Curso de Programación Desde Cero

Tema 7: Git y Control de Versiones

Duración estimada: 10 horas

1. ¿QuØ es el Control de Versiones?

El control de versiones es un sistema que registra los cambios realizados en archivos a lo largo del tiempo, permitiendo recuperar versiones específicas cuando sea necesario.

Problemas que Resuelve:

2 Historial completo de cambios en el proyecto
2 Colaboración segura entre mœltiples desarrolladores
2 Capacidad de revertir cambios problemÆticos
2 Ramificación para desarrollar características en paralelo
2 Respaldo distribuido del código fuente

2. Historia y Evolución

Los sistemas de control de versiones han evolucionado:

Sistemas Locales (RCS):

2 Guardaban parches en disco duro local
2 No permitían colaboración

Sistemas Centralizados (CVS, Subversion):

2 Servidor central con todo el historial
2 Clientes solo tenían œltima versión
2 Punto œnico de falla

Sistemas Distribuidos (Git, Mercurial):

2 Cada cliente tiene copia completa del historial
2 Trabajo offline completo
2 Mœltiples respaldos automÆticos
2 Ramificación y fusión muy eficientes

3. Introducción a Git

Git es el sistema de control de versiones mÆs popular del mundo, creado por Linus Torvalds en 2005 para el desarrollo del kernel de Linux. Características principales:

2 Velocidad: Operaciones locales muy rÆpidas
2 Diseæo simple: Conceptos claros y consistentes
2 Soporte para desarrollo no lineal: Ramificación eficiente
2 Completamente distribuido: Sin dependencia de servidor
2 Manejo eficiente de proyectos grandes

4. Configuración Inicial de Git

Antes de empezar a usar Git, debemos configurar nuestra identidad:

Configuración Global (para todos los repositorios):

```
git config --global user.name "Tu Nombre"
git config --global user.email "tu.email@ejemplo.com"
git config --global init.defaultBranch main
```

Configuración œtil adicional:

```
git config --global core.editor "code --wait"  # VS Code como editor
git config --global core.autocrlf true         # Windows
git config --global core.autocrlf input        # macOS/Linux
```

Verificar configuración:

```
git config --list
git config user.name
git config user.email
```

5. Conceptos Fundamentales

Repositorio (Repository):

Directorio que contiene tu proyecto y todo el historial de Git.

Working Directory (Directorio de Trabajo):

Los archivos actuales en los que estÆs trabajando.

Staging Area (´ndice):

Zona intermedia donde preparas los cambios para el próximo commit.

Commit:

InstantÆnea de tu proyecto en un momento específico del tiempo.

Estados de los Archivos:

2 Untracked: Archivos nuevos que Git no conoce
2 Modified: Archivos modificados pero no preparados
2 Staged: Archivos preparados para el próximo commit
2 Committed: Archivos guardados en el repositorio

6. Comandos BÆsicos de Git

Inicializar un repositorio:

```
git init                  # Crea repositorio en directorio actual
git init mi-proyecto      # Crea directorio y repositorio
```

Ver estado del repositorio:

```
git status                # Estado completo
git status -s             # Estado resumido
```

7. Flujo de Trabajo BÆsico

Aæadir archivos al staging area:

```
git add archivo.txt        # Aæadir archivo específico
git add .                  # Aæadir todos los archivos modificados
git add *.js               # Aæadir archivos por patrón
git add -A                 # Aæadir todo (incluso eliminados)
```

Crear commits:

```
git commit -m "Mensaje descriptivo del cambio"
git commit -am "Aæadir y commit en un paso"
git commit --amend         # Modificar œltimo commit
```

Ver historial:

```
git log                    # Historial completo
git log --oneline          # Una línea por commit
git log --graph            # Vista grÆfica
git log -p                 # Con diferencias
git log --since="2 weeks ago"
```

Ver diferencias:

```
git diff                   # Cambios no preparados
git diff --staged          # Cambios preparados
git diff HEAD~1            # Comparar con commit anterior
```

8. Trabajando con Ramas (Branches)

Las ramas permiten desarrollar características en paralelo sin afectar la rama principal del proyecto.

Comandos de ramas:

```
git branch                 # Ver ramas locales
git branch -a              # Ver todas las ramas
git branch nueva-feature   # Crear nueva rama
git checkout nueva-feature # Cambiar a la rama
git checkout -b mi-rama    # Crear y cambiar
git switch mi-rama         # Comando moderno para cambiar
git switch -c nueva-rama   # Crear y cambiar (moderno)
```

Fusionar ramas:

```
git checkout main          # Ir a rama principal
git merge nueva-feature    # Fusionar rama
git branch -d nueva-feature # Eliminar rama fusionada
git branch -D rama-forzar  # Eliminar forzosamente
```

Ejemplo de flujo con ramas:

```
git checkout -b feature/login-usuario
# Desarrollar funcionalidad de login
git add .
git commit -m "Implementar sistema de login"
git checkout main
git merge feature/login-usuario
git branch -d feature/login-usuario
```

9. Resolviendo Conflictos

Los conflictos ocurren cuando Git no puede fusionar automÆticamente los cambios porque diferentes ramas modificaron las mismas líneas.

Cómo se ve un conflicto:

```
function saludar(nombre) {
<<<<<<< HEAD
   console.log("Hola " + nombre + "!");
=======
   console.log(‘Hola ${nombre}!‘);
>>>>>>> nueva-rama
}
```

Pasos para resolver:

1. Abrir el archivo con conflicto
2. Editar manualmente para elegir quØ cambios mantener
3. Eliminar marcadores de conflicto (<<<<<<<, =======, >>>>>>>)
4. Aæadir el archivo resuelto al staging area
5. Hacer commit para completar la fusión

```
git add archivo-resuelto.js
git commit -m "Resolver conflicto en función saludar"
```

10. Trabajando con Repositorios Remotos

Clonar un repositorio:

```
git clone https://github.com/usuario/proyecto.git
git clone https://github.com/usuario/proyecto.git mi-carpeta
```

Ver repositorios remotos:

```
git remote -v               # Ver remotos configurados
git remote add origin https://github.com/usuario/proyecto.git
git remote rename origin nuevo-nombre
git remote remove nombre-remoto
```

Sincronizar con remoto:

```
git fetch origin            # Descargar cambios sin fusionar
git pull origin main        # Descargar y fusionar
git push origin main        # Subir cambios
git push -u origin main     # Configurar upstream
```

Trabajando con ramas remotas:

```
git checkout -b local-branch origin/remote-branch
git push origin nueva-feature
git push origin --delete rama-remota
```

11. Comandos œtiles para Corrección

Deshacer cambios:

```
git checkout -- archivo.txt # Descartar cambios no preparados
git reset HEAD archivo.txt  # Quitar del staging area
git reset --soft HEAD~1     # Deshacer commit, mantener cambios
```

12. GitHub: Colaboración y Hosting

GitHub es una plataforma de hosting para repositorios Git que facilita la colaboración y el desarrollo de software.

Características Principales:

- 2 Hosting gratuito para repositorios pœblicos
- 2 Herramientas de colaboración: Issues, Pull Requests
- 2 Integración con CI/CD
- 2 GitHub Pages para hosting de sitios web
- 2 Control de acceso y equipos

13. Pull Requests

Los Pull Requests permiten proponer cambios y solicitar revisión antes de fusionar código en la rama principal.

Flujo de trabajo con Pull Requests:

1. Crear rama para nueva funcionalidad
2. Desarrollar y hacer commits en la rama
3. Push de la rama al repositorio remoto
4. Crear Pull Request en GitHub
5. Revisión de código por parte del equipo
6. Discusión y mejoras
7. Fusión del Pull Request

```
# Flujo completo:
git checkout -b feature/nueva-funcionalidad
git add .
git commit -m "Implementar nueva funcionalidad"
git push origin feature/nueva-funcionalidad
# Crear PR en GitHub
# DespuØs de merge, limpiar:
git checkout main
git pull origin main
git branch -d feature/nueva-funcionalidad
```

14. Issues y Project Management

Issues (Problemas):

- 2 Reporte de bugs
- 2 Solicitud de nuevas características
- 2 Discusiones sobre el proyecto
- 2 Etiquetas para organización
- 2 Asignación a desarrolladores

Referencias en Commits:

```
git commit -m "Corregir bug de login - fixes #42"
git commit -m "Mejora performance - closes #15"
git commit -m "Avance en nueva feature - refs #23"
```

GitHub Projects:

15. Flujos de Trabajo Profesionales

Git Flow

Modelo de ramificación para proyectos con releases planificados:

2 main: Código en producción
2 develop: Integración de nuevas características
2 feature/*: Desarrollo de funcionalidades
2 release/*: Preparación de nuevas versiones
2 hotfix/*: Correcciones urgentes en producción

GitHub Flow

Flujo simplificado para desarrollo continuo:

1. Crear rama desde main
2. Desarrollar funcionalidad
3. Abrir Pull Request
4. Revisar y discutir
5. Deploy para testing
6. Merge a main
7. Deploy a producción

16. Mejores PrÆcticas

Mensajes de Commit:

2 Usar modo imperativo: "Agregar" no "Agregado"
2 Primera línea: resumen de 50 caracteres mÆx.
2 Segunda línea en blanco
2 Explicación detallada si es necesario

feat: agregar autenticación con JWT

Implementa sistema completo de autenticación incluyendo:
- Login con email y contraseæa
- Tokens JWT con expiración
- Middleware de autorización

Estructura de Ramas:

2 Nombres descriptivos: feature/user-authentication
2 Prefijos consistentes: feature/, bugfix/, hotfix/
2 Eliminar ramas despuØs de merge
2 Mantener main/develop siempre estables

Archivos .gitignore:

```
# Dependencias
node_modules/
# Archivos de compilación
dist/
build/
# Variables de entorno
.env
.env.local
```

17. Herramientas y Extensiones

Clientes GrÆficos:

2 GitHub Desktop: Interfaz simple y amigable
2 SourceTree: Herramienta avanzada gratuita
2 GitKraken: Cliente profesional con características premium
2 VS Code: Integración nativa de Git

Comandos Avanzados œtiles:

```
git stash                  # Guardar cambios temporalmente
git stash pop              # Recuperar cambios guardados
git cherry-pick <commit>   # Aplicar commit específico
git rebase -i HEAD~3       # Rebase interactivo
git reflog                 # Historial de referencias
git bisect start           # Buscar commit problemÆtico
```

Aliases œtiles:

```
git config --global alias.st status
git config --global alias.co checkout
git config --global alias.br branch
git config --global alias.ci commit
git config --global alias.unstage 'reset HEAD --'
```

18. Integración con CI/CD

GitHub Actions permite automatizar flujos de trabajo:

```
name: CI/CD Pipeline
on:
  push:
    branches: [ main, develop ]
  pull_request:
    branches: [ main ]
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v2
    - name: Setup Node.js
      uses: actions/setup-node@v2
      with:
        node-version: '16'
    - run: npm install
    - run: npm test
```

19. Resumen y Próximos Pasos

Has aprendido los fundamentos de Git y GitHub:
2 Control de versiones distribuido
2 Comandos esenciales de Git
2 Trabajo con ramas y fusiones
2 Colaboración en GitHub
2 Flujos de trabajo profesionales

Preparación para el Tema 8:

En el próximo tema integraremos todo lo aprendido: