7-3-2025

documentación Proyecto

Jorge morgado, Ángel Sánchez, Antoine López

Contenido

Introducción

El proyecto **Sistema de Gestión para Equipos de Fútbol** tiene como objetivo proporcionar una plataforma integral para la gestión administrativa y deportiva de equipos de fútbol. Este sistema permite gestionar jugadores, entrenadores, calendarios de partidos, estadísticas y más, brindando una herramienta eficiente y moderna para clubes de cualquier nivel.

Objetivos del Proyecto

1. **Digitalizar y centralizar** la información del equipo.
2. Facilitar la **gestión de recursos humanos** del club (jugadores y cuerpo técnico).
3. Proveer herramientas para el **análisis de desempeño deportivo** mediante estadísticas.
4. Diseñar una interfaz intuitiva para una facilidad de uso.
5. Brindar funcionalidades adicionales como la programación de partidos, registro de resultados y visualización de datos.

Equipo de Desarrollo

El proyecto **Sistema de Gestión para Equipos de Fútbol** fue desarrollado por un equipo multidisciplinario con roles y tareas claramente definidos, asegurando una colaboración efectiva y un producto final de alta calidad. A continuación, se presenta el equipo y su contribución principal:

1. Ángel Sánchez (Líder del Proyecto\Desarrolladore)

- Coordinación general del proyecto.
- Definición de requerimientos funcionales y técnicos.
- Supervisión del control de versiones y revisiones de código.

2. Jorge Morgado (Desarrollador)

- Desarrollo de Storages
- Creacion de documentos
- Implementación de autenticación y validación

3. Antoine Lopez (Desarrollador)

- Desarrollo de Repositorio CRUD
- Implementación Storage Json
- Implementación configuración

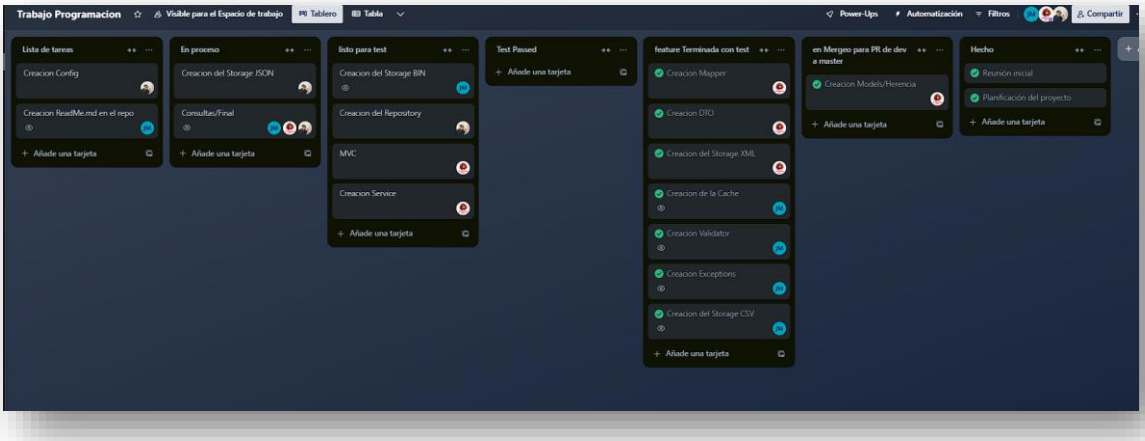
Gestión del Trabajo con Trello

Para la organización y gestión del proyecto. Este tablero permitió al equipo dividir las tareas en etapas claras, identificar prioridades y gestionar el progreso de manera eficiente.

Estructura del Tablero Trello

El tablero se organizó en diferentes listas, reflejando el flujo de trabajo del equipo:

1. **Lista de tareas:**
    - Contiene tareas identificadas, pero aún no iniciadas.
2. **En proceso:**
    - Tareas que están siendo trabajadas activamente.
3. **listo para test:**
    - Incluye tareas terminadas pero que requieren validación o pruebas.
4. **Test Passed:**
    - Registro de todas las Features con los test pasados
5. **feature Terminada con test:**
    - Tareas que tienen los test terminados y aprobados por el líder del proyecto
6. **en Mergeo para PR de dev a master:**
    - Tareas que están listas para mandarlas a producción



Beneficios del Uso de Trello

- **Colaboración Transparente:** Todo el equipo podía visualizar el estado y asignación de cada tarea en tiempo real.
- **Organización Clara:** Las listas facilitaron la transición fluida de tareas entre las etapas del proyecto.
- **Documentación de Avances:** El historial de movimientos en el tablero permitió realizar un seguimiento detallado del progreso.

Ejemplo de Tarjetas

Cada tarjeta incluía descripciones detalladas, asignación de responsables, fechas límite y comentarios para un seguimiento eficiente. Además, se emplearon etiquetas para clasificar las tareas por prioridad y categoría (backend, frontend, pruebas, etc.).

Análisis de Riesgos

El proyecto **Sistema de Gestión para Equipos de Fútbol** también incluyó un análisis de riesgos que ayudó al equipo a anticiparse a posibles problemas y mitigarlos proactivamente.

Riesgo	**Probabilidad**	**Impacto**	**Plan de Mitigación**
Retrasos en el desarrollo de módulos	Media	Alto	Sincronización semanal y pruebas intermedias para validar avances.
Fallos en la integración	Media	Medio	Realizar pruebas unitarias frecuentes y
Problemas con la configuración	Baja	Alto	Establecer un entorno de desarrollo unificado y documentación detallada.
Desacuerdos en diseño	Media	Medio	Reuniones con los desarrolladores para obtener una lluvia de ideas y acordar el diseño final.

Flujo de Trabajo con GitFlow

El proyecto **Sistema de Gestión para Equipos de Fútbol** adoptó el modelo de flujo de trabajo **GitFlow**, ampliamente utilizado para la gestión de ramas en proyectos colaborativos. Este flujo proporciona una estructura clara para el desarrollo y despliegue del software, organizando el trabajo en ramas específicas:

1. **Rama Principal (main):**
    - Contiene el código en estado estable y listo para producción.
    - Solo se actualiza tras completar una versión o mediante fusiones aprobadas.
2. **Rama de Desarrollo (dev):**
    - Actúa como la base para las nuevas funcionalidades y ajustes.
    - Recibe las integraciones desde las ramas de características y correcciones.
3. **Ramas de Funcionalidades (feature):**
    - Cada nueva funcionalidad o tarea específica se desarrolla en una rama derivada de feature.

Especificación de Requisitos

La especificación de requisitos define los aspectos esenciales del proyecto para asegurar que cumpla con las expectativas y necesidades del equipo de desarrollo y los usuarios finales. Se clasifican en tres categorías principales: funcionales, no funcionales y de información.

Requisitos Funcionales

Los requisitos funcionales describen las funcionalidades específicas que el sistema debe implementar. Estos incluyen:

1. **Gestión de Usuarios:**
    - Registrar, modificar y eliminar usuarios con diferentes roles (entrenador, jugador).
        - Asignación de permisos específicos según el rol.
    - Asignar jugadores y entrenadores a un equipo específico.
2. **Gestión de Partidos:**
    - Programar partidos, incluyendo fecha, hora y lugar.
    - Registrar resultados de los partidos jugados.
    - Generar informes automáticos basados en el desempeño.
3. **Estadísticas y Análisis:**
    - Generar y visualizar estadísticas detalladas de los jugadores.

Requisitos No Funcionales

Los requisitos no funcionales especifican las características cualitativas del sistema. Entre ellos se incluyen:

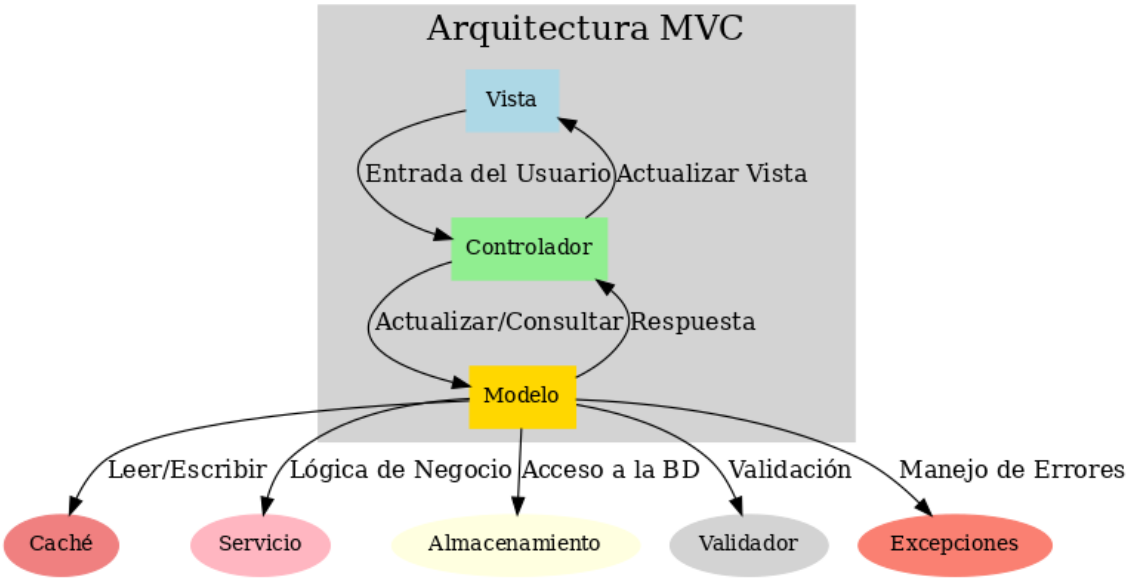
1. **Rendimiento:**
    - El sistema debe ser capaz de gestionar hasta 1,000 usuarios simultáneos sin degradación del rendimiento.
2. **Escalabilidad:**
    - La arquitectura debe permitir el crecimiento del sistema para soportar más funcionalidades o usuarios en el futuro.
3. **Disponibilidad:**
    - El sistema debe garantizar un uptime del 99.5% en entornos de producción.

4. **Usabilidad:**
    - Interfaz intuitiva y fácil de usar para usuarios no técnicos.

Requisitos de Información

Los requisitos de información detallan los datos que el sistema debe manejar y cómo deben organizarse:

1. **Información del Usuario:**
    - Datos personales: nombre, correo electrónico, rol.
    - Credenciales de acceso: nombre de usuario y contraseña encriptada.
2. **Información de Estadísticas:**
    - Rendimiento individual de jugadores: goles, asistencias, minutos jugados.



Sistema de Gestión de Personal Deportivo: Herramientas y Tecnologías

El desarrollo del **Sistema de Gestión de Personal Deportivo** se basa en un conjunto robusto de herramientas, lenguajes y tecnologías que aseguran un diseño eficiente, escalable y mantenible. A continuación, se detallan las principales herramientas empleadas:

1. Lenguajes de Programación y Frameworks

- **Kotlin:**
  Lenguaje principal para la implementación del sistema, aprovechando su sintaxis concisa, características modernas y compatibilidad con el ecosistema de Java.
- **Java:**
  Lenguaje complementario para integraciones y soporte en módulos específicos.
- **Gradle:**
  Herramienta de construcción utilizada para automatizar tareas de compilación, pruebas y gestión de dependencias.

2. Bibliotecas y Dependencias

- **JUnit 5:**
  Framework de pruebas utilizado para garantizar la calidad del código mediante pruebas unitarias y de integración.
- **Lighthouse Logger:**
  Sistema avanzado de logging para monitorear y depurar eventos dentro de la aplicación, facilitando el diagnóstico de problemas.
- **Json serialization y XML serialization:**
  Biblioteca para el procesamiento de JSON y XML y, utilizada en la serialización y deserialización de datos.
- **Mockk:**
  Framework especializado en pruebas de mocking, empleado para simular dependencias en escenarios de prueba.

3. Herramientas de Desarrollo

- **IntelliJ IDEA 2024.3:**
  IDE principal para el desarrollo del sistema, seleccionado por su integración avanzada con Kotlin, herramientas de depuración y soporte para Gradle.
- **Git:**
  Sistema de control de versiones utilizado para gestionar cambios en el código fuente de forma eficiente.
- **GitHub:**
  Repositorio remoto empleado para la colaboración entre los desarrolladores, almacenamiento del código y revisión mediante pull requests.

Beneficios del Uso de Estas Tecnologías

El conjunto de herramientas y tecnologías seleccionadas permite:

1. **Productividad:** Acelerar el desarrollo gracias a entornos integrados y automatización.
2. **Calidad:** Garantizar estándares altos mediante pruebas y logging robusto.
3. **Colaboración:** Facilitar el trabajo en equipo con un flujo eficiente de control de versiones.
4. **Escalabilidad:** Preparar la base tecnológica para integrar nuevas funcionalidades en el futuro.

Justificación de Elecciones de Software en el Proyecto "Gestión de Equipo de Fútbol"

El desarrollo del proyecto "Gestión de Equipo de Fútbol" ha requerido una selección meticulosa de tecnologías y herramientas de software para asegurar su eficiencia, escalabilidad y facilidad de mantenimiento. A continuación, se detallan las decisiones tomadas, las opciones consideradas y los pros y contras de cada elección.

Lenguaje de Programación: Kotlin

Justificación: Kotlin fue seleccionado como el lenguaje principal para este proyecto debido a sus múltiples ventajas en el desarrollo de aplicaciones modernas. Su sintaxis concisa y expresiva, junto con su interoperabilidad con Java, lo convierten en una opción ideal para proyectos que buscan eficiencia y claridad en el código.

Opciones Evaluadas:

- **Java:** Aunque es ampliamente utilizado y cuenta con una vasta comunidad, su sintaxis más verbosa puede conducir a un código más extenso y complejo de mantener.
- **Swift:** Excelente para desarrollo en plataformas Apple, pero no es adecuado para aplicaciones multiplataforma que incluyan Android.
- **JavaScript con frameworks como React Native:** Facilita el desarrollo multiplataforma, pero puede presentar limitaciones en el rendimiento y acceso a funcionalidades nativas avanzadas.

Pros de Kotlin:

- **Sintaxis Concisa:** Permite reducir la cantidad de código necesario, lo que facilita su lectura y mantenimiento.
- **Seguridad de Nulos:** Incluye mecanismos para manejar nulos de forma segura, reduciendo errores comunes en tiempo de ejecución.
- **Interoperabilidad con Java:** Facilita la integración y migración de proyectos existentes de Java a Kotlin, aprovechando bibliotecas y frameworks ya establecidos.
- **Soporte para Programación Asíncrona:** Las corrutinas de Kotlin optimizan la programación asíncrona, simplificando tareas como llamadas de red y acceso a bases de datos.

Contras de Kotlin:

- **Curva de Aprendizaje Inicial:** Aunque es similar a Java, puede requerir tiempo para adaptarse a sus características y paradigmas propios.

- **Ecosistema en Desarrollo:** Algunas bibliotecas y herramientas pueden no estar tan maduras o ampliamente adoptadas como las disponibles para Java.

Entorno de Desarrollo Integrado (IDE): IntelliJ IDEA

Justificación: IntelliJ IDEA fue elegido como el IDE principal debido a su excelente soporte para Kotlin y sus potentes herramientas de desarrollo.

Opciones Evaluadas:

- **Android Studio:** Basado en IntelliJ, ofrece herramientas específicas para el desarrollo de aplicaciones Android, pero puede ser más pesado en términos de rendimiento.
- **Eclipse con plugins:** Aunque es una opción popular, su soporte para Kotlin no es tan robusto como el de IntelliJ.

Pros de IntelliJ IDEA:

- **Soporte Nativo para Kotlin:** Ofrece integración completa con Kotlin, facilitando el desarrollo y depuración.
- **Herramientas Avanzadas:** Incluye características como refactorización inteligente, análisis estático de código y soporte para sistemas de control de versiones.
- **Personalización:** Permite adaptar el entorno de desarrollo según las necesidades del proyecto y las preferencias del desarrollador.

Contras de IntelliJ IDEA:

- **Consumo de Recursos:** Puede ser exigente en términos de memoria y CPU, especialmente en proyectos grandes.
- **Costo:** La versión Ultimate es de pago, aunque la versión Community es gratuita y suficientemente completa para muchos proyectos.

Sistema de Control de Versiones: Git

Justificación: Git se seleccionó para gestionar el control de versiones debido a su eficiencia, flexibilidad y amplia adopción en la industria del software.

Opciones Evaluadas:

- **Subversion (SVN):** Aunque es robusto, su modelo centralizado puede ser menos flexible que el enfoque distribuido de Git.
- **Mercurial:** Similar a Git en funcionalidad, pero con menor adopción y comunidad de soporte.

Pros de Git:

- **Control de Versiones Distribuido:** Cada desarrollador tiene una copia completa del historial del proyecto, lo que facilita el trabajo offline y la recuperación ante fallos.
- **Ramas y Fusiones Eficientes:** Permite gestionar múltiples líneas de desarrollo de manera ágil y efectiva.
- **Amplia Comunidad y Soporte:** Abundancia de recursos, herramientas y servicios integrados, como GitHub y GitLab.

Contras de Git:

- **Curva de Aprendizaje:** Su flexibilidad puede ser abrumadora para principiantes, requiriendo tiempo para dominar comandos y flujos de trabajo avanzados.
- **Complejidad en Proyectos Grandes:** En repositorios muy extensos, las operaciones pueden volverse lentas si no se gestionan adecuadamente.

Gestor de Dependencias: Gradle

Justificación: Gradle fue elegido para la automatización de la construcción del proyecto debido a su flexibilidad y potente integración con Kotlin.

Opciones Evaluadas:

- **Maven:** Popular y con una estructura convencional, pero menos flexible que Gradle y con una sintaxis más verbosa.
- **Ant:** Más antiguo y menos intuitivo, requiere más configuración manual y carece de convenciones modernas.

Pros de Gradle:

- **Flexibilidad:** Permite personalizar el proceso de construcción según las necesidades específicas del proyecto.
- **Integración con Kotlin DSL:** Facilita la escritura de scripts de construcción en Kotlin, mejorando la coherencia del código.
- **Rendimiento:** Ofrece tiempos de construcción rápidos y soporte para compilaciones incrementales.

Contras de Gradle:

- **Curva de Aprendizaje:** Su flexibilidad puede llevar a una mayor complejidad, requiriendo tiempo para dominar su configuración avanzada.
- **Documentación Dispersa:** Aunque abundante, la información puede estar fragmentada, dificultando la búsqueda de soluciones específicas.

Framework para Pruebas: JUnit 5

Justificación: JUnit 5 se seleccionó para la realización de pruebas unitarias debido a su robustez, flexibilidad y amplia adopción en la comunidad de desarrollo.

Opciones Evaluadas:

- **TestNG:** Ofrece funcionalidades similares, pero JUnit tiene una adopción más amplia y mejor integración con otras herramientas.
- **KotlinTest (ahora Kotest):** Específico para Kotlin, pero JUnit proporciona una mayor compatibilidad y soporte en el ecosistema Java/Kotlin.

Pros de JUnit 5 (continuación):

- **Integración:** Compatible con Gradle y otros sistemas de construcción, lo que facilita la ejecución de pruebas automatizadas dentro del flujo de desarrollo continuo.
- **Características Modernas:** Soporte para pruebas dinámicas, anotaciones avanzadas y una estructura más limpia en comparación con versiones anteriores.

Contras de JUnit 5:

- **Curva de Aprendizaje para Extensiones:** Aunque su modularidad es una ventaja, puede ser necesario invertir tiempo en aprender a configurar y utilizar módulos específicos.
- **Compatibilidad Limitada con Versiones Anteriores:** Puede requerir actualizaciones si se utiliza junto con pruebas escritas en versiones anteriores de JUnit.

Principios SOLID en el Proyecto de Gestión de Personal Deportivo

1. Single Responsibility Principle (SRP)

Implementación:

- Separación clara de responsabilidades en clases específicas

- Cada clase tiene una única razón para cambiar

```
// Controller: Solo maneja la lógica de interacción
class Controller(private val service: PersonalService) {
   fun crearJugador()
   fun actualizarMiembro()
   fun eliminarMiembro()
}

// Service: Gestiona la lógica de negocio
class PersonalService(private val repository: PersonalRepository) {
   fun save(personal: Personal)
   fun update(id: Int, personal: Personal)
   fun delete(id: Int)
}
```

2. Open/Closed Principle (OCP)

Implementación:

- Uso de clases abstractas y herencia

- Extensión sin modificación del código existente

```
// Clase base abstracta
abstract class Personal {
   abstract val id: Int
   abstract val nombre: String
   // Propiedades comunes
}

// Extensiones sin modificar la clase base
class Jugador(
   override val id: Int,
   override val nombre: String,
   val posicion: Posicion
) : Personal()
class Entrenador(
   override val id: Int,
   override val nombre: String,
   val especializacion: Especializacion
) : Personal()
```

3. **Liskov Substitution Principle (LSP)**

Implementación:

- Subtipos completamente sustituibles

- Comportamiento coherente en la jerarquía

```
class PersonalService {
   fun save(personal: Personal) {
      // Funciona con cualquier subtipo de Personal
      repository.save(personal)
   }

   fun update(id: Int, personal: Personal) {
      // Actualiza cualquier tipo de Personal
      repository.update(id, personal)
   }
}
```

4. **Interface Segregation Principle (ISP)**

Implementación:

- Interfaces específicas y cohesivas

- No forzar implementaciones innecesarias

```
interface PersonalStorage {

   fun readFromFile(file: File, fileFormat: FileFormat): List<Personal>

   fun writeToFile(file: File, fileFormat: FileFormat, personalList: List<Personal>)
}
interface PersonalStorageFile {

   fun readFromFile(file: File): List<Personal>

   fun writeToFile(file: File, personalList: List<Personal>)
}ç
```

5. **Dependency Inversion Principle (DIP)**

Implementación:

- Inyección de dependencias

- Abstracciones de alto nivel

```
// Interfaz de alto nivel
interface PersonalRepository {
   fun save(personal: Personal)
   fun findById(id: Int): Personal?
}
// Clase que depende de abstracción
class PersonalService(
   private val repository: PersonalRepository // Inyección de dependencia
) {
   fun save(personal: Personal) {
      repository.save(personal)
   }
}
```

Beneficios Obtenidos

1. Mantenibilidad
   - Código más limpio y organizado
   - Facilidad para realizar cambios
2. Extensibilidad
   - Nuevas funcionalidades sin modificar código existente
   - Reutilización de componentes
3. Testabilidad
   - Facilidad para escribir pruebas unitarias
   - Mejor cobertura de código

4. Escalabilidad

- Sistema modular y flexible

- Fácil integración de nuevas características

Patrones de Diseño Implementados en el Proyecto

El proyecto ha sido desarrollado utilizando patrones de diseño reconocidos que facilitan la organización, mantenimiento y escalabilidad del código. Entre ellos, destacan el patrón Modelo-Vista-Controlador (MVC) y la arquitectura por capas. A continuación, se detallan estos patrones y su aplicación en el proyecto.

Patrón Modelo-Vista-Controlador (MVC)

Descripción: El patrón MVC es una arquitectura de software que separa una aplicación en tres componentes principales:

- **Modelo:** Gestiona los datos y la lógica de negocio.
- **Vista:** Encargada de la presentación de la información al usuario.
- **Controlador:** Maneja la interacción del usuario, procesa la entrada y actualiza el modelo y la vista en consecuencia.

Esta separación permite una gestión más modular y facilita el mantenimiento y la escalabilidad de la aplicación.

Aplicación en el Proyecto: En el proyecto "Gestión de Equipo de Fútbol", el patrón MVC se implementa de la siguiente manera:

- **Modelo:** Representado por las clases que gestionan la lógica de negocio y el acceso a datos.
- **Vista:** Constituida por las interfaces de usuario que presentan la información.
- **Controlador:** Compuesto por las clases que manejan las interacciones del usuario y actualizan el modelo y la vista en consecuencia.

Arquitectura por Capas

Descripción: La arquitectura por capas es un patrón de diseño que organiza una aplicación en capas horizontales, cada una con responsabilidades específicas. Comúnmente, estas capas incluyen:

- **Capa de Presentación:** Gestiona la interfaz de usuario y la experiencia del usuario.
- **Capa de Aplicación:** Contiene la lógica de negocio y las reglas de la aplicación.
- **Capa de Datos:** Maneja el acceso y la gestión de los datos.

Esta estructura facilita la separación de responsabilidades y mejora la mantenibilidad del código.

Aplicación en el Proyecto: El proyecto "Gestión de Equipo de Fútbol" sigue una arquitectura por capas de la siguiente manera:

- **Capa de Presentación:** Incluye las interfaces de usuario desarrolladas para interactuar con el sistema.
- **Capa de Aplicación:** Contiene la lógica de negocio relacionada con la gestión de equipos de fútbol, como la gestión de jugadores, partidos y estadísticas.
- **Capa de Datos:** Encargada de la persistencia y recuperación de datos desde la base de datos.

Estimación Económica de la Ejecución del Proyecto

1. Costos de Recursos Humanos

El equipo está compuesto por tres integrantes que han trabajado en roles definidos:

1. **Desarrollador Principal:** Encargado del backend y la lógica de negocio en Kotlin.
2. **Tester y Soporte:** Realiza pruebas funcionales y asegura la calidad.

Dado que el proyecto se completó en dos semanas, se estima un promedio de **20 horas por semana por integrante**, lo que equivale a **120 horas en total** para el equipo completo. En un entorno profesional, con un costo promedio de 25 EUR por hora, el valor del esfuerzo sería de:

120 horas x 25 EUR = 3,000 EUR.

2. Costos de Infraestructura y Herramientas

Para desarrollar el proyecto, se han utilizado ciertas herramientas y servicios esenciales:

- **IntelliJ IDEA:** Aunque la versión gratuita es suficiente, si se utilizara la versión Ultimate, su costo sería de aproximadamente **50 EUR** por un mes.
- **Hosting:** Se considera un servicio básico de hosting en la nube durante un mes, con un costo estimado de **100 EUR**.
- **Base de Datos:** Un servicio gestionado como PostgreSQL en la nube, con un costo mensual aproximado de **50 EUR**.
- **Otros Gastos:** Incluye licencias menores y herramientas de soporte, estimados en **20 EUR**.

El costo total de infraestructura y herramientas es de aproximadamente **3220 EUR**.

3. Costos Indirectos y Contingencias

Para cubrir posibles imprevistos y otros costos no contemplados, se incluye un **10% adicional** sobre el costo total calculado hasta ahora:

(3,000 EUR + 220 EUR) x 10% = 322 EUR.

Estimación Total del Proyecto

Sumando todos los componentes, el costo total estimado del proyecto es:

3,000 EUR (recursos humanos) + 3220 EUR (infraestructura) + 322 EUR (contingencia) = 6,542 EUR.