



Clase 43. Programación Backend

DOCUMENTACIÓN EN API REST



OBJETIVOS DE LA CLASE

- Conocer y comprender acerca del concepto de API y API REST.
- Aprender acerca del módulo Express Generator para crear proyectos Node.
- Conocer acerca de la importancia de la documentación de una API REST, y cómo generarla utilizando Swagger.

CRONOGRAMA DEL CURSO

Clase 42



**Arquitectura del servidor:
Persistencia**

Clase 43



Documentación de APIs

Clase 44



GraphQL

API

¿Qué es una API?



- API significa interfaz de programación de aplicaciones. Es un conjunto de definiciones y protocolos que se utiliza para desarrollar e integrar el software de las aplicaciones.
- Las API permiten que sus productos y servicios se comuniquen con otros, sin necesidad de saber cómo están implementados. Esto simplifica el desarrollo de las aplicaciones y permite ahorrar tiempo y dinero.
- Le otorgan flexibilidad; simplifican el diseño, la administración y el uso de las aplicaciones, y proporcionan oportunidades de innovación, lo cual es ideal al momento de diseñar herramientas y productos nuevos (o de gestionar los actuales).

Usos y aplicaciones

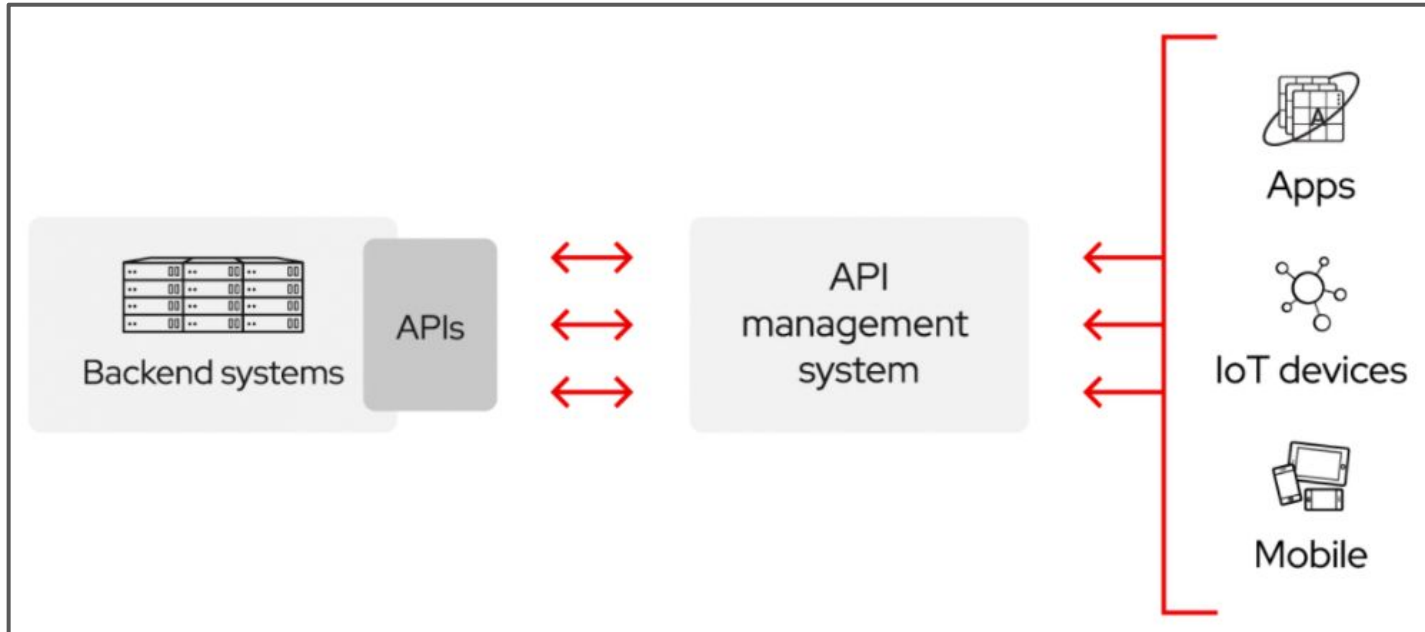


- Las API son un medio simplificado para conectar nuestra propia infraestructura a través del desarrollo de aplicaciones nativas de la nube, pero también nos permiten compartir sus datos con clientes y otros usuarios externos.
- Las API públicas representan un valor comercial único porque simplifican y amplían la forma en que se conecta con sus partners y, además, pueden rentabilizar sus datos (un ejemplo conocido es la API de Google Maps).
- Nos permiten habilitar el acceso a sus recursos y, al mismo tiempo, mantener la seguridad y el control.

Usos y aplicaciones



Diagrama de estructura de una aplicación completa usando una API.



Enfoques de las versiones de APIs



- Existen tres enfoques respecto a las políticas de las versiones de las API:
 - **Privado:** Las API solo se pueden usar internamente, así que las empresas tienen un mayor control sobre ellas.
 - **De partners:** Las API se comparten con partners empresariales específicos, lo cual puede ofrecer flujos de ingresos adicionales, sin comprometer la calidad.
 - **Público:** Todos tienen acceso a las API, así que otras empresas pueden desarrollar API que interactúen con las de usted y así convertirse en una fuente de innovaciones.

API REST



¿De qué se trata?



- Un servicio **REST** no es una arquitectura software, sino un *conjunto de restricciones* a tener en cuenta en la arquitectura software que usaremos para crear aplicaciones web respetando **HTTP**.
- las restricciones que definen a un sistema REST son:
 - **Cliente-servidor:** El servidor se encarga de controlar los datos mientras que el cliente se encarga de manejar las interacciones del usuario.
 - **Sin estado:** cada petición que recibe el servidor debería ser independiente y contener todo lo necesario para ser procesada.



¿De qué se trata?



- **Cacheable:** debe admitir un sistema de almacenamiento en caché. Esto evitará repetir varias conexiones entre el servidor y el cliente para recuperar un mismo recurso.
- **Interfaz uniforme:** define una interfaz genérica para administrar cada interacción que se produzca entre el cliente y el servidor de manera uniforme, lo cual simplifica y separa la arquitectura.
- **Sistema de capas:** el servidor puede disponer de varias capas para su implementación. Esto ayuda a mejorar la escalabilidad, el rendimiento y la seguridad.

👉 Hoy en día la mayoría de las empresas utilizan API REST para crear servicios web. Esto se debe a que es un estándar lógico y eficiente.



Las operaciones más importantes que nos permitirán manipular los recursos son:

- GET es usado para recuperar un recurso.
- POST se usa la mayoría de las veces para crear un nuevo recurso.
- PUT es útil para crear o editar un recurso. En el cuerpo de la petición irá la representación completa del recurso.
- PATCH realiza actualizaciones parciales. En el cuerpo de la petición se incluirán los cambios a realizar en el recurso.
- DELETE se usa para eliminar un recurso.



Otras operaciones menos comunes pero útiles son:

- HEAD funciona igual que GET pero no recupera el recurso. Se usa sobre todo para testear si existe el recurso antes de hacer la petición GET para obtenerlo.
- OPTIONS permite al cliente conocer las opciones o requerimientos asociados a un recurso antes de iniciar cualquier petición sobre el mismo.



Métodos



Se dice que los **métodos seguros** son aquellos que no modifican recursos (serían GET, HEAD y OPTIONS), mientras que los **métodos idempotentes** serían aquellos que se pueden llamar varias veces obteniendo el mismo resultado (GET, PUT, DELETE, HEAD y OPTIONS).

Características

- **El uso de hipermedios** (procedimientos para crear contenidos que contengan texto, imagen, vídeo, audio, etc.) para permitir al usuario navegar por los distintos recursos de una API REST a través de enlaces HTML.
- **Independencia de lenguajes**. La separación en capas de la API permite que el cliente se despreocupe del lenguaje en que esté implementado el servidor. Basta a ambos con saber que las respuestas se recibirán en el lenguaje de intercambio usado (que será XML o JSON).



Características



- **Los recursos en una API REST se identifican por medio de URI.** Será esa misma URI la que permitirá acceder al recurso o realizar cualquier operación de modificación sobre el mismo.
- **Las APIs deben manejar cualquier error que se produzca,** devolviendo la información de error adecuada al cliente.

GENERADOR DE PROYECTOS CON EXPRESS



¿De qué se trata?



- Express tiene un generador de aplicaciones, con el cual podemos generar la estructura de una API REST. Este módulo se llama **Express generator**.
- Vamos a ver un ejemplo de un proyecto creado con este módulo.
- Previo a empezar, en el proyecto de ejemplo se utilizan dos módulos que vamos a explicar a continuación.
- El módulo de **Debug** que se utiliza para loguear registros que no se deben mostrar luego en producción.
- El módulo de **Morgan** se utiliza para loguear en consola registros como errores.

Módulo Debug



- Express utiliza el módulo debug internamente para *registrar información* sobre las coincidencias de rutas, las funciones de middleware que se están utilizando, la modalidad de aplicación y el flujo del ciclo de solicitud/respuestas.
- Es como una versión aumentada de *console.log*, aunque a diferencia de éste, no tiene que comentar los registros debug en el código de producción.

Módulo Debug



- Para ver todos los registros internos utilizados en Express, establecer la variable de entorno DEBUG en express:* cuando iniciamos la aplicación con el comando:

```
> set DEBUG=express:* & node index.js
```
- Para utilizar Debug en una aplicación generada con Express, usamos el siguiente comando:

```
$ DEBUG=sample-app:* node ./bin/www
```

Donde sample-app es el nombre de nuestra aplicación.

Módulo Morgan



- Morgan es una herramienta de registro (middleware) que se puede usar en servidores HTTP implementados usando Express & Node. Se puede usar para registrar solicitudes, errores y más en la consola.
- Lo instalamos como siempre con el comando: `$ npm install morgan --save`
- Luego lo requerimos en el app.js: `const morgan = require('morgan');`
- Podemos ahora usarlo como middleware a nivel aplicación:

```
app.use(morgan('dev'))
```

Comenzando con Express Generator



- En primer lugar, comenzamos instalando globalmente el módulo de Express Generator: `$ npm install express-generator -g`
- Con el comando `$ express -h` se muestran las opciones disponibles para realizar:

```
Usage: express [options][dir]
```

```
Options:
```

```
-h, --help          output usage information
--version           output the version number
-e, --ejs           add ejs engine support
--hbs              add handlebars engine support
--pug              add pug engine support
-H, --hogan         add hogan.js engine support
--no-view          generate without view engine
-v, --view <engine> add view <engine> support (ejs|hbs|hjs|jade|pug|twig|vash) (defaults to jade)
-c, --css <engine>  add stylesheet <engine> support (less|stylus|compass|sass) (defaults to plain css)
--git              add .gitignore
-f, --force         force on non-empty directory
```

Comenzando con Express Generator



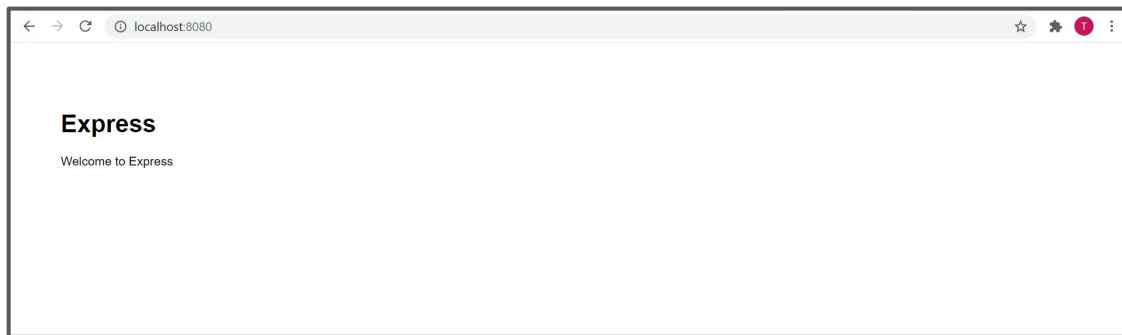
- Para crear una aplicación, por ejemplo llamada *myapp* y con el motor de vistas pug, usamos el siguiente comando: `$ express --view=pug myapp`
- Este crea una carpeta llamada *myapp* en el directorio en que estemos en la consola al momento de ejecutar ese comando. En la salida obtenemos:

```
create : myapp
create : myapp/package.json
create : myapp/app.js
create : myapp/public
create : myapp/public/javascripts
create : myapp/public/images
create : myapp/routes
create : myapp/routes/index.js
create : myapp/routes/users.js
create : myapp/public/stylesheets
create : myapp/public/stylesheets/style.css
create : myapp/views
create : myapp/views/index.pug
create : myapp/views/layout.pug
create : myapp/views/error.pug
create : myapp/bin
create : myapp/bin/www
```

Comenzando con Express Generator



- En nuestro caso, creamos una aplicación usando **handlebars** como motor de vistas.
- Luego ingresamos en la carpeta que se creó de nuestro proyecto e instalamos las dependencias.
- A continuación, podemos levantar el servidor con el comando “**npm start**” (en nuestro caso, queremos además pasar como variables de entorno: **DEBUG=myapp:***).
- Finalmente, podremos ingresar a nuestra aplicación desde el navegador ingresando en <http://localhost:3000> . Veremos entonces lo siguiente:



Comenzando con Express Generator



- La aplicación generada tiene la siguiente estructura de directorios:

```
.  
├── app.js  
├── bin  
│   └── www  
├── package.json  
├── public  
│   ├── images  
│   ├── javascripts  
│   └── stylesheets  
│       └── style.css  
├── routes  
│   ├── index.js  
│   └── users.js  
└── views  
    ├── error.pug  
    ├── index.pug  
    └── layout.pug
```

Archivos generados y agregados



1. El archivo de entrada a la aplicación, está en la carpeta bin y se llama www. Trae ya por defecto el siguiente código:

```
1 #!/usr/bin/env node
2
3 /**
4  * Module dependencies.
5  */
6
7 var app = require('../app');
8 var debug = require('debug')('myapp:server');
9 var http = require('http');
10
11 /**
12  * Get port from environment and store in Express.
13  */
14
15 var port = normalizePort(process.env.PORT || '8080');
16 app.set('port', port);
17
18 /**
19  * Create HTTP server.
20  */
21
22 var server = http.createServer(app);
```

```
24 /**
25  * Listen on provided port, on all network interfaces.
26  */
27
28 server.listen(port);
29 server.on('error', onError);
30 server.on('listening', onListening);
31
32 /**
33  * Normalize a port into a number, string, or false.
34  */
35
36 function normalizePort(val) {
37   var port = parseInt(val, 10);
38
39   if (isNaN(port)) {
40     // named pipe
41     return val;
42   }
43
44   if (port >= 0) {
45     // port number
46     return port;
47   }
48
49   return false;
50 }
```

```
52 /**
53  * Event listener for HTTP server "error" event.
54  */
55
56 function onError(error) {
57   if (error.syscall !== 'listen') {
58     throw error;
59   }
60
61   var bind = typeof port === 'string'
62     ? 'Pipe ' + port
63     : 'Port ' + port;
64
65   // handle specific listen errors with friendly messages
66   switch (error.code) {
67     case 'EACCES':
68       console.error(bind + ' requires elevated privileges');
69       process.exit(1);
70       break;
71     case 'EADDRINUSE':
72       console.error(bind + ' is already in use');
73       process.exit(1);
74       break;
75     default:
76       throw error;
77   }
78 }
```

```
80 /**
81  * Event listener for HTTP server "listening" event.
82  */
83
84 function onListening() {
85   var addr = server.address();
86   var bind = typeof addr === 'string'
87     ? 'pipe ' + addr
88     : 'port ' + addr.port;
89   debug('Listening on: ' + bind);
90   console.log('Servidor escuchando en el puerto ' + port);
91 }
92
```

Archivos generados y agregados



2. Luego, el servidor está en el archivo *app.js* que tiene el siguiente código donde se usan los middleware, y se definen las rutas.

```
1  var createError = require('http-errors');
2  var express = require('express');
3  var path = require('path');
4  var cookieParser = require('cookie-parser');
5  var logger = require('morgan');
6
7  var indexRouter = require('./routes/index');
8  var usersRouter = require('./routes/users');
9  var productsRouter = require('./routes/products');
10
11  var app = express();
12
13  // view engine setup
14  app.set('views', path.join(__dirname, 'views'));
15  app.set('view engine', 'hbs');
16
17  app.use(logger('dev'));
18  app.use(express.json());
19  app.use(express.urlencoded({ extended: false }));
20  app.use(cookieParser());
21  app.use(express.static(path.join(__dirname, 'public')));
22
23  app.use('/', indexRouter);
24  app.use('/users', usersRouter);
25  app.use('/products', productsRouter);
26
27  // catch 404 and forward to error handler
28  app.use(function(req, res, next) {
29    next(createError(404));
30  });
```

```
32  // error handler
33  app.use(function(err, req, res, next) {
34    // set locals, only providing error in development
35    res.locals.message = err.message;
36    res.locals.error = req.app.get('env') === 'development' ? err : {};
37
38    // render the error page
39    res.status(err.status || 500);
40    res.render('error');
41  });
42
43  module.exports = app;
44
```

Archivos generados y agregados



3. En la carpeta de rutas, se crean los archivos de rutas de *index* y de *users*. Creamos además en este caso el de *products*.

JS index.js X

```
myapp > routes > JS index.js > ...
1  var express = require('express');
2  var router = express.Router();
3
4  /* GET home page. */
5  router.get('/', function(req, res, next) {
6    res.render('index', { title: 'Express' });
7  });
8
9  module.exports = router;
10
```

JS users.js X

```
myapp > routes > JS users.js > ...
1  var express = require('express');
2  var router = express.Router();
3
4  /* GET users listing. */
5  router.get('/', function(req, res, next) {
6    res.send('respond with a resource GET');
7  });
8
9  router.post('/', function(req, res, next) {
10   res.send('respond with a resource POST');
11 });
12
13 router.put('/', function(req, res, next) {
14   res.send('respond with a resource PUT');
15 });
16
17 router.delete('/', function(req, res, next) {
18   res.send('respond with a resource DELETE');
19 });
20
21 module.exports = router;
22
```

JS products.js X

```
myapp > routes > JS products.js > ...
1  var express = require('express');
2  var router = express.Router();
3
4  router.use(express.urlencoded({ extended: true }));
5
6  /* GET users listing. */
7  router.get('/', function(req, res, next) {
8    res.render('form');
9  });
10
11 router.post('/', function(req, res, next) {
12   console.log('body', {...req.body})
13   //res.send('respond with a product POST');
14   res.redirect('/products')
15 });
16
17 router.put('/', function(req, res, next) {
18   res.send('respond with a product PUT');
19 });
20
21 router.delete('/', function(req, res, next) {
22   res.send('respond with a product DELETE');
23 });
24
25 module.exports = router;
```

Archivos generados y agregados



4. En la carpeta de vistas (*views*) van los archivos *hbs* de las vistas que queremos tener en nuestra aplicación, que se renderizan en las distintas rutas. La vista de *index* y de *error* se generan por defecto.

```
🐼 error.hbs X
myapp > views > 🐼 error.hbs > ...
1 <h1>{{message}}</h1>
2 <h2>{{error.status}}</h2>
3 <pre>{{error.stack}}</pre>
```

```
🐼 index.hbs X
myapp > views > 🐼 index.hbs > ...
1 <h1>{{title}}</h1>
2 <p>Welcome to {{title}}</p>
```

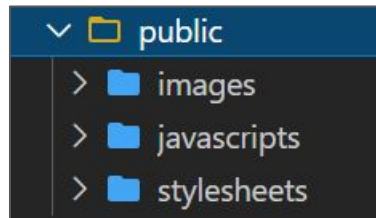
```
🐼 form.hbs X
myapp > views > 🐼 form.hbs > ...
1 <h2>Entrada de productos</h2>
2 <br>
3 <form action="/products" method="post">
4   <input type="text" name="nombre" placeholder="ingrese nombre">
5   <input type="number" name="precio" placeholder="ingrese precio">
6   <button>Enviar</button>
7 </form>
```

```
🐼 layout.hbs X
myapp > views > 🐼 layout.hbs > ...
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title>{{title}}</title>
5     <link rel="stylesheet" href="/stylesheets/style.css" />
6   </head>
7   <body>
8     {{{body}}}
9   </body>
10 </html>
```

Archivos generados y agregados



5. Finalmente, en la carpeta *public*, podemos incluir los archivos de Javascript de front-end, las imágenes que necesitemos en nuestra aplicación, y los archivos de hoja de estilo para el diseño de las vistas, cada uno en la carpeta correspondiente que ya vienen incluídas.



Estos son los archivos que genera de forma automática el módulo de Express generator. Nos facilita la tarea al comenzar a realizar una aplicación, dándonos una base, con un servidor y algunas dependencias ya por defecto.



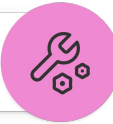
API REST CON EXPRESS GENERATOR

Tiempo: 15 minutos

API REST con Express Generator

Tiempo: 10 minutos

Desafío
generico



Instalar en forma global el generador de aplicaciones express y generar una aplicación de servidor que utilice handlebars como motor de plantillas.

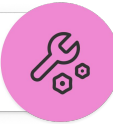
Modificar dicho servidor creando una apiRest full que me permita incorporar productos con su nombre y precio correspondiente y listarlos en forma total. Además debemos poder listar, editar y borrar cada producto por su id único, generado al momento de almacenarlos.

Para agregar un producto, emplearemos un formulario post para la entrada del nombre y precio. Dicho formulario lo ofreceremos en el root del servidor, a través de index.hbs. Para el resto de los endpoints podemos chequear su correcto funcionamiento usando un cliente HTTP.

API REST con Express Generator

Tiempo: 10 minutos

Desafío
generico



El servidor estará escuchando peticiones en el puerto 8080.

Verificar el funcionamiento del logger HTTPmorgan y al listar los usuarios en forma total, utilizar el módulo debug para mostrarlos en consola.

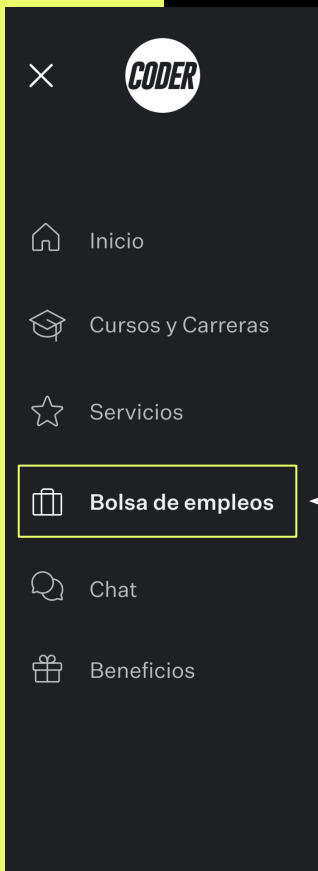
Notas:

- Renombrar `.bin/www` a `.bin/www.js` para el correcto funcionamiento de nodemon.
- No crear capas adicionales del servidor, la idea es hacer algo muy simple, que permita manejar la estructura que propone express generator.



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

MANEJO Y DOCUMENTACIÓN DE APIs

¿De qué se trata?



- Es fundamental mantener y distribuir una buena documentación de tu API.
- Una documentación bien estructurada facilita el proceso de aprendizaje sobre el nuevo entorno que se desarrollará basándose en tu servicio API.
- Debemos cumplir ciertos requisitos que diferencian una buena documentación “simplificada” de una mala documentación con páginas indescifrables en idiomas que ni conocías su existencia.

¿De qué se trata?



- Es una de las piezas básicas que determinan la experiencia de un desarrollador al hacer uso de nuestros servicios API, y va más allá del formato de respuesta API o el método de autenticación.
- La documentación debe ser interactiva, mostrando a los desarrolladores la estructura y la organización que les facilitará el trabajo.

Herramientas de documentación API: Swagger



- **Swagger** <https://swagger.io/> es una herramienta extremadamente útil para describir, producir, consumir y visualizar APIs RESTful. Es multiplataforma.
- El principal objetivo de este framework es unificar el sistema de documentación API con el propio código desarrollado para que esté sincronizado en cada cambio. Con Swagger documentamos nuestra API al mismo tiempo que creamos una nueva implementación.



Herramientas de documentación API: Swagger



- Como resultado final proporciona una interfaz web gráfica a modo de sandbox donde podemos testear los endpoints API a la vez que estamos consultando la documentación.
- Es útil tanto para desarrolladores como para usuarios no experimentados permitiendo probar en la misma documentación antes de programar.



EJEMPLO DE DOCUMENTACIÓN CON SWAGGER

Herramientas de documentación API: Swagger



```
# Product
components:
  schemas:
    Product:
      type: object
      required:
        - id
        - title
        - price
        - thumbnail
      properties:
        id:
          type: string
          description: The auto-generated id of the book.
        title:
          type: string
          description: The title of the product.
        price:
          type: number
          description: The price of the product.
        thumbnail:
          type: string
          description: The URL of the product thumbnail.
      example:
        id: 4ughd73658fnsk85dh58sk
        title: desktop computer
        price: 123.78
        thumbnail: http://photo.url.png
```

Acá vemos como declaramos, dentro del campo 'components/schemas', el esquema de Producto. Al ser un objeto con campos lo definimos como 'type: object'. Agregamos primero los campos requeridos, y luego definimos las características particulares de cada campo (particularmente su tipo de dato y una descripción para la documentación).

Finalmente, podemos agregar un producto de ejemplo.

Herramientas de documentación API: Swagger



```
# NewProduct
components:
  requestBodies:
    NewProduct:
      type: object
      required:
        - title
        - price
        - thumbnail
      properties:
        title:
          type: string
          description: The title of the product.
        price:
          type: number
          description: The price of the product.
        thumbnail:
          type: string
          description: The URL of the product thumbnail.
      example:
        title: desktop computer
        price: 123.78
        thumbnail: http://photo.url.png
```

En el campo ‘components/requestBodies’, podemos definir esquemas de objetos utilizados dentro de los cuerpos de las peticiones y respuestas, para evitar redundancia a futuro. En este caso, definimos que el objeto que se recibe para crear un nuevo producto es similar al Producto, pero carece de id (ya que este se agrega luego como parte de la lógica del negocio).

Herramientas de documentación API: Swagger



```
# create product
paths:
  /productos:
    post:
      summary: Creates a new product
      tags:
        - Productos
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/requestBodies/NewProduct'
      responses:
        "201":
          description: The created product.
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Product'
```

Finalmente, definimos las rutas en el campo 'paths'. En este primer caso, definimos la ruta '/productos', y dentro de ella, el método 'post'. Acá podemos definir la descripción del método, agregarle una etiqueta que defina a qué subcategoría de métodos pertenece (en este caso, a los métodos de Productos), y definimos las características de su petición y su respuesta, haciendo uso de los esquemas ya definidos anteriormente.

Herramientas de documentación API: Swagger



```
# get all products
paths:
  /productos:
    get:
      summary: Gets every product available
      tags:
        - Productos
      responses:
        "200":
          description: Every product available.
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/Product'
```

En este caso, vemos como podemos devolver un array de productos, definiendo el tipo del esquema como 'array' y definiendo que los ítems que contiene ese array serán de tipo Producto.

Herramientas de documentación API: Swagger



```
# get product by id
paths:
  /productos/{id}:
    get:
      summary: Gets the product with given id
      tags:
        - Productos
      parameters:
        - name: id
          in: path
          description: path parameter takes the product id
          required: true
          type: string
      responses:
        "200":
          description: The product with given id
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Product'
        "404":
          description: Product not found Error
```

En el caso de que el método pudiera devolver múltiples respuestas, como es el caso de los errores, podemos incluir fácilmente todas ellas, dentro del campo 'responses', encabezadas por su código de estado, y seguidas de su descripción y contenido como en los ejemplos anteriores. Si la respuesta no tiene ningún contenido, éste puede obviarse.

CONFIGURACIÓN DEL PROYECTO

Configuración del proyecto



```
{
  "name": "swagger-demo",
  "version": "1.0.0",
  "main": "main.js",
  "scripts": {
    "start": "nodemon ."
  },
  "dependencies": {
    "express": "^4.17.1",
    "swagger-jsdoc": "^6.1.0",
    "swagger-ui-express": "^4.1.6"
  }
}
```

Para que todo esto funcione, debemos instalar las dependencias correspondientes:

- express,
- swagger-jsdocs,
- swagger-ui-express,

todas disponibles vía NPM.

Configuración del proyecto



```
const express = require("express")
const swaggerUi = require("swagger-ui-express");
const swaggerJsdoc = require("swagger-jsdoc");

const options = {
  definition: {
    openapi: "3.0.0",
    info: {
      title: "Express API with Swagger",
      description: "A simple CRUD API application made with Express and documented with Swagger",
    },
  },
  apis: [ './docs/**/*.yaml' ],
};

const swaggerSpecs = swaggerJsdoc(options);

const app = express();

const specs = swaggerJsdoc(options);
app.use("/api-docs", swaggerUi.serve, swaggerUi.setup(specs));

const PORT = process.env.PORT || 8080
app.listen(PORT, () => {
  console.log(`Documentación disponible en http://localhost:${PORT}/api-docs`)
})
```

Luego, como se vé en el código, se definen las opciones de configuración de nuestras librerías, y se carga el middleware responsable de generar la documentación en html y servirla en la ruta correspondiente.

Una vez hecho esto, podemos levantar el servidor y dirigirnos a al ruta de la documentación:

localhost:8080/api-docs

BUENAS PRÁCTICAS PARA MANTENER UNA BUENA DOCUMENTACIÓN API


Descripción endpoint detallada



- Ampliar la descripción de los endpoints con texto explicativo de la forma más detallada posible, utilizando siempre un *lenguaje comprensible*, intentando no adquirir un tinte demasiado especializado o técnico porque no todos los desarrolladores tienen la misma experiencia o están familiarizados en el entorno de nuestra API.
- Aunque las API actúan como una capa de abstracción que sirve para ocultar ciertos detalles de las operaciones que se ejecutan, lo mejor para no confundir a los desarrolladores es precisamente *evitar abstracciones en la documentación*.

Detallar los parámetros y las respuestas esperadas



 Listar todos los parámetros de entrada y salida que proporcione cada endpoint, además de los tipos de respuesta HTTP esperados. Ésto evitará sorpresas a los desarrolladores que consuman nuestra API teniendo en cuenta las diferentes casuísticas que deberán valorar en el momento de desarrollar un frontend apropiado.

Un entorno multiplataforma



- Debemos señalar los diversos lenguajes de programación aceptados por nuestra API e, incluso, dar los ejemplos en cada uno de ellos.
- Una posibilidad es permitir que los usuarios elijan el lenguaje de programación que prefieran desde el principio, como es el caso de la documentación de Stripe API que ofrece una opción entre Curl, Ruby, Python, PHP, Java, Node.js, Go o .NET.

Versionado

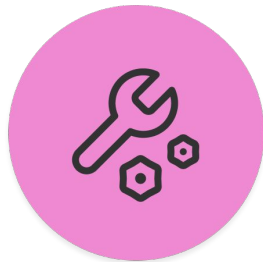


- El versionado es la clave del éxito a largo plazo de nuestra API. Los desarrolladores esperan que el funcionamiento no se vea alterado por nuevas implementaciones o innovaciones.
- Es muy importante facilitar el acceso a la documentación API de todas las versiones que se encuentran en producción, pero también lo es el dar acceso a la documentación API de versiones o funcionalidad deprecated que ya no lo estén.

Tutoriales como ejemplo



- Una muy buena opción es agregar tutoriales que actúan como un manual para mostrar a los desarrolladores ciertas funciones y ejemplos específicos de lo que pueden hacer con nuestra API.
- Conviene añadir en cada demostración el código que se requiere para llamar a la API desde la aplicación y ofrecer ejemplos del tipo de respuesta que se obtendrán.



DOCUMENTANDO CON SWAGGER

Tiempo: 10 minutos

DOCUMENTANDO CON SWAGGER

Tiempo: 15 minutos

Desafío
generico



Basándonos en el servidor del desafío anterior, realizar la documentación de su API REST utilizando swagger.

Cargar dicha documentación en la ruta '**/api/docs**' del servidor.

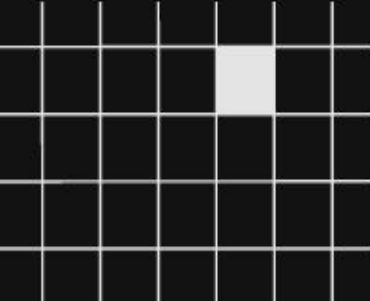
¿PREGUNTAS?





¡MUCHAS GRACIAS!

Resumen de lo visto en clase hoy:

- Conceptos de API, API REST.
 - Express Generator.
 - Documentación de APIs
- 



OPINA Y VALORA ESTA CLASE

#DEMOCRATIZANDO LA EDUCACIÓN

CODER HOUSE