# Ahoy App

Sergio Martín Segura

## Contents

## Introducción y Objetivos

El objetivo de este proyecto es diseñar e implementar un sistema distribuido, concretamente, una plataforma de chat en tiempo real.

Los principios fundamentales de este sistema serán:

- Aproximación *Cloud Native*: La distribución en la nube será un aspecto a considerar desde el inicio del diseño.
- Escalabilidad horizontal: Se tratará de maximizar el paralelismo.
- Alta disponibilidad: Se tratarán de minimizar los puntos de fallo único buscando replicación en los servicios críticos.
- Minimizar deuda técnica: Como un proyecto que aspira a un ciclo de vida largo, se buscará minimizar las decisiones corto-placistas que repercutan de forma negativa en la progresión y mantenimiento del proyecto.

## Diseño del sistema

Debido a las características solicitadas por el Dueño de Producto, podemos asumir que los principales requerimientos del sistema son:

- Una aplicación web cliente fácil de usar y compatible con los principales navegadores modernos.
- Una servidor que ofrezca una API que permita la cliente realizar operaciones síncronas en el sistema.
- Un servidor que ofrezca una API que permita al cliente recibir eventos relevantes para mantener su estado sincronizado con el del backend.

Ademas de estos requisitos básicos, podemos inferir que también necesitaremos:

- Un sistema de persistencia que descarge al servidor de la responsabilidad de mantener estado.
- Un sistema de mensajería que solucione el paso de eventos entre el servidor REST síncrono y el servidor WS de eventos
- Un sistema de almacenamiento de objetos para la subida y descarga de ficheros
- Un sistema de autenticación que no requiera verificación contra el sistema de persistencia ni afinidad de sesión

### Decisiones de diseño

Son muchas las opciones que un desarrollador web full-stack puede tomar a la hora de implementar un servicio web. El abanico de herramientas puede abrumar en un principio y a menudo las decisiones se basan en lo ya conocido o en sesgos propios. Para este

proyecto, se ha tratado de tensar el equilibrio entre aprender nuevas herramientas y aprovechar los conocimientos previos sobre otras.

### Node + Express vs Python + Flask vs Java + Spring

Empezando por el framework para el lado del servidor, se barajaron inicialmente tres opciones: Node, Python y Java.

Los puntos a favor de Python eran su aparentemente fácil desarrollo de código y su ecosistema Flask+Tornado que parecía ofrecer buenos resultados combinando HTTP y Web Sockets. No obstante, la experiencia manejando Python no era alta y se decidió desecharlo como opción en pos de un menor esfuerzo de aprendizaje.

El primer prototipo funcional se implementó en Java + Spring Boot con el módulo Spring Messaging. Si bien abstraía los detalles de implementación enormemente, no ofrecía el control de bajo nivel que se buscaba en la implementación para aplicar los patrones EIP que se desearan. Se encontraron otras aproximaciones dentro del ecosistema Spring como el manejo de Web Sockets a bajo nivel con el handleTextMessage, pero resultaban contra-intuitivas en la filosofía de Spring ya que obligaban a gestionar el estado de la conexión manualmente al combinarse con RabbitMQ. Además, era complicado encontrar documentación ya que el uso de Web Sockets en Spring parece estar *de facto* vinculado al uso de STOMP, cosa que no se buscaba en este proyecto.

Finalmente, se optó por Node por varios motivos. El primero de ellos era buen equilibrio que ofrecía entre control de bajo nivel y sencillez de programación al explotar la naturaleza funcional y asíncrona de Javascript. De ese modo, se podía declarar fácilmente una estructura de callbacks que orquestaran correctamente los eventos de Rabbit y del Web Socket sin preocuparse por mantener manualmente una lista de sesiones activas o similar. El otro motivo fue las mejoras en rendimiento que parecía ofrecer en varios *benchmarks* Node contra Java, especialmente en la gestión de bloqueos y concurrencia.

### REST vs GraphQL

Si bien GraphQL era una de esas tecnologías que se buscaba aprender, los propios requerimientos del Product Owner y el tiempo disponible para realizar el proyecto impidieron su inclusión. No obstante, se valora que habría sido especialmente interesante de cara a la implementación de la aplicación web ya que habría permitido desacoplarla de la API REST y delegar en ella el formato de consumo de los datos así como las optimizaciones y la carga especulativa. Por ello la solución escogida finalmente fue una API REST tradicional.

### WebSocket vs Server Side Events

En un punto intermedio del proyecto, se tomó la decisión indirecta de que los mensajes serían enviados del cliente al servidor vía API REST y no aprovechando la bidireccionalidad del canal Web Socket. El motivo de esta decisión no fue otro que la sencillez en la implementación, ya que esto permitía, por un lado, reutilizar todo el código similar del resto de controladores REST y por otro, delegar el canal de Web Socket a un canal unidireccional de eventos sin acciones persistentes.

Esta decisión abrió entonces la puerta a reemplazar la tecnología Web Socket por SSE. Los dos principales motivos para no hacerlo fueron, por un lado, la falta de soporte de algunos navegadores modernos, y por otro, el tiempo y esfuerzos extra que supondría adoptar una tecnología nueva, incluyendo readaptar el sistema actual de autentificación.

### JSON Web Token vs Session Cookie

Existen varias formas de autentificar a un usuario en un servicio web. Si bien las session cookies son ampliamente utilizadas, se buscaba una solución que permitiera eliminar la afinidad en el balanceo sin delegar la persistencia de la cookie en una base de datos. Por ello se decidió adoptar JWT como sistema, ya que delega la persistencia de la sesión en el cliente de un modo seguro. Este sistema ofrece, sin embargo, una desventaja: la dificultad de revocación de una sesión, problema que quedaba completamente fuera del alcance de este proyecto, por lo que no supuso un argumento de peso para no adoptar JWT.

### Mongo vs SQL

Actualmente es difícil encontrar un servicio web que no nazca sobre Mongo DB. Sus promesas de escalado y distribución son sólidas y existen numerosas herramientas que facilitan su uso hasta un punto caso trivial en la mayoría de casos cubiertos. Sin ánimo de caer en una moda irracional, y asumiendo que, en la mayoría de los casos, una base de datos relacional habría sido la opción correcta, se optó finalmente por Mongo debido a las herramientas mencionadas. Una solución más madura en un marco profesional podría ser una combinación de SQL para el modelo de Usuarios y Salas y una base de datos de baja latencia como Redis para los Mensajes.

### GridFS vs Minio

En una de las sesiones teóricas se introdujo Minio como almacén de objetos distribuido. A pesar de sus bondades, la experiencia con él y concretamente, la experiencia desplegándolo, hicieron que se abandonara desde un principio la idea de su uso. Ofrece un pobre soporte para las cuentas de usuario, lo que impide su exposición directa por motivos de seguridad.

Sabiendo esto, y asumiendo que será necesario entonces implementar un intermediario que autentifique al usuario y retransmita el fichero, se decidió aprovechar la base de datos ya desplegada, Mongo, y su soporte para ficheros de gran tamaño GridFS.

### React vs Vanilla

La elección tecnológica para el lado del cliente sufría una fuerte restricción: Cualquiera que fuera la herramienta escogida, tenía que permitir un control preciso del estado de la aplicación en todo momento; permitir su actualización de dicho estado tanto por acción del usuario como por eventos provenientes del servidor; y ofrecer una vista sincronizada con el estado en todo momento.

La complejidad de implementar algo así manualmente en vanilla era casi tan alta como la probabilidad de introducir serios errores de concurrencia y glitches visuales. Por ello se decidió utilizar un framework ya conocido: React. React, en conjunción con Redux, ofrecen un flujo de datos unidireccional y actualizaciones de estado secuenciales y deterministas así como vistas siempre sincronizadas con dicho estado.

## Escenarios de comportamiento

Los escenarios modelados a continuación representan las secuencias más representativos que se dan lugar en la aplicación.

### Sign Up

La secuencia de registro incluye, además, a la secuencia de login ya que, por diseño, se ha decidido que ambos procesos sean indistinguibles para el usuario. Esto es: La primera vez que un usuario acceda al sistema, se le creará una cuenta de usuario vinculada a sus credenciales de terceros, para que la próxima vez que acceda, se recupere dicha sesión.

Para el acceso, es necesario poseer previamente una cuenta en GitHub. En el proceso, de solicita a la API OAuth de GitHub un código que, por un lado, autentique al usuario y por otro, permita obtener un access_token con permisos de lectura sobre la información del usuario. De este modo, el servidor podrá obtener el correo electrónico y el nickname. En el futuro, sería posible incluir también el icono de usuario y otra información relevante.

Una vez creado o recuperado el usuario de la base de datos, se firma un token JWT propio del servicio web (no confundirlo con el JWT obtenido de GitHub y firmado por GitHub) y se le envía al cliente para que lo almacene de forma segura y lo use en futuras peticiones.

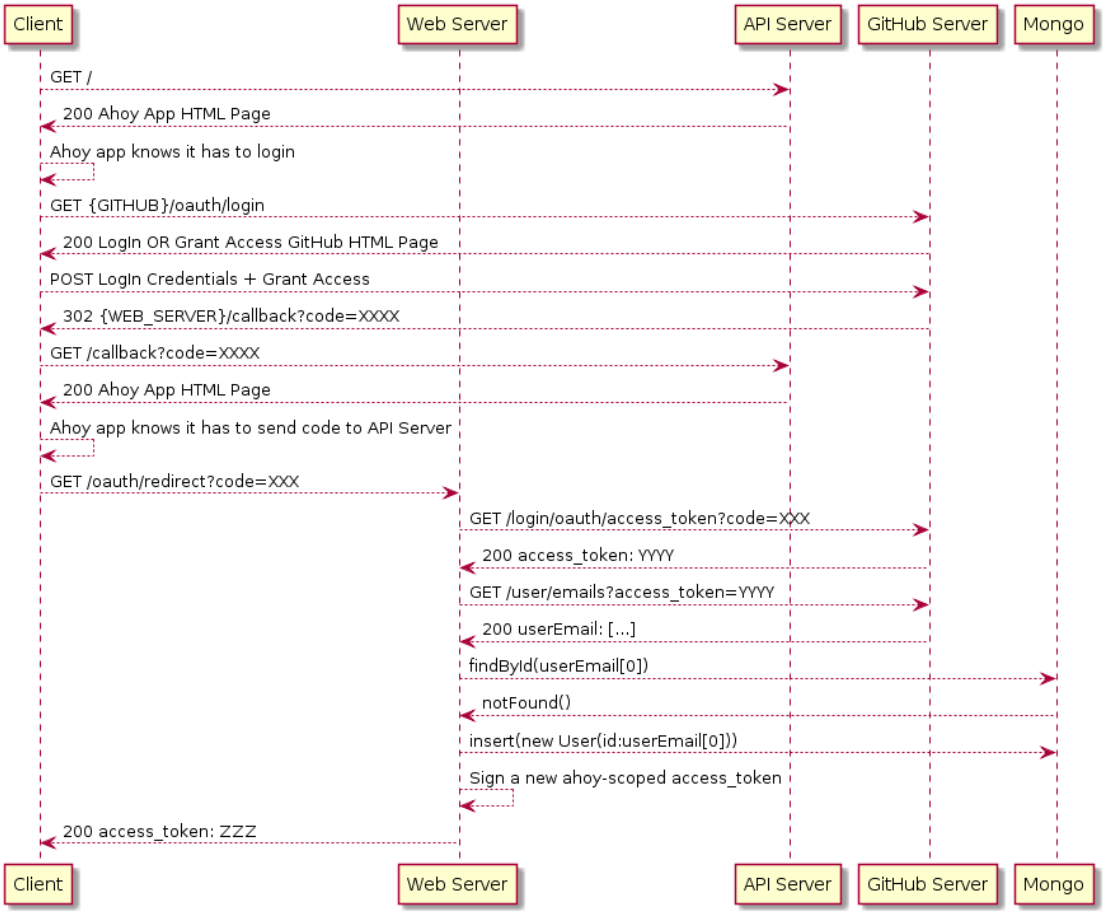


Figure 1: Sign Up

### Connect to WS

La conexión con el Web Socket se hace en dos pasos definidos por el protocolo: primero un handshake hecho sobre HTTP clásico y a continuación, el establecimiento de la conexión persistente. En la primera etapa, se usa el token enviado en la petición HTTP para autentificar al usuario y en la segunda etapa se recupera la información del relevante del usuario para suscribirlo a los tópicos relevantes, como pueden ser en este caos, las salas en las que está.

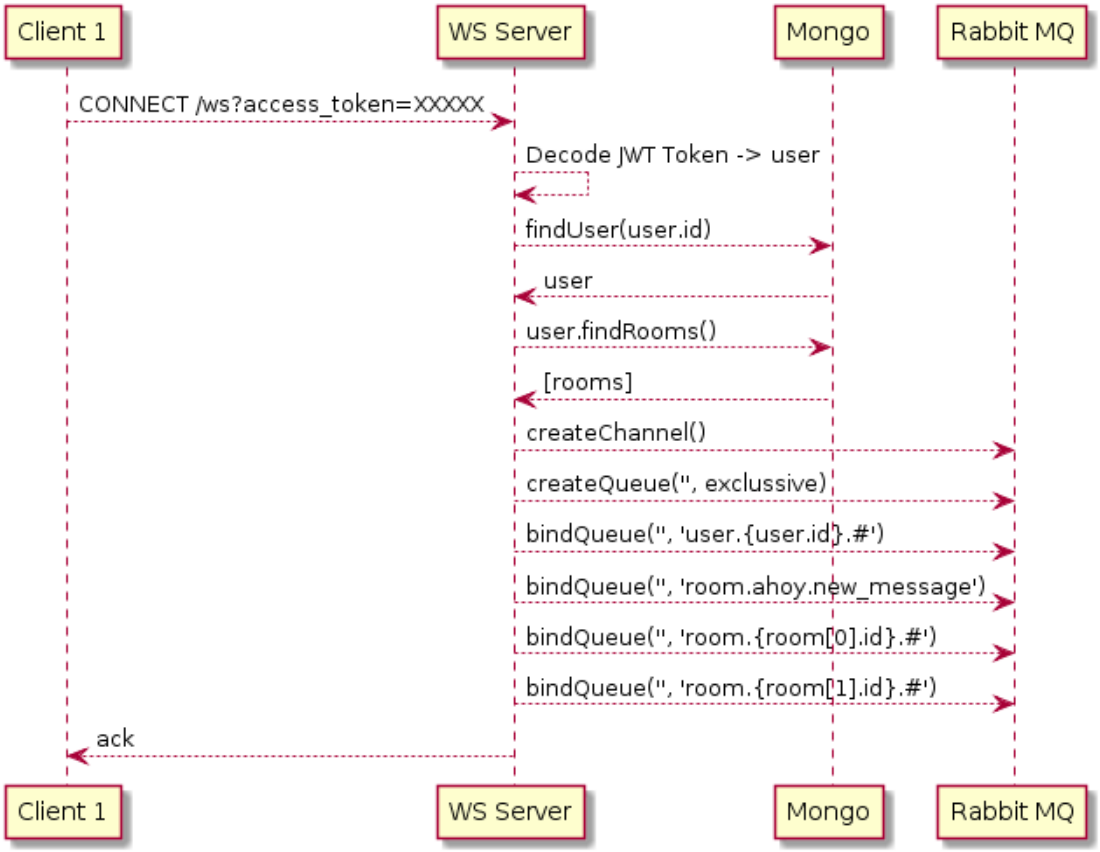


Figure 2: Connect WS

### Send Message

Este escenario concreto, no sólo representa la interacción más habitual en el sistema, sino que también representa al resto de peticiones HTTP de escritura del sistema. Se suelen componer estas de tres fases: lectura de la base de datos y verificación de permisos; escritura del resultado del cambio en la base de datos; y finalmente propagación de un evento que notifique el cambio.

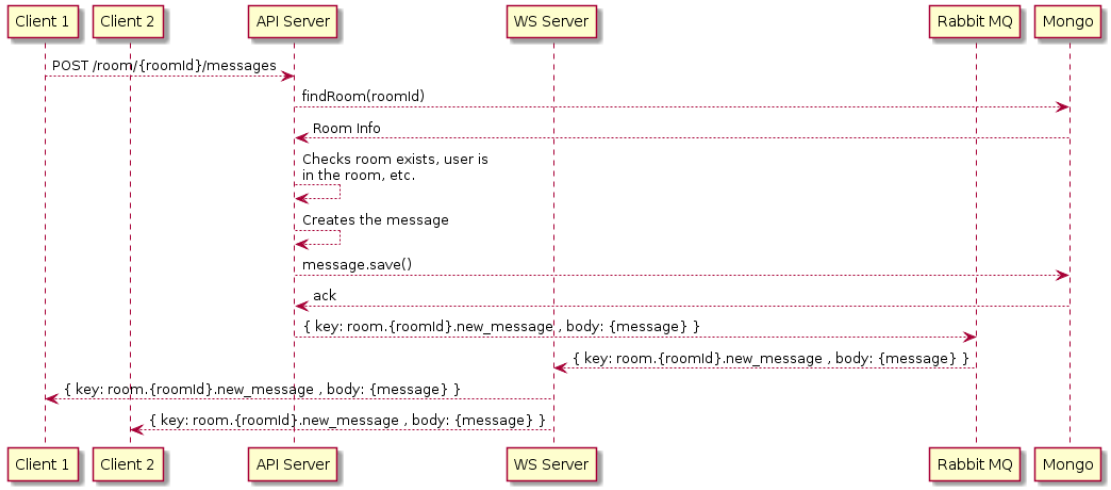


Figure 3: Send Message

### Kick User

Este último escenario extiende las fases del escenario anterior con un paso extra. Al añadir o echar a un usuario de una habitación, cada cola (vinculada a una sesión Web Socket activa) debe añadir o quitar un binding al tópico de esa habitación. De ese modo, conexiones existentes pueden escuchar en salas que no podían escuchar en su inicio.

## Decisiones Arquitecturales y Patrones implementados

La arquitectura escogida para esta aplicación es la tradicional Cliente-Servidor 3-Tier. El motivo de ello es la sencillez que ofrece frente a una 4-Tier, aspecto interesante dado el poco tiempo disponible para la implementación. Si bien habría sido deseable incluir un gateway que resolviera tanto el login como el logging, quedaba fuera del alcance del proyecto, además de que implicaría resolver problemas no triviales, como el forwarding de los Web Sockets.

### Depliegue

Empezaremos por la vista de despliegue para entender cuanto antes los nodos que conforman la aplicación en su conjunto.

Los nodos de la aplicación son entonces:

- Cliente con la aplicación web
- Servidor de la aplicación web cliente
- Servidor de API

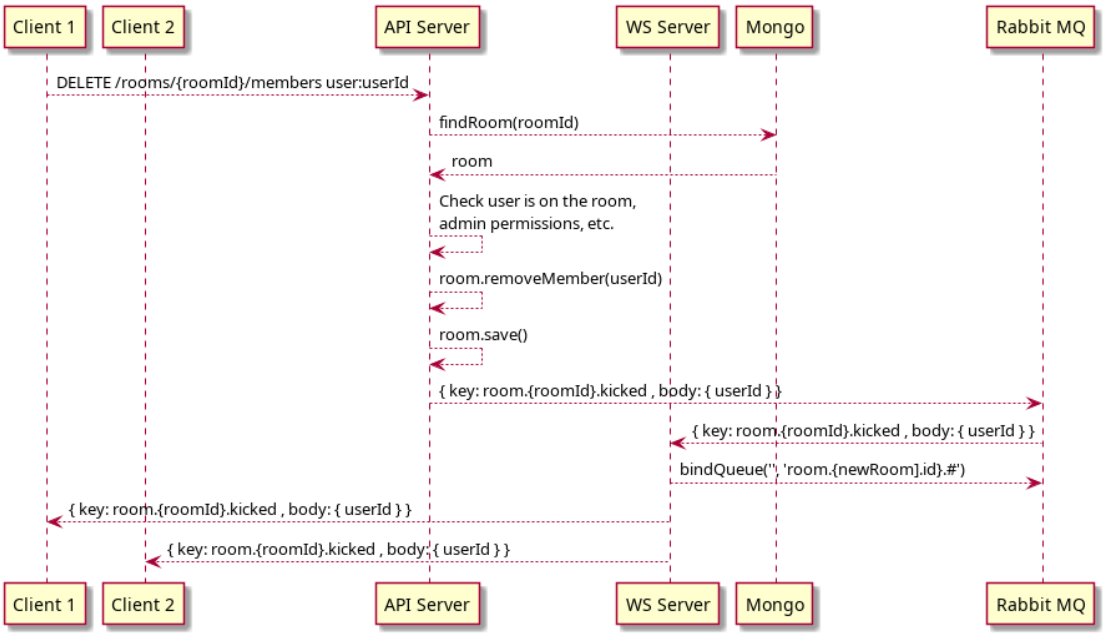


Figure 4: Kick User

- Servidor de Trendings
- Broker de mensajes RabbitMQ
- Base de Datos MongoDB

Tanto el broker como la base de datos han sido "contratados" como servicio. Esto implica, por un lado, delegar los problemas de administración y despliegue mientras, por otro lado, incrementamos el coste o sacrificamos prestaciones. Por último, también hay que considerar los posibles problemas de latencia si los nodos de persistencia se despliegan lejos de los nodos de la aplicación.

Los nodos de aplicación, sin embargo, se despliegan en el proveedor cloud Heroku. Al igual que sucedía con los nodos de persistencia, implica una relación coste-rendimiento peor que con unos servidores autogestionados con Kubernetes pero permite acceder a un tier gratuito que cumple los requisitos mínimos del proyecto. Sin embargo, obliga a sumir varios tradeoffs respecto a Kubernetes:

- No hay soporte para HTTP2
- No hay soporte nativo para configuraciones centralizadas
- No hay soporte nativo para descubrimiento de dinámico
- No hay opción de enrutado virtual interno: todas las conexiones se hacen vía red exterior

Cabe destacar que, aunque no se hayan estacado en el diagrama, todas las conexiones entre nodos se hacen vía conexiones seguras: HTTPS y WSS entre cliente y nodos de

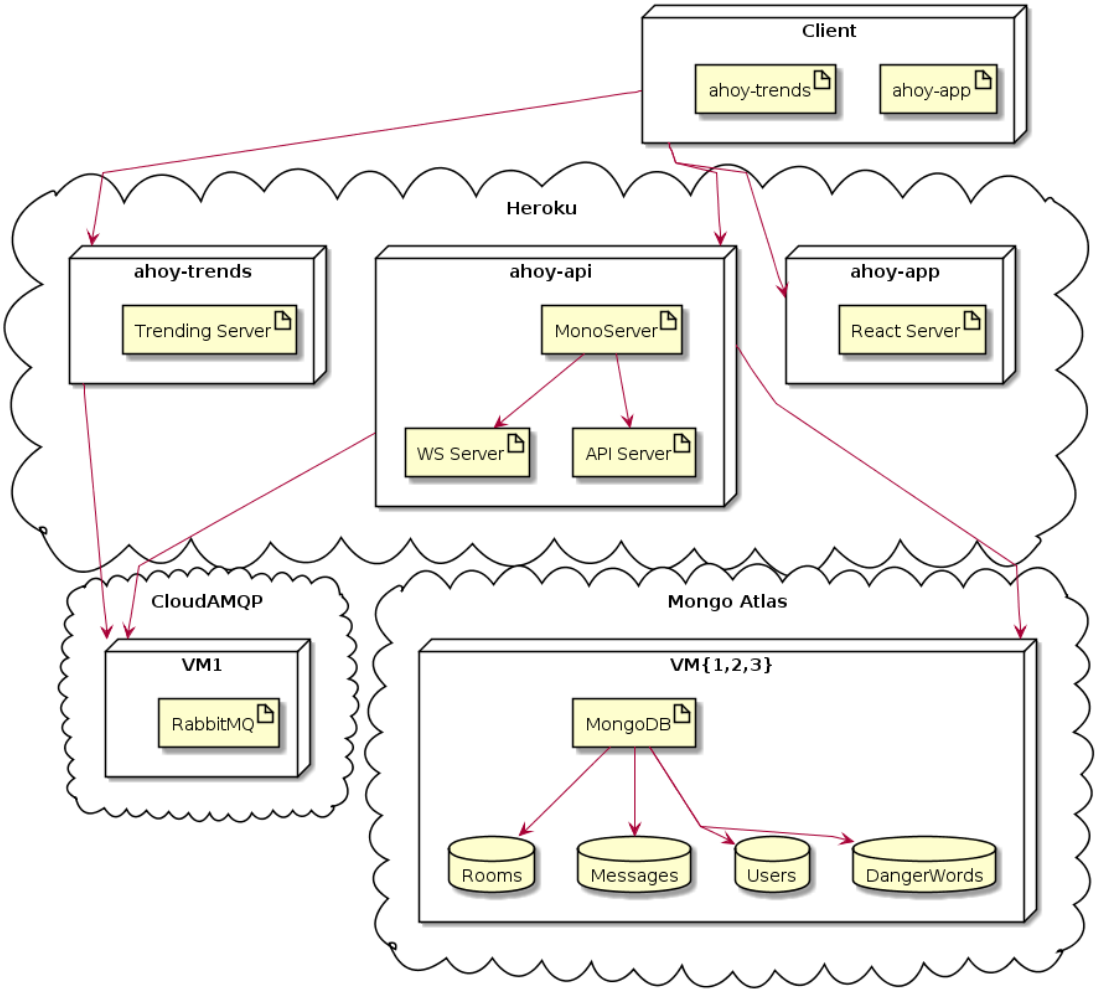


Figure 5: Componentes y Conectores - API Server

aplicación y AMQPS y Mongo+SSL entre nodos de aplicación y nodos de persistencia.

Por último, cabe destacar que el nodo del Servidor de API está desplegando conjuntamente el servidor HTTP y el servidor Web Socket. Esto se ha decidido hacer así por simplicidad en el despliegue pero será la última vez que los veamos juntos en un diagrama ya que son dos servicios autónomos.

## Vista de Componentes y Conectores

La vista de componentes y conectores nos muestra los componentes en ejecución y sus relaciones de uso entre ellos. Veremos primero la vista del Servidor de API ya que el Servidor de Trends comparte la mayoría de decisiones arquitecturales.

### API Server

Siguiendo el diagrama de arriba a abajo, podemos ver cómo la aplicación web desarrollada en React utiliza principalmente dos APIs: la Api del navegador no relacionada con el DOM (concretamente, la función fetch) y la API de React.

Continuando hacia abajo, React es el encargado de usar la API del DOM que ha sido deliberadamente separada para aclarar que la aplicación en React no tiene acceso ni ha de preocuparse por el acceso al DOM (una de las grandes ventajas de React).

Como en toda aplicación web, el navegador es el encargado de comunicarse con los servidores. En este caso, mediante la API HTTP y la API de eventos Web Socket. Cabe especificar que la API de Eventos no ha sido detallada en Open API al no adecuarse bien al paradigma. No obstante será especificada en el apartado de Interfaces.

El Servidor de API REST, a la izquierda, muestra cómo Espress expone la API REST y coordina los dos elementos importantes, los middlewares (llamado así en el ecosistema Express) y los Controllers (similares a los controladores del patrón MVC).

Los middlewares no son sino funciones de filtrado que toman un objeto request, un objeto response y un callback pasa pasar el control al siguiente filtro. Algunos middlewares de ejemplo pueden ser el del autentificación, que verifique el JWT; el de subida de ficheros, que intercepte las peticiones multipart; o el de censura, que intercepta las respuestas salientes que contienen mensajes y enmascara las palabras peligrosas.

Los controllers encapsulan las acciones de la API REST en una relación 1:1 comunicándose con el tier de persistencia a través de las APIs de los objetos de modelo y realizando las consultas o mutaciones pertinentes así como la consecuente emisión de eventos a tarvés de la API del sistema de emisión de eventos. Un requisito extra sería la posibilidad de hacer estas acciones de forma transaccional. No obstante, no se ha abordado esa funcionalidad a lo existir una acción crítica que dependiera de esa propiedad. CyC El Servidor de Eventos, a la derecha, muestra cómo el servidor Web Socket de Node utiliza las APIS

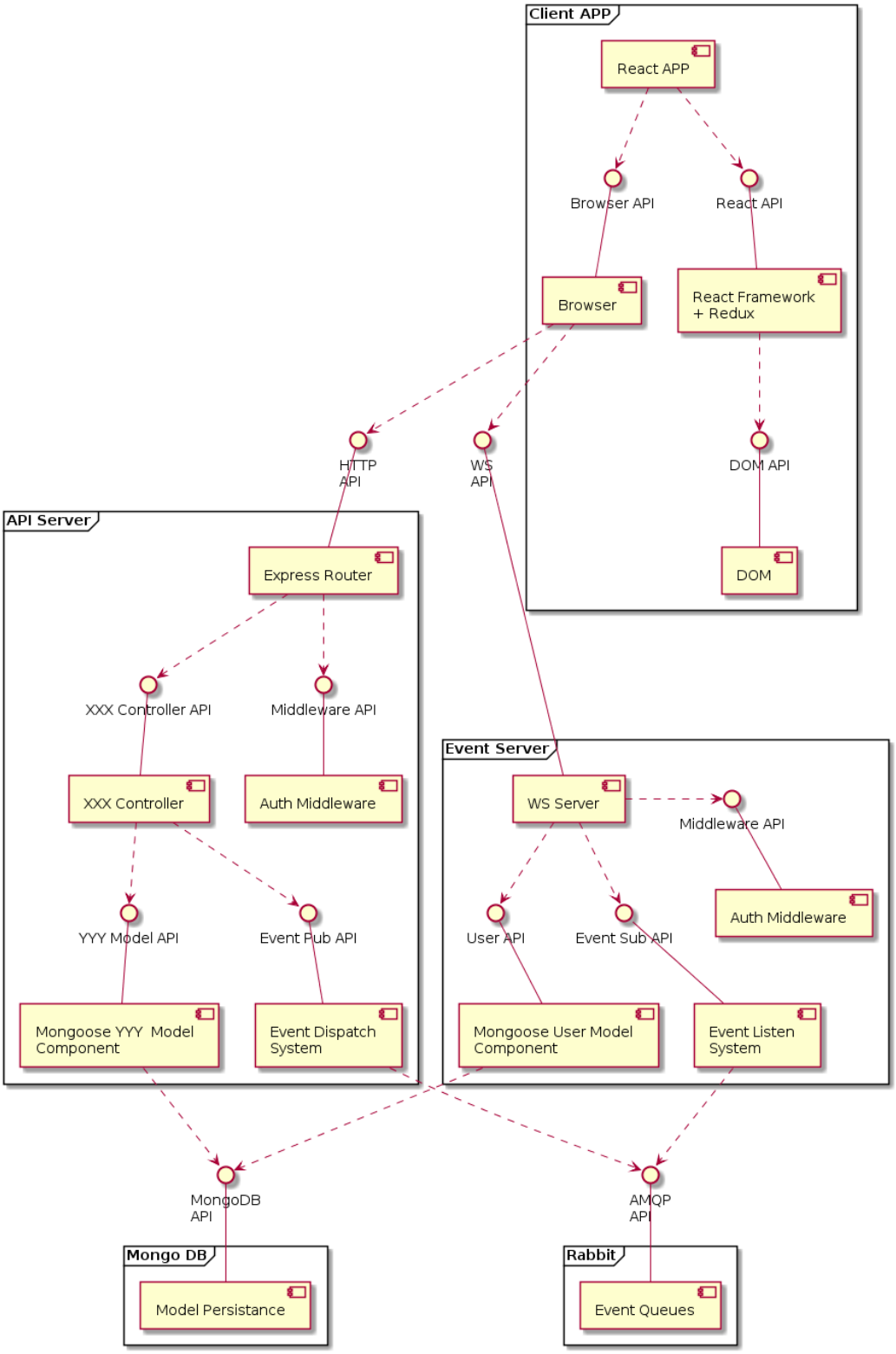
Client APP
React APP
Browser API
React API
Browser
React Framework
+ Redux
HTTP
API
WS
API
DOM API
DOM
API Server
Express Router
XXX Controller API
Middleware API
XXX Controller
Auth Middleware
YYY Model API
Event Pub API
Mongoose YYY Model
Component
Event Dispatch
System
Event Server
WS Server
Middleware API
Auth Middleware
User API
Event Sub API
Mongoose User Model
Component
Event Listen
System
MongoDB
API
AMQP
API
Mongo DB
Model Persistance
Rabbit
Event Queues

Figure 6: CyC

de tres componentes: El middleware de autentificación, que es comparte la lógica con el del Servidor de API REST a través de un adaptador, pero es un componente distinto en ejecución; el objeto del modelo que representa a los usuarios y se comunica con la capa de persistencia; y el sistema de escucha de eventos.

#### API Server

Las diferencias más significativas con el diagrama anterior son la eliminación de una aplicación cliente, la eliminación del servidor de eventos externo y la eliminación de los objetos del modelo. Las adiciones son, un Store en memoria en el lugar del modelo, un controlador nuevo para leer el estado de ese modelo y un sistema de escucha de eventos conectado a RabbitMQ que modifica el Store en base a los eventos recibidos: contando las palabras de los mensajes de texto.

### Packetes

El diagrama de paquetes muestra la distribución del código así como sus dependencias de uso. Para este proyecto, la arquitectura usada ha sido la tradicional de 3 niveles sin inversión de dependencias. Esto es, una capa de aplicación que orquesta a las capas inferiores, una capa de modelo que encapsula el modelo de negocio y una capa de infraestructura que encapsula las comunicaciones con sistemas externos. En contraposición a la arquitectura hexagonal, aquí la capa de modelo está completamente atada a la infraestructura por lo que un cambio en los sistemas subyacentes impactaría en la misma. Esta concesión se ha hecho deliberadamente considerando el alcance del proyecto, su complejidad y su proyección de vida útil.

### Entreprise Integration Patterns

Los patrones escogidos para el paso de mensajes son:

- Canal de tipo Publicación-Suscripción
- Mensajes de tipo Evento
- Enrutado Basado en el Contenido (concretamente, por tópico)

No se ha implementado un patrón Lista de Receptores porque eso implicaría que es el mensaje el que debe contener la información de sus destinatarios. En su lugar, son los destinatarios los que toman la responsabilidad de suscribirse a los tópicos que su lógica les ordene.

El Servidor de Eventos no es considerado un gateway de mensajería porque no sólo traduce los eventos, sino que realiza operaciones activas de suscripción y desuscripción a los tópicos.

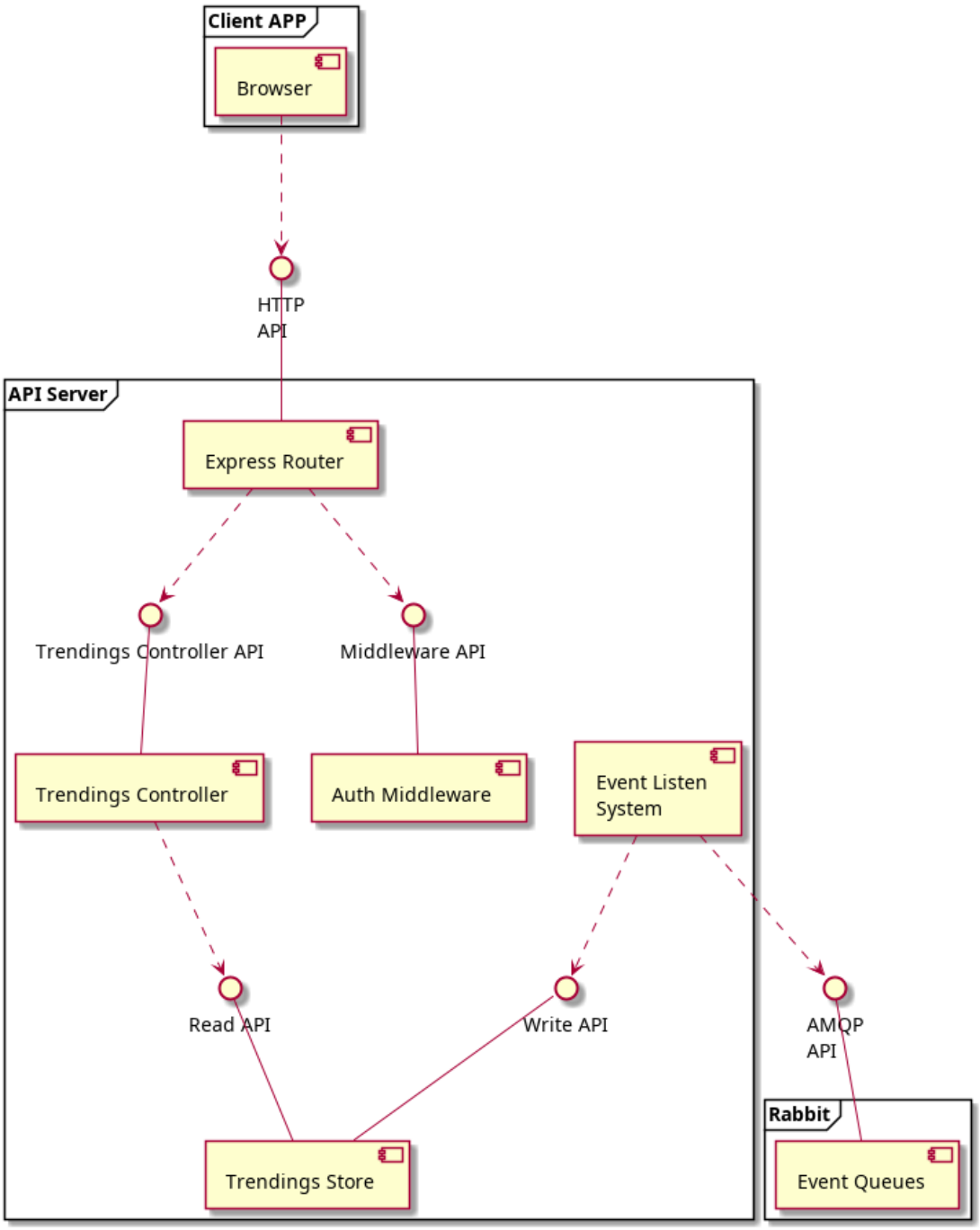


Figure 7: Componentes y Conectores - Métricas

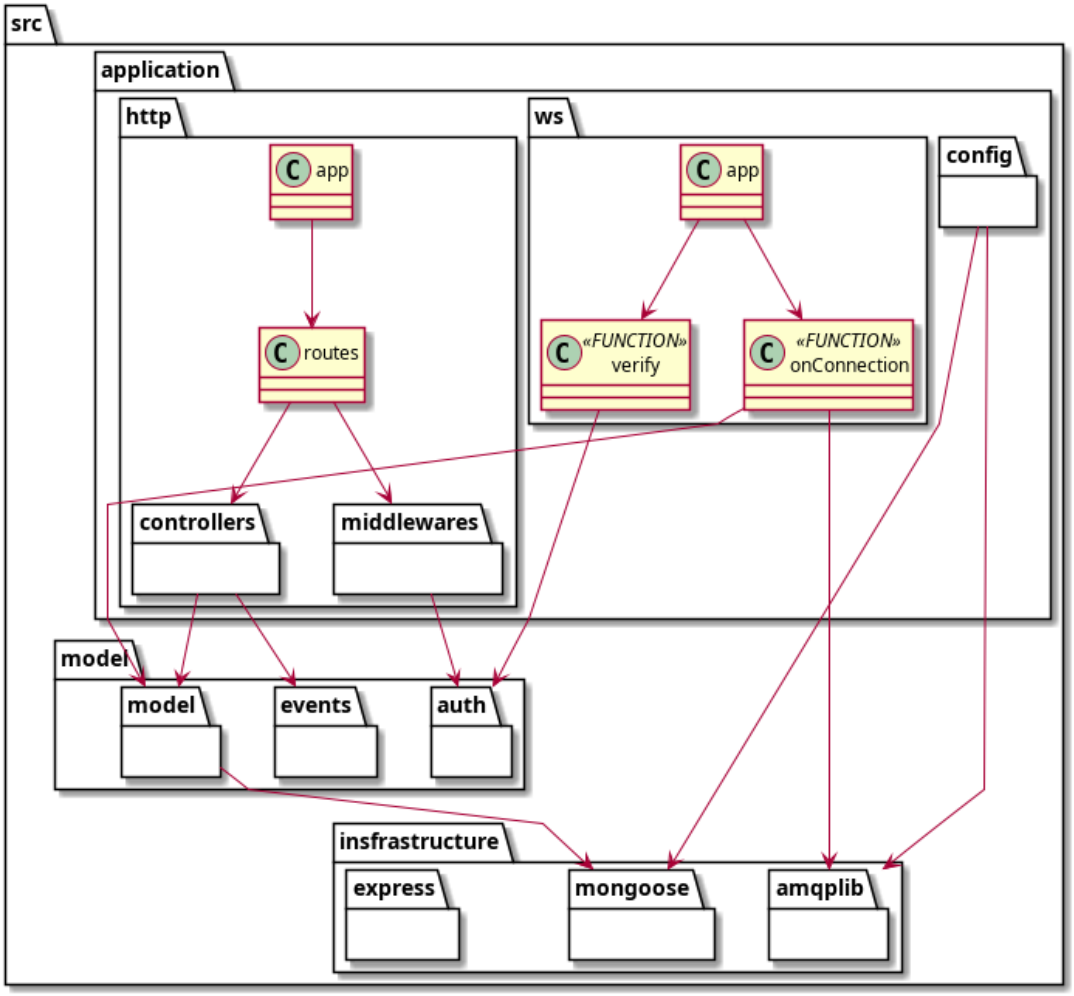


Figure 8: Diagrama de paquetes

## Interfaces del Sistema

Algunas de las interfaces presentadas a continuación ya han sido introducidas en el apartado de Arquitectura, no obstante, se recopilan aquí de nuevo para una mejor comprensión.

### API REST

La API REST está especificada en OpenAPI 3.0 en la siguiente dirección.

La única consideración a hacer al respecto una violación del diseño del verbo DELETE, en la cual hay un uso no permitido de un cuerpo de mensaje (body). Esto se debe a que la API fue diseñada antes de aprender el uso de la herramienta Open API. De haberse conocido esta antes y haberse usado para el diseño, el resultado habría sido, sin duda, una API mejor y más coherente con los principios REST.

### Middlewares

Los middlewares, como ya se ha comentado, siguen un patrón impuesto por el framework Express donde toman un objeto request y uno response así como una función para devolver el control al framework.

Un ejemplo de un middleware (simplificado) sería.

```
function auth(req, res, next) {
  if (checkUser(req.body.user)) {
    req.user = getUser(req.body.user);
    next();
  } else {
    res.status(403);
  }
}
```

Es por esto que no tiene sentido especificar la API de cada uno al compartir todos el método de llamada. En su lugar, cabe destacar las mutaciones que hacen a esos objetos. Los middlewares más interesantes son:

- Morgan: Un logger que registra cada petición HTTP
- CORS: Gestiona las cabeceras de seguridad para Cross Origin access
- BodyParser: Parsea los parámetros de la query y del el body en objetos JSON accesibles desde `req.body`
- censoreMessages + mung: Un middleware especial que actúa sobre la respuesta del Controller, en este caso, para censurar las palabras peligrosas.

- multer: intercepta mensajes de tipo `multipart/form-data` y crea un buffer de lectura para consumir el stream de bytes.
- dispatch: crea una función `dispatch(event)` que: crea un canal AMQP, envía el mensaje/evento `event` y cierra la conexión. Al hacer disponible esta función desde `req.dispatch`, los controller pueden emitir mensajes sin tener que conocer los detalles de la conexión con AMQP.
- verifyHTTPRequest: extrae el token de la URL o de las cabeceras HTTP y verifica su validez. Si es válido, escribe la información del usuario en `req.userId` y `req.admin`.

### Controller API

Las funciones de los controladores tienen un mapeo 1:1 con las funciones de la API REST. Cada endpoint representa una acción o caso de uso y tiene disponible una función en un controlador. De modo similar a cómo funcionan los middlewares en Express, estas funciones han de cumplir también con unas restricciones en cuanto a los parámetros que reciben.

```
function hello(req, res) {
  res.send(`Hello ${req.body.name}`);
}
```

Lo realmente interesante es conocer los atributos que utilizan y estos son los atributos disponibles en el body/query o en la URI (extraídos mediante patrones del tipo `/rooms/{roomId}`). Por ello, se considera que la especificación de OpenAPI describe las funciones de los Controladores mejor que su propia API y por ello se considera redundante profundizar en ellas más allá de un breve listado.

| Método | Ruta | Función |
|---|---|---|
| POST | /login: | AuthController.login |
| GET | /oauth/redirect: | AuthController.oauth_redirect |
| POST | /rooms: | RoomController.newGroupRoom |
| POST | /rooms/duo: | RoomController.newDuoRoom |
| GET | /rooms: | RoomController.getAllRooms |
| GET | /room/:roomId: | RoomController.getRoom |
| DELETE | /room/:roomId: | RoomController.deleteRoom |
| PUT | /room/:roomId/members: | RoomController.inviteUser |
| DELETE | /room/:roomId/members: | RoomController.kickoutUser |
| GET | /room/ahoy/messages: | MessageController.getBroadcastMessages |
| POST | /room/ahoy/messages: | MessageController.broadcastMessage |
| GET | /room/:roomId/messages: | MessageController.getMessages |
| GET | /room/:roomId/file/:fileId: | MessageController.getFile |
| POST | /room/:roomId/messages: | MessageController.postMessage |
| GET | /user: | UserController.getLoggedUser |

| Método | Ruta | Función |
|---|---|---|
| GET | /user/:userId: | UserController.getUser |

## Objetos del Modelo

Los objetos del modelo son implementados a través de el decorador de Mongoose.model (librería de mapeo objeto-documental). Por ello, todos los objetos instanciados comparten funciones genéricas como `.save()`. Además, la propia clase del objeto depresentado adquiere funciones de búsqueda típicas de un repositorio como `findById` o un `find()` genérico que acepta una query JSON para MongoDB. La verdadera "lógica" en estos objetos radica en su constructor y sus funciones de validación, que garantizan que el objeto creado o transformado es correcto. Muchas de estas funciones de validación, sin embargo son definidas declarativamente y es Mongoose quien las implementa.

Además, en el caso de las salas de chat, al poder ser de diversos tipos, se ha abstraído su construcción a funciones factoría que replican la estructura de los constructores, que requiriendo menos parámetros (al decidir estas el resto de ellos).

Las clases y los constructores/factorías resultantes son:

### Message

Los objetos de tipo Mensaje representan los mensajes enviados entre los Usuarios del sistema y pertenecen a una Sala concreta.

Función factoría: `createMessage({ from, to, content, type })`

- from: Usuario emisor.
- to: Sala destinataria.
- content: Contenido del mensaje (textual o el id del Attachment).
- type: Tipo del mensaje (text o file).

### Room

Los objetos de tipo Sala representan espacios lógicos a los que pertenecen los Usuarios y donde se envían Mensajes.

Función factoría de Sala Duo: `createDuoRoom({ members = [] })`

- members: Los dos Usuarios que formarán la Sala Duo. No importa el orden.

Función factoría Sala de Grupo: `createGroupRoom({ admin, name })`

- admin: el Usuario que ha creado la Sala
- name: el nombre de la Sala

### User

Los objetos de tipo Usuario representan a los usuarios que utilizan la aplicación de chat y permiten calcular sus permisos de acceso.

Constructor: `constructor({_id, name, role})`

- _id: el ID único del Usuario. En login con GitHub se usará el correo electrónico primario.
- name: el nombre del usuario. En login con GitHub se usará el nick.
- role: el Usuario rol del usuario (user o admin)

### Attachment

Los objetos de tipo Attachment representan los ficheros adjuntos en mensajes de tipo file.

Constructor: `constructor(metadata)`

- metadata: Contiene información como el nombre del fichero o el ID de la Sala a la que pertenece el Mensaje que lo contiene.

Método: `write(buffer)`

- buffer: El buffer de bytes que va a ser escrito en la base de datos

Método: `read(): buffer`

- buffer: El buffer de bytes del que poder leer de la base de datos

## Eventos

El sistema de eventos está inspirado en los patrones propuestos por la comunidad de Redux. En ésta, los eventos (llamados acciones) son objetos JSON planos con una estructura concreta. Además, una de las buenas prácticas es encapsular su creación en métodos parametrizados para garantizar que estén siempre bien formados.

El resultado de esto es un paquete de funciones que devuelven objetos evento, no los emiten. Para su emisión, se ha de tener acceso a un método `dispatch` previamente configurado (por ejemplo, en un middleware). De este modo, para emitir un evento que informe de que se ha creado un usuario habría que invocar: `req.dispatch(user_created(userId))`.

El listado de funciones creadoras de eventos es:

| Function | Event |
|---|---|
| user_authenticated(user) | `{ key:"auth.user_authenticated", body: user }` |
| user_created(user) | `{ key:"auth.user_created", body: user }` |

| Function | Event |
|---|---|
| new_message(message) | `{ key:"room.\${message.to}.new_message", body: message }` |
| room_deleted(roomId) | `{ key:"room.\${roomId}.deleted", body: { roomId } }` |
| room_created(room) | `{ key:"room.\${room.id}.created", body: room }` |
| user_invited((userId, room)) | `{ key:"user.\${userId}.invited", body: room }` |
| user_kicked((userId, room)) | `{ key:"user.\${userId}.kicked", body: room }` |

## Criterios de Aceptación

El objetivo de este proyecto era cumplir todos los requisitos que fueran compatibles propuestos por el Product Owner. Estos requisitos son, la combinación del desglose de las historias de usuario con los requisitos extra sobre la propia plataforma y su documentación.

Para corroborar el grado de cumplimiento de los requisitos, se han definido unas pruebas de aceptación manuales que cubren todos los requisitos de las historias de usuario.

### Pruebas de aceptación

#### Usuario

#### Log In con GitHub

##### U-1

- **Login** with GitHub with `you account`
- Expect the login to be successful

#### Sala

#### Base

##### S-B-1

- **Create** a `Duo Room` with the user `mike`

- **Fetch** the `Room information`
- Expect the `Room` to **be created**
- Expect the `Room` to **have admins** `you account` and `mike`

#### S-B-2

- **Create** a `Group Room` with a the title `San Pepe`
- **Fetch** the `Room information`
- Expect the `Room` to **be created**
- Expect the `Room` to **have admin** `you account`
- Expect the `Room` to **have** `0` **members**

#### S-B-3

- **Invite** other `mike`, `tom`, `hannah` and `lisa` to `San Pepe`
- **Fetch** the `Room information`
- Expect the `Room` to **be created**
- Expect the `Room` to **have admin** `you account`
- Expect the `Room` to **have members** `mike`, `tom`, `hannah` and `lisa`

#### S-B-4

- **Copy** the `URL` of the `Room`
- **Open** a new `Browser Tab`
- **Paste** the `URL` of the `Room`
- Expect the `Browser` to open in the `Room`

### Administrador de la sala

#### S-A-1

- **Login** with `lisa` on a `separate context`
- **Kick-out** `lisa` from `San Pepe` with `your account`
- Expect `lisa` to **be removed from the room on push**
- **Send** a `Text Message` with `your account` to `San Pepe`
- Expect `lisa` to **not receive it**

#### S-A-2

- **Login** with `lisa` on a `separate context`
- **Invite** `lisa` to `San Pepe` with `your account`
- Expect `lisa` to **be added** to the room on push

- **Send** a `Text Message` with `your account` to `San Pepe`
- Expect `lisa` to **receive it**

#### S-A-3

- **Login** with `lisa` on a `separate context`
- **Try to kick-out** `tom` from `San Pepe` with `lisa`
- Expect `lisa` to **be rejected**
- Expect `tom` to **be still in the room**

### Mensaje

### Texto

#### M-T-1

- **Send** a `Text Message 500 chars long` to `mike`
- Then **login** with `mike`
- Expect the `Message` to **arrive even when he was disconnected**
- Expect the `Message` to **arrive in less than 60 seconds**

#### M-T-2

- **Login** with `tom` on a `separate context`
- **Send** a `Text Message 500 chars long` with `your account` to `San Pepe`
- Expect the `Message` to **arrive on push** to `tom`
- Expect the `Message` to **arrive in less than 60 seconds**

### Attachment

#### M-A-1

- **Send** an `Attachment Message` with a `1MB file` to `mike`
- Then **login** with `mike`
- Expect the `Message` to **arrive** even **when he was disconnected**
- Expect the `Message` to **arrive in less than 60 seconds**
- **Click** the link in the `Message`
- Expect the `Browser` to **download** the `file`

#### M-A-2

- **Login** with `tom` on a `separate context`
- **Send** an `Attachment Message` with a `1MB file` with `your account` to `San Pepe`
- Expect the `Message` to **arrive on push** to `tom`
- Expect the `Message` to **arrive in less than 60 seconds**
- **Click** the link in the `Message`
- Expect the `Browser` to **download** the `file`

### Broadcast

#### M-B-1

- **Login** with `admin` on a `separate context`
- **Send** a `Text Message` with `admin` to `ahoy`

### Censura

#### M-C-1

- **Send** a `Text Message` to `San Pepe` saying `What do you think about blockchain?`
- **Add** `blockchain` to `Danger Word` list
- **Send** a `Text Message` to `San Pepe` saying `I think that Blockchain is the future`
- Expect the `Message` to **arrive censored**
- **Check** the `Process Logs`
- Expect the `Process Logs` to **report the issue**

### Trending Topics

#### M-X-1

- **Access** to `Ahoy Trends`
- Expect `top trends` to **appear sorted by popularity**

## Historias de Usuario

**Como usuario Quiero poder enviar mensajes a otros usuarios Para tener conversaciones puntuales**

| Requisito | Cubierto por |
|---|---|
| 500 chars | M-T-1 |
| Punto a punto sabiendo el id del otro usuario | S-B-1 |
| Recibir mensajes cuando no estabas conectado | M-T-1 |
| Recepción on push | M-T-2 |
| Compartir mensajes entre al menos 5 usuarios | S-B-3 y S-A-2 |
| Menos de 60 segundos de latencia | Transversal |

**Como usuaria Quiero poder compartir ficheros de mi equipo con otros usuarios Para complementar las conversaciones con información adicional**

| Requisito | Cubierto por |
|---|---|
| Ficheros de al menos 1Mb | M-A-1 |

**Como usuario Quiero poder crear salas de chat permanentes Para tener un registro de las conversaciones pasadas**

| Requisito | Cubierto por |
|---|---|
| Sólo el admin puede invitar y echar | S-A-1 y S-A-2 |
| Sólo los usuarios invitados pueden leerlos mensajes de la sala | S-B-3 |
| Acceder a salas por su URL única | S-B-4 |
| Mensajes persistentes | M-T-1 |

**Como superusuario Quiero poder enviar mensajes a todos los usuarios suscritos Para anuncios de interés y publicidad de nuevas características del sistema**

| Requisito | Cubierto por |
|---|---|
| Broadcast de mensajes de texto | M-B-1 |

**Como superusuaria Quiero poder censurar algunos mensajes Para cumplir con los requisitos legales exigidos en regímenes represivos con las libertades civiles pero con mercados económicamente rentables**

| Requisito | Cubierto por |
|---|---|
| Censura de las palabras peligrosas configuradas | M-C-1 |

| Requisito | Cubierto por |
|---|---|
| Registro de cada mensaje censurado incluyendo contenido, emisor y sala destinataria | M-C-1 |
| Reconfigurar las palabras peligrosas afecta en tiempo real al envío de mensajes | M-C-1 |

**Como superusuario Quiero saber en tiempo real de qué hablan los usuarios Para poder calcular trending topics etc. y vender esa información al mejor postor**

| Requisito | Cubierto por |
|---|---|
| Lista de trends generada en tiempo real | M-X-1 |
| Actualización en tiempo real de la aplicación cliente | No implementada |

### Requisitos extra

| Requisito | Cubierto por |
|---|---|
| Varios usuarios concurrentes. | S-A-2 y M-A-2 |
| Despliegue en, al menos, 5 máquinas. Las 5 en proveedores cloud. | Diagrama de despliegue |
| API REST documentada en OpenAPI. | Enlazado anteriormente |
| Configuración centralizada del sistema. | Dashboard de Heroku |
| Uso de HTTP2 sobre TLS. | HTTP2 no soportado por Heroku |
| Autentificación OAuth con GitHub. | U-1 |
| Documentar decisiones arquitecturales: Vista de CyC y Despliegue. | Mostrado anteriormente |
| Documentar patrones EIP. | Mostrado anteriormente |
| Diagramas de comportamiento. | Mostrado anteriormente |
| Costes operativos del sistema en proveedores cloud. | Mostrado a continuación |

## Costes operativos en distintos proveedores cloud

Estimar los costes operativos de un sistema así es, de entrada, imposible debido a que, al no ser una aplicación real con clientes reales, somos incapaces de estimar el volumen de tráfico o usuarios concurrentes. Además, los requerimientos del Product Owner son tan laxos (latencias de 60 segundos; ficheros de 1 MB) que prácticamente, cualquier sistema, por pequeño que sea, cubre las necesidades.

Asumiendo entonces:

- Un volumen de usuarios concurrentes bajo que no obliga a replicar nodos
- Un volumen de usuarios concurrentes bajo que no requiere máquina de altas capacidades
- Un volumen de datos saliente de menor a 5GB al mes (descargar 5000 veces ficheros de tamaño máximo)

Podemos asumir que el servicio entero puede estar distribuido en dos máquinas virtuales, una de ellas con persistencia en disco. De ese modo, reemplazaríamos en el diagrama de despliegue a Heroku por la primera máquina y a CloudAMQ y Mongo Atlas por la segunda.

Evidentemente, existiría siempre la opción de mantener los servicios en la distribución actual pagando un coste cero y ajustándose a las restricciones de los tres proveedores contratados.

## OVH

Esta primera solución es clave por su sencillez y por una cualidad específica del proveedor: No el tráfico de datos. Una empresa que conserva los modelos de hosting tradicionales donde pagas por la máquina que eliges, los discos adicionales y nada más.

### VPS

Contratando dos instancia de su VPS SSD 3 obtenemos:

- 2vCores (importante destacar que cada proceso de Node sólo consume un thread)
- 2GHz
- 8GB RAM
- 80GB SSD
- 100mbps de tráfico
- SLA del 99.95%
- Protección DDoS
- Una IP propia
- Monitoreo detallado del sistema desde el panel de cliente

Los costes serían de:

$$12eur/mes * 2vm = 24eur/mes$$

La desventaja es que la administración e instalación habrían de ser manuales como en cualquier máquina virtual.

### Dedicated Hosting + Kubernetes

Si la opción de los VPS no fuera suficientemente potente, o no nos gustara la administración manual, podríamos subir de nivel a un sistema de instancias cloud que, además de tener mejores prestaciones, permiten aprovechar el servicio de Kubernetes gratuito que ofrece OVH.

Contratando dos instancias B2-7 obtenemos:

- 2vCores (importante destacar que cada proceso de Node sólo consume un thread)
- 2.3GHz
- 7GB RAM
- 50GB SSD
- 250mbps de tráfico
- Una IP propia + posibilidad de comprar una ip-failover con pago único de $2eur$
- Cluster Kubernetes privado con todo lo que K8S ofrece: Configuración centralizada, descubrimiento automático, enrutado interno, etc.
- Posibilidad de escalar horizontalmente añadiendo un balanceador de carga por $20eur/mes$

Los costes serían de:

$$22eur/mes * 2inst = 44eur/mes$$

### Azure Kubernetes

Si buscamos un despliegue de similares prestaciones en Azure, descubriremos que también ofrece servicio gratuito de Kubernetes donde sólo hay que pagar por la infraestructura subyacente.

Contratando dos maquinas D2MS v3 + un disco de sistema SSD E4 obtenemos:

- 2vCores (importante destacar que cada proceso de Node sólo consume un thread)
- 8GB RAM
- 50GB SSD temporal
- Ancho de banda no especificado. Tráfico gratuito hasta 5GB al mes (descargar 5000 veces ficheros de tamaño máximo)
- Cluster Kubernetes privado con todo lo que K8S ofrece: Configuración centralizada, descubrimiento automático, enrutado interno, etc.
- Balanceador de carga gratuito

Los costes serían de:

$$53eur/mes * 2inst + 3eur/mes = 109eur/mes$$