Phase II: Todo Full-Stack Web Application

Basic Level Functionality

Objective: Using Claude Code and Spec-Kit Plus transform the console app into a modern multi-user web application with persistent storage.

Requirements

- Implement all 5 Basic Level features as a web application
- Create RESTful API endpoints
- Build responsive frontend interface
- Store data in Neon Serverless PostgreSQL database
- Authentication – Implement user signup/signin using Better Auth

Technology Stack

Layer	Technology
Frontend	Next.js 16+ (App Router)
Backend	Python FastAPI
ORM	SQLModel
Database	Neon Serverless PostgreSQL
Spec-Driven	Claude Code + Spec-Kit Plus
Authentication	Better Auth

API Endpoints

Method	Endpoint	Description
GET	/api/{user_id}/tasks	List all tasks
POST	/api/{user_id}/tasks	Create a new task
GET	/api/{user_id}/tasks/{id}	Get task details
PUT	/api/{user_id}/tasks/{id}	Update a task
DELETE	/api/{user_id}tasks/{id}	Delete a task
PATCH	/api/{user_id}tasks/{id}/complete	Toggle completion

Securing the REST API

Better Auth + FastAPI Integration

The Challenge

Better Auth is a JavaScript/TypeScript authentication library that runs on your **Next.js frontend**. However, your **FastAPI backend** is a separate Python service that needs to verify which user is making API requests.

The Solution: JWT Tokens

Better Auth can be configured to issue **JWT (JSON Web Token)** tokens when users log in. These tokens are self-contained credentials that include user information and can be verified by any service that knows the secret key.

How It Works

- User logs in on Frontend → Better Auth creates a session and issues a JWT token
- Frontend makes API call → Includes the JWT token in the Authorization: Bearer <token> header
- Backend receives request → Extracts token from header, verifies signature using shared secret
- Backend identifies user → Decodes token to get user ID, email, etc. and matches it with the user ID in the URL
- Backend filters data → Returns only tasks belonging to that user

What Needs to Change

Component	Changes Required
Better Auth Config	Enable JWT plugin to issue tokens
Frontend API Client	Attach JWT token to every API request header
FastAPI Backend	Add middleware to verify JWT and extract user
API Routes	Filter all queries by the authenticated user's ID

The Shared Secret

Both frontend (Better Auth) and backend (FastAPI) must use the **same secret key** for JWT signing and verification. This is typically set via environment variable **BETTER_AUTH_SECRET** in both services.

Security Benefits

Benefit	Description
User Isolation	Each user only sees their own tasks
Stateless Auth	Backend doesn't need to call frontend to verify users
Token Expiry	JWTs expire automatically (e.g., after 7 days)
No Shared DB Session	Frontend and backend can verify auth independently

API Behavior Change

After Auth:

All endpoints require valid JWT token
Requests without token receive 401 Unauthorized
Each user only sees/modifies their own tasks
Task ownership is enforced on every operation

Bottom Line

The REST API endpoints stay the same (**GET /api/user_id/tasks**, **POST /api/user_id/tasks**, etc.), but every request now must include a JWT token, and all responses are filtered to only include that user's data.

Monorepo Organization For Full-Stack Projects With GitHub Spec-Kit + Claude Code

This guide explains how to organize your Full-Stack Projects in a monorepo to integrate **GitHub Spec-Kit** for spec-driven development with **Claude Code**. This guide explains how to organize your repository so that Claude Code and Spec-Kit Plus can effectively edit both frontend (Next.js) and backend (FastAPI) code in a single context.

Spec-Kit Monorepo Folder Structure

```
hackathon-todo/
├── .spec-kit/                    # Spec-Kit configuration
│   └── config.yaml
├── specs/                        # Spec-Kit managed specifications
│   ├── overview.md               # Project overview
│   ├── architecture.md           # System architecture
│   ├── features/                 # Feature specifications
│   │   ├── task-crud.md
│   │   ├── authentication.md
│   │   └── chatbot.md
│   ├── api/                      # API specifications
│   │   ├── rest-endpoints.md
│   │   └── mcp-tools.md
│   ├── database/                 # Database specifications
│   │   └── schema.md
│   └── ui/                       # UI specifications
│       ├── components.md
│       └── pages.md
├── CLAUDE.md                     # Root Claude Code instructions
├── frontend/
│   ├── CLAUDE.md
│   └── ... (Next.js app)
├── backend/
│   ├── CLAUDE.md
│   └── ... (FastAPI app)
├── docker-compose.yml
└── README.md
```

Key Differences from Basic Monorepo

Aspect	Without Spec-Kit	With Spec-Kit
Specs Location	/specs (flat)	/specs (organized by type)
Config File	None	/.spec-kit/config.yaml
Spec Format	Freeform markdown	Spec-Kit conventions
Referencing	@specs/file.md	@specs/features/file.md

Spec-Kit Config File

```
# .spec-kit/config.yaml
name: hackathon-todo
version: "1.0"

structure:
  specs_dir: specs
  features_dir: specs/features
  api_dir: specs/api
  database_dir: specs/database
  ui_dir: specs/ui

phases:
```

```
  - name: phase1-console
    features: [task-crud]
  - name: phase2-web
    features: [task-crud, authentication]
  - name: phase3-chatbot
    features: [task-crud, authentication, chatbot]
```

CLAUDE.md Files

Create multiple CLAUDE.md files to provide context at different levels:

Root CLAUDE.md

```
# Todo App - Hackathon II

## Project Overview
This is a monorepo using GitHub Spec-Kit for spec-driven development.

## Spec-Kit Structure
Specifications are organized in /specs:
- /specs/overview.md - Project overview
- /specs/features/ - Feature specs (what to build)
- /specs/api/ - API endpoint and MCP tool specs
- /specs/database/ - Schema and model specs
- /specs/ui/ - Component and page specs

## How to Use Specs
1. Always read relevant spec before implementing
2. Reference specs with: @specs/features/task-crud.md
3. Update specs if requirements change

## Project Structure
- /frontend - Next.js 14 app
- /backend - Python FastAPI server

## Development Workflow
1. Read spec: @specs/features/[feature].md
2. Implement backend: @backend/CLAUDE.md
3. Implement frontend: @frontend/CLAUDE.md
4. Test and iterate

## Commands
- Frontend: cd frontend && npm run dev
- Backend: cd backend && uvicorn main:app --reload
- Both: docker-compose up
```

Frontend CLAUDE.md

```
# Frontend Guidelines

## Stack
- Next.js 14 (App Router)
- TypeScript
- Tailwind CSS

## Patterns
- Use server components by default
- Client components only when needed (interactivity)
- API calls go through `/lib/api.ts`

## Component Structure
- `/components` - Reusable UI components
- `/app` - Pages and layouts

## API Client
All backend calls should use the api client:

import { api } from '@/lib/api'
const tasks = await api.getTasks()

## Styling
- Use Tailwind CSS classes
```

```
- No inline styles
- Follow existing component patterns
```

Backend CLAUDE.md

```
# Backend Guidelines

## Stack
- FastAPI
- SQLModel (ORM)
- Neon PostgreSQL

## Project Structure
- `main.py` - FastAPI app entry point
- `models.py` - SQLModel database models
- `routes/` - API route handlers
- `db.py` - Database connection

## API Conventions
- All routes under `/api/`
- Return JSON responses
- Use Pydantic models for request/response
- Handle errors with HTTPException

## Database
- Use SQLModel for all database operations
- Connection string from environment variable: DATABASE_URL

## Running
uvicorn main:app --reload --port 8000
```

Example Spec Files

/specs/overview.md

```
# Todo App Overview

## Purpose
A todo application that evolves from console app to AI chatbot.

## Current Phase
Phase II: Full-Stack Web Application

## Tech Stack
- Frontend: Next.js 14, TypeScript, Tailwind CSS
- Backend: FastAPI, SQLModel, Neon PostgreSQL
- Auth: Better Auth with JWT

## Features
- [ ] Task CRUD operations
- [ ] User authentication
- [ ] Task filtering and sorting
```

/specs/features/task-crud.md

```
# Feature: Task CRUD Operations
```

```
## User Stories
- As a user, I can create a new task
- As a user, I can view all my tasks
- As a user, I can update a task
- As a user, I can delete a task
- As a user, I can mark a task complete

## Acceptance Criteria

### Create Task
- Title is required (1-200 characters)
- Description is optional (max 1000 characters)
- Task is associated with logged-in user

### View Tasks
- Only show tasks for current user
- Display title, status, created date
- Support filtering by status
```

/specs/api/rest-endpoints.md

```
# REST API Endpoints

## Base URL
- Development: http://localhost:8000
- Production: https://api.example.com

## Authentication
All endpoints require JWT token in header:
Authorization: Bearer <token>

## Endpoints

### GET /api/tasks
List all tasks for authenticated user.

Query Parameters:
- status: "all" | "pending" | "completed"
- sort: "created" | "title" | "due_date"

Response: Array of Task objects

### POST /api/tasks
Create a new task.

Request Body:
- title: string (required)
- description: string (optional)

Response: Created Task object
```

/specs/database/schema.md

```
# Database Schema

## Tables

### users (managed by Better Auth)
- id: string (primary key)
- email: string (unique)
- name: string
- created_at: timestamp

### tasks
- id: integer (primary key)
- user_id: string (foreign key -> users.id)
- title: string (not null)
- description: text (nullable)
- completed: boolean (default false)
- created_at: timestamp
- updated_at: timestamp

## Indexes
- tasks.user_id (for filtering by user)
- tasks.completed (for status filtering)
```

Workflow with Spec-Kit + Claude Code

- Write/Update Spec → @specs/features/new-feature.md
- Ask Claude Code to Implement → "Implement @specs/features/new-feature.md"
- Claude Code reads: Root CLAUDE.md, Feature spec, API spec, Database spec, Relevant CLAUDE.md
- Claude Code implements in both frontend and backend
- Test and iterate on spec if needed

Referencing Specs in Claude Code

```
# Implement a feature
You: @specs/features/task-crud.md implement the create task feature

# Implement API
You: @specs/api/rest-endpoints.md implement the GET /api/tasks endpoint

# Update database
You: @specs/database/schema.md add due_date field to tasks

# Full feature across stack
You: @specs/features/authentication.md implement Better Auth login
```

Summary

Component	Purpose
/.spec-kit/config.yaml	Spec-Kit configuration
/specs/overview.md	Project overview and status
/specs/features/	What to build (user stories, acceptance criteria)
/specs/api/	How APIs should work
/specs/database/	Data models and schema
/specs/ui/	UI components and pages
/CLAUDE.md	How to navigate and use specs
/frontend/CLAUDE.md	Frontend-specific patterns
/backend/CLAUDE.md	Backend-specific patterns

Key Point:
Spec-Kit provides organized, structured specs that Claude Code can reference. The CLAUDE.md files tell Claude Code how to use those specs and project-specific conventions.

Summary: Monorepo vs Separate Repos

Approach	Pros	Cons
Monorepo ☆	Single CLAUDE.md context, easier cross-cutting changes	Larger repo
Separate Repos	Clear separation, independent deployments	Claude Code needs workspace setup

Recommendation:
Use monorepo for the hackathon – simpler for Claude Code to navigate and edit both frontend and backend in a single context.

Key Benefits of This Structure

Benefit	Description
Single Context	Claude Code sees entire project, can make cross-cutting changes
Layered CLAUDE.md	Root file for overview, subfolder files for specific guidelines
Specs Folder	Reference specifications directly with @specs/filename.md
Clear Separation	Frontend and backend code in separate folders, easy to navigate