UT05.3: MVC

Contenido

1. Introducción

Los **patrones de diseño** los utilizamos para estructurar la arquitectura de una aplicación. En las primeras aplicaciones web, la **lógica de negocio** estaba implementada junto a la **interfaz de usuario** de la aplicación, lo que dificultaba su mantenimiento, el testeo, la reutilización del código, así como la escalabilidad.

El patrón de diseño **MVC** es el acrónimo de **Modelo-Vista-Controlador**, se trata del patrón de diseño más popular para implementar aplicaciones web, y es utilizado por una gran cantidad de frameworks en el mercado, pudiendo ser implementado en diferentes tipos de tecnologías. Este patrón tiene varias variantes con el MVVM o MVP más orientadas a aplicaciones de escritorio.

El objetivo de este apartado es familiarizarnos con la terminología de MVC, y empezar a estructurar nuestro código para hacerlo más reusable y mantenible, para ello vamos a implementar el patrón desde cero, y lo evolucionaremos en unidades posteriores.

1.1. Estructura de MVC

Para el explicar el patrón, vamos a utilizar la metáfora de cómo funciona un restaurante. En un restaurante tenemos tres actores, los clientes, los camareros y los cocineros.

De los tres componentes del patrón, la *Vista* tiene el rol del cliente ya que es el componente con el que interactúa el usuario.

El *Controlador* es el camarero. En un restaurante tenemos que hablar con un camarero para ordenar los platos que hemos elegido. En una aplicación, la *Vista* solicita el servicio al *Controlador* para ejecutar una operación. Además de servirnos la comanda, el camarero tiene otras funciones como cobrarnos o hacer reservas de mesas. En el *Controlador* tenemos implementado toda la **lógica de negocio** de la aplicación.

Por último, el *Modelo* realizan la función del cocinero. De la misma manera que el camarero lleva la comanda al cocinero para que la prepare, y una vez listo el plato, lo sirve en la mesa del cliente, el *Controlador* le pide servicio al *Modelo* para que prepare la información que el usuario quiere ver, enviándolo posteriormente a la *Vista* para que sea mostrado al usuario. El *Modelo* mantiene el **estado de la aplicación** independientemente de la estructura que hayamos elegido.

La siguiente imagen muestra un diagrama con la estructura del patrón MVC. El usuario realiza una solicitud al *Controlador* a través de los enlaces y formularios que tenemos en la *Vista* publicados. En función de la operación, pedirá los datos necesarios al *Modelo*, el cual tendrá que interactuar con el repositorio de información que tengamos, que podría ser tan complejo como una base de datos, o tan sencillo como un array. Cuando la información esté lista, el *Modelo* la envía como respuesta al *Controlador*, y éste la pasa a la *Vista* para que sea mostrada al usuario.

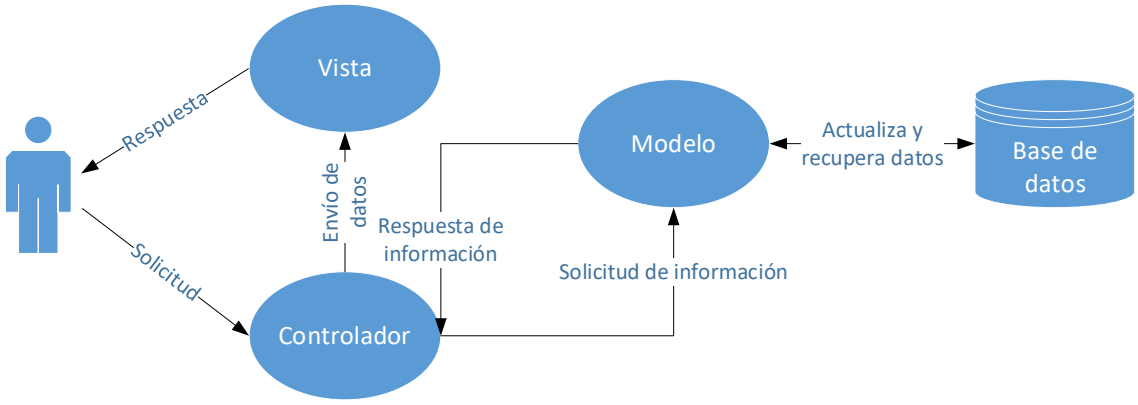

Imagen 1 Estructura MVC

En algunas implementaciones del patrón, el *Modelo* puede notificar los cambios directamente a la *Vista*, pero nosotros hemos elegido tener el menos acoplamiento posible entre *Vista* y *Modelo*, por lo que será el *Controlador* el que comunique cualquier cambio realizado en la *Vista*. Esta aproximación es más habitual en otras modalidades del patrón como MVVM para aplicaciones de escritorio.

1.2. Resumen

En la implementación del patrón, el *Modelo* representa los datos de forma independiente al *Controlador*, es decir, la lógica de negocios de nuestra aplicación, como de la *Vista*, el interfaz de usuario, teniendo un bajo acoplamiento entre los tres componentes.

Además de los beneficios que hemos citados en la introducción, también podemos citar:

1. Desarrolladores diferentes pueden trabajar simultáneamente en cada uno de los componentes de la aplicación.
2. Podemos tener diferentes "vistas" para la misma aplicación.
3. La lógica queda agrupada en el controlador.
4. La *Vista* puede ser modificada sin afectar al resto de componentes del patrón.
5. Podemos cambiar de modelo de datos sin que el *Controlador* o la *Vista* sean conscientes del cambio.

2. Implementación de MVC

Vamos a realizar una implementación del patrón utilizando como ejemplo nuestra aplicación **ShoppingCart**. Como hemos comentado en la introducción, en este primer ejemplo nos vamos a limitar a obtener datos del *Modelo*, para con ellos actualizar la *Vista*. En posteriores unidades trabajaremos la modificación de los datos en el *Modelo*. En la carpeta *shoppingcart*:

1. Renombramos *shoppingcart.js* como *shoppingCartModel.js* ya que las operaciones que hemos implementado en él actuarán de modelo.
2. Creamos un fichero para contener la aplicación denominado *ShoppingCartApp.js*.
3. Creamos un fichero para la vista denominado *ShoppingCartView.js*.
4. Creamos un fichero para el controlador denominado *ShoppingCartController.js*.

Todos los ficheros estarán implementados como módulos.

2.1. Inicializando el patrón

Para la *Vista* vamos a crear una clase **ShoppingCartView** de momento con un constructor vacío. La exportamos por defecto.

```
class ShoppingCartView {
  constructor(){

  }
}

export default ShoppingCartView;
```

Para el *Controlador* vamos a crear la clase **ShoppingCartController**. En su constructor vamos a definir dos parámetros, uno para el *Modelo* y otro para la *Vista* y los almacenamos cada uno en una propiedad privada implementada mediante `Symbol`. Al implementar la clase en un módulo, constantes con los *symbols* no son públicas mientras no los exportemos. Exportamos la clase por defecto.

```
const MODEL = Symbol('ShoppingCartModel');
const VIEW = Symbol('ShoppingCartView');

class ShoppingCartController {
  constructor(modelShoppingCart, viewShoppingCart) {
    this[MODEL] = modelShoppingCart;
    this[VIEW] = viewShoppingCart;
  }
}

export default ShoppingCartController;
```

En el fichero de aplicación, en primer lugar, debemos importar el controlador, la vista y el modelo, además vamos a importar las clases de entidad para realizar una carga inicial de datos. Debemos instanciar un controlador pasándole como argumentos la vista y el modelo. Por último, exportamos la instancia de la aplicación creada.

```
import ShoppingCart, {
  Product, Laptop, Camera, Smartphone, Tablet,
} from './shoppingCartModel.js';
import ShoppingCartController from './shoppingCartController.js';
import ShoppingCartView from './shoppingCartView.js';

const ShoppingCartApp = new
ShoppingCartController(ShoppingCart.getInstance(), new
ShoppingCartView());

export default ShoppingCartApp;
```

En el caso de que queramos utilizar en la vista el framework jQuery, debemos envolver la instanciación de la aplicación en el evento *ready*.

```
let ShoppingCartApp;
$(() => {
  ShoppingCartApp = new ShoppingCartController(
      ShoppingCart.getInstance(),
      new ShoppingCartView());
});
```

Para finalizar la inicialización, importamos el módulo en el módulo de *mitienda* para que esté accesible en la página.

```
import ShoppingCartApp from './shoppingcart/shoppingCartApp.js';
```

2.2. Cargar contenido de inicio

Tenemos el elemento `main` en la página cuyo contenido vamos a eliminar para poder generarlo desde el MVC. Este elemento lo utilizaremos como **ubicación del contenido** que vamos a crear desde MVC.

```
<main></main>
```

En el constructor de la *Vista* vamos a ir creando como **propiedades cualquier elemento de nuestra página con el que necesitemos interactuar** en el código JavaScript de la *Vista*. De momento solo almacenaremos el elemento `main`.

```
constructor() {
  this.main = document.getElementsByTagName('main')[0];
}
```

En la *Vista* vamos a añadir un método que nos permita crear el contenido inicial. Primero borramos todo el contenido de `main`, y luego añadimos el contenido inicial.

```
init() {
  this.main.replaceChildren();
  this.main.insertAdjacentHTML('afterbegin', `<div class="container
article-banner">
  <div class="row">
    <div class="col d-md-flex align-items-md-stretch flex-md-nowrap">
      <div class="article-banner-image flex-sm-grow-1">
        <h4 class="d-lg-none">Artículo semanal</h4>
      </div>
      <div
        class="article-banner-text d-flex flex-column align-items-center
justify-content-center flex-sm-grow-1">
        <h4>Artículo semanal</h4>
        <h5>Coworking</h5>
        <p>¿Cómo compartir espacios de trabajo?</p>
        <a id="button" class="btn" href="#">Ver artículo</a>
      </div>
    </div>
  </div>
</div>`);
}
```

Desde el *Controlador* necesitamos un método que invoque el método que acabamos de crear en la *Vista*. Como nomenclatura para nombrar este tipo de métodos vamos a utilizar la preposición *on* en respuesta a los eventos que nos ocurran en el *Controlador*. Estos eventos **los tenemos que invocar en respuesta a un cambio de datos** en el *Modelo*.

```
onInit = () => {
  this[VIEW].init();
};
```

En el constructor del *Controlador* debemos invocar el evento para cargar el contenido inicial.

```
// Eventos iniciales del Controlador
this.onInit();
```

Cargamos la página para comprobar que el contenido inicial se muestra correctamente.

2.3. Manejador para enlaces de inicio

Tenemos dos enlaces en la página que deberían construir el "inicio" de nuestra aplicación. Uno sería el enlace que representa el logotipo de la página, lo tenemos etiquetado con el identificador *logo*, el otro se trata del enlace *inicio* del menú de cabecera, le añadimos el identificador *init* para que sea más fácil identificarlo.

```
<a id="init" class="nav-link active" aria-current="page"
href="#">Inicio</a>
```

El *Controlador* debe disponer de métodos que **gestionen las peticiones** enviadas desde la *Vista*, también conocidos como **manejadores**. Para estos métodos utilizaremos la nomenclatura *handle* como nombre del método.

Creamos un manejador para la invocación del contenido de inicio. Este primer método tan solo tiene que disparar el evento de carga inicial de contenido.

```
handleInit = () => {
  this.onInit();
}
```

Desde la *Vista* necesitamos **enlazar el manejador en respuesta al evento de cliqueo** en los enlaces de inicio y en el logo. Creamos un método que reciba el manejador por parte del *Controlador* y lo asocie como manejador de eventos del evento `click` de ambos enlaces. La nomenclatura que vamos a utilizar es *bind*.

```
bindInit(handler) {
  document.getElementById('init').addEventListener('click', (event) => {
    handler();
  });
  document.getElementById('logo').addEventListener('click', (event) => {
    handler();
  });
}
```

Por último, en el constructor del *Controlador* debemos invocar los enlaces de los manejadores con los eventos de la *Vista*.

```
// Enlazamos handlers con la vista
this[VIEW].bindInit(this.handleInit);
```

Cada vez que cliquemos en los enlaces se cargará en la parte central el contenido inicial de la página.

En muchas ocasiones deberemos cancelar el comportamiento por defecto del evento con `event.preventDefault()`, pero en esta ocasión no es necesario.

Importante: El manejado es ejecutado desde la *Vista* pero su contexto debe ser el del *Controlador* para que pueda acceder a las propiedad donde hemos almacenado el *Modelo*. Si nos fijamos estamos utilizando una función *arrow* para declarar el método de enlace con el manejador. Tenemos que recordar que las funciones de tipo *arrow* no tienen contexto asignado, es decir, no tienen la referencia al contexto con `this`, por lo que al invocar el método del manejador estaríamos utilizando el contexto del *Controlador*.

En el caso de haber utilizado una función tradicional como método de enlace con el manejador, estaríamos utilizando el contexto de la *Vista* produciéndose un error en el código. Para resolverlo, en la invocación del método en el *Controlador* debemos pasar el manejador

cambiando el contexto utilizando el método `bind()` y el argumento `this` para indicar que el contexto será el del *Controlador* como hacemos en el ejemplo siguiente.

```
this.#viewShoppingCart.bindInit(this.handleInit.bind(this));
```

2.4. Evento onLoad()

Vamos a crear un evento en el *Controlador* que solamente se produzca con la carga de la aplicación. El evento **onLoad()** lo utilizaremos como carga inicial de datos. En el evento añadiremos al carrito los productos de testeo con los que estamos trabajando, recibirá como argumento un array con productos y las cantidades de los productos como carga inicial en el *Controlador*.

```
onLoad = (products) => {
  for (const product of products) {
    const quantity = product.quantity || 1;
    this[MODEL].addProduct(product.instance, quantity);
  }
};
```

Desde la aplicación vamos a instanciar una serie de productos e invocaremos el método anterior para hacer la carga inicial.

```
const p1 = new Laptop('111-111-111', 'HP', 'EliteBook', 1000);
p1.processor = 'i7 8 núcleos';
p1.memory = '16GB';
p1.hd = 'SDD';
p1.size = '1TB';
p1.system = 'Windows 10';
const p2 = new Camera('111-111-112', 'Canon', 'PowerShot', 500);
p2.type = 'Digital';
p2.resolution = 7;
p2.size = 7;
const p3 = new Smartphone('111-111-113', 'Samsung', 'Galaxy', 500);
p3.memory = '8GB';
p3.storage = '64GB';
p3.resolution = '1024x1024';
p3.size = 7;
p3.system = 'Android';
const p4 = new Tablet('111-111-114', 'Samsung', 'Galaxy', 500);
p4.memory = '8GB';
p4.storage = '64GB';
p4.resolution = '1024x1024';
p4.size = 15;
p4.system = 'Android';
const p5 = new Laptop('111-111-111', 'HP', 'EliteBook', 1000);
p5.processor = 'i7 8 núcleos';
p5.memory = '16GB';
p5.hd = 'SDD';
p5.size = '1TB';
p5.system = 'Windows 10';
```

```
ShoppingCartApp. onLoad([
  { instance: p1, quantity: 2 },
  { instance: p2, quantity: 3 },
  { instance: p3 },
  { instance: p4, quantity: 2 },
  { instance: p5 },
]);
```

Podemos hacer el testeo mostrando el número de productos del carrito al cliquear en el enlace de inicio. Lo añadimos en el manejador de inicio.

```
alert(this[MODEL].getNumberProducts()); // 4
```

2.5. Mostrar datos en la Vista

Vamos a ver dos ejemplos de cómo podemos obtener datos desde el *Modelo* para entregárselos a la *Vista*.

2.5.1. Mostrar número de productos del carrito

Comenzamos por la *Vista*. En el constructor creamos una propiedad con la selección del enlace del carrito del menú de cabecera.

```
this.linkShoppingcart = document.getElementById('shoppingcart');
```

Diseñamos el método que genere la vista para la funcionalidad. Como parámetro tenemos que recibir el número de productos que tenemos en el carrito.

```
showNumberProductsInCart(numProducts) {
  const spanNumProducts = this.linkShoppingcart.querySelector('span');
  if (spanNumProducts) spanNumProducts.remove();
  if (numProducts > 0) {
    this.linkShoppingcart.insertAdjacentHTML('beforeend', ` <span
class="rounded-circle px-2">${numProducts}</span>`);
    this.linkShoppingcart.classList.add('shopping');
    this.linkShoppingcart.href = '#shoppingcart-table';
  }
}
```

Pasamos al *Controlador*. Generamos un evento que invoque la vista de la funcionalidad. Como argumento debemos pasar el número de productos que tenemos en el carrito recuperados del *Modelo*.

```
onNumberProductsInCartChanged = () => {
  this[VIEW].showNumberProductsInCart(this[MODEL].getNumberProducts());
};
```

Lo invocamos en el constructor para contemplar el caso de que tengamos ya datos en el modelo, y en el evento `onLoad()` para el momento que carguemos datos debe actualizarse la vista.

```
this.onNumberProductsInCartChanged();
```

Al cargar la página debemos ver actualizado el enlace con el número de productos.

2.5.2. Mostrar tabla resumen del carrito

Vamos a crear una manejador en el componente *Controlador* que obtenga toda la información necesaria del modelo para generar una tabla con los productos añadidos en el carrito, así como la información de resumen para pasárselo a la *Vista*.

```
handleShowShoppingCart = () => {
  const data = {
    numProducts: this[MODEL].getNumberProducts(),
    products: this[MODEL][Symbol.iterator](),
    totalWithoutTaxes: this[MODEL].getTotalWithoutTaxes(),
    taxes: this[MODEL].getTaxes(),
    total: this[MODEL].getTotal(),
  };
  this[VIEW].showShoppingCart(data);
};
```

Desde la vista declaramos el método que genere la interfaz del usuario a partir del argumento con la información del modelo.

```
showShoppingCart(data) {
  if (data.numProducts > 0) {
    this.main.replaceChildren();
    const container = document.createElement('div');
    container.classList.add('container');
    container.classList.add('article-banner');
    container.insertAdjacentHTML('beforeend', `<div class="row">
      <div class="table-responsive" id="shoppingcart-table">
        <table class="table">
          <thead>
              <tr>
                  <th scope="col">#</th>
                  <th scope="col">Marca</th>
                  <th scope="col">Modelo</th>
                  <th scope="col" class="text-center">Cantidad</th>
                  <th scope="col" class="text-right">Precio</th>
              </tr>
          </thead>
          <tbody>
          </tbody>
          <tfoot>
          </tfoot>
        </table>
      </div>
    </div>`);
    this.main.append(container);

    const tbody = container.querySelector('tbody');
    for (const item of data.products) {
      const row = `<tr>
```

```
        <td scope="row">${item.product.serial}</td>
        <td>${item.product.brand}</td>
        <td>${item.product.model}</td>
        <td class="text-center">${item.quantity}</td>
        <td class="text-right">${(item.quantity *
item.product.price).toLocaleString('es-ES', { style: 'currency',
currency: 'EUR' })}</td>
      </tr>`;
      tbody.insertAdjacentHTML('beforeend', row);
    }

    const tfoot = container.querySelector('tfoot');
    const totalWithoutTaxes = `<tr>
      <td colspan="4">Total sin impuestos</td>
      <td class="text-right">${data.totalWithoutTaxes.toLocaleString('es-
ES', { style: 'currency', currency: 'EUR' })}</td>
    </tr>`;
    tfoot.insertAdjacentHTML('beforeend', totalWithoutTaxes);
    const taxes = `<tr>
      <td colspan="4">Impuestos</td>
      <td class="text-right">${data.taxes.toLocaleString('es-ES', {
style: 'currency', currency: 'EUR' })}</td>
    </tr>`;
    tfoot.insertAdjacentHTML('beforeend', taxes);
    const total = `<tr>
      <td colspan="4"><strong>Total</strong></td>
      <td class="text-right"><strong>${data.total.toLocaleString('es-ES',
{ style: 'currency', currency: 'EUR' })}</strong></td>
    </tr>`;
    tfoot.insertAdjacentHTML('beforeend', total);
  }
}
```

La interfaz será mostrada desde el enlace de la cabecera de 'mi carrito'. Necesitamos crear un método *bind* para asignar el manejador cada vez que se cliquee en el enlace.

```
bindShowShoppingCart(handler) {
  this.linkShoppingcart.addEventListener('click', (event) => {
    handler();
  });
}
```

Por último, en el constructor del *Controlador* invocamos al método para realizar el enlace.

```
this[VIEW].bindShowShoppingCart(this.handleShowShoppingCart);
```

2.6. Formato

Podemos borrar el alert de testeo del manjeador `handleInit()`.

Añadimos las siguientes CSS al fichero *main.css*.

```css
/* ShoppingCart link */
#navbarResponsive a.shopping {
    color: #faa541;
    text-decoration: none;
    cursor: pointer;
    transition: 0.3s;
}
#navbarResponsive a.shopping:hover,
#navbarResponsive a.shopping:focus {
    color: #e88006;
    text-decoration: none;
}
#navbarResponsive a.shopping span {
    background: #faa541;
    color: #343a40;
    margin-left: 0.5rem;
}
#navbarResponsive a.shopping span:hover {
    background: #e88006;
}
```