



Clase 48. Programación Backend

Manejo de dependencias con Deno



OBJETIVOS DE LA CLASE

- Desarrollar API REST con Deno.
- Realizar API REST con Deno y la dependencia Oak.

CRONOGRAMA DEL CURSO

Clase 47



**Deno: El futuro de
Nodejs?**

Clase 48



**Manejo de dependencias
con Deno**

MANEJO DE DEPENDENCIAS EN DENO



Como vimos la clase pasada, Deno utiliza URLs para el manejo de dependencias.

En proyectos grandes con muchas dependencias, actualizar los módulos será engorroso y llevará mucho tiempo si todos se importan individualmente en módulos individuales y sin un administrador de paquetes.



La práctica estándar para resolver este problema en Deno es crear un archivo ***deps.ts***.



Archivo `deps.ts`



- En este, se hace referencia a todas las dependencias remotas requeridas y los métodos y clases requeridos se re-exportan.
- Los módulos locales dependientes luego hacen referencia a los ***`deps.ts`*** en lugar de a las dependencias remotas.
- Si ahora, por ejemplo, se usa una dependencia remota en varios archivos, la actualización a una nueva versión de esta dependencia remota es mucho más simple, ya que esto se puede hacer solo dentro de ***`deps.ts`***.



Con todas las dependencias centralizadas en ***`deps.ts`***, gestionarlás se vuelve más fácil



Archivo deps.ts



- Las dependencias de desarrollo también se pueden administrar en un archivo ***dev_deps.ts*** separado, lo que permite una separación clara entre las dependencias que son solo de desarrollo y las de producción.
- Se puede exportar toda la dependencia, o solo algunos métodos. Vemos algunos ejemplos de módulos exportados desde el archivo `deps.ts`.

```
1 export * as ConsoleColor from "https://deno.land/x/colorlog@v1.0/mod.ts";
2
3 export { default as SQLQueryBuilder } from "https://raw.githubusercontent.com/denjucks/dex/master/mod.ts";
4
5 export { camelCase, snakeCase } from "https://deno.land/x/case/mod.ts";
6
7 export { Client as PostgresClient } from "https://raw.githubusercontent.com/deno-postgres/deno-postgres/master/mod.ts";
```

API REST CON OAK

OAK



¿De qué se trata?



- **Oak** es un framework de middleware para el servidor http de Deno, incluido un middleware de enrutador.
- Este framework de middleware está inspirado en Koa y enrutador de middleware inspirado en @koa/router.
- La arquitectura principal de los frameworks de middleware como oak es, como era de esperar, el concepto de middleware. Se trata de funciones que la aplicación ejecuta en un orden predecible entre el momento en que la aplicación recibe una solicitud y el envío de la respuesta.
- Estas funciones de middleware son las que nos permiten dividir la lógica de nuestro servidor en funciones separadas que la contienen.



Clase Application



- La clase principal de Oak es Application. Esta envuelve la función `serve()` del paquete `http`.
- Tiene dos métodos: **`.use()`** y **`.listen()`**. El middleware se agrega a través del método `.use()` y el método `.listen()` iniciará el servidor y comenzará a procesar solicitudes con el middleware registrado.

```
import { Application } from "https://deno.land/x/oak/mod.ts";

const app = new Application();

app.use((ctx) => {
  ctx.response.body = "Hello World!";
});

await app.listen({ port: 8000 });
```



Clase Application



Una instancia de Application también tiene algunas propiedades:

- **.keys:** se utilizarán al firmar y verificar cookies. El valor se puede establecer en un array de keys y una instancia de KeyStack, o un objeto que proporciona la misma interfaz que KeyStack
- **.proxy:** Tiene valor predeterminado false, pero se puede configurar a través de las opciones del constructor de la aplicación. Esto tiene la intención de indicar que la aplicación está detrás de un proxy y utilizará X-Forwarded-Proto, X-Forwarded-Host y X-Forwarded-For al procesar la solicitud, lo que debería proporcionar información más precisa sobre la solicitud.
- **.state:** Un registro del estado de la aplicación, que puede ser fuertemente tipado especificando un argumento genérico al construir una Application(), o inferido pasando un objeto de estado (por ejemplo, Application({estado})).



Context



A cada función de middleware se le pasa un contexto cuando se invoca. Este contexto representa "todo" lo que el middleware debe saber sobre la solicitud y la respuesta actual que está manejando la aplicación. El contexto también incluye otra información que es útil para procesar solicitudes.



Propiedades del contexto



- **.app:** Una referencia a Application que invoca este middleware.
- **.cookies:** La instancia de Cookies para este contexto que nos permite leer y configurar cookies.
- **.request:** El objeto Solicitud que contiene detalles sobre la solicitud.
- **.response:** El objeto Respuesta que se utilizará para formar la respuesta enviada al solicitante.
- **.respond:** Determina si cuando el middleware termina de procesarse, la aplicación debe enviar el .response al cliente. Si es verdadero, se enviará la respuesta, y si es falso, no se enviará la respuesta. El valor predeterminado es verdadero, pero ciertos métodos, como .upgrade() y .sendEvents() establecerán esto en falso.
- **.socket:** Esto no estará definido si la conexión no se ha actualizado a un socket web. Si la conexión se ha actualizado, se establecerá la interfaz .socket.
- **.state:** Un registro del estado de la aplicación, que puede ser fuertemente tipado especificando un argumento genérico al construir una Application(), o inferido pasando un objeto de estado (por ejemplo, Application({state})).



Métodos del contexto pasado al middleware



- **.assert():** Hace una aserción, que si no es verdadera, arroja un `HTTPError`, cuya subclase es identificada por el segundo argumento, siendo el mensaje el tercer argumento.
- **.send():** Transmite un archivo al cliente solicitante.
- **.sendEvents():** Convierte la conexión actual en una respuesta de evento enviada por el servidor y devuelve un ***ServerSentEventTarget*** donde los mensajes y eventos se pueden transmitir al cliente. Esto establecerá ***.respond*** en falso.
- **.throw():** Lanza un `HTTPError`, cuya subclase se identifica por el primer argumento, y el mensaje se pasa como el segundo.
- **.upgrade():** Intenta actualizar la conexión a una conexión de socket web y resuelve con una interfaz de socket web. Esto establecerá ***.respond*** en falso.



Router - Handlers



- En Oak tenemos un **router** que nos va a permitir definir las rutas de nuestra API REST.
- El router de Oak es también un middleware con la salvedad de que en él podemos definir el método (GET, POST, DELETE, PATCH, etc) al que el middleware va responder, la ruta específica y lógicamente un handler o función que se va a ejecutar cuando una petición se realice a dicha ruta/método.
- Los **handlers** son simplemente las funciones que responden a cada ruta. Cuando llamamos al método GET para obtener un registro es necesario acceder a los parámetros GET de la url. Esto en Oak se realiza mediante el **helper getQuery** que nos permite acceder a los slugs de la url, parámetros queryString etc.
- En el caso de los parámetros POST, en la propiedad **request** existe un método **body** que nos devuelve los valores pasados en el mismo.



Ejemplo servidor con Oak



- Vamos a hacer un simple servidor, con el método Application de la dependencia Oak.
- Importamos Application en el archivo deps.ts y luego creamos un servidor que simplemente va a escribir “Hola Mundo” como respuesta en la ruta “/”.

TS server.ts X

Oak > TS server.ts > ...

```
1  import { Application } from "../deps.ts";
2
3  const app = new Application();
4
5  app.use(ctx => {
6    ctx.response.body = 'Hola Mundo!';
7  });
8
9  console.log('Servidor Oak escuchando en el puerto 8080');
10
11 await app.listen({ port: 8080 });
```



Ejemplo servidor con Oak



- Para ejecutarlo, primero instalamos denon así podemos utilizar esta dependencia
- Luego, creamos el archivo denon.json para definir el script a ejecutar. En este caso necesitamos solamente permisos de red por lo que queda como:

```
{...} denon.json X
Oak > {...} denon.json > ...
1  {
2    "scripts": {
3      "start": {
4        "cmd": "deno run --allow-net server.ts"
5      }
6    }
7  }
```



Ejemplo servidor con Oak



- Podemos ahora entonces ejecutar el servidor con el comando `deno start`. Este ejecutará el script que definimos, por lo que se iniciará el servidor sin ningún problema.

The screenshot shows the Visual Studio Code interface with the following components:

- EXPLORER:** Shows the project structure with files like `denon.json`, `deps.ts`, `server.ts`, and `start.txt`.
- EDITOR:** Displays the `server.ts` file with the following TypeScript code:

```
1 import { Application } from './deps.ts';
2
3 const app = new Application();
4
5 app.use(ctx => {
6   ctx.response.body = 'Hola Mundo!';
7 });
8
9 console.log('Servidor Oak escuchando en el puerto 8080');
10
11 await app.listen({ port: 8080 });
```
- TERMINAL:** Shows the command `deno start` being executed. The output includes:

```
PS C:\Cursos\Coderhouse\CursoBackend\Clase48\Oak> deno start
[*] [main] v2.4.7
[*] [daem] watching path(s): **/*.
[*] [daem] watching extensions: ts,tsx,js,jsx,json
[!] [#0] starting "deno run --allow-net server.ts"
Servidor Oak escuchando en el puerto 8080
```

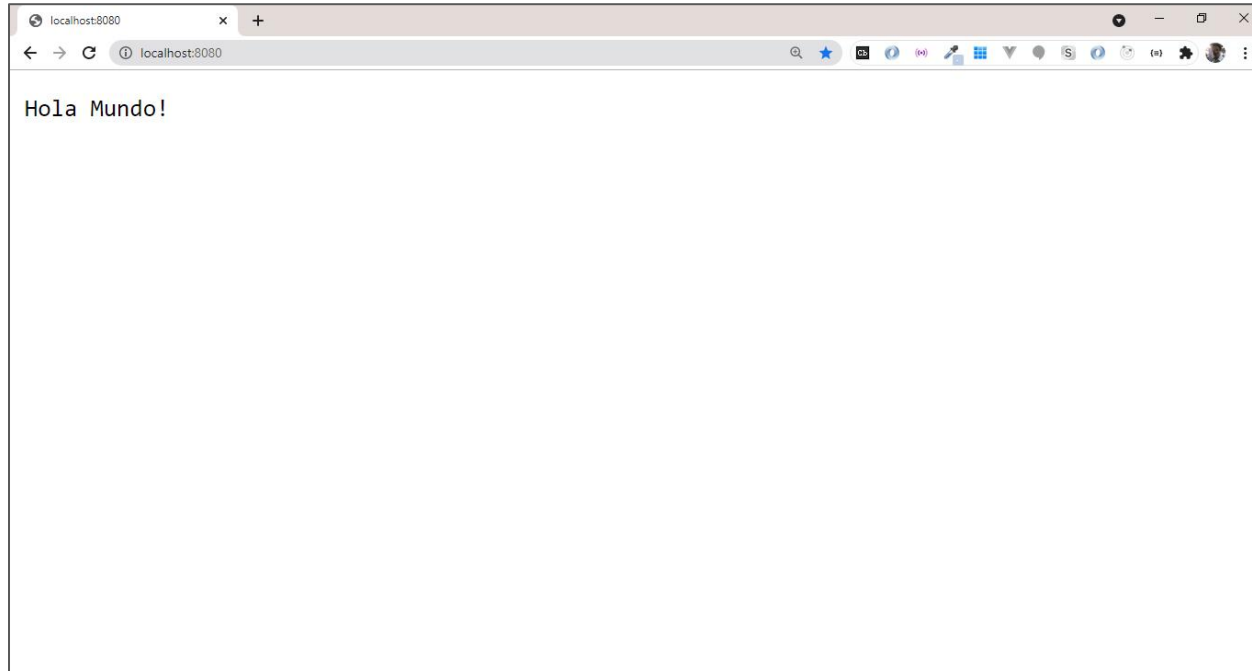
The status bar at the bottom indicates the file is at line 6, column 37, using UTF-8 encoding with CRLF line endings. It also shows extensions like Go Live, Deno 1.11.5, and Prettier.



Ejemplo servidor con Oak



- Si vamos entonces al navegador, en el puerto 8080 podemos ver el “Hola Mundo!” como respuesta de nuestro servidor.



API REST CON OAK - PERSISTENCIA EN MEMORIA



Dependencias



- Comenzamos importando las dependencias que vamos a utilizar en el archivo deps.ts.
- En este caso, importamos los métodos Application, Router, Context y helpers de Oak.
- Además, importamos el método config del módulo dotenv.

TS deps.ts X

APIREST_Oak > TS deps.ts

```
1 export { Application, Router, Context, helpers } from "https://deno.land/x/oak@v7.7.0/mod.ts";  
2 export { config } from "https://deno.land/x/dotenv@v2.0.0/mod.ts";
```



Modelo



- Definimos el modelo de usuario que vamos a usar en esta API REST. El archivo es **users.ts** dentro de la carpeta **types**.
- En este caso, es una interfaz con 3 parámetros. Esto son el Uuid que es el identificador del usuario, el nombre y la fecha de nacimiento.
- El objetivo de nuestra API REST será el de buscar, crear y actualizar usuarios.

```
TS user.ts  X
APIREST_Oak > src > types > TS user.ts > ...
1  export type Uuid = string;
2
3  export interface User {
4    uuid: Uuid;
5    name: string;
6    birthDate: Date;
7  }
```



Rutas



TS index.ts X

APIREST_Oak > src > routes > TS index.ts > ...

```
1  import { Router } from "../deps.ts";
2  import {
3    createUser,
4    deleteUser,
5    findUser,
6    updateUser,
7  } from "../handlers/user.ts";
8
9  export const router = new Router()
10    //User routes
11    .get("/api/users/:userId", findUser)
12    .delete("/api/users/:userId", deleteUser)
13    .patch("/api/users", updateUser)
14    .post("/api/users", createUser);
```

- En el archivo ***index.ts*** de la carpeta routes definimos las rutas.
- Usamos los métodos GET, DELETE, PATCH y POST de la instancia de ***Router***.
- Además, definimos qué método del handlers se corresponde con cada ruta.
- Al tener estos métodos separados, estamos usando una arquitectura en capas.



Handlers



```
TS user.ts X
APIREST_Oak > src > handlers > TS user.ts > ...
1 // deno-lint-ignore-file
2 import { Context, helpers } from "../../deps.ts";
3 import type { User } from "../../types/user.ts";
4 import * as db from "../../db/index.ts";
5
6 export const findUser = async (ctx: Context) => {
7   const { userId } = helpers.getQuery(ctx, { mergeParams: true });
8   try {
9     const user: User = await db.findUserById(userId);
10    ctx.response.body = user;
11  } catch (err) {
12    ctx.response.status = 404;
13    ctx.response.body = { msg: err.message };
14  }
15 };
16
17 export const createUser = async (ctx: Context) => {
18   try {
19     const { name, birthDate } = await ctx.request.body().value;
20     const createdUser: User = await db.createUser(name, birthDate);
21     ctx.response.body = createdUser;
22   } catch (err) {
23     ctx.response.status = 500;
24     ctx.response.body = { msg: err.message };
25   }
26 };
27
28 export const updateUser = async (ctx: Context) => {
29   ctx.response.body = { msg: "User updated!" };
30 };
31
32 export const deleteUser = async (ctx: Context) => {
33   ctx.response.body = { msg: "User deleted!" };
34 };
```

- Tenemos en la carpeta *handlers* el archivo ***users.ts***.
- En este, definimos las funciones de los métodos que corresponden a cada ruta.
- Vemos que usamos ***helpers.getQuery*** para obtener el id de parámetro de la ruta.
- Además, usamos ***ctx.request.body().value*** para tomar los datos en la ruta POST que vienen en el request.
- Usamos también ***ctx.response.body*** para devolver el usuario o mensaje de error y ***ctx.response.status*** para devolver el estado.



Queries



```
TS user.ts  X
APIREST_Oak > src > db > TS user.ts > ...
1 // deno-lint-ignore-file
2 import type { User, Uuid } from "../types/user.ts";
3 import { v4 } from "../../deps.ts";
4
5 //Fake Db Queries
6 export const findUserById = async (uuid: Uuid): Promise<User> => {
7   new Promise((resolve, reject) => {
8     if (uuid !== "23ceab21-98e3-42c1-85fa-d28ed3f5afb7") {
9       throw new Error("User not found");
10     }
11     setTimeout(() => {
12       resolve({
13         uuid,
14         name: "Paul",
15         birthDate: new Date(),
16       });
17     }, 50);
18   });
19 }
20 export const createUser = async (
21   name: string,
22   birthDate: Date,
23 ): Promise<User> => {
24   new Promise((resolve, reject) => {
25     setTimeout(() => {
26       resolve({
27         uuid: v4.generate(),
28         name,
29         birthDate,
30       });
31     }, 50);
32   });
33 }
```

- Con los datos que hemos recogido en los handlers, queremos ejecutar queries.
- En este ejemplo no vamos a implementar una conexión a base de datos real sino que vamos a simular una.
- Esta implementa los métodos `findUserById` que recibe el *uuid* del usuario y `createUser` que recibe los parámetros *name* y *birthdate* para "crear" un nuevo usuario, asignarle un *uuid* autogenerado y devolver el objeto.
- En ambos casos utilizamos una promesa con un timeout para simular una petición asíncrona con cualquier base de datos.



Logger



- En este archivo, hacemos un logger para nuestra API.
- Queremos mostrar por consola todas las peticiones que nos lleguen, el método de dicha petición mediante la propiedad *method* y los parámetros que ésta recibe cuando es una petición POST (por ejemplo) algo que podemos hacer de forma sencilla accediendo al body como vimos anteriormente.

TS logger.ts X

APIREST_Oak > src > middleware > TS logger.ts > ...

```
1 import type { Context } from "../../deps.ts";
2
3 export const logger = async (ctx: Context, next: () => void) => {
4   await next();
5   const body = await ctx.request.body().value;
6   const params = body ? `with params ${JSON.stringify(body)}` : "";
7   console.log(`${ctx.request.method} request to ${ctx.request.url} ${params}`);
8   };
```



Servidor



```
TS server.ts  X
APIREST_Oak > src > TS server.ts > ...
1  import { Application, config } from "../deps.ts";
2
3  import { router } from "../routes/index.ts";
4  import { logger } from "../middleware/logger.ts";
5
6  const { PORT } = config();
7  const app = new Application();
8
9  app.use(logger);
10 app.use(router.routes());
11
12 console.log(`Server up on port ${PORT}`);
13
14 await app.listen({ port: Number(PORT) });
```

- El server no es más que una instancia de Application al que pasamos todos los middlewares que queremos usar, en este caso el router y el logger.
- Usamos también el archivo *.env* gracias a la dependencia que importamos *dotenv*, donde guardamos la configuración y que podemos recuperar mediante *config()*.
- En este caso, nuestra única variable de entorno es el puerto PORT.



Ejecución de la API



```
src > TS server.ts > ...
1 import { Application, config } from "../deps.ts";
2
3 import { router } from "../routes/index.ts";
4 import { logger } from "../middleware/logger.ts";
5
6 const { PORT } = config();
7 const app = new Application();
8
9 app.use(logger);
10 app.use(router.routes());
11
12 console.log(`Server up on port ${PORT}`);
13
14 await app.listen({ port: Number(PORT) });
15
```

```
PS C:\Cursos\Coderhouse\CursoBackend\Claase48\APIREST_Oak> denon start
[?] [main] v2.4.7
[?] [daem] watching path(s): **/*.
[?] [daem] watching extensions: ts,tsx,js,jsx,json
[!] [d0] starting deno run --allow-net --allow-read --allow-env src/server.ts
Server up on port 8080
```

- Para ejecutar nuestro servidor con Denon, primero lo instalamos en este proyecto, con el mismo comando que vimos anteriormente.
- Luego, creamos el archivo denon.json con el script, en este caso, necesitamos permisos de red, de lectura y env (para las variables de entorno). Finalmente ejecutamos el servidor con denon start.

```
{..} denon.json X
APIREST_Oak > {..} denon.json > ...
1 {
2   "scripts": {
3     "start": {
4       "cmd": "deno run --allow-net --allow-read --allow-env src/server.ts"
5     }
6   }
7 }
```



Ejecución de la API



- Podemos ir ahora al navegador, y probar la ruta “**/api/users/id**”. Vemos que obtenemos el registro del usuario del id especificado en la ruta.

The screenshot shows a web browser window with a REST client interface. The address bar displays the URL `localhost:8080/api/users/23ceab21-98e3-42c1-85fa-d28ed3f5afb7`. The response body is a JSON object:

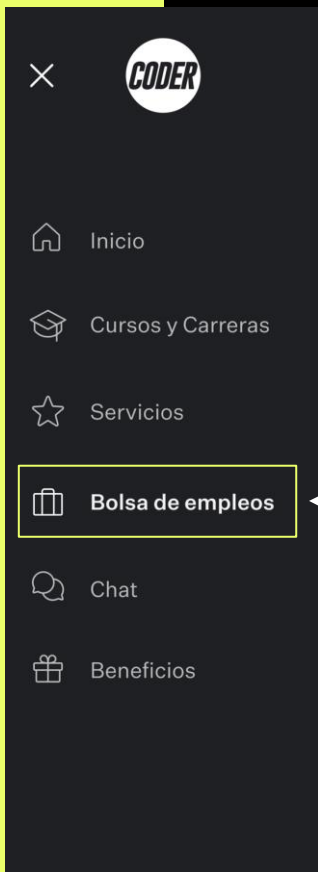
```
{  "uuid": "23ceab21-98e3-42c1-85fa-d28ed3f5afb7",  "name": "Paul",  "birthDate": "2021-07-09T17:09:33.524Z"}
```

On the right side of the response area, there are three icons: a gear for settings, a document labeled 'RAW' for viewing the raw response, and up/down arrows for scrolling.



BREAK

¡5/10 MINUTOS Y VOLVEMOS!



Nuevo

¡Lanzamos la Bolsa de Empleos!

Un espacio para seguir **potenciando tu carrera** y que tengas más **oportunidades de inserción laboral**.

Podrás encontrar la **Bolsa de Empleos** en el menú izquierdo de la plataforma.

Te invitamos a conocerla y ¡postularte a tu futuro trabajo!

Conócela

API REST CON OAK - PERSISTENCIA EN MONGO



Dependencias



- En este caso, vamos a usar los métodos **Application** y **Router** del módulo **Oak**.
- Además, vamos a usar una dependencia llamada **mongo**, de la cual vamos a usar el método **MongoClient**. Este módulo es un driver de base de datos MongoDB desarrollado para deno.
- Los importamos ambos en el archivo de **deps.ts**.

TS deps.ts X

APIREST_Oak_Mongo > TS deps.ts

```
1 export { Application, Router } from "https://deno.land/x/oak@v7.7.0/mod.ts";
2 export { MongoClient } from "https://deno.land/x/mongo@v0.22.0/mod.ts";
```



Servidor



```
TS server.ts X
APIREST_Oak_Mongo > TS server.ts > ...
1  import { Application } from "../deps.ts";
2  import router from "../routes.ts";
3  const PORT = 8080;
4
5  const app = new Application();
6
7  app.use(router.routes());
8  app.use(router.allowedMethods());
9
10 console.log(`Server escuchando en el puerto ${PORT}`);
11
12 await app.listen({ port: PORT });
```

- El servidor es, como ya vimos, una instancia de *Application* al que pasamos todos los middlewares que queremos usar.
- En este caso, le pasamos del archivo de rutas los métodos `routes()` y `allowedMethods()`.



Rutas



TS routes.ts X

APIREST_Oak_Mongo > TS routes.ts > ...

```
1  import { Router } from "../deps.ts";
2  import {
3    addQuote,
4    getQuotes,
5    getQuote,
6    updateQuote,
7    deleteQuote,
8  } from "../controllers/quotes.ts";
9
10 const router = new Router();
11
12 router
13   .get("/api/quote", getQuotes) // Get all quotes
14   .get("/api/quote/:id", getQuote) // Get one quote of quoteID: id
15   .post("/api/quote", addQuote) // Add a quote
16   .put("/api/quote/:id", updateQuote) // Update a quote
17   .delete("/api/quote/:id", deleteQuote); // Delete a quote
18
19 export default router;
```

- En el archivo de rutas, routes.ts, definimos las rutas con los métodos de Router de Oak.
- Vamos a usar los métodos GET, POST, PUT y DELETE.
- Las funciones de estas rutas las definimos en los controladores.



Modelo



TS types.ts X

APIREST_Oak_Mongo > TS types.ts > ...

```
1  export interface Quote {  
2    _id: { $oid: string };  
3    quote: string;  
4    quoteID: string;  
5    author: string;  
6  }
```

- Creamos el modelo de Quote para este ejemplo.
- En este caso, creamos la interfaz con id, quote, id de quote y el autor.



Controlador



TS quotes.ts X

APIREST_Oak_Mongo > controllers > TS quotes.ts > TypeScript > [e] getQuotes

```
1  // deno-lint-ignore-file
2  import { MongoClient } from "../deps.ts";
3  import { Quote } from "../types.ts";
4
5  const URI = "mongodb://127.0.0.1:27017";
6
7  // Mongo Connection Init
8  const client = new MongoClient();
9  try {
10     await client.connect(URI);
11     console.log("Base de datos conectada");
12 } catch (err) {
13     console.log(err);
14 }
15
16 const db = client.database("quotesApp");
17 const quotes = db.collection<Quote>("quotes");
```

- En el archivo **quotes.ts** de la carpeta **controllers**, vamos a definir todas las funciones de las rutas.
- En primer lugar, creamos la conexión a la base de datos de Mongo, con una instancia del método **MongoClient** que habíamos importado.



Controlador



```
19 // @description: GET all Quotes
20 // @route GET /api/quotes
21 const getQuotes = async ({ response }: { response: any }) => {
22   try {
23     const allQuotes = await quotes.find({}).toArray();
24     console.log(allQuotes);
25     if (allQuotes) {
26       response.status = 200;
27       response.body = {
28         success: true,
29         data: allQuotes,
30       };
31     } else {
32       response.status = 500;
33       response.body = {
34         success: false,
35         msg: "Internal Server Error",
36       };
37     }
38   } catch (err) {
39     response.body = {
40       success: false,
41       msg: err.toString(),
42     };
43   }
44 };
```

```
46 // @description: GET single quote
47 // @route GET /api/quotes/:id
48 const getQuote = async ({
49   params,
50   response,
51 }: {
52   params: { id: string };
53   response: any;
54 }) => {
55   console.log(params.id)
56   const quote = await quotes.findOne({ quoteID: params.id });
57
58   if (quote) {
59     response.status = 200;
60     response.body = {
61       success: true,
62       data: quote,
63     };
64   } else {
65     response.status = 404;
66     response.body = {
67       success: false,
68       msg: "No quote found",
69     };
70   }
71 };
```

- Tenemos entonces los métodos por GET para traer todas las quotes y para traer una por su id.



Es similar a lo que veníamos haciendo, con la diferencia que ahora podemos usar los métodos de Mongo para traer los datos.



Controlador



```
73 // @description: ADD single quote
74 // @route POST /api/quotes
75 const addQuote = async ({
76   request,
77   response,
78 }): {
79   request: any;
80   response: any;
81 }) => {
82   try {
83     if (!request.hasBody) {
84       response.status = 400;
85       response.body = {
86         success: false,
87         msg: "No Data",
88       };
89     } else {
90       const body = await request.body();
91       const quote = await body.value;
92       await quotes.insertOne(quote);
93       response.status = 201;
94       response.body = {
95         success: true,
96         data: quote,
97       };
98     }
99   } catch (err) {
100     response.body = {
101       success: false,
102       msg: err.toString(),
103     };
104   }
105   };
```

```
107 // @description: UPDATE single quote
108 // @route PUT /api/quotes/:id
109 const updateQuote = async ({
110   params,
111   request,
112   response,
113 }): {
114   params: { id: string };
115   request: any;
116   response: any;
117 }) => {
118   try {
119     const body = await request.body();
120     const inputQuote = await body.value;
121     await quotes.updateOne(
122       { quoteID: params.id },
123       { $set: { quote: inputQuote.quote, author: inputQuote.author } }
124     );
125     const updatedQuote = await quotes.findOne({ quoteID: params.id });
126     response.status = 200;
127     response.body = {
128       success: true,
129       data: updatedQuote,
130     };
131   } catch (err) {
132     response.body = {
133       success: false,
134       msg: err.toString(),
135     };
136   }
137   };
```

```
139 // @description: DELETE single quote
140 // @route DELETE /api/quotes/:id
141 const deleteQuote = async ({
142   params,
143   response,
144 }): {
145   params: { id: string };
146   request: any;
147   response: any;
148 }) => {
149   try {
150     await quotes.deleteOne({ quoteID: params.id });
151     response.status = 201;
152     response.body = {
153       success: true,
154       msg: "Product deleted",
155     };
156   } catch (err) {
157     response.body = {
158       success: false,
159       msg: err.toString(),
160     };
161   }
162   };
163
164 export { getQuotes, getQuote, addQuote, updateQuote, deleteQuote };
```

- De similar forma tenemos la función para crear Quote, modificarla y eliminarla.



Ejecución de la API



- Primero, instalamos Denon en este proyecto para poder usarlo, y lo hacemos con el mismo comando que vinimos usando en los proyectos anteriores.
- Luego, en el archivo denon.json tenemos el script para iniciar el servidor con permisos de red.

```
{...} denon.json X
APIREST_Oak_Mongo > {...} denon.json > ...
1  {
2    "scripts": {
3      "start": {
4        "cmd": "deno run --allow-net server.ts"
5      }
6    }
7  }
```



Ejecución de la API



- Para ejecutar el servidor debemos cuidar primero haber iniciado nuestro motor de base de datos MongoDB.
- Luego, iniciamos el servidor con el comando **denon start**.

The screenshot shows the Visual Studio Code interface with a project named 'APIREST_Oak_Mongo'. The Explorer sidebar on the left shows the file structure. The main editor displays the 'server.ts' file with the following code:

```
1 import { Application } from './deps.ts';
2 import router from './routes.ts';
3 const PORT = 8080;
4
5 const app = new Application();
6
7 app.use(router.routes());
8 app.use(router.allowedMethods());
9
10 console.log(`Server escuchando en el puerto ${PORT}`);
11
12 await app.listen({ port: PORT });
13
```

The TERMINAL panel at the bottom shows the command 'Oak_Mongo> denon start' and its output:

```
[*] [main] v2.4.7
[*] [daemon] watching path(s): **/*.ts
[*] [daemon] watching extensions: ts,tsx,jsx,json
[!] [#0] starting 'deno run --allow-net server.ts'
Base de datos conectada
Server escuchando en el puerto 8080
```

On the right side of the terminal, a PowerShell window shows the MongoDB command prompt and the 'mongo' command being executed, displaying the MongoDB shell prompt and some system information.



Ejecución de la API



- Vamos entonces al navegador, y probamos la ruta “/api/quotes” que lista todas las quotes que tenemos en la base de datos.

```
{
  success: true,
  - data: [
    - {
      _id: "60e87e0a0ca1af3e943cac86",
      quoteID: "1",
      quote: "comentario 1",
      author: "Ds"
    },
    - {
      _id: "60e87e230ca1af3e943cac88",
      quoteID: "3",
      quote: "comentario 3",
      author: "Sg"
    }
  ]
}
```



USANDO OAK

Tiempo: 15 minutos

USANDO OAK

Desafío
generico



Tiempo: 15 minutos

Crear un servidor API Restful con Deno Oak que permita:

- incorporar productos con su nombre, descripción y precio.
 - listar los productos totales y por su id.
 - modificar un producto por su id.
 - borrar un producto por su id.
- Utilizar denon para el reinicio del servidor en caso de cambiar el código fuente.
- Centralizar el uso de las dependencias en un sólo archivo deps.ts
- Utilizar postman para enviar todos los request http a la ruta '/api/productos' y también el navegador para visualizar la lista cargada.

CODER HOUSE



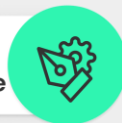
SERVIDOR EN DENO

SERVIDOR EN DENO

Formato: link a un repositorio en Github con el proyecto cargado.

Sugerencia: no incluir los node_modules

Desafío
entregable



>> Consigna:

1. Crear un servidor que utilice el módulo http server y genere la vista con React render.
2. Configurar deno para que, ante un cambio de código, el servidor se reinicie automáticamente.

El servidor presentará en su ruta raíz un formulario de ingreso de un color, que será enviado al mismo por método post. Dicho color (en inglés) será incorporado a un array de colores persistido en memoria.

Por debajo del formulario se deberán representar los colores recibidos en una lista desordenada (ul) utilizando el mismo color para la letra en cada caso. El color de fondo de la vista será negro.

NOTA: El servidor deberá tener extensión tsx para el correcto funcionamiento de la sintaxis de vista de React en Typescript.

CODER HOUSE



ENTREGA DEL PROYECTO FINAL

Deberás entregar tu aplicación eCommerce Backend correspondiente a la última entrega de tu proyecto final.

ENTREGA DEL PROYECTO FINAL

Formato: link a un repositorio en Github con el proyecto cargado.

Sugerencia: no incluir los node_modules

Proyecto
Final



>>Consigna: Para culminar con el proyecto final, vamos a realizar las últimas reformas al desarrollo backend e-Commerce para que quede estructurado de acuerdo a los criterios y mecanismos que fuimos aprendiendo en este último trayecto del curso.

- En primer lugar la aplicación de servidor debe tener sus capas MVC bien definidas y en archivos separados. Debe existir la capa de ruteo, el controlador, la capa de lógica de negocio con los casos de uso y las validaciones y la capa de persistencia con los DAOs/DTOs o Repositories necesarios para soportar el o los sistemas de persistencia elegidos. En caso de ser más de uno, utilizar una factory para que podamos elegir el sistema de almacenamiento al inicio del servidor.
- El servidor debe disponer de configuraciones mediante variables de entorno, que permitan crear un ambiente para desarrollo y otro para producción, elegibles desde la variable de environment `NODE_ENV` al desplegar la aplicación. Como variables de configuración deberían estar el puerto de escucha del servidor, la persistencia elegida, el string de conexión a la base de datos (si hubiera varios sistemas de persistencia en base de datos considerar todos los casos y sus diferencias), API keys y todo lo que sea necesario que esté en un archivo protegido fuera del código del servidor. Pensar en utilizar bases de datos y servidores locales para la configuración de desarrollo.

ENTREGA DEL PROYECTO FINAL

Formato: link a un repositorio en Github con el proyecto cargado.

Sugerencia: no incluir los node_modules

Proyecto
Final



- Se debe analizar que el hecho de incorporar un caso más de uso en la lógica del servidor, sea un proceso de agregar código y no de modificar el existente.
- Si agregamos un sistema más de persistencia, deberíamos agregar sólo el módulo nuevo y reformar la factory, mientras que resto del proyecto: router, controlador, lógica de negocio, validaciones y otros sistemas de persistencia no deberían sufrir modificaciones para soportar la nueva función.
- El código debe quedar bien tabulado, legible, ordenado y comentado ni por exceso ni por defecto.
- Las funciones o clases que se por sí solas expliquen su misión, no necesitan ser explicadas (salvo que amerite por complejidad).
- Para concluir, subir el desarrollo completo a Heroku o algún PASS de preferencia, seleccionando la configuración a producción de modo de utilizar los parámetros adecuados de funcionamiento y la persistencia en la nube a través de bases de datos como servicio (DBaaS).

👉 Para más detalle, puedes consultar la [Consigna Proyecto Final Curso Backend](#).

CODER HOUSE

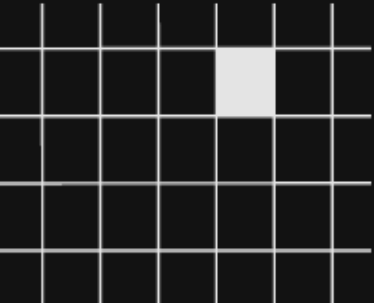
¿PREGUNTAS?





¡MUCHAS GRACIAS!

Resumen de lo visto en clase hoy:

- Manejo de dependencias en proyectos Deno.
 - API REST con Deno y Express.
 - API REST con Deno y Oak.
- 



OPINA Y VALORA ESTA CLASE

#DEMOCRATIZANDO LA EDUCACIÓN