Chapter 1: The Contract — OpenAPI Specification

Introduction: What You Will Learn

Before you write actual code, you need a **plan**. In web development, this plan is called an **API Specification**. It's a contract that says: - "These are the URLs you can call" - "This is the data you must send" - "This is the data you will receive"

By the end of this chapter, you will understand: 1. What OpenAPI/Swagger is 2. How to read and write YAML 3. Every section of our `api.yaml` file 4. Why we made specific design decisions

---

Part 1: What is OpenAPI?

The Problem: Miscommunication

Imagine this scenario:

Frontend Developer: "I need the list of messages" **Backend Developer**: "OK, call `/getMessages`" **Frontend Developer**: "It doesn't work" **Backend Developer**: "Oh, you need to send the conversation ID in the body" **Frontend Developer**: "Which field name? `conversationId`? `conv_id`? `id`?"

This is a disaster. People waste time on miscommunication.

The Solution: Write It Down

OpenAPI (formerly called Swagger) is a **formal document** that defines: - Every URL your API supports - What data each URL expects - What data each URL returns - What errors can happen

It's written in YAML or JSON format, and it can: - Generate documentation automatically - Generate code automatically - Be validated by tools

Our File: `service/api/api.yaml`

This file is approximately 450 lines. It defines EVERYTHING our backend does.

Part 2: YAML for Beginners

What is YAML?

YAML = "YAML Ain't Markup Language" (recursive acronym)

It's a human-readable data format. It uses **indentation** (spaces) to show structure.

Basic Rules

1. **Use spaces, not tabs** (2 spaces per level is standard)
2. **Colons separate keys from values**: `name: Alice`

3. **Dashes create lists**: `- item1`
4. **Nested items are indented**

Example: A Person in YAML

```yaml
person:
  name: Alice
  age: 25
  hobbies:
    - reading
    - coding
    - gaming
  address:
    city: Rome
    country: Italy
```

Translating to Understanding

The above YAML describes: - A person named Alice - She is 25 years old - She has 3 hobbies (a list) - She has an address (a nested object with city and country)

Part 3: The Structure of api.yaml

Our `api.yaml` has these main sections:

```yaml
openapi: 3.0.3              # Version of OpenAPI we're using
info:                       # Metadata about our API
  title: WASAText
  version: 1.0.0

servers:                    # Where the API runs
  - url: http://localhost:3000

tags:                       # Categories for endpoints
  - name: authentication
  - name: conversations
  - name: messages
  - name: groups

paths:                      # THE MAIN PART: All URLs
  /session:
    post: ...
  /conversations:
    get: ...
    post: ...

components:                 # Reusable definitions
```

```yaml
  schemas:                  # Data shapes
    User: ...
    Message: ...
  securitySchemes:          # Authentication methods
    bearerAuth: ...
```

Let me explain each section in detail.

Part 4: The `info` Section

```yaml
info:
  title: WASAText API
  description: A lightweight messaging application
  version: 1.0.0
```

This is metadata. It appears in generated documentation.

- **title**: The name of your API
- **description**: What it does
- **version**: Useful when you have multiple versions

Part 5: The `servers` Section

```yaml
servers:
  - url: http://localhost:3000
    description: Development server
```

This tells tools (like Swagger UI) where to send requests when testing.

In production, you'd add:

```yaml
  - url: https://api.wasatext.com
    description: Production server
```

Part 6: The `tags` Section

```yaml
tags:
  - name: authentication
    description: Login and user identity
  - name: conversations
    description: Chat management
  - name: messages
    description: Sending and receiving messages
  - name: groups
    description: Group management
```

Tags are **categories**. They organize your endpoints in documentation.

When you view the API in Swagger UI, endpoints are grouped by tag.

Part 7: The `paths` Section — The Heart of the API

This is where we define EVERY endpoint. Let me explain several examples in extreme detail.

Example 1: Login (`POST /session`)

```yaml
paths:
  /session:
    post:
      tags:
        - authentication
      operationId: doLogin
      summary: Log in or create account
      description: |
        If the username exists, returns the existing user.
        If the username doesn't exist, creates a new user.
      requestBody:
        required: true
        content:
          application/json:
            schema:
              type: object
              required:
                - username
              properties:
                username:
                  type: string
                  minLength: 3
                  example: "alice"
      responses:
        '201':
          description: User logged in successfully
          content:
            application/json:
              schema:
                type: object
                properties:
                  id:
                    type: integer
                    example: 1
                  token:
                    type: string
                    example: "1"
```

```
        '400':
          description: Invalid username
```

Let me break this down line by line:

Line	Meaning
`/session:`	The URL path is `/session`
`post:`	This is a POST request (we're creating something)
`tags: [authentication]`	This endpoint belongs to the "authentication" group
`operationId: doLogin`	**CRITICAL**: The function name in Go will be called `doLogin`
`summary:`	Short description shown in docs
`description:`	Long description (the `|` means multi-line text)
`requestBody:`	What the client MUST send
`required: true`	You CANNOT call this endpoint without a body
`content:`	The type of data (JSON in this case)
`schema:`	The structure of the data
`type: object`	It's a JSON object `{}`
`required: [username]`	The `username` field MUST be present
`properties:`	The fields in the object
`username:`	A field called "username"
`type: string`	It must be text
`minLength: 3`	At least 3 characters
`example: "alice"`	Example for documentation
`responses:`	What the server might return
`'201':`	If successful (201 Created)
`'400':`	If there's an error (400 Bad Request)

Example 2: Send a Message (`POST /conversations/{conversationId}/messages`)

```
  /conversations/{conversationId}/messages:
    post:
      tags:
        - messages
      operationId: sendMessage
      summary: Send a message to a conversation
      security:
        - bearerAuth: []
      parameters:
        - name: conversationId
          in: path
          required: true
          schema:
            type: integer
      requestBody:
        required: true
        content:
```

```yaml
        application/json:
          schema:
            type: object
            properties:
              content:
                type: string
                example: "Hello!"
              photo:
                type: string
                description: Base64 encoded image (optional)
              reply_to_message_id:
                type: integer
                description: ID of message being replied to (optional)
    responses:
      '201':
        description: Message sent
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Message'
      '401':
        description: Not authenticated
      '403':
        description: Not a member of this conversation
      '404':
        description: Conversation not found
```

Key new concepts:

Concept	Explanation
`{conversationId}`	This is a **path parameter**. The actual URL might be `/conversations/5/messages`
`in: path`	The parameter comes from the URL, not the body
`security: [bearerAuth: []]`	You must be logged in to use this endpoint
`$ref: '#/components/schemas/Message'`	Instead of defining the schema here, we reference a reusable definition

Example 3: Delete a Message (`DELETE /conversations/{conversationId}/messages/{messageId}`)

```yaml
  /conversations/{conversationId}/messages/{messageId}:
    delete:
      tags:
        - messages
      operationId: deleteMessage
      summary: Delete a message
      security:
        - bearerAuth: []
```

```yaml
    parameters:
      - name: conversationId
        in: path
        required: true
        schema:
          type: integer
      - name: messageId
        in: path
        required: true
        schema:
          type: integer
    responses:
      '204':
        description: Message deleted successfully
      '401':
        description: Not authenticated
      '403':
        description: Can only delete your own messages
      '404':
        description: Message or conversation not found
```

Key points: - **204 No Content**: The standard response for successful DELETE. There's nothing to return because the thing no longer exists. - **Two path parameters**: Both `conversationId` AND `messageId` are in the URL.

Part 8: The `components/schemas` Section — Reusable Data Shapes

Instead of defining the same structure over and over, we define it once and reference it.

The User Schema

```yaml
components:
  schemas:
    User:
      type: object
      properties:
        id:
          type: integer
          description: Unique identifier for the user
          example: 1
        username:
          type: string
          description: Display name
          example: "alice"
        photo:
          type: string
          description: Base64-encoded profile photo
```

The Message Schema

```yaml
    Message:
      type: object
      properties:
        id:
          type: integer
        conversation_id:
          type: integer
        sender_id:
          type: integer
        content:
          type: string
        timestamp:
          type: string
          format: date-time
        reply_to_message_id:
          type: integer
          nullable: true
        forward_from_message_id:
          type: integer
          nullable: true
        photo:
          type: string
        comment:
          type: string
```

Using References

When we want to return a Message, instead of rewriting all of this:

```yaml
responses:
  '200':
    content:
      application/json:
        schema:
          $ref: '#/components/schemas/Message'
```

The `$ref` says “look up the definition at components schemas Message”.

Part 9: The `securitySchemes` Section — How Authentication Works

```yaml
components:
  securitySchemes:
    bearerAuth:
      type: http
      scheme: bearer
      description: |
```

```
Send the token in the Authorization header:
Authorization: Bearer <your_token>
```

This defines that: 1. We use HTTP authentication 2. The scheme is “bearer” (a common pattern) 3. The token goes in the `Authorization` header

When an endpoint has `security: [bearerAuth: []]`, it means: - The client MUST send an `Authorization: Bearer <token>` header - If they don’t, return 401 Unauthorized

---

Part 10: Design Decisions — Why We Did It This Way

Decision 1: Why `/conversations/{id}/messages` instead of `/messages`?

The question: Should messages have their own top-level resource?

Our choice: Messages are nested under conversations.

Reasoning: - A message CANNOT exist without a conversation - When you fetch messages, you always want them for a specific conversation - This makes authorization natural: if you can access `/conversations/5`, you can access its messages

Alternative rejected: `/messages?conversation_id=5` - This works but is less RESTful - Harder to reason about permissions - Query parameters are typically for filtering, not scoping

Decision 2: Why `POST /session` for login instead of `POST /login`?

The question: What URL should handle login?

Our choice: `POST /session`

Reasoning: - Login creates a “session” — a user’s presence in the system - REST is about resources. A session IS a resource. - In a real app, you might have `DELETE /session` for logout

Decision 3: Why separate `operationId` for every endpoint?

The question: Do we need operationId?

Our choice: Yes, always.

Reasoning: - Code generators use operationId to name functions - Without it, you get auto-generated names like `postConversationsConversationIdMessages` - With it, you get clean names like `sendMessage`

Decision 4: Why `PUT /users/{id}/name` and `PUT /users/{id}/photo` separately?

The question: Should we have one endpoint to update all user fields?

Our choice: Separate endpoints for each field.

Reasoning: - The PDF requirement specified separate actions - Simpler validation (only validate what’s being changed) - Clearer permissions (maybe in future, only admins can change names)

Summary Checklist

Before moving to Chapter 2, make sure you understand:

- ☐ What OpenAPI is and why we use it
- ☐ How to read YAML (indentation, lists, objects)
- ☐ What `operationId` does
- ☐ The difference between `required` and `optional` parameters
- ☐ How `$ref` works for reusability
- ☐ What `security: [bearerAuth: []]` means
- ☐ Why we use 201 for creation and 204 for deletion

Oral Exam Questions

Q: Why did you use OpenAPI? "I used OpenAPI 3.0 because it's the industry standard for REST API documentation. It provides a single source of truth that both frontend and backend developers can reference. It also enables automatic code generation and interactive documentation through Swagger UI."

Q: Explain your URL structure. "I followed RESTful conventions. Resources are nouns (users, conversations, messages). Nested resources represent ownership (messages belong to conversations). This hierarchy makes the API intuitive and simplifies authorization."

Q: Why is operationId important? "OperationId is crucial for code generation. It provides human-readable function names instead of auto-generated ones. It also serves as a stable identifier that documentation tools use for linking."