Glaiel, Agustin

Plataforma web de pronósticos deportivos e información interactiva de Fórmula 1 - PREDI

Tesis para la obtención del título de grado de Ingeniero en Sistemas

Directores:

Carreño, Ignacio Luciano

Porrini, Federico

Universidad Católica de Córdoba
Facultad de Ingeniería
Carrera Ingeniería de Sistemas

Informe de Proyecto Integrador

Plataforma web de pronósticos deportivos e información interactiva de Fórmula 1 - PREDI

Autor
Glaiel, Agustin

Tutores
Ing. Ignacio Luciano Carreño
Ing. Federico Porrini

Tabla de contenido

Resumen

Predi es una plataforma web dedicada a los aficionados de la Fórmula 1, diseñada para ofrecer información actualizada y pronósticos interactivos de cada sesión de la temporada. La iniciativa surge de la combinación de una pasión personal por el automovilismo y la observación de la falta de un espacio similar a sitios de referencia en otros deportes, como “Promiedos” en el fútbol. El problema identificado es la dispersión y dificultad de acceso a datos relevantes de la F1, así como la ausencia de una herramienta que integre información y pronósticos en un solo lugar.

La solución propuesta consiste en un sitio web que centraliza datos de pilotos, circuitos y eventos, junto con un sistema de pronósticos deportivos (“prode”) que permite a los usuarios competir entre sí, ya sea de manera global o en grupos privados, acumulando puntos según sus aciertos. La plataforma incorpora también un foro para fomentar la interacción y el intercambio de opiniones.

El desarrollo se realizó con un enfoque centrado en la experiencia del usuario, priorizando la accesibilidad, el diseño responsivo y la facilidad de uso.

Los resultados muestran un prototipo funcional que permite consultar información de carreras pasadas y futuras, participar en pronósticos y visualizar clasificaciones. Se concluye que Predi cubre una necesidad latente entre los fanáticos de la F1, ofreciendo una experiencia unificada que combina datos, juego y comunidad.

Abstract

Predi is a web platform for Formula 1 fans, designed to provide up-to-date information and interactive predictions for each session of the season. The project originated from a personal passion for motorsports and the observation of the lack of a dedicated space similar to well-known sports websites such as “Promiedos” in football. The identified problem is the dispersion and difficulty of accessing relevant F1 data, along with the absence of a tool that integrates both information and predictions in one place.

The proposed solution is a website that centralizes driver, circuit, and event data, along with a sports prediction system (“prode”) that allows users to compete globally or in private groups, earning points based on their accuracy. The platform also features a forum to encourage interaction and discussion.

Development followed a user-centered approach, prioritizing accessibility, responsive design, and ease of use.

Results show a functional prototype that enables users to consult past and upcoming race information, participate in predictions, and view leaderboards. It is concluded that Predi addresses an unmet need among F1 fans by offering a unified experience that combines data, gameplay, and community.

Presentación del Tema

Este proyecto da vida a Predi, una plataforma web pensada para los fanáticos de la Fórmula 1, con la idea de que más adelante pueda crecer hasta convertirse en una aplicación móvil. La inspiración proviene de la pasión por este deporte y de años de interacción en grupos de amigos realizando pronósticos de resultados de carreras de una manera divertida y cercana. También surge de una observación: en el mundo de la Fórmula 1 falta algo importante. No existen plataformas interactivas que realmente inviten a los aficionados a involucrarse plenamente en el deporte mediante pronósticos de carreras y clasificaciones. Esto no solo limita la conexión entre los fans y su pasión, sino que también reduce la riqueza y emoción de la experiencia de seguir la Fórmula 1.

A diferencia de otros deportes, como el fútbol, donde los pronósticos deportivos de los mundiales o torneos locales despiertan gran interés y acercan más a los hinchas a la acción, la Fórmula 1 todavía no ha aprovechado ese potencial. Actualmente, los fans se encuentran dispersos en diversas plataformas, pero ninguna les ofrece un espacio donde puedan compartir predicciones, competir, y acceder a información detallada sobre el deporte en un solo lugar. Esto representa no solo una brecha que puede cubrirse, sino también una oportunidad única para construir una comunidad más unida y comprometida con la Fórmula 1.

Por ello, Predi busca ser la solución. Su propuesta se apoya en tres pilares principales: un sistema de pronósticos deportivos, un espacio con información clave sobre el deporte y un foro en el que los usuarios puedan debatir sobre cualquier aspecto relacionado con la Fórmula 1. Los participantes podrán anticipar resultados de clasificaciones y carreras, acumular puntos y competir en tablas globales o en grupos privados con amigos. Además, la plataforma ofrecerá datos actualizados sobre próximos eventos y resúmenes de eventos pasados, presentados de forma simple y directa. Predi no es solo un lugar para informarse, sino también para divertirse y

sentirse parte de la pasión por la Fórmula 1, elevando la experiencia de los aficionados a otro nivel.

Glosario

Grand Prix o Grande premio: Se le dice a un evento que implica todas las sesiones de un mismo fin de semana de carreras en un mismo circuito.

Práctica Libre: Sesión de entrenamiento durante un fin de semana de Gran Premio (FP1, FP2, FP3) donde los equipos prueban los autos, ajustan configuraciones y recopilan datos. Cada sesión suele durar una hora, permitiendo a los pilotos familiarizarse con el circuito.

Parrilla de salida: El orden en el que los pilotos comienzan la carrera principal, determinado por los resultados de la clasificación. Define las posiciones iniciales de los autos en la pista.

Qualy (Clasificación): Sesión dividida en tres etapas (Q1, Q2, Q3) que determina la parrilla de salida para la carrera principal. En el formato normal, Q1 dura 18 minutos, Q2 15 minutos y Q3 define los 10 primeros, con el objetivo de lograr el tiempo de vuelta más rápido.

Qualy Sprint: Una clasificación más corta usada en fines de semana con formato sprint. Consta de Q1 (12 minutos), Q2 (10 minutos) y Q3 (8 minutos), y define la parrilla de salida para la Carrera Sprint del sábado.

Carrera Normal: La carrera principal de un Gran Premio, disputada el domingo, con una duración de 1.5 a 2 horas o aproximadamente 305. Otorga puntos a los 10 primeros (25-18-15-12-10-8-6-4-2-1) y es el evento central del fin de semana.

Carrera Sprint: Una carrera corta (100 km o menos de la mitad de las vueltas de la carrera normal) que se disputa el sábado en algunos Grandes Premios. Otorga puntos a los 8 primeros (8-7-6-5-4-3-2-1) y puede influir en la parrilla de la carrera principal.

Campeonato de Pilotos: La competencia anual donde cada piloto suma los puntos obtenidos en las carreras (normales y sprints) para consagrarse como el campeón individual al final de la temporada. El piloto con más puntos gana el título.

Campeonato de Equipos (o Constructores): La competencia anual donde las escuderías (10 equipos con 2 pilotos cada uno) suman los puntos de ambos pilotos en todas las carreras. El equipo con más puntos al final de la temporada gana el título de constructores.

Backend: Parte de una aplicación web que se ejecuta en el servidor y maneja la lógica, los datos y las operaciones "detrás de escena".

Frontend: Parte de una aplicación web que corre en el navegador del usuario y muestra la interfaz con la que interactúan (botones, texto, imágenes).

API (Application Programming Interface): Conjunto de reglas que permite que dos sistemas (e.g., frontend y backend) se comuniquen.

API Gateway: Punto de entrada único para solicitudes en una app (especialmente microservicios), que enruta, autentica (e.g., con JWT) y gestiona tráfico.

Arquitectura Distribuida: Diseño donde la app se divide en servicios independientes (e.g., microservicios) que se comunican.

Arquitectura Monolítica: Diseño donde toda la app (frontend, backend, base) está en un solo bloque de código.

Balanceo de Carga: Técnica para repartir solicitudes entre varios servidores y evitar que uno se sature.

Base de Datos NoSQL: Tipo de base que almacena datos de forma flexible (e.g., documentos, clave-valor) sin esquemas fijos.

Base de Datos SQL: Tipo de base que organiza datos en tablas con relaciones definidas. Es como un archivador con carpetas etiquetadas: todo tiene su lugar y estructura.

Bibliotecas: Colecciones de funciones o herramientas específicas que podés usar en tu código para tareas comunes (e.g., manipular fechas o hacer gráficos). A diferencia de un framework, no impone una estructura. Es como una caja de herramientas: elegís lo que necesitás.

Caching: Técnica de guardar datos usados frecuentemente en memoria rápida (e.g., Redis) para responder sin consultas lentas.

CDN (Content Delivery Network): Red de servidores globales que guarda copias de contenido estático (imágenes, CSS) cerca de los usuarios.

CI/CD (Integración Continua/Despliegue Continuo): Proceso automatizado para probar y desplegar código cada vez que hay cambios.

Contenedor: Entorno ligero que empaqueta una app con sus dependencias (código, bibliotecas) y usa el kernel del sistema host para ejecutarse. Ejemplo: Docker.

CORS (Cross-Origin Resource Sharing): Mecanismo que permite a un frontend en un dominio solicitar recursos de otro, si el servidor lo autoriza.

Despliegue: Proceso de llevar una app desde el desarrollo a un entorno donde los usuarios la puedan usar (e.g., un servidor).

DNS (Domain Name System): Sistema que traduce nombres de dominio (e.g., www.google.com) en direcciones IP. Es como una agenda telefónica: buscás un nombre y te da el número para llamar.

Docker: Herramienta para crear y ejecutar contenedores, empaquetando apps con sus dependencias.

Endpoint: Punto específico de una API al que se envían solicitudes (e.g., /api/usuarios para obtener usuarios).

Fixes: Correcciones o ajustes en el código para solucionar errores (bugs) o mejorar algo.

Frameworks: Estructuras predefinidas de código que facilitan el desarrollo al ofrecer herramientas y patrones listos para usar. Ejemplo: React para frontend o Django para backend.

HTTPS: Versión segura de HTTP que cifra la comunicación entre cliente y servidor con TLS/SSL.

JSON (JavaScript Object Notation): Formato ligero para intercambiar datos entre sistemas, basado en pares clave-valor (e.g., {"nombre": "Ana", "edad": 25}). Es como una nota escrita en un idioma que todos entienden, simple y clara.

JWT (JSON Web Token): Token para autenticación que contiene datos firmados (e.g., ID de usuario) y se verifica sin guardar estado.

Kernel: Núcleo del sistema operativo que gestiona los recursos básicos (CPU, memoria, disco) y permite que el software hable con el hardware. En contenedores, todos comparten el kernel del host.

Lenguaje Compilado: Lenguaje que se traduce completamente a código máquina antes de ejecutarse. Ejemplo: Go.

Lenguaje Interpretado: Lenguaje de programación que se ejecuta línea por línea sin necesidad de compilarlo antes. Ejemplo: Python.

Máquina Virtual: Entorno simulado que actúa como una computadora completa dentro de otra, con su propio sistema operativo. Usa un hipervisor (e.g., VirtualBox) para dividir recursos físicos.

Microservicios: Estilo de arquitectura donde la app se divide en servicios pequeños e independientes que se comunican.

Middleware: Capa de software que intercepta solicitudes entre cliente y servidor para tareas como autenticación (e.g., verificar JWT).

MVC (Model-View-Controller): Patrón de diseño que separa la lógica (Model), la interfaz (View) y el control (Controller).

Nube: Servicios de computación (servidores, almacenamiento) ofrecidos a través de internet por proveedores como AWS o Google Cloud.

ORM (Object-Relational Mapping): Técnica que convierte datos de una base de datos en objetos del lenguaje de programación (e.g., una fila de usuarios en un objeto "Usuario"). Ejemplo: SQLAlchemy en Python. Es como un traductor que hace que el código y la base hablen el mismo idioma.

PaaS (Platform as a Service): Servicio que provee una plataforma para desarrollar y desplegar apps sin gestionar servidores (e.g., Heroku).

Refresh Token: Token duradero que renueva un JWT expirado sin pedir credenciales otra vez.

REST (Representational State Transfer): Estilo de diseño para APIs que usa métodos HTTP (GET, POST) y recursos (e.g., /usuarios) para comunicarse de forma simple y predecible.

Reverse Proxy: Servidor que recibe solicitudes y las pasa a los servidores backend, ocultándolos.

SQL Injection: Ataque donde se envía código malicioso a una base de datos para manipularla.

XSS (Cross-Site Scripting): Ataque que inyecta código dañino en el navegador del usuario (e.g., robando cookies).

Diagnóstico

El mundo de la Fórmula 1 está viviendo un momento único: la cantidad de fans no para de crecer, las redes sociales explotan con cada carrera y las transmisiones rompen récords de audiencia. Sin embargo, detrás de este boom, hay un vacío que pocos notan: los aficionados no tienen un lugar digital que los reúna de verdad y les dé herramientas para vivir el deporte más allá de mirar las carreras. Las plataformas actuales, como la web oficial de la Fórmula 1 o aplicaciones de estadísticas, ofrecen datos fríos y básicos —posiciones, tiempos, noticias— pero se quedan cortas en algo clave: no invitan a participar ni a conectar entre sí. Esto deja a los fans dependiendo de chats dispersos en WhatsApp, publicaciones en X o foros genéricos que no están pensados para el ritmo y la pasión de este deporte.

Este escenario no es solo un problema, sino una oportunidad que está pidiendo a gritos ser aprovechada. Para los seguidores, la falta de una plataforma interactiva significa menos diversión y menos formas de compartir su entusiasmo. Imagínate a un fan que quiere predecir si Verstappen va a ganar en Mónaco o si Sainz va a sorprender en la sesión de clasificación, pero no tiene dónde probar su instinto ni compararlo con otros. Para las comunidades de amigos que siguen la Fórmula 1, no hay un espacio que les deje competir y sacarse chispas como lo hacen los hinchas del fútbol con sus prodes. Y para el deporte mismo, esta desconexión digital frena el potencial de enganchar aún más a una audiencia que ya está loca por las carreras.

Predi aparece justo para llenar ese hueco. No se trata solo de solucionar algo que no anda, sino de aprovechar un momento dorado: darle a los fans una manera de meterse en el juego, de sentirse parte de la acción y de hacer que cada Gran Premio sea más que solo verlo por televisión. Es una oportunidad para transformar cómo los aficionados viven la Fórmula 1, llevándola de ser un espectáculo a una experiencia compartida y activa.

Objetivos

La propuesta de Predi implica obviamente un desarrollo, implementación y mantenimiento de un software que logre cumplir los siguiente objetivos:

Objetivo Global

Desarrollar una plataforma web dedicada a la realización de pronósticos deportivos sobre la Fórmula 1, fomentando la comunidad dinámica e interactiva mediante funcionalidades sociales, competitivas e informativas.

Objetivos Específicos

1. Diseñar una interfaz intuitiva, responsive y accesible desde múltiples dispositivos.
2. Implementar un módulo robusto para realizar y gestionar predicciones sobre clasificaciones y carreras.
3. Desarrollar un sistema que permita la creación y administración de grupos privados con tablas clasificatorias internas.
4. Integrar información oficial actualizada sobre eventos y resultados.
5. Proveer un espacio interactivo para la publicación y discusión de temas deportivos entre usuarios.
6. Garantizar la protección y confidencialidad de los datos personales de los usuarios mediante medidas de seguridad adecuadas.
7. Realizar pruebas sistemáticas con usuarios para optimizar la experiencia previa al lanzamiento.
8. Planificar y ejecutar una estrategia integral de lanzamiento orientada a captar usuarios desde el inicio.

Marco Teórico

Historia de la Fórmula 1

Para adentrarnos más en el deporte de la Fórmula 1, es necesario hablar un poco de su historia. El Campeonato Mundial de Fórmula 1 de la FIA, conocida como Fórmula Uno, Fórmula 1 o simplemente F1, es la máxima competencia de automovilismo internacional y el campeonato de deportes de motor más popular y prestigioso del mundo. Esta categoría está regida por la Federación Internacional del Automóvil (FIA).

Las raíces de las carreras de Gran Premio se remontan a 1894 en Francia, donde comenzaron como eventos individuales en caminos de tierra, sin conexión entre sí y con pocas limitaciones. Con el tiempo, estas competiciones ganaron terreno: entre 1927 y 1934, el número de carreras consideradas Gran Premio creció hasta alcanzar un máximo de dieciocho en 1934, justo antes de la Segunda Guerra Mundial. Antes del conflicto, se empezaron a establecer reglas para autos y pilotos, conocidas colectivamente como "Fórmula". Sin embargo, estas no se formalizaron hasta 1947, cuando la antigua AIACR se reorganizó en la FIA. Tras la guerra, en 1945, solo se disputaron cuatro carreras, pero el impulso continuó. Al final de la temporada de 1949, la FIA anunció que en 1950 unirían varios Grandes Premios nacionales para crear el primer Campeonato Mundial de Pilotos, marcando el nacimiento oficial de la Fórmula 1 moderna. Ese año, escuderías como Ferrari, Alfa Romeo y Maserati participaron, estableciendo un sistema de puntuación con siete carreras reconocidas. Por motivos económicos, entre 1952 y 1953 se usaron autos de Fórmula 2, y el calendario incluyó carreras no clasificadas como Grandes Premios hasta 1983.

Cada carrera se denomina Gran Premio, y el torneo completo es el Campeonato Mundial de Fórmula 1. Los eventos se llevan a cabo principalmente en autódromos, aunque también se utilizan circuitos callejeros, y en el pasado se emplearon circuitos ruteros. Los monoplazas, equipados con la última tecnología disponible (siempre regulada por un reglamento técnico), son

protagonistas. Algunas innovaciones, como el freno de disco, han pasado de la F1 a los autos comerciales. El campeonato ha visto la evolución de escuderías icónicas: mientras algunas como Ferrari y Alfa Romeo fueron fundacionales, otras como McLaren, Williams, Red Bull y Mercedes (que regresó con fuerza) han dominado en distintos momentos, alzándose con el Campeonato Mundial de Constructores. Por su parte, los pilotos necesitan la superlicencia de la FIA, obtenida por resultados en otras categorías, para competir.

Formato de la Competición

A medida que fue transcurriendo el tiempo, el formato de la competición fue cambiando junto con las reglas de esta. El formato anual de la temporada de Fórmula 1 se refiere a la estructura y organización que sigue el campeonato mundial de automovilismo de Fórmula 1 cada año. Es un conjunto de reglas, eventos y procedimientos que definen cómo se desarrolla la competencia a lo largo de un ciclo anual, generalmente desde marzo hasta noviembre o diciembre. Podemos definir el formato de la siguiente manera:

Calendario de Carreras:

- **Grandes Premios:** La temporada consta de una serie de carreras individuales conocidas como Grandes Premios (GP), que se disputan en circuitos de todo el mundo. En 2025, por ejemplo, el calendario incluirá 24 carreras, dependiendo de las decisiones de la FIA (Federación Internacional del Automóvil).
- **Duración:** Las carreras se distribuyen a lo largo del año, comenzando típicamente en marzo (como el GP de Australia) y terminando en noviembre o diciembre (como el GP de Abu Dhabi).

- **Ubicaciones:** Los circuitos varían entre pistas permanentes (como Monza o Silverstone) y circuitos callejeros (como Mónaco o Singapur), ofreciendo diversidad geográfica y desafíos técnicos.

Formato de cada Gran Premio:

Cada evento de Gran Premio sigue un formato estandarizado, pero puede sufrir algunas variaciones. De todas maneras, un Gran Premio siempre se lleva a cabo desde el Viernes de la semana, hasta el día Domingo.

- **Formato Normal:**
  - **Viernes**: En este día se suelen llevar a cabo las Prácticas 1 y 2 (FP1 y FP2). Éstas son sesiones para que los equipos prueben los autos, ajusten configuraciones y recopilen datos. Tienen una duración de una hora (puede haber una excepción y que por tests de neumáticos para futuras temporadas, alguna de las dos dure una hora y media en total).
  - **Sábado:** En este día se lleva a cabo la Práctica 3 (FP3) y la Qualy (Clasificación). La FP3 tiene el mismo formato que las FP1 y FP2. La Clasificación por su parte define la parrilla de salida. Ésta última está dividida en 3 etapas (Q1, Q2 y Q3). La Q1 consta de 18 minutos, y define las últimas 5 posiciones de la parrilla de largada. Seguido de la Q2, la cual dura 15 minutos y determina las siguientes 5 posiciones, y termina en la Q3 donde se define la posición de largada de los primero 10 pilotos. El objetivo de estas sesiones es lograr hacer el tiempo más bajo posible para lograr una mejor posición de largada.
  - **Domingo:** el día de la Carrera principal. En ella los pilotos buscan terminar lo más arriba posible en un evento que suele durar entre una hora y media y dos horas.

- **Formato Sprint:** El formato sprint es un cambio a la estructura tradicional de un fin de semana de carrera que se introdujo hace unos pocos años en donde surgen algunas variaciones a lo normal.
    - **Viernes:** Consta de una Práctica Libre 1 (FP1) y una Qualy Sprint. La Qualy Sprint sigue el mismo formato que la Qualy normal, pero surge cambios en la duración de cada instancia (Q1, Q2 y Q3). En ésta, la duración de cada una es de 12, 10 y 8 minutos respectivamente. Esta Qualy, define la parrilla de larga de la Carrera Sprint.
    - **Sábado:** Consta de la Carrera Sprint y de la Qualy. La Carrera Sprint, es al igual que la carrera principal, una carrera, pero con la cualidad de que es más corta que ésa última. Esta carrera consta de una menor cantidad de vuelta que la mitad de la carrera normal y a su vez, entrega menos puntos. Luego de la Carrera Sprint, se sigue con la Qualy que tiene el mismo formato que en el formato normal.
    - **Domingo:** Al igual que en el otro formato de fin de semana, es el día de la carrera principal.

Sistema de puntuación

- **Carrera principal:** Los 10 primeros lugares reciben puntos dependiendo de la posición en la que terminan la carrera. Los puntos que se entregan son 25-18-15-12-10-8-6-4-2-1 respectivamente siendo el puntaje más alto para el piloto que termina en la primera posición y siguiendo en orden ascendente.
- **Carrera Sprint:** A diferencia de la Carrera principal, ésta entrega puntos únicamente a los primeros 8 pilotos y los puntos que se entregan son 8-7-6-5-4-3-2-1 respectivamente siendo el puntaje más alto para el piloto que termina en la primera posición y siguiendo en orden ascendente.

El objetivo de los pilotos tanto como de las escuderías es sumar la mayor cantidad de puntos durante toda la temporada para poder así consagrarse campeones de cada campeonato. Es importante tener en cuenta que, actualmente el deporte consta de 10 escuderías con 2 pilotos cada una. Cada piloto, suma su puntaje obtenido en carrera para el campeonato de pilotos como así también suma puntos para la escudería en el campeonato de equipos.

Reglamento Técnico y Deportivo

- **Autos:** Los monoplazas cumplen con regulaciones estrictas de la FIA (peso, dimensiones, aerodinámica, motor híbrido, etc.), que se actualizan anualmente y que pueden sufrir algunos cambios a medida que avanza la temporada.
- **Paradas en boxes:** Los equipos cambian neumáticos y ajustan estrategias durante la carrera, con una parada mínima obligatoria en la mayoría de los casos.
- **Sanciones:** Los comisarios imponen penalizaciones (tiempo, posiciones o descalificación) por infracciones como adelantamientos ilegales o exceso de límites de pista.

En resumen, la temporada de Fórmula 1 es una serie de Grandes Premios con distintos formatos, donde pilotos y equipos compiten por puntos bajo un reglamento estricto, buscando los títulos de Pilotos y Constructores a lo largo del año.

Opciones similares en el mercado

Aunque el objetivo de Predi —integrar información detallada de la Fórmula 1, un sistema de pronósticos por evento y una dimensión social en una sola plataforma— no tiene un equivalente directo en el mercado actual, existen herramientas y servicios que abordan algunos de estos aspectos de manera parcial. Este análisis examina las opciones más relevantes, destacando sus fortalezas, debilidades y cómo dejan un nicho sin explotar que Predi busca llenar. A continuación, voy a mencionar algunas de las plataformas más representativas:

- **F1.com y F1 TV:** El sitio oficial de la Fórmula 1 (F1.com) y su servicio de streaming, F1 TV, son las principales fuentes de información para los aficionados. Ofrecen estadísticas en tiempo real, resultados históricos y transmisiones en vivo, respaldados por datos oficiales de la FIA. Sin embargo, su enfoque es estrictamente informativo lo que tiene como ventaja mostrar datos oficiales y confiables, pero a su vez, al ser únicamente informativo, tiene la desventaja de tener una ausencia de interacción social. Además, la API de F1 TV no es pública, sino que es paga, lo que limita su integración con sistemas externos.
- **F1 Fantasy:** Desarrollado por PlayON bajo licencia oficial de la Fórmula 1, F1 Fantasy es un juego el cual permite a los usuarios crear equipos virtuales con pilotos y escuderías, acumulando puntos según su desempeño real a lo largo de la temporada. Sin embargo, su mecánica se centra en selecciones a largo plazo, no en predicciones específicas por carrera, clasificación o sprint, lo que reduce su granularidad y dinamismo. Además, carece de una integración profunda con datos en tiempo real o espacios para debates entre usuarios. Sumado a que, como mencioné recién, se trata de un juego (una app) el cuál de por sí puede no tener el mayor alcance potencial dado que en ciertos rangos etarios no se suelen utilizar.

- **ProdeGP:** es una plataforma menos conocida que intenta aplicar el concepto de “prode” a la Fórmula 1. Este sitio web permite a los usuarios predecir resultados de carreras, como el ganador o el podio, y competir en rankings. Sin embargo, su alcance es limitado: además de no contar con una audiencia significativa ni haberse consolidado como un referente entre los aficionados de la F1, su funcionalidad es reducida, ya que únicamente ofrece la posibilidad de realizar pronósticos y consultar cronogramas de eventos, sin incorporar herramientas adicionales de interacción, análisis o acceso a información detallada sobre pilotos, circuitos o resultados históricos.

El análisis previo demuestra que, aunque plataformas como F1 TV, F1 Fantasy o ProdeGP abordan aspectos individuales del dominio de Predi pero ninguna logra integrar estos elementos en una experiencia cohesiva y específica para la Fórmula 1. Este vacío en el mercado no solo resalta la oportunidad para una solución innovadora, sino que también plantea desafíos técnicos significativos. Para lograr todo esto, es fundamental explorar las tecnologías disponibles que podrían ayudar a alzar desde 0 un proyecto de esta índole, desde lenguajes de programación hasta arquitecturas de sistemas y mecanismos de seguridad.

Tecnologías Disponibles

Para el desarrollo de Predi, un proyecto de software centrado en la interacción entre usuarios y diseñado para gestionar un elevado volumen de información, fue necesario llevar a cabo un estudio exhaustivo sobre diversas tecnologías y herramientas que garantizaran la eficacia en el logro de los objetivos planteados. Desde el inicio se identificó la importancia de contar con un ecosistema tecnológico robusto que permitiera gestionar adecuadamente grandes volúmenes de datos, favorecer la interacción entre usuarios y asegurar despliegues ágiles, además de cumplir criterios de seguridad y escalabilidad. A continuación, se describen diferentes categorías tecnológicas relevantes para el proyecto, indicando sus principales características, ventajas y desafíos relacionados con su implementación.

Lenguajes de Programación

Una plataforma como Predi, diseñada para proporcionar información actualizada sobre Fórmula 1 y ofrecer funcionalidades relacionadas con pronósticos deportivos y creación de comunidad, demanda lenguajes de programación que equilibren rendimiento, facilidad de desarrollo y compatibilidad con aplicaciones web modernas. Por esta razón, se investigaron distintas alternativas tecnológicas disponibles, contemplando tanto su idoneidad técnica como la experiencia previa del equipo desarrollador. A partir de dicho análisis, se identificaron las opciones más adecuadas para backend y frontend, resaltando aspectos fundamentales como eficiencia, flexibilidad y facilidad de uso.

Backend

El backend constituye la capa lógica y estructural de una aplicación web, encargándose del procesamiento de datos, gestión de bases de datos, seguridad y exposición de APIs que permiten la comunicación con el frontend. Por ello, resultó fundamental seleccionar tecnologías

que ofrezcan equilibrio entre rendimiento, escalabilidad, facilidad de desarrollo y mantenimiento. A partir de una evaluación técnica y considerando la experiencia previa del equipo desarrollador, se analizaron los siguientes lenguajes de programación:

- **Golang (Go):** Lenguaje de programación desarrollado por Google, caracterizado por su simplicidad y alto rendimiento en aplicaciones web y APIs. Su diseño orientado a concurrencia y la compilación directa a código máquina permiten una ejecución rápida y eficiente en situaciones de alta demanda. Go cuenta con frameworks como Gin y ORMs como GORM, facilitando desarrollos seguros y rápidos. Dispone de un recolector de basura eficiente que reduce considerablemente la latencia y maximiza el desempeño. Además, posee una sintaxis clara, tipado estático y ofrece gran compatibilidad con múltiples sistemas operativos, simplificando el despliegue y garantizando escalabilidad horizontal en aplicaciones con elevado tráfico concurrente. Go incorpora un modelo de concurrencia basado en goroutines, que son funciones ligeras ejecutadas de manera concurrente y gestionadas por el runtime del lenguaje, permitiendo manejar múltiples tareas simultáneamente con bajo consumo de recursos. Finalmente, Go enfatiza la seguridad y prevención de errores desde su diseño.
- **Python:** Lenguaje reconocido por su versatilidad, sintaxis clara y amplia comunidad de usuarios. Python es adecuado para la creación ágil de backends mediante frameworks populares como Django o Flask, y cuenta con una extensa biblioteca estándar que facilita la integración con bases de datos y otras tareas frecuentes. Sin embargo, al tratarse de un lenguaje interpretado, presenta menor rendimiento en comparación con Go o Java, lo cual podría afectar aplicaciones críticas en tiempo real o con alta demanda de recursos. La gestión automática de memoria, si bien simplifica el desarrollo, limita el control preciso en aplicaciones altamente exigentes en términos de recursos.

- **Java:** Lenguaje consolidado ampliamente utilizado en backends robustos y escalables, particularmente mediante frameworks como Spring Boot. Su paradigma orientado a objetos y la ejecución en la Máquina Virtual de Java (JVM) aseguran estabilidad, portabilidad y compatibilidad con diversos sistemas operativos. Java, aunque más verboso en comparación con alternativas como Python o Go, prioriza la mantenibilidad a largo plazo mediante tipado estático y detección temprana de errores durante la compilación. Además, incorpora administración automática de memoria mediante recolector de basura y soporte nativo para multihilos, facilitando la concurrencia y ejecución simultánea de múltiples procesos.

Frontend

El frontend representa la capa visual y de interacción directa con los usuarios en una aplicación web, encargado de presentar la información mediante elementos gráficos como textos, botones, tablas y formularios. En una plataforma como Predi, orientada a mostrar resultados y datos relacionados con la Fórmula 1, el frontend es clave para traducir la información procesada por el backend en contenidos claros y visualmente atractivos.

Existen diversas alternativas tecnológicas para construir el frontend, cuya elección depende de la complejidad del proyecto y la experiencia del desarrollador. Entre ellas destacan:

- **HTML, CSS y JavaScript puro:** Constituyen la base tradicional del desarrollo web. HTML estructura el contenido, CSS define su estilo visual y JavaScript añade interactividad básica. Sin embargo, depender exclusivamente de estas tecnologías puede resultar insuficiente para aplicaciones con interacciones avanzadas.

- **Frameworks y bibliotecas:** Herramientas como React, Vue.js y Angular simplifican el desarrollo del frontend mediante componentes reutilizables y estructuras organizadas, permitiendo crear interfaces interactivas eficientes y escalables. Su uso se recomienda especialmente en proyectos con funcionalidades dinámicas o que requieren mantener coherencia visual en múltiples vistas.
- **Generadores estáticos:** Tecnologías como Hugo o Gatsby permiten crear páginas pre-renderizadas, adecuadas para sitios con contenido estático que no requiere actualizaciones frecuentes.

La elección de un framework basado en JavaScript como React o Vue.js resulta apropiada debido a la organización modular del código en componentes reutilizables y a su soporte de interacciones complejas sin necesidad de recargas completas de página. En particular, React utiliza un DOM virtual, una representación en memoria que optimiza el rendimiento al actualizar únicamente los elementos necesarios, acelerando así la experiencia del usuario.

Entre los principales elementos y herramientas consideradas para el desarrollo frontend destacan:

HTML: Define la estructura fundamental del contenido web. Es esencial pero insuficiente por sí solo para ofrecer interactividad avanzada.

CSS: Controla el estilo visual del contenido definido por HTML, incluyendo aspectos como colores, fuentes, disposición y tamaños. Aunque imprescindible, puede resultar complejo gestionarlo exclusivamente en proyectos extensos.

Frameworks basados en JavaScript: Los frameworks son herramientas construidas sobre JavaScript que simplifican y aceleran el desarrollo del frontend al ofrecer estructuras predefinidas:

- **React**: Biblioteca JavaScript desarrollada por Facebook que utiliza el DOM virtual para mejorar la eficiencia y rendimiento. React favorece la reutilización del código mediante componentes, facilitando el mantenimiento y escalabilidad de la aplicación.
- **Vue.js:** Framework ligero basado en JavaScript, reconocido por su facilidad de aprendizaje y escalabilidad gradual desde aplicaciones simples hasta interfaces complejas.
- **Angular**: Framework completo desarrollado por Google, basado en TypeScript, que ofrece una solución estructurada y tipada para aplicaciones web con múltiples funcionalidades integradas.

Bases de Datos

Las bases de datos constituyen sistemas destinados al almacenamiento estructurado y eficiente de información, permitiendo su rápida consulta, actualización o eliminación. Existen fundamentalmente dos categorías principales de bases de datos consideradas para aplicaciones modernas:

- **Bases de datos Relacionales (SQL):** Usan tablas interrelacionadas para almacenar información estructurada. Ejemplos comunes son MySQL, PostgreSQL, SQLite y SQL Server. Se caracterizan por mantener la integridad y consistencia de los datos mediante esquemas rígidos y consultas complejas. Sin embargo, su estructura fija dificulta adaptaciones rápidas ante cambios frecuentes.
- **Bases de datos No Relacionales (NoSQL):** Permiten almacenamiento flexible de información en formatos menos estructurados, como documentos JSON, siendo especialmente útiles para aplicaciones que requieren escalabilidad rápida o manejan datos variables. Ejemplos comunes incluyen MongoDB, Firebase, Cassandra y Redis. Su

ventaja principal es la adaptabilidad ante cambios frecuentes, aunque sacrifican parcialmente la consistencia estricta típica de las bases relacionales.

La comunicación del backend con la base de datos ocurre mediante consultas SQL en bases relacionales o mediante herramientas como los ORM (Object-Relational Mapping), que facilitan la interacción traduciendo automáticamente código en consultas SQL. El uso de ORM permite simplificar significativamente el manejo de la información, acelerando el desarrollo y manteniendo claridad en el código.

Finalmente, las bases de datos pueden alojarse de diversas maneras:

- **Localmente:** Instalada directamente en el equipo de desarrollo, facilitando pruebas, accesibles y seguras.
- **Servidores dedicados:** Computadoras permanentemente conectadas a internet que alojan la base de datos, permitiendo acceso remoto.
- **Contenedores (Docker):** Empaquetan la base de datos junto con todo su entorno, simplificando despliegues y facilitando escalabilidad.
- **Servicios en la nube:** Proveedores como Amazon RDS, Google Cloud SQL o supabase administran aspectos como escalabilidad, seguridad y respaldo automático, resultando ideales para aplicaciones con gran cantidad de usuarios.

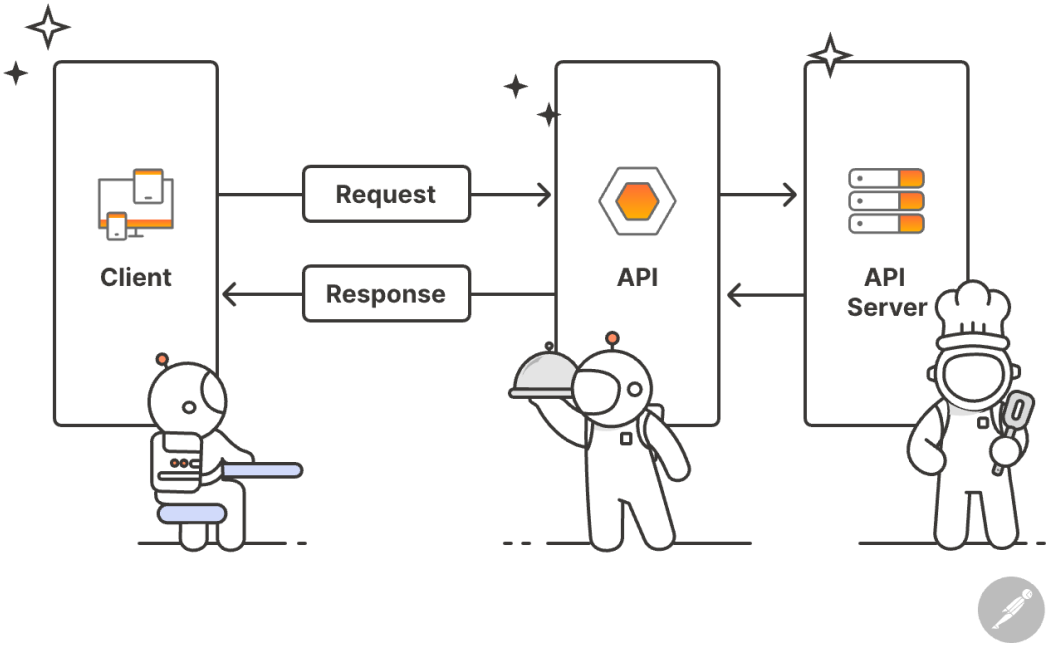
Comunicación entre Frontend y Backend (APIs)

En el desarrollo de aplicaciones web, la comunicación efectiva entre frontend y backend resulta fundamental para garantizar el intercambio fluido y ordenado de información. Esta interacción se realiza principalmente mediante Interfaces de Programación de Aplicaciones (APIs, por sus siglas en inglés), que establecen los protocolos y formatos con los cuales ambos componentes intercambian solicitudes y respuestas.

Una API puede definirse como un conjunto estructurado de reglas y métodos que permite a diferentes sistemas comunicarse entre sí, sin necesidad de conocer los detalles internos de cada uno. En este contexto, la API actúa como intermediario, posibilitando que el frontend solicite servicios específicos al backend, tales como obtener información, almacenar datos o ejecutar acciones concretas.

El proceso general de interacción mediante API sigue estos pasos:

1. **Solicitud del frontend:** El frontend genera una petición hacia el backend solicitando información o ejecución de alguna acción específica (por ejemplo, obtener detalles de un producto).
2. **Recepción y procesamiento por parte del backend:** La API interpreta la solicitud recibida, transfiriéndola al backend, donde se procesa la petición mediante lógica específica o consultas a la base de datos.
3. **Respuesta al frontend**: Una vez procesada la solicitud, la API formatea y envía la respuesta nuevamente hacia el frontend en un formato accesible y claro (habitualmente JSON).
4. **Presentación visual:** El frontend recibe dicha respuesta y la utiliza para actualizar la interfaz mostrada al usuario.



Tipos de APIs: Un enfoque en REST

Dentro del ámbito web, las APIs tipo REST (Representational State Transfer) son predominantes, debido a su simplicidad, flexibilidad y escalabilidad. REST emplea métodos HTTP específicos para expresar distintas operaciones:

- **GET:** Obtener información
- **POST:** Guardar nueva información
- **PUT:** Modificar información existente
- **DELETE:** Eliminar información

Cada recurso en REST se identifica con una URL específica (endpoint), permitiendo organizar claramente las acciones disponibles dentro del sistema. Asimismo, utiliza formatos de datos legibles y estructurados, principalmente JSON, para facilitar tanto el intercambio como la interpretación de la información.

Entre las ventajas más destacadas del uso de APIs REST figuran:

- **Interoperabilidad:** Facilitan la comunicación entre diferentes plataformas o sistemas.
- **Modularidad**: Permiten desarrollar y modificar componentes del sistema de forma independiente.
- **Reutilización:** Pueden aprovecharse para múltiples aplicaciones, incluyendo versiones web y móviles.
- **Escalabilidad:** Posibilitan la incorporación ágil de nuevos usuarios o funcionalidades.

Por otra parte, su uso también presenta ciertos desafíos:

- **Latencia:** El tiempo adicional que requiere cada solicitud y respuesta, aunque generalmente breve.
- **Seguridad:** Requieren mecanismos adicionales como cifrado o autenticación para evitar posibles vulnerabilidades.
- **Mantenimiento:** Exigen actualizaciones coordinadas si cambian las especificaciones.
- **Gestión de errores:** Deben contemplarse posibles fallos en las respuestas del backend o errores en las solicitudes.

En resumen, las APIs constituyen el núcleo que hace posible la colaboración efectiva entre frontend y backend, garantizando aplicaciones web funcionales, dinámicas y escalables.

Frameworks y Bibliotecas

Para optimizar el desarrollo de aplicaciones web y evitar la redundancia en la programación de tareas comunes, los desarrolladores suelen recurrir al uso de frameworks y

bibliotecas. Estas herramientas proporcionan estructuras o funcionalidades previamente establecidas, reduciendo considerablemente el esfuerzo necesario en la implementación de soluciones recurrentes.

Un framework representa un conjunto organizado y predefinido de código, que ofrece una estructura base para desarrollar aplicaciones completas. El uso de un framework implica seguir determinadas convenciones y reglas establecidas por su diseño, lo que facilita el desarrollo al proporcionar soluciones estándar a problemas comunes, como gestión de rutas, bases de datos o presentación visual de datos.

Por otro lado, una biblioteca es una colección de funciones específicas que pueden integrarse puntualmente al código según la necesidad particular de cada proyecto. A diferencia del framework, las bibliotecas no imponen una estructura rígida, otorgando al desarrollador mayor libertad y flexibilidad respecto a cuándo y cómo utilizarlas.

La principal diferencia entre ambas radica en el grado de control: mientras que en un framework el control lo tiene el propio framework (denominado "inversión de control"), en una biblioteca el desarrollador mantiene la autonomía sobre el flujo del programa.

Las razones principales por las que se emplean frameworks y bibliotecas en el desarrollo de software son:

- **Reducción del tiempo de desarrollo:** Evitan la repetición de tareas básicas, permitiendo centrarse en aspectos particulares del proyecto.
- **Minimización de errores: Al reutilizar código ampliamente probado, se disminuyen las probabilidades de fallos.**
- **Estandarización del código: Facilitan la comprensión y colaboración dentro de equipos de trabajo.**
- **Mantenimiento simplificado:** Una estructura clara y organizada facilita futuras modificaciones o actualizaciones.

Ejemplos concretos de frameworks y bibliotecas incluyen:

- En el Frontend:
    - **React:** Biblioteca basada en componentes reutilizables que utiliza el DOM virtual para optimizar la eficiencia en actualizaciones visuales.
    - **Vue.js:** Framework ligero que permite escalabilidad progresiva, desde aplicaciones simples hasta complejas.
    - **Angular:** Framework completo desarrollado en TypeScript, que ofrece soluciones integradas para la construcción de interfaces robustas.
- En el backend:
    - **Django (Python):** Framework estructurado que define claramente cómo gestionar bases de datos, usuarios y rutas, entre otros.
    - **Gin (Golang):** Framework que agiliza el desarrollo de APIs mediante convenciones claras y eficaces.

Entre las principales ventajas de usar estas herramientas destacan:

- **Rapidez y eficiencia:** Permiten centrarse en la lógica específica del proyecto.
- **Organización clara del código:** Facilitan el mantenimiento y la escalabilidad.
- **Amplia comunidad:** Existen recursos documentales y foros de ayuda.
- **Escalabilidad:** Facilitan la incorporación ágil de nuevos usuarios o funcionalidades.

Sin embargo, presentan también algunos desafíos potenciales:

- **Curva de aprendizaje:** Requieren tiempo inicial para aprender las convenciones propias.

- **Dependencia tecnológica:** El desarrollo queda supeditado a futuras actualizaciones o soporte del framework.
- **Peso adicional:** Incorporan código que en ciertas circunstancias podría resultar innecesario.
- **Rigidez:** Podrían limitar el desarrollo en proyectos altamente personalizados.

En definitiva, el uso estratégico de frameworks y bibliotecas es fundamental para construir aplicaciones modernas de manera eficiente, escalable y confiable, siempre considerando cuidadosamente las necesidades concretas y características particulares de cada proyecto.

Arquitecturas de Sistemas

La construcción de una aplicación web requiere definir una arquitectura que organice y articule todos los componentes seleccionados previamente, como lenguajes, frameworks y bases de datos. Una arquitectura de sistemas constituye una estructura lógica que determina la distribución de tareas, la comunicación entre módulos y el flujo de información, favoreciendo la eficiencia y escalabilidad del software.

Entre los tipos principales de arquitecturas se destacan:

- **Arquitectura Monolítica:** Agrupa frontend, backend y base de datos en una única unidad integrada. El código se ejecuta en un solo servidor, simplificando el desarrollo inicial. Sin embargo, presenta limitaciones significativas para escalar o mantener aplicaciones complejas.
- **Arquitectura Distribuida:** Divide la aplicación en componentes independientes o servicios que interactúan mediante APIs. Cada componente puede desplegarse de forma individual, ofreciendo mayor resistencia a fallos y facilidad para escalar recursos específicos.

- **Arquitectura en Capas (N-tier):** Organiza el software en capas horizontales con responsabilidades definidas: presentación (interfaz de usuario), lógica de negocio (reglas y validaciones) y acceso a datos (interacción con bases de datos). Este modelo puede aplicarse tanto en sistemas monolíticos como distribuidos.

Dentro de estas arquitecturas, existen patrones y estilos arquitectónicos específicos para resolver problemas recurrentes en el desarrollo de software:

- **Modelo-Vista-Controlador (MVC):** Divide claramente responsabilidades en tres componentes principales: el modelo (manejo de datos y lógica), la vista (interfaz que observa el usuario) y el controlador (maneja la comunicación entre modelo y vista). Este patrón facilita la organización del código, especialmente en aplicaciones monolíticas.
- **Microservicios:** Estilo de arquitectura distribuida en el que se crean múltiples servicios pequeños y especializados, independientes entre sí. Cada microservicio puede utilizar tecnologías diferentes, escalarse individualmente y comunicarse mediante APIs. Aporta flexibilidad y resistencia, aunque implica mayor complejidad operativa.

La elección de una arquitectura adecuada tiene un impacto directo sobre la facilidad de mantenimiento, la capacidad de escalado y la calidad general de la aplicación, facilitando su evolución frente al crecimiento en número de usuarios o funcionalidades.

Alojamiento y Despliegue

Un Una vez desarrollada la aplicación, es necesario alojarla en un entorno accesible y ejecutar un proceso de despliegue que permita su disponibilidad para los usuarios finales. El alojamiento proporciona el espacio físico o virtual necesario para que el software esté operativo continuamente, mientras que el despliegue representa la secuencia de acciones que trasladan y activan el código en ese entorno.

Las opciones más comunes para el alojamiento de aplicaciones son:

- Alojamiento local: Consiste en ejecutar la aplicación en equipos propios. Proporciona control absoluto y resulta apropiado únicamente para entornos de desarrollo o pruebas, debido a limitaciones en seguridad, disponibilidad y escalabilidad.
- Servidores Físicos: Máquinas físicas dedicadas o compartidas, generalmente alojadas en centros de datos. Ofrecen alto rendimiento y control, pero requieren mayor inversión inicial y esfuerzo de mantenimiento.
- Alojamiento en la Nube: Proveedores como Amazon Web Services, Google Cloud o Azure ofrecen servidores virtuales bajo demanda. Presentan alta escalabilidad, flexibilidad y pago por uso, aunque implican costos variables y dependencia de terceros.
- Contenedores (Docker): Tecnologías que encapsulan la aplicación junto a todas sus dependencias en contenedores portátiles y autónomos. Facilitan despliegues rápidos, escalabilidad y homogeneidad en múltiples entornos, aunque requieren configuraciones iniciales más complejas.
- Plataformas como Servicio (PaaS): Servicios como Heroku o Vercel automatizan tanto el alojamiento como el despliegue, simplificando considerablemente la

gestión operativa. Son especialmente útiles para desarrolladores con menos experiencia en administración de infraestructura, aunque presentan limitaciones en personalización y mayores costos en aplicaciones de gran escala.

El proceso general de despliegue incluye las siguientes etapas:

1. **Preparación del Código:** Compilación o empaquetado del software junto con dependencias y configuraciones necesarias.
2. **Subida al servidor:** Transferencia de archivos mediante herramientas como SSH, FTP o sistemas de control de versiones como Git.
3. **Configuración:** Instalación de componentes adicionales, configuración del entorno operativo, y definición de parámetros específicos (variables, accesos, puertos).
4. **Ejecución:** Inicio de la aplicación en el servidor para que esta responda a solicitudes externas
5. **Pruebas y monitoreo:** Verificación inicial y supervisión continua para asegurar estabilidad y rendimiento adecuado.

Además, para facilitar el acceso del usuario, la aplicación requiere una dirección web amigable denominada dominio, asociada a una dirección IP específica mediante el sistema DNS (Domain Name System). El DNS actúa como traductor, permitiendo a los usuarios acceder fácilmente sin manejar detalles técnicos.

Cada opción de alojamiento presenta ventajas y desafíos particulares, cuyo análisis cuidadoso condiciona aspectos fundamentales como costo, rendimiento, escalabilidad y facilidad operativa del sistema desplegado. En definitiva, seleccionar adecuadamente tanto el alojamiento como el método de despliegue resulta determinante para garantizar la accesibilidad, estabilidad y crecimiento sostenible de cualquier aplicación web.

Contenedores y Virtualización

La implementación efectiva de aplicaciones web requiere garantizar consistencia en distintos entornos operativos. Para este propósito, se utilizan técnicas como la virtualización y los contenedores, que ofrecen entornos aislados para ejecutar aplicaciones de manera homogénea y eficiente.

La virtualización consiste en crear entornos virtuales completos dentro de una máquina física. Un software llamado hipervisor administra los recursos de hardware (CPU, memoria, almacenamiento), permitiendo ejecutar múltiples máquinas virtuales (VMs), cada una con su propio sistema operativo independiente. Aunque las VMs ofrecen un aislamiento robusto, son relativamente pesadas y lentas en su inicialización.

Los contenedores, por otra parte, proporcionan un método más ligero de aislamiento. A diferencia de las VMs, los contenedores no incluyen un sistema operativo completo, sino que utilizan directamente el núcleo (kernel) del sistema anfitrión. Herramientas como Docker permiten empaquetar una aplicación junto con sus dependencias en un entorno autocontenido, facilitando el despliegue y garantizando que la aplicación se ejecute de manera uniforme en diferentes infraestructuras.

Docker

La herramienta principal para gestionar contenedores es Docker, que utiliza un archivo llamado Dockerfile para especificar instrucciones que construyen imágenes contenedoras. Estas imágenes pueden ejecutarse en cualquier entorno que disponga de Docker instalado. Docker emplea mecanismos como namespaces y grupos de control (control groups) para asegurar el aislamiento de procesos y limitar el consumo de recursos.

Ventajas del uso de Docker:

- Consistencia y portabilidad entre distintos entornos.

- Inicio rápido y eficiente en términos de recursos.
- Facilita el despliegue y escalabilidad mediante contenedores reproducibles.

Desafíos asociados a Docker:

- Requiere aprendizaje específico para escribir Dockerfiles y utilizar comandos.
- Depende de compatibilidad del sistema anfitrión (principalmente Linux o adaptaciones en Windows y Mac).
- Menor nivel de aislamiento en comparación con máquinas virtuales, aumentando riesgos de seguridad.

En síntesis, los contenedores simplifican considerablemente el proceso de desarrollo, prueba y despliegue de aplicaciones, resolviendo problemas comunes relacionados con la variabilidad de entornos y facilitando arquitecturas distribuidas.

Seguridad Informática

Al desplegar una aplicación web, esta queda expuesta a múltiples riesgos asociados con accesos no autorizados, ataques informáticos y pérdida accidental de información. La seguridad informática comprende técnicas y herramientas destinadas a proteger datos, usuarios y sistemas ante dichas amenazas.

Entre los riesgos más comunes se destacan:

- **Acceso no autorizado:** ingreso indebido a sistemas protegidos.
- **Inyección SQL:** introducción de código malicioso en consultas a bases de datos.
- **Cross-Site Scripting (XSS):** ejecución de scripts maliciosos en el navegador del usuario
- **Ataques Man-in-the-Middle (MITM):** interceptación de datos en tránsito entre cliente y servidor.
- **Pérdida o corrupción de datos:** debido a fallos técnicos o incidentes de seguridad.

Para contrarrestar estos riesgos, se aplican mecanismos básicos como:

- **Cifrado de comunicaciones (HTTPS con TLS/SSL):** garantiza la confidencialidad e integridad de los datos en tránsito.
- **Autenticación y autorización:** verifican la identidad del usuario y definen niveles de acceso permitidos.
- **Validación y sanitización de entradas:** evitan la introducción de datos maliciosos.
- **Respaldos regulares:** permiten recuperación rápida ante pérdida de información.

Uno de los métodos más utilizados actualmente para la autenticación segura es el uso de JSON Web Tokens (JWT). Un JWT está compuesto por tres secciones codificadas en Base64: encabezado (tipo y algoritmo), payload (información del usuario) y firma (para asegurar integridad). La autenticación mediante JWT se realiza mediante los siguientes pasos:

- El usuario envía credenciales al servidor.
- El servidor verifica credenciales y genera un JWT firmado.
- El token es enviado al cliente y utilizado en futuras solicitudes.
- El servidor valida la firma y vigencia del token para permitir acceso.

Debido a la corta duración de los JWT por razones de seguridad, frecuentemente se emplean Refresh Tokens, que permiten renovar tokens de acceso sin necesidad de autenticar nuevamente al usuario. Este método combina seguridad y comodidad para el usuario, aunque introduce complejidad adicional en su gestión y revocación.

La verificación eficiente de JWT y otros procesos de seguridad se maneja mediante componentes llamados middleware, funciones que interceptan solicitudes HTTP y realizan tareas como validar tokens, autorizaciones y prevenir accesos indebidos.

Otra consideración clave es la política de intercambio de recursos de origen cruzado (CORS, por sus siglas en inglés), mecanismo que controla el acceso a recursos desde dominios distintos al servidor principal. CORS define explícitamente qué solicitudes están permitidas mediante encabezados HTTP específicos, evitando así riesgos derivados de accesos no autorizados desde sitios externos.

Finalmente, la protección integral de aplicaciones también debe considerar:

- Uso obligatorio de HTTPS para proteger los datos en tránsito.
- Almacenamiento seguro de tokens mediante cookies protegidas con atributos como HttpOnly y Secure.

- Prevención de ataques comunes mediante consultas parametrizadas (SQL Injection), sanitización de entradas y Content Security Policy (XSS), y tokens anti-CSRF.
- Implementación de sistemas de monitoreo y registro para detectar y mitigar intentos de acceso no autorizados.

La adecuada implementación de medidas de seguridad informática asegura no solo la integridad y confidencialidad de los datos, sino también la confianza de los usuarios en la aplicación, fundamental en entornos digitales modernos.

Escalabilidad y Rendimiento

La construcción de una aplicación web requiere no solo lograr funcionalidad, sino también asegurar que pueda soportar un crecimiento significativo de usuarios manteniendo tiempos de respuesta adecuados. Para esto, se consideran dos aspectos fundamentales: la escalabilidad y el rendimiento.

La escalabilidad se refiere a la capacidad de una aplicación para gestionar incrementos sustanciales en la carga de trabajo, incluyendo usuarios concurrentes, solicitudes simultáneas y volumen de datos procesados. Existen dos enfoques principales de escalabilidad:

- Vertical: Aumentar los recursos de un servidor (más CPU, memoria). Es como agregar más caballos a un carro.
- Horizontal: Agregar más servidores para repartir la carga. Es como sumar más carros al equipo.

Por su parte, el rendimiento evalúa la rapidez y eficiencia con la que una aplicación responde a las solicitudes, directamente relacionado con la calidad percibida por los usuarios. Un rendimiento deficiente puede generar costos adicionales, degradación de la experiencia del usuario e incluso interrupciones en el servicio.

Estrategia para escalabilidad

Existen técnicas específicas destinadas a mejorar la escalabilidad, entre ellas:

- **Balanceo de carga:** Distribuye solicitudes entre múltiples servidores mediante un balanceador (load balancer) que dirige el tráfico según criterios específicos, optimizando el uso de recursos.
- **Escalado Horizontal con Contenedores:** Consiste en emplear herramientas como Docker para replicar la aplicación en múltiples instancias idénticas, facilitando así la gestión dinámica del tráfico según demanda.

- **Bases de datos escalables:** Utilizan métodos como réplicas de lectura y fragmentación (sharding) para distribuir consultas y almacenamiento de datos, mejorando el rendimiento y capacidad frente al aumento de la carga.

Estrategia para escalabilidad

Diversas técnicas permiten optimizar la rapidez y eficiencia en las respuestas:

- **Caching:** Almacena temporalmente datos frecuentemente solicitados en memoria rápida (e.g., Redis, Memcached), reduciendo consultas repetitivas a bases de datos.
- **Red de entrega de contenidos (CDN):** Distribuye copias de contenido estático en servidores ubicados estratégicamente cerca de los usuarios, disminuyendo significativamente la latencia.
- **Optimización del código:** Incluye mejoras en algoritmos, reducción de consultas innecesarias a bases de datos e implementación de índices, aumentando la eficiencia en la ejecución del código.

Reverse Proxy

El Reverse Proxy actúa como intermediario situado delante de los servidores de la aplicación, recibiendo solicitudes de clientes y gestionando su distribución y respuesta. Sus principales funciones incluyen balanceo de carga, almacenamiento de contenido estático en caché y filtrado de seguridad ante solicitudes sospechosas.

Ventajas de implementar un Reverse Proxy:

- Mejora la escalabilidad mediante distribución eficiente del tráfico.
- Optimiza el rendimiento mediante almacenamiento en caché de recursos frecuentes.

- Refuerza la seguridad al ocultar la estructura interna de la infraestructura.

Desafíos:

- Requiere configuración y mantenimiento cuidadosos.
- Constituye un punto único de fallo; si presenta errores, puede afectar toda la aplicación.

API Gateway

El API Gateway centraliza y optimiza el tráfico en aplicaciones con arquitecturas distribuidas o microservicios, actuando como punto de entrada único para todas las solicitudes de los clientes. Entre sus principales funciones se encuentran autenticación, enrutamiento inteligente de peticiones, limitación de tasas de solicitud (rate limiting) y transformación de formatos de datos.

Ventajas del API Gateway:

- Facilita la escalabilidad mediante gestión centralizada de las solicitudes entrantes.
- Mejora el rendimiento reduciendo tareas redundantes en múltiples servicios.
- Incrementa la seguridad aplicando validaciones y filtros sobre las solicitudes.

Desafíos:

- Introduce complejidad adicional en configuración y mantenimiento.
- Representa un componente crítico cuya falla impacta a todos los servicios dependientes.

En resumen, la escalabilidad y el rendimiento constituyen elementos esenciales para sostener el crecimiento y éxito de cualquier aplicación web. El uso adecuado de técnicas como balanceo de carga, caching, Reverse Proxy y API Gateway asegura una respuesta rápida y

continua, garantizando así la satisfacción de los usuarios y el aprovechamiento óptimo de los recursos tecnológicos.

Testing de Software

El testing de software constituye una fase esencial en el ciclo de vida de desarrollo, orientada a garantizar la calidad, la funcionalidad y la confiabilidad de las aplicaciones. Este proceso implica la ejecución sistemática de pruebas con el propósito de identificar defectos, verificar el cumplimiento de los requerimientos y asegurar que el sistema responda adecuadamente en distintos escenarios de uso. Existen diferentes niveles y tipos de pruebas que se aplican de manera complementaria:

- **Pruebas unitarias:** verifican de manera aislada el correcto funcionamiento de componentes o funciones específicas del sistema. Permiten detectar errores tempranos en la lógica interna del código y facilitan la mantenibilidad.
- **Pruebas de integración:** evalúan la interacción entre módulos o servicios, asegurando la correcta comunicación entre componentes del sistema, como por ejemplo la interoperabilidad entre backend, frontend y bases de datos.
- **Pruebas de sistema:** comprueban el comportamiento del software en su totalidad, evaluando tanto funcionalidades como aspectos no funcionales, incluyendo rendimiento y seguridad.
- **Pruebas de aceptación:** buscan confirmar que el producto cumple con las expectativas y requisitos definidos por el usuario o cliente final.
- **Pruebas beta:** consisten en la liberación controlada del sistema a un grupo limitado de usuarios reales, con el objetivo de recolectar retroalimentación en condiciones cercanas al uso productivo.

La planificación adecuada de estas pruebas permite no solo la detección de errores antes de la implementación definitiva, sino también la reducción de costos asociados a fallos en

producción. Asimismo, el uso de metodologías ágiles ha promovido la incorporación del testing continuo y la automatización de pruebas, lo cual posibilita una validación constante en cada etapa del desarrollo.

En síntesis, el testing de software constituye un mecanismo indispensable para asegurar la calidad del producto final, favoreciendo la confianza de los usuarios y la sostenibilidad del sistema a largo plazo.

Propuesta de solución

Luego de realizar un análisis de algunas las tecnologías disponibles, se seleccionaron aquellas más adecuadas en función de los alcances del sistema y la experiencia previa en su uso.

Backend

Para el desarrollo del backend de Predi se seleccionó Golang (Go) como lenguaje principal, debido a ventajas técnicas clave frente a alternativas evaluadas como Python y Java. En comparación con Python, Go ofrece un rendimiento notablemente superior gracias a su compilación directa a código máquina, lo que permite tiempos de respuesta reducidos en entornos con alta demanda concurrente, como el procesamiento de pronósticos en tiempo real. Frente a Java, mantiene la robustez y el tipado estático, pero con una sintaxis más concisa y un modelo de concurrencia más liviano, reduciendo tanto la complejidad de desarrollo como el consumo de recursos.

Su modelo de goroutines y canales facilita la gestión eficiente de múltiples solicitudes simultáneas con baja latencia, algo esencial para Predi durante fines de semana de carrera con picos de tráfico. La compatibilidad multiplataforma simplifica el despliegue en entornos de contenedores y servidores en la nube, y su ecosistema —incluyendo el framework Gin y el ORM GORM— agiliza la creación de APIs seguras, escalables y de alto rendimiento.

Estas características permitirán implementar de manera óptima funcionalidades como el manejo de pronósticos, la validación contra resultados reales, el cálculo de puntajes, la autenticación mediante JWT, la administración de grupos privados y la integración de datos actualizados de Fórmula 1, aprovechando la capacidad de Go para ejecutar operaciones concurrentes y trabajar con bases de datos relacionales y no relacionales.

Frontend

Para el desarrollo del frontend de Predi, se seleccionó React como biblioteca principal, complementada con Tailwind CSS para la gestión de estilos. Esta combinación permite construir una interfaz dinámica, modular y visualmente atractiva, optimizando los tiempos de desarrollo y garantizando una experiencia responsive y eficiente para los usuarios.

React, ampliamente adoptada en la industria, facilita la creación de interfaces basadas en componentes reutilizables. Esta modularidad contribuye a estructurar la plataforma mediante elementos independientes, como tablas de clasificación, formularios de pronósticos y secciones informativas, reduciendo la redundancia en el código y mejorando su mantenibilidad.

Tailwind CSS, por su parte, es un framework utility-first que proporciona clases predefinidas para aplicar estilos directamente en HTML o JSX, acelerando el proceso de diseño y asegurando una apariencia moderna y adaptable sin necesidad de escribir CSS personalizado desde cero.

La elección de estas tecnologías se fundamenta en las características y alcance del sistema, así como en la experiencia previa del equipo desarrollador, lo que garantiza un avance eficiente sin requerir tecnologías más complejas para cumplir con los objetivos funcionales establecidos.

Base de Datos

Para la gestión de datos en Predi, se optó por MySQL como sistema de base de datos relacional principal. Esta selección responde a la necesidad de mantener la integridad y consistencia de la información, aspectos esenciales para un sistema que gestiona pronósticos, resultados, puntajes y relaciones entre entidades como pilotos, eventos y grupos privados.

Asimismo, la experiencia previa con MySQL ha permitido aprovechar herramientas ORM, como GORM en combinación con Golang, para mapear objetos del backend a tablas relacionales, facilitando operaciones de consulta, inserción y actualización. La compatibilidad con índices en

MySQL contribuye a optimizar el rendimiento de las consultas, aspecto crítico para ofrecer respuestas ágiles al usuario en consultas de estadísticas y rankings.

Despliegue

Para el despliegue de Predi, tanto el frontend como el backend se alojarán en un servidor virtual (droplet) contratado en la plataforma DigitalOcean, con las siguientes especificaciones técnicas: 4 GB de RAM, 2 CPUs y 80 GB de almacenamiento SSD.

Ambos componentes fueron dockerizados para garantizar portabilidad, consistencia y facilidad en la gestión de entornos. El backend se organizará mediante microservicios independientes, cada uno ejecutándose en su propio contenedor, mientras que el frontend correrá en un contenedor separado. Dado que los proyectos de backend y frontend se encuentran en directorios distintos y no es posible vincularlos en un único archivo Docker Compose, se configuró una red Docker que permite la comunicación entre los contenedores.

En la máquina virtual se instaló y configuró un servidor Nginx, encargado de gestionar las solicitudes entrantes y direccionarlas adecuadamente al frontend o al backend, según corresponda. Docker se mantiene en ejecución permanente en el servidor, posibilitando el acceso constante a la plataforma.

Asimismo, se contrató un dominio que fue configurado para apuntar a la dirección IP del servidor virtual, asegurando así un acceso fácil y profesional a la aplicación.

Esta configuración garantiza un despliegue estable, escalable y alineado con las tecnologías seleccionadas, permitiendo gestionar eficientemente tanto la interfaz de usuario como la lógica de negocio y servicios asociados.

APIs: Implementación y Consumo de Datos

La comunicación entre el frontend y el backend de Predi, así como la integración de datos externos, se gestionará mediante APIs RESTful, garantizando una interacción eficiente,

escalable y predecible. Para la implementación, se empleará Gin en el backend desarrollado con Golang, complementado con un gateway que funcione como reverse proxy, y se consumirán datos oficiales de Fórmula 1 desde las APIs públicas de OpenF1.org.

Gin fue seleccionado por su capacidad para definir rutas claras, manejar solicitudes HTTP con enrutamiento optimizado y soportar middleware de forma nativa, facilitando la integración de funcionalidades esenciales como la verificación de tokens JWT y el registro de solicitudes para monitoreo. La experiencia previa con Golang y Gin asegura una configuración ágil de endpoints seguros y escalables, integrados con la base de datos MySQL mediante GORM para operaciones de almacenamiento y consulta de predicciones y resultados. Esto permite ofrecer respuestas rápidas incluso bajo alta concurrencia durante eventos de Fórmula 1.

El gateway que actúa como reverse proxy será el punto de entrada único para todas las solicitudes del frontend, dirigiendo el tráfico a los servicios correspondientes. Este componente permite balancear la carga entre múltiples instancias del backend, garantiza mayor seguridad al ocultar la estructura interna y filtra solicitudes maliciosas, además de centralizar tareas comunes como la compresión de respuestas y la gestión de CORS, facilitando la comunicación sin restricciones de origen cruzado. Aunque su configuración implica cierta complejidad inicial, su implementación es fundamental para preparar la plataforma ante una adopción masiva.

Para complementar la plataforma con información oficial y actualizada, se consumirá desde el backend en Golang la API pública de OpenF1.org, que provee datos en tiempo real y retrospectivos sobre pilotos, sesiones y resultados de Fórmula 1. Las solicitudes GET se realizarán utilizando la biblioteca estándar net/http o paquetes como http.Client, procesando los datos JSON recibidos y almacenándolos en la base de datos MySQL según sea necesario, asegurando la disponibilidad continua de la información aun en caso de interrupciones temporales en la API externa.

Alcance Funcional

El alcance funcional de Predi establece los límites y requerimientos del sistema, definiendo las funcionalidades que serán desarrolladas en esta versión inicial y los aspectos que quedarán excluidos, con el propósito de cumplir los objetivos globales y específicos previamente delineados.

Requerimientos

A partir de un profundo análisis del alcance del dominio de Predi, se logró identificar una serie de requerimientos funcionales y no funcionales. A continuación, presentaré estos en detalle:

- **Requerimientos Funcionales:**
    - **El sistema debe permitir el registro de usuarios a través de un formulario que solicite información básica como nombre, apellido, número de teléfono, correo electrónico y solicitar crear una contraseña.**
    - **Se debe implementar validaciones de seguridad en el formulario de registro, incluyendo verificación de la complejidad de la contraseña y la validación de formatos de correo electrónico y número de teléfono.**
    - **Permitir a los usuarios iniciar sesión utilizando correo electrónico.**
    - **Permitir el acceso completo a todas las funcionalidades del sistema a los usuarios registrados**
    - **Facilitar la gestión de perfil, donde los usuarios puedan actualizar su información personal y cambiar contraseña.**
    - **Permitir a los usuarios invitados visualizar información general de la F1 sin necesidad de registro restringiéndolos únicamente a esa acción**

- **Permitir a los administradores gestionar usuarios.**
- **Permitir a los administradores añadir, eliminar, editar información sobre sesiones pasadas/futuras.**
- **Permitir a los administradores gestionar los resultados ya sea a través de APIs externas y a su vez manualmente.**
- **Permitir a los usuarios realizar pronósticos para las distintas sesiones.**
- **Proporcionar especificaciones claras en cada formulario sobre cómo se asignan los puntos por predicciones.**
- **Calcular y actualizar el puntaje de los usuarios basándose en la precisión de sus pronósticos en comparación con los resultados reales de los eventos.**
- **Permitir a los usuarios visualizar su posición en la tabla de clasificación mundial en cualquier momento.**
- **Agrupar las sesiones según el gran premio al que pertenezcan y permitir diferenciar cada sesión específica.**
- **Mostrar los resultados de cada una de las sesiones pasadas**
- **Integrarse con APIs externas que brinden datos e información válida y oficial sobre los resultados de las sesiones tanto así como de las sesiones en sí.**
- **Presentar las distintas sesiones agrupadas por gran premio disputado de manera ordenada junto con información pertinente de horarios, locación, etc.**
- **Permitir visualizar las distintas sesiones, tanto pasadas como aquellas a disputar a futuro.**
- **Permitir a los usuarios crear grupos privados.**

- **Permitir a un usuario "administrador" del grupo aceptar o rechazar solicitudes de usuarios para unirse al grupo que administran.**
- **Mostrar tablas de clasificación internas para cada grupo privado, ordenando los miembros del grupo según sus puntajes en los pronósticos.**
- **Permitir a los usuarios interactuar a través de un foro.**

- **Requerimientos No Funcionales:**
  - **Organizar la información de manera que sea fácilmente accesible y comprensible, especialmente para usuarios que están haciendo pronósticos.**
  - **Implementar medidas de seguridad robustas para proteger la información personal y de uso de todos los usuarios, con especial atención en la protección contra accesos no autorizados y ataques de seguridad.**
  - **Diseñar una interfaz clara y fácil de usar para todos los tipos de usuarios, con adaptaciones específicas que faciliten la navegación y el acceso a las diferentes funcionalidades según el tipo de usuario.**
  - **El sistema debe ser capaz de soportar hasta al menos 1.000 usuarios concurrentes sin degradación significativa del rendimiento.**
  - **El sistema debe garantizar una disponibilidad del 99.5% durante los períodos de alta actividad, excluyendo mantenimientos programados.**
  - **La plataforma debe ser compatible con los navegadores web más utilizados y funcionar en sistemas operativos comunes a través de una interfaz responsive.**

Diseño

El diseño de Predi establece la estructura técnica y funcional del sistema, definiendo cómo se organizarán los componentes para cumplir con los requerimientos identificados en el alcance funcional. Esta sección abarca las interfaces de usuario (pantallas), la arquitectura general del software, los diagramas UML que representan las interacciones y entidades clave, y los patrones de diseño aplicados tanto en el backend como en el frontend. El enfoque adoptado prioriza la modularidad, la escalabilidad y la claridad, alineándose con las tecnologías seleccionadas (Golang con Gin, React, MySQL) y las necesidades de una plataforma interactiva para fanáticos de la Fórmula 1.

Predi contará con múltiples pantallas diseñadas para satisfacer la mayoría de los requerimientos funcionales establecidos, incluyendo interfaces para el registro e inicio de sesión, la gestión de pronósticos, la visualización de tablas de clasificación (globales y por grupo), la creación y administración de grupos privados, y la consulta de información sobre sesiones y resultados de Fórmula 1.

La arquitectura de Predi se basa en un enfoque de microservicios, implementado para dividir las funcionalidades del sistema en módulos independientes que operan de manera autónoma pero coordinada. Este diseño se eligió por su capacidad para escalar componentes específicos según la demanda, facilitar el mantenimiento individual de cada módulo y permitir una evolución flexible del sistema en el futuro. Los microservicios identificados son:

- **Microservicio de Usuarios:** Gestiona registro, autenticación, perfiles, administración de cuentas y manejo de usuarios por parte de los administradores.
- **Microservicio de Pilotos:** Maneja la información de pilotos, obtenida principalmente de APIs externas como OpenF1.org.
- **Microservicio de Resultados:** Administra los resultados oficiales de sesiones, integrándolos desde APIs externas o mediante ingresos manuales por administradores.

- **Microservicio de Sesiones:** Controla los datos de las sesiones de Fórmula 1, incluyendo horarios y detalles de Grandes Premios.
- **Microservicio de Pronósticos:** Permite a los usuarios ingresar y consultar pronósticos específicos para las sesiones de carrera y clasificación, calculando puntajes asociados.
- **Microservicio de Grupos:** Facilita la creación, gestión y clasificación interna de grupos privados.
- **Microservicio de Posteos:** Permite a los usuarios interactuar a través de posts y comentarios en el foro interactivo.

Un componente clave es el gateway desarrollado en Golang. Entre sus funciones destacan:

- **Validación de JWT:** Autenticación de solicitudes mediante verificación y generación de tokens JWT durante el inicio de sesión.
- **Reverse Proxy:** Redirección de solicitudes al microservicio correspondiente según la ruta solicitada.
- **Control de CORS:** Gestión de políticas de Cross-Origin Resource Sharing para permitir la interacción fluida entre el frontend y el backend.

El frontend, envía todas sus solicitudes al gateway mediante APIs REST, evitando comunicación directa con los microservicios. Estos no se comunican entre sí directamente; cuando un microservicio requiere datos de otro (por ejemplo, el microservicio de pronósticos solicita información de sesiones), realiza una solicitud HTTP al gateway, que la redirige al servicio adecuado.

La interacción entre el frontend, el gateway, los microservicios y la base de datos MySQL sigue un flujo cliente-servidor tradicional, optimizado por la independencia de los microservicios y la portabilidad que ofrece Docker para el despliegue del backend en la nube. Las APIs externas, son consumidas directamente por los microservicios relevantes, cuyos datos se procesan y almacenan en MySQL para garantizar disponibilidad y reducir latencia en consultas frecuentes.

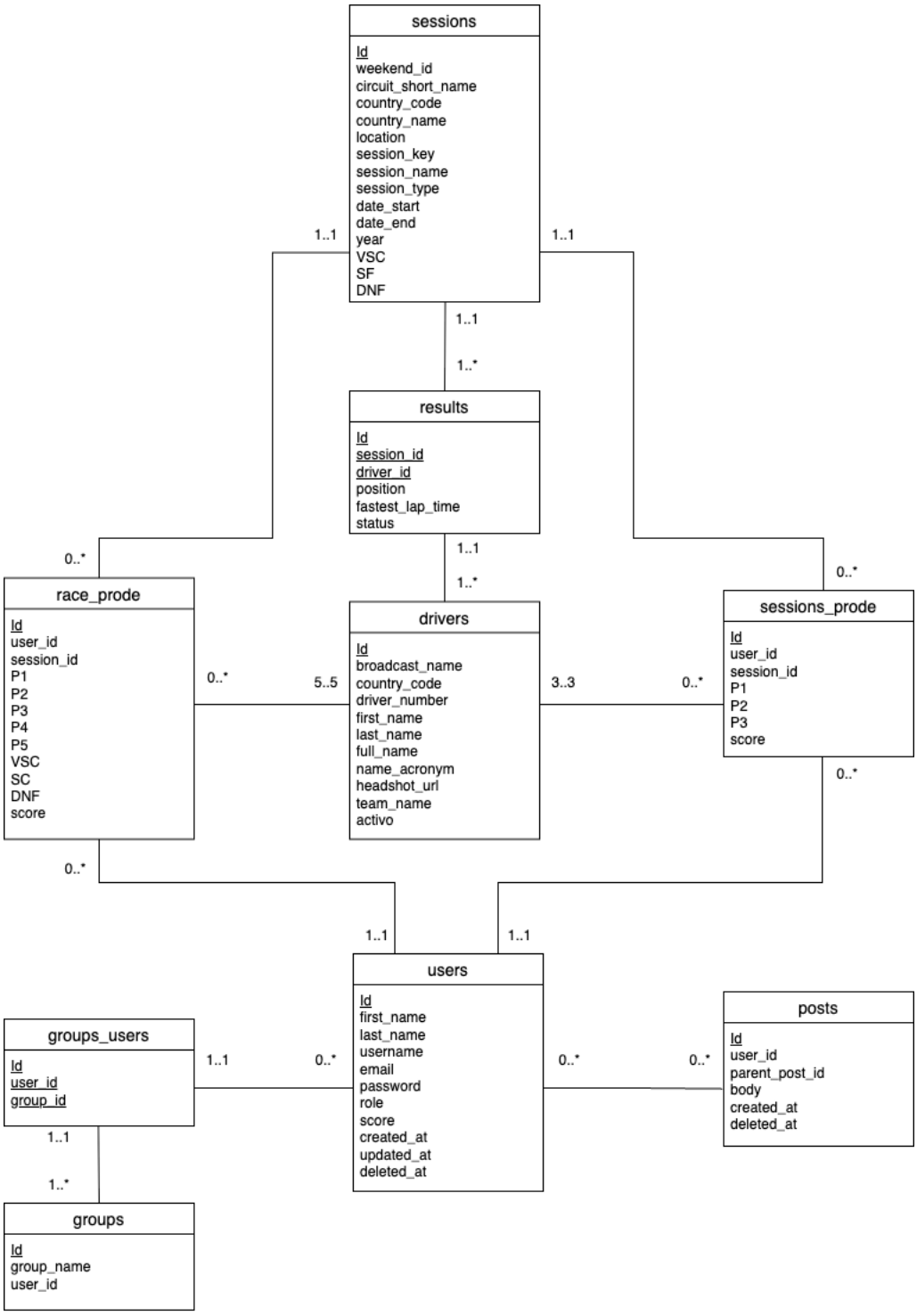
En el backend, cada microservicio sigue el patrón MVC (Model-View-Controller), adaptado para un sistema basado en APIs:

- **Modelo:** Define las estructuras de datos (por ejemplo, Usuario, Pronóstico) y encapsula la interacción con la base de datos MySQL mediante GORM, incluyendo migraciones y consultas SQL.
- **Controlador:** Recibe solicitudes HTTP a través de Gin, procesa parámetros y coordina las respuestas invocando la lógica necesaria.
- **Vista:** Se limita a la serialización de datos en formato JSON, retornados como respuestas RESTful.

Se ha incorporado una capa adicional denominada **Service**, que actúa como intermediaria entre controlador y modelo. Esta capa concentra la lógica de negocio y validaciones, mejorando la separación de responsabilidades y facilitando la realización de pruebas unitarias.

En el frontend, React sigue un enfoque basado en componentes, donde cada pantalla o funcionalidad se encapsula en componentes reutilizables. Los estados se gestionan con React Hooks, y las solicitudes al gateway se realizan mediante la biblioteca Axios, asegurando una interacción dinámica y eficiente con el backend.

Para modelar el sistema, se desarrollaron distintos diagramas UML. En particular, se llegó a una versión final del modelo relacional de base de datos, que se presenta a continuación:


sessions
Id
weekend_id
circuit_short_name
country_code
country_name
location
session_key
session_name
session_type
date_start
date_end
year
VSC
SF
DNF
1..1
1..1
1..1
1..*
results
Id
session_id
driver_id
position
fastest_lap_time
status
0..*
1..1
1..*
0..*
race_prode
Id
user_id
session_id
P1
P2
P3
P4
P5
VSC
SC
DNF
score
drivers
Id
broadcast_name
country_code
driver_number
first_name
last_name
full_name
name_acronym
headshot_url
team_name
activo
sessions_prode
Id
user_id
session_id
P1
P2
P3
score
0..*
5..5
3..3
0..*
0..*
0..*
1..1
1..1
users
Id
first_name
last_name
username
email
password
role
score
created_at
updated_at
deleted_at
groups_users
Id
user_id
group_id
1..1
0..*
0..*
0..*
posts
Id
user_id
parent_post_id
body
created_at
deleted_at
1..1
1..*
groups
Id
group_name
user_id

Explicación del diagrama Entidad-Relación

El diagrama entidad-relación (ER) representa las entidades principales del sistema, sus atributos y las relaciones entre ellas, asegurando que la base de datos MySQL cumpla con los requerimientos funcionales y no funcionales establecidos. Este diagrama refleja la estructura necesaria para gestionar usuarios, pronósticos, sesiones, resultados, grupos privados y otras entidades relacionadas, manteniendo la integridad y consistencia de los datos gracias al enfoque relacional de MySQL. A continuación, se describe el diagrama en detalle, incluyendo las razones detrás de decisiones clave como la separación de pronósticos en dos tablas.

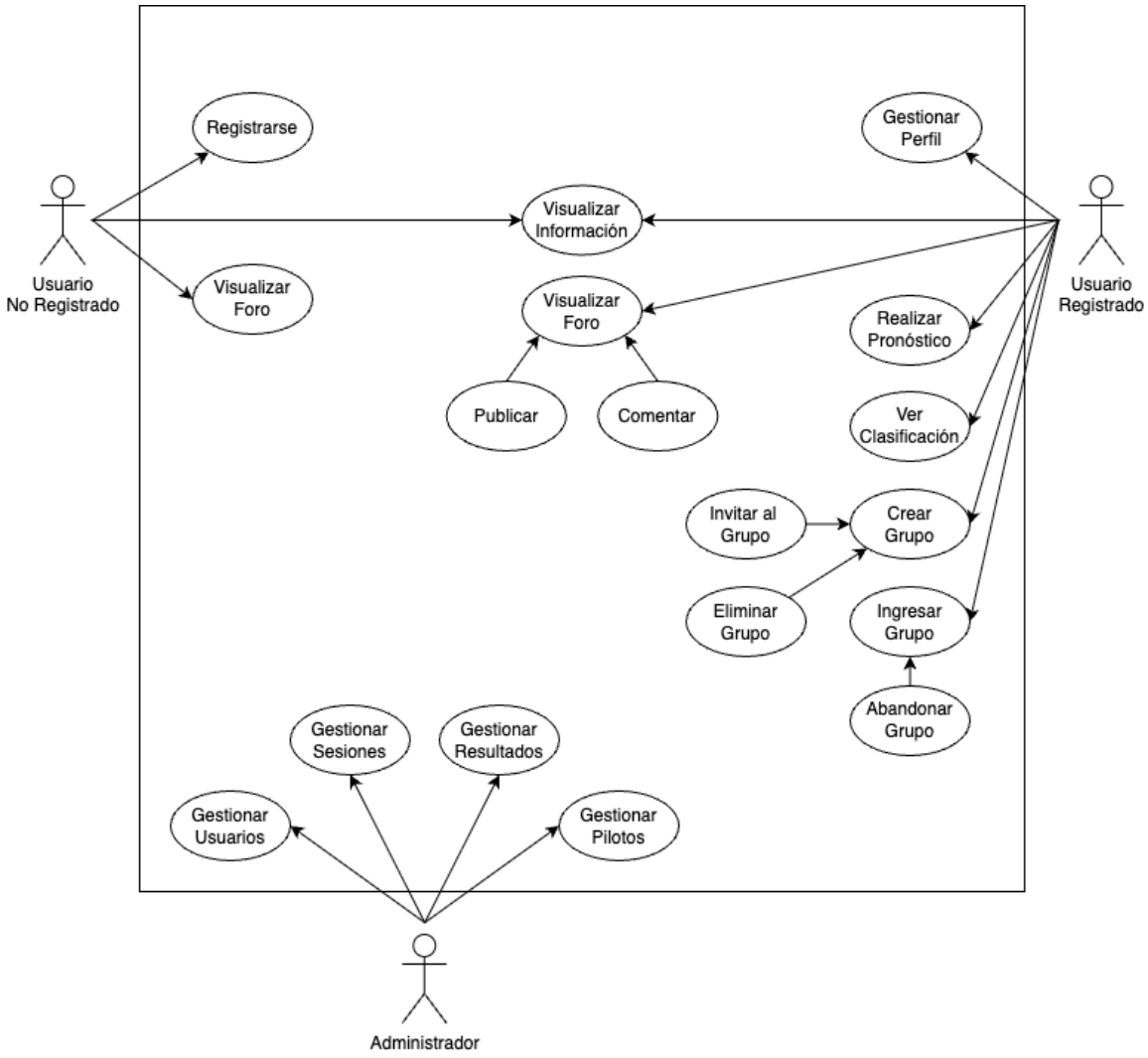
- Entidad: Sessions
    - Atributos: id (clave primaria), weekend_id, circuit_short_name, country_code, country_name, location, session_key, session_name, session_type, date_start, date_end, year, VSC (Virtual Safety Car), SF (Safety Car), DNF (Did Not Finish).
    - Propósito: Almacena información sobre las sesiones de Fórmula 1 agrupadas por Grandes Premios (weekend_id). Atributos como session_type permiten diferenciar entre tipos de sesiones (e.g., clasificación, carrera), mientras que date y location proporcionan contexto para los usuarios junto también como para obtener datos de la API externa. Campos como VSC, SF y DNF registran datos específicos que podrían influir en los pronósticos o resultados.
- Entidad: results
    - Atributos: id (clave primaria), session_id, driver_id, position, fastest_lap_time, status.
    - Propósito: Registra los resultados oficiales de cada sesión, vinculando pilotos con sus posiciones y tiempos más rápidos. Esto permite comparar los pronósticos de los usuarios con los resultados reales para calcular puntajes.

- Entidad: drivers
    - Atributos: id (clave primaria), broadcast_name, country_code, driver_number, first_name, last_name, full_name, name_acronym, headshot_url, team_name, activo
    - Propósito: Contiene información sobre los pilotos de Fórmula 1, esencial para mostrar datos en los pronósticos y resultados. Los datos se obtienen principalmente de la API de OpenF1.org.
- Entidad: race_prode
    - Atributos: id (clave primaria), user_id, session_id, P1, P2, P3, P4, P5, VSC, SC, DNF, score
    - Propósito: Almacena los pronósticos específicos para carreras, incluyendo las predicciones de las primeras cinco posiciones (P1 a P5) y eventos adicionales como Virtual Safety Car (VSC), Safety Car (SC) y pilotos que no terminan (DNF). Sumado a eso se almacena el score obtenido por el usuario para ese pronóstico.
- Entidad: sessions_prode
    - Atributos: id (clave primaria), user_id, session_id, P1, P2, P3.
    - Propósito: Almacena los pronósticos para sesiones de tipo clasificación, enfocándose en las primeras tres posiciones (P1 a P3), ya que estas sesiones suelen tener menos variables predictivas.
- Entidad: users
    - Atributos: id (clave primaria), first_name, last_name, username, email, password, role, score.
    - Propósito: Gestiona la información de los usuarios.

- Entidad: groups
    - Atributos: id (clave primaria), group_name.
    - Propósito: Representa los grupos privados creados por los usuarios para competir entre sí.
- Entidad: groups_users
    - Atributos: id (clave primaria), user_id, group_id.
    - Propósito: Tabla intermedia que modela la relación muchos a muchos entre usuarios y grupos, permitiendo que un usuario pertenezca a varios grupos y un grupo tenga varios usuarios.
- Entidad: posts
    - Atributos: id, user_id, parent_post_id, body, created_at, deleted_at
    - Propósito: Almacena los datos de un post (parent_post_id nulo) y de comentarios (paren_post_id con el valor del post al que comentó).

Diagrama de Casos de Uso

Debido a los distintos tipos de usuarios que he identificado para el uso de la aplicación web, he llegado a concluir en el siguiente diagrama de casos de uso donde se muestran estos últimos mencionados junto a sus posibles interacciones con el alcance del sistema:



Explicación del diagrama de Casos De Uso

El diagrama constituye una representación visual de los requisitos del sistema definidos previamente. En él se identifican distintos perfiles de usuarios con sus respectivas actividades. Los usuarios no registrados pueden únicamente registrarse en el sistema, consultar información general como eventos, fechas y resultados, o visualizar el foro sin posibilidad de interacción. Una vez registrados, los usuarios acceden a un conjunto ampliado de funcionalidades, que incluye la visualización de información, la generación de pronósticos para las distintas sesiones y la participación activa en el foro, contribuyendo así al cumplimiento de los objetivos de la aplicación

web. Finalmente, el usuario administrador posee privilegios para gestionar usuarios, administrar información de sesiones, resultados y datos relacionados con los pilotos.

Pantallas

Desde el inicio, el diseño de las pantallas se abordó con un enfoque mobile first, lo que implica un desafío adicional al tener que condensar mayor cantidad de información en espacios reducidos. En una primera etapa, se desarrollaron diversos mockups de las vistas potenciales, que sirvieron como base orientativa sin representar una restricción estricta. Este proceso permitió el diseño inicial de varias pantallas para el comienzo del proyecto. Posteriormente, las pantallas restantes fueron diseñadas en función de preferencias y criterios propios, prescindiendo de mockups para acelerar el desarrollo.

A continuación, se presentan los mockups generados, que se enfocaron principalmente en el usuario objetivo de la aplicación web, identificado en el diagrama de casos de uso como "Usuario Registrado":

Una de las múltiples vistas diseñadas se muestra en la parte superior. Para la elaboración de estas vistas se utilizó la herramienta Excalidraw, y el conjunto completo de vistas junto con su flujo de secuencia puede consultarse en el siguiente enlace:

https://excalidraw.com/#room=5ea3511e2cca950888d9,9LeAa9P1NG8-KZ_LJg9ctA

El propósito de estas vistas fue representar un flujo funcional del sistema para el usuario final, simulando el uso progresivo para cubrir todas las funcionalidades previstas. Cabe destacar que ciertas funciones, como la del foro —donde los usuarios registrados pueden publicar y comentar— no fueron representadas gráficamente, dado que se planifica adoptar un estilo similar al de la aplicación "X".

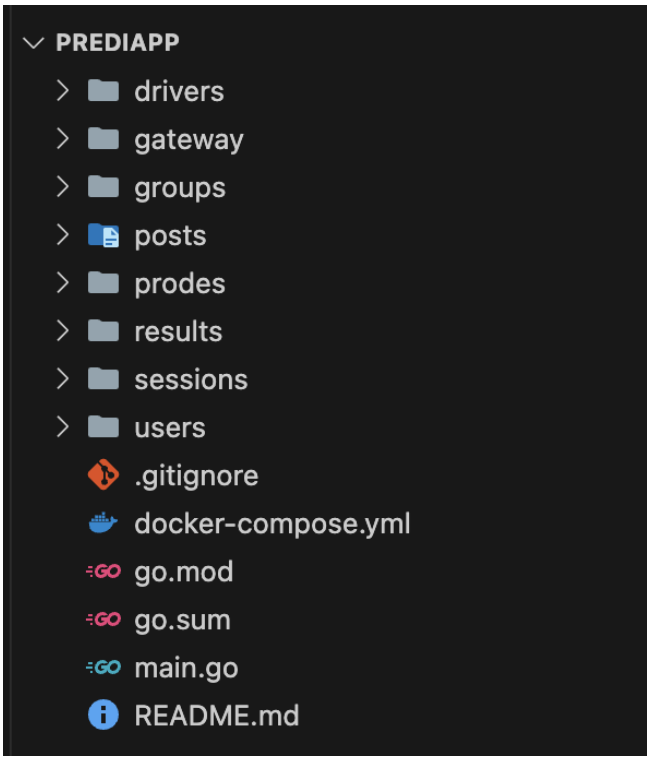
Implementación

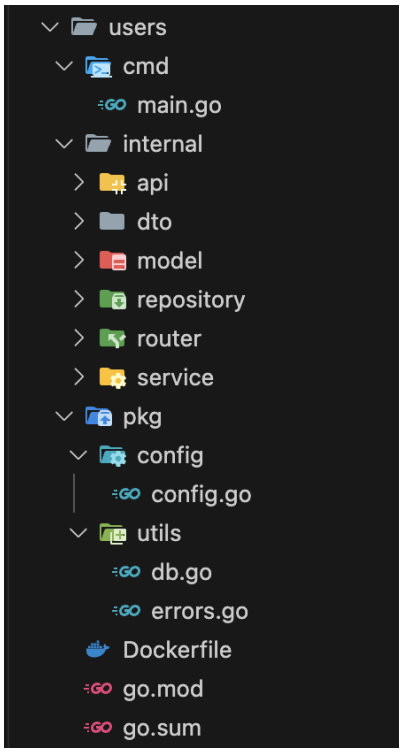
En función de lo expuesto en esta sección, a continuación, se presenta la aplicación práctica de dichos conceptos, iniciando con el backend de la aplicación, en el cual se ha priorizado la consistencia de los datos, la usabilidad y la seguridad del sistema. Tal como se mencionó anteriormente, el backend se compone de diversos microservicios, cada uno implementado bajo una arquitectura MVC, con la adición de una capa denominada Service, destinada a gestionar la lógica específica de cada microservicio.

Cabe destacar que, para optimizar la gestión de la base de datos y promover la reutilización del código, se centralizó toda la lógica relacionada con los modelos, migraciones e inicializaciones en un único módulo ubicado en el directorio db. Este módulo contiene las definiciones estructurales de los modelos de datos, la configuración de la conexión con la base de datos y las funciones para ejecutar las migraciones correspondientes.

Cada microservicio instancia este módulo db para acceder a las entidades y realizar operaciones sobre la base de datos, lo que garantiza una fuente única de verdad para el esquema

y evita la duplicación de definiciones. De esta manera, se mantiene la coherencia de los datos y se facilita el mantenimiento y escalabilidad del sistema. Además, el módulo db incorpora la configuración del pool de conexiones y permite una inicialización homogénea del entorno de base de datos en todos los microservicios.





En la imagen izquierda se presenta la estructura actual del directorio principal, donde se identifican los distintos microservicios junto con otros archivos esenciales de la aplicación. En la imagen derecha se muestra la estructura interna que sigue cada microservicio, la cual será detalladamente explicada a continuación.

Se considera pertinente iniciar con la explicación del microservicio Gateway, dada su función central en la arquitectura. Este microservicio cumple varias tareas clave: centraliza el acceso al sistema al fungir como único punto de entrada para las solicitudes provenientes del frontend, lo que evita accesos directos e indiscriminados a los microservicios individuales. Asimismo, gestiona la autenticación mediante la validación de credenciales y la generación de tokens JWT (JSON Web Tokens). Además, se encarga del enrutamiento, redirigiendo las

solicitudes hacia los microservicios correspondientes mediante la implementación de un proxy inverso. De forma transversal, aplica un middleware que administra aspectos críticos como la política de CORS (Cross-Origin Resource Sharing) y la verificación de autenticación basada en JWT.

En el interior del microservicio, el punto de entrada se encuentra en el archivo main.go, cuya función principal es inicializar el servidor y configurar el enrutamiento a través del framework Gin. Este archivo también implementa el middleware CORS de manera global, define rutas públicas destinadas a la autenticación y establece rutas dinámicas que soportan el funcionamiento del proxy inverso.

Adicionalmente, se han desarrollado controladores específicos para las operaciones relacionadas con la autenticación, permitiendo que los usuarios puedan registrarse e iniciar sesión en la plataforma. En secciones posteriores se presentará un ejemplo detallado de la implementación de los microservicios, tomando como referencia el controlador (handler) responsable del proceso de login.

```go
func LoginHandler(c *gin.Context) {
    // Define la URL del microservicio de usuarios
    usersServiceURL := os.Getenv("USERS_SERVICE_URL")
    secretKey := os.Getenv("JWT_SECRET")

    // 1. Recibir la Solicitud de Login
    var requestBody map[string]interface{}
    if err := c.ShouldBindJSON(&requestBody); err != nil {
        log.Printf("Error parsing request body: %v", err)
        c.JSON(http.StatusBadRequest, gin.H{"error": "invalid request"})
        return
    }

    // 2. Comunicarse con el Microservicio de Usuarios
    jsonBody, err := json.Marshal(requestBody)
    if err != nil {
        log.Printf("Error marshaling request body: %v", err)
        c.JSON(http.StatusInternalServerError, gin.H{"error": "internal server error"})
        return
    }

    req, err := http.NewRequest("POST", usersServiceURL+"/users/login", bytes.NewBuffer(
jsonBody))
    if err != nil {
        log.Printf("Error creating request to users service: %v", err)
        c.JSON(http.StatusInternalServerError, gin.H{"error": "internal server error"})
        return
    }
    req.Header.Set("Content-Type", "application/json")

    client := &http.Client{}
    resp, err := client.Do(req)
    if err != nil {
        log.Printf("Error sending request to users service: %v", err)
        c.JSON(http.StatusInternalServerError, gin.H{"error": "internal server error"})
        return
    }
    defer resp.Body.Close()
```

```go
// 3. Validar la Respuesta del Microservicio
    if resp.StatusCode != http.StatusOK {
        body, _ := ioutil.ReadAll(resp.Body)
        log.Printf("Error response from users service: %s", string(body))
        c.JSON(resp.StatusCode, gin.H{"error": "invalid credentials"})
        return
    }

    // Leer la respuesta del microservicio
    responseBody, err := ioutil.ReadAll(resp.Body)
    if err != nil {
        log.Printf("Error reading response body: %v", err)
        c.JSON(http.StatusInternalServerError, gin.H{"error": "internal server error"})
        return
    }

    var userResponse UserResponseDTO
    if err := json.Unmarshal(responseBody, &userResponse); err != nil {
        log.Printf("Error unmarshaling response body: %v", err)
        c.JSON(http.StatusInternalServerError, gin.H{"error": "internal server error"})
        return
    }

    // 4. Generar ambos tokens
    accessToken, refreshToken, err := GenerateTokens(userResponse.ID, userResponse.Role,
secretKey)
    if err != nil {
        log.Printf("Error generating tokens: %v", err)
        c.JSON(http.StatusInternalServerError, gin.H{"error": "internal server error"})
        return
    }

    // 5. Almacenar el refresh token en el microservicio de users
    refreshReqBody := map[string]interface{}{
        "user_id":    userResponse.ID,
        "token":      refreshToken,
        "expires_at": time.Now().Add(7 * 24 * time.Hour).Format(time.RFC3339),
    }
    refreshJson, err := json.Marshal(refreshReqBody)
    if err != nil {
        log.Printf("Error marshaling refresh token body: %v", err)
        c.JSON(http.StatusInternalServerError, gin.H{"error": "internal server error"})
        return
    }

    refreshReq, err := http.NewRequest("POST", usersServiceURL+"/users/refresh-token", bytes.
NewBuffer(refreshJson))
    if err != nil {
        log.Printf("Error creating refresh token request: %v", err)
        c.JSON(http.StatusInternalServerError, gin.H{"error": "internal server error"})
        return
    }
    refreshReq.Header.Set("Content-Type", "application/json")
    refreshResp, err := client.Do(refreshReq)
    if err != nil || refreshResp.StatusCode != http.StatusCreated {
        log.Printf("Error storing refresh token: %v", err)
        c.JSON(http.StatusInternalServerError, gin.H{"error": "internal server error"})
        return
    }
    defer refreshResp.Body.Close()

    // 6. Asignar tokens a la respuesta
    userResponse.Token = accessToken
    userResponse.RefreshToken = refreshToken

    // 7. Devolver la Respuesta con los tokens
    c.JSON(http.StatusOK, userResponse)
}
```

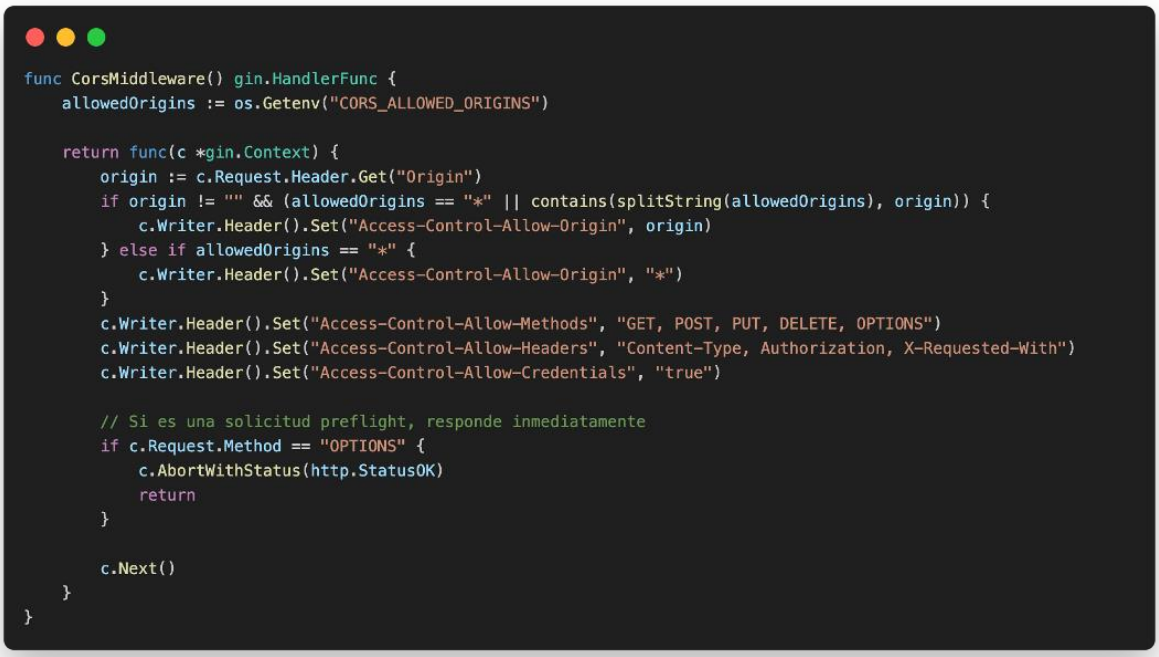
De forma resumida, el microservicio Gateway funciona inicialmente como el punto de entrada principal de la aplicación. En las funcionalidades específicas que involucran la asignación de tokens JWT a los usuarios, realiza una llamada al microservicio de usuarios. Solo si la respuesta es satisfactoria, continúa con la generación del token para el usuario. Esta generación del token se realiza dentro de este mismo microservicio:

```go
func GenerateTokens(userID int, role string, secretKey string) (string, string, error) {
    claims := Claims{
        UserID: userID,
        Role:   role,
        RegisteredClaims: jwt.RegisteredClaims{
            ExpiresAt: jwt.NewNumericDate(time.Now().Add(15 * time.Minute)),
        },
    }
    token := jwt.NewWithClaims(jwt.SigningMethodHS256, claims)
    accessToken, err := token.SignedString([]byte(secretKey))
    if err != nil {
        return "", "", fmt.Errorf("error generating access token: %w", err)
    }

    refreshTokenBytes := make([]byte, 32)
    if _, err := rand.Read(refreshTokenBytes); err != nil {
        return "", "", fmt.Errorf("error generating refresh token: %w", err)
    }
    refreshToken := base64.URLEncoding.EncodeToString(refreshTokenBytes)

    return accessToken, refreshToken, nil
}
```

Adicionalmente, se implementó un middleware encargado de validar los tokens JWT en las solicitudes dirigidas a rutas protegidas. Este middleware extrae el token enviado en el encabezado "Authorization" de la solicitud, lo valida utilizando la clave secreta definida en variables de entorno, verifica su validez y vigencia, y almacena el user_id en el contexto para su posterior uso. Asimismo, en este middleware se implementaron las validaciones correspondientes para la política CORS



```go
func CorsMiddleware() gin.HandlerFunc {
    allowedOrigins := os.Getenv("CORS_ALLOWED_ORIGINS")

    return func(c *gin.Context) {
        origin := c.Request.Header.Get("Origin")
        if origin != "" && (allowedOrigins == "*" || contains(splitString(allowedOrigins), origin)) {
            c.Writer.Header().Set("Access-Control-Allow-Origin", origin)
        } else if allowedOrigins == "*" {
            c.Writer.Header().Set("Access-Control-Allow-Origin", "*")
        }
        c.Writer.Header().Set("Access-Control-Allow-Methods", "GET, POST, PUT, DELETE, OPTIONS")
        c.Writer.Header().Set("Access-Control-Allow-Headers", "Content-Type, Authorization, X-Requested-With")
        c.Writer.Header().Set("Access-Control-Allow-Credentials", "true")

        // Si es una solicitud preflight, responde inmediatamente
        if c.Request.Method == "OPTIONS" {
            c.AbortWithStatus(http.StatusOK)
            return
        }

        c.Next()
    }
}
```

Esto permite recibir solicitudes únicamente de los orígenes especificados en variables de entorno configuradas como secretas. Finalmente, se implementó un proxy inverso encargado de enrutar todas las solicitudes hacia el microservicio correspondiente, lo que contribuye a evitar sobrecargas innecesarias y a mejorar los tiempos de respuesta.

```go
func ReverseProxy() gin.HandlerFunc {
    return func(c *gin.Context) {

        // 1. Manejar las preflight requests (OPCIONAL, si ya lo tenías)
        if c.Request.Method == http.MethodOptions {
            c.AbortWithStatus(http.StatusOK)
            return
        }


// 2. Normalizar la ruta para eliminar el slash final (p.ej.: "/driver
s/" => "/drivers")
        trimmedPath := strings.TrimSuffix(c.Request.URL.Path, "/")
        if trimmedPath == "" {
            trimmedPath = "/" // Evita quedarte sin barra en la raíz
        }
        c.Request.URL.Path = trimmedPath

        // 3. Determinar el microservicio y el path a partir de la ruta
        target, proxyPath := getTargetURL(c.Request.URL.Path)
        if target == "" {
            log.Printf("Service not found for path: %s", c.Request.URL.
Path)
            c.JSON(http.StatusNotFound, gin.H{"error":
"service not found"})
            return
        }

        // 4. Eliminar barra final también en proxyPath (por si acaso)
        if strings.HasSuffix(proxyPath, "/") && len(proxyPath) > 1 {
            proxyPath = strings.TrimSuffix(proxyPath, "/")
        }
```

```
// 5. Parsear la URL del target
        targetURL, err := url.Parse(target)
        if err != nil {
            c.JSON(http.StatusInternalServerError, gin.H{"error":
"internal server error"})
            return
        }

        // 6. Crear el Reverse Proxy de Go
        proxy := httputil.NewSingleHostReverseProxy(targetURL)


// Director: manipula la request ANTES de mandarla al microservicio
        proxy.Director = func(req *http.Request) {
            // Ajustamos la request al target final
            req.URL.Scheme = targetURL.Scheme
            req.URL.Host = targetURL.Host
            req.URL.Path = proxyPath

            // En caso de que venga body, se vuelve a leer
            if req.Body != nil {
                body, _ := ioutil.ReadAll(req.Body)
                req.Body = ioutil.NopCloser(bytes.NewBuffer(body))
                req.ContentLength = int64(len(body))
            }

            // Ajustamos el Host
            req.Host = targetURL.Host
        }

        proxy.ModifyResponse = func(res *http.Response) error {
            if res.StatusCode == http.StatusMovedPermanently || res.
StatusCode == http.StatusFound {
                location := res.Header.Get("Location")
                if location != "" {
                    c.Redirect(res.StatusCode, location)
                    return nil
                }
            }
            return nil
        }

        proxy.ServeHTTP(c.Writer, c.Request)
    }
}
```

Con esta implementación, el microservicio establece un único punto de entrada, simplificando la interacción del cliente con los diversos microservicios. Esta arquitectura facilita adaptaciones futuras, ya que cualquier cambio o incorporación de nuevos microservicios se traduce únicamente en la adición de nuevas rutas y URLs, lo que contribuye a una escalabilidad más eficiente. Además, se implementó la autenticación mediante JWT Token, garantizando un nivel de seguridad adecuado al proteger las rutas sensibles mediante middleware especializado.

Como se observó en la imagen al inicio de esta sección, cada microservicio contiene un directorio internal que organiza claramente las responsabilidades. En particular, el controlador actúa como capa intermediaria entre las solicitudes HTTP provenientes del frontend y el flujo de ejecución del sistema. Su función principal es validar las solicitudes antes de delegar la lógica al Service, evitando así sobrecargas innecesarias cuando se reciben datos erróneos o mal formateados. En síntesis, el controlador procesa las solicitudes, valida los datos de entrada, invoca la lógica del Service y finalmente devuelve las respuestas correspondientes. La definición del controlador se estructura de la siguiente manera:

```go
type UserController struct {
    userService service.UserServiceInterface
}

func NewUserController(userService service.
UserServiceInterface) *UserController {
    return &UserController{
        userService: userService,
    }
}
```

El segundo bloque de código corresponde a la inicialización del controlador mediante un constructor que refleja un diseño basado en principios de arquitectura de software, tales como la inyección de dependencias, la separación de responsabilidades y el principio de inversión de dependencias. En cuanto a la inyección de dependencias, el controlador no crea su propio servicio, sino que recibe una instancia preexistente, lo que proporciona mayor flexibilidad al permitir modificar la implementación del servicio sin alterar el controlador y contribuye a la escalabilidad del código.

Respecto a la separación de responsabilidades, el controlador se encarga exclusivamente de manejar las solicitudes HTTP, validar los datos de entrada y gestionar las respuestas, mientras que la lógica de negocio se delega al Service, lo que resulta en un código más limpio y facilita su mantenimiento. Por último, el principio de inversión de dependencias establece que los módulos de alto nivel, como el controlador, no deben depender de implementaciones concretas, sino de abstracciones, representadas por interfaces.

En los distintos microservicios, Gin es el framework utilizado para manejar las solicitudes HTTP provenientes del frontend (más precisamente, a través del gateway) y para conectar dichas solicitudes con la lógica del controlador. Gin permite un flujo de trabajo sencillo, eficiente y ordenado. A continuación, se explicará su funcionamiento tomando como ejemplo la función Login:

```go
func (ctrl *UserController) Login(c *gin.Context) {
    var request dto.UserLoginRequestDTO
    if err := c.ShouldBindJSON(&request); err != nil {
        apiErr := e.NewBadRequestApiError("invalid request")
        c.JSON(apiErr.Status(), apiErr)
        return
    }

    response, apiErr := ctrl.userService.Login(c.Request.Context(), request)
    if apiErr != nil {
        c.JSON(apiErr.Status(), apiErr)
        return
    }

    c.JSON(http.StatusOK, response)
}
```

Gin se encarga de gestionar todo lo relacionado con la solicitud HTTP. El parámetro c *gin.Context funciona como el núcleo de Gin, ya que contiene toda la información pertinente a la solicitud (método HTTP, cuerpo JSON, parámetros, entre otros) y proporciona herramientas para construir la respuesta. El proceso se desglosa de la siguiente manera:

Mediante c.ShouldBindJSON(&request), Gin convierte el cuerpo JSON de la solicitud en una estructura Go (UserLoginRequestDTO). En caso de que el JSON esté mal formado, se genera un error, facilitando la validación de la entrada.

A continuación, se invoca el método ctrl.userService.Login, al que se le pasa el contexto de la solicitud (c.Request.Context()) y el DTO correspondiente. El uso del contexto es fundamental, ya que permite al servicio gestionar aspectos como la cancelación en caso de interrupción de la solicitud. Si el servicio retorna un error, este se envía como respuesta JSON utilizando c.JSON.

Si la operación es exitosa, se responde con un estado HTTP 200 OK mediante c.JSON(http.StatusOK, response), enviando los datos en formato JSON. Gin se encarga de serializar el objeto y establecer el encabezado Content-Type: application/json.

Este flujo proporciona una gestión clara y directa de las solicitudes HTTP, eliminando la necesidad de escribir código repetitivo para el manejo de las mismas.

Posteriormente, en el archivo router.go se configura la conexión entre las rutas de la API y los manejadores definidos en el controlador.

```go
func MapUrls(engine *gin.Engine, userController *api.UserController) {
    usersGroup := engine.Group("/users")
    {
        usersGroup.POST("/login", userController.Login)
        // Otras rutas ..
    }

    fmt.Println("Finishing mappings configurations")
}
```

En este punto se observa el funcionamiento del enrutamiento proporcionado por Gin:

El objeto principal es el engine *gin.Engine, que administra todas las rutas y es instanciado en el archivo main.go para luego ser configurado mediante la función MapUrls.

La instrucción engine.Group("/users") crea un grupo de rutas que comparten el prefijo /users. Esto permite organizar y distinguir claramente las rutas correspondientes al microservicio de usuarios de aquellas pertenecientes a otros microservicios, como /drivers o /posts en el gateway.

La línea usersGroup.POST("/login", userController.Login) indica a Gin que, al recibir una solicitud HTTP POST en /users/login, debe invocar el método Login del controlador

correspondiente. Cuando dicha solicitud es recibida, Gin crea un objeto gin.Context, lo pasa al manejador y ejecuta la función mencionada previamente.

Este enfoque favorece la claridad del código y facilita la adición de nuevas rutas mediante la simple inclusión de una línea adicional. Tal como se ha explicado anteriormente, Gin fue seleccionado debido a su rapidez, facilidad de uso (por ejemplo, con métodos como ShouldBindJSON), soporte para arquitecturas basadas en microservicios, integración con el contexto de Go para manejar cancelaciones o timeouts —un aspecto relevante en aplicaciones donde las solicitudes pueden ser interrumpidas— y por su capacidad para favorecer la escalabilidad de los microservicios.

El uso de Gin se ha implementado de forma consistente en cada controlador de todos los microservicios, siguiendo las mismas buenas prácticas.

Adicionalmente, se ha incorporado una capa intermedia entre los controladores y los modelos denominada Service, que concentra la lógica de negocio. Esta capa asume la responsabilidad de procesar datos, realizar llamadas a otros microservicios y gestionar las reglas de la aplicación. Al igual que en el controlador, el Service se define en un paquete específico y utiliza una estructura con dependencias inyectadas, promoviendo la flexibilidad y escalabilidad del código.

```go
type userService struct {
    userRepo repository.UserRepository
}

type UserServiceInterface interface {
    SignUp(ctx context.Context, request dto.UserSignUpRequestDTO) (dto.UserSignUpResponseDTO, e.ApiError)
    Login(ctx context.Context, request dto.UserLoginRequestDTO) (dto.UserLoginResponseDTO, e.ApiError)
    // Otros métodos ..
}

func NewUserService(userRepo repository.UserRepository) UserServiceInterface {
    return &userService{
        userRepo: userRepo,
    }
}
```

El servicio userService cuenta con un repositorio de usuarios (userRepo) que se inyecta mediante el constructor NewUserService. En todas sus funciones, se pasa un parámetro de tipo Context, proveniente del controlador, cuyo propósito es gestionar cancelaciones, tiempos de espera o la transmisión de datos entre capas, aunque en este caso particular no se emplee para este último fin. Dicho contexto se propaga hacia el repositorio, asegurando que las consultas a la base de datos respeten cualquier cancelación o timeout que pueda ocurrir durante la ejecución.

Posteriormente, se implementó la capa de repositorio (repository), encargada de la interacción directa con la base de datos a través de los modelos centralizados en el módulo db/model. Esta capa administra operaciones de creación, consulta, actualización y eliminación de datos (CRUD) utilizando GORM como ORM, tal como se mencionó anteriormente. En concreto, en el microservicio de usuarios, se define una estructura denominada userRepository que tiene como única dependencia la conexión a la base de datos proporcionada por el módulo db. Esta dependencia se inyecta mediante el constructor NewUserRepository, siguiendo prácticas de diseño que promueven modularidad, flexibilidad y facilidad de mantenimiento.

En el punto de entrada del microservicio (archivo main.go), la conexión a la base de datos se inicializa mediante las funciones expuestas en el módulo db, que configuran la conexión

MySQL con GORM y ejecutan las migraciones correspondientes para asegurar que las tablas definidas en los modelos estén actualizadas. A continuación, se instancia el repositorio con dicha conexión, integrándose posteriormente con la capa de servicio y controlador.

Para ilustrar el funcionamiento de esta capa, se presenta como ejemplo una función típica del repositorio que obtiene un usuario por su email. El proceso es el siguiente:

```go
func (r *userRepository) GetUserByEmail(ctx context.Context, email string) (*model.User, e.ApiError) {
    var user model.User
    if err := r.db.WithContext(ctx).Where("email = ?", email).First(&user).Error; err != nil {
        if err == gorm.ErrRecordNotFound {
            return nil, e.NewNotFoundApiError("user not found")
        }
        return nil, e.NewInternalServerApiError("error finding user by email", err)
    }
    return &user, nil
}
```

- **Entrada:** recibe un contexto (ctx context.Context) y un email como parámetro.
- **Búsqueda:** utiliza el método Where("email = ?") de GORM para filtrar el registro por email, y First para cargar el resultado en la estructura model.User.
- **Manejo de errores:** si no se encuentra el registro, GORM devuelve el error gorm.ErrRecordNotFound, y el repositorio responde con un error HTTP 404 (Not Found). En caso de errores técnicos, como fallos en la conexión, se retorna un error HTTP 500 (Internal Server Error).
- **Respuesta**: en caso de éxito, retorna un puntero a la estructura User con los datos encontrados, sin error.

Esta capa actúa como puente entre el servicio y la base de datos, simplificando las consultas y manejando los errores de forma clara y consistente. La utilización de GORM facilita

la interacción con la base de datos mediante métodos intuitivos como Create, First y Where, además de gestionar automáticamente el mapeo entre estructuras y tablas. Adicionalmente, la migración automática ejecutada durante la inicialización garantiza que la base de datos esté sincronizada con los modelos definidos, lo que contribuye a un despliegue ágil y seguro.

En el punto de entrada del microservicio, el archivo main.go actúa como el orquestador principal del flujo de ejecución. Sus responsabilidades principales incluyen:

- **Lectura del puerto:** obtiene el valor del puerto en el que se ejecutará el servidor desde la variable de entorno PORT. En caso de no estar definido, el programa finaliza con un error para evitar configuraciones incorrectas.
- **Inicialización de la conexión a la base de datos:** llama a la función db.Init() (definida en el módulo db) para establecer la conexión con MySQL mediante GORM. Si la conexión falla, el programa se detiene con un error. Además, se asegura que la conexión a la base de datos se cierre correctamente al finalizar la ejecución mediante un defer db.DisconnectDB().
- **Inicialización de capas:** crea las instancias del repositorio, servicio y controlador, conectándolas mediante inyección de dependencias para mantener una arquitectura modular y desacoplada.
- **Configuración del router:** inicia el enrutador Gin con gin.Default() (que incluye middleware para logging y recuperación de pánicos) y asigna las rutas HTTP mediante la función router.MapUrls, que vincula los endpoints con los manejadores del controlador.
- **Ejecución del servidor:** arranca el servidor Gin en el puerto especificado, habilitando la recepción de solicitudes HTTP (por ejemplo, POST /users/login)

```go
func main() {
  // 1) Validar variables de entorno críticas
  requiredEnvVars := []string{
    "PORT",
    "JWT_SECRET",
    "DB_HOST", "DB_PORT", "DB_USER", "DB_PASS", "DB_NAME",
  }
  for _, envVar := range requiredEnvVars
    if os.Getenv(envVar) == ""          {
      log.Fatalf("%s is not set in the environment", envVar)
    }
  }

  // 2) Inicializar conexión (sin migrar)
  if err := db.Init(); err != nil
    log.Fatalf("db.Init failed: %v", err)
  }

  // 3) Inicializar repositorio, servicio y controlador
  userRepo := repository.NewUserRepository(db.DB)
  userService := service.NewUserService(userRepo)
  userController := api.NewUserController(userService)

  // 4) Configurar router
  ginRouter := gin.Default()
  router.MapUrls(ginRouter, userController)

  // 5) Iniciar servidor
  port := os.Getenv("PORT")
  go func() {
    fmt.Printf("Users service listening on port %s...\n", port)
    if err := ginRouter.Run(":" + port); err != nil
      log.Fatalf("Failed to run server on port %s: %v", port, err)
    }
  }()

  // 6) Esperar señal de interrupción
  sigs := make(chan os.Signal, 1)
  signal.Notify(sigs, syscall.SIGINT, syscall.SIGTERM)
  <-sigs
  log.Println("Stopping users service...")
}
```

En el contexto de la arquitectura de microservicios, cada microservicio posee su propio archivo main.go que cumple esta función. Las solicitudes externas son recibidas inicialmente por el gateway, que las redirige al microservicio correspondiente. En el microservicio users, Gin se encarga de gestionar dichas solicitudes, el controlador procesa y valida la entrada, el servicio aplica la lógica de negocio, y el repositorio interactúa con la base de datos. De este modo, el archivo main.go integra y coordina todos los componentes necesarios para que el microservicio funcione de manera eficiente y coherente.

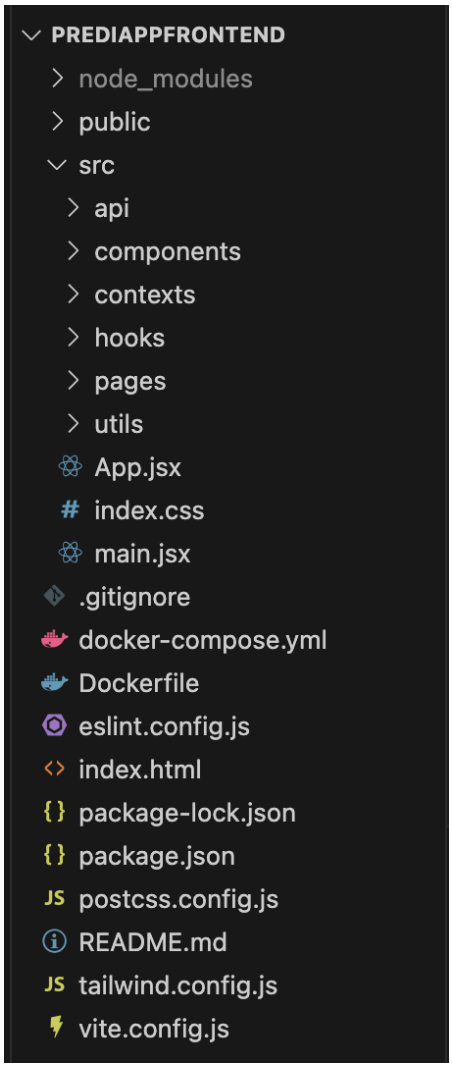
El frontend de la aplicación fue desarrollado utilizando React, Vite y Tailwind CSS, siguiendo un enfoque mobile-first y una arquitectura modular que facilita la escalabilidad y el mantenimiento. Se trata de una aplicación de página única (SPA) que integra de manera eficiente con el backend previamente descrito, empleando tecnologías modernas para optimizar tanto el proceso de desarrollo como la experiencia del usuario final. Las tecnologías y herramientas utilizadas incluyen:

- **Vite:** Herramienta de construcción que proporciona un entorno de desarrollo ágil, con recarga en caliente (hot reload) y generación de builds optimizados. Su configuración se encuentra en el archivo vite.config.js, donde se habilita soporte para React y Tailwind CSS.
- **React**: Biblioteca para la construcción de interfaces de usuario basada en componentes reutilizables. Se utiliza el paradigma de Hooks (useState, useEffect, useContext) para la gestión del estado y efectos secundarios. La navegación se administra mediante React Router, específicamente con el componente BrowserRouter definido en App.jsx.
- **Tailwind CSS:** Framework CSS utility-first que facilita la creación de estilos responsivos y personalizados. La configuración está definida en tailwind.config.js,

permitiendo la personalización de colores y temas, con un enfoque adaptado a dispositivos móviles.

- **Axios:** Librería empleada para realizar peticiones HTTP al backend desde la carpeta src/api, con manejo adecuado de errores y respuestas.
- **Context API:** Utilizada para la gestión del estado global de la aplicación, mediante contextos como AuthContext para la autenticación.

Se ha prestado especial atención a la organización del código y la estructuración de carpetas, con el objetivo de lograr una modularidad óptima y facilitar la comprensión y mantenimiento del código. Este diseño se alinea con las mejores prácticas recomendadas para el desarrollo con React y principios generales de programación modular.



El directorio src/api contiene las funciones encargadas de interactuar con el backend, organizadas y agrupadas por entidad para facilitar su mantenimiento y reutilización.

El directorio src/components alberga todos los componentes reutilizables, clasificados según su propósito y funcionalidad dentro de la aplicación, promoviendo la modularidad y la separación de responsabilidades.

El directorio src/contexts se utiliza principalmente para gestionar el estado global relacionado con la autenticación, mediante la implementación de contextos que permiten compartir información entre componentes sin necesidad de prop drilling.

El directorio src/hooks se dedica a la definición de hooks personalizados que encapsulan lógica reutilizable relacionada con la gestión del estado, efectos secundarios, y otras funcionalidades específicas.

El directorio src/pages contiene las páginas que cubren las diferentes funcionalidades del sistema presentadas al usuario final. Estas páginas funcionan como contenedores que orquestan los componentes y gestionan la navegación.

Siguiendo con el ejemplo de la funcionalidad de inicio de sesión (Login), se describe su implementación basada en la interacción entre la página LoginPage, el componente LoginForm, el contexto AuthContext y la función login definida en el módulo api.

La página LoginPage actúa como contenedor principal para el proceso de autenticación, donde se renderiza el componente reutilizable LoginForm. Este componente presenta un formulario sencillo con campos para correo electrónico y contraseña, diseñado con Tailwind CSS para asegurar un estilo limpio, accesible y responsivo. El diseño elimina elementos superfluos para maximizar la claridad y la usabilidad.

```jsx
import React from "react";
import LoginForm from "../components/LoginForm";
import Header from "../components/Header";

const LoginPage = () => {
  return (
    <div className="flex flex-col min-h-screen bg-gray-50">
      {/* Opcional: si quieres que en Login aparezca o no el Header global */}
      <Header />

      {/* main con flex-grow para que “empuje” al footer al fondo */}
      <main className="flex-grow flex justify-center items-center">
        <LoginForm />
      </main>

      <footer className="bg-gray-200 text-gray-700 text-center py-3 text-sm">
        <p>© 2025 PrediApp</p>
      </footer>
    </div>
  );
};

export default LoginPage;
```

LoginForm gestiona internamente el estado del formulario mediante el hook useState, que mantiene actualizados los valores de los campos email, password y posibles mensajes de error. Para la navegación post-login, se utiliza el hook useNavigate de React Router, que permite redirigir al usuario una vez que la autenticación es exitosa.

```
const LoginForm = () => {
  const { login: authLogin } = useContext(AuthContext);
  const navigate = useNavigate();
  const [credentials, setCredentials] = useState({ email: "", password: "" });
  const [error, setError] = useState(null);

  const handleChange = (e) => {
    setCredentials({ ...credentials, [e.target.name]: e.target.value });
  };

  const handleSubmit = async (e) => {
    e.preventDefault();
    try {
      setError(null);
      const response = await login(credentials);
      authLogin(response);

      if (response.role === "admin") {
        navigate("/admin", { replace: true });
      } else {
        navigate("/", { replace: true });
      }
    } catch (err) {
      setError(
        err.message ||

"Error al iniciar sesión. Verifica tus credenciales o contacta al soporte."
      );
    }
  };
```

El manejo de eventos en LoginForm se realiza a través de la función handleChange, que actualiza el estado con los valores ingresados por el usuario, y la función asíncrona handleSubmit, que se encarga de procesar el envío del formulario. Esta última invoca la función login del módulo api, la cual realiza una petición HTTP POST al endpoint /login del backend, enviando las credenciales ingresadas.

En caso de éxito, la respuesta contiene un token JWT junto con los datos del usuario (ID, nombre, rol, etc.). El contexto AuthContext utiliza este token para actualizar el estado global de autenticación mediante su función login, almacenando el token en el almacenamiento local (localStorage) y decodificando su contenido para poblar la información del usuario.

Posteriormente, en función del rol asignado al usuario (admin o usuario común), LoginForm realiza una redirección condicional hacia la ruta correspondiente, garantizando así una navegación acorde a los permisos.

En caso de error durante el proceso de autenticación (como credenciales incorrectas o problemas de red), el componente actualiza el estado de error para mostrar un mensaje claro y conciso al usuario, facilitando la retroalimentación inmediata.

El contexto AuthContext gestiona el estado global de autenticación, almacenando y validando el token JWT, así como la información del usuario asociada. Este contexto implementa funciones para el inicio de sesión, cierre de sesión, y refresco del usuario, asegurando la persistencia y sincronización del estado de autenticación en toda la aplicación.

Finalmente, la función login del módulo api realiza la comunicación con el backend utilizando Axios para enviar las credenciales y recibir la respuesta. Esta función se encarga de almacenar el token recibido y de retornar los datos relevantes del usuario para su procesamiento en el contexto.

```
export const login = async (userData) => {
  try {
    const { data } = await axios.post('/login', userData);

// data = { id, first_name, last_name, username, email, role, token, created_at }
    const {
      token,
      id,
      first_name,
      last_name,
      username,
      email,
      role,
      created_at,
    } = data;
    if (!token) throw new Error("No se recibió token en login");

    // Guardar solo el JWT
    localStorage.setItem("jwtToken", token);
    setAuthToken(token);

    // Devolver toda la info (incluido el token)
    return {
      token,
      id,
      first_name,
      last_name,
      username,
      email,
      role,
      created_at,
    };
  } catch (err) {
    console.error("Login error:", err.response?.data || err.message);
    throw new Error(err.response?.data?.error || "Error al iniciar sesión");
  }
};
```

En la totalidad del frontend, se aplicaron consistentemente prácticas modernas de desarrollo basadas en React y Hooks, con el objetivo de simplificar el manejo del estado y fomentar la reutilización de componentes, evitando la duplicación de código. Tailwind CSS fue

empleado para la estilización eficiente, permitiendo aplicar diseños rápidos y responsivos sin la necesidad de desarrollar CSS personalizado para cada clase.

Para la gestión de estados y datos compartidos entre componentes, se optó por el uso de Context API, evitando el prop-drilling y proporcionando una solución más ligera en comparación con otras librerías de gestión de estado como Redux.

Esta arquitectura y elección tecnológica buscó alcanzar diversos beneficios fundamentales para la aplicación web:

- **Modularidad:** La organización en componentes reutilizables, páginas específicas y funciones API desacopladas facilita la incorporación de nuevas funcionalidades sin necesidad de reestructurar el sistema completo. Esto reduce considerablemente los tiempos de desarrollo y mejora la mantenibilidad del código.
- **Rendimiento:** La herramienta Vite optimiza el proceso de desarrollo mediante recargas rápidas en caliente y produce builds minificados para producción, lo que mejora la eficiencia en la entrega del frontend. React, mediante el uso de Hooks, minimiza re-renderizados innecesarios, mientras que Tailwind CSS realiza purgas automáticas del CSS no utilizado. En conjunto, estas tecnologías garantizan una experiencia de usuario fluida, incluso en condiciones de conexión lentas.
- **Usabilidad:** El diseño mobile-first, complementado con elementos como modales y formularios claros, contribuye a una interfaz intuitiva y accesible. La navegación es consistente gracias a componentes como la NavigationBar, facilitando la interacción del usuario con la aplicación sin importar su nivel de experiencia.
- **Escalabilidad:** La arquitectura implementada soporta el aumento en la cantidad de usuarios, sesiones y pilotos, con contextos y APIs diseñados para adaptarse al crecimiento. Además, la integración con APIs externas permite la incorporación de

nuevos datos y funcionalidades, posibilitando la expansión de Predi hacia otras categorías de carreras o un mayor volumen de usuarios.

De esta manera es como una vez finalizado el desarrollo del sistema en su completitud, se optó por llevar a cabo un despliegue del mismo utilizando la herramienta DigitalOcean como se mencionó en la apertura de esta sección. De esta manera podrá visualizar el sistema en funcionamiento en la siguiente URL: https://testing.gopredi.com

En resumen, tanto el backend como el frontend están diseñados para crecer juntos. Nuevas funcionalidades pueden implementarse de manera rápida y sencilla simplemente añadiendo páginas, componentes y nuevos endpoints sin romper la arquitectura del sistema.

Validación y pruebas realizadas

Durante el desarrollo del sistema no se implementaron pruebas unitarias ni de integración debido a que el alcance del proyecto no requería un nivel de complejidad elevado en esta etapa inicial. En su lugar, se adoptó una metodología de verificación basada en la técnica de prueba y error, utilizando herramientas como Postman para evaluar el correcto funcionamiento de los distintos endpoints de la API y la coherencia en las respuestas del backend.

Una vez completada la fase principal de desarrollo, se realizaron pruebas de tipo beta en un entorno controlado. Para ello, se invitó a un grupo reducido de usuarios cercanos a interactuar con la aplicación, con el objetivo de identificar errores, inconsistencias o comportamientos no esperados. La retroalimentación obtenida de estas pruebas permitió detectar casos particulares de uso, corregir fallos menores y mejorar la experiencia de usuario antes del despliegue definitivo.

Este enfoque de validación, aunque más limitado en comparación con metodologías formales de testing, resultó adecuado para un proyecto de esta envergadura, garantizando que

la aplicación cumpliera con los objetivos funcionales definidos y ofreciera un desempeño estable para el usuario final.

Impacto Económico

Este informe presenta un análisis del impacto económico asociado con el desarrollo e implementación del sistema Predi. El análisis abarca los costos de desarrollo, los costos de implementación, los ahorros potenciales para los usuarios y una evaluación del retorno de inversión (ROI). El objetivo es proporcionar una visión clara y fundamentada de los aspectos económicos del proyecto.

Costos de Desarrollo

El desarrollo de Predi requirió aproximadamente 500 horas de trabajo, abarcando el diseño, codificación, pruebas y optimización del backend y el frontend. Para estimar el costo de desarrollo, se consideró la tarifa promedio por hora de un desarrollador en 2024 redondeada a $8,53 USD por hora. De esta manera, en base a cálculos aproximados de 500 horas de trabajo junto con el salario promedio, se estima que el costo de desarrollo es de $4265 USD. Cabe señalar que se trata de una estimación basada en valores de referencia y no de un monto exacto.

Costos de Implementación

La implementación de Predi contempla el despliegue tanto del frontend como del backend en la nube, utilizando una única máquina virtual (Droplet) en DigitalOcean para alojar ambos componentes. A continuación, se describen los costos asociados a esta configuración.

El Droplet seleccionado en DigitalOcean cuenta con las siguientes especificaciones técnicas: 2 vCPU, 4 GB de memoria RAM, 80 GB de almacenamiento SSD y 2 TB de transferencia de datos mensuales. Este tipo de instancia resulta adecuado para soportar la carga esperada durante las primeras etapas de implementación, considerando un tráfico bajo a moderado.

El costo mensual del Droplet es de 24 USD, lo que implica un costo anual estimado de 288 USD. De esta manera, el costo total mensual de implementación de Predi se consolida en 24 USD, integrando tanto el frontend como el backend en un entorno único y optimizado.

Ahorros Potenciales

Dado que Predi es una plataforma sin fines de lucro destinada a la comunidad de Fórmula 1, los ahorros se reflejan principalmente en beneficios operativos y en la accesibilidad para los usuarios, más que en términos monetarios directos.

La plataforma fue diseñada bajo un enfoque mobile-first y optimizada para ofrecer un desempeño ágil y eficiente en dispositivos con recursos limitados, como teléfonos de gama media o baja. Esto permite a los usuarios acceder a la plataforma sin la necesidad de invertir en equipos de alta gama, lo que representa un ahorro indirecto en términos de hardware.

Al tratarse de una solución web alojada en la nube, Predi elimina la dependencia de infraestructura física, como servidores locales o espacios presenciales, así como la necesidad de personal dedicado para la gestión de comunidades relacionadas con la Fórmula 1. Esta centralización facilita el acceso gratuito para los usuarios, evitando la necesidad de participar en eventos presenciales o utilizar plataformas comerciales para la interacción comunitaria, y reduce significativamente los costos operativos asociados con hardware local y mantenimiento.

Retorno de Inversión (ROI)

El proyecto Predi no busca un retorno de inversión monetario, dado que su propósito principal es fomentar la unión y el desarrollo social de la comunidad de Fórmula 1. Los objetivos fundamentales son de índole social y personal, orientados a fortalecer la interacción comunitaria sin fines comerciales.

Impacto Social

El sistema Predi, una plataforma web diseñada para conectar a los aficionados de la Fórmula 1 mediante pronósticos, foros y grupos privados, trasciende sus objetivos técnicos al generar un impacto social relevante en la comunidad global de este deporte. Su propósito principal consiste en proporcionar un espacio digital gratuito y accesible que facilite la interacción entre entusiastas, fortaleciendo los lazos comunitarios sin barreras económicas.

Al centralizar la interacción en un entorno inclusivo, Predi fomenta la colaboración entre personas de diversas nacionalidades, edades y contextos socioeconómicos, promoviendo un sentido de pertenencia en una audiencia que abarca millones de seguidores alrededor del mundo.

La accesibilidad representa un pilar fundamental de esta contribución social, al eliminar los costos asociados con eventos presenciales o plataformas premium, comúnmente vinculados con la Fórmula 1, deporte frecuentemente percibido como exclusivo. La optimización de la plataforma para dispositivos de gama media y baja posibilita la participación activa de usuarios con recursos tecnológicos limitados. Esta optimización contribuye a reducir la brecha digital, asegurando que incluso usuarios con conexiones a internet inestables puedan acceder a una experiencia fluida, democratizando el acceso a un espacio de entretenimiento de calidad.

Asimismo, Predi promueve la equidad al crear un entorno en el que aficionados de todos los géneros y orígenes culturales interactúan en condiciones de igualdad. En un deporte históricamente dominado por audiencias masculinas, la plataforma acoge a la creciente comunidad femenina de seguidores, así como a subcomunidades especializadas que apoyan a pilotos, equipos o países específicos. La funcionalidad de grupos privados facilita la formación de estas conexiones, permitiendo a los usuarios construir redes sociales basadas en intereses compartidos, fortaleciendo la identidad colectiva y el bienestar emocional de los participantes.

Conclusión

El desarrollo de Predi ha sido una experiencia transformadora, tanto en el ámbito técnico como personal, que me permitió fusionar mi pasión por la Fórmula 1 con los conocimientos adquiridos a lo largo de mi carrera en Ingeniería de Sistemas. Este proyecto, resultado de meses de investigación, planificación y codificación, creo que no solo cumplió con los objetivos propuestos, sino que también me desafió a salir de mi zona de confort, enfrentarme a problemas complejos y buscar soluciones de manera autónoma, ayudándome a crecer y ganar confianza para el ámbito profesional.

Uno de los aprendizajes más valiosos fue la gestión del tiempo, una habilidad que perfeccioné al equilibrar las demandas técnicas y académicas del proyecto. En el aspecto técnico, profundizar en Golang fue un desafío enriquecedor. Enfrenté complicaciones significativas, especialmente al implementar el gateway con reverse proxy y autenticación JWT, lo que me obligó a investigar y experimentar hasta encontrar soluciones efectivas. Este proceso reforzó mi capacidad para resolver problemas de manera independiente, una lección que considero fundamental para mi futuro profesional. El desarrollo del frontend con React y Tailwind CSS, aunque más exigente de lo esperado, me dejó satisfecho con el resultado: una interfaz intuitiva y optimizada para dispositivos móviles que refleja el enfoque mobile-first de Predi. Este proyecto me permitió aplicar casi todos los conocimientos adquiridos en la carrera, desde diseño de bases de datos hasta arquitectura de microservicios, demostrando que la universidad me preparó para enfrentar retos del mundo real, pero también que la verdadera innovación surge al buscar soluciones más allá del aula.

En cuanto a los objetivos, Predi cumplió con éxito la mayoría de los establecidos. El sistema de pronósticos, los grupos privados y la integración de datos oficiales de Fórmula 1 a través de la API de OpenF1.org funcionan como se esperaba, ofreciendo a los usuarios una experiencia interactiva y accesible. El foro, aunque operativo, podría beneficiarse de mejoras

futuras, como notificaciones en tiempo real, que no se implementaron por limitaciones de tiempo. Una restricción externa que enfrenté fue la escasez de APIs abiertas al inicio del proyecto, lo que me obligó a adaptar las entidades de la base de datos a la estructura de OpenF1.org, en lugar de modelarlas completamente a mi criterio. Mis tutores, Ing. Ignacio Carreño e Ing. Federico Porrini, fueron fundamentales en este aspecto, guiándome en el diseño de la base de datos relacional con su experiencia y ayudándome a tomar decisiones informadas que optimizaron la estructura del sistema.

A nivel personal, Predi es mucho más que un proyecto académico: es la materialización de un sueño que comenzó en mi infancia, haciendo pronósticos caseros con amigos y familia. Transformar esa pasión en una plataforma que conecta a aficionados de todo el mundo me llena de orgullo y esperanza. Ver a Predi funcionando, con su potencial para unir a la comunidad global de Fórmula 1, reafirma mi creencia en el poder de la tecnología para crear experiencias significativas. Este proceso no estuvo exento de retos, pero cada obstáculo me enseñó a perseverar y a valorar el aprendizaje que surge de la práctica.

De cara al futuro, veo a Predi como una plataforma con un gran potencial de expansión. La posibilidad de convertirla en una aplicación móvil, incorporar nuevas funcionalidades como integración con redes sociales, o incluso extenderla a otras categorías de automovilismo, me entusiasma. Espero que Predi inspire a otros estudiantes o desarrolladores a continuar su evolución, añadiendo innovaciones que lo lleven más lejos. Este proyecto me ha preparado para abordar desafíos profesionales con una mentalidad práctica y creativa, y me ha mostrado que combinar pasión y tecnología puede generar un impacto real..

Bibliografía

Historia de la Fórmula 1 - https://www.abc.es/deportes/formula-1/abci-historia-formula-1-202007141357_reportaje.html

Formato de la Fórmula 1 - https://www.sportingnews.com/ar/formula-1/news/guia-de-la-formula-1-para-principiantes-sistema-puntuacion-sprint-salarios-paradas-boxes-mas/jbf1nmywxgn4rxbwcn4htakp

Golang - https://www.mytaskpanel.com/lenguaje-programacion-go/

Golang - https://go.dev/doc/security/best-practices

Python - https://keepcoding.io/blog/ventajas-y-desventajas-de-python/

Golang vs Python - https://www.bmc.com/blogs/go-vs-python/

Golang vs Java - https://www.semanticscholar.org/paper/Concurrency-in-Go-and-Java%3A-Performance-analysis-Togashi-Klyuev/4b73e80c19f9cbb3881379f73e4bb134ea9d3cf8

Effective Go - https://go.dev/doc/effective_go

Gin - https://dev.to/leapcell/performance-best-practices-with-gin-25ci

GORM - https://tillitsdone.com/blogs/gorm-model-and-query-best-practices/

GORM - https://gorm.io/docs/index.html

GORM - https://www.pingcap.com/article/building-robust-go-applications-with-gorm-best-practices/

React - https://technostacks.com/blog/react-best-practices/

Tailwind CSS - https://www.uxpin.com/studio/blog/tailwind-best-practices/

Vite - https://dev.to/oppaaaii/best-practices-for-using-typescript-in-react-with-vite-1dhf

MySQL - https://www.geeksforgeeks.org/best-practices-for-mysql-database-connection-management/

Vercel - https://kapsys.io/user-experience/deploying-to-vercel-step-by-step-tutorial

Docker - https://snyk.io/es/blog/10-docker-image-security-best-practices/

Digital Ocean - https://appmaster.io/blog/how-to-secure-your-digitalocean-cloud

Open F1 - https://openf1.org/

Axios - https://climbtheladder.com/10-axios-best-practices/

JWT Token - https://blog.bitsrc.io/best-practices-for-using-jwt-df3788433fd3?gi=6525e1905fcc

Reverse Proxy - https://www.cloudflare.com/learning/cdn/glossary/reverse-proxy/

Reverse Proxy - https://kinsta.com/blog/reverse-proxy/

CORS - https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/CORS

Arquitecturas de Software - https://www.redhat.com/en/blog/14-software-architecture-patterns

MVC - https://www.codecademy.com/article/mvc

Microservicios - https://konghq.com/blog/learning-center/what-are-microservices