express # guide

Intro to Express

TCSS 460 — Client/Server Programming

Express is the web framework you will use to build every API in this course. It runs on Node.js, handles HTTP requests, and lets you send back JSON responses — the core mechanics behind every web API. This guide walks you through creating a server, defining routes, and handling the different ways clients send data to your API.

---

1 What is Express?

Express is a minimal web framework for Node.js. It gives you just enough to receive HTTP requests and send responses — nothing more. There is no XML configuration, no project wizard, and no generated boilerplate. You install it, write a few lines of TypeScript, and you have a running web server.

1.1 Express 5

This course uses **Express 5**, the current major version. Express 5 includes built-in support for `async` route handlers and several improvements over Express 4. If you see older tutorials online that reference Express 4 patterns (like wrapping async handlers), know that Express 5 handles these automatically.

> **Gen AI & Learning: Express Version Matters**
>
> AI coding assistants are trained on millions of Express 4 examples. If you ask one to help with Express code, it may suggest patterns from Express 4 that are unnecessary or incorrect in Express 5. Always specify that you are using Express 5 in your prompts, and verify suggestions against the Express 5 documentation.

---

2 Your First Express App

Let's start with the absolute minimum: a server that listens on a port and responds to one request.

2.1 Installation

You need a Node.js project with TypeScript configured before adding Express. If you do not have one yet, follow Creating a TypeScript Project from Scratch in the Node.js Setup guide.

Once your project is set up, install Express and its type definitions:

```
npm install express
npm install --save-dev @types/express
```

2.2 The Minimal Server

Create a file called `src/index.ts`:

```
import express from 'express';

const app = express();
const PORT = 3000;

app.get('/', (request, response) => {
    response.json({ message: 'Hello from Express' });
});

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

That is the entire server. Four things happen here:

1. **Import Express** — `express` is a function that creates an application instance
2. **Create the app** — `const app = express()` creates your server
3. **Define a route** — `app.get('/')` says "when a GET request arrives at `/`, run this function"
4. **Start listening** — `app.listen(PORT)` opens the port and waits for requests

Run it:

```
npx tsx src/index.ts
```

Then visit `http://localhost:3000` in your browser. You should see `{"message":"Hello from Express"}`.

2.3 Using Environment Variables for the Port

Hardcoding the port works for a quick test, but real projects read it from an environment variable. This is what the lecture demo does.

First, install the `dotenv` package:

```
npm install dotenv
```

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

```
PORT=3000
```

Add `.env` to your `.gitignore` (your starter projects already include this, but if you are starting from scratch, do not forget — `.env` files often contain secrets like API keys and database passwords):

```
.env
```

Now update your code to read from the environment:

```
import 'dotenv/config';
import express from 'express';

const app = express();
const PORT = parseInt(process.env.PORT || '3000', 10);

app.get('/', (request, response) => {
    response.json({ message: 'Hello from Express' });
});

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

The `dotenv` package loads variables from a `.env` file in your project root:

```
PORT=3000
```

> **Why Not Hardcode the Port?**
>
> When you deploy your API, the hosting platform assigns the port. If your code says `app.listen(3000)` and the platform gives you port `8080`, your server never receives traffic. Reading from `process.env.PORT` lets the platform control it.

---

3 Routes — Responding to Requests

A **route** is a combination of an HTTP method and a URL path. When a request matches a route, Express runs the function you provided. That function receives two objects: the **request** (`request`) and the **response** (`response`).

3.1 The Request/Response Pattern

Every route handler follows the same signature:

```
app.get('/hello', (request, response) => {
    // request = information about the incoming request
    // response = tools for sending a response back
    response.json({ message: 'Hello, you sent a GET request' });
});
```

This is the pattern for every route you will ever write in Express. The request comes in, you do something with it, and you send a response back.

3.2 Route Handler Functions

In the lecture demo, route handlers are defined as standalone functions in separate files. Here is the `getHello` handler from `controllers/hello.ts` :

```
import { Request, Response } from 'express';

export const getHello = (request: Request, response: Response) => {
    response.json({ message: 'Hello, you sent a GET request' });
};
```

Notice the type annotations: `request: Request` and `response: Response` . These are imported from Express and give you full autocomplete and type checking in your editor. The underscore prefix (`_request`) is a TypeScript convention for parameters you must accept

but do not use — you will see this in the lecture demo when the handler ignores the request object.

---

4 HTTP Methods in Express

HTTP defines several methods (also called "verbs") that indicate what the client wants to do. Express gives you a function for each one:

```
app.get('/hello', handler);     // Read data
app.post('/hello', handler);    // Create data
app.put('/hello', handler);     // Replace data
app.patch('/hello', handler);   // Update part of data
app.delete('/hello', handler);  // Remove data
```

4.1 One Path, Multiple Methods

The same URL path can respond to different HTTP methods. The lecture demo defines all five methods on `/v1/hello`, each returning a message that identifies which method was used:

```
import { Request, Response } from 'express';

export const getHello = (_request: Request, response: Response) => {
    response.json({ message: 'Hello, you sent a GET request' });
};

export const postHello = (_request: Request, response: Response) => {
    response.status(201).json({ message: 'Hello, you sent a POST request' });
};

export const putHello = (_request: Request, response: Response) => {
    response.json({ message: 'Hello, you sent a PUT request' });
};

export const patchHello = (_request: Request, response: Response) => {
    response.json({ message: 'Hello, you sent a PATCH request' });
};

export const deleteHello = (_request: Request, response: Response) => {
    response.json({ message: 'Hello, you sent a DELETE request' });
};
```

Notice that `postHello` uses `response.status(201)` — a 201 status means "created," which is the conventional response for POST requests that create a new resource.

4.2 Testing Beyond GET

Your browser's address bar can only send GET requests. To test POST, PUT, PATCH, and DELETE, you need a tool like **Postman** or **Thunder Client** (a VS Code extension). We cover these in the API Testing Tools guide.

> **Try It Yourself**
>
> 1. Start the lecture demo: `npm run dev`
> 2. Open your browser and visit `http://localhost:3000/v1/hello`
> 3. You will see the GET response
> 4. Open Postman and send a POST request to the same URL
> 5. Notice the different message and the `201` status code

---

5 Route Parameters

Route parameters let the client specify a value as part of the URL path. They are defined with a colon (`:`) prefix:

```
app.get('/users/:id', (request, response) => {
    const { id } = request.params;
    response.json({ message: 'User details', params: { id } });
});
```

When a client sends `GET /users/42`, Express extracts `42` and puts it in `request.params.id`.

5.1 From the Lecture Demo

The lecture demo's `getUserById` handler in `controllers/input.ts` shows this pattern:

```
export const getUserById = (request: Request, response: Response) => {
    const { id } = request.params;
    response.json({
        message: 'User details',
        params: { id },
    });
};
```

The route is defined in `routes/v1/input.ts`:

```
inputRouter.get('/users/:id', getUserById);
```

So a request to `GET /v1/input/users/42` produces:

```
{
    "message": "User details",
    "params": { "id": "42" }
}
```

5.2 Parameters Are Always Strings

Gen AI & Learning: Route Parameters and AI Tools

When you ask an AI agent to "add a route that takes a user ID," it will almost always generate correct parameter extraction (`request.params.id`). But it often forgets the string-to-number conversion and validation. Always check generated route handlers for proper type conversion and error handling on parameters — the happy path will work, but the sad path (invalid input) is where bugs hide.

Common Mistake: Assuming Params Are Numbers

`request.params.id` is always a `string`, even if the URL contains `42`. If you need a number, you must convert it yourself:

```
const id = Number(request.params.id);
if (!Number.isInteger(id) || id <= 0) {
    response.status(400).json({ error: 'Parameter "id" must be a
positive integer' });
    return;
}
```

This exact validation appears in the lecture demo's `validateNumericId` middleware — you will see it in the Routing & Middleware guide.

---

6 Query Strings

Query strings are key-value pairs appended to the URL after a `?`. They are commonly used for search, filtering, and pagination:

```
GET /search?q=express&limit=10
```

Express parses these automatically into `request.query`:

```
app.get('/search', (request, response) => {
    const { q, limit } = request.query;
    response.json({
        message: 'Search results',
        query: { q: q ?? null, limit: limit ?? null },
    });
});
```

6.1 From the Lecture Demo

The `searchByQuery` handler in `controllers/input.ts`:

```
export const searchByQuery = (request: Request, response: Response) => {
    const { q, limit } = request.query;
    response.json({
        message: 'Search results',
        query: {
            q: q ?? null,
            limit: limit ?? null,
        },
    });
};
```

A request to `GET /v1/input/search?q=hello&limit=5` produces:

```
{
    "message": "Search results",
    "query": { "q": "hello", "limit": "5" }
}
```

> ⚠ **Query Values Are Also Strings**
>
> Just like route parameters, query string values are always strings. The `limit` above is `"5"` (a string), not `5` (a number). Convert when needed.

6.2 Params vs. Query Strings

When should you use each?

Use	When	Example
Route parameters	Identifying a specific resource	`GET /users/42`
Query strings	Filtering, searching, or optional modifiers	`GET /users?role=admin&limit=10`

A good rule of thumb: if the value identifies **which** resource, use a parameter. If it modifies **how** you retrieve or filter resources, use a query string.

---

7 Request Body (POST/PUT)

When a client creates or updates a resource, it sends data in the **request body** as JSON. Express does not parse JSON bodies by default — you must enable it.

7.1 Enabling JSON Parsing

Add this line before your routes:

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

This is **middleware** (covered in depth in the Routing & Middleware guide). For now, just know it must be present or `request.body` will be `undefined`.

The lecture demo enables it in `app.ts`:

```
const app = express();
app.use(cors());
app.use(express.json());
```

7.2 Reading the Body

Once JSON parsing is enabled, `request.body` contains the parsed JSON object:

```
export const createUser = (request: Request, response: Response) => {
    const { name, email } = request.body;
    response.status(201).json({
        message: 'User created',
```

```
        body: { name, email },
    });
};
```

A POST request with this JSON body:

```
{
    "name": "Alice",
    "email": "alice@example.com"
}
```

Produces this response:

```
{
    "message": "User created",
    "body": {
        "name": "Alice",
        "email": "alice@example.com"
    }
}
```

---

8 Sending Responses

Express provides several methods on the `response` object for sending responses back to the client.

8.1 JSON Responses

The most common response in an API is JSON:

```
response.json({ message: 'success', data: someObject });
```

`response.json()` calls `JSON.stringify()` internally, sets the `Content-Type` header to `application/json`, and sends the result. You pass it a raw JavaScript object and the serialization happens automatically — you never call `JSON.stringify()` yourself in a route handler.

8.2 Status Codes

Chain `response.status()` before `response.json()` to set the HTTP status code:

```
// 201 Created — resource was successfully created
response.status(201).json({ message: 'User created', body: { name, email } });

// 404 Not Found — resource does not exist
response.status(404).json({ error: 'User not found' });

// 400 Bad Request — client sent invalid data
response.status(400).json({ error: 'Missing required field: name' });
```

If you do not call `response.status()`, Express defaults to `200 OK`.

8.3 Common Status Codes

You will use these status codes throughout the course:

Code	Meaning	When to Use
`200`	OK	Successful GET, PUT, PATCH, DELETE
`201`	Created	Successful POST that created a resource
`400`	Bad Request	Client sent invalid data
`401`	Unauthorized	Missing or invalid authentication
`403`	Forbidden	Authenticated but insufficient permissions
`404`	Not Found	Resource does not exist
`500`	Internal Server Error	Something unexpected broke

Important

Always send a JSON body with error responses. A raw `404` with no body is unhelpful — the client has no idea what went wrong. Always include a message: `response.status(404).json({ error: 'User not found' })`.

---

9 Request Headers

Clients can send additional information in HTTP **headers** — key-value pairs attached to the request. Common headers include authentication tokens, content types, and custom request IDs.

9.1 Reading Headers

Access headers through `request.headers`:

```
export const echoHeaders = (request: Request, response: Response) => {
    response.json({
        message: 'Headers received',
        headers: {
            'x-request-id': request.headers['x-request-id'] ?? null,
            'x-custom-header': request.headers['x-custom-header'] ?? null,
            'content-type': request.headers['content-type'] ?? null,
        },
    });
};
```

Tip

Header names are always **lowercase** in `request.headers`, regardless of how the client sent them. Use `request.headers['content-type']`, not `request.headers['Content-Type']`.

Notice the bracket notation (`request.headers['x-request-id']`) instead of dot notation. Header names contain hyphens, and `request.headers.x-request-id` would be interpreted as `request.headers.x` minus `request` minus `id` — JavaScript sees the hyphens as subtraction. Bracket notation avoids this.

---

10 Project Structure

For a "hello world" server, putting everything in one file is fine. But as your API grows, you will split code into separate files by responsibility.

10.1 The Lecture Demo Structure

The lecture demo organizes code like this:

```
src/
├── index.ts          ← Server startup (app.listen)
├── app.ts            ← Express app creation and middleware
```

```
├── controllers/      ← Route handler functions
│   ├── hello.ts
│   └── input.ts
├── middleware/        ← Middleware functions
│   ├── logger.ts
│   └── validation.ts
└── routes/           ← Route definitions (which paths use which handlers)
    ├── index.ts
    ├── v1/
    │   ├── index.ts
    │   ├── hello.ts
    │   └── input.ts
    └── v2/
        ├── index.ts
        ├── hello.ts
        └── input.ts
```

10.2 Why Split?

Concern	File	What It Does
Starting the server	`index.ts`	Loads environment, calls `app.listen()`
Configuring the app	`app.ts`	Creates Express app, registers middleware
Defining routes	`routes/*.ts`	Maps URL paths to handler functions
Handling requests	`controllers/*.ts`	The actual logic for each route
Cross-cutting logic	`middleware/*.ts`	Logging, validation, authentication

This separation means you can test your app without starting a server (import `app` directly), add routes without touching middleware, and add middleware without touching route handlers.

Don't Worry About This Yet

You will start with a single file and grow into this structure naturally. The Routing & Middleware guide covers `Router` and multi-file organization in detail. For now, understand that the lecture demo is organized this way and know where to find things.

Gen AI & Learning: Project Structure and Context Engineering

A well-organized project structure is one of the most effective forms of context engineering. When an AI coding agent opens your project and sees clearly named directories — `controllers/`, `routes/`, `middleware/` — it immediately understands where new code should go. If you ask it to "add a route for movies," it knows to create `controllers/movies.ts` and `routes/v1/movies.ts` because it learned the pattern from your existing files. The structure teaches the agent your conventions without you having to explain them.

---

11 Putting It Together

Here is a complete single-file Express app that demonstrates everything covered in this guide:

```
import express, { Request, Response } from 'express';

const app = express();
const PORT = 3000;

// Enable JSON body parsing
app.use(express.json());

// GET — read
app.get('/hello', (_request: Request, response: Response) => {
    response.json({ message: 'Hello from Express' });
});

// POST — create
app.post('/users', (request: Request, response: Response) => {
    const { name, email } = request.body;
    response.status(201).json({ message: 'User created', body: { name, email }
});
});

// Route parameter
app.get('/users/:id', (request: Request, response: Response) => {
    const { id } = request.params;
    response.json({ message: 'User details', params: { id } });
});

// Query string
app.get('/search', (request: Request, response: Response) => {
    const { q, limit } = request.query;
    response.json({ query: { q: q ?? null, limit: limit ?? null } });
});

app.listen(PORT, () => {
```

```
    console.log(`Server running at http://localhost:${PORT}`);
});
```

Try It Yourself

1. Create a new file called `src/index.ts` in a project with Express installed
2. Paste the code above
3. Run it with `npx tsx src/index.ts`
4. Test each endpoint:
   - Browser: `http://localhost:3000/hello`
   - Browser: `http://localhost:3000/users/42`
   - Browser: `http://localhost:3000/search?q=express&limit=5`
   - Postman: POST to `http://localhost:3000/users` with JSON body `{"name": "Alice", "email": "alice@example.com"}`

---

12 Summary

Concept	Key Point
Express	Minimal Node.js web framework — handles HTTP for you
`app.listen(PORT)`	Starts the server on the given port
`app.get()`, `.post()`, etc.	Define routes for different HTTP methods
`request.params`	Values from URL path segments (`:id`) — always strings
`request.query`	Values from query strings (`?q=hello`) — always strings
`request.body`	Parsed JSON from the request body (requires `express.json()`)
`request.headers`	HTTP headers sent by the client — always lowercase keys

Concept	Key Point
`response.json()`	Send a JSON response
`response.status(code)`	Set the HTTP status code before sending
Controllers	Separate files for route handler functions
Routes	Separate files that map paths to handlers

---

13 References

Official Documentation:

- Express.js — Getting Started — Installation and hello world
- Express.js — Routing — Route methods, paths, and parameters
- Express.js — API Reference: Request — `request.params`, `request.query`, `request.body`
- Express.js — API Reference: Response — `response.json()`, `response.status()`
- MDN — HTTP Status Codes — Complete reference

TCSS 460 Lecture Demo

github.com/UWT-SET-TCSS460-LECTURE-MATERIALS/TCSS-460-Backend-1 — Express, TypeScript, routes, and controllers

---

14 Further Reading

External Resources

- Express 5 Migration Guide — Changes from Express 4 to Express 5
- MDN — HTTP Request Methods — Definitions of GET, POST, PUT, PATCH, DELETE
- The dotenv Package — Environment variable management

This guide is part of TCSS 460 — Client/Server Programming, School of Engineering and Technology, University of Washington Tacoma.