Codebase to Tutorial

Open Source 11.1K

Health-Tracker

Siddharthakhandelwal/Health-Tracker

english

gemini-2.5-flash

03b7bde

Aug 3, 2025

Chapters

Overview of Health-Tracker

Chapter 1: FastAPI Application

Chapter 2: API Endpoints

Chapter 3: Request Handling Pipeline

Chapter 4: AI Integration (Gemini Model)

Chapter 5: Prompt Generation

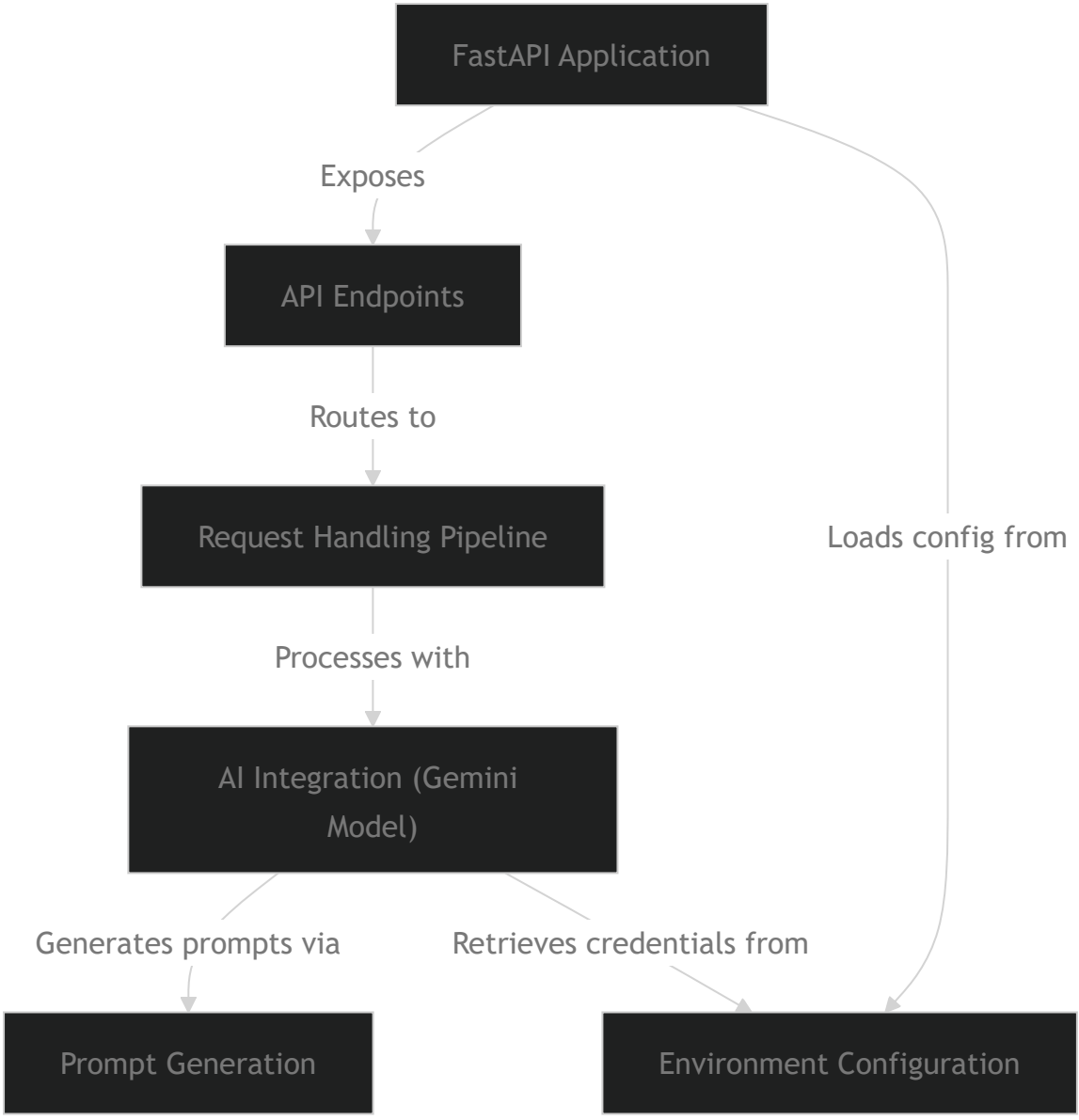
Chapter 6: Environment Configuration

Show Raw Markdown

Tutorial: Health-Tracker

The Health-Tracker is a **smart application** that helps you keep track of your daily health activities like *water intake*, *gym workouts*, and *food consumption*. It uses **Artificial Intelligence (AI)**, specifically Google's Gemini model, to process your data and give you *personalized feedback*, *motivation*, and *health insights* through simple web interactions.

Visual Overview



Chapters

1. FastAPI Application
2. API Endpoints
3. Request Handling Pipeline
4. AI Integration (Gemini Model)
5. Prompt Generation
6. Environment Configuration

Terms of Service Privacy Policy

Health-Tracker

Siddharthakhandelwal/Health-Tracker

english

gemini-2.5-flash

03b7bde

Aug 3, 2025

Chapters

Overview of Health-Tracker

Chapter 1: FastAPI Application

Chapter 2: API Endpoints

Chapter 3: Request Handling Pipeline

Chapter 4: AI Integration (Gemini Model)

Chapter 5: Prompt Generation

Chapter 6: Environment Configuration

Chapter 1: FastAPI Application

Imagine you're using a super helpful app on your phone called "Health Tracker." You just drank a glass of water, and you want to record it. You tap a button, type "1 glass," and hit "Save." What happens next? How does the app know what to do with your water intake data?

This is where the **FastAPI Application** comes in!

Think of our Health Tracker as a big, bustling restaurant. You, the user, are a customer. The "FastAPI Application" is like the restaurant's **main control panel** or the **central kitc management system**. Its job is to:

1. **Listen for orders:** It constantly waits for requests from your phone app (or other p the Health Tracker system).

2. **Understand the order:** It figures out what you want to do (e.g., record water, log a gym session, track food).

3. **Process the order:** It sends your request to the right "chefs" or "departments" inside the system to handle it.

4. **Send back a response:** It prepares a reply, like "Got it! Good job drinking water!" or "Here's some advice for your gym session!"

Without this central control panel, our Health Tracker wouldn't know how to receive your data, process it, or send you helpful responses.

What is FastAPI?

FastAPI is like a **high-speed construction kit** for building web APIs. An **API (Application Programming Interface)** is simply a way for different computer programs to talk to each other. It's like a menu in our restaurant analogy – it lists all the "dishes" (actions) the server can perform and what information it needs for each.

FastAPI helps us build this "main control panel" very efficiently. It's known for being fast and easy to use, making it perfect for our Health Tracker.

Setting Up Our FastAPI Application

Let's look at how our Health Tracker project sets up this central control panel. All the magic starts in the `main.py` file.

First, we need to tell Python that we want to use the FastAPI library:

```
from fastapi import FastAPI

# This line loads settings from a special file (we'll learn more about this later!)
from dotenv import load_dotenv
load_dotenv()

```

The most important line here is `from fastapi import FastAPI`. This imports the necessary tools. The `load_dotenv()` part helps us manage secret keys later on, but don't worry about it for now!

Next, we create our FastAPI application:

```
app = FastAPI()

```

This line `app = FastAPI()` is like opening our restaurant for business! The `app` variable now holds our entire web server application.

We also have a small section for `CORS`:

```
# Enable CORS (for cross-platform/browser access)
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],  # Allows connections from anywhere (for development)
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

```

This piece of code (called "middleware") is like setting up a friendly greeter at the restaurant's door. It helps make sure that requests coming from different places (like your phone app or a website) are allowed to talk to our server. For now, `allow_origins=["*"]` means it's super friendly and lets everyone in, which is great for testing!

Finally, to make our server actually run and listen for requests, we use `uvicorn`:

```
import uvicorn

if __name__ == "__main__":
    uvicorn.run("main:app", host="0.0.0.0", port=8000, reload=True)

```

This part is like hiring a manager for our restaurant. `uvicorn` is a super-fast server that actually runs our FastAPI application (`main:app`).

- `host="0.0.0.0"` means it's accessible from anywhere on your network.
- `port=8000` means it listens for requests on port 8000 (think of it as a specific door number for the server).

- `reload=True` is handy during development, as it automatically restarts the server when you make changes to your code.

To "use" the FastAPI application, you simply run this `main.py` file.

How to Run It:

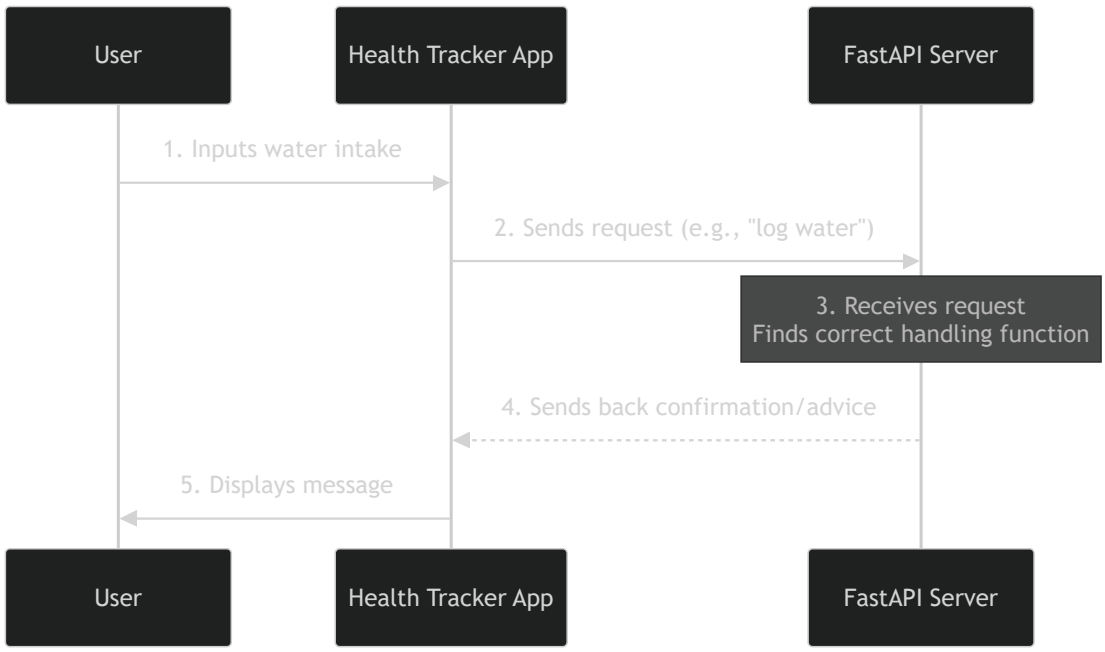
Open your terminal or command prompt, navigate to your project folder, and type:

```
python main.py

```

When you run this, you'll see messages from Uvicorn, indicating that your FastAPI application (our central control panel!) has started and is now listening for requests!

How It All Works (Under the Hood)

Let's trace what happens when you log your water intake:



1. **User Inputs Water:** You type "1 glass" into your Health Tracker app.
2. **App Sends Request:** Your app packages this information and sends it over the ur running **FastAPI Server**.

3. **FastAPI Server Receives:** The `uvicorn` part of our FastAPI application catches this incoming request. Our FastAPI application then looks at the request and figures out *what kind* of request it is (e.g., is it for water, gym, or food?). It then passes it to the correct internal function to handle it. This process of figuring out which function should handle a request is called "routing," and we'll learn more about it in API Endpoints.

4. **FastAPI Server Processes & Responds:** After processing your water intake (which might involve some smart AI suggestions, as we'll see in AI Integration (Gemini Model)), the FastAPI server sends a reply back to your app.

5. **App Displays Message:** Your app receives the reply and shows you a friendly message, like "Great job! Keep hydrating!"

Essentially, the FastAPI Application is the central hub that makes sure all these communication steps happen smoothly.

Conclusion

In this chapter, we learned that the **FastAPI Application** is the central "brain" or "control panel" of our Health Tracker project. It's built using the FastAPI framework, which helps us create a robust web server that listens for requests, processes them, and sends back responses. We saw how `app = FastAPI()` initializes our application and how `uvicorn.run()` makes it start listening for incoming messages.

Now that we understand what the central server is, let's dive into how it organizes what it can do. In the next chapter, we'll explore **API Endpoints**, which are like the specific "menu items" or "services" our FastAPI application offers to users.

Next Chapter: API Endpoints

Generated by AI Codebase Knowledge Builder. **References**: [1], [2]

Health-Tracker

Siddharthakhandelwal/Health-Tracker

english

gemini-2.5-flash

03b7bde

Aug 3, 2025

Chapters

Overview of Health-Tracker

Chapter 1: FastAPI Application

Chapter 2: API Endpoints

Chapter 3: Request Handling Pipeline

Chapter 4: AI Integration (Gemini Model)

Chapter 5: Prompt Generation

Chapter 6: Environment Configuration

Chapter 2: API Endpoints

In Chapter 1: FastAPI Application, we learned that our Health Tracker's **FastAPI Application** is like the central "control panel" or "kitchen management system" of a restaurant. It's always running, ready to take orders from your phone app.

But a big kitchen needs organization! How does the kitchen know if you're ordering water, a gym session log, or a food entry? This is where **API Endpoints** come in!

What are API Endpoints?

Imagine our Health Tracker restaurant has several different windows or counters, ea
specializing in a specific type of order:

- One window is *only* for ordering drinks.
- Another window is *only* for ordering main dishes.
- And yet another window is *only* for ordering desserts.

API Endpoints are exactly like these specific "service windows" or "doors" in our application. They are unique addresses that clients (like your phone app) use to interact with specific parts of our Health Tracker.

For example:

- `/water` is the window for sending water intake data.
- `/gym` is the window for sending gym activity logs.
- `/food` is the window for sending food logs.

Each of these "windows" (endpoints) knows exactly how to receive a specific type of data, process it according to its purpose (like sending it to our smart AI!), and then send back a helpful response.

Defining Endpoints in FastAPI

In FastAPI, we define these "windows" using special decorators (pieces of code that add functionality). Look at our `main.py` file:

```python
from fastapi import FastAPI # From Chapter 1!

app = FastAPI() # Our central control panel!

# ... other setup code (like CORS) ...

@app.post("/water")
async def water_endpoint(request: Request):
    # This function handles requests to the /water endpoint
    pass # We'll see what's inside soon!

@app.post("/gym")
async def gym_endpoint(request: Request):
    # This function handles requests to the /gym endpoint
    pass # More details below!

@app.post("/food")
async def food_endpoint(request: Request):
    # This function handles requests to the /food endpoint
```

```
    pass # Let's dive deeper!

# ... uvicorn run command ...

```

Let's break down one of these definitions:

```
@app.post("/water")
async def water_endpoint(request: Request):
    # This function gets called when someone sends data to /water
    # It processes the water intake data
    pass

```

- `@app.post("/water")` : This is the most important part for endpoints!
- `@app` : This tells FastAPI that we're adding something to our `app` (our main FastAPI application).
- `.post` : This indicates the "HTTP Method." Think of it as the *type* of interaction. `POST` is used when you want to *send* new data to the server (like logging new water intake). There are other methods like `GET` (to *get* data), `PUT` (to *update* data), `DELETE` (to *remove* data), but for our Health Tracker, we primarily `POST` new activity.
- `("/water")` : This is the **path** or the specific address of this "window." When your app sends data to `http://localhost:8000/water` , this is the code that gets activated.
- `async def water_endpoint(request: Request):` : This defines the actual Python function that will run when a request arrives at the `/water` endpoint using the `POST` method.
- `async def` : This means the function can run "asynchronously," which is fancy talk for saying it can do other things while waiting for slow operations (like talking to the AI). Don't worry too much about `async` for now, just know it's a good practice in modern web applications.
- `water_endpoint` : This is just the name we gave to our function. You could call it anything, but it's good practice to make it descriptive.
- `(request: Request)` : This `request` object holds all the information sent by the client, like the water amount you logged.

We have similar definitions for `/gym` and `/food` , each linking to its own specific func

Endpoint Path	HTTP Method	What it's for
`/water`	POST	Log new water intake
`/gym`	POST	Log new gym activity
`/food`	POST	Log new food consumption

How Clients Use Endpoints

Your phone app (or our `test_api.py` script) knows these specific "window addresses." When you want to log your water, the app sends data specifically to the `/water` endpoint.

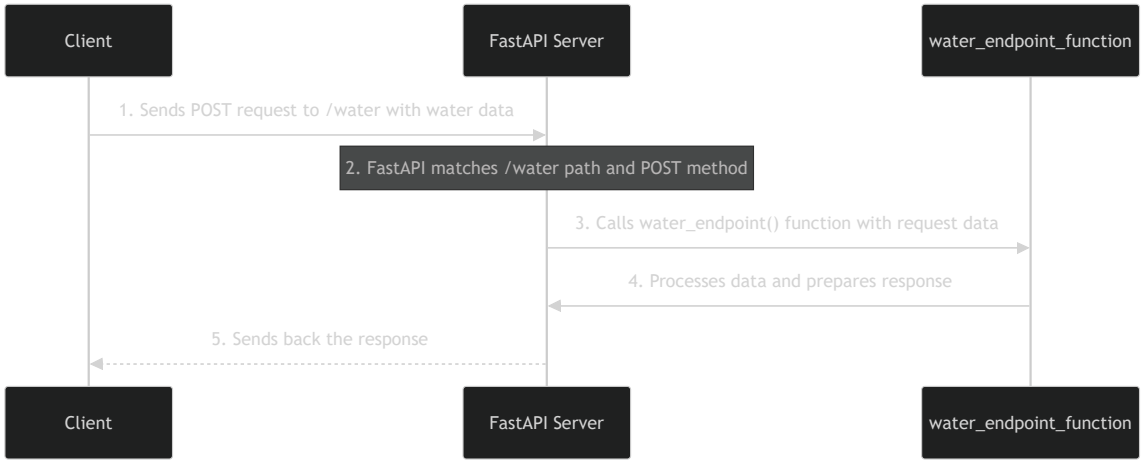
Look at how our `test_api.py` script sends data:

```python
import requests

BASE_URL = "http://localhost:8000"

endpoints = ["water", "gym", "food"]
sample_data = {
    "water": {"amount_liters": 2, "time": "2024-06-01T10:00:00"},
    # ... other sample data ...
}

for endpoint in endpoints:
    url = f"{BASE_URL}/{endpoint}" # This builds the full URL, e.g., http://localho
    data = sample_data[endpoint]
    response = requests.post(url, json=data) # This sends a POST request with JSON
    print(f"Response from /{endpoint}: {response.status_code}")
    print(response.json())

```

This `test_api.py` script is acting like a simple client. It constructs the full address (like `http://localhost:8000/water`) and then uses `requests.post()` to send data specifically to that endpoint.

How It All Works (Under the Hood)

Let's trace what happens when `test_api.py` sends water data to the `/water` endpoint.



1. **Client Sends Request:** Our `test_api.py` script sends a `POST` request to `http://localhost:8000/water` with your water intake details (e.g., `{"amount_liters": 2}`).
2. **FastAPI Server Receives & Routes:** Our running FastAPI application (from Chapter 1: FastAPI Application) receives this request. It looks at the path (`/water`) and the method (`POST`). It then "routes" (sends) this incoming request to the specific Python function we've linked to that endpoint.
3. **Endpoint Function Handles Request:** The `water_endpoint` function is now activated. It gets access to the data sent by the client.
4. **Processes and Prepares Response:** Inside `water_endpoint`, our application will do its magic: it extracts the water data, sends it to the AI for suggestions (we'll see this in Chapter 4: AI Integration (Gemini Model)), and prepares a message back to the client.
5. **Sends Back Response:** The FastAPI server sends this prepared message back to the `test_api.py` script, which then prints it for you to see!

Looking at the Real Code

In our `main.py`, the actual functions linked to these endpoints are quite simple, thanks to a helper function:

```
# From main.py
# ... other code ...

# Generic handler (We'll dive into this in Chapter 3!)
async def handle_request(request: Request, endpoint: str):
    try:
        data = await request.json() # Get the data sent by the client
        # This is where we talk to the AI, but it's hidden inside send_to_gemini!
```

```
        response_text = send_to_gemini(data, endpoint)
        return JSONResponse(content={"message": response_text})
    except Exception as e:
        raise HTTPException(status_code=400, detail=str(e))

@app.post("/water")
async def water_endpoint(request: Request):
    # This simply calls our generic handler for "water"
    return await handle_request(request, "water")

@app.post("/gym")
async def gym_endpoint(request: Request):
    # This simply calls our generic handler for "gym"
    return await handle_request(request, "gym")

@app.post("/food")
async def food_endpoint(request: Request):
    # This simply calls our generic handler for "food"
    return await handle_request(request, "food")

# ... uvicorn run command ...

```

As you can see, the `@app.post("/path")` lines are what *define* the endpoints and link them to our `water_endpoint`, `gym_endpoint`, and `food_endpoint` functions. These functions then call `handle_request`, which is a clever way to reuse code for all three, but the core idea is that each path (`/water`, `/gym`, `/food`) has a dedicated entry point.

Conclusion

In this chapter, we learned that **API Endpoints** are like specialized "service windows" in our FastAPI application. Each endpoint (like `/water`, `/gym`, `/food`) has a unique address and handles a specific type of data input using HTTP methods like `POST`. We saw how `@app.post("/path")` is used to define these entry points and link them to Python functions that process the incoming requests.

Now that we understand how requests arrive at specific "windows," the next step is to understand what happens *inside* that window. In the next chapter, we'll explore the **Request Handling Pipeline**, which explains how the data is actually processed after it hits

Next Chapter: Request Handling Pipeline

Generated by AI Codebase Knowledge Builder. **References**: [1], [2]

Health-Tracker

Siddharthakhandelwal/Health-Tracker

english

gemini-2.5-flash

03b7bde

Aug 3, 2025

Chapters

Overview of Health-Tracker

Chapter 1: FastAPI Application

Chapter 2: API Endpoints

Chapter 3: Request Handling Pipeline

Chapter 4: AI Integration (Gemini Model)

Chapter 5: Prompt Generation

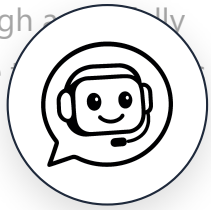
Chapter 6: Environment Configuration

Show Raw Markdown

Chapter 3: Request Handling Pipeline

In Chapter 2: API Endpoints, we learned that our Health Tracker has special "service windows" (API Endpoints) like `/water`, `/gym`, and `/food` that are ready to receive your activity data. But once your data *arrives* at one of these windows, what happens next? How does it get processed? This is where the **Request Handling Pipeline** comes in!

Imagine our Health Tracker is a busy factory. When an order (your request) comes in, it doesn't just magically turn into a finished product (an AI response). Instead, it goes through a carefully organized assembly line. Each station on the line does a specific job to make sure the final product is perfect every time.



The **Request Handling Pipeline** is exactly like this assembly line for every request that comes into our Health Tracker. It's the systematic, step-by-step process that ensures your request, whether it's logging water or food, is handled consistently and correctly from start to finish.

Let's stick with our example: you log "1 glass of water" into your app. This request hits the `/water` endpoint. The pipeline then takes over to:

1. **Receive** your water data.
2. **Extract** just the important bits (like "1 glass").
3. Pass it to our **smart AI** for intelligent processing (e.g., to get a motivational message).
4. **Prepare** the AI's response.
5. **Send** that response back to your app.

The Stages of the Pipeline

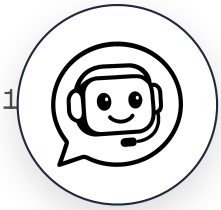
Think of the **Request Handling Pipeline** as these distinct stages for every incoming request:

Stage Name	What Happens	Analogy
1. Receive Request	The FastAPI server catches the incoming data from your app.	Customer's order arrives at the service window.
2. Extract Data	The raw data from the request is carefully pulled out and understood.	The order taker reads the specific items.
3. AI Processing	The extracted data is sent to our smart AI (Gemini model) for analysis/response.	The order goes to the expert chef.
4. Prepare Response	The AI's output is formatted into a clear, structured message.	The chef's dish is plated nicely.
5. Send Response	The formatted message is sent back to your app, completing the interaction.	The waiter delivers the dish to the customer.

How It Solves Our Water Logging Use Case

Let's trace our "log 1 glass of water" example through this pipeline:

1. **You:** Tap "Log Water" in your Health Tracker app and enter the amount.
2. **Your App:** Sends a `POST` request with JSON data (e.g., `{"amount_liters": 1` `http://localhost:8000/water` address.



3. **FastAPI Server (Stage 1: Receive Request):** Our running FastAPI application (from Chapter 1: FastAPI Application) catches this incoming request at the `/water` endpoint.

4. **FastAPI Server (Stage 2: Extract Data):** The server looks inside the request and carefully pulls out the important information: `{"amount_liters": 1}`.

5. **FastAPI Server (Stage 3: AI Processing):** It then takes `{"amount_liters": 1}` and asks our smart AI (the Gemini model), "What advice or motivation should I give the user who just logged 1 liter of water?" (We'll explore this in detail in Chapter 4: AI Integration (Gemini Model)).

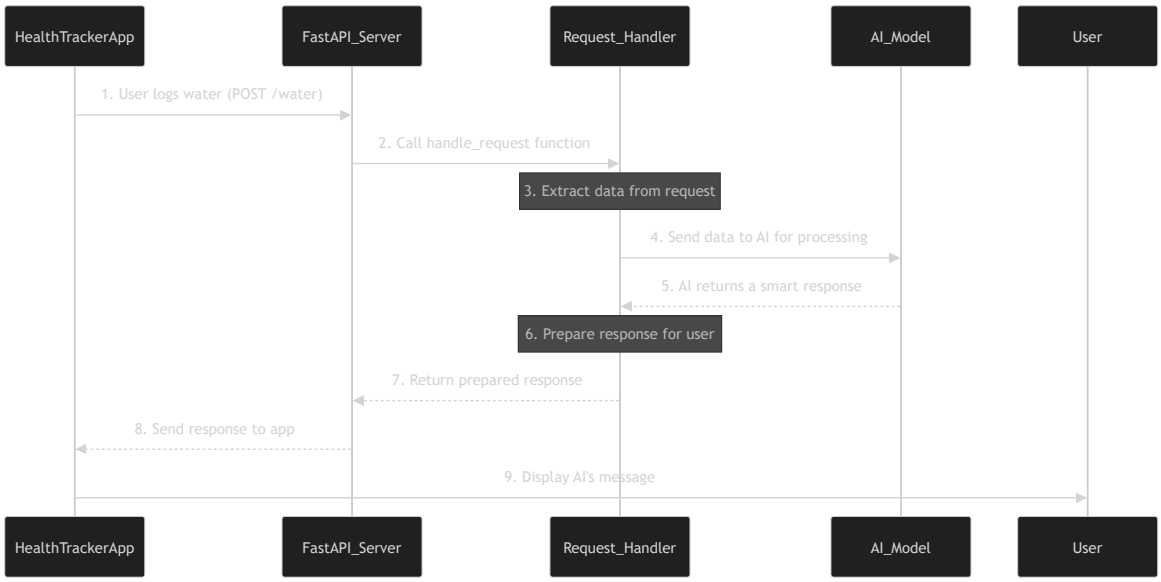
6. **FastAPI Server (Stage 4: Prepare Response):** Once it gets the AI's suggestion (e.g., "Good job! You've had 1L. Aim for 2L today!"), it formats this message into a standard structure.

7. **FastAPI Server (Stage 5: Send Response):** Finally, it sends this formatted message back to your app.

8. **Your App:** Receives the message and displays the AI's encouraging words to you!

How It All Works (Under the Hood)

Let's visualize this step-by-step flow with a diagram:



You might recall from Chapter 2: API Endpoints that our `/water`, `/gym`, and `/food` endpoints all call a special function named `handle_request`. This `handle_request` function is where our "assembly line" truly begins for every single request!

Let's look at the core of our `main.py` file, focusing on this `handle_request` func

```python
# From main.py

# This function acts as our request handling pipeline
async def handle_request(request: Request, endpoint: str):
    try:
        # Stage 1 & 2: Receive request and Extract Data
        data = await request.json() # Pulls out the JSON data sent by the user

        # Stage 3: AI Processing
        # This function sends data to our AI and gets a smart reply
        response_text = send_to_gemini(data, endpoint) # More in Chapter 4!

        # Stage 4 & 5: Prepare and Send Response
        # Formats the AI's reply nicely and sends it back
        return JSONResponse(content={"message": response_text})
    except Exception as e:
        # If anything goes wrong, we send an error message
        raise HTTPException(status_code=400, detail=str(e))

```

This `handle_request` function contains the steps of our pipeline. Let's break down each important part:

1. Receiving and Extracting Data (`data = await request.json()`)

When a request arrives at an endpoint (like `/water`), FastAPI automatically gives our `handle_request` function a `request` object. This `request` object contains everything the client sent.

The very first step in our pipeline is to get the actual information (like `{"amount_liters": 1}`) out of this `request` object.

```python
# Inside handle_request function in main.py

# This line handles Stage 1 (Receiving) and Stage 2 (Extracting)
data = await request.json()

```

This line `data = await request.json()` does two important things:

- It uses `await` because receiving and processing data from the internet can take a tiny bit of time, and `await` allows our server to do other things while waiting.
- It extracts the JSON-formatted data sent by your app (e.g., `{"amount_liters": 1}`) and stores it in a variable called `data`. Now we have the user's input, ready for the next stage!

2. AI Processing (`response_text = send_to_gemini(...)`)

Once we have the user's data, the core of our Health Tracker is to get smart advice or motivation. This is where our AI comes in!

```
# Inside handle_request function in main.py

# This line handles Stage 3 (AI Processing)
# The 'endpoint' tells us if it's water, gym, or food data
response_text = send_to_gemini(data, endpoint)

```

This line calls another function, `send_to_gemini`. This function is specifically designed to:

- Take the `data` (e.g., `{"amount_liters": 1}`).
- Know which `endpoint` it came from (e.g., `"water"`), so it can ask the AI the right kind of question.
- Communicate with the Google Gemini AI model.
- Receive the AI's intelligent response (like "Great start! Aim for 2.5L daily.").

We will dive deep into how `send_to_gemini` works and how it talks to the AI in Chapter 4: AI Integration (Gemini Model) and Chapter 5: Prompt Generation.

3. Preparing and Sending Response (`return JSONResponse(...)`)

Finally, once we have the AI's response, we need to send it back to the user's app in a way the app can understand.

```
# Inside handle_request function in main.py

# This line handles Stage 4 (Prepare) and Stage 5 (Send)
return JSONResponse(content={"message": response_text})

```

This line creates a `JSONResponse`:

- `JSONResponse` is a special object from FastAPI that helps us send data back in JSON format, which is very common for web APIs.
- `content={"message": response_text}` puts the AI's message into a structured format. So, your app will receive something like `{"message": "Good job! Aim for 2.5L daily."}`. This makes it easy for your app to display the message to you.

If anything goes wrong during this entire process (for example, if the data is not in the correct format or the AI connection fails), the `try...except` block catches the error, and we send back an `HTTPException` with a `400 Bad Request` status code, letting the client know something went wrong.

Conclusion

In this chapter, we unpacked the **Request Handling Pipeline**, the systematic "assembly line" that processes every user request in our Health Tracker. We saw how a request moves from being received, through data extraction, intelligent AI processing, and finally, response preparation and delivery. The `handle_request` function in `main.py` is the core of this pipeline, ensuring a consistent and robust flow for all interactions.

Now that we understand the full journey of a request, it's time to zoom in on the most exciting part: the **AI Integration**. In the next chapter, we'll discover how our Health Tracker communicates with the powerful AI Integration (Gemini Model) to provide personalized advice and motivation.

Next Chapter: AI Integration (Gemini Model)

Generated by AI Codebase Knowledge Builder. **References**: [1]

Health-Tracker

Siddharthakhandelwal/Health-Tracker

english

gemini-2.5-flash

03b7bde

Aug 3, 2025

Chapters

Overview of Health-Tracker

Chapter 1: FastAPI Application

Chapter 2: API Endpoints

Chapter 3: Request Handling Pipeline

Chapter 4: AI Integration (Gemini Model)

Chapter 5: Prompt Generation

Chapter 6: Environment Configuration

Chapter 4: AI Integration (Gemini Model)

In Chapter 3: Request Handling Pipeline, we followed your health data as it traveled through our FastAPI server's "assembly line," from the moment it was received to when it was ready for the final step: getting smart advice. But where does this "smart advice" come from? How does our Health Tracker actually become intelligent?

This is where **AI Integration (Gemini Model)** comes in!

Imagine your Health Tracker isn't just a basic logbook, but also has a super smart, **health coach** built right into it. When you log your activities – whether it's a glass of [illegible] gym

session, or a meal – this "coach" instantly analyzes your data and gives you personalized tips, motivation, or helpful estimates.

This "coach" is powered by **Google's Gemini AI model**. It's the "brain" of our Health Tracker, making it intelligent and truly helpful.

What is Gemini AI Integration?

Our project uses Google's **Gemini AI** model. Think of Gemini as an incredibly powerful, knowledgeable assistant that can understand questions and generate helpful, human-like responses.

In our Health Tracker, the **AI Integration (Gemini Model)** abstraction represents the part of our system responsible for:

1. **Sending your health data securely** to the Gemini AI model.
2. **Asking Gemini smart questions** based on your input.
3. **Receiving intelligent responses** from Gemini (like a motivational message, a personalized health tip, or a calorie estimate).
4. **Bringing that valuable insight back to you** through the app.

This is how our Health Tracker moves beyond simply recording data to providing actionable and encouraging feedback.

Our Use Case: Getting AI Feedback for Water Intake

Let's go back to our example: you log "1 glass of water."

- Without AI: The app might just say "Water logged."
- **With AI (Gemini Model):** The app can now say, "Great start! You've had 1L. Aim for 2.5L today for optimal hydration!" – much more encouraging and helpful!

This AI integration is what makes our Health Tracker truly interactive and personalized.

The Core: `send_to_gemini` Function

The central piece of code that handles all the communication with the Gemini AI is a function called `send_to_gemini`. You saw it briefly in Chapter 3: Request Handling Pipeline as the part of the `handle_request` pipeline that does the "AI Processing."

Let's look at it in `main.py`:

```python
# From main.py

# Gemini call handler
def send_to_gemini(data: dict, endpoint: str) -> str:
    # ... (code to talk to Gemini) ...
    pass

```

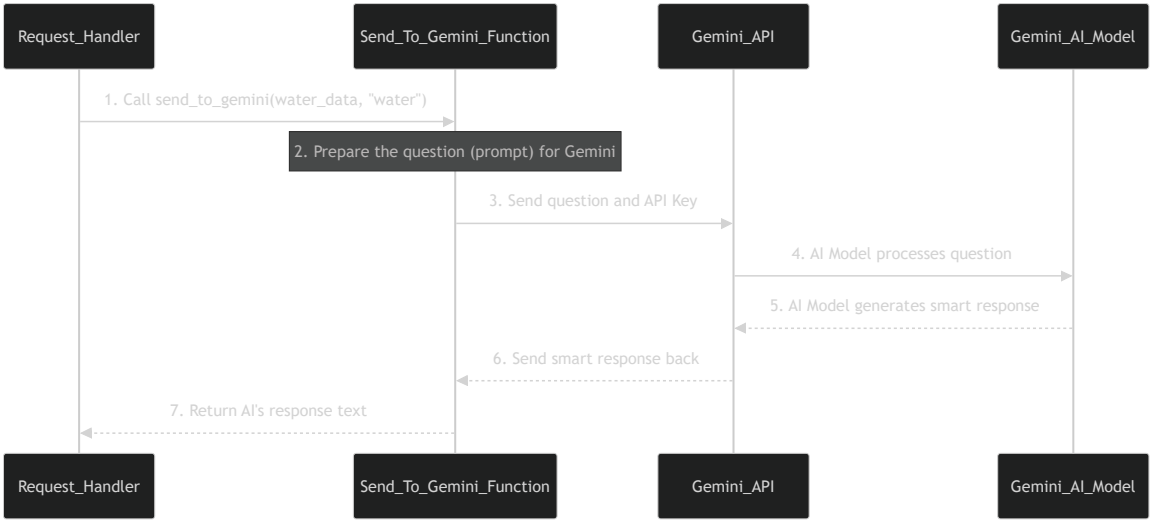
This function takes two main pieces of information:

- `data` : This is the actual health information you logged (e.g., `{"amount_liters": 1}`).
- `endpoint` : This tells us *what kind* of data it is (e.g., "water", "gym", or "food"), so Gemini can give relevant advice.

It then does all the heavy lifting to talk to the AI and return Gemini's response as a text message.

How It All Works (Under the Hood)

Let's trace what happens when the `handle_request` function (our "assembly line" manager) calls `send_to_gemini` with your water intake data:



1. **Call to `send_to_gemini`** : The `handle_request` function in our pipeline passes your water data and tells `send_to_gemini` that it's "water" data.
2. **Prepare the Question**: Inside `send_to_gemini` , our code takes your `water_da` and creates a specific question (called a "prompt") for Gemini. For example, "A person drank 1 lit[illegible]ater.

Give them a motivational message and how much more they should drink." We'll learn how these questions are built in Chapter 5: Prompt Generation.

3. **Send Question to Gemini API**: Using a special Python library, `send_to_gemini` securely sends this question to Google's Gemini servers (the "Gemini API"), along with a secret key that identifies our application.

4. **Gemini AI Processes**: Google's servers receive the question and send it to the powerful Gemini AI model.

5. **Gemini Generates Response**: The Gemini AI processes the question and generates an intelligent, helpful response, like "Great job! You're on track for hydration."

6. **Response Back to Our Server**: The Gemini API sends this response back to our `send_to_gemini` function.

7. **Return AI's Response**: Finally, `send_to_gemini` extracts the plain text message from Gemini's response and sends it back to the `handle_request` function, which then prepares it to be sent back to your app!

Diving into the `send_to_gemini` Code

Let's look at the actual code in `main.py` that makes this happen:

```python
# From main.py

# Gemini call handler
def send_to_gemini(data: dict, endpoint: str) -> str:
    try:
        # Step 1: Get our secret key for Gemini
        api_key = os.getenv("GEMINI_API_KEY") # We'll learn about this in Chapter
        if not api_key:
            raise EnvironmentError("Gemini API key not found.")

        # Step 2: Tell the Gemini library our secret key
        genai.configure(api_key=api_key)

        # Step 3: Choose which Gemini model to use
        model = genai.GenerativeModel("gemini-2.5-flash")

        # Step 4: Build the specific question for Gemini
        prompt = build_prompt(data, endpoint) # More in Chapter 5!

        # Step 5: Send the question to Gemini and get a response
        response = model.generate_content(prompt)

```

```
        # Step 6: Extract and return Gemini's text answer
        return response.text
    except Exception as e:
        print(f"Error communicating with Gemini: {e}")
        return "Sorry, there was an error processing your request."

```

Let's break down the important lines:

1. Getting the Secret Key (`api_key = os.getenv( ... )`)

```
api_key = os.getenv("GEMINI_API_KEY")
if not api_key:
    raise EnvironmentError("Gemini API key not found.")

```

To talk to Google's Gemini AI, we need a special "secret key" (like a password) that identifies our application. This line securely gets that key from our project's settings. We'll learn more about setting up these secret keys in Chapter 6: Environment Configuration. Without this key, Gemini won't know who we are!

2. Configuring the Gemini Library (`genai.configure( ... )`)

```
genai.configure(api_key=api_key)

```

`genai` is the Python library that helps us easily talk to Google's Gemini AI. This line is like telling the library, "Here's my secret key, now you're ready to communicate with Gemini!"

3. Choosing a Gemini Model (`model = genai.GenerativeModel( ... )`)

```
model = genai.GenerativeModel("gemini-2.5-flash")

```

Google offers different versions of its Gemini AI model. Some are faster, some are ul. `"gemini-2.5-flash"` is a good choice for quick, conversational responses, perfec Health Tracker's motivational messages. This line loads the specific "brain" we want to

4. Building the Question (`prompt = build_prompt( ... )`)

```
prompt = build_prompt(data, endpoint)

```

This is a crucial step! We can't just send raw user data to Gemini. We need to turn it into a clear, specific question that the AI can understand and respond to intelligently. This `build_prompt` function creates that question (e.g., "A user logged 2 liters of water. Give them a motivating message and suggest a daily goal."). We will explore exactly how these "prompts" are crafted in the next chapter, Chapter 5: Prompt Generation.

5. Talking to Gemini (`response = model.generate_content(prompt)`)

```
response = model.generate_content(prompt)

```

This is the moment of truth! This line sends our carefully crafted `prompt` (the question) to the Gemini AI model we selected. The `model.generate_content()` function handles all the complex communication behind the scenes and waits for Gemini to send back its intelligent response.

6. Getting Gemini's Answer (`return response.text`)

```
return response.text

```

Once Gemini sends back its response, it usually comes in a structured format. This line simply extracts the main text part of Gemini's answer (like "Excellent! Hydration is key..") so we can send it back to the user's app.

The `try ... except` block is there to catch any issues that might occur, like if there's no internet connection or if the Gemini service isn't available. In such cases, it will return a friendly error message instead of crashing.

Conclusion

In this chapter, we discovered that the **AI Integration (Gemini Model)** is the "bra Health Tracker, providing intelligent, personalized feedback. We learned that the `send_to_gemini` function in `main.py` is the core component that securely communicates with

Google's Gemini AI, sends it questions based on your health data, and receives smart responses. This is what transforms our simple tracker into a truly intelligent health coach.

Now that we understand *how* we talk to Gemini, the next logical step is to understand *what* we say to it. In the next chapter, we'll dive into **Prompt Generation**, where we'll learn the art of crafting the perfect questions to get the best advice from our AI coach.

Next Chapter: Prompt Generation

Generated by AI Codebase Knowledge Builder. **References**: [1], [2], [3]

Health-Tracker

Siddharthakhandelwal/Health-Tracker

english

gemini-2.5-flash

03b7bde

Aug 3, 2025

Chapters

Overview of Health-Tracker

Chapter 1: FastAPI Application

Chapter 2: API Endpoints

Chapter 3: Request Handling Pipeline

Chapter 4: AI Integration (Gemini Model)

Chapter 5: Prompt Generation

Chapter 6: Environment Configuration

Chapter 5: Prompt Generation

In Chapter 4: AI Integration (Gemini Model), we learned that our Health Tracker has a super-smart "health coach" powered by Google's Gemini AI. This coach gives you personalized tips and motivation based on your logged activities. We also saw that the `send_to_gemini` function is responsible for talking to this AI.

But how does our AI coach know what *kind* of advice to give? If you just send it "I ran for 45 minutes," how does it know you want a motivational message about exercise intensity or a recipe for a post-workout meal?

This is where **Prompt Generation** comes in!

Think of **Prompt Generation** as the "translator" or "scriptwriter" for our AI coach. It takes your raw, simple health data (like "I ran for 45 minutes" or "I ate a burger") and transforms it into a clear, specific question or instruction that the Gemini AI can understand perfectly. This ensures that the AI receives exactly the right context to provide accurate, helpful, and tailored advice for your water, gym, or food intake.

What is a "Prompt"?

In the world of AI, a **"prompt"** is essentially the question or instruction you give to an AI model. It's how you tell the AI what you want it to do or what information you need.

Imagine you're talking to a very smart friend. If you just say "water," your friend might be confused. Do you want a glass of water? Are you asking about the chemical composition of water? But if you say, "I just drank a glass of water. Can you give me a motivating message about hydration and suggest how much more I should drink today?", your friend knows exactly how to respond.

That clear, specific instruction is what a "prompt" is for an AI. It guides the AI to generate the exact kind of response you're looking for.

Why Do We Need Prompt Generation?

Our Health Tracker needs **Prompt Generation** for a few key reasons:

1. **Context is King:** The AI needs to know *what* the data is about (water, gym, food) and *what kind of response* we expect (motivation, calorie estimate, exercise advice).
2. **Specificity:** We want the AI to give consistent, relevant advice, not just random thoughts. Prompt generation makes sure the AI focuses on our specific health tracking needs.
3. **Efficiency:** A well-crafted prompt helps the AI understand quickly and respond accurately, saving time and ensuring quality.

Our Use Case: Crafting the Perfect Question

Let's revisit our "log water intake" example. When you send `{"amount_liters": 1}` for water:

- **Raw Data:** `{"amount_liters": 1}`
- **The Goal:** Get a motivational message and a daily water goal.
- **Without Prompt Generation:** The AI might not know what to do with just `{"amount_liters": 1}`.
- **With Prompt Generation:** We create a prompt like: "You have the following d… `{'amount_liters': 1}` of a person's water intake. Give the amount of water they sh… drink

today and a motivating line. Limit response to 25 words." Now, the AI knows exactly what to analyze and how to respond!

The Core: `build_prompt` Function

In Chapter 4: AI Integration (Gemini Model), we briefly saw a function called `build_prompt` inside `send_to_gemini`. This `build_prompt` function is our dedicated "scriptwriter" for the AI.

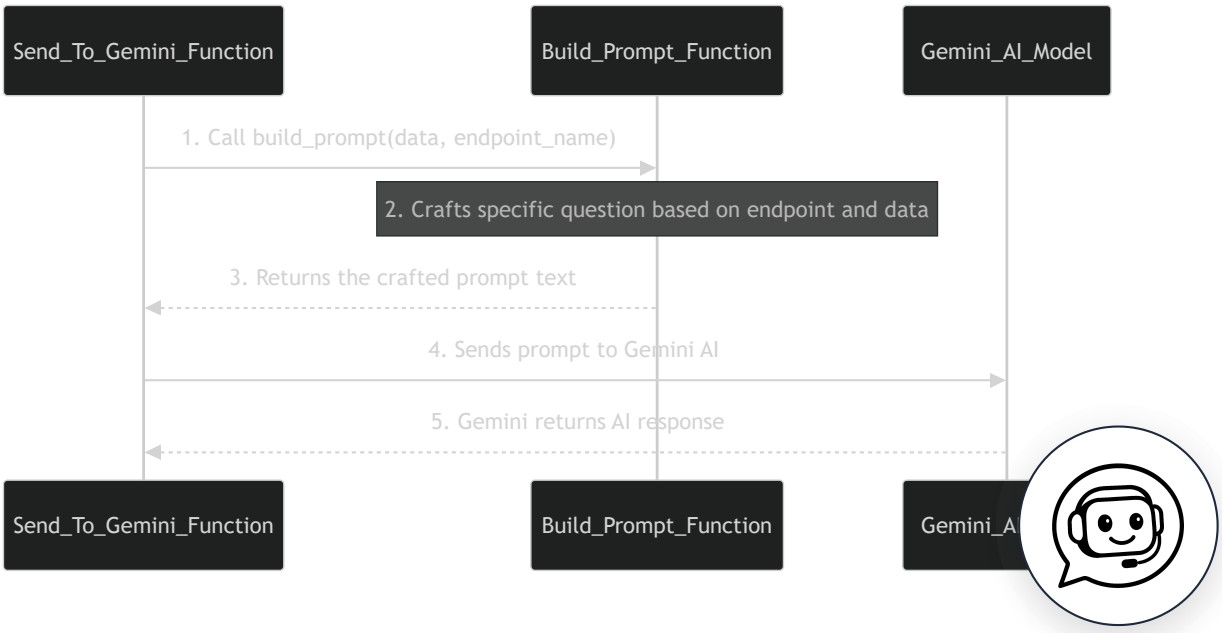
It takes the raw `data` (like `{"amount_liters": 1}`) and the `endpoint` name (like `"water"`, `"gym"`, or `"food"`) and uses them to create the perfect prompt string for Gemini.

```
# From main.py

# Helper: Create prompt based on endpoint
def build_prompt(data: dict, endpoint: str) -> str:
    # ... code inside this function ...
    pass # We'll break this down next!

```

This function is super important because it ensures that for every type of health activity, Gemini receives a tailored question.

How It All Works (Under the Hood)

Let's trace how the `build_prompt` function works within our AI communication flow:



1. **Call `build_prompt`**: The `send_to_gemini` function asks `build_prompt` to create a question, giving it the `data` (e.g., `{"amount_liters": 1}`) and the `endpoint` (e.g., `"water"`).

2. **Craft the Question**: Inside `build_prompt`, the function checks the `endpoint` type. Based on whether it's "water," "gym," or "food," it chooses a specific "script" or template for the question. It then inserts the actual `data` into this script.

3. **Return the Prompt**: The `build_prompt` function then sends the complete, well-formed question (the "prompt" string) back to the `send_to_gemini` function.

4. **Send to Gemini**: With the perfect question in hand, `send_to_gemini` securely sends this prompt directly to the Gemini AI model.

5. **Gemini Responds**: Gemini processes the prompt and sends back its intelligent answer, which then makes its way back to your app!

Diving into the `build_prompt` Code

Let's look at the actual code for `build_prompt` in `main.py` and break down how it crafts these questions:

```python
# From main.py

# Helper: Create prompt based on endpoint
def build_prompt(data: dict, endpoint: str) -> str:
    if endpoint == "water":
        return (
            f"You have the following data: {data} of a person's water intake. "
            "Give the amount of water they should drink today and a motivating line
            "Limit response to 25 words."
        )
    # ... other endpoint checks ...
    else:
        raise ValueError("Invalid endpoint")

```

The `build_prompt` function uses `if` and `elif` (short for "else if") statements to check the value of the `endpoint` variable.

1. Water Prompt Generation (`if endpoint == "water"`)

When `endpoint` is `"water"`, this part of the code runs:

```
# Inside build_prompt function in main.py
if endpoint == "water":
    return (
        f"You have the following data: {data} of a person's water intake. "
        "Give the amount of water they should drink today and a motivating line. "
        "Limit response to 25 words."
    )

```

- `f"You have the following data: {data} ..."` : This is an f-string, which allows us to easily embed the actual `data` (e.g., `{"amount_liters": 1}`) directly into the prompt string. This way, Gemini gets the specific amount.
- `"Give the amount of water they should drink today and a motivating line."` : This is the instruction for Gemini, telling it *what kind of information* we want back.
- `"Limit response to 25 words."` : This is a crucial instruction! It tells Gemini to keep its answer short and to the point, which is great for a quick app notification.

2. Gym Prompt Generation (`elif endpoint == "gym"`)

If the `endpoint` is `"gym"`, a different prompt is created:

```
# Inside build_prompt function in main.py
elif endpoint == "gym":
    return (
        f"You have the following data: {data} of a person's gym activity. "
        "Motivate based on their routine, suggest exercise intensity, and advise he
        "Limit response to 25 words."
    )

```

Again, the `data` (like `{"duration_minutes": 45, "activity_type": "running"}`) is embedded, and the instructions are tailored for gym activities: "Motivate based on their routine, suggest exercise intensity, and advise healthy eating."

3. Food Prompt Generation (`elif endpoint == "food"`)

And for `endpoint` being `"food"` :

```
# Inside build_prompt function in main.py
elif endpoint == "food":
    return (
        f"You have the following data: {data} of a person's food intake.A detail me
        "Encourage healthy meals and motivate them to eat better. "
        "Limit response to 25 words."
    )

```

Here, the instructions are for food intake: "calculate the user calories intake and give your response. Encourage healthy meals and motivate them to eat better."

What if it's an unknown endpoint?

```
# Inside build_prompt function in main.py
else:
    raise ValueError("Invalid endpoint")

```

If the `endpoint` is anything other than "water," "gym," or "food," the `build_prompt` function will raise an error. This is important for preventing our AI from getting confused by unexpected requests.

Examples of Prompt Generation in Action

Here's how `build_prompt` acts as our AI's "scriptwriter" for different inputs:

| Endpoint | Raw User Data (example) | Generated Prompt for Gemini ``` In this chapter, we learned that **Prompt Generation** is the strategic art of creating specific and tailored questions for the `send_to_gem` function to ask the Gemini AI. It acts as a bridge between your raw health data and the AI's intelligent responses, ensuring context and a helpful output. We understood how the `build_prompt` function leverages the `data` and `endpoint` to craft precise instructions for the AI, resulting in personalized advice for your water, gym, or food logs.

Now that we know how to configure our application and how to intelligently commun with the AI, the final chapter will guide you through setting up the necessary environm project.

Next Chapter: Environment Configuration

Generated by AI Codebase Knowledge Builder. **References**: [1]

Health-Tracker

Siddharthakhandelwal/Health-Tracker

english

gemini-2.5-flash

03b7bde

Aug 3, 2025

Chapters

Overview of Health-Tracker

Chapter 1: FastAPI Application

Chapter 2: API Endpoints

Chapter 3: Request Handling Pipeline

Chapter 4: AI Integration (Gemini Model)

Chapter 5: Prompt Generation

Chapter 6: Environment Configuration

Chapter 6: Environment Configuration

In our previous chapters, especially in Chapter 4: AI Integration (Gemini Model), we saw how our Health Tracker talks to Google's super-smart Gemini AI. Remember this line of code from `send_to_gemini`?

```
1  api_key = os.getenv("GEMINI_API_KEY")
2
```

We mentioned it was getting a "secret key" for Gemini. But where does this secret come from? And why don't we just write it directly into our code?

This is where **Environment Configuration** comes in!

Imagine you have a very important, secret password – maybe for your personal diary. Would you write that password directly on the first page of the diary for everyone to see? Probably not! You'd keep it somewhere safe, like a special, locked vault, separate from the diary itself.

Environment Configuration is exactly like this secure, external vault for sensitive information and settings in our Health Tracker project. Instead of embedding crucial details like our Gemini AI's "secret password" (API key) directly into the program's code, this mechanism allows our application to load these values from separate, safe files when it starts up.

Why Do We Need a "Secret Vault" (Environment Configuration)?

There are a few big reasons why keeping secrets separate is super important:

1. **Security (Most Important!)**: If you write your `GEMINI_API_KEY` directly in `main.py`, and then you share your code (for example, by putting it on a public website like GitHub), *everyone* will see your secret key! This is like shouting your password in a crowded room. Environment configuration keeps your secrets hidden.
2. **Flexibility**: What if you want to use a different Gemini key for testing versus when your app is used by real people? Or what if the key changes? With environment configuration, you just update one simple file, not the main code.
3. **Separation**: It helps keep your core program code clean and focused on *what* it does, not *what specific keys* it uses.

For our Health Tracker, the most important "secret" we're protecting is our `GEMINI_API_KEY`. Without it, our smart AI health coach can't talk to Google's Gemini!

The Magic of the `.env` File

The "special, safe file" we use for environment configuration is usually named `.env`. It's a simple text file that stores a list of `KEY=VALUE` pairs. Think of it as a small, hidden notepad where you write down all your app's secrets.

Here's what our `.env` file might look like:

```
GEMINI_API_KEY="AIzaSyDCtqkcX4YzR8Y_LbcW1Y6PvdP184U-rw"
# GROQ_API_KEY="gsk_l0I12kKv6FJgypdifVd7WGdyb3FYKNl7UVLIOzKef50vOUal
```

- `GEMINI_API_KEY`: This is the **name** of our secret.

- `="AIzaSyDCtqkcX4YzR8Y_LbcW1Y6PvdP184U-rw"` : This is the **value** of our secret. (Note: This is just an example key; your actual key will be different!).

How Our Health Tracker Uses Environment Configuration

Let's see the steps to use this "secret vault" for our `GEMINI_API_KEY` :

Step 1: Create the `.env` File

In the very top folder of your project (where `main.py` is located), create a new file and name it exactly `.env` .

Step 2: Add Your Gemini API Key

Open the `.env` file and add the following line, replacing the example key with your *actual* Gemini API Key:

```
GEMINI_API_KEY="YOUR_ACTUAL_GEMINI_API_KEY_HERE"
```

Important: If you don't have a Gemini API key yet, you'll need to get one from Google AI Studio. (Search for "Google AI Studio get API key" for instructions.)

Step 3: Tell Python to Load the `.env` File

In our `main.py` file, right at the very beginning, we need to add two lines of code to tell our application to "open the vault" and load the secrets from the `.env` file.

You've already seen them in Chapter 1: FastAPI Application!

```python
# From main.py

from dotenv import load_dotenv # 1. Import the tool
load_dotenv() # 2. Use the tool to load secrets!

# ... rest of your FastAPI application setup ...

```

- `from dotenv import load_dotenv` : This line imports a special function calle `load_dotenv` from a helpful library named `python-dotenv` . This library knows h `.env` files.

- `load_dotenv()` : This single line is like the magic spell! When your program starts, this function looks for the `.env` file in your project, reads all the `KEY=VALUE` pairs, and then makes them available to your program.

Step 4: Access the Key in Your Code

Now that `load_dotenv()` has done its job, any part of your Python code can "ask for" these secrets by their name. We do this using `os.getenv()` .

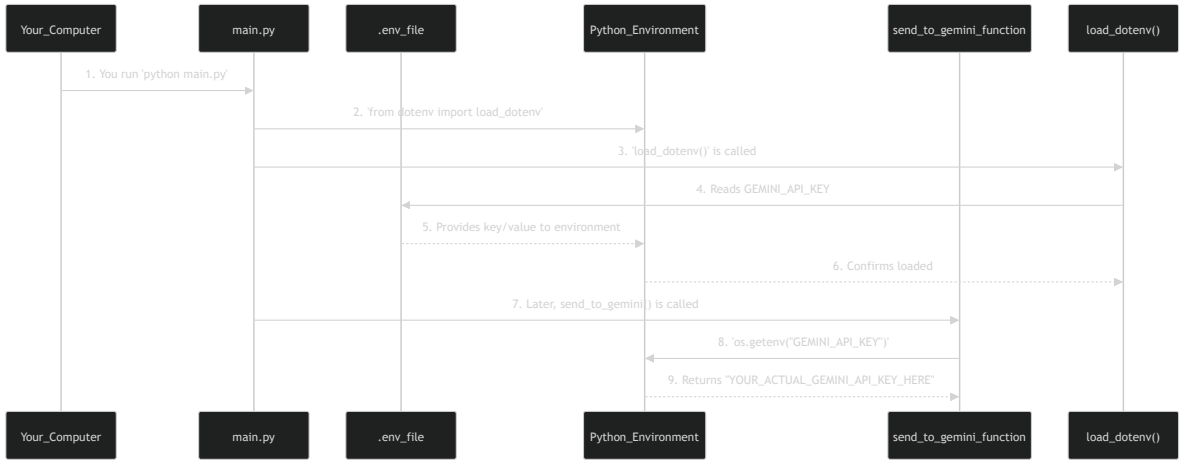
You already saw this in `main.py` inside our `send_to_gemini` function:

```
# From main.py (inside send_to_gemini function)

import os # Don't forget to import 'os' at the top of your file!

# This line asks for the "GEMINI_API_KEY" from our environment (where load_dotenv p
api_key = os.getenv("GEMINI_API_KEY")
if not api_key:
    raise EnvironmentError("Gemini not set in environment")

```

- `import os` : The `os` module (short for "operating system") is a built-in Python module that helps your program talk to the computer's operating system, including getting "environment variables" (which is what `load_dotenv` turns our `.env` entries into).
- `os.getenv("GEMINI_API_KEY")` : This is like asking the "vault manager" (the `os` module), "Please give me the value for the secret named `GEMINI_API_KEY` ." If the secret was loaded, it gets the value (`"AIzaSyDCtqkc...."`); otherwise, it gets `None` .
- `if not api_key: ...` : This is a check to make sure we actually got the key. If not, it means something went wrong (like the `.env` file wasn't there, or the key wasn't set), and we raise an error.

How It All Works (Under the Hood)

Let's trace how our `GEMINI_API_KEY` makes its way from your `.env` file to our `send_to_gemini` function:



1. **You Run `main.py`**: When you start our Health Tracker by running `python main.py`, the Python interpreter begins executing the code.

2. **`load_dotenv()` Activates**: The very first thing our `main.py` does (after imports) is call `load_dotenv()`.

3. **Reads `.env`**: The `load_dotenv()` function looks for the `.env` file in your project folder.

4. **Loads into Environment**: It reads the `GEMINI_API_KEY="value"` line from `.env` and **loads** this key-value pair into your Python program's "environment." Think of this as adding it to a temporary, secure list that only your running program can see.

5. **Program Continues**: The `main.py` script continues running, setting up FastAPI, defining endpoints, and so on. The `GEMINI_API_KEY` is now safely stored in the program's environment.

6. **`send_to_gemini()` Needs the Key**: When a request comes in (e.g., you log water), eventually our `handle_request` function calls `send_to_gemini`. Inside `send_to_gemini`, it needs that `GEMINI_API_KEY`.

7. **`os.getenv()` Retrieves**: The line `api_key = os.getenv("GEMINI_API_KEY")` is executed. The `os.getenv()` function looks into the Python program's "environment" (where `load_dotenv` put the key earlier) for a variable named `GEMINI_API_KEY`.

8. **Key is Used**: It finds the value and returns it. Now, `send_to_gemini` has the secret key it needs to securely talk to the Gemini AI!

The Role of `.gitignore`

This part is super important for security! Remember our "secret vault" analogy? We don't want the `.env` file (our secret notepad) to accidentally get into a public place like GitHub.

This is where the `.gitignore` file comes in. It's a special file that tells Git (our version control system) which files to **ignore** and *not* include when you "commit" and "push" your code to GitHub.

Look at the relevant lines in our project's `.gitignore`:

```
 # From .gitignore

# Environments
.env # This line tells Git to ignore our .env file!
.venv
env/
# ... many other lines ...
# API keys and secrets (extra protection)
*api_key*
*secret*
```

The line `.env` specifically tells Git: "No matter what, do NOT include the `.env` file when I share my code online." This ensures that your actual API key stays private, even if you share your entire project repository.

Conclusion

In this final chapter, we learned that **Environment Configuration** is a crucial practice for keeping sensitive information, like our `GEMINI_API_KEY`, secure and flexible. We discovered that the `.env` file acts as a secure external vault for these secrets. By using `load_dotenv()` at the start of our `main.py` and `os.getenv()` to retrieve values, our Health Tracker can access these critical settings without hardcoding them. Finally, we emphasized the vital role of `.gitignore` in preventing our secrets from being accidentally exposed when sharing our code.

You've now learned how to build a smart, secure Health Tracker that can talk to AI and manage its important settings! Congratulations on completing this tutorial.