Academia JAVA

Presentation 2023

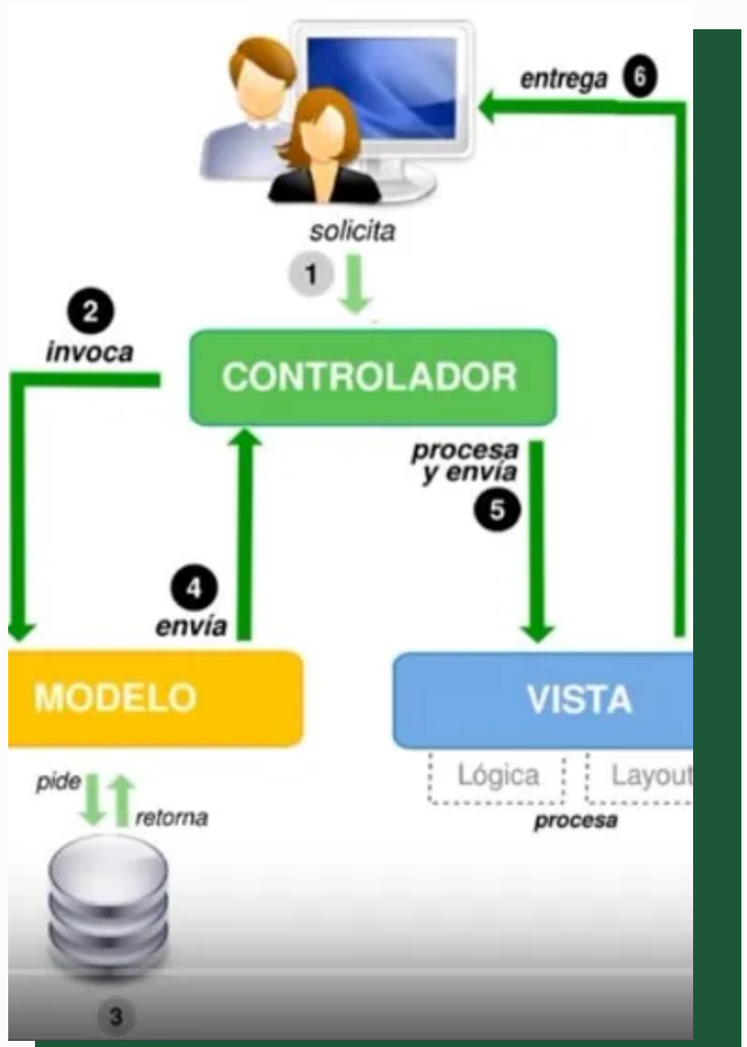
ARQUITECTURA MVC

Presentado por: Guadalupe López Velázquez

¿Qué es MVC?

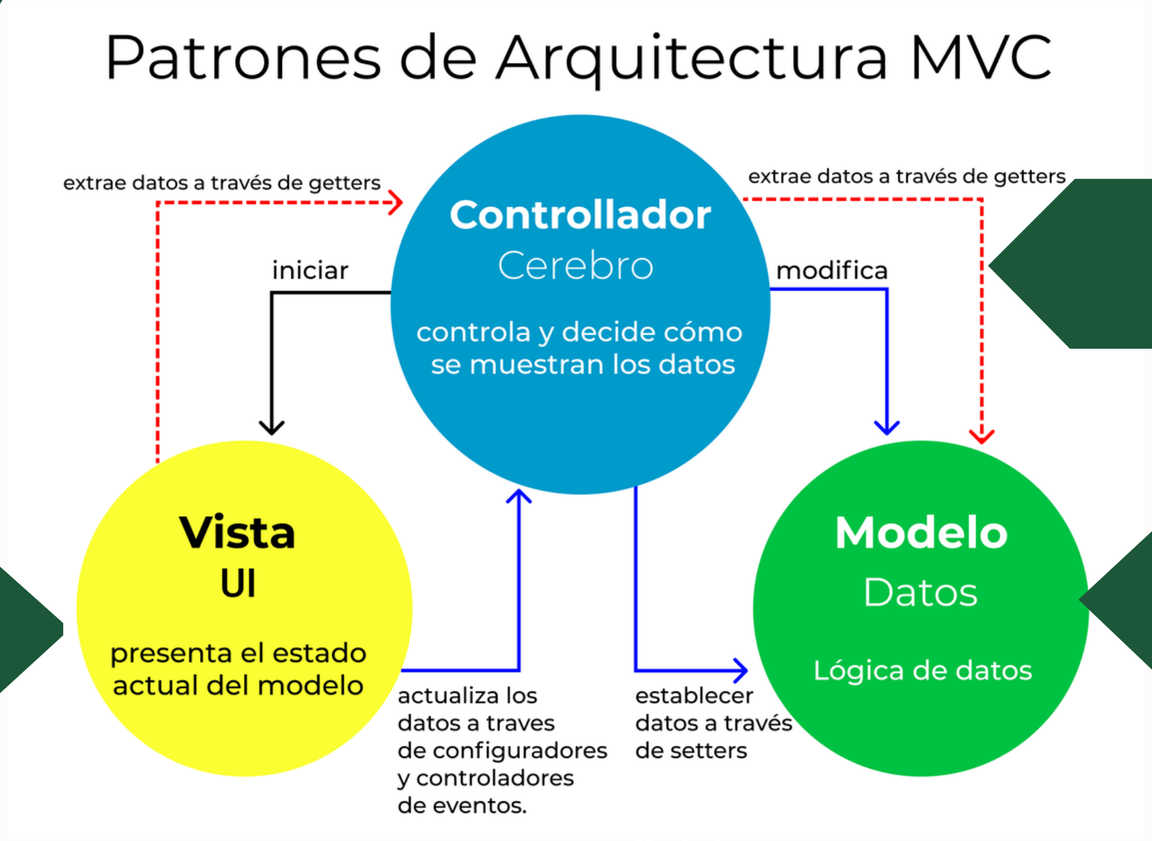
MVC es una propuesta de arquitectura del software utilizada para separar el código por sus distintas responsabilidades, manteniendo distintas capas que se encargan de hacer una tarea muy concreta, lo que ofrece beneficios diversos.
MVC se usa inicialmente en sistemas donde se requiere el uso de interfaces de usuario, aunque en la práctica el mismo patrón de arquitectura se puede utilizar para distintos tipos de aplicaciones.

Su fundamento es la separación del código en tres capas diferentes, acotadas por su responsabilidad, en lo que se llaman Modelos, Vistas y Controladores, o lo que es lo mismo, Model, Views & Controllers,



MVC

El patrón MVC te ayuda a dividir el código frontend y backend en componentes separados. De esta manera, es mucho más fácil administrar y hacer cambios a cualquiera de los lados sin que interfieran entre sí.



- **Controlador: El cerebro de la aplicación que controla como se muestran los datos.**

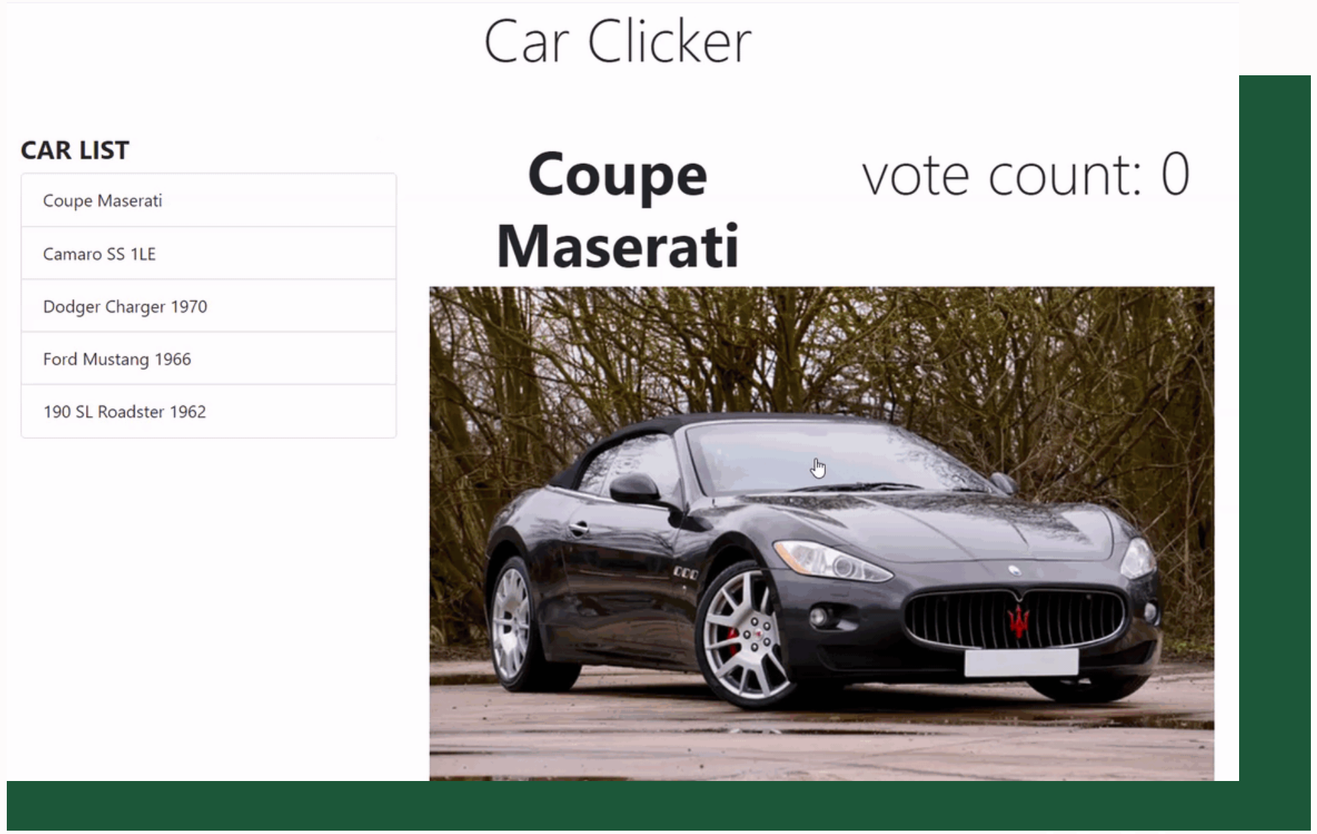
- **Vista**: El frontend o interfaz gráfica de usuario (GUI)

Modelo: El backend que contiene toda la lógica de datos

EJEMPLO DE UNA APLICACIÓN WEB

La aplicación "My Car Clicker" es una variación de una aplicación conocida como "Cat Clicker". Estas son algunas de las principales diferencias en mi aplicación:

1. No hay gatos, solo imágenes de carros potentes (¡lo siento, amantes de los gatos!)
2. Se enumeran varios modelos de automóviles
3. Hay varios contadores de clics
4. Solo muestra el coche seleccionado



Modelo (datos)

El trabajo del modelo es simplemente administrar los datos. Ya sea que los datos provengan de una base de datos, una API o un objeto JSON, el modelo es responsable de administrarlos.

En la aplicación Car Clicker, el objeto modelo contiene un arreglo de objetos car con toda la información (datos) necesaria para la aplicación. También gestiona el carro actual que se muestra con una variable que se establece inicialmente en null.

```
const model = {
    currentCar: null,
    cars: [
        {
            clickCount: 0,
            name: 'Coupe Maserati',
            imgSrc: 'img/black-convertible-coupe.jpg',
        },
        {
            clickCount: 0,
            name: 'Camaro SS 1LE',
            imgSrc: 'img/chevrolet-camaro.jpg',
        },
        {
            clickCount: 0,
            name: 'Dodger Charger 1970',
            imgSrc: 'img/dodge-charger.jpg',
        },
        {
            clickCount: 0,
            name: 'Ford Mustang 1966',
            imgSrc: 'img/ford-mustang.jpg',
        },
        {
            clickCount: 0,
            name: '190 SL Roadster 1962',
            imgSrc: 'img/mercedes-benz.jpg',
        },
    ],
};
```

Vistas (UI)

El trabajo de la vista es decidir qué verá el usuario en su pantalla y cómo.

La aplicación "Car Clicker" tiene dos vistas: carListView y CarView.

Ambas vistas tienen dos funciones críticas que definen lo que cada vista quiere inicializar y renderizar.
Estas funciones son donde la aplicación decide lo que el usuario verá y cómo.

carListVIew

```
const carListView = {
    init() {
        // almacene el elemento DOM para un fácil acceso más tarde
        this.carListElem = document.getElementById('car-list');

        // renderizar esta vista (actualizar los elementos DOM con los valores correctos)
        this.render();
    },

    render() {
        let car;
        let elem;
        let i;
        // obtener los carros para ser renderizados desde el controlador
        const cars = controller.getCars();

        // para asegurarse de que la lista está vacía antes de renderiz
        this.carListElem.innerHTML = '';

        // bucle sobre el arreglo de carros
        for(let i = 0; i < cars.length; i++) {
            // este es el carro que tenemos en bucle
            car = cars[i];

            // hacer un nuevo elemento de la lista de carros y establecer su texto
            elem = document.createElement('li');
            elem.className = 'list-group-item d-flex justify-content-between lh-condensed';
            elem.style.cursor = 'pointer';
            elem.textContent = car.name;
            elem.addEventListener(
                'click',
                (function(carCopy) {
                    return function() {
                        controller.setCurrentCar(carCopy);
                        carView.render();
                    };
                })(car)
            );
            // finalmente, agregua el elemento a la lista
            this.carListElem.appendChild(elem);
        }
    },
};rrentCar.imgSrc;
        this.carImageElem.style.cursor = 'pointer';
    },
};
```

CarView

```
const carView = {
    init() {
        // almacene punteros a los elementos DOM para un fácil acceso más tarde
        this.carElem = document.getElementById('car');
        this.carNameElem = document.getElementById('car-name');
        this.carImageElem = document.getElementById('car-img');
        this.countElem = document.getElementById('car-count');
        this.elCount = document.getElementById('elCount');


        // al hacer clic, aumentar el contador del carro actual
        this.carImageElem.addEventListener('click', this.handleClick);

        // renderizar esta vista (actualizar los elementos DOM con los valores
correctos)this.render();
    },

    handleClick() {
        return controller.incrementCounter();
    },

    render() {
        // actualizar los elementos DOM con valores del carro actual
        const currentCar = controller.getCurrentCar();
        this.countElem.textContent = currentCar.clickCount;
        this.carNameElem.textContent = currentCar.name;
        this.carImageElem.src = currentCar.imgSrc;
        this.carImageElem.style.cursor = 'pointer';
    },
};
```

Controlador (cerebro)

La responsabilidad del controlador es extraer, modificar y proporcionar datos al usuario. Esencialmente, el controlador es el enlace entre y el modelo.

A través de las funciones getter y setter, el controlador extrae datos del modelo e inicializa las vistas.

Si hay alguna actualización desde las vistas, modifica los datos con una función setter.

```
const controller = {
    init() {
        // establecer el carro actual como el primero en la lista
        model.currentCar = model.cars[0];

        // indicar a las vistas que inicialicen
        carListView.init();
        carView.init();
    },

    getCurrentCar() {
        return model.currentCar;
    },

    getCars() {
        return model.cars;
    },

    // establecer el carro seleccionado actualmente en el objeto que se pasa
en  setCurrentCar(car) {
        model.currentCar = car;
    },

    // incrementar el contador para el coche seleccionado actualmente
    incrementCounter() {
        model.currentCar.clickCount++;
        carView.render();
    },
};

controller.init();
```

Conclusión

El concepto más atractivo del patrón MVC es la separación de preocupaciones.

Las aplicaciones web modernas son muy complejas, y hacer un cambio a veces puede ser un gran dolor de cabeza.

Administrar el frontend y el backend en componentes separados más pequeños permite que la aplicación sea escalable, mantenible y fácil de expandir.