THE OPENAI API

Goal of This Talk

- You already know what an API is.
- You have written simple FastAPI servers with routes, request bodies, and response objects.
- This talk explains how the OpenAI API is designed and how its major endpoints fit together.
- The main question is: when your application needs AI behavior, which OpenAI endpoint should your backend call, and what shape should the request have?

The Big Picture

- The OpenAI API is a hosted model platform exposed through HTTP APIs.
- Your application sends inputs to OpenAI.
- OpenAI runs a model or platform service.
- OpenAI returns structured output that your application can store, display, transform, or use to trigger other work.
- In a web application, your backend should call OpenAI; your browser frontend should not call OpenAI directly with a secret API key.

Mental Model

- Think of OpenAI as a set of API resources, not just one “chat endpoint.”
- The same platform includes endpoints for:
  - model responses
  - chat completions
  - embeddings
  - images
  - audio
  - moderation
  - files
  - vector stores
  - batches
  - fine-tuning
  - realtime interactions
- Most applications use only a small subset.
- The right endpoint depends on the product behavior you are building.

Base URL and Version

- The API is organized under a versioned base path.
- Most REST endpoints use:

  `https://api.openai.com/v1`
- Example full endpoint:

  `POST https://api.openai.com/v1/responses`
- The `/v1` prefix is part of the API contract.
- Individual resources then appear after `/v1`, such as `/responses`, `/embeddings`, or `/audio/transcriptions`.

Authentication

- OpenAI API requests use API keys.
- The key is sent with HTTP Bearer authentication:

  `Authorization: Bearer OPENAI_API_KEY`
- API keys are secrets.
- Do not put an API key in browser JavaScript, mobile app code, public GitHub repositories, or screenshots.
- In server applications, load the key from an environment variable or a secrets manager.
- If a request needs to specify a particular organization or project, headers can be used for that, but many simple projects only need the API key.

SDKs vs. Raw HTTP

- You can call OpenAI with raw HTTP using `curl`, `requests`, or any HTTP client.
- Official SDKs wrap the HTTP details in a library.
- The SDK still sends ordinary API requests under the hood.
- SDKs are useful because they handle:
  - request construction
  - response parsing
  - file uploads
  - streaming helpers
  - typed objects in some languages
- For learning the design, it is still useful to understand the raw endpoint paths and JSON bodies.

The Main Model Endpoint: Responses

- For new text, image-input, tool-using, and multi-step model interactions, the main endpoint is the Responses API.
- Endpoint:

  `POST /v1/responses`
- It creates a model response from input.
- The input can be simple text or a structured list of messages and content parts.
- Responses can also use tools, structured outputs, conversation state, streaming, and background mode.
- If you are starting a new app, this is usually the first endpoint to understand.

Minimal Responses Request

- A minimal request needs a model and input.
- Conceptually:

```
{
  "model": "MODEL_NAME",
  "input": "Explain what an API endpoint is
in one paragraph."
}
```

- The model name chooses which model handles the request.
- The input is the user's task or message.
- The response object contains output items, status information, and usage information.

Responses Request Shape

- Common request fields include:
  - `model`: which model to use
  - `input`: text, messages, images, files, or mixed content
  - `instructions`: high-level behavior for the model
  - `tools`: tools the model may call
  - `tool_choice`: whether tool use is automatic or forced
  - `text.format`: output format, including structured JSON formats
  - `stream`: whether to stream partial output
  - `previous_response_id`: prior response for multi-turn context
  - `conversation`: a server-side conversation resource
  - `metadata`: application-defined metadata
  - `max_output_tokens`: output budget
- Not every field is needed for simple applications.

Responses Output Shape

- A response object is more than a string.
- It can include:
  - an `id`
  - a `status`
  - the `model` used
  - an `output` array
  - text output items
  - tool call items
  - usage information
  - error or incomplete details
- Application code usually extracts the text output for display.
- More advanced applications inspect the output array to handle tool calls, citations, or structured data.

Stateless Calls

- The simplest model call is stateless.
- Your app sends all necessary context in the request.
- OpenAI returns an answer.
- The next request starts fresh unless you include the prior context again.
- Stateless calls are easy to reason about.
- They are useful for:
  - summarization
  - classification
  - one-off generation
  - extraction
  - short assistant replies where your app manages history

Stateful Conversations

- Some applications need multi-turn context.
- There are several ways to manage state:
  - store conversation history in your own database and include relevant messages each time
  - use `previous_response_id` to build on a prior response
  - use a conversation resource where supported
- Your database is still important if your product needs durable user-visible history.
- Model-side state is useful for request context, but product state belongs in your application.

Messages and Roles

- Model requests often include messages with roles.
- Common roles include:
  - `developer` or `system` for instructions
  - `user` for user input
  - `assistant` for prior model output
  - tool-related roles or output items for tool results
- Instruction hierarchy matters.
- Higher-priority instructions should not be mixed into normal user content.
- A chat app should not trust the user to provide its system or developer instructions.

Instructions vs. User Input

- Instructions define how the model should behave.
- User input defines the task the user is asking about.
- Keep them separate.
- Example:
  - instruction: “Answer as a concise database teaching assistant.”
  - user input: “What is a foreign key?”
- If you concatenate everything into one string, it becomes harder to reason about prompt injection and policy boundaries.
- Structured request fields help preserve intent.

Structured Outputs

- Many applications need data, not prose.
- Structured outputs let you ask the model to produce JSON matching a schema.
- This is useful for:
  - extracting fields from text
  - classifying content
  - generating records for a database
  - returning predictable data to a frontend
- The important idea is that the model output should match an application contract.
- Your backend should still validate the returned object before trusting it.

Tool Use

- Tool use lets the model ask your application to perform an action.
- The model does not magically call your private systems.
- Instead, it returns a tool call with arguments.
- Your application decides whether to execute the tool.
- Your application runs the tool code.
- Your application sends the result back to the model if another model step is needed.
- This is a control loop, not direct model access to your server internals.

Function Calling

- Function calling is the most common custom tool pattern.
- You describe a function name, description, and JSON schema for arguments.
- The model may return a request to call that function.
- Your code parses and validates the arguments.
- Your code calls the real function.
- Your code returns the function result to the model or directly to the user.
- Never execute tool arguments blindly.

Built-In Tools

- The Responses API can also use built-in tools.
- Examples include:
  - file search
  - web search
  - computer use
  - code interpreter, depending on the product surface and access
  - image generation as a tool in some workflows
- Built-in tools are useful when the task requires capabilities beyond plain generation.
- They also make responses more complex because the output may include tool call items, citations, or intermediate results.

Streaming

- Some model responses take time.
- Streaming sends partial output as it becomes available.
- This improves user experience for chat applications.
- Instead of waiting for the entire answer, the frontend can display tokens or text chunks incrementally.
- In HTTP APIs, this is often implemented with server-sent events.
- In a FastAPI app, your backend can receive the OpenAI stream and forward a stream to the browser.

Streaming Design

- Streaming adds complexity.
- You need to think about:
  - connection lifetime
  - client disconnects
  - partial output
  - error handling after some text has already been sent
  - whether to store partial or final messages
- For a first implementation, non-streaming is simpler.
- For a polished chat interface, streaming feels much better.

Background Mode

- Some tasks are too slow for a normal request-response cycle.
- Background mode lets a response continue running asynchronously.
- The app can create a job-like response and check status later.
- This pattern is useful for:
  - long document processing
  - multi-step tool work
  - tasks where the user can wait or come back later
- The design resembles a FastAPI app that starts a background task and returns a job ID.

Retrieve, Delete, Cancel, and Compact Responses

- The Responses API includes more than creation.
- Common response-related endpoints include:
  - `POST /v1/responses`
  - `GET /v1/responses/{response_id}`
  - `DELETE /v1/responses/{response_id}`
  - `GET /v1/responses/{response_id}/input_items`
  - `POST /v1/responses/{response_id}/cancel`
  - `POST /v1/responses/compact`
  - `POST /v1/responses/count_tokens`
- These endpoints treat responses as resources.
- They support real application needs such as retrieval, cleanup, cancellation, and context management.

Conversations Endpoints

- Conversation resources group input and output items across turns.
- Common conversation endpoints include:
  - `POST /v1/conversations`
  - `GET /v1/conversations/{conversation_id}`
  - `POST /v1/conversations/{conversation_id}`
  - `DELETE /v1/conversations/{conversation_id}`
  - `POST /v1/conversations/{conversation_id}/items`
  - `GET /v1/conversations/{conversation_id}/items`
  - `GET /v1/conversations/{conversation_id}/items/{item_id}`
  - `DELETE /v1/conversations/{conversation_id}/items/{item_id}`
- This looks like normal REST resource design.
- A conversation has child items.
- Items can represent messages, tool calls, and outputs.

Files

- File resources let you upload data for use by other endpoints.
- Common file endpoints include:
  - `POST /v1/files`
  - `GET /v1/files`
  - `GET /v1/files/{file_id}`
  - `DELETE /v1/files/{file_id}`
  - `GET /v1/files/{file_id}/content`
- Files are used by workflows such as:
  - fine-tuning
  - batch processing
  - retrieval
  - assistants or legacy workflows
  - tool-based model interactions
- File upload requests usually use `multipart/form-data`, not plain JSON.

Models

- Model endpoints let you inspect model resources available to your account.
- Common model endpoints include:
  - `GET /v1/models`
  - `GET /v1/models/{model}`
  - `DELETE /v1/models/{model}` for supported fine-tuned models
- Most apps hard-code or configure the model they want.
- Listing models is useful for debugging account access and availability.
- Model capabilities differ, so endpoint support and parameters may vary by model.

Batch API

- Batch processing is for large groups of offline requests.
- Common endpoints include:
  - `POST /v1/batches`
  - `GET /v1/batches/{batch_id}`
  - `GET /v1/batches`
  - `POST /v1/batches/{batch_id}/cancel`
- Batch requests are useful when the user is not waiting interactively.
- Examples:
  - classify thousands of records
  - summarize many documents
  - generate labels or metadata offline
- This is not the right design for live chat.

Fine-Tuning

- Fine-tuning creates a custom model variant from training data.
- Common fine-tuning endpoints include:
  - `POST /v1/fine_tuning/jobs`
  - `GET /v1/fine_tuning/jobs/{fine_tuning_job_id}`
  - `GET /v1/fine_tuning/jobs`
  - `POST /v1/fine_tuning/jobs/{fine_tuning_job_id}/cancel`
  - `GET /v1/fine_tuning/jobs/{fine_tuning_job_id}/events`
- Fine-tuning is not the first tool to reach for.
- Many applications should start with better prompting, retrieval, or tool use.
- Fine-tuning is useful when you need repeated behavior that examples can teach.

Error Handling

- OpenAI API calls can fail for normal API reasons:
  - missing or invalid API key
  - malformed request body
  - unsupported parameter for a model
  - rate limit
  - timeout
  - service error
  - content or safety-related refusal
- Your backend should catch exceptions and return an application-appropriate error.
- Do not pass raw provider error details directly to users if they expose internals.
- Log enough detail to debug the problem.

Request IDs

- API responses include request identifiers in headers.
- Request IDs are useful for debugging.
- In production systems, log the OpenAI request ID along with your application request ID.
- This helps connect:
  - a user's failed request
  - your FastAPI route logs
  - the OpenAI API request
  - provider support or dashboard debugging
- This is the same reason you might log request IDs in any distributed system.

Rate Limits

- Rate limits protect the service and vary by account, model, and project.
- Common rate-limit dimensions include requests and tokens.
- Your backend should expect rate limit errors.
- For interactive apps:
  - show a friendly “try again soon” message
  - avoid automatic retry storms
  - apply your own per-user limits
- For background work:
  - use queues
  - use backoff
  - retry only when safe

Tokens and Cost

- Model APIs usually bill around input and output token usage.
- A token is roughly a chunk of text, not exactly a word.
- Long prompts cost more.
- Long outputs cost more.
- Conversation history can quietly become expensive if you resend all prior messages every turn.
- Good applications manage context deliberately:
  - keep only relevant history
  - summarize old context
  - use retrieval when the source material is large
  - set output limits when appropriate

Timeouts and Retries

- A model call may take longer than a normal database query.
- Your backend should set reasonable timeouts.
- Retry only when it is safe.
- Retrying a generation request can create duplicate work or different output.
- For idempotent read-like calls, retries are usually simpler.
- For user-visible generation, it may be better to show an error and let the user retry.

Design Pattern: Chat App

- Browser route:
  - `GET /chat`
  - `POST /api/chat`
- Backend responsibilities:
  - authenticate the user
  - load recent user-specific history
  - call `POST /v1/responses`
  - store user and assistant messages
  - return the assistant response
- OpenAI responsibilities:
  - generate the model response
  - optionally use tools
  - report usage and status

Design Pattern: Document Q&A

- User uploads documents.
- Backend stores files and metadata.
- Backend creates embeddings or uploads files into a vector store.
- User asks a question.
- Backend calls Responses with file search or retrieval context.
- Model answers using relevant retrieved content.
- Backend stores the answer and citations if needed.
- Useful endpoint families:
  - Files
  - Vector Stores
  - Responses
  - Embeddings, depending on the retrieval design

Design Pattern: Data Extraction

- User submits text or a file.
- Backend calls Responses with a structured output schema.
- Model returns JSON matching the desired fields.
- Backend validates the JSON.
- Backend writes clean data into a database.
- Useful endpoint:
  - `POST /v1/responses`
- Optional endpoint:
  - `POST /v1/moderations` if input safety classification is required

Design Pattern: Offline Processing

- User or administrator uploads a dataset.
- Backend builds a batch of model requests.
- Backend submits the batch.
- Backend checks status later.
- Backend downloads or reads the results.
- Useful endpoint families:
  - Files
  - Batches
  - Responses or another supported request type inside the batch
- This avoids tying up a live HTTP request for a long-running job.