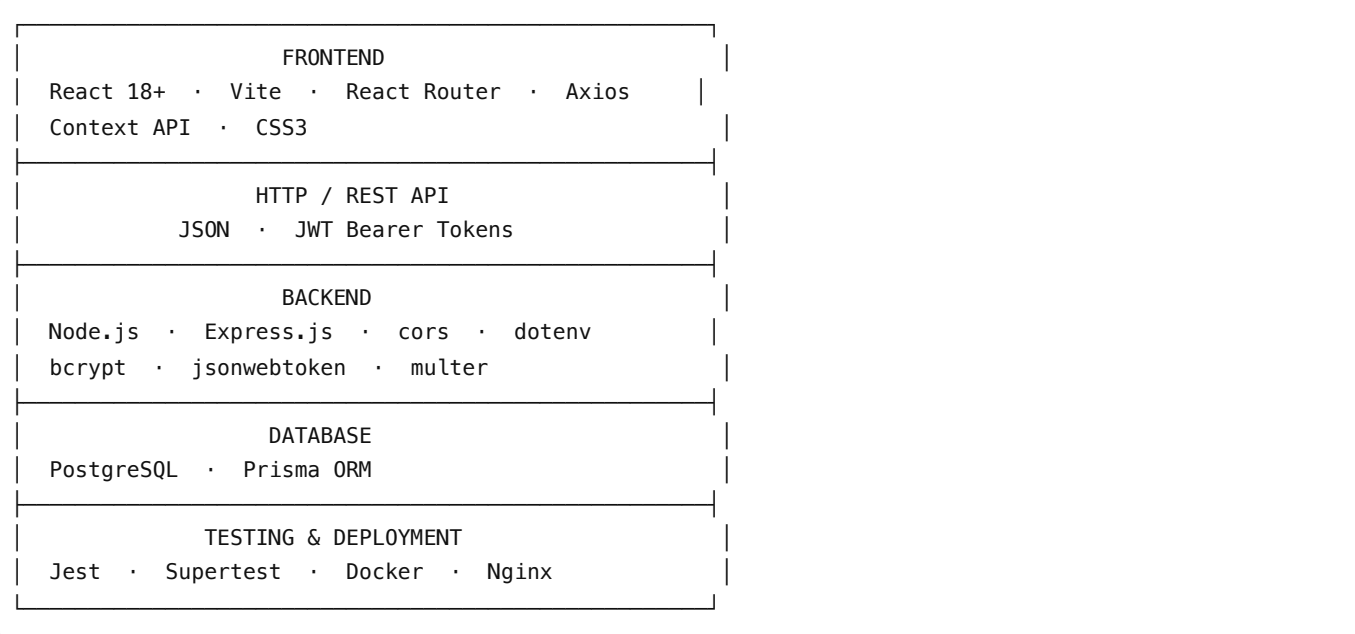
Full-Stack Web Development: Technology Reference Guide

Course: Full-Stack Web Application Development **Stack:** React 18+ · Node.js · Express.js · PostgreSQL · Prisma ORM **Duration:** 4 Weeks — 12 Sessions — 48 Contact Hours

---

Course Architecture Overview



Weekly Roadmap

Week	Sessions	Focus
1	1–3	React components, state, forms, routing, Express API basics
2	4–6	PostgreSQL, Prisma ORM, CRUD refactor, authentication
3	7–9	Frontend auth, full-stack integration, file uploads
4	10–12	Testing, Docker deployment, capstone presentations

Frontend Technologies

React 18+

What it is: A JavaScript library for building user interfaces using a component-based architecture.

Why we use it: React breaks UIs into reusable components, manages state efficiently with hooks, and updates the DOM automatically when data changes.

When it's introduced: Session 1

Core Concepts

Components — Reusable building blocks of the UI:

```
function TaskCard({ title, status, onToggle }) {
  return (
    <div className={`task-card task-${status}`}>
      <h3>{title}</h3>
      <span className="badge">{status}</span>
      <button onClick={onToggle}>Toggle</button>
    </div>
  );
}
```

useState — Manages local component state:

```
import { useState } from 'react';

function Counter({ startValue = 0 }) {
  const [count, setCount] = useState(startValue);

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>+</button>
      <button onClick={() => setCount(count - 1)}>−</button>
    </div>
  );
}
```

useEffect — Runs side effects (API calls, timers, subscriptions):

```
import { useState, useEffect } from 'react';

function TaskList() {
  const [tasks, setTasks] = useState([]);
  const [isLoading, setIsLoading] = useState(true);

  useEffect(() => {
```

```
    const fetchTasks = async () => {
      try {
        const { data } = await api.get('/tasks');
        setTasks(data);
      } catch (err) {
        console.error('Failed to fetch tasks:', err);
      } finally {
        setIsLoading(false);
      }
    };
    fetchTasks();
  }, []); // Empty array = run once on mount

  if (isLoading) return <p>Loading...</p>;
  return tasks.map(t => <TaskCard key={t.id} task={t} />);
}
```

Context API — Shares state globally without prop drilling:

```
import { createContext, useContext, useState } from 'react';

const AuthContext = createContext(null);

export function AuthProvider({ children }) {
  const [user, setUser] = useState(null);
  const [token, setToken] = useState(null);

  const login = async (email, password) => { /* ... */ };
  const logout = () => { /* ... */ };

  return (
    <AuthContext.Provider value={{ user, token, login, logout }}>
      {children}
    </AuthContext.Provider>
  );
}

export const useAuth = () => useContext(AuthContext);
```

Vite 5+

What it is: A fast build tool and development server for modern web applications.

Why we use it: Vite provides instant hot module replacement (HMR), fast builds, and zero-config React support. It replaces older tools like Create React App.

When it's introduced: Session 1

Usage

Create a project:

```
npm create vite@latest my-app -- --template react
cd my-app
npm install
npm run dev
```

Project structure:

```
my-app/
  index.html          ← Entry HTML (Vite serves this)
  vite.config.js      ← Configuration
  src/
    main.jsx          ← JavaScript entry point
    App.jsx           ← Root React component
    App.css           ← Styles
```

Configuration with API proxy (for development):

```
// vite.config.js
import { defineConfig } from 'vite';
import react from '@vitejs/plugin-react';

export default defineConfig({
  plugins: [react()],
  server: {
    proxy: {
      '/api': {
        target: 'http://localhost:3001',
        changeOrigin: true,
      },
    },
  },
});
```

Build for production:

```
npm run build    # Outputs to dist/
npm run preview  # Preview the production build locally
```

React Router v6

What it is: A client-side routing library for React that enables navigation between pages without full page reloads.

Why we use it: Single-page applications need client-side routing to show different views based on the URL.

When it's introduced: Session 2

Usage

```
import { BrowserRouter, Routes, Route, Link, Navigate } from 'react-router-dom';

function App() {
  return (
    <BrowserRouter>
      <nav>
        <Link to="/">Home</Link>
        <Link to="/tasks">Tasks</Link>
        <Link to="/profile">Profile</Link>
      </nav>

      <Routes>
        <Route path="/" element={<HomePage />} />
        <Route path="/login" element={<LoginPage />} />
        <Route path="/register" element={<RegisterPage />} />
        <Route path="/tasks" element={
          <PrivateRoute><TasksPage /></PrivateRoute>
        } />
        <Route path="/profile" element={
          <PrivateRoute><ProfilePage /></PrivateRoute>
        } />
        <Route path="*" element={<Navigate to="/" />} />
      </Routes>
    </BrowserRouter>
  );
}
```

Protected Route pattern:

```
function PrivateRoute({ children }) {
  const { user, isLoading } = useAuth();

  if (isLoading) return <p>Loading...</p>;
  if (!user) return <Navigate to="/login" replace />;
  return children;
}
```

Axios

What it is: A promise-based HTTP client for making API requests from the browser.

Why we use it: Axios provides interceptors for automatic token injection and error handling, automatic JSON parsing, and a cleaner API than native `fetch`.

When it's introduced: Session 8

Usage

Create an API client with interceptors:

```
// src/api/client.js
import axios from 'axios';

const api = axios.create({
  baseURL: '/api',
});

// Automatically attach JWT token to every request
api.interceptors.request.use((config) => {
  const token = localStorage.getItem('token');
  if (token) {
    config.headers.Authorization = `Bearer ${token}`;
  }
  return config;
});

// Handle 401 responses (auto-logout)
api.interceptors.response.use(
  (response) => response,
  (error) => {
    if (error.response?.status === 401) {
      localStorage.removeItem('token');
      localStorage.removeItem('user');
      window.location.href = '/login';
    }
    return Promise.reject(error);
  }
);

export default api;
```

Making requests:

```
// GET
const { data: tasks } = await api.get('/tasks');

// GET with query params
const { data } = await api.get('/tasks', {
  params: { status: 'pending' }
```

```
});

// POST
const { data: newTask } = await api.post('/tasks', {
  title: 'Build login page',
  priority: 'high',
});

// PUT
await api.put(`/tasks/${id}`, { status: 'completed' });

// DELETE
await api.delete(`/tasks/${id}`);

// File upload
const formData = new FormData();
formData.append('avatar', file);
await api.post('/users/avatar', formData, {
  headers: { 'Content-Type': 'multipart/form-data' },
});
```

Backend Technologies

Node.js 20+ LTS

What it is: A JavaScript runtime built on Chrome's V8 engine that lets you run JavaScript on the server.

Why we use it: Node.js enables full-stack JavaScript — the same language on frontend and backend. Its event-driven, non-blocking I/O model handles concurrent connections efficiently.

When it's introduced: Session 3

Key Concepts

- **npm** — Package manager for installing dependencies
- **package.json** — Project manifest with dependencies and scripts
- **CommonJS modules** — `require()` and `module.exports` (used in this course)
- **Event loop** — Non-blocking I/O for handling many requests simultaneously

Usage

```
# Initialize a project
npm init -y

# Install dependencies
npm install express cors dotenv

# Install dev dependencies
npm install --save-dev prisma jest supertest

# Run scripts
npm start          # node src/app.js
npm run dev        # node --watch src/app.js
npm test           # jest --runInBand --forceExit
```

Express.js 4.x

What it is: A minimal, flexible Node.js web framework for building APIs and web applications.

Why we use it: Express provides routing, middleware support, and a clean structure for handling HTTP requests. It's the most widely used Node.js framework.

When it's introduced: Session 3

Core Concepts

Server setup:

```
const express = require('express');
const cors = require('cors');
const app = express();
```

```
// Middleware (order matters!)
app.use(cors());                    // Enable cross-origin requests
app.use(express.json());            // Parse JSON request bodies
app.use(express.static('uploads')); // Serve static files

// Routes
app.use('/api/auth', authRoutes);
app.use('/api/tasks', authenticate, taskRoutes);
app.use('/api/users', authenticate, userRoutes);

// Error handling (must be last)
app.use(notFoundHandler);
app.use(errorHandler);

const PORT = process.env.PORT || 3001;
app.listen(PORT, () => console.log(`Server running on port ${PORT}`));
```

Route definitions:

```
const router = require('express').Router();

// GET all tasks (with optional query filter)
router.get('/', async (req, res, next) => {
  try {
    const where = { userId: req.userId };
    if (req.query.status) where.status = req.query.status;

    const tasks = await prisma.task.findMany({ where });
    res.json(tasks);
  } catch (err) {
    next(err);
  }
});

// POST new task
router.post('/', async (req, res, next) => {
  try {
    const { title, description, status, priority } = req.body;
    if (!title) return res.status(400).json({ error: 'Title is required' });

    const task = await prisma.task.create({
      data: {
        title,
        description,
        status: status || 'pending',
        priority: priority || 'medium',
        userId: req.userId,
      },
    });
    res.status(201).json(task);
  } catch (err) {
    next(err);
```

```
  }
});

module.exports = router;
```

Middleware pattern:

```
// Middleware is a function with (req, res, next)
function logger(req, res, next) {
  const start = Date.now();
  res.on('finish', () => {
    const duration = Date.now() - start;
    console.log(`${req.method} ${req.path} ${res.statusCode} ${duration}ms`);
  });
  next(); // Pass control to the next middleware
}

// Error-handling middleware has 4 parameters: (err, req, res, next)
function errorHandler(err, req, res, next) {
  console.error(err.stack);
  res.status(err.status || 500).json({
    error: err.message || 'Internal server error',
  });
}
```

HTTP Methods and Status Codes

Method	Purpose	Example	Success Code
`GET`	Read data	`GET /api/tasks`	200 OK
`POST`	Create data	`POST /api/tasks`	201 Created
`PUT`	Update data	`PUT /api/tasks/5`	200 OK
`DELETE`	Remove data	`DELETE /api/tasks/5`	204 No Content

Status	Meaning	When to use
200	OK	Successful GET, PUT
201	Created	Successful POST
204	No Content	Successful DELETE
400	Bad Request	Invalid input / validation failure
401	Unauthorized	Missing or invalid auth token
403	Forbidden	Valid token but not allowed
404	Not Found	Resource doesn't exist
409	Conflict	Duplicate (email already registered)

500	Server Error	Unexpected error

cors

What it is: Express middleware that enables Cross-Origin Resource Sharing — allowing your frontend (on port 5173) to make requests to your backend (on port 3001).

Why we use it: Browsers block cross-origin requests by default for security. CORS headers tell the browser which origins are allowed.

When it's introduced: Session 3

Usage

```
const cors = require('cors');

// Development: allow all origins
app.use(cors());

// Production: whitelist specific origins
app.use(cors({
  origin: ['http://localhost:5173', 'https://your-app.com'],
  methods: ['GET', 'POST', 'PUT', 'DELETE'],
  allowedHeaders: ['Content-Type', 'Authorization'],
}));
```

---

dotenv

What it is: A module that loads environment variables from a `.env` file into `process.env`.

Why we use it: Keeps secrets (database URLs, JWT keys) out of source code. Different environments (dev, test, production) use different `.env` files.

When it's introduced: Session 3

Usage

```
# .env (never commit this file!)
DATABASE_URL=postgresql://postgres:postgres@localhost:5432/taskflow
JWT_SECRET=your-super-secret-key-at-least-32-chars
PORT=3001
NODE_ENV=development
```

```
require('dotenv').config(); // Load at the top of your entry file

const PORT = process.env.PORT || 3001;
const secret = process.env.JWT_SECRET;
```

---

bcrypt

What it is: A library for securely hashing passwords using the bcrypt algorithm.

Why we use it: Passwords must never be stored as plain text. bcrypt adds a random salt and is intentionally slow, making brute-force attacks impractical.

When it's introduced: Session 6

Usage

```
const bcrypt = require('bcrypt');
const SALT_ROUNDS = 10; // Higher = slower but more secure

// Registration: hash the password before storing
const hashedPassword = await bcrypt.hash(password, SALT_ROUNDS);
await prisma.user.create({
  data: { email, username, password: hashedPassword },
});

// Login: compare the submitted password with the stored hash
const user = await prisma.user.findUnique({ where: { email } });
const isMatch = await bcrypt.compare(password, user.password);
if (!isMatch) {
  return res.status(401).json({ error: 'Invalid email or password' });
}
```

How It Works

```
password: "mypass123"
     ↓
bcrypt.hash(password, 10)
     ↓
"$2b$10$N9qo8uLOickgx2ZMRZoMy..."  ← stored in database
     ↓
bcrypt.compare("mypass123", hash)  → true
bcrypt.compare("wrongpass", hash)  → false
```

- The salt is embedded in the hash — no separate storage needed
- Same password produces different hashes each time (due to random salt)
- `compare()` extracts the salt from the hash to verify

---

jsonwebtoken (JWT)

What it is: A library for creating and verifying JSON Web Tokens — compact, signed tokens used for authentication.

Why we use it: After login, the server issues a JWT. The client sends it with every request to prove identity without re-sending credentials.

When it's introduced: Session 6

Usage

```
const jwt = require('jsonwebtoken');
const SECRET = process.env.JWT_SECRET;

// After login: create a token
const token = jwt.sign(
  { userId: user.id },     // Payload (claims)
  SECRET,                   // Secret key
  { expiresIn: '24h' }     // Options
);

// Middleware: verify the token on protected routes
function authenticate(req, res, next) {
  const authHeader = req.headers.authorization;
  if (!authHeader?.startsWith('Bearer ')) {
    return res.status(401).json({ error: 'No token provided' });
  }

  try {
    const token = authHeader.split(' ')[1];
    const decoded = jwt.verify(token, SECRET);
    req.userId = decoded.userId;  // Attach to request
    next();
  } catch (err) {
    return res.status(401).json({ error: 'Invalid or expired token' });
  }
}
```

JWT Structure

```
eyJhbGciOiJIUzI1NiJ9.eyJ1c2VySWQiOjF9.abc123...
|         Header        |     Payload      | Signature |

Header:    { "alg": "HS256", "typ": "JWT" }
Payload:   { "userId": 1, "iat": 1708200000, "exp": 1708286400 }
Signature: HMACSHA256(header + "." + payload, secret)
```

---

multer

What it is: Express middleware for handling `multipart/form-data`, used for file uploads.

Why we use it: Standard `express.json()` can't handle file uploads. Multer processes uploaded files, saves them to disk, and provides metadata on `req.file`.

When it's introduced: Session 9

Usage

```
const multer = require('multer');
const path = require('path');

// Configure where and how files are saved
const storage = multer.diskStorage({
  destination: (req, file, cb) => cb(null, 'uploads/'),
  filename: (req, file, cb) => {
    const uniqueName = Date.now() + '-' + file.originalname;
    cb(null, uniqueName);
  },
});

// Only allow image files
const fileFilter = (req, file, cb) => {
  const allowed = ['image/jpeg', 'image/png', 'image/gif', 'image/webp'];
  cb(null, allowed.includes(file.mimetype));
};

const upload = multer({
  storage,
  fileFilter,
  limits: { fileSize: 5 * 1024 * 1024 }, // 5MB max
});

// Upload route
router.post('/avatar', authenticate, upload.single('avatar'),
  async (req, res) => {
    // req.file = { filename, path, mimetype, size }
    await prisma.user.update({
      where: { id: req.userId },
      data: { avatarUrl: `/uploads/${req.file.filename}` },
    });
    res.json({ avatarUrl: `/uploads/${req.file.filename}` });
  }
);

// Serve uploaded files
app.use('/uploads', express.static('uploads'));
```

Database Technologies

PostgreSQL 15+

What it is: A powerful, open-source relational database management system.

Why we use it: PostgreSQL is production-grade, supports complex queries and relationships, has strong data integrity with constraints, and is widely used in the industry.

When it's introduced: Session 4

Core SQL

Create tables:

```
CREATE TABLE users (
  id         SERIAL PRIMARY KEY,
  username   VARCHAR(50) UNIQUE NOT NULL,
  email      VARCHAR(100) UNIQUE NOT NULL,
  password   VARCHAR(255) NOT NULL,
  avatar_url VARCHAR(255),
  created_at TIMESTAMP DEFAULT NOW()
);

CREATE TABLE tasks (
  id          SERIAL PRIMARY KEY,
  title       VARCHAR(255) NOT NULL,
  description TEXT,
  status      VARCHAR(20) DEFAULT 'pending'
                CHECK (status IN ('pending', 'in-progress', 'completed')),
  priority    VARCHAR(10) DEFAULT 'medium'
                CHECK (priority IN ('low', 'medium', 'high')),
  due_date    TIMESTAMP,
  user_id     INTEGER NOT NULL REFERENCES users(id) ON DELETE CASCADE,
  created_at  TIMESTAMP DEFAULT NOW(),
  updated_at  TIMESTAMP DEFAULT NOW()
);
```

Common queries:

```
-- Select with filter
SELECT * FROM tasks WHERE status = 'pending' AND user_id = 1;

-- Join tables
SELECT t.title, t.status, u.username
FROM tasks t
JOIN users u ON t.user_id = u.id;

-- Count by status
SELECT status, COUNT(*) FROM tasks GROUP BY status;
```

```sql
-- Update
UPDATE tasks SET status = 'completed' WHERE id = 5;

-- Delete
DELETE FROM tasks WHERE id = 5;
```

psql Commands

Command	What it does
`psql -U postgres`	Connect to PostgreSQL
`\l`	List all databases
`\c taskflow`	Connect to a database
`\dt`	List tables
`\d tasks`	Describe a table's columns
`\q`	Quit psql

Prisma 5.x

What it is: A modern ORM (Object-Relational Mapping) toolkit that provides type-safe database access, schema management, and migrations.

Why we use it: Prisma replaces raw SQL with a JavaScript API, handles migrations automatically, prevents SQL injection, and makes database operations more intuitive.

When it's introduced: Session 4 (schema), Session 5 (CRUD)

Schema Definition

```
// prisma/schema.prisma
datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}

generator client {
  provider = "prisma-client-js"
}

model User {
  id        Int      @id @default(autoincrement())
  username  String   @unique @db.VarChar(50)
  email     String   @unique @db.VarChar(100)
  password  String   @db.VarChar(255)
  avatarUrl String?  @map("avatar_url")
  createdAt DateTime @default(now()) @map("created_at")
  tasks     Task[]   // One-to-many relation

  @@map("users")
}

model Task {
  id          Int       @id @default(autoincrement())
  title       String    @db.VarChar(255)
  description String?
  status      String    @default("pending") @db.VarChar(20)
  priority    String    @default("medium") @db.VarChar(10)
  dueDate     DateTime? @map("due_date")
  userId      Int       @map("user_id")
  user        User      @relation(fields: [userId], references: [id])
  createdAt   DateTime  @default(now()) @map("created_at")
  updatedAt   DateTime  @updatedAt @map("updated_at")

  @@map("tasks")
}
```

CLI Commands

```
npx prisma init              # Create prisma/ directory and .env
npx prisma migrate dev       # Create and apply migration
npx prisma generate          # Generate Prisma Client
npx prisma studio            # Visual database browser (localhost:5555)
npx prisma db seed           # Run seed script
```

CRUD Operations

```
const { PrismaClient } = require('@prisma/client');
const prisma = new PrismaClient();

// CREATE
const task = await prisma.task.create({
  data: { title: 'Build API', status: 'pending', userId: 1 },
});

// READ (many)
const tasks = await prisma.task.findMany({
  where: { userId: 1, status: 'pending' },
  orderBy: { createdAt: 'desc' },
});

// READ (one)
const task = await prisma.task.findUnique({
  where: { id: 5 },
  include: { user: true }, // Join related user
});

// UPDATE
const updated = await prisma.task.update({
  where: { id: 5 },
  data: { status: 'completed' },
});

// DELETE
await prisma.task.delete({ where: { id: 5 } });

// COUNT
const count = await prisma.task.count({
  where: { userId: 1, status: 'completed' },
});
```

Testing Technologies

Jest 29.x

What it is: A JavaScript testing framework with built-in assertions, mocking, and code coverage.

Why we use it: Jest is zero-config for Node.js, runs tests in parallel, provides clear error messages, and includes everything needed for both unit and integration testing.

When it's introduced: Session 10

Usage

Test structure:

```
describe('isValidEmail', () => {
  it('should accept valid email addresses', () => {
    expect(isValidEmail('user@example.com')).toBe(true);
    expect(isValidEmail('test.name@domain.org')).toBe(true);
  });

  it('should reject invalid email addresses', () => {
    expect(isValidEmail('not-an-email')).toBe(false);
    expect(isValidEmail('@missing-local.com')).toBe(false);
    expect(isValidEmail('')).toBe(false);
  });

  it('should handle edge cases', () => {
    expect(isValidEmail(null)).toBe(false);
    expect(isValidEmail(undefined)).toBe(false);
    expect(isValidEmail(42)).toBe(false);
  });
});
```

Common matchers:

Matcher	Use Case
`toBe(value)`	Exact equality (primitives)
`toEqual(obj)`	Deep equality (objects/arrays)
`toBeTruthy()` / `toBeFalsy()`	Truthy/falsy checks
`toContain(item)`	Array contains element
`toHaveLength(n)`	Array or string length
`toMatchObject(obj)`	Partial object match
`toHaveProperty(key)`	Object has property

`toThrow()`	Function throws an error

Lifecycle hooks:

```
beforeAll(async () => { /* Run once before all tests */ });
afterAll(async () => { /* Run once after all tests */ });
beforeEach(async () => { /* Run before each test */ });
afterEach(async () => { /* Run after each test */ });
```

Run commands:

```
npm test                    # Run all tests
npm run test:watch          # Re-run on file changes
npm run test:coverage       # Generate coverage report
npx jest validators.test.js # Run a specific file
```

Supertest 6.x

What it is: An HTTP assertion library for testing Express.js APIs without starting a real server.

Why we use it: Supertest makes HTTP requests to your Express app in-process, so you can test your API endpoints with real request/response cycles — no server port needed.

When it's introduced: Session 10

Usage

```
const request = require('supertest');
const app = require('../app'); // Import Express app (not listening)

describe('POST /api/auth/register', () => {
  it('should register a new user', async () => {
    const res = await request(app)
      .post('/api/auth/register')
      .send({
        username: 'testuser',
        email: 'test@example.com',
        password: 'password123',
      });

    expect(res.status).toBe(201);
    expect(res.body).toHaveProperty('token');
    expect(res.body.user.email).toBe('test@example.com');
    expect(res.body.user).not.toHaveProperty('password');
  });
});

describe('GET /api/tasks', () => {
  it('should return tasks for authenticated user', async () => {
    const res = await request(app)
```

```
      .get('/api/tasks')
      .set('Authorization', `Bearer ${token}`);

    expect(res.status).toBe(200);
    expect(Array.isArray(res.body)).toBe(true);
  });

  it('should return 401 without token', async () => {
    const res = await request(app).get('/api/tasks');
    expect(res.status).toBe(401);
  });
});
```

Key Pattern: Export App Without Listening

```
// src/app.js
const app = express();
// ... routes and middleware ...

// Only listen if this file is run directly (not imported by tests)
if (require.main === module) {
  app.listen(PORT, () => console.log(`Server on port ${PORT}`));
}

module.exports = app; // Export for Supertest
```

Deployment Technologies

Docker & Docker Compose

What it is: Docker packages applications into containers — isolated environments that include everything needed to run. Docker Compose orchestrates multiple containers as one application.

Why we use it: Containers ensure "it works on my machine" isn't a problem. Docker Compose lets you run the database, backend, and frontend together with one command.

When it's introduced: Session 11

Backend Dockerfile

```
FROM node:20-slim
WORKDIR /app

# Install dependencies first (layer caching)
COPY package*.json ./
RUN npm ci --production

# Copy application code
COPY . .

# Generate Prisma Client
RUN npx prisma generate

EXPOSE 3001
CMD ["node", "src/app.js"]
```

Frontend Dockerfile (Multi-Stage Build)

```
# Stage 1: Build the React app
FROM node:20-slim AS build
WORKDIR /app
COPY package*.json ./
RUN npm ci
COPY . .
RUN npm run build

# Stage 2: Serve with Nginx (tiny image)
FROM nginx:alpine
COPY --from=build /app/dist /usr/share/nginx/html
COPY nginx.conf /etc/nginx/conf.d/default.conf
EXPOSE 80
```

Docker Compose

```yaml
version: '3.8'
services:
  db:
    image: postgres:15-alpine
    environment:
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: postgres
      POSTGRES_DB: taskflow
    ports: ['5432:5432']
    volumes: [pgdata:/var/lib/postgresql/data]
    healthcheck:
      test: ['CMD-SHELL', 'pg_isready -U postgres']
      interval: 5s
      timeout: 5s
      retries: 5

  backend:
    build: ./backend
    ports: ['3001:3001']
    environment:
      DATABASE_URL: postgresql://postgres:postgres@db:5432/taskflow
      JWT_SECRET: change-this-to-a-secure-random-string
      PORT: 3001
    depends_on:
      db:
        condition: service_healthy

  frontend:
    build: ./frontend
    ports: ['8080:80']
    depends_on: [backend]

volumes:
  pgdata:
```

Commands

```
docker compose up --build       # Build and start everything
docker compose up --build -d    # Run in background
docker compose logs -f          # Follow logs
docker compose down             # Stop containers
docker compose down -v          # Stop and delete all data
docker compose exec backend npx prisma migrate deploy  # Run migrations
```

Nginx

What it is: A high-performance web server used to serve the production React build and handle SPA routing.

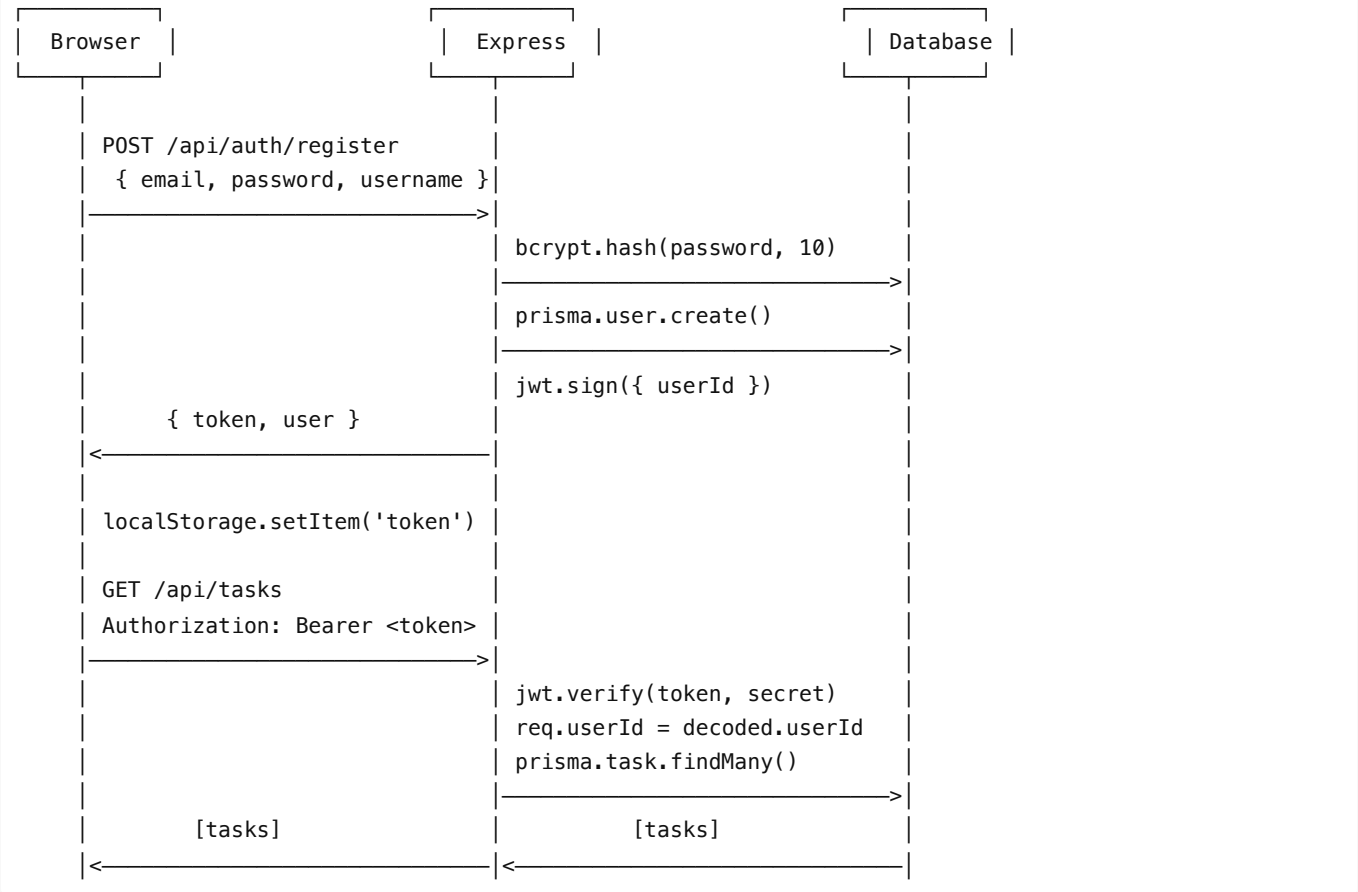
Why we use it: After `npm run build`, React outputs static files. Nginx serves these files efficiently and redirects all routes to `index.html` for client-side routing to work.

When it's introduced: Session 11

Configuration

```
server {
    listen 80;
    root /usr/share/nginx/html;
    index index.html;

    # SPA routing: send all routes to index.html
    location / {
        try_files $uri $uri/ /index.html;
    }

    # Cache static assets (JS, CSS, images)
    location /assets/ {
        expires 1y;
        add_header Cache-Control "public, immutable";
    }

    # Don't cache index.html (always serve latest version)
    location = /index.html {
        add_header Cache-Control "no-cache";
    }

    # Security headers
    add_header X-Content-Type-Options "nosniff";
    add_header X-Frame-Options "DENY";

    # Gzip compression
    gzip on;
    gzip_types text/plain text/css application/json application/javascript;
}
```

Complete Authentication Flow



Quick Reference: All Technologies

Technology	Category	Version	Purpose	Session
React	Frontend	18+	UI components and state management	1
Vite	Frontend	5+	Build tool and dev server	1
React Router	Frontend	6+	Client-side page routing	2
Axios	Frontend	1.x	HTTP client with interceptors	8
Node.js	Runtime	20+ LTS	JavaScript server runtime	3
Express.js	Backend	4.x	Web framework and routing	3
cors	Backend	2.x	Cross-origin request handling	3
dotenv	Backend	16.x	Environment variable management	3
bcrypt	Backend	5.x	Password hashing	6

jsonwebtoken	Backend	9.x	JWT authentication tokens	6
multer	Backend	1.x	File upload processing	9
PostgreSQL	Database	15+	Relational database	4
Prisma	Database	5.x	ORM and migration toolkit	4
Jest	Testing	29.x	Test runner and assertions	10
Supertest	Testing	6.x	HTTP API testing	10
Docker	Deployment	—	Application containerization	11
Nginx	Deployment	—	Static file server for production	11
