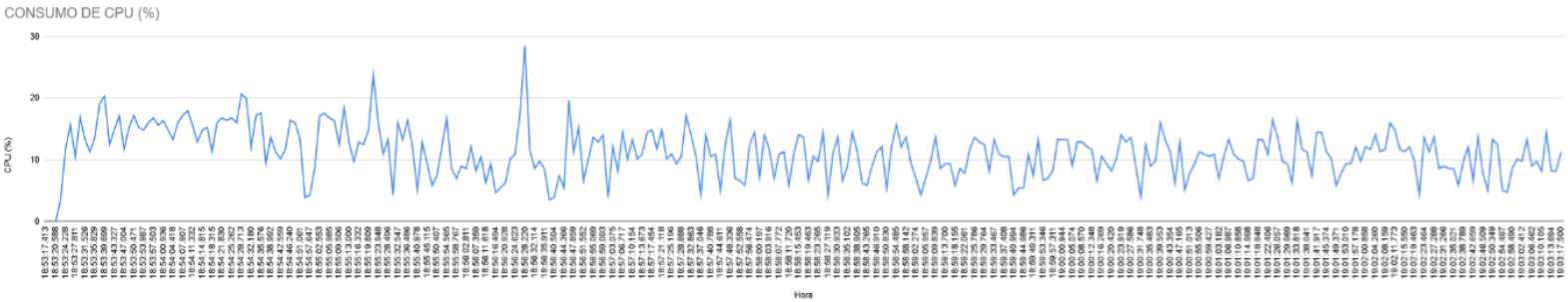


Figura 8.79: Porcentaje de consumo de CPU de la aplicación Mule durante 10 minutos

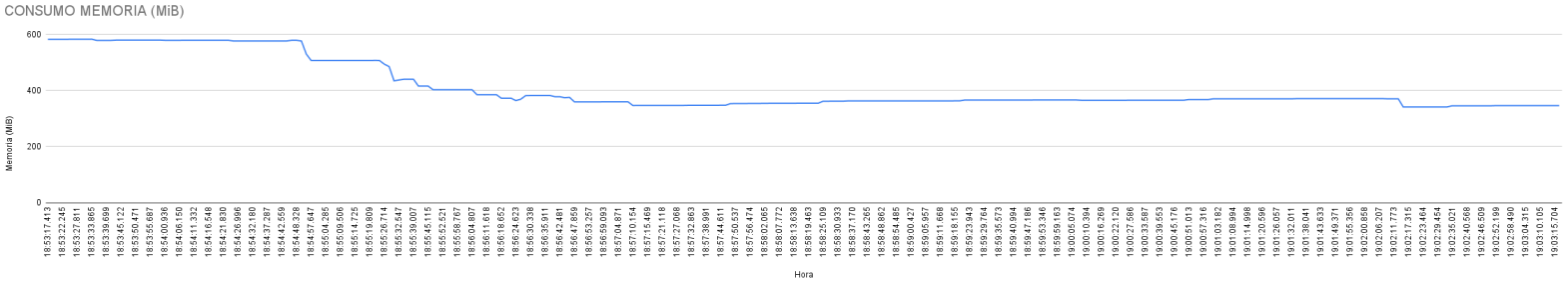


Figura 8.80: Consumo de memoria en MiB de la aplicación Mule durante 10 minutos

Podemos ver que los picos de consumo tanto de CPU como de Memoria son mayores en el primer análisis, y esto se debe a que en este se estudiaba el consumo de recursos mientras se consumían eventos recogidos de la base de datos, pero también los obtenidos por medio de una petición a la API externa. En este segundo estudio se analiza el consumo de recursos durante 10 minutos en los que únicamente se consumen eventos simples generados desde nITROGEN con la cadencia ya mencionada. A diferencia del primer estudio, analizar resultados durante este tiempo agranda también el intervalo de resultados, que oscila desde un 4 % y un 28 % para el porcentaje de consumo de CPU, y desde 341 MiB a 582 MiB de consumo de memoria. Dado que los picos de consumo de recursos son menores en este estudio, cuyo tiempo es significativamente mayor que en el primero, concluimos que los resultados son aún mas farovables que los del anterior análisis teniendo en cuenta que se realizan las pruebas bajo las mismas condiciones (a excepción de la ejecución en ese tiempo de el flujo de recogida de eventos de la base de datos y petición a la API externa) y en el mismo equipo con las mismas especificaciones.

8.4.1.3. Visualización de resultados del programa Oshi.jar

Durante la realización de estas pruebas, el programa no mostraba los resultados por consola. Se hizo uso del entorno de desarrollo IntelliJ IDEA para poder ver las clases java del archivo .jar (añadiéndolo como librería) y su contenido, para ver cómo se mostraban los resultados. Al encontrar la clase Main y leer su código fuente, se observó que los resultados se mostraban por medio del método "logger.info" de la clase "Logger". Tras investigar, se averiguó que la causa de que dicha información no se mostrara por consola podía estar relacionada con la configuración dada por el archivo log4j2.xml, el archivo de configuración de Log4j (Apache, 2025). Se analizó el contenido de dicho fichero, y se observó que los mensajes de nivel "info" no eran mostrados por consola, sino escritos en un fichero del cual se especificaba el directorio y nombre del fichero mediante la asignación de la expresión "`${sys:user.home}/stats.log`" a la propiedad "fileName".

8.4.2. Análisis de tiempo de respuesta

Se quiere estudiar el tiempo de respuesta de la aplicación Mule, es decir, el tiempo que transcurre desde que se recibe un evento simple que cumple una condición de algún patrón y el tiempo en que, consecuentemente, se genera el evento complejo. Para realizar este estudio, se ha escogido por tanto analizar el tiempo de respuesta de la generación de un tipo concreto de evento complejo cuyo patrón no genere ventanas temporales y cuya detección sea dentro de lo posible inmediata: el evento complejo "AforoLimitado".

Para realizar el estudio se ha diseñado un esquema para el tipo evento simple "dummy", que servirá para detectar el momento en milisegundos en que es recibido por un lado el evento simple, y por otro el momento en que el complejo se genera. Esto se consigue gracias al diseño de tres patrones:

- **dummyEventosMadrid**: Detecta el momento en que el evento simple de tipo "EventosMadrid" es recibido.

Extracto de código 8.6: Definición del patrón "dummyEventosMadrid"

```
@public @Name('dummyEventosMadrid')
INSERT INTO dummyEventosMadrid
select 'dummyEventosMadrid' as eventTypeName,
    e.id as eventID, d.EPOCH_now as EPOCH_now
from pattern
[
(every-distinct (e.id) e=EventosMadrid -> d=
   dummy)
];
```

- **dummyAforoLimitado**: Detecta el momento en que el evento complejo de tipo "AforoLimitado" es recibido.

Extracto de código 8.7: Definición del patrón "dummyAforoLimitado"

```
@public @Name('dummyAforoLimitado')
INSERT INTO dummyAforoLimitado
select 'dummyAforoLimitado' as eventTypeName,
    a.id as eventID, d.EPOCH_now as EPOCH_now
from pattern
[
(every a=AforoLimitado -> d=dummy)
];
```

- **TiempoRespuestaAforoLimitado**: Calcula la diferencia de tiempo en milisegundos entre los dos momentos recogidos por los dos patrones anteriores.

Extracto de código 8.8: Definición del patrón "TiempoRespuestaAforoLimitado"

```
    @public @Name('
       TiempoRespuestaAforoLimitado')
INSERT INTO TiempoRespuestaAforoLimitado
select 'TiempoRespuestaAforoLimitado' as
   eventTypeName, e.eventID as eventID,(a.
   EPOCH_now - e.EPOCH_now) as
   tiempoRespuesta_ms
from pattern
[
(every e=dummyEventosMadrid -> every a=
   dummyAforoLimitado)
OR
(every a=dummyAforoLimitado -> every e=
   dummyEventosMadrid)
]
where e.eventID=a.eventID ;
```

En la imagen 8.81 podemos ver el tiempo de respuesta en la detección del evento complejo “AforoLimitado” para diferentes eventos simples de tipo “EventosMadrid”. Estos resultados se han generado a partir de los eventos recogidos directamente de la base de datos siguiendo el flujo implementado en Mule, en cuyo caso han sido un total de 2096 eventos simples que se consumen en cuestión de unos segundos. Por tanto se puede concluir que el tiempo de respuesta se encuentra por debajo del milisegundo y por tanto, puede considerarse despreciable. Esto se debe a que la carga de datos es muy baja , ya que no se espera que se generen miles de eventos culturales por segundo.

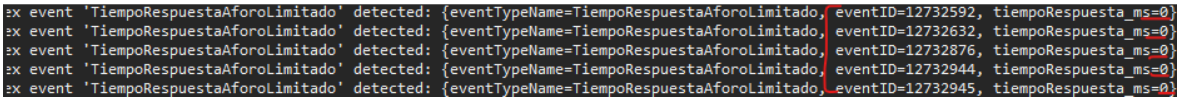


Figura 8.81: Tiempo de respuesta con eventos dummy desde Mule Anypoint

8.5. Verificación de la calidad

En este apartado se detalla cómo el sistema cumple con los diferentes Requisitos no funcionales.

- **Disponibilidad**: Tras las pruebas se ha comprobado que cada componente del sistema permanece en ejecución de forma continua sin interrupciones.
- **Fiabilidad**: Se ha probado que el sistema sea capaz de formatear de manera correcta en el correspondiente flujo de la aplicación Mule los datos de entrada que se reciben de la API externa, aunque estos presenten un formato no deseado (como por ejemplo, que presenten el caracter “\” en alguna cadena de texto). Además, la correcta funcionalidad del sistema queda verificada tras la realización de las pruebas realizadas.
- **Usabilidad**: Se han realizado Pruebas de aceptación en las que tras la participación de usuarios y del posterior estudio de la usabilidad (aparte de la utilidad) del sistema en base a las respuestas obtenidas, el resultado ha sido favorable.
- **Mantenibilidad**: La modularidad del código hace del sistema uno fácil de mantener, posibilitando que una modificación en los módulos no provoque tener que realizar cambios en todo el proyecto. Más concretamente, esto ha ayudado a agilizar el proceso de cambios que tuvo que realizarse tras el transcurso de algunas pruebas.
- **Eficiencia**: El cumplimiento de este requisito no funcional queda respaldado por el estudio realizado y detallado en el apartado Pruebas de rendimiento de la aplicación Mule.

Parte III

Epílogo

9. Conclusiones

En este último capítulo se detallan las lecciones aprendidas tras el desarrollo del presente proyecto y se identifican las posibles oportunidades de mejora sobre el software desarrollado.

9.1. Objetivos alcanzados y valoración de la funcionalidad del proyecto

Analizamos qué objetivos se han alcanzado al finalizar el proyecto:

- Se ha conseguido planear un caso de uso interesante dados los datos ofrecidos por una fuente de datos externa.
- Se ha conseguido desarrollar la aplicación móvil implementando su front-end, back-end y además, una aplicación Mule que se retroalimentan entre sí para dar el funcionamiento esperado a la aplicación.
- Se ha conseguido que la aplicación Mule sea capaz de mandar notificaciones a la aplicación móvil y por otro lado notificar por correo electrónico.
- Se ha conseguido implementar un chat asíncrono en la aplicación móvil.
- Se ha conseguido, haciendo uso de React Native en el front-end, que la aplicación tenga una interfaz vistosa y que sus componentes reaccionen a la interacción del usuario de manera adecuada.
- Se ha conseguido implementar un mapa en la aplicación móvil.

Centrándonos en la funcionalidad que el sistema ofrece, podemos decir que es interesante para el tipo de usuario por el que se formalizó la idea del proyecto, y que explicamos con más detalle en el apartado dedicado a la Motivación. Las distintas funcionalidades del sistema buscan que el usuario se mantenga actualizado de los eventos que suceden a su alrededor, y que eso mismo sea lo que le anime a descubrir nuevos tipos de ocio. A su vez, el chat de la aplicación y el sistema de "me gusta" es ideal para aquellas personas que también buscan formar nuevas relaciones sociales. Los avisos sobre plazas libres de parking y la visualización de dicha información en el mapa buscan dar solución a uno de los grandes problemas al moverse por la ciudad en vehículo: el aparcamiento.

9.2. Lecciones aprendidas y valoración personal del proyecto

Dada la decisión de desarrollar la aplicación móvil usando React Native, se tuvo que empezar a estudiar desde cero toda su semántica y sintaxis. La curva de aprendizaje fue algo costosa al principio, pero con el tiempo y práctica se fueron entendiendo conceptos clave y la aplicación empezaba a cobrar forma, plasmando las funcionalidades que se querían implementar en el código. Además se fueron refrescando conceptos referentes al uso de estilos CSS, y la aplicación también iba adoptando una interfaz más vistosa.

Por otro lado, el desarrollo del back-end costó algo menos debido a que ya se había utilizado Nodejs previamente, por lo que sólo se tuvo que recordar conceptos más básicos para empezar a desarrollarlo. No obstante, sí que fue trabajoso aprender a utilizar Socket.io para el desarrollo del chat o la subida de imágenes por parte del usuario. Un error del chat no se detectaría hasta prácticamente el final del proyecto, cuando se decidió realizar las Pruebas de aceptación. Gracias a ello, se comprendió un detalle muy importante que se pasó por alto durante la implementación del chat, y cuya ausencia en el código no permitía que los mensajes solo se emitieran a su correspondiente chat.

Se tuvo que repasar también conceptos vistos en la asignatura "Ingeniería de Sistemas de Información" para implementar patrones que cumplieran con la funcionalidad buscada. Además, se tuvieron que aprender a usar componentes no vistos en la asignatura para recoger datos de fuentes externas mediante request, como son el componente "Poll" y "Quartz". Se implementaron clases Java para formatear los datos recogidos, para enviarlos a la base de datos y también a la cola de RabbitMQ en algunos flujos. Otras clases Java que se implementaron tenían el propósito de enviar notificaciones al dispositivo móvil desde el que el usuario mantenía su sesión activa o el envío de correo electrónico, siendo cada mensaje y notificación diferente según el evento complejo detectado.

Me llevo después de la realización del proyecto todo lo aprendido, y a pesar de la dificultad en general y de la ambición que el proyecto pueda alcanzar, la satisfacción personal por ver que todo el sistema en su conjunto cumple con los requisitos. Todo esto ya hace que el trabajo haya merecido la pena.

9.3. Trabajo futuro

Cuando se trata de proponer nuevas funcionalidades para un sistema las posibilidades son tan amplias como uno sea capaz de plasmar o traducir esas ideas a código, o quizá ese abanico de posibilidades sea directamente proporcional a su creatividad e inspiración en un momento dado. Podrían plantearse muchos otros nuevos diseños de patrones, incluso con datos simulados, aunque en el caso de este proyecto se ha querido dar protagonismo a procesar datos reales proporcionados por APIs externas.

Podrían implementarse nuevos patrones de eventos que aprovechen otra información recogida desde la API para cada evento, como el campo "excluded_days" que contiene los días para los que el evento no está programado, o el campo "audience", que indica para qué público está dirigido el evento. Podría implementarse en la aplicación la funcionalidad de que el usuario indicara el tipo de público al que pertenece, por ejemplo. Podría implementarse también la obtención de información sobre transporte público, o el envío de imágenes y vídeos a través del chat. En el apartado dedicado a Pruebas de aceptación, en concreto el apartado de preguntas dedicadas a analizar la Utilidad del sistema, podemos observar la opinión de los usuarios participantes en dichas pruebas sobre algunas de las posibles mejoras planteadas en la pregunta número doce.

Bibliografía

Apache (2025). Archivo de configuración log4j2. URL https://logging.apache.org/log4j/2.x/manual/configuration.html.[Accedido por última vez : 14 de mayo de 2025].

Auth0 (2024). Json web tokens. URL https://jwt.io/.[Accedido por última vez : 18 de octubre de 2024].

Becker, A. (2024). Heidisql. URL https://www.heidisql.com/#google_vignette.[Accedido por última vez : 2 de octubre de 2024].

contributors, W. (2024). Fórmula de haversine. URL https://es.wikipedia.org/wiki/F%C3%B3rmula_del_semiverseno.[Accedido por última vez : 22 de noviembre de 2024].

Corporation, O. (2024). Java. URL https://www.java.com/en/download/help/whatis_java.html.[Accedido por última vez : 3 de octubre de 2024].

Cousens, D. (2023). bcrypt. URL https://www.npmjs.com/package/bcrypt.[Accedido por última vez : 18 de octubre de 2024].

Dodds, K. C. (2024). cross-env. URL https://www.npmjs.com/package/cross-env.[Accedido por última vez : 16 de diciembre de 2024].

EsperTech (2024). Espertech. URL https://www.espertech.com/esper/.[Accedido por última vez : 1 de octubre de 2024].

Expo (2024). Expo go. URL https://docs.expo.dev/.[Accedido por última vez : 2 de octubre de 2024].

Foundation, N. (2024). Node.js. URL https://nodejs.org/en.[Accedido por última vez : 3 de octubre de 2024].

Genymobile (2025). scrcpy. URL https://github.com/Genymobile/scrcpy.[Accedido por última vez : 01 de abril de 2025].

Girges, R. (2024). express-fileupload. URL https://www.npmjs.com/package/express-fileupload.[Accedido por última vez : 05 de noviembre de 2024].

Google (2024a). Android studio. URL https://developer.android.com/studio/intro?hl=es-419.[Accedido por última vez : 2 de octubre de 2024].

Google (2024b). Android studio - virtual device manager. URL https://developer.android.com/studio/run/managing-avds?hl=es-419.[Accedido por última vez : 2 de octubre de 2024].

Jest (2025a). Compatibilidad de jest con ecmascriptmodules. URL https://jestjs.io/es-ES/docs/ecmascript-modules.[Accedido por última vez : 21 de enero de 2025].

Jest (2025b). Jest. URL https://www.npmjs.com/package/supertest.[Accedido por última vez : 10 de marzo de 2025].

jetbrains (2025). Intellij idea. URL https://www.jetbrains.com/idea/.[Accedido por última vez : 14 de mayo de 2025].

ladjs (2025). Supertest. URL https://www.npmjs.com/package/supertest.[Accedido por última vez : 10 de marzo de 2025].

MariaDB Corporation, M. F. (2024a). Formato utilizado por mariadb para el tipo datetime. URL https://mariadb.com/kb/en/datetime/.[Accedido por última vez : 27 de noviembre de 2024].

MariaDB Corporation, M. F. (2024b). Mariadb server. URL https://mariadb.org/es/#compatibilidad-con-mysql-postgres-mongodb-y-oracle.[Accedido por última vez : 2 de octubre de 2024].

Meta Platforms, I. (2024a). Bottom tab navigator. URL https://reactnavigation.org/docs/bottom-tab-navigator/.[Accedido por última vez : 16 de octubre de 2024].

Meta Platforms, I. (2024b). Flatlist. URL https://reactnative.dev/docs/flatlist.[Accedido por última vez : 23 de octubre de 2024].

Meta Platforms, I. (2024c). React native. URL https://reactnative.dev/.[Accedido por última vez : 3 de octubre de 2024].

Meta Platforms, I. (2024d). Stack navigator. URL https://reactnavigation.org/docs/stack-navigator/.[Accedido por última vez : 16 de octubre de 2024].

Meta Platforms, I. (2024e). usecallback. URL https://react.dev/reference/react/useCallback.[Accedido por última vez : 23 de octubre de 2024].

Meta Platforms, I. (2024f). usefocuseffect. URL https://reactnavigation.org/docs/use-focus-effect/.[Accedido por última vez : 23 de octubre de 2024].

Meta Platforms, I. (2024g). useroute. URL https://reactnavigation.org/docs/use-route/.[Accedido por última vez : 23 de octubre de 2024].

Microsoft (2024). Visual studio code. URL https://code.visualstudio.com/docs.[Accedido por última vez : 3 de octubre de 2024].

MuleSoft (2024). Anypoint studio. URL https://docs.mulesoft.com/studio/latest/.[Accedido por última vez : 1 de octubre de 2024].

Ordoñez, J. O. (2024). Implementación de la fórmula de haversine en lenguaje java. URL https://github.com/Fhernd/Java-Ejercicios/blob/master/JavaEjercicios/src/main/java/ejercicio0000504/DistanciaPuntosTierra.java.[Accedido por última vez : 22 de noviembre de 2024].

Palmer, J. (2024). Formik. URL https://formik.org/.[Accedido por última vez : 17 de octubre de 2024].

Prado, A. G. D. (2024). nitrogen. URL https://ucase.uca.es/nITROGEN/.[Accedido por última vez : 2 de octubre de 2024].

Quense, J. (2024). Yup. URL https://www.npmjs.com/package/yup/v/1.0.0-alpha.3.[Accedido por última vez : 17 de octubre de 2024].

RabbitMQ (2024). Rabbitmq. URL https://www.rabbitmq.com/.[Accedido por última vez : 1 de octubre de 2024].

Socket.IO (2024). socket.io-connection-state-recovery. URL https://socket.io/docs/v4/connection-state-recovery.[Accedido por última vez : 04 de noviembre de 2024].

Team, E. (2024a). expo-image-picker. URL https://docs.expo.dev/versions/latest/sdk/imagepicker/.[Accedido por última vez : 31 de octubre de 2024].

Team, E. (2024b). expo-location. URL https://docs.expo.dev/versions/latest/sdk/location/.[Accedido por última vez : 06 de noviembre de 2024].

Team, E. (2024c). Expo notifications. URL https://docs.expo.dev/versions/latest/sdk/notifications/.[Accedido por última vez : 17 de octubre de 2024].

Team, E. (2024d). Express. URL https://expressjs.com/.[Accedido por última vez : 3 de octubre de 2024].

Parte IV

Anexos

En esta última parte quedarán recogidas los manuales necesarios para el manejo de la aplicación resultado del desarrollo. Si se ha realizado algún tipo de evaluación de la solución proporcionada, más allá de las pruebas del sistema, también deberá venir recogida en un capítulo separado dentro de esta parte. Pueden consultarse diversos tipos de evaluaciones sobre sistemas de información en hevner2004: casos de estudio, análisis estático, análisis dinámico, simulación, experimento controlado, etc.

A. Manual de desarrollador

A continuación se recogen las instrucciones necesarias para evolucionar el software. Este manual está dirigido a los desarrolladores que pretenden extender o modificar el código fuente, con el fin de incorporar nuevas funcionalidades o modificar las ya existentes. A lo largo de este capítulo se deberán hacer referencias explicitas a aquellos epígrafes de los capítulos de Diseño, Construcción, Pruebas y Despliegue del Sistema que resulten de interés.

A.1. Introducción

Resumen de los principales objetivos, ámbito, características y alcance del software desarrollado.

A.2. nITROGEN

Para ello hay que descargarse el software la siguiente página web y descomprimir el archivo: https://ucase.uca.es/nITROGEN/.

Para ejecutar el programa, basta con ejecutar el archivo "nITROGEN GUI.exe" para acceder a su interfaz gráfica.

A.3. Anypoint Studio

Necesitas instalar el JDK8 (Java Development Kit 8), por ejemplo desde https://adoptium.net/es/temurin/releases/?version=8. Para el desarrollo del proyecto se utilizó la versión 6.6 de Anypoint Studio descargada desde el siguiente enlace: https://www.mulesoft.com/lp/dl/studio/previous Actualmente la siguiente versión disponible es la versión 7.20, para la cual no existe servidor gratuito y motivo por el que se usó la versión 6.6.

Se debe descomprimir el archivo descargado en una ruta corta y sin espacios, cerca del directorio raíz.

Abrimos el archivo de configuración "AnypointStudio.ini" y añadimos, después de la sentencia del launcher y antes de los -vmargs, la siguiente sentencia con la ruta donde hemos instalado JDK8:

-vm
C:/[Ruta de instalación de Java]/bin/javaw.exe

Al abrir Anypoint Studio por primera vez, deberemos asignar una ruta como workspace, también corta, sin espacios y cerca del directorio raíz.

Permitir el acceso a Anypoint Studio cuando el cortafuegos lo solicite.

Hacer lo siguiente una sola vez: Después, haremos clic en Help → Install new Software. En la ventana emergente abrimos el desplegable "Work with" y seleccionamos Mule Runtimes for Anypoint Studio. Desplegamos entonces las opciones de Anypoint Studio Community Runtime y seleccionamos Mule ESB Server Runtime 3.9.0 CE. Tras la instalación se solicitará el reinicio de Anypoint Studio.

A.3.1. Importar aplicación Mule en Anypoint Studio

Para importar la aplicación Mule, haremos clic en File → Import → Anypoint Studio → Anypoint Studio generated Deployable Archive (.zip) y seleccionaremos nuestro proyecto.

A.4. RabbitMQ

A.4.1. Erlang

Para un correcto funcionamiento de RabbitMQ se necesita tener instalada una versión de 64 bits de Erlang para Windows, y tener en cuenta los siguientes aspectos:

- Sólo debe haber una versión de Erlang instalada a la vez
- Erlang debe instalarse usando una cuenta de administrador
- Es muy recomendable que RabbitMQ se instale también con una cuenta de administrador
- La ruta de instalación sólo debe contener caracteres ASCII
- Hay que permitir las comunicaciones de Erlang en el cortafuegos

Podemos descargar Erlang desde https://erlang.org/download/otp_versions_tree.html.

Tras la instalación de Erlang, debemos actualizar las variables de entorno del sistema:

- Crear ERLANG_HOME `C:\Program Files\Erlang OTP`
- Añadir al Path `%ERLANG_HOME\bin`

A.4.2. Servidor RabbitMQ

Descargamos la ultima versión actual con la configuración por defecto del instalador de RabbitMQ, rabbitmq-server-version.exe desde https://www.rabbitmq.com/

install-windows.html

Permitiremos el acceso al cortafuegos cuando se solicite. Tras ello, debemos actualizar las variables de entorno del sistema:

- Crear RABBITMQ_SERVER `C:\Program Files\RabbitMQSERVER\rabbitmq_server-[versión instalada]`
- Añadir al Path `%RABBITMQ_SERVER%\sbin`

Comprobamos que se puede ver la ayuda de RabbitMQ tecleando rabbitmqctl.bat en cmd.

A.4.3. Administrador de RabbitMQ

Asumimos que RabbitMQ está instalado y funcionando en localhost en el puerto estándar (5672). Escribimos `rabbitmq-plugins enable rabbitmq_management` en la terminal y descargamos el administrador desde http://localhost:15672/cli/rabbitmqadmin, almacenándolo en la ruta `C:\Program Files\RabbitMQSERVER\rabbitmq_server-[versión instalada]\sbin`. Debemos descargar e instalar python en el dispositivo desde https://www.python.org/downloads/. Al instalar, debemos hacer clic en "añadir python al PATH". Debemos asegurarnos de que se añade al PATH de sistema.

Para comprobar que todo ha ido bien y está correctamente instalado, debemos ir a la ubicación `C:\Program Files\RabbitMQSERVER\rabbitmq_server-[versión instalada]\sbin` y escribir **python.exe rabbitmqadmin -help**.

A.4.4. Panel de control gráfico de RabbitMQ

Comprobamos el estado del servicio con **rabbitmqctl status**. Podremos parar el servicio con **rabbitmqctl stop** e iniciarlo con **rabbitmq-server start**. Para acceder a la consola gráfica entraremos en el enlace http://localhost:15672/(introducir el usuario y la contraseña por defecto: "guest").

A.5. Base de datos

Descarga e instala una base de datos SQL, por ejemplo, **heidiSQL** desde https://www.heidisql.com/download.php#google_vignette (podría ser otra, como por ejemplo MariaDB).

Al ejecutar el instalador, aceptamos términos de condiciones y procedemos a instalarlo en la ruta que seleccionemos. Una vez instalado, hacemos clic en "Nueva" para crear una nueva conexión. Seleccionamos el tipo de red , como por ejemplo "MariaDB or MySQL(TCP/IP)". Escogemos un usuario y contraseña. Una vez abramos la conexión podremos crear las tablas de nuestra base de datos que usará nuestro sistema. Hacemos

clic derecho sobre la conexión y clic sobre Crear nuevo → Base de datos. Le damos el nombre de "tfg".

Crearemos otra base de datos llamada "tfg_test" donde crearemos las mismas tablas. Esta base de datos servirá como su nombre indica, para realizar las Pruebas de integración del back-end con la base de datos usando Jest Y SuperTest.

Extracto de código A.1: Sentencias de creación de tablas sql

```
CREATE TABLE IF NOT EXISTS `eventos` (
    `id` int(11) NOT NULL AUTO_INCREMENT,
    `event_type` varchar(255) DEFAULT NULL,
    `title` varchar(255) NOT NULL,
    `description` text DEFAULT NULL,
    `is_free` tinyint(1) DEFAULT 0,
    `price` varchar(500) DEFAULT NULL,
    `dtstart` timestamp NULL DEFAULT NULL,
    `EPOCHdtstart` bigint(20) DEFAULT NULL,
    `dtend` timestamp NULL DEFAULT NULL,
    `EPOCHdtend` bigint(20) DEFAULT NULL,
    `excluded_days` longtext DEFAULT NULL,
    `EPOCH_excluded_days` longtext DEFAULT NULL,
    `daysOfWeek` varchar(255) DEFAULT NULL,
    `latitude` double DEFAULT NULL,
    `longitude` double DEFAULT NULL,
    PRIMARY KEY (`id`)
  )

CREATE TABLE Users (
    id INT AUTO_INCREMENT PRIMARY KEY,
    nombre VARCHAR(50) NULL,
    apellidos VARCHAR(50) NULL,
    nickname VARCHAR(50) UNIQUE NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL,
    password VARCHAR(255) NOT NULL,
    image MEDIUMBLOB,
    descripcion VARCHAR(255)
);

CREATE TABLE Sesiones (
    id INT AUTO_INCREMENT PRIMARY KEY,
    userID INT NOT NULL,
    state TINYINT NOT NULL,
    created_at TIMESTAMP NOT NULL DEFAULT
       CURRENT_TIMESTAMP,
    closed_at TIMESTAMP NULL DEFAULT NULL ON UPDATE
       CURRENT_TIMESTAMP,
    expoPushToken VARCHAR(255),
```

```
    FOREIGN KEY (userID) REFERENCES Users(id) ON DELETE
       CASCADE
);




CREATE TABLE IF NOT EXISTS ‘intereses‘ (
‘id‘ int(11) NOT NULL AUTO_INCREMENT,
‘userID‘ int(11) DEFAULT NULL,
‘eventID‘ int(11) DEFAULT NULL,
PRIMARY KEY (‘id‘),
UNIQUE KEY ‘unique_userID_eventID‘ (‘userID‘,‘eventID‘),
KEY ‘fk_intereses_prueba‘ (‘eventID‘),
CONSTRAINT ‘fk_intereses_prueba‘ FOREIGN KEY (‘eventID‘)
   REFERENCES ‘prueba‘ (‘id‘) ON DELETE CASCADE,
CONSTRAINT ‘fk_users‘ FOREIGN KEY (‘userID‘) REFERENCES ‘
   users‘ (‘id‘)
)


CREATE TABLE IF NOT EXISTS ‘parking‘ (
‘id‘ int(11) NOT NULL AUTO_INCREMENT,
‘nombre‘ varchar(250) DEFAULT NULL,
‘direccion‘ varchar(250) DEFAULT NULL,
‘nickName‘ varchar(250) DEFAULT NULL,
‘areaCode‘ varchar(250) DEFAULT NULL,
‘momento‘ datetime DEFAULT NULL,
‘latitud‘ double DEFAULT NULL,
‘longitud‘ double DEFAULT NULL,
‘plazasLibres‘ int(11) DEFAULT NULL,
PRIMARY KEY (‘id‘)
)


CREATE TABLE IF NOT EXISTS ‘chats‘ (
‘id‘ int(11) NOT NULL AUTO_INCREMENT,
‘isGroup‘ tinyint(1) NOT NULL,
‘fecha_creacion‘ timestamp NULL DEFAULT current_timestamp
   (),
‘groupName‘ varchar(50) DEFAULT NULL,
‘groupImage‘ mediumblob DEFAULT NULL,
PRIMARY KEY (‘id‘)
)
```

```
CREATE TABLE IF NOT EXISTS `chat_users` (
`chatID` int(11) NOT NULL,
`userID` int(11) NOT NULL,
PRIMARY KEY (`chatID`,`userID`),
KEY `userID` (`userID`),
CONSTRAINT `chat_users_ibfk_2` FOREIGN KEY (`userID`)
   REFERENCES `users` (`id`),
CONSTRAINT `fk_chat_users_chatID` FOREIGN KEY (`chatID`)
   REFERENCES `chats` (`id`) ON DELETE CASCADE
)


CREATE TABLE IF NOT EXISTS `mensajes` (
`id` int(11) NOT NULL AUTO_INCREMENT,
`cuerpo` varchar(255) DEFAULT NULL,
`username` varchar(50) DEFAULT NULL,
`chatID` int(11) DEFAULT NULL,
PRIMARY KEY (`id`),
KEY `fk_mensajes_chatID` (`chatID`),
CONSTRAINT `fk_mensajes_chatID` FOREIGN KEY (`chatID`)
   REFERENCES `chats` (`id`) ON DELETE CASCADE
)
```

A.6. Expo Go

Descargamos e instalamos en el dispositivo Android físico o emulado la aplicación Expo Go desde https://play.google.com/store/apps/details?id=host.exp.exponent&hl=es_419. Nos situaremos en el directorio `MadridTFG\server` y arrancamos el servidor ejecutando npm run dev. Por otro lado, ejecutamos dentro del directorio `MadridTFG\client` y ejecutamos npx expo start –no-dev -c –port 8082. Lo ejecutamos en ese puerto porque el 8081 es usado por la aplicación Mule. Ahora solo debemos desde Expo Go escanear el código QR que se genera, o bien introducir manualmente la url.

A.7. Fichero .env del directorio `MadridTFG\client`

Es importante que modifiquemos la variable de entorno "IP_SERVER" del fichero .env, pues para realizar una request desde el front-end se usará dicha ip para enviar la solicitud a cualquier endpoint del backend. Esto ocurre ya que si usamos "localhost" el dispositivo móvil no detectará la ip del equipo donde corre el back-end, sino la del propio dispositivo Android.

B. Manual de usuario

B.1. Inicio de sesión y registro

Al ejecutar la aplicación, lo primero que veremos será la pantalla de inicio de sesión (ver imagen B.1). Si aún no nos hemos registrado, podemos hacerlo pulsando sobre la palabra “Regístrate”.

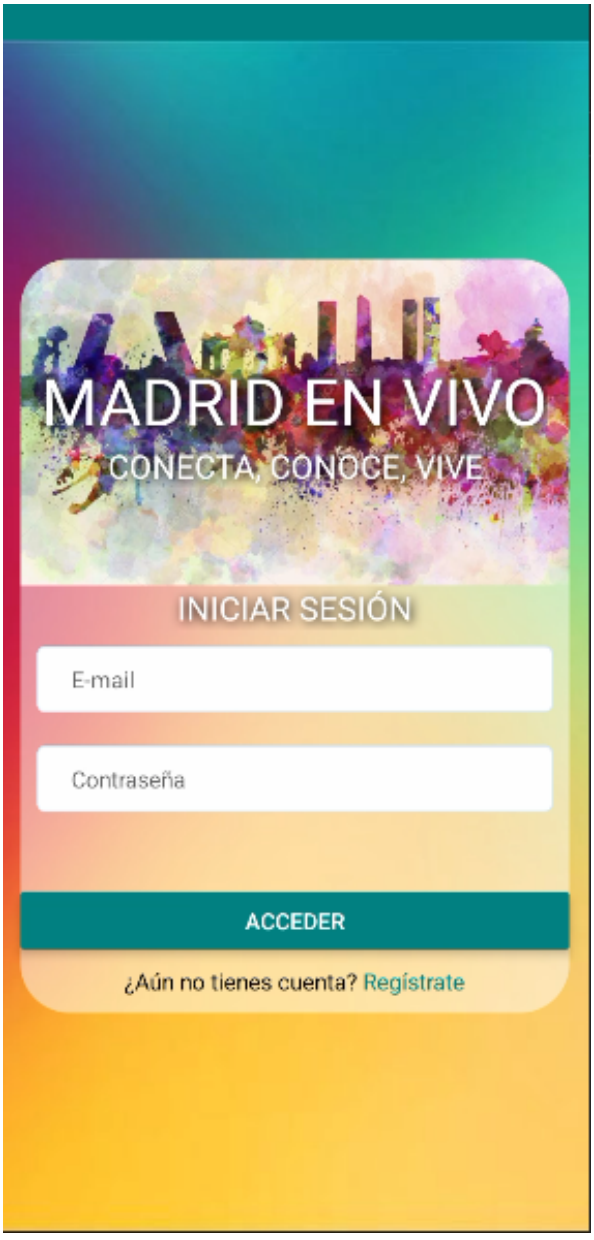


Figura B.1: Pantalla de inicio de sesión

Al pulsar dicho botón, navegaremos hacia la pantalla de registro, como podemos

apreciar en la imagen B.2.



Figura B.2: Pantalla de registro

B.2. Explorador de eventos

Una vez registrados, podremos iniciar sesión desde la pantalla de inicio de sesión. Tras iniciar sesión, se navegaremos hasta la pantalla del explorador de eventos (ver imagen B.3). En dicha pantalla podremos elegir de entre los diferentes tipos de eventos que podremos visualizar haciendo scroll, para obtener información sobre los eventos disponibles del tipo que seleccionemos.

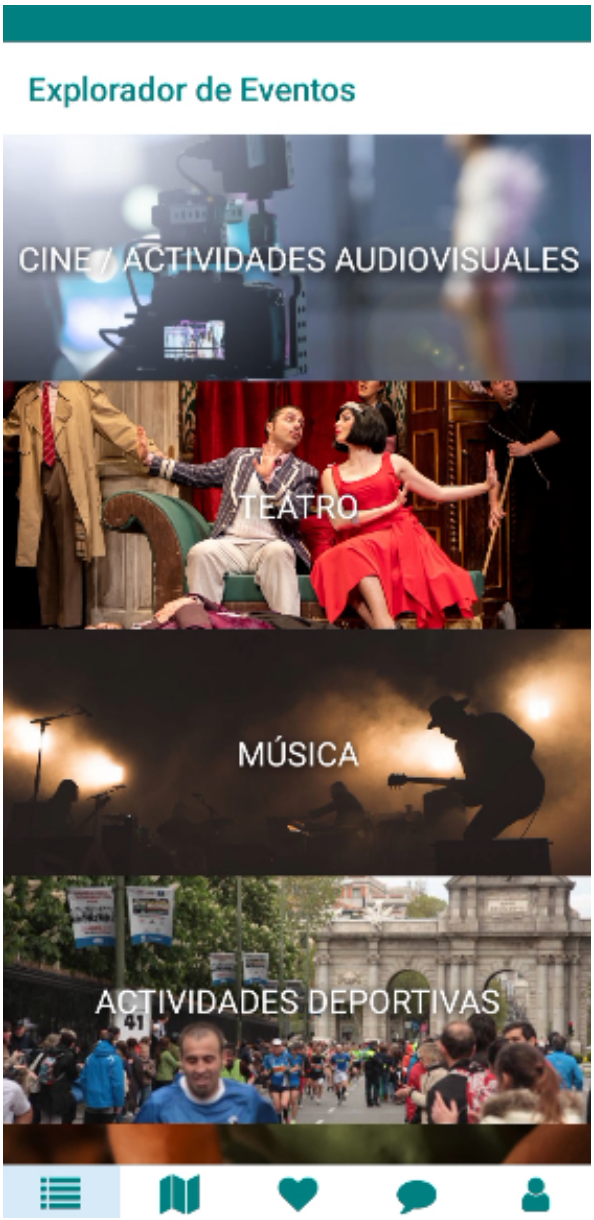


Figura B.3: Pantalla del explorador de eventos

Si seleccionamos un tipo evento, como por ejemplo “Fiestas”, podremos ver la lista de eventos que están por celebrarse de este tipo, dispuestos de manera en que los eventos que acaben primero (su fecha de finalización es más próxima) aparezcan con prioridad en la lista (ver imagen B.4). Cada evento contará con sus propios despegables de información, y podremos dejar un “me gusta” pulsando el botón correspondiente. Tras ello el botón pasará a tener el propósito de desmarcar el “me gusta”, y lo indicará con el texto “Ya No Me Gusta”.



Figura B.4: Pantalla de lista de eventos de la categoría seleccionada

B.3. Mapa

Podremos ver de igual manera los eventos desde la pantalla del mapa, navegado a ella pulsando su correspondiente botón en el menú que encontramos en el extremo inferior de pantalla. Vemos en la imagen B.5 lo que visualizamos al navegar a esta pantalla. Visualizamos en el extremo inferior derecho de la pantalla una leyenda que nos indica el significado de cada uno de los iconos que representan parking de Madrid.



Figura B.5: Pantalla del mapa

Por otro lado, en el extremo superior de la pantalla, vemos un icono con la lettra "i", que al ser pulsado nos brindará información sobre cómo utilizar esta pantalla de la aplicación (ver imagen B.6).



Figura B.6: Pantalla del mapa - información de uso

Si pulsamos sobre un Parking visualizaremos su nombre y datos sobre plazas libres disponibles como podemos observar en la imagen B.7.

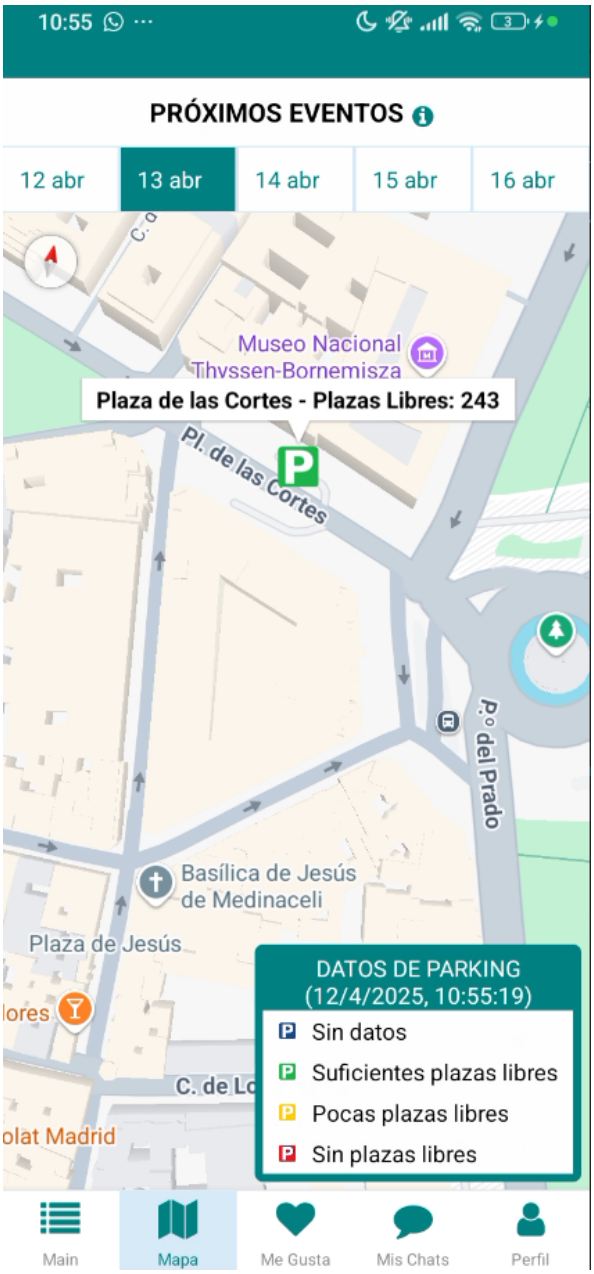


Figura B.7: Pantalla del mapa - información de plazas libres

Podemos seleccionar alguno de los próximos cuatro días para ver qué eventos están programados para entonces. Al pulsar sobre un evento, nos aparecerá un recuadro con desplegables del mismo modo que aparecían para los eventos de una categoría seleccionada en el Explorador de eventos, pudiendo dejar o no un "me gusta" (ver imagen B.8).

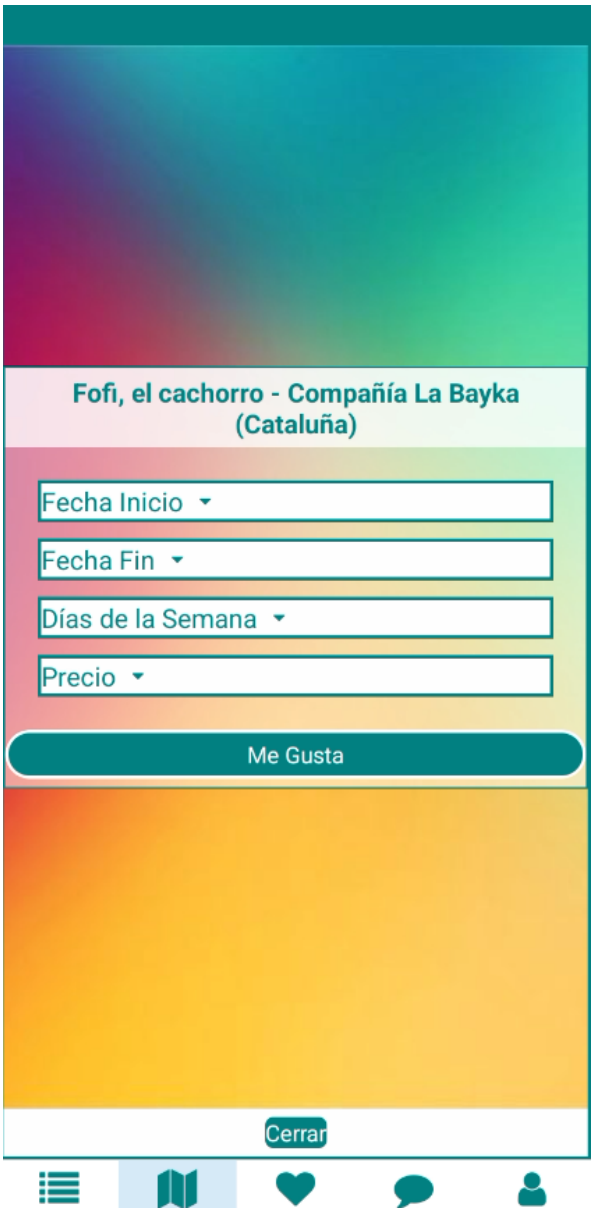


Figura B.8: Pantalla del mapa - información de evento seleccionado

B.4. Me gusta

Accedemos a esta pantalla del mismo modo que a las anteriores, seleccionando su icono en el menu del extremo inferior de la pantalla. Esta pantalla muestra los "me gusta" que el usuario ha dejado, mostrando información sobre los eventos del mismo modo que la pantalla Explorador de eventos, con la diferencia de que muestra un botón más con el texto "Mira a quién más le gusta este evento" (ver imagen B.9).

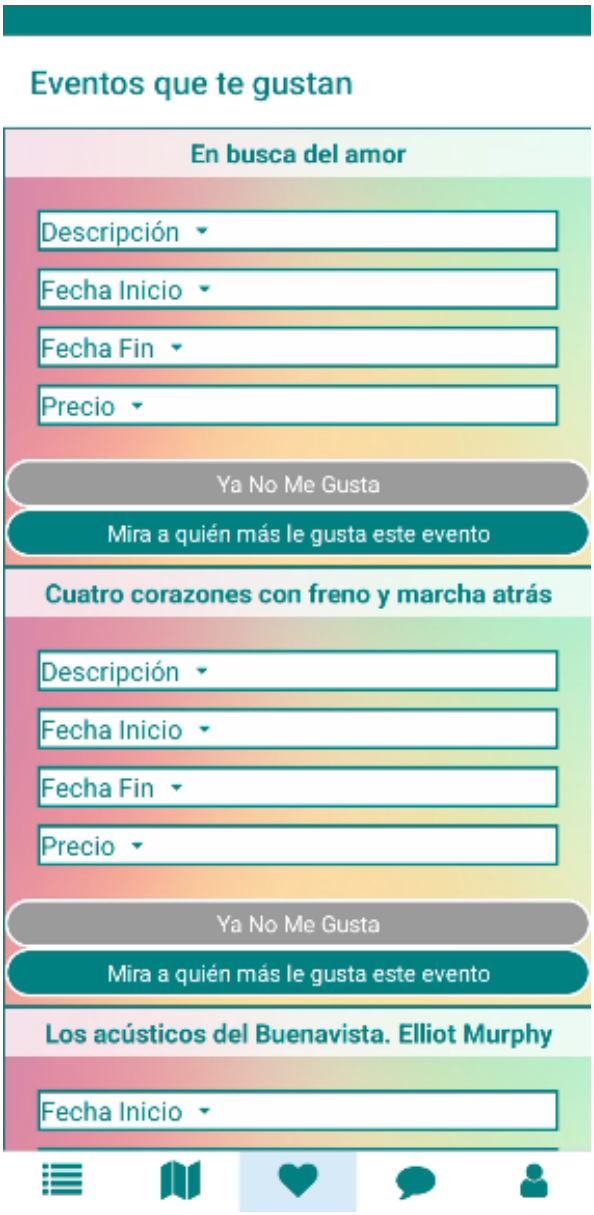


Figura B.9: Pantalla de me gusta

Si aún no hemos dejado ningún "me gusta", en vez de una lista de eventos nos aparecerá el texto que se puede observar en la imagen B.10.

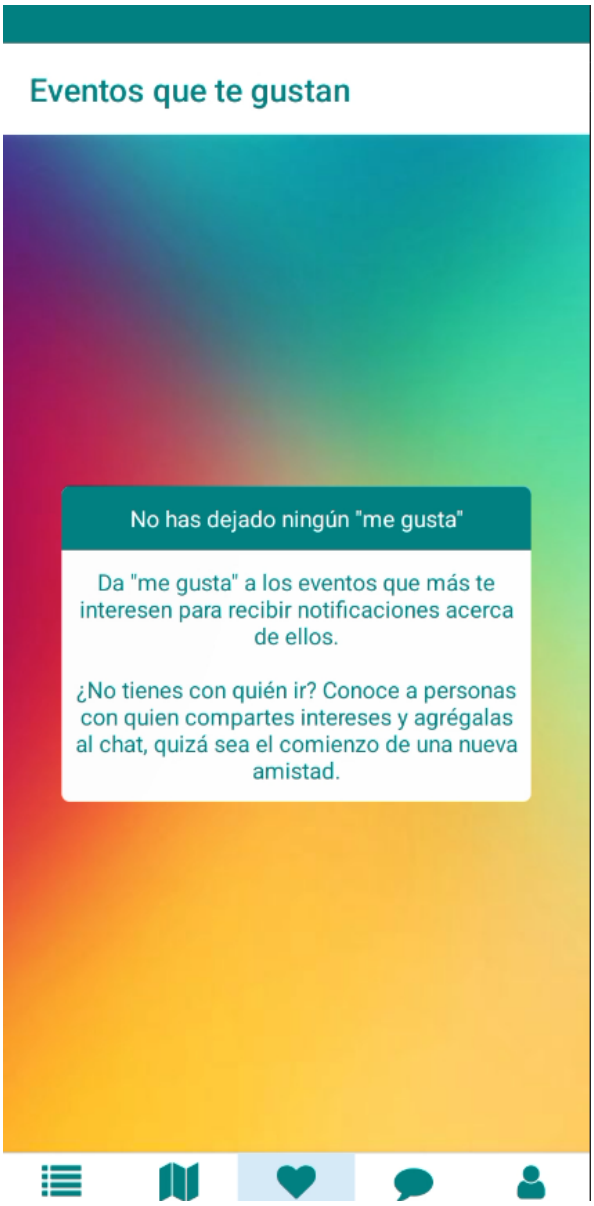


Figura B.10: Pantalla de me gusta en caso de no haber dejado ningún "me gusta"

Al pulsar sobre el botón "Mira a quién más le gusta este evento", como podemos intuir por el texto que contiene, podremos visualizar una lista de usuarios que también han dejado un "me gusta" al evento. En caso de haber sido el único usuario que ha dejado un "me gusta" al evento, se mostrará en vez de la lista el mensaje que vemos en la imagen B.11.

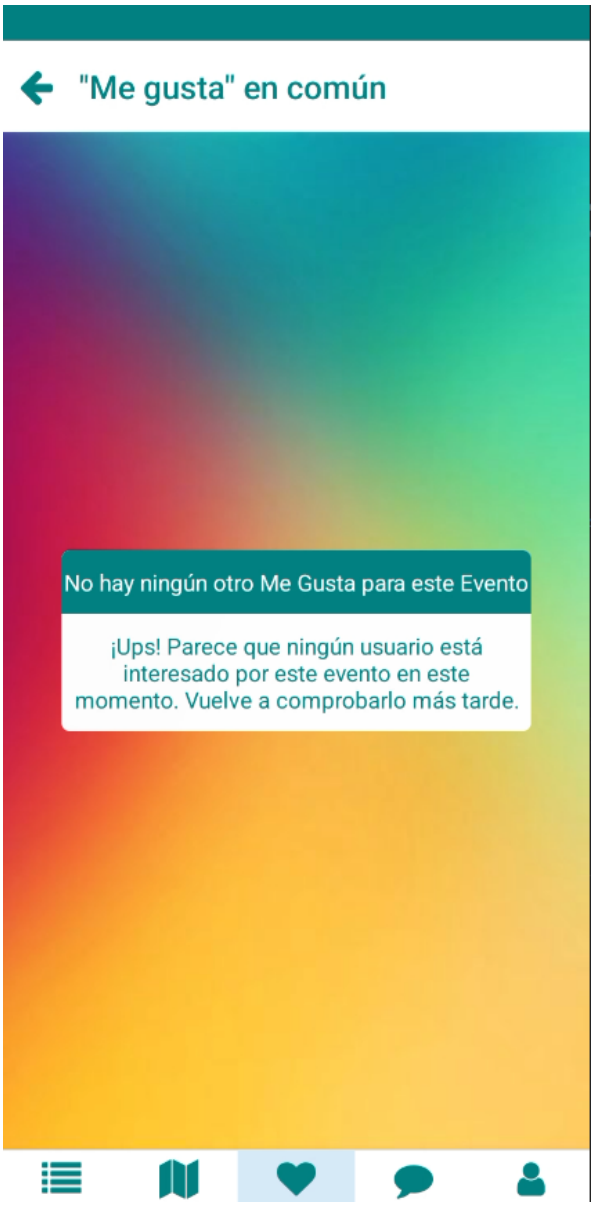


Figura B.11: Lista de “me gusta” en común vacía

Cuando otros usuarios han dejado un “me gusta” al evento, se mostrará la lista de dichos usuarios como vemos en la imagen B.12.

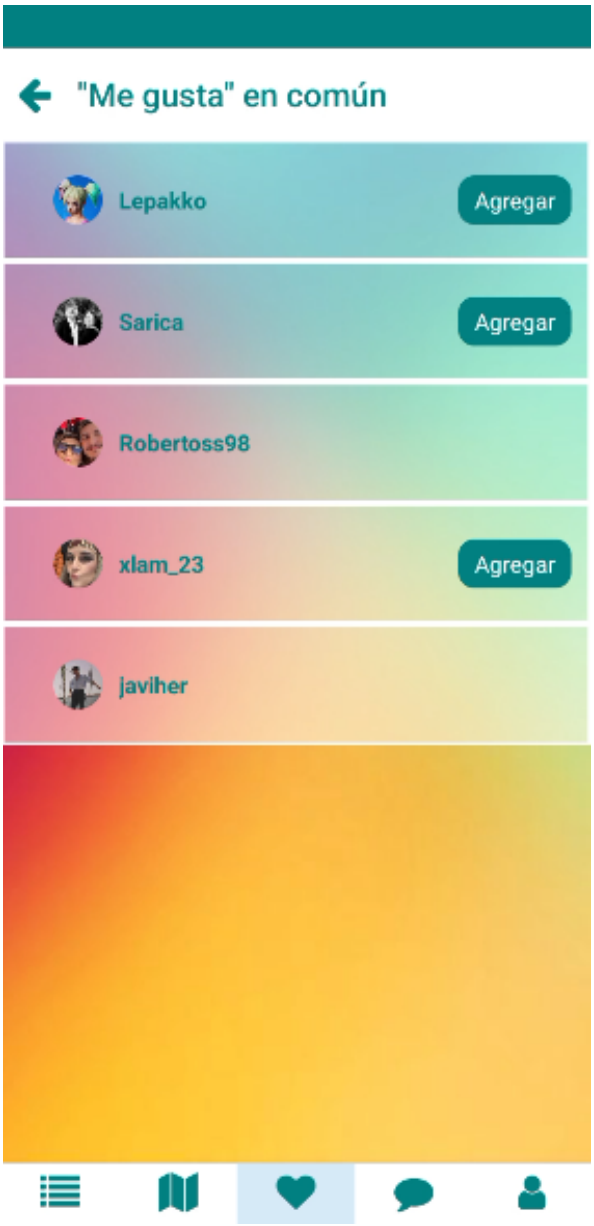


Figura B.12: Lista de usuarios con un “me gusta” en común

Al pulsar un usuario, podremos visualizar su perfil, donde encontraremos una lista de los diferentes tipos de eventos ordenadas en orden descendente según el número de eventos de ese tipo al que el usuario ha dejado “me gusta”. Además podremos visualizar su foto de perfil, nombre de usuario y descripción en caso de tenerla (ver imagen B.13).

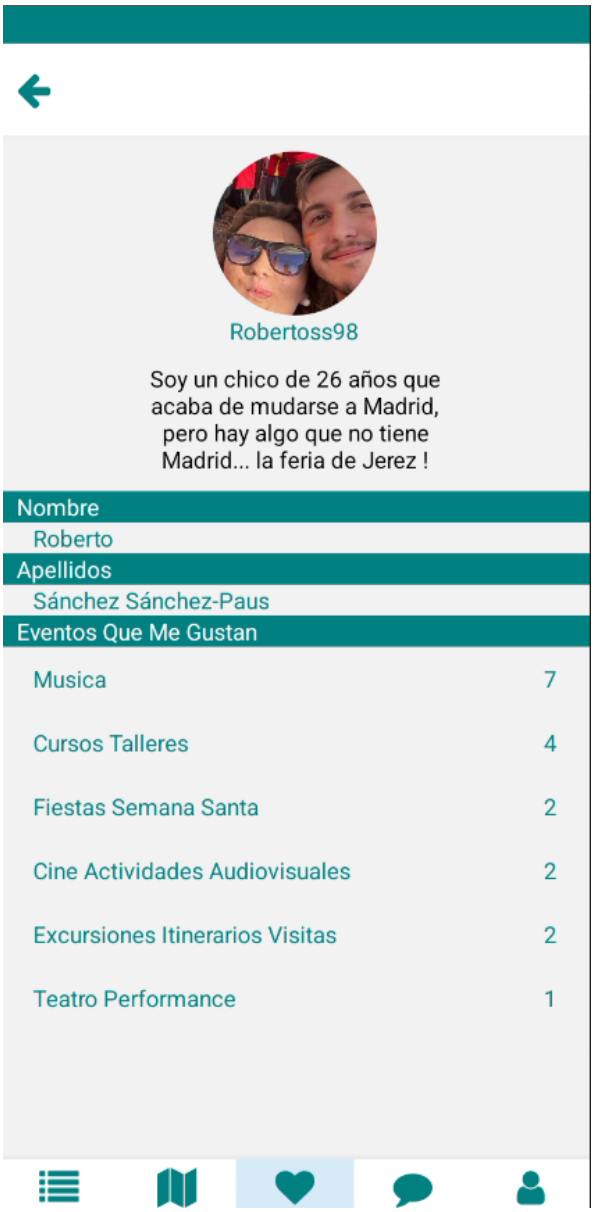


Figura B.13: Consulta de perfil de otro usuario

Si volvemos la vista a la imagen B.12 podemos observar que cada usuario tiene, a excepción de uno, el botón "Agregar a mis Chats". Esto ocurre porque ese usuario ya ha sido agregado previamente.

B.5. Chats

Se accede a esta pantalla del mismo modo que a las anteriores, seleccionando su icono en el menu del extremo inferior de la pantalla. Esta pantalla muestra los chats del usuario. Para un usuario que no pertenezca a ningún chat, la pantalla mostrará el mensaje que podemos observar en la imagen B.14.

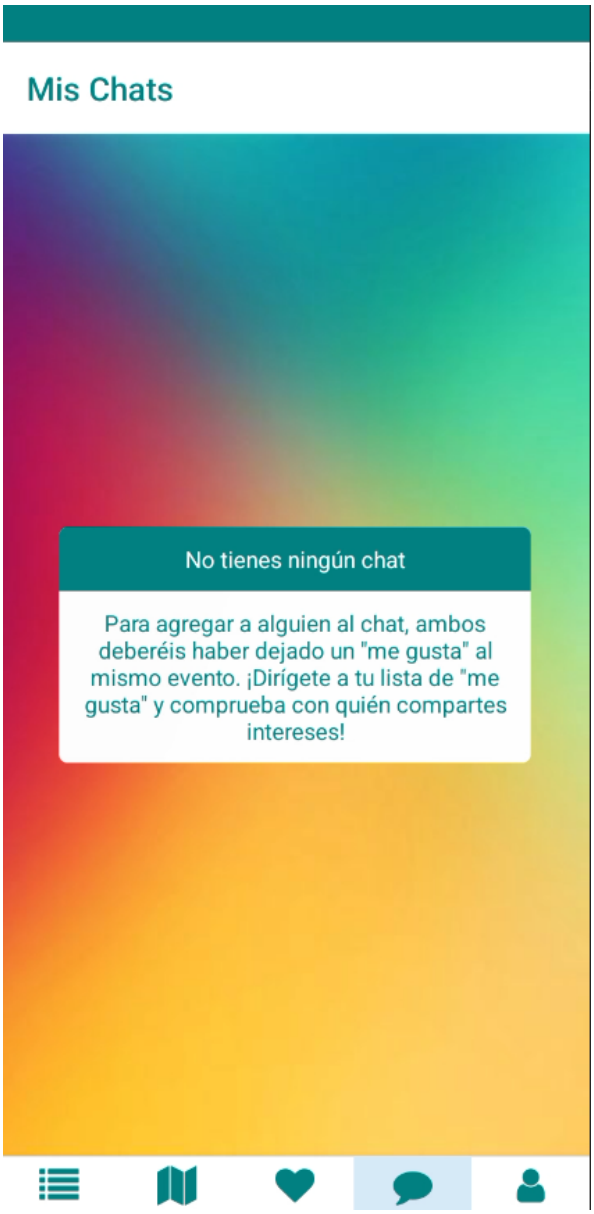


Figura B.14: Lista de chats vacía

Para un usuario que pertenece a algún chat, la pantalla mostrará una lista de sus chats(ver imagen B.15).

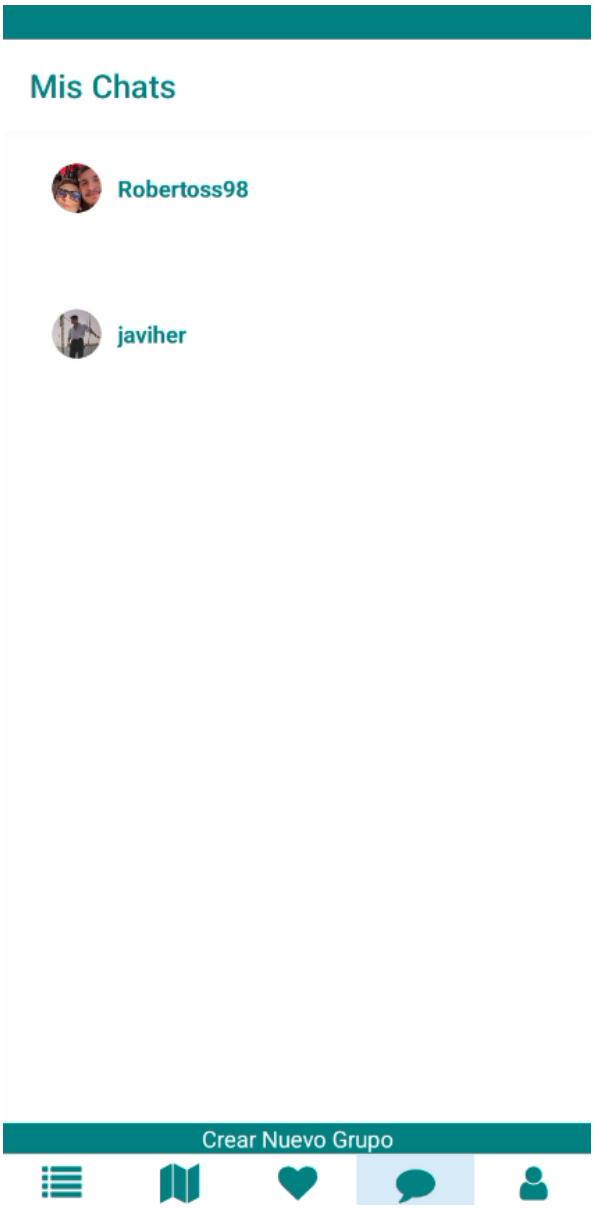


Figura B.15: Lista de chats

Podemos acceder a un chat pulsando sobre él. Veremos la pantalla que se muestra en la imagen B.16, donde podremos intercambiar mensajes con el otro usuario. El botón del extremo inferior derecho de la pantalla sirve para bajar rápidamente hasta el último de los mensajes del chat.

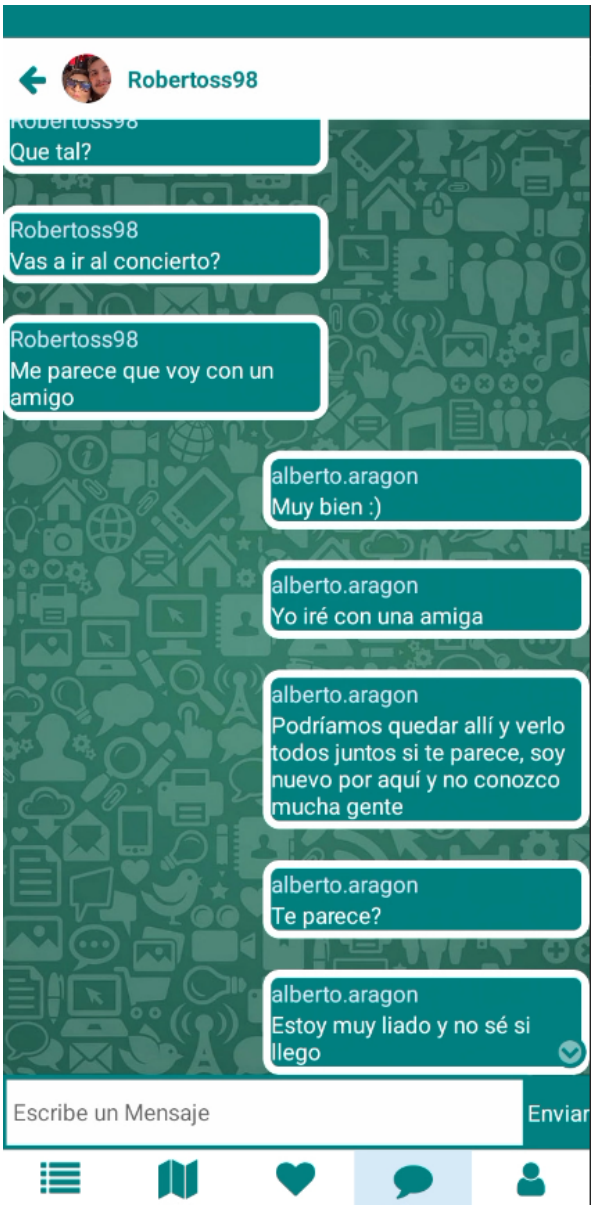


Figura B.16: Intercambio de mensajes en un chat

Podemos acceder también al perfil del usuario pulsando sobre su nombre de usuario en el extremo superior de la pantalla, y nos mostraría la misma pantalla que veíamos al acceder al perfil de otro usuario desde la pantalla de Me gusta.

Volviendo la vista a la imagen B.15, vemos que en el extremo inferior de la pantalla hay un botón con el texto "Crear Nuevo Grupo". Al pulsarlo, veremos la pantalla que muestra la imagen B.17. Podremos seleccionar una imagen para el grupo de la galería de nuestro dispositivo y darle un nombre al grupo. Disponemos de una lista de usuarios que tenemos agregados al chat y podremos añadirlos al grupo.

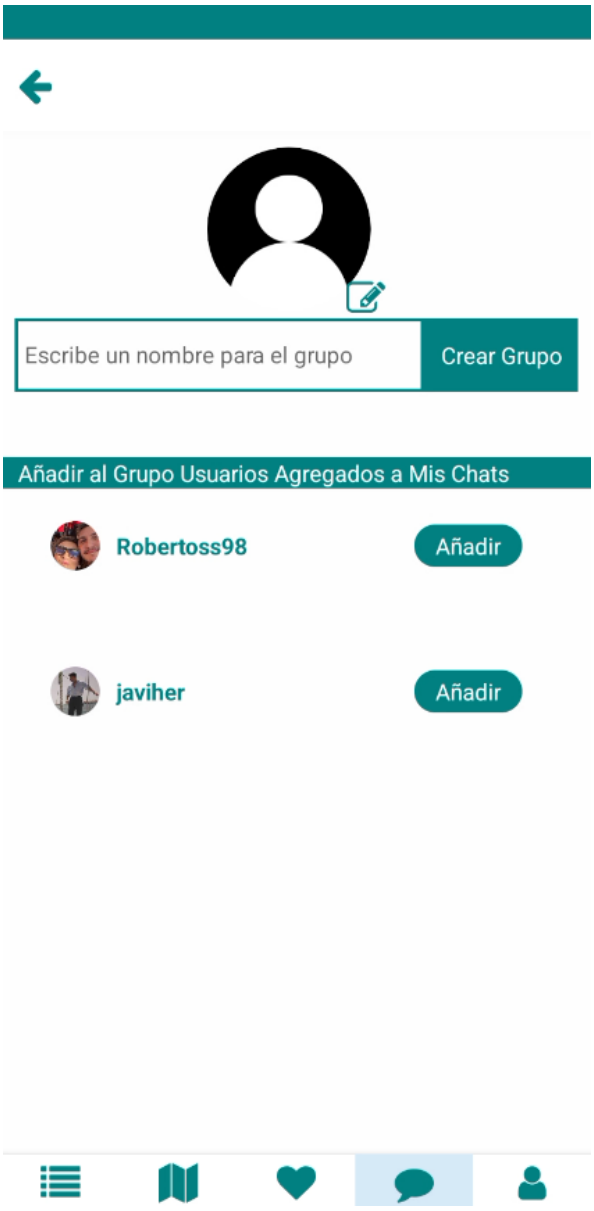


Figura B.17: Creación de un chat grupal

En caso de intentar crear el grupo sin añadir a ningún usuario, saldrá una alerta con el mensaje que vemos en la imagen B.18.

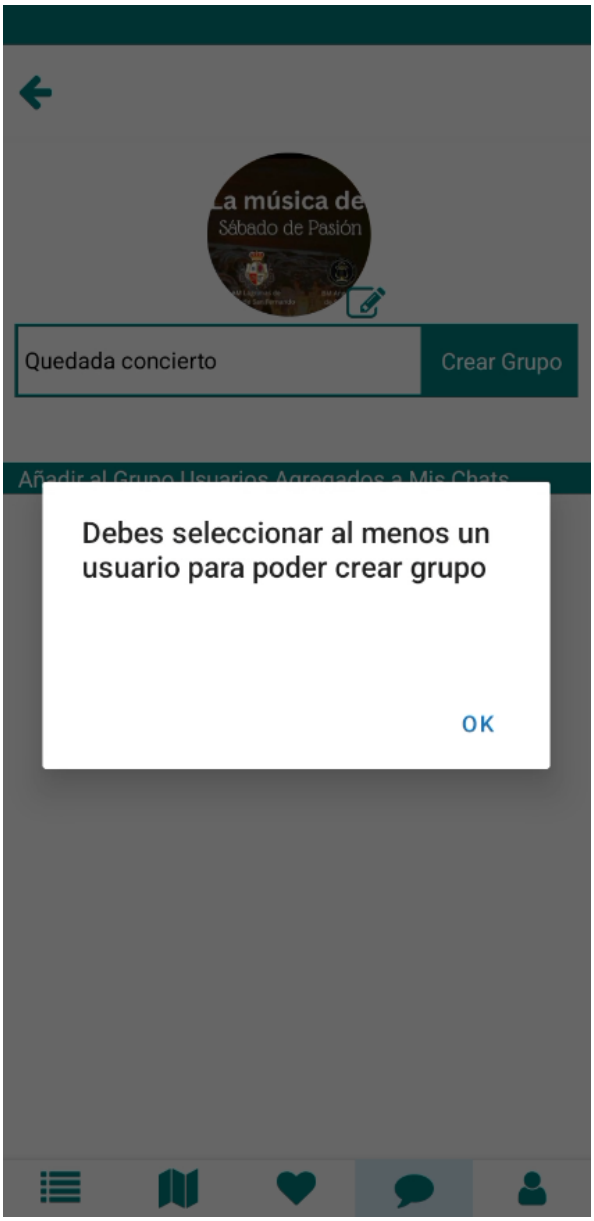


Figura B.18: Alerta al intentar crear un chat grupal sin añadir usuarios

Una vez creado el grupo, nos aparecerá en nuestra lista de chats. Al entrar al chat grupal, podremos intercambiar mensajes con el resto de componentes (ver imagen B.19).

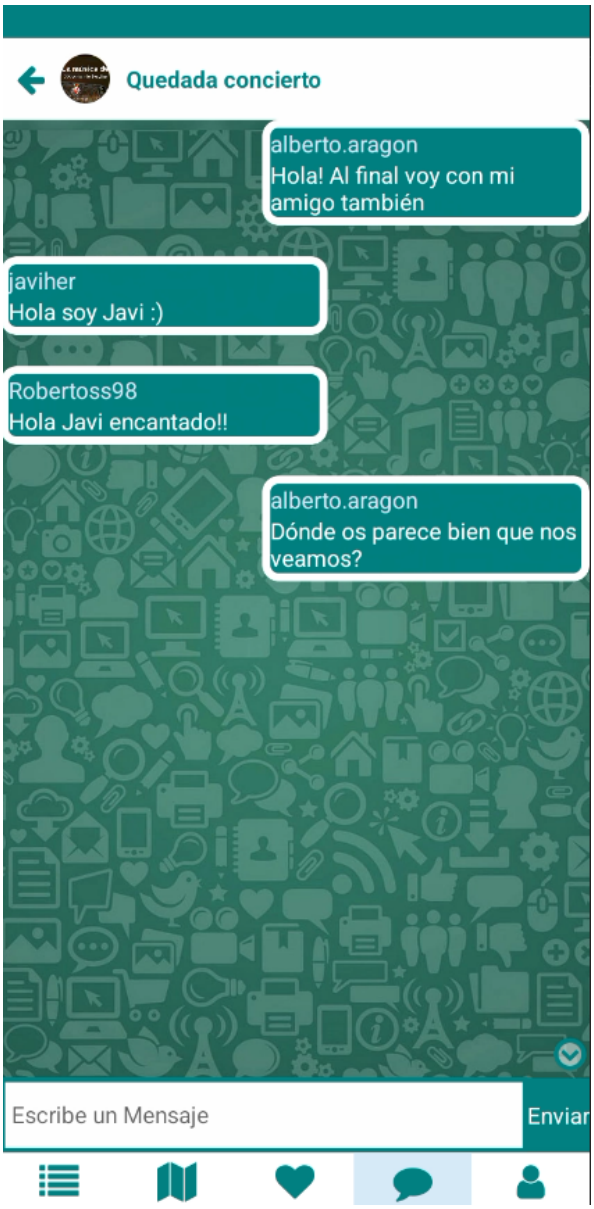


Figura B.19: Intercambio de mensajes en un chat grupal

Al pulsar el nombre del grupo en el extremo superior de la pantalla, navegaremos a la pantalla de la imagen B.20, desde donde podremos eliminar a algún componente del grupo o podremos abandonar el grupo nosotros mismos.

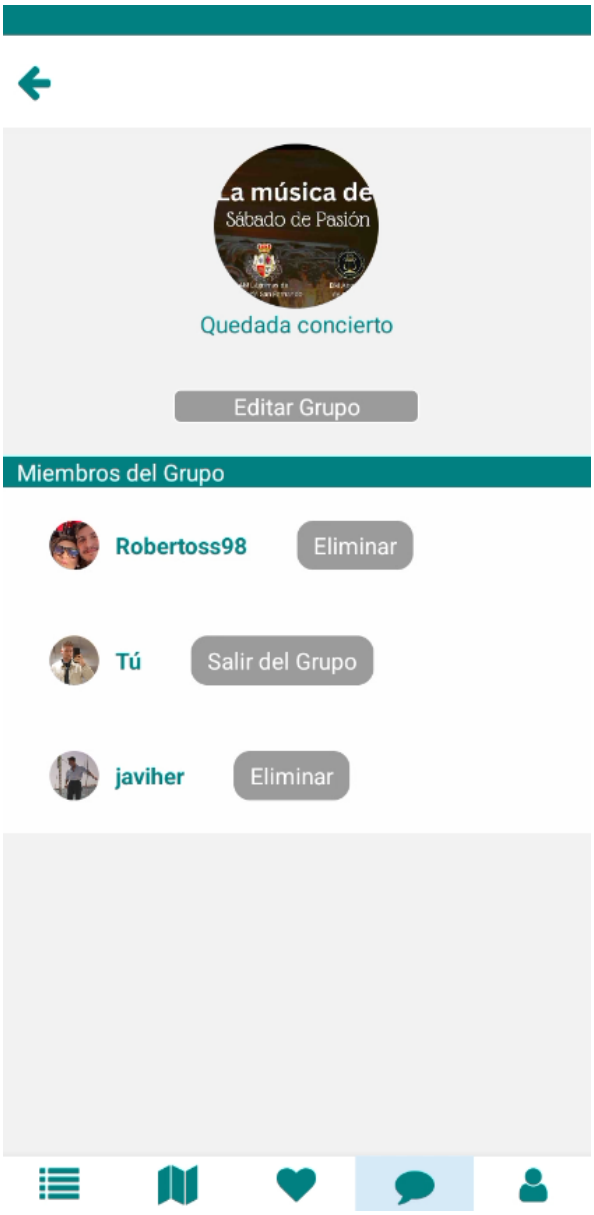


Figura B.20: Pantalla de consulta de grupo

Al pulsar el botón "Editar Grupo", navegaremos a una pantalla casi idéntica a la que se accede al crear un nuevo grupo , pudiendo realizar cambios en su nombre e imagen, añadir y eliminar usuarios del grupo (ver imagen B.21).

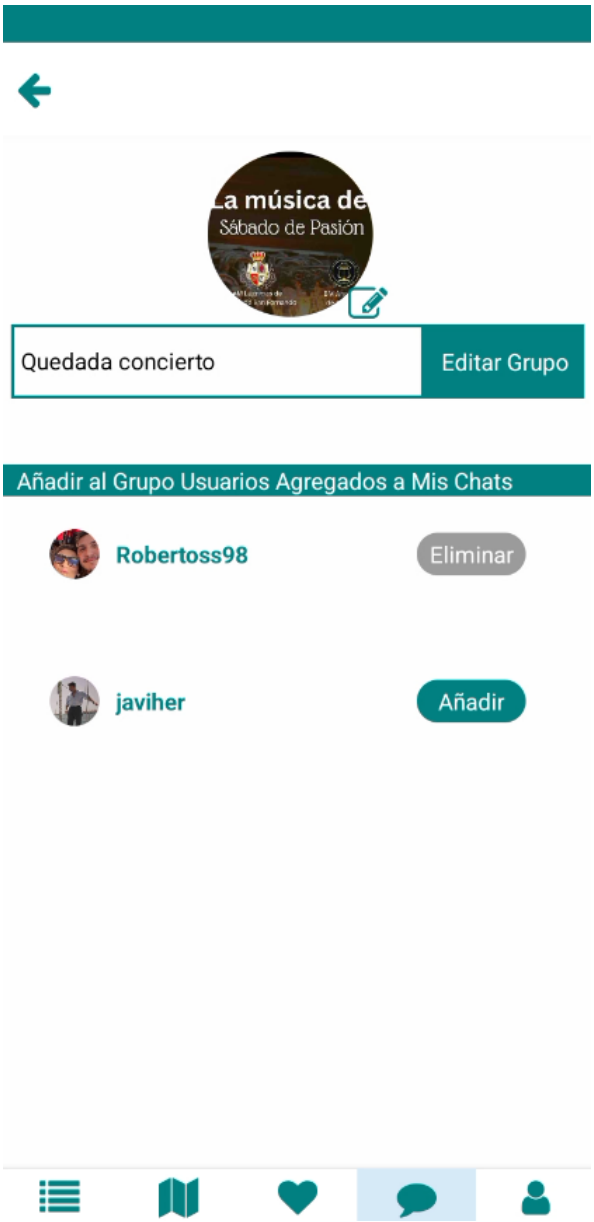


Figura B.21: Pantalla de edición de grupo

En el caso de que un grupo esté compuesto solo por otro usuario y nosotros no podremos eliminarlo ni abandonar el grupo, se mostrará el mensaje de la imagen B.22 y la única opción sería eliminar el grupo desde la pantalla de lista de chats.

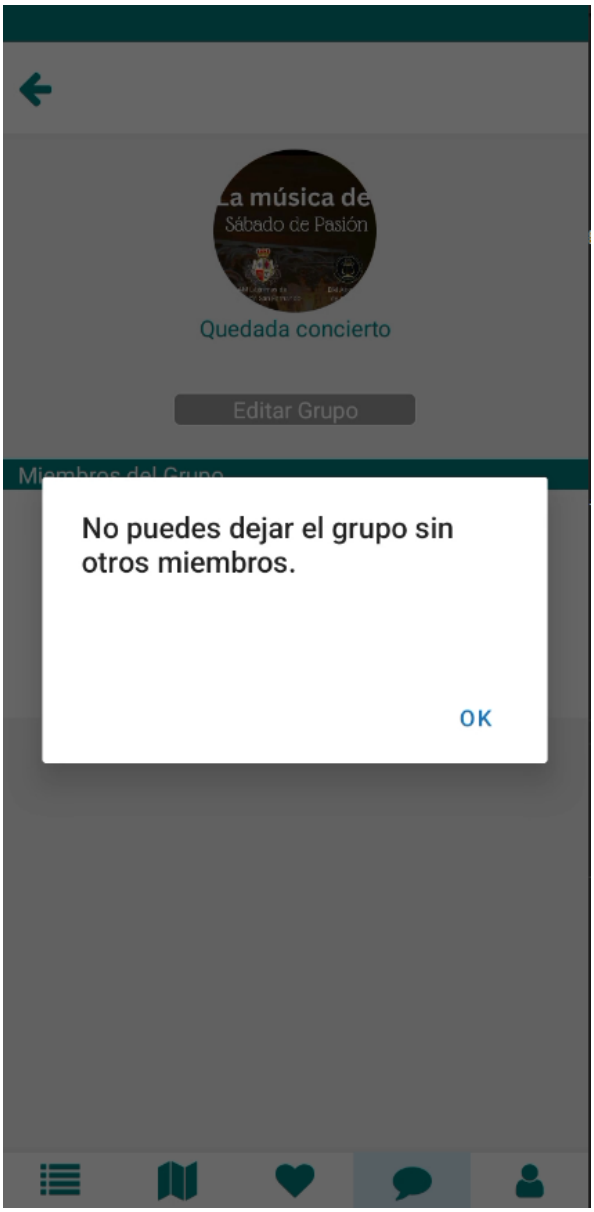


Figura B.22: Alerta al intentar dejar el grupo con un solo miembro

Para eliminar un chat, debemos mantener pulsado sobre el nombre del chat en la lista de chats, y nos saldrá el mensaje de la imagen B.23.

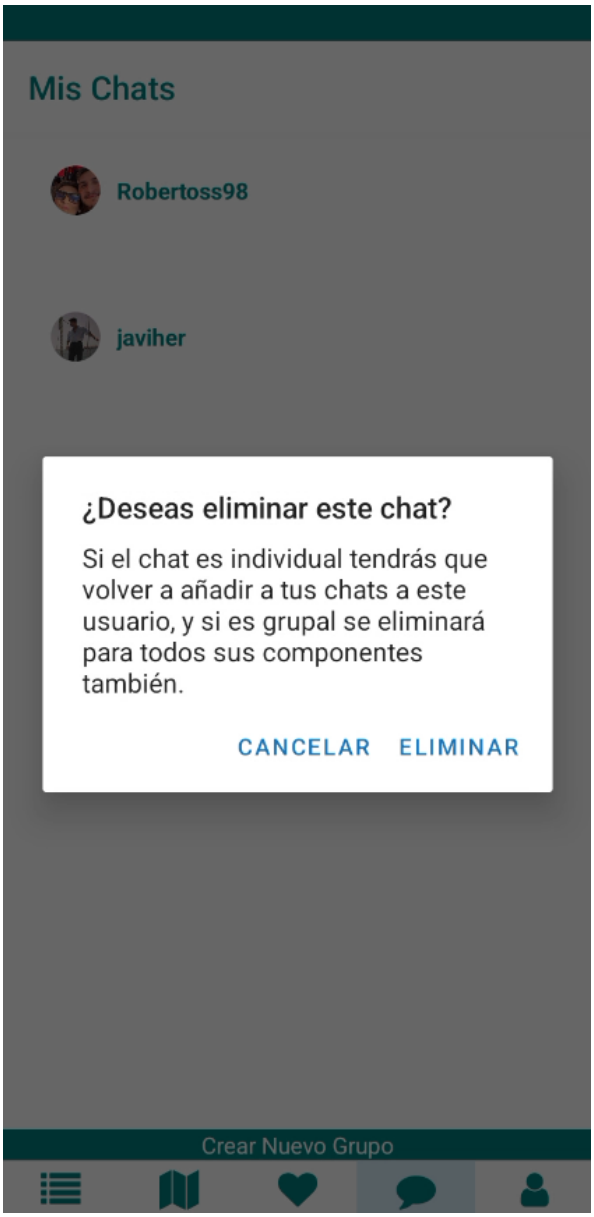


Figura B.23: Mensaje al intentar eliminar chat

B.6. Perfil

Accedemos a esta pantalla pulsando su icono en el menú del extremo inferior de la pantalla. Si observamos la imagen B.24 vemos que se nos muestra una pantalla al igual que la que navegábamos al consultar el perfil de otro usuario, a diferencia de que en esta se nos permite cerrar sesión (en cuyo caso navegaremos a la pantalla de inicio de sesión) y editar el perfil.

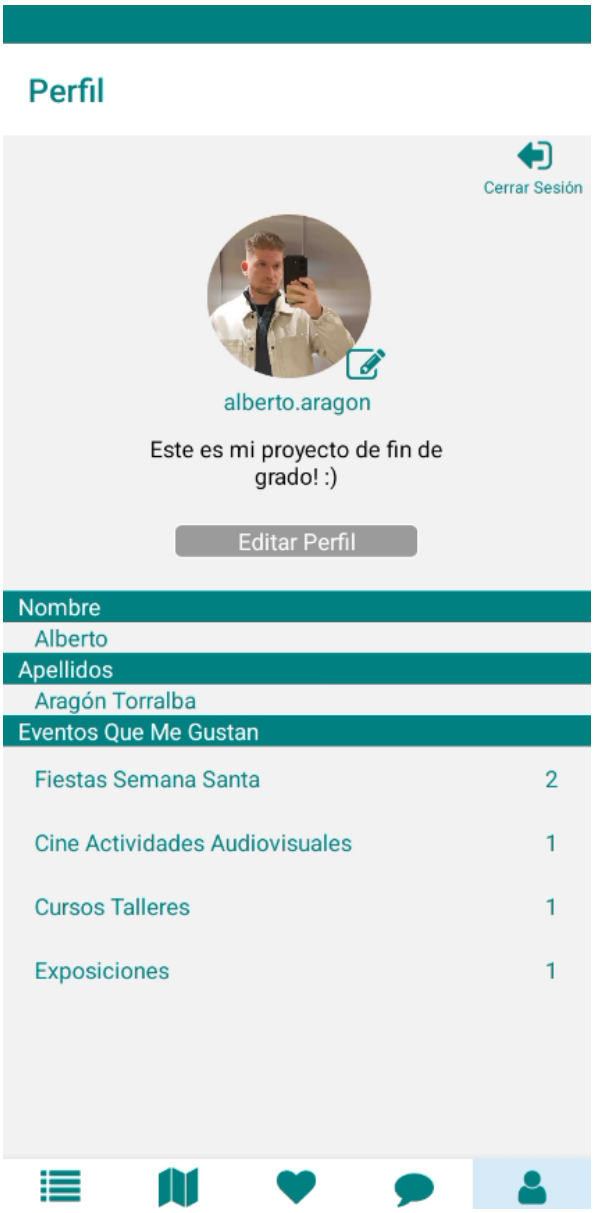


Figura B.24: Pantalla de perfil de usuario

En el caso de que no hayamos dado "me gusta", se mostrará en vez de la correspondiente lista el mensaje de la imagen B.25.

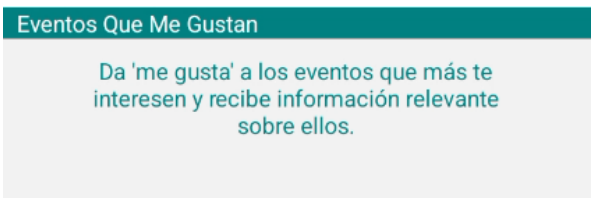


Figura B.25: Pantalla de perfil de usuario sin "me gusta"

Se nos permite editar la imagen sin pulsar el botón "Editar Perfil", pero para editar el resto de datos sí tendremos que pulsarlo, y veremos la pantalla que se muestra

en la imagen B.26.



Figura B.26: Pantalla de edición de perfil de usuario