RAML is an example of an API specification language. Another API specification language is the **OpenAPI Specification (OAS)**

**What are API specifications?**

Some APIs follow strictly defined protocols. However, RESTful APIs follow an architectural style that was laid out by Roy Fielding in his doctoral dissertation. This means that there are general principles laid out that they follow, but there is not a strict protocol that they must adhere to. This balance between structure and flexibility has been a powerful concept and has contributed to the widespread adoption of this kind of API architecture. There is no such thing as a perfect solution, however, and this is no different. One of the benefits of **SOAP APIs** is that the structure must be specified following strict programmatic rules. This means that the API definition must be written in a way that computers can easily understand. This makes it easy to create a lot of general-purpose automation for these kinds of APIs. If a computer can understand the layout of an API, you can automatically generate some kinds of documentation, tests, and even code from that specification.

Without that strict specification, each **RESTful API** may have some nuances to it that are different from others. As humans, this is generally fine to manage. We can quickly and easily figure out where those differences are and accommodate for them. Computers struggle with this, however. Things must be very clearly and explicitly laid out for a computer to use it. This challenge with RESTful APIs was recognized pretty early on, and so some API specification formats were proposed.

API specifications provide a structured way to define an API. If you follow the rules of the specification, you can interact with the API at a higher level of abstraction. For example, an API specification could be used to automatically create mock servers that you can use for testing and experimentation during development. You can also do things such as automatically generate documentation and contract tests. You can even use these specifications to generate some kinds of code, for both server implementations and client-side code that calls the API.

**Подходы к созданию API**

* **Code First**
  + При таком подходе в начале пишется код, а потом автоматически генерируется документация в Swagger.
    - Плюсы:
      * Легкое создание документации
      * Синхронизация кода и документации API (так как при изменении в коде, подтягиваются изменения в документацию)
    - Минусы
      * Отсутствие параллельной разработки (человек кто захочет с нами интегрироваться и заранее писать код, до того как мы создадим сервис этого сделать не сможет, так как еще не будет документации)
      * Отсутствие цели у команд (так как нет понимаю что мы будет делать)
      * Отсутствие кросс-платформенной совместимости
* **Contract First**
  + При таком подходе в начале пишется документация, а потом пишется код
    - Плюсы:
      * Возможность параллельной разработки
      * Есть цели у команд
      * Кросс-платформенная совместимость
      * Возможность повторно использовать схему
    - Минусы:
      * Дополнительные начальные затраты (нужно определить соглашения в обслуживании – договориться что договор четко определен и не меняется очень часто)
      * Механизм для обновления контракта и обмена

**Types of API specification**

**API specification languages** – another term for this is an **API description format**. You can follow the API description format to create an API description itself, which is the actual metadata that describes what the API contains and can do. There are multiple API description formats, and there are also tools that will allow you to take an API description that is written in one format and convert it to another.

There are three main RESTful API specification languages:

1. RAML
2. API Blueprint
3. OpenAPI/Swagger

**RAML**

This specification language uses the YAML format. YAML is a human-readable file format that makes it easy to structure data by following a few simple rules. The rules for structuring YAML are in the YAML spec (<https://yaml.org/spec/1.2/spec.html>). Since YAML is designed to be human-readable, it is quite easy to get started with and helps to make RAML intuitive to understand. Much of the structure is built in a way that matches how you should think about and design APIs. This makes RAML a good choice if you are trying to create "design-first" APIs where you plan out the API ahead of time and then create an API that matches that design.

RAML is supported by Postman and is probably the second most popular API specification language. Although not as popular as OpenAPI, it has broad community support as well, and so if you are looking for a design-first approach, it may be a good choice

**API Blueprint**

The API Blueprint specification uses Markdown as the format. You can read more about Markdown here: https://www.markdownguide.org/cheat-sheet/. Using Markdown makes specifications written in this format more readable. In fact, it reads almost exactly like consumer-facing documentation. It is also very easy to understand and get started with. However, it isn't as widely supported by the community. If you want to use it in Postman, you have to first use a tool to convert it into a format that Postman can use.

The API Blueprint format is integrated into some tool stacks such as apiary (https:// apiary.io), but if you are working with Postman, it is probably better to choose a different API specification language if you can.

**OpenAPI/Swagger**

The most used API specification language is OpenAPI. Since it was originally called Swagger, many of the tools that support the OAS are called Swagger tools, while the specification itself is called OpenAPI. This specification language is flexible and powerful and has a lot of tooling to support it. You can directly import descriptions written in this format into Postman, and you can use a lot of other tools to automate things related to it as well. This large toolset and broad adoption has in some ways made it the de facto standard for API specifications. If you aren't sure which specification you want to use, it's probably best to just default to using OpenAPI.

**Swagger** - это фреймворк для спецификации RESTful API. Его прелесть заключается в том, что он дает возможность не только интерактивно просматривать спецификацию, но и отправлять запросы – так называемый Swagger UI. С 2016 года он получил новое обновление и стал называться OpenAPI Specification.

**OpenAPI** - это спецификация/стандарты

**Swagger имеет два подхода к написанию документации:**

**Документация пишется на основании вашего кода.**

* Данный подход позиционируется как "очень просто". Нам достаточно добавить несколько зависимостей в проект, добавить конфигурацию и уже мы будем иметь нужную документацию, хоть и не настолько описанной какою мы хотели.
* Код проекта становиться не очень читабельным от обилия аннотаций и описания в них.
* Вся документация будет вписана в нашем коде (все контроллеры и модели превращаются в некий Java Swagger Code)
* Подход не советуют использовать, если есть возможности, но его очень просто интегрировать.

**Документация пишется отдельно от кода.**

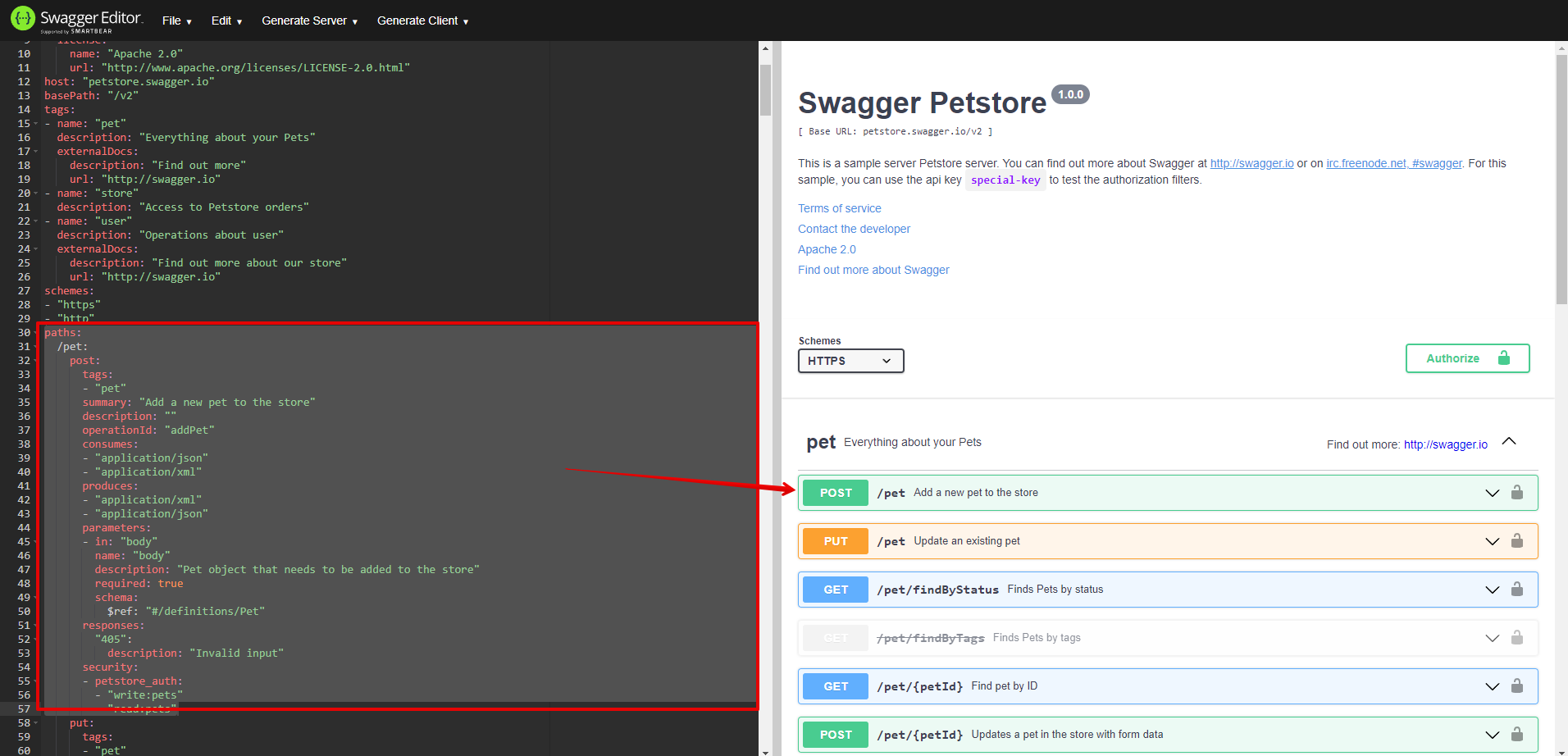
* Данный подход требует знать синтаксис Swagger Specification.
* Документация пишется либо в YAML/JSON файле, либо в редакторе Swagger Editor.

**Swagger или OpenAPI framework состоит из 4 основных компонентов:**

* **Swagger Core** - позволяет генерировать документацию на основе существующего кода основываясь на Java Annotation.
* **Swagger Codegen** - позволит генерировать клиентов для существующей документации.
* **Swagger UI** - красивый интерфейс, который представляет документацию. Дает возможность просмотреть какие типы запросов есть, описание моделей и их типов данных.
* **Swagger Editor** [**https://editor.swagger.io/**](https://editor.swagger.io/) - Позволяет писать документацию в YAML или JSON формата.

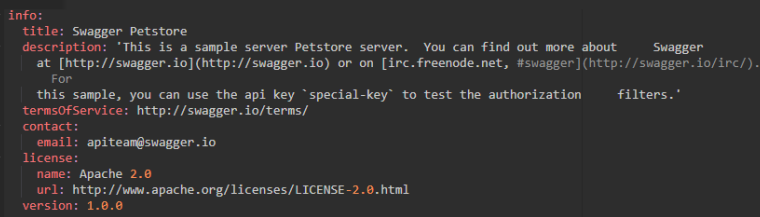
**Swagger Editor**

On the right-hand side, the editor shows a preview of the documentation that can be produced from the specification. On the left we see graphical representation of what we can do. As soon as we update data on the left side – it’s updated on the right side

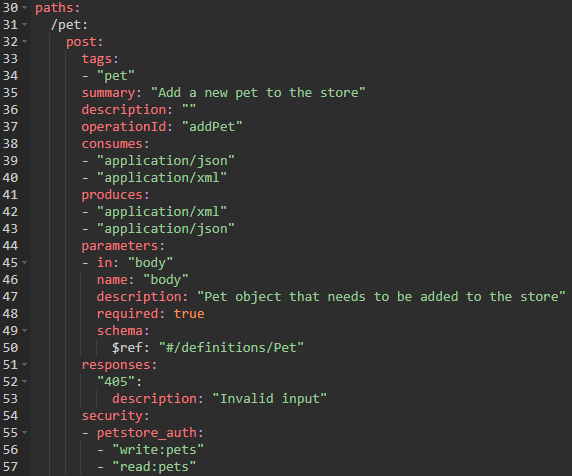


**Parts of an OAS in Swagger**

* **info**. This is where the base data for the API object is defined. At a minimum you will need to include the title, description, and version options. There are a few other optional high-level tags that help specify metadata about the API. These tags are helpful but not necessary for describing your API. The following screenshot shows different options in the info section



* **paths**. This section is critically important and there are a lot of options in here. Obviously, it is the place where you define your endpoints, and under each endpoint, you can define the various actions available on that endpoint. You can also define a lot of detail about each endpoint/action combination, things such as *what kind of responses this endpoint can give and what kind of data it requires, as well as descriptions of what the API does*.

let's walk through the **/pet** endpoint to get a bit more of a feel for this.  
 

The first thing that is defined is the actual endpoint itself (**/pet**). The next thing you specify is what actions you can do on that endpoint. In our case there are 2 methods for our endpoint – POST and PUT

The next things under those verbs are some parts of the spec that define some information about the endpoint. These include things such as summary, operationID, and the request body description. Most of these kinds of keys are optional but can be helpful for documenting and organizing your API.

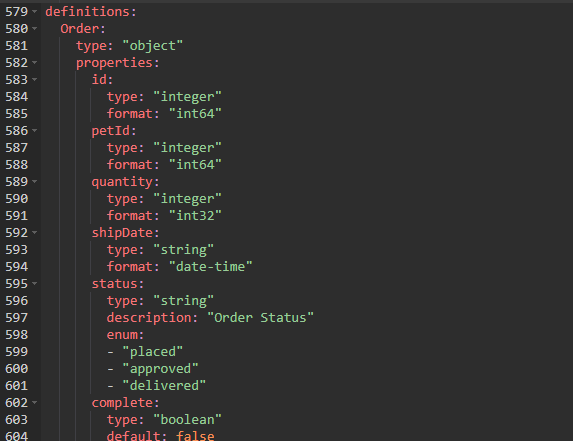
The **consumes** section defines what needs to go into the body of the API call. In this section, we need to define the content type (it can be either application/json or application/xml in this example) and then we need to define the **schema** or structure that this content needs to adhere to. You will notice that the schemas are referenced using a **$ref** key. This is merely a key that points to values that are stored in another part of the document. You could spell out the entire schema directly in this part of the document. However, in this case, the same schema is being used in multiple requests and so it has been moved to another section of the document called components. This makes it possible to reference the same thing from multiple places.

Another important part of any API call is the **response**. The spec defines what should happen for each response that the API can give in the responses section. These responses are described in the spec under the different HTTP error codes that you want your API to support. The specification also lays out the security options for each action on the endpoint. The actual definition of the security options for the API can be found in the securitySchemes section near the bottom of the file.

The paths sections lay out most of the details of what each endpoint can do, but as I noted earlier, the schema for the bodies of these requests is defined in another part of the document called **schemas**

At its most basic, a **schema** is just a plan, usually in the form of a model or theory. Schemas are common in software development. For example, databases have schemas that show how the various tables are connected to each other. In a similar way, an API can have a schema. An API schema shows how the data should be structured. It will specify **what a response should contain**, and **what data you need to include in the body of POST or PUT requests**.

Schemas usually start in the same document, in the end of it. It starts with a word **definitions**

In many ways these are the core of the spec. They define what the incoming and outgoing data in the API calls need to look like

Any request that uses this schema is required to have a **name** field and a **photoUrls** field. The name field needs to be of the string type and the photoUrls field needs to be an array of strings. There are also some other optional fields that you can include, such as status and tags. Another interesting thing to note is that the optional category field references another schema. In other words, to fully see the schema, we have to scroll up to the section that has the category schema in order to see that it is an object with an ID and name.