Express.js : API REST

What is a REST API? Click here!

A REST API must follow the principles of the model **CRUD**.

CRUD is an acronym that refers to the four basic operations that can be performed on data in an application:
Create(C), **Read**(R), **Update**(U) i **Delete**(D).

Create (**HTTP POST**): This operation is used to add new records to the database.
Example: Adding a new user to the database.

Read (**HTTP GET**): This operation allows you to retrieve existing data, one or more records.
Example: Get the list of all users.

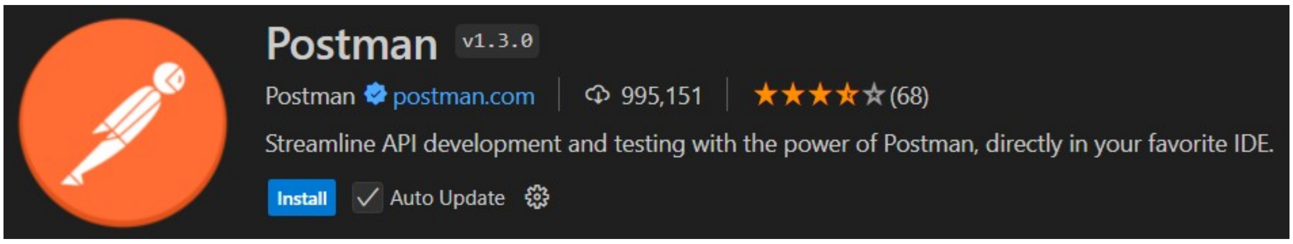
Update (**HTTP PUT** or **HTTP PATCH**): This operation modifies existing records.
Example: Update the information of a specific user.

Delete (**HTTP DELETE**): This operation is responsible for removing records from the database.
Example: Delete a user from the database.

To test and verify that a REST API correctly follows the CRUD principles can be used **Postman**, a very popular tool that makes it easy to send HTTP requests and view responses.

Postman allows simulate CRUD operations by sending HTTP requests such as POST, GET, PUT, PATCH and DELETE directly from its interface. Additionally, you have the ability to configure parameters and headers, adding HTTP headers, request parameters, and variables to emulate real-world situations, making it easy to send information to the server accurately. The tool also allows you to view the API responses, so you can analyze the data returned by the server to verify that the results are correct for each operation.

We can install it as an extension of **Visual Studio Code**:



Postman v1.3.0
Postman postman.com | 995,151 | ★★★★☆ (68)
Streamline API development and testing with the power of Postman, directly in your favorite IDE.

To make requests click **"New HTTP Request"** in the Postman extension.
Check the results using the **Postman Console**.

Regulate API access with CORS

CORS (Cross-Origin Resource Sharing) is a security mechanism that allows web pages to request information from other pages that are from different domains. By default, this is not possible.
This system controls and allows access to shared resources between different origins (domains), helping to prevent attacks such as cross-site scripting (XSS).
CORS works by using HTTP headers that indicate whether a resource can be shared with a different origin.

With the following code, you manage to enable CORS in your API Express with the default settings:

```
import cors from 'cors';

// Enable CORS with default settings (allow all origins)
app.use(cors());
```

You allow access to your API from any source, making testing and development easier. In other words, you can test the server using a web page located on a computer on the same network.

But, by not limiting access, you expose your API to possible security risks. Therefore, for production environments, it is recommended to restrict access by specifying which domains can make requests to your API.

User authentication with JWT

JWT (JSON Web Token) is used in web applications to manage user authentication using tokens that securely store information and can be used to identify and authenticate users.
JWT is a type of **Bearer Token**.

Example of JWT usage:
https://github.com/xbaubes/DesenvolupamentWeb/tree/main/Backend/Node.js/Express.js/JWT
For greater security, we will use environment variables to save the key using the **dotenv** package.
The file is in **.mjs** format because it uses the ES6 import without configuring the package.json.

Code Operation example:

1. Start of Session
   To authenticate as an API user you need to obtain a token, you must make a POST request to the **/login** endpoint with the following information in JSON format in the body of the request:

   ```
   {
         "username": "user",
         "password": "password"
   }
   ```

   If the login is correct, you will receive a token in the body of the response.
   If the credentials are incorrect, you will receive an error response with a 401 status code.

2. Access to Protected Routes
   Save the token, with it you can make requests to protected routes.
   Do a GET request to **/perfil**, including the token in the header **Authorization**.

   If the token is valid, you will receive a welcome message with the username in the response body.
   If the token is invalid or not provided, you will receive a 403 or 401 status code respectively.

Validation and sanitizing data with express-validator

Simple example for the validation of user data:

```
// npm install express express-validator
import express from 'express';
import { body, validationResult } from 'express-validator';
const app = express();
const port = 3000;

app.use(express.json());

// Path to create a user
app.post('/users', [
      // Field validation
      body('name').notEmpty().withMessage('Name is required.'),
      body('email').isEmail().withMessage('The format of email is incorrect.'),
      body('password').isLength({min: 6}).withMessage('Password minimum 6 characters.')
], (req,res)=>{
      // Check for validation errors
      const errors = validationResult(req);
      if(!errors.isEmpty()) {
            return res.status(400).json({errors: errors.array() });
      }

      // If everything is correct, process the request
      const {name,email,password} = req.body;
      res.status(201).json({message: 'User created', name, email});
});

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

body and **validationResult** are express-validator functions to define validation rules and handle errors.

If we try to create a user with the following data:

```
{
	"name": "John",
	"email": "john@example.com",
	"password": "secret123"
}
```

We will pass the validation.Response Body:

```
{
	"message":"User created","name":"John","email":" john@example.com "
}
```

If we try to create a user with the following data:

```
{
	"email": "john@example.com",
	"password": "secret123"
}
```

We will not pass the validation. Response Body:

```
{
	"errors":[{"type":"field","msg":"Name is required.","path":"name","location":"body"}]
}
```

Dynamic routes

Dynamic routes in Express allow you to create URLs that include variable parameters. This makes it easier to manage resources with variable identities, such as getting an element by ID.
Fields included in dynamic routes are mandatory when making the call and cannot be disordered.

```
app.get('/dynamic/:id',function(req,res){
	res.send('The id you specified is ' + req.params.id);
});
```

A call to the following endpoint: http://localhost:3000/dynamic/12345
It would result in **req.params**.id would have the value: 12345.
":id" is a variable that can take different value depending on the call, allowing Express to capture that value and use it within the route.
This flexibility allows handling a wide variety of requests with a single route, as the dynamic parameter ":id" can represent different resource identifiers depending on the request.

If we put 2 or 3 dynamic routes in a row, we will use a structure that includes informational fields:

Declaration	Use
/products/:products/brand/:brand	/products/tv/brand/panasonic

We will not put more than 3 dynamic routes in a row, if we need more fields then we will complement it using query parameters, unlike dynamic routes they are not mandatory and can be messed up.
These are best practices in a RESTful API.

ACTIVITIES

1. API for book management

API must allow to manage bookstores, books and their sales.
The layout includes 3 data models related to each other, saved in the following files JSON: https://github.com/xbaubes/DesenvolupamentWeb/tree/main/Backend/Node.js/Express.js/books

Tasks:

- Access to the data requires authentication. An endpoint must be provided for user creation that returns a **JWT** to access the data. Registered users and their passwords will be stored in an encrypted file: users.json.
- Assess which **CRUD** operations make sense for each model and design them; ensure you understand the relationships between models before starting. Notes: The inventory of the bookstore must be kept up to date. sales.json is a record of completed sales, so updating or deleting data is unnecessary; consider how to handle returns.
- Only return responses in **JSON** format, using res.json().
- Return **sorted** data. You can use JavaScript's sort() method.
- **Validate** incoming data, ensuring it meets the expected format, type, and range before saving.
- **Sanitize** incoming data, such as removing unnecessary spaces.
- **Handle any error** that may occur.
- Return appropriate **HTTP status codes** for each request.
- Implement **CORS** to allow access from other domains to the API.
- Data identifiers are generated using **uuid**, a 128-bit identifier that, due to probability, is almost guaranteed to be unique.
- Generate **logs** for interactions with the API: log all requests to the API and their results.
- Include start in the scripts object of the package.json. If correctly defined, for example if the main file is server.js it would be: "start": "node server.js".
  It allows starting the server without knowing anything about the project using the command: npm start.
- Update **updatedAt** included in each file with each data modification. You can use JavaScript's Date object.
- The API will start with the following route: **api/v1**. Using a version indicator like v1 allows for managing changes and incompatibilities as the API evolves. We could keep the old version to avoid issues for applications that use it while developing new features in v2.
- It should allow **filtering** of data according to specified fields. Implement filtering using parameters sent by URL with GET. For example:
  - Retrieve books written by a particular author.
- Develop **operations with more complexity or that relate multiple files**. For example:
  - Obtain the number of copies sold of a specific book.
  - Retrieve bookstores that have more than 20 copies of a specific book.
  - Get the revenue of a specific bookstore.

Project Organization:

```
api
│
├── routes/
├── controllers/
├── models/
│
├── middleware/
│
├── server.js // Main application file
├── config.js //cconfiguration API; for example, the port of the server
│
├── .env //Environment variables
├── .gitignore // Files and folders ignored by Git; for example, .env
│
├── server.log //API logs
│
├── package.json
├── package-lock.json
└── node_modules/
```

Create a routes file and handler for each JSON data file, this will make the application easier to maintain and scalable.

You can further modularize the application, for example, by separating data access.

BIBLIOGRAPHY AND WEBGRAPHY

«Brave Developer». https://bravedeveloper.com/
«DiegoSF». https://diegosf.es/big-data-y-analisis-de-datos/que-es-crud/
«ExpressJS». https://expressjs.com/es/api.html
"Microsoft Learn". https://learn.microsoft.com/es-es/training/