ADAF Tennis Simulator, una aplicación web para simular partidos de tenis
ADAF Tennis Simulator: a web application for simulating tennis matches

Trabajo de Fin de Grado
Curso 2025–2026

Autor

Álvaro Adrada Martínez-Flórez
Diego Fernández Albert

Director

Ramón González del Campo Rodríguez Barbero

Grado en Ingeniería del Software
Facultad de Informática
Universidad Complutense de Madrid

ADAF Tennis Simulator, una aplicación web para simular partidos de tenis
ADAF Tennis Simulator: a web application for simulating tennis matches

Trabajo de Fin de Grado en Ingeniería del Software

Autor

Álvaro Adrada Martínez-Flórez

Diego Fernández Albert

Director

Ramón González del Campo Rodríguez Barbero

Convocatoria: *Junio 2026*

Calificación final individual:

Álvaro Adrada Martínez-Flórez: *9.0*

Diego Fernández Albert: *9.0*

Grado en Ingeniería del Software

Facultad de Informática

Universidad Complutense de Madrid

11 de mayo de 2026

Dedicatoria

De Álvaro:

A mi familia, por su apoyo incondicional.
A mis padres, por su cariño infinito, por confiar siempre en mí y por animarme a seguir adelante durante este camino.
A mis hermanos, por las risas y el cariño de cada día.
A mi novia, por todo lo que has significado para mí y por todo lo que has hecho por mí durante esta etapa tan bonita.
A mis amigos de siempre, por estar ahí en todo momento.
A los amigos que me ha dado esta etapa, por hacerla mucho mejor y por todos los recuerdos que me llevo conmigo.

De Diego:
A mis padres, por apoyarme con los exámenes y animarme con la carrera.
A mi hermana, por las conversaciones tan graciosas y los ánimos diarios.
A mis amigos de siempre, por ser familia de padres distintos, y a todas aquellas personas que he conocido durante esta etapa.
Y a mi novia, por apoyarme siempre.

Agradecimientos

Álvaro

Quiero agradecer a todas las personas que me han acompañado durante estos años de carrera y que, de una forma u otra, han hecho más llevadero el camino hasta este trabajo.

En primer lugar, agradezco a los profesores de la facultad que me han transmitido conocimientos y me han ayudado a formarme durante esta etapa universitaria.

También quiero dar las gracias a mis compañeros de carrera, por el apoyo, la ayuda en los momentos difíciles y todos los buenos momentos compartidos a lo largo de estos años.

Por último, quiero agradecer a mi familia su apoyo constante durante todo este proceso. Su cariño, paciencia y confianza han sido una parte fundamental para poder llegar hasta aquí.

Diego

Deseo dejar constancia de mi agradecimiento a todas las personas que han contribuido, directa o indirectamente, a mi formación y a la realización de este Trabajo de Fin de Grado.

A mis compañeros, por el compañerismo demostrado y el aprendizaje conjunto.

Y, de manera muy especial, a mi familia, por su respaldo constante, su dedicación y su apoyo incondicional en cada paso de mi trayectoria académica.

Resumen

ADAF Tennis Simulator, una aplicación web para simular partidos de tenis

Nuestro Trabajo de Fin de Grado consiste en el diseño e implementación de una aplicación web de simulación de tenis denominada ADAF Tennis Simulator. La aplicación permite crear jugadores personalizados, simular partidos completos punto a punto y analizar los resultados mediante estadísticas detalladas.

El sistema se basa en un motor de simulación probabilístico desarrollado en Python que modela el transcurso de un partido teniendo en cuenta factores técnicos como la calidad del saque o la consistencia, y factores dinámicos como la fatiga y el estado mental. El backend está construido con FastAPI y PostgreSQL, y la interfaz web con Jinja2, Tailwind CSS y JavaScript.

Además de la simulación estándar, la aplicación ofrece un modo estratégico en el que el usuario toma decisiones tácticas punto a punto, un modo de análisis masivo que simula miles de partidos, y un sistema de torneos eliminatorios. El proyecto incluye autenticación con JWT y pruebas automatizadas que validan el comportamiento del motor.

Palabras clave

Tenis, Simulador, Aplicación web, Python, FastAPI, PostgreSQL, JavaScript, Modelo probabilístico, Sistema de usuarios, Visualización en tiempo real.

Abstract

ADAF Tennis Simulator: a web application for simulating tennis matches

This Bachelor's Thesis presents the design and implementation of a web-based tennis simulation application called ADAF Tennis Simulator. The application allows users to create custom players defined by numerical attributes, simulate full matches point by point, and analyse the results through detailed statistics.

The core of the system is a probabilistic simulation engine developed in Python that models the progression of a tennis match by combining technical factors such as serve quality and baseline consistency with dynamic variables like accumulated fatigue and the player's mental state. The backend is built with FastAPI and PostgreSQL, while the frontend uses Jinja2, Tailwind CSS and JavaScript to render the match in real time with a textual commentary of each point.

Beyond the standard simulation, the application offers three advanced modes: a strategic mode in which the user makes tactical decisions point by point, a Big Data mode that runs thousands of matches and extracts statistical trends, and an elimination tournament system with a full bracket and automatic round progression. The project also includes JWT-based user authentication and an automated test suite that validates the behaviour of the simulation engine.

Keywords

Tennis, simulator, web application, Python, FastAPI, PostgreSQL, JavaScript, probabilistic model, user authentication, real-time visualisation.

Índice

1. Introducción **1**

1.1. Motivación . . . 1

1.2. Planteamiento del problema . . . 2

1.3. Objetivos del proyecto . . . 2

1.3.1. Objetivo general . . . 2

1.3.2. Objetivos específicos . . . 2

1.4. Alcance del proyecto . . . 3

1.5. Metodología y plan de trabajo . . . 3

2. Contexto del problema y fundamentos del simulador **5**

2.1. El tenis como sistema de puntuación y competición . . . 5

2.2. Elementos clave de un partido de tenis . . . 6

2.2.1. Puntuación: puntos, juegos, sets y partido . . . 6

2.2.2. *Tie-break* y situaciones de presión . . . 6

2.2.3. Factores que influyen en el rendimiento de un jugador . . . 6

2.3. Simulación deportiva y modelos probabilísticos . . . 7

2.4. Enfoque adoptado en el proyecto . . . 7

3. Análisis de requisitos **9**

3.1. Visión general del sistema . . . 9

3.2. Tipos de usuario . . . 9

3.3. Requisitos funcionales . . . 10

3.3.1. Gestión de usuarios y autenticación . . . 10

3.3.2. Gestión de jugadores . . . 11

3.3.3. Simulación de partidos . . . 12

3.3.4. Gestión de torneos . . . 13

3.3.5. Historial y consulta de estadísticas . . . 14

3.4. Requisitos no funcionales . . . 14

3.4.1. Usabilidad . . . 14

3.4.2. Seguridad y autenticación . . . 15

3.4.3. Reproducibilidad y mantenibilidad . . . 16

3.5. Casos de uso principales . . . 16

4. Diseño de la arquitectura del sistema **25**
4.1. Visión general de la arquitectura 25
4.2. Arquitectura cliente-servidor 26
4.3. Componentes principales 26
4.3.1. Frontend SSR con Jinja2 y JavaScript 26
4.3.2. Backend con FastAPI 27
4.3.3. Base de datos PostgreSQL 27
4.3.4. Sistema de autenticación con JWT 28
4.4. Flujo de interacción entre componentes 29
4.5. Decisiones de diseño y justificación tecnológica 29

5. Diseño del motor de simulación **33**
5.1. Objetivos del motor de simulación 33
5.2. Modelado del jugador 34
5.2.1. Atributos estáticos 34
5.2.2. Variables dinámicas: estamina y momentum 35
5.3. Modelado del punto 37
5.3.1. Primer saque 38
5.3.2. Segundo saque 39
5.3.3. Resto 40
5.3.4. Rally golpe a golpe 42
5.3.5. Finalización del punto 43
5.4. Influencia del clutch en puntos clave 44
5.5. Reglas de puntuación implementadas 45
5.5.1. Juego estándar 45
5.5.2. *Tie-break* 45
5.5.3. Set 46
5.5.4. Partido al mejor de tres y al mejor de cinco 46
5.6. Control de aleatoriedad y reproducibilidad 46
5.7. Limitaciones actuales del modelo 47

6. Implementación del sistema **49**
6.1. Implementación del backend 49
6.1.1. Organización del proyecto 49
6.1.2. Endpoints principales 51
6.1.3. Serialización y estructura de respuestas 52
6.1.4. Implementación del CRUD de jugadores 52
6.2. Implementación del frontend 53
6.2.1. Estructura de vistas SSR 53
6.2.2. Interacción dinámica con JavaScript 59
6.2.3. Marcador en tiempo real 59
6.2.4. Feed narrativo punto a punto 60
6.2.5. Visualización de estadísticas con Chart.js 60
6.3. Implementación del sistema de autenticación 61

6.3.1. Registro y login . . . 61
6.3.2. Gestión de tokens . . . 61
6.3.3. Protección de rutas y sesiones . . . 62

7. Persistencia de datos y funcionalidades avanzadas 63
7.1. Diseño de la base de datos . . . 63
7.1.1. Modelo entidad-relación . . . 63
7.1.2. Tablas principales . . . 64
7.1.3. Relaciones entre usuarios, jugadores, partidos y torneos . . . 69
7.2. Historial de partidos y almacenamiento de estadísticas . . . 69
7.3. Simulación de torneos eliminatorios . . . 70
7.4. Modos de uso del sistema . . . 70
7.4.1. Partido rápido . . . 70
7.4.2. Modo estratégico . . . 71
7.4.3. Modo Big Data . . . 71

8. Pruebas y validación 73
8.1. Estrategia de validación . . . 73
8.2. Pruebas manuales de la aplicación . . . 73
8.3. Pruebas automatizadas del motor y del sistema . . . 75

9. Conclusiones y trabajo futuro 77
9.1. Conclusiones generales . . . 77
9.2. Principales aportaciones del proyecto . . . 77
9.3. Limitaciones actuales y trabajo futuro . . . 78

Introduction 81

Contribuciones Personales 85

A. Repositorio del proyecto 91

B. Ejecución de la aplicación en local 93
B.1. Requisitos previos . . . 93
B.2. Obtención del código fuente . . . 93
B.3. Preparación de la base de datos . . . 94
B.4. Configuración de variables de entorno . . . 94
B.5. Creación del entorno virtual . . . 95
B.6. Instalación de dependencias . . . 95
B.7. Ejecución del servidor . . . 95
B.8. Documentación interactiva de la API . . . 96
B.9. Ejecución de pruebas . . . 96

Índice de figuras

5.1. Diagrama de estados del desarrollo de un punto en el motor de simulación. 37

6.1. Organización de carpetas del backend 50
6.2. Página de inicio de la aplicación 54
6.3. Menú principal de la aplicación 54
6.4. Formulario de creación de jugador 55
6.5. Formulario de configuración de partido rápido 55
6.6. Vista de reproducción de un partido rápido 56
6.7. Vista de estadísticas de un partido 56
6.8. Vista del historial de partidos de un usuario 57
6.9. Gráficos de estadísticas de un partido 57
6.10. Vista de creación de torneo 58
6.11. Vista del cuadro de torneo 58
6.12. Vista del perfil del usuario 59

7.1. Modelo entidad-relación de la base de datos 64

Índice de tablas

3.1. Caso de uso: Registrarse en el sistema 16
3.2. Caso de uso: Iniciar sesión 17
3.3. Caso de uso: Crear un jugador personalizado 18
3.4. Caso de uso: Simular un partido 19
3.5. Caso de uso: Organizar un torneo 20
3.6. Caso de uso: Consultar el historial de un jugador 21
3.7. Caso de uso: Ejecutar análisis Big Data 22
3.8. Caso de uso: Simular un partido en modo estratégico 23

5.1. Atributos técnicos del jugador y significado dentro del modelo. 35

7.1. Tabla `usuarios` 65
7.2. Tabla `jugadores` 66
7.3. Tabla `partidos` 67
7.4. Tabla `estadisticas_partido` 67
7.5. Tabla `torneos` 68
7.6. Tabla `torneo_partidos` 68

8.1. Resumen de pruebas manuales realizadas sobre la aplicación 74

Capítulo 1

Introducción

1.1. Motivación

La motivación principal de este proyecto nace de nuestro interés personal por el tenis. Desde pequeños hemos practicado y seguido este deporte, por lo que, cuando se nos propuso desarrollar un simulador deportivo como Trabajo de Fin de Grado, vimos claro que el tenis podía ser una buena elección. Era un deporte que conocíamos de cerca y que, además, ofrecía suficientes elementos técnicos y estratégicos como para plantear una simulación interesante.

Elegir el tenis nos permitía trabajar sobre un sistema de puntuación diferente al de otros deportes, con puntos, juegos, sets, *tie-breaks* y situaciones de presión. También nos parecía interesante intentar representar aspectos propios del juego, como la importancia del saque, la consistencia durante los intercambios, la fatiga acumulada o la influencia de los puntos importantes. En tenis, un jugador puede ser superior en términos generales y aun así perder momentos clave que cambian por completo el desarrollo del partido.

Desde el inicio queríamos que el simulador no se limitase a calcular un porcentaje general de victoria o a resolver el partido de forma directa. La idea era construir una simulación lo más desglosada posible, en la que el partido se generase punto a punto y, dentro de cada punto, golpe a golpe. De esta manera, el resultado final no aparecería como un dato aislado, sino como consecuencia de todo lo que hubiera ocurrido durante el encuentro.

Además, queríamos que el proyecto combinase desarrollo web con una parte algorítmica propia. No se trataba únicamente de crear formularios y guardar datos en una base de datos, sino de desarrollar un sistema capaz de generar partidos, reproducirlos visualmente y extraer estadísticas a partir de esa simulación. Esta combinación entre aplicación web, motor probabilístico y visualización interactiva fue una de las razones por las que el proyecto nos resultó especialmente atractivo.

1.2. Planteamiento del problema

Una vez definida la idea de desarrollar un simulador de tenis, el siguiente paso fue analizar qué debía aportar el sistema. Existen herramientas orientadas a la predicción estadística de partidos (17), pero muchas de ellas se centran en estimar probabilidades generales de victoria a partir de datos históricos. Este enfoque puede ser útil para el análisis, pero no permite al usuario observar cómo se construye el partido ni experimentar con jugadores creados por él mismo.

El problema que se plantea en este proyecto es, por tanto, construir un simulador que no resuelva el partido únicamente mediante una probabilidad final, sino que genere el encuentro de forma progresiva. Para ello, el sistema debe simular el desarrollo del partido punto a punto y, dentro de cada punto, representar fases como el saque, el resto y el peloteo. Este enfoque permite que el resultado final sea consecuencia de una secuencia de acciones, errores, golpes ganadores, rachas y situaciones de presión.

Además, el simulador debía estar integrado en una aplicación web completa. Esto implicaba que el usuario pudiera crear jugadores con atributos personalizados, configurar partidos, visualizar el marcador en tiempo real, consultar estadísticas, disputar torneos y utilizar modos adicionales como el modo estratégico o el análisis masivo de simulaciones.

Por tanto, el reto del proyecto no se limitaba al diseño de un motor probabilístico, sino que incluía varios frentes de trabajo: la construcción del modelo de simulación, la organización del backend, la persistencia de datos, la autenticación de usuarios y el desarrollo de una interfaz clara e interactiva. La dificultad principal consistía en integrar todas estas partes en un sistema coherente y usable para el usuario final.

1.3. Objetivos del proyecto

1.3.1. Objetivo general

Desarrollar una aplicación web completa que permita crear jugadores personalizados, simular partidos y torneos de tenis, visualizar el desarrollo del partido en tiempo real y analizar los resultados mediante estadísticas.

1.3.2. Objetivos específicos

- Implementar un motor de simulación punto a punto basado en atributos numéricos y variables dinámicas, como la fatiga y el estado mental, que permita representar de forma coherente el desarrollo de un partido de tenis.

- Diseñar una base de datos relacional que permita almacenar usuarios, jugadores, partidos, estadísticas y torneos, manteniendo la información asociada a cada usuario.

- Desarrollar una API con FastAPI que exponga las funcionalidades principales del simulador de forma organizada.

- Construir una interfaz web que permita al usuario crear jugadores, configurar partidos, seguir el marcador actualizado punto a punto y visualizar la narración de cada jugada.
- Añadir modos de uso avanzados, como un modo estratégico en el que el usuario toma decisiones durante el partido, un modo de análisis masivo que simula múltiples partidos para extraer tendencias estadísticas y un sistema de torneos eliminatorios con avance automático de rondas.
- Validar el sistema mediante pruebas manuales y automatizadas, comprobando que el motor de simulación se comporta de forma coherente y reproducible, y que las funcionalidades principales de la aplicación responden correctamente.

1.4. Alcance del proyecto

El proyecto cubre el diseño e implementación de una aplicación web funcional, incluyendo el motor de simulación, el backend, la base de datos, el frontend y el sistema de autenticación. También se incluyen funcionalidades como la creación de jugadores personalizados, la simulación de partidos, los torneos eliminatorios, el modo estratégico, el análisis masivo de simulaciones y la consulta de resultados.

1.5. Metodología y plan de trabajo

El desarrollo se llevó a cabo de forma iterativa e incremental. En primer lugar, se implementó el motor de simulación de forma aislada, con el objetivo de comprobar que podía generar partidos completos y resultados coherentes. Después, se desarrollaron la API y la base de datos para poder gestionar usuarios, jugadores, partidos y torneos.

Una vez construida la parte principal del backend, se desarrolló la interfaz web mediante plantillas Jinja2 y JavaScript. En las fases finales se incorporaron las funcionalidades avanzadas, como el modo estratégico, el modo Big Data y los torneos eliminatorios. Por último, se realizaron pruebas manuales y automatizadas para validar el comportamiento del sistema.

Durante el desarrollo se utilizó Git como sistema de control de versiones, lo que permitió mantener un seguimiento de los cambios realizados y trabajar de forma ordenada sobre las distintas partes del proyecto.

Capítulo 2

Contexto del problema y fundamentos del simulador

En este capítulo se introduce el contexto general del proyecto. Para ello, se explica primero el funcionamiento básico del tenis como deporte competitivo, prestando especial atención a su sistema de puntuación y a los elementos que influyen en el desarrollo de un partido. Después, se presentan algunas ideas generales sobre la simulación deportiva y los modelos probabilísticos. Finalmente, se describe el enfoque adoptado en ADAF Tennis Simulator.

2.1. El tenis como sistema de puntuación y competición

El tenis es un deporte de raqueta que puede disputarse entre dos jugadores, en la modalidad individual, o entre dos parejas, en la modalidad de dobles. En este proyecto se trabaja únicamente con partidos individuales, ya que el objetivo principal es simular el enfrentamiento entre dos tenistas.

Una de las características más importantes del tenis es su sistema de puntuación. A diferencia de otros deportes, los puntos no se acumulan de forma lineal hasta alcanzar una puntuación final, sino que se organizan en varios niveles: puntos, juegos, sets y partido. Esta estructura hace que no todos los puntos tengan la misma importancia, ya que algunos pueden decidir un juego, un set o incluso el partido completo.

Desde el punto de vista de la simulación, esta jerarquía de puntuación resulta especialmente interesante, porque obliga a modelar no solo quién gana cada punto, sino también cómo ese punto afecta al marcador global del encuentro.

2.2. Elementos clave de un partido de tenis

2.2.1. Puntuación: puntos, juegos, sets y partido

La unidad mínima de juego es el **punto**. Cada punto ganado modifica el marcador del juego actual. En un juego estándar, la puntuación se representa con la nomenclatura tradicional del tenis: 0, 15, 30, 40 y juego ganado.

Para ganar un juego, un jugador debe conseguir al menos cuatro puntos y mantener una diferencia mínima de dos puntos sobre su rival. Si ambos jugadores alcanzan 40, se entra en una situación de iguales, conocida como *deuce*. A partir de ese momento, un jugador debe ganar dos puntos consecutivos para cerrar el juego: el primero le da ventaja y el segundo le permite ganar el juego.

Los juegos se agrupan en **sets**. Normalmente, un set se gana al alcanzar seis juegos con una diferencia mínima de dos. Si el marcador llega a 6–6, puede disputarse un *tie-break*, dependiendo del formato elegido. Finalmente, el partido se decide al mejor de un número determinado de sets, normalmente al mejor de tres o al mejor de cinco.

2.2.2. *Tie-break* y situaciones de presión

El *tie-break* es un juego especial que se disputa cuando un set llega a 6–6. En este caso, los puntos se cuentan de forma numérica directa, es decir, 1, 2, 3, etc. El objetivo habitual es alcanzar siete puntos con una diferencia mínima de dos.

El servicio también sigue una rotación distinta a la de un juego normal. El primer jugador sirve un único punto y, a partir de ese momento, cada jugador sirve dos puntos consecutivos por turno. Esta regla hace que el *tie-break* tenga una dinámica diferente y que cada punto tenga un peso importante en el resultado del set.

Además del *tie-break*, existen otras situaciones de presión dentro de un partido. Un ejemplo claro son los puntos de ruptura o *break points*, en los que el restador tiene la oportunidad de ganar el juego al jugador que está sacando. También pueden considerarse puntos importantes las situaciones de ventaja, los puntos de set o los puntos de partido. En este tipo de momentos, el componente mental puede influir en el rendimiento del jugador.

2.2.3. Factores que influyen en el rendimiento de un jugador

El rendimiento de un tenista durante un partido depende de varios factores. Algunos de ellos son relativamente estables y forman parte del perfil técnico del jugador, como la calidad del saque, la capacidad de restar, la consistencia desde el fondo de la pista, la movilidad o la calidad de sus golpes de derecha y revés.

Sin embargo, no todos los factores permanecen constantes durante el partido. El desgaste físico acumulado puede afectar al rendimiento de un jugador, especialmente en intercambios largos o partidos de varios sets. Del mismo modo, el estado mental también puede modificar el desarrollo del encuentro, ya que una racha positiva puede aumentar la confianza del jugador, mientras que una secuencia de puntos perdidos puede afectar negativamente a su rendimiento.

La superficie de juego también es un factor relevante en el tenis real, ya que no se juega igual en tierra batida, hierba o pista dura. Cada superficie modifica aspectos como la velocidad de la bola, el bote o la duración media de los puntos. En este proyecto, la superficie se incluye como parámetro de configuración del partido, aunque su influencia física sobre el motor no se modela de forma específica en la versión actual, tal y como se explica en las limitaciones del modelo.

2.3. Simulación deportiva y modelos probabilísticos

La simulación deportiva consiste en representar mediante un modelo computacional el comportamiento de una competición o de una parte de ella. En el caso del tenis, una simulación puede centrarse únicamente en el resultado final del partido o, de forma más detallada, en la evolución punto a punto del encuentro.

Existen distintos enfoques para construir simuladores deportivos. Uno de ellos consiste en utilizar datos reales de jugadores y partidos anteriores. A partir de estadísticas históricas, como porcentajes de primer saque, puntos ganados al resto o resultados previos, se pueden estimar probabilidades de victoria o de ganar determinados puntos. Este enfoque puede ofrecer un alto grado de realismo, pero depende de disponer de datos completos y fiables (17).

Otro enfoque es el uso de modelos paramétricos. En este caso, cada jugador se representa mediante un conjunto de atributos numéricos, y el sistema calcula las probabilidades de cada acción combinando esos atributos mediante fórmulas. Este tipo de modelo es habitual cuando se quiere permitir la creación de jugadores ficticios o cuando no se dispone de una base de datos real suficientemente amplia.

También existen modelos probabilísticos más formales, como los basados en cadenas de Markov (18; 19), que resultan útiles para analizar deportes con estados bien definidos. En tenis, el marcador puede verse como una sucesión de estados que cambian después de cada punto. Sin embargo, este tipo de modelos suele centrarse más en la probabilidad de ganar juegos, sets o partidos que en la generación detallada de cada golpe.

2.4. Enfoque adoptado en el proyecto

ADAF Tennis Simulator adopta un enfoque paramétrico y probabilístico. Cada jugador se describe mediante nueve atributos principales en una escala de 1 a 100, que representan aspectos como el saque, el resto, la movilidad, la consistencia, la derecha, el revés, el físico y el rendimiento bajo presión.

A partir de estos atributos, el motor calcula el desarrollo de cada punto mediante funciones probabilísticas. Además, el modelo incorpora variables dinámicas como la estamina y el *momentum*, que evolucionan durante el partido y permiten que el rendimiento del jugador no dependa únicamente de sus valores iniciales.

Se ha elegido este enfoque por varias razones. En primer lugar, permite crear jugadores personalizados sin depender de datos reales de tenistas profesionales. En segundo lugar, facilita que el usuario entienda el significado de cada atributo y pue-

da modificarlo para observar cómo afecta al rendimiento del jugador. Por último, permite generar partidos variados, ya que el resultado no está completamente determinado por los atributos iniciales.

El objetivo del modelo no es reproducir con exactitud el comportamiento de jugadores reales del circuito profesional, sino generar simulaciones coherentes, variadas y comprensibles para el usuario. De esta forma, el sistema combina una estructura basada en reglas de tenis con un componente probabilístico que introduce incertidumbre en el resultado de cada punto.

Capítulo 3

Análisis de requisitos

3.1. Visión general del sistema

ADAF Tennis Simulator es una aplicación web que permite simular partidos de tenis entre jugadores personalizados. El usuario configura los jugadores, la superficie de juego y el formato del partido, y el sistema ejecuta la simulación de forma automática, generando una narración punto a punto y estadísticas detalladas del encuentro.

Además de la simulación individual, la aplicación ofrece un modo torneo, un modo estratégico en el que el usuario toma decisiones tácticas durante el partido, y un modo de análisis que ejecuta un gran número de simulaciones para comparar el rendimiento estadístico de dos jugadores.

3.2. Tipos de usuario

El sistema contempla dos tipos de usuario con capacidades diferenciadas:

- **Usuario no registrado.**

  Es cualquier persona que accede a la aplicación sin haber iniciado sesión. Este tipo de usuario puede acceder a la funcionalidad de partido rápido, que permite simular un encuentro entre jugadores predefinidos en el sistema sin necesidad de disponer de una cuenta. También puede navegar por las páginas públicas de la aplicación, como la página de inicio, el formulario de registro y el formulario de acceso.

  Sin embargo, no puede crear jugadores propios, guardar resultados ni consultar historiales. El objetivo de este modo es que cualquier persona pueda probar la simulación sin necesidad de realizar un registro previo.

- **Usuario registrado.**

  Es el tipo de usuario principal del sistema. Dispone de una cuenta creada mediante el formulario de registro y accede a la aplicación mediante sus credenciales. Un usuario registrado puede:

- Crear y gestionar su propia lista de tenistas personalizados.
- Simular partidos entre sus tenistas o entre tenistas predefinidos del sistema.
- Consultar el historial completo de los partidos que ha simulado.
- Acceder a estadísticas detalladas de cada enfrentamiento.
- Organizar y disputar torneos.
- Usar el modo estratégico y el modo de análisis *Big Data*.
- Consultar la información de su cuenta.

No existe un rol de administrador explícito con panel de control dentro de la aplicación.

3.3. Requisitos funcionales

Los requisitos funcionales describen qué debe ser capaz de hacer el sistema, es decir, las acciones concretas que los usuarios pueden realizar a través de la aplicación.

3.3.1. Gestión de usuarios y autenticación

En este apartado se recogen los requisitos relacionados con el registro, el acceso y la protección de las funcionalidades privadas de la aplicación.

- **RF01. Registro de cuenta**

  El sistema debe permitir que cualquier visitante cree una cuenta proporcionando un nombre de usuario único, una dirección de correo electrónico válida y una contraseña de al menos ocho caracteres. De forma opcional, el usuario también puede indicar otros datos personales, como nombre, apellido, nacionalidad y género.

  El sistema debe rechazar el registro si el nombre de usuario o el correo electrónico ya están en uso, informando al usuario del motivo del error.

- **RF02. Inicio de sesión**

  El sistema debe permitir que un usuario registrado inicie sesión utilizando indistintamente su nombre de usuario o su dirección de correo electrónico, junto con su contraseña.

- **RF03. Cierre de sesión**

  El usuario debe poder cerrar su sesión en cualquier momento. Tras el cierre de sesión, el token utilizado para mantener la sesión deja de estar disponible en el cliente, por lo que el usuario pierde el acceso a las funcionalidades privadas hasta que vuelva a iniciar sesión.

- **RF04. Consulta del perfil**

  El sistema debe ofrecer a cada usuario registrado una página de perfil donde pueda consultar su información personal, como el nombre, el correo electrónico y la fecha de registro. Además, esta página debe mostrar un resumen de sus jugadores y partidos asociados.

- **RF05. Protección de rutas**

  El acceso a cualquier funcionalidad privada, como la creación de jugadores, la simulación de partidos o la consulta del historial, debe requerir una sesión activa. Si un usuario no autenticado intenta acceder a una de estas rutas, el sistema debe redirigirle a la página de inicio de sesión.

3.3.2. Gestión de jugadores

En este apartado se recogen los requisitos relacionados con la creación, consulta y selección de jugadores dentro de la aplicación.

- **RF06. Creación de jugadores**

  El usuario registrado debe poder crear tenistas personalizados. Para ello, el sistema debe solicitar la siguiente información:

  - Nombre y apellidos del jugador.
  - Edad y altura, expresada en centímetros.
  - Nacionalidad y brazo dominante, pudiendo ser derecho o izquierdo.
  - Nueve atributos numéricos en una escala del 1 al 100, que representan las principales características del tenista: primer saque, segundo saque, resto, golpe de derecha, golpe de revés, movilidad, consistencia, factor clutch y condición física.

  Además de estos atributos configurables, durante la simulación se utilizan variables dinámicas como la estamina y el momentum. Estas no son introducidas manualmente por el usuario, sino que se inicializan al comienzo del partido y evolucionan en función del desarrollo del encuentro.

- **RF07. Listado de jugadores**

  El usuario debe poder consultar la lista de jugadores que ha creado. Además, el sistema debe proporcionar un catálogo de jugadores por defecto que cualquier usuario pueda utilizar en las simulaciones.

- **RF08. Selección de jugadores para partidos**

  En cualquier modo de simulación que requiera la selección de jugadores, el sistema debe mostrar al usuario tanto sus jugadores creados como los jugadores predefinidos del sistema, permitiendo elegir entre ellos.

3.3.3. Simulación de partidos

En este apartado se recogen los requisitos relacionados con la configuración, ejecución y visualización de los partidos simulados.

- **RF09. Partido rápido sin registro**

  El sistema debe ofrecer un modo de simulación accesible sin necesidad de iniciar sesión, en el que el usuario pueda enfrentar a jugadores predefinidos sobre la superficie y con el formato que elija. El resultado de estos partidos no se almacena en la base de datos.

- **RF10. Partido rápido para usuario registrado**

  El usuario registrado debe poder configurar y simular un partido completo indicando los siguientes parámetros:

  - Ambos jugadores.
  - Superficie de juego: tierra batida, pista dura o hierba.
  - Formato del partido: al mejor de 1, 3 o 5 sets.
  - Uso de *tie-break* en el set decisivo.

  Una vez configurado el partido, el sistema debe ejecutar la simulación y presentar el resultado mediante una animación del marcador y una narración punto a punto.

- **RF11. Visualización de la simulación**

  Durante el transcurso del partido simulado, el sistema debe mostrar la evolución del encuentro de forma comprensible para el usuario. Para ello, debe incluir:

  - El marcador actualizado, incluyendo sets ganados, juegos y puntos del juego en curso.
  - Una narración de lo que ocurre en cada punto, indicando acciones como aces, dobles faltas, winners o errores.
  - Información sobre el jugador que va por delante en el marcador.

  El usuario debe poder controlar la velocidad de reproducción de la simulación y pausar o reanudar el partido en cualquier momento.

- **RF12. Resumen y estadísticas del partido**

  Al finalizar el partido, el sistema debe presentar una pantalla de resumen con estadísticas comparativas entre ambos jugadores. Entre estas estadísticas se incluyen aces, dobles faltas, porcentaje de puntos ganados con primer y segundo saque, winners, errores no forzados y rendimiento por set.

  Cuando sea necesario para facilitar la interpretación de los datos, estas estadísticas podrán visualizarse mediante gráficas.

- **RF13. Modo estratégico**

  En este modo, el sistema debe permitir al usuario seleccionar la estrategia que desea aplicar a su jugador, pudiendo elegir entre una estrategia agresiva, neutral o defensiva. La estrategia seleccionada modifica los pesos probabilísticos utilizados en el cálculo del punto siguiente.

  El partido avanza punto a punto bajo la dirección del usuario, que puede ajustar su planteamiento táctico durante el desarrollo del encuentro.

- **RF14. Modo Big Data**

  El usuario debe poder solicitar la ejecución de varias simulaciones consecutivas del mismo enfrentamiento, dentro del rango permitido por la aplicación. El sistema ejecutará las repeticiones solicitadas y mostrará el avance del proceso.

  Al finalizar, debe presentar estadísticas agregadas, como el porcentaje de victorias de cada jugador, el promedio de aces, errores, duración de los puntos y distribución de resultados por set.

3.3.4. Gestión de torneos

En este apartado se recogen los requisitos relacionados con la creación, seguimiento y simulación de torneos dentro de la aplicación.

- **RF15. Creación de torneo**

  El usuario registrado debe poder crear un torneo configurando los siguientes datos:

  - Nombre del torneo.
  - Número de participantes: 4, 8 o 16 jugadores.
  - Jugadores que participarán en el torneo.
  - Superficie y formato de partido para todas las rondas.

  Una vez introducida esta información, el sistema debe generar automáticamente el cuadro inicial de emparejamientos.

- **RF16. Simulación de rondas**

  El usuario debe poder simular cada ronda del torneo de forma individual. Tras cada ronda, el sistema debe determinar los ganadores de los partidos y avanzarlos automáticamente a la siguiente fase del cuadro.

- **RF17. Consulta del cuadro del torneo**

  El usuario debe poder acceder en cualquier momento a la vista del cuadro del torneo para consultar qué partidos se han jugado, qué jugadores han avanzado y cuáles han quedado eliminados.

3.3.5. Historial y consulta de estadísticas

En este apartado se recogen los requisitos relacionados con el almacenamiento y la consulta de los resultados generados por las simulaciones.

- **RF18. Guardado automático de partidos**

  Cada partido simulado por un usuario registrado en una simulación completa debe guardarse automáticamente en la base de datos al finalizar, sin que el usuario tenga que realizar ninguna acción adicional.

- **RF19. Historial de partidos por jugador**

  El sistema debe permitir consultar el historial de partidos disputados por un jugador concreto, incluyendo información como la fecha, el rival, la superficie, el resultado final en sets y juegos, y el ganador.

- **RF20. Consulta del detalle de un partido**

  El usuario debe poder acceder al detalle de cualquier partido de su historial para consultar las estadísticas completas del encuentro. Estas estadísticas deben corresponderse con las mostradas en la pantalla de resumen generada al finalizar la simulación.

- **RF21. Historial del usuario**

  La página de perfil del usuario debe mostrar un resumen de los últimos partidos simulados, incluyendo un acceso rápido al detalle de cada uno.

3.4. Requisitos no funcionales

Los requisitos no funcionales describen las cualidades que debe tener el sistema. No se centran en las funcionalidades concretas que ofrece la aplicación, sino en la forma en que estas se proporcionan. Por tanto, afectan a aspectos como la facilidad de uso, la seguridad de los datos, el rendimiento y la mantenibilidad del sistema.

3.4.1. Usabilidad

- **RNF01. Interfaz intuitiva**

  La aplicación debe poder utilizarse sin necesidad de formación previa. Un usuario debe ser capaz de crear un jugador, configurar y lanzar una simulación, y consultar el resultado sin tener que recurrir a un manual externo.

- **RNF02. Diseño adaptable**

  La interfaz debe adaptarse correctamente a distintos tamaños de pantalla, especialmente ordenadores de escritorio, portátiles y tablets. Los formularios, el marcador y las gráficas deben mantenerse legibles y ser operables en estos dispositivos.

- **RNF03. Retroalimentación visual**

  El sistema debe informar visualmente al usuario del resultado de las acciones relevantes, como el inicio de sesión, los errores en formularios, el avance de una simulación masiva o la finalización de un partido. Los mensajes de error deben ser claros y estar redactados en un lenguaje comprensible para el usuario.

- **RNF04. Velocidad de respuesta**

  Las páginas principales de la aplicación deben cargarse de forma fluida en condiciones normales de uso. Del mismo modo, las acciones habituales, como acceder al perfil, consultar jugadores o configurar una simulación, deben ofrecer una respuesta suficientemente rápida para no dificultar la experiencia del usuario.

3.4.2. Seguridad y autenticación

- **RNF05. Contraseñas protegidas**

  Las contraseñas de los usuarios no deben almacenarse en texto plano. El sistema debe aplicar una función de hash con sal, como *bcrypt* (10), antes de guardar cualquier contraseña en la base de datos, de modo que no sea posible obtener directamente las contraseñas originales a partir de la información almacenada.

- **RNF06. Tokens de sesión seguros**

  La autenticación debe implementarse mediante tokens firmados digitalmente, concretamente mediante JWT (*JSON Web Token*) (11). Estos tokens deben tener una caducidad definida y no deben transmitirse como parámetros visibles en la URL. El servidor debe validar el token antes de conceder acceso a funcionalidades o datos privados.

- **RNF07. Aislamiento de datos entre usuarios**

  Ningún usuario debe poder acceder a los jugadores, partidos o historiales pertenecientes a otro usuario. Para ello, el sistema debe verificar la identidad del solicitante en cada operación de lectura o escritura sobre datos privados.

- **RNF08. Validación en servidor**

  Todas las entradas de datos procedentes del cliente, como formularios, parámetros de simulación o configuración de torneos, deben validarse en el servidor antes de ser procesadas. Esta validación debe realizarse independientemente de cualquier comprobación existente en el navegador.

- **RNF09. Comunicación segura**

  En un entorno de despliegue real, las comunicaciones entre el navegador y el servidor deben realizarse mediante HTTPS, evitando el envío de información sensible a través de conexiones no cifradas.

3.4.3. Reproducibilidad y mantenibilidad

- **RNF10. Simulaciones reproducibles**

  El motor de simulación debe estar diseñado de forma que facilite la reproducción de resultados durante las pruebas. Para ello, cuando se utilice una semilla aleatoria fija, una misma configuración de jugadores y partido debe producir el mismo resultado, lo que ayuda a depurar el modelo y verificar el comportamiento del sistema.

- **RNF11. Arquitectura modular**

  El código del sistema debe estar organizado en módulos con responsabilidades diferenciadas, como autenticación, gestión de jugadores, gestión de partidos, torneos, motor de simulación e interfaz. Esta organización facilita el mantenimiento del proyecto y permite introducir cambios en una parte del sistema reduciendo el impacto sobre el resto.

3.5. Casos de uso principales

A continuación se describen los escenarios de uso más representativos del sistema, redactados desde el punto de vista del usuario.

CU-01. Registrarse en el sistema

Actor principal	Visitante
Descripción	El usuario crea una cuenta en la aplicación introduciendo sus datos básicos de registro.
Precondición	El usuario no tiene una cuenta creada en el sistema.
Flujo principal	1. El usuario accede a la página de registro. 2. Rellena el nombre de usuario, el correo electrónico, la contraseña y los datos opcionales. 3. El sistema valida que los datos introducidos son correctos y que no existe otro usuario con el mismo nombre de usuario o correo electrónico. 4. El sistema crea la cuenta del usuario. 5. El usuario es redirigido a la página de inicio de sesión.
Flujo alternativo	Si el nombre de usuario o el correo electrónico ya están en uso, el sistema muestra un mensaje de error y solicita al usuario que introduzca datos distintos.
Postcondición	El usuario tiene una cuenta creada en el sistema, pero todavía debe iniciar sesión para acceder a las funcionalidades privadas.

Tabla 3.1: Caso de uso: Registrarse en el sistema

CU-02. Iniciar sesión

Actor principal	Usuario registrado
Descripción	El usuario accede a la aplicación introduciendo su nombre de usuario o correo electrónico y su contraseña.
Precondición	El usuario debe tener una cuenta creada y no haber iniciado sesión.
Flujo principal	1. El usuario accede al formulario de inicio de sesión. 2. Introduce su nombre de usuario o correo electrónico junto con su contraseña. 3. El sistema valida las credenciales introducidas. 4. Si son válidas, el sistema inicia la sesión y redirige al usuario al menú principal de la aplicación.
Flujo alternativo	Si las credenciales no son válidas, el sistema muestra un mensaje de error y permite al usuario volver a intentarlo.
Postcondición	El usuario tiene una sesión activa y puede acceder a las funcionalidades privadas.

Tabla 3.2: Caso de uso: Iniciar sesión

CU-03. Crear un jugador personalizado

Actor principal	Usuario registrado
Descripción	El usuario crea un tenista personalizado introduciendo sus datos personales y sus atributos deportivos.
Precondición	El usuario ha iniciado sesión.
Flujo principal	1. El usuario accede a la sección *Crear jugador*. 2. Introduce el nombre, los datos biográficos y los nueve atributos configurables del tenista. 3. Confirma el formulario de creación. 4. El sistema valida los datos introducidos. 5. El sistema guarda el jugador y lo añade al catálogo personal del usuario.
Flujo alternativo	Si alguno de los datos introducidos no es válido, el sistema muestra un mensaje de error y permite al usuario corregir el formulario.
Postcondición	El jugador queda almacenado en la base de datos y disponible para futuras simulaciones.

Tabla 3.3: Caso de uso: Crear un jugador personalizado

CU-04. Simular un partido

Actor principal	Usuario registrado
Descripción	El usuario configura un partido entre dos jugadores y el sistema genera una simulación completa del encuentro.
Precondición	El usuario tiene al menos dos jugadores disponibles, ya sean propios o predefinidos por el sistema.
Flujo principal	1. El usuario accede a la sección *Crear partido* y selecciona los dos jugadores. 2. Elige la superficie, el formato de sets y la configuración del *tie-break*. 3. Inicia la simulación. 4. El sistema ejecuta el partido en el servidor y genera la información punto a punto del encuentro. 5. La interfaz reproduce el marcador y la narración del partido. 6. Al finalizar, el sistema guarda el partido automáticamente y muestra la pantalla de resumen con las estadísticas del encuentro.
Flujo alternativo	Si no hay jugadores disponibles, el sistema informa al usuario y le indica que debe crear o seleccionar jugadores antes de iniciar la simulación.
Postcondición	El partido queda guardado en el historial del usuario.

Tabla 3.4: Caso de uso: Simular un partido

CU-05. Organizar un torneo

Actor principal	Usuario registrado
Descripción	El usuario crea un torneo eliminatorio, selecciona sus participantes y simula las rondas hasta obtener un campeón.
Precondición	El usuario tiene al menos cuatro jugadores disponibles.
Flujo principal	1. El usuario accede a la sección *Torneo*. 2. Elige el número de participantes, pudiendo ser 4, 8 o 16 jugadores. 3. Selecciona los jugadores participantes, la superficie y el formato de los partidos. 4. El sistema genera el cuadro inicial de emparejamientos. 5. El usuario simula la primera ronda y el sistema avanza automáticamente a los ganadores. 6. El proceso se repite hasta que se disputa la final y se determina el campeón.
Postcondición	El torneo queda guardado en la base de datos de forma persistente, junto con los resultados de sus rondas y el cuadro completo.

Tabla 3.5: Caso de uso: Organizar un torneo

CU-06. Consultar el historial de un jugador

Actor principal	Usuario registrado
Descripción	El usuario consulta los partidos almacenados asociados a un jugador y accede al detalle de un encuentro concreto.
Precondición	El usuario tiene partidos simulados guardados en el sistema.
Flujo principal	1. El usuario accede al perfil de uno de sus jugadores o a la sección de historial de partidos. 2. El sistema muestra la lista de partidos disputados, incluyendo información como fecha, rival, superficie y resultado. 3. El usuario selecciona un partido concreto. 4. El sistema muestra las estadísticas detalladas de ese encuentro.
Postcondición	El usuario consulta la información solicitada sin que se modifique ningún dato del sistema.

Tabla 3.6: Caso de uso: Consultar el historial de un jugador

CU-07. Ejecutar análisis Big Data

Actor principal	Usuario registrado
Descripción	El usuario ejecuta un conjunto de simulaciones repetidas entre dos jugadores para comparar su rendimiento estadístico.
Precondición	El usuario tiene dos jugadores disponibles y quiere comparar su rendimiento estadístico.
Flujo principal	1. El usuario accede al modo *Big Data* y configura el enfrentamiento, seleccionando los jugadores, la superficie y el formato. 2. Indica el número de simulaciones que desea ejecutar, dentro del rango permitido por la aplicación. 3. El sistema ejecuta las simulaciones solicitadas y muestra el avance del proceso. 4. Al finalizar, muestra estadísticas agregadas como el porcentaje de victorias, los aces de media, los errores y la distribución de resultados.
Postcondición	El usuario dispone de un análisis estadístico del enfrentamiento. Los partidos individuales generados en este modo no se guardan en el historial.

Tabla 3.7: Caso de uso: Ejecutar análisis Big Data

CU-08. Simular un partido en modo estratégico

Actor principal	Usuario registrado
Descripción	El usuario participa activamente en la simulación tomando decisiones tácticas durante el desarrollo del partido.
Precondición	El usuario ha seleccionado dos jugadores y ha activado el modo estratégico.
Flujo principal	1. El sistema muestra el estado actual del partido y solicita al usuario la estrategia que desea aplicar en el siguiente punto. 2. El usuario elige una estrategia: agresiva, neutral o defensiva. 3. El sistema calcula el punto aplicando los modificadores correspondientes. 4. El sistema muestra el resultado del punto y actualiza el marcador. 5. El proceso se repite hasta que el partido concluye.
Postcondición	El partido finaliza con la participación activa del usuario en las decisiones tácticas.

Tabla 3.8: Caso de uso: Simular un partido en modo estratégico

Capítulo 4

Diseño de la arquitectura del sistema

En este capítulo se describe la arquitectura del sistema, es decir, cómo se organiza la aplicación en capas, cómo se comunican sus componentes principales y qué decisiones de diseño se han tomado para estructurar el proyecto.

4.1. Visión general de la arquitectura

La aplicación se organiza en cuatro capas diferenciadas:

- **Frontend.** Es la capa de presentación. Se encarga de mostrar al usuario las distintas vistas de la aplicación, como el menú principal, la creación de jugadores, la simulación de partidos, la consulta de resultados y la gestión de torneos. Además, gestiona parte de la interacción dinámica en el navegador.
- **Backend de servicios.** Es la capa encargada de la lógica de negocio. Define las rutas de la aplicación, gestiona la autenticación, coordina las operaciones de creación, lectura, actualización y eliminación de datos, y delega las simulaciones al motor correspondiente.
- **Motor de simulación.** Es un componente ejecutado dentro del backend, pero con una lógica interna diferenciada. Su función es generar partidos completos punto a punto aplicando modelos probabilísticos. Debido a su importancia dentro del proyecto, se describe con mayor detalle en el Capítulo 5.
- **Base de datos.** Es la capa de persistencia del sistema. Almacena de forma relacional la información de usuarios, jugadores, partidos, estadísticas y torneos.

Estas capas se comunican de forma ordenada. La interfaz web envía peticiones al backend, el backend invoca al motor de simulación cuando es necesario y accede a la base de datos para leer o escribir información. Posteriormente, el backend devuelve la respuesta a la interfaz, ya sea como una página renderizada en el servidor o como datos en formato JSON.

4.2. Arquitectura cliente-servidor

La aplicación sigue un modelo cliente-servidor, en el que el navegador web actúa como cliente y el backend de la aplicación actúa como servidor. Aunque la interfaz se presenta como una aplicación web tradicional, internamente combina renderizado en el servidor con peticiones asíncronas en formato JSON.
El cliente se encarga de presentar la interfaz al usuario y gestionar las interacciones dinámicas que se producen en el navegador. Entre estas tareas se encuentran reproducir la simulación punto a punto, actualizar el marcador, mostrar la narración del partido, enviar formularios y presentar gráficos o estadísticas del encuentro. La información mostrada por el cliente procede del servidor, lo que permite mantener una separación clara entre la capa de presentación y la lógica principal de la aplicación.
El servidor se encarga de la lógica de negocio del sistema. Entre sus responsabilidades se incluyen la gestión de usuarios, la validación de los datos de entrada, la ejecución de las simulaciones, la aplicación de las reglas de puntuación y el acceso a la base de datos. Además, el servidor genera las páginas HTML mediante plantillas Jinja2 (12), de forma que el navegador recibe vistas ya renderizadas.
La comunicación entre cliente y servidor utiliza un enfoque híbrido. La navegación y la carga inicial de las páginas se realizan mediante peticiones `GET`, que devuelven HTML renderizado en el servidor. En cambio, las operaciones que modifican o consultan datos de la aplicación, como el inicio de sesión, la creación de jugadores o el lanzamiento de simulaciones, se realizan mediante peticiones asíncronas `POST` a endpoints de la API. Estas peticiones envían la información en formato JSON y son validadas en el backend mediante modelos de datos.
Este diseño permite mejorar la experiencia de usuario, ya que las acciones principales pueden ejecutarse sin recargar la página completa. Además, separa la lógica de presentación, basada en plantillas Jinja2, de la lógica de intercambio de datos, basada en una API JSON.
Las peticiones autenticadas incluyen un token JWT (11) en la cabecera `Authorization: Bearer <token>`. De esta forma, el servidor puede identificar al usuario en las operaciones privadas sin necesidad de solicitar sus credenciales en cada petición.

4.3. Componentes principales

En esta sección se describen los componentes principales que forman la arquitectura de la aplicación. Cada uno de ellos cumple una responsabilidad concreta dentro del sistema y se comunica con el resto mediante interfaces bien definidas.

4.3.1. Frontend SSR con Jinja2 y JavaScript

El frontend de la aplicación está basado en el renderizado de plantillas Jinja2 desde el servidor, complementado con JavaScript para gestionar la interacción dinámica en el navegador.
Esta elección se debe a que la aplicación está formada por varias vistas independientes, como el menú principal, la creación de jugadores, la configuración de partidos, la

simulación, el resumen de resultados, los torneos y el perfil de usuario. Estas vistas comparten una estructura visual similar y no requieren una arquitectura frontend especialmente compleja.
El renderizado en el servidor, conocido como SSR (*Server-Side Rendering*), permite simplificar la estructura del proyecto, ya que no requiere herramientas de compilación, empaquetadores ni gestores de paquetes específicos del lado del cliente. De esta forma, se reduce la complejidad frente a otros enfoques basados en frameworks frontend como React.
La interactividad del usuario se implementa mediante módulos de JavaScript. Estos se encargan de tareas como interceptar formularios, realizar peticiones asíncronas al backend, actualizar el marcador durante la simulación y mostrar mensajes o resultados sin necesidad de recargar la página completa.
En cuanto al aspecto visual, la aplicación utiliza Tailwind CSS 3 (13) para definir los estilos de la interfaz, junto con iconos vectoriales en formato SVG integrados directamente en las plantillas.

4.3.2. Backend con FastAPI

El backend se ha desarrollado con FastAPI (1), un framework web de Python orientado a la construcción de APIs y aplicaciones web. FastAPI se basa en el estándar ASGI, que permite ejecutar aplicaciones Python con soporte para comunicación asíncrona. En este proyecto, la aplicación se ejecuta sobre el servidor Uvicorn (2).
El backend es el encargado de coordinar la lógica principal del sistema. Sus responsabilidades principales son:

- Exponer los endpoints de la API REST que utiliza el frontend en las distintas funcionalidades de la aplicación.
- Coordinar la simulación de partidos, haciendo uso del motor de simulación, descrito en el Capítulo 5, para ejecutar los encuentros y procesar los resultados obtenidos.
- Implementar la lógica de negocio del sistema, incluyendo la validación de datos de entrada mediante esquemas Pydantic (6), la extracción de estadísticas a partir del desarrollo del partido y la normalización de los datos.
- Gestionar la autenticación de usuarios, verificando los tokens JWT en las rutas protegidas y asociando cada operación al usuario que la realiza.
- Comunicarse con la base de datos a través de SQLAlchemy, el ORM utilizado para mapear las tablas de PostgreSQL a clases Python y gestionar las sesiones de conexión.

4.3.3. Base de datos PostgreSQL

La base de datos del sistema se implementa mediante PostgreSQL (4), un sistema gestor de bases de datos relacional. En ella se almacena la información persistente

de la aplicación, incluyendo usuarios, jugadores creados, partidos simulados, estadísticas asociadas a cada partido y torneos.
Se ha optado por una base de datos relacional en lugar de una alternativa NoSQL porque las entidades del sistema presentan relaciones claras entre sí. Por ejemplo, un usuario puede tener varios jugadores, los jugadores participan en partidos, los partidos pueden pertenecer a torneos y las estadísticas se vinculan tanto a partidos como a jugadores. Este tipo de estructura hace conveniente el uso de integridad referencial mediante claves primarias y claves foráneas.
Además, PostgreSQL permite definir restricciones a nivel de base de datos, como restricciones `CHECK` para validar que determinados atributos se encuentren dentro de un rango permitido. También permite utilizar claves foráneas, eliminación en cascada y valores por defecto, aportando una capa adicional de validación independiente de la lógica de la aplicación.
El acceso a la base de datos se realiza mediante SQLAlchemy (3). Este ORM permite trabajar con las tablas de PostgreSQL mediante clases Python, lo que simplifica las operaciones de consulta, inserción, actualización y eliminación de datos desde el backend.

4.3.4. Sistema de autenticación con JWT

La autenticación del sistema se basa en JWT (*JSON Web Token*), un estándar que permite verificar la identidad del usuario en cada petición sin necesidad de mantener una sesión tradicional en el servidor.
El flujo de autenticación se divide en las siguientes fases:

1. **Registro.** El usuario crea una cuenta introduciendo un nombre de usuario, un correo electrónico y una contraseña. Antes de almacenar la contraseña, el sistema aplica una función de hash mediante *bcrypt*, a través de la librería *passlib* (10). De esta forma, no se guarda la contraseña en texto plano, sino una representación irreversible de la misma.

2. **Inicio de sesión.** El usuario introduce sus credenciales. El servidor busca la cuenta utilizando el nombre de usuario o el correo electrónico, compara la contraseña introducida con el hash almacenado y, si la verificación es correcta, genera un token JWT firmado con una clave secreta.

3. **Uso del token.** El cliente almacena el token en `localStorage` y lo adjunta en la cabecera `Authorization: Bearer <token>` de las peticiones posteriores que requieren autenticación.

4. **Verificación en rutas protegidas.** Los endpoints que requieren autenticación extraen el token de la cabecera, validan su firma y comprueban su caducidad. Si el token es válido, se obtiene el identificador del usuario y se utiliza para asociar la petición a dicho usuario. Si el token es inválido o ha expirado, la petición es rechazada.

4.4. Flujo de interacción entre componentes

Para explicar cómo interactúan los cuatro componentes principales del sistema, a continuación se describe el flujo de simulación de un partido. Se ha elegido este caso porque representa una de las funcionalidades centrales de la aplicación y requiere la participación del frontend, el backend, el motor de simulación y la base de datos.

- **Petición de página.** El usuario accede a la vista de creación de partido. El backend recibe una petición `GET`, renderiza la plantilla correspondiente mediante Jinja2 y devuelve la página HTML completa al navegador.
- **Configuración del partido.** El usuario selecciona dos jugadores de su lista o del catálogo del sistema, elige la superficie de juego y define el formato del partido. JavaScript realiza una primera validación de los campos del formulario en el cliente.
- **Envío de la simulación.** Al pulsar el botón de simulación, el módulo JavaScript encargado de la comunicación con la API envía una petición `POST` al endpoint correspondiente. En dicha petición se incluyen los datos de los jugadores y la configuración del partido en formato JSON, junto con el token JWT necesario para autenticar la operación.
- **Procesamiento en el backend.** FastAPI recibe la petición, valida los datos mediante esquemas Pydantic y comprueba el token JWT. A continuación, invoca al motor de simulación para ejecutar el partido completo punto a punto. Una vez finalizada la simulación, el sistema calcula las estadísticas del encuentro a partir del flujo generado.
- **Persistencia de resultados.** Si la simulación corresponde a un usuario registrado y debe almacenarse, el backend guarda en PostgreSQL la información del partido, el marcador, las estadísticas y las relaciones con los jugadores correspondientes.
- **Respuesta al cliente.** El servidor devuelve una respuesta en formato JSON con la información principal del partido, incluyendo el ganador, el marcador por sets, el flujo completo del encuentro y las estadísticas de ambos jugadores. El navegador utiliza estos datos para preparar la visualización de la simulación.
- **Visualización.** En la pantalla de simulación, los módulos JavaScript del frontend reproducen el partido punto a punto. El marcador se actualiza dinámicamente, se muestra la narración de las acciones y el usuario puede controlar la reproducción, la pausa y la velocidad. Al finalizar, el usuario puede acceder a la vista de resumen, donde se presentan las estadísticas del encuentro mediante gráficos.

4.5. Decisiones de diseño y justificación tecnológica

En esta sección se explican las principales decisiones tecnológicas tomadas durante el desarrollo del proyecto, así como las razones por las que se han elegido frente a

otras alternativas posibles.

- **FastAPI como framework backend.**

  Para el backend se valoraron otras alternativas como Django y Flask. Finalmente, se eligió FastAPI por su integración con Pydantic para la validación automática de datos, su buen rendimiento y su soporte para aplicaciones asíncronas ejecutadas sobre servidores ASGI como Uvicorn. Además, su forma de definir rutas y modelos resulta clara para una aplicación basada en endpoints JSON.

- **Jinja2 frente a otros frameworks frontend o HTML puro.**

  El uso de Jinja2 permite aplicar renderizado en el lado del servidor, lo que simplifica la estructura de la aplicación al no requerir herramientas de compilación, empaquetadores ni gestores de paquetes del lado del cliente. Cada URL puede ser gestionada por el servidor, que genera la página correspondiente y la devuelve ya renderizada al navegador.

  También se descartó trabajar únicamente con HTML estático, ya que el motor de plantillas permite reutilizar estructuras comunes mediante herencia, como barras laterales, cabeceras o pies de página. Además, facilita la inserción dinámica de datos generados por el servidor, como el nombre del usuario autenticado, la lista de jugadores o la información de un partido.

- **PostgreSQL frente a bases de datos NoSQL.**

  Como se detalla en la Sección 4.3.3, los datos del proyecto son principalmente relacionales. Existen relaciones claras entre usuarios, jugadores, partidos, estadísticas y torneos, por lo que era preferible utilizar un sistema gestor de bases de datos que garantizase integridad referencial. PostgreSQL se eligió además por su robustez y por su buena integración con SQLAlchemy.

- **SQLAlchemy como ORM.**

  SQLAlchemy permite definir los modelos de datos como clases Python mapeadas a las tablas de la base de datos, reduciendo la necesidad de escribir consultas SQL manuales para las operaciones habituales. Su integración con FastAPI mediante dependencias facilita la gestión de sesiones de conexión y mantiene separado el acceso a datos de la lógica principal de la aplicación.

  Además, el uso de un ORM resultaba familiar por su similitud conceptual con herramientas vistas durante la carrera, como JPA (*Java Persistence API*), lo que facilitó la comprensión del modelo de persistencia y la organización de las entidades.

- **JWT para autenticación.**

  Se eligió JWT (*JSON Web Token*) porque permite identificar al usuario en cada petición sin mantener una sesión tradicional en el servidor. El cliente envía el token en la cabecera de las peticiones autenticadas y el backend verifica su firma y validez antes de permitir el acceso a las rutas protegidas.

Esta decisión se complementa con el uso de *bcrypt* para el hash de contraseñas, evitando almacenar las contraseñas en texto plano en la base de datos.

- **Chart.js para visualización de estadísticas.**

  Para la representación gráfica de las estadísticas se utiliza Chart.js (14). Esta librería permite generar distintos tipos de gráficos, como gráficos de líneas, barras, radar, donut e histogramas, utilizando Canvas HTML5. Se eligió por su sencillez de uso, su rendimiento adecuado para la aplicación y porque no requiere herramientas de build adicionales.

- **Tailwind CSS frente a Bootstrap o CSS manual.**

  Tailwind CSS se eligió por su enfoque basado en clases de utilidad aplicadas directamente en el HTML. Esto permite construir interfaces personalizadas de forma rápida sin depender tanto de componentes predefinidos como ocurre en Bootstrap. Frente a escribir todo el CSS de forma manual, Tailwind reduce la cantidad de estilos personalizados necesarios y facilita mantener una apariencia visual coherente en toda la aplicación.

  Además, los iconos se integran como elementos SVG dentro de las plantillas, evitando depender de una librería externa de iconografía.

Capítulo 5

Diseño del motor de simulación

En este capítulo se describe el diseño del motor de simulación, que constituye el núcleo funcional de ADAF Tennis Simulator. Su propósito es reproducir partidos de tenis completos de forma coherente con las principales reglas de puntuación del deporte, modelando el desarrollo de cada punto a partir de los atributos de los jugadores y de variables dinámicas que evolucionan durante el encuentro.
A lo largo del capítulo se presenta la representación computacional de los jugadores, el flujo lógico de la simulación punto a punto y la incorporación de factores contextuales como la estamina, el *momentum* y el *clutch*. También se describen las reglas de puntuación implementadas para construir juegos, sets y partidos completos. Finalmente, se exponen los mecanismos de reproducibilidad del sistema y las principales limitaciones del modelo actual.

5.1. Objetivos del motor de simulación

El motor de simulación de ADAF Tennis Simulator constituye el núcleo computacional de la aplicación. Mediante un modelo probabilístico orientado a objetos, reproduce el desarrollo punto a punto de un partido de tenis entre dos jugadores parametrizados. Para funcionar correctamente, el motor debe satisfacer un conjunto de objetivos funcionales y de diseño.
En primer lugar, el motor debe ser capaz de generar un partido completo conforme a las principales reglas de puntuación del tenis, incluyendo el sistema de puntuación con ventaja, la muerte súbita o desempate a siete puntos, conocido como *tie-break*, y las modalidades de partido a un set, al mejor de tres sets o al mejor de cinco sets. En segundo lugar, cada punto simulado debe ser el resultado de una cadena de sucesos determinada mediante funciones probabilísticas, en las que los atributos de los jugadores influyen de forma significativa. En tercer lugar, el modelo debe incorporar variables dinámicas que evolucionen durante el partido y afecten al desarrollo de la simulación, aportando contexto al encuentro.
Un aspecto central en el diseño del motor es su naturaleza no determinista. A diferencia de un modelo basado únicamente en reglas fijas, en el que el resultado sería completamente predecible a partir de los atributos de entrada, este motor utiliza aleatoriedad controlada para que cada partido pueda desarrollarse de forma distin-

ta.
La variabilidad se introduce mediante el uso de distribuciones de probabilidad, principalmente normales, en la generación de cada golpe. El sistema utiliza los atributos de los jugadores para modificar la probabilidad de éxito de las acciones, favoreciendo estadísticamente al jugador con mayor calidad técnica. Sin embargo, la existencia de una varianza residual permite la aparición de errores no forzados en jugadores de alto nivel o de golpes excepcionales en jugadores de menor nivel. De este modo, aunque el modelo tiende a favorecer al jugador superior, mantiene abierta la posibilidad de resultados inesperados, reflejando parte de la incertidumbre propia del deporte real.
El motor está implementado íntegramente en Python (7) y se organiza en los módulos `models.py`, `utils.py`, `shots.py`, `point.py`, `scoring.py`, `scorekeeper.py`, `strategy.py` y `api.py`. Cada uno de estos módulos agrupa una parte concreta de la lógica del simulador: los modelos de datos, funciones auxiliares, generación de golpes, simulación de puntos, reglas de puntuación, gestión del marcador en modos interactivos, estrategias y punto de entrada público del motor.
La función pública de entrada para la simulación automática es `run_match()`, ubicada en `api.py` dentro del paquete del motor de simulación. Esta función recibe los atributos de ambos jugadores y un diccionario de configuración, inicializa la semilla aleatoria cuando corresponde y delega la ejecución del encuentro en la clase `TennisMatch`.

5.2. Modelado del jugador

En la aplicación existen dos formas de representar a un jugador. Por un lado, la entidad persistente almacenada en la base de datos contiene información biográfica y de perfil, como la edad, la altura, la nacionalidad o el brazo dominante. Por otro lado, para ejecutar una simulación, el motor instancia la clase `Player`, definida como `dataclass` en el módulo `models.py`.
La clase `Player` agrupa los atributos técnicos necesarios para simular el comportamiento de un tenista durante un partido. A partir de estos atributos estáticos, el sistema utiliza también variables dinámicas que evolucionan durante el encuentro y modifican el rendimiento del jugador en función del contexto de juego.

5.2.1. Atributos estáticos

Los atributos estáticos describen el perfil técnico inicial del jugador y permanecen constantes a lo largo de la simulación. La Tabla 5.1 recoge el nombre de cada atributo, la abreviatura utilizada en el modelo y su interpretación funcional.
Los símbolos utilizados en la tabla corresponden a abreviaturas habituales dentro del modelo. En particular, *FH* procede de *Forehand*, es decir, golpe de derecha, y *BH* procede de *Backhand*, es decir, golpe de revés.
Estos atributos se almacenan en una escala numérica comprendida entre 1 y 100, y se normalizan antes de ser utilizados en los cálculos internos del motor. Por ejemplo, el atributo de primer saque se transforma de la siguiente forma:

Atributo	Símbolo	Descripción
`Primer_Saque`	S_1	Calidad técnica del primer saque.
`Segundo_Saque`	S_2	Calidad técnica del segundo saque.
`Fisico`	F	Condición física general del jugador, utilizada para modelar potencia y resistencia.
`Consistencia`	C	Regularidad del jugador, reduciendo la probabilidad de errores y la variabilidad de los golpes.
`Clutch`	K	Capacidad de rendir bajo presión en puntos importantes.
`Derecha`	FH	Calidad del golpe de derecha.
`Reves`	BH	Calidad del golpe de revés.
`Resto`	RET	Calidad de la devolución del saque.
`Movilidad`	MOV	Capacidad de desplazamiento y llegada a la bola.

Tabla 5.1: Atributos técnicos del jugador y significado dentro del modelo.

$$S_1 = \frac{\texttt{Primer_Saque}}{100} \tag{5.1}$$

De esta manera, los atributos pasan a trabajar en una escala comprendida aproximadamente entre 0,01 y 1, lo que facilita su uso dentro de las fórmulas probabilísticas del motor.
La selección del lado de golpe durante el peloteo y el resto se realiza mediante el método `pick_side()`. Este método elige entre derecha y revés de forma probabilística, teniendo en cuenta la calidad relativa de ambos golpes:

$$\Pr(\text{derecha}) = \frac{FH}{FH + BH + \epsilon}, \qquad \epsilon = 10^{-9} \tag{5.2}$$

El término ϵ se introduce para evitar una posible división por cero. Una vez elegido el lado del golpe, el valor correspondiente, ya sea FH o BH, se utiliza en las fórmulas de potencia, precisión y probabilidad de error.

5.2.2. Variables dinámicas: estamina y momentum

A diferencia de los atributos estáticos, la estamina y el *momentum* no son valores fijos del perfil del jugador. Ambas variables se actualizan durante el desarrollo del partido y permiten que el rendimiento no dependa únicamente de los atributos iniciales. Al comienzo de cada encuentro, el método `reset_dynamic_state()` reinicia estas variables a sus valores iniciales.

Estamina. La estamina, representada como E, modela el desgaste físico acumulado del jugador durante el partido. Su valor se mantiene en una escala $[0, 100]$, donde 100 representa el estado físico inicial y 0 un nivel de fatiga máximo.

Tras cada punto, se aplica una reducción de estamina proporcional a la duración del intercambio y condicionada por la condición física del jugador:

$$\Delta E_{\text{fatiga}} = (0{,}5 + n_{\text{rally}} \cdot 0{,}2) \cdot \left(1{,}5 - \frac{\texttt{Fisico}}{100}\right) \tag{5.3}$$

donde n_{rally} representa el número de golpes de peloteo registrados en el punto. La actualización de la estamina se realiza evitando que el valor pueda ser negativo:

$$E \leftarrow \text{máx}\,(0,\ E - \Delta E_{\text{fatiga}}) \tag{5.4}$$

Al final de cada juego, ambos jugadores recuperan una pequeña cantidad de estamina en función de su condición física:

$$E \leftarrow \text{mín}\left(100,\ E + 0{,}3 + 0{,}8 \cdot \frac{\texttt{Fisico}}{100}\right) \tag{5.5}$$

Al final de cada set, la recuperación es mayor:

$$E \leftarrow \text{mín}\left(100,\ E + 2 + 5 \cdot \frac{\texttt{Fisico}}{100}\right) \tag{5.6}$$

Para ser utilizada dentro de las fórmulas del motor, la estamina se normaliza como:

$$\hat{E} = \frac{E}{100} \tag{5.7}$$

De esta forma, un jugador con baja estamina tiende a producir golpes de menor calidad media y con mayor irregularidad.

Momentum. El *momentum*, representado como M, modela la influencia de las rachas positivas o negativas durante el partido. Su valor se mantiene dentro del intervalo $[-50, 50]$. Un valor positivo indica una dinámica favorable para el jugador, mientras que un valor negativo representa una pérdida de confianza o de control del partido.
Para actualizar esta variable, el motor utiliza una variable auxiliar denominada `streak`, que registra la racha de puntos consecutivos ganados o perdidos por cada jugador. Tras cada punto, la racha del ganador aumenta y la del perdedor disminuye:

$$\texttt{streak}_{\text{gan}} \leftarrow \text{máx}(1,\ \texttt{streak}_{\text{gan}} + 1) \tag{5.8}$$

$$\texttt{streak}_{\text{per}} \leftarrow \text{mín}(-1,\ \texttt{streak}_{\text{per}} - 1) \tag{5.9}$$

A partir de estas rachas se actualiza el *momentum* de ambos jugadores:

$$M_{\text{gan}} \leftarrow \text{mín}\,(50,\ M_{\text{gan}} + 2 \cdot |\texttt{streak}_{\text{gan}}|) \cdot 0{,}97 \tag{5.10}$$

$$M_{\text{per}} \leftarrow \text{máx}\,(-50,\ M_{\text{per}} - 2 \cdot |\texttt{streak}_{\text{per}}|) \cdot 0{,}97 \tag{5.11}$$

El factor de decaimiento 0,97 evita que el *momentum* crezca o disminuya de forma indefinida, simulando una tendencia natural a estabilizar el estado mental del partido. Además, al ganar o perder un juego completo se aplica una variación adicional de ± 8 puntos, y al ganar o perder un set una variación de ± 15 puntos.

Finalmente, el *momentum* se normaliza antes de utilizarse en los cálculos de golpes y probabilidades:

$$\hat{M} = \frac{M}{100}, \qquad \hat{M} \in [-0{,}5, 0{,}5] \tag{5.12}$$

Este valor actúa como modificador contextual, favoreciendo ligeramente al jugador que atraviesa una buena racha y penalizando al que encadena puntos o juegos perdidos.

5.3. Modelado del punto

La clase `PointSimulator`, definida en `point.py`, coordina la simulación de un punto completo en tres fases secuenciales: saque, resto y peloteo. Cada fase produce o consume un objeto `Ball`, que transporta la potencia (*pot*), la precisión (*prec*) y el lado del golpe entre las distintas etapas del punto.
En las fórmulas de este apartado, los atributos técnicos de los jugadores se consideran normalizados, es decir, transformados a partir de su escala original $[1, 100]$ dividiendo su valor entre 100. La calidad de un golpe se define como una combinación lineal de potencia y precisión:

$$shotQ = \text{clip}(0{,}65 \cdot pot + 0{,}35 \cdot prec,\ 0,\ 1) \tag{5.13}$$

Esta métrica resume en un único valor la dificultad que el golpe recibido impone al jugador contrario.
La lógica general de la simulación puede representarse mediante un diagrama de estados, en el que se recogen las transiciones principales desde el inicio del punto hasta su finalización, incluyendo las posibles bifurcaciones asociadas al saque, el resto y el desarrollo del rally.

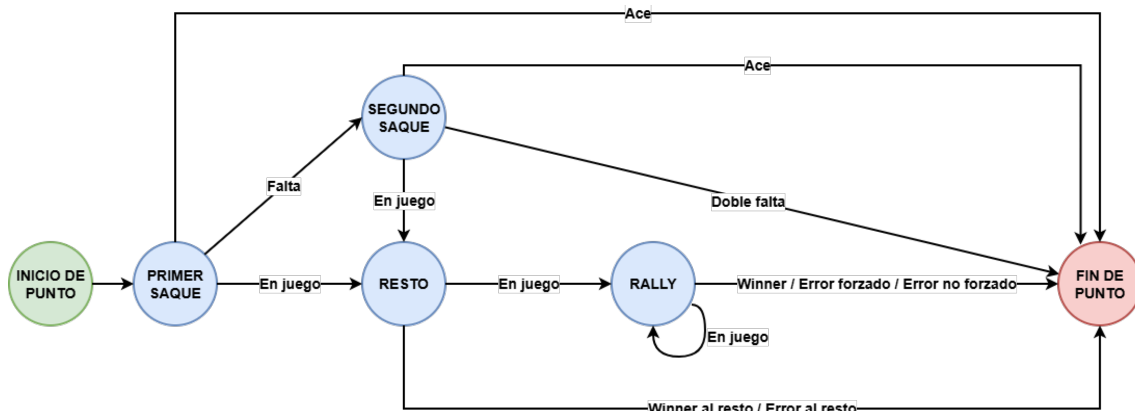

Figura 5.1: Diagrama de estados del desarrollo de un punto en el motor de simulación.

Como puede observarse en la Figura 5.1, todo punto comienza con un primer saque. Si este no entra, la simulación pasa al estado de segundo saque; si entra en juego, el receptor ejecuta la fase de resto. A partir de ese momento, el punto puede finalizar directamente por ace, doble falta, winner o error, o bien continuar mediante un rally

en el que se encadenan golpes sucesivos hasta producirse un desenlace definitivo. En los apartados siguientes se describen con mayor detalle los modelos probabilísticos empleados en cada una de estas fases.

5.3.1. Primer saque

El primer servicio se implementa en la clase `Serve`, definida en `shots.py`. Su comportamiento sigue una lógica de mayor riesgo, ya que el jugador puede buscar un saque más agresivo para forzar el error del rival o conseguir un punto directo mediante un *ace*. Esta agresividad se traduce en una mayor penalización asociada a la potencia del golpe.

Modelado de la variabilidad humana. Para evitar un comportamiento determinista, el sistema no asigna valores fijos de potencia y precisión. En su lugar, cada ejecución del saque se considera un evento único cuyos parámetros se obtienen a partir de distribuciones normales $\mathcal{N}(\mu, \sigma)$.
Las medias (μ) representan el rendimiento esperado del jugador en esa acción bajo las condiciones actuales del punto. La potencia depende principalmente de la técnica específica de primer saque (S_1), de la condición física (F) y de la estamina normalizada (E). La precisión depende de la técnica de saque, la consistencia (C) y la estamina:

$$\mu_{pot}^{(1)} = 0{,}5 \cdot S_1 + 0{,}35 \cdot F + 0{,}15 \cdot E \tag{5.14}$$
$$\mu_{prec}^{(1)} = 0{,}42 \cdot S_1 + 0{,}38 \cdot C + 0{,}20 \cdot E \tag{5.15}$$

Las desviaciones estándar (σ) modelan la variabilidad mecánica del golpe. Un jugador con menor consistencia o con mayor fatiga presenta una desviación más elevada, lo que aumenta la dispersión de los resultados y hace que el saque sea más irregular:

$$\sigma_{pot}^{(1)} = 0{,}04 + 0{,}08 \cdot (1 - C) + 0{,}04 \cdot (1 - E) \tag{5.16}$$
$$\sigma_{prec}^{(1)} = 0{,}05 + 0{,}10 \cdot (1 - C) + 0{,}05 \cdot (1 - E) \tag{5.17}$$

Una vez definidos estos parámetros, los valores de potencia y precisión del saque se obtienen mediante muestreo y se limitan al intervalo $[0{,}01, 1]$:

$$\begin{aligned} pot_1 &\sim \text{clip}\Big(\mathcal{N}\Big(\mu_{pot}^{(1)}, \sigma_{pot}^{(1)}\Big), 0{,}01,\ 1\Big), \\ prec_1 &\sim \text{clip}\big(\mathcal{N}\big(\mu_{prec}^{(1)}, \sigma_{prec}^{(1)}\big), 0{,}01,\ 1\big) \end{aligned} \tag{5.18}$$

Ajuste por momentum y compromiso de riesgo. Tras el muestreo, el modelo puede aplicar un factor de *momentum* para representar el estado de confianza del jugador. Una racha positiva incrementa ligeramente la calidad mecánica del golpe, mientras que una racha negativa puede reducirla. Este ajuste permite que el rendimiento no dependa únicamente de los atributos estáticos del jugador, sino también del contexto reciente del partido.

El éxito del saque no se calcula como una probabilidad fija, sino a partir de una penalización por riesgo. Cuanto mayor es la potencia del servicio, menor es el margen de seguridad y, por tanto, más difícil resulta que el saque entre. Esta relación se expresa mediante el siguiente *score* técnico:

$$score^{(1)} = 0{,}6 \cdot prec_1 - 0{,}3 \cdot pot_1 + 0{,}2 \cdot S_1 + 0{,}2 \cdot C \tag{5.19}$$

El valor obtenido se normaliza y posteriormente se transforma en una probabilidad de acierto. En el primer saque, esta probabilidad se mantiene entre un mínimo del 50 % y un máximo del 75 %, evitando porcentajes extremos:

$$\tilde{s}^{(1)} = \mathrm{clip}\left(\frac{score^{(1)} + 0{,}3}{1{,}3},\ 0,\ 1\right) \tag{5.20}$$

$$p_{\text{in}}^{(1)} = 0{,}5 + 0{,}25 \cdot \tilde{s}^{(1)} \tag{5.21}$$

Finalmente, la validez del saque se decide mediante un ensayo de Bernoulli con probabilidad $p_{\text{in}}^{(1)}$. En caso de fallo, el motor activa automáticamente la fase de segundo saque.

5.3.2. Segundo saque

El segundo servicio, también gestionado por la clase `Serve`, responde a una lógica de mayor control. Tras fallar el primer intento, el jugador prioriza la seguridad mecánica frente a la agresividad, ya que un segundo fallo implica la pérdida automática del punto por doble falta.

Ajuste de potencia y control de dispersión. Al igual que en el primer saque, los parámetros base se obtienen mediante distribuciones normales. Sin embargo, en este caso se utiliza el atributo técnico de segundo saque (S_2) y se reducen las desviaciones estándar. Esta reducción de la varianza representa un gesto técnico más conservador y menos propenso a errores no forzados:

$$\mu_{pot}^{(2)} = 0{,}5 \cdot S_2 + 0{,}35 \cdot F + 0{,}15 \cdot E \tag{5.22}$$

$$\mu_{prec}^{(2)} = 0{,}42 \cdot S_2 + 0{,}38 \cdot C + 0{,}20 \cdot E \tag{5.23}$$

$$\sigma_{pot}^{(2)} = 0{,}02 + 0{,}04 \cdot (1 - C) + 0{,}02 \cdot (1 - E) \tag{5.24}$$

$$\sigma_{prec}^{(2)} = 0{,}025 + 0{,}10 \cdot (1 - C) + 0{,}05 \cdot (1 - E) \tag{5.25}$$

A partir de estos parámetros se muestrean la potencia y la precisión base del segundo saque. La precisión resultante puede verse modificada posteriormente por factores contextuales, como el *momentum* o el atributo *clutch* en situaciones de presión.
Para mantener una diferencia realista entre ambos servicios, la potencia final del segundo saque (pot_2) se calcula de forma relativa a la potencia generada para el primer saque. El modelo impone que el segundo servicio sea menos agresivo, situándose dentro de un rango acotado respecto al primer intento:

$$pot_2 = \text{clip}\big(pot_{\text{raw}}^{(2)} \cdot r + \varepsilon,\ \text{máx}(0{,}7 \cdot pot_1,\ 0{,}50),\ \text{mín}(0{,}8 \cdot pot_1,\ 0{,}66)\big) \tag{5.26}$$

En esta expresión, r es un factor aleatorio uniforme en el intervalo $[0{,}70, 0{,}80]$, utilizado para reducir la potencia del segundo saque respecto al primero. Por su parte, ε es una pequeña perturbación aleatoria uniforme en el intervalo $[-0{,}02, 0{,}02]$, introducida para evitar que todos los segundos saques presenten un comportamiento idéntico.

El factor psicológico y el atributo *clutch*. A diferencia del primer saque, el segundo servicio es más sensible al estado mental del jugador en momentos críticos, como una bola de break o un punto de set. En estas situaciones, el atributo *clutch* (K) modula directamente la precisión ya muestreada del segundo saque:

$$prec_2 \leftarrow prec_2 \cdot (1 + (K - 0{,}5) \cdot 0{,}20) \tag{5.27}$$

Esta fórmula premia a los jugadores con mayor temple ($K > 0{,}5$), permitiéndoles mantener mejor la precisión bajo presión, mientras que penaliza a aquellos con menor resistencia psicológica.

Resolución de la probabilidad de éxito. La probabilidad de que el segundo saque sea válido ($p_{\text{in}}^{(2)}$) es superior a la del primero, ya que el jugador adopta un golpe más seguro. El cálculo parte de una base dependiente de la calidad del segundo saque y de la consistencia:

$$score^{(2)} = 0{,}6 \cdot S_2 + 0{,}4 \cdot C \tag{5.28}$$

$$p_{\text{base}}^{(2)} = 0{,}86 + 0{,}09 \cdot score^{(2)} \tag{5.29}$$

$$p_{\text{in}}^{(2)} = \text{clip}\Big(p_{\text{base}}^{(2)} \cdot (1 + \delta_{\text{clutch}} + \delta_M),\ 0,\ 1\Big) \tag{5.30}$$

donde δ_{clutch} representa el ajuste psicológico aplicado en puntos de presión y δ_M el ajuste asociado al *momentum* del jugador.
Si el ensayo de Bernoulli resultante es negativo, el motor registra una doble falta y concede el punto al restador sin iniciar la fase de peloteo.

5.3.3. Resto

La devolución del saque, implementada en la clase `ReturnShot`, constituye la primera fase del punto en la que interactúan ambos jugadores. En esta situación, el objetivo del restador es neutralizar la ventaja inicial del sacador y conseguir que la bola vuelva a estar en juego. En algunos casos, si la devolución es especialmente efectiva, el punto puede finalizar directamente con un golpe ganador de resto (*return winner*).

Determinación del alcance. Antes de evaluar la calidad de la devolución, el sistema determina si el restador es capaz de alcanzar la bola. Esta probabilidad de

alcance, p_{reach}, surge de comparar la dificultad del saque recibido con la capacidad defensiva del jugador. La dificultad del saque se calcula a partir de su potencia y precisión, mientras que la capacidad del restador depende de su atributo de resto, movilidad y estamina:

$$dif = \text{clip}(0{,}7 \cdot pot + 0{,}3 \cdot prec,\ 0,\ 1) \tag{5.31}$$
$$cap = 0{,}5 \cdot RET + 0{,}3 \cdot MOV + 0{,}2 \cdot E \tag{5.32}$$

La probabilidad base parte de un 85 % y se ajusta según la diferencia entre ambos factores. Además, el modelo añade un margen adicional, δ_{seg}, cuando el saque recibido es un segundo servicio, ya que este suele ser menos agresivo y más fácil de leer para el restador:

$$p_{\text{reach}} = \text{clip}(0{,}85 + (cap - dif) \cdot 0{,}25 + \delta_{\text{seg}},\ 0{,}70,\ 0{,}97) \tag{5.33}$$

Si el ensayo de Bernoulli asociado a esta probabilidad falla, el restador no alcanza la bola y el punto finaliza como *ace* a favor del sacador.

Calidad de la devolución y neutralización. Si el restador alcanza la bola, el sistema genera la devolución. Para ello, calcula primero la potencia y precisión base del golpe, ponderando el atributo de resto, la condición física, la estamina, la movilidad, la consistencia y el lado del golpe seleccionado, ya sea derecha o revés:

$$\mu^{\text{ret}}_{pot} = 0{,}35 \cdot RET + 0{,}25 \cdot F + 0{,}20 \cdot E + 0{,}20 \cdot side \tag{5.34}$$
$$\mu^{\text{ret}}_{prec} = 0{,}40 \cdot RET + 0{,}25 \cdot C + 0{,}20 \cdot side + 0{,}15 \cdot MOV \tag{5.35}$$

Estas medias se ajustan en función de la calidad del saque recibido. El modelo no aplica una penalización directa, sino una corrección centrada en valores de referencia. De esta forma, un saque de calidad media apenas modifica el rendimiento del restador, un saque difícil reduce la calidad de la devolución y un saque más débil puede facilitar una respuesta más cómoda:

$$\mu^{\text{ret}*}_{pot} = \text{clip}\big(\mu^{\text{ret}}_{pot} + 0{,}18 \cdot (0{,}50 - shotQ_{\text{in}}),\ 0{,}15,\ 0{,}75\big) \tag{5.36}$$
$$\mu^{\text{ret}*}_{prec} = \text{clip}\big(\mu^{\text{ret}}_{prec} + 0{,}22 \cdot (0{,}60 - shotQ_{\text{in}}),\ 0{,}20,\ 0{,}92\big) \tag{5.37}$$

A partir de estas medias ajustadas se realiza el muestreo estocástico para obtener la potencia y precisión finales de la devolución.

Probabilidad de éxito en el impacto. Una vez generada la devolución, el motor calcula la probabilidad de que esta caiga dentro del campo. Para ello se parte de una probabilidad base, p_{in}, asociada a la calidad técnica del golpe, a la que se añade un componente de ruido proporcional a la inconsistencia del jugador:

$$\varepsilon \sim \mathcal{N}(0,\ 0{,}05 \cdot (1 - C)) \tag{5.38}$$

La probabilidad final integra el efecto del *momentum* y, cuando el punto es especialmente importante, el atributo *clutch*:

$$p_{\text{final}} = \text{clip}\Big(\text{clip}(p_{\text{in}} + \varepsilon,\ 0{,}02,\ 0{,}99) \cdot (1 + 0{,}10 \cdot \hat{M}) \cdot b_K,\ 0{,}02,\ 0{,}99 \Big) \tag{5.39}$$

donde b_K representa el modificador asociado al atributo K en puntos de presión. Si el ensayo de Bernoulli resultante falla, el punto se contabiliza como error de resto y se concede al sacador sin que se inicie el peloteo.

5.3.4. Rally golpe a golpe

Si la devolución es válida, el punto entra en la fase de peloteo. Esta etapa se gestiona de forma iterativa mediante la clase `RallyShot`, donde cada iteración representa un golpe ejecutado por uno de los jugadores. El bucle continúa hasta que se produce una condición de finalización: un jugador no alcanza la bola o el golpe ejecutado no entra dentro de los límites de la pista.

Alcance en el peloteo. A diferencia del resto, la probabilidad de alcanzar la bola durante el peloteo depende de forma más directa de la movilidad, la estamina y la calidad del golpe recibido. El sistema utiliza la siguiente expresión:

$$p^{\text{rally}}_{\text{reach}} = \text{clip}\Big(0{,}72 + 0{,}30 \cdot (MOV - 0{,}70) + 0{,}20 \cdot (E - 0{,}70) - 0{,}55 \cdot (shotQ_{\text{in}} - 0{,}60),\ 0{,}02,\ 0{,}98 \Big) \tag{5.40}$$

Esta fórmula penaliza la probabilidad de alcance cuando la calidad del golpe entrante supera el umbral de referencia. Así, un golpe especialmente potente o preciso puede convertirse en un golpe ganador si el rival no consigue llegar a la bola.

Dinámica de golpeo y calidad del tiro. Si el jugador alcanza la posición de golpeo, el sistema calcula la potencia y precisión del nuevo golpe. En esta fase, la estamina y la consistencia tienen un papel relevante, ya que determinan la capacidad del jugador para mantener un intercambio prolongado sin perder calidad:

$$\mu^{\text{rly}}_{pot} = 0{,}45 \cdot F + 0{,}25 \cdot E + 0{,}20 \cdot side + 0{,}10 \cdot C \tag{5.41}$$

$$\mu^{\text{rly}}_{prec} = 0{,}40 \cdot C + 0{,}30 \cdot side + 0{,}20 \cdot MOV + 0{,}10 \cdot E \tag{5.42}$$

Al igual que en la devolución, estas medias se ajustan en función de la calidad del golpe anterior. El modelo utiliza una corrección centrada en valores de referencia, de forma que una bola entrante de calidad media apenas altera la ejecución, una bola difícil reduce la calidad del golpe y una bola débil puede facilitar una respuesta más agresiva:

$$\mu_{pot}^{\text{rly}*} = \text{clip}\Big(\mu_{pot}^{\text{rly}} + 0{,}18 \cdot (0{,}50 - shotQ_{\text{in}}),\ 0{,}15,\ 0{,}75\Big) \tag{5.43}$$

$$\mu_{prec}^{\text{rly}*} = \text{clip}\big(\mu_{prec}^{\text{rly}} + 0{,}22 \cdot (0{,}60 - shotQ_{\text{in}}),\ 0{,}20,\ 0{,}92\big) \tag{5.44}$$

Además, el factor de *momentum* se aplica con un peso mayor durante el peloteo:

$$f_M = 1 + 0{,}15 \cdot \hat{M} \tag{5.45}$$

Este factor permite que una racha positiva mejore ligeramente la ejecución del jugador durante intercambios abiertos, mientras que una dinámica negativa puede reducir su rendimiento.

Resolución del intercambio. La validez del golpe de peloteo se decide combinando la calidad técnica del lado utilizado, derecha o revés, con la consistencia del jugador. Además, se introduce un ajuste por la calidad del golpe recibido:

$$Q = 0{,}6 \cdot side + 0{,}4 \cdot C \tag{5.46}$$

$$Q_{\text{adj}} = Q - 0{,}5 \cdot (shotQ_{\text{in}} - 0{,}5) \tag{5.47}$$

$$p_{\text{in}} = \text{clip}(0{,}65 + 0{,}3 \cdot (Q_{\text{adj}} - 0{,}5),\ 0{,}05,\ 0{,}99) \tag{5.48}$$

A esta probabilidad se añade un ruido gaussiano proporcional a la inconsistencia del jugador:

$$\varepsilon \sim \mathcal{N}(0,\ 0{,}05 \cdot (1 - C)) \tag{5.49}$$

Si el ensayo de Bernoulli determina que la bola no entra, el punto finaliza y se concede al jugador que ejecutó el último golpe válido. En caso contrario, la nueva bola generada pasa al rival y el proceso se repite.

5.3.5. Finalización del punto

Además de determinar qué jugador gana el punto, el motor registra el motivo por el que se ha producido el desenlace. Esta información se almacena en el objeto `PointResult` y posteriormente se utiliza para construir las estadísticas del partido. El motor distingue entre los siguientes tipos principales de finalización:

Doble falta (`doble_falta`). El sacador falla sus dos intentos de saque, por lo que el punto se concede directamente al restador.

Ace (`ace`). El saque entra en juego, pero el restador no consigue alcanzar la bola. El punto se concede al sacador.

Error de resto (`error_resto`). El restador alcanza la bola, pero la devolución no entra en el campo contrario. El punto se concede al sacador.

No llega durante el peloteo (`no_llega`). Durante el intercambio, un jugador no consigue alcanzar el golpe del rival. Esta situación se interpreta como un golpe ganador o como un error forzado, según el contexto del punto.

Error durante el peloteo (`error_golpe`). El jugador alcanza la bola y ejecuta el golpe, pero este no entra dentro de los límites de la pista. El punto se concede al rival.

Durante la simulación, estos eventos se registran en la estructura `MatchStats`. Además del ganador de cada punto, el sistema almacena información adicional como la duración del peloteo, el tipo de saque utilizado, la existencia de aces, dobles faltas, errores y otros indicadores relevantes.
Gracias a este registro, al finalizar el partido se pueden generar estadísticas agregadas, como el porcentaje de primeros saques, el número de aces, las dobles faltas, los errores cometidos o el rendimiento de cada jugador en distintas fases del encuentro. De esta forma, el resultado probabilístico del motor se transforma en información estadística interpretable para el usuario.

5.4. Influencia del clutch en puntos clave

En un partido de tenis, no todos los puntos tienen la misma importancia dentro del marcador. Un punto disputado en una situación neutra no tiene el mismo impacto que una bola de ruptura (*break point*) o un punto decisivo dentro de un *tie-break*. Para modelar esta presión competitiva, el motor de simulación incorpora el atributo *Clutch*, que representa la capacidad de un jugador para mantener su rendimiento en momentos importantes del partido.

Activación de la presión. El sistema identifica automáticamente estos escenarios mediante el método `is_clutch_point()`. En el modelo se consideran puntos de presión las siguientes situaciones:

- **Marcadores de igualdad o ventaja.** Cuando el juego llega a *deuce* o ventaja, ya que el siguiente punto puede acercar mucho a uno de los jugadores a cerrar el juego.
- **Oportunidades de ruptura.** Cuando el restador tiene la posibilidad de ganar el juego al saque del rival, es decir, en una situación de *break point*.
- ***Tie-break.*** Durante el *tie-break*, todos los puntos se consideran de presión, ya que cada intercambio tiene una influencia directa en la resolución del set.

Efecto sobre el rendimiento. Cuando el motor detecta un punto de presión, aplica un modificador b_K sobre la precisión del jugador. Esta decisión parte de la idea de que la presión afecta especialmente al control del golpe, más que a la potencia. Es decir, un jugador puede seguir golpeando con fuerza bajo presión, pero su capacidad para dirigir la bola con precisión puede verse alterada.
El factor de ajuste se calcula mediante la siguiente expresión:

$$b_K = 1 + (K - 0{,}5) \cdot 0{,}20 \tag{5.50}$$

donde K es el atributo de *clutch* del jugador normalizado en el intervalo $[0, 1]$. A partir de esta fórmula, el comportamiento del modelo puede interpretarse de la siguiente forma:

- **Jugadores con alto *clutch* ($K > 0{,}5$).** Obtienen una mejora de precisión en puntos clave, con un incremento máximo aproximado del 10 %.
- **Jugadores neutrales ($K = 0{,}5$).** No reciben modificación en su rendimiento por efecto de la presión.
- **Jugadores con bajo *clutch* ($K < 0{,}5$).** Sufren una reducción de precisión en puntos clave, con una penalización máxima aproximada del 10 %.

De esta forma, el atributo *clutch* permite diferenciar jugadores que mantienen mejor su nivel en momentos decisivos de aquellos que tienden a cometer más errores bajo presión. Esto aporta variedad al comportamiento de los tenistas simulados y afecta a estadísticas como los puntos de ruptura convertidos o salvados.

5.5. Reglas de puntuación implementadas

La puntuación en tenis se organiza mediante una estructura jerárquica: los puntos forman juegos, los juegos forman sets y los sets determinan el resultado final del partido. El motor implementa las reglas principales de puntuación necesarias para simular partidos completos, incluyendo juegos con ventaja, *tie-breaks*, sets y partidos al mejor de tres o cinco sets.

5.5.1. Juego estándar

Aunque internamente el motor cuenta los puntos de forma numérica, el marcador se presenta utilizando la nomenclatura habitual del tenis: 0, 15, 30, 40 y ventaja. Para ganar un juego estándar es necesario alcanzar al menos cuatro puntos y mantener una diferencia mínima de dos puntos respecto al rival:

$$\text{Ganar juego} \iff P \geq 4 \ \wedge \ P \geq P_{\text{rival}} + 2 \tag{5.51}$$

donde P representa los puntos ganados por el jugador dentro del juego actual. Si el marcador llega a 40-40, el motor entra en la fase de iguales y ventaja, que continúa hasta que uno de los jugadores consigue la diferencia de dos puntos necesaria para cerrar el juego.

5.5.2. *Tie-break*

Cuando un set alcanza un empate a seis juegos, el motor puede activar una muerte súbita o desempate, conocido como *tie-break*, según la configuración del partido. En esta situación, los puntos se cuentan de forma numérica directa. El objetivo es alcanzar siete puntos, manteniendo siempre una diferencia mínima de dos puntos respecto al rival.

En el caso de un *tie-break* de set decisivo configurado a diez puntos, el objetivo pasa a ser alcanzar diez puntos con diferencia de dos. Esta variante permite simular formatos en los que el desempate final se juega como *super tie-break*, es decir, un desempate a diez puntos.
La rotación del saque sigue el patrón habitual: el primer jugador sirve un punto y, a partir de ese momento, cada jugador sirve dos puntos consecutivos por turno. Además, todos los puntos disputados durante el *tie-break* se consideran puntos de presión, por lo que pueden activar el efecto del atributo *clutch*.

5.5.3. Set

Un set se gana cuando un jugador alcanza seis juegos con una diferencia mínima de dos juegos sobre el rival. Si ambos jugadores llegan a seis juegos, el comportamiento depende de la configuración establecida para el partido: puede disputarse un *tie-break* o continuar hasta que uno de los jugadores consiga una diferencia de dos juegos.

$$\text{Ganar set} \iff G \geq 6 \ \land \ G \geq G_{\text{rival}} + 2 \tag{5.52}$$

donde G representa el número de juegos ganados por el jugador en el set actual.

5.5.4. Partido al mejor de tres y al mejor de cinco

El partido finaliza cuando uno de los jugadores alcanza el número de sets necesarios según la configuración elegida. Para un partido al mejor de tres sets, un jugador necesita ganar dos sets. Para un partido al mejor de cinco sets, necesita ganar tres. De forma general, el número de sets necesarios para ganar el partido se calcula como:

$$\text{Sets necesarios} = \left\lfloor \frac{N}{2} \right\rfloor + 1 \tag{5.53}$$

donde N representa el número máximo de sets configurado para el partido.
Cada vez que comienza una nueva simulación, el motor reinicia el estado dinámico de los jugadores, restaurando la estamina inicial y eliminando las rachas previas. Esto garantiza que cada partido se simule como un evento independiente.

5.6. Control de aleatoriedad y reproducibilidad

La capacidad de replicar resultados es un requisito importante en cualquier sistema de simulación. Aunque el motor de ADAF Tennis Simulator es de naturaleza estocástica, el uso controlado de generadores de números aleatorios permite que el sistema se comporte de forma determinista cuando se fija una semilla inicial. Esta característica resulta útil durante las fases de depuración, validación y comparación de resultados.
El motor gestiona la aleatoriedad a través de dos fuentes principales: la biblioteca estándar de Python y la librería NumPy (8). Para garantizar que ambas fuentes trabajen de forma coordinada, se utiliza una función de inicialización global denominada `seed_all`, encargada de fijar el estado de ambos generadores:

```
def seed_all(seed: int | None) -> None:
    if seed is not None:
        random.seed(seed)
        np.random.seed(seed)
```

Esta función se ejecuta en la capa de entrada pública `run_match()` antes de comenzar cualquier cálculo técnico. Si se proporciona un valor entero como semilla, la secuencia de sucesos del partido será idéntica en ejecuciones posteriores bajo la misma configuración. Por el contrario, si el valor de la semilla es nulo, el sistema no fija explícitamente el estado aleatorio y cada simulación puede producir un desarrollo diferente.

Esta característica permite reproducir errores concretos o comportamientos anómalos durante la simulación para analizarlos paso a paso. También facilita la comparación entre distintos perfiles de jugadores, ya que permite evaluar configuraciones diferentes bajo condiciones controladas.

5.7. Limitaciones actuales del modelo

Aunque el motor de simulación permite generar partidos completos de forma coherente y con variabilidad probabilística, el modelo presenta algunas limitaciones respecto al comportamiento real del tenis. Estas simplificaciones se han adoptado para mantener el sistema dentro de un alcance asumible para el proyecto y para evitar una complejidad excesiva en el cálculo de cada punto.

Una primera limitación es el tratamiento de la superficie de juego. Actualmente, aunque el usuario puede seleccionar la superficie del partido, esta no modifica de forma específica el comportamiento físico de la bola ni los parámetros del motor. En el tenis real, superficies como la tierra batida, la hierba o la pista dura influyen en la velocidad del bote, el deslizamiento y el tipo de intercambios, por lo que su incorporación supondría una mejora relevante del modelo.

Otra limitación es la ausencia de posicionamiento espacial del jugador. El motor no representa coordenadas dentro de la pista ni calcula la posición exacta de cada tenista tras cada golpe. Por este motivo, la elección entre derecha y revés se realiza de forma probabilística a partir de los atributos del jugador, y no en función de su ubicación real o de la dirección concreta de la bola.

El modelo de fatiga también está simplificado. La estamina se representa mediante una única variable dinámica que disminuye en función de la duración de los intercambios y se recupera parcialmente entre juegos y sets. Sin embargo, no se diferencian distintos tipos de desgaste físico, como la fatiga muscular, la resistencia aeróbica o el impacto de condiciones externas como la temperatura o la altitud.

En cuanto al componente mental, el *momentum* se calcula a partir de rachas de puntos, juegos y sets ganados o perdidos. Esto permite introducir una dinámica contextual en el partido, pero no contempla factores externos como la presión del público, la experiencia previa del jugador, el cansancio psicológico acumulado o eventos concretos del encuentro.

Por último, el simulador no modela la dirección exacta del saque ni la zona de destino de los golpes. Por ejemplo, no se distingue si un saque va abierto, al cuerpo o a la

línea central, ni si un golpe de fondo se dirige cruzado o paralelo. Estas decisiones tácticas podrían incorporarse en versiones futuras para aumentar el realismo del punto y enriquecer el modo estratégico.
En conjunto, estas limitaciones no impiden que el motor cumpla su objetivo principal, pero delimitan el alcance del modelo actual. Su identificación permite plantear líneas de mejora futuras orientadas a conseguir una simulación más precisa y cercana al comportamiento real del tenis.

Capítulo 6

Implementación del sistema

En este capítulo se describe la implementación del sistema desarrollado, partiendo de la arquitectura definida en los capítulos anteriores. Se detallan la organización del backend, los principales endpoints de la API, la estructura del frontend basado en plantillas Jinja2 y JavaScript, la visualización dinámica de la simulación y el sistema de autenticación mediante JWT.

6.1. Implementación del backend

Como se explicó en el Capítulo 4, la aplicación presenta una arquitectura modular en la que cada parte del sistema agrupa una responsabilidad concreta. En el backend, esta separación se refleja en módulos funcionales que contienen sus propios modelos ORM, esquemas de validación Pydantic, rutas o endpoints y, cuando es necesario, funciones auxiliares específicas.
Esta organización permite desarrollar, probar y modificar cada parte del sistema de forma más independiente, reduciendo el acoplamiento entre funcionalidades como autenticación, gestión de jugadores, simulación de partidos, torneos, modo entrenador o análisis Big Data.

6.1.1. Organización del proyecto

El código del servidor se estructura dentro del directorio `backend`, siguiendo una organización orientada a módulos funcionales.

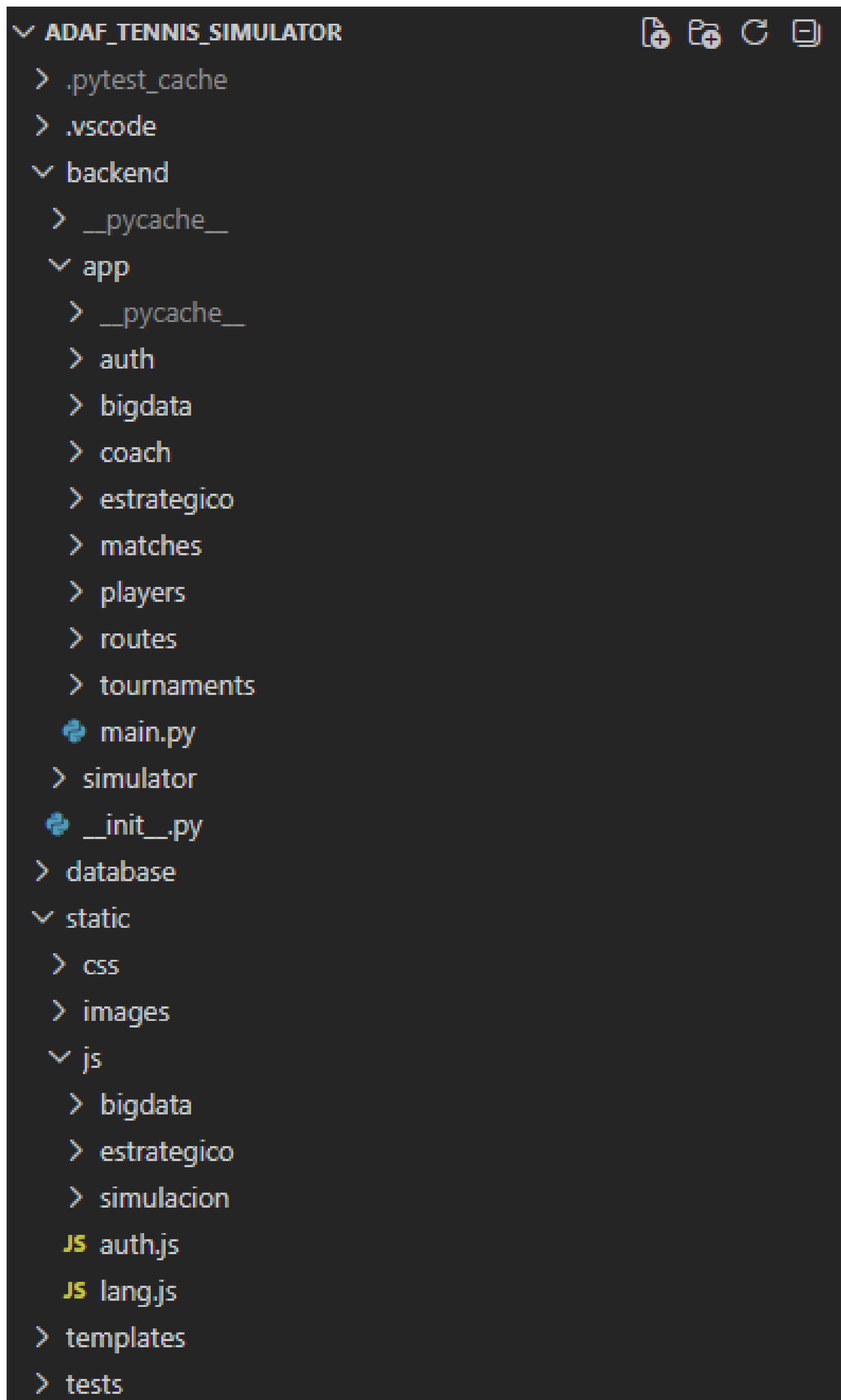

Figura 6.1: Organización de carpetas del backend

El fichero `main.py` constituye el punto de entrada de la aplicación. En él se instancia el objeto principal de FastAPI, se configura el middleware CORS para permitir

peticiones desde el cliente, se monta el directorio de archivos estáticos `static` y se registran los distintos routers bajo sus respectivos prefijos.
Entre estos prefijos se encuentran `/api/auth`, `/api/players`, `/api/matches`, `/api/tournaments`, `/api/estrategico`, `/api/bigdata`, `/api` y `/`, este último destinado a las rutas que devuelven páginas renderizadas mediante Jinja2. Esta organización permite añadir o modificar un módulo funcional sin afectar de forma directa al resto de la aplicación.

6.1.2. Endpoints principales

La API se organiza en varios grupos de endpoints, cada uno asociado a una funcionalidad principal del sistema:

- **Simulación de partidos** (`POST /api/simulate_match`).

  Es el endpoint central de la aplicación. Recibe los datos de ambos jugadores, incluyendo nombre, identificador y atributos numéricos, junto con la configuración del partido, como formato, uso de *tie-break*, semilla o superficie.

  Este endpoint delega la ejecución del partido en la función `run_match()` del motor de simulación y devuelve una respuesta JSON con el ganador, el marcador por sets, el *timeline* completo y las estadísticas agregadas de cada jugador, calculadas por el módulo `stats.py`.

  El flujo completo de interacción entre componentes se describió en la Sección 4.4. En el caso de partidos simulados por usuarios registrados, los resultados se almacenan automáticamente en la base de datos.

- **Autenticación** (`POST /api/auth/register`, `POST /api/auth/login`, `GET /api/auth/me`, `GET /api/auth/me/matches`).

  Estos endpoints se encargan del registro de usuarios, la validación de duplicados, el inicio de sesión mediante nombre de usuario o correo electrónico y la emisión de tokens JWT. También permiten consultar la información del usuario autenticado y su historial de partidos simulados.

- **Gestión de jugadores** (`GET /api/players`, `POST /api/players`).

  Permiten consultar los jugadores disponibles para el usuario, incluyendo tanto los jugadores propios como los jugadores predefinidos del sistema. También permiten crear nuevos jugadores personalizados con sus datos personales y atributos técnicos.

- **Consulta de partidos** (`GET /api/matches/player/{id}`, `GET /api/matches/{id}`).

  Estos endpoints permiten consultar el historial de partidos de un jugador concreto y acceder al detalle completo de un partido, incluyendo marcador y estadísticas desglosadas.

- **Torneos** (`POST /api/tournaments`, `POST /api/tournaments/simulate-round`).

Permiten crear torneos eliminatorios y simular sus rondas. El backend se encarga de generar los emparejamientos, ejecutar los partidos correspondientes y avanzar automáticamente a los ganadores dentro del cuadro.

- **Modo estratégico** (`POST /api/estrategico/start`, `POST /api/estrategico/next-point`, `POST /api/estrategico/set-strategy`).

  Estos endpoints gestionan sesiones interactivas en memoria del servidor, donde el usuario dirige a un jugador punto a punto eligiendo la estrategia entre puntos.

- **Simulación masiva o Big Data** (`POST /api/bigdata/simulate`, `POST /api/bigdata/simulate-stream`).

  Permiten ejecutar múltiples simulaciones del mismo enfrentamiento y generar estadísticas agregadas a partir de los resultados obtenidos.

6.1.3. Serialización y estructura de respuestas

La comunicación entre el backend y el frontend se realiza principalmente mediante respuestas en formato JSON. FastAPI utiliza esquemas de datos definidos con Pydantic para validar la información recibida en las peticiones y estructurar los datos devueltos al cliente.
En las operaciones de creación, como el registro de usuarios o la creación de jugadores, los esquemas Pydantic permiten comprobar que los campos recibidos cumplen el formato esperado antes de ejecutar la lógica de negocio. En el caso de los jugadores, se validan tanto los datos personales como los nueve atributos de rendimiento en escala de 1 a 100.
En las operaciones de simulación, el backend devuelve objetos JSON adaptados a las necesidades del frontend. Por ejemplo, en una simulación de partido se devuelve el ganador, el marcador por sets, el flujo punto a punto del encuentro y las estadísticas agregadas. Esta estructura permite que los módulos JavaScript puedan reproducir visualmente el partido y construir posteriormente la pantalla de resumen.

6.1.4. Implementación del CRUD de jugadores

La gestión de jugadores se implementa como un módulo independiente dentro de `players/`. Este módulo permite crear y consultar jugadores, cuyos datos quedan asociados al usuario autenticado y disponibles para las simulaciones.
Cada jugador se modela mediante la clase ORM `Jugador`, que almacena información personal como nombre, apellido, nacionalidad, altura y mano dominante. Además, contiene nueve atributos de rendimiento valorados en una escala de 1 a 100: primer saque, segundo saque, resto, derecha, revés, movilidad, consistencia, *clutch* y físico. El modelo incluye restricciones `CHECK` a nivel de base de datos que impiden almacenar valores fuera del rango válido para cada atributo. Estas restricciones actúan como una segunda capa de validación, adicional a las comprobaciones realizadas en el backend mediante Pydantic.

Creación de jugadores. En el endpoint `POST /api/players`, el usuario envía los datos personales del jugador y el diccionario con los nueve atributos técnicos. El backend valida el token JWT, normaliza la nacionalidad a un código de tres caracteres en mayúsculas y convierte la mano dominante del formato textual utilizado en la interfaz al formato almacenado en la base de datos.
A continuación, se crea la instancia `Jugador` asociada al usuario autenticado y se persiste en PostgreSQL.

Consulta de jugadores. El endpoint `GET /api/players` devuelve una lista combinada de jugadores propios del usuario y jugadores predefinidos del sistema. Estos últimos se corresponden con jugadores sin creador asignado. La consulta filtra los jugadores activos y los devuelve ordenados alfabéticamente.
El esquema de respuesta `JugadorOut` omite campos internos de la base de datos y devuelve únicamente la información necesaria para que el frontend pueda mostrar y seleccionar jugadores.

6.2. Implementación del frontend

6.2.1. Estructura de vistas SSR

Las vistas de la aplicación se generan en el servidor mediante Jinja2 y se sirven a través del router `pages.py`. Cada ruta asociada a una vista renderiza una plantilla HTML que extiende una plantilla base o una plantilla específica para autenticación.
La plantilla `base.html` define la estructura común de las páginas privadas de la aplicación. En ella se incluye la carga de Tailwind CSS, las animaciones CSS principales y dos zonas principales: la barra lateral de navegación, incluida como *partial* mediante `sidebar.html`, y el área de contenido principal. La barra lateral dispone de estados expandido y colapsado, con transiciones CSS suaves, y el margen del contenido se ajusta en función de su estado.
Cada página concreta rellena bloques como `title`, `head-extra` y `content`. Para las vistas de autenticación, como login y registro, se utiliza una plantilla alternativa, `auth.html`, que omite la barra lateral y presenta un diseño centrado, ya que el usuario todavía no ha iniciado sesión.
Las páginas principales del sistema son:

- `index.html`: página de inicio con presentación del proyecto.
- `menu.html`: menú principal con acceso a las funcionalidades principales.
- `crear-jugador.html`: formulario de creación de jugador con controles para sus atributos.
- `crear-partido.html` y `partido-rapido.html`: configuración y lanzamiento de partidos.
- `simulacion.html`: pantalla de reproducción de la simulación.
- `resumen.html`: vista de estadísticas posteriores al partido.

- `resultados.html`: historial de partidos simulados.
- `torneo-setup.html` y `torneo-bracket.html`: configuración y visualización de torneos.
- `perfil.html`: perfil del usuario autenticado.

A continuación se muestran algunas capturas representativas de las vistas principales de la aplicación.

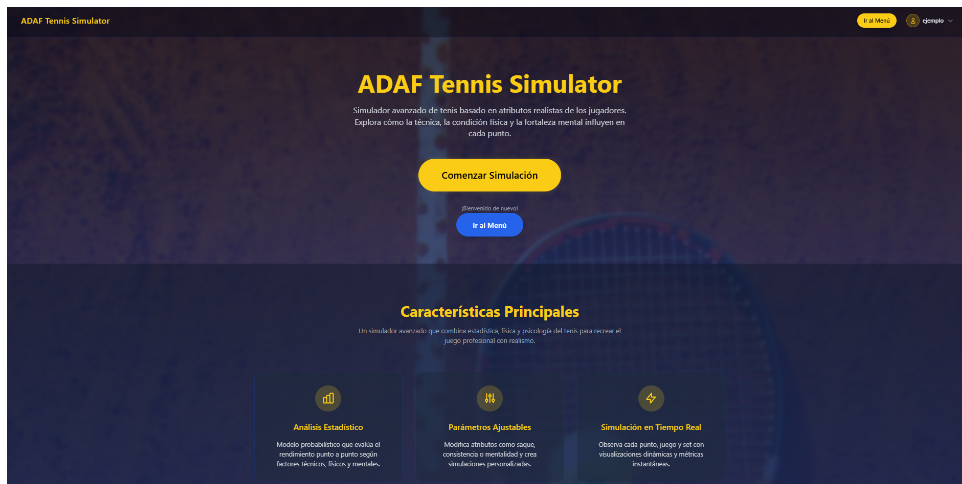

Figura 6.2: Página de inicio de la aplicación

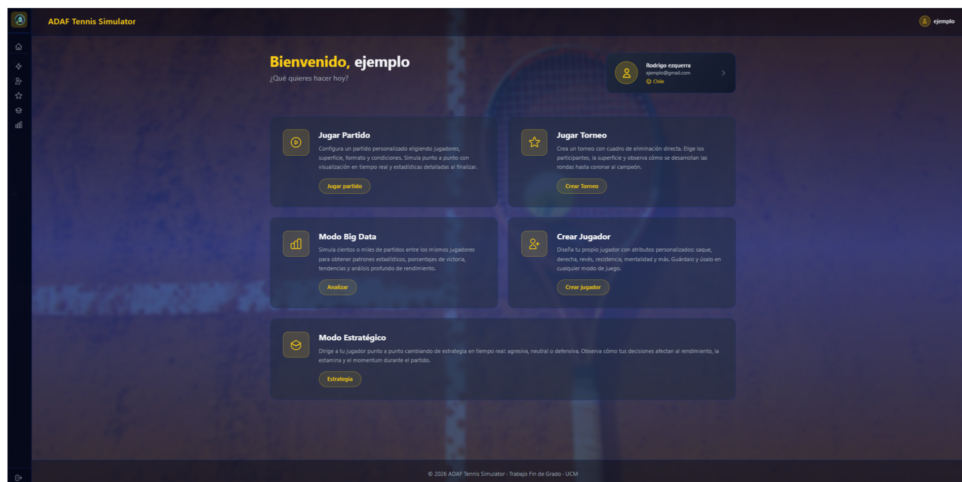

Figura 6.3: Menú principal de la aplicación

Figura 6.4: Formulario de creación de jugador

Figura 6.5: Formulario de configuración de partido rápido

Figura 6.6: Vista de reproducción de un partido rápido

Figura 6.7: Vista de estadísticas de un partido

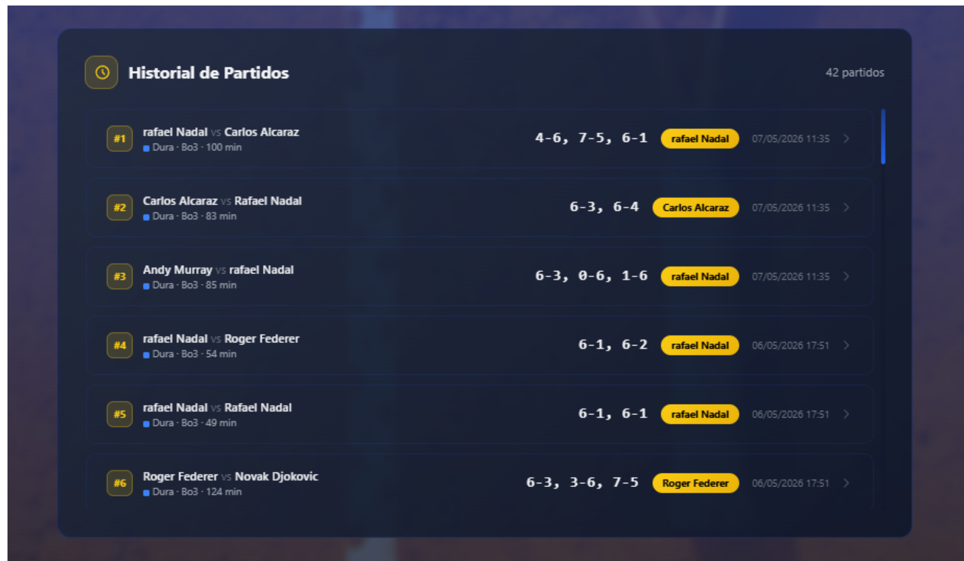

Figura 6.8: Vista del historial de partidos de un usuario

Figura 6.9: Gráficos de estadísticas de un partido

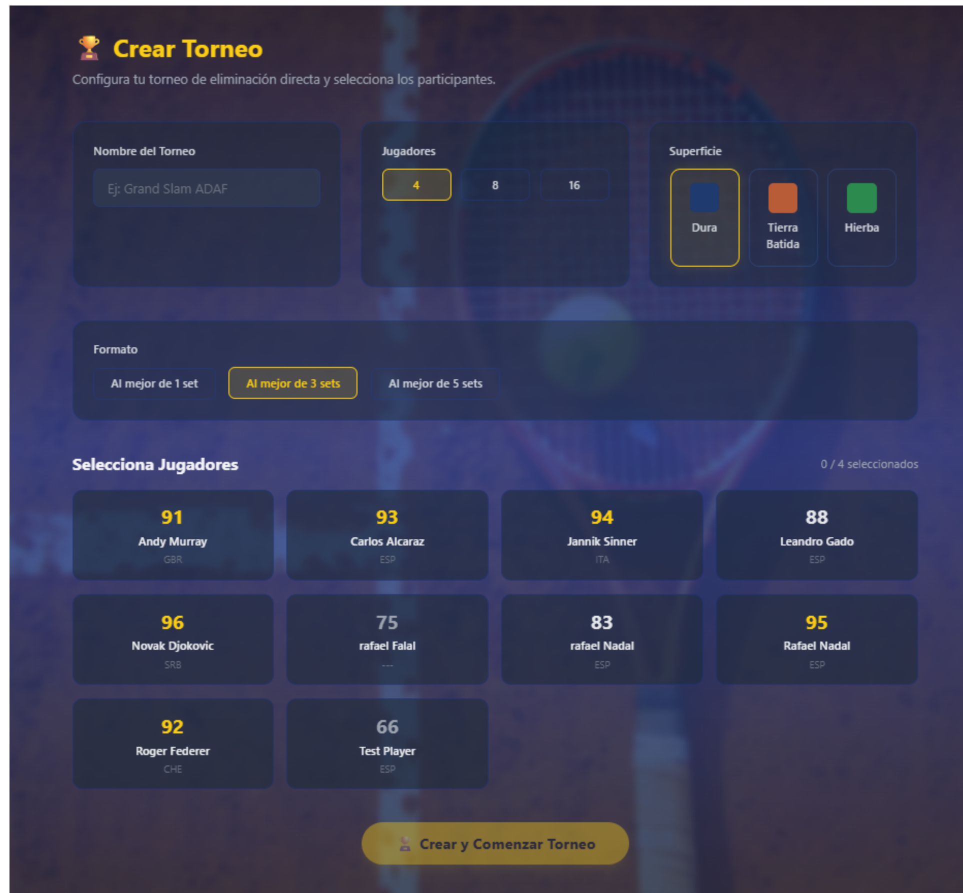

Figura 6.10: Vista de creación de torneo

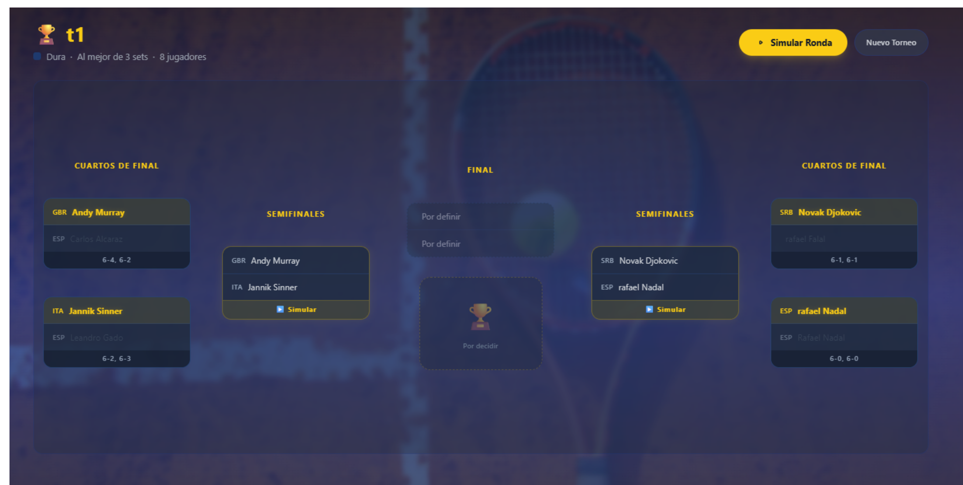

Figura 6.11: Vista del cuadro de torneo

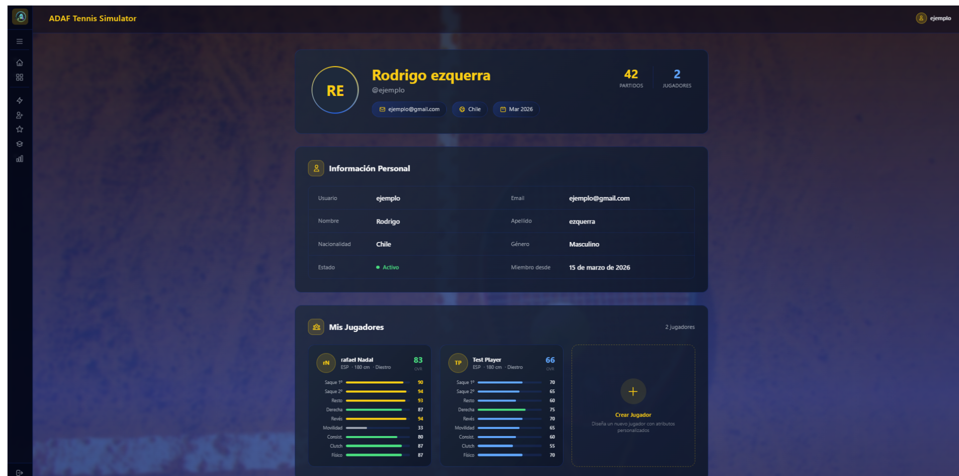

Figura 6.12: Vista del perfil del usuario

6.2.2. Interacción dinámica con JavaScript

El comportamiento dinámico del frontend se ha estructurado mediante módulos JavaScript. Esta lógica se utiliza especialmente en la pantalla de simulación, donde es necesario reproducir el partido punto a punto, actualizar el marcador y mostrar la narración del encuentro.
El archivo principal, `app.js`, actúa como controlador de la pantalla de simulación. Cuando el usuario accede a esta vista, se ejecutan varias acciones de inicialización:

1. Se inicializa el generador de narración cargando las bibliotecas JSON de frases mediante `fetch`.

2. Se recuperan los datos del partido desde `sessionStorage` (16), donde fueron almacenados tras recibir la respuesta del backend.

3. Se guardan los datos en el estado centralizado, se establecen los nombres de los jugadores en la interfaz, se inicializan los paneles de estadísticas en vivo y se activan los controles de simulación.

El módulo `state.js` implementa un gestor de estado centralizado. Este módulo almacena el JSON completo del partido, la línea temporal de puntos, el índice del punto actual y las estadísticas acumuladas en vivo. También expone funciones de lectura y actualización, como `getState()`, `getCurrentPoint()`, `setMatchData()`, `nextPoint()`, `previousPoint()`, `endMatch()` y `resetState()`.
Fuera de la pantalla de simulación, también se utilizan peticiones asíncronas mediante `fetch` para operaciones como el inicio de sesión, el registro, la creación de jugadores, la carga del historial de partidos y la consulta de datos del perfil, evitando recargas completas de página.

6.2.3. Marcador en tiempo real

El módulo `scoreboard.js` mantiene actualizado el marcador visual durante la reproducción de la simulación. Cada vez que se avanza un punto, la función

`updateScoreboard()` recibe el objeto correspondiente y actualiza la interfaz.
Esta actualización incluye:

- Los juegos de cada set, mostrando los sets completados, el set en curso y dejando vacías las columnas de sets futuros.
- Los puntos del juego actual, utilizando la notación 0, 15, 30, 40 y ventaja en juegos normales, y puntuación numérica directa en *tie-breaks*.
- El indicador de saque, activando una animación junto al nombre del jugador que está sacando.
- El resaltado visual del jugador que va por delante en el partido.

La sincronización con los datos del backend es directa. Como el servidor ya calcula y envía el marcador exacto tras cada punto, incluyendo juegos, puntos y *tie-breaks*, el frontend se limita a representar esa información en pantalla sin recalcular el resultado del partido.

6.2.4. Feed narrativo punto a punto

El sistema de narración genera frases que describen lo sucedido en cada punto, imitando el estilo de una retransmisión deportiva.
La generación de frases se realiza mediante la clase `PointFeedGenerator`. Su constructor recibe una biblioteca de frases cargada desde ficheros JSON y organizada en varias categorías, como saques, restos, peloteos, alcance de la bola y mensajes generales.
Para cada punto, el método `generateFeedForPoint()` recorre el conjunto de acciones registradas por el motor de simulación y selecciona una frase aleatoria de la categoría correspondiente. Estas frases pueden incluir marcadores de posición que se sustituyen por datos dinámicos, como el nombre real del jugador. Al final del punto se añade una frase de cierre según el resultado producido, como ace, doble falta, error no forzado o winner.
La visualización en pantalla la gestiona el módulo `liveFeed.js`. Para cada punto, este módulo genera un elemento HTML con un icono representativo, una descripción textual y un estilo visual asociado al jugador ganador. En modo paso a paso, las frases se muestran secuencialmente con un retardo entre ellas. En modo de reproducción automática, se muestra directamente la frase resumen para mantener la fluidez de la simulación.

6.2.5. Visualización de estadísticas con Chart.js

Al finalizar el partido, la vista `resumen.html` presenta las estadísticas completas generadas a partir de la simulación. El módulo `summary.js` lee los datos del partido desde `sessionStorage` y construye los distintos elementos de la pantalla de resumen.
Entre estos elementos se incluyen:

- Un banner con el ganador y el marcador final.

- Una tabla comparativa organizada por categorías, como saque, resto y puntos, con barras de progreso para comparar visualmente ambos jugadores.
- Estadísticas como aces, dobles faltas, porcentaje de primer saque, puntos ganados con primer y segundo saque, puntos ganados al saque y al resto, bolas de break convertidas, winners, errores no forzados, ratio entre winners y errores, media de duración del rally y rally más largo.
- Gráficos generados con Chart.js, incluyendo gráficos de donut, barras, radar, histograma y líneas.
- Un gráfico de evolución del *momentum*, que representa el impulso de cada jugador a lo largo de los puntos del partido.

6.3. Implementación del sistema de autenticación

El diseño general del sistema de autenticación, incluyendo el uso de *bcrypt* para el hash de contraseñas, la emisión de tokens JWT y su verificación en peticiones protegidas, se describió en la Sección 4.3.4. En este apartado se detallan los aspectos concretos de su implementación.

6.3.1. Registro y login

El registro se implementa en el endpoint `POST /api/auth/register`. El cuerpo de la petición se valida mediante el esquema Pydantic `RegisterRequest`, que exige nombre de usuario, correo electrónico, contraseña y nombre, y acepta opcionalmente apellido, nacionalidad y género.
Antes de crear el usuario, el sistema comprueba que no exista otro registro con el mismo nombre de usuario o correo electrónico. En caso de duplicado, devuelve un error `409 Conflict`. Si los datos son válidos, la contraseña se transforma mediante *bcrypt* antes de almacenarse y se crea una instancia del modelo `Usuario` en PostgreSQL.
El inicio de sesión se implementa en el endpoint `POST /api/auth/login`. Este endpoint admite tanto el nombre de usuario como el correo electrónico como identificador. El sistema verifica la contraseña introducida comparándola con el hash almacenado y comprueba que la cuenta esté marcada como activa. En caso de éxito, genera un token JWT y lo devuelve junto con los datos básicos del usuario mediante el esquema `TokenResponse`.

6.3.2. Gestión de tokens

Los tokens se generan mediante la función `create_access_token()`, utilizando el algoritmo `HS256` y una clave secreta configurable mediante variable de entorno. La caducidad por defecto del token es de siete días.
La verificación se realiza mediante la función `decode_access_token()`, que devuelve el contenido del token si este es válido, o `None` si ha expirado o ha sido manipulado. Ambas funciones utilizan la librería `python-jose`.

En el lado del cliente, el token se almacena en `localStorage`. El módulo `api.js` lo adjunta automáticamente en la cabecera `Authorization: Bearer <token>` de cada petición a la API que requiere autenticación.

6.3.3. Protección de rutas y sesiones

Los endpoints que requieren autenticación utilizan el esquema `HTTPBearer` de FastAPI. FastAPI extrae el token de la cabecera de la petición y el backend lo decodifica para obtener el identificador del usuario.
Este identificador se utiliza para filtrar resultados, asociar acciones al usuario correcto y verificar que la cuenta sigue activa en la base de datos. Si el token es inválido, ha expirado o no se proporciona, la petición se rechaza con un error `401 Unauthorized`.
El endpoint `GET /api/auth/me` permite al frontend comprobar en cualquier momento si la sesión sigue activa y obtener la información básica del usuario autenticado.

Capítulo 7

Persistencia de datos y funcionalidades avanzadas

En este capítulo se describe cómo se almacena, organiza y recupera la información generada por ADAF Tennis Simulator. Para ello, se presenta el diseño de la base de datos, las tablas principales y las relaciones entre usuarios, jugadores, partidos, estadísticas y torneos.
Además, se explican varias funcionalidades avanzadas de la aplicación que dependen de esta persistencia de datos o amplían el uso básico del simulador: el historial de partidos, los torneos eliminatorios, el partido rápido, el modo estratégico y el modo Big Data.

7.1. Diseño de la base de datos

La base de datos forma parte del *backend* de la aplicación, junto con el motor de simulación y la lógica encargada de procesar las peticiones del usuario. Su función es garantizar que la información generada por la aplicación se conserve de forma persistente y pueda ser recuperada posteriormente.
Los jugadores creados por los usuarios, los partidos simulados, las estadísticas asociadas y los torneos quedan almacenados de forma estructurada en PostgreSQL, el sistema gestor de bases de datos utilizado en el proyecto.

7.1.1. Modelo entidad-relación

El modelo entidad-relación permite representar las entidades principales que la aplicación necesita almacenar y las relaciones existentes entre ellas. En ADAF Tennis Simulator, el modelo se construye en torno a cinco entidades principales:

- **Usuario**: representa a la persona que utiliza la aplicación. Almacena información como el nombre de usuario, el correo electrónico y la contraseña protegida mediante hash.
- **Jugador**: representa a un tenista que puede participar en simulaciones. Contiene datos personales y atributos deportivos. Puede pertenecer a un usuario

concreto o ser un jugador predefinido del sistema.

- **Partido**: representa una simulación disputada entre dos jugadores. Almacena información como los participantes, el ganador, el marcador final y la configuración utilizada.

- **Estadísticas de partido**: recoge los datos estadísticos de cada jugador dentro de un partido concreto, como aces, dobles faltas, winners o errores no forzados.

- **Torneo**: representa una competición eliminatoria formada por varios partidos y asociada a un usuario creador.

La Figura 7.1 muestra el modelo entidad-relación utilizado en la aplicación.

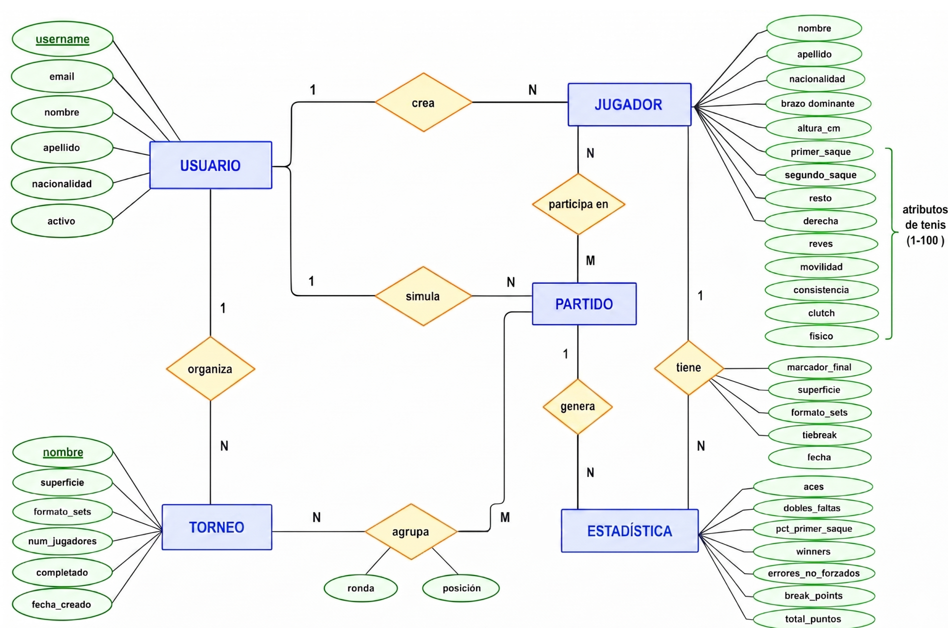

Figura 7.1: Modelo entidad-relación de la base de datos

7.1.2. Tablas principales

Cada una de las entidades anteriores se implementa mediante una tabla en la base de datos. A continuación se describen las tablas principales del sistema.

Campo	Tipo	Descripción
`id`	SERIAL PK	Identificador único autoincremental.
`username`	VARCHAR(50)	Nombre de usuario único.
`email`	VARCHAR(100)	Correo electrónico único.
`password_hash`	VARCHAR(255)	Contraseña protegida mediante hash.
`nombre`	VARCHAR(100)	Nombre real del usuario.
`apellido`	VARCHAR(100)	Apellido del usuario.
`nacionalidad`	VARCHAR(50)	País de origen.
`activo`	BOOLEAN	Indica si la cuenta está activa.
`created_at`	TIMESTAMP	Fecha de registro.

Tabla 7.1: Tabla `usuarios`

Campo	Tipo	Descripción
id	SERIAL PK	Identificador único.
nombre	VARCHAR(100)	Nombre del jugador.
apellido	VARCHAR(100)	Apellido del jugador.
nacionalidad	CHAR(3)	Código de país del jugador.
altura_cm	INT	Altura en centímetros.
brazo_bueno	CHAR(1)	Brazo dominante: R o L.
id_creador	INT FK	Usuario propietario. Si es NULL, se trata de un jugador del sistema.
attr_primer_saque	INT	Calidad del primer saque, en escala 1–100.
attr_segundo_saque	INT	Calidad del segundo saque, en escala 1–100.
attr_resto	INT	Capacidad de resto, en escala 1–100.
attr_derecha	INT	Calidad del golpe de derecha, en escala 1–100.
attr_reves	INT	Calidad del golpe de revés, en escala 1–100.
attr_movilidad	INT	Movilidad y cobertura de pista, en escala 1–100.
attr_consistencia	INT	Regularidad del jugador, en escala 1–100.
attr_clutch	INT	Rendimiento en puntos importantes, en escala 1–100.
attr_fisico	INT	Condición física del jugador, en escala 1–100.

Tabla 7.2: Tabla jugadores

Campo	Tipo	Descripción
`id`	SERIAL PK	Identificador único.
`id_jugador_1`	INT FK	Primer jugador del partido.
`id_jugador_2`	INT FK	Segundo jugador del partido.
`id_ganador`	INT FK	Jugador ganador.
`id_usuario_creador`	INT FK	Usuario que lanzó la simulación.
`marcador_final`	VARCHAR(50)	Resultado final del partido.
`duracion_minutos`	INT	Duración estimada del partido.
`superficie`	VARCHAR(20)	Superficie configurada para el partido.
`formato_sets`	INT	Formato del partido: 1, 3 o 5 sets.
`tiebreak_ultimo_set`	BOOLEAN	Indica si se juega *tie-break* en el set decisivo.
`fecha_jugado`	TIMESTAMP	Fecha y hora de la simulación.

Tabla 7.3: Tabla `partidos`

Campo	Tipo	Descripción
`id`	SERIAL PK	Identificador único.
`id_partido`	INT FK	Partido al que pertenecen las estadísticas.
`id_jugador`	INT FK	Jugador al que corresponden las estadísticas.
`aces`	INT	Número de aces.
`dobles_faltas`	INT	Número de dobles faltas.
`primeros_saques_in`	INT	Primeros saques válidos.
`primeros_saques_total`	INT	Total de primeros saques intentados.
`winners`	INT	Golpes ganadores.
`errores_no_forzados`	INT	Errores no forzados.
`break_points_convertidos`	INT	Oportunidades de rotura convertidas.
`break_points_oportunidades`	INT	Oportunidades de rotura totales.
`total_puntos_ganados`	INT	Total de puntos ganados en el partido.

Tabla 7.4: Tabla `estadisticas_partido`

Campo	**Tipo**	**Descripción**
`id`	SERIAL PK	Identificador único.
`nombre`	VARCHAR(150)	Nombre del torneo.
`id_usuario_creador`	INT FK	Usuario que organizó el torneo.
`superficie`	VARCHAR(20)	Superficie configurada para todos los partidos.
`formato_sets`	INT	Formato de los partidos.
`num_jugadores`	INT	Número de participantes: 4, 8 o 16.
`id_ganador`	INT FK	Jugador campeón. Es `NULL` hasta que el torneo finaliza.
`completado`	BOOLEAN	Indica si el torneo ha finalizado.
`fecha_creado`	TIMESTAMP	Fecha de creación.

Tabla 7.5: Tabla `torneos`

Además de estas tablas principales, se utiliza una tabla intermedia para representar la relación entre torneos y partidos. Esta tabla es necesaria porque la relación contiene información propia, como la ronda y la posición dentro del cuadro. Estos datos no pertenecen únicamente al torneo ni únicamente al partido, sino al vínculo entre ambos.

Campo	**Tipo**	**Descripción**
`id`	SERIAL PK	Identificador único.
`id_torneo`	INT FK	Torneo al que pertenece el enfrentamiento.
`id_partido`	INT FK	Partido simulado.
`ronda`	INT	Ronda del torneo.
`posicion`	INT	Posición dentro de la ronda.
`id_jugador_1`	INT FK	Primer participante del enfrentamiento.
`id_jugador_2`	INT FK	Segundo participante del enfrentamiento.
`id_ganador`	INT FK	Ganador del enfrentamiento.
`completado`	BOOLEAN	Indica si el enfrentamiento ya se ha simulado.

Tabla 7.6: Tabla `torneo_partidos`

En el modelo también existe una relación entre jugadores y partidos. Un jugador puede participar en muchos partidos y cada partido tiene dos jugadores. Aunque conceptualmente puede verse como una relación de muchos a muchos, en este caso se ha decidido modelarla directamente en la tabla `partidos` mediante los campos `id_jugador_1`, `id_jugador_2` e `id_ganador`.
Esta decisión se tomó porque un partido individual de tenis siempre tiene exactamente dos participantes con roles conocidos. Por tanto, una tabla intermedia habría añadido complejidad sin aportar una ventaja clara para el alcance del proyecto.

7.1.3. Relaciones entre usuarios, jugadores, partidos y torneos

Las tablas se conectan entre sí mediante claves foráneas. Por ejemplo, el campo `id_creador` de la tabla `jugadores` referencia a la tabla `usuarios`, indicando qué usuario ha creado cada jugador. Estas relaciones permiten mantener la coherencia de los datos y evitar referencias a elementos inexistentes.
Como se observa en la Figura 7.1, las relaciones principales son las siguientes:

- Un usuario puede crear muchos jugadores.
- Un usuario puede simular muchos partidos.
- Un jugador puede participar en muchos partidos.
- Un partido genera dos registros de estadísticas, uno por cada jugador.
- Un usuario puede organizar muchos torneos.
- Un torneo agrupa varios enfrentamientos a través de la tabla `torneo_partidos`.

El sistema aplica reglas de integridad referencial para evitar datos huérfanos. En algunos casos se utiliza borrado en cascada, mientras que en otros se conserva la información histórica eliminando únicamente la referencia al usuario asociado. Esta política permite mantener coherencia en la base de datos sin perder necesariamente toda la información histórica generada por la aplicación.

7.2. Historial de partidos y almacenamiento de estadísticas

Cada vez que un usuario registrado simula un partido completo, el resultado se guarda automáticamente en la base de datos al finalizar la ejecución del motor. Este proceso se realiza sin que el usuario tenga que realizar ninguna acción adicional.
El endpoint `POST /api/simulate-match`, cuyo flujo general se explicó en la Sección 4.4, desencadena el guardado tras recibir el resultado generado por `run_match()`.
Por cada simulación se almacena:

- Una fila en la tabla `partidos`, con el marcador final, la superficie, el formato, la duración estimada y el jugador ganador.
- Dos filas en la tabla `estadisticas_partido`, una por cada jugador, con valores como aces, dobles faltas, porcentaje de primer saque, winners, errores no forzados, bolas de break y total de puntos ganados.

El usuario puede consultar posteriormente sus partidos desde la sección de historial de la aplicación. Esta vista muestra los partidos ordenados por fecha, junto con información como los jugadores, el marcador, la superficie y la duración estimada. Al acceder al detalle de un partido se presenta una comparación estadística completa de ambos jugadores.

7.3. Simulación de torneos eliminatorios

El sistema permite organizar y simular torneos en formato eliminatorio directo. En este formato, el perdedor de cada partido queda eliminado y el ganador avanza a la siguiente ronda hasta que se determina un campeón.

Creación del torneo. El usuario selecciona entre 4, 8 o 16 jugadores, asigna un nombre al torneo y define una superficie y un formato de partido común para todos los enfrentamientos. En el momento de crear el torneo, el sistema genera automáticamente el cuadro inicial, emparejando a los participantes de la primera ronda y dejando reservados los espacios de las rondas posteriores.

Simulación por rondas. El usuario simula las rondas de una en una. Al pulsar el botón correspondiente, el sistema ejecuta automáticamente todos los partidos de la ronda actual utilizando el mismo motor de simulación empleado en el resto de la aplicación. Cada partido se calcula completo, con su resultado y estadísticas, y queda almacenado en la base de datos.
Los ganadores avanzan automáticamente al espacio correspondiente de la siguiente ronda. Por ejemplo, en un torneo de 8 jugadores el proceso sería:

- Primera ronda o cuartos de final: 4 partidos y 8 jugadores, de los que avanzan 4 ganadores.
- Semifinales: 2 partidos y 4 jugadores, de los que avanzan 2 ganadores.
- Final: 1 partido y 2 jugadores, del que sale el campeón.

Visualización del cuadro. La vista del torneo muestra el cuadro eliminatorio con los enfrentamientos, los resultados de los partidos ya simulados y los huecos pendientes. Una vez completadas todas las rondas, el torneo queda marcado como finalizado y el campeón se registra en la base de datos.

7.4. Modos de uso del sistema

Con el objetivo de adaptarse a distintos tipos de uso, la aplicación ofrece varios modos además de la simulación estándar de un partido completo.

7.4.1. Partido rápido

El partido rápido es el modo principal de simulación. El usuario accede a la pantalla de creación de partido, selecciona dos jugadores, configura la superficie, el formato del partido y el uso de *tie-break* en el último set. Al iniciar la simulación, el motor calcula el partido completo y el usuario puede reproducirlo punto a punto desde la pantalla de simulación.
Durante la reproducción se muestra el marcador actualizado, la narración textual de cada punto y, al finalizar, el resumen estadístico completo del encuentro.

Este modo también está disponible en una variante de exhibición sin necesidad de registro, con partidos predefinidos entre tenistas del sistema. Su objetivo es permitir que cualquier visitante pueda probar la aplicación sin crear una cuenta.

7.4.2. Modo estratégico

El modo estratégico permite al usuario intervenir durante el partido tomando decisiones tácticas punto a punto. En lugar de limitarse a reproducir una simulación ya calculada, el usuario dirige a uno de los jugadores y selecciona la estrategia que desea aplicar antes de cada punto.
El usuario elige dos jugadores y decide cuál de ellos va a controlar. Antes de cada punto puede seleccionar una de las siguientes estrategias:

- **Agresiva**: el jugador asume más riesgos y busca golpes más determinantes, a costa de aumentar la probabilidad de error.
- **Neutral**: el jugador mantiene el comportamiento base del motor, sin aplicar modificaciones relevantes.
- **Defensiva**: el jugador prioriza la consistencia y reduce riesgos, aunque puede ceder iniciativa al rival.

Cada estrategia modifica los parámetros utilizados por el motor para calcular el punto correspondiente. Tras cada punto, el sistema muestra el resultado, el marcador actualizado y el estado dinámico de ambos jugadores, incluyendo estamina y *momentum*. De esta forma, el usuario puede adaptar sus decisiones al desarrollo del partido.

7.4.3. Modo Big Data

El modo Big Data está diseñado para obtener conclusiones estadísticas sobre el rendimiento comparado de dos jugadores. En lugar de simular un único partido, este modo ejecuta múltiples partidos entre los mismos jugadores y agrega los resultados obtenidos.
El usuario configura el número de simulaciones, los dos jugadores, la superficie y el formato del partido. Al lanzar el análisis, el sistema simula todos los encuentros de forma automatizada y calcula para cada jugador distintos indicadores:

- Porcentaje de victorias sobre el total de partidos simulados, entendido como una probabilidad estimada dentro del modelo.
- Distribución de marcadores, indicando con qué frecuencia aparece cada resultado.
- Medias estadísticas, como aces, dobles faltas, winners, errores no forzados o porcentaje de bolas de break convertidas.
- Distribución de la duración de los puntos, agrupando los rallies por número de golpes.

- Evolución del porcentaje de victorias a medida que aumenta el número de simulaciones.

Para simulaciones grandes, la aplicación utiliza un sistema de streaming que envía el progreso al navegador en tiempo real. De esta forma, el usuario puede ver una barra de progreso y la evolución parcial de los resultados mientras el servidor calcula los partidos.

Capítulo 8

Pruebas y validación

En este capítulo se describe el proceso seguido para comprobar el correcto funcionamiento de ADAF Tennis Simulator. La validación se ha centrado tanto en la aplicación web completa como en el motor de simulación, ya que este último constituye la parte más crítica del sistema.
El objetivo de estas pruebas no es demostrar que el sistema esté libre de errores en cualquier situación posible, sino verificar que las funcionalidades principales se comportan de forma coherente, que los datos se almacenan y recuperan correctamente, y que el motor genera resultados válidos y reproducibles cuando se utiliza una semilla aleatoria fija.

8.1. Estrategia de validación

La validación del sistema se ha llevado a cabo en dos niveles. Por un lado, se realizaron pruebas manuales sobre la aplicación completa, comprobando que los flujos principales de usuario funcionaban correctamente desde la interfaz web. Por otro lado, se escribieron pruebas automatizadas para verificar el comportamiento del motor de simulación, que es el componente más crítico del proyecto.
Las pruebas manuales se centraron en validar la experiencia de uso, la navegación entre vistas, la persistencia de datos y el funcionamiento de las principales funcionalidades del sistema. Las pruebas automáticas, en cambio, se orientaron a comprobar la coherencia interna del motor, la corrección de las reglas de puntuación y la reproducibilidad de las simulaciones mediante semilla aleatoria.
Las pruebas automatizadas se encuentran en la carpeta `tests/` del proyecto y se ejecutan mediante `pytest`. Estas pruebas se dividen en pruebas unitarias, centradas en componentes concretos del motor, y pruebas de integración, centradas en la ejecución completa de partidos simulados.

8.2. Pruebas manuales de la aplicación

Las pruebas manuales consistieron en recorrer los principales flujos de uso de la aplicación desde el navegador, comprobando tanto el funcionamiento esperado como

la respuesta ante entradas incorrectas. Esta validación permitió comprobar que la interfaz se comunicaba correctamente con el backend y que los datos generados por el sistema se mostraban de forma adecuada al usuario.
La Tabla 8.1 resume las principales pruebas realizadas.

Prueba	**Descripción**	**Resultado**
Registro e inicio de sesión	Se comprobó la creación de cuentas, el inicio de sesión con credenciales válidas y el rechazo de credenciales incorrectas.	Correcto
Creación de jugadores	Se validó la creación de jugadores personalizados y la comprobación de atributos fuera de rango o campos obligatorios vacíos.	Correcto
Simulación de partidos	Se configuraron partidos con distintos jugadores, formatos y superficies, verificando la generación del marcador, el feed narrativo y el resumen final.	Correcto
Modo estratégico	Se comprobó la simulación punto a punto con selección de estrategias durante el partido.	Correcto
Modo Big Data	Se ejecutaron simulaciones masivas y se verificó la visualización de estadísticas agregadas.	Correcto
Gestión de torneos	Se crearon torneos, se simularon rondas y se comprobó el avance automático de ganadores en el cuadro.	Correcto
Historial y perfil	Se revisó la consulta del historial de partidos y la visualización de información asociada al usuario.	Correcto
Protección de rutas	Se intentó acceder a vistas privadas sin sesión activa, comprobando la redirección o bloqueo correspondiente.	Correcto
Textos y etiquetas	Se comprobó que los textos principales, etiquetas de formularios, mensajes y estadísticas se muestran correctamente en español.	Correcto

Tabla 8.1: Resumen de pruebas manuales realizadas sobre la aplicación

Además de estas pruebas, se revisó el comportamiento de los formularios con datos inválidos, como contraseñas incorrectas, usuarios duplicados o atributos numéricos fuera del rango permitido. En estos casos, la aplicación mostró mensajes de error adecuados y evitó que se almacenaran datos no válidos.

8.3. Pruebas automatizadas del motor y del sistema

Las pruebas automatizadas cubren principalmente el motor de simulación, ya que es el componente central y más complejo del proyecto. El objetivo de estas pruebas es comprobar que las reglas internas del simulador se mantienen coherentes y que una simulación completa produce siempre una salida válida.
Se implementaron dos conjuntos de pruebas:

- **Pruebas unitarias** (`tests/test_unit.py`).

  Estas pruebas comprueban componentes aislados del motor, como la función de recorte de valores, las propiedades normalizadas del modelo `Player`, los valores por defecto de `Config` y el comportamiento de las clases `TennisGame` y `TennisSet`. También se verifica que un juego produce un ganador válido y que, al fijar una semilla aleatoria, los resultados son reproducibles.

- **Pruebas de integración** (`tests/test_integration.py`).

  Estas pruebas ejercitan la función `run_match()` de extremo a extremo. Se comprueba que la respuesta contiene los campos esperados, que el ganador es siempre uno de los dos jugadores, que el número de sets ganados es coherente con el formato elegido y que la reproducibilidad por semilla funciona correctamente.

  También se incluye una prueba de cordura estadística, en la que un jugador con atributos claramente superiores debe ganar la mayoría de los enfrentamientos simulados. En concreto, esta prueba permite detectar errores graves en el comportamiento probabilístico del motor, comprobando que la diferencia de atributos se refleja de forma razonable en los resultados.

Todas las pruebas automatizadas se ejecutan correctamente mediante `pytest` y finalizan en un tiempo reducido. Esto permite repetirlas durante el desarrollo para comprobar que los cambios realizados no rompen el comportamiento principal del motor.

Capítulo 9

Conclusiones y trabajo futuro

9.1. Conclusiones generales

El desarrollo de ADAF Tennis Simulator ha permitido construir una aplicación web completa alrededor de un motor de simulación deportiva. A lo largo del proyecto se ha diseñado e implementado un sistema capaz de crear jugadores personalizados, simular partidos de tenis punto a punto, mostrar el desarrollo del encuentro en tiempo real, almacenar resultados y ofrecer estadísticas finales al usuario.
Uno de los aspectos más importantes del trabajo ha sido la integración de distintas partes dentro de una misma aplicación. El proyecto no se ha limitado a una interfaz web o a una base de datos, sino que ha combinado un backend con FastAPI, una base de datos PostgreSQL, plantillas Jinja2, módulos JavaScript para la interacción dinámica y un motor probabilístico desarrollado en Python. Esta combinación ha permitido aplicar conocimientos de varias áreas del grado, como desarrollo web, diseño de bases de datos, arquitectura de software, programación orientada a objetos y pruebas.
También se ha conseguido que el simulador no resuelva los partidos únicamente mediante un porcentaje global de victoria. En su lugar, el motor genera el encuentro punto a punto y, dentro de cada punto, tiene en cuenta fases como el saque, el resto y el peloteo. Gracias a esto, el resultado final se construye a partir de una secuencia de acciones y no aparece simplemente como un valor calculado de forma directa.
En conjunto, se considera que el proyecto cumple los objetivos planteados inicialmente. La aplicación permite al usuario experimentar con distintos perfiles de jugadores, visualizar partidos simulados y consultar estadísticas que ayudan a interpretar lo ocurrido.

9.2. Principales aportaciones del proyecto

La principal aportación del proyecto es el desarrollo de un simulador de tenis integrado en una aplicación web funcional. El usuario no solo puede ejecutar una simulación, sino también crear jugadores, configurar partidos, consultar historiales, disputar torneos y analizar resultados.
Otra aportación importante es el diseño del motor de simulación. Este motor uti-

liza atributos técnicos de los jugadores y variables dinámicas como la estamina, el *momentum* y el *clutch*. De esta forma, el rendimiento de un jugador no depende únicamente de sus valores iniciales, sino también del contexto del partido y de la evolución de los puntos.
También destaca la visualización del partido en tiempo real. El sistema reproduce el marcador, genera una narración punto a punto y permite al usuario seguir el desarrollo del encuentro de forma más cercana que si únicamente se mostrase el resultado final. Esta parte aporta valor a la experiencia de uso y hace que la simulación sea más comprensible.
Además, el proyecto incorpora funcionalidades complementarias como el modo estratégico, el modo Big Data y la gestión de torneos. Estas funciones permiten utilizar el simulador de distintas formas: como una experiencia interactiva, como una herramienta de comparación estadística o como un sistema para organizar cuadros eliminatorios.
Por último, se ha trabajado en una estructura modular que facilita la evolución del sistema. La separación entre backend, frontend, base de datos y motor de simulación permite modificar una parte del proyecto sin afectar directamente al resto, lo que resulta importante de cara a futuras ampliaciones.

9.3. Limitaciones actuales y trabajo futuro

Aunque el sistema cumple los objetivos principales planteados, existen algunas limitaciones que conviene tener en cuenta. Estas limitaciones no impiden el funcionamiento de la aplicación, pero sí marcan posibles líneas de evolución para versiones futuras. A continuación se recogen las más relevantes.

- **Calibración del motor de simulación.**

  El motor probabilístico se ha desarrollado con los conocimientos disponibles durante el proyecto y a partir de una interpretación razonada del tenis. Sin embargo, los pesos asignados a los atributos y a las distintas fórmulas no están calibrados con una base de datos profesional de partidos reales.

  Como trabajo futuro, sería interesante comparar las estadísticas generadas por el simulador con datos reales de partidos profesionales. Esto permitiría ajustar mejor los pesos del modelo, reducir la parte subjetiva del diseño y hacer que los resultados se aproximasen más al comportamiento observado en el tenis real. Además, una posible evolución sería orientar esta calibración hacia técnicas de *Machine Learning*, de forma que el sistema pudiera aprender o ajustar automáticamente parte de las fórmulas matemáticas a partir de grandes volúmenes de datos.

- **Influencia de la superficie de juego.**

  En la versión actual, la superficie puede seleccionarse al configurar el partido, pero no modifica de forma específica el comportamiento del motor. Es decir, el modelo no altera todavía la duración de los puntos, la probabilidad de error, la importancia del saque o la movilidad según se juegue en tierra batida, hierba o pista dura.

Una mejora futura sería incorporar un modelo de superficie que afectase a las probabilidades internas del motor. Por ejemplo, la tierra batida podría favorecer intercambios más largos y dar más importancia a la movilidad, mientras que la hierba podría aumentar el peso del saque y reducir la duración media de los puntos.

- **Operaciones de edición y eliminación.**

  La aplicación permite crear jugadores, consultar información, simular partidos y guardar resultados, pero algunas operaciones de mantenimiento no se han desarrollado en profundidad. Por ejemplo, en futuras versiones sería útil permitir editar o eliminar jugadores creados por el usuario, modificar información del perfil, borrar historiales antiguos o gestionar de forma más completa torneos ya creados.

  Estas operaciones no se consideraron prioritarias frente al desarrollo del motor de simulación y de las funcionalidades principales, pero mejorarían la experiencia de usuario y harían que la aplicación fuese más completa desde el punto de vista de gestión.

- **Ampliación del modo Big Data.**

  El modo Big Data permite ejecutar muchas simulaciones consecutivas de un mismo enfrentamiento y obtener estadísticas agregadas. No obstante, actualmente está centrado principalmente en la comparación entre dos jugadores bajo una configuración concreta.

  Como línea futura, este modo podría ampliarse para permitir más combinaciones de análisis, como comparar varios jugadores entre sí, generar matrices de enfrentamientos, crear rankings simulados, filtrar resultados por formato de partido o exportar los datos a formatos como CSV o PDF. Esto convertiría el modo Big Data en una herramienta de análisis más potente.

- **Profundidad del modo estratégico.**

  El modo estratégico permite que el usuario intervenga durante el partido tomando decisiones tácticas, pero en la versión actual el número de variables modificables es limitado. La estrategia influye en el desarrollo del punto, aunque todavía podría ofrecer más posibilidades.

  En futuras versiones se podrían añadir decisiones más concretas, como variar la agresividad del primer saque, jugar de forma más conservadora en puntos importantes, atacar el revés del rival, subir más a la red o gestionar la energía del jugador durante el partido. Esto haría que la participación del usuario tuviera más peso en el desarrollo táctico del encuentro.

- **Funcionalidades sociales.**

  Actualmente, cada usuario trabaja principalmente con sus propios jugadores, partidos e historiales. La aplicación no incorpora todavía un sistema social que permita interactuar con otros usuarios.

Una posible mejora sería añadir funcionalidades como agregar amigos, consultar perfiles de otros usuarios, ver sus jugadores creados, compartir jugadores o incluso utilizarlos en simulaciones propias. También se podrían plantear torneos con jugadores de distintas cuentas. Esta línea convertiría la aplicación en una plataforma más colaborativa y no solo en un simulador individual.

- **Accesibilidad.**

  Durante el desarrollo se ha priorizado la construcción del motor de simulación, la arquitectura del backend, la persistencia de datos y las funcionalidades principales de la aplicación. Por este motivo, la accesibilidad de la interfaz no se ha trabajado con el mismo nivel de profundidad.

  Como trabajo futuro, sería importante revisar la aplicación desde el punto de vista de la accesibilidad, mejorando aspectos como el contraste visual, la navegación mediante teclado, la claridad de los mensajes y la compatibilidad con lectores de pantalla. Esto permitiría que la aplicación fuera más cómoda y usable para un mayor número de usuarios.

- **Despliegue y escalabilidad.**

  El proyecto se ha desarrollado como una aplicación funcional, pero todavía no se ha abordado un despliegue completo en producción. En una versión futura se podría desplegar el sistema en una infraestructura en la nube, configurar un entorno de producción y optimizar aspectos como la gestión de sesiones interactivas o la ejecución de simulaciones masivas.

  Esta mejora sería especialmente importante si la aplicación llegase a utilizarse por varios usuarios simultáneamente o si el modo Big Data aumentase su carga de procesamiento.

Introduction

Motivation

The main motivation behind this project stems from our personal interest in tennis. We have both practiced and followed the sport since we were young, so when we were given the opportunity to develop a sports simulator as our Final Degree Project, it was clear to us that tennis was a suitable choice. It was a sport we knew well and one that offered enough technical, strategic, and probabilistic elements to make for an interesting simulation.
Choosing tennis allowed us to work with a scoring system unlike that of most other sports, built around points, games, sets, *tie-breaks*, and high-pressure situations. We were also drawn to the challenge of representing aspects inherent to the game, such as the importance of the serve, consistency during rallies, accumulated fatigue, and the influence of key moments. In tennis, a player can be superior overall and still lose critical points that completely change the course of a match.
From the beginning, we wanted the simulator to go beyond simply computing a general win probability or resolving the match directly. The goal was to build a detailed simulation in which the match would be generated point by point and, within each point, shot by shot. In this way, the final result would not appear as an isolated figure, but as the consequence of everything that had happened throughout the match.
We also wanted the project to combine web development with an original algorithmic component. This was not just about building forms and storing data in a database, but about developing a system capable of generating matches, replaying them visually, and extracting statistics from the simulation. This combination of web application, probabilistic engine, and interactive visualisation was one of the main reasons the project appealed to us.

Problem Statement

Once the idea of building a tennis simulator had been defined, the next step was to analyse what the system needed to provide. There are tools aimed at the statistical prediction of matches, but many of them focus on estimating general win probabilities from historical data. This approach can be useful for analysis, but it does not

allow the user to observe how a match unfolds or to experiment with custom-created players.
The problem addressed in this project is therefore to build a simulator that does not resolve the match through a single final probability, but instead generates it progressively. To achieve this, the system must simulate the development of the match point by point and, within each point, represent phases such as the serve, the return, and the rally. This approach ensures that the final result emerges from a sequence of actions, errors, winners, streaks, and pressure situations.
Furthermore, the simulator needed to be integrated into a complete web application. This meant that the user should be able to create players with custom attributes, configure matches, visualise the scoreboard in real time, review statistics, compete in tournaments, and make use of additional modes such as a strategic mode or a bulk simulation analysis tool.
The challenge of the project therefore extended beyond designing a probabilistic engine. It involved several areas of work: building the simulation model, organising the backend, handling data persistence, implementing user authentication, and developing a clear and interactive interface. The main difficulty lay in integrating all these components into a coherent system that would be usable by the end user.

Project Objectives

General Objective

To develop a complete web application that allows users to create custom players, simulate tennis matches and tournaments, visualise the match as it unfolds in real time, and analyse results through statistics.

Specific Objectives

- Implement a point-by-point simulation engine based on numerical attributes and dynamic variables, such as fatigue and mental state, that can coherently reproduce the development of a tennis match.
- Design a relational database capable of storing users, players, matches, statistics, and tournaments, keeping all information associated with each user.
- Develop an API using FastAPI that exposes the main functionalities of the simulator in an organised manner.
- Build a web interface that allows the user to create players, configure matches, follow the scoreboard updated point by point, and visualise the commentary for each point.
- Add advanced usage modes, including a strategic mode in which the user makes decisions during the match, a bulk analysis mode that simulates multiple matches to extract statistical trends, and an elimination tournament system with automatic round progression.

- Validate the system through manual and automated testing, verifying that the simulation engine behaves coherently and reproducibly, and that the main functionalities of the application respond correctly.

Project Scope

The project covers the design and implementation of a functional web application, including the simulation engine, the backend, the database, the frontend, and the authentication system. It also encompasses features such as the creation of custom players, match simulation, elimination tournaments, the strategic mode, bulk simulation analysis, and the retrieval of results.

Methodology and Work Plan

Development was carried out in an iterative and incremental manner. First, the simulation engine was implemented in isolation, with the aim of verifying that it could generate complete matches and produce coherent results. Afterwards, the API and the database were developed to handle users, players, matches, and tournaments.
Once the core backend had been built, the web interface was developed using Jinja2 templates and JavaScript. In the final phases, the advanced features were incorporated: the strategic mode, the Big Data mode, and the elimination tournament system. Finally, manual and automated tests were carried out to validate the behaviour of the system.
Throughout development, Git was used as the version control system, allowing changes to be tracked and the different parts of the project to be managed in an orderly manner.

Contribuciones Personales

Álvaro Adrada

Motor de simulación probabilístico. Lideró el diseño e implementación del motor de simulación punto a punto, que constituye el núcleo técnico del proyecto. Desarrolló los módulos `point.py`, `shots.py` y `scoring.py`, que gestionan respectivamente la lógica de cada punto, los golpes del intercambio y el sistema de puntuación jerárquico (puntos, juegos, sets y partido). Definió las fórmulas de combinación de atributos que determinan las probabilidades de ace, winner, error no forzado o falta en cada fase del punto. Para ello, investigó cómo modelar matemáticamente la interacción entre dos jugadores con atributos distintos, de forma que un jugador con mejor saque tuviera más probabilidades de ganar los puntos al servicio, pero sin que el resultado fuera determinista. Incorporó además las variables dinámicas del modelo: la fatiga acumulada a lo largo del partido, que penaliza progresivamente el rendimiento del jugador según su atributo físico, y el estado mental, que amplifica o reduce el rendimiento en función de las rachas y los momentos de presión.
Backend y API REST. Desarrolló la estructura principal del backend con FastAPI, incluyendo la organización en módulos por dominio (auth, players, matches, tournaments, simulator) y la configuración de los ocho routers de la aplicación. Implementó los endpoints de gestión de jugadores (creación, edición, listado y eliminación), el endpoint principal de simulación de partidos y la lógica de almacenamiento automático de resultados y estadísticas tras cada simulación. Se encargó de la configuración de SQLAlchemy como ORM, la definición de los modelos de base de datos como clases Python y el sistema de inyección de dependencias con `get_db()`, que garantiza que cada petición trabaja con su propia sesión de base de datos y que esta se cierra correctamente al finalizar.
Estadísticas de partido. Implementó el cálculo y almacenamiento de todas las estadísticas generadas durante la simulación. Por cada partido simulado, el sistema calcula y guarda: aces, dobles faltas, primeros saques realizados y entrados, puntos ganados con primer y segundo saque, winners, errores no forzados, break points convertidos y oportunidades, y total de puntos ganados. Diseñó la tabla `estadisticas_partido` para almacenar dos filas por partido, una por jugador, con todas sus restricciones de integridad (por ejemplo, no pueden existir más break points convertidos que oportunidades). También implementó el endpoint que devuelve el detalle estadístico completo de un partido concreto para su consulta en el historial.

Modo estratégico. Implementó la lógica del modo estratégico en el backend. Desarrolló los tres modificadores de estrategia (agresivo, neutro y defensivo) y definió cómo cada uno altera los parámetros del motor en ese punto concreto: el modo agresivo incrementa la probabilidad de winner pero también la de error no forzado, el defensivo reduce ambas priorizando la consistencia, y el neutro mantiene los valores base del jugador. La implementación requirió modificar el flujo del motor para que, en lugar de calcular el punto de forma autónoma, esperara la decisión del usuario antes de proceder, manteniendo el estado del partido entre peticiones.
Seguridad y autenticación. Implementó el sistema de autenticación con JWT mediante `python-jose` y el cifrado de contraseñas con `passlib` y bcrypt. Diseñó el flujo completo: en el registro, la contraseña se cifra antes de almacenarse en la base de datos y nunca se guarda en texto plano; en el login, se verifica el hash y se genera un token JWT firmado con una clave secreta configurable mediante variable de entorno; en cada petición protegida, el token se verifica y se extrae el usuario asociado. Configuró una expiración de 7 días para los tokens y estableció las políticas de borrado en cascada en la base de datos para garantizar que no queden datos huérfanos al eliminar un usuario o un jugador.
Sistema de torneos. Contribuyó al diseño de la tabla `torneo_partidos` y a la integración del motor de simulación con la lógica de rondas. Adaptó la función `run_match()` para que pudiera invocarse de forma encadenada durante la simulación automática de cada ronda, asegurando que los resultados se almacenaran correctamente y que los ganadores quedaran registrados para la ronda siguiente. También adaptó el motor para soportar la ejecución de miles de partidos de forma eficiente en el modo Big Data, optimizando el uso de memoria y evitando cuellos de botella en simulaciones masivas.

Diego Fernández

Sistema de torneos eliminatorios. Lideró el diseño e implementación del módulo de torneos. Desarrolló la lógica completa del cuadro eliminatorio: el algoritmo de generación automática de emparejamientos a partir del número de jugadores seleccionados (4, 8 o 16), el sistema de avance de ganadores entre rondas y la detección del campeón una vez completada la ronda final. Implementó los endpoints de la API para crear torneos, simular rondas completas y consultar el estado actualizado del cuadro en cualquier momento. Diseñó la representación interna del cuadro de forma que los huecos de rondas futuras quedaran reservados desde el momento de la creación del torneo, rellenándose automáticamente con los ganadores a medida que se simulaban las rondas. También desarrolló la vista del bracket en el frontend, con la visualización gráfica del cuadro completo y su actualización dinámica tras cada ronda simulada.
Interfaz web y experiencia de usuario. Se encargó del desarrollo general de las plantillas HTML con Jinja2 y del sistema de estilos con Tailwind CSS. Diseñó y maquetó todas las páginas principales de la aplicación: la página de inicio con las secciones de presentación, el menú de usuario con acceso a todas las funcionalidades, la pantalla de creación de partido con sus controles de configuración, la vista de historial de resultados y la pantalla de perfil de usuario. Aseguro la consistencia del diseño de las páginas y de la experiencia en dispositivos de distinto tamaño, aplicando clases responsivas de Tailwind. También implementó el sistema de herencia de plantillas con `base.html` como plantilla raíz, los layouts de autenticación y los componentes parciales reutilizables (cabecera, pie de página, sidebar).
Pantalla de simulación en tiempo real. Implementó la pantalla de simulación como una experiencia interactiva en el navegador. Desarrolló el sistema de reproducción punto a punto con control de velocidad, el marcador visual actualizado en tiempo real, la narración textual de cada punto y las transiciones de estado entre juegos y sets. Organizó el código JavaScript del frontend en módulos ES6 independientes: `app.js` como orquestador, `state.js` para el estado de la simulación, `controls.js` para los controles de reproducción, `scoreboard.js` para el marcador y `liveFeed.js` para la narración.
Modo Big Data. Implementó el modo de análisis masivo, incluyendo el sistema de streaming con Server-Sent Events que envía el progreso al navegador en tiempo real mientras el servidor ejecuta los partidos en segundo plano. Desarrolló todas las gráficas de resultados con Chart.js: la distribución de marcadores, las medias estadísticas comparadas por jugador, la distribución de la duración de los puntos y la progresión de la probabilidad de victoria a medida que se acumulan simulaciones.
Modo estratégico. Desarrolló la interfaz del modo estratégico en el frontend. Implementó los controles de selección de táctica que aparecen antes de cada punto, la visualización del estado físico y mental de los jugadores (estamina y momentum) actualizada tras cada jugada, y la narración del resultado de cada punto con información sobre la estrategia aplicada. Coordinó con Álvaro la integración entre los controles del frontend y la lógica del backend para que la elección del usuario se transmitiera correctamente al motor antes de calcular cada punto.
Seguridad y autenticación — lado cliente. Se encargó de la parte cliente del

sistema de autenticación: los formularios de registro e inicio de sesión con validación de datos en el navegador, el almacenamiento seguro del token JWT en el cliente, la lógica de protección de rutas privadas que redirige al login cuando el token no está presente o ha expirado, y la gestión del estado de sesión en las distintas páginas de la aplicación.

Estadísticas de partido — visualización. Participó en la definición de qué estadísticas eran más relevantes para el usuario y desarrolló la vista comparativa de estadísticas al final de cada partido. Implementó la tabla de datos con los valores de ambos jugadores y las gráficas generadas con Chart.js: la comparativa de aces y dobles faltas, los porcentajes de primer saque, los winners y errores no forzados, y los break points.

Motor de simulación probabilístico. Colaboró en la fase de validación del motor, definiendo los perfiles de los jugadores de prueba y comprobando que los resultados generados eran coherentes con el comportamiento esperado de un partido real. Participó en la detección y corrección de casos extremos en los que el motor producía resultados estadísticamente anómalos.

Pruebas y validación. Lideró la escritura de las pruebas automatizadas del módulo de simulación. Desarrolló las pruebas unitarias que verifican el comportamiento de componentes individuales como la función `clip()`, los valores por defecto de `Config` y la lógica de `TennisGame`, y las pruebas de integración que validan partidos completos ejecutados con `run_match()`: estructura del resultado, validez del ganador, formato de los marcadores, reproducibilidad con semilla fija y comportamiento con distintos formatos (al mejor de 1, 3 o 5 sets).

Trabajo conjunto

Las decisiones de diseño más importantes del proyecto fueron tomadas conjuntamente tras varias iteraciones y discusiones entre ambos. La estructura de la base de datos, la organización de la API en módulos, el modelo de jugador con sus nueve atributos y la arquitectura general del sistema son el resultado de ese proceso compartido.

La integración del módulo de torneos con el motor de simulación requirió una coordinación estrecha: uno adaptó el motor para que pudiera invocarse de forma encadenada en cada ronda, mientras el otro ajustó la lógica del cuadro para consumir correctamente los resultados. De forma similar, la pantalla de simulación en tiempo real fue un trabajo coordinado, con uno definiendo la estructura del timeline de puntos que genera el motor y el otro implementando su representación visual en el navegador.

El diseño de la base de datos también fue revisado y refinado entre los dos, especialmente en lo relativo a las políticas de borrado en cascada y a la tabla `torneo_partidos`, cuya estructura requirió varias iteraciones para cubrir correctamente todos los casos del sistema de torneos.

Ambos participaron en la depuración de errores a lo largo del desarrollo, revisaron mutuamente el código de las partes desarrolladas por cada uno y contribuyeron a la redacción de esta memoria.

Bibliografía

[1] Sebastián Ramírez. FastAPI documentation. Available in: https://fastapi.tiangolo.com

[2] Encode. Uvicorn documentation. Available in: https://www.uvicorn.org

[3] SQLAlchemy Authors. SQLAlchemy documentation. Available in: https://docs.sqlalchemy.org

[4] The PostgreSQL Global Development Group. PostgreSQL documentation. Available in: https://www.postgresql.org/docs

[5] Daniele Varrazzo. Psycopg2 documentation. Available in: https://www.psycopg.org/docs

[6] Pydantic Authors. Pydantic documentation. Available in: https://docs.pydantic.dev

[7] Python Software Foundation. Python. Available in: https://www.python.org

[8] NumPy Developers. NumPy documentation. Available in: https://numpy.org/doc

[9] pytest Developers. pytest documentation. Available in: https://docs.pytest.org

[10] Passlib Authors. Passlib documentation. Available in: https://passlib.readthedocs.io

[11] Michael Hart. python-jose documentation. Available in: https://python-jose.readthedocs.io

[12] Pallets Projects. Jinja2 documentation. Available in: https://jinja.palletsprojects.com

[13] Tailwind Labs. Tailwind CSS documentation. Available in: https://tailwindcss.com/docs

[14] Chart.js Contributors. Chart.js documentation. Available in: https://www.chartjs.org/docs

[15] Fonticons Inc. Font Awesome documentation. Available in: `https://fontawesome.com/docs`

[16] Mozilla Contributors. JavaScript. Available in: `https://developer.mozilla.org/en-US/docs/Web/JavaScript`

[17] Jeff Sackmann. Tennis Abstract. Available in: `https://www.tennisabstract.com`

[18] Mark Walker y John Wooders. Minimax Play at Wimbledon. *American Economic Review*, 91(5):1521–1538, 2001.

[19] Paul K. Newton y Joseph B. Keller. Probability of Winning at Tennis I. Theory and Data. *Studies in Applied Mathematics*, 114(3):241–269, 2005.

Apéndice A

Repositorio del proyecto

El código fuente de ADAF Tennis Simulator se encuentra alojado en un repositorio de GitHub. En dicho repositorio se incluye el código del backend, el frontend, el motor de simulación, los scripts de base de datos, las pruebas automatizadas y los recursos estáticos necesarios para ejecutar la aplicación.

`https://github.com/AlvaroAdrada26/ADAF_Tennis_Simulator`

Además, el repositorio contiene un archivo `README.md` con información general del proyecto e instrucciones para su instalación y ejecución en un entorno local.
Dado que el repositorio es privado, el acceso al código fuente no es público. Para poder visualizarlo o clonar el proyecto es necesario disponer de una cuenta de GitHub y solicitar acceso previamente a los autores del trabajo.

Apéndice B

Ejecución de la aplicación en local

Este apéndice recoge los pasos necesarios para ejecutar ADAF Tennis Simulator en un entorno local de desarrollo. Estas instrucciones complementan la información disponible en el archivo `README.md` del repositorio del proyecto.

B.1. Requisitos previos

Antes de ejecutar la aplicación, es necesario disponer de los siguientes elementos instalados en el equipo:

- Python 3.10 o superior.
- PostgreSQL instalado y en ejecución.
- Git, en caso de querer clonar el repositorio desde GitHub.
- Un terminal o consola de comandos.

Además, se recomienda utilizar un entorno virtual de Python para instalar las dependencias del proyecto sin afectar a otras instalaciones del sistema.

B.2. Obtención del código fuente

El primer paso consiste en obtener el código fuente del proyecto desde el repositorio de GitHub. Una vez concedido el acceso al repositorio privado, puede clonarse mediante el siguiente comando:

```
git clone https://github.com/AlvaroAdrada26/ADAF_Tennis_Simulator.git
cd ADAF_Tennis_Simulator
```

En caso de disponer ya del proyecto descargado, únicamente es necesario situarse en el directorio raíz del mismo mediante la terminal.

B.3. Preparación de la base de datos

La aplicación utiliza PostgreSQL como sistema gestor de base de datos. Por defecto, el código está configurado para conectarse a una base de datos local con los siguientes valores:

- Usuario: `postgres`.
- Contraseña: `root`.
- Base de datos: `adaf`.
- Puerto: `5432`.

Por tanto, para ejecutar la aplicación sin modificar la configuración, debe existir una base de datos llamada `adaf` en PostgreSQL. Puede crearse desde `psql` o desde una herramienta gráfica como pgAdmin:

```
CREATE DATABASE adaf;
```

Posteriormente, desde la raíz del proyecto, se debe ejecutar el script de inicialización incluido en la carpeta `database`. Este script crea las tablas necesarias e inserta los datos iniciales del sistema:

```
psql -U postgres -d adaf -f database/newDB.sql
```

En este comando, `postgres` representa el usuario utilizado para conectarse a PostgreSQL. Si se utiliza otro usuario, debe sustituirse por el correspondiente.

B.4. Configuración de variables de entorno

El proyecto no carga automáticamente variables desde un archivo `.env`. En su lugar, lee directamente las variables definidas en el entorno del sistema operativo mediante `os.getenv()`.
La variable principal para la conexión a la base de datos es `DATABASE_URL`. Si no se define, la aplicación utiliza por defecto la siguiente cadena de conexión:

```
postgresql://postgres:root@localhost:5432/adaf
```

En caso de que la instalación local de PostgreSQL utilice otro usuario, contraseña, puerto o nombre de base de datos, debe definirse la variable `DATABASE_URL` antes de iniciar el servidor.
En Windows PowerShell:

```
$env:DATABASE_URL="postgresql://usuario:contrasena@localhost:5432/adaf"
```

En Windows CMD:

```
set DATABASE_URL=postgresql://usuario:contrasena@localhost:5432/adaf
```

En macOS o Linux:

```
export DATABASE_URL="postgresql://usuario:contrasena@localhost:5432/adaf"
```

El sistema también permite configurar la variable `JWT_SECRET_KEY`, utilizada para firmar los tokens JWT, y la variable `ACCESS_TOKEN_EXPIRE_MINUTES`, utilizada para definir la duración de los tokens. Si no se definen, el código utiliza valores por defecto adecuados para un entorno local de desarrollo.

B.5. Creación del entorno virtual

Se recomienda crear un entorno virtual para aislar las dependencias del proyecto. En Windows puede hacerse con los siguientes comandos:

```
python -m venv .venv
.venv\Scripts\activate
```

En macOS o Linux, los comandos equivalentes son:

```
python3 -m venv .venv
source .venv/bin/activate
```

Una vez activado el entorno virtual, el nombre del entorno suele aparecer al inicio de la línea de comandos.

B.6. Instalación de dependencias

Con el entorno virtual activado, se instalan las dependencias necesarias mediante el archivo `requirements.txt`:

```
pip install -r requirements.txt
```

Este comando instala las librerías utilizadas por la aplicación, incluyendo FastAPI, SQLAlchemy, Uvicorn, Python-JOSE, Passlib, bcrypt, NumPy y las dependencias necesarias para la ejecución de pruebas.

B.7. Ejecución del servidor

Una vez preparada la base de datos e instaladas las dependencias, la aplicación puede iniciarse mediante Uvicorn desde la raíz del proyecto:

```
uvicorn backend.app.main:app --reload
```

La opción `-reload` permite que el servidor se reinicie automáticamente cuando se detectan cambios en el código, por lo que resulta útil durante el desarrollo.
Si la ejecución se realiza correctamente, el servidor quedará disponible en la dirección:

```
http://localhost:8000
```

Desde esa URL se puede acceder a la interfaz web de ADAF Tennis Simulator.

B.8. Documentación interactiva de la API

FastAPI genera automáticamente una documentación interactiva de los endpoints disponibles. Una vez iniciado el servidor, puede consultarse desde:

`http://localhost:8000/docs`

Esta vista permite revisar los endpoints definidos en la aplicación y probar peticiones directamente desde el navegador.

B.9. Ejecución de pruebas

El proyecto incluye pruebas automatizadas en la carpeta `tests/`. Para ejecutarlas, debe mantenerse activado el entorno virtual y lanzar el siguiente comando desde la raíz del proyecto:

```
pytest
```

Estas pruebas permiten comprobar el comportamiento del motor de simulación, la reproducibilidad mediante semilla y la coherencia de las simulaciones completas.