**ESCUELA POLITÉCNICA NACIONAL**
**FACULTAD DE INGENIERÍA DE SISTEMAS**
**CONSTRUCCIÓN Y EVOLUCIÓN DE SOFTWARE**

**Integrantes:** Gabriel Maldonado, Daniel Moncayo y Jhair Zambrano

**Curso:** GR2SW

**Fecha:** 6 de diciembre de 2025

# Proyecto

## Aplicación

HabitFlow

## Descripción

Este proyecto es un sistema web de seguimiento de hábitos personales diseñado para ayudar a los usuarios a construir y mantener rutinas positivas. La plataforma permite establecer hábitos organizados en distintas categorías (personal, deporte, académico, salud, etc.) y definir tareas específicas para cada uno.

Además, el sistema proporciona un módulo de visualización de progreso que muestra estadísticas detalladas sobre el cumplimiento de los hábitos, permitiendo al usuario evaluar su constancia y mejorar su desempeño a lo largo del tiempo.

## Plan de SCM

1. Planificación y diseño
   a. Configuración del entorno y repositorio

### Herramientas

Se utilizará Git/GitHub como sistema de control de versiones.

- **Frontend:** Se desarrollará utilizando React.js para crear una interfaz dinámica e interactiva.
- **Backend:** Se implementará con Node.js y Express para la gestión de la API.
- **Base de Datos:** Se utilizará MongoDB para almacenar la información de usuarios, hábitos y registros de seguimiento de forma flexible.

### Estructura del repositorio

Se trabajará con un monorepo llamado HabitFlow, donde se centralizará todo el código fuente del sistema (frontend y backend en carpetas separadas).

### Política de ramas

La rama principal del proyecto será main, la cual estará protegida de acuerdo con reglas previamente establecidas para asegurar la estabilidad del código en producción.

b. Definición del flujo de trabajo

**Estándar nombres de ramas**

Se utilizará una convención clara y uniforme para nombrar las ramas del proyecto. Los tipos más comunes serán:

- **feature/:** para nuevas funcionalidades.
  - *Ej: feature/crear-habito-deporte*
- **bugfix/: para corregir errores no críticos.**
  - *Ej: bugfix/error-calculo-estadistica*
- **hotfix/: para errores urgentes en producción.**
  - *Ej: hotfix/login-fallido*
- **refactor/: cambios de código sin alterar funcionalidad.**
  - *Ej: refactor/optimizar-componente-calendario*
- **chore/: tareas rutinarias (configuraciones, dependencias).**
  - *Ej: chore/actualizar-dependencias-npm*
- **docs/: documentación.**
  - *Ej: docs/actualizar-readme*

**Reglas de nomenclatura:**

- Las ramas deben escribirse en minúsculas.
- No se permiten espacios; se deben usar guiones para separar palabras.
- Cada rama debe corresponder a un objetivo claro y específico.

**Política para intercambiar cambios**

Todo cambio en el repositorio deberá integrarse mediante un pull request.

- Las revisiones deben ser realizadas por al menos un integrante del equipo (o revisadas por el propio autor con rigor si es individual), verificando que el código cumple con el requisito funcional.
- El revisor debe comprobar que la lógica de cálculo de hábitos es correcta y no afecta a las estadísticas históricas.

c. Gestión de artefactos

Los requisitos funcionales, las definiciones de las categorías de hábitos y los diseños de la interfaz (mockups) se documentarán y almacenarán dentro de la Wiki de GitHub del repositorio.

**2.** Codificación y pruebas

a. Gestión de cambios en acción

Cuando un desarrollador finaliza una tarea (ej. "Implementar gráfico de barras para hábitos académicos"), debe realizar un pull request. Si la tarea cumple con los requisitos y pasa las pruebas, el pull request será aprobado y fusionado a la rama principal.

b. Integración Continua

El proyecto implementará procesos de integración continua (CI) basado en GitHub Actions. La configuración ejecutará dos tareas esenciales en cada push y pull request:

1. **Linting:** Se usará ESLint para garantizar un estilo de código consistente en JavaScript/React y detectar errores de sintaxis tempranos.
2. **Pruebas unitarias:** Estas serán ejecutadas con Jest, permitiendo validar la integridad de las funciones críticas, como el cálculo de rachas de hábitos y porcentajes de cumplimiento.

c. Gestión de líneas base

La línea base representa un conjunto de componentes del sistema que se consideran estables.

Para el proyecto HabitFlow, se establecerá una línea base cuando se completen los módulos fundamentales. Una vez finalizados y validados los módulos de "Gestión de Categorías", "CRUD de Hábitos" y "Visualización de Estadísticas Básicas", se creará un tag en el repositorio para marcar esta versión como estable.

El tag definido para esta primera línea base será: *v1.0-base*.

**3.** Despliegue y mantenimiento

a. Gestión de releases y despliegue

Se utilizará un proceso controlado para generar versiones estables del sistema HabitFlow.

**Versionamiento**

Se utilizará Versionamiento Semántico (SemVer) con el formato: MAJOR.MINOR.PATCH.

- PATCH (v1.0.1): Correcciones menores en la interfaz o ajustes en fórmulas de estadísticas.
- MINOR (v1.1.0): Nuevas funcionalidades, como "Modo Oscuro" o "Nuevas categorías de deporte".

- MAJOR (v2.0.0): Cambios grandes, como una reescritura del motor de base de datos o cambio total de interfaz.

**Flujo para generar un Release**

1. Un cambio termina su desarrollo en una rama feature/.
2. Se realiza un pull request hacia main y es aprobado.
3. Una vez fusionado, se crea un tag con la versión correspondiente (ej. v1.0.0).
4. Ese tag representa una línea base estable de HabitFlow.

**Despliegue**

Dado que HabitFlow es una aplicación web (MERN Stack), el despliegue inicial será manual controlado o mediante servicios como Vercel/Render.

- Las versiones etiquetadas (tags) serán las únicas autorizadas para desplegarse en el entorno de producción para asegurar la estabilidad.

**Entornos**

- **Staging:** Entorno de pruebas para validar que las estadísticas se calculan bien antes de liberar.
- **Producción:** La versión estable disponible para los usuarios finales.

**Hotfixes**

En caso de un error crítico (ej. el sistema no guarda el progreso diario):

1. Se crea una rama hotfix/ desde main.
2. Se corrige el problema y se fusiona a main.
3. Se publica un release tipo PATCH (ej. v1.0.1).

b. Plan de mantenimiento proactivo

**Proceso para bugs reportados**

- Reporte: Los usuarios reportan errores (ej. "No puedo crear un hábito de deporte") mediante un issue en GitHub con etiqueta bug.
- **Clasificación:**
  - Crítico (Hotfix): El sistema no carga o se pierden datos. Etiqueta priority: critical.
  - Grave: Funcionalidad parcial afectada (ej. gráficas no cargan, pero los datos están). Etiqueta priority: high.
  - Menor: Errores visuales o de texto. Etiqueta priority: low.

**Calendario de mantenimiento programado**

- Mensual: Actualizar dependencias de npm (React/Node) y revisar logs de errores.
- Trimestral: Auditoría de seguridad de la base de datos y optimización de consultas de estadísticas.
- Semestral: Revisión de la arquitectura y actualización de versiones menores de librerías.
- Anual: Evaluación para actualización de versión mayor (Major Release) y limpieza profunda de base de datos.

## 4. Referencias

[1]. Sommerville, *Software Engineering*, 10th ed. Boston, MA, USA: Pearson, 2016.
[2]. S. Chacon and B. Straub, *Pro Git*, 2nd ed. Apress, 2014. [En línea]. Disponible en: https://git-scm.com/book/en/v2
[3]. K. Beck et al., "Manifesto for Agile Software Development," Agile Alliance, 2001. [En línea]. Disponible en: https://agilemanifesto.org/
[4]. GitHub, "GitHub Flow," GitHub Docs, 2024. [En línea]. Disponible en: https://docs.github.com/en/get-started/using-github/github-flow
[5]. T. Preston-Werner, "Semantic Versioning 2.0.0," SemVer.org, 2013. [En línea]. Disponible en: https://semver.org/