Express Course

Flavio Copes

Updated August 6, 2026

Contents

Build an Express 5 application with clear routing, middleware, validation, sessions, errors, tests, security, and production behavior.

What you'll learn: Ship a tested notes application with HTML and JSON paths, secure input and sessions, useful errors, observability, and graceful shutdown.

Express foundations

Follow requests, construct the app, validate configuration, and serve static assets.

Follow one Express request

Trace a request through Node, Express middleware, a route handler, and the final response.

Before choosing a tool or writing more code, make the requirement concrete. A request for `/notes` enters the app, crosses ordered middleware, matches one route, and ends when a response is sent or delegated.

Express is a thin request-routing and middleware layer on top of the Node HTTP server. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

> **Diagram: One Express request.** Node receives an HTTP request, Express runs ordered middleware, a matching route handles it, and the response returns to the browser. Middleware can also forward an error. A common failure is to describe Express as a complete server that hides the Node request and response lifecycle. The happy path may still work, which makes the mistake easy to miss. Draw the ordered request path and identify exactly which function ends or forwards it. Keep the first version small enough that every important input and output remains visible.

This supports the module goal: Start a small Express 5 application whose configuration and request path can be explained from the first byte. Capture the request, command, trace, screenshot, test result, or other evidence that proves the result.

Add three tiny middleware functions that log entry and exit, then predict and verify their order. Change one condition on purpose, predict the result, and compare the prediction with what actually happened.

Create the application

Set up an Express 5 project with explicit scripts, environment requirements, and a minimal start file.

Start from the behavior you can observe. The notes app exports the Express application for tests and starts listening from a separate executable module.

The first application should reveal the server, port, and startup failure instead of hiding them behind scaffolding. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

It is tempting to mix app construction, listener startup, tests, and environment mutation in one unimportable file. That trades a clear decision for an assumption you will eventually have to debug. The stronger move is to separate app creation from process startup and fail clearly when required configuration is missing. Make the boundary explicit in code and in the project notes.

Tie the decision back to the larger job: Start a small Express 5 application whose configuration and request path can be explained from the first byte. Record enough evidence that another developer can repeat the result without relying on your memory.

Start the app, import it without opening a port, and prove a test can exercise it in isolation. Repeat it once with an invalid or hostile condition and write down the boundary the failure revealed.

Configure without global mystery

Read, validate, and pass configuration deliberately instead of reaching into process.env everywhere.

The session secret is required, the port has a safe local default, and environment-specific trust proxy behavior is explicit. That contrast gives us something useful to test instead of a rule to memorize.

Configuration is input to the application and should have a schema, defaults, and an obvious production failure mode. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

The happy path can hide a bad design: you can scatter environment lookups through handlers and discover invalid values on the first user request and still get one successful demo. Push past the demo. validate configuration once at startup and inject the resulting values where they are needed, then test the condition most likely to prove the choice wrong.

The module is moving toward one outcome: Start a small Express 5 application whose configuration and request path can be explained from the first byte. Save the before-and-after evidence now; it will make the final project review much more honest.

Run the server with a missing secret, an invalid port, and a complete local configuration; capture each outcome. Save the evidence, then explain which requirement would force you to choose a different design.

Serve static assets

Mount static files at a clear URL boundary and understand cache and path behavior.

The notes stylesheet lives under `/assets/`, uses a resolved directory path, and receives a deliberate cache policy. This is a small part of an Express 5 notes application with an HTML interface, a JSON API, sessions, tests, and production failure handling, but the boundary it creates affects everything that follows.

Static middleware maps a URL prefix to a filesystem directory before later dynamic routes run. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

Watch for one shortcut in particular: trying to serve the repository root or depend on the current working directory accidentally. It makes later failures harder to locate. Instead, mount the smallest public directory, resolve its path explicitly, and test missing files and caching headers. Prefer an implementation you can explain from the outside before optimizing it.

Keep the wider goal in view: Start a small Express 5 application whose configuration and request path can be explained from the first byte. A short observation with concrete evidence is more useful than a confident sentence with no reproduction path.

Request a known asset, a missing asset, and a private source file; only the intended public file should succeed. Hand the result to someone else and see whether the behavior is clear without an oral explanation.

Check your understanding: express foundations

Check application setup, request flow, configuration, static files, and the Express 5 runtime boundary.

Answer each question before continuing. Read the explanation after every answer, including the

ones you get right.

Take this interactive quiz online

Routes and responses

Match routes, read inputs, negotiate representations, and finish responses once.

Design route order

Arrange specific, parameterized, router, and fallback routes so matching is predictable.

Before choosing a tool or writing more code, make the requirement concrete. The literal `/notes/new` route appears before `/notes/:id`, and the final not-found handler runs only after every router declines.

Express evaluates routes and middleware in registration order, which makes order part of application behavior. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

A common failure is to register broad parameter or wildcard routes first and debug why later handlers never run. The happy path may still work, which makes the mistake easy to miss. list concrete route examples, order the narrow matches first, and test the fallback explicitly. Keep the first version small enough that every important input and output remains visible.

This supports the module goal: Design routes that match intentionally, parse input from the correct location, and return one complete response. Capture the request, command, trace, screenshot, test result, or other evidence that proves the result.

Create a route table with conflicting paths, predict the handler for each request, and verify it with tests. Change one condition on purpose, predict the result, and compare the prediction with what actually happened.

Read params, query, and body

Distinguish path identity, optional query controls, and parsed request bodies.

Start from the behavior you can observe. A note id comes from `req.params`, a search filter from `req.query`, and editable content from a parsed form or JSON body.

Parameters, query strings, and bodies have different semantics even though every value still needs validation. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

It is tempting to pull every value from whichever object happens to contain a matching property name. That trades a clear decision for an assumption you will eventually have to debug. The stronger move is to define the location and schema of every input in the route contract before reading it. Make the boundary explicit in code and in the project notes.

Tie the decision back to the larger job: Design routes that match intentionally, parse input from the correct location, and return one complete response. Record enough evidence that another developer can repeat the result without relying on your memory.

Send the same key in all three locations and prove the handler uses only the documented source. Repeat it once with an invalid or hostile condition and write down the boundary the failure revealed.

Send one clear response

Choose status, headers, content type, and body deliberately and return after completing the response.

Creating a note returns the appropriate success status and representation, while an empty delete does not invent a JSON body. That contrast gives us something useful to test instead of a rule to memorize.

A handler must either finish the response, delegate, or throw; continuing after a send causes double-response bugs. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

The happy path can hide a bad design: you can send a response and continue into code that can send or mutate state again and still get one successful demo. Push past the demo. make every branch visibly return, delegate, or fail and test the response status and content type, then test the condition most likely to prove the choice wrong.

The module is moving toward one outcome: Design routes that match intentionally, parse input from the correct location, and return one complete response. Save the before-and-after evidence now; it will make the final project review much more honest.

Instrument one handler with success, validation, missing-record, and failure branches and prove each terminates once. Save the evidence, then explain which requirement would force you to choose a different design.

Negotiate HTML and JSON

Keep representation logic explicit when one application serves pages and an API.

The form flow redirects after a successful POST, while the API returns JSON and a location header. This is a small part of an Express 5 notes application with an HTML interface, a JSON API, sessions, tests, and production failure handling, but the boundary it creates affects everything that follows.

The same underlying note can be rendered as HTML or serialized as JSON, but redirects and API responses have different client expectations. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

Watch for one shortcut in particular: trying to guess the client type from a fragile user-agent string. It makes later failures harder to locate. Instead, use separate routes or explicit content negotiation and keep the domain operation independent of rendering. Prefer an implementation you can explain from the outside before optimizing it.

Keep the wider goal in view: Design routes that match intentionally, parse input from the correct location, and return one complete response. A short observation with concrete evidence is more useful than a confident sentence with no reproduction path.

Exercise the same create operation from a browser form and a JSON client, then compare the two response contracts. Hand the result to someone else and see whether the behavior is clear without an oral explanation.

Check your understanding: routes and responses

Check route matching, parameters, queries, representations, redirects, status codes, and response completion.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Middleware and input

Compose middleware and handle bodies, validation, uploads, cookies, and sessions.

Compose middleware

Use small ordered middleware for cross-cutting behavior without turning request flow into a maze.

Before choosing a tool or writing more code, make the requirement concrete. A request id and logger run globally, while note ownership checks live only on the notes router.

Middleware is powerful because it can observe, change, stop, or forward a request; that power makes scope and order critical. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

A common failure is to make every helper global and let unrelated routes pay its cost or inherit its assumptions. The happy path may still work, which makes the mistake easy to miss. mount middleware at the narrowest useful scope and document whether it reads, changes, stops, or forwards. Keep the first version small enough that every important input and output remains visible.

This supports the module goal: Accept useful input while keeping parsing, validation, state, and privilege boundaries visible. Capture the request, command, trace, screenshot, test result, or other evidence that proves the result.

Write an order diagram for parsing, authentication, rate limiting, authorization, handler, and error processing. Change one condition on purpose, predict the result, and compare the prediction with what actually happened.

Parse only expected bodies

Enable JSON and URL-encoded parsers with explicit size limits where routes need them.

Start from the behavior you can observe. The JSON API accepts a small JSON document; the form route accepts URL-encoded input; neither accepts an unlimited arbitrary body.

Body parsing allocates work before business validation, so content type and size are part of the security boundary. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

It is tempting to install every parser globally with large defaults and accept mismatched content types. That trades a clear decision for an assumption you will eventually have to debug. The stronger move is to mount the required parser, enforce a conservative limit, and reject unsupported media types before domain work. Make the boundary explicit in code and in the project notes.

Tie the decision back to the larger job: Accept useful input while keeping parsing, validation, state, and privilege boundaries visible. Record enough evidence that another developer can repeat the

result without relying on your memory.

Send valid, malformed, oversized, and wrong-content-type bodies and record the exact response for each. Repeat it once with an invalid or hostile condition and write down the boundary the failure revealed.

Validate and normalize input

Validate types, ranges, shapes, and business rules, then pass a clean value into the domain layer.

A note title is trimmed and length-bounded, but HTML is not destroyed merely because one renderer must escape it. That contrast gives us something useful to test instead of a rule to memorize.

Parsing answers “can I read this?”; validation answers “is this allowed here?”; output encoding protects a later rendering context. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

The happy path can hide a bad design: you can call broad sanitization on every string and assume that replaces validation and output encoding and still get one successful demo. Push past the demo. validate at the boundary, normalize only by explicit product rules, and encode at the output context, then test the condition most likely to prove the choice wrong.

The module is moving toward one outcome: Accept useful input while keeping parsing, validation, state, and privilege boundaries visible. Save the before-and-after evidence now; it will make the final project review much more honest.

Create a table of malformed, boundary, Unicode, and malicious-looking inputs and test the resulting stored and rendered values. Save the evidence, then explain which requirement would force you to choose a different design.

Handle uploads, sessions, and cookies

Treat uploaded bytes and session identifiers as untrusted inputs with controlled storage and lifetime.

The app limits upload size and type, stores files outside public paths, and sends only an opaque session id in a secure cookie. This is a small part of an Express 5 notes application with an HTML interface, a JSON API, sessions, tests, and production failure handling, but the boundary it creates affects everything that follows.

Upload middleware, session stores, and cookie attributes introduce storage and trust decisions beyond ordinary form fields. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

Watch for one shortcut in particular: trying to trust an extension, store uploads under a directly served path, or keep production sessions in process memory. It makes later failures harder to locate. Instead, validate content and size, generate server-side names, use a production session store, and configure restrictive cookies. Prefer an implementation you can explain from the outside before optimizing it.

Keep the wider goal in view: Accept useful input while keeping parsing, validation, state, and privilege boundaries visible. A short observation with concrete evidence is more useful than a confident sentence with no reproduction path.

Upload valid and deceptive files, restart the server, log out, and verify both file and session behavior. Hand the result to someone else and see whether the behavior is clear without an oral explanation.

Check your understanding: middleware and input

Check middleware composition, body parsers, validation, uploads, cookies, sessions, and trust boundaries.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Errors, security, and observation

Handle failures, proxy policy, CORS, logs, and shutdown safely.

Handle async errors in Express 5

Use rejected promises and thrown errors correctly, then verify the Express 5 behavior you depend on.

Before choosing a tool or writing more code, make the requirement concrete. A database rejection reaches the centralized error path, while an expected missing note becomes a controlled not-found response.

Express 5 forwards rejected promises from async handlers to error middleware, reducing the need for repetitive wrappers. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

A common failure is to catch every error only to log and forget it, leaving the request open. The happy path may still work, which makes the mistake easy to miss. throw unexpected failures, translate expected domain outcomes, and test promise rejection through the real router. Keep the first version small enough that every important input and output remains visible.

This supports the module goal: Make failures safe for users and useful for operators without leaking secrets or losing requests silently. Capture the request, command, trace, screenshot, test result, or other evidence that proves the result.

Make the repository reject and verify the request receives one safe response and the operator log retains the cause. Change one condition on purpose, predict the result, and compare the prediction with what actually happened.

Design error middleware

Place one final four-argument error handler that maps known failures and hides internal details.

Start from the behavior you can observe. Validation returns a stable client error, while an unknown database failure returns a generic response with a request id.

Error middleware is part of the public API and the operational record; users and logs need different information. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

It is tempting to return stack traces and raw database messages to every client. That trades a clear decision for an assumption you will eventually have to debug. The stronger move is to classify expected failures, log structured internal context, and send a minimal stable external shape. Make the boundary explicit in code and in the project notes.

Tie the decision back to the larger job: Make failures safe for users and useful for operators without leaking secrets or losing requests silently. Record enough evidence that another developer can repeat the result without relying on your memory.

Trigger validation, authorization, missing-record, and unknown failures and compare client bodies with server logs. Repeat it once with an invalid or hostile condition and write down the boundary the failure revealed.

Configure proxies, CORS, and headers

Set trust proxy and cross-origin policy from the real deployment path instead of copying permissive snippets.

The app trusts only its known proxy hop and allows one documented frontend origin with the required methods and headers. That contrast gives us something useful to test instead of a rule to memorize.

Proxy trust affects client IP and secure-cookie behavior; CORS controls which browser origins may read responses, not who can call the server. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

The happy path can hide a bad design: you can enable trust proxy blindly and use wildcard CORS with credentials and still get one successful demo. Push past the demo. model the proxy chain, allow only required origins and methods, and add defensive response headers at the app or edge, then test the condition most likely to prove the choice wrong.

The module is moving toward one outcome: Make failures safe for users and useful for operators without leaking secrets or losing requests silently. Save the before-and-after evidence now; it will make the final project review much more honest.

Test direct and proxied requests plus allowed and disallowed browser origins; inspect derived IP, protocol, and CORS headers. Save the evidence, then explain which requirement would force you to choose a different design.

Log and shut down

Correlate requests and close the listener and dependencies gracefully when the process receives a stop signal.

Structured logs carry a request id, route, status, duration, and safe error context; shutdown stops new connections and drains bounded work. This is a small part of an Express 5 notes application with an HTML interface, a JSON API, sessions, tests, and production failure handling, but the boundary it creates affects everything that follows.

A production service must explain failures and stop accepting work before its process disappears. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

Watch for one shortcut in particular: trying to log entire bodies and secrets while omitting request outcome and duration. It makes later failures harder to locate. Instead, define redaction, lifecycle metrics, readiness, and a time-bounded graceful shutdown sequence. Prefer an implementation you can explain from the outside before optimizing it.

Keep the wider goal in view: Make failures safe for users and useful for operators without leaking

secrets or losing requests silently. A short observation with concrete evidence is more useful than a confident sentence with no reproduction path.

Start slow requests, send a termination signal, and prove the process stops accepting new work while completed requests keep valid responses. Hand the result to someone else and see whether the behavior is clear without an oral explanation.

Check your understanding: errors, security, and observation

Check async failures, error middleware, safe responses, proxy trust, CORS, logging, and shutdown.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Test and ship Express

Test the HTTP contract, separate domain logic, and verify production configuration.

Test the HTTP contract

Exercise the application as HTTP instead of calling route handlers like ordinary functions.

Before choosing a tool or writing more code, make the requirement concrete. The test suite creates the app with fake dependencies and sends requests without binding a public port.

Black-box request tests cover routing, middleware, parsing, status, headers, body, and error behavior together. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

A common failure is to unit-test req and res mocks until they reproduce a different framework from Express. The happy path may still work, which makes the mistake easy to miss. test public HTTP behavior and reserve unit tests for domain logic that has no transport concerns. Keep the first version small enough that every important input and output remains visible.

This supports the module goal: Ship the notes app with black-box tests, production controls, and a failure matrix that can be rerun after changes. Capture the request, command, trace, screenshot, test result, or other evidence that proves the result.

Write tests for the happy path, malformed input, unauthorized access, missing note, and dependency failure. Change one condition on purpose, predict the result, and compare the prediction with what actually happened.

Separate domain and transport

Keep note rules independent of Express so route tests stay focused and domain tests stay fast.

Start from the behavior you can observe. A note service owns title rules and ownership, while the router owns params, status codes, and representations.

Routes translate HTTP into domain calls and domain outcomes back into HTTP; they should not contain every business rule. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

It is tempting to pass Express request objects deep into the data and domain layers. That trades a clear decision for an assumption you will eventually have to debug. The stronger move is to define plain inputs and outcomes at the boundary and inject dependencies into app construction. Make the boundary explicit in code and in the project notes.

Tie the decision back to the larger job: Ship the notes app with black-box tests, production controls, and a failure matrix that can be rerun after changes. Record enough evidence that another developer can repeat the result without relying on your memory.

Move one complicated route rule into a plain function and compare the resulting route and domain tests. Repeat it once with an invalid or hostile condition and write down the boundary the failure revealed.

Prepare production configuration

Review TLS termination, proxy trust, session storage, limits, secrets, dependencies, and runtime version before deployment.

The deployment pins a supported Node version, injects secrets, uses an external session store, and documents its proxy chain. That contrast gives us something useful to test instead of a rule to memorize.

Local defaults are not production architecture, especially for memory state, cookies, proxy metadata, and shutdown. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

The happy path can hide a bad design: you can promote the local process and memory store unchanged because the demo passed and still get one successful demo. Push past the demo. create a production checklist from the application trust boundaries and verify it in a staging environment, then test the condition most likely to prove the choice wrong.

The module is moving toward one outcome: Ship the notes app with black-box tests, production controls, and a failure matrix that can be rerun after changes. Save the before-and-after evidence now; it will make the final project review much more honest.

Deploy a disposable staging copy and verify cookies, IPs, limits, persistence, logs, health, and shutdown. Save the evidence, then explain which requirement would force you to choose a different design.

Finish the notes application

Complete the HTML and JSON paths, security controls, tests, observability, and recovery notes as one coherent service.

The final repository explains setup, route contracts, configuration, data lifetime, session behavior, test commands, and deployment limits. This is a small part of an Express 5 notes application with an HTML interface, a JSON API, sessions, tests, and production failure handling, but the boundary it creates affects everything that follows.

A finished Express app is not the number of routes; it is a request path you can reason about in success and failure. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

Watch for one shortcut in particular: trying to add another feature while known failure paths and operating assumptions remain undocumented. It makes later failures harder to locate. Instead, run

the complete test and deployment matrix, fix evidence-backed gaps, and publish the operational contract. Prefer an implementation you can explain from the outside before optimizing it.

Keep the wider goal in view: Ship the notes app with black-box tests, production controls, and a failure matrix that can be rerun after changes. A short observation with concrete evidence is more useful than a confident sentence with no reproduction path.

Hand the app to a clean environment and have another person start, test, use, stop, and diagnose it from the documentation. Hand the result to someone else and see whether the behavior is clear without an oral explanation.

Check your understanding: test and ship express

Check application tests, dependency boundaries, production configuration, health, performance, and final delivery.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

The interactive version of this course is available at https://flaviocopes.com/courses/express/.