La fase de implementación

Dentro del ciclo de desarrollo de software, la fase de implementación es aquella en la que se le da forma a los diseños establecidos en la fase de análisis. Es decir, es la fase en la que se programa el sistema. Y para programar, los informáticos generalmente emplean muchas herramientas.

En esta unidad vamos a introducirnos a esas herramientas, concretamente veremos:

- Diferentes editores de código y entornos de desarrollo integrados, para escribir el código
- Compiladores, intérpretes y máquinas virtuales, para ejecutar el código
- Herramientas para el control de versiones, para controlar el progreso del código

> **Actividad**
> Reflexiona sobre la siguiente pregunta:
>
> ¿Por qué crees que en los equipos profesionales no se programa directamente en un bloc de notas o editor básico como el del móvil?
>
> Después, debate en la clase los siguientes puntos:
>
> - ¿Qué ventajas ofrecen los entornos de desarrollo modernos?
> - ¿Qué problemas pueden surgir si no usamos herramientas como control de versiones?
> - ¿Qué diferencias hay entre programar solo para uno mismo y hacerlo como parte de un equipo?
> - ¿Creéis que conocer muchas herramientas os hace mejores desarrolladores?

Entornos de Desarrollo Integrados (IDEs) y otros editores

Tipos de Editores de Código

Los editores de código se dividen en dos categorías principales: **editores de texto** y **IDEs**. Los editores de texto son más livianos y sencillos, pero no integran de forma nativa todas las herramientas necesarias para la ejecución de código. Los IDEs, en cambio, integran en un solo programa tanto el editor de texto, como las herramientas de compilación y depuración de programas. Sin embargo, son más pesados y menos versátiles precisamente debido a esto. A continuación, vamos a repasar una serie de editores e IDEs:

Editores de Texto

1. **Notepad++**
   - **Características**: Ligero, soporte para múltiples lenguajes, resaltado de sintaxis, y una interfaz simple.
   - **Limitaciones**: Funcionalidades básicas; carece de herramientas avanzadas como depuración.
   - **Licencia**: Gratuito y de código abierto.
2. **Visual Studio Code**

- **Características**: Extensible mediante plugins, integración con Git, terminal integrado y soporte para múltiples lenguajes. Con las extensiones adecuadas, se comporta casi como si fuera un IDE. Tanto es así que en Mac OSX el IDE Visual Studio for Mac ha sido descontinuado en favor de este editor, con la extensión C# Dev Kit.
- **Limitaciones**: Requiere configuración adicional para funciones avanzadas.
- **Licencia**: Gratuito.

3. **Sublime Text**
   - **Características**: Rápido, interfaz minimalista, soporte para múltiples lenguajes.
   - **Limitaciones**: Versión de pago después de un periodo de evaluación; algunas características son limitadas.
   - **Licencia**: Gratuito en versión de evaluación; pago para uso continuado.

IDEs

1. **IntelliJ IDEA**
   - **Características**: IDE potente para Java, con soporte para Kotlin, Groovy y otros lenguajes. Ofrece integración con sistemas de control de versiones, herramientas de refactorización, y una interfaz intuitiva.
   - **Limitaciones**: Requiere hardware medianamente potente.
   - **Licencia**: Community Edition es gratuito y de código abierto; Ultimate Edition es de pago, con licencia educativa disponible.
2. **Eclipse**
   - **Características**: IDE versátil principalmente para Java, con soporte para plugins que permiten el desarrollo en otros lenguajes. Ofrece herramientas de depuración y una gran comunidad.
   - **Limitaciones**: La interfaz no es anticuada y puede no ser todo lo eficiente que uno desearía.
   - **Licencia**: Gratuito y de código abierto.
3. **NetBeans**
   - **Características**: IDE oficial para Java SE, pero también admite PHP y C/C++. Es fácil de usar y configurar.
   - **Limitaciones**: Menos extensible que otros IDEs; a veces más lento en comparación con IntelliJ.
   - **Licencia**: Gratuito y de código abierto.
4. **Visual Studio**
   - **Características**: IDE completo para C#, .NET y desarrollo web, herramientas de depuración avanzadas, integración con Azure. Muy usado también para C++.
   - **Limitaciones**: Solo disponible en Windows;

- **Licencia**: Community Edition es gratuito; Professional y Enterprise son de pago, con licencias educativas disponibles.

5. **Godot** (Editor de Juegos)
    - **Características**: Herramientas para desarrollo de juegos 2D y 3D, scripting en GDScript, interfaz amigable.
    - **Limitaciones**: Menos soporte para lenguajes convencionales; no es un IDE generalista aunque se pueden realizar aplicaciones con él.
    - **Licencia**: Gratuito y de código abierto.
6. **Unreal Engine** (Editor de Juegos)
    - **Características**: Motor para desarrollo de juegos 3D, herramientas visuales de creación de contenido y programación con Blueprints. Sirve también para crear contenido audiovisual en 3D y ha sido usado en series como *The Mandalorian*.
    - **Limitaciones**: Requiere hardware potente; curva de aprendizaje pronunciada.
    - **Licencia**: Gratuito, con royalties aplicables a los ingresos por juegos. Se pueden comprar licencias alternativas.

Herramientas que Componen un IDE

Los IDEs están compuestos por varias herramientas que facilitan el desarrollo de software. A continuación se presentan ejemplos de herramientas comunes:

1. **Editor de Código**: Donde se escribe el código fuente. Ejemplos incluyen el editor de IntelliJ y el editor de Visual Studio.
2. **Compilador/Intérprete**: Convierte el código fuente en un formato ejecutable. Por ejemplo, el compilador de Java en IntelliJ o el compilador de C# en Visual Studio.
3. **Herramientas de Depuración**: Permiten encontrar y corregir errores en el código. Tanto IntelliJ como Visual Studio cuentan con potentes herramientas de depuración integradas.
4. **Gestión de Proyectos**: Facilita la organización de archivos y recursos. Los IDEs permiten gestionar múltiples proyectos desde una única interfaz.

Diferencias entre compiladores, intérpretes y máquinas virtuales

Compiladores: La Traducción Completa

Un **compilador** actúa como un traductor completo de un idioma a otro. Toma el código fuente escrito en un lenguaje de programación de alto nivel (como C++ o C) y lo convierte, de una sola vez,

en código máquina o en un formato intermedio que la computadora puede entender directamente. Este proceso de compilación genera un archivo ejecutable independiente.

Las ventajas de la compilación radican en la velocidad de ejecución: una vez compilado, el programa se ejecuta de forma muy eficiente. Sin embargo, cualquier cambio en el código fuente requiere una nueva compilación de todo el programa. Ejemplos clásicos incluyen el compilador GCC para C/C++ y el compilador de Java que genera bytecode.

Intérpretes: La Ejecución Línea a Línea

A diferencia de los compiladores, un **intérprete** no traduce el código fuente por completo antes de la ejecución. En su lugar, lee y ejecuta el código línea por línea, o instrucción por instrucción, en tiempo real. Esto significa que el código fuente se traduce y ejecuta simultáneamente.

Los intérpretes ofrecen mayor flexibilidad y son ideales para el desarrollo rápido, ya que los cambios en el código se pueden probar de inmediato sin una fase de compilación. Sin embargo, la ejecución suele ser más lenta que la de los programas compilados debido al proceso de traducción en cada ejecución. Python, JavaScript y Ruby son ejemplos prominentes de lenguajes que suelen ser interpretados.

Máquinas Virtuales: Un Entorno de Ejecución Abstracción

Las **máquinas virtuales (VM)** añaden otra capa de abstracción. No son ni compiladores ni intérpretes en sí mismas, sino un entorno de software que emula un sistema informático completo o una plataforma de hardware específica. Su objetivo es proporcionar un entorno aislado y consistente para la ejecución de programas, independientemente del hardware o sistema operativo subyacente.

En el contexto de la ejecución de código, muchas máquinas virtuales trabajan en conjunto con un compilador o un intérprete. Por ejemplo, la Máquina Virtual de Java (JVM) toma el bytecode compilado de Java y lo interpreta o lo compila "justo a tiempo" (JIT) en código máquina para su ejecución. Esto permite que los programas Java sean "escritos una vez, ejecutados en cualquier lugar". Otro ejemplo son las máquinas virtuales de sistemas operativos (como VMware o VirtualBox) que permiten ejecutar múltiples sistemas operativos en un solo equipo físico.

> **Actividad**
> Busca 3 lenguajes compilados (que no sean C y C++), tres lenguajes interpetados (que no sean python, Javascript ni Ruby) y otros lenguajes que funcionen con un sistema de máquina virtual similar al de Java.

> **Actividad**
> Busca qué significa el término *transpilación*

Características Comunes en los IDEs

IntelliSense

- **Descripción**: Autocompletado de código y sugerencias contextuales.
- **Ejemplo**: En **Visual Studio Code**, al escribir una función en JavaScript, el IDE sugiere automáticamente los parámetros y funciones disponibles. En **IntelliJ**, la funcionalidad es similar para Java y otros lenguajes, mejorando la productividad del desarrollador.
- En Visual Studio Code, Intellisense se instala por separado para cada lenguaje de programación.

Multicursor

- **Descripción**: Permite editar múltiples líneas de código simultáneamente.
- **Ejemplo**: En **Visual Studio Code**, puedes colocar cursores en diferentes líneas y editar a la vez, lo que acelera cambios repetitivos. IntelliJ también ofrece esta funcionalidad, facilitando la edición de múltiples instancias de una variable o función.

Ejecución Paso a Paso

- **Descripción**: Herramientas de depuración que permiten seguir el flujo del programa línea por línea.
- **Ejemplo**: En **IntelliJ**, puedes establecer puntos de ruptura y seguir la ejecución de tu código para identificar errores. **Visual Studio** ofrece una funcionalidad similar, permitiendo a los desarrolladores depurar aplicaciones complejas.

IDEs Especializados por Lenguaje

Los IDEs y editores se especializan en diferentes lenguajes de programación. A continuación, se presentan algunos ejemplos destacados:

1. **Java**:
    - **IntelliJ IDEA**: Community Edition es gratuito; Ultimate Edition de pago, pero dispone de gratuicidad con la versión educativa.
    - **Eclipse**: Gratuito y de código abierto.

> **Actividad**
> Instala IntelliJ IDEA y ejecuta un hola mundo.

2. **C/C++**:
    - **Visual Studio**: Community Edition es gratuito; versiones Professional y Enterprise son de pago con licencias educativas.
    - **Code::Blocks**: Gratuito y de código abierto.
    - **CLion**: IDE de Jetbrains, gratuito para uso no comercial.

> **Actividad**
> Instala CLION y ejecuta un hola mundo en C (usando printf) y otro en C++ (usando cout)

3. **C#**:
    - **Visual Studio**: Community Edition es gratuito; versiones de pago con licencias educativas.
    - **Visual Studio Code + C# Dev Kit**: Gratuito; ideal para desarrollo de aplicaciones C# multiplataforma.
    - **Rider**: IDE de Jetbrains, gratuito para uso no comercial

> **Actividad**
> Instala Rider y ejecuta un hola mundo.

4. **Python**:
    - **PyCharm**: Community Edition es gratuito; Professional Edition de pago con licencia educativa.
    - **Visual Studio Code**: Gratuito; soporte para Python mediante extensiones. Necesitas instalar Python 3.

> **Actividad**
> Instala Pycharm y también Python. Ejecuta el mismo hola mundo en Pycharm y en VS Code usando Python3.

5. **JavaScript**:
    - **Visual Studio Code**: Gratuito; muy popular para desarrollo web.
    - **WebStorm**: De pago, con licencia educativa disponible.
6. **SQL**:
    - **DataGrip**: De pago, con licencia educativa.
    - **DBeaver**: Gratuito y de código abierto.

Licencias Educativas de JetBrains

Para obtener la **licencia gratuita para estudiantes de JetBrains**, sigue estos pasos:

1. Visita el sitio web de JetBrains.
2. Selecciona el producto que deseas utilizar (IntelliJ, PyCharm, etc.).
3. Regístrate con tu cuenta de correo electrónico institucional (por ejemplo, **@alu.edu.gva.es**).
4. Completa el formulario de solicitud y sigue las instrucciones para activar tu licencia.

Puedes usar tu identidad digital (@alu.edu.gva.es) para crear una cuenta en JetBrains y aplicar para solicitar la licencia educativa. Dicha licencia es renovable mientras dure tu educación.
Ten en cuenta que tu identidad digital no será accesible cuando acabes tu etapa en el instituto, pero puedes transferir o añadir más correos electrónicos a tu cuenta para poder seguir accediendo a Jetbrains.

Cuando acaba tu periodo educativo, dispones de un descuento del 40% para comprar una licencia anual, tanto del pack completo de IDEs de Jetbrains como los IDEs en específico que más emplees. Esta licencia ya no tiene las limitaciones de las licencias educativas y la puedes usar para tu día a día.
Con la licencia anual, Jetbrains ofrece una licencia de reserva perpetua. Esta licencia te permite usar para toda la vida la versión del IDE más actual en el momento de la compra.

Aunque con los IDEs community edition o gratuitos sin uso comercial nos es suficiente en la mayoría de casos, el poder acceder a IntelliJ IDEA Ultimate de forma gratuita nos permite, por ejemplo, configurar proyectos con Jakarta EE.

> **Actividad**
> Solicita tu licencia educativa de JetBrains usando tu carnet de estudiante. Cuando lo tengas, instala desde su aplicación JetBrains Toolbox todos los IDEs que necesites, especialmente IntelliJ IDEA Ultimate Edition.

Servicios Educativos de Otras Plataformas

Además de JetBrains, los estudiantes pueden acceder a otros servicios gratuitos o con licencia educativa:

- **Microsoft Azure**: Acceso gratuito a servicios en la nube, ideal para aprender sobre desarrollo y administración de sistemas.
- **GitHub**: Licencias educativas que ofrecen herramientas premium de desarrollo colaborativo.
- **AWS Educate**: Acceso a recursos y herramientas de Amazon Web Services para aprender sobre computación en la nube.
- **Google Cloud Services**: Ofrece créditos gratuitos para estudiantes que deseen experimentar con servicios de Google.

Configuración de los entornos de desarrollo: Visual Studio Code

En este apartado, vamos a preparar **Visual Studio Code** para que sea un potente editor de código y, especialmente, para que nos sirva como una herramienta eficaz para generar documentación en **Markdown**, para programar en **Javascript**, **Typescript** y **Python**.

> **Actividad**
> A medida que sigas este tutorial, realiza una memoria de todos los pasos, con capturas de pantalla, de cada una de las configuraciones y procesos que se explican a continuación.

Visual Studio Code. Instalación y configuración

Lo primero es tener Visual Studio Code instalado en tu sistema.

1. Instalación de Visual Studio Code

1. **Descarga el instalador:** Abre tu navegador web y ve a la página oficial de Visual Studio Code: https://code.visualstudio.com/.
2. **Selecciona tu sistema operativo:** El sitio web detectará automáticamente tu sistema operativo (Windows, macOS, Linux). Haz clic en el botón de descarga correspondiente.

3. **Ejecuta el instalador:** Una vez descargado, abre el archivo.
    - **En Windows:** Sigue las instrucciones del asistente de instalación. Te recomiendo marcar la opción **"Agregar al PATH"** durante la instalación, ya que esto te permitirá abrir VS Code desde la línea de comandos en cualquier directorio.
    - **En macOS:** Arrastra el icono de la aplicación a tu carpeta de "Aplicaciones".
    - **En Linux:** Dependiendo de tu distribución, puedes usar el archivo `.deb` (Debian/Ubuntu) o `.rpm` (Fedora/RHEL), o seguir las instrucciones para tu gestor de paquetes.
4. **Inicia Visual Studio Code:** Una vez finalizada la instalación, abre la aplicación.

2. Primeros ajustes y bienvenida

Al abrir VS Code por primera vez, verás una pantalla de bienvenida. Puedes explorarla o cerrarla.

- **Idioma:** Si VS Code no está en tu idioma preferido, puedes instalar un **Language Pack**.
    i. Ve a la vista de **Extensiones** (icono de los cuadrados en la barra lateral izquierda, o atajo `Ctrl+Shift+X`).
    ii. Busca "Spanish Language Pack" e instálalo.
    iii. Reinicia VS Code cuando te lo pida.
- **Tema:** Puedes cambiar el tema visual de VS Code desde `Archivo > Preferencias > Tema de color` (o `Code > Preferencias > Tema de color` en macOS). Experimenta con los temas "Dark Modern", "Light Modern", o descarga otros temas desde el Marketplace de extensiones.

Sin embargo, es recomendable que trabajes con VS Code en inglés, ya que la mayoría de la documentación está en ese idioma.

3. Configuración de las extensiones para Markdown

Las extensiones son el corazón de la personalización y funcionalidad de VS Code. Para instalarlas, usa la vista de **Extensiones** y busca la que necesites.

Si realizaste la actividad de ampliación de la unidad de programación 2, probablemente ya tengas preparado Visual Studio Code para usar Markdown. Si no, sigue estos pasos:

- **Para la documentación en Markdown:**
    i. **Markdown All in One:**
        - En la barra de búsqueda de extensiones, escribe `Markdown All in One`.
        - Busca la extensión publicada por "Yuya Saito" y haz clic en **Instalar**.

- Esta extensión te dará atajos de teclado para formato, una tabla de contenidos automática, y previsualización mejorada.

ii. **Markdown Preview Enhanced:**

- En la barra de búsqueda de extensiones, escribe `Markdown Preview Enhanced`.
- Busca la extensión publicada por "WangKou" y haz clic en **Instalar**.
- Ofrece una previsualización aún más rica, con soporte para diagramas, matemáticas y más. Para activarla en un archivo Markdown, haz clic derecho y selecciona **"Open Preview"** o el icono de previsualización en la esquina superior derecha.

4. Configuración para Javascript en Visual Studio Code

VS Code tiene un excelente soporte para JavaScript. Aquí vemos cómo complementarlo.

4.1. Uso de la consola del navegador (Chrome)

Aunque esto no es una configuración de VS Code per se, la consola del navegador es indispensable para el desarrollo frontend con JavaScript.

1. **Abre el navegador Chrome.**
2. **Abre las Herramientas de Desarrollador:**
   - Haz clic derecho en cualquier parte de la página web y selecciona **"Inspeccionar"**.
   - O usa el atajo de teclado: `Ctrl+Shift+I` (Windows/Linux) o `Cmd+Option+I` (macOS).
3. **Navega a la pestaña "Console":** Aquí es donde verás los errores de JavaScript, los mensajes de `console.log()`, y podrás ejecutar código JavaScript directamente.
4. **Prueba un comando:** Escribe `console.log("Hola desde la consola!");` y presiona `Enter`. Verás el mensaje impreso.

4.2. Configuración y uso de Node.js

Node.js te permite ejecutar JavaScript fuera del navegador, en el servidor o para tareas de desarrollo.

1. **Instalación de Node.js:**
   - Ve a la página oficial de Node.js: https://nodejs.org/.
   - Descarga la **versión LTS (Long Term Support)**, que es la más estable y recomendada.
   - Ejecuta el instalador y sigue los pasos. Asegúrate de que las opciones de instalación de **npm (Node Package Manager)** y **agregar Node.js al PATH** estén seleccionadas.
2. **Verifica la instalación:**
   - Abre el **Terminal integrado de VS Code** (`Ctrl+Ñ` o `View > Terminal`).
   - Escribe `node -v` y presiona `Enter`. Debería mostrar la versión de Node.js instalada.
   - Escribe `npm -v` y presiona `Enter`. Debería mostrar la versión de npm instalada.

3. **Extensiones de VS Code para Node.js:**
    - **ESLint:** Para ayudarte a escribir código JavaScript sin errores y con un estilo consistente.
        - Instala la extensión `ESLint` (publicada por "Dirk Baeumer").
        - A menudo, necesitarás configurar un archivo `.eslintrc.js` en tu proyecto para definir las reglas.
    - **Prettier - Code formatter:** Formatea tu código automáticamente al guardarlo.
        - Instala la extensión `Prettier – Code formatter` (publicada por "Prettier").
        - Para que formatee al guardar, ve a `Archivo > Preferencias > Configuración` (o `Code > Preferencias > Configuración` en macOS), busca `format on save` y marca la casilla.
4. **Ejecutar un archivo Node.js en VS Code:**
    - Crea un archivo llamado `app.js` con el siguiente contenido:

```
console.log("¡Hola desde Node.js!");
```

    - En el **Terminal integrado de VS Code**, navega hasta el directorio donde guardaste `app.js` (p.ej., `cd mi_proyecto`).
    - Ejecuta el archivo con `node app.js`. Verás la salida en el terminal.

4.3. Configuración y uso de Deno (admite Typescript de forma nativa)

Deno es una alternativa moderna a Node.js, con soporte nativo para TypeScript.

1. **Instalación de Deno:**
    - Abre tu terminal (o la línea de comandos).
    - Sigue las instrucciones de instalación de la web oficial de Deno: https://deno.land/#installation. Sigue el getting started.
2. **Verifica la instalación:**
    - En tu terminal (o el Terminal integrado de VS Code), escribe `deno --version`. Debería mostrar la versión de Deno.
3. **Extensión de VS Code para Deno:**
    - Ve a la vista de **Extensiones** en VS Code.
    - Busca `Deno` y instala la extensión oficial (publicada por "denoland").
4. **Habilitar Deno para tu proyecto en VS Code:**
    - Abre la **Paleta de Comandos** (`Ctrl+Shift+P` o `Cmd+Shift+P`).
    - Escribe `Deno: Initialize Workspace` y selecciona la opción. Esto creará un archivo `.vscode/settings.json` o actualizará el existente en tu proyecto, configurando VS Code para usar Deno.
    - Verifica que el archivo `settings.json` contenga algo similar a:

```
{
    "deno.enable": true,
    "deno.lint": true,
    "deno.unstable": true,
    "[typescript]": {
        "editor.defaultFormatter": "denoland.vscode-deno"
    },
    "[typescriptreact]": {
        "editor.defaultFormatter": "denoland.vscode-deno"
    },
    "[javascript]": {
        "editor.defaultFormatter": "denoland.vscode-deno"
    },
    "[javascriptreact]": {
        "editor.defaultFormatter": "denoland.vscode-deno"
    }
}
```

5. **Ejecutar un archivo Deno (TypeScript) en VS Code:**
    - Crea un archivo llamado `main.ts` con el siguiente contenido (TypeScript):

```
function greet(name: string): void {
    console.log(`Hola, ${name} desde Deno!`);
}

greet("Mundo");
```

    - En el **Terminal integrado de VS Code**, navega hasta el directorio del archivo.
    - Ejecuta el archivo con `deno run main.ts` . Si es la primera vez, Deno puede pedirte permiso para acceder a la red o al sistema de archivos (si tu código lo requiere).

5. Python en VS Code

VS Code es un entorno extremadamente popular y potente para el desarrollo en Python.

5.1. Instalación de Python 3

Para trabajar con Python, primero debes tenerlo instalado en tu sistema.

1. **Descarga Python 3:**
    - Ve a la página oficial de descargas de Python: https://www.python.org/downloads/.
    - Descarga la última versión estable de **Python 3**.

- **Ejecuta el instalador.**
    - **MUY IMPORTANTE (Windows):** Durante la instalación, asegúrate de marcar la casilla **"Add Python to PATH"** (Agregar Python al PATH). Esto es fundamental para que puedas ejecutar Python desde cualquier ubicación en tu terminal.
    - Sigue los pasos restantes del instalador.

2. **Verifica la instalación:**
    - Abre el **Terminal integrado de VS Code** (`Ctrl+Ñ`).
    - Escribe `python --version` y presiona `Enter` . Debería mostrar la versión de Python 3 que acabas de instalar (p.ej., `Python 3.10.x`).
    - En algunos sistemas, puede que necesites usar `python3 --version` .

5.2. Extensiones interesantes de VS Code relacionadas con Python

Estas extensiones transformarán tu experiencia de desarrollo Python en VS Code.

1. **Python (Microsoft):** Esta es la extensión fundamental y obligatoria.
    - Ve a la vista de **Extensiones** (`Ctrl+Shift+X`).
    - Busca `Python` y busca la extensión publicada por **"Microsoft"**. Haz clic en **Instalar**.
    - Esta extensión proporciona IntelliSense, depuración, formateo, linting, soporte para entornos virtuales, y mucho más.
2. **Pylance (Microsoft):** Complementa y mejora enormemente la extensión Python.
    - Busca `Pylance` y haz clic en **Instalar**.
    - Ofrece un análisis de código estático más rápido, sugerencias de autocompletado más precisas y una mejor experiencia con el tipado de Python.
3. **Jupyter:** Si trabajas con ciencia de datos, machine learning o notebooks, esta extensión es crucial.
    - Busca `Jupyter` y haz clic en **Instalar**.
    - Te permitirá abrir y ejecutar archivos `.ipynb` (Jupyter Notebooks) directamente en VS Code.
4. **Black Formatter:** Un formateador de código Python muy popular que aplica un estilo consistente automáticamente.
    - Busca `Black Formatter` y haz clic en **Instalar**.
    - Para usarlo, necesitas instalar `black` en tu entorno Python: `pip install black` .
    - Luego, en VS Code, puedes configurarlo como tu formateador predeterminado en `Archivo > Preferencias > Configuración` y buscando `python.formatting.provider` y seleccionando `black` . También puedes habilitar `Editor: Format On Save` .

5.3. Seleccionar el Intérprete de Python en VS Code

Es importante que VS Code sepa qué versión de Python (o entorno virtual) debe usar para tu proyecto.

1. **Abre la Paleta de Comandos:** `Ctrl+Shift+P` (Windows/Linux) o `Cmd+Shift+P` (macOS).
2. **Escribe** `Python: Select Interpreter` y selecciona la opción.
3. **Elige tu intérprete:** Verás una lista de los intérpretes de Python que VS Code ha detectado en tu sistema. Selecciona la versión de Python 3 que instalaste.
   - Si tienes varios proyectos o entornos virtuales, este paso es crucial para asegurar que VS Code use el intérprete correcto para cada uno.

Probando el mismo código en diferentes IDE y diferentes códigos en el mismo editor

> **Actividad**
> Entra en la página web mycompiler.io. Se trata de un emulador virtual de entornos de programación. Selecciona un lenguaje, por ejemplo Python. Aparecerá el editor con un código para el programa `Hola Mundo` . Dale a ejecutar. Prueba ese mismo código en:
>
> - pycharm (instálalo a través de Jetbrains Toolbox, con la licencia educativa que conseguiste en el apartado anterior)
> - vscode usando python3
>
> Haz lo mismo con otros lenguajes de programación, pruébalos en diversos editores.

> **Actividad**
> Usando VS Code y las extensiones correspondientes, ejecuta un `Hola Mundo` en diferentes lenguajes (python, typescript y javascript, por ejemplo). Extrae el código de mycompiler.io.

Introducción a Java. Gestores de dependencias. Aprender con Inteligencia Artificial. JSP.

Gestores de dependencias

En IntelliJ tienes varias opciones para gestionar las dependencias de tus proyectos Java, como el gestor de dependencias propio de IntelliJ, **Maven** o **Gradle**. Aquí te doy una visión rápida de cada uno, con recomendaciones sobre cuál podría ser la mejor opción según tu caso.

1. Gestor de dependencias de IntelliJ

El gestor de dependencias integrado en IntelliJ se encarga de gestionar bibliotecas manualmente a través de configuraciones locales en el proyecto. Aunque puede ser útil para proyectos pequeños o sencillos, tiene limitaciones importantes:

Ventajas del gestor de IntelliJ

- Fácil de usar para proyectos muy pequeños o para aquellos que no requieren muchas dependencias externas.
- Permite agregar librerías manualmente (en formato `.jar`) sin necesidad de un gestor externo.

Desventajas del gestor de IntelliJ

- **No es escalable**: Si el proyecto crece y tiene muchas dependencias, manejar esto manualmente puede ser muy complicado y propenso a errores.
- **No maneja automáticamente las dependencias transitorias**: Si tu biblioteca necesita otras bibliotecas, tendrás que gestionarlas tú manualmente.
- **No es compatible con automatización o integración continua**: No se integra fácilmente con herramientas de automatización de construcción como Jenkins o CI/CD.
- **Menor soporte para plugins o configuraciones avanzadas**.

¿Cuándo usaremos el gestor de IntelliJ?

Usar el gestor de dependencias propio de IntelliJ solo es recomendable para proyectos **muy pequeños** o **prototipos** que no requieran integración continua, automatización, o un sistema de construcción más avanzado.

> **Actividad**
>
> Crea un proyecto en Java usando el gestor de dependencias propio de IntelliJ y un programa que, al ejecutarlo, escriba por consola "Hola Mundo".

2. Maven

Maven es un gestor de proyectos y dependencias ampliamente utilizado en proyectos Java. Define la estructura del proyecto y gestiona las dependencias a través de un archivo XML llamado `pom.xml`.

Ventajas de Maven

- **Estandarización**: Tiene una estructura estándar de proyectos Java, lo que facilita la colaboración entre equipos.
- **Automatización de dependencias**: Puedes agregar dependencias fácilmente, y Maven descargará automáticamente las bibliotecas necesarias y sus dependencias transitorias desde repositorios centralizados.
- **Integración continua**: Está bien soportado en herramientas de integración continua (CI) como Jenkins, GitLab CI, GitHub Actions, etc.
- **Popularidad y soporte**: Hay una enorme comunidad y mucha documentación disponible.

Desventajas de Maven

- **XML**: Algunos desarrolladores consideran que editar archivos `pom.xml` es tedioso y difícil de leer cuando el archivo crece.
- **Rendimiento**: Comparado con Gradle, Maven puede ser un poco más lento, especialmente en proyectos grandes.

¿Cuándo usaremos Maven?

Maven es una excelente opción si trabajas en **proyectos medianos o grandes** o si tu equipo ya está familiarizado con él. También es ideal para proyectos con equipos distribuidos, ya que su estructura estándar y amplio uso en la industria lo hacen fácil de adoptar.

Configuración de un proyecto con Maven

1. **Crear un proyecto con Maven**:
    - Al crear un nuevo proyecto en IntelliJ, selecciona **Maven** como opción de proyecto.
    - IntelliJ generará automáticamente un archivo `pom.xml` donde podrás agregar tus dependencias.
2. **Agregar dependencias en Maven**:
    - Abre el archivo `pom.xml` y añade las dependencias dentro de la sección `<dependencies>`:

```
<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.13.1</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

3. **Ejecutar Maven**:
    - Puedes ejecutar comandos de Maven directamente en IntelliJ desde el panel de Maven (en la barra lateral derecha).
    - Comandos comunes incluyen `clean`, `install`, y `package`.

> **Actividad**
>
> Crea un proyecto en Java usando Maven y un programa que, al ejecutarlo, escriba por consola "Hola Mundo, vivo en Maven".

3. Gradle

Gradle es un gestor de construcción más moderno y flexible que Maven. En lugar de usar XML, Gradle utiliza archivos de configuración en **Groovy** o **Kotlin** para definir dependencias y tareas, lo que lo hace más legible y potente.

Ventajas de Gradle

- **Flexibilidad**: Gradle es más flexible que Maven, permitiendo configuraciones avanzadas sin tantas restricciones.
- **Rendimiento**: Gradle es generalmente más rápido que Maven debido a su sistema de ejecución incremental y su caché local.
- **Soporte para múltiples lenguajes**: Aunque es muy popular en proyectos Java, Gradle también es común en proyectos Android, Kotlin, y otros lenguajes como Scala o Groovy.
- **DSL legible**: Al usar un DSL en Groovy o Kotlin, los archivos de configuración (`build.gradle`) son más compactos y fáciles de leer que los archivos XML de Maven.

Desventajasde Gradle

- **Curva de aprendizaje**: Aunque más potente, puede ser más difícil de aprender al principio si vienes de Maven, sobre todo si necesitas personalizar tareas específicas.

- **Menos estándar**: Aunque cada vez es más común, no todos los proyectos Java adoptan Gradle. Por lo tanto, puede no ser la mejor opción si tu equipo ya está familiarizado con Maven.

¿Cuándo usamos Gradle?

Gradle es ideal para **proyectos grandes**, especialmente si necesitas flexibilidad o si trabajas en proyectos **multilenguaje** o **Android**. También es una excelente opción si te importa el rendimiento y la eficiencia en la construcción.

Configuración de un proyecto con Gradle

1. **Crear un proyecto con Gradle**:
   - Al crear un nuevo proyecto, selecciona **Gradle** como el sistema de construcción.
   - Se generará un archivo `build.gradle` que puedes usar para gestionar las dependencias.
2. **Agregar dependencias en Gradle**:
   - Abre el archivo `build.gradle` y agrega las dependencias en la sección `dependencies`:

```
dependencies {
    testImplementation 'junit:junit:4.13.1'
}
```

3. **Ejecutar Gradle**:
   - Al igual que con Maven, puedes ejecutar tareas de Gradle directamente desde IntelliJ usando el panel de Gradle. Tareas comunes incluyen `build`, `test`, y `clean`.

> **Actividad**
>
> Crea un proyecto en Java usando Gradle y un programa que, al ejecutarlo, escriba por consola "Hola Mundo, vivo en Gradle".

Durante el curso emplearemos Maven, por su mayor implantación en la actualidad.

Uso de funciones Hash

Programación en Java. Aprendizaje a través de la Inteligencia Artificial

Java es un lenguaje muy popular en el desarrollo de aplicaciones web, se emplea sobre todo en el campo del backend aunque puede ser empleado para el frontend usando extensiones como JavaServer Pages (JSP). Java emplea la misma lógica que otros lenguajes como C# o C++.

Al crear nuestro proyecto en Maven, en la carpeta src introducimos nuestros archivos de código fuente, en los que insertamos nuestras clases. En java, los archivos deben tener el mismo nombre que las clases públicas definidas y solo puede haber una clase pública por archivo. Dentro de la clase pública, podemos crear métodos estáticos, que nos permiten tratar la clase como si fuera un contenedor de funciones, o definirla como una clase normal, como las que trabajamos en la Unidad de Programación 3 sobre diseño orientado a objetos.

- *archivo Main.java*

```
import java.util.*;
import java.lang.*;
import java.io.*;

// The main method must be in a class named "Main".
public class Main {
    public static void main(String[] args) {
        System.out.println("Hello world!");
    }
}
```

Este es el código de un `Hola Mundo` extraído de MyCompiler.io. Si usamos la autocompleción de IntelliJ, los `import` se añadirán solos. Es importante, por lo tanto, emplear esas herramientas que nos facilita el IDE, conocidas como herramientas de refactorización.

Una vez que conocemos la estructura básica de un programa en Java, es hora de transferir nuestros conocimientos en cualquier otro lenguaje de programación a Java. Para ello, una herramienta fundamental es la **Inteligencia Artificial**. Podemos usar ChatGPT, Gemini o cualquier otra. Incluso podemos ir alternando para obtener diferentes perspectivas. La versión de Java que trabajaremos será la 21 LTS (Long Term Support), por temas de compatibilidad.

> **Actividad**
> Coge tus apuntes de Programación sobre `Entrada y salida de datos por consola` y pídele a un modelo de lenguaje que te los traduzca a Java.

Más adelante aprenderemos a traducir lo aprendido en la unidad de programación 3 a lenguaje Java y, aunque no es necesario para este módulo, sí que sería recomendable hacer un esfuerzo en aprender los siguientes puntos sobre Java:

- Uso de if, if-else, switch y enhaced switch en Java.
- Uso de while, do-while, for y enhanced for en Java.

> **Actividad**
> Pídele a un modelo de lenguaje que te haga un tutorial para aprender a manejar cada uno de los siguientes puntos en Java. Pídele un tutorial distinto por punto, para evitar respuestas demasiado largas o complejas. Pídele que te proponga ejercicios para que tú los resuelvas y luego él te los corrija.
>
> - Uso de if y de if-else
> - Uso de switch-case
> - Uso de while y do-while
> - Uso de for
> - Creación y consulta de arrays
> - Uso de enhanced for para recorrer arrays

Puedes emplear también la información obtenida en el apartado de Diagramas de Actividad de la Unidad de Programación 2, pero ten en cuenta que los diagramas que tratan la concurrencia son de carácter avanzado y pueden hacerse complicados de trabajar y depurar en Java.

Conforme vayas avanzando en el módulo de programación, procura tratar de convertir esos conocimientos a Java. Te será de utilidad en un futuro.

Uso de JSP y Jakarta EE en IntelliJ Ultimate

Jakarta EE es una plataforma de programación para desarrollar y ejecutar aplicaciones en lenguaje Java. Entre sus estándares incluye el uso de servlets y de JavaServer Pages (JSP), herramientas para trabajar en la programación de aplicaciones web usando Java.

JSP es similar a PHP, en el sentido que combina el lenguaje HTML con, en este caso, fragmentos de código en Java. En un archivo .jsp, introduce tu código html como harías de forma normal y, cuando requieras algo de java, introduce el código entre `<% %>`. Si deseas imprimir directamente un valor por pantalla, puedes usar `<%= %>`. Al código introducido se lo conoce como `scriptlet`.

A nivel interno, el archivo .jsp se traduce a un archivo .java que emplea un objeto derivado de la clase `HttpServlet` y el escribe todo dentro del método `doGet` a través de un objeto PrintWriter.out. Necesita un servidor web, como Tomcat, para poder funcionar. Vamos a aprender cómo configurarlo fácilmente en IntelliJ IDEA Ultimate Edition. Recuerda que con la licencia educativa puedes acceder de forma gratuita a ese IDE.

Necesitarás:

1. **IntelliJ IDEA Ultimate Edition:** Asegúrate de que tienes la versión Ultimate, ya que la Community Edition no tiene soporte para Java EE/Web.
2. **Java Development Kit (JDK):** Necesitas un JDK instalado en tu sistema. Normalmente, IntelliJ lo instala de forma guiada y fácil.
3. **Apache Tomcat** Descargado y descomprimido en tu sistema. Puedes bajarlo de aquí. Si tienes un Mac o un Linux, baja la versión tar.gz. Guárdalo en una carpeta que te sea accesible.

Pasos para crear y ejecutar un proyecto JSP en IntelliJ IDEA Ultimate:

1. Crear un Nuevo Proyecto Web

1. Abre IntelliJ IDEA.
2. Desde la pantalla de bienvenida, haz clic en **"New Project"**. Si ya tienes un proyecto abierto, ve a `File > New > Project...` .
3. En el panel izquierdo del asistente "New Project", selecciona **"Jakarta EE"** (o "Java Enterprise" en versiones más antiguas).
4. En el panel derecho:
   - **Project name:** Dale un nombre a tu proyecto (ej., `MiPrimerJSP`).
   - **Location:** Elige la ubicación donde se guardará tu proyecto.
   - **Build system:** Selecciona **"Maven"** (recomendado para proyectos Java EE modernos) o "Gradle". Para este tutorial, usaremos Maven.
   - **JDK:** Asegúrate de que se selecciona el JDK correcto. Si no tienes uno configurado, haz clic en "Add JDK..." y navega a la ubicación de tu JDK.
   - **Template:** Selecciona **"Web Application"**. Esto creará la estructura básica para una aplicación web, incluyendo un `index.jsp` y `web.xml` .
   - **Dependencies (solo si usas Maven/Gradle):** Por ahora, puedes dejarlo tal cual.
5. Haz clic en **"Next"**.

2. Configurar el Servidor de Aplicaciones (Tomcat)

1. En la siguiente pantalla, bajo **"Application Server"**:
   - Haz clic en el botón **"New..."**.
   - Selecciona **"Tomcat Server"**.
   - En la ventana emergente, haz clic en el botón `...` (Browse) y navega a la **carpeta raíz de tu instalación de Apache Tomcat** (donde se encuentran `bin` , `conf` , `lib` , etc.).
   - IntelliJ IDEA detectará la versión de Tomcat. Haz clic en "OK".

- Asegúrate de que el Tomcat recién configurado esté seleccionado en la lista "Application Server".

2. En la misma ventana:
   - **Jakarta EE version:** Elige la versión de Jakarta EE que deseas usar. Para la mayoría de los casos, la última versión estable (ej., `Jakarta EE 9` o `10`) estará bien. Si necesitas compatibilidad con versiones antiguas, elige `Jakarta EE 8` (que es el nombre anterior de Java EE 8).
   - **Language:** Deja "Java".
   - **Servlet version:** Deja la versión por defecto o la más reciente (ej., `6.0`).
   - **Generate web.xml: Asegúrate de que esta casilla esté marcada.** Es crucial para la configuración de tu aplicación web.
3. Haz clic en **"Create"**.

3. Explorar la Estructura del Proyecto

IntelliJ IDEA creará el proyecto con una estructura similar a esta (si usas Maven):

```
MiPrimerJSP/
├── .idea/                (Configuración de IntelliJ)
├── src/
│   └── main/
│       ├── java/         (Para tus clases Java, servlets, etc.)
│       └── webapp/       (La raíz de tu aplicación web)
│           ├── WEB-INF/
│           │   └── web.xml   (Descriptor de despliegue)
│           └── index.jsp  (Tu primera página JSP)
└── pom.xml               (Archivo de configuración de Maven)
```

- **`src/main/webapp/`**: Aquí es donde colocarás todos tus archivos JSP, HTML, CSS, JavaScript e imágenes. IntelliJ ya habrá creado un `index.jsp` por defecto.
- **`src/main/webapp/WEB-INF/web.xml`**: Este archivo es el descriptor de despliegue de tu aplicación web. Contiene configuraciones importantes, como la bienvenida, mapeos de servlets, etc.
- **`src/main/java/`**: Si vas a escribir servlets o clases Java que tu JSP va a usar, irán aquí.

4. Modificar el `index.jsp` (Opcional)

Abre el archivo `src/main/webapp/index.jsp`. Verás un código HTML básico. Puedes modificarlo para incluir un poco de código JSP para probar:

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>Mi Primer JSP</title>
</head>
<body>
<h1>¡Hola desde JSP en IntelliJ!</h1>
<%-- Código Java dentro de un JSP --%>
<%
    String mensaje = "La fecha actual es: " + new java.util.Date();
    out.println("<p>" + mensaje + "</p>");
%>
</body>
</html>
```

5. Configurar la Ejecución (Run Configuration)

IntelliJ IDEA suele crear automáticamente una configuración de ejecución para Tomcat cuando creas un proyecto web.

1. Mira en la barra de herramientas superior, cerca de los botones de "Play" (Run) y "Debug". Deberías ver un desplegable con el nombre de tu proyecto (ej., `Tomcat [nombre de tu proyecto]`).
2. Si no está o quieres verificar la configuración:
    - Haz clic en el desplegable y selecciona **"Edit Configurations..."**.
    - En el panel izquierdo, bajo "Tomcat Server", selecciona **"Local"** (o el nombre de tu configuración).
    - Asegúrate de que en la pestaña **"Server"**:
        - `Application server` apunta a tu instalación de Tomcat.
        - `URL` muestra algo como `http://localhost:8080/[nombre_de_tu_proyecto]_war_exploded` (esto es el "context path" por defecto que usa IntelliJ para el despliegue de desarrollo).
    - En la pestaña **"Deployment"**:
        - Deberías ver una entrada con el nombre de tu proyecto seguido de `_war_exploded`. Esto significa que IntelliJ está desplegando tu proyecto de forma "descomprimida" para un desarrollo rápido.
        - El "Application context" (la URL por la que se accederá) debería ser `/MiPrimerJSP` (o el nombre que le diste a tu proyecto). Si quieres cambiarlo, haz clic en el lápiz al final de la línea.

    - Asegúrate de que está seleccionada la opción `Deploy at startup`.
3. Haz clic en **"OK"**.

6. Ejecutar el Proyecto

1. En la barra de herramientas superior, haz clic en el botón verde de **"Play"** (Run) junto al desplegable de la configuración de Tomcat.
2. IntelliJ IDEA:
    - Construirá tu proyecto (Maven lo compilará y empaquetará).
    - Iniciará el servidor Tomcat.
    - Desplegará tu aplicación web en Tomcat.
    - Automáticamente abrirá tu navegador predeterminado en la URL de tu aplicación (ej., `http://localhost:8080/MiPrimerJSP/`).

Deberías ver tu `index.jsp` renderizado en el navegador.

Consejos adicionales:

- **Cambios en JSP:** Si modificas tu archivo `.jsp`, simplemente **refresca tu navegador**. IntelliJ IDEA y Tomcat están configurados para detectar cambios en JSPs sin necesidad de reiniciar el servidor.
- **Cambios en clases Java/Servlets:** Si modificas clases Java (archivos `.java`), necesitarás **recompilar el proyecto** (Build > Rebuild Project) y luego **reiniciar el servidor Tomcat** (puedes detenerlo y volver a iniciarlo con los botones de la barra de herramientas, o usar el botón de "Restart" que aparece a veces).
- **Ventana "Run":** La ventana "Run" (en la parte inferior de IntelliJ) mostrará los logs de Tomcat, lo cual es muy útil para depurar si algo sale mal.
- **Depuración:** Puedes poner puntos de interrupción en tu código Java (incluso en los scriptlets JSP) y ejecutar en modo "Debug" para depurar tu aplicación.

Uso de JSP en la actualidad

JSP no es una tecnología muy popular en la actualidad, ya que en general las páginas dinámicas (las que se generan en el servidor) han caído en popularidad en favor de las páginas estáticas generadas con React, Angular y otras tecnologías web. Sin embargo, JSP tiene algunas ventajas y además se utiliza todavía en páginas antiguas.

La ventaja es que podemos trabajar con los conocimientos que adquiramos en Java de forma directa y es una forma muy sencilla de poder conectar nuestros proyectos de Java a una interfaz gráfica

diseñada en HTML5, CSS y Javascript, tal y como se trabaja en el módulo profesional de Lenguaje de Marcas y Gestión de Sistemas de la información.

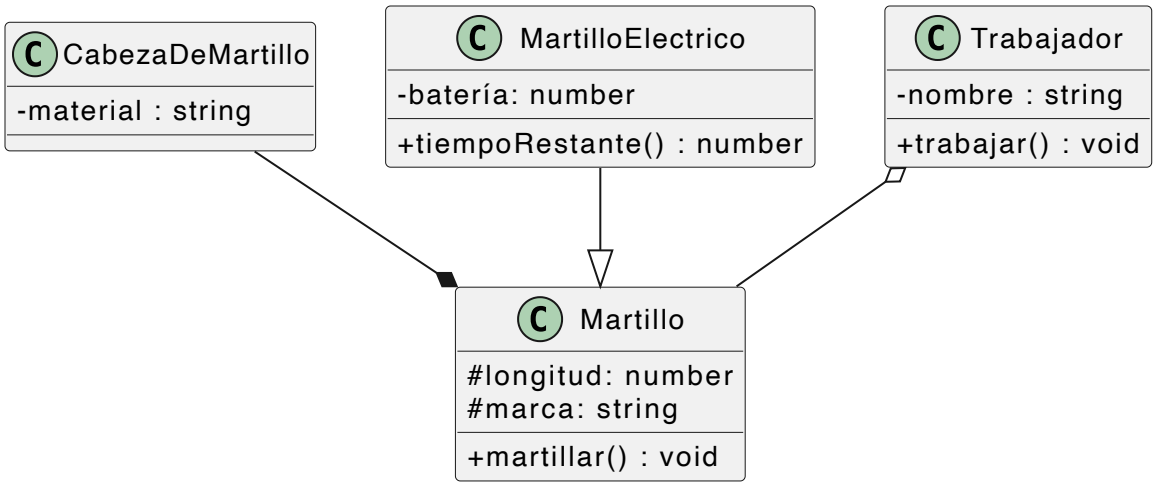
Si quieres profundizar más sobre el uso de JSP, puedes realizar la actividad de ampliación relacionada.

> **Actividad**
> Crea una página en JSP que calcule los primeros 7 números primos y los imprima en la página web. Usa scriptlets para el cálculo.

Java. Del diagrama al código e Ingeniería inversa. Métodos y atributos estáticos

Del UML al Código Java: Un Ejemplo Práctico

En Java, como en cualquier otro lenguaje de programación, podemos trasladar cualquier diagrama UML de clases a programación. Tomemos como ejemplo el siguiente diagrama UML y su implementación en Java:



```java
// Martillo.java
public class Martillo {
    protected float longitud;
    protected String marca;
    protected CabezaDeMartillo cabeza; // Composición: Martillo "tiene" una CabezaDeMar

    public Martillo() {
        this.cabeza = new CabezaDeMartillo(); // La CabezaDeMartillo se crea con el Mar
    }

    public void martillar() {
        // Implementación del método martillar
    }
}

// MartilloElectrico.java
public class MartilloElectrico extends Martillo { // Herencia: MartilloElectrico "es un
    private float bateria;

    public float getBateria() {
        return bateria;
    }

    public void setBateria(float bateria) {
        this.bateria = bateria;
    }

    public int tiempoRestante() {
        // Cálculo y retorno del tiempo restante de la batería
        return 0; // Placeholder
    }
}

// Trabajador.java
public class Trabajador {
    private String nombre;
    private Martillo martillo; // Agregación: Trabajador "usa un" Martillo

    public Trabajador(String nombre, Martillo martillo) { // El Martillo se pasa al con
        this.nombre = nombre;
        this.martillo = martillo;
    }
```

```
    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public Martillo getMartillo() {
        return martillo;
    }

    public void setMartillo(Martillo martillo) {
        this.martillo = martillo;
    }

    public void trabajar() {
        // Implementación del método trabajar
    }
}

// CabezaDeMartillo.java
public class CabezaDeMartillo {
    private String material;

    public CabezaDeMartillo() {
        // Constructor por defecto
    }

    public CabezaDeMartillo(String material) {
        this.material = material;
    }

    public String getMaterial() {
        return material;
    }

    public void setMaterial(String material) {
        this.material = material;
    }
}
```

Adaptación de los tipos de Relaciones en Java

Herencia (`extends`)

La **herencia** se expresa explícitamente usando la palabra reservada `extends` . Por ejemplo, `class MartilloElectrico extends Martillo` indica que `MartilloElectrico` hereda todas las características y comportamientos de `Martillo` , además de tener los suyos propios. En un diagrama UML, esto se representa con una flecha con punta triangular que va de la subclase a la superclase. En el caso de usar interfaces, se emplearía `implements` . En Java, la herencia es simple, pero se pueden usar múltiples interfaces.

Relaciones de Pertenencia (Asociación, Agregación y Composición)

Las relaciones donde una clase "contiene" o "usa" objetos de otra clase se representan como **atributos** dentro de la clase contenedora. La clave está en cómo se maneja la vida útil de los objetos relacionados:

- **Agregación** (representada por un rombo vacío en UML, `o--` en PlantUML): En una agregación, la "parte" (el objeto contenido) puede existir independientemente del "todo" (el objeto que lo contiene). En Java, esto se traduce en pasar la instancia de la "parte" como **parámetro al constructor** de la clase "todo", o asignarla a través de un *setter*. Observa cómo el `Trabajador` recibe un `Martillo` en su constructor, lo que implica que el `Martillo` puede existir y ser usado por otros `Trabajador` es o fuera de cualquier `Trabajador` .
- **Composición** (representada por un rombo relleno en UML, `--*` en PlantUML): En una composición, la "parte" es una parte integral y dependiente del "todo". Si el "todo" deja de existir, la "parte" también deja de tener sentido o se destruye con él. En Java, la instancia de la "parte" se **crea directamente dentro del constructor** de la clase "todo". En nuestro ejemplo, la `CabezaDeMartillo` se instancia dentro del constructor de `Martillo` , lo que significa que un `Martillo` siempre "posee" y es responsable de la vida de su `CabezaDeMartillo` .

Constructores y Convenciones de Nomenclatura

Constructores

Un **constructor** es un método especial que se invoca automáticamente cuando se crea un nuevo objeto de una clase. Su objetivo principal es inicializar el estado del objeto. Si no defines uno, Java proporciona un **constructor por defecto** sin parámetros.

Para definir un constructor, simplemente creas un método con el mismo nombre que la clase. Puedes pasarle parámetros para inicializar los atributos del objeto, lo cual es común en las relaciones de asociación y agregación, como vimos con `Trabajador`. Para la composición, el objeto "parte" se suele crear directamente dentro del constructor del "todo".

Convenciones de Nomenclatura en Java

- **Clases**: Se escriben con la primera letra en mayúscula y utilizando **CamelCase** (por ejemplo, `MartilloElectrico`).
- **Variables y Métodos**: La primera letra es minúscula y se usa CamelCase para las palabras siguientes (por ejemplo, `longitud`, `tiempoRestante`).

Archivos y Visibilidad

En Java, generalmente, cada **clase pública** reside en un archivo `.java` que lleva el mismo nombre de la clase. Aunque es posible tener múltiples clases no públicas en un solo archivo, no es una práctica común para proyectos grandes.

Si no especificas un modificador de acceso (como `public`, `private`, `protected`), la visibilidad por defecto es de **paquete**. Esto significa que los miembros (atributos y métodos) son accesibles desde cualquier otra clase dentro del mismo paquete, pero son privados para clases fuera de ese paquete. Es una visibilidad muy útil que explorarás más a fondo a medida que avances.

> **Actividad**
>
> Crea un proyecto nuevo en IntelliJ. Añade un archivo, Libro.java, y crea en él la clase Libro, cuyo diagrama creaste en la actividad 1.
>
> Después, crea un diagrama en uml para representar una estantería de libros. Crea el archivo Estanteria.java (no uses tildes) y codifica la clase.
>
> Haz un programa `main` de la siguiente forma:

```
public class Main{
    public static void main(String [] args){
        Libro libro = new Libro();
        Estanteria estanteria = new Estanteria();
    }
}
```

Ingeniería Inversa: Del código a los diagramas de clases

La ingeniería inversa o retroingeniería es el proceso llevado a cabo con el objetivo de obtener información o un diseño a partir de un producto, con el fin de determinar cuáles son sus componentes y de qué manera interactúan entre sí y cuál fue el proceso de fabricación. En nuestro caso, nos vamos a centrar en el código como producto y vamos a tratar de elaborar el diagrama de clases correspondiente.

Si somos capaces de traducir un diagrama de clases a código, también somos capaces de realizar ingeniería inversa y transferir un código a un diagrama de clases, para su posterior refinamiento o su estudio.

> **Actividad**
> Crea un diagrama de clases a partir de los siguientes códigos en Java

```java
// Archivo: Persona.java
package com.universidad;

public abstract class Persona {
    protected String nombre;
    protected String id;

    public Persona(String nombre, String id) {
        this.nombre = nombre;
        this.id = id;
    }

    public String getNombre() {
        return nombre;
    }

    public String getId() {
        return id;
    }

    public abstract void realizarActividadPrincipal();
    public void mostrarInformacionBasica() {
        // Método sin implementar
    }
}

// Archivo: Estudiante.java
package com.universidad;

import java.util.List;
import java.util.ArrayList;

public class Estudiante extends Persona { // Herencia
    private String carrera;
    private List<Curso> cursosInscritos; // Asociación (muchos a muchos implícita)

    public Estudiante(String nombre, String id, String carrera) {
        super(nombre, id);
        this.carrera = carrera;
        this.cursosInscritos = new ArrayList<>();
    }

    public String getCarrera() {
```

```
        return carrera;
    }

    public List<Curso> getCursosInscritos() {
        return cursosInscritos;
    }

    public void inscribirCurso(Curso curso) {
        // Método sin implementar
    }

    @Override
    public void realizarActividadPrincipal() {
        // Método sin implementar
    }
}

// Archivo: Profesor.java
package com.universidad;

import java.util.List;
import java.util.ArrayList;

public class Profesor extends Persona { // Herencia
    private String departamento;
    private List<Curso> cursosImpartidos; // Agregación (Profesor tiene Cursos, pero Cu

    public Profesor(String nombre, String id, String departamento) {
        super(nombre, id);
        this.departamento = departamento;
        this.cursosImpartidos = new ArrayList<>();
    }

    public String getDepartamento() {
        return departamento;
    }

    public List<Curso> getCursosImpartidos() {
        return cursosImpartidos;
    }

    public void asignarCurso(Curso curso) {
        // Método sin implementar
```

```
    }

    @Override
    public void realizarActividadPrincipal() {
        // Método sin implementar
    }
}

// Archivo: Curso.java
package com.universidad;

import java.util.List;
import java.util.ArrayList;

public class Curso {
    private String nombre;
    private String codigo;
    private int creditos;
    private Profesor profesorAsignado;
    private Departamento departamento;

    public Curso(String nombre, String codigo, int creditos) {
        this.nombre = nombre;
        this.codigo = codigo;
        this.creditos = creditos;
    }

    public String getNombre() {
        return nombre;
    }

    public String getCodigo() {
        return codigo;
    }

    public int getCreditos() {
        return creditos;
    }

    public Profesor getProfesorAsignado() {
        return profesorAsignado;
    }
```

```
    public void setProfesorAsignado(Profesor profesorAsignado) {
        // Método sin implementar
    }

    public Departamento getDepartamento() {
        return departamento;
    }

    public void setDepartamento(Departamento departamento) {
        // Método sin implementar
    }

    public void añadirEstudiante(Estudiante estudiante) {
        // Método sin implementar
    }
}

// Archivo: Departamento.java
package com.universidad;

import java.util.List;
import java.util.ArrayList;

public class Departamento {
    private String nombre;
    private String codigo;
    private List<Profesor> profesores;
    private Direccion direccion;

    public Departamento(String nombre, String codigo, String calle, String ciudad) {
        this.nombre = nombre;
        this.codigo = codigo;
        this.profesores = new ArrayList<>();
        this.direccion = new Direccion(calle, ciudad);
    }

    public String getNombre() {
        return nombre;
    }

    public String getCodigo() {
        return codigo;
    }
```

```
    public List<Profesor> getProfesores() {
        return profesores;
    }

    public Direccion getDireccion() {
        return direccion;
    }

    public void añadirProfesor(Profesor profesor) {
        // Método sin implementar
    }

    public void mostrarInformacionDepartamento() {
        // Método sin implementar
    }
}

// Archivo: Direccion.java
package com.universidad;

public class Direccion { // Clase parte para Composición
    private String calle;
    private String ciudad;

    public Direccion(String calle, String ciudad) {
        this.calle = calle;
        this.ciudad = ciudad;
    }

    public String getCalle() {
        return calle;
    }

    public String getCiudad() {
        return ciudad;
    }

    public String obtenerDireccionCompleta() {
        // Método sin implementar
        return null;
    }
}
```

Métodos y atributos estáticos

Existe una palabra reservada que hemos estado utilizando hasta ahora sin saber muy bien qué era: `static`. Los métodos y atributos estáticos son aquellos que no pertenecen al objeto, sino a la clase.

Esto quiere decir que una vez que el programa es ejecutado, son siempre accesibles independientemente de si hay o no instancias de la clase que los contiene.

En las primeras etapas del curso hemos usado los elementos estáticos para simular el paradigma de programación procedural en Java, es decir, la resolución de problemas mediante funciones/procedimientos y no mediante objetos.

Si declaramos una atributo como estático, se convierte en una especie de variable con ámbito dentro de toda la clase y a la cual podemos acceder desde fuera dependiendo de su visibilidad. Lo mismo sucede con los métodos.

Una cosa que hay que tener en cuenta es que no se pueden usar atributos de la clase dentro de métodos estáticos (porque, en realidad, no tiene sentido ya que no hay instancia de la clase), pero sí podemos usar atributos estáticos dentro de métodos del objeto.

En cierto modo, podemos usar las clases como contenedores de métodos estáticos, que se comportan como librerías de funciones. Tal es el caso de la clase `Math`, que incluye métodos estáticos para operaciones matemáticas complejas.

De esta manera, podemos emplear los atributos y métodos estáticos como herramientas útiles para emplear dentro o fuera de la clase.

> **Actividad**:
> Uso de métodos y atributos estáticos
>
> Crea una variable estática dentro de la clase libro que sirva para contar el número de libros (instancias) que hay. Empléala para darles a cada libro un identificador único. Para ello, añade un atributo privado `id` de tipo entero. El `id` no debe pasarse como atributo al constructor ni debe poder ser modificado con un setter.
>
> Después, crea un método estático dentro de la clase Estantería que reciba como parámetro una estantería y devuelva un array de string. El contenido del array de string es cada uno de los parámetros, incluido el `id`, separado por el caracter `;`. Puedes rodear cada parámetro entre `` si dentro de alguno de los campos hubiera un `;`. Cada libro ocupa un espacio distinto del array. A este formato de texto se le conoce como CSV (Comma Separated Values) y es muy empleado en la codificación de datos.

Control de versiones

Sistemas de control de versiones

Los sistemas de control de versiones (VCS) son herramientas que pueden registrar cualquier cambio en cualquier archivo o conjunto de archivos a lo largo del tiempo, de manera que podamos recuperar fácilmente cualquier versión anterior. Pueden ser utilizados no solo con archivos fuente, sino también con cualquier otro tipo de archivo.

Un VCS nos permite revertir el estado de cualquier archivo o incluso de un proyecto completo, comparar archivos a lo largo del tiempo, determinar quién cambió un archivo en un momento determinado y mucho más. Además, si algún archivo se daña o se pierde, podemos volver a una versión anterior en la historia y recuperarlo.

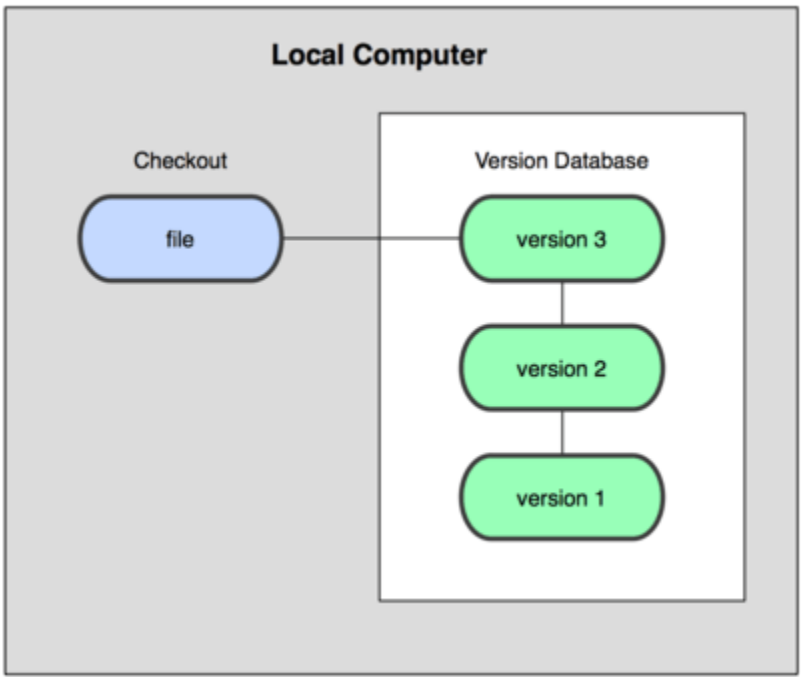
1. Tipos de VCS

1.1 VCS Local

Los VCS pueden utilizarse tanto en línea como en modo local. Este último modo es particularmente útil porque podemos crear fácilmente una copia de seguridad de un proyecto y almacenarla localmente, de modo que podamos restaurarla más tarde si la necesitamos (en caso de un error, por ejemplo) y volver a una versión estable.

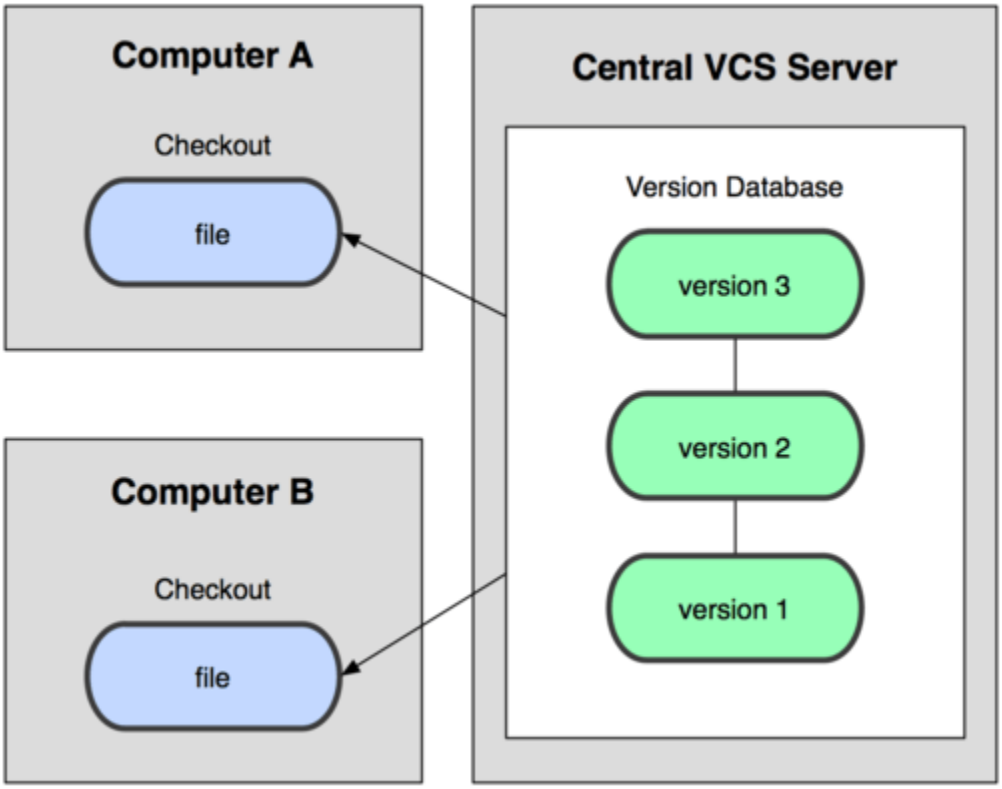
La principal ventaja de esto es su simplicidad, y el principal inconveniente es que debemos gestionar el control de versiones manualmente, por lo que podemos cometer errores en el proceso. Por ejemplo, podemos olvidarnos de que estamos en la carpeta equivocada y luego modificar el archivo de copia de seguridad en lugar del archivo actual.

Para hacer frente a estos problemas, existen algunas herramientas interesantes que nos ayudan a gestionar los archivos y los cambios. Una de las más populares es un sistema llamado *rcs*, que aún se encuentra en muchos ordenadores. Esta herramienta básicamente almacena un conjunto de parches o diferencias entre archivos de una versión a otra. Estos cambios se almacenan en un tipo especial de archivo, y luego el sistema puede recuperar cualquier estado anterior de cualquier archivo, añadiendo o quitando los parches correspondientes.



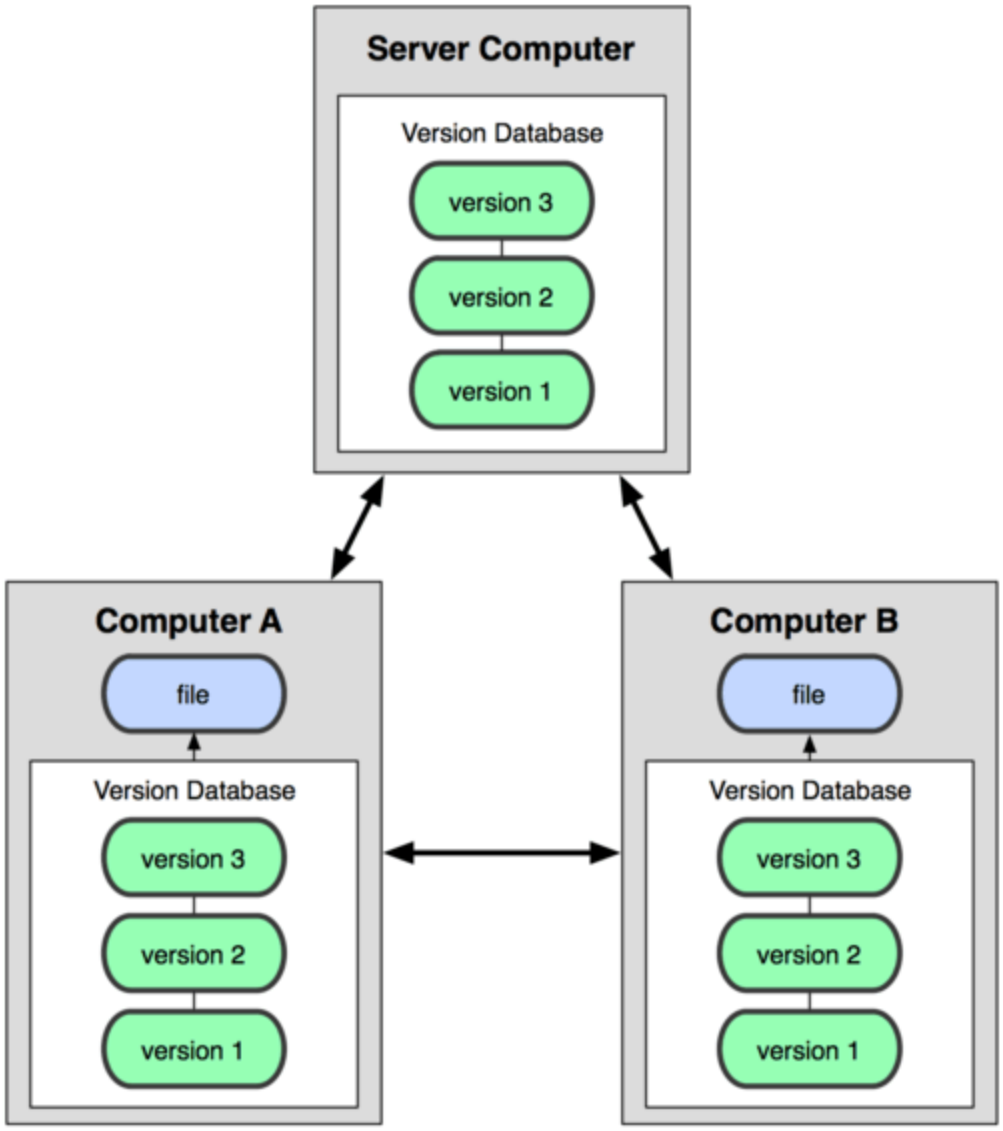
1.2 VCS Centralizado

Los VCS locales no son adecuados cuando necesitamos colaborar con otros miembros del equipo. Para resolver este problema, también existen los VCS centralizados (CVCS). Estos sistemas se instalan en un solo servidor que contiene todos los archivos y sus diferentes versiones. Entonces, muchos clientes pueden conectarse a este servidor y descargar/subir cambios a estos archivos. Esta segunda manera de controlar versiones fue un estándar durante muchos años, ya que tenía grandes ventajas sobre los sistemas CVS locales, pero su principal inconveniente es que, si el servidor falla, podríamos perder todo el proyecto.



1.3 VCS Distribuido

Los VCS distribuidos (DVCS) surgieron para resolver el principal inconveniente del CVCS. En un DVCS (como Git, Mercurial, Bazaar o Darcs), los clientes no solo se conectan al servidor, sino que también descargan todo el repositorio. Así, si un servidor falla, cualquiera de los repositorios locales de los clientes puede copiarse de nuevo al servidor, y el proyecto puede ser restaurado. Cada vez que descargamos cualquier cosa del repositorio, estamos haciendo una copia de seguridad completa de los datos.



2. Git

Git fue desarrollado por el equipo de Linux una vez que rompieron su relación con BitKeeper, la herramienta que usaban para el control de versiones antes. A partir de las carencias observadas en esta herramienta, decidieron algunos de los objetivos principales del nuevo sistema a desarrollar:

- Velocidad
- Diseño sencillo
- Fuerte soporte para el desarrollo no lineal (miles de ramas paralelas)
- Completamente distribuido
- Adecuado para grandes proyectos (como el núcleo de Linux)
- Eficiencia (en términos de velocidad y tamaño de los datos)

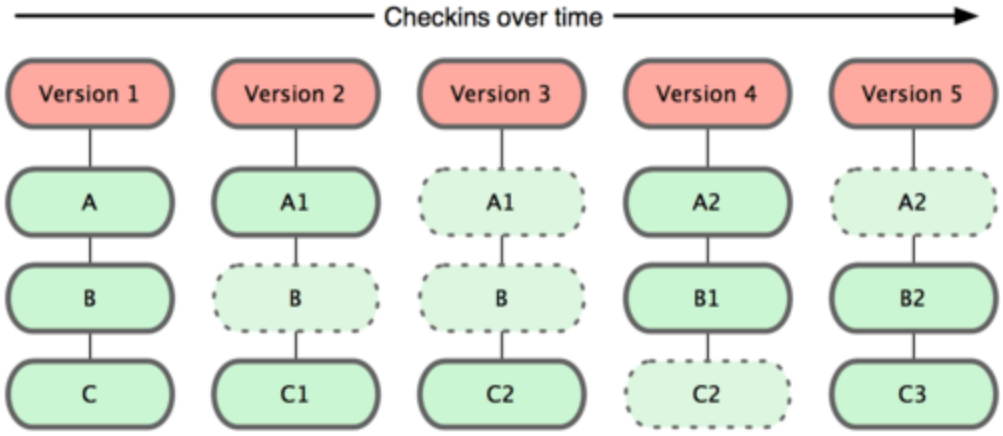
Desde su nacimiento en 2005, Git ha evolucionado y se ha vuelto cada vez más fácil de usar. Es realmente rápido y eficiente con proyectos grandes, y tiene un sistema de ramificación excepcional.

2.1. Fundamentos de Git

Veamos algunos de los conceptos básicos relacionados con el sistema de control de versiones Git.

Modelado de datos

Git almacena una especie de conjunto de "instantáneas" de su sistema de archivos, en lugar de almacenar una lista de cambios. Cada vez que subimos un nuevo cambio, básicamente toma una foto de cada archivo en ese momento y almacena una referencia a esta instantánea. Si el archivo no ha sido modificado, entonces Git no guarda una copia del mismo, solo un enlace a la versión anterior idéntica.



Esta es una diferencia importante entre Git y casi cualquier otro VCS, y hace que Git reconsidere estos aspectos de las generaciones anteriores de VCS. Por lo tanto, se parece más a un pequeño sistema de archivos con algunas herramientas útiles, que a un VCS.

Trabajo local

La mayoría de las funciones de Git solo necesitan archivos y recursos locales para funcionar. Como el historial del proyecto se almacena localmente, muchas operaciones son inmediatas y nos permite trabajar en un proyecto incluso si no estamos conectados a Internet. Los cambios se almacenan localmente y, tan pronto como tenemos una conexión, el repositorio externo puede actualizarse.

Integridad

Git utiliza el algoritmo hash SHA-1 para almacenar la información, por lo que los datos siempre se verifican y, en caso de que se modifiquen, Git lo notificaría.

Solo añade información

Cada operación de Git consiste en añadir información, por lo que todo se puede deshacer fácilmente (la información no se borra). Después de confirmar una instantánea, la información se almacena de

forma segura.

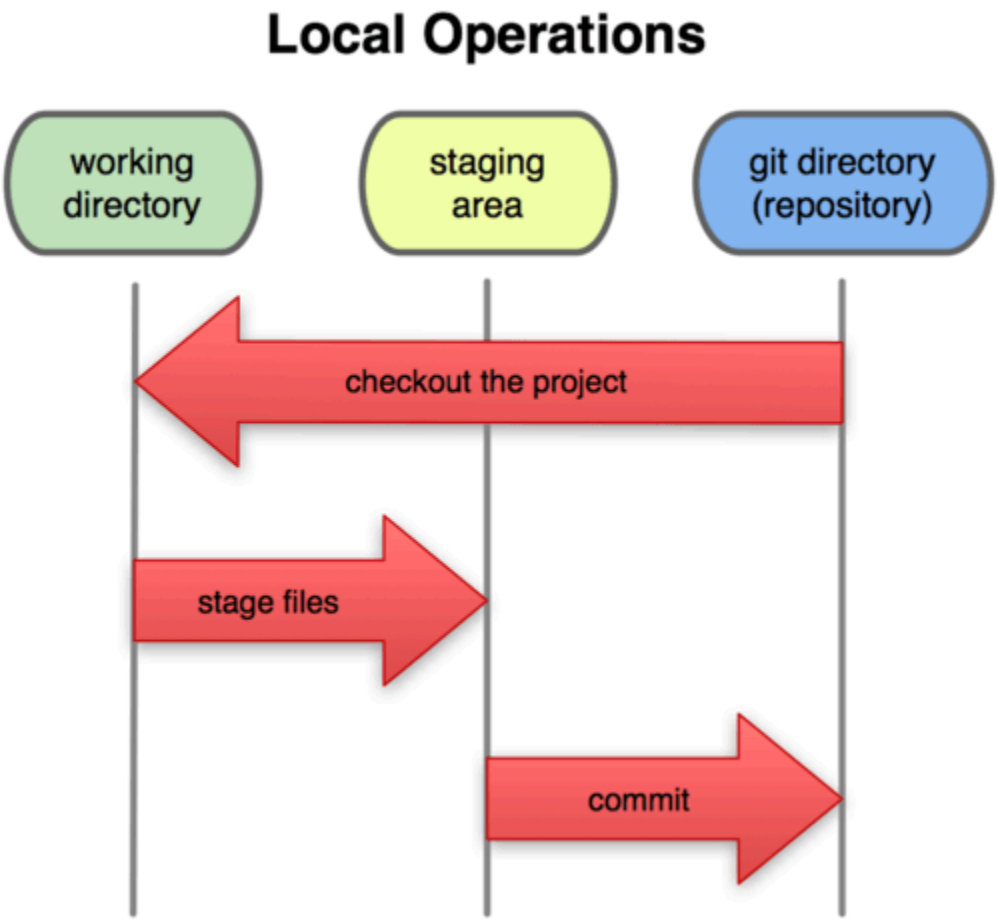
Estados del proyecto

Git tiene tres estados principales en los que puede encontrarse cada archivo de un proyecto:

- **Modificado:** los datos se han cambiado localmente, pero aún no se han confirmado.
- **Preparado (Staged):** los datos se han etiquetado para ser enviados en la próxima confirmación (commit).
- **Confirmado (Committed):** los datos se almacenan de forma segura en un almacenamiento local.

Por lo tanto, hay tres secciones en Git:

- **Directorio Git:** donde Git almacena los metadatos y la base de datos de los elementos del proyecto. Esta parte es lo que copiamos cuando clonamos el repositorio desde otra computadora.
- **Directorio de trabajo:** es una copia de una versión del proyecto. Estos archivos se extraen de la base de datos de Git y se colocan en una carpeta, listos para ser utilizados.
- **Área de preparación (Staging area):** es un archivo simple almacenado en el directorio Git que contiene información sobre los archivos que se enviarán en la próxima confirmación. También se le llama *índice*.


Local Operations
working directory
staging area
git directory (repository)
checkout the project
stage files
commit

Fuente

Uso de Git y Github

Como hemos visto antes, Git es un sistema de control de versiones distribuido (DVCS) creado por el equipo de Linux. Actualmente, se utiliza en muchos servidores de control de versiones, como GitHub, BitBucket o GitLab, para almacenar proyectos de forma remota. Pero, si queremos interactuar con estos proyectos o repositorios remotos desde nuestra máquina local, debemos instalar Git localmente y utilizar los diferentes comandos que proporciona. En este documento aprenderemos cómo instalar Git y cómo utilizar algunos de los comandos básicos.

Instalación de Git en tu máquina

1. Instalación y configuración de Git

La instalación de Git depende del sistema operativo en el que queremos instalarlo.

Para sistemas Linux, solo tenemos que ejecutar el comando especificado para instalar Git. Por ejemplo, en sistemas Ubuntu tenemos que ejecutar este comando:

```
sudo apt-get install git
```

Para Windows y Mac, tenemos que ir a la página web de Git y descargar la versión adecuada. En el caso de Mac, también se puede instalar Git a través de la instalación de XCode.

1.1. Configuración de Git

Antes de utilizar los comandos de Git, deberíamos configurar algunas variables predeterminadas en nuestro sistema, para conectarnos fácilmente a los servidores y almacenar nuestras credenciales para conexiones posteriores. Utilizaremos el comando `git config` para almacenar estas variables, y las podemos guardar a tres niveles diferentes:

- **Sistema**: utilizando el parámetro `--system`, la configuración se aplica a todos los usuarios del sistema.
- **Usuario**: utilizando el parámetro `--global`, la configuración se aplica solo al usuario actual del sistema. Esta es la opción que utilizaremos en esta sección.
- **Repositorio**: cada repositorio tendrá sus propios parámetros de configuración de Git.

Primero de todo, definimos nuestro nombre completo con este comando (sustituye "John Doe" por tu nombre real):

```
git config --global user.name "John Doe"
```

A continuación, especificamos el correo electrónico con el que creamos la cuenta de GitHub:

```
git config --global user.email yourEmail@server.com
```

Después, podemos especificar el editor predeterminado de Git. Este paso no es necesario, pero si Git necesita abrir un fichero de texto para mostrar información, este será el editor que utilizaremos. Por ejemplo, podemos utilizar el Bloc de notas en Windows de esta manera:

```
git config --global core.editor notepad
```

Finalmente, debemos especificar cómo Git almacenará nuestras credenciales, de manera que no tengamos que introducirlas cada vez que nos conectemos a los repositorios. El ayudante que utilizamos para guardar las credenciales depende del sistema operativo en el que estamos utilizando Git, pero el comando general es este:

```
git config --global credential.helper <helper>
```

donde `<helper>` depende del sistema operativo:

- Para Windows utilizamos `wincred`
- Para Linux utilizamos `cache`
- Para Mac OSX utilizamos `osxkeychain`

Así que, si queremos configurar el ayudante de credenciales en Windows, por ejemplo, escribimos algo así:

```
git config --global credential.helper wincred
```

De esta manera, ya estamos preparados para utilizar Git, incluso desde diferentes entornos de desarrollo, como veremos en otras secciones. Podemos comprobar la configuración actual utilizando el comando `git config --list`. También podemos comprobar la versión de Git que tenemos instalada actualmente con el comando `git version`.

2. Comandos locales básicos útiles

Veamos ahora algunos comandos que podemos utilizar para trabajar con proyectos locales (sin conectarnos a ningún repositorio o servidor remoto). Estos comandos son útiles tanto para proyectos locales como para proyectos remotos que hayamos descargado previamente, si queremos trabajar localmente con ellos durante un tiempo.

2.1. Crear un repositorio local

Si queremos inicializar o crear un nuevo repositorio local, primero debemos crear la carpeta donde se guardará este proyecto. Después, podemos inicializarla como repositorio Git con este comando (desde dentro de la carpeta del proyecto):

```
git init
```

Esto inicializará esta carpeta como carpeta Git, creando una subcarpeta oculta llamada `.git`, donde se almacenará la base de datos del repositorio. No nos debemos preocupar por esta subcarpeta.

Cada fichero dentro de este repositorio estará en uno de los tres estados mencionados en secciones anteriores (committed, staged o modified), y podemos cambiar el estado de cada fichero escribiendo algunos de los comandos que veremos ahora. También podemos comprobar el estado del repositorio en cualquier momento con el comando `git status` (lo debemos ejecutar desde la carpeta raíz del repositorio). Nos informará si todo está "commitado", o si hay algún fichero con cambios sin guardar.

> **Actividad**
>
> 1. Instala Git en tu ordenador, en la máquina virtual facilitada en clase o en una máquina virtual. En este último caso, puedes descargar una imagen de Linux como por ejemplo Lubuntu y utilizar VirtualBox (o cualquier otro). Para poder utilizar una máquina virtual en tu ordenador, **debes permitirlo en la configuración de la BIOS** (permitir virtualización).
> 2. Configura Git con tu cuenta creada en el apartado anterior.
> 3. Crea una carpeta llamada **GitExercises** en tu sistema. Almacenaremos algunos repositorios dentro de ella. Para empezar, crea dentro de esta carpeta una subcarpeta nueva llamada **MyFirstLocalRepo**, entra dentro de esta carpeta y ejecuta el comando `git init` para inicializar esta carpeta como repositorio Git.

2.2. Añadir o editar ficheros en el repositorio

Si añadimos algún fichero nuevo a la carpeta del repositorio (por ejemplo, un fichero llamado `file.txt`) y ejecutamos el comando `git status`, Git mostrará que hay algunos ficheros que deben ser añadidos al repositorio.
Estos ficheros se encuentran en estado "modificado". Si utilizamos el comando `git add`, el (los) fichero(s) se marcarán como "preparados" (staged). Si solo queremos añadir un único fichero, especificaremos este nuevo fichero como parámetro:

```
git add file.txt
```

No obstante, puede haber muchos cambios en nuestro repositorio. Si queremos añadirlos todos a la vez, utilizamos `.` como parámetro:

```
git add .
```

Después de cada nuevo cambio que hagamos en el repositorio (ya sea añadiendo, editando o eliminando ficheros), debemos repetir este comando para preparar los cambios. Una vez los cambios estén preparados, este es el resultado del comando `git status`:

Como puedes ver en la imagen anterior, podemos utilizar el comando `git rm` para desmarcar este fichero, si queremos:

```
git rm --cached file.txt
```

2.3. Guardar o *commitear* cambios

Después de añadir o preparar los cambios, debemos hacer un último paso para actualizar la base de datos del repositorio. Esta operación es el "**commit**", y la podemos hacer mediante el comando `git commit` . Podemos ejecutarla después de una o varias operaciones de `git add` que hayan preparado uno o más ficheros.

Esta es la estructura general del comando `git commit` :

```
git commit -m "Mi primer commit"
```

El parámetro `-m` nos permite especificar un mensaje de commit. Este mensaje es obligatorio para guardar el commit, de manera que, si queremos recuperarlo más adelante, podemos identificar este mensaje en la lista de commits. Después de commitear los cambios, si ejecutamos `git status` , deberíamos ver que no hay nada por commitear:

```
On branch master
nothing to commit, working tree clean
```

Alternativamente, también podemos utilizar el parámetro `-a` para añadir o preparar automáticamente los cambios antes de commitearlos. Este comando combina un `git add .` y un comando `git commit` :

```
git commit -a -m "Tu mensaje de commit"
```

Mostrar el historial de commits

Si queremos ver el historial de commits de nuestro repositorio, podemos escribir este comando:

```
git log
```

Cada commit tiene una etiqueta que consiste en una secuencia de dígitos y letras. En el ejemplo anterior, nuestro commit se ha etiquetado como `08f4ed1751…` . Esta etiqueta será útil para comprobar el commit más adelante, aunque no es necesario recordar todos estos caracteres, solo el prefijo inicial.

Mostrar los cambios

También podemos ver los cambios entre dos versiones consecutivas del repositorio. Hay muchas maneras de hacerlo:

- `git show` : muestra los cambios realizados en el último commit.
- `git show cb1fd6f8` : muestra los cambios hechos en el commit etiquetado con el prefijo `cb1fd6f8` (como se puede ver, no necesitamos escribir toda la etiqueta).
- `git diff` : muestra los cambios realizados en la última versión que todavía no se ha commiteado.
-

> **Actividad**
>
> Haz los siguientes cambios en el repositorio **MyFirstLocalRepo** que has creado en el ejercicio anterior:
>
> 1. Crea un fichero nuevo llamado `file.txt` con el texto "Mi primer fichero de texto". Guarda los cambios en este fichero.
> 2. Ejecuta el comando `git add .` para preparar este fichero.
> 3. Ejecuta el comando `git commit` con el mensaje "Mi primer commit" para guardar los cambios a la base de datos.
> 4. Edita el fichero `file.txt` y añade una segunda línea con tu nombre.
> 5. Ejecuta el comando `git commit –a –m` para preparar y commitear automáticamente los cambios con el mensaje "Mi segundo commit".
> 6. Ejecuta el comando `git log` para ver el historial de commits. Deberías ver algo similar a esto:

```
commit 08f4ed1751…
Author: John Doe <john@example.com>
Date: Wed Apr 14 15:00 2023
    Mi primer commit
commit cb1fd6f8…
Author: John Doe <john@example.com>
Date: Wed Apr 14 16:00 2023
    Mi segundo commit
```

> 7. Ejecuta el comando `git show` para ver los cambios hechos en el último commit. Deberías ver algo parecido a esto:
>
> Los nuevos cambios se muestran en verde si han sido añadidos (en este caso, tu nombre al final del contenido del fichero), o en rojo si han sido eliminados.

Etiquetar commits

Podemos añadir manualmente etiquetas a un commit determinado, de manera que lo podemos encontrar fácilmente más adelante cuando queramos mostrar sus cambios. Utilizamos el comando `git tag` seguido del nombre de la etiqueta:

```
git tag v1.0
```

Esto se aplica al último commit enviado. A continuación, podemos mostrar los cambios de este commit con este comando:

```
git show v1.0
```

Si queremos etiquetar un commit que no es el último, entonces debemos especificar la etiqueta anterior de ese commit (o su prefijo inicial), después de la nueva etiqueta que queremos asignarle:

```
git tag v1.0 cb1fd6f8
```

2.4. Deshacer cambios

¿Y si queremos volver a un commit anterior y deshacer los cambios hechos en el (los) último(s) commit(s)? Podemos utilizar el comando `git reset` . Este comando se puede utilizar de muchas maneras, pero aquí explicaremos una de ellas: debemos identificar la etiqueta del commit que queremos establecer como el actual y después escribir este comando:

```
git reset --hard 0305afd
```

donde `0305afd` es el prefijo de la etiqueta del commit que queremos establecer como nuestro estado actual activo.

2.5. El fichero .gitignore

En cada repositorio Git, podemos añadir manualmente un fichero llamado `.gitignore` . Es solo un fichero de texto que contiene una lista de ficheros y carpetas que deben ser ignorados cuando subamos nuevos cambios. Por ejemplo, si estamos trabajando en un proyecto C#, no necesitamos subir ficheros `.exe` al repositorio, ya que podemos recompilar el proyecto de nuevo. Así que podemos editar este fichero y especificarlo así:

```
*.exe
```

Esto saltará todos los ficheros `.exe` de la carpeta principal del proyecto. De la misma manera, podemos añadir tantos ficheros y carpetas como necesitemos en este fichero. Por ejemplo:

```
node_modules/
*.exe
*.tmp
```

Esto omite la carpeta `node_modules` y todos los ficheros `.exe` o `.tmp` en la carpeta raíz. Aquí puedes encontrar ficheros `.gitignore` típicos para muchos tipos diferentes de proyectos, como proyectos Node, Laravel, etc.

NOTA: El fichero `.gitignore` **NO** excluye ficheros que ya han sido *commiteados* previamente. Por ejemplo, si le decimos a este fichero que ignore ficheros `.exe` pero previamente hemos commiteado un fichero `.exe` al repositorio, este fichero no se eliminará del mismo.

3. Trabajando con repositorios remotos

Ahora que hemos aprendido cómo añadir y editar contenido en un repositorio local, veamos cómo conectarnos a un repositorio remoto de GitHub para descargar o subir los cambios. Primeramente, si queremos trabajar con repositorios remotos almacenados en GitHub, necesitaremos crear este repositorio remoto en GitHub.

3.1. Clonar repositorios

Una vez tengamos nuestro repositorio creado en GitHub, necesitaremos copiarlo a nuestra máquina local. Esta operación se conoce como una operación de **clonación**, y la hacemos con el comando `git clone`, especificando la URL del repositorio, la cual se puede recuperar del botón *Clone or download* en el mismo repositorio.

Ejemplo de página principal del repositorio:

Si tuviéramos un repositorio en GitHub, el comando adecuado para clonarlo sería así:

```
git clone https://github.com/johndoe/test
```

Este comando creará una carpeta llamada **test** en el directorio desde el cual estamos ejecutando esta comando, así que asegúrate de ejecutarla dentro de la carpeta donde quieras colocar tu proyecto.

> **Actividad**
> Clona el repositorio donde se almacenan estos apuntes:
> https://github.com/arturoalbero/Materiales-EDE-2425

3.2. Actualizar los cambios remotos en local

Una vez tengamos el repositorio clonado localmente, si estamos trabajando en equipo o gestionando el mismo repositorio desde diferentes ordenadores, quizás necesitaremos descargar los últimos cambios de este repositorio para actualizar nuestra copia local. Este paso es esencial para asegurarnos que tenemos los contenidos actualizados antes de hacer nuevos cambios.

Para hacer esto, simplemente utilizamos el comando `git pull` desde la carpeta del repositorio:

```
git pull
```

Esto descarga automáticamente los últimos cambios y actualiza los ficheros afectados.

3.3. Subir los cambios locales al repositorio remoto

Si tenemos nuestro repositorio local actualizado y hacemos nuevos cambios a cualquier fichero, podemos subir estos cambios al repositorio remoto. Los pasos necesarios son los siguientes:

1. Haz cambios en el(los) fichero(s) deseado(s).
2. Marca los ficheros como preparados con el comando `git add .` que ya hemos visto antes.
3. Haz un **commit** de los cambios localmente con el comando `git commit` que también hemos visto antes.
4. Sube este commit (o los últimos commits, si hay más de uno) con el comando `git push` .

```
git push
```

Ejercicio 3:

1. Clona el repositorio de GitHub **MyFirstRepo** que habrás creado en el documento anterior. Clónalo dentro de la misma carpeta principal donde estás creando el resto de repositorios locales de este documento. Verás una nueva carpeta llamada **MyFirstRepo** que contiene todos los elementos de tu repositorio remoto.
2. Aplica estos cambios:
    - Añade un nuevo fichero llamado `shopping_list.txt` con una lista de artículos que quieras comprar.
    - Sube este fichero al repositorio remoto (recuerda, primero añade los cambios, después haz el commit y finalmente haz el push).
3. Ve a GitHub y comprueba que el nuevo fichero se ha subido correctamente.
4. Ve a una carpeta diferente del ordenador y clona una copia nueva del mismo repositorio.

5. Desde esta segunda carpeta, añade un nuevo fichero llamado `to_do.txt` y añade algunas tareas pendientes para las próximas semanas.
6. Sube los cambios al repositorio remoto.
7. Vuelve a tu carpeta original **MyFirstRepo** y haz un comando `git pull`. Comprueba si el nuevo fichero **to_do.txt** se ha descargado en esta copia local.

3.4 Crear un repositorio remoto a partir de nuestro repositorio local

Para crear un repositorio a partir de nuestro repositorio local, lo que debemos hacer es conectar nuestro repositorio local a un repositorio remoto vacío (para evitar conflictos de fusión). Para ello, seguimos estos pasos:

1. **Creamos el repositorio de Git si no lo tenemos con `git init`**
2. **Añadimos los archivos que queramos y hacemos al menos un commit**
3. **Creamos un Nuevo Repositorio en GitHub:**

- Ve a GitHub.com e inicia sesión en tu cuenta.
- Haz clic en el signo **+** en la esquina superior derecha y selecciona **"New repository"** (Nuevo repositorio).
- Dale un **nombre** a tu repositorio (normalmente, es buena idea que coincida con el nombre de tu carpeta local, pero no es obligatorio).
- Puedes añadir una **descripción** si quieres.
- **¡Importante!** No marques las opciones "Initialize this repository with a README", "Add .gitignore" ni "Choose a license". Así evitamos conflitos.
- Finalmente, haz clic en el botón **"Create repository"** (Crear repositorio).

4. **Conecta tu Repositorio Local con el de GitHub:**

Después de crear el repositorio en GitHub, verás una página con instrucciones. Busca la sección que dice "...or push an existing repository from the command line." (o sube un repositorio existente desde la línea de comandos). Usaremos los siguientes dos comandos:

- **Añade el origen remoto:** Este comando le dice a tu Git local dónde está tu repositorio de GitHub. Asegúrate de reemplazar `TU_NOMBRE_DE_USUARIO` y `TU_NOMBRE_DEL_REPOSITORIO` con tu información real.

```
git remote add origin https://github.com/TU_NOMBRE_DE_USUARIO/TU_NOMBRE_DEL_REPOSITORIO
```

Ejemplo: `git remote add origin https://github.com/octocat/mi-proyecto-genial.git`

- **Verifica el remoto (Opcional pero recomendado):** Puedes comprobar que el remoto se añadió correctamente:

```
git remote -v
```

Deberías ver algo así:

```
origin  https://github.com/TU_NOMBRE_DE_USUARIO/TU_NOMBRE_DEL_REPOSITORIO.git (fetch)
origin  https://github.com/TU_NOMBRE_DE_USUARIO/TU_NOMBRE_DEL_REPOSITORIO.git (push)
```

`fetch` es recoger, `push` es subir.

5. **Sube tu Repositorio Local a GitHub:**

- `git branch -M main` : Renombra tu rama actual a `main` . GitHub usa `main` como nombre de rama principal por defecto. Si tu rama local ya se llama `main` o prefieres mantenerla como `master` , puedes omitir este comando o ajustarlo (por ejemplo, `git push -u origin master`).
- `git push -u origin main` : Sube tu rama local `main` al repositorio remoto llamado `origin` . La opción `-u` (o `--set-upstream`) es crucial porque establece la rama remota de seguimiento, lo que significa que en el futuro, los comandos `git push` y `git pull` sabrán con qué rama remota interactuar sin que tengas que especificarla cada vez.

```
git branch -M main
git push -u origin main
```

Tabla resumen de comandos básicos

Comando	Utilidad
`git init`	Inicializa un nuevo repositorio Git en la carpeta actual.
`git status`	Muestra el estado del repositorio y los ficheros modificados o no commiteados.
`git add <fichero>`	Añade un fichero específico al área de "stage" para el siguiente commit.
`git add .`	Añade todos los ficheros modificados al área de "stage".

Comando	Utilidad
`git commit -m "<mensaje>"`	Realiza un commit de los cambios con un mensaje descriptivo.
`git commit -a -m "<mensaje>"`	Añade y commitea automáticamente los cambios sin necesidad de `git add .`
`git log`	Muestra el historial de commits del repositorio.
`git show`	Muestra los cambios realizados en el último commit o en un commit específico.
`git diff`	Compara los cambios entre las versiones no commiteadas del repositorio.
`git rm --cached <fichero>`	Elimina un fichero del área de "stage" sin borrarlo del sistema.
`git reset --hard <commit>`	Restaura el estado del repositorio a un commit anterior.
`git tag <etiqueta>`	Añade una etiqueta a un commit específico.
`git clone <URL>`	Clona un repositorio remoto en la máquina local.
`git pull`	Actualiza la copia local con los cambios del repositorio remoto.
`git push`	Sube los cambios locales al repositorio remoto.
`git config --global user.name "<nombre>"`	Configura el nombre de usuario para los commits.
`git config --global user.email "<email>"`	Configura el correo electrónico para los commits.
`git config --list`	Muestra la configuración actual de Git.
`git version`	Muestra la versión de Git instalada.

Uso de ramas en Git

En esta segunda parte, nos centraremos en el trabajo con **ramas** (**branches**) de Git, una de las funcionalidades más poderosas de este sistema de control de versiones. Las ramas permiten trabajar en funciones o mejoras de manera independiente sin afectar la rama principal del proyecto (normalmente llamada `main` o `master`). Así, podemos desarrollar, probar y modificar nuevas funcionalidades sin interferir en la versión estable del proyecto.

1. ¿Qué es una rama en Git?

Una rama en Git es esencialmente una secuencia de commits que comienza a partir de un punto específico en la historia del repositorio. La rama principal del proyecto es generalmente `main` o `master`, pero podemos crear tantas ramas como necesitemos para desarrollar nuevas características o corregir errores de manera paralela.

Cuando trabajamos con ramas, podemos hacer cambios en una rama sin que afecten a las otras, y posteriormente podemos combinar las ramas para integrar esos cambios en el proyecto principal.

2. Crear y cambiar de ramas

2.1. Crear una nueva rama

Para crear una nueva rama, utilizamos el comando `git branch`, seguido del nombre de la nueva rama. Esta nueva rama se creará a partir del estado actual del repositorio, es decir, del commit donde estamos en este momento.

```
git branch nombre-rama
```

Por ejemplo, si quieres crear una rama para desarrollar una nueva funcionalidad llamada "nueva-funcion", puedes hacerlo así:

```
git branch nueva-funcion
```

2.2. Cambiar de rama

Una vez creada la nueva rama, si queremos trabajar en ella, debemos cambiar a ella. Para ello, utilizamos el comando `git checkout` seguido del nombre de la rama a la que queremos cambiar:

```
git checkout nueva-funcion
```

A partir de este momento, cualquier cambio que hagamos se aplicará a esta rama, sin afectar la rama principal (`main` o `master`).

Nota: A partir de Git 2.23, también podemos utilizar `git switch` para cambiar de rama, una alternativa más clara a `git checkout` :

```
git switch nueva-funcion
```

2.3. Crear y cambiar de rama en un solo paso

Es posible crear una rama y cambiar a ella en un solo paso con este comando:

```
git checkout -b nombre-rama
```

Por ejemplo:

```
git checkout -b nueva-funcion
```

Este comando crea la rama `nueva-funcion` y automáticamente nos cambia a ella.

3. Trabajar en una rama

Una vez estemos trabajando en una nueva rama, podemos hacer modificaciones, añadir ficheros y realizar commits de la misma manera que lo haríamos en la rama principal. Estos cambios quedarán aislados dentro de la rama y no afectarán a ninguna otra rama hasta que decidamos combinarlas.

3.1. Ver las ramas disponibles

Podemos ver todas las ramas existentes en nuestro repositorio utilizando este comando:

```
git branch
```

La rama en la que estamos trabajando actualmente estará marcada con un asterisco `*` .

4. Fusionar ramas (merge)

Una vez hayamos terminado el desarrollo o la corrección de errores en nuestra rama, es momento de fusionar los cambios con la rama principal. Este proceso se denomina **fusión** o **merge**.

Antes de realizar la fusión, es recomendable asegurarse de que estamos en la rama donde queremos aplicar los cambios (normalmente `main` o `master`). Para ello, cambiamos a la rama con:

```
git checkout main
```

Una vez estamos en la rama principal, podemos utilizar el comando `git merge` para fusionar los cambios de la rama secundaria en la que hemos estado trabajando (por ejemplo, `nueva-funcion`):

```
git merge nueva-funcion
```

Si no hay conflictos entre los cambios, Git combinará automáticamente las dos ramas. Si hay conflictos (por ejemplo, si se han modificado las mismas líneas de código en ambas ramas), Git nos avisará y tendremos que resolverlos manualmente.

5. Resolver conflictos de fusión

Cuando aparecen conflictos durante una fusión, Git marca las líneas de código en los ficheros afectados que tienen conflictos. Deberíamos revisar estas líneas manualmente, escoger qué versión queremos mantener, y después marcar los conflictos como resueltos con `git add`. Finalmente, hacemos un commit para completar la fusión:

```
git add <fichero>
git commit
```

6. Borrar ramas

Después de fusionar los cambios de una rama secundaria a la rama principal, es una buena práctica eliminar esa rama para mantener el repositorio limpio. Podemos hacerlo con el comando `git branch -d`:

```
git branch -d nueva-funcion
```

Si la rama no ha sido fusionada, Git no permitirá borrarla. Si estamos seguros de que queremos eliminarla aunque no haya sido fusionada, podemos forzar la eliminación con `git branch -D` :

```
git branch -D nueva-funcion
```

7. Trabajo con ramas remotas

Cuando trabajamos con repositorios remotos (como GitHub o GitLab), las ramas locales no se suben automáticamente al servidor remoto. Para subir una nueva rama, debemos utilizar el comando `git push` especificando el nombre de la rama:

```
git push origin nueva-funcion
```

Una vez la rama esté subida al repositorio remoto, otros colaboradores podrán acceder a ella.

> **Actividad**
> Trabajando con ramas en Git
>
> En este ejercicio, aprenderás a trabajar con ramas para desarrollar nuevas funcionalidades o hacer correcciones sin afectar el código de la rama principal. Seguirás el proceso completo de creación de ramas, fusión de cambios y resolución de conflictos, todo en un escenario práctico.
>
> **Escenario:**
>
> Añade al repositorio antes creado, MyFirstRepo, un fichero llamado `index.html` a la rama principal (`main`). Tu objetivo es añadir dos nuevas funcionalidades de manera independiente, trabajando en ramas separadas y, después, fusionarlas con la rama principal. Documenta los pasos con capturas de pantalla y comparte la memoria del ejercicio.
> Puedes emplear el siguiente código como base del fichero:

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Archivo</title>
</head>
<body>

</body>
</html>
```

> **Pasos a seguir:**
>
> 1. **Clona el repositorio y sitúate en la rama principal:**

```
# Si aún no lo has hecho
git clone <URL_DE_TU_REPOSITORIO_MyFirstRepo>
cd MyFirstRepo
git checkout main # Asegúrate de estar en la rama main
```

> Crea el archivo `index.html` con el contenido proporcionado en la rama `main` y súbelo al repositorio remoto.
>
> 2. **Crea una nueva rama para añadir un título a la página web:**
>
> - Crea la rama `añadir-titulo` y cambia a esta rama:

```
git checkout -b añadir-titulo
```

> - Abre el fichero `index.html` y añade un título dentro de la etiqueta `<body>` :

```
<h1>Bienvenidos a mi página web</h1>
```

> - Guarda los cambios y haz un commit:

```
git add index.html
git commit -m "Añadido título a la página web"
```

> 3. **Crea otra rama para añadir un párrafo de introducción:**

- Cambia a la rama principal (`main`) y después crea una nueva rama `añadir-introduccion` :

```
git checkout main
git checkout -b añadir-introduccion
```

- Modifica el fichero `index.html` y añade un párrafo de introducción bajo el título:

```
<p>Esta es una página web de ejemplo creada para aprender Git.</p>
```

- Guarda los cambios y haz un commit:

```
git add index.html
git commit -m "Añadido párrafo de introducción"
```

4. **Fusiona las ramas con la rama principal:**

- Vuelve a la rama principal:

```
git checkout main
```

- Fusiona la rama `añadir-titulo` con la rama principal:

```
git merge añadir-titulo
```

- A continuación, fusiona la rama `añadir-introduccion` con la rama principal:

```
git merge añadir-introduccion
```

5. **Resuelve un conflicto de fusión (opcional):**

Si hubiera conflictos durante la fusión (por ejemplo, si el título y el párrafo hubieran sido añadidos en las mismas líneas del fichero `index.html`), resuélvelos de la siguiente manera:

- Abre el fichero conflictivo (`index.html`) y modifica las líneas marcadas por Git para mantener las dos funcionalidades (el título y el párrafo).
- Después, marca los conflictos como resueltos:

```
git add index.html
```

- Haz un commit para completar la fusión:

```
git commit -m "Resueltos conflictos y fusionadas ramas"
```

6. **Borra las ramas:**

Después de fusionar las ramas, puedes borrarlas para mantener el repositorio limpio:

```
git branch -d añadir-titulo
git branch -d añadir-introduccion
```

Tabla resumen de comandos de ramas en Git

Comando	Utilidad
`git branch`	Lista todas las ramas del repositorio.
`git branch <nombre-rama>`	Crea una nueva rama con el nombre especificado.
`git checkout <nombre-rama>`	Cambia a la rama especificada.
`git checkout -b <nombre-rama>`	Crea una nueva rama y cambia a ella en un solo paso.
`git switch <nombre-rama>`	Cambia a la rama especificada (alternativa a `git checkout`).
`git merge <nombre-rama>`	Fusiona la rama especificada con la rama actual.
`git branch -d <nombre-rama>`	Borra una rama local después de fusionarla.
`git branch -D <nombre-rama>`	Fuerza la eliminación de una rama no fusionada.
`git push origin <nombre-rama>`	Sube una nueva rama al repositorio remoto.
`git pull origin <nombre-rama>`	Descarga e integra los cambios de una rama remota en la rama local.
`git stash`	Guarda los cambios no commiteados temporalmente para cambiar de rama.
`git stash apply`	Recupera los cambios guardados anteriormente con `git stash`.

Integración de github y git en un IDE

Autenticación en Git con Tokens Personales (PAT)

A partir del 13 de agosto de 2021, GitHub ya no acepta contraseñas de cuenta para la autenticación de operaciones Git. Es necesario utilizar un **PAT (Token de Acceso Personal)**. Puedes seguir el siguiente método para añadir un PAT a tu sistema.

Crear un Token de Acceso Personal en GitHub

1. Desde tu cuenta de GitHub, ve a **Settings** → **Developer Settings** → **Personal Access Token** → **Tokens (classic)** → **Generate New Token**.
2. Introduce tu contraseña, rellena el formulario y haz clic en **Generate Token**.
3. Copia el Token generado, que tendrá un formato similar a:
   `ghp_sFhFsSHhTzMDreGRLjmks4Tzuzgthdvfsrta` .

Métodos según tu sistema operativo:

Para Windows:

1. Ve al **Administrador de credenciales** desde el **Panel de Control** → **Credenciales de Windows** → busca `git:https://github.com` → **Editar**.
2. Sustituye la contraseña por tu **GitHub Personal Access Token**.
3. Si no encuentras `git:https://github.com` , haz clic en **Agregar una credencial genérica**.
   - La dirección de Internet será `git:https://github.com` .
   - Introduce tu nombre de usuario, y para la contraseña, utiliza tu **Personal Access Token**.
4. Haz clic en **Aceptar** y ya estará configurado.

Para macOS:

1. Haz clic en el icono de la lupa (Spotlight) en la parte derecha de la barra de menú.
2. Escribe **Acceso a Llaveros** y pulsa la tecla Intro para abrir la aplicación.
3. Busca `github.com` dentro de **Acceso a Llaveros**.
4. Encuentra la entrada de la contraseña de Internet para `github.com` y edita o elimina la entrada según sea necesario.
   Ya deberías haber terminado.

Para sistemas basados en Linux:

1. Necesitas configurar el cliente local de GIT con un nombre de usuario y dirección de correo electrónico:

```
$ git config --global user.name "tu_nombre_de_usuario_github"
$ git config --global user.email "tu_correo_github"
$ git config -l
```

2. Una vez GIT esté configurado, puedes empezar a utilizarlo para acceder a GitHub. Ejemplo:

```
$ git clone https://github.com/TU-NOMBRE-DE-USUARIO/TU-REPOSITORIO
> Cloning into `TU-REPOSITORIO`...
Username: <introduce tu nombre de usuario>
Password: <introduce tu contraseña o token de acceso personal de GitHub>
```

3. Ahora, puedes guardar (cachear) el token en tu ordenador para recordarlo:

```
$ git config --global credential.helper cache
```

4. Si necesitas borrar la memoria caché en algún momento:

```
$ git config --global --unset credential.helper
$ git config --system --unset credential.helper
```

5. Puedes verificar con la opción `-v` cuando hagas un pull:

```
$ git pull -v
```

6. Para clonar en Debian u otras distribuciones de Linux:

```
git clone https://<tu_token_aquí>@github.com/<usuario>/<repositorio>.git
```

Integración con diferentes IDEs

Para IDEs de JetBrains (IntelliJ, PhpStorm, WebStorm, etc.):

1. Consulta la página de ayuda del IDE que utilices para más información sobre cómo iniciar sesión con un Token.
2. A continuación, tienes un resumen rápido de cómo hacerlo en **IntelliJ**:
   - Abre los **ajustes** pulsando ⌘Cmd0/CtrlAlt0 (las teclas pueden variar).
   - Selecciona **Version Control | GitHub**.
   - Haz clic en el botón **Añadir**.
   - Elige la opción **Iniciar Sesión con Token**.
   - Introduce el token generado en el campo de texto correspondiente.
   - Haz clic en **Añadir Cuenta**.

Información extraída de stackoverflow.

Integrar Git y GitHub en VSCode

Una de las herramientas más potentes de VSCode es la **integración con Git**, que te permite gestionar repositorios locales y remotos (como GitHub) directamente desde el editor.

Crear un repositorio Git

Para empezar a trabajar con Git, primero necesitas crear un repositorio:

1. **Inicializar un repositorio**
   Abre la carpeta de tu proyecto en VSCode y haz clic en el **panel de control de Git** (icono de ramas). Allí encontrarás la opción "Initialize Repository". Esto creará un repositorio Git en tu carpeta de trabajo.
2. **Añadir ficheros al repositorio**
   Una vez creado, todos los ficheros no seguidos aparecerán en la sección "Changes". Para añadirlos a tu commit, selecciona los ficheros y haz clic en el icono "+".
3. **Hacer un commit**
   Después de añadir ficheros, puedes crear un **commit**, que es un punto de restauración dentro de tu proyecto. Escribe un mensaje descriptivo en el campo de texto y haz clic en el icono de confirmación.

Clonar un repositorio de GitHub

Si ya tienes un repositorio en GitHub y quieres trabajar en él desde VSCode, sigue estos pasos:

1. **Clonar el repositorio**
   Ve a `Ctrl+Shift+P` o `Cmd+Shift+P` y escribe "Git: Clone". Introduce la URL del repositorio GitHub y selecciona la carpeta donde quieres guardarlo.
2. **Hacer cambios y hacer un commit**
   Una vez clonado el repositorio, puedes hacer cambios en los ficheros como de costumbre. Después añade los ficheros cambiados a Git, crea un commit y finalmente envíalo a GitHub.

Push y pull

1. **Hacer push**
   Para enviar los cambios a tu repositorio remoto (GitHub), haz clic en el icono de sincronización (arriba en la barra de Git) o escribe "Git: Push" en la paleta de comandos.
2. **Hacer pull**
   Si otros colaboradores han hecho cambios en el repositorio remoto, puedes descargar esos cambios haciendo un `pull`. Escribe "Git: Pull" en la paleta de comandos o haz clic en el icono de sincronización.

Otros comandos útiles de Git

- **Git: Checkout to...** permite cambiar de rama.
- **Git: Merge Branch...** permite fusionar ramas.
- **Git: Create Branch...** crea una nueva rama para trabajar.

Integrar IntelliJ IDEA y PyCharm con Git y GitHub

Configurar Git en IntelliJ IDEA o PyCharm

1. **Abrir las preferencias del IDE**
   Ve a `File > Settings` (o `Preferences` en macOS) y luego busca `Version Control > Git`.
2. **Comprobar la ruta de Git**
   El IDE intentará detectar automáticamente la ubicación del binario de Git en tu sistema. Si no lo hace correctamente, especifica la ruta manualmente. La ruta típica para Git es:
   - Windows: `C:\Program Files\Git\bin\git.exe`
   - macOS y Linux: `/usr/bin/git` o `/usr/local/bin/git`
3. **Comprobar la configuración**
   Haz clic en el botón `Test` para verificar que Git está configurado correctamente y funcionando. Si todo está bien, verás el mensaje "Git executable is found".

Crear un repositorio Git en IntelliJ IDEA o PyCharm

Inicializar un repositorio Git

1. **Abrir un proyecto o crear uno nuevo**
   Abre el proyecto existente o crea uno nuevo en IntelliJ o PyCharm.
2. **Inicializar un repositorio Git**
   Ve a `VCS` (Version Control System) en el menú superior y selecciona `Enable Version Control Integration`. Aparecerá una ventana emergente donde debes seleccionar `Git` como sistema de control de versiones.
3. **Añadir ficheros al repositorio**
   Una vez inicializado el repositorio, podrás ver los **ficheros no seguidos** en el panel de "Version Control". Para añadirlos al seguimiento de Git, selecciónalos, haz clic con el botón derecho y selecciona `Git > Add`.
4. **Hacer un commit**
   Después de añadir los ficheros, podrás hacer un **commit**, que es un punto de restauración del proyecto. Ve a `VCS > Commit` o haz clic en el icono de commit en la barra de control de versiones. Escribe un mensaje descriptivo y confirma el commit.

Conectar con GitHub

Para trabajar con GitHub, primero tienes que **conectar tu cuenta al IDE**.

Configurar el cuenta de GitHub

1. **Acceder a GitHub desde el IDE**
   Ve a `File > Settings` (o `Preferences` en macOS) y busca `Version Control > GitHub`.
2. **Iniciar sesión en GitHub**
   Haz clic en `Add Account`. Esto abrirá una ventana donde podrás iniciar sesión con tu cuenta de GitHub. Si lo prefieres, también puedes utilizar un **token de acceso personal (PAT)** que puedes generar en tu cuenta de GitHub.

Crear un repositorio GitHub desde IntelliJ IDEA o PyCharm

1. **Crear el repositorio remoto**
   Una vez tienes Git integrado en tu proyecto, puedes subir tu proyecto a GitHub directamente desde el IDE. Ve a `VCS > Import into Version Control > Share Project on GitHub`.
2. **Añadir una descripción**
   Especifica el nombre del repositorio y su descripción (opcional). El IDE creará automáticamente el repositorio remoto en GitHub y hará un `push` de los ficheros al repositorio.

Clonar un repositorio GitHub en IntelliJ IDEA o PyCharm

Si ya tienes un repositorio en GitHub y quieres clonarlo para trabajar en él localmente:

1. **Clonar el repositorio**
   Ve a `File > New > Project from Version Control`. Selecciona `Git` e introduce la URL del repositorio GitHub que quieres clonar.
2. **Seleccionar la ubicación**
   Indica dónde quieres guardar el proyecto clonado en tu ordenador. Haz clic en `Clone` y el IDE descargará el repositorio en la ubicación seleccionada.

Push y pull en GitHub

1. **Hacer push de los cambios a GitHub**
   Una vez has hecho cambios y creado commits, puedes enviar esos cambios a tu repositorio remoto en GitHub haciendo clic en el icono `Push` en la barra de control de versiones o yendo a `VCS > Git > Push`. Esto enviará tus commits al repositorio remoto.

2. **Hacer pull de los cambios**
   Si alguien más ha hecho cambios en el repositorio remoto, puedes hacer un `pull` para actualizar tu repositorio local. Ve a `VCS > Git > Pull` para descargar los cambios.

Otras funcionalidades de Git dentro de IntelliJ IDEA y PyCharm

- **Ramas (branches)**: Puedes crear nuevas ramas para trabajar en funcionalidades o correcciones específicas sin afectar la rama principal (normalmente `main` o `master`). Ve a `VCS > Git > Branches > New Branch` para crear una nueva.
- **Merge**: Cuando terminas de trabajar en una rama, puedes fusionarla con la rama principal. Selecciona la rama con la que quieres trabajar en `VCS > Git > Branches` y luego selecciona `Merge` para fusionar las ramas.
- **Resolución de conflictos**: Si hay conflictos entre diferentes cambios (por ejemplo, si tú y otro colaborador habéis hecho cambios en el mismo fichero), el IDE te mostrará una interfaz para resolver los conflictos de manera visual.

> **Actividad**
> Haz una memoria en la que guardes los pasos exactos que haces para integrar github en un IDE a tu elección.

Reto cooperativo de la Unidad de Programación 04

Herramientas para el desarrollo de software

Formación del equipo

Para esta unidad de programación, la formación de los equipos se hará a través del método *Dinámica de los colores*. Observa la tabla:

¿Con qué colores te identificas más? Escoge en la encuesta en FORMS tu primera y tu segunda opción. En base a ella, se crearán los equipos de la clase intentando tener representados todos los colores.

Tarea

Continúa el proyecto que se te ha asignado. Este proyecto fue comenzado por otros equipos, que se encargaron del **Diseño de los diagramas de comportamiento** y del **Diseño de los diagramas de clases y propuesta de traducción a entidad relación**. Tu labor es:

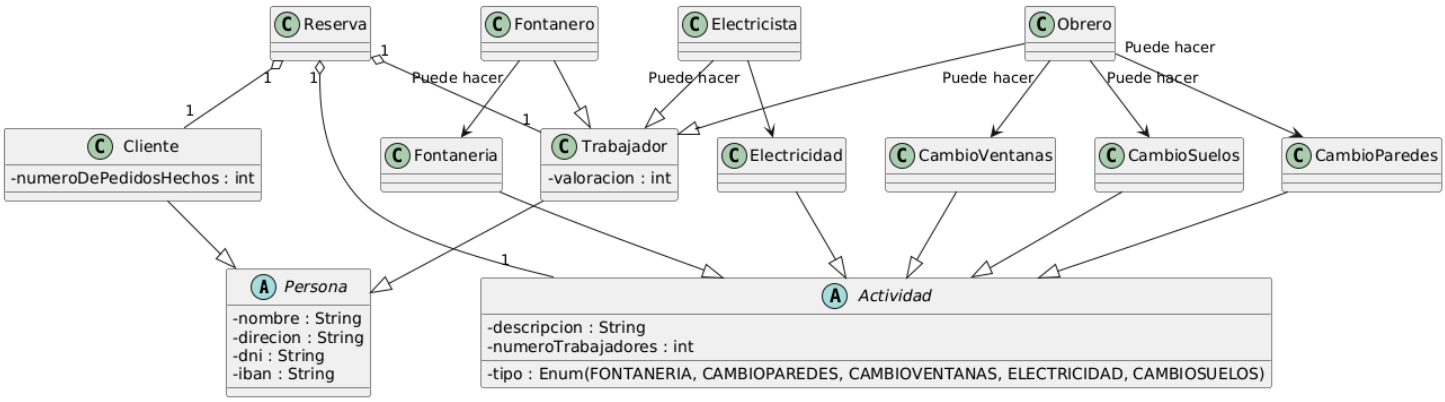
- Corregir posibles errores del producto entregado por el equipo anterior antendiendo a la especificación dada
- Traducir las clases a Java, sin implementar los métodos. Usa Maven para construir el proyecto.

- Traducir las clases a Python o Javascript o Typescript, sin implementar los métodos
- Subir el proyecto a un repositorio de github. Usa un repositorio diferente para cada versión del proyecto.
- Realizar una memoria de cómo configurar el IDE (IntelliJ, VS Code o Pycharm) para el proyecto.

Reto individual

> **Actividad**
> Dado el siguiente diagrama de clases (trabajado en la unidad anterior), realiza las siguientes tareas.

- Crea un proyecto en IntelliJ IDEA usando Maven (puedes elegir community edition o Ultimate). Usa Java 21 LTS.
- Traduce las clases a código Java. Añade los atributos y métodos que consideres oportunos (no es necesario implementar los métodos).
- Genera constructores, getters y setters usando las herramientas incluidas en IntelliJ IDEA
- Crea una versión mejorada del diagrama con el código nuevo que has generado.
- Crea un repositorio en Github
- Conecta tu proyecto con el repositorio de Github
- Repite lo mismo, pero usando Python, Javascript o Typescript



Rubrica

	ÍTEM	Criterio Evaluación	PESO
	Creación del proyecto en Maven	2a, 2b, 2c, 2d	1
	Adecuación del proyecto	2a, 2b, 2c, 2d	1
	Traducción del diagrama al código JAVA	5e	2
	Generación correcta de métodos	2e	1
	Traducción del código al diagrama	5f	1

	ÍTEM	Criterio Evaluación	PESO
	Repositorio de Github	4f	1
	Traducción del proyecto a otro lenguaje	2e, 2f, 2g	2
	Presentación de la memoria	TODOS	1

Se debe entregar un enlace al repositorio y una memoria explicando el proceso punto por punto

EXAMEN (tipo Test)

1. ¿Qué es un compilador?
   - o A) Un programa que convierte código fuente en lenguaje máquina
   - o B) Un editor de texto especializado en programación
   - o C) Un entorno gráfico para depuración de errores
   - o D) Una herramienta para documentar proyectos de software
2. ¿Qué sistema utiliza el lenguaje de programación Java para ejecutarse?
   - o A) Intérprete
   - o B) Máquina virtual
   - o C) Compilador
   - o D) Ensamblador
3. ¿Cuál es la diferencia entre un IDE (Entorno de desarrollo integrado) y un editor de código como Visual Studio Code?
   - o A) Un IDE incluye herramientas adicionales para la depuración, compilación y administración de proyectos, mientras que un editor de código es más simple
   - o B) Un IDE es un editor de texto sin formato, mientras que un editor de código tiene herramientas avanzadas
   - o C) No hay ninguna diferencia entre un IDE y un editor de código
   - o D) Un editor de código solo permite escribir código en lenguajes específicos
4. ¿Qué es un sistema de control de versiones?
   - o A) Un sistema que almacena archivos en la nube
   - o B) Un sistema que facilita la creación de bases de datos
   - o C) Un sistema que registra y gestiona los cambios realizados en el código fuente de un proyecto
   - o D) Un sistema para detectar errores en el código fuente
5. ¿Cuál de estas características de GIT no es cierta?
   - o A) Es un sistema de control de versiones
   - o B) Solo funciona con una cuenta de Github
   - o C) Solo crea versiones de archivos modificados, para el resto usa enlaces
   - o D) Trabaja asignado estados a los diferentes archivos del repositorio
6. ¿Qué es un pull en GitHub?
   - o A) Una solicitud para eliminar un repositorio
   - o B) Una operación para descargar y fusionar cambios de un repositorio remoto
   - o C) Una función para crear un backup de los archivos
   - o D) Una herramienta para analizar la calidad del código

7. ¿Qué operaciones se pueden hacer con un repositorio en Git?
   - o A) Solo se pueden leer archivos
   - o B) Solo se pueden añadir archivos nuevos
   - o C) Se pueden añadir, modificar y eliminar archivos, además de gestionar el historial de versiones
   - o D) Solo se pueden compartir archivos con otros usuarios
8. ¿Qué es un intérprete?
   - o A) Un programa que convierte código fuente en código máquina de una vez
   - o B) Un programa que convierte el código fuente línea por línea durante su ejecución
   - o C) Un programa que optimiza el rendimiento del sistema operativo
   - o D) Un programa que analiza errores de sintaxis en el código fuente
9. ¿Qué es un scriptlet?
   - o A) La parte del código en HTML de un archivo JSP
   - o B) La parte del código en Python de un archivo JSP
   - o C) La parte del código en Java de un archivo JSP
   - o D) Otro nombre para los archivos JSP
10. ¿Qué es DENO?
    - o A) Un runtime para Javascript
    - o B) Un runtine para RUST
    - o C) Un runtine solo para Typescript
    - o D) El antecesor de NODE.js
11. ¿Cuál de estos IDE es adecuado para código en C# sin necesidad de extensiones?
    - o A) Rider
    - o B) VS CODE
    - o C) IntelliJ IDEA ULTIMATE
    - o D) Netbeans
12. ¿Con qué lenguaje relacionamos principalmente al gestor Maven?
    - o A) Con Rust
    - o B) Con Java
    - o C) Con HTML
    - o D) Con Typescript

Cuaderno de ejercicios 04

Completa los siguientes tutoriales

- VS CODE de HolaMundo
- IntelliJ Idea
- Git y Github de MoureDev

Responde a las siguientes preguntas de tipo test

¿Qué es un compilador?
o A) Un programa que convierte código fuente en lenguaje máquina
o B) Un editor de texto especializado en programación
o C) Un entorno gráfico para depuración de errores
o D) Una herramienta para documentar proyectos de software

¿Qué sistema utiliza el lenguaje de programación Java para ejecutarse?
o A) Intérprete
o B) Máquina virtual
o C) Compilador
o D) Ensamblador

¿Cuál es la diferencia entre un IDE (Entorno de desarrollo integrado) y un editor de código como Visual Studio Code?
o A) Un IDE incluye herramientas adicionales para la depuración, compilación y administración de proyectos, mientras que un editor de código es más simple
o B) Un IDE es un editor de texto sin formato, mientras que un editor de código tiene herramientas avanzadas
o C) No hay ninguna diferencia entre un IDE y un editor de código
o D) Un editor de código solo permite escribir código en lenguajes específicos

¿Qué es un sistema de control de versiones?
o A) Un sistema que almacena archivos en la nube
o B) Un sistema que facilita la creación de bases de datos
o C) Un sistema que registra y gestiona los cambios realizados en el código fuente de un proyecto
o D) Un sistema para detectar errores en el código fuente

¿Cuál de estas características de GIT no es cierta?

o A) Es un sistema de control de versiones

o B) Solo funciona con una cuenta de Github

o C) Solo crea versiones de archivos modificados, para el resto usa enlaces

o D) Trabaja asignado estados a los diferentes archivos del repositorio

¿Qué es un pull en GitHub?

o A) Una solicitud para eliminar un repositorio

o B) Una operación para descargar y fusionar cambios de un repositorio remoto

o C) Una función para crear un backup de los archivos

o D) Una herramienta para analizar la calidad del código

¿Qué operaciones se pueden hacer con un repositorio en Git?

o A) Solo se pueden leer archivos

o B) Solo se pueden añadir archivos nuevos

o C) Se pueden añadir, modificar y eliminar archivos, además de gestionar el historial de versiones

o D) Solo se pueden compartir archivos con otros usuarios

Introducción al uso de Docker

Docker desglosado

Este apartado es un **tutorial opcional** para aprender a dominar Docker en su totalidad. Algunos de los conceptos que se verán ya se han tratado en el apartado anterior. Esto se debe a que este documento trabaja tomando como referencia el siguiente vídeo de youtube:

Este es el vídeo que vamos a tomar como referencia

> **Actividad**
> Realiza una memoria de todo el proceso, hasta configurar tu servidor MySQL en Docker.

Contenedor

Los contenedores son una forma de empaquetar una aplicación y todas sus dependencias.
De esta forma, se consigue una mayor portabilidad y se mejora la posibilidad de compartir la aplicación en diferentes equipos, facilitando el desarrollo y despliegue de aplicaciones.

Los contenedores de Docker se almacenan en repositorios, ya sean privados o públicos, como dockerhub, que es el repositorio oficial.

Para unificar las herramientas de desarrollo de un equipo, un contenedor almacena todas las dependecias necesarias. Se construyen sobre una imagen de Linux. De esta forma, se evitan errores relacionados con compatibilidad y dependencias, al tenerlas almacenadas en imágenes. Por lo tanto, los contenedores son un tipo de virtualización.

Para entender la diferencia entre una máquina virtual y un contenedor, debemos saber que un ordenador se divide en diferentes capas: la capa de hardware, el kernel y la capa de aplicaciones. Con una máquina virtual, se virtualizan tanto el kernel como la capa de aplicaciones. El contenedor, por su parte, solo virtualiza la capa de aplicaciones, empleando el kernel del sistema operativo anfitrión (Linux o Mac OSX) o, en el caso de windows, WSL2 (Windows Subsystem for Linux).

Al virtualizar solamente la capa de aplicaciones, los contenedores son más rápidos y ocupan menos que las máquinas virtuales, por lo que son más apropiados para el desarrollo.

Instalar Docker

Para instalar Docker debes ir a su página web y descargar Docker Desktop. Docker Desktop incluye herramientas como Docker-compose y CLI (la interfaz línea de comandos).

Manejo de imágenes

Para manejar Docker, emplearemos la línea de comandos. Para que funcione, debemos tener ejecutándose Docker Desktop, programa que habilita el uso de CLI.

```
$ docker images
```

Este comando muestra todas las imágenes descargadas. Nos muestra los siguientes campos:

- **Repository**: Nombre de la imagen.
- **Tag**: Etiqueta de la imagen.
- **Image Id**: Identificador de la imagen.
- **Created**: Información sobre cuándo fue creada la imagen.
- **Size**: Tamaño de ocupa la imagen.

```
$ docker pull node
$ docker pull node:lts-alpine3.19
```

En el primer caso, descarga la imagen de `node` etiquetada como `latest`. En el segundo caso, descarga la imagen de `node` etiquetada como `lts-alpine3.19`. En ambos casos, descargará el contenido del repositorio en caso de que no haya sido descargado previamente. El contenido de las imágenes está dividido en módulos para hacer más eficiente la descarga de imágenes, ya que diferentes imágenes pueden compartir algunos módulos, o la misma imagen podría estar etiquetada de diferentes formas (cosa que suele ocurrir con aquella etiquetada con `latest`).

En ocasiones, es necesario especificar la plataforma (especialmente si no se corresponde con la nuestra). Por ejemplo, si estamos usando un Macbook Air con el procesador M1 y queremos descargar una imagen de `mysql` debemos emplear el siguiente comando:

```
$ docker pull --platform linux/x86_64 mysql
```

Para borrar una imagen que hayamos descargado, empleamos el siguiente comando:

```
$ docker image rm node:lts-alpine3.19
```

En este caso, borraremos la imagen de `node` etiquetada con `18` que habíamos descargado antes.

Manejo de contenedores

Los contenedores son el elemento principal de Docker y se crean empleando una imagen como base.

```
$ docker pull mongo
$ docker create mongo
```

En este caso, primero descargamos la imagen de mongo con el comando `pull` y, después, creamos un contenedor con el comando `create`. Este comando nos devuelve un id, que debemos copiar y añadir al siguiente comando:

```
$ docker start [id]
```

En este caso, sustituimos `[id]` por la id que habíamos copiado y ejecuta el contenedor, devolviéndonos la propia id de nuevo. Si ejecutamos el siguiente comando:

```
$ docker ps
```

Veremos todos los contenedores en ejecución. Si añadimos el argumento `-a` veremos todos los contenedores en la máquina, tanto en ejecución como no.

Este comando nos devuelve una tabla similar a la que obteníamos con `$ docker images`, que nos revela la siguiente información:

- **Container ID**: La id, en formato reducido, del contenedor.
- **Image Command**: La imagen base del contenedor.
- **Created**: La fecha de creación del contenedor.
- **Status**: El estado (activo o inactivo) del contenedor.
- **Ports**: Los puertos que usa el contenedor.
- **Name**: El nombre del contenedor. Docker asigna nombres aleatorios al contenedor, pero podemos asignarlo nosotros de forma manual con el siguiente comando:

```
$ docker create --name mimongo mongo
```

En este caso, creará una imagen basada en mongo y llamada `mimongo`. Para ejecutarla, podemos usar `$ docker start` y añadir al final el nombre o la id del contenedor.

```
$ docker stop mimongo
$ docker rm mimongo
```

El primer comando sirve para detener la ejecución del contenedor llamado `mimongo`. En lugar del nombre, también se puede usar la id. El segundo comando sirve para borrar el contenedor `mimongo` y, de la misma forma que antes, el nombre se puede sustituir por la id.

Es recomendable usar nombres personalizados, ya que facilita el uso de los contenedores.

Port Mapping

En telemática y redes informáticas, los puertos son como ventanas virtuales que permiten que una computadora se conecte con varias otras al mismo tiempo. Cada ventana está numerada del 0 al 65,535 y sirve para dirigir la información entrante al programa correcto que está esperándola.

El puerto que aparecía cuando ejecutábamos el comando `$ docker ps` y teníamos un contenedor corriendo era un puerto interno del contenedor, pero cerrado y no accesible desde nuestro equipo. Para poder acceder a un puerto de un contenedor, debemos mapearlo a uno de nuestro equipo.

```
$ docker create -p27017:27017 mongo
```

Con este comando, mapeamos el puerto 27017 del contenedor a nuestro puerto 27017. Es una práctica común mapear los puertos internos del contenedor al mismo número en nuestro equipo y, en caso de no poderse, a un número similar.

En la instrucción `-p27017:27017`, el primer puerto es el de nuestra máquina y el segundo el del contenedor. El nombre extendido del puerto de nuestra máquina es, en realidad, `0.0.0.0:27017` o `localhost:27017`.

Además del comando anterior, también podemos no especificar un puerto de nuestro equipo y que Docker lo asigne arbitrariamente usando `-p27017`. Sin embargo, es preferible asignar los puertos de manera manual.

De esta forma, podremos acceder al contenedor a través de su puerto. Para ello, abrimos un navegador y nos dirigimos a la dirección de nuestro equipo asociada al contenedor, como por ejemplo http://localhost:27017. Mientras tanto, en el terminal, podemos acceder a los logs internos del contenedor de la siguiente forma:

```
$ docker logs mimongo
$ docker logs --follow mimongo
```

En el primer caso, vemos los logs del contenedor `mimongo` en el momento dado en el que ejecutamos el comando. En el segundo caso, el terminal se queda escuchando los logs internos del contenedor `mimongo`.

Docker Run

Docker Run es un comando que sirve para crear y ejecutar un contenedor en un solo paso. Es decir, es una combinación de los comandos `pull`, `create` y `start`. Todos los parámetros anteriores son compatibles.

```
$ docker run mongo
```

Descarga la imagen de mongo si no está presente en el sistema, crea el contenedor y lo ejecuta. Muestra los logs del contenedor, por lo que si cerramos el terminal cerraremos el contenedor.

```
$ docker run -d mongo
```

El argumento `-d` indica que se va a ejecutar en modo "detached", que significa despegado. Esto quiere decir que vamos a poder seguir introduciendo comandos, ya que no se verán los logs.

```
$ docker run --name mimongo -p27017 -d mongo
```

Por último, con este comando estamos creando y ejecutando un contenedor de nombre `mimongo` basado en la imagen `mongo` y, además, estamos mapeando el puerto interno 27017 del contenedor a nuestro puerto 27017.

El comando `$ docker run` siempre crea un contenedor nuevo cuando se ejecuta. Si queremos ejecutar un contenedor ya creado, debemos usar `$ docker start` y el nombre o la id del contenedor.

Variables de entorno y cómo conectarse a un contenedor

En node.js existe una librería llamada mongoose que nos permite conectarnos a bases de datos mongo. Imagina la siguiente instrucción de javascript:

```
mongoose.connect('mongodb://nombre:password@localhost:27017/miapp?authSource=admin');
```

En esta instrucción, le estamos diciendo al objeto mongoose que se conecte a `mongoddb` usando como nombre `nombre` y como password `password`. La @ sirve para separar esos datos de dónde vamos a conectarnos, en este caso a `localhost:27017' que es nuestro equipo en su puerto 27017. Finalmente, el URI nos indica a qué parte del programa nos conectamos, en e` miapp `es la aplicación y después de ? se colocan los parámetros de configuración,` authSource=admin`.

Para poder realizar correctamente esta conexión, debemos haber configurado las **variables de entorno** del contenedor. Cada imagen tiene sus propias variables de entorno, que debemos consultar en el mismo repositorio. En el caso de la imagen de mongo, solamente necesitamos configurar las siguientes variables de entorno para que funcione todo:

```
MONGO_INITDB_ROOT_USERNAME
MONGO_INITDB_ROOT_PASSWORD
```

Para ello, empleamos el siguiente comando:

```
$ docker create -p27017:27017 --name mimongo
-e MONGO_INITDB_ROOT_USERNAME=nombre
-e MONGO_INITDB_ROOT_PASSWORD=password
mongo
```

El argumento `-e` indica que lo que viene a continuación es una variable de entorno. Para inicializarlas, colocamos su nombre, el signo = y el valor que queremos asignarle. Recuerda que las variables de entorno se colocan antes de la imagen. Aunque en el ejemplo estén en diferentes líneas, se trata de un solo comando.

Dockerfile, o cómo crear imágenes personalizadas

Los archivos dockerfile sirven para construir imágenes personalizadas. Dentro de él se escriben instrucciones para crear imágenes que ejecutarán nuestro código, basadas generalmente en otras imágenes.

```
FROM node:lts-alpine3.19
RUN mkdir -p /home/app
COPY . /home/app
EXPOSE 3000
CMD ["node", "/home/app/index.js"]
```

Suponiendo que index.js es un archivo que hemos creado nosotros, el anterior archivo dockerfile hace lo siguiente:

- `FROM node:lts-alpine3.19` : Le indica a Docker cuál es la imagen sobre la que creamos nuestra imagen propia.
- `RUN mkdir -p /home/app` : Crea directorio llamado "app" dentro del directorio "/home". La opción -p le indica al comando mkdir que cree el directorio y cualquier directorio padre que aún no exista.
- `COPY . /home/app` : Copia el contenido de la dirección `.` de nuestro ordenador en la dirección `/home/app` de la imagen. `.` representa el directorio en el que se encuentra el archivo dockerfile.
- `EXPOSE 3000` : Expone el puerto 3000, de forma similar a como si hiciéramos `-p3000:3000` en los comandos vistos en los apartados anteriores.
- `["node", "/home/app/index.js"]` : Ejecuta dentro de la imagen el comando `node /home/app/index.js` . Esto permite que nuestro código se ejecute en la imagen.
  Con estas instrucciones, puedes probar a ejecutar tu propio código javascript en un contenedor de node, en lugar de tener que descargarte una versión de node en tu ordenador.

Para montar la imagen, empleamos el siguiente comando:

```
$ docker build -t miapp:1 .
```

El comando `$ docker build` construye la imagen usando el archivo dockerfile como referencia. El argumento `-t` sirve para etiquetar la imagen. En este caso, `miapp` es el nombre que le damos a la imagen y `1` la etiqueta que le asignamos. Finalmente, el '.' sirve para indicarle dónde está el archivo dockerfile, en este caso, indica el directorio actual.

Para probar que esto funciona bien, vamos a usar el siguiente código (archivo index.js):

```
const express = require('express');

var app = express();

app.get('/', function(req, res) {
  res.send('hello world');
});
app.listen(3000);
```

En este caso, necesitamos el módulo `express` que instalaremos con la orden `$ npm install express` en el directorio raíz.

Para ejecutar el código a través de la imagen de docker, debemos introducir la siguiente instrucción:

```
$ docker run -p 3000:3000 miapp:1
```

De esta forma, comunicaremos el puerto expuesto dentro de docker usando `EXPOSE 3000` con el puerto 3000 de nuestra máquina. Al ir al navegador y conectarnos a `http://localhost:3000/` obtendremos como respuesta `hello world` .

Redes internas de docker

Para que un contenedor pueda conectarse a otro contenedor, necesitamos definir una red interna. Docker genera algunas de forma automática, aunque si no usamos `docker-compose` deberemos especificarlas de forma manual.

```
$ docker network ls
```

Este comando lista todas las redes de docker.

```
$ docker network create mired
```

Crea una red interna de docker con el nombre `mired` .

```
$ docker network rm mired
```

Borra la red interna de docker llamada `mired` .

Para conectarse entre sí, los contenedores de una misma red se comunican a través de su nombre. Para ello, debemos añadir el argumento `--network` a la creación de cada contenedor. Supongamos que tenemos nuestra imagen personalizada `miapp` creada del paso anterior:

```
$ docker create -p27017:27017 --name mimongo --network mired
-e MONGO_INITDB_ROOT_USERNAME=nombre
-e MONGO_INITDB_ROOT_PASSWORD=password
mongo

$ docker create -p3000:3000 --name app --network mired miapp:1
```

En este caso, al crear los dos contenedores, estamos asignándoles a cada uno la red que hemos creado anteriormente.

A continuación, la línea de javascript para conectarse a mongo desde nuestro contenedor con imagen personalizada:

```
mongoose.connect('mongodb://nombre:password@mimongo:27017/miapp?authSource=admin');
```

Como se puede observar, se sustituye `localhost` (nombre de nuestra máquina) por `mimongo` (nombre del contenedor de MongoDB).

El archivo completo en javascript, `index.js` es como sigue:

```
const express = require('express');
const mongoose = require('mongoose');

var app = express();
mongoose.connect('mongodb://nombre:password@mimongo:27017/miapp?authSource=admin');

const kittySchema = new mongoose.Schema({
    name: String
});
const Kitten = mongoose.model('Kitten', kittySchema);
const silence = new Kitten({ name: 'Silence' });
silence.save();

app.get('/', (req, res) => {
    Kitten.find().then(result => {
        res.send(result);
    }).catch(error=>{
        let data = errorMsg(error);
        res.send(data);
    });
});
app.listen(3000);
```

Se trata de una aplicación `javascript` que usa los módulos `express` y `mongoose` , este último para conectarse a `mimongo` .

Para usar `mongo` , primero se crea un esquema para la base de datos y después se crea el modelo. A través del modelo se hacen todas las peticiones, desde la creación y guardado con `save()` en la base de datos hasta la consulta con `find()` . En este caso, dicha consulta ocurrirá cuando accedamos a `http://localhost:3000/` .

Para lanzarlo bien, debemos ejecutar, aparte del los comandos para lanzar el contenedor de mongo, los siguientes cada vez que hagamos un cambio en nuestro archivo `index.js` :

```
$ docker stop app
$ docker rm app

$ docker build -t miapp:1 .
$ docker create -p3000:3000 --name app --network mired miapp:1
$ docker start app
```

Docker Compose

Como se ha podido observar en las líneas de terminal empleadas en el ejemplo anterior, hasta ahora, para poder emplear un contenedor de docker de forma efectiva hemos tenido que realizar una gran cantidad de pasos. Sin embargo, todo ese proceso se puede simplicar empleando el comando docker-compose y un archivo de configuración de tipo `.yml` que llamaremos 'docker-compose.yml`.

El lenguaje **yml**(pronunciado yámel) se emplea para crear archivos de configuración. En este lenguaje, como ocurre en otros como **python** una indentación adecuada es necesaria para que se ejecute correctamente. Vigila, por lo tanto, los espacios.

```
version: "3.9"
services:
    app:
        build: .
        ports:
            - "3000:3000"
        links:
            - mimongo
    mimongo:
        image: mongo
        ports:
            - "27017:27017"
        environment:
            - MONGO_INITDB_ROOT_USERNAME=nombre
            - MONGO_INITDB_ROOT_PASSWORD=password
```

A continuación, pasamos a explicar las líneas de código.

```
version: "3.9"
```

Esta línea especifica la versión de Docker Compose que se utilizará. En este caso, se está utilizando la versión 3.9.

Esta línea es opcional. Si se omite la declaración de la versión en el archivo docker-compose.yml, Docker Compose intentará inferir automáticamente la versión correcta basándose en las características y sintaxis utilizadas en el archivo.

```
services:
```

Esta línea comienza un bloque que define los servicios que se ejecutarán como contenedores.

```
    app:
        build: .
```

Con estas líneas, se define el servicio `app` , que se construirá a partir del Dockerfile presente en el directorio actual (`.`). Este servicio se ejecutará en un contenedor y representará la aplicación.

```
        ports:
            - "3000:3000"
```

Esta sección especifica el mapeo de puertos para el servicio `app` . El puerto 3000 del contenedor se mapeará al puerto 3000 de la máquina host, lo que significa que la aplicación en el contenedor estará disponible en `localhost:3000` en la máquina host.

```
        links:
            - mimongo
```

Esta sección especifica a qué otros servicios se enlaza el servicio `app`. De esta manera, docker gestiona de forma automática las redes internas.

```
    mimongo:
        image: mongo
```

Se define el servicio `mimongo`, que se creará a partir de la imagen `mongo` disponible en Docker Hub. Este servicio se ejecutará en un contenedor y representará una instancia de MongoDB.

```
        ports:
            - "27017:27017"
```

Esta sección especifica el mapeo de puertos para el servicio `mimongo`.

```
        environment:
            - MONGO_INITDB_ROOT_USERNAME=nombre
            - MONGO_INITDB_ROOT_PASSWORD=password
```

Aquí se definen las variables de entorno para el servicio `mimongo`. Estas variables se utilizan para establecer el nombre de usuario y la contraseña de la base de datos MongoDB. En este caso, el nombre de usuario se establece como `nombre` y la contraseña como `password`.

Para que docker ejecute todos los comandos especificados en el archivo docker-compose.yml debemos usar el siguiente comando:

```
$ docker compose up
```

Se le puede añadir el argumento `-d` para que se ejecute en modo *detached*. Si no lo hacemos, cuando pulsemos `ctrl + C`, se terminará la ejecución de los contenedores, aunque seguirán creados.

Para detener y/o eliminar lo creado con `docker compose up`, se emplea el siguiente comando:

```
$ docker compose down
```

Es importante borrar todo lo creado cuando se realizen cambios, antes de volver a subirlo.

De esta manera se integran todos los comandos anteriores en solamente dos, lo que facilita el trabajo con contenedores de docker. Sin embargo, la información interna de estos contenedores no es persistente, lo que significa que lo que guardemos en la base de datos `mimongo` se borrará al detener la ejecución del contenedor. Además, para poder hacer pruebas en el archivo `index.js` tenemos que estar alternando entre `down` y `up` constantemente. Para solucionar estos dos inconvenientes, tenemos a nuestra disposición la herramienta `volumes`.

Volúmenes

Los volúmenes en Docker son un mecanismo para persistir datos fuera del sistema de archivos de un contenedor. Esto es especialmente útil para bases de datos, archivos de configuración o cualquier dato que deba sobrevivir a la eliminación o reinicio de un contenedor. Los volúmenes permiten compartir datos entre el host y los contenedores, o entre múltiples contenedores.

Tipos de volúmenes

1. **Volúmenes con nombre**:
   - Son gestionados por Docker y se almacenan en un directorio específico del host (generalmente en `/var/lib/docker/volumes` en Linux).
   - Son ideales para persistir datos de manera independiente al ciclo de vida de los contenedores.
2. **Bind mounts**:
   - Permiten mapear un directorio o archivo específico del host a un directorio o archivo dentro del contenedor.
   - Útiles cuando necesitas acceso directo a archivos del host o para desarrollo.
3. **Tmpfs mounts**:
   - Almacenan datos en la memoria RAM del host.

- Son temporales y se eliminan cuando el contenedor se detiene.

Creación y uso de volúmenes

Crear un volumen con nombre

Para crear un volumen con nombre, usa el siguiente comando:

```
docker volume create mi_volumen
```

Esto creará un volumen llamado `mi_volumen` que podrás usar en tus contenedores.

Usar un volumen en un contenedor

Para usar un volumen en un contenedor, utiliza el argumento `-v` o `--mount` en el comando `docker run`. Por ejemplo:

```
docker run -d --name mi_contenedor -v mi_volumen:/ruta/en/contenedor nginx
```

En este caso:

- `mi_volumen` es el nombre del volumen.
- `/ruta/en/contenedor` es la ruta dentro del contenedor donde se montará el volumen.

Usar bind mounts

Para mapear un directorio del host a un contenedor, usa:

```
docker run -d --name mi_contenedor -v /ruta/en/host:/ruta/en/contenedor nginx
```

Aquí:

- `/ruta/en/host` es la ruta absoluta en el host.
- `/ruta/en/contenedor` es la ruta dentro del contenedor.

Gestión de volúmenes a través de Docker CLI

Listar volúmenes

Para ver todos los volúmenes creados en tu sistema, usa:

```
docker volume ls
```

Inspeccionar un volumen

Para obtener detalles sobre un volumen específico, usa:

```
docker volume inspect mi_volumen
```

Esto mostrará información como la ubicación del volumen en el host y su configuración.

Eliminar un volumen

Para eliminar un volumen que ya no necesitas, usa:

```
docker volume rm mi_volumen
```

Si el volumen está en uso por un contenedor, primero debes detener y eliminar el contenedor.

Ejemplo práctico: Persistencia de datos en MySQL

Supongamos que quieres ejecutar un contenedor de MySQL y persistir los datos de la base de datos en un volumen.

1. Crea un volumen para MySQL:

```
docker volume create mysql_data
```

2. Ejecuta el contenedor de MySQL usando el volumen:

```
docker run -d --name mysql_db -e MYSQL_ROOT_PASSWORD=contraseña -v mysql_data:/var/lib/mysql -p 3306:3306 mysql
```

   Aquí:
   - `mysql_data` es el volumen que almacenará los datos de MySQL.
   - `/var/lib/mysql` es la ruta dentro del contenedor donde MySQL guarda sus datos.
3. Verifica que los datos persisten:
   - Detén y elimina el contenedor:

```
docker stop mysql_db
docker rm mysql_db
```

   - Vuelve a crear el contenedor con el mismo volumen:

```
docker run -d --name mysql_db -e MYSQL_ROOT_PASSWORD=contraseña -v mysql_data:/var/lib/mysql -p 3306:3306 mysql
```

   - Los datos de la base de datos seguirán intactos.

Algunas imágenes oficiales ya llevan incluida la creación de volúmenes anónimos (sin nombre) cuando ejecutamos su archivo Dockerfile, y esto es una práctica cada vez más frecuente. Si no estamos seguros, cuando ejecutemos por primera vez el Docker Run podemos comprobar si se ha creado un volumen nuevo (y qué nombre tiene) o bien a través de la línea de comandos o a través de la aplicación Docker Desktop.

> **Actividad**
> Crea una conexión a tu servidor corriendo en Docker desde DBeaver u otro editor SQL. Puedes descargar DBeaver aquí.

Anexo: Express y API REST

`Express` es un marco de aplicación web rápido, minimalista y flexible para `Node.js`. Se utiliza para crear aplicaciones web y API REST de manera sencilla y eficiente.

Una API REST (Representational State Transfer) es un estilo arquitectónico para diseñar sistemas distribuidos y servicios web que se basa en los principios del protocolo HTTP. Aquí hay una descripción detallada de los conceptos clave asociados con una API REST:

1. **Recursos:** En una API REST, los recursos son entidades de datos que pueden ser accedidas o manipuladas mediante el protocolo HTTP. Estos recursos pueden ser cualquier cosa, desde objetos de datos simples hasta entidades más complejas. Por ejemplo, en una aplicación de redes sociales, los recursos podrían ser usuarios, publicaciones, comentarios, etc.
2. **URIs (Identificadores de Recursos Uniformes):** Cada recurso en una API REST se identifica mediante un URI único. Los URIs son las rutas a través de las cuales se accede a los recursos en el servidor. Por ejemplo, `/users`, `/posts`, `/users/123`, etc.
3. **Métodos HTTP:** Los métodos HTTP son verbos que indican la acción que se debe realizar en un recurso dado. Los métodos HTTP comúnmente utilizados en una API REST son:
   - **GET:** Se utiliza para recuperar datos de un recurso.
   - **POST:** Se utiliza para crear un nuevo recurso.
   - **PUT:** Se utiliza para actualizar un recurso existente.
   - **DELETE:** Se utiliza para eliminar un recurso.
4. **Representaciones:** Los datos asociados con un recurso pueden ser representados en diferentes formatos, como JSON, XML, HTML, etc. En una API REST, los clientes pueden solicitar la representación de un recurso en un formato específico utilizando encabezados HTTP como `Accept`.
5. **Estado del Cliente y Estado del Servidor:** En una API REST, el servidor no mantiene ningún estado de cliente entre las solicitudes. Cada solicitud del cliente al servidor debe contener toda la información necesaria para que el servidor comprenda y procese la solicitud. El estado de la aplicación reside completamente en el servidor, y el cliente puede cambiar el estado del servidor enviando solicitudes HTTP.
6. **HATEOAS (Hypertext As The Engine Of Application State):** Este principio de diseño de API REST propone que las respuestas de la API deben incluir enlaces hipertextuales que permitan a los clientes descubrir de manera dinámica y navegar a través de los recursos disponibles. Esto promueve la independencia entre el cliente y el servidor, ya que el cliente puede navegar por la API sin necesidad de conocimiento previo de sus rutas.

En segundo curso trabajarás en profundidad esta materia, pero si despierta tu curiosidad, aquí tienes esta información introductoria.

Ampliación de conocimientos de JSP

Uso de clases personalizadas en un proyecto JSP

Para utilizar una clase Java personalizada en un archivo JSP dentro de un proyecto Jakarta EE en IntelliJ IDEA Ultimate, puedes seguir estos pasos:

1. Colocar la Clase en el Directorio `WEB-INF/classes` (Método Sencillo para Desarrollo)

Este es el método más rápido para probar durante el desarrollo, pero no es el más recomendado para despliegues de producción.

- **Paso 1: Crea tu clase Java.**
  Por ejemplo, crea un archivo `MiClase.java` en tu proyecto.

```java
// src/main/java/com/tuempresa/miproyecto/MiClase.java (o donde sea tu paquete)
package com.tuempresa.miproyecto;

public class MiClase {
    private String mensaje;

    public MiClase(String mensaje) {
        this.mensaje = mensaje;
    }

    public String getMensaje() {
        return mensaje;
    }

    public void setMensaje(String mensaje) {
        this.mensaje = mensaje;
    }

    public String saludar() {
        return "Hola desde MiClase: " + mensaje;
    }
}
```

- **Paso 2: Compila tu clase.**
  IntelliJ IDEA normalmente compila tus clases Java automáticamente. Si no, puedes ir a `Build > Build Project` o `Build > Rebuild Project`. Asegúrate de que el archivo `.class` generado se coloque en la ruta de salida de tu módulo (generalmente `target/classes` si usas Maven/Gradle, y luego IntelliJ lo copia a `WEB-INF/classes` cuando despliegas).
- **Paso 3: Accede a la clase en tu JSP.**

```jsp
<%-- webapp/miPagina.jsp --%>
<%@ page import="com.tuempresa.miproyecto.MiClase" %>
<!DOCTYPE html>
<html>
<head>
    <title>Uso de MiClase en JSP</title>
</head>
<body>
    <h1>Ejemplo de Uso de Clase Propia</h1>

    <%
        // Crear una instancia de MiClase
        MiClase miObjeto = new MiClase("¡Este es un mensaje desde JSP!");

        // Llamar a un método de la clase
        String saludo = miObjeto.saludar();
    %>

    <p><%= saludo %></p>

    <%
        // También puedes guardar el objeto en un ámbito (request, session, applica
        request.setAttribute("miObjeto", miObjeto);
    %>

    <p>Mensaje desde el atributo de request: <%= ((MiClase)request.getAttribute("mi

</body>
</html>
```

2. Usar JavaBeans con `jsp:useBean` (Recomendado para Modelos Simples)

Esta es la forma más idiomática y recomendada para usar objetos Java en JSP, especialmente si tu clase sigue la convención de JavaBeans (constructor sin argumentos y métodos `get` / `set` para sus propiedades).

- **Paso 1: Asegúrate de que tu clase sea un JavaBean.**
  Esto significa que debe tener un constructor público sin argumentos (incluso si tiene otros constructores) y métodos `getPropiedad()` y `setPropiedad()` para cada propiedad que quieras exponer.

```java
// src/main/java/com/tuempresa/miproyecto/Usuario.java
package com.tuempresa.miproyecto;

import java.io.Serializable; // Buena práctica para JavaBeans

public class Usuario implements Serializable {
    private String nombre;
    private int edad;

    // Constructor sin argumentos (obligatorio para JavaBeans)
    public Usuario() {
    }

    // Constructor con argumentos (opcional, pero útil)
    public Usuario(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public int getEdad() {
        return edad;
    }

    public void setEdad(int edad) {
        this.edad = edad;
    }

    public String obtenerSaludoPersonalizado() {
        return "Hola, " + nombre + ". Tienes " + edad + " años.";
    }
}
```

- **Paso 2: Accede a la clase en tu JSP usando `jsp:useBean`.**

```
<%-- webapp/perfilUsuario.jsp --%>
<!DOCTYPE html>
<html>
<head>
    <title>Perfil de Usuario</title>
</head>
<body>
    <h1>Perfil del Usuario</h1>

    <%--
        jsp:useBean busca un objeto de la clase especificada en el ámbito especificado.
        Si no lo encuentra, lo crea e inicializa.
        id: el nombre de la variable que usarás en el JSP
        class: el nombre completo de la clase (incluyendo el paquete)
        scope: el ámbito donde se buscará/almacenará el bean (page, request, session, a
    --%>
    <jsp:useBean id="usuarioBean" class="com.tuempresa.miproyecto.Usuario" scope="reque

    <%--
        Puedes establecer propiedades usando jsp:setProperty,
        ya sea manualmente o automáticamente desde parámetros de request.
    --%>
    <%-- Si viniera de un formulario: <jsp:setProperty name="usuarioBean" property="*"
    <%
        // Para este ejemplo, establecemos las propiedades manualmente
        usuarioBean.setNombre("Ana López");
        usuarioBean.setEdad(30);
    %>

    <p>Nombre: <jsp:getProperty name="usuarioBean" property="nombre" /></p>
    <p>Edad: <jsp:getProperty name="usuarioBean" property="edad" /></p>
    <p>Saludo: <%= usuarioBean.obtenerSaludoPersonalizado() %></p>

</body>
</html>
```

3. Usar JSTL (JSP Standard Tag Library) con EL (Expression Language) y Modelos (Recomendado para Aplicaciones Robustas)

Esta es la forma más moderna, limpia y preferida de trabajar con JSP y Java, ya que separa mejor la lógica de presentación del código Java. Necesitarás agregar las dependencias de JSTL a tu proyecto.

- **Paso 1: Agrega las dependencias de JSTL.**
  Si usas Maven, agrega esto a tu `pom.xml` :

```
<dependencies>
    <dependency>
        <groupId>jakarta.platform</groupId>
        <artifactId>jakarta.jakartaee-api</artifactId>
        <version>10.0</version> <scope>provided</scope>
    </dependency>

    <dependency>
        <groupId>jakarta.servlet.jsp.jstl</groupId>
        <artifactId>jakarta.servlet.jsp.jstl-api</artifactId>
        <version>3.0.0</version> </dependency>
    <dependency>
        <groupId>org.glassfish.web</groupId>
        <artifactId>jakarta.servlet.jsp.jstl</artifactId>
        <version>3.0.1</version> </dependency>

    </dependencies>
```

  Si usas Gradle, agrega a `build.gradle` :

```
dependencies {
    // Jakarta EE API (depende de tu configuración específica)
    providedCompile 'jakarta.platform:jakarta.jakartaee-api:10.0' // O tu versión

    // JSTL para Jakarta EE
    implementation 'jakarta.servlet.jsp.jstl:jakarta.servlet.jsp.jstl-api:3.0.0'
    runtimeOnly 'org.glassfish.web:jakarta.servlet.jsp.jstl:3.0.1'

    // Para Java EE 8 o anterior
    // implementation 'javax.servlet:jstl:1.2'
    // runtimeOnly 'taglibs:standard:1.1.2'
}
```

- **Paso 2: Crea tus clases de modelo (POJOs/JavaBeans).**
  Utiliza la misma clase `Usuario` del ejemplo anterior (que es un JavaBean).
- **Paso 3: Prepara tus datos en un Servlet o Controlador.**
  En una aplicación real, no harías la lógica de negocio directamente en el JSP. Usarías un Servlet o un controlador (por ejemplo, con Spring MVC, JSF, etc.) para preparar los datos y luego pasarlos al JSP.

```java
// src/main/java/com/tuempresa/miproyecto/UsuarioServlet.java
package com.tuempresa.miproyecto;

import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

@WebServlet("/usuarios")
public class UsuarioServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        // Lógica de negocio para obtener datos
        Usuario usuario1 = new Usuario("Carlos Ruiz", 25);
        Usuario usuario2 = new Usuario("María García", 40);

        List<Usuario> listaUsuarios = new ArrayList<>();
        listaUsuarios.add(usuario1);
        listaUsuarios.add(usuario2);

        // Poner los objetos en el ámbito de la solicitud para que el JSP los vea
        request.setAttribute("usuarioPrincipal", usuario1);
        request.setAttribute("usuariosRegistrados", listaUsuarios);

        // Redirigir al JSP para la vista
        request.getRequestDispatcher("/WEB-INF/jsp/mostrarUsuarios.jsp").forward(re
    }
}
```

- **Paso 4: Accede a los datos en tu JSP usando JSTL y EL.**

```
<%-- webapp/WEB-INF/jsp/mostrarUsuarios.jsp --%>
<%@ taglib prefix="c" uri="jakarta.tags.core" %> <%-- O "http://java.sun.com/jsp/jstl/c
<!DOCTYPE html>
<html>
<head>
    <title>Lista de Usuarios</title>
</head>
<body>
    <h1>Detalles del Usuario Principal</h1>
    <p>Nombre: ${usuarioPrincipal.nombre}</p>
    <p>Edad: ${usuarioPrincipal.edad}</p>
    <p>Saludo: ${usuarioPrincipal.obtenerSaludoPersonalizado()}</p>

    <h2>Usuarios Registrados</h2>
    <c:choose>
        <c:when test="${not empty usuariosRegistrados}">
            <ul>
                <c:forEach var="usuario" items="${usuariosRegistrados}">
                    <li>${usuario.nombre} (${usuario.edad} años)</li>
                </c:forEach>
            </ul>
        </c:when>
        <c:otherwise>
            <p>No hay usuarios registrados.</p>
        </c:otherwise>
    </c:choose>

</body>
</html>
```

Nota importante sobre las URIs de las Taglibs:

- Para **Jakarta EE 9+** (JSTL 3.0+), la URI correcta para el core de JSTL es `jakarta.tags.core`.
- Para **Java EE 8 o anterior** (JSTL 1.2), la URI correcta es `http://java.sun.com/jsp/jstl/core`. Asegúrate de usar la URI correcta según la versión de Jakarta EE/Java EE que estés utilizando en tu proyecto de IntelliJ.

Configuración en IntelliJ IDEA Ultimate (Jakarta EE Web Application)

IntelliJ IDEA Ultimate está diseñado para manejar esto de manera muy fluida:

1. **Estructura del Proyecto:** Asegúrate de que tus clases Java estén en la carpeta de fuentes (`src/main/java` si usas Maven/Gradle, o tu carpeta de fuentes configurada).
2. **Artefacto de Despliegue (WAR):**
   - Ve a `File > Project Structure...` (`Ctrl+Alt+Shift+S`).
   - Selecciona `Artifacts` en el panel izquierdo.
   - Asegúrate de tener un artefacto de tipo `Web Application: Exploded` o `Web Application: Archive` (WAR).
   - En la pestaña `Output Layout` , verifica que la salida de tus módulos (clases compiladas) esté incluida. IntelliJ lo hará automáticamente si sigues la estructura de proyecto estándar, colocando tus `.class` files en `WEB-INF/classes` .
3. **Configuración del Servidor de Aplicaciones:**
   - Ve a `Run > Edit Configurations...` .
   - Selecciona tu configuración de `Tomcat Server` (o el servidor de aplicaciones que uses).
   - En la pestaña `Deployment` , asegúrate de que tu artefacto web esté configurado para ser desplegado. La "Application context" será la URL base de tu aplicación (ej., `/miaplicacion`).
4. **Ejecutar/Depurar:**
   - Una vez configurado, puedes ejecutar tu aplicación (`Run > Run 'TuConfiguracionTomcat'`). IntelliJ compilará, empaquetará el WAR y lo desplegará en tu servidor de aplicaciones.

Puntos Clave a Recordar:

- **Compilación:** IntelliJ IDEA compila automáticamente tus clases Java. Asegúrate de que no haya errores de compilación.
- **Classpath:** Cuando se despliega un WAR, el contenedor de servlets (como Tomcat, GlassFish, WildFly) pone las clases de `WEB-INF/classes` y los JARs de `WEB-INF/lib` en el classpath de la aplicación web, lo que permite que tus JSPs las vean.
- **Separación de Responsabilidades (MVC):** Aunque puedes poner lógica en JSP usando scriptlets (`<% %>`), la mejor práctica en aplicaciones Jakarta EE modernas es usar Servlets (o frameworks MVC como Spring MVC/Jakarta Faces/Struts) para manejar la lógica de negocio y pasar los datos a los JSPs (vistas) usando JavaBeans, JSTL y EL. Esto sigue el patrón MVC (Modelo-Vista-Controlador).
- **Rutas de JSP:** Si colocas tus JSPs dentro de `WEB-INF/jsp/` (como en el ejemplo de JSTL), los usuarios no podrán acceder a ellos directamente por la URL, lo cual es una buena práctica de seguridad y un mejor control del flujo de la aplicación a través de Servlets.