Integrantes: Fernando Huilca, Mateo Simbaña

Curso: GR2SW

Fecha: 14 de diciembre de 2025

Índice

1. Definición de proyecto

1.1 Nombre

Amauta

1.2 Descripción

Este proyecto es un sistema de gestión de cursos académicos que permite a los profesores crear y publicar sus propios cursos de forma sencilla, incluyendo materiales de estudio como documentos PDFs, videos, imágenes y otros recursos académicos. Por su parte, los estudiantes pueden matricularse en los cursos disponibles y aprender de manera dinámica e interactiva.

2. Plan de SCM

2.1 Planificación y diseño

2.1.1 Configuración del entorno y repositorio

Herramientas

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

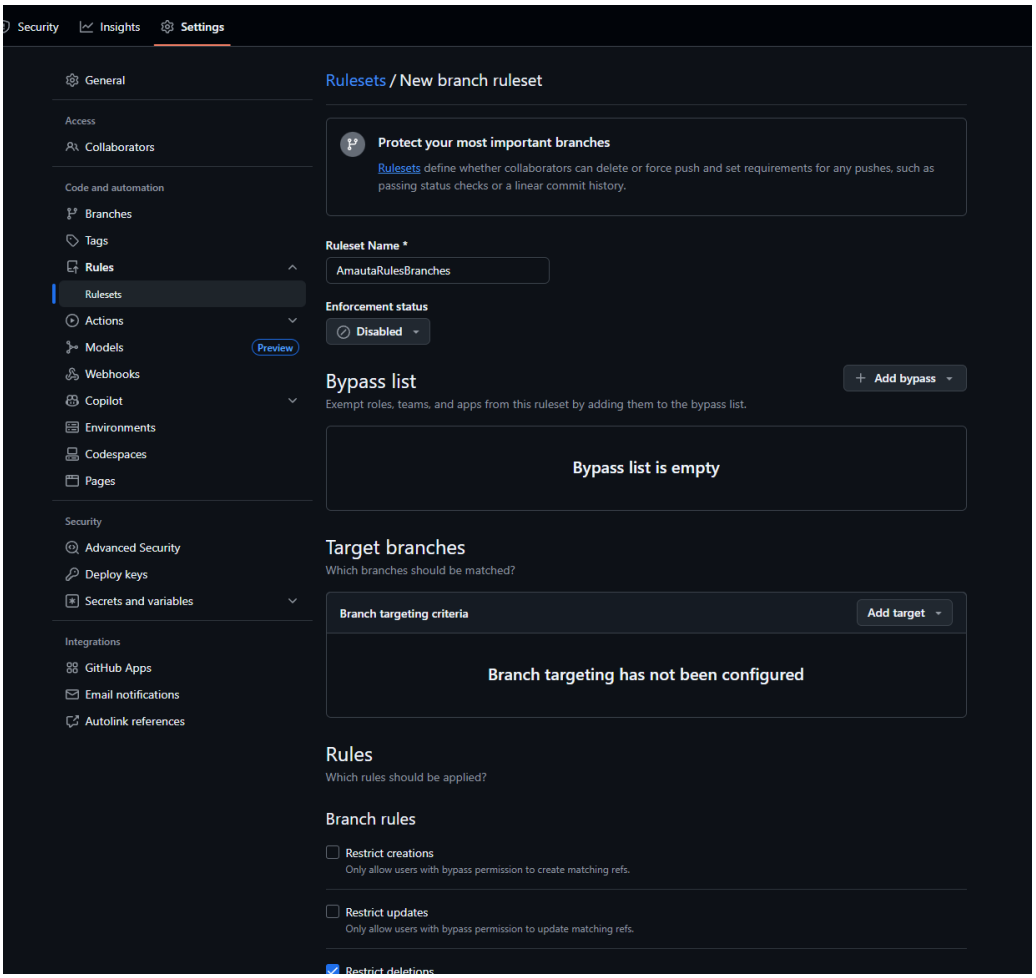
Para el desarrollo del frontend se emplearán **HTML, CSS y JavaScript**, mientras que el backend estará implementado en **PHP**. La base de datos será gestionada con **MySQL**.

Estructura del repositorio

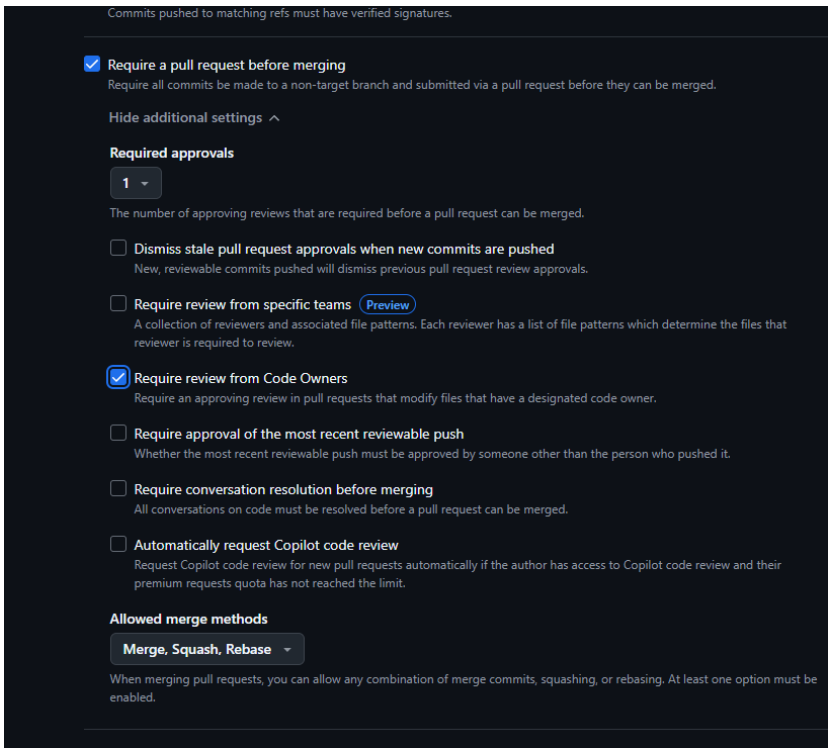
Se trabajará con un **monorepo** llamado *Amauta*, donde se centralizará todo el código fuente del sistema.

Política de ramas

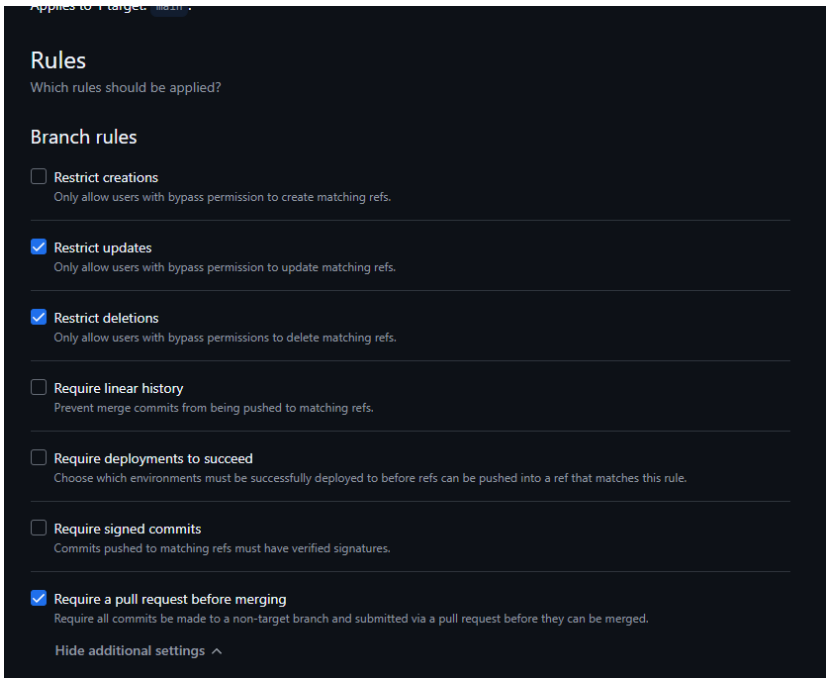
La rama principal del proyecto será **main**, la cual estará protegida de acuerdo con reglas previamente establecidas.



La regla colocada hace que ningún miembro del equipo pueda hacer un cambio directo al **main** sin antes hacer un **pull request**. Asimismo, solo se efectuarán los cambios si el creador del repositorio lo aprueba.



Además, se añadieron restricciones adicionales para asegurar que ningún miembro del equipo pueda **actualizar, eliminar o forzar actualizaciones (force push)** sobre la rama main, a excepción de las personas explícitamente autorizadas.



2.1.2 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/login-de-usuarios
- **bugfix/:** para corregir errores
  Ej: bugfix/imagen-no-carga
- **hotfix/:** para errores urgentes en producción
  Ej: hotfix/error-en-pago
- **refactor:** cambios de código sin alterar funcionalidad
  Ej: refactor/ordenar-servicios
- **chore/:** tareas rutinarias (config, dependencias)
  Ej: chore/actualizar-eslint
- **docs/:** documentación
  Ej: docs/agregar-readme

Las reglas de nomenclatura son las siguientes:

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

Política para intercambiar cambios

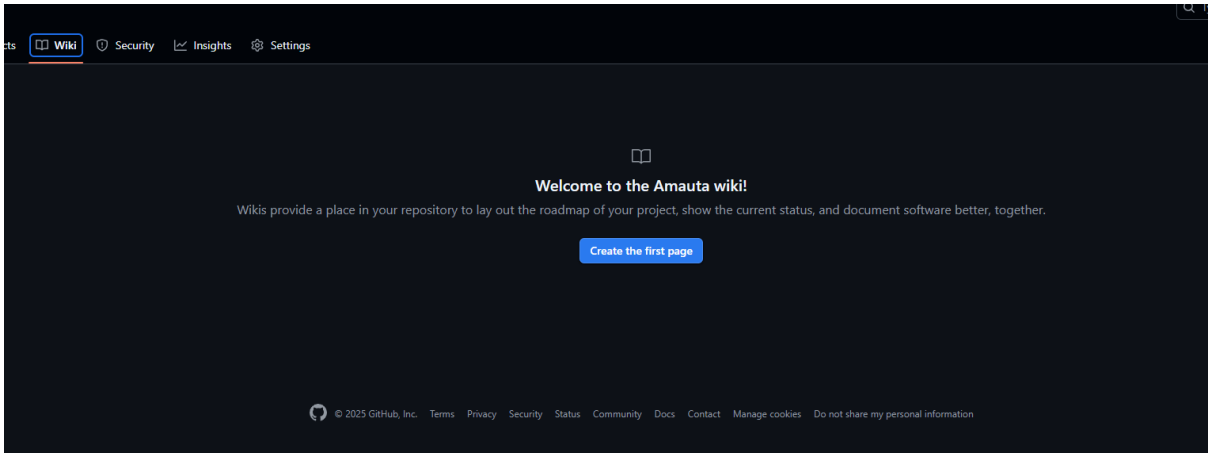
Todo cambio en el repositorio deberá integrarse mediante un **pull request**.

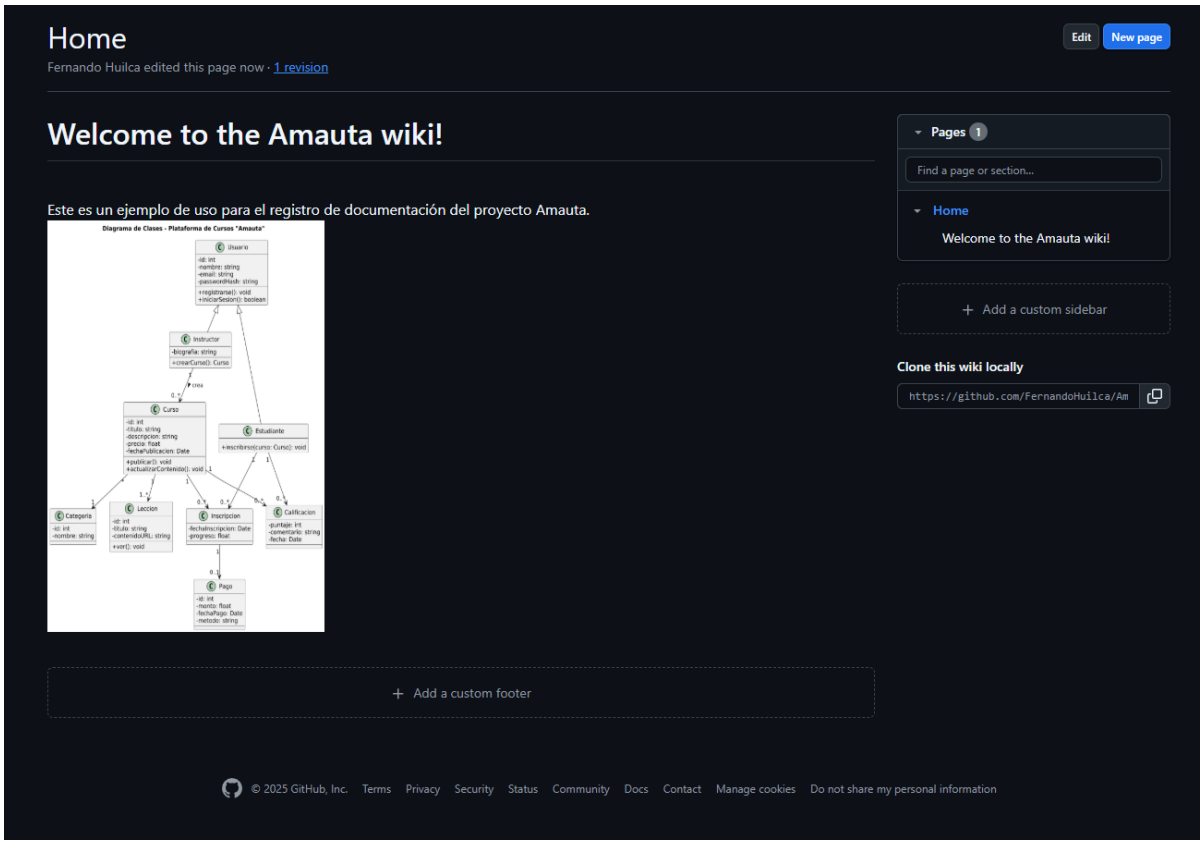
Las revisiones deben ser realizadas por **dos integrantes del equipo**, quienes tienen prohibido aprobar usando únicamente el mensaje superficial **LGTM (Looks Good To Me)**.

El revisor debe comprobar que el cambio cumple con el objetivo asignado, respeta la estructura del proyecto y no afecta funciones existentes.

2.1.3 Gestión de artefactos

Los requisitos y el enlace de los diseños de interfaz se almacenarán dentro de la Wiki de Github.





2.2 Codificación y pruebas

2.2.1 Gestión de cambios en acción

La persona que se haya encargado de un requisito debe realizar un pull request. La persona encargada de la revisión debe encargarse de verificar que la tarea cumple con el requisito establecido. Si la tarea es satisfactoria el pull request será aprobado.

2.2.2 Integración continua

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

Linting, se usará PHP_CodeSniffer para garantizar un estilo de código consistente y detectar malas prácticas tempranas.

Pruebas unitarias, estas serán ejecutadas en PHPUnit, permitiendo validad la integridad de las funcionalidades a medida que el proyecto crece.

2.2.3 Gestión de líneas base

La línea base representa un conjunto de componentes del sistema que se consideran estables y listos para ser utilizados como punto de referencia para futuras integraciones o versiones.

Para el proyecto Amauta, se establecerá una línea base cuando se completen los módulos fundamentales del sistema.

Una vez finalizados y validados los módulos de Registro de Cursos, Gestión de Profesores y Autenticación inicial, 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**

Esta línea base servirá como punto de partida para el desarrollo de nuevas funcionalidades, permitiendo comparar cambios, realizar integraciones controladas y asegurar que el equipo siempre pueda regresar a un estado conocido y funcional del sistema.

2.3 Despliegue y mantenimiento

2.3.1 Gestión de releases y despliegue

Para Amauta usaremos un proceso sencillo pero controlado para generar versiones estables del sistema y desplegarlas en el servidor donde estarán disponibles para los usuarios.

Versionamiento

Se utilizará Versionamiento Semántico (SemVer) con el formato: MAJOR.MINOR. PATCH (ej. *v1.0.0*)

- **PATCH (v1.0.1):** correcciones menores o arreglos de bugs.
- **MINOR (v1.1.0):** nuevas funcionalidades que no rompen compatibilidad.
- **MAJOR (v2.0.0):** cambios grandes que modifican estructuras del sistema o rompen compatibilidad.

Cada versión estable del proyecto será etiquetada con un tag de Git, por ejemplo: v1.0.0, v1.1.0, v1.1.1.

Flujo para generar un Release

1. Un cambio termina su desarrollo en una rama feature/, bugfix/ o hotfix/.
2. Se realiza un **pull request hacia main**, revisado por los miembros del equipo.
3. Si el PR es aprobado, se fusiona en la rama principal.
4. Una vez que los cambios están en main, se crea un **tag** con la versión correspondiente (v1.x.x).
5. Ese tag representa una **línea base** estable de Amauta.

Despliegue

Debido a que Amauta usa tecnologías simples (PHP, HTML, CSS, JS, MySQL), el despliegue será:

- **Manual controlado** en esta primera etapa.
- El equipo subirá los archivos actualizados de la versión etiquetada (tag) al servidor o hosting del proyecto (ej. panel del hosting, FTP o Git Pull).
- Las actualizaciones de la base de datos (si las hubiera) se aplicarán solo después de confirmar que el código ya está desplegado correctamente.

Este proceso garantiza que solo versiones estables y probadas lleguen al entorno donde los profesores y estudiantes usarán el sistema.

Entornos

- **Staging (pruebas):** versión previa para validar funcionalidades antes de liberar un release oficial.
- **Producción:** la versión estable para los usuarios.

Las versiones etiquetadas como v1.0.0, v1.1.0, etc. serán las que se desplegarán en producción.

Hotfixes

En caso de un error crítico:

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

2.3.2 Plan de mantenimiento proactivo

Proceso para bugs reportados

- **Reporte:** Los usuarios (profesores/estudiantes) reportan bugs mediante formulario web que crea automáticamente un GitHub Issue con etiqueta bug.
- **Clasificación:**
    - **Crítico (Hotfix):** Bugs que impiden funcionalidad esencial (ej: no se pueden subir archivos, no funciona el pago, error 500 general). Se etiquetan como priority: critical.
    - **Grave:** Bugs que afectan experiencia, pero no bloquean (ej: formato incorrecto en materiales, problemas en visualización móvil). Etiqueta priority: high.
    - **Menor:** Bugs cosméticos o de baja prioridad. Etiqueta priority: low.

Calendario de mantenimiento programado

Mensual (Primer Lunes de cada mes):

- Actualizar dependencias (via Dependabot PRs)
- Revisar y rotar credenciales de acceso
- Análisis de logs para patrones problemáticos
- Limpieza de archivos temporales y cache

Trimestral:

- Auditoría de seguridad completa
- Optimización de índices de base de datos
- Revisión y actualización de documentación
- Prueba de restauración de backups

Semestral (Enero y Julio):

- Actualización de versión menor de PHP/Symfony
- Revisión de arquitectura y propuestas de mejora
- Capacitación equipo en nuevas tecnologías relevantes
- Revisión de métricas de negocio vs. técnico

Anual (Diciembre):

- Actualización de versión mayor (si aplica)
- Revisión completa de infraestructura

- Plan de mantenimiento para próximo año
- Retrospectiva de mantenimiento del año

3. Referencias

[1] I. Sommerville, *Software Engineering*, 10th ed. Boston, MA, USA: Pearson, 2016.

[2] R. C. Seacord, D. Plakosh, and G. A. Lewis, *Modernizing Legacy Systems: Software Upgrade and Migration*. Boston, MA, USA: Addison-Wesley, 2003.

[3] E. Spinellis, *Version Control with Git: Powerful Tools and Techniques for Collaborative Software Development*, 2nd ed. Sebastopol, CA, USA: O'Reilly Media, 2012.

[4] K. Beck et al., "Manifesto for Agile Software Development," Agile Alliance, 2001. [En línea]. Disponible en: https://agilemanifesto.org/

[5] M. Fowler and M. Foemmel, "Continuous Integration," ThoughtWorks, 2006. [En línea]. Disponible en: https://martinfowler.com/articles/continuousIntegration.html