

INVESTIGACION INICIAL

||♦] FRONT - END

♦ Vue

Es un **framework open source progresivo** de Javascript para crear interfaces de usuario, o en otras palabras, se trata de una «capa» añadida a Javascript formada por herramientas, convenciones de trabajo y un lenguaje particular que nos permite crear aplicaciones de forma rápida, agradable, sencilla y práctica.

[n] PROPOSITO:

- Permite la creación de **interfaces de usuario y aplicaciones de una sola página** (single-page application o SPA).

[n] CARACTERISTICAS:

- [] La **curva de aprendizaje** corresponde a la mas **sencilla** de los tres frameworks más populares: React, Vue y Angular.
- [] Se trata de un **framework amigable y respetuoso** con las **tecnologías de frontend y los estándares**. Utiliza HTML, CSS y Javascript y es compatible con WebComponents.
- [] Se trata de un **framework progresivo**. Esto significa que es **ideal** para **migrar y adaptar proyectos existentes** hechos en otras tecnologías y pasarlos poco a poco a Vue.
- [] Provee mayor **protagonismo** al enfoque **tradicional «centrado en HTML»**, así como a los **sistemas de plantillas**.

♦ Vuetify

Vuetify es un **framework de componentes** que sigue las especificaciones del **Material Design** de Google.

Está **construido sobre Vue.js**, uno de los frameworks de JavaScript más populares y versátiles.

Proporciona una vasta **colección de componentes UI** listos para usar que son **responsivos y accesibles**, facilitando enormemente la tarea de **diseño de interfaces**.

CARACTERISTICAS:

- **Componentes ricos y listos para usar:** Desde botones, tarjetas, hasta complejos menús y diálogos, Vuetify ofrece todo lo que necesitas para construir interfaces detalladas.
- **Soporte para temas personalizados:** Puedes ajustar fácilmente los colores, tamaños y otros aspectos visuales para que coincidan con la identidad de tu marca.
- **Optimización para móviles:** Todos los componentes están diseñados para ser completamente responsivos y funcionar bien en una variedad de dispositivos.
- **Integración con herramientas de desarrollo:** Vuetify se integra sin problemas con herramientas modernas de desarrollo frontend, como Webpack, y frameworks como Nuxt.js.

◆ Axios

Axion es una librería JavaScript quien puede ejecutarse en el navegador y node.js, la cual nos permite hacer sencillas las operaciones como cliente HTTP.

Se encuentra basada en promesas para node.js y el navegador. Permite configurar y realizar solicitudes a un servidor y procesar fácilmente de las respuestas recibidas.

Es isomorfo, por lo que puede ejecutarse en el navegador y nodejs con el mismo código base. En el lado del servidor usa el modulo nativo http de node.js, mientras que en el lado del cliente (navegador) utiliza XMLHttpRequest.

CARACTERISTICAS:

- **Hace XMLHttpRequest desde el navegador.**
- **Hace peticiones http desde node.js.**
- **Soporta el API de Promesa.**
- **Intercepta petición y respuesta.**
- **Transforma petición y datos de respuesta.**
- **Cancela peticiones.**
- **Transformación automática de datos JSON.**
- **Soporte para proteger al cliente contra XSRF.**

PROCESO DE INSTALACION:

IMPLEMENTACION A TRAVEZ DE CDN

```
<script src='https://unpkg.com/axios/dist/axios.min.js'></script>
```

INSTALACION A TRAVEZ DE NPM

```
npm install axios
```

| ♣️ | EJEMPLO DE IMPLEMENTACION:

RECUPERACION DE TEXTO PLANO

```
var loading = document.getElementById('loading');
var mensaje = document.getElementById('mensaje');

var boton = document.getElementById('carga_ajax');
boton.addEventListener('click', function() {
  loading.style.display = 'block';
  axios.get('texto.txt', {
    responseType: 'text'
  })
  .then(function(res) {
    if(res.status==200) {
      mensaje.innerHTML = res.data;
    }
    console.log(res);
  })
  .catch(function(err) {
    mensaje.innerText = 'Conexión errónea ' + err;
  })
  .then(function() {
    loading.style.display = 'none';
  });
});
```

ACCESO A UN RECURSO API REST [GET]

```
var boton = document.getElementById('json_get');
boton.addEventListener('click', function() {
```

```

loading.style.display = 'block';
axios.get('https://jsonplaceholder.typicode.com/todos/1', {
  responseType: 'json'
})
.then(function(res) {
  if(res.status==200) {
    console.log(res.data);
    mensaje.innerHTML = res.data.title;
  }
  console.log(res);
})
.catch(function(err) {
  console.log(err);
})
.then(function() {
  loading.style.display = 'none';
});
});

```

EJEMPLO DE ENVIO DE DATOS [POST]

```

var boton = document.getElementById('json_post');
boton.addEventListener('click', function() {
  loading.style.display = 'block';
  axios.post('https://jsonplaceholder.typicode.com/posts', {
    data: {
      userId: 1,
      title: 'Post nuevo',
      body: 'Ejemplo de librería Axios.'
    }
  })
  .then(function(res) {
    if(res.status==201) {
      mensaje.innerHTML = 'El Post ha sido guardado bajo el id: ' + res.data.id;
    }
  })
  .catch(function(err) {
    console.log(err);
  })
  .then(function() {
    loading.style.display = 'none';
  });
});

```

◆ i18N en vuetify

I18N o internatilizacion, corresponde al soporte de **internacionalización de los textos** presentes en los **diversos componentes UI de Vue**, propios a botones, tablas, diálogos, paginación, etc..., beneficio de la **implementación del framework Vuetify**.

El cual permite la **utilización de traducciones incorporadas**, propios a **diversos lenguajes predefinidos**, así como la generación de **equivalentes personalizados**.

Facilitando el **cambio de idiomas en tiempo real**.

◆ Pinia

Pinia es una **biblioteca de gestión de estados** para **Vue.js** que utiliza la **API de Composición de Vue 3**. A diferencia de **Vuex**, que usa objetos centrales para almacenar el estado, **Pinia** emplea **módulos para definir estados** más **modulares y fáciles de reutilizar** en diferentes partes de la aplicación.

CARACTERISTICAS:

- **Modularidad:** los estados en Pinia son más modulares que en Vuex, lo que permite una mayor reutilización en diferentes partes de la aplicación. También permite una mejor escalabilidad y mantenimiento del código.
- **Reactividad:** Pinia utiliza la API de Composición de Vue 3, lo que significa que los estados y efectos son más reactivos y se actualizan automáticamente cuando cambian los datos.
- **Tipado fuerte:** Pinia usa TypeScript de forma nativa, lo que permite definir tipos de datos precisos para los estados, evitando errores comunes.
- **Curva de aprendizaje:** aunque la API de Composición puede parecer más compleja que la API de objetos centrales de Vuex, Pinia proporciona una API intuitiva y fácil de aprender para la gestión de estados.

EJEMPLO DE IMPLEMENTACION:

ESTRUCTURA DE TIENDA DE DATOS

```
// stores/counter.js
import { defineStore } from 'pinia'
```

```

export const useCounterStore = defineStore('counter', {
  state: () => {
    return { count: 0 }
  },
  // también se puede definir como
  // state: () => ({ count: 0 })
  actions: {
    increment() {
      this.count++
    },
  },
})

```

IMPLEMENTACION EN COMPONENTE VUE

```

<script setup>
import { useCounterStore } from '@stores/counter'

const counter = useCounterStore()

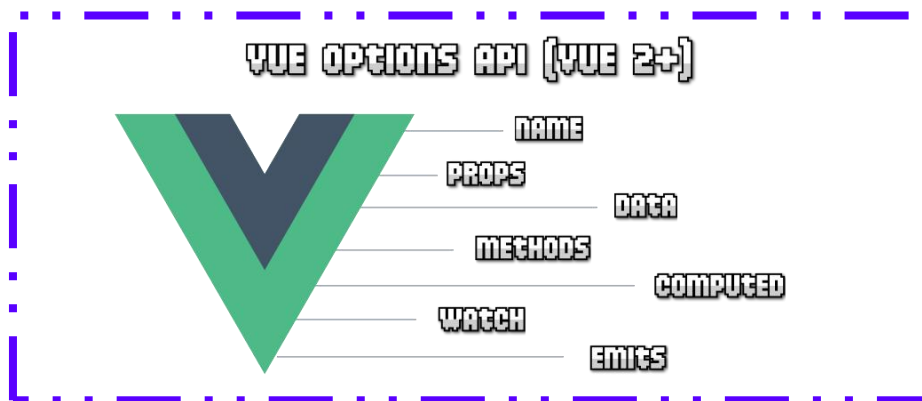
counter.count++
// con autocompletado ✨
counter.$patch({ count: counter.count + 1 })
// o usando una acción en su lugar
counter.increment()
</script>

<template>
  <!-- Accede al estado desde el almacén directamente -->
  <div>Current Count: {{ counter.count }}</div>
</template>

```

◆ Options API en vue

La conocida como «Options API» es la forma tradicional de trabajar con Vue, estando disponible tanto en la versión Vue 2 como en Vue 3. Esta modalidad de API se basa en el uso de un objeto que contiene varias **propiedades clave** para el funcionamiento de los componentes Vue, como son las **propiedades props, data, computed, methods, etc...**



Cada una de estas **opciones** permitiendo **definir un aspecto clave de Vue**.

ESTRUCTURA DE COMPONENTE VUE [Options API]

```
<template>
  <!-- Código HTML -->
</template>

<script>
export default {
  name: "BaseComponent",
  props: {},
  data() {
    return {}
  },
  computed: {},
  methods: {}
}
</script>

<style>
/* Estilos CSS */
</style>
```

ELEMENTOS DE LA OPTIONS API:

OPCION	DESCRIPCION
◆ name	Establece un nombre al componente. De lo contrario aparece como "Anónimo" en DevTools.
◆ props	Lista de atributos (props) aceptados desde el

componente padre.

- ◆ **data** Función que devuelve un objeto con las variables del componente Vue.
- ◆ **computed** Lista de funciones que se ejecutarán cuando se acceda a la propiedad en cuestión.
- ◆ **methods** Lista de funciones (métodos) disponibles en el componente Vue.
- ◆ **watch** Lista de funciones que se disparan cuando detecten cambios en variables con su nombre.
- ◆ **emits** Lista de custom events que pueden ser emitidos desde el componente.



||♦] BACK-END

♦ Metodos HTTP

HTTP define un **conjunto de métodos** de **petición** para indicar la **acción que se desea realizar** para un **recurso determinado**. Aunque estos también pueden ser sustantivos, estos métodos de solicitud a veces son llamados HTTP verbs.

|n] PETICIONES HTTP:

- **] GET:** El método GET solicita una representación de un recurso específico. Las peticiones que usan el método GET sólo deben recuperar datos.
- **] HEAD:** El método HEAD pide una respuesta idéntica a la de una petición GET, pero sin el cuerpo de la respuesta.
- **] POST:** El método POST se utiliza para enviar una entidad a un recurso en específico, causando a menudo un cambio en el estado o efectos secundarios en el servidor.
- **] PUT:** El modo PUT reemplaza todas las representaciones actuales del recurso de destino con la carga útil de la petición.
- **] DELETE:** El método DELETE borra un recurso en específico.
- **] CONNECT:** El método CONNECT establece un túnel hacia el servidor identificado por el recurso.
- **] OPTIONS:** El método OPTIONS es utilizado para describir las opciones de comunicación para el recurso de destino.
- **] TRACE:** El método TRACE realiza una prueba de bucle de retorno de mensaje a lo largo de la ruta al recurso de destino.
- **] PATCH:** El método PATCH es utilizado para aplicar modificaciones parciales a un recurso.

♦ Clases

En programación, una clase es una **plantilla** o **modelo** para **crear objetos**. Define las **propiedades (atributos)** y **comportamientos (métodos)** que los objetos tendrán.

En el **desarrollo backend**, donde se maneja la lógica del servidor, acceso a datos, autenticación, procesamiento, etc..., las **clases se utilizan para organizar y estructurar el código** de forma lógica y reutilizable, a método de la **generación de entidades** del sistema.

- Como parte del backend, se usan clases para representar entidades como usuarios, órdenes o productos, encapsular lógica de negocio, estructurar servicios y facilitar el mantenimiento y la escalabilidad.

APLICACIONES EN BACKEND:

- **[] Representar entidades del sistema** (usuarios, productos, órdenes, etc.).
- **[] Encapsular lógica de negocio** (por ejemplo, calcular impuestos, validar usuarios).
- **[] Interactuar con la base de datos** (por ejemplo, un modelo User).
- **[] Crear servicios reutilizables** (como EmailService, PaymentProcessor).
- **[] Facilitar la modularidad y el mantenimiento del código.**

◆ Interfaces

En **programación orientada a objetos**, una **interfaz** (o protocolo en algunos lenguajes) es una **definición abstracta de métodos y propiedades** que una **clase** debe implementar **sin proporcionar lógica interna**.

Representa un **contrato de comportamiento** que cualquier **clase** concreta debe cumplir.

APLICACIONES EN BACKEND:

- **[] Definir contratos entre componentes** (como servicios, repositorios, adaptadores).
- **[] Permitir el polimorfismo:** múltiples implementaciones que pueden intercambiarse fácilmente.
- **[] Facilitar la inyección de dependencias y lograr código más limpio, probado y desacoplado**

◆ Mappers

Un **mapper** consiste en un **esquema con la estructura de los datos** en la **forma que preferimos**, la cual en momento de recibir **respuesta de la API**, permitirá la **adaptación de los datos recibidos**, al **formato de los inputs** propios a nuestros componentes.

Todo lo que hace el mapper es mapear el input al output.

- Crear **mappers puede ser útil** cuando se tiene que lidiar con **datos de fuentes externas**, de las cuales **no se tiene un control sobre los datos**.
- Los casos de buen uso incluyen Unit Testing y componentes (no sólo componentes de React).

Sin la implementación de los **mappers**, cualquier **cambio en la fuente y estructura de los datos retornados por la API**, puede representar un **incremento significativo de la complejidad**.

- Se requeriría la review de numerosos componentes y unit tests, por tanto, un mapper igualmente ayuda a estandarizar el formato de input requerido por los componentes.

🃏 | EJEMPLO DE IMPLEMENTACION:

IMPLEMENTACION DE MAPPER

```
// Your API Response or another data source
const API_RESPONSE = {
  userName: "Talysson",
  userLastName: "Cassiano",
  birth: "01/04/1992",
  evaluation: {
    average: 4.5,
    total: 20,
  }
}

const apiResponseMapper => data => {
  const {
    userName,
    userLastName,
    birth,
    evaluation
  } = data

  return {
    name: userName,
    lastName: userLastName,
    fullName: `${userName} ${userLastName},
```

```

    birthDate: new Date(birth),
    userRating: evaluation.average,
    ratingsTotal: evaluation.total
  }
}

const userDataMapper = ({ data, origin }) => {
  const mappers = {
    API: apiResponseMapper
  }

  return mappers[origin] ? mappers[origin](data) : data
}

// Good way to write components and tests, having same names in both
// components for the same data
const UserProfile = ({ fullName, birthDate, userRating }) => {...}

describe('UserProfile', () => {
  it('renders with correct props', () => {
    const MOCK_USER = userDataMapper({ data: API_RESPONSE, origin:
'API' })
    ...
  })
})

const UserRating = ({ fullName, userRating, ratingsTotal }) => {... }

describe('UserRating', () => {
  it('renders with correct props', () => {
    const MOCK_USER = userDataMapper({ data: API_RESPONSE, origin:
'API' })
    ...
  })
})

```

◆ Autenticación con JWT

JSON Web Token (JWT) es un estándar abierto (RFC 7519) que define una forma compacta y autónoma de transmitir información de forma segura entre partes como un objeto JSON.

Esta información se puede verificar y es confiable gracias a su firma digital.

Los JWT se pueden firmar mediante un **secreto** (con el algoritmo HMAC) o un **par de claves pública/privada** mediante **RSA** o **ECDSA** .

ESCENARIOS DE USO:

- **[] Autorización:** Este es el escenario más común para el uso de JWT. Una vez que el usuario inicia sesión, cada solicitud posterior incluirá el JWT, lo que le permitirá acceder a rutas, servicios y recursos permitidos con ese token.

El inicio de sesión único (SSO) es una función ampliamente utilizada actualmente en JWT debido a su bajo consumo de recursos y su facilidad de uso en diferentes dominios.

- **[] Intercambio de información:** Los tokens web JSON son una buena forma de transmitir información de forma segura entre partes.

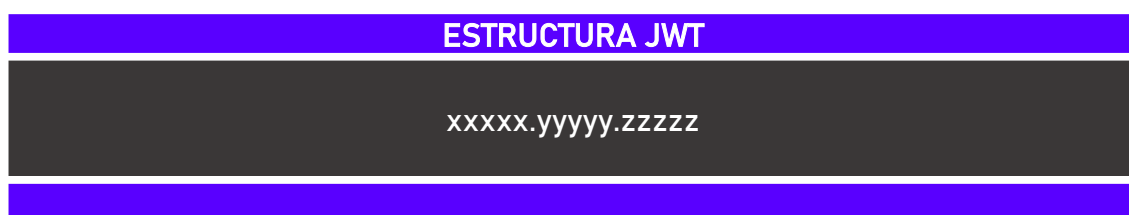
Dado que los JWT se pueden firmar (por ejemplo, mediante pares de claves pública y privada), puede estar seguro de que los remitentes son quienes dicen ser. Además, como la firma se calcula utilizando el encabezado y la carga útil, también puede verificar que el contenido no haya sido manipulado.

ESTRUCTURA:

En su forma compacta, los **tokens web JSON** constan de **tres partes separadas por puntos (.)**, que son:

- ❖ **Encabezamiento [x]**
- ❖ **Carga útil [y]**
- ❖ **Firma [z]**

Por lo tanto, un JWT normalmente luce así:



- **[] Encabezamiento:** El encabezado generalmente consta de dos partes: el tipo de token, que es JWT, y el algoritmo de firma utilizado, como HMAC SHA256 o RSA.



```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

Luego, este JSON se codifica en Base64Url para formar la primera parte del JWT.

- **]] Carga útil:** La segunda parte del token es la carga útil, que contiene las reclamaciones.

Las reclamaciones son declaraciones sobre una entidad (normalmente, el usuario) y datos adicionales. Existen tres tipos de reclamaciones: registradas, públicas y privadas.

- ❖ **Reclamos registrados:** Se trata de un conjunto de reclamos predefinidos que no son obligatorios, pero sí recomendados, para proporcionar un conjunto de reclamos útiles e interoperables.

Algunos de ellos son: iss (emisor), exp (fecha de vencimiento), sub (asunto), aud (audiencia), entre otros.

- ❖ **Reclamos públicos:** Quienes usan JWT pueden definirlos libremente. Sin embargo, para evitar colisiones, deben definirse en el Registro de Tokens Web JSON de la IANA o como un URI que contenga un espacio de nombres resistente a colisiones.

- ❖ **Reclamaciones privadas:** son reclamaciones personalizadas creadas para compartir información entre partes que acuerdan usarlas y no son reclamaciones registradas ni públicas.

ESTRUCTURA PAYLOAD

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

Luego, la carga útil se codifica en Base64Url para formar la segunda parte del token web JSON.

- **I) Firma:** Para crear la parte de la firma, se debe tomar el encabezado codificado, la carga codificada, un secreto, el algoritmo especificado en el encabezado y firmarlo.

Por ejemplo, si desea utilizar el algoritmo HMAC SHA256, la firma se creará de la siguiente manera:

GENERACION DE FIRMA

```

HMACSHA256(
    base64UrlEncode(header) + "." +
    base64UrlEncode(payload),
    secret)

```

La firma se utiliza para verificar que el mensaje no se modificó durante el proceso y, en el caso de tokens firmados con una clave privada, también puede verificar que el remitente del JWT es quien dice ser.

- **II Estructura final:** La salida son tres cadenas URL Base64 separadas por puntos que se pueden pasar fácilmente en entornos HTML y HTTP, y son más compactas en comparación con estándares basados en XML como SAML.

A continuación se muestra un JWT que tiene el encabezado y la carga anterior codificados, y está firmado con un secreto.

RESULTADO DE JWT

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4
gRG9lIiwiaXNTb2NpYWwiOnRydWV9.
4pcPyMD09o1PSyXnrXCjTwXyr4BsezdI1AVTmud2fU4

◆ CORS

El Intercambio de recursos de origen cruzado (CORS), es un mecanismo basado en cabeceras HTTP que permite a un **servidor** indicar cualquier **dominio, esquema o puerto** con un **origen distinto** del suyo desde el que un navegador debería **permitir la carga de recursos**.

Por razones de seguridad, los **navegadores restringen las peticiones HTTP de origen cruzado iniciadas desde scripts**.

- Por ejemplo, **XMLHttpRequest** y la **API Fetch** siguen la **Política Same-origin**.

Esto significa que una aplicación web que utilice esas API solo puede solicitar recursos del mismo origen desde el que se cargó la aplicación, a menos que la respuesta de otros orígenes incluya las cabeceras CORS adecuadas.

◆ **Clean Architecture**

Filosofía de diseño de software que enfatiza la separación de preocupaciones, haciendo que sea más fácil administrar, probar y mantener sistemas de software complejos.

Organiza el código en capas, cada una con responsabilidades y dependencias distintas, y está diseñado para ser independiente de marcos, interfaces de usuario, bases de datos y organismos externos.

PRINCIPIOS BASICOS:

- **]] Separación de preocupaciones:** Diferentes preocupaciones (por ejemplo, UI, lógica empresarial, acceso a los datos) están aisladas en capas distintas.
- **]] Independencia de los frameworks:** La lógica central de la aplicación no depende de frameworks externos, lo que facilita el intercambio de frameworks o su actualización sin afectar la lógica central.
- **]] Testeabilidad:** La separación de las preocupaciones permite que cada capa se pruebe de forma independiente.
- **]] Flexibilidad y mantenimiento:** Al organizar el código en capas, el sistema puede adaptarse a los cambios más fácilmente, y el mantenimiento se vuelve menos engorroso.
- **]] Regla de Dependencia:** Las Dependencias deben apuntar hacia el centro de la aplicación.
- Las capas exteriores dependen de capas internas, pero las capas internas son independientes de las capas externas.

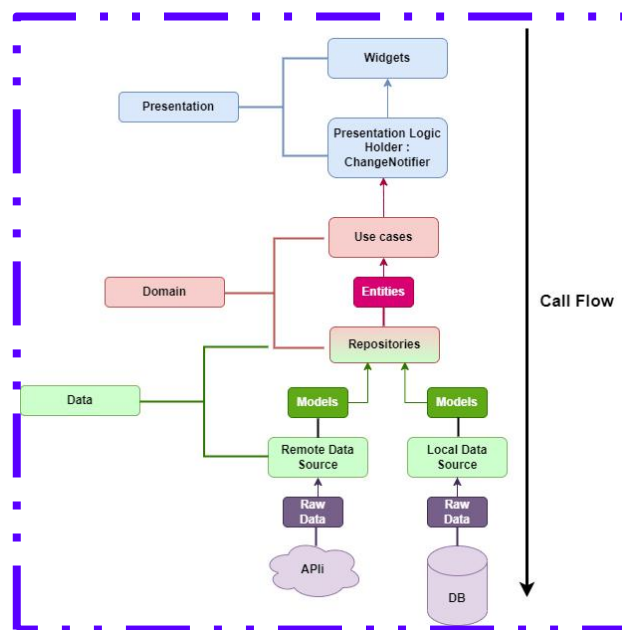
CAPAZ DE LA ARQUITECTURA:

- **|| Capa de datos:** La capa de datos es responsable de el manejo de datos de fuentes externas, como bases de datos, servicios web o sensores de dispositivos. Consiste en las subdivisiones siguientes:
 - ❖ **Fuentes de datos:** Contiene las clases responsables de obtener datos de diversas fuentes, como API REST, almacenamiento local o bases de datos.
 - ❖ **Modelos:** Define los modelos de datos utilizados en toda la aplicación.
 - ❖ **Repositorio:** Actúa como una capa de abstracción sobre las fuentes de datos, proporcionando una API limpia para acceder y administrar datos.
- **|| Capa de dominio:** La capa de dominio contiene la lógica empresarial básica y las reglas de la aplicación. Es independiente de las dependencias externas y consiste en las siguientes subdivisiones:
 - ❖ **Entidades:** Representan objetos de negocio básicos con sus propiedades y comportamientos.
 - ❖ **Casos de uso:** Contiene reglas de negocio y lógica específica de la aplicación, orquestando interacciones entre entidades y fuentes de datos.
 - ❖ **Interfaces de repositorio:** Define interfaces para repositorios utilizados para acceder a los datos, disociando la capa de dominio de fuentes de datos específicas.
- **|| Capa de presentación:** La capa de presentación es responsable de manejar la lógica de la interfaz de usuario y las interacciones. Consiste en las subdivisiones siguientes:
 - ❖ **BLoC (Business Logic Component):** Gestiona la lógica de estado y de negocio de la aplicación, a menudo basada en el patrón BLoC.
 - ❖ **Páginas:** Representa pantallas o puntos de vista individuales en la aplicación, típicamente implementados como widgets de Flutter.
 - ❖ **Widgets:** Componentes de interfaz de usuario reutilizables utilizados en múltiples pantallas o páginas.
 - ❖ **Controladores de UI:** Maneja entradas y eventos de usuario, coordinando con BLoCs y otros componentes para actualizar la interfaz de usuario.
- **|| Capa de datos:** La capa de datos es responsable de administrar las fuentes de datos de la aplicación y proporcionar datos a la capa de dominio. Normalmente incluye varias subpartes clave:
 - ❖ **Repositorios:** Los repositorios actúan como intermediarios entre la capa de dominio y las fuentes de datos (remota o local).
 - ❖ Absuelven las complejidades del acceso a los datos, proporcionando una API simple para que la capa de dominio recupere o persista los datos.

- ❖ **Fuentes de datos (remota y local):** Las fuentes de datos son responsables de la recuperación o almacenamiento de datos reales (datos de redes, API, bases de datos locales o cache).
- ❖ **Modelos (o entidades):** Los modelos dentro de la capa de datos representan la estructura de los datos que se transfieren entre los distintos componentes de la aplicación.

Actúan como contenedores de datos que sostienen la información recuperada de fuentes de datos, transformada para su procesamiento, y pasadas entre diferentes capas de la aplicación.

- ❖ **Mapeadores de datos.**

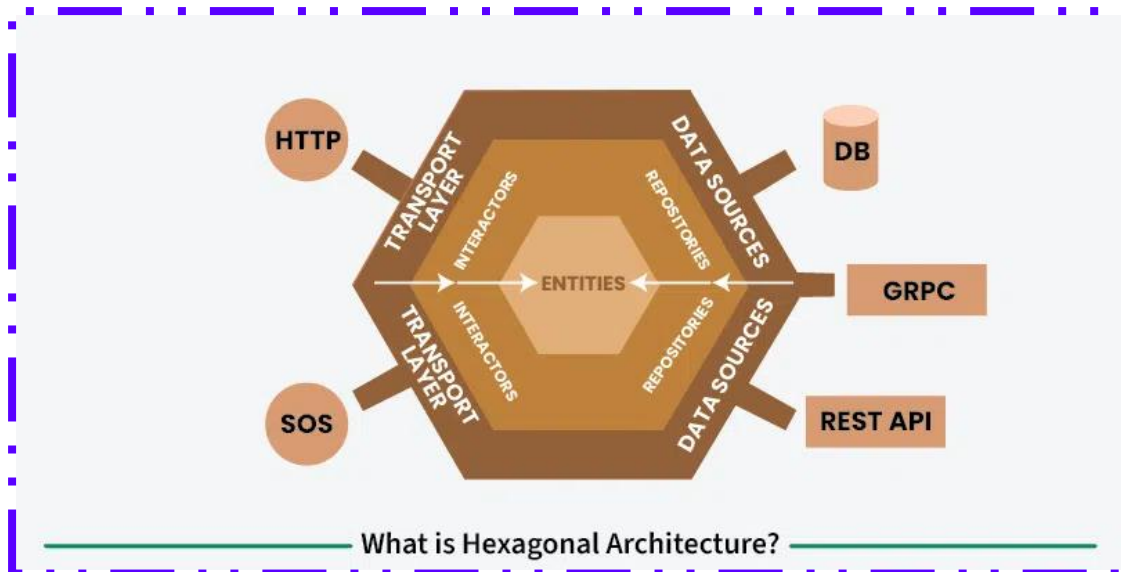


◆ Arquitectura Hexagonal

Hexagonal Architecture, también conocido como **Ports and Adapters Architecture**, es un enfoque de diseño de sistemas dirigido a desvincular la lógica empresarial central de un software de sus interacciones externas.

La imagen representa visualmente esta arquitectura colocando las **entidades básicas (lógica de negocio)** en el **centro del sistema**, protegidas de **dependencias externas** como bases de datos, API y colas de mensajes.

El **núcleo** se **comunica con el mundo exterior** a través de **puertos (interfaces)**, que actúan como límites que definen lo que el sistema central espera con respecto a la entrada y salida.



Alrededor del núcleo se encuentran **adaptadores** que **interactúan con componentes externos**. Estos adaptadores implementan la funcionalidad requerida por los puertos.

- Tiene como principal motivación separar la aplicación en distintas capas o regiones con su propia responsabilidad. De esta manera permitiendo que evolucionen de manera aislada.
- Debido a dispone del sistema separado por responsabilidades, facilita la reutilización de sus componentes.

QUALIDADES PRINCIPALES:

- **Separación de preocupaciones:** La arquitectura impone una clara separación entre la lógica empresarial central y los sistemas externos, como bases de datos, API o interfaces de usuario.

Al aislar el núcleo, asegura que la lógica empresarial no se entrelaza con infraestructuras o preocupaciones externas.

- **Adaptabilidad y Flexibilidad:** La arquitectura Hexagonal presenta de la capacidad de adaptarse a los cambios en los sistemas externos sin requerir cambios en la lógica empresarial central.

Si una empresa decide cambiar de una base de datos relacional a una base de datos NoSQL, solo debe modificarse el adaptador responsable de la interacción de la base de datos. La lógica central no se ve afectada.

- **Testeabilidad mejorada:** Debido a que las dependencias externas como bases de datos o API se extraen, la lógica central se puede probar fácilmente sin necesidad de mockups o simular dependencias complejas, como son bases de datos o servicios web.

El uso de puertos y adaptadores permite burlar interfaces para sistemas externos, permitiendo que el núcleo se ensaye de forma aislada.

- **] Desarrollo paralelo:** Diferentes equipos pueden trabajar independientemente en varias partes del sistema.

Un equipo podría trabajar en la lógica empresarial central, mientras que otro se centra en desarrollar adaptadores para API, bases de datos u otros sistemas externos.

- **] Componentes intercambiables:** Hexagonal Architecture facilita la sustitución o la integración de nuevos componentes sin afectar al sistema central.

Si se introduce una nueva API o si se desprecia una más antigua, el manejo del adaptador que API se puede intercambiar sin perturbar la lógica empresarial central.

] COMPONENTES DE LA ARQUITECTURA:

- **] Entidades (Core Business Logic):** En el corazón de la Arquitectura Hexagonal se encuentran Entidades, que representan la lógica empresarial básica y las reglas de la aplicación.

Son completamente independientes de los sistemas externos y permanecen aisladas de preocupaciones como bases de datos o interfaces de usuario.

El objetivo es asegurar que la lógica empresarial se encapsule y pueda operar independientemente de cómo se reciben o persisten los datos.

- **] Puertos (Interfaces):** Los puertos son las interfaces que definen cómo los sistemas externos se comunican con la lógica central.

Estos puertos actúan como límites entre el núcleo y el mundo exterior. Existen dos tipos de puertos:

- ❖ **Puertos entrantes:** Definen cómo el mundo externo (interfaz de usuario, API o queue de mensajes) puede interactuar con el sistema central.

Son responsables de recibir inputs y desencadenar la lógica de negocio en el núcleo.

- ❖ **Puertos salientes:** Definen cómo interactúa el sistema central con sistemas externos (como son bases de datos o API externas) para enviar o recibir datos.

Abstraen los detalles de la comunicación y aseguran que el núcleo solo interactúe con componentes externos a través de estas interfaces.

- **]] Adaptadores (Implementación de Puertos):** Los adaptadores son la implementación de los puertos. Son responsables de traducir datos externos (como solicitudes HTTP, consultas de bases de datos o queue de mensajes) en algo que el núcleo puede entender, y viceversa.

Permiten que diferentes sistemas externos interactúen con el sistema central sin que el núcleo sea directamente consciente de los detalles del sistema externo.

- ❖ **Adaptadores entrantes:** Administran la comunicación de fuentes externas, como controladores HTTP, componentes de interfaz de usuario o sistemas de mensajería, y los convierten a un formato entendido por el núcleo a través de puertos entrantes.
- ❖ **Adaptadores salientes:** Administran la comunicación con servicios externos o bases de datos, tales como implementaciones de repositorios, API externas o sistemas de almacenamiento de datos.

Toman datos del núcleo, los procesan según sea necesario y envían a los sistemas externos apropiados.

- **]] Servicios de aplicación (casos de uso e Interactors):** Los Servicios de Aplicación, también conocidos como Interactors o Casos de Uso, actúan como intermediario entre las entidades (lógica empresarial básica) y las capas externas (puertos y adaptadores).

Estos servicios coordinan el flujo de datos entre los puertos entrantes y los puertos de salida.

Contienen la lógica de aplicación que maneja el proceso de negocio pero permanecen separadas de las reglas de negocio básicas.

- ❖ **Por ejemplo,** un Interactor podría tomar la entrada del usuario de un formulario web (a través de un adaptador entrante), procesarlo a través de reglas de negocio, y luego persistir los resultados usando un repositorio (a través de un adaptador de salida).

- **]] Repositorios (Capa de persistencia):** Los repositorios se utilizan para gestionar la persistencia de los datos. Proporcionan una capa entre la lógica central y el sistema de almacenamiento de datos.

Se implementan típicamente como adaptadores salientes que interactúan con la base de datos a través de puertos de salida.

- ❖ Encapsulan la lógica necesaria para recuperar y almacenar datos en sistemas de almacenamiento externos, permitiendo que la lógica central siga siendo agnóstico de la base de datos.

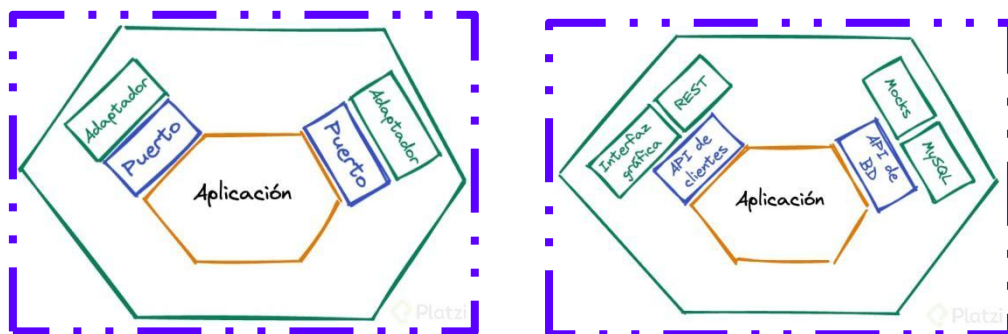
- **Capa de transporte:** La Capa de Transporte se encarga de la comunicación entre usuarios o sistemas externos y la aplicación. Esto puede incluir el manejo de solicitudes HTTP, colas de mensajes u otras formas de operaciones de entrada/salida.

La capa de transporte interactúa con los adaptadores entrantes para comunicar las solicitudes al sistema central.

- ❖ Esta capa se desvincula de la lógica empresarial, permitiendo que el sistema soporte múltiples mecanismos de comunicación (por ejemplo, REST API, WebSocket, etc.).

- **Sistemas externos (Infraestructura):** Los sistemas externos incluyen bases de datos, API, colas de mensajes y otros componentes de infraestructura con los que la aplicación interactúa.

Estos sistemas están conectados a la lógica central a través de adaptadores y puertos de salida, asegurando que el sistema central no sea consciente de los detalles del sistema externo.



◆ Patrones de diseño (Criteria, singleton)

Los **patrones de diseño** son **soluciones** habituales a **problemas** que **ocurren con frecuencia** en el diseño de software. Son como **planos prefabricados** que se pueden personalizar **para resolver un problema de diseño** recurrente en tu código.

No se puede elegir un patrón y copiarlo en el programa como si se tratara de funciones o bibliotecas ya preparadas. El **patrón no es una porción específica de código**, sino un **concepto general para resolver un problema** particular.

- Mientras que un algoritmo siempre define un grupo claro de acciones para lograr un objetivo, un patrón es una descripción de más alto nivel de una solución.
- El código del mismo patrón aplicado a dos programas distintos puede ser diferente.

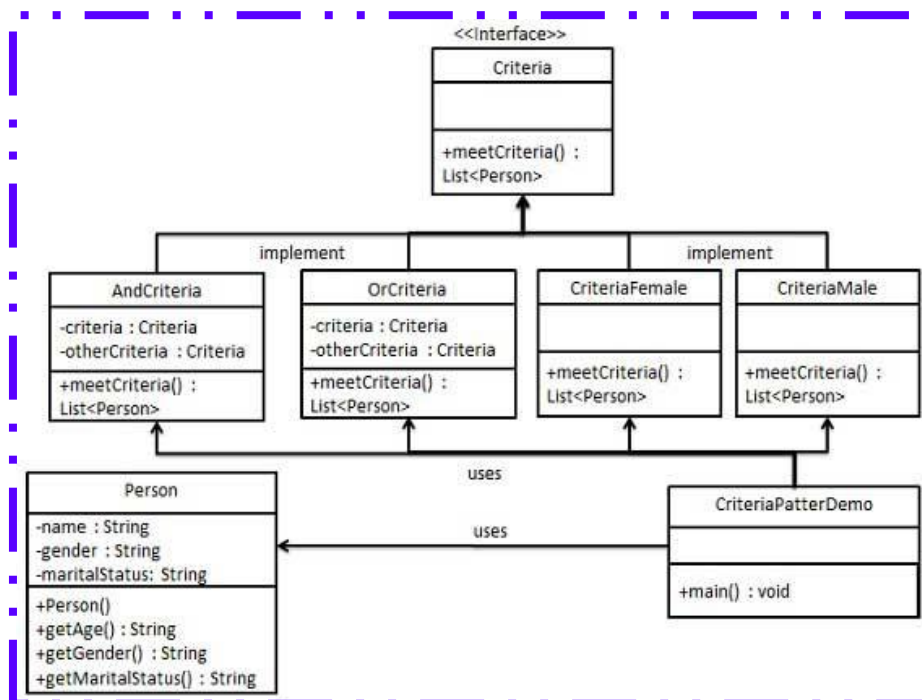
CRITERIA:

El **patrón de filtro** o **patrón de Criteria** es un patrón de diseño que permite a los desarrolladores **filtrar un conjunto de objetos** usando **diferentes criterios** y encadenándolos de forma desacoplada a través de operaciones lógicas.

Permite la **encapsulación de la lógica de la consulta** en los objetos, **abstrayendo** las complejidades de las **operaciones de base de datos**.

Estos objetos, conocidos como "**criteria**", se pasan sin problemas a **través de capas de aplicación**, sirviendo como un **modelo para obtener datos** de acuerdo a varias condiciones.

- Es una poderosa herramienta para construir especificaciones de consulta flexibles y reutilizables en una aplicación.
- promueve la separación de preocupaciones y la mantenibilidad.



♣️ EJEMPLO DE IMPLEMENTACION:

CLASE EN QUE APLICAR CRITERION [Person]

```
public class Person {  
  
    private String name;  
    private String gender;
```

```

private String maritalStatus;

public Person(String name, String gender, String maritalStatus){
    this.name = name;
    this.gender = gender;
    this.maritalStatus = maritalStatus;
}

public String getName() {
    return name;
}
public String getGender() {
    return gender;
}
public String getMaritalStatus() {
    return maritalStatus;
}
}

```

INTERFAZ PARA CRITERION [Criteria]

```

import java.util.List;

public interface Criteria {
    public List<Person> meetCriteria(List<Person> persons);
}

```

IMPLEMENTACION DE INTERFAZ CRITERION [CriteriaMale]

```

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

public class CriteriaMale implements Criteria {

    @Override
    public List<Person> meetCriteria(List<Person> persons) {
        List<Person> malePersons = new ArrayList<Person>();

        for (Person person : persons) {
            if(person.getGender().equalsIgnoreCase("MALE")){

```



```

        malePersons.add(person);
    }
}
return malePersons;
}
}

```

IMPLEMENTACION DE INTERFAZ CRITERION [CriteriaFemale]

```

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

public class CriteriaFemale implements Criteria {

    @Override
    public List<Person> meetCriteria(List<Person> persons) {
        List<Person> femalePersons = new ArrayList<Person>();

        for (Person person : persons) {
            if(person.getGender().equalsIgnoreCase("FEMALE")){
                femalePersons.add(person);
            }
        }
        return femalePersons;
    }
}

```

IMPLEMENTACION DE INTERFAZ CRITERION [AndCriteria]

```

import java.util.List;

public class AndCriteria implements Criteria {

    private Criteria criteria;
    private Criteria otherCriteria;

    public AndCriteria(Criteria criteria, Criteria otherCriteria) {
        this.criteria = criteria;
        this.otherCriteria = otherCriteria;
    }
}

```

```

@Override
public List<Person> meetCriteria(List<Person> persons) {

    List<Person> firstCriteriaPersons = criteria.meetCriteria(persons);

    return otherCriteria.meetCriteria(firstCriteriaPersons);
}
}

```

IMPLEMENTACION DE INTERFAZ CRITERION [OrCriteria]

```

import java.util.List;

public class OrCriteria implements Criteria {

    private Criteria criteria;
    private Criteria otherCriteria;

    public OrCriteria(Criteria criteria, Criteria otherCriteria) {
        this.criteria = criteria;
        this.otherCriteria = otherCriteria;
    }

    @Override
    public List<Person> meetCriteria(List<Person> persons) {
        List<Person> firstCriteriaItems = criteria.meetCriteria(persons);
        List<Person> otherCriteriaItems = otherCriteria.meetCriteria(persons);

        for (Person person : otherCriteriaItems) {
            if(!firstCriteriaItems.contains(person)){
                firstCriteriaItems.add(person);
            }
        }
        return firstCriteriaItems;
    }
}

```

DEMO DE CRITERION [CriteriaPatternDemo]

```

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

public class CriteriaPatternDemo {
    public static void main(String[] args) {
        List<Person> persons = new ArrayList<Person>();

        persons.add(new Person("Robert","Male", "Single"));
        persons.add(new Person("John", "Male", "Married"));
        persons.add(new Person("Laura", "Female", "Married"));
        persons.add(new Person("Diana", "Female", "Single"));
        persons.add(new Person("Mike", "Male", "Single"));
        persons.add(new Person("Bobby", "Male", "Single"));

        Criteria male = new CriteriaMale();
        Criteria female = new CriteriaFemale();
        Criteria single = new CriteriaSingle();
        Criteria singleMale = new AndCriteria(single, male);
        Criteria singleOrFemale = new OrCriteria(single, female);

        System.out.println("Males: ");
        printPersons(male.meetCriteria(persons));

        System.out.println("\nFemales: ");
        printPersons(female.meetCriteria(persons));

        System.out.println("\nSingle Males: ");
        printPersons(singleMale.meetCriteria(persons));

        System.out.println("\nSingle Or Females: ");
        printPersons(singleOrFemale.meetCriteria(persons));
    }

    public static void printPersons(List<Person> persons){

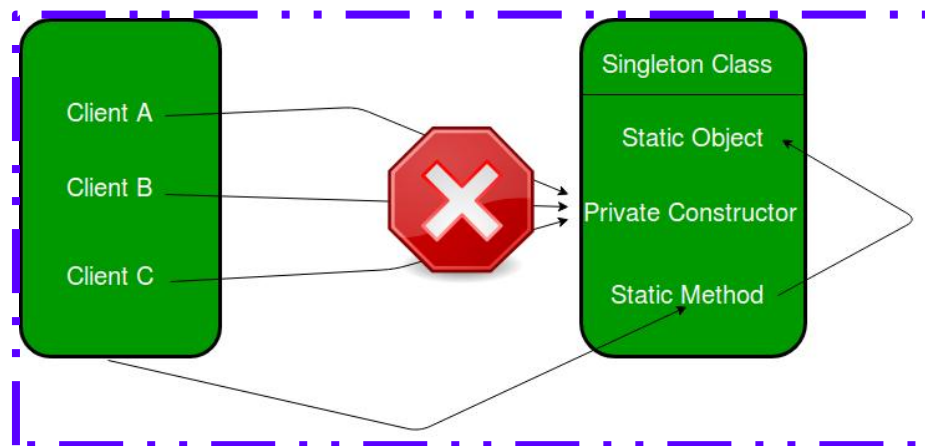
        for (Person person : persons) {
            System.out.println("Person : [ Name : " + person.getName() + ", Gender : " + person.getGender() + ", Marital Status : " + person.getMaritalStatus() + " ]");
        }
    }
}

```






 SINGLETON:

El **método o patrón de diseño Singleton** es un patrón de diseño creacional que **permite asegurar** que una **clase** tenga una **sola instancia**, al tiempo que proporciona un **punto de acceso global** a esta.

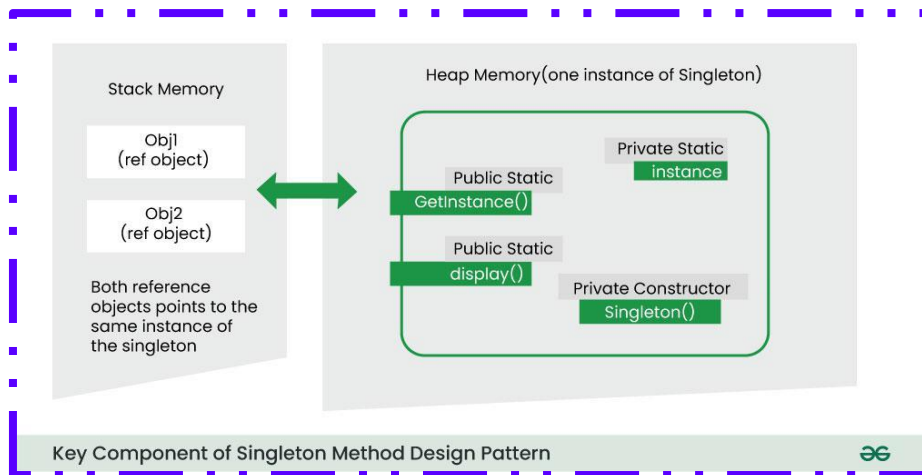
- Es ideal para escenarios que requieren un control centralizado, como la gestión de conexiones de base de datos o la configuración.



PRINCIPIOS DEL PATRON SINGLETON:

-  **Instancia única:** Singleton asegura que sólo exista una instancia de la clase a lo largo de la aplicación.
-  **Acceso global:** Se proporciona un punto de acceso global a la instancia.
-  **Inicialización perezosa o ansiosa:** Soporta la creación de la instancia cuando sea necesario (perezosa) o cuando la clase encuentra cargada (ansiosa).
-  **Seguridad de los hilos:** Implementa mecanismos para evitar que múltiples hilos creen instancias separadas simultáneamente.
-  **Constructor privado:** Restringe la instantiación directa haciendo privado el constructor, forzando con ello el uso del punto de acceso

COMPONENTES CLAVE DE SINGLETON:



- **MIEMBRO ESTÁTICO:** El patrón Singleton emplea a un miembro estático dentro de la clase.

Este miembro estático asegura que la memoria se asigne sólo una vez, preservando la única instancia de la clase Singleton.

MIEMBRO ESTATICO

```
// Static member to hold the single instance
private static Singleton instance;
```

- **CONSTRUCTOR PRIVADO:** El patrón Singleton incorpora un constructor privado, quien sirve como una barricada contra intentos externos de crear instancias de la clase Singleton.

Esto asegura que la clase tenga el control sobre su proceso de instantiación.

CONSTRUCTOR PRIVADO

```
// Private constructor to
// prevent external instantiation
class Singleton {

    // Making the constructor as Private
    private Singleton()
    {
        // Initialization code here
    }
}
```

- **|| Método estático de fabricación** : Un aspecto crucial del patrón Singleton es la presencia de un método de fabricación estático.

Este método actúa como una puerta de entrada, proporcionando un punto de acceso global al objeto Singleton.

Cuando alguien solicita una instancia, este método crea una nueva instancia (si no existe) o devuelve la instancia existente al que llama.

METODO ESTATICO DE FABRICACION

```
// Static factory method for global access
public static Singleton getInstance()
{
    // Check if an instance exists
    if (instance == null) {
        // If no instance exists, create one
        instance = new Singleton();
    }
    // Return the existing instance
    return instance;
}
```

| ♣ || EJEMPLO DE IMPLEMENTACION:

DEMO DE CRITERION [CriteriaPatternDemo]

```
/*package whatever //do not write package name here */
import java.io.*;
class Singleton {
    // static class
    private static Singleton instance;
    private Singleton()
    {
        System.out.println("Singleton is Instantiated.");
    }
    public static Singleton getInstance()
    {
        if (instance == null)
```

```
        instance = new Singleton();  
        return instance;  
    }  
    public static void doSomething()  
    {  
        System.out.println("Somethong is Done.");  
    }  
}  
  
class GFG {  
    public static void main(String[] args)  
    {  
        Singleton.getInstance().doSomething();  
    }  
}
```

||♦] OTRAS

◆ Comandos basicos de GIT

COMANDO	DESCRIPCION	EJEMPLO
git init	Inicializa un nuevo repositorio de Git en el directorio actual.	git init
git clone <url>	<p>Crea una copia de un repositorio de Git existente en la maquina local.</p> <p>■ NOTA: Se especifica el repositorio a copiarse por medio de la adición al comando, de su ruta determinada.</p>	git clone https://github.com/usuario/repositorio.git
git add <archivo>	<p>Añade contenido del directorio de trabajo al área de ensayo (staging area o 'index') para la próxima confirmación.</p> <p>■ NOTA 1: Al ejecutarse el comando git commit, este únicamente confirma los cambios establecidos en el área de ensayo.</p> <p>Por tanto se considera, git add, permite fabricar una instantánea de lo deseado a cometer.</p> <p>■ NOTA 2: sustituir el archivo por el carácter ".", permite agregar la totalidad de los archivos modificados al área de preparación.</p>	git add index.html git add .
git status	<p>Muestra los diferentes estados de los archivos en el directorio de trabajo y área de ensaño.</p> <p>Provee noción de qué archivos están modificados y sin seguimiento y cuáles con seguimiento pero no confirmados aún.</p>	git status
git diff git difftool	<p>Muestra las diferencias entre archivos modificados y el último commit.</p> <p>■ NOTA: La sustitución del comando diff por difftool, sencillamente lanza una herramienta externa para mostrar la diferencia entre dos árboles.</p> <p>En caso se desees utilizar algo que no sea el comando <code>git diff</code> incorporado.</p>	git diff git difftool

git commit git commit -m "mensaje"	Toma los contenidos de los archivos a los que se les realiza el seguimiento con git add y registra una nueva instantánea permanente en la base de datos. ■ NOTA: La adición al comando de los términos -m "mensaje", permiten la inclusión de un mensaje descriptivo al historial de cambios cometidos.	git commit git commit -m "Agrega navbar y estilos"
git reset <archivo>	Deshace los cambios en los archivo del directorio de trabajo. Permite limpiar o eliminar por completos los cambios que no se hayan enviado al repositorio publico.	git reset index.html
git rm <archivo>	Permite eliminar archivos del área de ensayo y el directorio de trabajo para Git.	git rm archivo.txt
git clean	se utiliza para eliminar archivos no deseados de tu directorio de trabajo.	git clean

◆ Que es un PR en GITHUB?

Un Pull Request (PR) es una solicitud formal para fusionar cambios realizados en una rama (o fork) hacia otra rama de un repositorio en GitHub.

- Se presenta como una propuesta para revisar y debatir las modificaciones, previo a su integración a la base de código principal.

◆ Que es un Fork en GITHUB

Un fork en github se define como una operación en la cual, se crea una copia de un repositorio en la cuenta del usuario quien la solicita.

Este repositorio copiado será igual al repositorio desde el que se realizo el fork en Git.

- Creada la copia, cada repositorio se ubicará en espacios diferentes y tendrán la posibilidad de evolucionar de forma particular, de acuerdo a las acciones del usuario.