ESTADO DE EJECUCIÓN

Plan de proyecto y tareas programadas

En base al proyecto a desarrollar se decide el stack de tecnologías a usar, estructurándose de la siguiente manera:

- Estudiar y documentarse sobre las tecnologías a utilizar.
- Para el flujo de trabajo se usarán metodologías ágiles, haciendo sprints por cada funcionalidad.
- El trabajo en equipo se realizará a través de GitHub y haciendo uso de GIT para el control de versiones. Se crean ramas según la funcionalidad, manteniendo una clara separación entre el entorno de producción y el de desarrollo.
- Se realiza un análisis funcional y no funcional donde se reúnen los requisitos.
- Se realiza el diseño del prototipo incorporándose en el mismo los diagramas de flujo.

En consecuencia de todo lo mencionado con anterioridad, se establecen las siguientes tareas y objetivos:

0 - Estudio de tecnologías
1 – Estructura y configuración de proyecto
2 - Modelo de datos (BD relacional)
3 – Registro y login de usuarios
4 – Consumir APIs
5 – Buscador
6 – Filtros
7 – Perfil de usuarios
8 – UI / UX

Problemas surgidos y redefinición de objetivos

Dificultades críticas

El mayor impedimento a la hora de implementar partes del proyecto ha venido de la mano de las APIs, surgiendo problemas críticos que han obligado a tomar una decisión drástica o arriesgarnos a no conseguir los objetivos establecidos. Esta decisión ha cambiado la propuesta inicial, teniendo que redefinir el proyecto en fechas muy cercanas a la entrega.

La propuesta inicial consistía en consultar varias APIs de plataformas de videojuegos, mostrando la información relevante a cada videojuego.

Estas APIs eran de uso gratuito pero todas requerían de un acceso con token o API key, consumiéndolas a través del método POST. En un principio no nos pareció un impedimento, ya que habíamos realizado este tipo de consultas anteriormente. Además se ejecutaron con Postman satisfactoriamente.

Al realizar la implementación de dicha funcionalidad nos encontramos con problemas, ya que la consulta requería un header para que los datos de los tokens no fueran visibles. Consultando la documentación oficial de las APIs se vio que usar este método podría conllevar problemas de CORS, siendo recomendable acceder a esta información a través de un proxy.

Ante estos tres problemas nos resultó muy complicado seguir esta vía inicial, ya que requería bastantes más horas de investigación además de estar la fecha límite cada vez más cercana. Se expuso el impedimento en la tutoría del día 7 de mayo y se tomó la decisión de abandonar las APIs de videojuegos y en su lugar reemplazarlas por una API que se consuma a través del método GET, eligiéndose la PokeAPI.

Tanto el diseño del prototipo como la idea general se siguen manteniendo.

Dificultades superadas

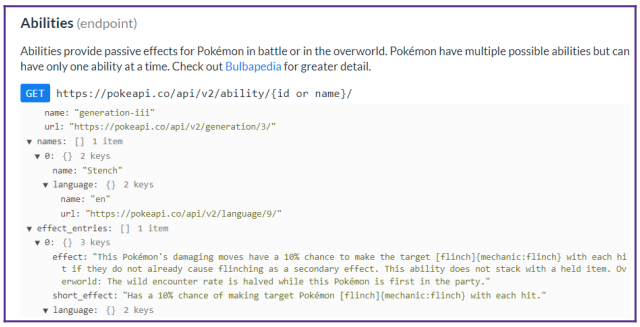
Autenticación: para gestionar roles y usuarios se utilizó la dependencia de NextAuth.js. Al intentar implementarlo dio errores incongruentes que referenciaban líneas exentas de código. Tras varios días depurando e código y buscando en la documentación oficial y en internet, se llegó a la solución de instalar una versión anterior de la dependencia NextAuth.js. Cuando NextAuth.js es instalado por primera vez se hace por defecto en su última versión, que en el caso de nuestro proyecto derivaba en un conflicto con las dependencias de Prisma.

Control de acceso: la página de perfil de usuario contiene datos sensibles y no debería ser accesible sin una sesión iniciada. En un primer momento se suplió con una redirección al login cuando se intentaba usar esta ruta. Este método hacía que la aplicación no funcionara correctamente y lanzara errores. Leyendo la documentación de Next.js observamos que para controlar el enrutamiento existe el archivo middleware.ts. Este se ejecuta con cada petición, pudiendo modificar, reescribir o redirigir la respuesta. De esta forma pudimos controlar la restricción de acceso de usuarios que no tengan sesión.

PokeAPI: la estructura de la API tiene se conforma de:

Endpoints generales

Endpoints específicos



Para extraer datos generales de los que obtener datos específicos (por ejemplo la lista de los Pokémon) se necesitan varias llamadas encadenadas a la API. Esto supuso invertir más tiempo, ya que requiere crear y optimizar las funciones para reutilizarlas siempre que sea posible.

Al estar usando Typescript todas las respuestas requieren de un tipado, por lo que es necesario realizar interfaces para cada estructura de datos diferente. La documentación nos sirve de guía para precisar estas intefaces.

Acumulación de peticiones: el primer planteamiento para mostrar todos los Pokémon (en total 1.302) fue un único fecth general que contiene un array con todos los nombres y la URL específica. Después se iteraba el array a través del método Promise.all() lo que conllevaba un tiempo de respuesta de más de 30 segundos, o incluso rechazar la respuesta al fallar una de las promesas. Se estudió utilizar Promise.allSettled() ya que aunque una promesa sea rechazada sigue devolviendo el valor del resto. No se utilizó por tener conflictos con Next.js.

Para optimizar los tiempos de espera y evitar el riesgo de rechazo se implementa la paginación, donde se mantiene la petición general pero el array se divide en segmentos de 10 elementos, los que se mostrarán por página.

Toggles: se decidió usar este tipo de componente para hacer filtros ya que devuelve un valor booleano, permitiendo un estado activado o desactivado. Se planteó poder filtrar por varios tipos a la vez, pero al formar la url con los parámetros seleccionados no siempre se realizaba con éxito ya que necesita un tiempo específico para modificar la URL. Al pulsar varios filtros consecutivamente no ejecutaba bien las instrucciones, pudiendo llevar al usuario a confusión. Se decidió que solo se filtraría por un tipo a la vez.

TailwindCSS dinámico: para poner colores de manera dinámica a los filtros y a los tipos de los Pokémon se crean colores propios en **tailwind.config.ts**. Cuando se iteran los elementos que incorporan alguna de estas clases se agrega el nombre del filtro al campo correspondiente.

Por como funciona Tailwind, la clase de estilos se aplica antes del renderizado, por lo que si hay alguna variable en el className será visible en el código pero tailwind no creará la clase correspondiente porque no la conoce. Para resolver este problema se usa el safelist, que sería equivalente a tener clases estáticas. Esto hace que estas clases se creen independientemente de si existen el prerenderizado o no. Para optimizar el safelist se utiliza un RegEx.

Ejecución del proyecto y temporalización

De todos los objetivos marcados se han cumplido los siguientes, teniendo en cuenta los cambios del proyecto:

- Entrega propuesta de proyecto **(31/3/2024)**
- Entrega anteproyecto **(14/4/2024)**
- Estudio de tecnologías **(a partir del 14/4/2024)**
  - Next.js
  - Typescript
  - Prisma
  - PostgreSQL
  - TailwindCSS
  - Vercel
- Diseño y prototipado **(30/4/2024)**
- Estructura y configuración de proyecto **(1/5/2024)**
  - Creación del proyecto
  - Organización de dependencias y módulos
  - Jerarquía de carpetas
- Modelo de datos **(3/5/2024)**
  - Creación de base de datos relacional (Supabase)
  - Triggers de actualización y borrado de usuarios
- Entrega del prototipo **(5/5/2024)**
- Realizar login y registro **(5/5/2024)**
  - Autenticación
  - Validación
  - CRUD usuarios
  - Tratamiento de sesiones
- Consumir la API **(8/5/2024)**
  - Estudiar estructura
  - Crear interfaces para tratar las respuestas en formato JSON

- Buscador **(10/5/2024)**
  - Consultas especificas a la API según la query
  - Paginación para mostrar resultados y mejorar el rendimiento de la aplicación

- Despliegue **(12/5/2024)**

- Filtrado **(15/5/2024)**
  - Definir los filtros por tipos
  - Respetar la QUERY de búsqueda al filtrar

- Perfil usuarios **(17/5/2024)**
  - Modificar datos de usuarios
  - Eliminación de cuenta

- UI/UX **(22/5/2024)**
  - Perfil de usuario
  - Cartas Pokémon
  - Detalles Pokémon

- Despliegue **(26/5/2024)**
  - Actualización del entorno de producción

Estado del proyecto

Salvo por el problema inicial que retrasó todo el proyecto, los objetivos e hitos de entregas se fueron cumpliendo satisfactoriamente, quedando todavía por desarrollar los siguientes:

- Mostrar datos de API
  - Contenido para la página home

- Perfil de usuarios
  - Agregar Pokémon favoritos al perfil

- Realizar frontend
  - Diseño de experiencia de usuario
  - Página de un Pokémon

- Agregar funciones de administrador
  - Rol de administración
  - Mostrar estadísticas de usuarios (última fecha de login, usuarios borrados)

Estado actual del código

- Proyecto en GitHub: https://github.com/CMallen29/PFG

- Proyecto desplegado: https://unify-sage.vercel.app/