Option 1: Flask (Python)

Best for: very small, straightforward REST APIs; fast learning curve.

Pros

- Minimal and easy to understand, good for beginners
- Lots of tutorials and examples
- Flexible: you can structure it simply and grow over time
- Works well as a pure API layer in front of someone else’s database

Cons

- You have to choose and wire up pieces yourself (auth, validation, background jobs, websockets)
- Easy to end up with inconsistent structure if the team is not disciplined
- For larger apps, you will eventually need conventions (or it becomes messy)

Option 2: FastAPI (Python)

Best for: a clean REST API with strong validation, modern patterns, and auto-generated docs.

Pros

- Very beginner-friendly once set up, and harder to “mess up” than Flask
- Automatic request validation using typed models
- Auto-generated interactive API docs (great for teamwork and testing)
- Generally strong performance
- Easy to enforce consistent API shapes

Cons

- Requires learning Python type hints and Pydantic models (not hard, but new for some)
- Background tasks and websockets are possible but add complexity depending on hosting
- Slightly more “framework-y” than Flask

Option 3: Django + Django REST Framework (Python)

Best for: larger apps with lots of features, admin tools, and a longer lifetime.

Pros

- Very complete framework with lots built in (auth, admin panel, security defaults)
- Django REST Framework is powerful for APIs
- Strong conventions reduce chaos in bigger teams
- Good if the project grows into a serious long-term system

Cons

- Heavyweight for a small team and early MVP
- More boilerplate and configuration
- Can feel slow to iterate at the start compared to Flask/FastAPI
- People may end up fighting the framework for simple tasks

Option 4: Express (Node.js)

Best for: teams already comfortable with JavaScript and building simple APIs quickly.

Pros

- Huge ecosystem and lots of community examples
- Very flexible, easy to start fast

- Matches frontend JavaScript skills, one language across stack

Cons

- Like Flask, you must pick and assemble many parts yourself
- Easy to create inconsistent patterns without a standard structure
- Requires careful attention to security and validation patterns

Option 5: NestJS (Node.js, TypeScript)

Best for: a structured, maintainable Node backend that scales with a team.

Pros

- Clear conventions and project structure from day one
- TypeScript improves reliability and reduces runtime bugs
- Built-in patterns for auth, validation, modules, testing
- Better long-term maintainability than “free-form Express”

Cons

- Steeper learning curve than Express or Flask
- More framework concepts to learn upfront
- Can feel like overkill for a small MVP

Option 6: Serverless Functions Only (no always-on server)

Examples: Cloudflare Workers, Vercel Functions, Netlify Functions.

Best for: minimal ops and a small set of endpoints, especially if you do not want to manage deployments on a VM.

Pros

- No server to run or maintain
- Deploy is usually very simple (connect to GitHub, push to deploy)
- Can be cheap or free at low usage
- Good for scheduled tasks and webhook-style endpoints (depends on provider)

Cons

- Harder to support persistent connections for real-time features
- Local development and debugging can be trickier
- You still need to implement auth, validation, background workflows
- Free tier limits may bite if usage grows