

Electronic Business Development — Lab 4 — Building a REST API

Contacts

Course Instructor

Dr. Nourhan Hamdi

nourhan.hamdi@giu-uni.de
M2.312

Teaching Assistant

Mr. Nour Gaser

nour.aly@giu-uni.de
M2.419

Teaching Assistant

Mr. Omar Alaa

omar.alaaeldin@giu-uni.de
M2.419

Teaching Assistant

Ms. Mariam Khalid

mariam.khalid@giu-uni.de
M2.414

Teaching Assistant

Mr. Younes Zied

younes.zied@giu-uni.de
M2.506

Teaching Assistant

Mr. Mahmoud Hani

TBD
M2.506



Lab agenda

Duration Topic

10 min	Recap: where we left ebd bn'b
--------	-------------------------------

25 min	Routes, controllers & models
--------	------------------------------

10 min	Testing with an API client
--------	----------------------------

15 min	Middleware & the request body
--------	-------------------------------

25 min	Create, update & delete
--------	-------------------------

5 min	What happens on restart?
-------	--------------------------



Where we left ebd bn'b

Last lab you set up an npm project, started an Express server, and served listings with two `GET` routes.

Method	URL	Status
GET	/api/listings	Done
GET	/api/listings/:id	Done
POST	/api/listings	Today
PUT	/api/listings/:id	Today
DELETE	/api/listings/:id	Today

Today we finish the API and give the code a shape that lasts the whole semester.



Didn't finish Lab 3?

Start from here and you are exactly where Lab 3 ended:

1. Create a folder called `ebd-bnb` and open it in [VS Code](#).
2. Create `package.json` and paste in the code on the right.
3. In the terminal, run `npm install`.
4. Create `.gitignore` containing `node_modules/`.
5. Create `server.js` and paste in the code from the **next slide**.
6. Run `npm run dev` and open `http://localhost:3000/api/listings`. You should see three listings.
7. Commit and push it to a new [GitHub](#) repo.

package.json

```
{
  "name": "ebd-bnb",
  "version": "1.0.0",
  "type": "module",
  "main": "server.js",
  "scripts": {
    "start": "node server.js",
    "dev": "node --watch server.js"
  },
  "dependencies": {
    "express": "^5.1.0"
  }
}
```

Want to know what each step does? Lab 3, from "npm init" onwards, walks through it.

Lab 3's server.js

server.js

```
import express from "express";

const app = express();

// Our fake database for now
const listings = [
  { id: 1, title: "Nile View Apartment", pricePerNight: 85 },
  { id: 2, title: "Beach House", pricePerNight: 140 },
  { id: 3, title: "Desert Eco-Lodge", pricePerNight: 60 },
];

app.get("/", (req, res) => {
  res.send("Welcome to ebd bn'b");
});
```

Copy each half into `server.js`, left then right. Indentation lost in the paste? **Shift+Alt+F** in VS Code puts it back.

server.js (continued)

```
app.get("/api/listings", (req, res) => {
  res.json(listings);
});

app.get("/api/listings/:id", (req, res) => {
  const id = Number(req.params.id);
  const listing = listings.find((listing) => listing.id === id);

  if (!listing) {
    return res.status(404).json({ error: "Listing not found" });
  }
  res.json(listing);
});

app.listen(3000, () => {
  console.log("Server running on http://localhost:3000");
});
```

Today's question

```
// ebd bn'b: the listings API
// We build this in Lab 4. Try it before then!

// 1. Import express and create an app

// ===== MIDDLEWARE =====
// 2. Read JSON request bodies (express.json)

// ===== DATABASE =====
// 3. A "listings" array of rental properties,
//     each with an id, title and pricePerNight

// ===== MODEL =====
// 4. Functions that are the ONLY code
//     allowed to touch the array:
//     getAllListings()
//     getListingById(id)
//     addListing(data)
//     updateListing(id, data)
//     removeListing(id)
```

```
// ===== CONTROLLERS =====
// 5. Read the request, call the model,
//     send the response:
//     handleGetAll -> 200 + all listings
//     handleGetOne -> 200 + listing, or 404
//     handleCreate -> 201 + new listing, or 400
//     handleUpdate -> 200 + listing, or 400/404
//     handleDelete -> 204 (no body), or 404

// ===== ROUTES =====
// 6. Connect each method + URL to a controller
//     GET    "/api/listings"
//     GET    "/api/listings/:id"
//     POST   "/api/listings"
//     PUT    "/api/listings/:id"
//     DELETE "/api/listings/:id"

// ===== START =====
// 7. Start the server on port 3000
```

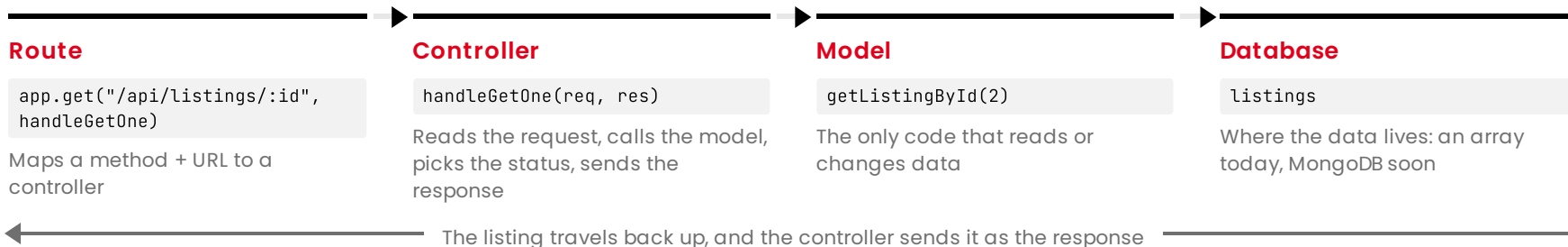


Routes, controllers & models

Who does what, when a request arrives



Four layers, one job each



- A request travels **down** the layers; the answer comes back **up**.
- Each layer only talks to its neighbours. A route never touches the array; a model never sees `req` or `res`.
- **Why bother?** When MongoDB replaces the array, only the **model** and **database** change. Routes and controllers stay as they are.

One file today, many files later

Today everything stays in `server.js`, split into sections by banners:

```
// ===== MODEL =====
```

Read the file top to bottom and you meet the layers in order.

As ebd bn'b grows, each section moves into its own file. Nothing inside it changes; it only moves.

```
ebd-bnb/  
├─ server.js      MIDDLEWARE, ROUTES, START  
├─ db.js          DATABASE  
├─ models/  
│   └─ Listing.js  MODEL  
├─ controllers/  
│   └─ listingController.js  
└─ routes/  
    └─ listingRoutes.js
```

This is the usual Express layout; we get there in a few labs.

The model

```
function getAllListings() {
  return listings;
}

function getListById(id) {
  return listings.find((listing) => listing.id === id);
}

function addListing(data) {
  const listing = {
    id: nextId++,
    title: data.title,
    pricePerNight: data.pricePerNight,
  };
  listings.push(listing);
  return listing;
}
```

```
function updateListing(id, data) {
  const listing = getListById(id);
  if (listing) {
    listing.title = data.title;
    listing.pricePerNight = data.pricePerNight;
  }
  return listing;
}

function removeListing(id) {
  const listing = getListById(id);
  listings = listings.filter((item) => item.id !== id);
  return listing;
}
```

- Plain functions, and the **only** code that touches `listings`.
- They know nothing about requests or status codes; they just find and change data.
- In the MongoDB lab, each one becomes a single database call.

Controllers and routes

ROUTES

```
app.get("/", (req, res) => {
  res.send("Welcome to ebd bn'b");
});

app.get("/api/listings", handleGetAll);
app.get("/api/listings/:id", handleGetOne);
app.post("/api/listings", handleCreate);
app.put("/api/listings/:id", handleUpdate);
app.delete("/api/listings/:id", handleDelete);
```

In Lab 3 the handler was written **inside** `app.get( ... )`. Now it has a name, and the route only says which one to run.

CONTROLLERS

```
function handleGetAll(req, res) {
  res.json(getAllListings());
}

function handleGetOne(req, res) {
  const listing = getListingById(Number(req.params.id));
  if (!listing) {
    return res.status(404).json({ error: "Listing not found" });
  }
  res.json(listing);
}
```

`handleGetOne` never touches the array, and `getListingById` never sees `req` or `res`.



Split your server into layers

1. Add the section banners to `server.js`: MIDDLEWARE, DATABASE, MODEL, CONTROLLERS, ROUTES, START.
2. Move the `listings` array under DATABASE.
3. Under MODEL, write `getAllListings()` and `getListingById(id)`.
4. Turn your two inline handlers into the controllers `handleGetAll` and `handleGetOne`.
5. Make the routes point at them: `app.get("/api/listings", handleGetAll)`.
6. Test in the browser. **Everything should behave exactly as before.** That is what a refactor is.
7. Commit and push.



Testing with an API client

The address bar only sends GET

API clients

Typing a URL in the address bar always sends a `GET`, with no body.

An API client lets you choose **everything**: the method, the URL, the headers, and the body.

-  **Postman**: a desktop app, the most widely used.
-  **Thunder Client**: the same idea, inside  **VS Code**.

Both are fine. Pick one and stick to it.



Postman



Thunder Client

Sending a request

The screenshot displays the Postman interface with a GET request and its response. Red circles and lines highlight specific components: 1. The HTTP method 'GET'. 2. The URL 'http://localhost:3000/api/listings/2'. 3. The 'Send' button. 4. The status bar showing 'Status: 200 OK', 'Size: 50 Bytes', and 'Time: 38 ms'. 5. The response body, which is a JSON object:

```
{  "id": 2,  "title": "Beach House",  "pricePerNight": 140}
```

Query Parameters

parameter	value

Response

Chart

1 method · 2 URL · 3 Send · 4 status · 5 response body. Postman has the same parts in the same places.



Your GET routes, again

1. Start your server with `npm run dev`.
2. In your API client, send `GET http://localhost:3000/api/listings`.
3. Send `GET http://localhost:3000/api/listings/99`. Find the **status code**. Where is it shown?
4. Try `POST http://localhost:3000/api/listings`. What comes back, and why?



Middleware & the request body

What app.use actually does

A mystery

Say we write a `handleCreate` controller and send it a new listing:



We **did** send a title. Why does the server say it is missing?

Add `console.log(req.body)` to the controller and send it again. It prints:

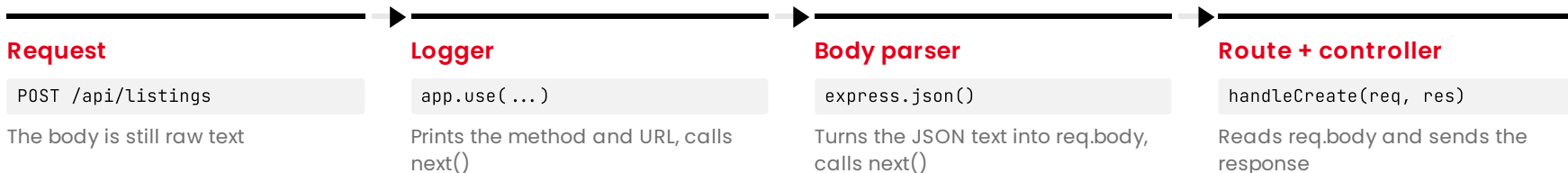
```
undefined
```

A body arrives as **raw text**, in pieces over the network. Express does not read it unless you ask it to.

That is a job for **middleware**.



Middleware



- **Middleware** is a function that runs on a request **before** it reaches any route.
- `app.use(fn)` adds one. They run **in the order you add them**, for every request.
- Each gets `(req, res, next)`. It can read or change `req`, then call `next()` to pass it on, or answer straight away with `res`.
- Forget to call `next()` and the request hangs forever.

Two pieces of middleware

server.js

```
// Log every request, then pass it on
app.use((req, res, next) => {
  console.log(req.method, req.url);
  next();
});

// Read JSON request bodies into req.body
app.use(express.json());
```

- The **logger** is ours: every request now shows up in the terminal.
- `express.json()` is built into Express. When a request says its body is JSON, it parses it and puts the result on `req.body`.
- That is all **body parsing** means: turning the text of the body into a JavaScript object.
- Both come **before** the routes, so they run first.



Create, update & delete

Create: POST

```
function handleCreate(req, res) {  
  const data = req.body || {}; // no body sent? use an empty object  
  if (!data.title || !data.pricePerNight) {  
    return res.status(400).json({ error: "title and pricePerNight are required" });  
  }  
  res.status(201).json(addListing(data));  
}
```

- **Never trust the client.** Check the body before handing it to the model.
- Missing fields? `400 Bad Request`, and say what is wrong.
- Created? `201 Created`, and send the new listing back, with the `id` the server gave it.
- `req.body` is `undefined` when no body was sent, so `|| {}` gives us an empty object to check instead.

Sending a JSON body

The screenshot shows the Postman interface with a POST request to `http://localhost:3000/api/listings`. The request body is set to JSON and contains the following content:

```
{
  "title": "Lake View Penthouse",
  "pricePerNight": 400
}
```

The response status is `201 Created`, the size is `58 Bytes`, and the time is `46 ms`. The response body is also shown as JSON:

```
{
  "id": 4,
  "title": "Lake View Penthouse",
  "pricePerNight": 400
}
```

1 POST · 2 Body · 3 JSON · 4 the listing · 5 201 Created. In Postman: **Body** → **raw** → **JSON**. Either way the `Content-Type: application/json` header is set, which is what `express.json()` looks for.

Update: PUT

```
function handleUpdate(req, res) {
  const data = req.body || {};
  if (!data.title || !data.pricePerNight) {
    return res.status(400).json({ error: "title and pricePerNight are required" });
  }
  const listing = updateListing(Number(req.params.id), data);
  if (!listing) {
    return res.status(404).json({ error: "Listing not found" });
  }
  res.json(listing);
}
```

- The same checks as create, plus a 404 when the id does not exist.
- PUT **replaces** the whole listing, so send **every** field, not only the one that changed.

PUT or PATCH? PATCH updates only the fields you send, so { "pricePerNight": 500 } would change the price and keep the rest. Many APIs offer both. We stick to PUT here; a PATCH version of updateListing would only copy the fields that were sent.

Delete: DELETE

```
function handleDelete(req, res) {  
  const listing = removeListing(Number(req.params.id));  
  if (!listing) {  
    return res.status(404).json({ error: "Listing not found" });  
  }  
  res.status(204).end(); // done, nothing to send back  
}
```

- 204 No Content : it worked, and there is nothing to send back.
- .end() finishes the response with no body.
- Deleting something that is not there is a 404, like everywhere else.



The status codes we use

Code	Name	When
200	OK	A read or an update worked; the data is in the body
201	Created	A new listing was made; it is in the body
204	No Content	A delete worked; there is no body
400	Bad Request	The client sent something invalid
404	Not Found	No listing has that <code>id</code>
500	Internal Server Error	Our code crashed. Express sends this for you

2xx it worked · 4xx the client asked for something wrong · 5xx the server broke

Complete the API

1. Add the logger and `express.json()` middleware at the top.
2. Add `addListing`, `updateListing` and `removeListing` to the model, then `handleCreate`, `handleUpdate`, `handleDelete` and their routes.
3. Test each request in order, and predict the status code first:

Request	Body	Expect
GET /api/listings	—	200
POST /api/listings	{ "title": "Lake View Penthouse", "pricePerNight": 400 }	201
POST /api/listings	{ "title": "No price" }	400
PUT /api/listings/1	{ "title": "Nile View Apartment", "pricePerNight": 500 }	200
DELETE /api/listings/1	—	204
DELETE /api/listings/1	—	404

4. Watch the terminal: the logger prints every request. Commit and push.



Now restart the server

Stop it with **Ctrl+C**, start it again, and send `GET /api/listings`.

- Where did **Lake View Penthouse** go? Why is listing 1 back?
- The array lives in the server's **memory**. When the process stops, it is gone.
- `--watch` restarts on every save, which is why your test data kept disappearing today.

Real applications keep their data in a **database**. That is next: MongoDB replaces the DATABASE section and the insides of the MODEL functions. The routes and controllers you wrote today stay.

The full solution

The complete `server.js`, section by section, as we built it today, with its `package.json` :

[The ebd bn'b solution on GitHub Gist](#)

Compare it with yours. If yours works but looks different, that is fine; ask us if you are unsure why.





Useful references

EXPRESS

[Express: routing guide](#)

[Express: using middleware](#)

[Express: writing middleware](#)

[Express API: `express.json\(\)`](#)

HTTP

[MDN: HTTP request methods](#)

[MDN: PATCH](#), and how it differs from PUT

[MDN: HTTP response status codes](#)

API CLIENTS

[Postman: send your first request](#)

[Thunder Client on the VS Code Marketplace](#)

Nothing on this slide is required.

Questions?

Open floor for any questions

