Express.js

Version 5.2.1 · verified 2026-07-31

Minimalist web framework for Node.js, typed for safety.

Contents

Mental model

Express is a thin pipeline over Node's native HTTP module. Every request passes through a chain of middleware functions, in the order they were registered. Each function can modify the request, terminate the cycle by responding, or pass control to the next function.

App setup

Concept	Usage
Instantiation	`const app = express()`
Listen	`app.listen(port, callback)`
JSON parsing	`app.use(express.json())`
URL-encoded	`app.use(express.urlencoded())`

Express applications are typically created by calling the exported `express()` function. Middlewares like `express.json()` must be mounted before route handlers that expect `req.body`.

```
import express from "express";

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

app.listen(3000, () => {
  console.log("Server running on port 3000");
});
```

> **Warning:** Without `app.use(express.json())`, `req.body` will be `undefined` for JSON payloads.

Routing

Method	Syntax
GET	`app.get(path, handler)`
POST	`app.post(path, handler)`
All methods	`app.all(path, handler)`

Routes map HTTP methods and URLs to handler functions. A handler receives the request (`req`) and

response (`res`) objects. Path strings can contain route parameters.

```
import express from "express";
import type { Request, Response } from "express";
const app = express();

app.get(
  "/users/:id",
  (req: Request, res: Response) => {
    const id = req.params.id; // string
    res.json({ id, name: "Alice" });
  }
);
```

> **Gotcha:** `req.params` values are always strings. If you need a number, you must explicitly parse it (e.g., `parseInt(req.params.id)`).

Middleware

Type	Signature
Standard	`(req, res, next) => void`
Error handling	`(err, req, res, next) => void`

Middleware functions execute sequentially. They must either send a response or call `next()`. If neither happens, the request will hang permanently.

```
import express from "express";
import type {
  Request, Response, NextFunction
} from "express";
const app = express();

function logger(
  req: Request, res: Response, next: NextFunction,
) {
  console.log(`${req.method} ${req.path}`);
  next(); // Pass control to the next middleware
```

```
}

app.use(logger); // Apply globally
```

> **Gotcha:** If you call `next()` after `res.send()`, subsequent middlewares will still run, but they cannot modify the response headers.

Error handling

Tool	Purpose
`next(err)`	Passes an error to error middleware
Error handler	Catches errors from synchronous code

Express catches synchronous errors automatically. For asynchronous handlers, you must pass the error to `next(err)` or use a wrapper like `express-async-errors`. Error middlewares must have exactly four arguments to be recognized by Express.

```
import express from "express";
import type {
  Request, Response, NextFunction
} from "express";
const app = express();

app.use((
  err: Error, req: Request,
  res: Response, next: NextFunction,
) => {
  console.error(err.stack);
  res.status(500).send("Something broke!");
});
```

> **Warning:** Error handling middleware must be defined last, after all other `app.use()` and route calls.

Typing Request and Response

Interface	Usage
`Request`	Types the incoming HTTP request
`Response`	Types the outgoing HTTP response

The `Request` type accepts generics for params, response body, request body, and query. This provides strict typing for incoming data.

```
import express from "express";
import type { Request, Response } from "express";
const app = express();
type P = { id: string };
type B = { message: string };

// Request<Params, ResBody, ReqBody, ReqQuery>
app.post(
  "/users/:id",
  (req: Request<P, B>, res: Response<B>) => {
    res.json({ message: "Success" });
```

```
  }
);
```

> **Tip:** You usually only need to provide the generics you care about, defaulting the rest to `any` or `unknown`.

Further reading

- Express routing documentation
- Writing middleware
- Error handling in Express