DOCUMENTACIÓN DEL PROYECTO INTERMODULAR

Jon Cantero, Ariel Carrera, Dagner Ramos

# ÍNDICE

[ÍNDICE 1](#_Toc34667066)

[Identificación del proyecto 2](#_Toc34667067)

[1. Activar el equipo 2](#_Toc34667068)

[2. Plantear e identificar el reto 2](#_Toc34667069)

[Preparación del proyecto 2](#_Toc34667070)

[1. Aplicación Android 2](#_Toc34667071)

[2. Repositorio 2](#_Toc34667072)

[3. Planificación 2](#_Toc34667073)

[4. Diagrama de clases 2](#_Toc34667074)

[Puesta en marcha 3](#_Toc34667075)

[1. Base de datos 3](#_Toc34667076)

[2. Servidor 3](#_Toc34667077)

[3. Aplicación Android 6](#_Toc34667078)

[4. Multilenguaje 7](#_Toc34667079)

[Conclusión 7](#_Toc34667080)

[1. Feed-back 7](#_Toc34667081)

# Identificación del proyecto

## Activar el equipo

El primer paso del proyecto fue firmar el contrato del proyecto por todos los integrantes

del grupo.

## Plantear e identificar el reto

En este paso, los miembros del equipo tuvimos que decidir a ver quién impartía cada uno de los siguientes roles:

* Moderador: Jon
* Secretario: Ariel
* Controlador del tiempo: Dagner

# Preparación del proyecto

## Aplicación Android

Tuvimos que decidir entre las diferentes aplicaciones de los integrantes. Finalmente nos quedamos con la aplicación de Ariel por ser la más completa.

El diseño por otra parte se adapta tanto a teléfonos móviles como tablets.

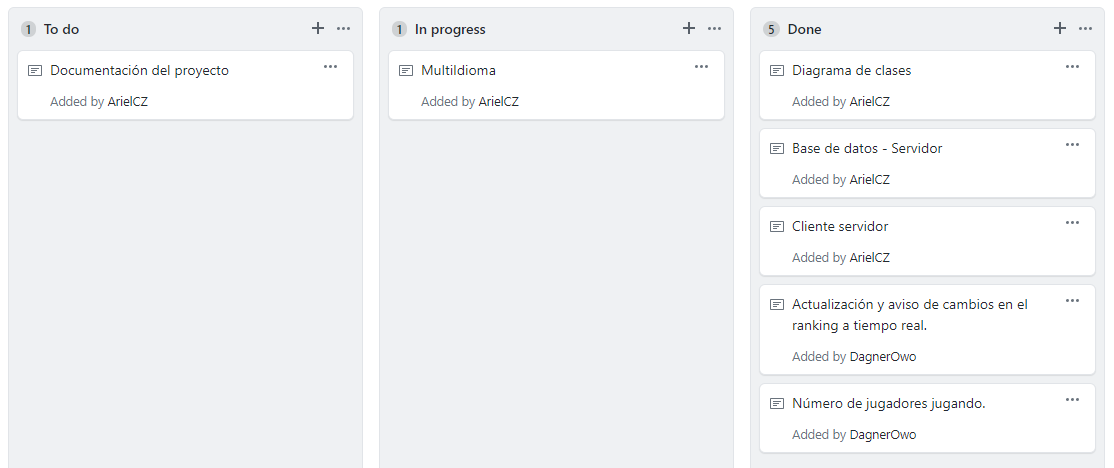
## Repositorio

Después de decidir la aplicación íbamos a usar, subimos el proyecto a un repositorio en GitHub, para que todos los miembros del equipo pudiéramos acceder a él.

<https://github.com/ArielCZ/ProyectoHundirLaFlota>

## Planificación

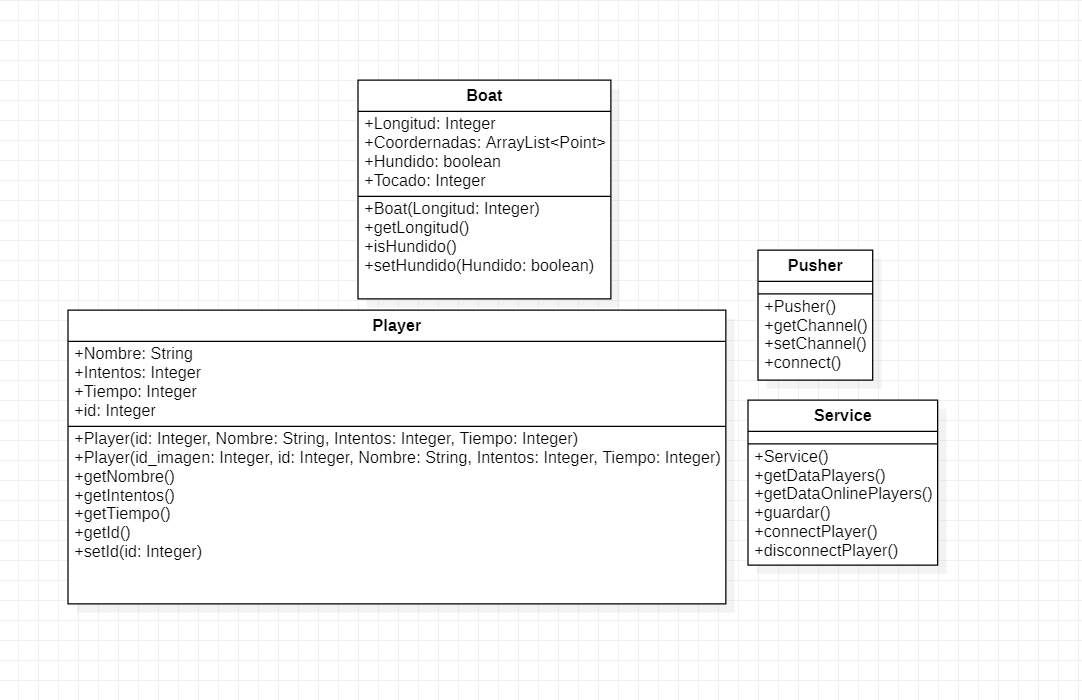
Generáramos el tablero Kanban con toda la planificación de las tareas.



## Diagrama de clases

Hemos desarrollado el diagrama de clases de la aplicación, representando las siguientes clases:

* Boat: longitud, coordenadas, hundido, tocado
* Player: nombre, intentos, tiempo, id
* Pusher: getChannel(), setChannel(), Conncet()
* Service: getDataPlayers(), getDataOnlinePlayers(), guardar(), connectPlayer(), disconnetPlayer()



# Puesta en marcha

## Base de datos

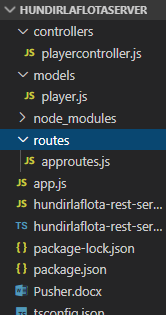
En un principio, estuvimos debatiendo sobre qué tipo de base de base de datos implementar, y finalmente decidimos usar MongoDB porque es el que más dominábamos.

En nuestra base de datos vamos a almacenar la clase “Player”, que almacena los datos del jugador (El nombre, intentos, tiempo…), para poder almacenar las puntuaciones.

## Servidor

Creamos el servidor siguiendo las instrucciones del pdf de myStore – Web Server.

La estructura de carpetas que hemos planteado es la siguiente:

En la carpeta controllers tenemos todos los métodos que interactúan con la base de datos.

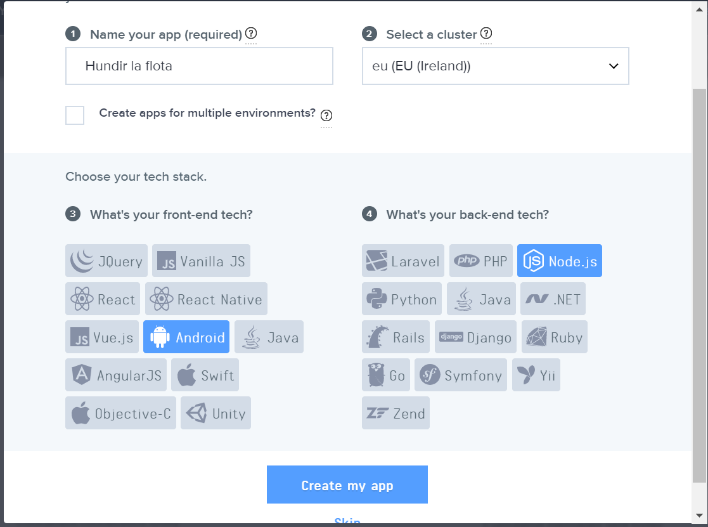
La carpeta models tiene definidos los esquemas necesarios utilizados por el controlador.

En cuanto a las dependencias hemos instalado Pusher, Mongoose, Express y BodyParser los cuales están contenidas en la carpeta node\_modules.

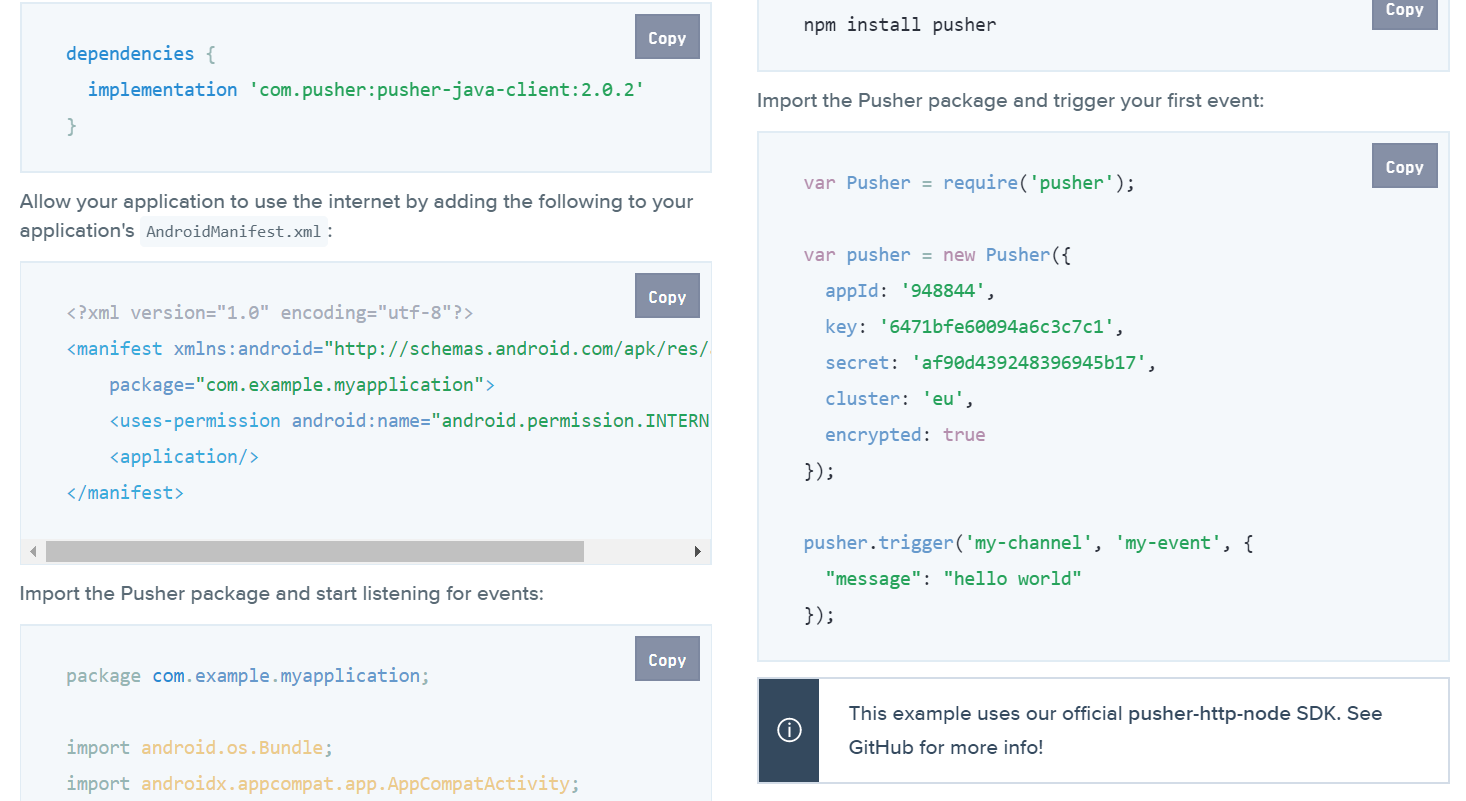
En la carpeta routes están definidas las rutas que hacen uso de los métodos del controlador.

Después, buscamos una forma de sincronizar los datos en tiempo real, y decidimos quedarnos con Pusher, una API bidireccional.

Para ello es necesario registrarse en la página de Pusher y crear un nuevo proyecto.



Al crear el proyecto necesitaremos añadir las dependencias, editar el fichero Manifest.xml, configurar Pusher y crear los métodos necesarios. Toda esta información te la proporciona Pusher al crear el Proyecto.



Para hacer uso de Pusher necesitamos implementar los siguientes métodos, los cuales crean una conexión mediante un canal y recibe los datos o consecuentes errores que puedan haber.





Para hacer uso de Pusher en el lado del servidor primero:

* Tenemos que instanciar el Pusher con los parámetros de configuración del proyecto creado.

var Pusher = require('pusher');

var onlinePlayers = [];

var pusher = new Pusher({

    appId: '948844',

    key: '6471bfe60094a6c3c7c1',

    secret: 'af90d439248396945b17',

    cluster: 'eu',

    encrypted: true

});

* Después notificar a la aplicación de que se ejecute un método haciendo uso del canal creado y un evento.

        jugador.save((err, playerStored)=>{

            if (err) return res.status(500).send({

                message: 'Error al guardar'

            });

            if (!playerStored) return res.status(404).send({

                message: 'No se ha podido guardar'

            })

            pusher.trigger('playerschannel', 'player-save', {

                player:playerStored

            });

            return res.status(200).send({

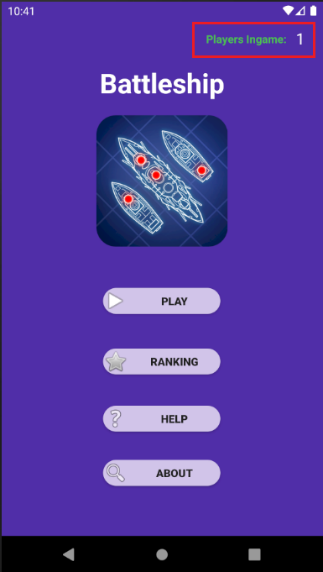
                player: playerStored

            })

## Aplicación Android

En este paso, tuvimos que adaptar el proyecto Android para poder trabajar junto con el servidor que creamos.

Para ello hemos optado por controlar que hay algún jugador en la pantalla de juego. Si un nuevo jugador se conecta y entra a jugar, los demás usuarios serán notificados, reflejándose en la pantalla de inicio.



En el caso de que un jugador termine de jugar y guarde sus estadísticas, de la misma manera se notificará en la pantalla principal de que el ranking de jugadores ha recibido una actualización.



## Multilenguaje

Hemos añadido la opción de multilenguaje dependiendo de la zona en la que se ejecuta la aplicación.

También, decidimos poner un icono flotante en la pantalla principal del proyecto para que el usuario pueda cambiar el idioma a su elección sin tener que cambiar el del teléfono.



# Conclusión

## Feed-back

El proyecto nos ha parecido interesante por el hecho de trabajar con servidores y datos en tiempo real.

También nos ha gustado la libertad de trabajo que hemos tenido a la hora de programar por tener que investigar por nosotros mismos.

Nos habría gustado implementar alguna cosa extra, pero no hemos podido por falta de tiempo.