

# Day 2 Lab: Claude Code Fundamentals

---

**Duration:** 90 minutes | **Prerequisites:** Terminal/CLI basics, Node.js installed

**Objective:** Install Claude Code, build a real project from scratch, and master core workflows.

---

## Pre-Lab Setup

---

Before starting, ensure you have: - [ ] Node.js 18+ installed ( `node --version` ) - [ ] A terminal (VS Code terminal, iTerm2, Windows Terminal) - [ ] An Anthropic API key or Claude Max subscription - [ ] Git installed ( `git --version` )

---

## Exercise 1: Installation & First Magic Moment (15 min)

---

### Step 1: Install Claude Code

```
npm install -g @anthropic-ai/claude-code
```

Verify the installation:

```
claude --version
```

**Expected output:** Version number (e.g., `1.x.x` )

### Step 2: Authenticate

```
claude
```

Follow the prompts to authenticate. Choose your preferred method: - **Anthropic API key** — paste your key when prompted - **Claude Max** — sign in with your browser

### Step 3: Your First Prompt

Create a working directory and launch Claude Code:

```
mkdir todo-app && cd todo-app
claude
```

Now type this prompt:

```
Create a simple Express.js REST API for a todo app with these endpoints:
- GET /todos - list all todos
- POST /todos - create a new todo
- PUT /todos/:id - update a todo
- DELETE /todos/:id - delete a todo

Use an in-memory array for storage. Include proper error handling and status co
```

**Watch what happens.** Claude Code will: 1. Create `package.json` with dependencies 2. Create `server.js` with full API code 3. Run `npm install` automatically 4. The app is ready to test

### Step 4: Test Your API

Open a new terminal tab and test:

```
# List todos (empty)
curl http://localhost:3000/todos

# Create a todo
curl -X POST http://localhost:3000/todos \
  -H "Content-Type: application/json" \
  -d '{"title": "Learn Claude Code", "completed": false}'

# List todos (should show your new todo)
curl http://localhost:3000/todos

# Update it
```

```
curl -X PUT http://localhost:3000/todos/1 \
  -H "Content-Type: application/json" \
  -d '{"completed": true}'

# Delete it
curl -X DELETE http://localhost:3000/todos/1
```

**What just happened?** Claude Code analyzed your request, chose the right tools (file creation, npm install), wrote production-quality code, and set up the project — all from a single natural language prompt. This is the agentic loop in action.

❑ **Checkpoint:** You should have a running Express API with 4 working endpoints.

---

## Exercise 2: CLAUDE.md Mastery (15 min)

---

### Step 1: Create Your CLAUDE.md

In the `todo-app` directory, ask Claude Code:

```
Create a CLAUDE.md file for this project. Include:
- Project overview
- Tech stack (Express.js, Node.js)
- Coding conventions: use const/let (no var), semicolons required, single quote
- File structure description
- How to run and test the project
```

### Step 2: Review the CLAUDE.md

```
cat CLAUDE.md
```

**Expected:** A well-structured markdown file documenting your project.

### Step 3: Test Convention Enforcement

Now ask Claude Code to add a feature:

```
Add a GET /todos/stats endpoint that returns: total count, completed count, and pending count
```

**Check the code.** Does it follow your conventions? - Single quotes? ✓ - Semicolons? ✓ - const/let (no var)? ✓ - 2-space indentation? ✓

### Step 4: Compare Without CLAUDE.md

Try this experiment:

```
mkdir ../test-no-claude && cd ../test-no-claude
claude
```

Ask the same question:

```
Create a stats endpoint for a todo API that returns total, completed, and pending count
```

**Notice the difference.** Without CLAUDE.md, Claude Code has no project context — it might use different conventions, different variable names, different structure.

```
cd ../todo-app
```

**What just happened?** CLAUDE.md acts as persistent project memory. It tells Claude Code your preferences, conventions, and context — so every interaction is consistent with your project's style.

☐ **Checkpoint:** You have a CLAUDE.md that enforces coding conventions.

---

## Exercise 3: Context Management & @ Mentions (15 min)

---

### Step 1: Reference Specific Files

Back in your `todo-app`, try:

```
Look at @server.js and add request logging middleware that logs: timestamp, me
```

**Notice:** The `@server.js` tells Claude Code exactly which file to focus on, reducing token usage and improving accuracy.

### Step 2: Reference Multiple Files

```
Look at @server.js and @package.json – add a test script using Jest and write
```

Claude Code will: 1. Read both files for context 2. Add Jest to package.json 3. Create a test file 4. Write meaningful tests

### Step 3: Verify Tests Pass

```
Run the tests and fix any failures.
```

### Step 4: Practice Context Efficiency

Try this comparison:

**Broad (expensive):**

```
Add input validation to the API.
```

**Focused (efficient):**

```
In @server.js, add input validation to the POST /todos endpoint: title must be
```

**What just happened?** The @ mention system gives you precise control over context. More specific context = better output + fewer tokens. Think of it as pointing Claude Code's attention exactly where it needs to look.

☐ **Checkpoint: Your API now has logging middleware, tests, and input validation.**

---

## Exercise 4: Git Workflow with Claude Code (20 min)

---

### Step 1: Initialize Git

```
Initialize a git repository, create a .gitignore for Node.js, and make an init
```

Check the result:

```
git log --oneline
git status
cat .gitignore
```

### Step 2: Feature Branch Workflow

```
Create a new branch called "feature/search" and add a GET /todos/search?q=term
```

### Step 3: Meaningful Commits

```
Commit this change with a descriptive conventional commit message.
```

**Expected:** Something like `feat: add todo search endpoint with case-insensitive matching`

## Step 4: Create a PR Description

```
Write a pull request description for the feature/search branch. Include: what
```

**Expected:** A professional PR description you could actually submit.

## Step 5: Review a Diff

```
Show me the diff between main and feature/search and explain each change.
```

**What just happened?** Claude Code integrates deeply with Git. It creates branches, writes meaningful commits, generates PR descriptions, and reviews diffs — all the tedious parts of version control, automated.

☐ **Checkpoint:** You have a clean Git history with conventional commits and a PR-ready branch.

---

## Exercise 5: Real-World Mini-Project (25 min)

---

### The Challenge

Start a brand new project from scratch. You'll use everything you've learned.

```
mkdir ../md-converter && cd ../md-converter
claude
```

### Step 1: Project Setup (5 min)

```
Create a Node.js CLI tool called "mdx" that converts Markdown files to HTML. R
- Accept a file path as argument: mdx input.md
- Support --output flag for custom output path: mdx input.md --output result.ht
- Support --watch flag to auto-reconvert on file changes
- Use marked for markdown parsing and highlight.js for code syntax highlighting
- Include a CLAUDE.md with project conventions
```

Set up the project with proper package.json, bin configuration, and a shebang line.

## Step 2: Test It (3 min)

Create a test markdown file:

Create a sample.md file with: a heading, a paragraph, a code block (JavaScript), and a list.

## Step 3: Add Features (10 min)

Add these features to mdx:

1. A --style flag that accepts "github", "dark", or "minimal" and embeds the code in a `<pre>` tag.
2. A --toc flag that auto-generates a table of contents from headings.
3. Error handling for missing files with helpful error messages.

## Step 4: Iterate and Refactor (5 min)

Review @src/ for code quality. Refactor any functions longer than 30 lines, extract constants, and improve comments.

## Step 5: Ship It (2 min)

Create a comprehensive README.md with: installation instructions, usage examples, and a contribution guide.

**What just happened?** You built a complete, publishable CLI tool in 25 minutes. You used CLAUDE.md for conventions, @ mentions for context, Git integration for version control, and iterative prompting to refine the result. This is the Claude Code workflow.

❑ **Final Checkpoint:** You have a working CLI tool with multiple features, tests, documentation, and clean Git history.

---

## Troubleshooting

---

Issue	Solution
<code>claude: command not found</code>	Run <code>npm install -g @anthropic-ai/claude-code</code> again. Check your PATH includes npm global bin.
Authentication fails	Try <code>claude auth</code> to re-authenticate. Check your API key is valid.
<code>EACCES</code> permission error	Use <code>sudo npm install -g @anthropic-ai/claude-code</code> or fix npm permissions.
Claude Code hangs	Press <code>Ctrl+C</code> to cancel, then try again with a simpler prompt.
Tests fail after generation	Ask Claude Code: "Run the tests and fix any failures." It's good at self-correction.
Port 3000 already in use	Kill the existing process: <code>lsof -ti:3000   xargs kill</code> or use a different port.

---

## Challenge Extensions (For Fast Finishers)

---

1. **Add a SQLite database** to the todo app instead of in-memory storage
  2. **Add authentication** with JWT tokens to the API
  3. **Add Dockerfile** and docker-compose.yml to the md-converter project
  4. **Create a custom slash command** that runs your test suite and reports results
-