# Software Engineering

A Faculty of Engineering Course: CSEN 303

## Back-end Design & Technologies with the PERN stack

8

**Dr. Iman Awaad**

iman.awaad@giu-uni.de

GIU GERMAN INTERNATIONAL UNIVERSITY
الجامعة الألمانية الدولية

# Acknowledgments

The slides are **heavily** based on the **slides** by **Prof. Dr. John Zaki**.

His contribution is gratefully acknowledged.

Any additional sources are referenced.

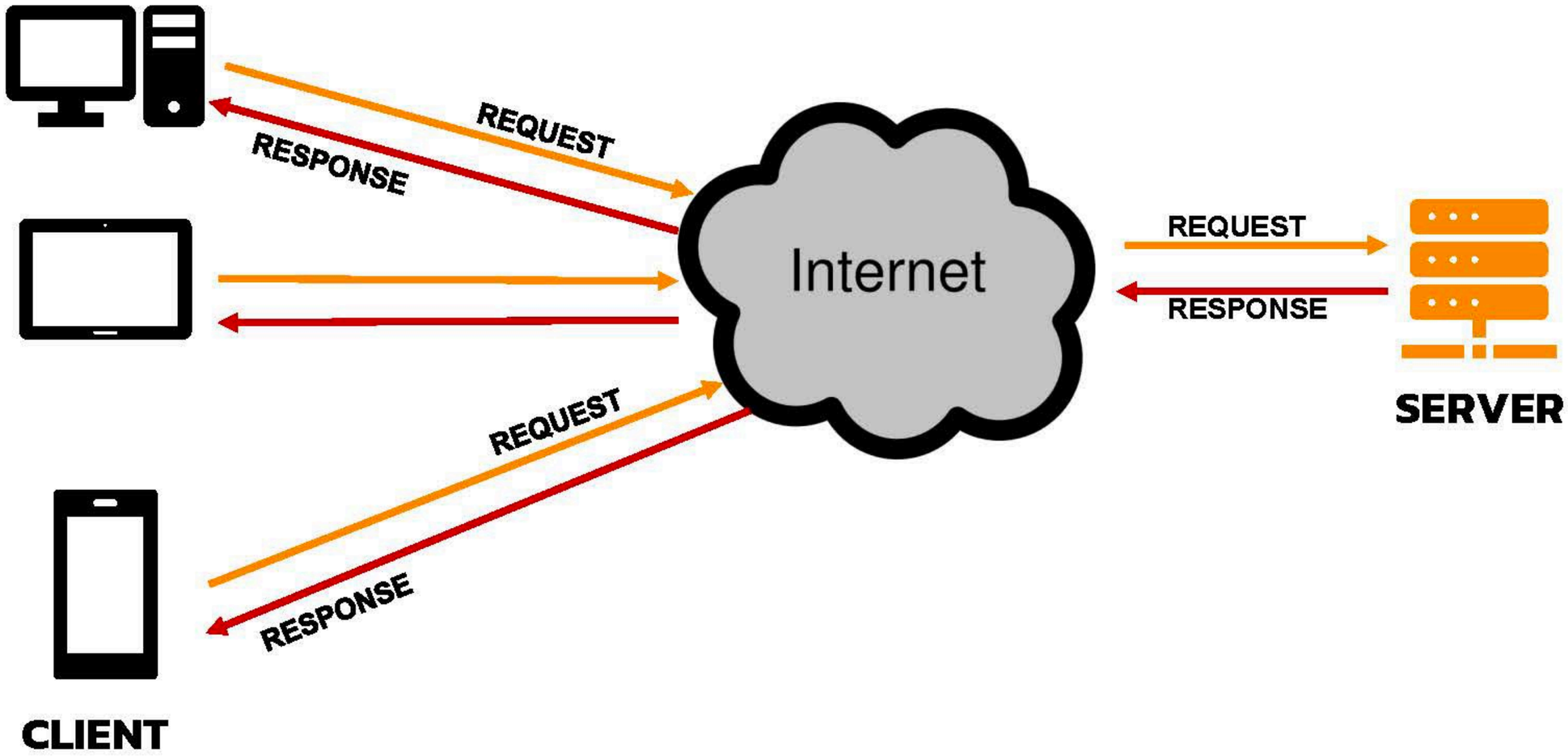
# Back-end Design & Technologies

- PERN stack
- Backend development
- Middleware
- Express Server
- Routes
- Postman
- DB Query

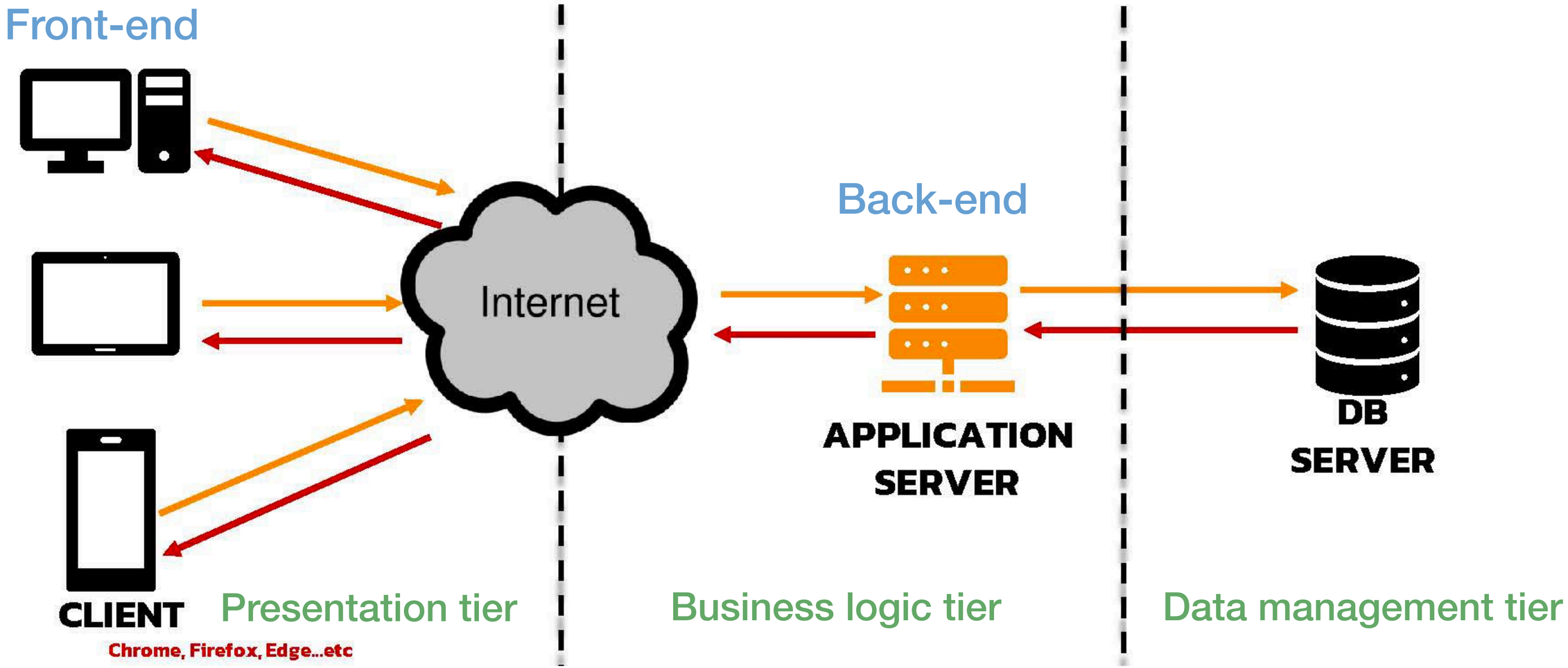
How do we build and deliver robust and dynamic web applications?

# The Internet in a nutshell…

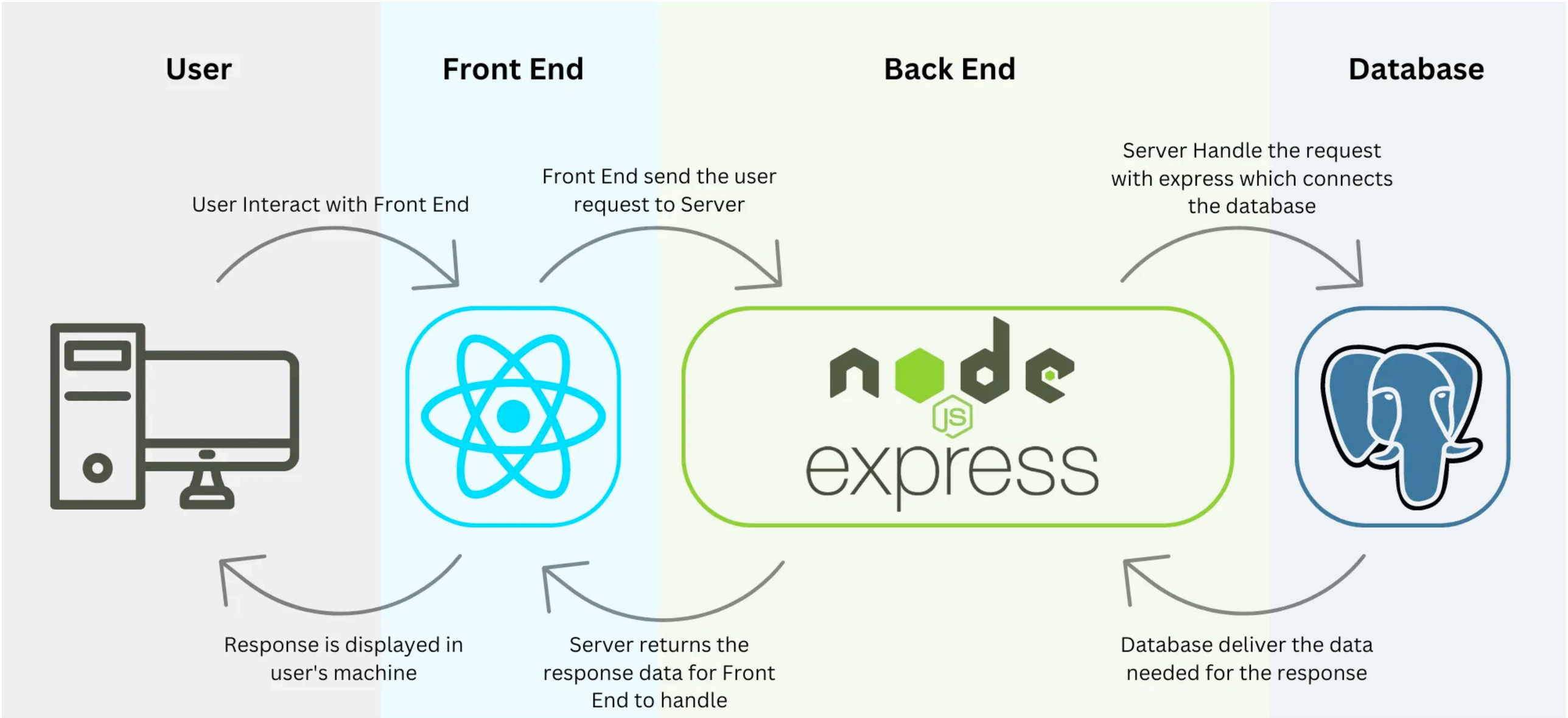
Remember this?



# Web application

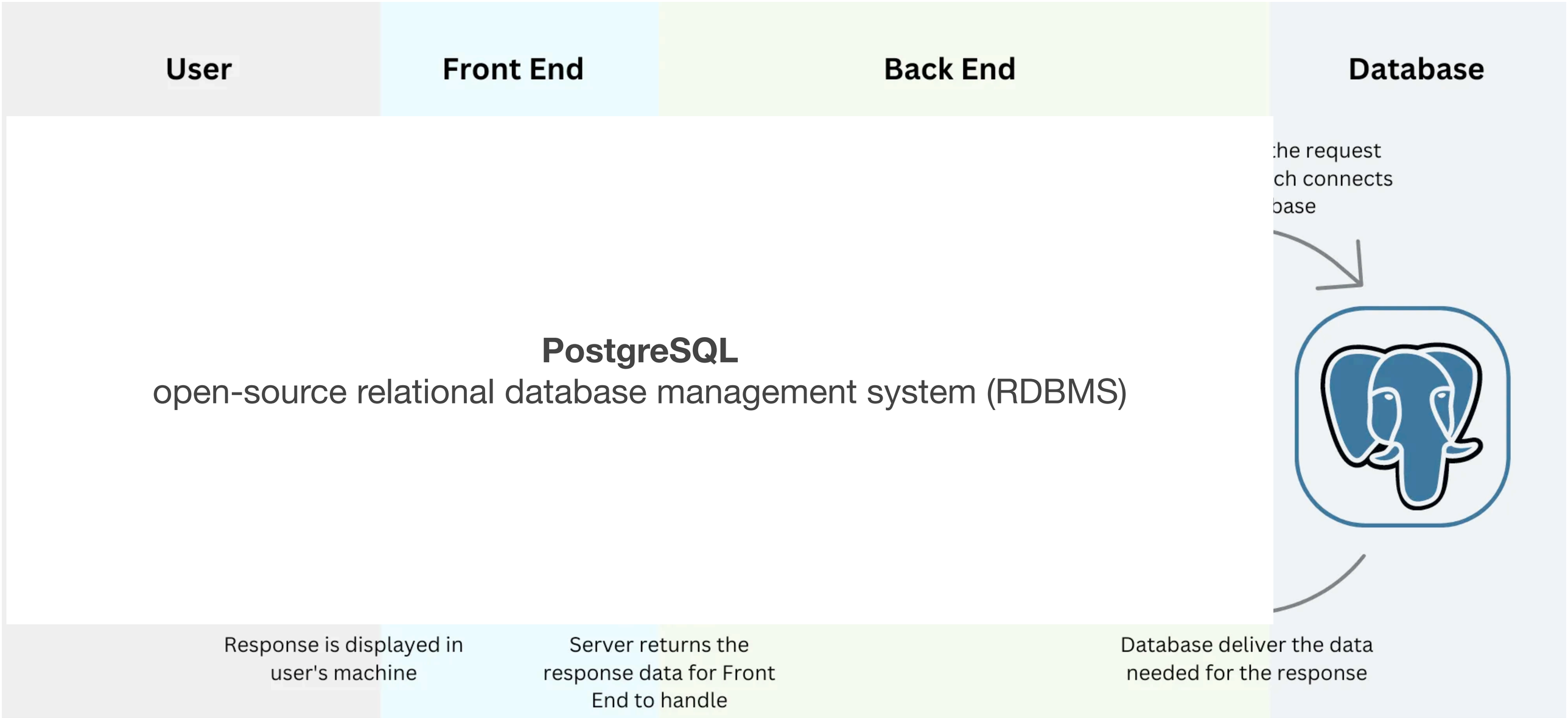


# The PERN stack




https://medium.com/@snehadissanayaka56/exploring-the-pern-stack-build-robust-and-dynamic-web-applications-1bf04d6e5c82

# The PERN stack



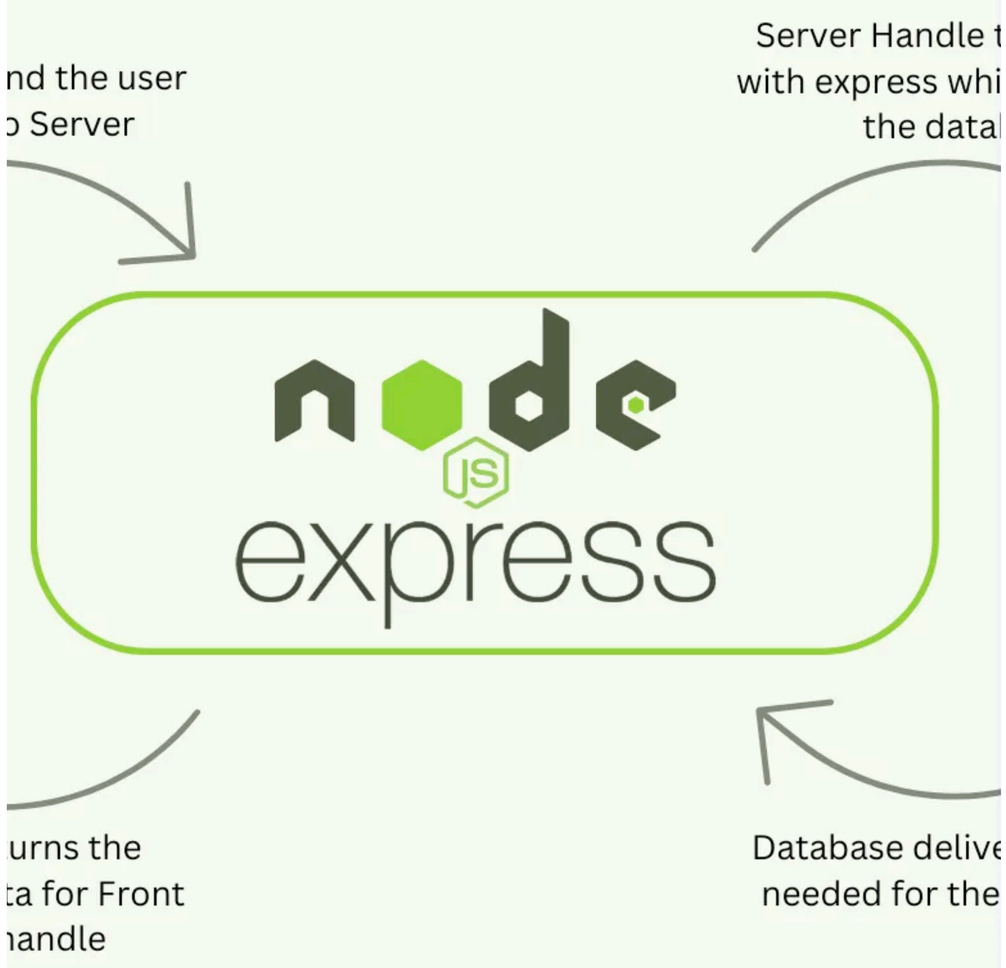

https://medium.com/@snehadissanayaka56/exploring-the-pern-stack-build-robust-and-dynamic-web-applications-1bf04d6e5c82

# The PERN stack

| User | Front End | Back End | Database |
|---|---|---|---|

**Node**
open-source server-side JavaScript runtime

Enables developers to build scalable, efficient, real-time applications, APIs and web servers




**Express.js**
minimal, flexible web application framework for **Node.js**
Enables developers to define routes, handle requests and responses, manage middleware, and connect to databases

https://medium.com/@snehadissanayaka56/exploring-the-pern-stack-build-robust-and-dynamic-web-applications-1bf04d6e5c82

# Back-end development

## Server-side logic & DB interaction

**Back-end/server-side development of a web application involves**

working with the 1) **server**, 2) **DB** and 3) **application logic**
to ensure that the front-end and back-end work together seamlessly to process requests from the front-end, and handle the DB in order to return the correct response!

# Back-end development

## Server-side logic & DB interaction

**Back-end/server-side development of a web application involves**

working with the 1) **server**, 2) **DB** and 3) **application logic**
to ensure that the front-end and back-end work together seamlessly to process requests from the front-end, and handle the DB in order to return the correct response!

### Server

provides functionality for other programs or devices (*i.e.* clients)

### DB

organized collection of data

### Application Logic

code that defines the behaviour, rules, and operations of the application, *e.g.*

processing user inputs, performing calculations, interacting with the DB

# Back-end development

## Server-side logic & DB interaction

**Back-end/server-side development of a web application involves**

working with the 1) **server**, 2) **DB** and 3) **application logic**
to ensure that the front-end and back-end work together seamlessly to process requests from the front-end, and handle the DB in order to return the correct response!

This involves writing **server-side logic** that handles the business logic and functionality of a web application. It ensures that the application behaves correctly and securely. *e.g.* **data processing**, **calculations**, **authentication**, **authorisation**, …

The backend communicates with **databases** to **store**, **retrieve**, and **update** data. It is responsible for **managing database connections**, **executing queries**, and **handling data persistence**

# Back-end development

API (a set of rules that allow two applications to share resources)

**Back-end/server-side development of a web application involves**

working with the 1) **server**, 2) **DB** and 3) **application logic**
to ensure that the front-end and back-end work together seamlessly to process requests from the front-end, and handle the DB in order to return the correct response!

**Back-end development** involves **creating back-end APIs** to enable the **front-end** to **request** and **receive** data from the server

*act as an intermediary layer, intercepting incoming requests before they reach the actual route handler*

APIs can be **public** or **private**. They use **middleware** to protect sensitive **routes**.

**CRUD** (**Create**, **Read**, **Update**, **Delete**) **operations** are typically implemented through these **API endpoints**. **API responses** are usually **formatted** as **JSON** or **XML**

# Back-end development

## API **endpoint** (the location of the resource)

Credit: CBS Photo archive

**One end of a communication channel**, typically represented by a unique URL which combines a base URL (identifying the API or service) with a specific path that points to the desired resource or function.

*e.g.* `https://api.example.com/users/123`

# Back-end development

## API **endpoint**

**Back-end/server-side development of a web application involves**

working with the 1) **server**, 2) **DB** and 3) **application logic**
to ensure that the front-end and back-end work together seamlessly to process requests from the front-end, and handle the DB in order to return the correct response!

*e.g.*

**GET** /books
Retrieves a list of all books

**GET** /books/{id}
Retrieves a specific book by its ID

**POST** /books
Creates a new book

**PUT** /books/{id}
Updates an existing book by its ID

**DELETE** /books/{id}
Deletes a book by its ID

/books is the **base endpoint** for managing books, and {id} is a **placeholder** for the book's unique identifier

# Back-end development

## Integration of third-party services

**Back-end/server-side development of a web application involves**

working with the 1) **server**, 2) **DB** and 3) **application logic**
to ensure that the front-end and back-end work together seamlessly to process requests from the front-end, and handle the DB in order to return the correct response!

**Back-end development** often also involves integrating **third-party services** and **APIs** to enhance the functionality of the application… *e.g.* payment gateways, social media integration, and external data sources

# Back-end development

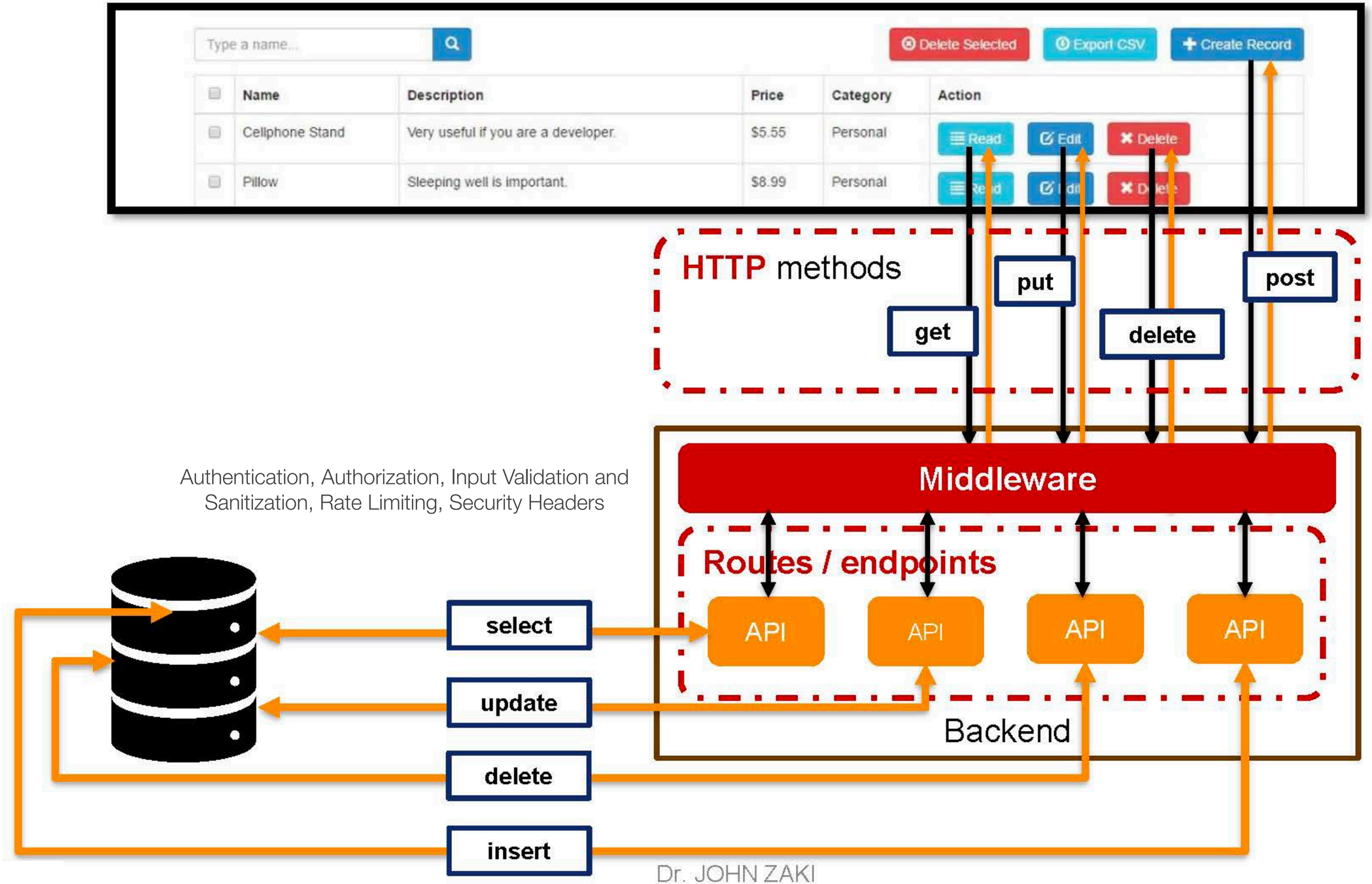
## Error-handling and logging

**Back-end/server-side development of a web application involves**

working with the 1) **server**, 2) **DB** and 3) **application logic**
to ensure that the front-end and back-end work together seamlessly to process requests from the front-end, and handle the DB in order to return the correct response!

The back-end is responsible for **handling errors**. It **logs errors**, **sends appropriate error responses** to the **client**, and **provides diagnostic information** for **debugging**

It has **built-in error handling** and also **enables the definition** of **custom error handling**.




Authentication, Authorization, Input Validation and Sanitization, Rate Limiting, Security Headers

Dr. JOHN ZAKI

# Middleware

…is the **software layer** that sits **between** the **user interface** and the **back-end logic**.

It facilitates **communication** between **applications**, **services**, and **systems** that may not interact directly.

Often considered the “**glue**” holding software components together (manages data exchange, authorisation, and service orchestration).

https://dreamertechnoland.com/difference-between-backend-and-middleware/

# Middleware

…is the **software layer** that sits **between** the **user interface** and the **back-end logic**.

It facilitates **communication** between **applications**, **services**, and **systems** that may not interact directly.

Often considered the “**glue**” holding software components together (manages data exchange, authorisation, and service orchestration).

While the API tells developers what can be done,
middleware determines how it gets done efficiently and securely

https://dreamertechnoland.com/difference-between-backend-and-middleware/

# Middleware

**Authentication** …verifies if a user is authenticated by checking for valid credentials, *e.g.* via an access token or session cookie. If the user is not authenticated, it can deny access or redirect them to a login page.

**Authorisation** …it can enforce authorisation rules to ensure that the authenticated user has the necessary permissions or roles to access a specific resource or perform a particular action.

JavaScript

```
// Example of authentication middleware in Express.js
function authenticateToken(req, res, next) {
    const authHeader = req.headers['authorization'];
    const token = authHeader && authHeader.split(' ')[1];

    if (token == null) return res.sendStatus(401); // Unauthorized

    jwt.verify(token, process.env.ACCESS_TOKEN_SECRET, (err, user) =>
        if (err) return res.sendStatus(403); // Forbidden
        req.user = user;
        next(); // Proceed to the route handler
    });
}
```

JavaScript

```
// Example of authorization middleware
function authorizeRole(role) {
    return (req, res, next) => {
        if (req.user.role !== role) {
            return res.sendStatus(403); // Forbidden
        }
        next();
    };
}
```

# Middleware

**Input Validation and Sanitization** …it can validate and sanitise incoming request data to prevent common vulnerabilities like SQL injection or cross-site scripting (XSS).

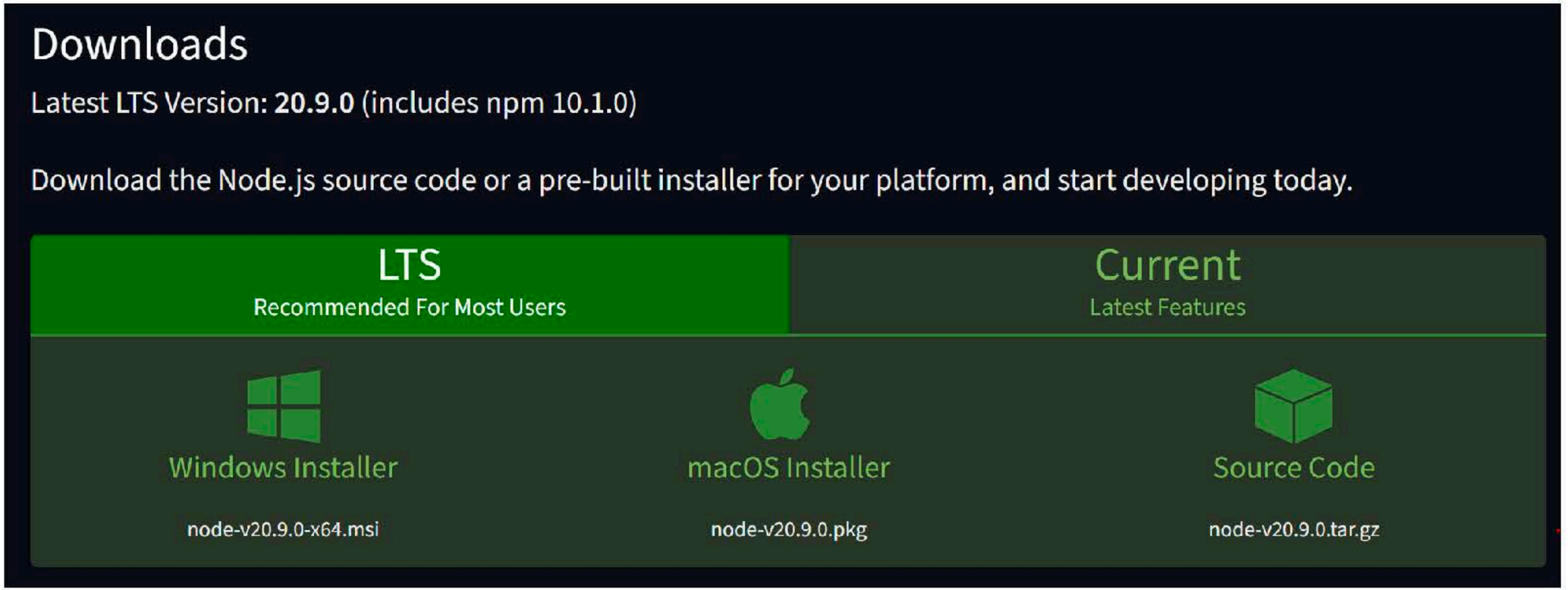
**Rate Limiting** …to prevent abuse and denial-of-service attacks by restricting the number of requests a single client can make within a given timeframe.

**Security Headers** …it can set various HTTP security headers (*e.g.* Helmet in Express.js) to mitigate risks like clickjacking, XSS, and content sniffing.

By centralizing these security checks in middleware, developers can **avoid duplicating logic** across multiple route handlers, leading to **cleaner**, **more maintainable**, and **more secure** code

# Back-end development: Key aspects

## Node setup

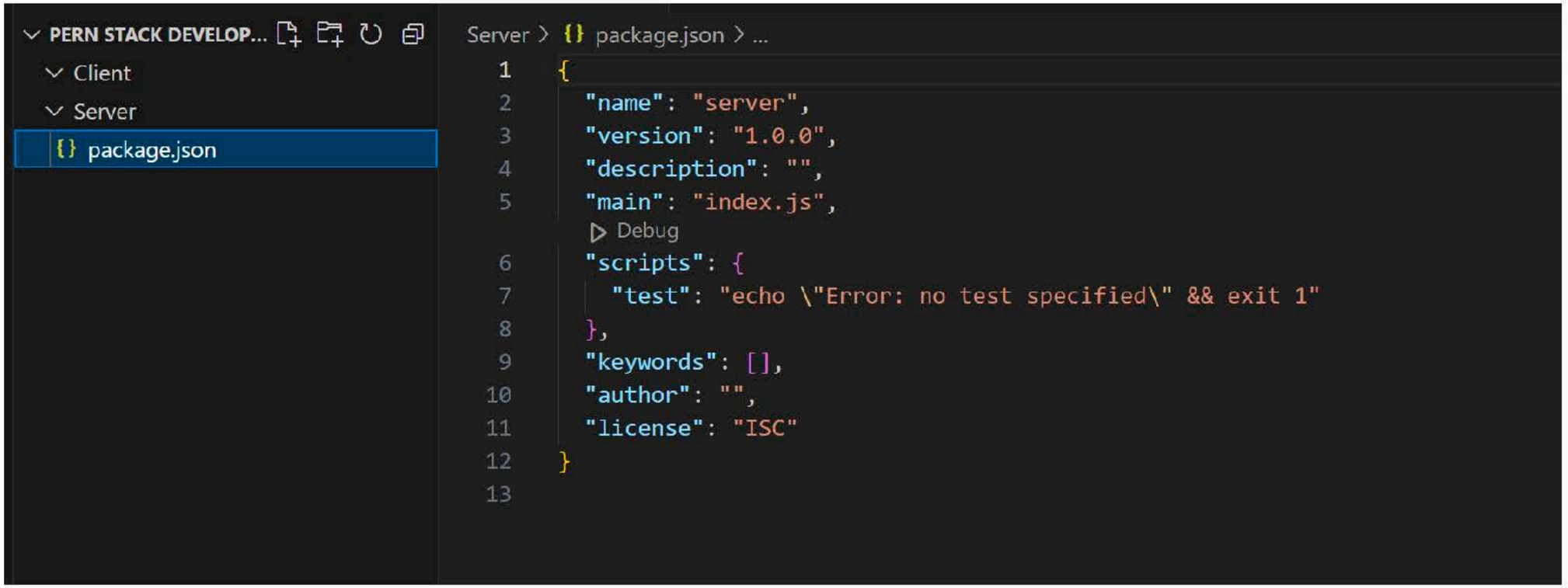



https://nodejs.org/en/download

# Back-end development: Key aspects

## Package.JSON

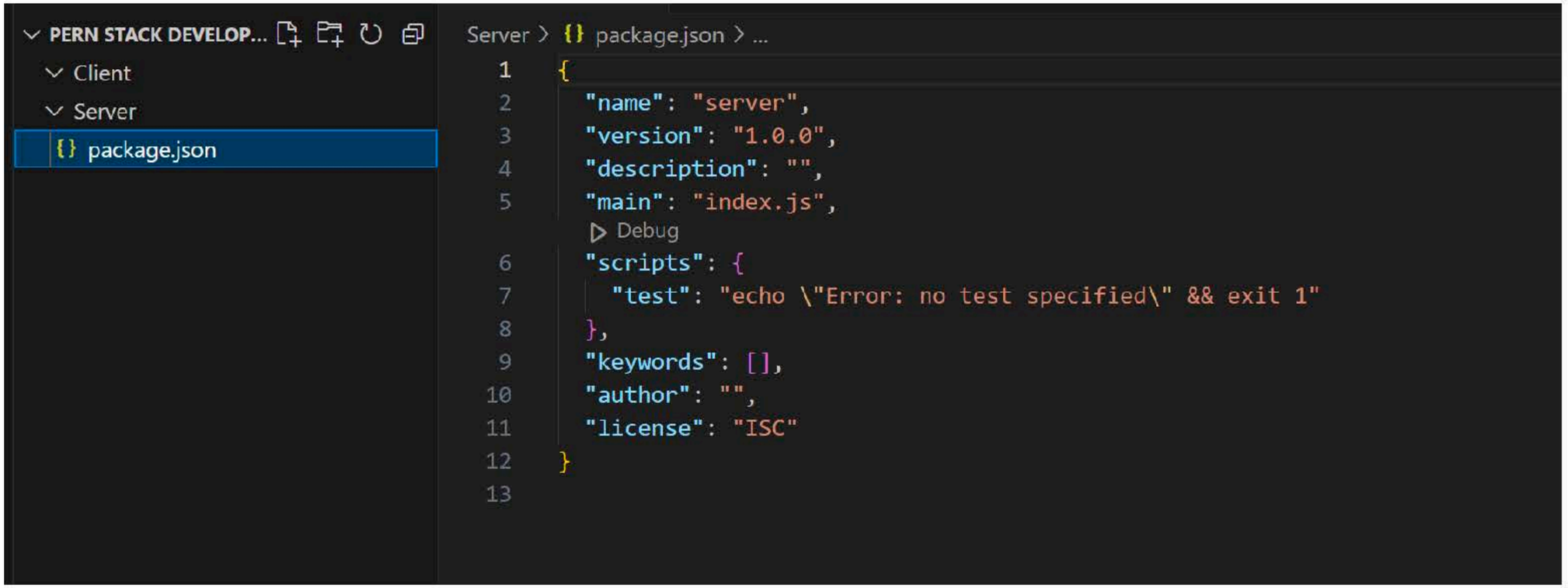
```
npm init -y
```



# Back-end development: Key aspects

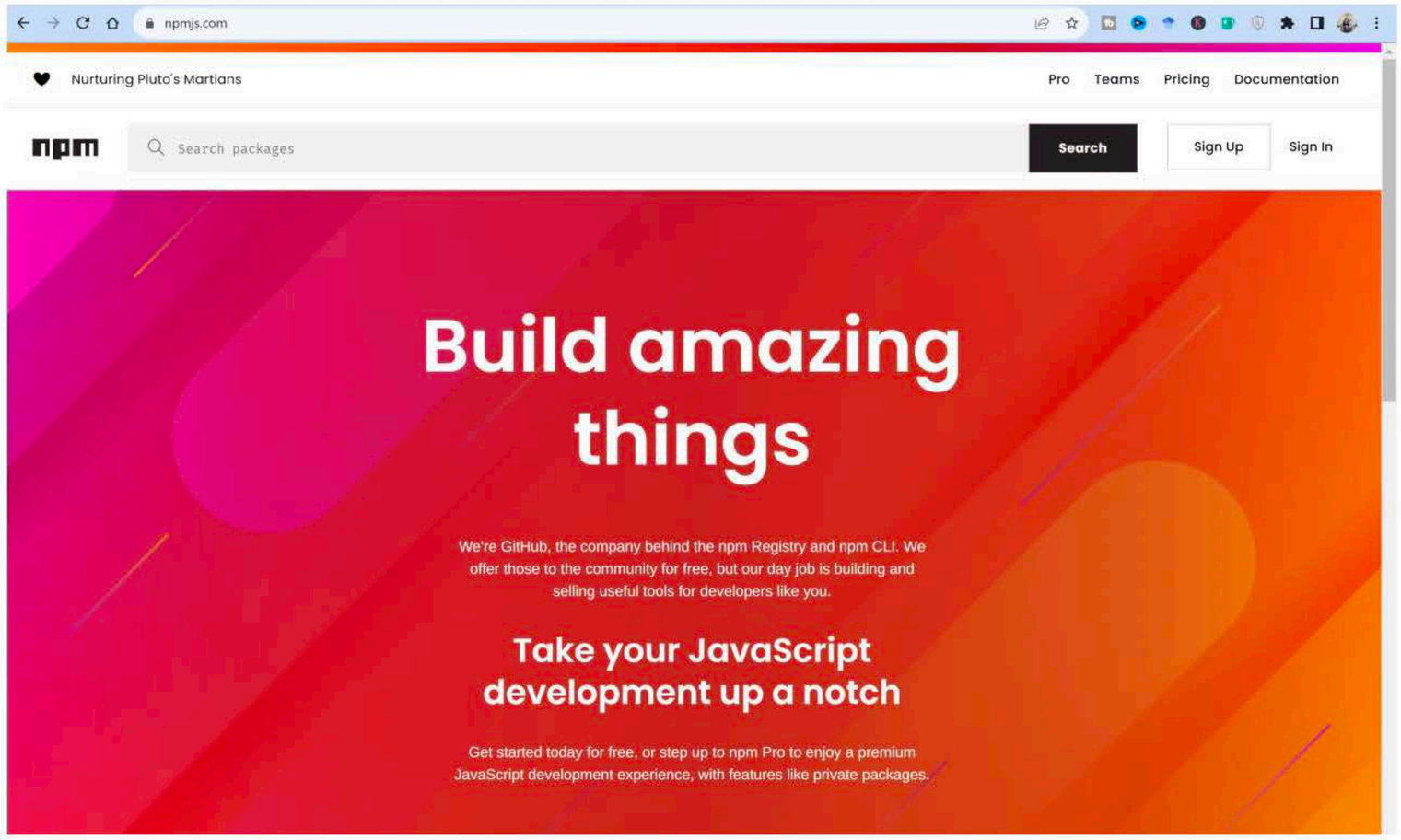
## Package.JSON

```
npm init -y
```



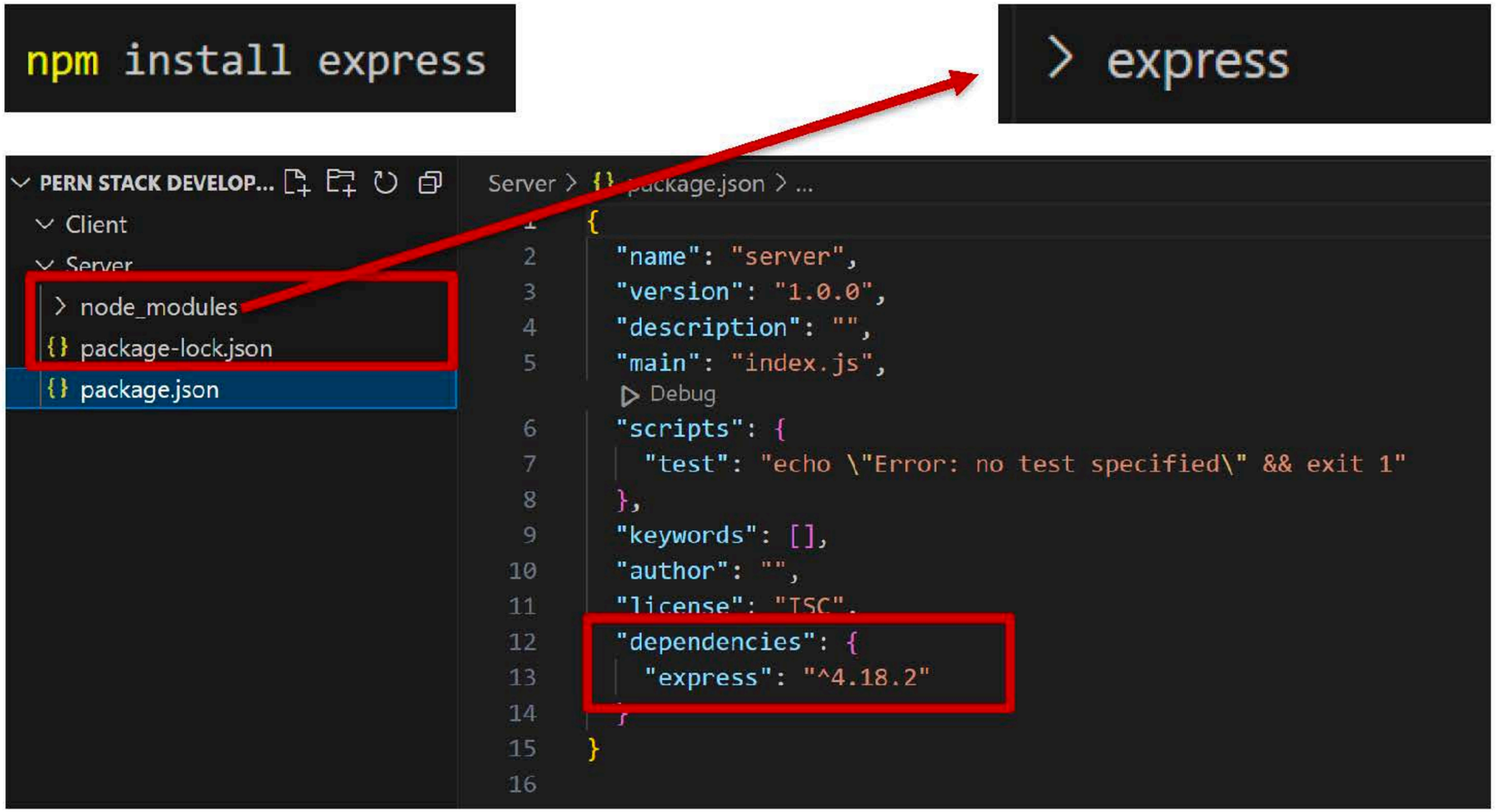
# Back-end development: Key aspects

## NPMJS.COM



# Back-end development: Key aspects

## Install Express



# Back-end development: Key aspects

## Create your entry point

Create your first JS file (`server.js` or `index.js`): the entry point every time you run your application.

For each package or module that you want to include in your application, you should use `require` to import it.

Create an instance of your Express app and start **listening** on a particular **port**. *e.g.* port 3000 or 3001 …Just be sure that no other app is listening on the same port at the same time…

# Express Server

## A basic server…

```
1   // import the express app
2   const express = require("express");
3
4   //create an instance of the express app and store it in a variable called app
5   const app = express();
6
7   // instruct our express app to listen to a certain port
8   const port = 3000
9   app.listen(port,()=>{
10  console.log(`server is running: listening on port ${port}`);
11
12  })
13
```

# Express Server

## Environment Variables

Good practice to keep the environment variables in a separate file (for security purposes and for managing the different environments)

Use package `dotenv` → `npm install dotenv`

…**dynamic**, **named values** that are accessible by processes and applications on a computer to **provide configuration** and **system-specific information**. …used to store settings like temporary file **paths**, **user directories**, or **API keys**, allowing software to run consistently across different environments without needing to change the source code

```
PS C:\Users\Surface\Downloads\GIU Work\winter Semester Preparation\Software Engineering\PERN Stack Development\server> npm i dotenv
```

After installation → check package.json file

```
"dotenv": "^16.3.1",
"express": "^4.18.2"
```

Don't forget to include the package using require

https://en.wikipedia.org/wiki/Environment_variable

# Express Server

## dotenv

### Install

1

```
# install locally (recommended)
npm install dotenv --save
```

Or installing with yarn? `yarn add dotenv`

### Usage

2

Create a `.env` file in the root of your project:

```
S3_BUCKET="YOURS3BUCKET"
SECRET_KEY="YOURSECRETKEYGOESHERE"
```


As early as possible in your application, import and configure dotenv:

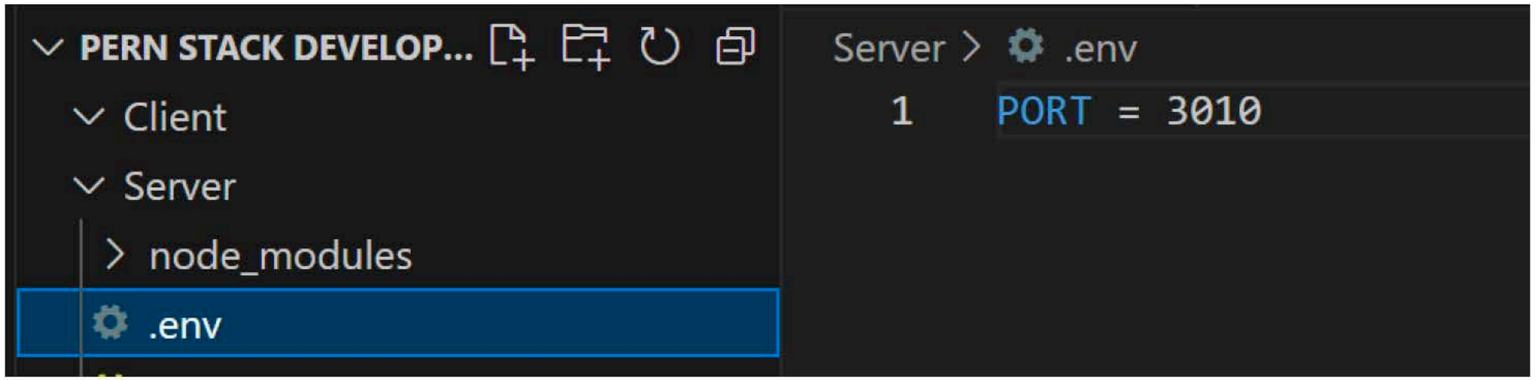
3

```
require('dotenv').config()
console.log(process.env) // remove this afte
```

# Express Server

## dotenv

Common practice to write the environment variables in CAPITAL letters




Set the port in the `server.js` file to read from the environment variable…

```
const port = process.env.PORT || 3001;
```

# Express Server

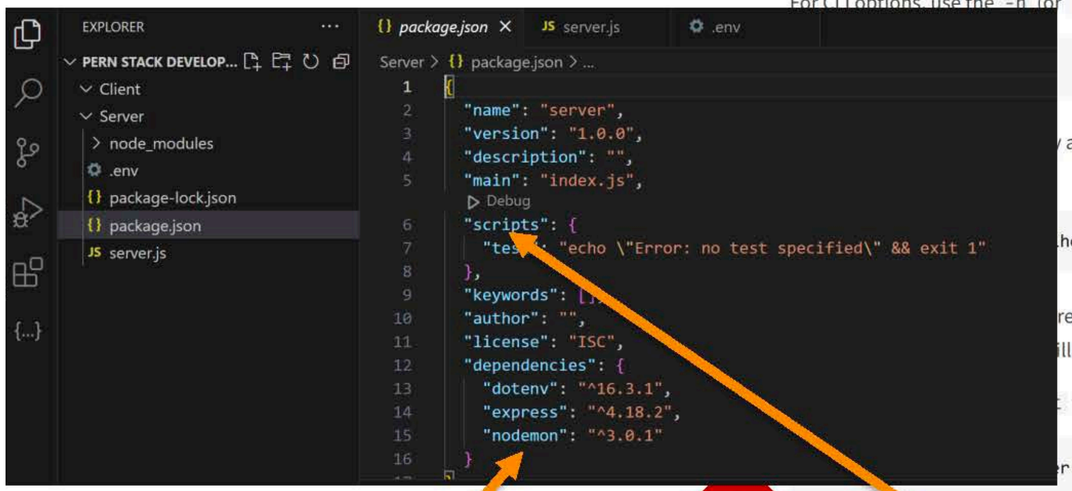
## nodemon

Every time you change something in the code, you need to stop and start your server (because the server keeps running and listening to the port)

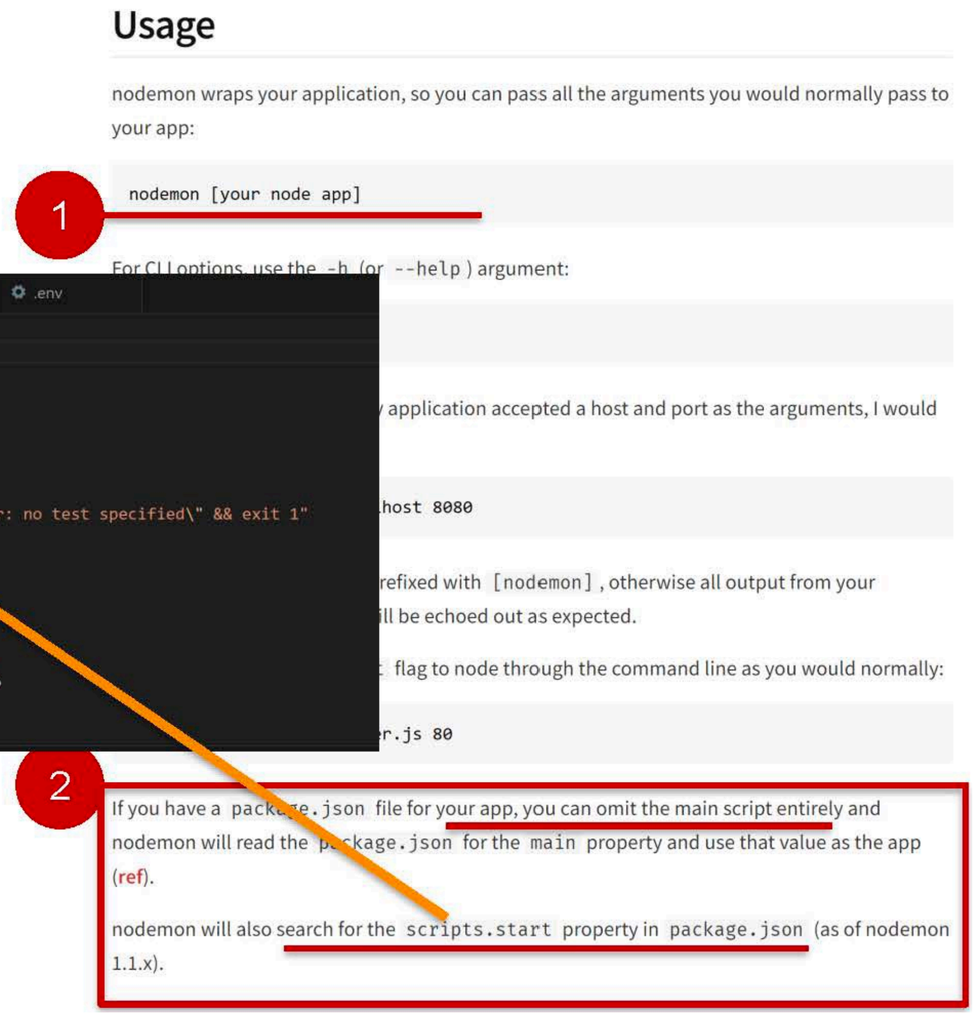
`nodemon` (node monitor) solves this issue

Install `nodemon` using

```
npm i -d nodemon
```




CHECK THE DEPENDENCY




*i.e. it automatically restarts your Node.js server whenever it detects changes in your project's files.*

# Express Server

## nodemon

Add the script to the `package.JSON` file in the format

```
6     "scripts": {
7
8       "start": "nodemon server.js"
9     },
```

To run your server

```
PS C:\Users\Surface\Downloads\GIU Work\winter Semester Preparation\Software Engineering\PERN Stack Development\server> npm start

> server@1.0.0 start
> nodemon server.js

[nodemon] 3.0.1
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,cjs,json
[nodemon] starting `node server.js`
server is running: listening on port 3010
```

# Routing

…is the process of **defining** and **mapping endpoints** to **specific functionality** within the **API**. It involves **specifying** the **HTTP method** (*i.e.* GET, POST, PUT, DELETE) and **URL path** of a request to determine the corresponding action that the API should take

…is **directing** incoming **requests** to the appropriate **resource** **or controller** that will **handle** the **request** and **generate** the appropriate **response**

https://roadmap.alexhyett.com/backend-developer-roadmap/

# Your first route (API)

Remember this?

```
CREATE TABLE table_name (
Column1 datatype (size),
Column2 datatype (size),
.....);
```

**CREATE**: creates a new table, view, index or other object in the DB

| Customers | | |
|---|---|---|
| PK | CustomerID | int |
| | FirstName | varchar(50) |
| | LastName | varchar(50) |
| | Street | varchar(50) |
| | City | varchar(50) |
| | ZipCode | varchar(50) |
| | Phone | varchar(11) |

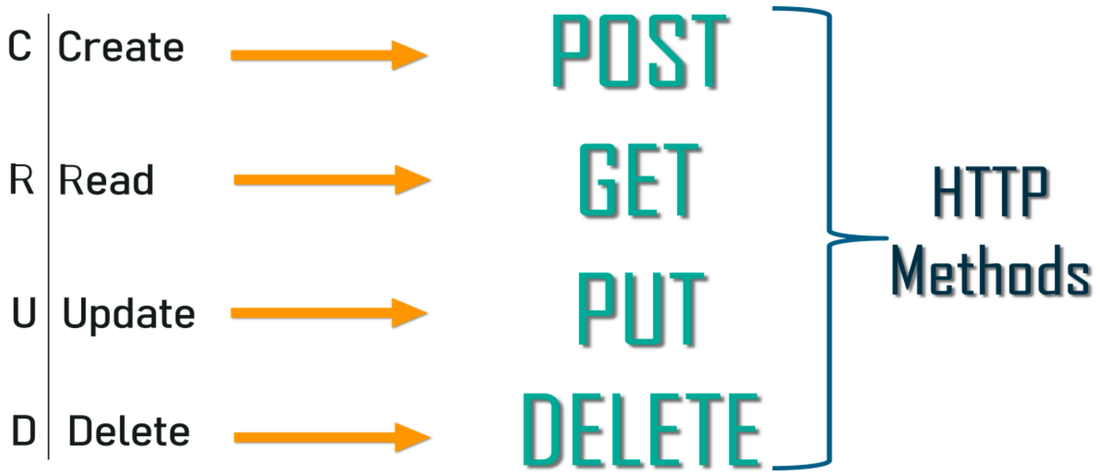
```
CREATE TABLE "Customers" (
  "CustomerID" INT,
  "FirstName" varchar(50),
  "LastName" varchar(50),
  "Street" varchar(50),
  "City" varchar(50),
  "ZipCode" varchar(50),
  "Phone" varchar(11),
PRIMARY KEY ("CustomerID")
);
```

**PRIMARY KEY**

1. UNIQUE
2. AUTO INCREMENT
3. NOT NULL

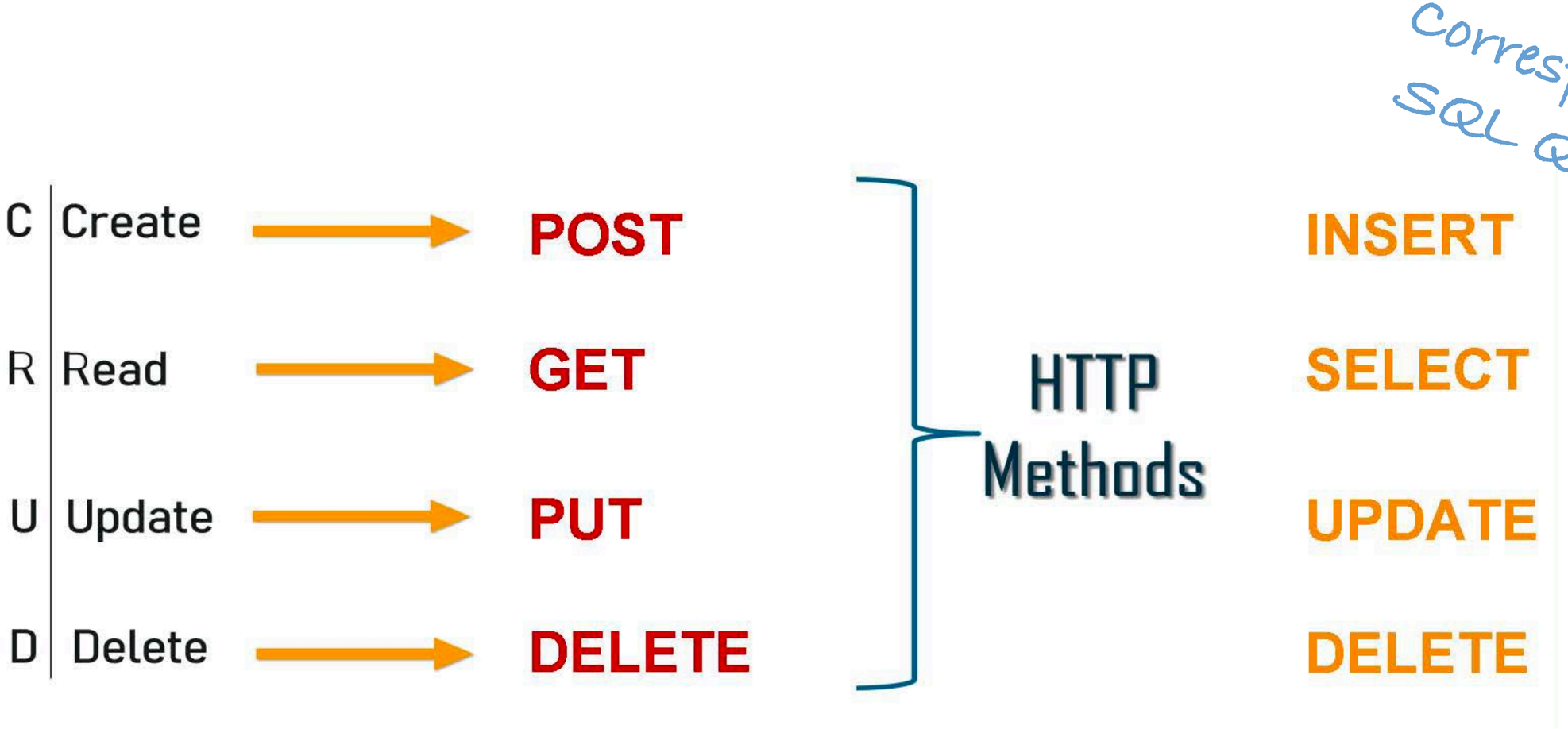
# Your first route (API)

```sql
INSERT INTO "Customers" ("CustomerID","FirstName", "LastName", "Street", "City", "ZipCode", "Phone")
VALUES
(1, 'John', 'Zaki', 'ABC', 'Mansoura', '35511', '01000100100'),
( 2, 'Ahmed', 'Sherif', 'DEF', 'Cairo', '51144', '01111101111'),
( 3,'Amir', 'Haitham', 'XYZ', 'Ismailia', '33311', '01222112222');
```




For every CRUD operation
on the `CUSTOMERS` table,
in the customer DB,
we need to create a route!

# Your first route (API)



# Your first route (API)

To define the route, reference the Express instance that was previously defined and stored within the app variable

```
//create an instance of the express app and store it in a variable called app
const app = express();
```

Define the HTTP method you want to use

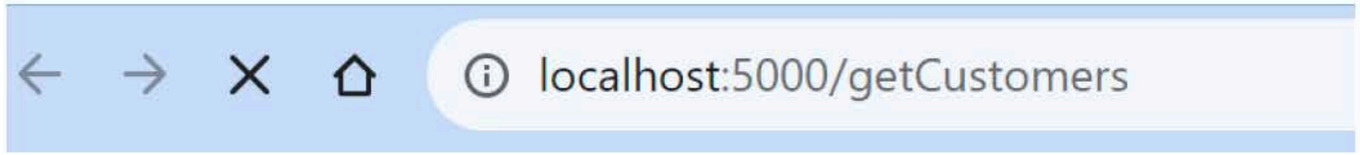
```
/*
- your frontend request is going to hit this route.
- your get method first parameter is the URL which is for
- your current development is for instance:
- http://localhost:5000/getCustomers
- in a production env. the base URL might be different (another IP address with a domain name)
- for instance http://mywebsite.com:PORT/ but your API URL is the same /getCustomers
*/
app.get("/getCustomers",(req, res)=>{
console.log("i\'m going to get all the customers...");

})
```

# Your first route (API)

## Testing it…

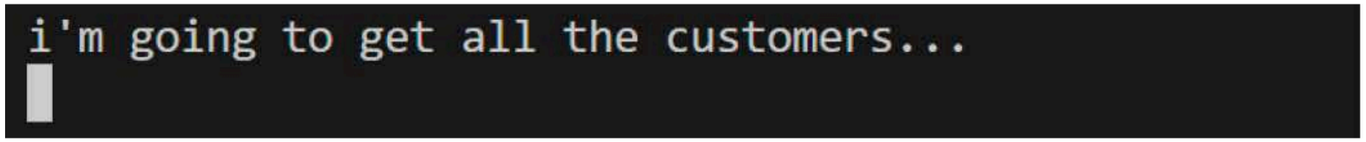
Go to the browser and type your full URL:

http://localhost:5000/getCustomers




Noting the port number you used and the API URL

If everything is correct you should receive the message you wrote in the callback function on the terminal




However…

# Your first route (API)

## Testing it…

Replace `console.log` with a response to be sent back to the customer

Refresh your page… and you will receive a response from server!

```
app.get("/getCustomers",(req, res)=>{
console.log("i\'m going to get all the customers...");

})
```

```
app.get("/getCustomers",(req, res)=>{
//console.log("i\'m going to get all the customers...");
res.send("Server received the request to get all customers...")
})
```

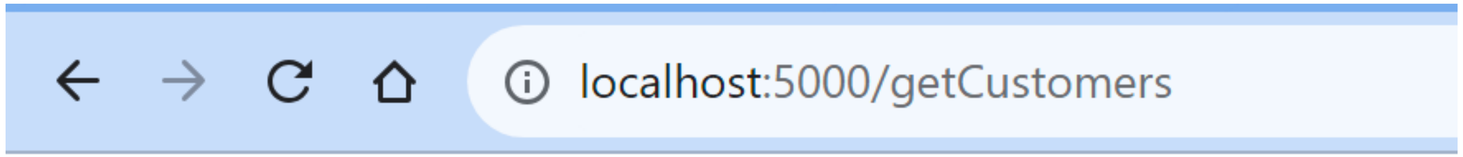
# Your first route (API)

## Testing it…

We will not be sending messages to the frontend, we need to send data → use the JSON format

Refresh your page…. You will receive the JSON format response

```
// res.send("Server received the request to get all customers...");
res.json({
    status:"success",
    firstName: "John",
    lastName: "Zaki"
});
```



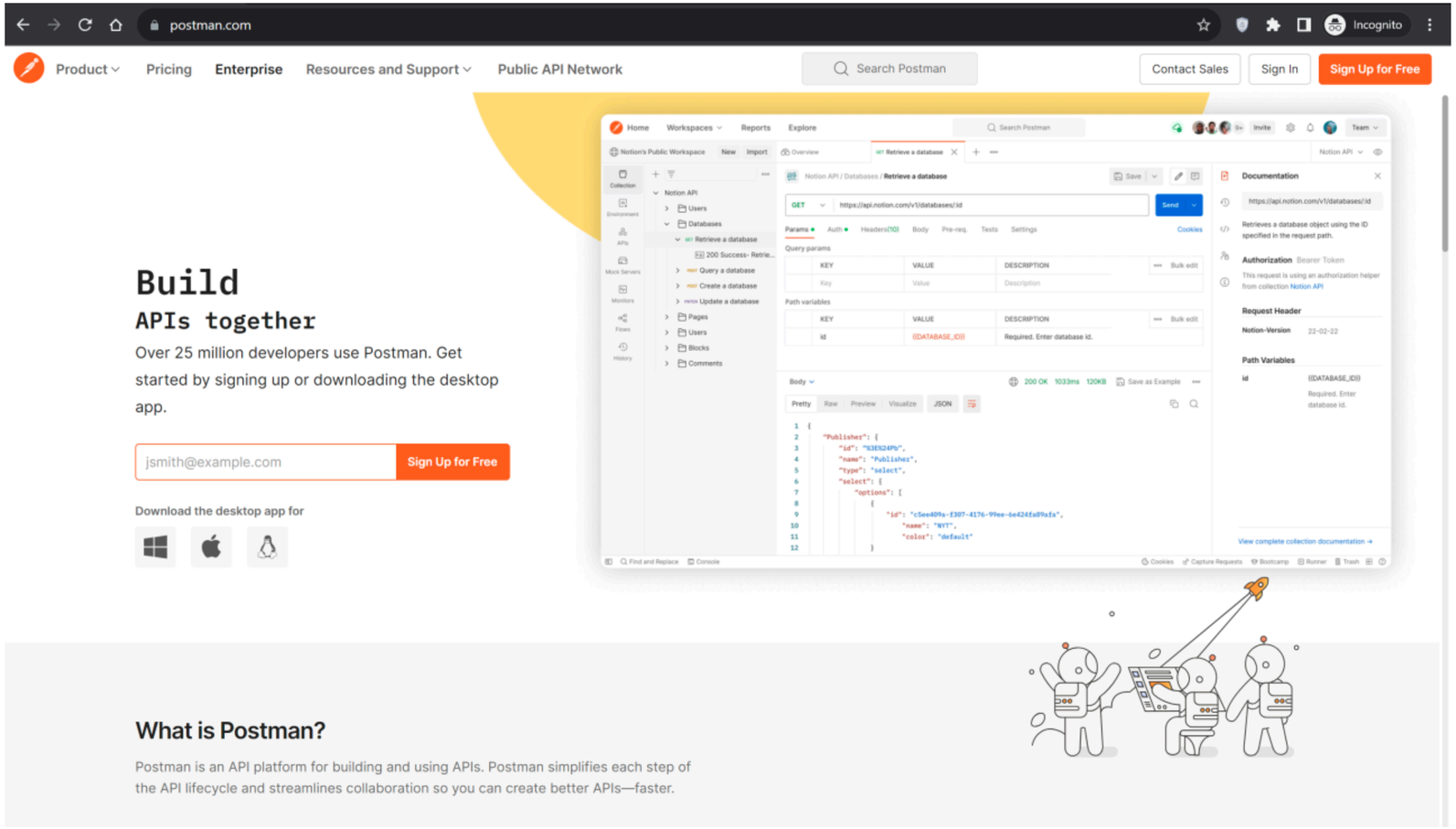

{"status":"success","firstName":"John","lastName":"Zaki"}

# Your first route (API)

## Postman collection

Testing the rest of the routes and HTTP methods is not going to be very practical from the browser, and our front-end is not there yet, so… use the **Postman collection**:
[ postman.com ]

*A tool for interacting with and managing back-end APIs*



1. Choose your HTTP method

RESTful API basics: CRUD, test & variable / **Get data**

Save

GET {{base_url}}/info?id=1

Send

GET
POST
PUT
PATCH
DELETE
HEAD
OPTIONS
Type a new method

Headers (5) Body Pre-request Script Tests Settings Cookies

| | Value | Description | Bulk Edit |
|---|---|---|---|
| | 1 | | |
| | Value | Description | |

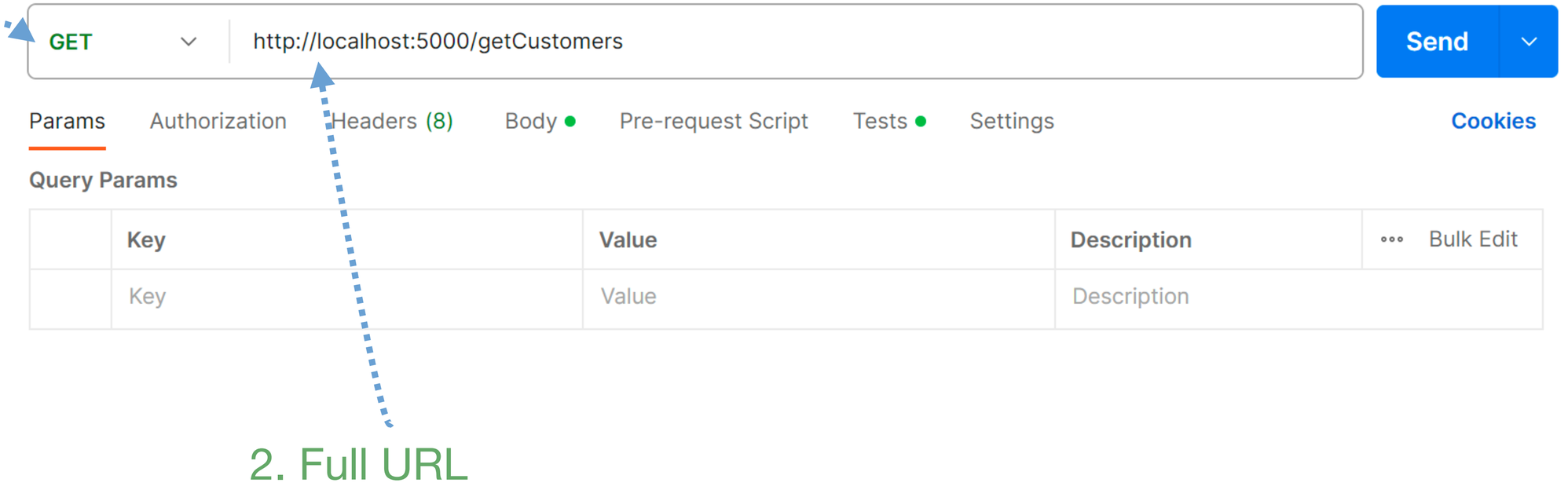
2. Your full URL: Base & Route

3. Press Send!

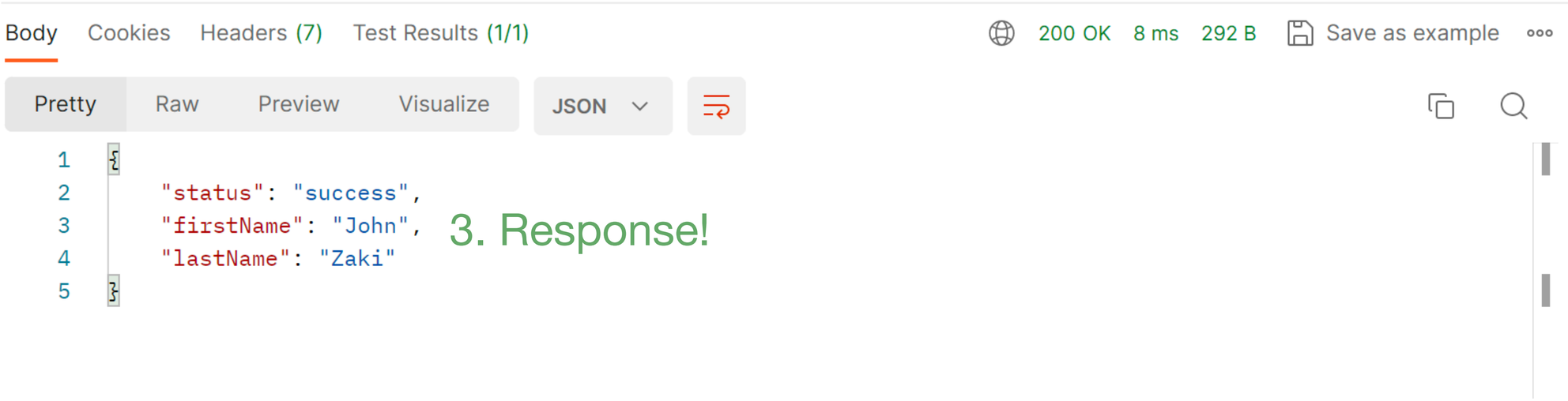
**Response**

Click Send to get a response

1. Testing GET




2. Full URL




3. Response!

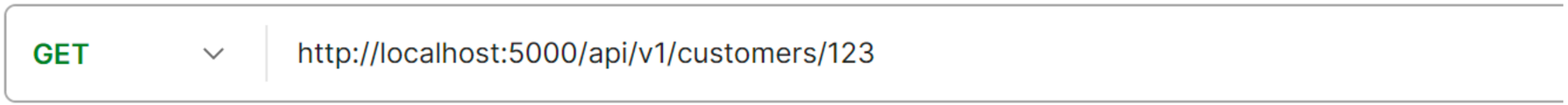
# Better naming convention for API

| CRUD | HTTP METHOD | URL |
|---|---|---|
| **READ** ALL CUSTOMERS | GET | /api/v1/customers |
| **READ** A CUSTOMER | GET | /api/v1/customers/:id |
| **CREATE** A CUSTOMER | POST | /api/v1/customers |
| **UPDATE** A CUSTOMER | PUT | /api/v1/customers/:id |
| **DELETE** A CUSTOMER | DELETE | /api/v1/customers/:id |

# Get one customer

Log the request object to check if the right ID was passed from the postman to the server

```
// get one customer
app.get("/api/v1/customers/:id",(req, res)=>{
// let's log the response to find the ID passed from the postman to the server
console.log(req);

});
```

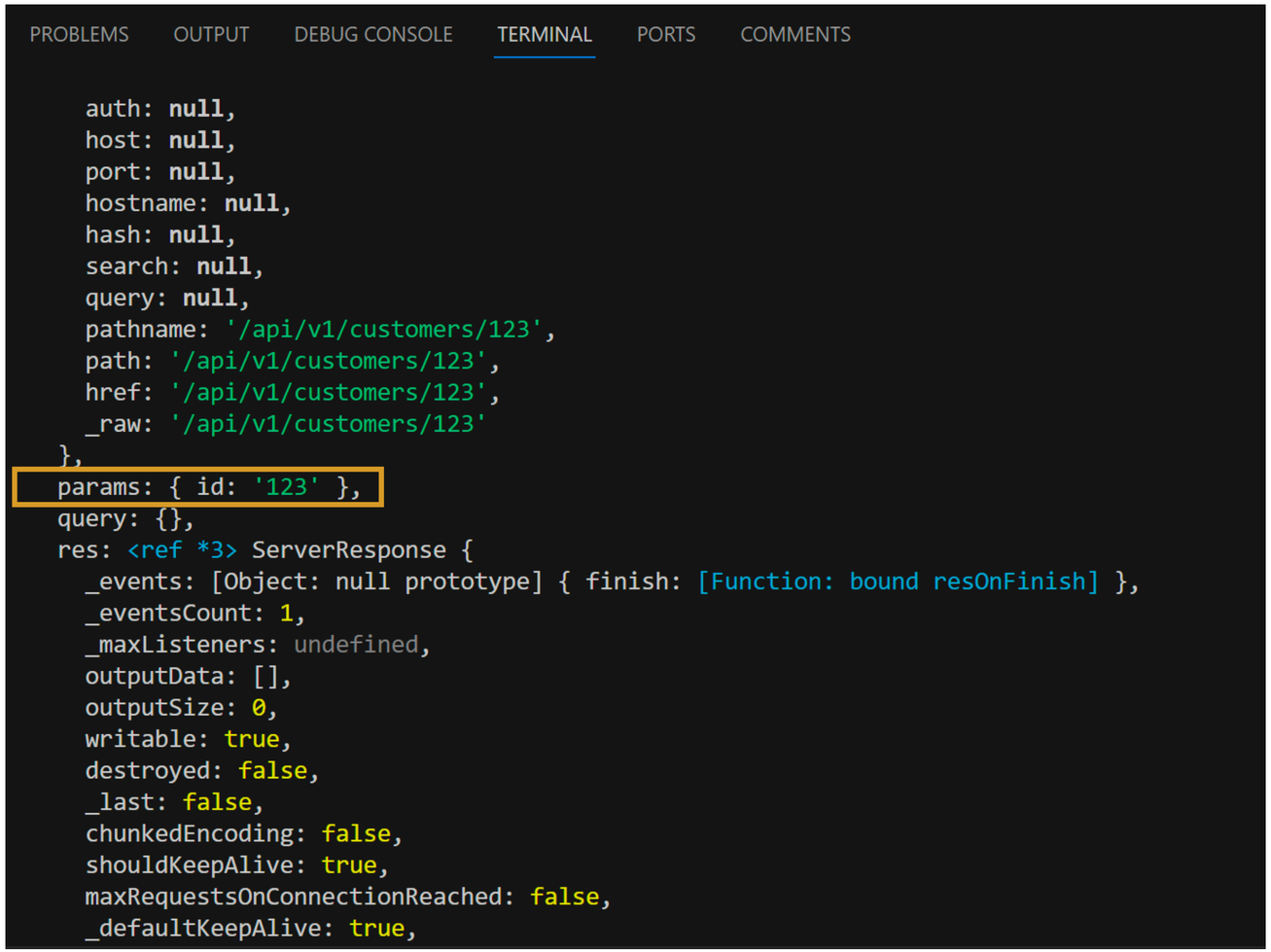


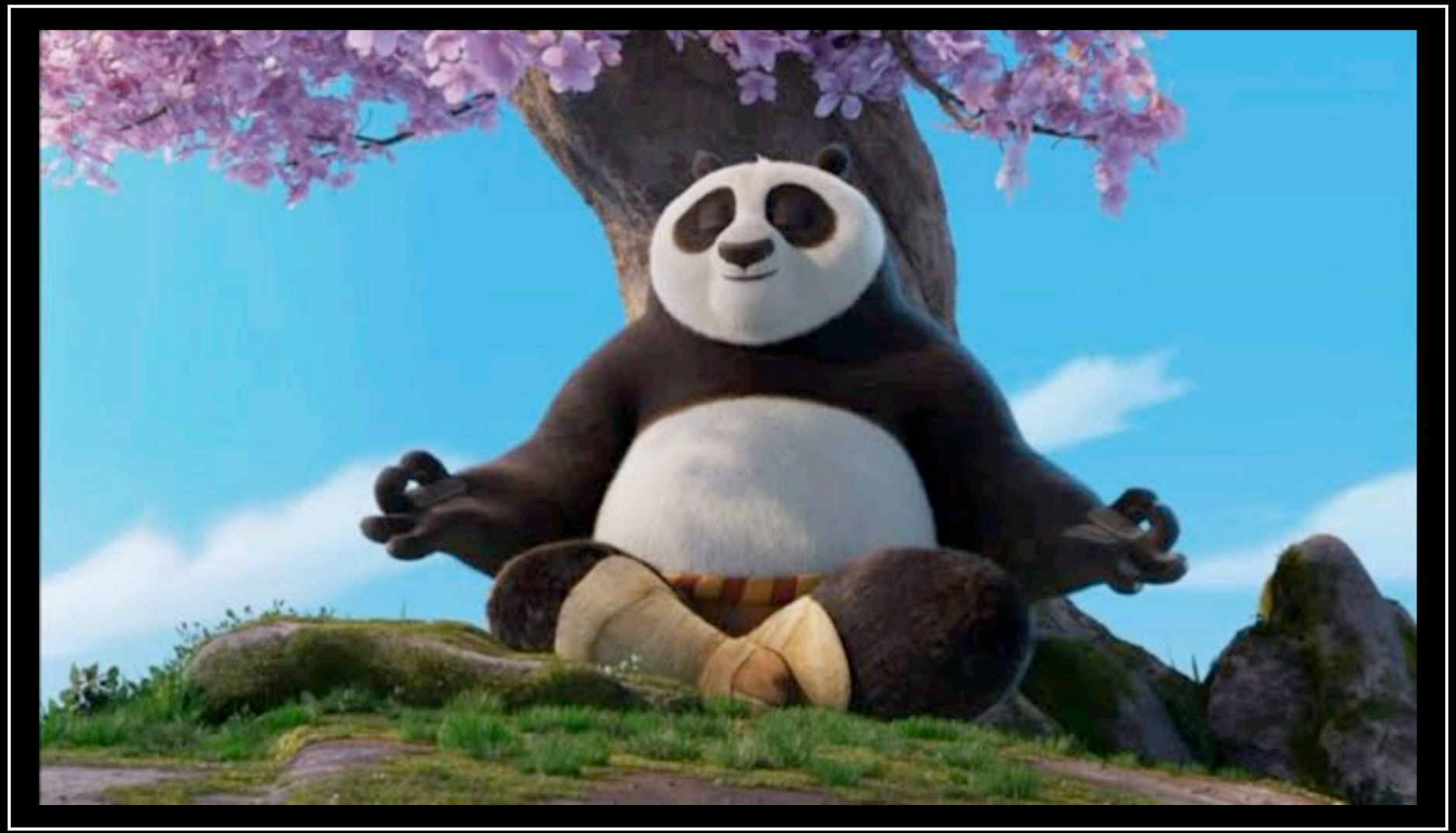
# Get one customer

GET http://localhost:5000/api/v1/customers/123

Log the request object to check if the right ID was passed from the postman to the server…

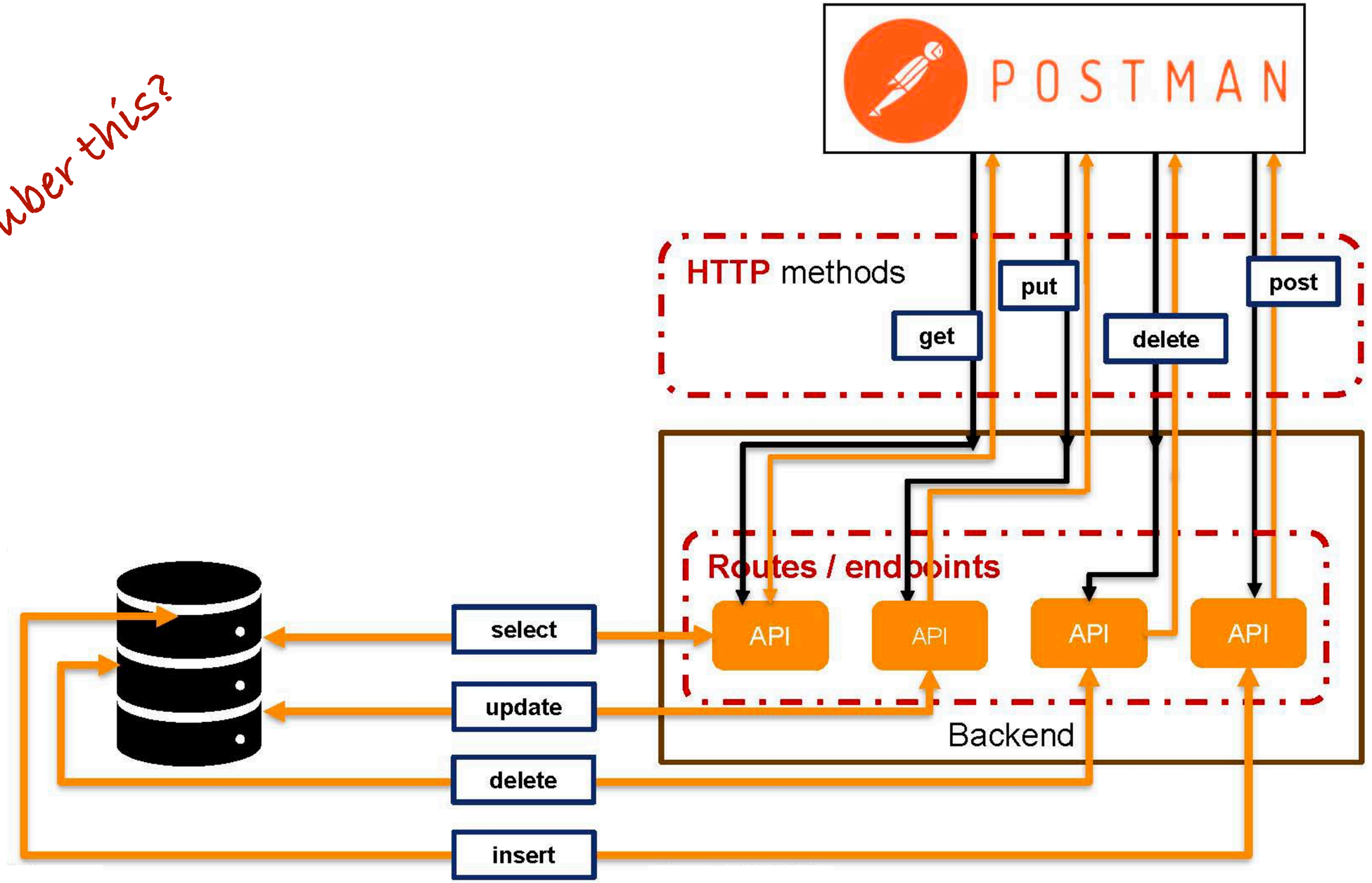
Search in the request printed in the terminal



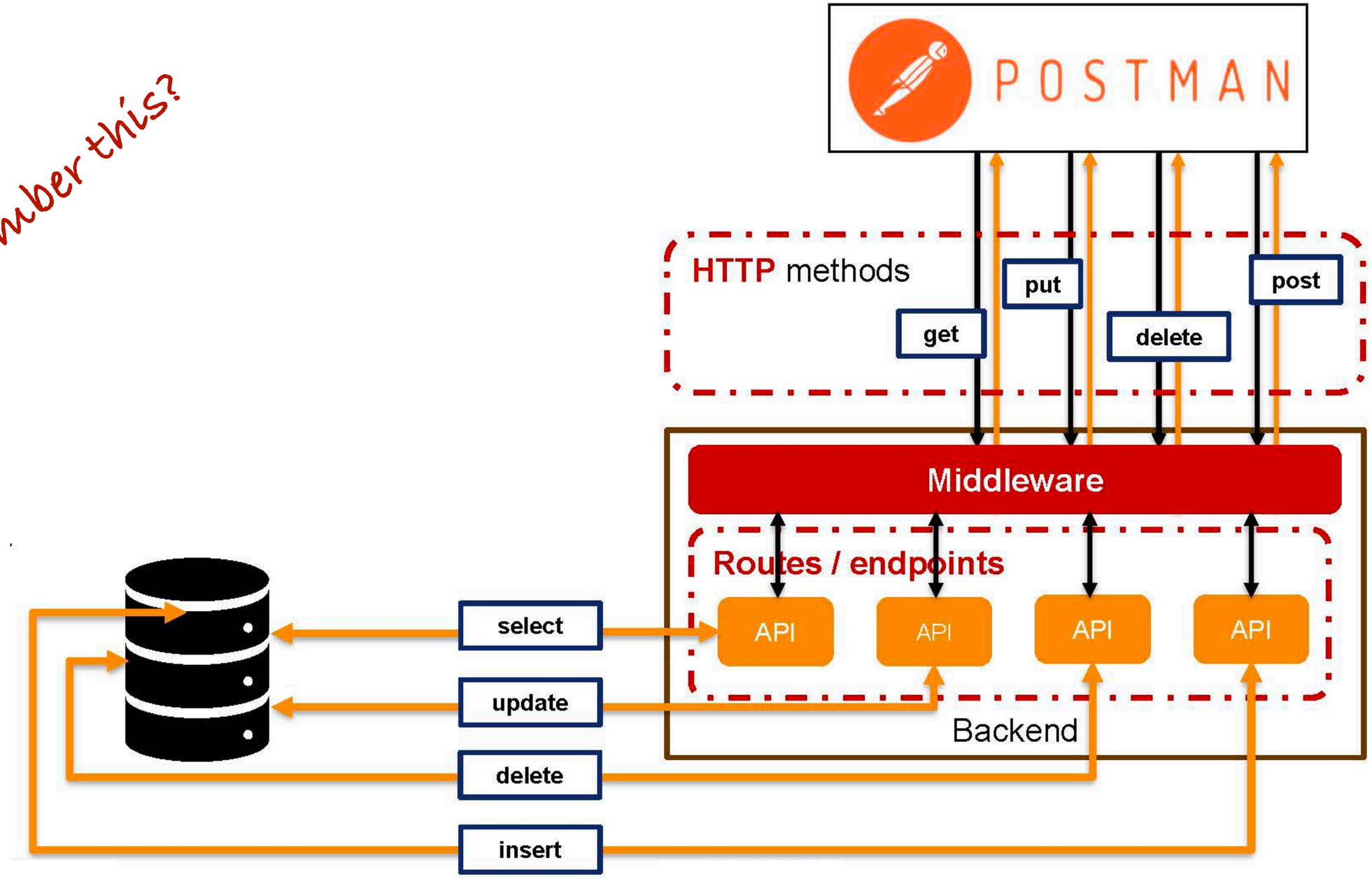


# INNER PEACE

Remember this?



Remember this?



# Middleware

Just another layer of API…

Use your middleware or a third-party middleware…

Define your middleware as early as possible **before** all your **routes**!!

```
// add the middleware
app.use((req, res, next )=>{
    console.log("the middleware received the request");
    next();  // middleware passes the request to the next route handler (API)
});
```

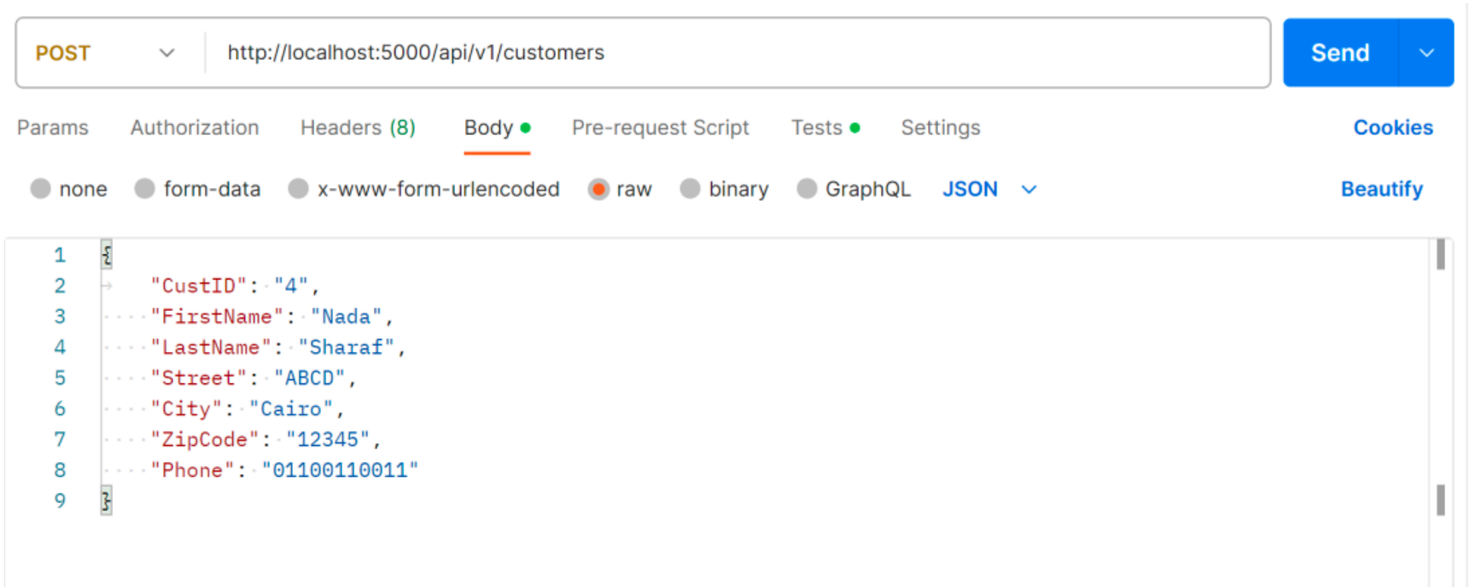
# Express middleware

Add the Express middleware

Use the `express.json` property to get the body of the request to be sent to the API.

…attached as a property to the request…

```
// ADD EXPRESS MIDDLEWARE

app.use(express.json());
```



# Express middleware

## POST

Back to our original problem of creating an API for the POST, PUT and DELETE messages…

Result of the body sent by the Postman appears in the teminal, stored as a JS object, meaning, we can send it to the DB once the connection is there…

```
// create a new customer .... data is sent from the client to the server
// use postman body to send the data - body - raw - JSON
app.post("/api/v1/customers",(req, res)=>{
    console.log(req.body);

})
```

```
{
  CustID: '4',
  FirstName: 'Nada',
  LastName: 'Sharaf',
  Street: 'ABCD',
  City: 'Cairo',
  ZipCode: '12345',
  Phone: '01100110011'
}
```

# Express middleware

## PUT, DELETE

We are just building the skeleton for the routes and ensuring we are receiving the correct data from  the Postman…

Once connected to the DB, we will no longer need `console.log`

```
// update an existing customer
app.put("/api/v1/customers/:id",(req, res)=>{
    console.log(req.params);  // check that we are passing the customer ID we would like to update
    console.log(req.body); // get the details of the body from the request
})

// delete a customer
app.delete("/api/v1/customers/:id",(req, res)=>{
    console.log(req.params);
    console.log(req.body);
})
```

# Your first route (API)

```
CREATE TABLE table_name (
Column1 datatype (size),
Column2 datatype (size),
.....);
```

**CREATE**: creates a new table, view, index or other object in the DB

| Customers | | |
|---|---|---|
| PK | CustomerID | int |
| | FirstName | varchar(50) |
| | LastName | varchar(50) |
| | Street | varchar(50) |
| | City | varchar(50) |
| | ZipCode | varchar(50) |
| | Phone | varchar(11) |

```
CREATE TABLE "Customers" (
  "CustomerID" INT,
  "FirstName" varchar(50),
  "LastName" varchar(50),
  "Street" varchar(50),
  "City" varchar(50),
  "ZipCode" varchar(50),
  "Phone" varchar(11),
PRIMARY KEY ("CustomerID")
);
```

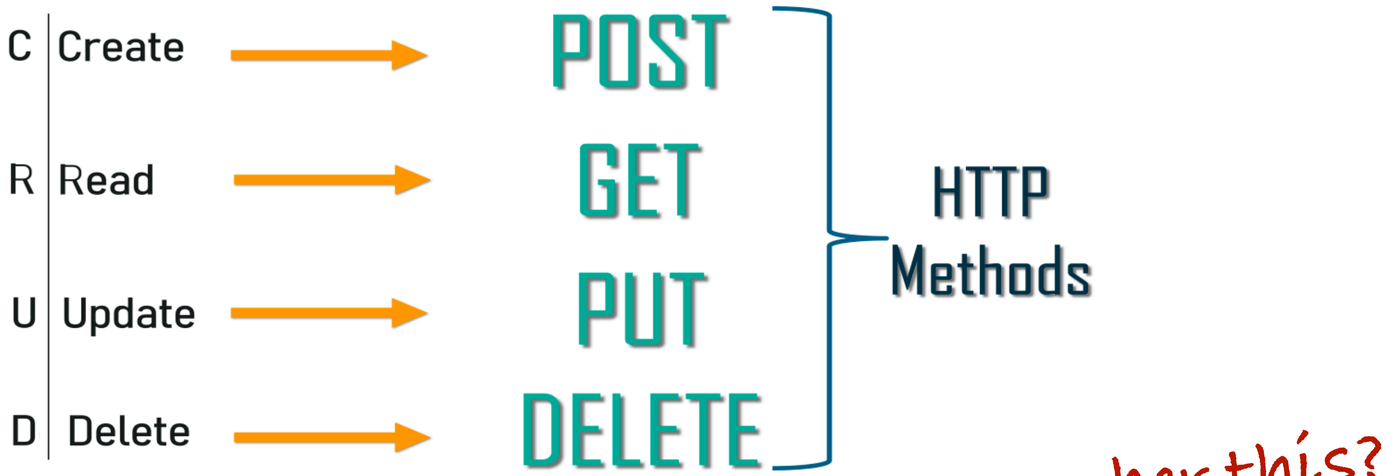
Remember this?

**PRIMARY KEY**

1. UNIQUE
2. AUTO INCREMENT
3. NOT NULL

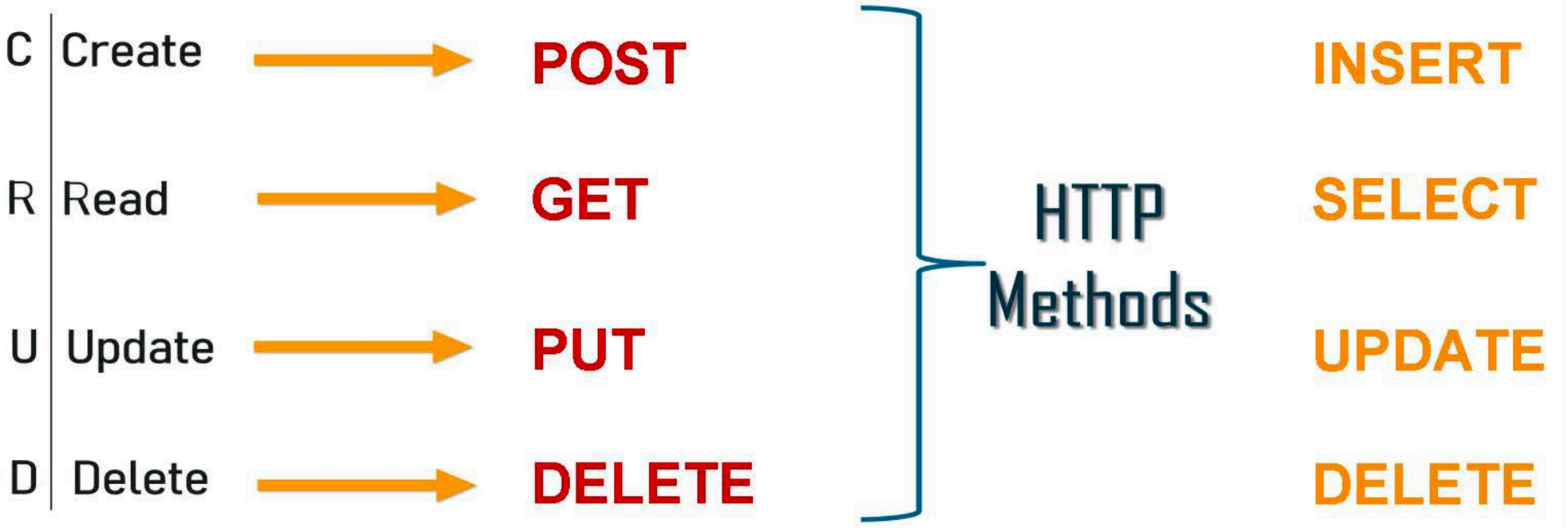
# Your first route (API)

```sql
INSERT INTO "Customers" ("CustomerID","FirstName", "LastName", "Street", "City", "ZipCode", "Phone")
VALUES
(1, 'John', 'Zaki', 'ABC', 'Mansoura', '35511', '01000100100'),
( 2, 'Ahmed', 'Sherif', 'DEF', 'Cairo', '51144', '01111101111'),
( 3,'Amir', 'Haitham', 'XYZ', 'Ismailia', '33311', '01222112222');
```



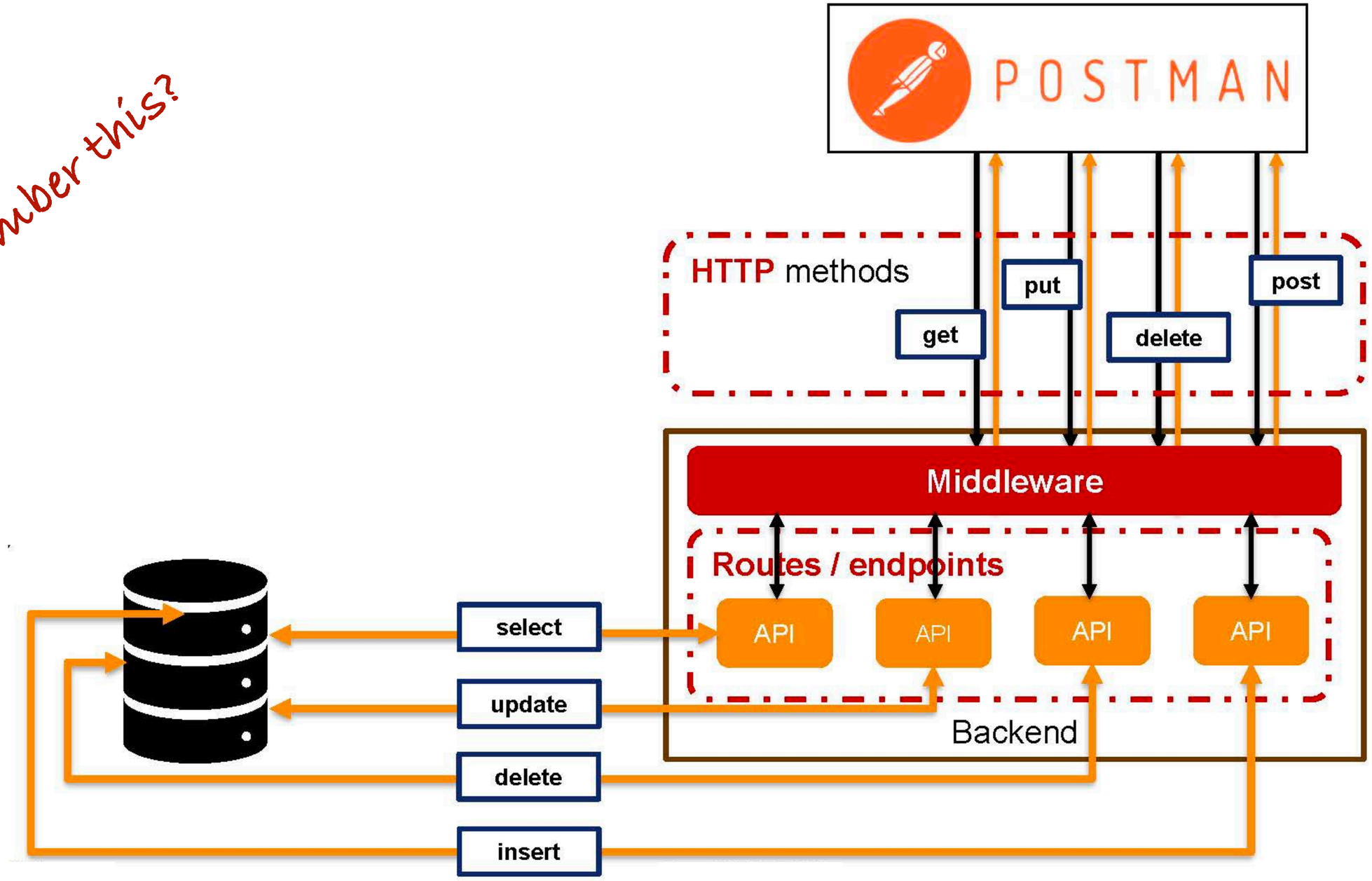

For every CRUD operation
on the `CUSTOMERS` table,
in the customer DB,
we need to create a route!

# Your first route (API)




Remember this?

Remember this?



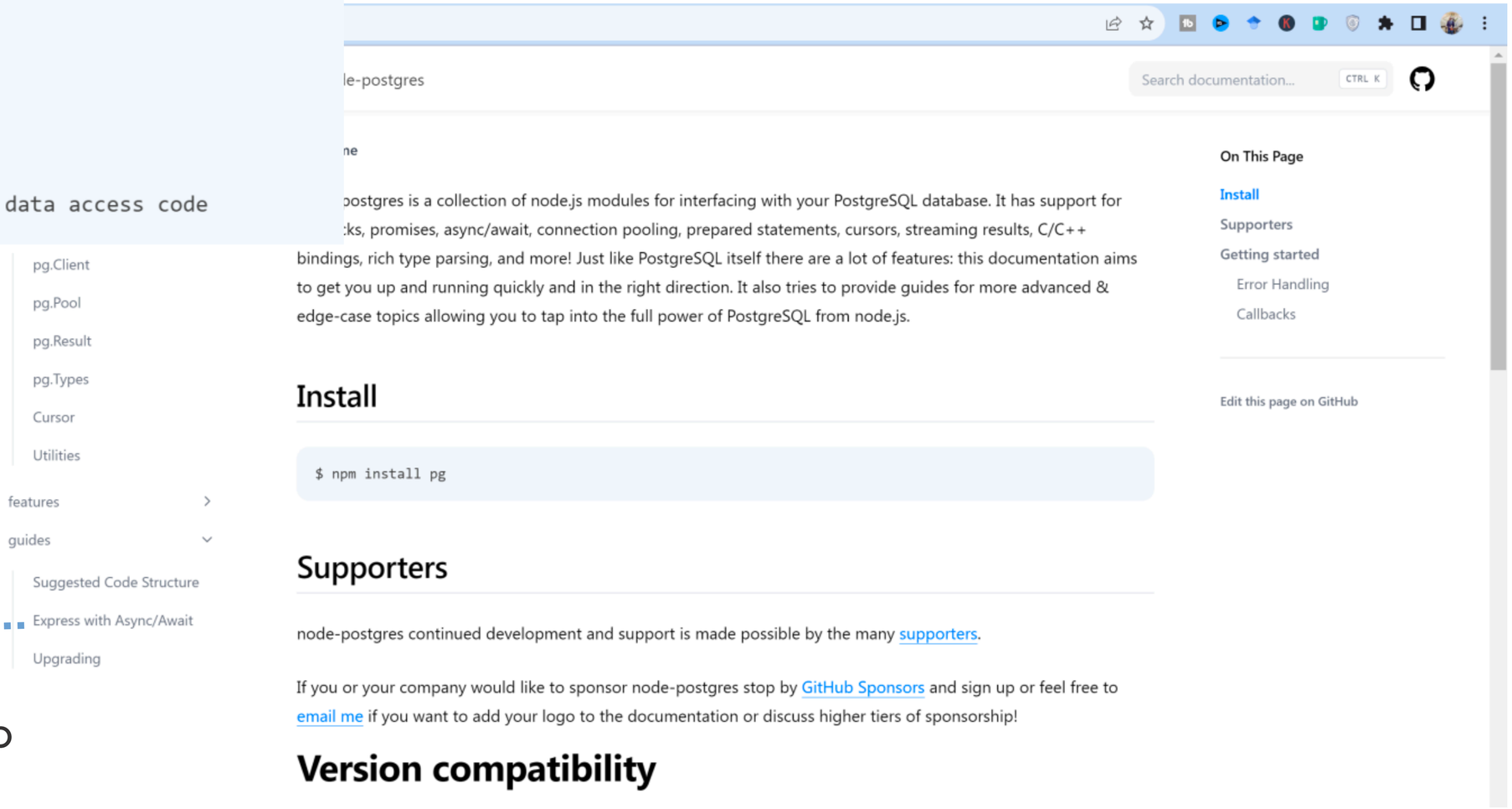
# Connecting to the DB

```
- app.js
- index.js
- routes/
  - index.js
  - photos.js
  - user.js
- db/
  - index.js <--- this is where I put data access code
```

Install the library using: $ *npm* install *pg* (a non-blocking PostgreSQL client for Node.js)

Create a folder called `db`

Create `index.js`



# Connecting to the DB

Importing `db/index.js`

Server > db > JS index.js > ...

```
1
2   const {Pool} = require('pg');
3
4   const pool = new Pool(); // creating a new pool
5
6   module.exports = {
7       query: (text, param) => pool.query(text, param)
8   }
```

Connecting to the DB using environment variables

Stored in `.env` file

```
3   PGUSER=postgres
4   PGHOST=localhost
5   PGPASSWORD=admin
6   PGDATABASE=Customers
7   PGPORT=5432
```

# Query the DB

## Return all customers

```
app.get("/api/v1/Customers", async (req, res)=>{

    // attempts to make a query to db is recommended to be captured in a try-catch block
    try{
        const sqlResult = await db.query("SELECT * FROM \"Customers\"");  // this returns a promise - so use async await

        /* you will see in the console that the results coming from the DB appear in a property called rows.
        so you send sqlResults.rows to the frontend */
        console.log(sqlResult);

        res.status(200).json({
                status:"success",
                results: sqlResult.rows.length,
                data: {
                 customers: sqlResult.rows
                  },
     });
    } catch(err){

        console.log(err);

    }

});
```

# What to do next…

Complete to other queries

Google for “Parameterised Queries”

Google Public vs Private APIs

😎 https://roadmap.alexhyett.com/backend-developer-roadmap/

# Back-end development

## Terminology

### Back-end/server-side development of a web application

**Test API Endpoint:** used during development and testing phases to validate the behavior and functionality of the API before it is deployed in a production environment

**API Call:** a request made to an API endpoint to retrieve data, perform an action, or execute a specific operation

**REST** (Representational State Transfer) **API Endpoint:** uses a set of architectural principles that define the structure of the endpoints: CRUD (Create, Read, Update, Delete) operations. REST APIs can serve data in a variety of **formats** to better suit different client needs:

> **JSON** (JavaScript Object Notation)**:** By far the most popular choice for REST APIs, JSON is lightweight, easy to read, and works seamlessly with JavaScript and most modern programming languages.
>
> **XML** (eXtensible Markup Language)**:** While not as ubiquitous as it once was, XML is still supported by many APIs, especially those with legacy integrations.
>
> …also **CSV**, **YAML**, **RSS/ATOM**…

**API Endpoint:** APIs can have multiple endpoints, each serving a specific purpose or resource. These endpoints are defined by their URLs and are used to access different parts of the API

**API Endpoint Structure:** The structure of an API endpoint is typically defined by its URI path, HTTP method (GET, POST, PUT, DELETE), query parameters, headers, and payload (if any). It determines how clients interact with the API

**API Security:** Security measures such as authentication, authorization, HTTPS encryption, and rate limiting are implemented to protect API endpoints from unauthorized access and attacks

https://qodex.ai/blog/what-is-an-api-endpoint