1000 Programadorxs

>Introducción a la Programación con Python 2023

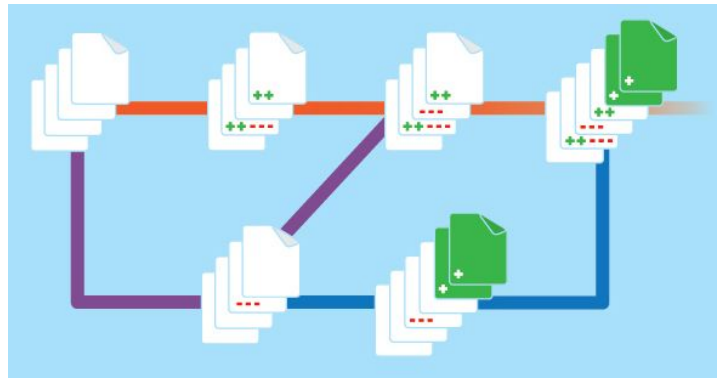
academia CIMNE IBER | 1000 PROGRAMADORXS Salta | Ministerio de Economía y Servicios Públicos Gobierno de Salta | GOBIERNO DE SALTA

módulo 5

>Control de Versiones

academia CIMNE IBER | 1000 PROGRAMADORXS Salta | Ministerio de Economía y Servicios Públicos Gobierno de Salta | GOBIERNO DE SALTA

Introducción



Hasta este punto, hemos visto los **contenidos esenciales** para desarrollar programas en Python. El siguiente paso es adoptar un **control de versiones**.

Esta práctica esencial nos permite **rastrear** y **gestionar** cambios en el código, nos brinda un **historial** detallado, **colaboración** eficiente, y mejorar nuestro **flujo de trabajo** en proyectos individuales y en equipo.

Es el siguiente paso lógico para el desarrollo.

Control de Versiones

El control de versiones es una práctica fundamental en el desarrollo de software que permite rastrear y gestionar los cambios en el código. Proporciona las siguientes ventajas:

- **Registro de modificaciones**: Mantiene un historial de cambios del código, facilitando la comparación con versiones anteriores y la corrección de errores.
- **Protección del código**: Evita la pérdida o corrupción de código, ya que se puede restaurar una versión anterior del código.
- **Facilita el trabajo en equipo**: Permite a los desarrolladores trabajar de manera concurrente, ya que se gestiona de manera eficiente la combinación de cambios realizados por diferentes personas.

Control de Versiones

El control de versiones es una práctica fundamental en el desarrollo de software que permite rastrear y gestionar los cambios en el código. Proporciona las siguientes ventajas:

- **Mejora la productividad**: Al brindar un flujo de trabajo estructurado y la capacidad de revertir cambios problemáticos, el control de versiones reduce el tiempo de desarrollo y el trabajo para corregir problemas.
- **Soporte para multiplataforma**: Los sistemas de control de versiones son compatibles con diversas plataformas y entornos de desarrollo, permitiendo una integración sin problemas en diferentes proyectos y equipos.

Git

Git es un sistema de control de versiones distribuido y de código abierto. Fue creado por Linus Torvalds en 2005 para gestionar el desarrollo del kernel de Linux.

El principal propósito de Git es rastrear los cambios en los archivos de un proyecto y coordinar el trabajo que varias personas realizan sobre los mismos archivos.

Git - Rendimiento

Git destaca por su rendimiento. Las operaciones más comunes como confirmar cambios o revisar el historial, se realizan en local, lo que significa que no se necesita conectividad a internet para trabajar.

Git es altamente eficiente en la gestión de grandes proyectos debido a su arquitectura distribuida, lo que lo hace más rápido en comparación con otros sistemas de control de versiones.

Git - Seguridad

En términos de seguridad, Git utiliza un modelo de seguridad de acceso directo. Esto significa que solo las personas autorizadas pueden realizar cambios en los archivos del proyecto.

Además, Git utiliza un mecanismo de hashing criptográfico llamado SHA1 para asegurar la integridad y consistencia de los datos.

Git - Flexibilidad

Git ofrece una gran **flexibilidad**. Permite tener múltiples **ramas de trabajo** que pueden ser **fusionadas** en cualquier momento.

También se pueden crear etiquetas para **puntos específicos** de la historia de un proyecto, lo que facilita el seguimiento de versiones específicas.

Git - Control de Versiones

El control de versiones con Git implica el seguimiento de los cambios en el código fuente durante el tiempo. Cuando se guarda un cambio, Git crea una instantánea del código y un puntero hacia esa instantánea.

Los desarrolladores pueden revisar y revertir a versiones anteriores del código fácilmente. Además, el control de versiones con Git permite a los equipos de desarrollo trabajar de forma concurrente, ayudando a evitar conflictos y a gestionar el código de forma eficaz.

Herramientas que Suma Git

El cambio de un **sistema de control** de versiones centralizado a Git cambia la forma en la que un **equipo de desarrollo** crea software. A continuación introduciremos las herramientas y **metodologías** más importantes que nos aporta Git.

Flujo de Trabajo en Ramas

El flujo de trabajo de ramas en Git permite a los desarrolladores crear, cambiar y fusionar ramas con facilidad. Cada rama es una copia del proyecto en un punto determinado y sirve como un entorno aislado para el desarrollo.

Esto permite a los equipos trabajar en características o arreglos separados sin interferir con el código principal hasta que estén listos para ser integrados.

Desarrollo Distribuido

Git es un sistema de control de versiones **distribuido**. Esto significa que cada desarrollador tiene una **copia completa** del repositorio en su máquina local, incluyendo el **historial completo** de cambios.

Esta característica facilita el desarrollo **colaborativo**, ya que los cambios pueden ser **sincronizados** entre las distintas copias del repositorio.

Pull Requests

Los **pull requests** son una funcionalidad de las plataformas de alojamiento de Git como GitHub, que facilitan la colaboración y **revisión de código**. Un pull request es una **propuesta de cambios** que un desarrollador hace a un repositorio.

Otros miembros del equipo pueden revisar, discutir y sugerir **modificaciones** a estos cambios antes de ser **integrados** al código principal.

Instalación de Git

Para empezar a trabajar con Git, debemos instalarlo en nuestra computadora. Simplemente visitamos el siguiente link y seguimos los pasos de la instalación:

Link: https://git-scm.com/download/win

En la página hacemos clic en Click here to download.

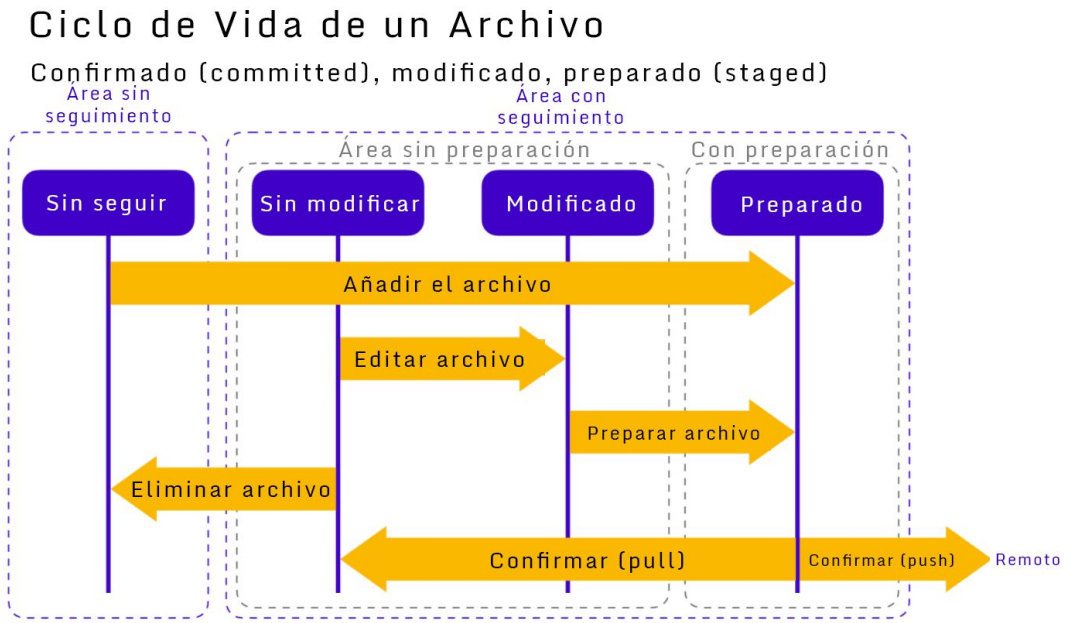
Para las diapositivas posteriores, en el explorador hacemos clic derecho a una carpeta y seleccionamos la opción Git Bash Here. Esta opción nos abrirá una línea de comandos para trabajar con Git en este directorio.

Ciclo de Vida de un Archivo

Antes de empezar a trabajar en Git debemos tener en claro las **etapas** que atraviesa cada archivo dentro de un proyecto.

Para trabajar con git en un repositorio, una vez instalado el software, vamos a la **línea de comandos** y nos dirigimos a la carpeta donde trabajaremos.

Allí lo primero que escribimos será `git init`



Ciclo de Vida de un Archivo

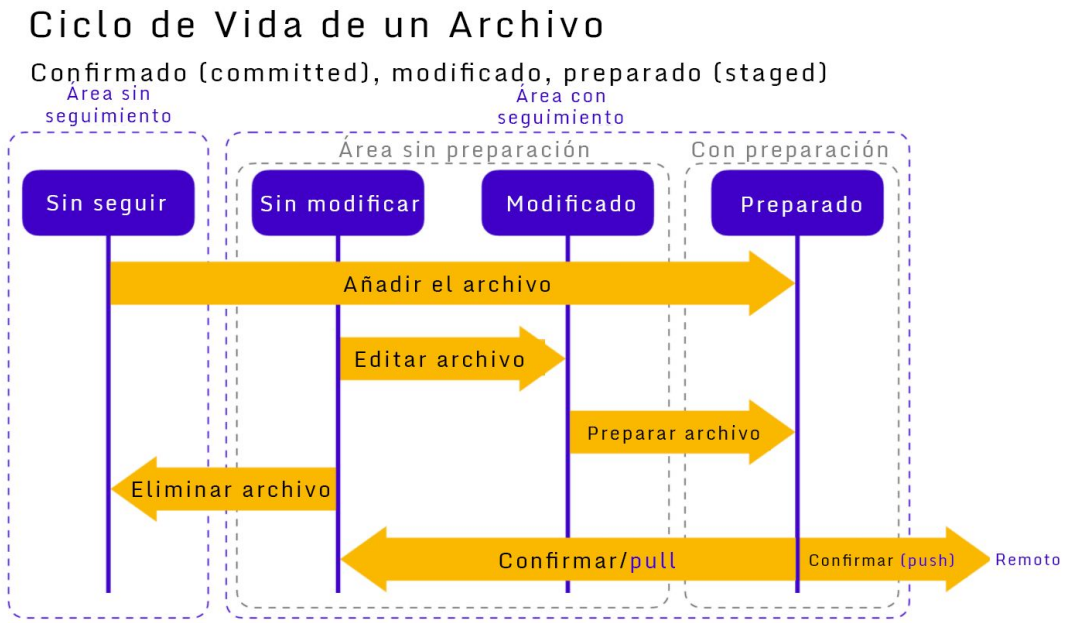
Ahora estamos en nuestro directorio con git habilitado y vemos que aparece **main**.

Viendo el gráfico nos encontramos en la **primera etapa**.

La acción de **añadir** archivos se considera tanto al crear, copiar en el directorio, o modificar archivos existentes.

Para cualquiera de los casos el comando es:

`git add nombre-archivo` o, `git add .` si queremos agregar **todos** los archivos.



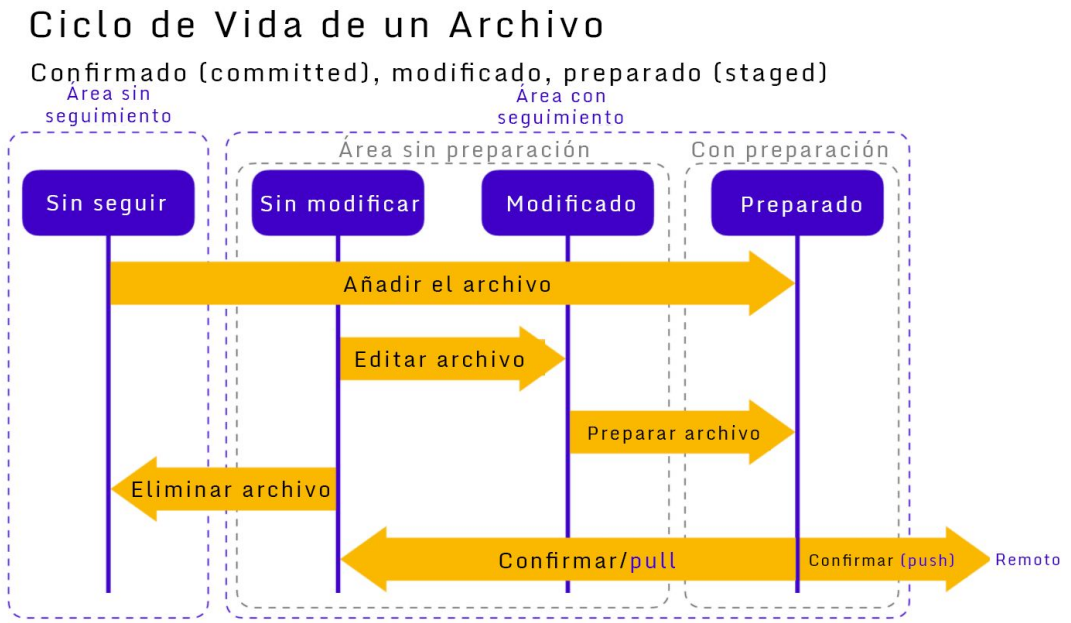
Ciclo de Vida de un Archivo

Si agregamos archivos nuevos o modificamos existentes podemos ver el **estado** del repositorio con el comando `git status`.

Aquí veremos mensajes sobre los cambios en dos tipos:

- Cambios no preparados (unstaged)
- Archivos sin seguir (unstaged)

Para pasar al **staging**, tenemos que utilizar nuevos comandos.

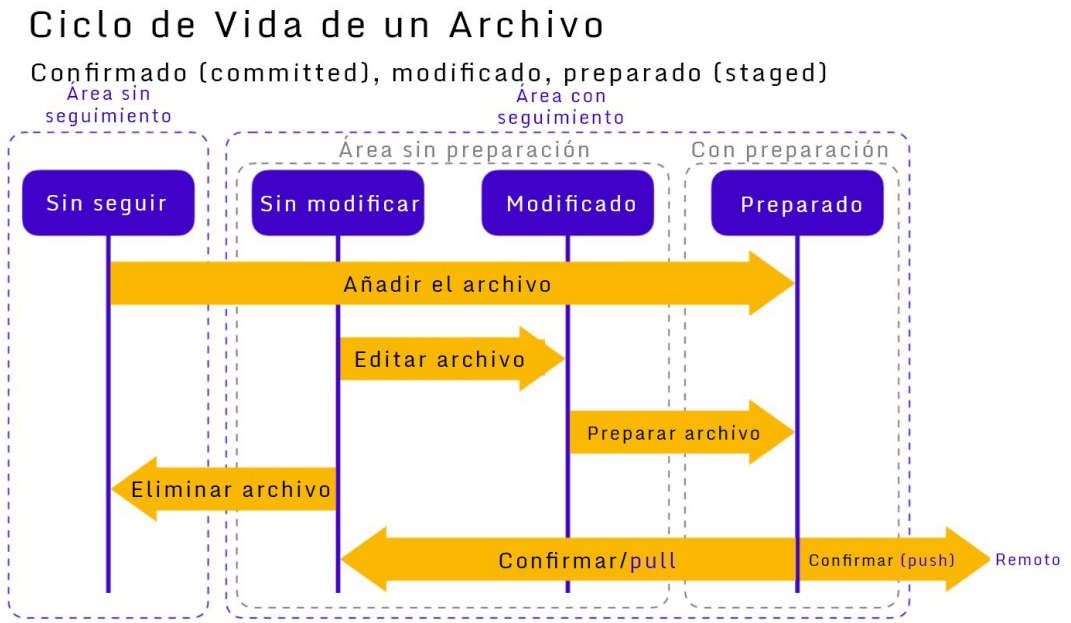


Ciclo de Vida de un Archivo

Cuando ya tenemos todos nuestros archivos (nuevos, modificados) en staging con `git add`, el siguiente paso es **confirmar** esta modificación. Esto creará una instantánea (snapshot) y volveremos a las primeras dos zonas de nuestro diagrama.

Para esto usamos el comando:

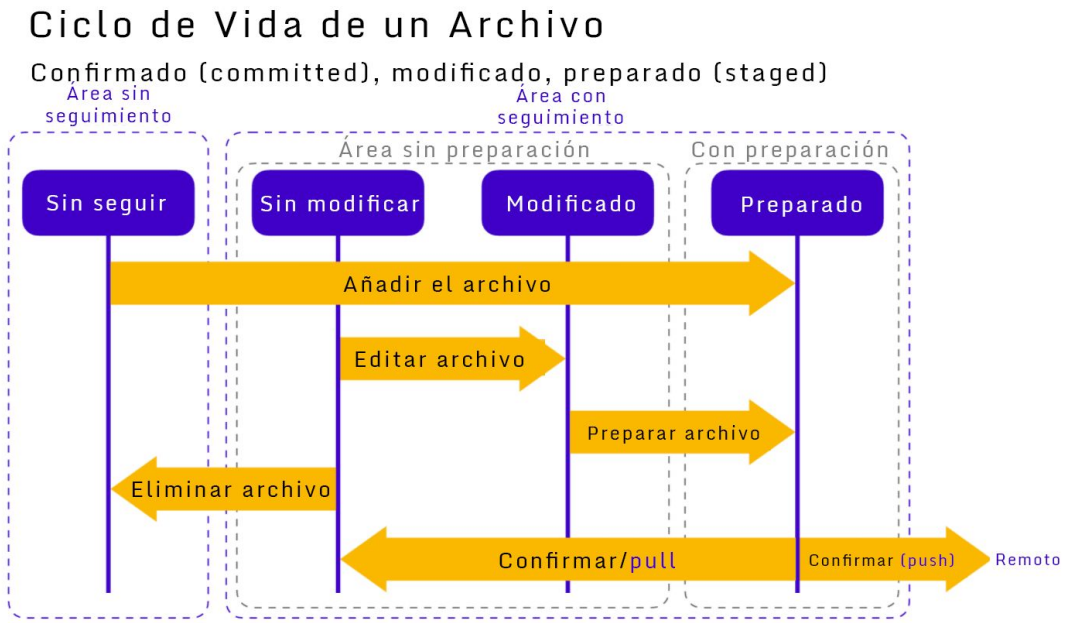
```
git commit -m “Mensaje relevante sobre los cambios”
```



Ciclo de Vida de un Archivo

Al correr `git status` nuevamente nos dirá que no hay nada que confirmar (commit), y volvemos al principio del ciclo de vida.

En el gráfico tenemos dos operaciones en púrpura. Estas se relacionan más con el trabajo con repositorios remotos, y las veremos en detalle en la siguiente clase.

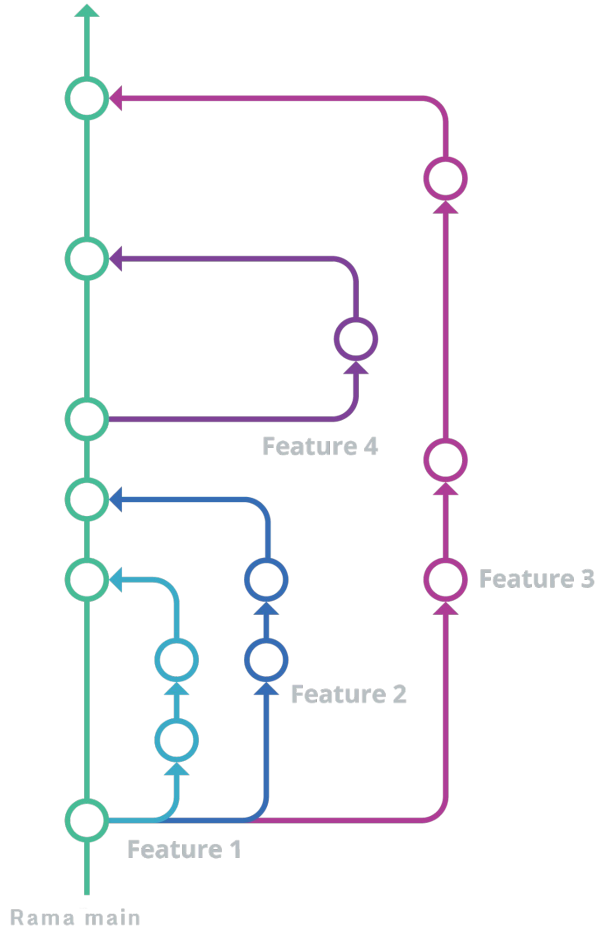


Introducción a Ramas

Una vez dominados los **conceptos básicos** del ciclo de vida de los archivos, podemos proceder con el **trabajo por ramas**.

Una rama es una línea de desarrollo independiente que nos permite trabajar en **diferentes versiones** de un proyecto de manera paralela.

Al crear un repositorio **nuevo** en Git, se crea **automáticamente** la rama **main** que representa la versión **principal** o **estable** del proyecto.



Creación de una Rama

En la consola vemos la rama actual luego del directorio de trabajo.
Para crear una nueva, escribimos `git checkout -b nombre-rama`. El nombre debe ser representativo de las tareas planificadas a trabajar en la misma.

Supongamos que trabajamos en algunos archivos en la rama **main** y consideramos esto como la **estructura base** del proyecto. Ahora deseamos desarrollar un modelo **Cliente** en una rama nueva. Los pasos a seguir serán:

1. En la rama main, escribir `git checkout -b feature/crear-cliente`.
2. Ahora volvemos al directorio y trabajamos en la **característica**.
3. Cuando terminemos de trabajar usamos los comandos **`add`** y **`commit`** en la rama.

Revisión de Ramas

Lo que tenemos actualmente es una rama main con la estructura base, y una rama feature/crear-cliente con código adicional del modelo Cliente.

Si usamos el comando `git branch` veremos una lista de nuestras ramas y en verde con un asterisco, la rama en la que estamos trabajando. Para regresar a la rama main (o cambiar a cualquier otra), usamos el comando git checkout main.

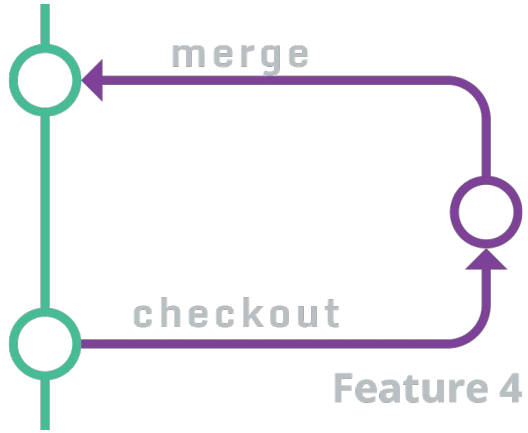
Si tenemos el explorador abierto, veremos cómo “desaparece” el archivo que trabajamos en la otra rama. Esto se debe a que los cambios están aislados por rama.

Fusión de Ramas

Si nos enfocamos en algunas de las ramas veremos el siguiente comportamiento. Ya vimos cómo creamos las ramas, ahora lo que nos interesa es, una vez completada la tarea deseamos llevar todos estos cambios a nuestra rama estable.

Para eso utilizamos `git merge`. El procedimiento es así:

1. Con `git checkout` regresamos a la rama base
2. Aquí hacemos git merge branch para traer los cambios de la rama branch a la rama base en la que estemos
3. Ahora si vemos el directorio, la rama base tendrá todos los cambios de la rama branch



Nombre para Ramas

Existen varias metodologías para nombrar ramas, aquí veremos las más sencillas de explicar su funcionamiento. Primero que nada debemos considerar lo siguiente:

1. **Uso de minúsculas**: Es preferible utilizarlas para mantener consistencia en el repositorio.
2. **Separar palabras con guiones**: Usaremos el guión - para separar las palabras del nombre de la rama para mejorar la legibilidad.
3. **Ser descriptivo**: El nombre de la rama debe ser lo suficientemente descriptivo para comprender su propósito sin necesidad de consultar más documentación.

Nombre para Ramas

Usando estas consideraciones, podemos usar **tres categorías** para nombrarlas:

- **`feature/modelo-cliente`**: Esta rama se puede usar si trabajamos en una **nueva característica** relacionada con el modelo de Cliente.
- **`bugfix/validacion-cliente`**: Si estamos solucionando un **error** relacionado con la **validación** de Cliente, este nombre puede dar esa información.
- **`refactor/reestructurar-modelo-cliente`**: Si estamos realizando una **reestructuración importante** en la implementación del modelo de Cliente, así podemos reflejar en el nombre.

Con estas tres categorías podemos hacer más **legible** nuestro trabajo.

muchas gracias.

academia CIMNE IBER | 1000 PROGRAMADORXS Salta | Ministerio de Economía y Servicios Públicos Gobierno de Salta | GOBIERNO DE SALTA