**Chapter 4**

**API Design and Modeling**

Sanjay Patnia\*

a UCSC Extension, Santa Clara, California, USA

**Abstract**

This chapter starts with API design strategies and then goes into API creation process and modeling. Best practices for REST API design are discussed, followed by API solution architecture. In the exercises, a simple API is designed for podcasts subscription and then modeling using OpenAPI.

**API Design Strategies**

As UI is to UX (User Experience), API is to APX (Application Programming Experience). In APX it is important to answer following questions:

* What should be exposed?
* What is the best way to expose the data?
* How should API be adjusted and improved?

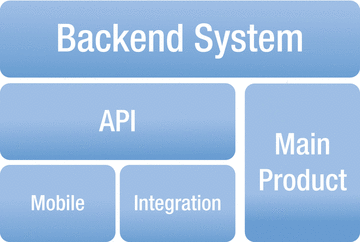
In addition, let’s discuss why we should develop a nice Application Programming Experience?

A nice API will encourage the developers to use it and share it with others, creating a virtuous cycle where each additional successful implementation leads to more engagement and more contributions from developers who add value to your service. I’ll start by saying that API design is hard.

Also, a nice API will help to grow an ecosystem of employees, customers, and partners who can use and help to continue to evolve your API in ways that are mutually beneficial.

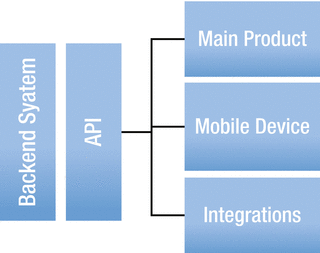
There are four strategies for API design:

* Bolt-on strategy: This is when you have an existing application and add an API after the fact. This takes advantage of existing code and systems (Figure [2-1](#bookmark)).



***Figure 2-1.*** *Bolt-on strategy*

* Greenfield strategy: This is the other extreme. This is a strategy behind “API-first” or “Mobile first,” and is the easiest scenario to develop an API. Since you’re starting from scratch, you can make use of technologies and concepts that may not have been available before (Figure [2-2](#bookmark1)).



***Figure 2-2.*** *Greenfield strategy*

Greenfield or API-first strategy is a simulation-based design implementation.

Simulation of a back-end system is development of a back-end system without need

ing fully implemented back-end systems. With simulation of APIs, consumers can

start development of apps without fully developed APIs.

* Agile design strategy: Agility is based on the premise that you can start without a full set of specs. You can always adapt and change the specs later, as you go and learn more. Through multiple iterations, architectural design can converge to the right solution. Agile approach should only be applied until API is published.
* Finally, you have the facade strategy, which is the middle ground between Greenfield and bolt-on. In this case, you can take advantage of existing business systems, yet shape them to what you prefer and need. This gives them the ability to keep working systems in place while making the underlying architecture better.

**API Creation Process and Methodology**

In this section we are going to review API creation process and methodology. In order to deliver great APIs, the design must be a first-order concern. Like optimizing for UX (User Experience) has become a primary concern in UI development, also optimizing for APX (API User Experience) should be a primary concern in API development.

**Process**

First determine your business value. When thinking about business value, think of the “elevator pitch” about why you need an API. Developer engagement is not a great goal; you need a tangible goal: increase user engagement, move activity off the main product to the API, engage and retain partners, and so on.

Choose your metrics, e.g.:

* Number of developer keys in use
* Number of applications developed
* Number of users interacting via API
* Number of partner integrations
* How API is enhancing goals of the company as a whole rather than simply determining how many people have begun to integrate

**API Methodology**

Consists of 5 phases in the case of agile strategy:

* Domain analysis or API description
* Architecture design
* Prototyping
* Building API for production, then
* Publishing the API

**Domain Analysis or API Description**

Define your use cases for domain analysis. Who are the participants? Are they external or internal? Which API solutions do consumers want to build with the API? Which other API solutions would be possible with the API?

Activities participant takes on consumer view: What would the API that the consumer wants to use look like? What apps does the consumer want to build? What data or domain objects does the consumer want to use in his app?

Break activities into steps or write down the usage scenario.

* A dependent resource can not exist without another.
* For example, the association of a podcast and its consumer can not be determined unless the podcast and its consumer are created.
* An independent resource can exist without another.
* For example, a podcast resource can exist without any dependency.
* An associative resource exists independently but still has some kind of relation, i.e., it may be connected by reference.
* As mentioned above

The next step is to identify possible transitions between resource states. Transitions between states provide an indicator of the HTTP method that needs to be supported. For the example of the podcast which could be added to a playlist, let’s analyze different states:

***Table 2-1.*** *Domain analysis example*

|  |  |  |  |
| --- | --- | --- | --- |
| **State** | **Operation** | **Domain Objet** | **Description** |
| CREATE | POST | PODCAST | Creates podcast |
| READ | GET | PODCAST | Read podcasts |
| READ | GET/{podcast\_id} | PODCAST | Reads podcast |
| UPDATE | PUT/{playlist\_id} | PODCAST | Updates podcast |

Also, verify by building a simple demo app. More than curl calls, this demo app provides a showcase for the API and can be reused in later stages.

**Architecture Design**

In this phase, API description or analysis phase is further redefined. Architecture design should make decisions about

* Protocol
* End points
* URI design
* Security
* Performance or availability

Detail design description:

* Resources
* Representations
* Content types
* Parameters
* HTTP methods
* HTTP status codes
* Consistent naming

In addition, look into reusability by looking at common APIs in the API Portfolio. Design decisions should be consistent with the API in the API Portfolio. The API Portfolio is a collection of APIs in an Enterprise, as discussed in Chapter [5](http://dx.doi.org/10.1007/978-1-4842-2665-0_5).

As part of the design verification, the demo app can be further extended here with design decisions. Issues to be verified are that:

* the API is still easy to use;
* the API is simple and supports use cases; and,
* the API follows architectural style.

**Prototyping**

Prototyping is the preparation for the production implementation. Take complex use cases and implement end-to-end with high-fidelity. The prototype is incomplete and uses shortcuts. It can have a simulation of API if the back-end functionality is not available at the time of building the prototype. Once the prototype is made, then there is the acceptance test with pilot consumers as verification of the API. Pilot customers are internal customers from the API provider’s team.

**Implementation**

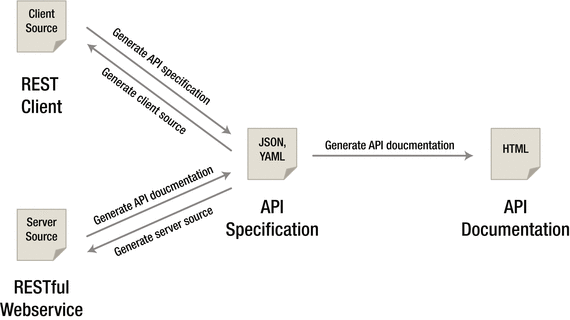
The implementation needs to conform to the API description and needs to be delivered as soon as possible. In addition, the API is fully integrated into the back-end system and API Portfolio. This should have all the desired functionality as well as non-functional aspects of the API, like performance, security, and availability. At this stage, the API description should be stable since it has gone through multiple iterations. For verification, hand-picked API consumers could be identified at this stage.

**Publish**

Publishing of the API does not require a lot of work, but this is a big milestone for the API. From an organizational perspective, the responsibility of the API is transferred from development to the operational unit. After publishing, there is no agility in the development process. Any change requires traditional change management process. As part of the verification, there is analysis on successful vs. failed API calls and documentation gaps which are supported by the maintenance team.

**API Modeling**

Modeling the schema for your API means creating a design document that can be shared with other teams, customers, or executives. A schema model is a contract between your organization and the clients who will be using it. A schema model is essentially a contract describing what the API is, how it works, and exactly what the endpoints are going to be. Think of it as a map of the API, a user-readable description of each endpoint, which can be used to discuss the API before any code is written. Figure [2-3](#bookmark2) below shows the API Modeling framework where you have API specifications defined and generate API documentation. Also, generate server and client source code.



***Figure 2-3.*** *API Modeling*

Creating this model before starting development helps you to ensure that the API you create will meet the needs described by the use cases you’ve identified. The three schema modeling systems and the markup languages they use are:

* RAML: markdown, relatively new. Good online modeling tool: RESTful API Modeling Language
* OpenAPI(Swagger:) JSON, large community
* Blueprint: markdown, low adoption

The OpenAPI(Swagger) exercise in this chapter shows the modeling done for the podcast resource.

Each of the schema modeling languages has tools available to automate testing or code creation based on the schema model you’ve created, but even without this functionality the schema model helps you to have a solid understanding of the API before a single line of code is written.

Figure [2-4](#bookmark3) below shows the API Modeling tool.



**Figure 2-4.** *APIModeling tool*

**Comparison of API Modeling**

**Table 2-2.** *Comparision of API Modeling tools*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Category | Property | RAML | API Blueprint | Swagger |
| What is behind name? | Format | YAML | Markdown (MOSN) | JSON |
|  | Available at | Github | Github | Github |
|  | Sponsored By | Mulesoft | Apiary | SmartBear |
|  |  |  |  |  |
|  | Initial Commit | Sep 2013 | Apr 2013 | Jul 2011 |
|  | Commercial Offering | Yes | Yes | Yes |
| How does it model REST? | Resources | X | X | X(“api”) |
|  | Methods/  Actions | X(“methods”) | X(“actions”) | X(“operations”) |
|  | Query Parameters | X | X | X |
|  | Path / URL Parameters | X | X | X |
|  | Representation | X | X | X |
|  | Header Parameters | X | X | X |
|  | Documentation | X | X | X |
|  | References | <http://raml.org> | <https://apiblueprint.org> | <http://swagger.io> |
|  | Design | API-first | Design First | Existing API |
|  | Code Generation | X |  | X |
| Who are customers? |  |  |  | APIGEE, Microsoft, Paypal |

In summary:

* Swagger has a very strong modeling language for defining exactly what’s expected of the system—very useful for testing and creating coding stubs for a set of APIs.
* RAML is designed to support a design-first development flow, and focuses on consistency.
* Apiary blueprint is more documentation-focused, with user-readable models and documentation as its first priority.

Each project brings different strengths and weaknesses to the table, and in the end it’s really about what strengths you need and which weaknesses you cannot afford. Overall, RAML fared the best in these different categories and, while the developer community is not as large as the others, I think it’s safe to say it will keep growing.

Overall Winner: RAML

**Best Practices**

REST is an architectural style and not a strict standard; it allows for a lot of flexibly. Because of that flexibility and freedom of structure, there is also a big appetite for design best practices. These best practices are discussed here in this section.

**Keep your base URL simple and intuitive**

The base URL is the most important design affordance of your API. A simple and intuitive base URL design makes using your API easy. Affordance is a design property that communicates how something should be used without requiring documentation. A door handle’s design should communicate whether you pull or push. For Web API design, there should be only two base URLs per resource. Let’s model an API around a simple object or resource (a customer) and create a Web API for it. The first URL is for a collection; the second is for a specific element in the collection:

* /customers - Collection
* /customers/1 - Specific element

Boiling it down to this level will also force the verbs out of your base URLs. Keep verbs out of your URLs as shown in table below:

**Table 2-3.** Nouns and verbs

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Resource | POST Create | GET Read | PUT Update | DELETE Delete |
| /customers | New customer | List customers | Bulk update | Delete all |
| /customers/12 | - | Show customer 12 | If exists update  If not error | Delete customer 12 |

In summary:

* Use two base URLs per resource. Keep verbs out of your base URLs. Use HTTP verbs to operate on the collections and elements.
* The level of abstraction depends on your scenario. You also want to expose a manageable number of resources.
* Aim for concrete naming and to keep the number of resources between 12 and 24.
* An intuitive API uses plural rather than singular nouns, and concrete rather than abstract nouns.
* Resources almost always have relationships to other resources. What’s a simple way to express these relationships in a Web API? Let’s look again at the API we modeled in nouns are good, verbs are bad—the API that interacts with our podcasts resource. Remember, we had two base URLs: /podcasts and /podcasts/1234. We’re using HTTP verbs to operate on the resources and collections. Our podcasts belong to customers. To get all the podcasts belonging to a specific customer, or to create a new podcast for that customer, do a GET or a POST:
* GET /customers/5678/podcasts
* POST /customers/5678/podcasts
* Sweep complexity under the “?”. Make it simple for developers to use the base URL by putting optional states and attributes behind the HTTP question mark. To get all customers in sfo city of ca state of usa country:
* GET /customers?country=usa&state=ca&city=so

**Error Handling**

Many software developers, including myself, don’t always like to think about exceptions and error handling, but it is a very important piece of the puzzle for any software developer, and especially for API designers. Why is good error design especially important for API designers? From the perspective of the developer consuming your Web API, everything at the other side of that interface is a black box. Errors therefore become a key tool providing context and visibility into how to use an API. First, developers learn to write code through errors. The “test-first” concepts of the extreme programming model and the more recent “test-driven development” models represent a body of best practices that have evolved because this is such an important and natural way for developers to work. Second, in addition to when they’re developing their applications, developers depend on well-designed errors at the critical times when they are troubleshooting and resolving issues after the applications they’ve built using your API are in the hands of their users.

Handling errors: Let’s take a look at how three top APIs approach:

* Facebook

HTTP Status Code: 200

{"type" : "OauthException", "message":"(#803) Some of the aliases you requested do not exist: foo.bar"}

* Twillo

HTTP Status Code: 401

{"status" : "401", "message":"Authenticate","code": 20003, "more info": "http://www.twilio.com/docs/errors/20003"}

* Another example of error messaging from SimpleGeo

HTTP Status Code: 401

{"code" : 401, "message": "Authentication Required"}

When you boil it down, there are really only 3 outcomes in the interaction between an app and an API:

* Everything worked—success.
* The application did something wrong—client error.
* The API did something wrong—server error.

**Error Code**

Start by using the following 3 codes which should map to the 3 outcomes above. If you need more, add them. But you shouldn’t need to go beyond:

* 200 - OK
* 400 - Bad Request
* 500 - Internal Server Error

If you’re not comfortable reducing all your error conditions to these 3, try picking among these additional 5:

* 201 - Created
* 304 - Not Modified
* 404 - Not Found
* 401 - Unauthorized
* 403 - Forbidden

Check out this good Wikipedia entry for all HTTP Status codes: <https://en.wikipedia.org/wiki/List_of_HTTP_status_codes>.

**Versioning**

Never release an API without a version.

* Make the version mandatory.
* Specify the version with a “v” prefix. Move it all the way to the left in the URL so that it has the highest scope (e.g., /v1/dogs).
* Use a simple ordinal number. Don’t use the dot notation like v1.2, because it implies a granularity of versioning that doesn’t work well with APIs—it’s an interface, not an implementation. Stick with v1, v2, and so on.
* How many versions should you maintain? Maintain at least one version back.
* For how long should you maintain a version? Give developers at least one cycle to react before obsoleting a version.
* There is a strong school of thought about putting format (xml or json) and version in the header. Simple rules we follow: If it changes the logic you write to handle the response, put it in the URL so you can see it easily. If it doesn’t change the logic for each response (like OAuth information), put it in the header.

**Partial Response**

Partial response allows you to give developers just the information they need. Take, for example, a request for a tweet on the Twitter API. You’ll get much more than a typical twitter app often needs, including the name of person, the text of the tweet, a timestamp, how often the message was retweeted, and a lot of metadata. Let’s look at how several leading APIs handle giving developers just what they need in responses, including Google, who pioneered the idea of partial response:

* LinkedIn

/people:(id,first-name,last-name,industry)This request on a person returns the ID, first name, last name, and the industry

* Facebook

/joe.smith/friends?fields=id,name,picture

* Google

?fields=title,media

Google and Facebook have a similar approach, which works well. They each have an optional parameter called “fields” after which you put the names of fields you want to be returned. As you see in this example, you can also put sub-objects in responses to pull in other information from additional resources.

**Pagination**

Make it easy for developers to paginate objects in a database. Let’s look at how Facebook, Twitter, and LinkedIn handle pagination. Facebook uses offset and limit. Twitter uses page and rpp (records per page). LinkedIn uses start and count semantically. Facebook and LinkedIn do the same thing, that is, the LinkedIn start and count.

To get records 50 through 75 from each system, you would use:

* Facebook - offset 50 and limit 2
* Twitter - page 3 and rpp 25 (records per page)
* LinkedIn - start 50 and count 25

**Multiple Formats**

We recommend that you support more than one format—that you push things out in one format and accept as many formats as necessary. You can usually automate the mapping from format to format. Here’s what the syntax looks like for a few key APIs.

* Google Data: ?alt=json
* Foursquare: /venue.json
* Digg\*: Accept: application/json

**API Façade**

Use the façade pattern when you want to provide a simple interface to a complex subsystem. Subsystems often get more complex as they evolve.

Implementing an API façade pattern involves three basic steps:

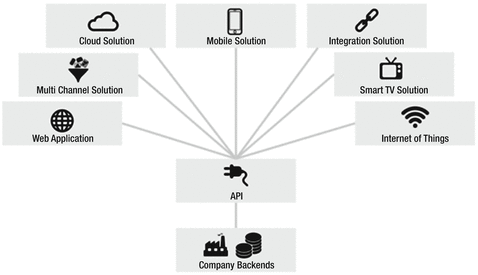
1. Design the ideal API—design the URLs, request parameters and responses, headers, query parameters, and so on. The API design should be self-consistent. This means you give the developers the information they need.
2. Implement the design with data stubs. This allows application developers to use your API and give you feedback even before your API is connected to internal systems
3. Mediate or integrate between the façade and the systems.

**API Solution Architecture**

Developers and architects often think of APIs as a continuation of the integration-based architectures that have long been in use within enterprise IT. But this is a narrow view.

To understand the demands and requirements on APIs, let’s discuss typical solutions that are enabled by APIs.

Figure [2-5](#bookmark4) below shows API Solution Architecture.



***Figure 2-5.*** *API Solution Architecture*

API solutions typically consists of two components:

* Exposes API
* Exposed API resides server-side, e.g., in the cloud or on premise.
* Consumes API
* Web or mobile apps and embedded devices on IoT

**Mobile Solutions**

Mobile apps need to connect to the servers on the Internet to be useable at all or at least to be usable to their full potential—some business logic on the app and heavy duty processing logic on servers on the cloud. Functionality hosted on these servers can be reached by APIs calls. Data captured on mobile devices is sent to servers by APIs calls, which hands to services and then to databases. Data delivered by APIs needs to be lightweight. This ensures APIs can be consumed by devices with limited processing power. Typically, the mobile app provider provides the APIs for the mobile app.

**Cloud Solutions**

SaaS cloud solutions typically consist of a web application and APIs. The web application is visible for the consumers. Under the hood, cloud solutions usually offer an API as well. Examples: Dropbox, Salesforce, Workday, Oracle Cloud

**Web Solutions**

Web applications display dynamic web pages based upon user requests; web pages are created on the fly with data available from the back end. The web application pulls raw data from the APIs, processes the data (JSON, XML), and displays in HTML, e.g., podcast or customer API.

**Integration Solutions**

APIs provide capabilities which are essential for connecting, extending the integrating software. By integrating software APIs, businesses can connect with other businesses. The business of an enterprise can be expanded by linking business to a partner. Integration not only makes sense externally, but also internally for integrating internal systems.

**Multi-channel Solutions**

Today, an e-commerce system offers customers shopping on multiple platforms—mobile, web, tablet. It is required to provide a seamless experience when a consumer moves from one platform to another. This can be accomplished by providing a common API, which supports a multichannel maintaining state of user experience.

**Smart TV Solutions**

Smart TV offers not only TV channels, but provides interaction capabilities. These are all implemented by API calls to the servers.

**Internet-of-Things**

The Internet of Things is made up of physical devices with an Internet connection. The device connects to smart functions (e.g., sensors, scanners, etc.) which are exposed on the Internet via APIs.

**Stakeholders in API Solutions**

In API Solutions, stakeholders are API Providers, API Consumers and End Users. We will discuss the roles of each here in this section.

**API Providers**

API providers develop, design, deploy, and manage APIs. API providers define the API portfolio, roadmap, and product mode. It is the responsibility of an API provider to decide which functionality is exposed by the API. In the solution-driven approach, only those APIs are built which are required by the consumer. In the top-down approach, API providers provide APIs which are good from an internal perspective, e.g., from a reusability perspective.

**API Consumers**

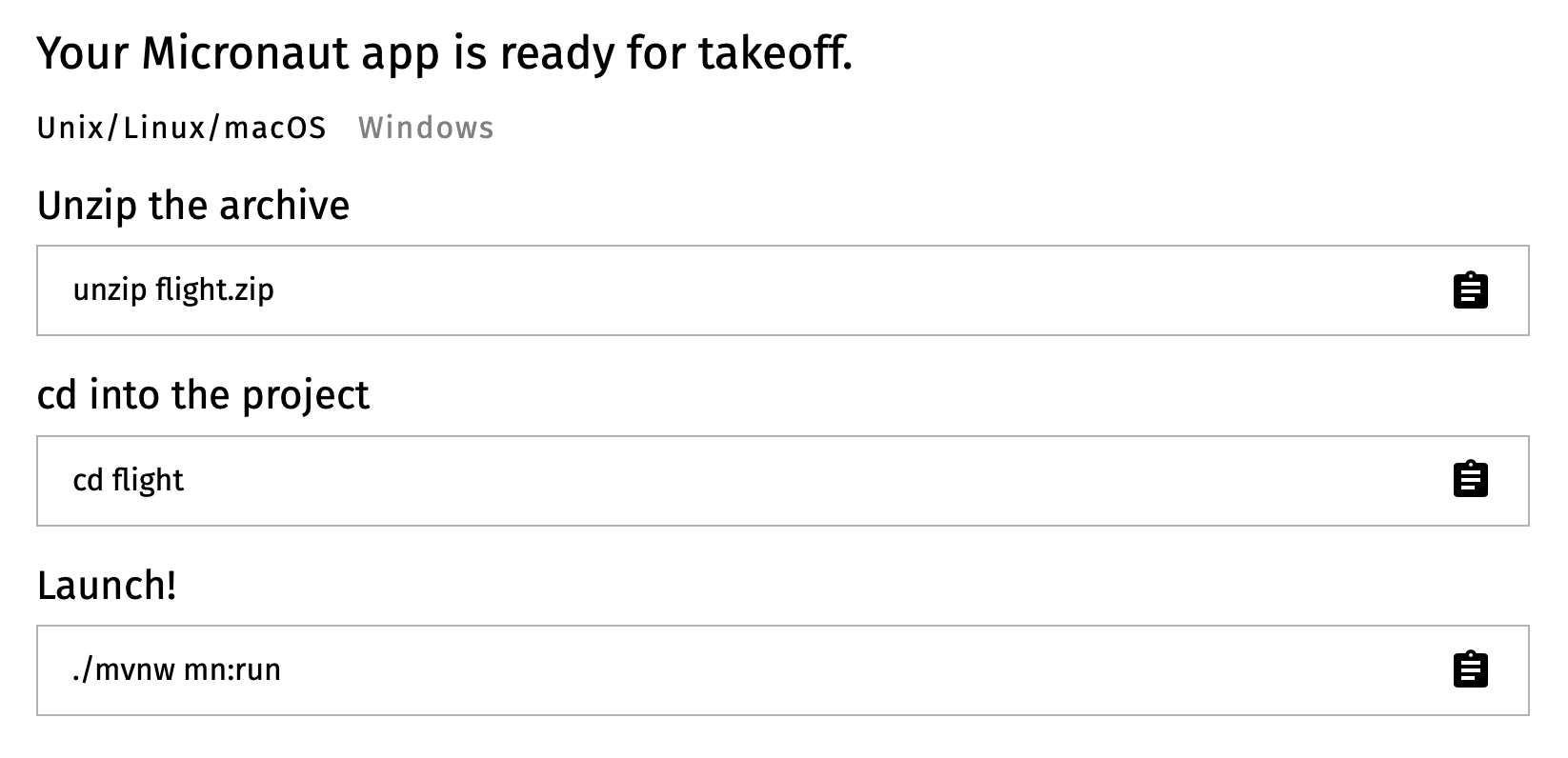
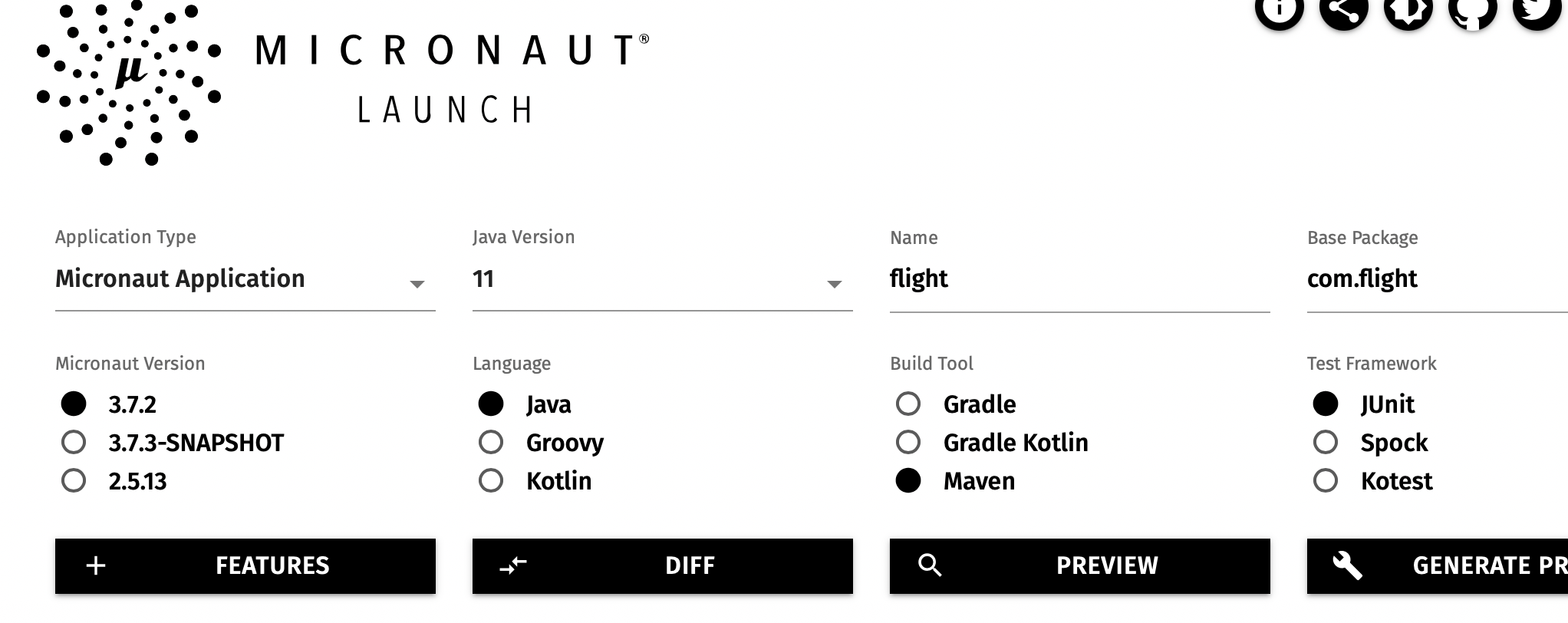
Consumers need to know how to call API and build an API client. API providers should provide a demo app to consume their API for the consumers.

**End users**

End users do not call the API directly, but use the app developed by API consumers.

**API Modeling**

**OpenAPI(Swagger)**

****

This tutorial walks you through steps for creating OpenAPI specs using Swagger(Micronaut for flight passenger API.

Import flight folder in Visual Studio Code(VSC)

Create model folder in VSC

Create controllers folder in VSC

Create Service Folder in VSC

Create Flight class in model folder using VSC. Paste following definition of the attributes of Flight class and the select pasted code and using light bulb generate getter and setter methods.

Create Passenger Class in model folder using VSC. Paste passenger attributes code and then generate getter and setter methods like for flight object.

package com.rest.domain;

import io.swagger.v3.oas.annotations.media.Schema;

@Schema(description="Passenger")

public class Passenger {

private String id;

private String name;

public String getId() {

return id;

}

public void setId(String id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

}

Create FlightService class in service folder using VSC and paste following folder. In this code we are creating flightRepo for storing flights in memory. Get methods will be implemented to fetch details of a flight and list of all flights.

package com.rest.service;

import com.rest.domain.Flight;

import java.util.Map;

import java.util.List;

import java.util.ArrayList;

import java.util.HashMap;

import java.util.concurrent.atomic.AtomicInteger;

public class FlightService {

static private Map<Integer, Flight> flightRepo = new HashMap<Integer, Flight>();

static private AtomicInteger idCounter = new AtomicInteger();

public Flight getFlight(String id) {

Flight flight = flightRepo.get(id);

return flight;

}

public List<Flight> getFlightsByPassenger(String passengerId) {

return new ArrayList<Flight>(flightRepo.values());

}

}

Create PassengerService class in service folder using VSC and paste following code

package com.rest.service;

import com.rest.domain.Passenger;

import java.util.Map;

import java.util.List;

import java.util.ArrayList;

import java.util.HashMap;

import java.util.concurrent.atomic.AtomicInteger;

public class PassengerService {

static private Map<Integer, Passenger> passengerRepo = new HashMap<Integer, Passenger>();

static private AtomicInteger idCounter = new AtomicInteger();

public Passenger getPassenger(int id) {

Passenger passenger = passengerRepo.get(id);

return passenger;

}

public List<Passenger> getPassengers() {

return new ArrayList<Passenger>(passengerRepo.values());

}

}

Create FlightController class in controller folder using VSC and paste following code

package com.rest.controller;

import com.rest.domain.Flight;

import com.rest.service.FlightService;

import io.micronaut.http.annotation.Get;

import io.micronaut.http.annotation.Controller;

import io.micronaut.http.HttpHeaders;

import io.micronaut.http.HttpResponse;

import io.micronaut.http.MediaType;

import io.micronaut.http.annotation.Produces;

import io.micronaut.http.annotation.Controller;

import io.micronaut.http.annotation.Delete;

import io.micronaut.http.annotation.Get;

import io.micronaut.http.annotation.Post;

import io.micronaut.http.annotation.Put;

import io.micronaut.http.annotation.Body;

import java.util.List;

@Controller("/flight") // <2>

public class FlightController {

FlightService flightService;

public FlightController(FlightService flightService) { // <3>

this.flightService = flightService;

}

@Get("/{id}")

public Flight getFlight(String id) {

Flight flight = flightService.getFlight(id);

return flight;

}

@Get("/passenger/{id}")

public List<Flight> getFlightsByPassenger(String id) {

List<Flight> flights = flightService.getFlightsByPassenger(id);

return flights;

}

}

Create PassengerController class in controller folder using VSC and paster following code

package com.rest.controller;

import com.rest.domain.Passenger;

import com.rest.service.PassengerService;

import io.micronaut.http.annotation.Get;

import io.micronaut.http.annotation.Controller;

import io.micronaut.http.HttpHeaders;

import io.micronaut.http.HttpResponse;

import io.micronaut.http.MediaType;

import io.micronaut.http.annotation.Produces;

import io.micronaut.http.annotation.Controller;

import io.micronaut.http.annotation.Delete;

import io.micronaut.http.annotation.Get;

import io.micronaut.http.annotation.Post;

import io.micronaut.http.annotation.Put;

import io.micronaut.http.annotation.Body;

import java.util.List;

@Controller("/passenger") // <2>

public class PassengerController {

PassengerService passengerService;

public PassengerController(PassengerService passengerService) { // <3>

this.passengerService = passengerService;

}

@Get("/{id}")

public Passenger getPassenger (int id) {

Passenger passenger = passengerService.getPassenger(id);

return passenger;

}

@Get

public List<Passenger> getPassengers() {

List<Passenger> passengers = passengerService.getPassengers();

return passengers;

}

}

To get started add Micronaut’s openapi to the annotation processor scope of build configuration in pom.xml file.

<path>

<groupId>io.micronaut.openapi</groupId>

<artifactId>micronaut-openapi</artifactId>

<version>4.0.1</version>

</path>

For swagger annotation add following to pom.xml file -

<dependency>

<groupId>io.swagger.core.v3</groupId>

<artifactId>swagger-annotations</artifactId>

</dependency>

Once dependencies has been configured minimum requirement is to add to Application class.

import io.swagger.v3.oas.annotations.OpenAPIDefinition;

import io.swagger.v3.oas.annotations.info.Contact;

import io.swagger.v3.oas.annotations.info.Info;

import io.swagger.v3.oas.annotations.info.License;

@OpenAPIDefinition(

info = @Info(

title = “Flight",

version = "0.1",

description = "Flight API",

license = @License(name = "Apache 2.0", url = "https://foo.bar"),

contact = @Contact(url = "https://gigantic-server.com", name = "Fred", email = "Fred@gigagantic-server.com")

))

Compile application using command “mvn package”.

cd target/classes/META-INF/swagger

Generated OpenAPI YAML in file flight-0.1.yml.

Once you have modeled API, you can generate a document which could be shared with API consumers. Swagger allows to make API access in the browser and more readable. Next we will configure swagger.

Configure following in application.yml file to enable swagger. You could find application.yml file is src/main/resources folder:

micronaut:

router:

static-resources:

swagger:

paths: classpath:META-INF/swagger

mapping: /swagger/\*\*

With the above configuration in place when you run your application you can access your Swagger documentation http://localhost:8080/swagger/flight-0.1.yml at

To Review

In this chapter we started with API design strategies and then looked into API creation process and modeling. Best practices for REST API design are discussed, followed by API solution architecture. We compared API modeling tools, and then developed an API for flight passenger using Micronaut.