express # guide # tooling

API Testing

TCSS 460 — Client/Server Programming

You have been building routes, adding middleware, and validating input. But how do you know it all actually works? Manual testing — sending requests in Postman and eyeballing the response — works for a few endpoints, but it does not scale. When your API has dozens of routes, you need automated tests that verify every endpoint still behaves correctly after every change. This guide covers API testing with Jest and Supertest, the same tools used in the lecture demo.

---

1 Why Test a Web API?

Your API is a **contract**. It promises: "If you send a GET request to `/api/movies/550`, I will return that movie's data with a 200 status code. If the movie does not exist, I will return a 404 with an error message." Tests verify that contract holds — and keeps holding as you add features, fix bugs, and refactor code.

1.1 Manual Testing Does Not Scale

When you have three routes, you can test them by hand in Postman. When you have thirty routes with validation, authentication, and error handling, manual testing falls apart:

- You forget to test an edge case after a code change
- You test the route you just modified, but not the five routes it indirectly affects
- Your teammate changes a shared middleware function and breaks your routes — nobody notices until production

Automated tests catch these problems instantly. You run `npm test`, and in seconds you know whether every endpoint in your API still works correctly.

1.2 The Java Analogy

If you used JUnit in TCSS 305 or 360, API testing follows the same idea:

JUnit (Java)	Jest + Supertest (TypeScript)
`@Test` method	`it('description', async () => { ... })`
`assertEquals(expected, actual)`	`expect(actual).toBe(expected)`
Test a method's return value	Test an HTTP response's status code and body
Run with `mvn test` or IDE	Run with `npm test`

The difference: instead of calling a method directly, you send an HTTP request to your Express app and check what comes back.

> **Gen AI & Learning: AI and Test Writing**
>
> Writing tests is one of the tasks AI coding assistants do best. If you describe what an endpoint should do, an agent can generate test cases — including edge cases you might miss. But you need to understand what a good test looks like before you can evaluate what the agent generates. Learn the patterns in this guide first, then let AI help you write more tests faster.

---

2 Types of API Tests

Not all tests are the same. Understanding the categories helps you choose the right approach.

2.1 Unit Tests

A **unit test** tests a single function in isolation — no HTTP, no middleware, no routing. For example, testing a validation function directly:

```
import { validateNumericId } from '../src/middleware/validation';

// Unit test: call the function directly, check the result
```

Unit tests are fast and focused, but they do not tell you whether your routes, middleware, and handlers work together correctly.

2.2 Integration Tests

An **integration test** tests a route end-to-end: the HTTP request goes through your middleware, hits the route handler, and you check the HTTP response. This is what Supertest does:

```
const response = await request(app).get('/v1/hello');
expect(response.status).toBe(200);
```

Integration tests are the most practical starting point for API development because they test what actually matters: "Does my API respond correctly to real HTTP requests?"

2.3 Regression Tests

A **regression test** verifies that something that used to work still works after a change. You add a new route, refactor a middleware function, or update a dependency — and your existing tests tell you immediately if you broke something.

This is the core value of automated testing: not just proving your new code works, but proving your old code still works. Without regression tests, every change is a gamble — you fix one route and silently break three others. With them, you run `npm test` and know in seconds.

In this course, your group project grows every sprint. Sprint 3 adds new routes on top of Sprint 2's routes. If Sprint 3 breaks a Sprint 2 endpoint and nobody notices until the front-end team tries to use it, that is a regression. Tests catch it before it reaches anyone else.

2.4 What This Guide Covers

This guide focuses on **integration tests with Supertest** — the approach used in the lecture demo. These tests exercise your routes, middleware, and handlers together, exactly as a real client would.

> **Note**
>
> End-to-end (E2E) tests that spin up a database, seed data, and make requests through a deployed server also exist. They are more realistic but slower and harder to set up. We do not cover them in this course.

---

3 The Testing Stack

Three packages work together to make API testing possible in TypeScript.

3.1 Jest

Jest is a test runner and assertion library. It finds your test files, runs them, and reports which tests pass or fail. Think of it as JUnit for the JavaScript/TypeScript ecosystem.

3.2 Supertest

Supertest makes HTTP requests to your Express app *without starting a real server*. Instead of calling `app.listen()` and sending requests to `localhost:3000`, Supertest binds directly to your Express app in memory. This means tests run fast and do not need an open port.

3.3 ts-jest

ts-jest is a Jest preset that lets Jest understand TypeScript files. Without it, Jest only knows JavaScript.

3.4 Installation

```
npm install --save-dev jest @types/jest supertest @types/supertest ts-jest
```

3.5 Configuration

The lecture demo uses this `jest.config.js`:

```
module.exports = {
    preset: 'ts-jest',
    testEnvironment: 'node',
    roots: ['<rootDir>/tests'],
    testMatch: ['**/*.test.ts'],
    moduleFileExtensions: ['ts', 'js', 'json'],
};
```

Option	What It Does
`preset: 'ts-jest'`	Use ts-jest to compile TypeScript
`testEnvironment: 'node'`	Run tests in a Node.js environment (not a browser)
`roots`	Where to look for test files
`testMatch`	File naming pattern for tests

---

4 The Key Insight — Testing Without a Server

This is the most important concept in API testing: **you do not start the server to run tests.**

4.1 Why app.ts and index.ts Are Separate

Look at the lecture demo's project structure:

```
// src/index.ts — starts the server (used in development/production)
import 'dotenv/config';
import { app } from './app';

const PORT = parseInt(process.env.PORT || '3000', 10);
app.listen(PORT, () => {
    console.log(`Server running at http://localhost:${PORT}`);
});
```

```
// src/app.ts — creates and configures the Express app (used by tests)
import express from 'express';
// ... middleware, routes ...
export { app };
```

The `app` is created and exported in `app.ts`. The server is started in `index.ts`. Tests import from `app.ts` — they get the fully configured Express app without calling `app.listen()`.

4.2 How Supertest Uses the App

```
import request from 'supertest';
import { app } from '../../src/app';
```

```
const response = await request(app).get('/v1/hello');
```

`request(app)` tells Supertest: "bind to this Express app and send HTTP requests to it directly." No port, no `localhost`, no network. The request goes straight from Supertest into Express's request handling pipeline.

> **Important**
>
> This separation is why the lecture demo has two files. If you put `app.listen()` in the same file as your routes, tests would start a real server every time they run — leading to port conflicts and slow tests. Always keep app creation and server startup in separate files.

---

5 What Should Your Tests Verify?

Before writing your first test, you need to answer a fundamental question: **what is the source of truth for how your API should behave?**

5.1 Test Against Documentation, Not Implementation

The answer is your **API documentation** — your Swagger/OpenAPI spec, your route descriptions, your assignment requirements. These define what the API *should* do. Your code is an attempt to implement that specification, and it might have bugs.

If you write tests by reading your implementation code and asserting what it currently does, your tests will pass — but they will also pass when the code is wrong. The tests become a mirror of your mistakes, not a check against them.

Instead, write tests from the specification:

- "The spec says `GET /users/:id` returns `200` with user data when the ID exists"
- "The spec says `GET /users/:id` returns `404` with an error message when the ID does not exist"
- "The spec says `POST /users` returns `400` when the `name` field is missing"

These tests verify the contract your API promises to fulfill. If a test fails, either the code or the documentation is wrong — both need investigation.

> **API Documentation**
>
> If you have not documented your API yet, see the OpenAPI Documentation guide. Your OpenAPI spec is both documentation for your teammates and the source of truth for your tests.

5.2 Tests Verify Your Documentation Too

This is the underappreciated benefit of testing from documentation: you are not just testing your code — you are testing your docs. If your spec says a route returns `201` but your test (written from the spec) fails because the code returns `200`, you have found a real discrepancy. Either the code needs to change or the docs need to change — but either way, something is wrong, and a client relying on your docs would be surprised.

In this course, other teams consume your API based on your Swagger documentation. If your docs say one thing and your code does another, the front-end team building against your API is going to have a bad time.

> **Gen AI & Learning: AI and Test Generation**
>
> If you use an AI agent to generate tests, **feed it your API documentation, not your source code.** An agent that reads your implementation will write tests that match what the code does — including any bugs. An agent that reads your Swagger spec will write tests that match what the API *should* do. When those spec-based tests fail, you have found a real issue.
>
> For regression testing (verifying that existing behavior does not break after changes), implementation-based tests are appropriate. But for contract verification — "does my API do what it promises?" — always start from the documentation.

---

6 Your First Test

Let's walk through the simplest test in the lecture demo — `tests/v1/hello.test.ts`:

```
import request from 'supertest';
import { app } from '../../src/app';

describe('v1 Hello Routes', () => {
    it('GET /v1/hello - returns hello message', async () => {
        const response = await request(app).get('/v1/hello');
        expect(response.status).toBe(200);
        expect(response.body.message).toBe('Hello, you sent a GET request');
    });
```

```
    it('POST /v1/hello – returns hello message with 201', async () => {
        const response = await request(app).post('/v1/hello');
        expect(response.status).toBe(201);
        expect(response.body.message).toBe('Hello, you sent a POST request');
    });
});
```

6.1 Breaking It Down

Code	Purpose	Java Equivalent
`describe('v1 Hello Routes', () => { ... })`	Group related tests together	A test class
`it('GET /v1/hello – returns hello message', ...)`	A single test case	A `@Test` method
`async () => { ... }`	Tests are async because HTTP is async	N/A (JUnit is synchronous)
`request(app).get('/v1/hello')`	Send a GET request to the route	Calling a method on an object
`expect(response.status).toBe(200)`	Assert the status code	`assertEquals(200, status)`
`expect(response.body.message).toBe(...)`	Assert the response body	`assertEquals("Hello ...", msg)`

6.2 Testing All HTTP Methods

The lecture demo tests all five methods on the same route:

```
it('PUT /v1/hello – returns hello message', async () => {
    const response = await request(app).put('/v1/hello');
    expect(response.status).toBe(200);
    expect(response.body.message).toBe('Hello, you sent a PUT request');
});

it('PATCH /v1/hello – returns hello message', async () => {
    const response = await request(app).patch('/v1/hello');
    expect(response.status).toBe(200);
    expect(response.body.message).toBe('Hello, you sent a PATCH request');
```

```
});

it('DELETE /v1/hello – returns hello message', async () => {
    const response = await request(app).delete('/v1/hello');
    expect(response.status).toBe(200);
    expect(response.body.message).toBe('Hello, you sent a DELETE request');
});
```

Each test follows the same three-step pattern: **send a request → check the status → check the body.**

> **Try It Yourself**
>
> 1. Open the lecture demo project
> 2. Run `npm test` in the terminal
> 3. Watch the output — you should see all tests pass
> 4. Break something in `controllers/hello.ts` (change a status code or message)
> 5. Run `npm test` again — notice which test fails and what the error message says
> 6. Fix it and run again

---

7 Testing Different Input Types

The `tests/v1/input.test.ts` file shows how to test routes that accept different kinds of input.

7.1 Query Strings

Append query parameters directly to the URL:

```
it('echoes query params', async () => {
    const response = await request(app).get('/v1/input/search?
q=test&limit=5');
    expect(response.status).toBe(200);
    expect(response.body.query.q).toBe('test');
    expect(response.body.query.limit).toBe('5');
});
```

Notice that `limit` comes back as the string `'5'`, not the number `5`. Query parameters are always strings — your test should reflect that.

7.2 Route Parameters

Put the parameter value directly in the URL path:

```
it('echoes the route param', async () => {
    const response = await request(app).get('/v1/input/users/42');
    expect(response.status).toBe(200);
    expect(response.body.params.id).toBe('42');
});
```

7.3 Request Body

Use `.send()` to attach a JSON body to POST, PUT, or PATCH requests:

```
it('echoes the request body', async () => {
    const response = await request(app)
        .post('/v1/input/users')
        .send({ name: 'Jane', email: 'jane@example.com' });
    expect(response.status).toBe(201);
    expect(response.body.body.name).toBe('Jane');
    expect(response.body.body.email).toBe('jane@example.com');
});
```

Supertest automatically sets the `Content-Type: application/json` header when you use `.send()` with an object.

7.4 Custom Headers

Use `.set()` to attach headers to the request:

```
it('echoes custom headers', async () => {
    const response = await request(app)
        .get('/v1/input/headers')
        .set('x-request-id', 'req-123')
        .set('x-custom-header', 'my-value');
    expect(response.status).toBe(200);
    expect(response.body.headers['x-request-id']).toBe('req-123');
    expect(response.body.headers['x-custom-header']).toBe('my-value');
});
```

---

8 Testing Validation — Happy and Sad Paths

Good tests verify that your API works correctly with valid input **and** fails correctly with invalid input. The lecture demo's v2 routes add validation middleware, so the v2 tests check both paths.

8.1 Happy Path — Valid Input

```
it('returns results when q is provided', async () => {
    const response = await request(app).get('/v2/input/search?q=test');
    expect(response.status).toBe(200);
    expect(response.body.query.q).toBe('test');
});
```

The happy path confirms the route works when the client sends correct input.

8.2 Sad Path — Invalid Input

```
it('returns 400 when q is missing', async () => {
    const response = await request(app).get('/v2/input/search');
    expect(response.status).toBe(400);
    expect(response.body.error).toMatch(/q/i);
});
```

The sad path confirms the route rejects bad input with the right status code and a helpful error message. The `toMatch(/q/i)` assertion checks that the error message mentions the missing parameter `q` (case-insensitive).

8.3 Testing Multiple Validation Failures

The user creation route validates both `name` and `email`. The tests verify each individually and together:

```
it('returns 400 when name is missing', async () => {
    const response = await request(app)
        .post('/v2/input/users')
        .send({ email: 'jane@example.com' });
    expect(response.status).toBe(400);
    expect(response.body.details).toEqual(
        expect.arrayContaining([expect.stringMatching(/name/i)])
    );
});

it('returns 400 with both errors when body is empty', async () => {
    const response = await request(app).post('/v2/input/users').send({});
    expect(response.status).toBe(400);
    expect(response.body.details).toHaveLength(2);
});
```

The empty body test checks that both validation errors are returned at once — the API does not stop at the first error.

8.4 Testing Edge Cases

The numeric ID validation is tested with several invalid values:

```
it('returns 400 when id is not numeric', async () => {
    const response = await request(app).get('/v2/input/users/abc');
    expect(response.status).toBe(400);
});

it('returns 400 when id is zero', async () => {
    const response = await request(app).get('/v2/input/users/0');
    expect(response.status).toBe(400);
});

it('returns 400 when id is negative', async () => {
    const response = await request(app).get('/v2/input/users/-1');
    expect(response.status).toBe(400);
});
```

Always Test the Sad Path

It is tempting to only write happy-path tests — "it works when I send good data." But most bugs live in the sad path: missing fields, wrong types, boundary values, empty strings. If you only test the happy path, you are only testing the easy part.

Gen AI & Learning: AI-Generated Edge Cases

AI agents are surprisingly good at identifying edge cases you might miss. If you show an agent your Swagger spec for an endpoint and ask "what edge cases should I test?", it will often suggest scenarios you did not consider — empty strings, extremely long input, special characters, concurrent requests, boundary values. Use this as a brainstorming tool, then write the tests yourself (or have the agent write them from the spec, not the code).

8.5 v1 vs. v2 — Why Both Exist

The lecture demo tests the same controllers through two sets of routes:

- **v1 tests** verify the handlers work correctly with no validation
- **v2 tests** verify the validation middleware catches bad input before the handler runs

This side-by-side comparison is a teaching tool. In your own projects, you will have one set of routes with validation — and your tests should cover both happy and sad paths for those routes.

---

9 Mocking External Dependencies

> **Looking Ahead**
>
> This section covers testing patterns for routes that call external APIs — like a proxy route that fetches data from a third-party service. If you have not built a proxy route yet, that is expected. The Proxy Pattern guide walks through the implementation. Read this section now to understand the testing patterns, and come back when you are building your proxy routes.

When your API talks to something outside its own process — a third-party HTTP service like OpenWeatherMap, or a database accessed through a client like Prisma — your tests should **not** make real calls to it. Real calls are slow, require credentials or running services, can cost money, and fail when the dependency is down. Instead, you **mock** the external dependency so your test controls exactly what it returns.

The first six subsections walk through mocking an HTTP call (`fetch`). Section 9.7 covers the same pattern applied to the Prisma database client.

9.1 Why Mock?

Real API Calls in Tests	Mocked API Calls in Tests
Slow (network latency)	Fast (in-memory)
Flaky (service might be down)	Reliable (you control the response)
Requires API keys in CI	No keys needed
Costs money (rate limits, paid tiers)	Free
Cannot test error scenarios easily	Can simulate any response

9.2 Setting Up the Mock

The lecture demo mocks `global.fetch` — the built-in function that makes HTTP requests:

```
import request from 'supertest';
import { app } from '../src/app';

// Create a mock function to replace fetch
const mockFetch = jest.fn();
global.fetch = mockFetch;

// A fake response matching OpenWeatherMap's shape
const mockWeatherResponse = {
    name: 'Seattle',
    sys: { country: 'US' },
    main: { temp: 12, feels_like: 10, temp_min: 9, temp_max: 14, humidity: 72
},
    weather: [{ description: 'light rain' }],
    wind: { speed: 5.2 },
};

beforeEach(() => {
    mockFetch.mockReset();
    process.env.WEATHER_API_KEY = 'test-api-key';
});
```

`jest.fn()` creates a mock function — a fake version of `fetch` that you control. `beforeEach()` resets the mock before every test so tests do not interfere with each other.

9.3 Simulating a Successful Response

```
it('returns full weather data on success', async () => {
    mockFetch.mockResolvedValue({
        ok: true,
        status: 200,
        json: async () => mockWeatherResponse,
    });

    const response = await request(app).get('/proxy/weather?city=Seattle');
    expect(response.status).toBe(200);
    expect(response.body.name).toBe('Seattle');
    expect(response.body.main.temp).toBe(12);

expect(mockFetch).toHaveBeenCalledWith(expect.stringContaining('q=Seattle'));
});
```

`mockResolvedValue()` tells the mock: "When someone calls `fetch()`, return this fake response." The test then sends a request to your proxy route, which calls `fetch()`, gets the mock response, and returns it to the client. Your test verifies the final HTTP response.

The last assertion — `toHaveBeenCalledWith` — verifies that your proxy actually called the external API with the right URL.

9.4 Simulating Errors

You can simulate different failure scenarios:

Upstream API returns an error (e.g., city not found):

```
it('returns upstream error status on API failure', async () => {
    mockFetch.mockResolvedValue({
        ok: false,
        status: 404,
        json: async () => ({ message: 'city not found' }),
    });

    const response = await request(app).get('/proxy/weather?
city=FakeCity123');
    expect(response.status).toBe(404);
});
```

Network failure (server unreachable):

```
it('returns 502 when fetch throws', async () => {
    mockFetch.mockRejectedValue(new Error('Network error'));

    const response = await request(app).get('/proxy/weather?city=Seattle');
    expect(response.status).toBe(502);
});
```

`mockRejectedValue()` makes the mock throw an error — simulating a network failure. Your proxy handler catches this and returns a `502 Bad Gateway`.

9.5 Testing a Transformed Response

The lecture demo's `/proxy/summary` route fetches raw weather data and reshapes it. The test verifies the exact transformed shape:

```
it('returns a simplified weather summary', async () => {
    mockFetch.mockResolvedValue({
        ok: true,
        status: 200,
        json: async () => mockWeatherResponse,
    });

    const response = await request(app).get('/proxy/summary?city=Seattle');
    expect(response.status).toBe(200);
    expect(response.body).toEqual({
        city: 'Seattle',
```

```
        country: 'US',
        temperature: {
            current: 12,
            feelsLike: 10,
            min: 9,
            max: 14,
        },
        conditions: 'light rain',
        humidity: 72,
        windSpeed: 5.2,
    });
});
```

`toEqual()` does a deep comparison — it checks every property and nested property. This is how you verify that your proxy correctly transforms the external API's response into the shape your clients expect.

9.6 Testing Middleware with Mocks

The proxy routes require an API key. The test verifies the middleware blocks requests when the key is missing:

```
it('returns 500 when WEATHER_API_KEY is not set', async () => {
    delete process.env.WEATHER_API_KEY;

    const response = await request(app).get('/proxy/weather?city=Seattle');
    expect(response.status).toBe(500);
    expect(response.body.error).toMatch(/WEATHER_API_KEY/);
});
```

By deleting the environment variable before the request, the test triggers the `requireEnvVar` middleware's error response. This is a pattern you will use whenever you need to test middleware that depends on configuration.

9.7 Mocking the Database Client

Your routes also depend on an external service — the database — through the Prisma client. Tests should not hit a real database for the same reasons they should not hit a real weather API: speed, reliability, setup cost, and the difficulty of forcing specific rows or error conditions on demand.

The mocking mechanics differ slightly from the `fetch` case. The weather proxy calls the globally-available `fetch()`, so the test replaces `global.fetch`. The Prisma client is imported from your own module, so the test replaces the whole module with `jest.mock()`.

Replacing the Prisma module:

```
import request from 'supertest';
import { app } from '../src/app';
import { prisma } from '../src/prisma';

jest.mock('../src/prisma', () => ({
    prisma: {
        message: {
            findMany: jest.fn(),
            findUnique: jest.fn(),
            create: jest.fn(),
            update: jest.fn(),
            delete: jest.fn(),
            count: jest.fn(),
        },
    },
}));

const mockMessage = prisma.message as jest.Mocked<typeof prisma.message>;

beforeEach(() => {
    jest.clearAllMocks();
});
```

`jest.mock()` intercepts the `import` for `../src/prisma` and substitutes the factory's object for the real module. Every Prisma method your route calls must be listed here as a `jest.fn()` — if a route calls `prisma.message.findMany` and the mock does not define it, the call returns `undefined` and the test fails with a confusing error. The `mockMessage` alias is just a typed handle so the test body gets autocomplete for `findMany`, `create`, etc.

`jest.clearAllMocks()` in `beforeEach` resets the call history and any queued return values between tests. Without it, state leaks from one test to the next.

Simulating a query result:

```
it('returns a message by id', async () => {
    (mockMessage.findUnique as jest.Mock).mockResolvedValueOnce({
        id: 1,
        content: 'Hello',
        author: { id: 1, username: 'jchen' },
    });

    const response = await request(app).get('/v2/messages/1');

    expect(response.status).toBe(200);
    expect(response.body.data.id).toBe(1);
});
```

`mockResolvedValueOnce()` queues a return value for the next call only — a good fit for database tests, where each test arranges the exact row (or rows) Prisma should "return" for that scenario. Pass `null` to simulate "record not found" and trigger your 404 path:

```
it('returns 404 for non-existent message', async () => {
    (mockMessage.findUnique as jest.Mock).mockResolvedValueOnce(null);

    const response = await request(app).get('/v2/messages/999');

    expect(response.status).toBe(404);
});
```

Verifying the query Prisma received:

Section 9.3 verified the URL passed to `fetch()`. The database equivalent is verifying the `where`, `data`, or `orderBy` passed to Prisma:

```
it('filters by authorId', async () => {
    (mockMessage.findMany as jest.Mock).mockResolvedValueOnce([]);
    (mockMessage.count as jest.Mock).mockResolvedValueOnce(0);

    await request(app).get('/v2/messages?authorId=5');

    expect(mockMessage.findMany).toHaveBeenCalledWith(
        expect.objectContaining({
            where: expect.objectContaining({ authorId: 5 }),
        }),
    );
});
```

`expect.objectContaining()` matches only the fields you care about instead of asserting the entire call object. The test stays focused on the behavior under test — "did the query filter by `authorId`?" — rather than tied to every incidental option (pagination, ordering, includes) that Prisma also receives.

Simulating database errors:

As with `fetch`, `mockRejectedValueOnce()` makes the mock throw — simulating a database failure:

```
it('returns 500 when the database throws', async () => {
    (mockMessage.findMany as jest.Mock).mockRejectedValueOnce(
        new Error('connection refused'),
    );

    const response = await request(app).get('/v2/messages');

    expect(response.status).toBe(500);
});
```

This lets you exercise your error-handling middleware without actually taking the database down.

---

10 Running Tests

10.1 Basic Commands

```
# Run all tests once
npm test

# Run tests in watch mode (re-runs on file changes)
npm run test:watch
```

10.2 Reading the Output

When all tests pass:

```
PASS  tests/v1/hello.test.ts
PASS  tests/v1/input.test.ts
PASS  tests/v2/hello.test.ts
PASS  tests/v2/input.test.ts
PASS  tests/proxy.test.ts

Test Suites: 5 passed, 5 total
Tests:       35 passed, 35 total
```

When a test fails, Jest shows you exactly what went wrong:

```
FAIL  tests/v1/hello.test.ts
  ● v1 Hello Routes › GET /v1/hello — returns hello message

    expect(received).toBe(expected)

    Expected: 200
    Received: 404

      5 |   it('GET /v1/hello — returns hello message', async () => {
      6 |     const response = await request(app).get('/v1/hello');
    > 7 |     expect(response.status).toBe(200);
      8 |     expect(response.body.message).toBe('Hello, you sent a GET
request');
```

The failure message tells you: which test failed, what you expected, what you actually got, and the exact line in your test file. This is your starting point for debugging.

Gen AI & Learning: Using AI to Debug Failing Tests

When a test fails and you cannot figure out why, paste the test code, the error output, and the relevant route handler into your AI coding agent. Ask it to explain why the test is failing. The agent can often spot the mismatch — a wrong status code, a missing field in the response body, a middleware that is not being applied — faster than you can by reading the code yourself. This is one of the highest-value uses of an AI agent: debugging, not generating.

10.3 Tests in CI

The lecture demo runs tests automatically on every pull request via GitHub Actions. If a test fails, the PR cannot be merged. This ensures that broken code never reaches the main branch.

```
# .github/workflows/ci.yml (simplified)
- run: npm test
```

This is the same `npm test` command you run locally — CI just runs it automatically.

---

11 Summary

Concept	Key Point
Why test?	Automated tests verify your API contract and catch regressions
Integration tests	Test routes end-to-end: request → middleware → handler → response
Jest	Test runner and assertion library (`describe`, `it`, `expect`)
Supertest	Sends HTTP requests to Express without starting a server
`app.ts` vs `index.ts`	Separate app creation from server startup so tests can import the app
`request(app).get(url)`	Supertest sends a request directly to the Express app

Concept	Key Point
`.send()`	Attach a JSON body to the request
`.set()`	Attach headers to the request
Happy path	Test that valid input produces the expected response
Sad path	Test that invalid input produces the correct error
Mocking	Replace external API calls with controlled fake responses
`jest.fn()`	Create a mock function
`mockResolvedValue()`	Mock returns a successful response
`mockRejectedValue()`	Mock throws an error (simulates network failure)
`beforeEach()`	Reset mocks between tests to prevent interference

---

12 References

Official Documentation:

- Jest — Getting Started — Installation and first test
- Jest — Expect API — All assertion methods (`toBe` , `toEqual` , `toMatch` , etc.)
- Jest — Mock Functions — `jest.fn()` , `mockResolvedValue` , `mockRejectedValue`
- Supertest — GitHub — API reference and examples
- ts-jest — Documentation — TypeScript support for Jest

TCSS 460 Lecture Demo

github.com/UWT-SET-TCSS460-LECTURE-MATERIALS/TCSS-460-Backend-1 — Full test suite in `tests/`

---

13 Further Reading

External Resources

- Jest — Using Matchers — Complete guide to Jest's assertion methods
- Jest — Setup and Teardown — `beforeEach`, `afterEach`, `beforeAll`, `afterAll`
- Express.js — Testing — Official Express testing guide

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