PLAN DE GESTIÓN DE LA CONFIGURACIÓN

FISIO FIND

Antonio Macías Ferrera, Benjamín Ignacio Maureira Flores

04/02/2025

Índice

Ficha del documento

- **Nombre del Proyecto:** FISIO FIND
- **Autores:** Antonio Macías Ferrera, Benjamín Ignacio Maureira Flores
- **Fecha de Creación:** 04/02/2025
- **Versión:** v2.0

Histórico de Modificaciones

Fecha	Versión	Realizada por	Descripción de los cambios
04/02/2025	v1.0	Antonio Macías Ferrera	Elaboración de la plantilla del documento.
05/02/2025	v2.0	Benjamín Ignacio Maureira Flores	Completado según la metodología a seguir en el proyecto.
13/02/2025	v2.1	Antonio Macías Ferrera	Adecuación al nuevo proyecto Fisio Find.

1. NORMAS Y PROCEDIMIENTOS APLICABLES

Este plan contiene la información sobre cómo el equipo de trabajo realizará el seguimiento y control del cambio durante el desarrollo del proyecto, además de cómo se llevará a cabo el control de versiones a lo largo del mismo.

2. ELEMENTOS CONFIGURABLES

Los elementos configurables del proyecto incluyen aquellos artefactos y entregables que pueden estar sujetos a modificaciones durante el desarrollo. Cada uno de estos elementos tendrá un identificador único, y se someterá a control de versiones.

1. **Documentación**: todos los documentos que surjan a lo largo de todas las fases del proyecto deberán estar sujetos a un sistema de control de versiones específico y unificado.
2. **Requisitos e HU**: define los requisitos del sistema. De este documento saldrán también los registros de HU y de casos de uso del sistema, que también serán configurables y estarán debidamente trazados.
3. **Código fuente**: código del producto software, sujeto a control de versiones.
4. **Tareas**: las fechas y condiciones para la entrega de los principales entregables de cada Sprint también serán elementos configurables.

3. HERRAMIENTAS A UTILIZAR

Tecnología	Elementos Configurables	Descripción
Pandoc y Eisvogel	Documentación, Registros	Plataforma para la edición de documentos en Markdown.
Clockify	Tareas	Seguimiento del tiempo de trabajo por actividad.
Taiga.io	Hitos, requisitos, HU	Herramienta ágil para gestión de proyectos.
Git	Código fuente	Control de versiones del código fuente.
GitHub	Código fuente, Hitos, HU	Gestión del desarrollo colaborativo.

4. CONTROL y POLÍTICA DE VERSIONADO

4.1. Versionado de Documentación y Registros

Se sigue un esquema de versionado semántico:

- **Mayor:** Cambios significativos en el contenido (reestructuración o nueva sección).
- **Menor:** Actualizaciones dentro de secciones existentes o correcciones menores.

Se deberá modificar la versión siempre que se realice alguna modificación en el documento. La versión se indicará tanto en el título del documento como en la tabla de control de versiones que se encontrará tras el índice del documento.

Ejemplo de versionado: v1 . 2 (vMayor.Menor)

4.2. HU, tareas y actividades

Cada historia de usuario y caso de uso tiene un identificador único asignado tras su creación.

Cambios importantes en la descripción o criterios de aceptación de una HU se anotan y actualizaremos la versión. Se seguirá un versionado similar al visto en el punto anterior. La versión de cada elemento se añadirá en un campo específico que encontraremos en su tabla.

- **Mayor**: Cambios significativos en el contenido debido a un cambio en los requisitos.
- **Menor**: Actualización de información, correcciones ortográficas, formato o pequeños ajustes de expresión.

Ejemplo de versionado: v1 . 2 (vMayor.Menor).

A la hora de crear nuevas tareas en *Taiga.io* en base a las HU, se deberá usar un nombre descriptivo y breve, similar al título de la HU asociada. Como esta tarea estará asociada a una Issue de GitHub irá acompañada de un código de la Issue(#XX). ***Esto ayudará a llevar una trazabilidad entre Issue - Tarea - HU - Clockify***.

La nomenclatura a seguir para los distintos requisitos, CU e HU se con un esquema de numeración para cada tipo de registro:

Categoría	Código
Objetivos de alto nivel	OBJ-XXX
Requisitos del proyecto	PRO-XXX
Requisitos de información	RI-XXX
Requisitos de reglas de negocio	RN-XXX
Requisitos de conducta	CON-XXX
Requisitos de fiabilidad	FIB-XXX
Requisitos de portabilidad	POR-XXX
Requisitos de seguridad	SEG-XXX
Requisitos de organización (incluye entrega, uso de estándares y tecnología)	ORG-XXX
Requisitos de factores ambientales (incluye requisitos legislativos y de privacidad)	FA-XXX
Cambios	CAM-XXX
Historias de usuario	HU-XXX
Casos de Uso	CU-XXX

4.3. Control del tiempo

Clockify no tiene control de versiones explícito, pero el seguimiento se realiza registrando las entradas de tiempo y las tareas completadas para cada miembro del equipo siguiendo la misma nomenclatura que las tareas de *Taiga.io*.

Las tareas se numeran en orden cronológico y deben incluir un identificador único en el formato #XX.

Se excluyen de esta nomenclatura las tareas de las fases de Inicio y Planificación. Todas las tareas deberán estar asignadas a UNA SOLA persona y a un proyecto de Clockify previamente creado. Esto ayudará evaluar mejor el desempeño y el tiempo empleado en las tareas.

Ejemplo de versionado: `Realización de ventana de inicio de sesión` #53.

4.4. Versionado de Código Fuente en Git y GitHub

Estructura de Ramas - **`main`** → Producción
- **`develop`** → Desarrollo
- **`feat/nueva_funcionalidad`** → Funcionalidades nuevas
- **`test/nueva_funcionalidad`** → Ramas de pruebas
- **`hotfix/corrección`** → Correcciones urgentes

Versionado semántico

X.y.z

- **X - Versión mayor**: Cambios mayores, rompe la compatibiladad de la API, la versión inicial será 0.y.z, la versión 1.0.0 definirá la primera API pública.
- **y - Versión menor**: Cambios menores, no rompen la compatibilidad de la API, incluyen nuevas funcionalidades y mejoras, puede incluir parches.
- **z - Parche**: Eliminación de bugs, no rompen la compatibilidad de la API, sólo cambios internos para arreglar comportamientos incorrectos.
- **Reglas de versionado**: Cuando la versión mayor sea incrementada se resetean las demás, cuando se incremente la versión menor, se resetea el parche. Seguramente se hagan de manera automática usando un workflow.

Commits (Conventional Commits) La política de nombrado de commits se ajustará a las directrices de *Conventional Commits*, siendo estos **siempre en inglés**:

```
<tipo>: <descripción breve>
[opcional] Cuerpo detallado del mensaje
[opcional] Pie de mensaje (referencias a tickets, breaking changes, etc.)
    El <tipo> especifica la naturaleza del cambio realizado. Los tipos más comunes incluyen:
    - feat: Una nueva funcionalidad.
    - fix: Corrección de errores.
    - docs: Cambios en la documentación (no relacionados con el código).
    - style: Cambios que no afectan la lógica del código (formato, espacios en blanco, etc.).
    - refactor: Cambios en el código que no corrigen errores ni añaden funcionalidades.
    - test: Adición o modificación de pruebas.
```

La descripción debe ser concisa y clara, expresando en una sola línea el propósito del commit. Se recomienda utilizar el modo imperativo (por ejemplo, “add...”, “fix...”) y evitar el uso de mayúsculas al inicio, salvo para nombres propios.

El cuerpo se utiliza para detallar el contexto del cambio, explicar el por qué detrás del commit, y describir cualquier implicación importante.

El pie del mensaje puede incluir: - Referencias a tickets o tareas: Refs #123. - Cambios significativos (breaking changes): BREAKING CHANGE: seguido de una descripción del cambio.

Se recomienda encarecidamente usar **Close #XX** cuando se cierre una Issue con dicho commit, para hacer los commits y las Issues lo más trazable posible. Esto se puede añadir a la descripción del commit o al cuerpo.

Ejemplo de mensaje de commit

```
feat: add support for token-based authentication Close #01

This change introduces a new authentication system based on JWT tokens.
    The user module has been updated, and a new dependency has been
   added.

BREAKING CHANGE: The login API now requires a JWT token instead of a
   cookie-based session.
```

4.5. Solicitudes de cambio (Registro de cambios)

Los registros se mantienen como archivos en formato *Markdown*. Cada cambio, HU o caso de uso tiene un identificador único (ejemplo: HU-001, CU-003). Cambios aprobados se reflejan en el historial de cambios del documento correspondiente actualizando su versión acorde a lo mencionado anteriormente.

Para las solicitudes de cambio formales, se deberá seguir la plantilla ubicada en el ***Plan De Gestión Del Cambio***.

5. ESTRATEGIA DE RAMAS

La estrategia de ramas se basa en **Git Flow** con algunas modificaciones para garantizar un desarrollo ágil y seguro.

Desarrollo de ramas que añadan funcionalidad:

1. Crear una rama `feat/` desde `develop`.
2. Desarrollar la funcionalidad en la rama creada.
3. Hacer Pull Request solicitando a un revisor del equipo antes de fusionar con `develop`.

Desarrollo de ramas que prueben una funcionalidad:

1. Crear una rama `test/` desde `develop` una vez se haya fusionado la funcionalidad a probar.
2. Desarrollar y ejecutar los tests en la rama creada.
3. Hacer Pull Request solicitando a un revisor del equipo antes de fusionar con `develop`.

Preparación de releases

- Se usará un workflow para hacer commit diario desde la rama `develop` a `main`, de manera que siempre tengamos un producto funcional actualizado en producción.

Arreglar bugs en producción

1. Crear una rama de hotfix desde `main`.
2. Corregir el error.
3. Fusionar la rama de hotfix con `main` y `develop`.

Revisión de las Pull Requests

Una vez acabada una tarea, se creará una Pull Request y se tendrá que notificar al resto de miembros del equipo, para que, con la mayor brevedad posible, un miembro la revise y dé feedback (aprobandola o solicitando cambios).

Se recomienda encarecidamente **no aprobar una Pull Request si no se han pasado los checks a la hora de ejecutar los workflows correspondientes**, para no arrastrar errores.

6. ESTÁNDARES DE CODIFICACIÓN

Para garantizar la consistencia, legibilidad y mantenibilidad del código, seguiremos estándares de codificación reconocidos en la industria para Python (backend) y JavaScript (frontend). A continuación, se detallan las prácticas recomendadas y ejemplos ilustrativos.

Python (Backend)

1. Nombrado

- **Positivo**:

```python
# Variables en camelCase y funciones en snake_case
userName = "John"
def calculate_total_price(items):
    pass

# Constantes en UPPER_CASE
MAX_CONNECTIONS = 100
```

- **Negativo**:

```python
  # Variables en UPPER_CASE (no recomendado)
USER_NAME = "John"

# Constantes en snake_case (no recomendado)
max_connections = 100
```

2. Indentación y espaciado

- **Positivo**:

```python
# Usar 4 espacios para la indentación
def greet(name):
    if name:
        print(f"Hello, {name}!")
    else:
        print("Hello, World!")
```

- **Negativo**:

```python
# Usar tabulaciones o 2 espacios (no recomendado)
def greet(name):
  if name:
    print(f"Hello, {name}!")
```

```
  else:
    print("Hello, World!")
```

3. Docstrings y comentarios

- **Positivo**:

```
# Docstrings para documentar funciones
def add(a, b):
    """
    Suma dos números y devuelve el resultado.

    :param a: Primer número
    :param b: Segundo número
    :return: Suma de a y b
    """
    return a + b

# Comentarios para explicar algo específico
counter = 0  # inicializa el contador
```

- **Negativo**:

```
# Sin docstrings o comentarios (no recomendado)
def add(a, b):
    return a + b

# Usando comentarios como docstrings y viceversa
def add(a, b):
    # Suma dos números y devuelve el resultado.

    # :param a: Primer número
    # :param b: Segundo número
    # :return: Suma de a y b
    return a + b

counter = 0  '''inicializa el contador'''
```

4. Manejo de excepciones

- **Positivo**:

```
# Capturar excepciones específicas
try:
    result = 10 / 0
except ZeroDivisionError as e:
    print(f"Error: {e}")
```

- **Negativo**:

```python
# Capturar excepciones genéricas (no recomendado)
try:
    result = 10 / 0
except:
    print("Ocurrió un error")
```

JavaScript (Frontend)

1. Nombrado

- **Positivo**:

```javascript
// Variables y funciones en camelCase
let userName = "John";
function calculateTotalPrice(items) {
    // ...
}

// Constantes en UPPER_CASE
const MAX_CONNECTIONS = 100;
```

- **Negativo**:

```javascript
// Variables en snake_case (no recomendado en JavaScript)
let user_name = "John";

// Constantes en camelCase (no recomendado)
const maxConnections = 100;
```

2. Uso de `let` y `const`

- **Positivo**:

```javascript
// Usar `const` para valores que no cambian
const PI = 3.1416;

// Usar `let` para valores que pueden cambiar
let counter = 0;
counter += 1;
```

- **Negativo**:

```javascript
// Usar `var` (no recomendado)
var counter = 0;
```

```
3
4 // Usar `let` para constantes (no recomendado)
5 let PI = 3.1416;
```

3. Formato y espaciado

- **Positivo**:

```
1 // Usar llaves y espacios consistentemente
2 function greet(name) {
3     if (name) {
4         console.log(`Hello, ${name}!`);
5     } else {
6         console.log("Hello, World!");
7     }
8 }
```

- **Negativo**:

```
1 // Formato inconsistente (no recomendado)
2 function greet(name){
3 if(name){
4 console.log(`Hello, ${name}!`);
5 }else{
6 console.log("Hello, World!");
7 }}
```

4. Manejo de promesas

- **Positivo**:

```
1 // Usar async/await para manejar promesas
2 async function fetchData() {
3     try {
4         const response = await fetch('https://api.example.com/data');
5         const data = await response.json();
6         console.log(data);
7     } catch (error) {
8         console.error("Error fetching data:", error);
9     }
10 }
```

- **Negativo**:

```
1 // Usar callbacks o .then() en exceso (no recomendado)
2 fetch('https://api.example.com/data')
3     .then(response => response.json())
```

```
4       .then(data => console.log(data))
5       .catch(error => console.error("Error fetching data:", error));
```

Todos estos ejemplos son una muestra de los más importantes que tendremos en cuenta (que quiere decir que no tenemos en cuenta el estandar al 100%), es responsabilidad de cada miembro del grupo adherirse a los estándares de código para garantizar un trabajo correcto.

7. GESTIÓN DEL CAMBIO

8. CONCLUSIÓN

Este documento define las políticas de gestión de configuración del proyecto **FISIO FIND**. Es crucial que estas buenas prácticas sean aplicadas por todos los miembros del equipo a lo largo de todo el proyecto para procurar un orden, trazabilidad y, en resumen, una buena calidad en el desarrollo y resultado del producto.