ESCUELA POLITÉCNICA NACIONAL

Facultad de Ingeniería de Sistemas
Asignatura: Construcción y Evolución de Software

Proyecto Final

Daniel Moncayo
Jhair Zambrano
Gabriel Maldonado

Curso: GR2SW

2025B

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

1. Introducción

Habit Tracker es un sistema de seguimiento de hábitos desarrollado como proyecto académico en el curso de Metodologías Ágiles (ISWD622-25B). El proyecto busca resolver el problema de la **falta de visibilidad y seguimiento continuo en el cambio de hábitos**, proporcionando a los usuarios una plataforma integral para:

- Registrar y monitorear hábitos personales (positivos y negativos)
- Rastrear el progreso y rachas de cumplimiento
- Registrar el estado emocional y su relación con los hábitos
- Realizar journaling y reflexiones sobre el progreso
- Visualizar estadísticas e insights sobre el comportamiento

Este es un proyecto educativo desarrollado bajo metodología SCRUM, con énfasis en prácticas de ingeniería de software modernas: integración continua (CI), entrega continua (CD), pruebas automatizadas y gestión ágil del código.

---

2. Arquitectura del Proyecto

Backend:
- Lenguaje: Java 21 (Maven)
- Framework Web: Servlets & JSP (Jakarta Servlet API 6.0 / Jakarta JSP 3.1)
- ORM: Hibernate 6.3.1 (JPA 3.1)
- Bases de Datos: PostgreSQL (Supabase) en producción / H2 en desarrollo local

Frontend:
- Vistas: JSP (JavaServer Pages)
- Componentes: HTML5, CSS3, JavaScript vanilla
- Despliegue: Tomcat 10.1

Testing:
- Unit Tests: JUnit 4.13.2 + JUnit Jupiter (5.10.2)
- Mocking: Mockito 5.11.0

- Test Coverage: JaCoCo

DevOps & CI/CD:
- Plataforma: Azure DevOps / Azure Pipelines
- Artefactos: WAR (Web Application Archive)
- Despliegue: Azure App Service

---

3. Estrategia de Pipelines (CI/CD)

Pasos del Pipeline CI:

1. **Compilación y Build (Maven)**
   - Tarea: `Maven@3` - Build WAR with Maven (Java 21)
   - Comando: `mvn clean package`
   - Objetivo: Compilar código fuente, ejecutar pruebas unitarias automáticamente, generar artefacto WAR
   - Entrada: `pom.xml`, código fuente en `src/main/java`
   - Salida: `sistema-seguimiento.war`

2. **Publicación de Artefactos**
   - Tarea: `PublishBuildArtifacts@1`
   - Publica el WAR compilado al repositorio de artefactos de Azure DevOps
   - Permite descarga por pipelines posteriores (CD)

Ejecución Temporal: ~5-10 minutos

Pipeline de Entrega Continua (CD): `azure-pipelines-1.yml`

Descripción: Se ejecuta tras finalización exitosa del pipeline CI, pero solo cuando se aprueban cambios a `main`.

Pasos del Pipeline CD:

1. **Descarga de Artefactos**
   - Descarga el WAR generado por el pipeline CI
   - Ubicación: `$(Pipeline.Workspace)/ciPipeline/drop/*.war`

2. **Deploy a Azure App Service**
   - Tarea: `AzureWebApp@1` - Deploy WAR to Azure App Service (Tomcat 10.1)
   - Suscripción Azure: `AzureForStudents-HabitTracker3`
   - Aplicación objetivo: `habit-tracker-app-ma`
   - Entorno: Tomcat 10.1 en Azure App Service
   - El WAR se despliega y la aplicación queda disponible en producción/staging

Ejecución Temporal: ~5-15 minutos

---

4. Estrategia de Flujos de Desarrollo

El proyecto utiliza un modelo de branching inspirado en Git Flow, adaptado para SCRUM:

main (rama principal - producción)
│
├─── hotfix/correccion-urgente (correcciones críticas)
│
develop (rama de integración)
│
├─── feature/HU-001-establecer-habito
├─── feature/HU-002-registrar-emociones
├─── feature/HU-003-ver-progreso
├─── feature/HU-004-journaling
└─── feature/HU-005-estadísticas

Descripción de cada rama:

- `main`: Rama de producción. Solo contiene código verificado y liberado. Cada commit debe tener un tag de versión (v1.0.0, v1.1.0, etc.). Solo se reciben cambios mediante PRs desde `develop`.

- `develop`: Rama de integración del equipo. Contiene la versión más reciente con nuevas funcionalidades. Cada historia de usuario completa se integra aquí mediante Pull Request.

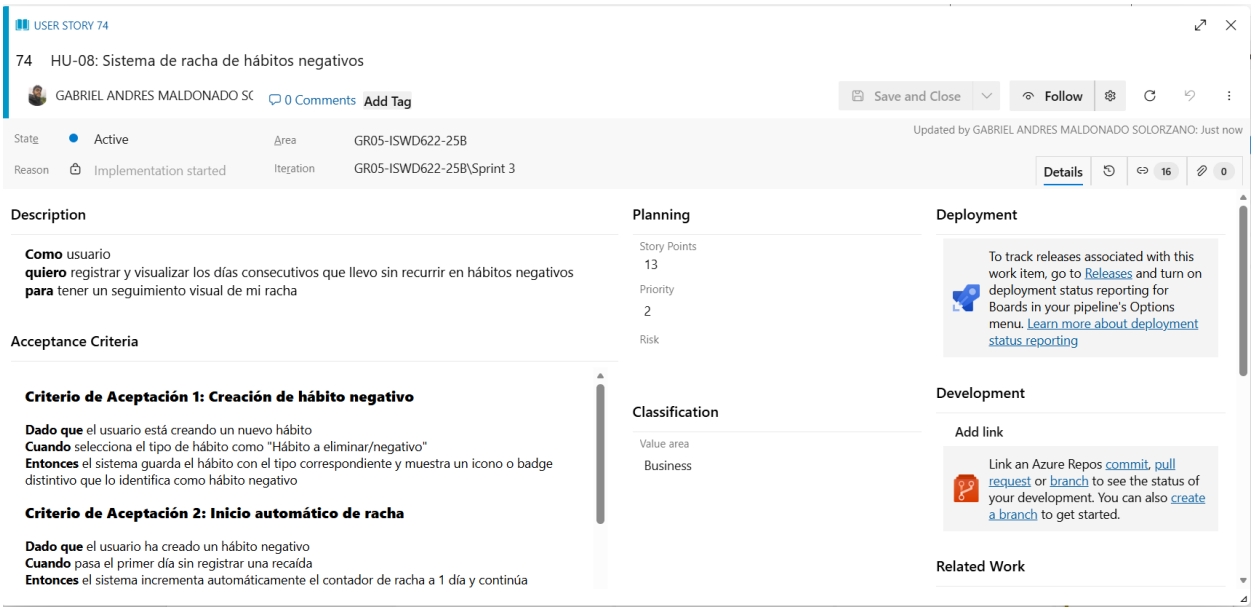
- `feature/HU-XXX-descripcion`: Rama para cada historia de usuario. Se crea a partir de `develop`. Al terminar el desarrollo, se abre un PR hacia `develop` solicitando revisión.

- `hotfix/descripcion-correccion`: Rama para correcciones urgentes en producción. Se crea a partir de `main`, se corrige el bug, y se fusiona tanto a `main` como a `develop` para sincronización.

---

5. Gestión de Historias de Usuario

El proyecto adopta el formato estándar de historias de usuario SCRUM:
Como [rol], quiero [funcionalidad], para [beneficio/valor].



---

6. Estrategia de Revisiones y Aprobaciones

Estructura obligatoria de un PR:
Descripción
Breve resumen de cambios (2-3 líneas)

Tipo de Cambio
- [ ] Corrección de bug
- [ ] Nueva funcionalidad
- [ ] Cambio en funcionalidad existente
- [ ] Cambio que requiere actualización de docs

Historia de Usuario

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

Vinculado a: HU-XXX - [Título de Historia]

Cambios Realizados
- Cambio 1
- Cambio 2
- Cambio 3

Pruebas Ejecutadas
- [ ] Pruebas unitarias ejecutadas localmente
- [ ] Pruebas de integración pasadas
- [ ] Cobertura de código verificada (>60%)

Capturas de Pantalla (si aplica)
[Adjuntar screenshots de UI changes]

Checklist de Revisión
- [ ] Código cumple con estándares de estilo (Java conventions)
- [ ] Pruebas unitarias escritas y exitosas
- [ ] Documentación actualizada (Javadoc)
- [ ] No hay warnings en el IDE
- [ ] Build exitoso en pipeline CI
- [ ] Sin conflictos de merge

Proceso de Revisión

Responsabilidades del Revisor:

1.Claridad: ¿Es claro qué hace el código?
2. Correctitud: ¿El código resuelve el problema sin bugs?
3. Estándares: ¿Cumple con convenciones del proyecto?
4. Pruebas: ¿Hay pruebas adecuadas? ¿Cobertura > 80%?
5. Performance: ¿Hay problemas de performance?
6. Documentación: ¿Está documentado apropiadamente?

Decisiones de Revisión:

Aprobar: El código está listo para merge
Comentar: Requiere cambios menores (conversar con autor)

Solicitar Cambios: Requiere cambios significativos antes de merge

---

7. Herramientas y Conexiones

En cuanto a la gestión de tareas y seguimiento del proyecto, se utiliza Azure DevOps o Jira para crear historias de usuario, organizar sprints, y mantener un registro centralizado del trabajo realizado. Estas plataformas actúan como el centro neurálgico donde el equipo planifica, asigna y rastrea el progreso de cada funcionalidad. El control de versiones se maneja a través de GitHub o Azure Repos, donde se almacena todo el código fuente, se gestionan las ramas según el modelo Git Flow, y se mantiene un historial completo de todos los commits y cambios realizados.
Para la automatización del ciclo de vida del software, Azure Pipelines orquesta toda la cadena de CI/CD, disparándose automáticamente en cada push a las ramas principales. La compilación y gestión de dependencias se realiza con Maven, que interpreta el archivo pom.xml para resolver todas las librerías necesarias, compilar el código Java 21, empaquetar la aplicación en un archivo WAR, y preparar los artefactos para deployment.
Las pruebas del sistema se ejecutan mediante JUnit 5 para pruebas unitarias y Mockito para crear mocks de componentes dependientes, permitiendo aislar y validar la lógica de negocio de cada clase. JaCoCo trabaja en conjunto con JUnit para medir la cobertura de código, generando reportes detallados que indican qué porcentaje de líneas de código fueron ejecutadas durante las pruebas. De forma opcional, SonarQube puede integrarse para proporcionar análisis estático profundo, detectando problemas de calidad, vulnerabilidades de seguridad y deuda técnica potencial.
La persistencia de datos se maneja a través de PostgreSQL alojado en la plataforma Supabase, que proporciona una base de datos relacional completamente gestionada. Hibernate/JPA abstrae las operaciones de base de datos, mapeando las entidades Java directamente a las tablas SQL, permitiendo que los desarrolladores trabajen con objetos en lugar de escribir SQL manualmente.
El hosting y despliegue de la aplicación se realiza en Azure App Service, que ejecuta un servidor Tomcat 10.1 compatible con Jakarta Servlet. Este servicio proporciona escalabilidad automática, monitoreo de telemetría, y una plataforma robusta para ejecutar aplicaciones web Java en producción.

8. Conclusiones

1. Calidad del Código

- Pruebas automatizadas en cada build (JUnit + Mockito)
- Cobertura de código medida y reportada (JaCoCo > 80%)
- Revisión obligatoria de pares antes de merge
- Análisis de código estático (SonarQube optional)
- Validación automática en PRs con Devin AI

2. **Trazabilidad**

- Cada commit vinculado a historia de usuario (HU-XXX)
- PRs con descripción detallada y checklist de calidad
- Integración GitHub ↔ Jira automatizada
- Historial completo de cambios con justificación
- Tags de versión en `main` para releases

3. Automatización (CI/CD)

- Build automático en cada commit a ramas principales
- Pruebas ejecutadas automáticamente (sin overhead manual)
- Deploy automático a Azure App Service tras merge
- Reducción de errores manuales y downtime
- Feedback inmediato al desarrollador

4. Evolución Sostenible

- Metodología SCRUM con sprints de 2 semanas

- Modelo de ramas claro (Git Flow) facilita trabajo en paralelo
- Separación clara de responsabilidades (arquitectura N-Tier)
- Documentación de procesos y estándares
- Escalabilidad: Fácil agregar nuevos desarrolladores

5. Confiabilidad en Producción

- Pipeline CI valida cada cambio antes de deploy
- Ambiente de staging para pre-release testing
- Rollback fácil: Tags de versión en repositorio
- Monitoreo continuo (Azure App Service telemetry)
- Recuperación ante fallos: Hotfix rápido via `hotfix/*`