

EXPRESS.JS

Routes & Rest

BUT FIRST...

POP QUIZ



CLIENT

Something that makes (HTTP) requests



SERVER

Something that responds to (HTTP) requests



REQUEST

*A formatted message sent over the network by a client.
Contains VERB, URI (route), headers, and body.*



RESPONSE

*A server's reply to a request (formatted message).
Contains headers, payload, and status.*



REQUEST-RESPONSE CYCLE

*The client **always** initiates by sending a request, and the server completes it by sending **exactly one** response*



EXPRESS MIDDLEWARE

A function that receives the request and response objects of an HTTP request/response cycle.



EXPRESS MIDDLEWARE

A function that receives the request and response objects of an HTTP request/response cycle.

```
function(req, res, next){...}
```

EXPRESS MIDDLEWARE CAN...

- Execute any code (such as logging) **then** move to the **next** middleware function in the chain
- Modify the request and the response objects **then** pass them to the **next** middleware function in the chain
- End the request-response cycle (E.g. `res.send()`)

EXPRESS ROUTER

EXPRESS ROUTER

- Express provides a Router middleware to create modular, mountable route handlers.
- Think of it as a “mini-app” that nest within an exiting app.
- It let you break up the major parts of your application into separate modules.

A Router instance is a complete middleware and routing system; for this reason, it is often referred to as a “mini-app”.

```
App.js

const express = require("express");
const morgan = require("morgan");
const client = require("../db");
const postList = require("../views/postList");
const postDetails = require("../views/postDetails");

const app = express();

app.use(morgan("dev"));
app.use(express.static(__dirname + "/public"));

app.get("/", async (req, res) => {
  const data = await client.query("SELECT...");
  res.send(postList(data.rows));
});

app.get("/posts/:id", async (req, res) => {
  const data = await client.query("SELECT ...");
  const post = data.rows[0];
  res.send(postDetails(post));
});

const PORT = 1337;

app.listen(PORT, () => {
  console.log(`App listening in port ${PORT}`);
});
```

This is similar to the pair exercise you did before. The code has been simplified to fit on screen, but in general this represents a working express application with two routes.

You can imagine that as your application grows, you will need more and more routes. Having all of them on the same file is a recipe for unmaintainable code.

App.js

```
const express = require("express");
const morgan = require("morgan");
const postList = require("../views/postList");
const postDetails = require("../views/postDetails");
const routes = require("../routes");

const app = express();

app.use(morgan("dev"));
app.use(express.static(__dirname + "/public"));
app.use(routes);

const PORT = 1337;

app.listen(PORT, () => {
  console.log(`App listening in port ${PORT}`);
});
```

routes.js

```
const express = require('express');
const router = express.Router();
const client = require("../db");

app.get("/", async (req, res) => {
  const data = await client.query("SELECT...");
  res.send(postList(data.rows));
});

app.get("/posts/:id", async (req, res) => {
  const data = await client.query("SELECT ...");
  const post = data.rows[0];
  res.send(postDetails(post));
});

module.exports = router;
```

App.js

```
const express = require("express");
const morgan = require("morgan");
const postList = require("../views/postList");
const postDetails = require("../views/postDetails");
const routes = require("../routes");

const app = express();

app.use(morgan("dev"));
app.use(express.static(__dirname + "/public"));
app.use(routes);

const PORT = 1337;

app.listen(PORT, () => {
  console.log(`App listening in port ${PORT}`);
});
```

routes.js

```
const express = require('express');
const router = express.Router();
const client = require("../db");

router.get("/", async (req, res) => {
  const data = await client.query("SELECT...");
  res.send(postList(data.rows));
});

router.get("/posts/:id", async (req, res) => {
  const data = await client.query("SELECT ...");
  const post = data.rows[0];
  res.send(postDetails(post));
});

module.exports = router;
```

... Of course, you don't have an "app" variable in your routes file. Instead, we will use the router instance.

We can also split the application's routes in more than one file, which begs the question: How to organize our routes?

REST

REST stands for Representational State Transfer. (It is sometimes spelled "ReST".) It relies on a stateless, client-server, cacheable communications protocol -- and in virtually all cases, the HTTP protocol is used. REST is an architecture style for designing networked applications

REST

- **Architecture style for designing backend applications.**
- **Helps answer the question on how to organize routes and how to map functionality to URIs and Methods:**
 - Paths represent "nouns" or *resources*
 - HTTP methods maps to data operations

Users & Posts are examples of resources.

paths represent "nouns" or resources, and methods represent actions to apply to those resources. For example:



REST - RESOURCES

GET /api/users
GET /users
GET /api/users?name="Rubeus"



REST - RESOURCES

GET	/users	Show all users
GET	/users/4	Show a single user (whose ID=4 in the db)
POST	/users	Create a new user in the DB
PUT	/users/4	Update user 4 in the db
DELETE	/users/4	Delete user 4 from the db

Our express routes tend to mimic the REST structure, having one route file per resource.

App.js

```
const express = require("express");
const app = express();
app.use(morgan("dev"));
app.use(express.static(__dirname + "/public"));

app.use('/posts', require('./routes/posts'));
app.use('/users', require('./routes/users'));

const PORT = 1337;

app.listen(PORT, () => {
  console.log(`App listening in port ${PORT}`);
});
```

posts.js

users.js

```
const express = require('express');
const router = express.Router();
const client = require("../db");

router.get("/", async (req, res) => {
  const data = await client.query("SELECT...");
  res.send(postList(data.rows));
});

router.get("/:id", async (req, res) => {
  const data = await client.query("SELECT ...");
  const post = data.rows[0];
  res.send(postDetails(post));
});

module.exports = router;
```

REQUEST BODY & BODY-PARSER

- **POST, PUT (and the less used PATCH) HTTP requests can contain information in the body**
- **The request body is streamed and frequently compressed**
- **Body-parser is an official Express middleware to automatically parse incoming request bodies and make the data available under `req.body`**

The request body is streamed and frequently compressed - In other words: It's not trivial to get the body information out of a POST/PUT/PATCH request in your express application.



BODY-PARSER

verb route

```
POST /books HTTP/1.1
Host: www.test101.com
Accept: */*
```

headers

In express...
request.body = {bookId:12345, author: 'Nimit'}

```
bookId=12345&author=Nimit
```

body



BODY-PARSER

```
npm install body-parser
```

```
const bodyParser = require('body-parser');  
app.use(bodyParser.urlencoded({ extended: false }));
```

 FULLSTACK

24

EXPRESS • ROUTES & REST

The bodyParser object exposes various factories to create middlewares.
The most common are urlEncoded (for data sent through forms) and json (which we will use later).