# Claude Code

## Interactive Tutorial & Workshop

**AI-First Development for Modern Software Teams**

https://github.com/emmanuelandre/claude-tutorial

# Workshop Agenda

- Part 1: Philosophy & Foundation
- Part 2: Getting Started Hands-On
- Part 3: Core Workflow (Interactive)
- Part 4: Testing Strategy (Live Demo)
- Part 5: Real-World Application
- Part 6: Q&A; and Practice

# Part 1: Philosophy & Foundation

# The AI-First Philosophy

- Traditional: Human writes code → AI assists
- AI-First: AI executes 100% → Human validates 100%

**This means:**

- AI handles ALL coding, testing, documentation
- Human handles ALL validation, review, decisions
- Clear handoff points at each step
- Systematic quality gates

# Core Development Philosophy

### API-First Development:

- Specifications drive implementation
- Database schema → API contracts → UI components
- E2E tests validate complete user journeys

### Micro-Teams of 2:

- 1 human + Claude = redundancy without overhead
- Each team owns end-to-end features
- Parallel development without bottlenecks

### Zero External Dependencies:

- Be your own QA engineer
- Be your own DevOps engineer

- Own the entire vertical slice

# Testing Philosophy

- E2E tests are MANDATORY (API + UI)
- Unit tests are OPTIONAL (only for complex algorithms)

**Why this works:**

- Test user journeys, not implementation details
- Catch integration issues early
- Enable confident refactoring
- Tests are your regression safety net

- Measure test effectiveness - don't assume value

# Part 2: Getting Started

# What You Need

- Git 2.x or higher
- Code editor (VS Code, Cursor)
- Claude Code access (claude.ai/code)

**Language-specific tools:**

- Go 1.21+ for backend
- Node.js 18+ (via nvm) for React
- Docker for deployment

- GitHub CLI (gh) for PR management

# ■ Hands-On: Create Your First Project

## YOUR PROMPT:

```
mkdir my-api-project cd my-api-project git init # Now ask Claude: "Help me create a CLAUDE.md file for a Go API project with: - PostgreSQL database - JWT authentication - RESTful endpoints - Docker deployment Follow the template structure from github.com/emmanuelandre/claude-tutorial"
```

## What You Should See:

• Claude will create a complete CLAUDE.md with your project structure, conventions, and commands

# The CLAUDE.md File

- Your project's instruction manual for Claude
- Persists context across ALL sessions

**Essential sections:**

- Philosophy & team structure
- Architecture diagram
- Tech stack (Go, React, PostgreSQL, NATS)
- Git workflow and commit format
- Pre-commit checks (mandatory)
- Code patterns and conventions

- Update it as your project evolves

# Hands-On: Verify Your CLAUDE.md

## YOUR PROMPT:

```
"Read my CLAUDE.md and summarize: 1. What tech stack am I using? 2. What's my git workflow? 3. What checks must pass before committing? 4. What's my testing strategy?"
```

## What You Should See:

- Claude should accurately describe all your conventions from CLAUDE.md

# Part 3: AI-First Workflow

# The 10-Step Process

- 1. Human: Write detailed specification
- 2. AI: Design database schema → Human: Review
- 3. AI: Implement repository layer (Go) → Human: Review
- 4. AI: Create API endpoints (Go) → Human: Review
- 5. AI: Write API E2E tests (Cypress) → Human: Verify pass
- 6. AI: Build React components → Human: Review
- 7. AI: Write UI E2E tests (Cypress) → Human: Verify pass
- 8. AI: Update documentation → Human: Review
- 9. Human: Conduct security & code review
- 10. AI: Execute deployment → Human: Verify

# ■ Hands-On: Step 1 - Write Specification

## YOUR PROMPT:

```
"I need to implement user profile editing. Please help me plan: Requirements: - Users can update name, email, bio, avatar - Email must be unique and validated - Changes persist to PostgreSQL - Show success/error messages in React UI Before implementing: 1. What database changes do we need? 2. What API endpoints are required? 3. What security considerations? 4. What edge cases to handle?"
```

## What You Should See:

• Claude provides detailed implementation plan with DB schema, endpoints, security measures, and edge cases

# Hands-On: Step 2 - Database Schema

## YOUR PROMPT:

```
"Create PostgreSQL migration for user profile updates: - Add bio TEXT column - Add avatar_url
VARCHAR(500) column - Add email_verified BOOLEAN - Follow our naming conventions from CLAUDE.md"
```

## What You Should See:

• Claude creates migration files (up.sql and down.sql) following your project patterns

# Example: Migration Output

## SQL

```
-- migrations/003_user_profile.up.sql

ALTER TABLE users

ADD COLUMN bio TEXT,

ADD COLUMN avatar_url VARCHAR(500),

ADD COLUMN email_verified BOOLEAN DEFAULT FALSE,

ADD COLUMN updated_at TIMESTAMP DEFAULT NOW();


CREATE INDEX idx_users_email ON users(email)

WHERE deleted_at IS NULL;


-- migrations/003_user_profile.down.sql

ALTER TABLE users

DROP COLUMN bio,
```

```
DROP COLUMN avatar_url,

DROP COLUMN email_verified;
```

# ■ Hands-On: Step 3 - Repository Layer

## YOUR PROMPT:

```
"Implement Go repository methods for user profile: - UpdateProfile(ctx, userID, data) -
UpdateEmail(ctx, userID, newEmail) - check uniqueness - GetProfile(ctx, userID) Use database/sql
with prepared statements. Follow the Handler → Repository pattern from CLAUDE.md"
```

## What You Should See:

• Claude creates repository methods with proper error handling, SQL injection prevention, and context support

# Example: Go Repository

### Go

```
// internal/repository/user_repository.go

func (r *UserRepository) UpdateProfile(

ctx context.Context,

userID int,

data UpdateProfileData,

) (*User, error) {

query := `

UPDATE users

SET name = $1, bio = $2, avatar_url = $3, updated_at = NOW()

WHERE id = $4 AND deleted_at IS NULL

RETURNING id, email, name, bio, avatar_url

`
```

```
var user User
err := r.db.QueryRowContext(ctx, query,
data.Name, data.Bio, data.AvatarURL, userID,
).Scan(&user.ID;, &user.Email;, &user.Name;,
&user.Bio;, &user.AvatarURL;)

if err == sql.ErrNoRows {
```

# ■ Hands-On: Step 4 - API Endpoints

## YOUR PROMPT:

```
"Create Go API endpoint for profile update: - PUT /api/v1/users/:id/profile - Require JWT
authentication - Check user can only update their own profile - Validate all inputs - Return 200
with updated user or 400/403 on error Use gorilla/mux for routing."
```

## What You Should See:

• Claude creates handler with auth middleware, validation, and proper error responses

# Example: Go HTTP Handler

## Go

```
// internal/handlers/user_handler.go

func (h *UserHandler) UpdateProfile(w http.ResponseWriter, r *http.Request) {

userID := r.Context().Value("user_id").(int)

targetID, _ := strconv.Atoi(mux.Vars(r)["id"])


// Authorization check

if userID != targetID {

respondError(w, http.StatusForbidden,

"Cannot update another user's profile")

return

}


var data UpdateProfileData
```

```
if err := json.NewDecoder(r.Body).Decode(&data;); err != nil {
respondError(w, http.StatusBadRequest, "Invalid request")
return
}

user, err := h.repo.UpdateProfile(r.Context(), userID, data)
if err != nil {
```

# ■ Hands-On: Step 5 - API E2E Tests

## YOUR PROMPT:

```
"Write Cypress E2E tests for profile update API: - Test successful profile update - Test 403
when updating another user - Test 400 for invalid email format - Test 409 for duplicate email -
Test 401 without auth token Save in tests/e2e/api/user-profile.cy.js"
```

## What You Should See:

• Claude creates comprehensive Cypress tests covering all scenarios

# Example: Cypress API Test

### JavaScript

```
// tests/e2e/api/user-profile.cy.js

describe('API: User Profile', () => {

let authToken, userId


before(() => {

cy.request('POST', '/api/v1/auth/login', {

email: 'test@example.com',

password: 'TestPass123!'

}).then((res) => {

authToken = res.body.token

userId = res.body.user.id

})
```

```
})

it('updates profile successfully', () => {
cy.request({
method: 'PUT',
url: `/api/v1/users/${userId}/profile`,
headers: { Authorization: `Bearer ${authToken}` },
body: {
```

# ■ Hands-On: Step 6 - React Components

## YOUR PROMPT:

```
"Create React component for profile editing: - Use React Hook Form for validation - Make PUT request to API - Show loading state during save - Display success/error messages - Optimistic UI updates - Use Zustand for state if needed Create in src/components/ProfileEditor.jsx"
```

## What You Should See:

• Claude creates React component with form validation, API integration, and proper error handling

# Example: React Component

### JavaScript (React)

```
// src/components/ProfileEditor.jsx

import { useState } from 'react'

import { useForm } from 'react-hook-form'

import { api } from '../api/client'


export function ProfileEditor({ user }) {

const [loading, setLoading] = useState(false)

const [message, setMessage] = useState(null)


const { register, handleSubmit, formState: { errors } } = useForm({

defaultValues: {

name: user.name,

bio: user.bio || ''
```

```
}
})

const onSubmit = async (data) => {
setLoading(true)
try {
await api.updateProfile(user.id, data)
```

# ■ Hands-On: Step 7 - UI E2E Tests

## YOUR PROMPT:

```
"Write Cypress UI E2E tests for profile editing: - Test user can update their profile - Test
form validation (required fields) - Test error message display - Test success message display -
Test loading state visibility Use data-test attributes for selectors."
```

## What You Should See:

• Claude creates UI tests that verify the complete user journey in the browser

# Example: Cypress UI Test

### JavaScript

```
// tests/e2e/ui/profile.cy.js

describe('UI: Profile Editing', () => {

beforeEach(() => {

cy.login('test@example.com', 'TestPass123!')

cy.visit('/profile/edit')

})


it('updates profile successfully', () => {

cy.get('[data-test="name-input"]')

.clear()

.type('New Name')


cy.get('[data-test="bio-input"]')
```

```
.type('My new bio')

cy.get('[data-test="save-button"]').click()

cy.get('[data-test="success-message"]')
.should('be.visible')
.and('contain', 'Profile updated')
```

# Part 4: Testing Strategy

# E2E First Philosophy

- Inverted testing pyramid: E2E tests are primary

**Why E2E first?**

- Test actual user flows
- Catch integration issues
- Verify the whole system
- Enable confident refactoring

- Component tests: Only for complex UI logic
- Unit tests: Optional - only for critical algorithms
- Measure effectiveness, don't assume value

# ■ Hands-On: Write Your First E2E Test

## YOUR PROMPT:

```
"Write a Cypress E2E test for user login: 1. Visit /login page 2. Enter email and password 3.
Click login button 4. Verify redirect to /dashboard 5. Verify JWT token in localStorage 6.
Verify welcome message visible Use data-test attributes for selectors."
```

## What You Should See:

• Claude creates a complete E2E test covering the entire login flow

# E2E Testing Best Practices

- Test user journeys, not implementation details
- Use stable data-test attributes, never CSS selectors

**Coverage priorities:**

- Happy path - must always pass
- Critical failures - auth, permissions, validation
- Edge cases - boundary conditions
- Error scenarios - network failures

- Separate API and UI E2E tests
- Create reusable test commands

# ■ Hands-On: Create Reusable Test Commands

## YOUR PROMPT:

```
"Create Cypress custom commands for: 1. cy.login(email, password) - Login and store token 2.
cy.createUser(userData) - Create user via API 3. cy.deleteUser(userId) - Clean up test data Save
in cypress/support/commands.js"
```

## What You Should See:

• Claude creates reusable commands that simplify your test code

# Example: Cypress Commands

### JavaScript

```
// cypress/support/commands.js

Cypress.Commands.add('login', (email, password) => {

cy.request({

method: 'POST',

url: '/api/v1/auth/login',

body: { email, password }

}).then((response) => {

window.localStorage.setItem('token', response.body.token)

window.localStorage.setItem('user',

JSON.stringify(response.body.user))

})

})
```

```
Cypress.Commands.add('createUser', (userData) => {
return cy.request({
method: 'POST',
url: '/api/v1/users',
headers: {
Authorization: `Bearer ${Cypress.env('adminToken')}`
},
```

# Part 5: Best Practices

# Git Workflow Standards

### Branch naming: /

- feature/, fix/, refactor/, docs/, test/

### Conventional commits: (scope): subject

- feat, fix, refactor, docs, test, chore

### HARD RULES - never break:

- Never commit directly to main
- Never merge without review
- Never commit code that fails tests
- Pre-commit checks MUST pass

# ■ Hands-On: Practice: Create Feature Branch

## YOUR PROMPT:

```
"Help me create a feature branch for adding password reset: 1. What should I name the branch? 2. What pre-commit checks do I need to run? 3. What should my commit message be? 4. How do I create the PR?"
```

## What You Should See:

• Claude provides step-by-step git workflow commands following your conventions

# Pre-Commit Checks (Mandatory)

**Backend (Go):**

■ go fmt ./... (format code)

■ go test -v -race ./... (run tests)

■ go build ./... (verify builds)

■ golangci-lint run (if installed)

**Frontend (React):**

■ npm run lint (ESLint)

■ npm run build (Vite build)

**E2E Tests:**

■ npm run test:e2e (all tests must pass)

• DO NOT commit if ANY check fails!

# ■ Hands-On: Run Pre-Commit Checks

## YOUR PROMPT:

```
"I've made changes to my Go API. Walk me through the pre-commit checks: 1. Show me exact commands to run 2. What does each check verify? 3. What do I do if a check fails? 4. When can I commit?"
```

## What You Should See:

• Claude provides exact command sequence with explanations for each check

# Prompt Engineering Tips

- Be specific, not vague - include exact requirements
- Provide context - architecture, patterns, constraints
- Include examples - API contracts, expected behavior

**Multi-step requests:**

- Break complex features into clear steps
- Define validation criteria for each step
- Request tests and documentation explicitly

- Reference existing code patterns for consistency

# ■ Hands-On: Practice: Good vs Bad Prompts

## YOUR PROMPT:

```
BAD: "Add user authentication" GOOD: "Implement JWT authentication for my Go API: - POST
/api/v1/auth/login endpoint - Accept email and password - Return JWT token valid for 15 minutes
- Include refresh token valid for 7 days - Use bcrypt for password hashing (10 rounds) - Follow
the handler pattern from CLAUDE.md - Include comprehensive E2E tests"
```

## What You Should See:

• Claude implements exactly what you specified vs. making assumptions with vague prompts

# Part 6: Real-World Application

# Common Patterns

### API Layer (Go):

- Route → Handler → Repository pattern
- Middleware for auth, CORS, logging
- Prepared statements prevent SQL injection
- Context for cancellation and timeouts

### Frontend (React):

- Page → Container → Component pattern
- Zustand/Redux for state management
- React Query for API caching
- React Hook Form for validation

# ■ Hands-On: Exercise: Complete Feature

## YOUR PROMPT:

```
NOW IT'S YOUR TURN! Implement a complete "Add Comment" feature: Specification: - Users can add comments to posts - Comments have: text (max 500 chars), user_id, post_id - API: POST /api/v1/posts/:id/comments - React component shows comment form - Real-time update after submission Ask Claude to: 1. Design database schema 2. Create Go repository and handler 3. Write API E2E tests 4. Build React component 5. Write UI E2E tests Go through all 10 steps!
```

## What You Should See:

• Complete implementation with all tests passing and documentation updated

# Debugging with Claude

### When you encounter errors:

- Share exact error message and stack trace
- Describe what you were doing
- Show relevant code snippets
- Mention recent changes

### Claude can help:

- Analyze error messages
- Review code for issues
- Suggest fixes with explanations
- Prevent similar issues in future

# ■ Hands-On: Practice: Debug an Issue

## YOUR PROMPT:

```
"I'm getting this error when calling my API: Error: connect ECONNREFUSED 127.0.0.1:8080 What I did: 1. Started my Go server with 'go run cmd/api/main.go' 2. Made request from React app 3. Got this error Recent changes: - Added new endpoint for comments - Updated .env file Help me debug this."
```

## What You Should See:

• Claude systematically debugs: checks if server is running, verifies port, checks CORS, reviews .env configuration

# Keys to Success

### 1. CLAUDE.md is essential

- Keep it current and comprehensive
- Document all conventions and patterns

### 2. Clear handoffs between AI and human

- AI completes a step fully before handoff
- Human approves or requests changes

### 3. E2E tests are non-negotiable

- Write tests before considering feature complete
- Tests are your confidence for refactoring

### 4. Iterate quickly

- Start working, refine as you go

- Commit small, atomic changes

# Summary & Next Steps

# What You Learned Today

- AI-first development philosophy
- The 10-step systematic workflow
- E2E-first testing strategy
- Go backend + React frontend patterns
- Git workflow and quality gates
- Effective prompt engineering
- Real-world debugging techniques

# Your Action Plan

### This Week:

- ■ Create your project's CLAUDE.md
- ■ Set up pre-commit checks
- ■ Write your first E2E test

### This Month:

- ■ Implement one complete feature using 10-step workflow
- ■ Build reusable test commands library
- ■ Document your patterns and learnings

### Ongoing:

- ■ Measure your testing effectiveness
- ■ Update CLAUDE.md as you learn

- Share knowledge with your team

# Resources

### Tutorial Repository:

- github.com/emmanuelandre/claude-tutorial

### Documentation:

- Complete CLAUDE.md template
- Step-by-step workflow guides
- Testing strategy guide
- Troubleshooting common issues

### Official Resources:

- Claude Code: claude.ai/code
- Anthropic Docs: docs.anthropic.com

# Thank You!

## Questions & Practice Time

**github.com/emmanuelandre/claude-tutorial**