API mastery with OpenAPI

JSON and Servers and Clients, Oh My!

20 February 2024

Dr. Jacob Hochstetler
Distinguished Engineer, Vice President, Fidelity Investments
Clinical Assistant Professor, University of North Texas

Agenda

- Client-Server Architecture
- OpenAPI
- Workshop
  - Creating an OpenAPI Specification
  - Generating a Server (Go)
  - Generating a Client (Python)
- Next Steps
- Conclusion

Client-Server: The model

What is a **client**?

> Software that users *directly* interact with to **request** data.

What is a **server**?

> Software designed to *process* and *respond* to **requests** from clients.

- **1960s-1970s: Emergence**
  - Time-sharing systems introduce early client-server model concepts.
- **1980s: Expansion**
  - Personal computers and LANs popularize the client-server model.
- **1990s-Present: Internet Era**
  - Web technologies evolve to become cloud computing and microservices.

Client-Server: Protocols

A *protocol* is a set of rules that dictate how data is exchanged over a network.

- Hypertext Transfer Protocol Secure (HTTPS): encrypted web traffic communication.
- SSH File Transfer Protocol (SFTP): encrypted file transfers.
- Network Time Protocol (NTP): time synchronization.
- Simple Mail Transport Protocol (SMTP): sending mail service.
- Post Office Protocol (POP3): receiving mail service.

Specification Implementation

Example of a driving protocol specification and its implementation.

Client-Server: Types/Examples

Types of Clients

- **Web** Browsers: Chrome, Firefox
- **Mobile** Applications: Instagram, Snapchat
- **Desktop** Applications: Microsoft Outlook, Slack, VSCode
- **API** Clients [raw API]: Bruno, Insomnia, Postman

Types of Servers

- **Proxy/Web** Servers: Envoy, Caddy, Traefik, Apache, `nginx`
- **Database** Servers: PostgreSQL, MySQL, MongoDB
- **File** Servers: SMB/CIFS, NFS, AFS
- **Mail** Servers: Postfix, Microsoft Exchange
- **Application** Servers: Glassfish, Gunicorn, Tomcat, Node.js

Client-Server: Communication elements & flow

Request-Response cycle

- Clients request data & servers respond with information.

Stateless interactions (for HTTP)

- Each request is treated independently (no state is saved).

Protocol-Based

- Uses *protocols* (HTTP, FTP) for standardized communication.

Data format

- Defined by a standard/specification, or negotiated by the client and server.

Client-Server: Data formats

Data formats define how information is structured/encoded for exchange:

- **JSON** (JavaScript Object Notation): A lightweight format for data interchange.
- **XML** (eXtensible Markup Language): Legacy format and a "cousin" of HTML.
- `protobuf` (Protocol Buffers): Binary format for serializing structured data.

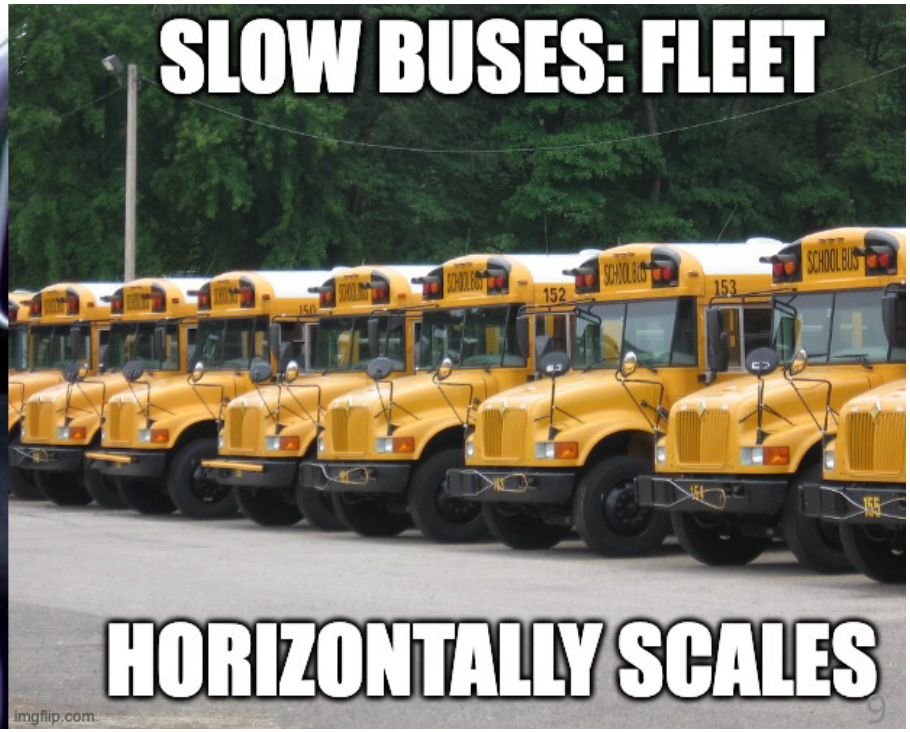
Different kinds of vehicles are like different kinds of formats. Each has its own strengths and weaknesses.

Client-Server: What is JSON? JAY-Sawn? JEH-Sun?

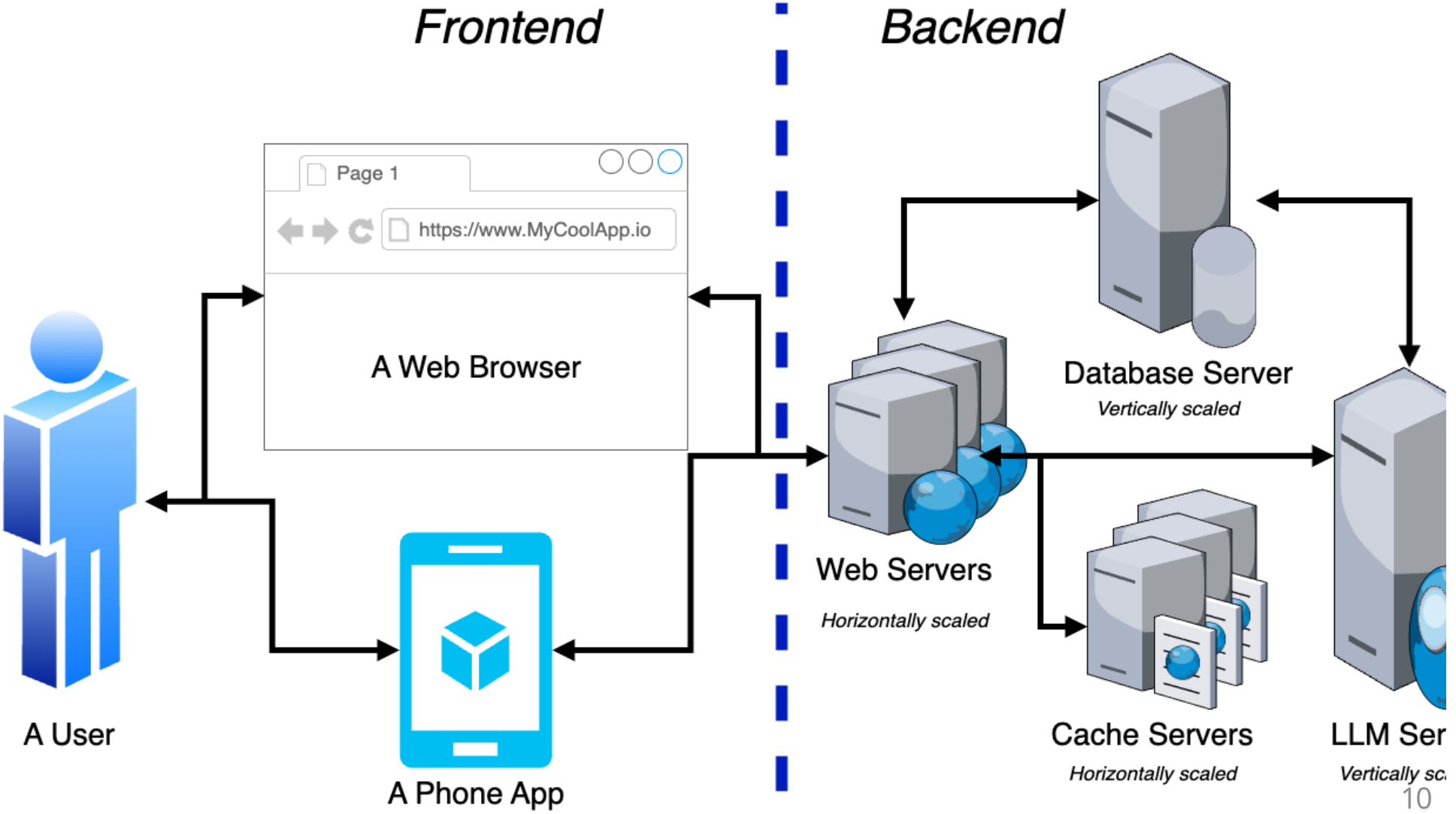
- **J**ava**S**cript **O**bject **N**otation.
- **Readable**: Stored as text (not binary) easy for humans and machines to read/write.
- **Value Types**:
  - String: sequence of characters enclosed in double quotes → `"Hello!"`
  - Number: integer or floating-point number → 42, `3.14`
  - Boolean: a true or false value → `true`, `false`
  - Null: a null value, representing the absence of a value → `null`
- **Complex Types**:
  - Object: Key-value pairs, enclosed in braces → `{"name": "Jay", "age": 30}`
  - Array: An ordered list of values, enclosed in brackets → `["apple", "banana"]`

Client-Server: Generic application components and scaling terms

- **Frontend:** What you see and interact with on a website, e.g. the layout/design.
- **Backend:** The database(s), server(s), and application logic hidden from the user.
- **Vertical:** Increasing the power (CPU/RAM) of individual servers to handle more load.
- **Horizontal:** Distributing your workload across a pool of servers.



Client-Server: Generic application components and scaling diagram

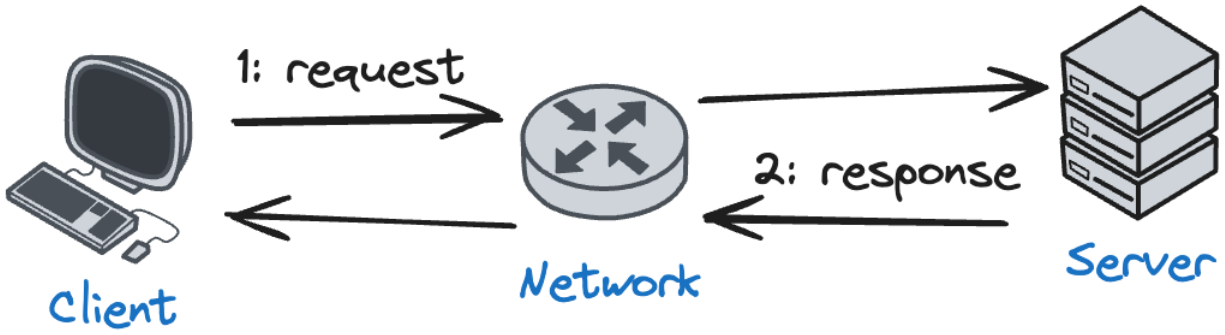


Client-Server: What is an Application Programming Interface (API)?

A set of rules that allow different software applications to communicate with each other.

For example, an HTTP *API* is composed of six parts of the HTTP *protocol*:

- **Headers**: Optional metadata about the request. (e.g., authN, accept, encoding, etc.)
- **Endpoint**: The specific URL where requests can be sent.
- **Method**: The type of request being made (e.g., `GET`, `POST`, `DELETE`).
- **Body**: Optional content being sent with the request or response.
- **Status Code**: A code that indicates the success or failure of a request (e.g., 200, 404).
- **Trailers**: Optional metadata about the response.

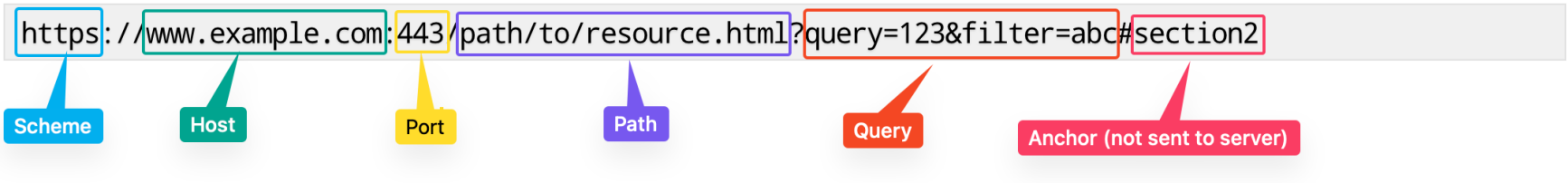


Client-Server: What is a Uniform Resource Locators (URLs)?

- **URL**: A web address that specifies the location of a resource on the internet.

Composed of several parts:

- **Scheme**: The protocol used to access the resource (e.g., `sftp`, `https`).
- **Host**: The domain name or IP address of the server.
- **Port**: The specific port on the server to connect to (optional).
- **Path**: The specific location of a resource on the server.
- **Query**: Additional parameters for the request (optional).
- **Anchor**: A named anchor pointing to a DOM ID (optional and *not* sent to the server).



Full example of a URL with Scheme, Host, Port, Path, Query, and Anchor.

Client-Server: Representational State Transfer (REST) in API Design

REST was developed by Roy Fielding in his 2000 doctoral dissertation.
✎ ***Not*** a protocol, but a set of constraints for creating web services.

Web APIs that adhere to the REST architectural constraints are called RESTful APIs:

- **Uniform Interface**: Decouples client-server & enables independent evolution.
  - **Resources**: Central concept in REST, each identified by a URL.
  - **Stateless**: Each request contains all information needed to fulfill the request.
  - **CRUD Operations**: Utilizes standard HTTP methods:
    - Create: `POST`
    - Read: `GET`
    - Update: `PUT`
    - Delete: `DELETE`
  - **Representation**: Uses formats chosen through client↔server negotiation.

Client-Server: Summary

- Client-Server model
- Protocols & Types
- Communication elements & flow
- Data formats
- Generic application terms
- HTTP APIs & REST

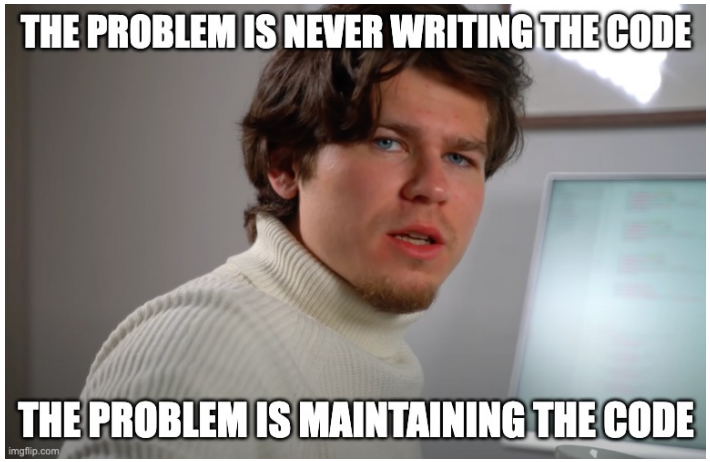
The role and impact of APIs:

1. Translation for interaction & provides universal vocabulary
2. Facilitates cooperation & enables integration
3. Simplifies complexity & hides implementation details

OpenAPI: Software Engineering Challenges

HTTP APIs are widely used for web services but they come with challenges:

- **Validation**: Ensure the integrity and validity of data being exchanged.
- **Documentation maintenance**: Keeping docs up-to-date with the implementation.
- **Version control**: Manage changes and backward compatibility of APIs over time.
- **Error handling**: Meaningful error messages and status codes for various failures.
- **Compatibility**: Ensure APIs work across diverse clients & programming languages.
- **Security**: Implement secure and flexible authN/authZ mechanisms.



OpenAPI: Enter the specification

OpenAPI is a specification for building APIs that aims to address a range of challenges associated with *designing*, *developing*, *documenting*, and even *using* web APIs:

- **Validation**: Mechanisms for validating query params, headers, and body content.
- **Documentation**: Automatically generates interactive documentation.
- **Client SDK Generation**: Enables easy creation of client libraries across languages.
- **Versioning**: Helps manage different versions of an API effectively.
- **Error handling**: Allows for the definition of standard error structures and messages.
- **Interface consistency**: Ensures consistency of HTTP methods, codes, and payloads.
- **Content negotiation**: Specifies support for multiple request/response formats.
- **Security definitions**: Standardizes the way API security schemes are defined.
- **Discovery**: Makes it easier for clients to discover the capabilities of an API.
- **Testing**: Enables easy creation of tests for API validation and verification.

OpenAPI: What is a schema?

A schema is a formal description of the structure of data and can validate the content.

```json
{
    "id": 2,
    "category": "Pet Supplies",
    "price": 1
}
```

A basic JSON payload example.

```json
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "title": "Product",
    "description": "A product from the catalog",
    "type": "object",
    "properties": {
        "id": {
            "description": "The unique identifier for a product",
            "type": "integer"
        },
        "category": {
            "description": "Name of the product",
            "type": "string"
        },
        "price": {
            "type": "number",
            "minimum": 1,
            "exclusiveMinimum": true
        }
    },
    "required": ["id", "category", "price"]
}
```

A basic JSON schema used to validate the example above.

OpenAPI: The specification

The OpenAPI Specification (OAS) uses schemas to define request/response payloads.

- **Data types**: Define the type of data (e.g., `string`, `number`, `boolean`).
- **Properties**: Define the structure of an object (e.g., "name", "age", "address").
- **Required fields**: Define which fields are mandatory for a request or response.
- **Nested objects**: Define complex data structures with nested objects and arrays.

An OAS document serves as a blueprint for:

1. Generating client & server (stub) code.
2. Generating interactive documentation.
3. Generating test cases.
4. Creating mock servers.

✎ The doc can be JSON or YAML format but YAML is preferred for commentability.

OpenAPI: Main parts of the specification

The OAS includes:

- **Paths**: URLs to access various API functionalities. Represents endpoints.
- **Operations**: HTTP methods applied to paths (e.g., GET, POST).
- **Parameters**: Details on request info like headers, query strings.
- **Responses**: Expected data structure and status codes from API calls.

```
openapi: 3.0.0
info:
  title: Books API
  version: 1.0.0
paths:
  /books:
    get:
      summary: Get a list of books
      responses:
        200:
          description: Success
```

An example minimum viable OpenAPI document.

OpenAPI: Code-first or Schema-first?

Code-first:

1. Design your program "normally" then start writing code/manually writing docs.
2. Generate an OpenAPI spec from code annotations or comments (*optional*).

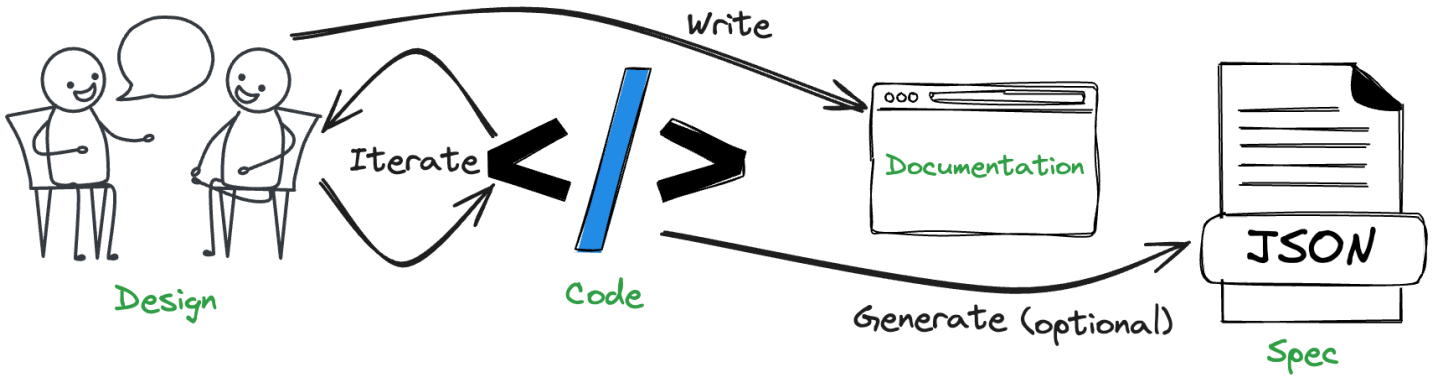
- *Pros*: Easy for developers, keeps code and spec in sync.
- *Cons*: Can lead to less comprehensive specs, depends on tooling.

Schema-first, also referred to as **Spec-first**:

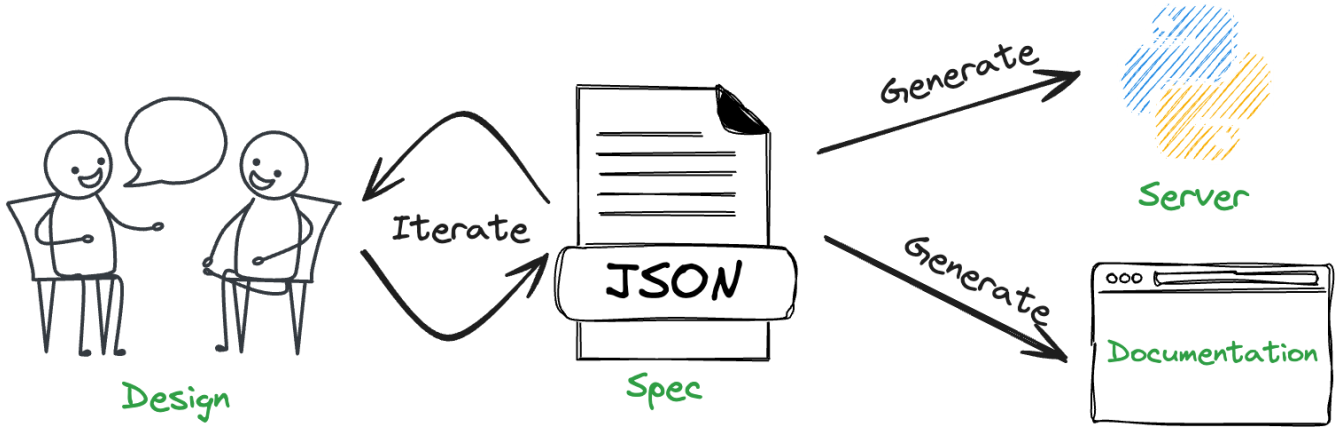
1. Design the OpenAPI spec, then generate server stubs, client SDKs, and docs.
2. Implement business logic in the stubs (*required...it's not magic*).

- *Pros*: Clear API contract, promotes design thinking, tool-agnostic.
- *Cons*: Requires upfront design effort, potential for spec-code divergence.

OpenAPI: Code-first or Schema-first flowcharts



Code-first approach: Start with code and generate an OpenAPI spec from code annotations or comments.



Schema-first approach: Design the OpenAPI spec and then generate server stubs, client SDKs, and docs.

OpenAPI: The tools

Pages of OpenAPI tools are available at OpenAPI.Tools and OpenAPI Tooling.

Official Swagger tooling:

- **Editor**: A web-based tool for creating and editing OpenAPI documents.
- **UI**: A tool for visualizing and interacting with the API's resources.
- **Codegen**: Generates client and server stubs from an OpenAPI document.

✎ "*OpenAPI*" vs. "*Swagger*"

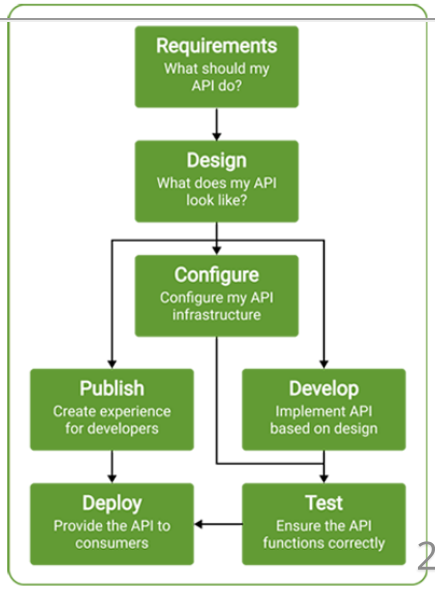
- Swagger is the original toolkit for building APIs with a specification format.
- OpenAPI Specification versions (3.0+) are evolved from the original Swagger (2.0).
- "*Swagger*" now refers to the *API tools* supporting the OpenAPI Specification.

Workshop: The plan

0. **Tutorial**: OpenAPI tutorial

---

1. **Editor**: Design an OpenAPI spec for `PetStore™`.
2. **UI**: Demonstrate Swagger UI.
3. **CodeGen**: Generate a Go server from spec.
4. **Server**: Sprinkle in business logic and deploy.
5. **Client**: Create a Python CLI from spec.



Workshop: Swagger Editor

"Official" editors:

1. Swagger Editor
2. Swagger Editor Next
3. Swagger Offline Editor

Open online editors (and many others):

- Apibldr
- Frogment

VSCode plugin:

- OpenAPI Editor

Workshop: Swagger UI

"Official":

1. Online: Swagger UI

Opensource:

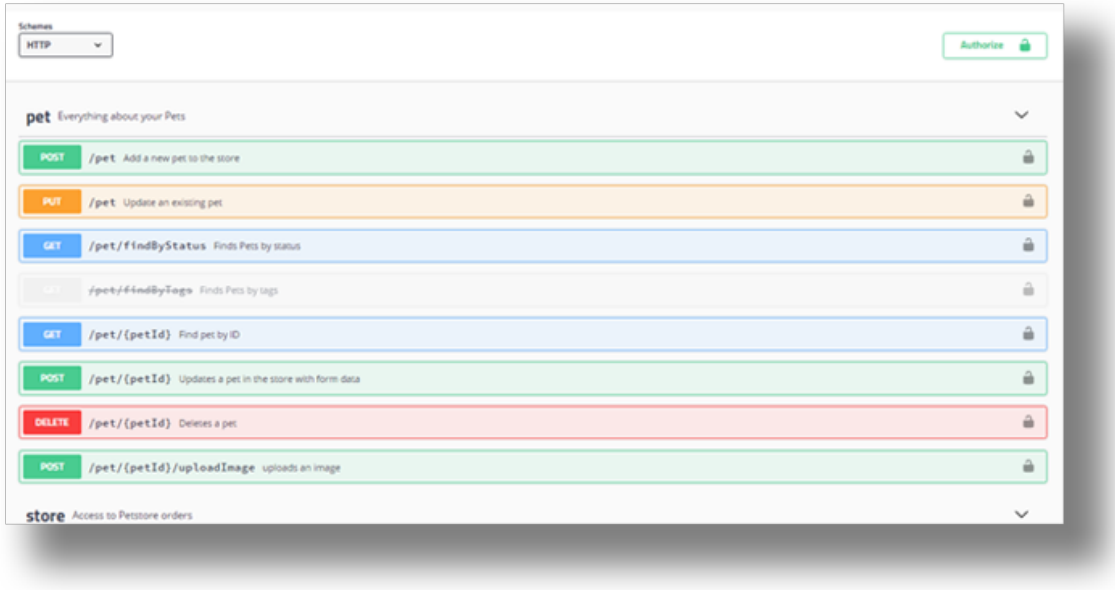
2. Desktop: Bruno
3. Desktop: Hoppscotch

Forces you to have an account:

3. Desktop: Insomnia
4. Desktop: Postman

Embeddable:

5. Package: SWGUI



Workshop: Swagger Codegen

Since the OpenAPI Spec is open-source, many different generators are available...

I'll use `oapi-codegen` from DeepMap, Inc since it can generate a Chi router (I like Chi).

```
git clone https://github.com/jh125486/petstore && cd petstore
make generate
```

This will:

1. Install the the `oapi-gen` code generator, and
2. Execute the `go generate` comments in the `petstore.go` file.

✎ More info on `generate` comments can be found in the blog post "Generating code".

Two files will be generated based on the OAPI-codegen YAML configs:

- `api/petstore-server.gen.go`: Petstore endpoint handlers (interface)
- `api/petstore-types.gen.go`: Petstore models

Workshop: Server implementation

Now that we've got our handlers interface and models, we'll add the business logic.

For our simple Petstore™, we'll just use a mutexed `map` to store our `Pets`:

```
type PetStore struct {
    Pets   map[int64]Pet
    NextId int64
    Lock   sync.Mutex
}
```

Then we'll write some functions to satisfy the `ServerInterface`:

```
type ServerInterface interface {
    // Returns all pets
    FindPets(w http.ResponseWriter, r *http.Request, params FindPetsParams)
    // Creates a new pet
    AddPet(w http.ResponseWriter, r *http.Request)
    // Deletes a pet by ID
    DeletePet(w http.ResponseWriter, r *http.Request, id int64)
    // Returns a pet by ID
    FindPetByID(w http.ResponseWriter, r *http.Request, id int64)
}
```

Workshop: Server deployment

With the business logic added, we can serve our handler in a few lines from `main.go`:

```
r := chi.NewRouter()                                    // Create the Chi router.
spec, _ := api.GetSwagger()                             // Get OpenAPI spec and ignore any errors
r.Use(mw.OapiRequestValidator(spec))                    // Use OpenAPI spec request middleware.
handler := api.NewStrictHandler(api.NewPetStore(), nil) // Use the repo to create our handler.
api.HandlerFromMux(handler, r)                          // Add the handler  to the Chi router.
log.Fatal(http.ListenAndServe(":"+port, r))             // Serve HTTP until the world ends.
```

Then we can manually test our server:

```
PORT=3000 go run .
```

OK, that works (hopefully ☝️), so let's deploy to Google Cloud Run:

```
make deploy
```

```
Building using Buildpacks and deploying container to Cloud Run service [petstore] in project [present-403020] region [us-south1]
✓ Building and deploying... Done.
  ✓ Uploading sources...
  ✓ Building Container... Logs are available at [https://console.cloud.google.com/cloud-build/builds/88866f66-7e85-4ab0-9125-d0e1a274229f?project=538351792545
  ✓ Creating Revision...
  ✓ Routing traffic...
Done.
Service [petstore] revision [petstore-00009-klm] has been deployed and is serving 100 percent of traffic.
Service URL: https://petstore-2jruimmcrq-vp.a.run.app
```

Workshop: Server deployment verification

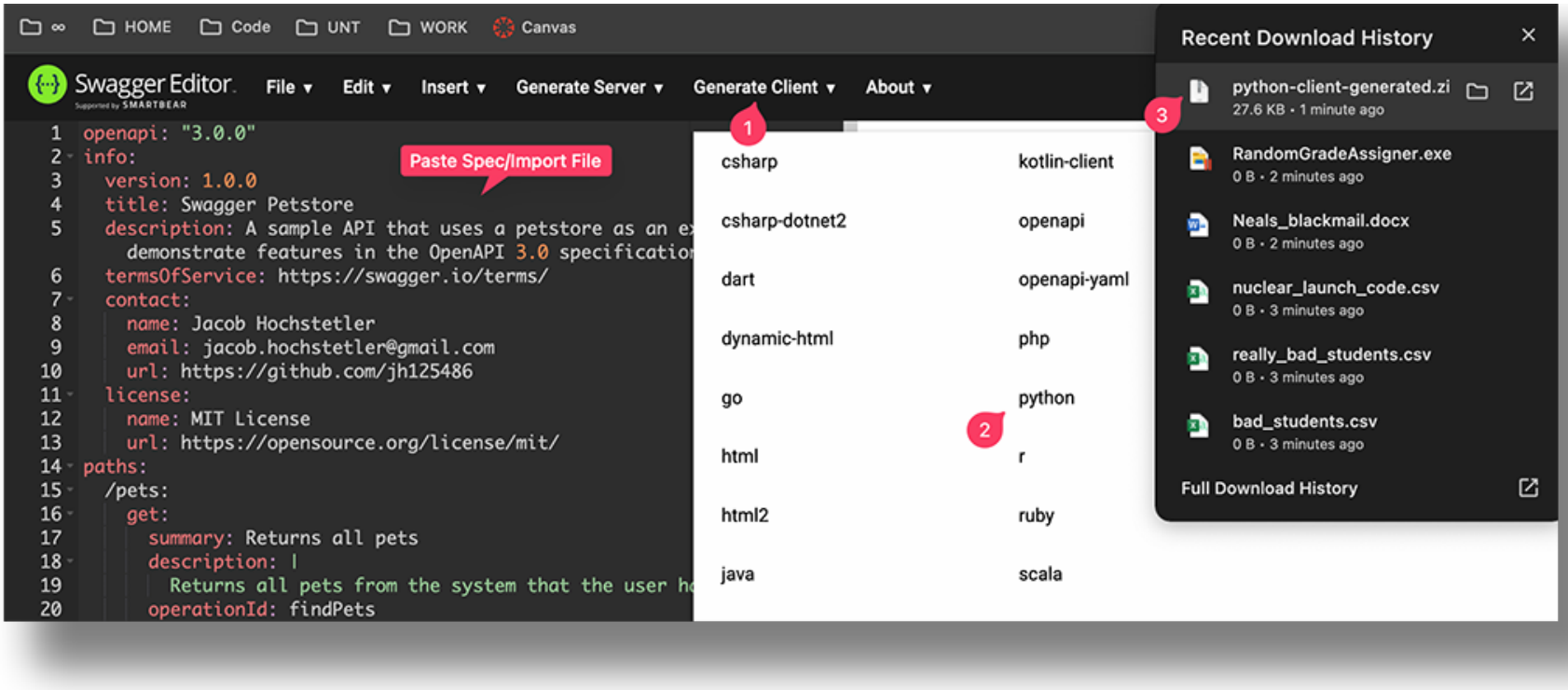
Open your browser and head to the Cloud Run App URL

It should return a 404 since we don't have any handler defined for '/'

We've embedded a Swagger UI though, at /docs/

Workshop: Client generation

I don't have an opinion on client-generators, so I'll just use Swagger Editor.



This uses the Java swagger-codegen CLI from Swagger API.

There's plenty of other generators, like this pure Python one from openapi-generators.

Workshop: Client implementation

Now that the client is generated, we'll have to wire up `main.py` to actually call the API:

```
# main.py
# Setup the client configuration and point the `host` to our server.
configuration = swagger_client.Configuration()
configuration.host = "https://petstore-2jruimmcrq-vp.a.run.app/"

# Create an API instance from the client configuration.
api_instance = swagger_client.DefaultApi(swagger_client.ApiClient(configuration))

# Create a cat named "Hank" and print the response.
pprint(api_instance.add_pet(swagger_client.NewPet("Hank", "cat")))

# Delete the pet with ID #1000
api_instance.delete_pet(1000)

# Find a pet with ID #1001 and print the response.
pprint(api_instance.find_pet_by_id(1001))

# Find first 3 pets tagged with "dog"
pprint(api_instance.find_pets(tags=['dog'], limit=3))
```

✎ In reality, you'd want to have `try / except` blocks to catch any API errors returned.

Next Steps

- **Security**:
  - Define authN security schemes in the OpenAPI spec (e.g., OAuth2, JWT).
- **Mocking**:
  - Prism by Stoplight: A set of packages for API mocking and contract testing.
  - Mocks Server: A Node.js mock server running live mocks in place of real APIs.
- **Testing**:
  - Dredd: A tool for validating API document against implementation of the API.
- **Versioning**:
  - RESTful API best practices: *When* and *How* to version a REST API.
  - Semantic Versioning: Uses three numbers to convey changes and compatibility.

Summary

What we discussed today:

1. *Client-server*
    - From protocols to REST to data formats.
2. *OpenAPI*
    - The specification and the problems it aims to solve.
3. *Workshop activities*
    - Generated a server, deployed it, and then called it with a generated client.
4. *Next Steps*
    - Resources for Security, Mocking, Testing and Versioning OpenAPIs.

Thank you

Dr. Jacob Hochstetler
Distinguished Engineer, Vice President, Fidelity Investments
Clinical Assistant Professor, University of North Texas
Jacob.Hochstetler@UNT.edu
Jacob.Hochstetler@Fidelity.com
https://github.com/jh125486