7-3-2025

documentación Proyecto

Jorge morgado, Ángel Sánchez, Antoine López

Contenido

[**Introducción** 2](#_Toc192499642)

[**Equipo de Desarrollo** 2](#_Toc192499643)

[**Gestión del Trabajo con Trello** 3](#_Toc192499644)

[**Flujo de Trabajo con GitFlow** 5](#_Toc192499645)

[**Especificación de Requisitos** 6](#_Toc192499646)

# **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.

El repositorio del proyecto puede ser consultado en: [Repositorio en GitHub](https://github.com/JorgeMrj/Proyecto-Gestion-Equipo-Futbol/tree/features).

**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. Carlos Rodríguez (Desarrollador)**

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

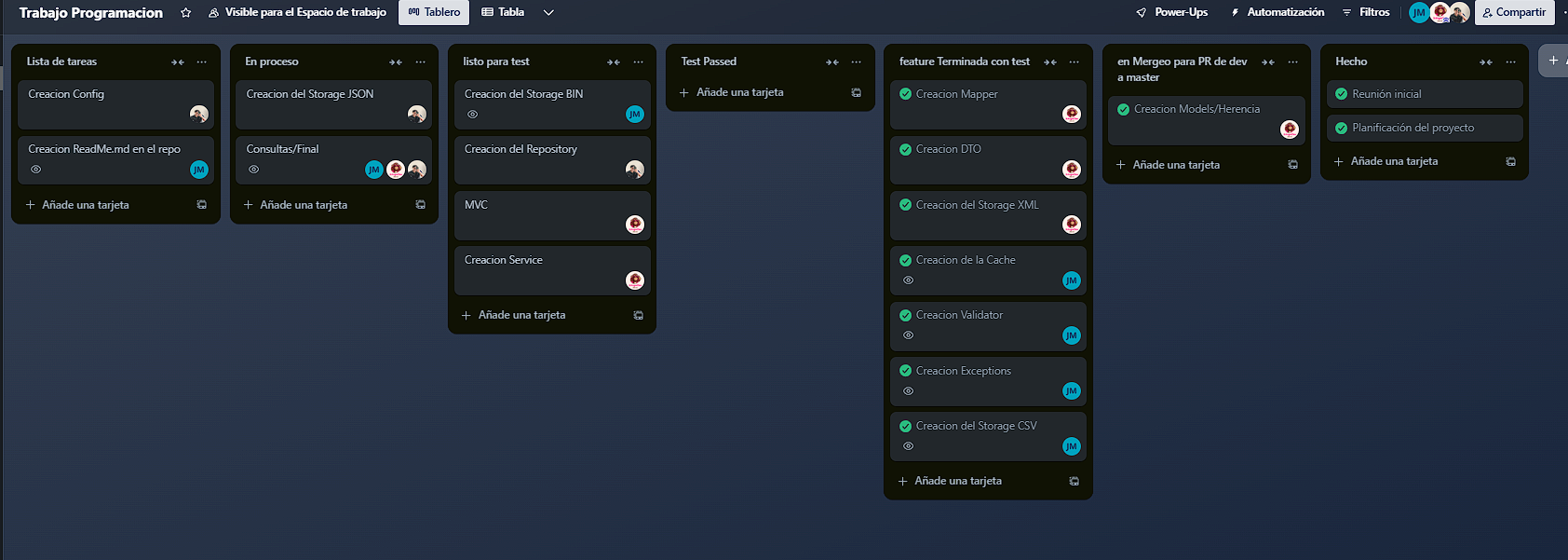
# **Gestión del Trabajo con Trello**

Para la organización y gestión del proyecto, se utilizó el tablero de Trello disponible en el siguiente enlace: [Tablero de Trabajo - Programación](https://trello.com/b/hWvpQUiC/trabajo-programacion). 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.  
       
       
       
     FALTA IMAGEN RAMAS JEJE

# **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.

Diagrama por hacer

# **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 cada uno de los elementos software existente justificando de las opciones posibles y porqué se ha decantado el equipo por ella, justificado pros y contras de la solución parcial aportada por dicho elemento.

PARA VOSOTROS

# **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