

Clase 16: Material Complementario

Sitio:	Centro de E-Learning - UTN.BA	Imprimido	Nelson Brian Avila Solano
Curso:	Curso de Backend Developer - Turno Noche	por:	
Libro:	Clase 16: Material Complementario	Día:	Tuesday, 30 de December de 2025, 19:54

Tabla de contenidos

1. Desarrollo Backend con API y Arquitectura MVC

1.1. Desarrollo Backend con API y Arquitectura MVC

2. Peticiones, respuestas y templates

2.1. Peticiones, respuestas y templates

1. Desarrollo Backend con API y Arquitectura MVC

Desarrollo Backend con API y Arquitectura MVC

1.1. Desarrollo Backend con API y Arquitectura MVC

Material de lectura: Desarrollo Backend con API y Arquitectura MVC

En el desarrollo backend, una de las estructuras más comunes y eficaces para organizar proyectos es la **arquitectura MVC** (Modelo, Vista, Controlador). En esta arquitectura, los distintos componentes de la aplicación se dividen en tres partes principales:

- **Modelo (Model):** Define y maneja los datos y la lógica de negocio de la aplicación.
- **Vista (View):** Responsable de la interfaz de usuario y la presentación de datos (generalmente no se usa directamente en el backend, sino en el frontend).
- **Controlador (Controller):** Gestiona la lógica de la aplicación, conecta el modelo con las rutas (enrutamiento) y responde a las solicitudes del cliente.

En el caso de una API desarrollada con Express, la arquitectura MVC se adapta perfectamente. A continuación, veremos cómo aplicar esta arquitectura en una API con una estructura común para múltiples entidades.

Componentes de una API con Arquitectura MVC

1. Rutas (Routes)

Las rutas definen las URL a las que los usuarios pueden acceder y cómo la aplicación responde a esas solicitudes. Las rutas se encargan de redirigir las solicitudes HTTP (GET, POST, PUT, DELETE, etc.) hacia los controladores.

Ejemplo de una ruta básica:

```
import { Router } from "express";
import { getUsers, getUserByData } from
"../controllers/userController.js";
const userRoutes = Router();

// Todas las peticiones que llegan a "/api/users/"
userRoutes.get("/", getUsers); // GET: /api/users

// Obtener un registro por un parámetro dinámico (id o nombre)
userRoutes.get("/:data", getUserByData); // GET: /api/users/1 o
/api/users/Pepito

export { userRoutes };
```

En este ejemplo, tenemos una entidad de **usuarios**, pero el mismo esquema se puede aplicar a otras entidades, como productos, órdenes, etc. Cada una de estas entidades tendrá su propio archivo de rutas, organizando mejor el código.

2. Controladores (Controllers)

Los controladores contienen la lógica para manejar las solicitudes que llegan a las rutas. Son los responsables de procesar los datos y comunicarse con el modelo para obtener o manipular información, devolviendo la respuesta adecuada al cliente.

Ejemplo de un controlador:

```
import UserModel from "../models/userModel.js";

const getUsers = (req, res) => {
  try {
    const users = UserModel.getUsers(); // Llama al modelo
    res.json(users); // Devuelve los datos en formato JSON
  } catch (error) { res.status(500).json({ error: "Internal Server Error" }); } };

const getUserByData = (req, res) => {
  try {
    const { data } = req.params;
    const user = UserModel.getUserByData(data); // Llama al modelo para buscar por ID o nombre
    if (!user) {
      res.status(404).json({ error: "User not found" });
    } else {
      res.json(user);
    }
  } catch (error) { res.status(500).json({ error: "Internal Server Error" }); } };
  export { getUsers, getUserByData };
```

En este caso, el controlador maneja las solicitudes GET para obtener todos los usuarios o un usuario específico. Al recibir una solicitud, llama a las funciones del **modelo** para obtener los datos y los devuelve al cliente en formato JSON.

3. Modelo (Model)

El modelo representa la fuente de datos de la aplicación. Aquí es donde se define cómo se manejan los datos, ya sea que estén en una base de datos, un archivo o incluso en memoria.

Ejemplo de un modelo simple:

```
import { users } from "../data/users.js"; // Simula una base de datos

const getUsers = () => {
    return users; // Retorna todos los usuarios
};

const getUserByData = (data) => { return users.find((user) =>
    user.id === data || user.name ===
data); // Busca por id o nombre
}; export default { getUsers, getUserByData }
```

En este caso, el modelo obtiene los datos de una fuente simulada, como un array en memoria. En un escenario real, el modelo interactuaría con una base de datos (por ejemplo, MySQL, MongoDB, etc.).

4. Punto de Entrada (Server)

El archivo principal de la aplicación (a menudo llamado index.js o app.js) configura el servidor Express, carga las rutas y maneja las solicitudes HTTP.

Ejemplo:

```
import express from "express";
import dotenv from "dotenv";
dotenv.config();

import { userRoutes } from "./src/routes/userRoutes.js"; // Importa las rutas de usuario
import { productRoutes } from "./src/routes/productRoutes.js"; // Importa las rutas de productos
const PORT = process.env.PORT; // Puerto definido en el archivo .env

const app = express();
app.use(express.json()); // Permite manejar solicitudes con JSON
// Rutas principales
app.use("/api/users", userRoutes); // Rutas para la entidad de usuarios
app.use("/api/products", productRoutes); // Rutas para la entidad de productos

// Manejo de rutas no existentes
app.use("*", (req, res) => {
  res.status(404).json({ error: "Resource not found" });
});

app.listen(PORT, () => {
  console.log(`Server running on http://localhost:${PORT}`);
});
```

En este ejemplo, el servidor Express se configura para escuchar en el puerto especificado en el archivo .env. Se cargan las rutas para diferentes entidades, como usuarios y productos, y se define un manejador para todas las solicitudes no encontradas (respuesta 404).

Organización del Proyecto

La estructura del proyecto en este ejemplo sigue el patrón MVC, con una separación clara de responsabilidades:

```
/src
  /controllers
    userController.js
    productController.js
  /models
    userModel.js
    productModel.js
  /routes
    userRoutes.js
    productRoutes.js
  /data
    users.js
  /env
index.js
```

Beneficios de la Arquitectura MVC

- **Modularidad:** Se puede trabajar en distintos componentes del sistema de manera independiente.
- **Mantenibilidad:** Facilita el mantenimiento y escalabilidad del código, ya que los cambios se pueden realizar en un componente sin afectar a los otros.
- **Reusabilidad:** El código puede reutilizarse en diferentes partes de la aplicación, como cuando un modelo es utilizado por múltiples controladores.

Conclusión

En este material, hemos cubierto los aspectos fundamentales de la arquitectura MVC aplicada a una API RESTful. La separación entre rutas, controladores y modelos permite mantener un código organizado, escalable y fácil de mantener. La estructura que has visto puede adaptarse fácilmente para manejar múltiples entidades en un proyecto real, como usuarios, productos, pedidos, etc.

2. Peticiones, respuestas y templates

Temario:

- Captura de datos mediante POST y GET.
- Templates de Handlebars.
- Enviar datos al navegador.

2.1. Peticiones, respuestas y templates

Captura de datos mediante POST y GET

Los parámetros **POST** son aquellos que se envían desde la página web al servidor sin que sean visibles en la URL.

Lo primero que deberemos de conocer es que los parámetros POST no se envían en la URL si no que se envían en el cuerpo de la petición. Siendo el tipo de petición recibida por los servidores como **application/x-www-form-urlencoded**. Si bien el formato de los parámetros es el mismo que el utilizado por los parámetros **GET**.

Para conformar la petición **POST** vamos a utilizar el método **.post**

El objeto **req** representa la solicitud HTTP y tiene propiedades para la cadena de consulta de la solicitud, parámetros, cuerpo, encabezados HTTP, etc. En esta documentación y por convención, el objeto siempre se denomina **req** (y la respuesta HTTP es **res**), pero su nombre real está determinado por los parámetros de la función de devolución de llamada en la que estás trabajando.

Propiedades del objeto Request

req.body

Contiene pares de datos clave-valor enviados en el cuerpo de la solicitud. De forma predeterminada, no está definido y se rellena cuando utiliza middleware de análisis corporal como **express.json()** o **express.urlencoded()**.

```
app.post('/saludo', function (req, res) {
  var nombre = req.body.nombre || '';
  var saludo = '';

  if (nombre != '')
    saludo = "Hola " + nombre;

  res.send(saludo);
})
```

req.query

A la hora de acceder al parámetro GET utilizamos el objeto req.query seguido del nombre que tenía el parámetro en el formulario (el nombre definido mediante el atributo name).

```
app.get('/saludo', function (req, res) {  
    var nombre = req.query.nombre || '';  
});
```

Templates con Handlebars

El Motor de plantilla (referido como "motor de vistas" por Express) le permite definir la estructura de documento de salida en una plantilla, usando marcadores de posición para datos que serán llenados cuando una página es generada. Las plantillas son utilizadas generalmente para crear HTML, pero también pueden crear otros tipos de documentos.

Express tiene soporte para numerosos motores de plantillas , como es el caso de Handlebars que es una extensión de Mustache.js y es un motor de plantillas muy popular ya que es basado en JavaScript y podemos utilizarlo tanto en lado servidor como en el cliente.

Handlebars nos permite escribir etiquetas HTML y luego dentro con código del motor propio podemos definir que imprime del contexto y la forma en que lo hace.

Instalar handlebars

Cuando nosotros instalamos express, instalamos también el motor de plantillas de handlebars, ya que queda prolíjamente declarado en los archivos.

```
npx express-generator --view=hbs
```

Imprimir elementos del contexto

Nuestra vista genera como resultado datos que debemos mostrar al usuario, dichos datos los pasamos a través del contexto a nuestra plantilla y aquí es donde los imprimimos. Para imprimir esto simplemente debemos encerrar la variable o el elemento en llaves dobles como lo siguiente:

```
{{nombre}}
```

Eso nos lleva a ver el contenido de "nombre" que hayamos definido en nuestra vista, estas dobles llaves llevan el escape de caracteres de forma automática, de tal manera que no resulte la impresión de código no permitido por omisión del desarrollador.

Ahora si queremos imprimir un texto sin escapar debemos utilizar triples llaves, esto le indica a Handlebars que no debe escapar nada, veamos el ejemplo:

```
{{{nombres}}}
```

Comentarios

Los comentarios se declaran así:

```
{!! esto es un comentario en Handlebars !!}
```

Renderización de data (vistas)

Las plantillas se almacenan en el directorio views (como se especifica en app.js) y se les da la extensión de archivo .hbs .

El método Response.render() se utiliza para representar una plantilla específica junto con los valores de las variables nombradas que se pasan en un objeto y luego enviar el resultado como respuesta. En el siguiente código se puede ver cómo una ruta muestra una respuesta usando la plantilla "index".

```
/* GET home page. */
router.get('/', function(req, res) {
  res.render('index');
});
```

Enviar datos al navegador

Podemos enviar datos al navegador del usuario pasando un objeto como segundo parámetro del método render(). Dentro de este objeto incluiremos un conjunto de datos definidos como clave-valor donde cualquier tipo de dato es válido.

```
res.render('plantilla', {  
    titulo: 'Lista de países',  
    lista: ['Argentina', 'Uruguay', 'Brasil'],  
    activado: true,  
    cantidad: 10  
});
```

```
<h1>{{ titulo }}</h1>  
  
<ul>  
  {{#each lista}}  
    <li>{{this}}</li>  
  {{/each}}  
</ul>  
  
<{{#if activado}}>  
<h3>Si estoy activo muestro {{cantidad}}</h3>  
<{{/if}}>
```

En el primer archivo podemos ver como estamos llamando al método render para enviar al navegador el template llamado plantilla.hbs junto con 4 tipos de datos distintos: un string, un array, un booleano y un número.

En el segundo archivo vemos cómo se interpretan estos valores en la plantilla y vemos 2 de los bloques principales de Handlebars. Tanto el string de título como el número de cantidad se representan directamente, mientras que el valor de activado lo usamos dentro del bloque `{{#if}}{{/if}}` para evaluar si es verdadero, lo que hace que se muestre todo lo incluido en el bloque. En caso contrario el usuario nunca vería el h3 ni el valor de la variable cantidad.

Para el array lista el bloque empleado es el `{{#each}}{{/each}}` que nos permite recorrer arrays. Para acceder al valor de cada interacción del bucle Handlebars usa la variable this.

Bibliografía:

- Express . Disponible desde la URL: <https://expressjs.com/es/>
- Handlebars . Disponible desde la URL:<https://handlebarsjs.com/>

