Assignment: Build an App for AI Agents

Course: MIT — Building with AI Agents **Due:** See Canvas for deadline **Format:** Individual or teams of 2

Overview

You will build a **web application that AI agents can use autonomously**. Your app will expose a set of API endpoints and a `skill.md` file that teaches any OpenClaw agent how to interact with it — without human intervention.

You can build any app you want. The only requirement is that agents can discover it, learn it, and use it through the skill.md protocol. Some ideas:

- **Study group finder** — agents find classmates studying the same topics
- **Event planner** — agents coordinate meetups, RSVPs, and schedules
- **Book/movie club** — agents recommend and discuss media with each other
- **Marketplace** — agents post and browse listings on behalf of their humans
- **Debate forum** — agents argue positions on topics and vote
- **Recipe exchange** — agents share and rate recipes from their humans
- **Fitness challenge tracker** — agents log workouts and compete
- **Research paper matchmaker** — agents find collaborators based on research interests
- **Lost & found board** — agents post and search for lost items
- **Confession wall** — agents post anonymous confessions and react to others

Or anything else. Get creative.

What You're Building

Your app has three parts:

1. Protocol Files (how agents discover and learn your app)

These are markdown/JSON files served at specific URLs that teach agents what your app does and how to use it.

File	URL	Purpose
skill.md	`/skill.md`	Complete API documentation — teaches agents every endpoint, with examples
heartbeat.md	`/heartbeat.md`	Task loop — tells agents what to do and when they're done
skill.json	`/skill.json`	Package metadata — name, version, description, emoji

skill.md is the most important file. Think of it as a user manual written for AI instead of humans. When an OpenClaw agent reads your `skill.md`, it should be able to start using your app immediately — registering, authenticating, and calling every endpoint without any human help.

heartbeat.md is a task loop that drives the agent forward. It's not a passive check-in — it tells the agent: "Here's what you need to accomplish. Keep going until you're done. If something goes wrong, ask your human."

skill.json is simple metadata so agent platforms can display your app's name, description, and emoji.

2. Backend API (what agents actually call)

A set of REST API endpoints that agents interact with. At minimum, your app needs:

- **Agent registration** — an endpoint where agents register themselves and get an API key
- **Agent claiming** — a way for humans to claim/verify ownership of their agent
- **Core functionality** — whatever your app does (posting, browsing, messaging, voting, etc.)
- **Bearer token auth** — every request (except registration) requires an API key

3. Frontend (what humans see)

A web interface where humans can:

- See what's happening in the app (browse content, view activity)
- Claim their agent (click a link, that's it)
- View results/output of whatever your app does

The frontend doesn't need to be fancy, but it should be functional and look decent.

Step-by-Step Guide

Step 1: Set Up Your Project

Use whatever tech stack you want. The example below uses Next.js + MongoDB, but you can use Flask + SQLite, Express + PostgreSQL, Django + Supabase, Rails, Go — whatever you're comfortable with. The only thing that matters is that your app serves `skill.md`, `heartbeat.md`, `skill.json`, has API endpoints, and has a frontend.

Example with Next.js + MongoDB (recommended if you don't have a strong preference):

```
npx create-next-app@latest my-agent-app --typescript --app --tailwind
cd my-agent-app
npm install mongoose nanoid
```

Pick any database you want. Some free options:

Database	Free Tier	Good for
MongoDB Atlas	512MB free forever	Document-based, flexible schemas
Supabase	500MB, 2 projects free	PostgreSQL, built-in auth
PlanetScale	1 DB free	MySQL, great for relational data
Turso	9GB free	SQLite at the edge, simple
Neon	512MB free	Serverless PostgreSQL
SQLite file	Unlimited, free	Simplest possible, works locally

If using MongoDB Atlas:

1. Go to cloud.mongodb.com → create a free M0 cluster
2. Create a database user → get connection string

3. Replace `<password>` in the URI with your actual password

Create `.env.local` (or equivalent for your framework):

```
MONGODB_URI=mongodb+srv://username:password@cluster.mongodb.net/?
retryWrites=true&w=majority
MONGODB_DB=your-app-name
APP_URL=http://localhost:3000
NEXT_PUBLIC_APP_URL=http://localhost:3000
ADMIN_KEY=pick-any-secret-string
```

Important: Never push credentials to GitHub. Add `.env*.local` to `.gitignore`.

Step 2: Database Connection

(Skip this if you're using a different database — adapt to your ORM/driver.)

If using MongoDB with Mongoose, create `lib/db/mongodb.ts` — this handles connection pooling for serverless environments:

```
import mongoose from 'mongoose';

const MONGODB_URI = process.env.MONGODB_URI!;
const MONGODB_DB = process.env.MONGODB_DB || 'my-agent-app';

if (!MONGODB_URI) throw new Error('Missing MONGODB_URI');

let cached = (global as any).mongoose;
if (!cached) cached = (global as any).mongoose = { conn: null, promise: null };

export async function connectDB() {
  if (cached.conn) return cached.conn;
  if (!cached.promise) {
    cached.promise = mongoose.connect(MONGODB_URI, { dbName: MONGODB_DB });
  }
  cached.conn = await cached.promise;
  return cached.conn;
}
```

Step 3: Define Your Models

(Adapt this to your database. The schema is what matters, not the ORM.)

At minimum you need an **Agent** model. Here's an example with Mongoose — if you're using Prisma, Drizzle, SQLAlchemy, or raw SQL, just create the equivalent table/schema. Create `lib/models/Agent.ts`:

```
import mongoose, { Schema, Document } from 'mongoose';

export interface IAgent extends Document {
  name: string;
  description: string;
  apiKey: string;
  claimToken: string;
  claimStatus: 'pending_claim' | 'claimed';
  ownerEmail?: string;
  lastActive: Date;
}

const AgentSchema = new Schema<IAgent>({
  name: { type: String, required: true, unique: true },
  description: { type: String, required: true },
  apiKey: { type: String, required: true, unique: true },
  claimToken: { type: String, required: true, unique: true },
  claimStatus: { type: String, default: 'pending_claim' },
  ownerEmail: String,
  lastActive: { type: Date, default: Date.now },
}, { timestamps: true });

export default mongoose.models.Agent || mongoose.model<IAgent>('Agent',
AgentSchema);
```

Then define models for whatever your app does — posts, events, reviews, messages, etc.

Step 4: Helper Utilities

Create `lib/utils/api-helpers.ts` with reusable functions:

```
import { NextResponse } from 'next/server';
import { nanoid } from 'nanoid';

// Standard success response
export function successResponse(data: any, status = 200) {
  return NextResponse.json({ success: true, data }, { status });
}

// Standard error response
export function errorResponse(error: string, hint: string, status: number) {
  return NextResponse.json({ success: false, error, hint }, { status });
}

// Generate API key for agents
export function generateApiKey(): string {
  return `yourapp_${nanoid(32)}`;
}

// Generate claim token
export function generateClaimToken(): string {
  return `yourapp_claim_${nanoid(24)}`;
}

// Extract API key from Authorization header
export function extractApiKey(header: string | null): string | null {
  if (!header) return null;
  return header.replace('Bearer ', '').trim() || null;
}
```

Install nanoid: `npm install nanoid`

Step 5: Build Your API Routes

Registration: `app/api/agents/register/route.ts`

This is the first endpoint any agent calls. It creates an agent and returns an API key.

```
import { NextRequest } from 'next/server';
import { connectDB } from '@/lib/db/mongodb';
import Agent from '@/lib/models/Agent';
import { successResponse, errorResponse, generateApiKey, generateClaimToken } from
'@/lib/utils/api-helpers';

export async function POST(req: NextRequest) {
  await connectDB();
  const { name, description } = await req.json();

  if (!name || !description) {
    return errorResponse('Missing fields', 'Both "name" and "description" required',
400);
  }

  // Check if name is taken
  const existing = await Agent.findOne({ name: new RegExp(`^${name}$`, 'i') });
  if (existing) {
    return errorResponse('Name taken', 'Choose a different name', 409);
  }

  const apiKey = generateApiKey();
  const claimToken = generateClaimToken();
  const baseUrl = process.env.APP_URL || process.env.NEXT_PUBLIC_APP_URL ||
'http://localhost:3000';

  await Agent.create({ name, description, apiKey, claimToken });

  return successResponse({
    agent: {
      name,
      api_key: apiKey,
      claim_url: `${baseUrl}/claim/${claimToken}`,
    },
    important: 'SAVE YOUR API KEY! You cannot retrieve it later.',
  }, 201);
}
```

Claim page: `app/claim/[token]/page.tsx`

A simple page where the human clicks to claim their agent. No complicated verification — just click and done.

Auth middleware pattern

For all other endpoints, extract and validate the API key:

```
const apiKey = extractApiKey(req.headers.get('authorization'));
if (!apiKey) return errorResponse('Missing API key', 'Include Authorization header',
401);

const agent = await Agent.findOne({ apiKey });
if (!agent) return errorResponse('Invalid API key', 'Agent not found', 401);
```

Your app's endpoints

Build whatever endpoints your app needs. For example, if you're building an event planner:

- `POST /api/events` — create an event
- `GET /api/events` — list events
- `POST /api/events/:id/rsvp` — RSVP to an event
- `GET /api/events/:id` — get event details

Every endpoint should follow this pattern:

- Authenticate via Bearer token
- Do the thing
- Return `{ success: true, data: {...} }` or `{ success: false, error: "...", hint: "..." }`

Step 6: Write Your skill.md

This is the most important part. Create `app/skill.md/route.ts` :

```
import { NextResponse } from 'next/server';

export async function GET() {
  const baseUrl = process.env.APP_URL || process.env.NEXT_PUBLIC_APP_URL ||
'http://localhost:3000';

  const markdown = `---
name: your-app-name
version: 1.0.0
description: One sentence describing what your app does.
homepage: ${baseUrl}
metadata: {"openclaw":
{"emoji":"🎯","category":"social","api_base":"${baseUrl}/api"}}
---

# Your App Name

Describe what your app does in 2-3 sentences.
## Step 1: Register

\\\`\\\`\\\`bash
curl -X POST ${baseUrl}/api/agents/register \\\\
  -H "Content-Type: application/json" \\\\
  -d '{"name": "YourAgentName", "description": "What you do"}'
\\\`\\\`\\\`

Response: { "data": { "agent": { "api_key": "...", "claim_url": "..." } } }

Save your api_key. Send the claim_url to your human.

## Step 2: Get Claimed

Your human clicks the claim link. Done.

## Step 3: [Your core functionality]

Document every endpoint with:
- What it does
- The curl command to call it
- What the response looks like

...continue for all your endpoints...

## Authentication

All requests (except register) require:
\\\`\\\`\\\`
Authorization: Bearer YOUR_API_KEY
\\\`\\\`\\\`

## Response Format

Success: { "success": true, "data": {...} }
Error: { "success": false, "error": "...", "hint": "..." }
`;

  return new NextResponse(markdown, {
    headers: { 'Content-Type': 'text/markdown; charset=utf-8' },
  });
}
```

Tips for writing good skill.md:

- **Be explicit.** Don't assume the agent knows anything. Show exact curl commands.
- **Show response examples.** Agents need to know what to expect back.
- **Use numbered steps.** Walk the agent through the flow from registration to completion.
- **Include error hints.** Tell agents what to do if something goes wrong.
- **Mention escalation.** Tell agents: "If you don't know something about your human, message them and ask."

Step 7: Write Your heartbeat.md

Create `app/heartbeat.md/route.ts`. The heartbeat is a **continuous task loop** — not a passive check-in. It tells the agent what to accomplish and keeps it going until the task is done.

```
import { NextResponse } from 'next/server';

export async function GET() {
  const baseUrl = process.env.APP_URL || process.env.NEXT_PUBLIC_APP_URL ||
'http://localhost:3000';

  const markdown = `# Your App — Heartbeat

**Keep running this loop until you've completed the goal.**

If anything goes wrong, message your human and ask what to do.

## Your Goal

[Describe what "done" looks like. For example:]
- Post at least 3 items
- Respond to all incoming messages
- Browse and interact with at least 5 other agents' content

## Loop

### Step 1: Check your setup
[Am I registered? Claimed? Profile created?]

### Step 2: Handle incoming activity
[Check for new messages, requests, notifications]

### Step 3: Do the main thing
[Post content, browse others, interact, etc.]

### Step 4: Check if done
[Have I met the goal? If yes, tell my human. If no, go back to Step 2.]

## Error Handling

If something fails:
1. Message your human and explain what happened
2. Ask them what to do
3. Try again once you have an answer

Never silently fail.
`;

  return new NextResponse(markdown, {
    headers: { 'Content-Type': 'text/markdown; charset=utf-8' },
  });
}
```

Step 8: Write Your skill.json

Create `app/skill.json/route.ts` :

```
import { NextResponse } from 'next/server';

export async function GET() {
  const baseUrl = process.env.APP_URL || process.env.NEXT_PUBLIC_APP_URL ||
'http://localhost:3000';

  return NextResponse.json({
    name: 'your-app-name',
    version: '1.0.0',
    description: 'One sentence about your app.',
    homepage: baseUrl,
    metadata: {
      openclaw: {
        emoji: '🎯',
        category: 'social',
        api_base: `${baseUrl}/api`,
      },
    },
  });
}
```

Step 9: Build Your Frontend

Your app needs a web interface. At minimum:

- **Landing page** (`/`) — what the app is, how it works, link to skill.md
- **Claim page** (`/claim/[token]`) — where humans claim their agent
- **Content pages** — whatever your app shows (events, posts, reviews, etc.)

Use Tailwind CSS for styling. Here's a minimal landing page pattern:

```
export default function HomePage() {
  return (
    <div className="max-w-4xl mx-auto px-4 py-16 text-center">
      <h1 className="text-5xl font-bold mb-4">Your App Name</h1>
      <p className="text-xl text-gray-600 mb-8">
        What your app does in one sentence.
      </p>
      <div className="bg-gray-900 rounded-xl p-6 mb-8">
        <p className="text-gray-300 mb-2">Tell your OpenClaw agent:</p>
        <code className="text-green-400 text-lg">
          Read https://your-url/skill.md
        </code>
      </div>
    </div>
  );
}
```

Step 10: Environment Variables for Production

Important: `NEXT_PUBLIC_*` variables get baked in at build time. For URLs that need to resolve correctly in production, use a non-prefixed variable like `APP_URL` and read it at runtime:

```
// DO THIS — works in production
const baseUrl = process.env.APP_URL || process.env.NEXT_PUBLIC_APP_URL ||
'http://localhost:3000';

// NOT THIS — gets baked as "localhost" at build time
const baseUrl = process.env.NEXT_PUBLIC_APP_URL || 'http://localhost:3000';
```

Add `APP_URL` to your deployment environment set to your production URL.

Step 11: Deploy to Railway

We'll use [Railway](https://railway.com) for deployment. (You can also use Vercel, Render, Fly.io, or any other platform — your app just needs to be live at a public URL.)

1. Push your code to GitHub (make sure `.env*.local` is in `.gitignore`)
2. Create an account at railway.com
3. Click **New Project** → **Deploy from GitHub repo** → select your repo
4. Add environment variables in the Railway dashboard (click on your service → **Variables**):
    - Your database connection string (e.g. `MONGODB_URI`)
    - `APP_URL` — your Railway URL (e.g. `https://my-app.up.railway.app`)
    - `ADMIN_KEY` — a secret string for admin endpoints
    - Any other env vars your app needs
5. Create `railway.json` in your project root:

```
{
  "$schema": "https://railway.com/railway.schema.json",
  "build": { "builder": "NIXPACKS" },
  "deploy": {
    "startCommand": "npm start",
    "restartPolicyType": "ON_FAILURE",
    "restartPolicyMaxRetries": 10
  }
}
```

6. Push to GitHub — Railway auto-deploys on every push

7. Go to your service's **Settings** → **Networking** → **Generate Domain** to get your public URL

Verify your deployment:

```
# Check skill.md serves correctly with your production URL
curl https://your-app.up.railway.app/skill.md

# Test registration
curl -X POST https://your-app.up.railway.app/api/agents/register \
  -H "Content-Type: application/json" \
  -d '{"name": "TestAgent", "description": "Testing deployment"}'
```

Make sure `skill.md` shows your production URL (not `localhost`). If it shows `localhost`, check Step 10 — you probably need to set `APP_URL` in your Railway environment variables.

Testing Your App

Before submitting, test the full agent flow yourself using curl:

```
# 1. Read skill.md — does it explain everything clearly?
curl https://your-url/skill.md

# 2. Register an agent
curl -X POST https://your-url/api/agents/register \
  -H "Content-Type: application/json" \
  -d '{"name": "TestAgent", "description": "Test"}'

# 3. Claim the agent (use the claim_url from registration)

# 4. Use your API key to call every endpoint documented in skill.md
# 5. Follow the heartbeat.md loop — can you complete the goal?
```

Ask yourself: If I were an AI agent reading skill.md for the first time, would I know exactly what to do? If the answer is no, your skill.md needs more detail.

Reference Example: ClawMatchStudio

This is just one example. It uses Next.js + MongoDB + Railway, but you can use completely different tools. What matters is the protocol (skill.md, heartbeat.md, skill.json) and that agents can use your app.

- **GitHub:** github.com/mariagorskikh/homework2_example
- **Live app:** clawmatch.up.railway.app
- **skill.md:** clawmatch.up.railway.app/skill.md
- **heartbeat.md:** clawmatch.up.railway.app/heartbeat.md
- **Assignment doc:** github.com/mariagorskikh/homework2_example/blob/main/assignment.md

ClawMatchStudio is a team matching app where agents have conversations with each other to find compatible teammates. Study it for the patterns — the protocol files, the API design, the auth flow, the frontend — but build something different and make it your own.

Other references:

- **OpenClaw** — the agent framework your agent runs on: openclaw.com
- **Moltbook** — a social network for agents that uses the same skill.md protocol: moltbook.com

---

FAQ

Q: Do I have to use Next.js? A: No. Use whatever you want — Flask, Express, Django, Rails, Go, anything. The example uses Next.js because it bundles frontend + API + deployment nicely, but it's just one option. Pick whatever you're most productive with.

Q: Do I have to use MongoDB? A: No. Use any database — PostgreSQL, MySQL, SQLite, Supabase, PlanetScale, Turso, Neon, even a JSON file if your app is simple enough. The example uses MongoDB Atlas because the free tier is generous, but it's just one option.

Q: Do I have to deploy on Railway? A: No. Deploy anywhere — Vercel, Render, Fly.io, Netlify, your own VPS, whatever. Your app just needs to be live at a public URL so agents can reach it.

Q: Do I have to follow the exact code structure from the example? A: No. The example is one way to do it. What matters is: your app serves `skill.md` , `heartbeat.md` , and `skill.json` at the right URLs, has API endpoints that work, has auth, and has a frontend. How you organize the code internally is up to you.

Q: Do I need agent-to-agent conversations? A: No. That's what the example does, but your app can be anything. An event planner where agents RSVP, a review board where agents post ratings, a marketplace — whatever you want. Agents just need to be able to use it through skill.md.

Q: How detailed should skill.md be? A: Very. Include curl commands for every endpoint, show example responses, explain what to do on errors. The agent has never seen your app before — skill.md is the only documentation it gets. If an agent can't figure out how to use your app from skill.md alone, it needs more detail.

Q: What if my agent doesn't know something about my human? A: That's what OpenClaw channels are for. Your skill.md should tell agents: "If you don't know something about your human, message them through your channel (WhatsApp, Telegram, Discord, Slack, OpenClaw chat, etc.) and ask."

Q: Can I work in a team? A: Teams of up to 2. Both members should contribute and understand the full codebase.

Good luck, and build something cool. 🦀