Instructivo de los 10 Tipos de Conventional Commits versionando una Página Web

1) **Conventional Commits** es una especificación para escribir *mensajes de commit estructurados y consistentes* en proyectos gestionados con Git. Esto mejora la legibilidad del historial, facilita la colaboración en equipo y permite automatizar tareas como la generación de changelogs o versiones. Estructura básica:

```
<tipo>(<ámbito>): <descripción corta>
[opcional cuerpo del mensaje]
[opcional pie de página]
```

- **Tipo**: Indica el propósito del commit (Ejem: `feat`, `fix`, `docs`).
- **Ámbito** (opcional): Especifica el módulo o componente afectado (e.g., `auth`, `api`, `config`).
- **Descripción corta**: Resumen breve (máximo 50 caracteres).
- **Cuerpo** (opcional): Detalles del cambio, explicando el **qué** y **por qué**.
- **Pie de página** (opcional): Referencias a issues o notas (e.g., `Closes #123`).

2) **Cómo funciona la estructura**

Cada commit sigue un formato estándar:

- **Primera línea**: `<tipo>(<ámbito>): <descripción corta>`. Ejemplo: `feat(tasks-api): add endpoint to create tasks` indica que añadimos un endpoint.
- **Cuerpo**: Explica detalles. Por ejemplo, en `fix(task-service)`, describimos cómo asociamos tareas al usuario autenticado.
- **Pie de página**: Conecta con issues (Ejem:, `Closes #15`) o agrega notas.

3) **Beneficios**

- Historial de Git claro y organizado, esencial para proyectos Spring Boot con múltiples capas.
- Facilita la colaboración en equipos, especialmente en proyectos con controladores, servicios y entidades.
- Automatiza tareas como generar changelogs o releases con herramientas como **Semantic Release**.
- Mejora la trazabilidad, útil para debugging y auditorías en aplicaciones empresariales.

4) **Los 10 Tipos Más Útiles** de Conventional Commits para versionar el código fuente de en un sitio web con HTML, Tailwind CSS, JavaScript y Python (Flask):

- `feat`: Nueva funcionalidad para el usuario.
- `fix`: Corrección de errores.
- `docs`: Cambios en documentación.
- `style`: Cambios de formato o estilo (sin afectar la lógica).
- `refactor`: Reestructuración de código sin cambiar funcionalidad, reducir líneas de código, mejorar, etc.
- `test`: Agregar o modificar pruebas.
- `chore`: Tareas de mantenimiento (e.g., actualizar dependencias).
- `build`: Cambios en el sistema de compilación (e.g., Maven/Gradle).
- `ci`: Cambios en la configuración de integración continua.
- `perf`: Mejoras de rendimiento.

5) Ejemplo con un Código Fuente Real

El proyecto es un sitio web de lista de tareas (To-Do) con un frontend en **HTML**, **Tailwind CSS** y **JavaScript**, y un backend en **Python** con ***Flask*** y una base de datos SQLite. Los usuarios pueden agregar, listar y eliminar tareas.

Estructura del Proyecto

```
project/
├── static/
│   ├── css/
│   │   ├── styles.css  # Tailwind CSS compilado
│   ├── js/
│   │   ├── app.js      # Lógica del frontend
├── templates/
│   ├── index.html      # Página principal
├── tests/
│   ├── test_api.py     # Pruebas para el backend
├── app.py              # Backend Flask
├── requirements.txt    # Dependencias de Python
├── README.md
├── .github/
│   ├── workflows/
│   │   ├── ci.yml     # Configuración de CI
├── package.json        # Para Tailwind CSS y herramientas
frontend
├── tailwind.config.js # Configuración de Tailwind
├── .gitignore
```

6) Ejemplos de los 10 Tipos de Commits

1. **`feat`** ***(nueva funcionalidad)***
   Escenario: Añadimos un formulario en el frontend para crear tareas.
   Cambio en el código:
   Archivo: templates/index.html

```
html
<!-- Antes -->
<div class="container mx-auto p-4">
  <h1 class="text-2xl font-bold">To-Do List</h1>
</div>

<!-- Después -->
<div class="container mx-auto p-4">
  <h1 class="text-2xl font-bold">To-Do List</h1>
+ <form id="task-form" class="mt-4">
+   <input type="text" id="task-input" class="border p-2" placeholder="Nueva tarea">
+   <button type="submit" class="bg-blue-500 text-white p-2">Agregar</button>
+ </form>
</div>
```

Archivo: static/js/app.js

```
javascript
+document.getElementById('task-form').addEventListener('submit', async (e) => {
+    e.preventDefault();
+    const title = document.getElementById('task-input').value;
+    if (title) {
+        await fetch('/api/tasks', {
+            method: 'POST',
+            headers: { 'Content-Type': 'application/json' },
+            body: JSON.stringify({ title })
+        });
+        document.getElementById('task-input').value = '';
+    }
+});
```

Mensaje de Commit:

```
feat(frontend): add task creation form
Implement form in index.html with Tailwind CSS styling.
Add JavaScript to send POST request to /api/tasks.
Closes #5
```

Este commit añade un formulario en el frontend para que los usuarios creen tareas. El cambio incluye HTML con Tailwind CSS para el diseño y JavaScript para enviar la tarea al backend. El ámbito `frontend` *indica que el cambio está en la interfaz, y el cuerpo explica los componentes añadidos. El pie de página cierra el issue #5.*

2. `fix` ***(corrección de errores)***

Escenario: Corregimos un error en el backend que no validaba el título de la tarea.
Cambio en el código:
Archivo: app.py

```
python
@app.route('/api/tasks', methods=['POST'])
def create_task():
    data = request.get_json()
-   title = data.get('title')
+   title = data.get('title')
+   if not title or title.strip() == '':
+       return jsonify({'error': 'Title is required'}), 400
    task = {'id': len(tasks) + 1, 'title': title}
    tasks.append(task)
    return jsonify(task), 201
```

Mensaje de Commit:

```
fix(tasks-api): validate empty title in task creation
Add validation to prevent empty task titles in POST /api/tasks.
Return 400 error if title is missing or empty.
Closes #8
```

Corregimos un error en el backend que permitía crear tareas con títulos vacíos, lo que podía causar problemas en la base de datos. El commit `fix` *soluciona errores funcionales. El ámbito*

`tasks-api` *apunta al endpoint, y el cuerpo detalla la validación añadida. El pie cierra el issue #8.*

3. `docs` ***(Documentación)***
Escenario: Actualizamos el README.md con instrucciones de instalación.
Cambio en el código:
Archivo: README.md

```markdown
# To-Do List App

## Installation
+1. Clone the repository: `git clone https://github.com/user/todo-app.git`
+2. Install Python dependencies: `pip install -r requirements.txt`
+3. Install frontend dependencies: `npm install`
+4. Build Tailwind CSS: `npm run build`
+5. Run the app: `python app.py`
```

Mensaje de Commit:

```
docs(readme): add installation and setup instructions
Provide steps to clone, install dependencies, and run the application.
Closes #100
```

Actualizamos el `README.md` *con instrucciones claras para instalar y ejecutar el proyecto. Los commits* `docs` *son para cambios en documentación que no afectan al código. El ámbito* `readme` *especifica el archivo, y el mensaje es breve porque el cambio es simple. El pie cierra el issue #100.*

4. `style` ***(Cambios de Estilo)***
Escenario: Mejoramos el formato de los estilos en styles.css para consistencia.
Cambio en el código:
Archivo: static/css/styles.css

```css
-/* Antes */
-.container { margin: 0 auto; padding: 1rem; }
-.task-item { padding: 0.5rem; border-bottom: 1px solid #ccc;}
+/* Después */
+.container {
+    margin: 0 auto;
+    padding: 1rem;
+}
+.task-item {
+    padding: 0.5rem;
+    border-bottom: 1px solid #ccc;
+}
```

Mensaje de Commit:

```
style(frontend): format CSS for consistency
Apply consistent spacing and formatting in styles.css.
Closes #150
```

Mejoramos el formato de los estilos CSS generados por Tailwind para seguir convenciones de escritura. Los

commits `style` no cambian la lógica, solo la presentación. El ámbito `frontend` indica que el cambio está en los estilos del frontend. El pie cierra el issue #150

5. `refactor` ***(Refactorización)***

Escenario: eorganizamos el código JavaScript para separar la lógica de envío de formularios.
Cambio en el código:
Archivo: static/js/app.js

```javascript
-document.getElementById('task-form').addEventListener('submit', async (e) => {
-    e.preventDefault();
-    const title = document.getElementById('task-input').value;
-    if (title) {
-        await fetch('/api/tasks', {
-            method: 'POST',
-            headers: { 'Content-Type': 'application/json' },
-            body: JSON.stringify({ title })
-        });
-        document.getElementById('task-input').value = '';
-    }
-});
+const createTask = async (title) => {
+    if (!title) return;
+    await fetch('/api/tasks', {
+        method: 'POST',
+        headers: { 'Content-Type': 'application/json' },
+        body: JSON.stringify({ title })
+    });
+};
+
+document.getElementById('task-form').addEventListener('submit', async (e) => {
+    e.preventDefault();
+    const title = document.getElementById('task-input').value;
+    await createTask(title);
+    document.getElementById('task-input').value = '';
+});
```

Mensaje de Commit:

```
refactor(frontend): extract task creation logic
Move task creation to a separate function for better maintainability.
Closes #50
```

Reorganizamos el código JavaScript para separar la lógica de creación de tareas en una función independiente, mejorando la mantenibilidad. Los commits `refactor` mejoran el código sin alterar su comportamiento. El ámbito `frontend` apunta al JavaScript del frontend. El pie cierra el issue #50

6. `test` ***(Pruebas)***

Escenario: Añadimos pruebas unitarias para el endpoint de creación de tareas.
Cambio en el código:
Archivo: tests/test_api.py

```python
```

```
+import unittest
+from app import app
+
+class TestTasksAPI(unittest.TestCase):
+    def setUp(self):
+        self.app = app.test_client()
+
+    def test_create_task(self):
+        response = self.app.post('/api/tasks', json={'title': 'Test task'})
+        self.assertEqual(response.status_code, 201)
+        self.assertEqual(response.json['title'], 'Test task')
+
+if __name__ == '__main__':
+    unittest.main()
```

Mensaje de Commit:

```
test(tasks-api): add unit tests for task creation
Test POST /api/tasks endpoint for valid task creation.
Verifies status code and response data.
```

Añadimos pruebas unitarias para el endpoint de creación de tareas en el backend. Los commits `test` son para pruebas nuevas o modificadas. El ámbito `tasks-api` indica que las pruebas están en la API, y el cuerpo detalla qué se prueba. El pie describe el estado de la prueba.

7. `chore` *(Mantenimiento)*
Escenario: Actualizamos Flask a una nueva versión.
Cambio en el código:
Archivo: requirements.txt

```txt
-Flask==2.0.1
+Flask==2.3.2
```

Mensaje de Commit:

```
chore(deps): upgrade Flask to version 2.3.2
Update Flask dependency to the latest stable version.
Verify Flask version.
```

Actualizamos Flask a una nueva versión en `requirements.txt`. Los commits `chore` manejan tareas de mantenimiento, como actualizar dependencias. El ámbito `deps` indica que es una tarea de dependencias. El pie indica verificar que se tiene la versión correcta.

8. `build` ***(Sistema de Compilación)***
Escenario: Añadimos un script para compilar Tailwind CSS.
Cambio en el código:
Archivo: package.json

```json
{
  "scripts": {
+    "build": "npx tailwindcss -i ./static/css/input.css -o
```

```
./static/css/styles.css --minify"
  }
}
```

Mensaje de Commit:

```
build(frontend): add Tailwind CSS build script
Add npm script to compile and minify Tailwind CSS.
Verify Tailwind version.
```

Añadimos un script en `package.json` para compilar Tailwind CSS con minificación. Los commits `build` afectan el sistema de compilación del frontend. El ámbito `frontend` especifica la parte afectada. El pie indica verificar que se tiene la versión deseada de Tailwind.

9. `ci` ***(Integración Continua)***
Escenario: Configuramos GitHub Actions para pruebas automáticas.
Cambio en el código:
Archivo: .github/workflows/ci.yml

```yaml
+name: CI
+on: [push]
+jobs:
+  test:
+    runs-on: ubuntu-latest
+    steps:
+      - uses: actions/checkout@v3
+      - name: Set up Python
+        uses: actions/setup-python@v4
+        with:
+          python-version: '3.9'
+      - name: Install dependencies
+        run: pip install -r requirements.txt
+      - name: Run tests
+        run: python -m unittest discover tests
```

Mensaje de Commit:

```
ci(github): add GitHub Actions workflow for tests
Configure CI to run Python tests on push events with Python 3.9.
Closes pipeline.
```

Configuramos GitHub Actions para ejecutar pruebas de Python automáticamente. Los commits `ci` son para cambios en integración continua. El ámbito `github` indica la plataforma, y el cuerpo detalla la configuración. El pie indica que se debe cerrar el pipeline.

10. `perf` ***(Mejoras de Rendimiento)***
Escenario: Optimizamos la carga de tareas en el frontend usando lazy loading.
Cambio en el código:
Archivo: static/js/app.js

```java
+const loadTasks = async () => {
+   const response = await fetch('/api/tasks');
```

```
+  const tasks = await response.json();
+  const taskList = document.getElementById('task-list');
+  tasks.forEach(task => {
+    const li = document.createElement('li');
+    li.textContent = task.title;
+    taskList.appendChild(li);
+  });
+};
+
+document.addEventListener('DOMContentLoaded', () => {
+  const observer = new IntersectionObserver((entries) => {
+    if (entries[0].isIntersecting) {
+      loadTasks();
+      observer.disconnect();
+    }
+  });
+  observer.observe(document.getElementById('task-list'));
+});
```

Mensaje de Commit:

```
perf(frontend): implement lazy loading for task list
Use IntersectionObserver to load tasks only when visible.
Improves initial page load performance.
```

Implementamos lazy loading en el frontend para cargar la lista de tareas solo cuando es visible, mejorando el rendimiento. Los commits `perf` optimizan el rendimiento. El ámbito `frontend` apunta al JavaScript del frontend. El pie indica validar la mejora en la carga de la página web.

7) Consejos para Estudiantes

- **Practica en proyectos simples:** Usa Conventional Commits en tus proyectos web, como una lista de tareas. Por ejemplo, usa `feat` para un nuevo formulario o `test` para pruebas.
- **Usa herramientas**:
  - Instala **Commitizen** (`npm install -g commitizen`) y usa `git-cz` para escribir commits.
  - Instala **Husky** para validar commits automáticamente.
  - **Maven Release Plugin** o **Semantic Release**: Automatiza versiones.
  - Usa **Semantic Release** para automatizar changelogs.
- **Revisa el historial**: Usa `git log --oneline` para ver cómo los commits estructurados hacen el historial claro.
- **Colabora con reglas**: En proyectos grupales, acuerda con tu equipo usar estos 10 tipos.
- **Sé breve**: La primera línea debe ser menor a 50 caracteres.
- **Explora proyectos reales**: Mira repositorios como `flask/flask` en GitHub para aprender cómo usan commits estructurados.

8) Recursos adicionales

- **Conventional Commits**: https://www.conventionalcommits.org/
- **Commitizen**: https://commitizen.github.io/cz-cli/
- **Husky**: https://typicode.github.io/husky/
- **Semantic Release**: https://semantic-release.gitbook.io/
- **Flask Projects**: https://github.com/pallets/flask