Contenido

1 Base de datos: tecnología y gestión

1.1 Tecnologías usada para la BD

1.1.1 SQLite

Es una base de datos **embebida en un solo fichero** (***db/skyport.sqlite3***).

- No hay servidor aparte (como MySQL/Postgres), solo un archivo en disco.
- Para este proyecto tipo juego individual – prototipo, nos ha parecido parece adecuada: simple, portable y sin instalaciones complicadas.

1.1.2 Knex

Es un **query builder** para Node.js. Tiene las siguientes ventajas:

- Se escriben las consultas en **JS** (no cadenas SQL sueltas por todo el código).
- Tiene **migrations** → se puede versionar el esquema (crear/alterar tablas).
- Tiene **seeds** → se pueden poblar datos iniciales fácilmente.
- Si algún día decidimos migrar de SQLite a Postgres, casi todo el código de datos se podría mantener (según la información consultada).

1.2 Ficheros clave en la BD del proyecto

1.2.1 knexfile.js (en la raíz)

Es el **archivo de configuración de Knex** para trabajar por línea de comandos:

```
const path = require('path');

module.exports = {
 development: {
  client: 'sqlite3',
  connection: {
   filename: path.join(__dirname, 'db', 'skyport.sqlite3')
  },
  useNullAsDefault: true,
  migrations: { directory: path.join(__dirname, 'migrations') },
  seeds: { directory: path.join(__dirname, 'seeds') }
 }
};
```

Este archivo indica a Knex:

- Qué motor usar (sqlite3).
- Dónde está la BD (db/skyport.sqlite3).
- Dónde buscar migrations y seeds.

1.2.2 db/knex.js

Este es el módulo que usa **el código de la aplicación** para conectarse a la base de datos:

```
const knexConfig = require('../knexfile');
const env = process.env.NODE_ENV || 'development';
```

```
const knex = require('knex')(knexConfig[env]);

module.exports = knex;
```

Crea una instancia de Knex usando la config de ***knexfile.js*** y la exporta: el resto del proyecto. Simplemente habrá que hacer:

```
const db = require('../db/knex');.
```

1.2.3 Creación base de datos: migrations/

Cada archivo ***20xxxxx_create_XXXX_table.js*** contiene algo así:

```
exports.up = function(knex) {
 return knex.schema.createTable('users', table => {
  table.increments('id').primary();
  table.string('username').notNullable().unique();
  table.string('password_hash').notNullable();
  table.timestamps(true, true);
 });
};

exports.down = function(knex) {
 return knex.schema.dropTableIfExists('users');
};
```

- **up**: cómo crear la tabla.
- **down**: cómo revertirla.

Cuando se ejecuta:

```
npx knex migrate:latest:
```

Knex recorre **todas** las migrations pendientes en orden y crea/modifica las tablas.

1.2.4 Carga de datos: seeds/

Son scripts que insertan datos iniciales, usan Knex:

```
exports.seed = async function(knex) {
 await knex('users').del();                    // limpia
 await knex('users').insert([
  { id: 1, username: 'julio', password_hash: '...' },
  // ...
 ]);
};
```

Con:

```
npx knex seed:run [1]
```

se ejecutan todos los seed.

1.2.5 La base de datos - bd/skyport.sqlite3 y db/skyport.empty.sqlite3

- ***skyport.sqlite3***: BD que usa juego.

---

[1] ***npx knex seed:run --specific 001_nombre_del_seed.js*** (para ejecutar un único seed).

- ***skyport.empty.sqlite3***: plantilla vacía (sin datos) para poder "resetear" copiándola encima si algo se rompe.

1.3 Instalar, crear y poblar la base de datos desde cero

Imaginemos que borramos ***db/skyport.sqlite3*** y empezamos de cero.

1. Instalar dependencias

// sqlite3 es el driver. knex es el query builder.
- npm install knex sqlite3.

// Si queremos usar la CLI de Knex:
- npx knex --help

2. Configurar knexfile.js

Lo importante es que **apunte a la ruta correcta** del fichero *.sqlite3* y a las carpetas migrations y seeds.

```
const path = require('path');

module.exports = {
 development: {
  client: 'sqlite3',
  connection: {
   filename: path.join(__dirname, 'db', 'skyport.sqlite3')
  },
  useNullAsDefault: true,
  migrations: { directory: path.join(__dirname, 'migrations') },
  seeds: { directory: path.join(__dirname, 'seeds') }
 }
};
```

3. Crear la base de datos (fichero físico)

Con SQLite no hace falta "crear" la BD con un comando específico:

- Si **no existe *"db/skyport.sqlite3"***, se creará cuando se ejecuten las migrations.
- Solo debemos asegurarnos de que la carpeta "***db/"*** existe.

4. Ejecutar migrations para levantar el esquema

Desde la raíz del proyecto:

```
npx knex migrate:latest --env development
```

¿Qué hace esto?

- Mira ***knexfile.js***
- Se conecta a ***db/skyport.sqlite3***.
- Crea todas las tablas que están definidas en ***migrations/*** y marca en una tabla interna (***knex_migrations***) cuáles están aplicadas.

Si algún día se quiere revertir la última migration:

```
npx knex migrate:rollback --env development
```

5. Ejecutar seeds . para cargar datos iniciales

```
npx knex seed:run --env development
```

- Esto ejecuta en orden las seeds que tengamos en la carpeta /seeds/:
    - 01_users.js
    - 02_aircraft_types.js
    - 03_account_initial_credit.js
    - 04_missions.js
- Al terminar, la BD ya tiene:
    - Usuarios de prueba.
    - Tipos de avión.
    - Crédito inicial.
    - Misiones base.

A partir de ahí, se puede arrancar la app (***node app.js***) y el juego ya puede leer de la BD.

1.4 Uso básico de Knex

En el proyecto NO se crean instancias nuevas de Knex. Siempre usamos **la misma**, que es la que exporta ***db/knex.js***.

Importar la conexión

Ejemplo en ***data/usersStore/db.js***:

```
const knex = require('../../db/knex');
async function findUserByUsername(username) {
    const row = await knex('users').where({ username }).first();
    return mapRow(row);
}
```

Consultas típicas: SELECT, INSERT, UPDATE…

Ejemplo de JOIN sencillo con Knex

```
// --- Flota del usuario ---
const knex = require('../../db/knex');
async function getFleetForUser(userId) {    return
knex('user_aircraft')
        .join('aircraft_types', 'user_aircraft.aircraft_type_id',
'aircraft_types.id')
        .where('user_aircraft.user_id', userId)
        .select(
            'user_aircraft.id as id',
            'user_aircraft.nickname as nickname',
            'user_aircraft.status as status',
            'user_aircraft.purchased_price as purchasedPrice',
            'user_aircraft.purchased_at as purchasedAt',
            'aircraft_types.id as typeId',
            'aircraft_types.role as role',
            'aircraft_types.name as model',
            'aircraft_types.base_price as basePrice',
            'aircraft_types.description as description'
        );
}
```

Todo esto se hace *con async/await*, porque **Knex devuelve Promesas**.

1.5 Cómo se conecta todo

La cadena es:

1. **db/knex.js**

   Crea **una** instancia de Knex y la exporta.

2. **data/*Store/db.js**

   Importan esa instancia y definen funciones de acceso a datos:

   - getUserById, createUser, getUserFleet, getAvailableMissions, etc.

3. **routes/api/*.js**

   - Reciben peticiones HTTP.
   - Llaman a los stores de data/ para leer/escribir en la BD.
   - Devuelven JSON al front (public/js/*.js).

4. **Front (public/js/*.js)**

   - Hace *fetch('/api/game/fleet'), fetch('/api/game/missions'),* etc.
   - Pinta en views/game/*.html usando DOM.

2 Esquema de Base de Datos - SkyPort

- Base de datos: **SQLite 3**
- Gestor de migraciones: **Knex.js**
- Modelo lógico: **orientado a usuario**

2.1 Tabla: users

2.1.1 **Descripción general**

Contiene la información principal de los usuarios registrados del sistema. Es la entidad base de la que dependen la mayoría de relaciones.

2.1.2 Columnas

Campo	Tipo	Not Null	Default	PK	Descripción
id	varchar(255)	NO	—	✓	Identificador único del usuario.
username	varchar(255)	✓	—	—	Nombre de usuario único.
email	varchar(255)	✓	—	—	Email del usuario, único.
password_hash	varchar(255)	✓	—	—	Hash bcrypt de la contraseña.
roles	TEXT	✓	'[]'	—	Lista JSON con roles (ej: ["player"]).
is_active	boolean	✓	'1'	—	Usuario activo/inactivo.
current_balance	INTEGER	✓	'0'	—	Saldo actual de la cuenta del jugador.
last_login_at	datetime	NO	—	—	Última fecha de inicio de sesión.
created_at	datetime	NO	CURRENT_TIMESTAMP	—	Fecha de creación.
updated_at	datetime	NO	CURRENT_TIMESTAMP	—	Última actualización.

2.2 Tabla: user_airports

2.2.1 **Descripción**

Aeropuertos controlados por cada usuario. Permite que un jugador tenga uno o varios aeropuertos base para operar.

2.2.2 **Columnas**

Campo	Tipo	Not Null	Default	PK	Descripción
id	varchar(255)	NO	—	✓	Identificador único del aeropuerto del usuario.

Campo	Tipo	Not Null	Default	PK	Descripción
user_id	varchar(255)	✓	—	—	FK → users.id.
name	varchar(255)	✓	—	—	Nombre del aeropuerto (ej: "Barajas").
level	INTEGER	✓	'1'	—	Nivel del aeropuerto.
xp	INTEGER	✓	'0'	—	Experiencia acumulada.
created_at	datetime	NO	CURRENT_TIMESTAMP	—	Fecha creación.
updated_at	datetime	NO	CURRENT_TIMESTAMP	—	Última actualización.

2.3 Tabla: aircraft_types

2.3.1 **Descripción**

Catálogo global de tipos de avión disponibles en el juego. No son aviones del usuario, sino la "plantilla".

2.3.2 **Columnas**

Campo	Tipo	Not Null	Default	PK	Descripción
id	varchar(255)	NO	—	✓	Identificador del tipo de avión.
name	varchar(255)	✓	—	—	Nombre comercial (A320, B738, C17...).
role	varchar(255)	✓	—	—	Rol: pasajeros, carga, militar, reconocimiento, transporte...
base_price	INTEGER	✓	—	—	Precio base del avión.
description	TEXT	NO	—	—	Descripción opcional del avión.
is_active	boolean	✓	'1'	—	Visible/no visible para compra.
created_at	datetime	NO	CURRENT_TIMESTAMP	—	Fecha creación.
updated_at	datetime	NO	CURRENT_TIMESTAMP	—	Ult. actualización.

2.4 Tabla: user_aircraft

2.4.1 **Descripción**

Instancias concretas de aviones comprados por cada usuario. Relaciona un *user_id* con un *aircraft_type*.

2.4.2 **Columnas**

Campo	Tipo	Not Null	Default	PK	Descripción
id	varchar(255)	NO	—	✓	Identificador único del avión del usuario.
user_id	varchar(255)	✓	—	—	FK → users.id.
aircraft_type_id	varchar(255)	✓	—	—	FK → aircraft_types.id.
status	varchar(255)	✓	'idle'	—	Estado: idle, running, maintenance, sold...
purchased_price	INTEGER	✓	—	—	Precio pagado.
purchased_at	datetime	✓	—	—	Fecha compra.
sold_at	datetime	NO	null	—	Fecha venta (si aplica).
sold_price	INTEGER	NO	—	—	Precio venta (si aplica).
nickname	varchar(255)	NO	—	—	Nombre personalizado.
created_at	datetime	NO	CURRENT_TIMESTAMP	—	Fecha creación.
updated_at	datetime	NO	CURRENT_TIMESTAMP	—	Última actualización.

2.5 Tabla: missions

2.5.1 **Descripción**

Misiones disponibles globalmente en el juego. Cada misión tiene un coste, recompensa, duración y un tipo asociado a un rol de avión.

2.5.2 **Columnas**

Campo	Tipo	Not Null	Default	PK	Descripción
id	varchar(255)	NO	—	✓	Identificador único de misión.
name	varchar(255)	✓	—	—	Nombre de la misión.
type	varchar(255)	✓	—	—	Rol de misión (coincide con rol del avión).
cost	INTEGER	✓	—	—	Coste a descontar al empezar.
reward	INTEGER	✓	—	—	Recompensa al completar.
duration_seconds	INTEGER	✓	—	—	Duración total.
description	TEXT	NO	—	—	Detalles de la misión.

Campo	Tipo	Not Null	Default	PK	Descripción
level_required	INTEGER	✓	'1'	—	Nivel mínimo del jugador.
is_active	boolean	✓	'1'	—	Visible/no visible.
created_at	datetime	NO	CURRENT_TIMESTAMP	—	Fecha creación.
updated_at	datetime	NO	CURRENT_TIMESTAMP	—	Última actualización.

2.6 Tabla: user_missions

2.6.1 **Descripción**

Misiones aceptadas por cada usuario, asociadas a un avión concreto. Permite controlar misiones en progreso, fallidas o completadas.

2.6.2 **Columnas**

Campo	Tipo	Not Null	Default	PK	Descripción
id	varchar(255)	NO	—	✓	Identificador único de misión del usuario.
user_id	varchar(255)	✓	—	—	FK → users.id.
mission_id	varchar(255)	✓	—	—	FK → missions.id.
aircraft_id	varchar(255)	✓	—	—	FK → user_aircraft.id.
status	varchar(255)	✓	'running'	—	running / done / failed / cancelled
started_at	datetime	✓	—	—	Fecha de inicio.
finished_at	datetime	NO	—	—	Fecha fin.
cost_at_start	INTEGER	✓	—	—	Coste aplicado al inicio.
reward_on_success	INTEGER	✓	—	—	Recompensa prevista.
cost_applied	boolean	✓	'0'	—	Indica si se descontó el coste.
reward_applied	boolean	✓	'0'	—	Indica si se abonó la recompensa.
failure_reason	TEXT	NO	—	—	Explicación si falla.
created_at	datetime	NO	CURRENT_TIMESTAMP	—	Fecha creación.
updated_at	datetime	NO	CURRENT_TIMESTAMP	—	Última actualización.

2.7 Tabla: account_movements

2.7.1 **Descripción**

Histórico de movimientos económicos del usuario. Registra ingresos y gastos generados por misiones, compras, ventas, etc.

2.7.2 **Columnas**

Campo	Tipo	Not Null	Default	PK	Descripción
id	varchar(255)	NO	—	✓	Identificador único de movimiento.
user_id	varchar(255)	✓	—	—	FK → users.id.
created_at	datetime	✓	CURRENT_TIMESTAMP	—	Fecha del movimiento.
type	varchar(255)	✓	—	—	credit / debit.
amount	INTEGER	✓	—	—	Importe del movimiento.
description	TEXT	NO	—	—	Comentario (ej: “compra avión”).
related_aircraft_id	varchar(255)	NO	—	—	FK opcional → user_aircraft.id.
related_mission_id	varchar(255)	NO	—	—	FK opcional → user_missions.id.

2.8 Relaciones principales del modelo

- users (1) --- (N) user_aircraft
- users (1) --- (N) user_airports
- users (1) --- (N) user_missions
- users (1) --- (N) account_movements

==========

- aircraft_types (1) --- (N) user_aircraft

==========

- missions (1) --- (N) user_missions

==========

- user_aircraft (1) --- (N) user_missions

==========

- user_missions (1) --- (1) account_movements