OpenAPI and Service Integration

Vijay Paul Nayar

2023-08-29 Tue
1 Introduction

2 What is OpenAPI?

3 Managing OpenAPI Specs

4 Useful D Features

5 D Project: openapi-client
Who am I?

Vijay Paul Nayar
- Java developer and CTO of a FinTech
- Left CTO role to found own company

Funnel-Labs.io: Performant D apps.
- Funnel: High performance data storage system for ride-hailing and micro-mobility companies.
- Fiveum: Office chat and video built to minimize interruptions and improve focus.
How did OpenAPI Come Up?

- Built Funnel Service MVP...
- How do customers pay for the service?
  - Most services use credit-cards
- How to easily add credit-card support?
  - Stripe is popular and common
- How to use Stripe?
  - Stripe has a REST API, but it’s huge
- How do Java/Python do this?
  - Generated OpenAPI client
- Do such tools exist in D?
  - No, but they could.
Introduction
External Service Interoperability

Companies often depend on useful external services. For example:

- Stripe (financial transactions)
- OpenAI (categorize sentiment, question/answer, content generation)
- Slack (real-time communication)

Handwritten clients are time-consuming and error-prone.
Even internal services face interoperability challenges:

- Communication must be secure
- Interfaces should be understandable and standardized
- Multiple programming languages must be supported (companies change technologies, different employees have different skills, etc.)
REST Interfaces

(Re)presentational (S)tate (T)ransfer is an architectural style designed for the web

- Many forms, typically JSON/Avro/Protobuf over HTTPS
- URLs arranged into "nouns" with HTTP Methods representing "verbs"
- By itself, too vague to be uniform
- Minor performance penalty for increased clarity
Presentation Agenda

What is OpenAPI?
OpenAPI Specification is open standard to define HTTP APIs for external consumers

- Builds upon JSON Schemas
  https://json-schema.org/
- Builds upon Swagger API description and documentation
  https://swagger.io/
- Split from Swagger in 2016 to become the OpenAPI Initiative, a Linux Foundation project
Benefits of OpenAPI Usage

- Commonly used by major services, e.g. Stripe, Slack, OpenAI, and 2500+ more: https://apis.guru/
- Standard formats mean tools can be used to generate client code with:
  - request and responses
  - documentation
  - success and error codes
- Creating an OpenAPI Specification enables low-effort cross-compatibility
Swagger Sample App

This is a sample server Petstore server. You can find out more about Swagger at http://swagger.wordnik.com or on irc.freenode.net, #swagger. For this sample, you can use the api key "special-key" to test the authorization filters

Terms of service
Contact the developer
Apache 2.0

pet : Operations about pets

GET /pet/{petId}

DELETE /pet/{petId}

PATCH /pet/{petId} (partial updates to a pet)

POST /pet/{petId} (Updates a pet in the store with form data)

Parameters
- petId (required) ID of pet that needs to be updated
- name Updated name of the pet
- status Updated status of the pet

Data Type
- path
- form
- string

Response Messages
- HTTP Status Code: 400 Reason: Invalid input
OpenAPI Specification is itself a JSON/YAML document

### OpenAPI Major Top-Level Attributes

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>servers</td>
<td>[Server Object]</td>
<td>Connection info. for servers offering the API.</td>
</tr>
<tr>
<td>paths</td>
<td>Paths Object</td>
<td>Method-specific actions by URL path.</td>
</tr>
<tr>
<td>components</td>
<td>Components Object</td>
<td>Re-usable schemas for data by name.</td>
</tr>
<tr>
<td>security</td>
<td>[Security Object]</td>
<td>Lists security mechanisms to access the API.</td>
</tr>
</tbody>
</table>
Defining API Endpoints - #/paths

- Mapping from endpoint URL to details

```json
{
    "paths": {
        "/files/{file_id}" : {  // URL => Path Item
            "delete": {  // Method => Operation
                "operationId": "deleteFile",  // API-unique identifier
                "tags": [  // Tags for grouping documentation
                    "OpenAI"
                ],
                "summary": "Delete a file.",  // A 1-liner for documentation.
                "parameters": [  // Request parameters in path/query/header/cookie
                    {
                        "in": "path",
                        "name": "file_id",
                        "required": true,
                        "schema": {  // JSON Schema format is used.
                            "type": "string"
                        },
                        "description": "The ID of the file to use for this request"
                    }
                ],
                "responses": {  // Response data format by HTTP status
                    "200": {  // Response data format by HTTP status
                        "description": "OK",
                        "content": {
                            "application/json": {
                                "$ref": "#/components/schemas/DeleteFileResponse"
                            }
                        }
                    }
                }
            }
        }  // All other methods: GET, POST, PUT, OPTIONS, HEAD
    }
}
```
JSON Schemas

- All data represented in JSON can be described using JSON Schemas.
- Assertions are used to validate if data matches the schema:
  - `type` Primitive values like null, boolean, object, array, number, string
  - `format` How a type is used, e.g. date-time, email, uri, ipv4, etc.
  - `enum` Limit value to a predefined list.
  - `allOf` All validations must be satisfied.
  - `anyOf` One or more validation must be satisfied.
  - `oneOf` Exactly one validation must be satisfied.
An example schema:

```json
{
    "components": {
        "schemas": {
            "CreateChatCompletionResponse": {
                "type": "object",
                "properties": {
                    "id": {
                        "type": "string"
                    },
                    "model": {
                        "type": "string"
                    },
                    "choices": {
                        "type": "array",
                        "items": {
                            "type": "object",
                            "required": [
                                "index",
                                "message",
                                "finish_reason"
                            ],
                            "properties": {
                                ...
                            }
                        }
                    }
                }
            }
        }
    }
}
```

An example instance complying with the schema:

```json
{
    "id": "3d5e3472-3057-11ee-89d4-c3a0bb88b99f",
    "model": "gpt-3.5-turbo",
    "choices": [
        {
            "index": 3,
            "finish_reason": "length",
            "message": {
                ...
            }
        },
        ...
    ]
}
```
Data Types:

- The "type" field corresponds broadly to a JSON type.
- The "format" field clarifies details and usage.

<table>
<thead>
<tr>
<th>type</th>
<th>format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>int32</td>
<td>signed 32 bits</td>
</tr>
<tr>
<td>integer</td>
<td>int64</td>
<td>signed 64 bits</td>
</tr>
<tr>
<td>number</td>
<td>float</td>
<td></td>
</tr>
<tr>
<td>number</td>
<td>double</td>
<td></td>
</tr>
<tr>
<td>string</td>
<td>password</td>
<td>A hint to UIs to obscure input</td>
</tr>
</tbody>
</table>
APIs commonly have shared data types between paths.

- Error and Created responses
- Query Parameters
- Security headers

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>schemas</td>
<td>string =&gt; SchemaObj</td>
<td>Common schemas by name.</td>
</tr>
<tr>
<td>responses</td>
<td>string =&gt; ResponseObj</td>
<td>Path responses, e.g. errors.</td>
</tr>
<tr>
<td>parameters</td>
<td>string =&gt; ParameterObj</td>
<td>Request parameter types.</td>
</tr>
<tr>
<td>requestBodies</td>
<td>string =&gt; RequestBodyObj</td>
<td>Request bodies for POST, PUT.</td>
</tr>
<tr>
<td>headers</td>
<td>string =&gt; HeaderObj</td>
<td>Common data in HTTP headers.</td>
</tr>
<tr>
<td>securitySchemes</td>
<td>string =&gt; SecuritySchemeObj</td>
<td>E.g. OAuth, Basic Auth, etc.</td>
</tr>
</tbody>
</table>
Reusing Components

- Once defined, components can be referenced by their location in the OpenAPI Schema.
- Substitute type definition with a "$ref" to a component.

```json
"properties": {
  "index": {
    "type": "integer"
  },
  "message": {
    "$ref": "#/components/schemas/ChatCompletionResponseMessage"
  }
}
```
Managing OpenAPI Specs
Creating OpenAPI Specifications

- D currently lacks tools to extract specification from code.
- Open question whether it is better to:
  - Generate specification from code
    - Easier to keep specification up to date
    - Language/Framework-specific projects like SpringDoc
  - Generate interfaces from specification
    - Easier tool integration and multi-language support
    - Projects like openapi-generator
OpenAPI Specification from Code

- Systems like SpringDoc are specific to language (Java) and web framework (Spring)
- OpenAPI Specification is updated with code changes
What happens when a service is split?
What if multiple technologies are used?

D Code
Java Code
Python Code
OpenAPI Spec
JS Client
Dart Client
Public Docs
Code from OpenAPI Specification

- Requires clients/servers to regenerate code after changes
@SecurityScheme(name = "petstore_auth", type = SecuritySchemeType.OAUTH2, flows = @OAuthFlows(implicit = @OAuthFlow(authorizationUrl = "https://petstore3.swagger.io/oauth/authorize", scopes = {
@OAuthScope(name = "write:pets", description = "modify pets in your account"),
@OAuthScope(name = "read:pets", description = "read your pets") })))
@Tag(name = "pet", description = "the pet API")
public interface PetApi {
    @Operation(summary = "Add a new pet to the store",
               description = "Add a new pet to the store",
               security = { @SecurityRequirement(name = "petstore_auth", scopes = { "write:pets", "read:pets" }) },
               tags = { "pet" })
    @ApiResponse(responseCode = "200",
                  description = "Successful operation",
                  content = {
                              @Content(mediaType = "application/xml", schema = @Schema(implementation = Pet.class)),
                              @Content(mediaType = "application/json", schema = @Schema(implementation = Pet.class)) }),
    @ApiResponse(responseCode = "405", description = "Invalid input")
    @PostMapping(value = "/pet", consumes = { "application/json", "application/xml", "application/x-www-form-urlencoded" })
    default void addPet(
        @Parameter(description = "Create a new pet in the store", required = true) @Valid @RequestBody Pet pet) {
        // return getDelegate().addPet(pet);
    }
Useful D Features
Mixins

The mixin expression takes a list of string arguments representing a complete D statement and turns them into code.

- Can make use of variables known at compile-time, e.g. those provided by templates
- Useful for code that declares variables or methods with parameterized identifiers

```d
mixin("private bool _myValue;");
string N = "yourVal";
mixin("private bool", ", ", N, ";");
```
Mixin Templates

A mixin template encloses declarations of fields, functions, classes, structs, etc. When referenced in code with compile-time parameters, it inserts those declarations in the scope in which it was called.

**Mixin Templates**: Re-useable code generation

```plaintext
import std.traits : isAssignable;
import std.string : capitalize;
import std.typecons : Nullable;

mixin template AddField(C, T, string N) {

    // Declare the variable.
    mixin(T, " ", N, ";");
    mixin(
        // Define setter function.
        C, " set", capitalize(N), "(ST)(ST val) ",
        "if (isAssignable!(T, ST)) {",
        " this.", N, " = val;",
        " return this;",
        "}");
}

// Example usage
class Fish {
    mixin AddField!(typeof(this), Nullable!int, "age");
    mixin AddField!(typeof(this), Nullable!string, "job");
}

unittest {
    import std.stdio;
    Fish f = new Fish()
        .setAge(42)
        .setJob("Accountant");
    writeln(f.age, " ", f.job);
}
```
Static ForEach

static foreach statements generate repeated lines of code in the same scope in which they occur.

- static foreach: Loop over compile-time data, such as class members.

```plaintext
import std.traits : Fields, FieldNameTuple,
                  BaseClassesTuple;
// Add setters for a single class.
mixin template AddClassSetters(C) {
    static foreach (size_t i; iota(Fields!(C).length)) {
        mixin AddSetter!(Fields!(C)[i], FieldNameTuple!(C)[i]);
    }
}

// Add setters for full class hierarchy.
mixin template AddSetters(C) {
    static foreach (B; BaseClassesTuple!(C)) {
        mixin AddSetters!(B);
    }
    mixin AddClassSetters!(C);
}
```

class A {
    int a1;
    string a2;
}

class B : A {
    float b1;
    mixin AddSetters!(typeof(this));
}

unittest {
    import std.stdio;
    B b = new B()
        .setA1(3)
        .setA2("ham")
        .setB1(2.9);
}
D Project: openapi-client
code.dlang.org project: openapi-client

- Consistent interface created/updated in seconds
- Creates data types from OpenAPI Specification
- Creates client to call endpoints
- Configurable server and security controls
openai-client: Creating an OpenAI Client

1. Download the OpenAPI Specification from GitHub:
   
   curl https://raw.githubusercontent.com/openai/openai-openapi/master/openapi.yaml
   -o openapi.yaml

2. Convert to JSON format:
   
   yq openapi.yaml -o json > openapi.json

3. Invoke openapi-client to generate code:
   
   dub run openapi-client@2.0.1 --
   --openApiSpec=json/openapi.json
   --packageRoot=openai

4. Done!
class CreateImageEditRequest {

/**
 * The number of images to generate. Must be between 1 and 10.
 */
@vibeName("n")
@vibeOptional
@vibeEmbedNullable
Nullable!(int) n;

/**
 * The image to edit. Must be a valid PNG file, less than 4MB, and square. If mask is not
 * provided, image must have transparency, which will be used as the mask.
 */
@vibeName("image")
@vibeOptional
string image;

Optional fields are Nullable.

- Nested objects as static inner classes
- Documentation included
- Builder pattern used to ease object creation
class ImagesEditsService {
    /**
     * Creates an edited or extended image given an original image
     * See_Also: HTTP POST `/images/edits`
     */
    void createImageEdit(
        CreateImageEditRequest requestBody,
        CreateImageEditResponseHandler responseHandler,
    ) {
        ApiRequest requestor = new ApiRequest(
            HTTPMethod.POST,
            Servers.getServerUrl(),
            "// images/edits";
        requestor.setHeaderParam("Content-Type", "multipart/form-data");
        Security.apply(requestor);
        requestor.makeRequest(requestBody, responseHandler);
    }
}
// Service classes group API functionality by path, e.g. /completions
auto service = new CompletionsService();
// Invoke an API endpoint, this one is for POST /completions
service.createCompletion(
    // Define the request body with a builder.
    CreateCompletionRequest.builder()
        .model("text-davinci-003")
        .prompt(Json("What is the cutest breed of rabbit? "))
        .echo(true)
        .maxTokens(2048)
        .build(),
    // ResponseHandlers have an attribute for each valid response.
    CompletionsService.CreateCompletionResponseHandler.builder()
        .handleResponse200((CreateCompletionResponse response) {
            logDebug("%s", serializeToJson(response).toString());
        })
        .build());
The debate for which rabbit breed is the cutest is subjective, as it will depend on what the individual finds appealing. Some popular breeds that are known for their cute looks include the Holland Lop, Mini Rex, Jersey Wooly, Mini Lop, and Netherland Dwarf.
Future Plans

- Move spec-first efforts to more mature projects like openapi-generator
  - Add D client and server-stub generators
- Consider code-first integration via annotations in frameworks like Vibe.d
Thank you for your interest and attention!