# API Contracts

An API contract is a definition of how API services are to be consumed by the API consumers. In the first stages of API development, as instructed in previous Section 2.1, Development Lifecycle, an API contract is to be drawn up between the API provider and API consumer. Creating such an API contract (or service definition) is an important initial step in API development that is often overlooked. API definitions (sometimes referred to as API specifications) are designed to provide a format for describing APIs that is readable by both humans and machines. They are language-agnostic and commonly based on a widely agreed-upon format such as OpenAPI Specification (aka Swagger), WSDL (in a case of SOAP services), or RAML.

One of the most important aspects of microservices architecture is the ability to deploy microservices independently of one another. To achieve such microservice independence, each microservice must provide a **versioned, well-defined contract** to its consumers. Each service must not break these versioned contracts until it's known that no other service consumer relies on a particular, versioned contract.

Therefore, the following architecture aspects are critical when thinking about API contract design:

* Create the API contract early on in your design of the microservice or an API service
* Define API contract format and adherence to a particular well-established contract standard, such as OpenAPI, WSDL, or RAML
* Version your API contract and notify your API/microservice consumer when a new version is released
* Don’t take down an API contract version unless you know that no other service consumer relies on that particular contract version

## Design First Methodology

The Design First methodology instructs that we ought to create an API contract or service definition before even starting our first line of development. Creating such an API contract is a key part of the design phase, hence following this methodology, we are creating the API contracts first, publish them to the API Developer Portal, mock the APIs, and only then start full development of “live” API service operations.

An API Contract is created in the initial design phase of any microservice API development. Following the Development Lifecycle, defined in Section 2.1, API Contract is to be initially drafted in the very first phase of API development lifecycle, the “Design” stage. Design First methodology instructs the developer to design their API first by creating an API contract and socializing that contract with any potential API consumer or other microservice developers who would be integrating their services with yours.

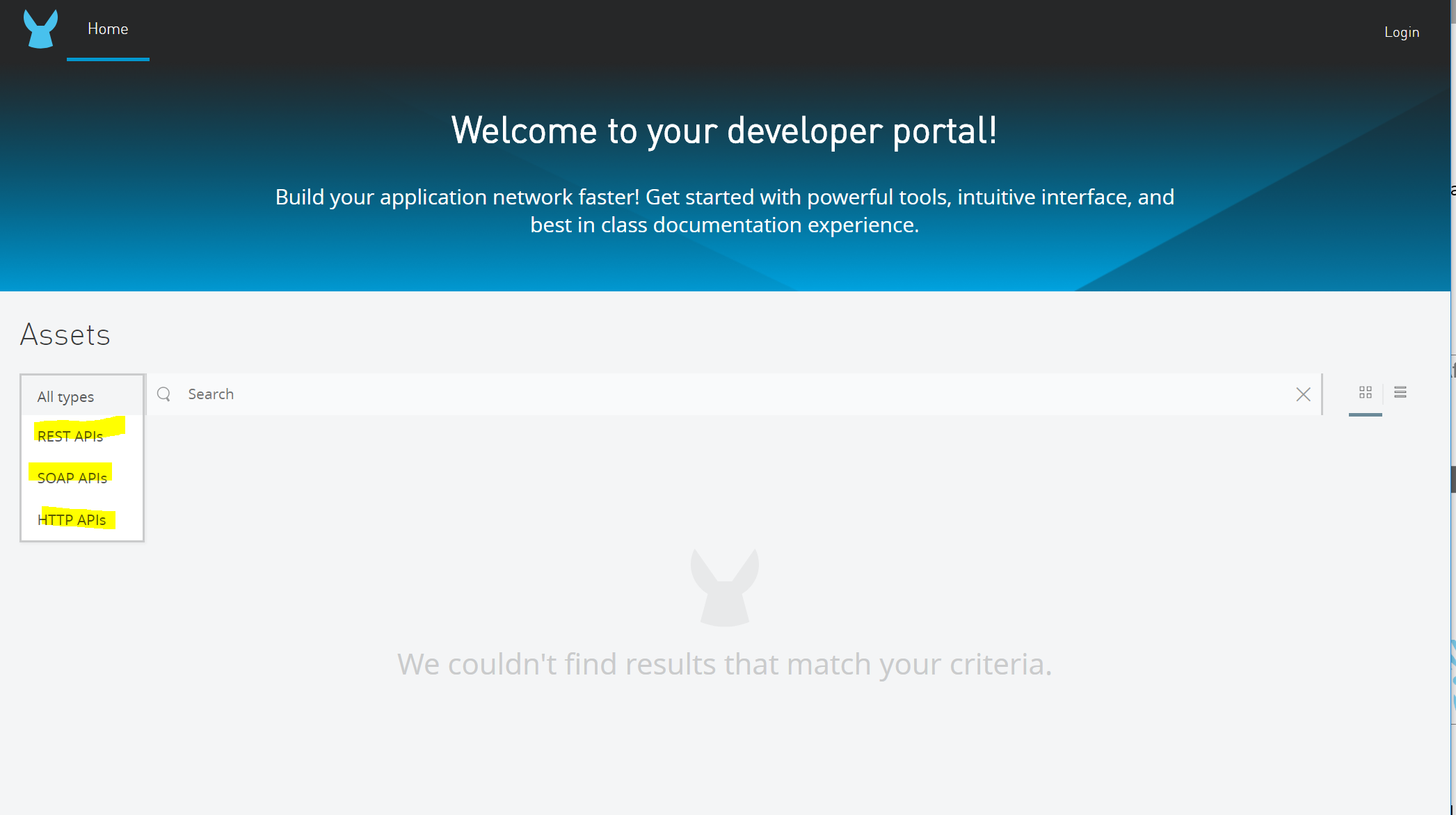
Example

As an example, as a developer, you are given a business use case to retrieve a data object from the source system such as a relational database via a programming language of choice (.NET, Java, NodeJS, Python, etc) and expose this database object in a form of a JSON object to be send over HTTP/REST to an external data consumer. As a developer, you should first draft up an API contract by defining the JSON schema of an object to be fetched via your microservice to be consumed by other microservices or platforms. To draft such a contract, you should define the following:

1. API Services to be consumed by your service consumers (e.g., “xyz-database-retrieval-service”)
2. Service Operations (e.g., Retrieve single record, Retrieve a collection, Insert New Record, Update a Record, etc)
   1. In REST APIs, express these operations as HTTP/REST operations such as GET, POST, PUT, DELETE
   2. In SOAP APIs, express these operations as separate SOAP Operations inside a service definition WSDL file
3. Request Requirements
   1. What inputs are expected for each service operation (e.g., a unique Customer ID to fetch customer details from the database)
   2. Which of these inputs are required
   3. What data type is associated with each of the inputs
4. Response Requirements
   1. What outputs are expected for each service operation
   2. What data type is associated with each of the outputs
   3. What is an example of a data item returned back (provide either dummy data or real customer information, scrubbed from any sensitive information, for security reasons)
5. Service Security Requirements
   1. Does the service expect an OAuth 2.0 Bearer token
   2. Does the service expect an WSSE Security Header (for SOAP services)
   3. Does the service expect a Basic Auth user name and password

Once the contract is drawn up, you can proceed to socialize that contract with the potential consumers of your microservice. To socialize the API contract, you should publish the contract into an API Portal. An API Portal is a web-based publishing platform which allows developers to share their API contract definitions, inform others of new APIs that are to be developed, find new APIs, and share API contracts with external partners and consumers.

An example API Developer Portal with some sample APIs is provided below. The screenshot shows REST APIs, SOAP APIs, or HTTP APIs that can be easily discovered by searching for specific term or just listing all that are available.



Once the API contract is created and published to the API Developer Portal, you can proceed with creating a mock API to be published together with the API contract. Creating a mock allows your API consumers to immediately test out your API operations, without guessing what data is expected to be returned.

Luckily, if you are have chosen to create your API Contract either using OpenAPI Specification or RAML, the mock service can be auto-generated using an API Console that can interpret your API contract and, provided you have provided examples for each of your inputs and outputs, generate mock API operations.

For example, Swagger UI available within SwaggerHub or within a programming language such as Java and NodeJS (if your microservice is written in those languages) already auto-generates mock services based on your contract. As does the Mulesoft API Design Center, which allows a quick generation of mock API requests/responses by a turn of a configuration switch.

## API Contract Standards

### Contract Standards

Depending on the type of an API service written (REST or SOAP), there are these widely-used API Contract standards in the API development community:

1. REST API Contract Standards:
   1. OpenAPI Specification
   2. RESTful API Modelling Language (RAML)
2. SOAP API Contract Standards:
   1. SOAP 1.1
   2. SOAP 1.2

OpenAPI Specification

OpenAPI Specification (OAS) is a specification for machine-readable interface files for describing, producing, consuming, and visualizing RESTful web service, overseen by the Open API Initiative, under the Linux Foundation. Certain commercial platforms, such as SmartBear SwaggerHub, have been launched to be based on the OAS. These platforms allow developers to create Swagger 1.x, 2.x, or (lately) 3.0 specifications for their APIs, as well as publish and socialize these Swagger API contracts with others within the organization or with their customers.

RAML

While Mulesoft has recently joined OAS and now actively supports Swagger, Mulesoft continues to maintain its own API Contract Standard called RESTful API Modelling Language (RAML). Mulesoft provides funding for the RAML initiative, but the RAML itself is considered to also be open to the API development community for contributing to the RAML specification.

SOAP

SOAP contracts are expressed in a form of Web Service Definition Language (WSDL), an XML-based contract that describes a SOAP service, its operations and custom entities. Original SOAP 1.1 places restriction on all serializations to follow the XML 1.0 serialization standard. Version 1.2 places no such restriction, allowing serialization to either follow XML 1.0 serialization or HTTP, for example.

### API Specifications

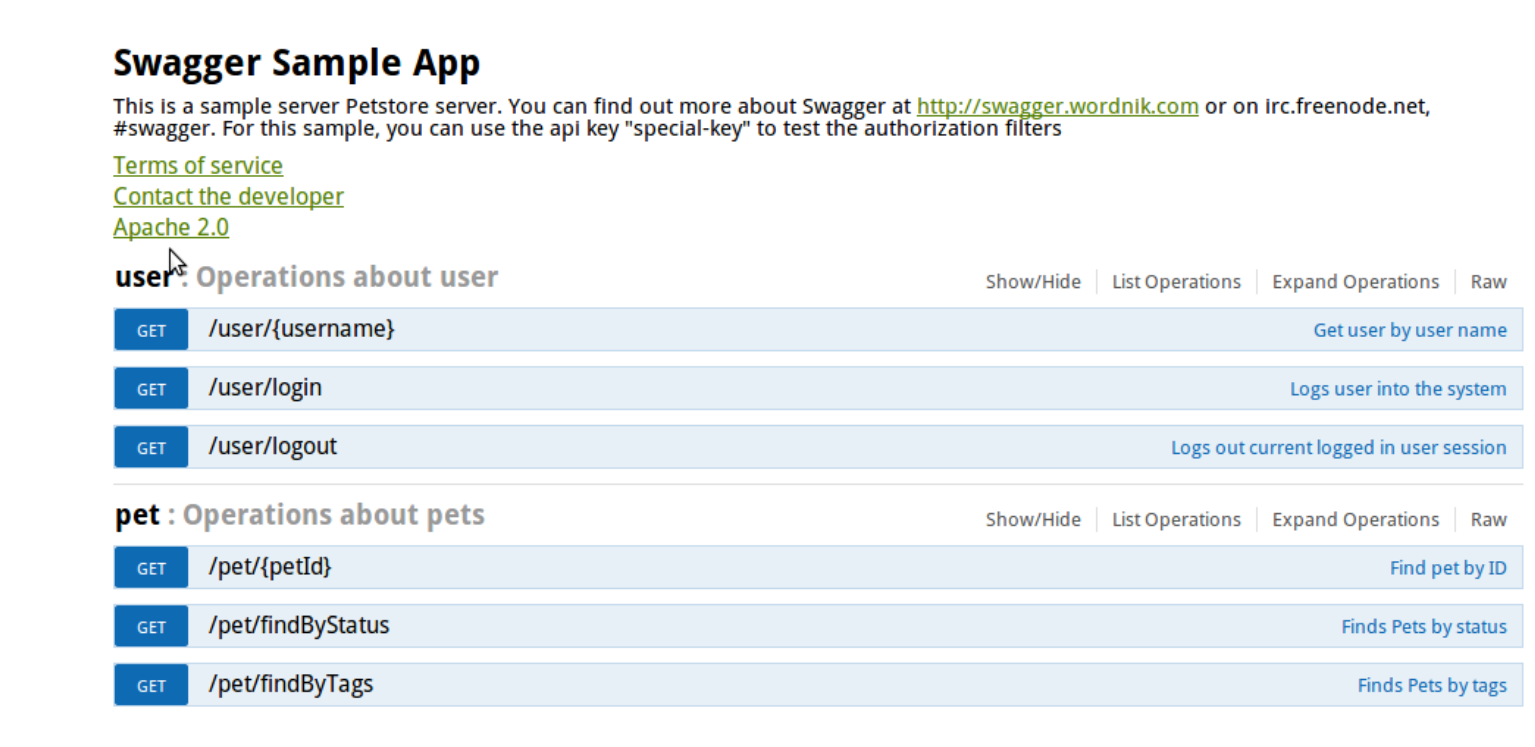
When developing an API Contract, follow these key guidelines to make sure to include all pertinent service definition information is included in the contract:

API Service Operations

Define all API Services to be consumed by your service consumers (e.g., “xyz-database-retrieval-service”). Then, define all Service Operations within each service (e.g., Retrieve single record, Retrieve a collection, Insert New Record, Update a Record, etc).

* + In REST APIs, express these operations as HTTP/REST operations such as GET, POST, PUT, DELETE
  + In SOAP APIs, express these operations as separate SOAP Operations inside a service definition WSDL file

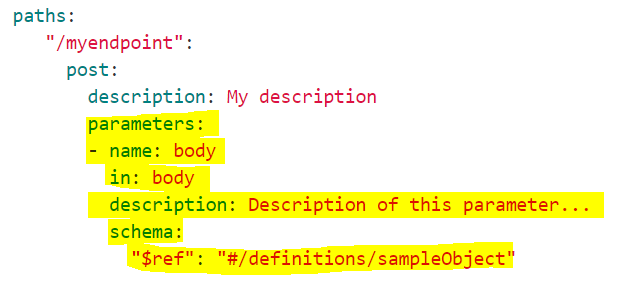
An example of a service operation is provided below for a RESTful operation:



Request Requirements

In the request section of the API contract, list all inputs, their data types, whether they are required, and how they are expected to be passed in (in header, part of the path, on a querystring, or in the request body for POST and PUT requests).

An example of a request definition in the API contract is provided below:



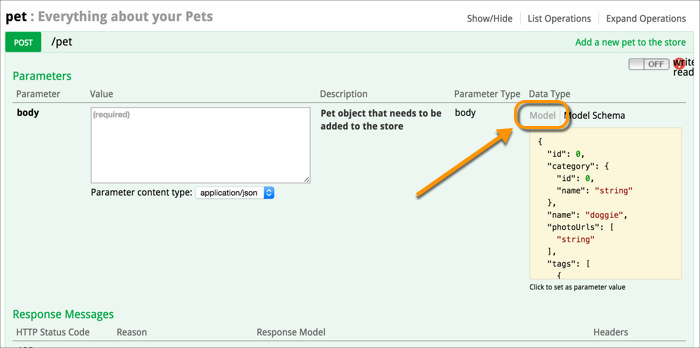
Response Requirements

In the response section of the API contract, list all structures and attributes expected to be returned back in the response for each operation. In the case of RESTful services returning JSON, define the JSON schema files and reference those schema files inside your Swagger or RAML contract definition. In the case of SOAP, define all outputs as part of WSDL contract.

For JSON-based responses, define the following:

* Parent JSON structure (object or collection)
* Attribute data types
* Any nested objects or collections within your parent JSON structure
* Each attribute listed should contain the following:
  1. Data Type (e.g., string, number, date, boolean, array)
  2. Example (i.e., provide a concrete example for each data item returned, with any sensitive information scrubbed or replaced with dummy data for an example)
  3. Formatting required (e.g., whether special formatting for emails, phones, dates is required)

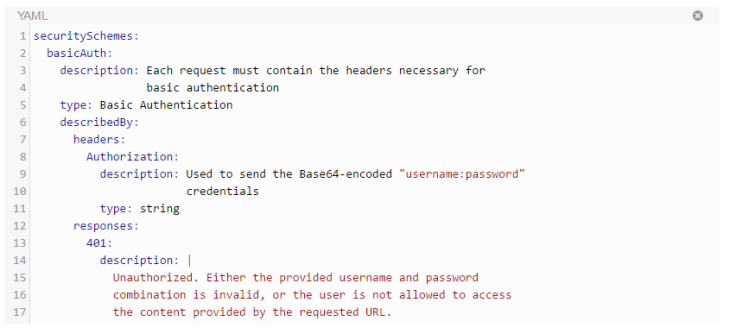
Example of a response model for a Swagger sample pet store is provided below:



Service Security Requirements

Security required for an API can be defined in a form of a Swagger or RAML security definition section of the contract. A security section defines whether the service is to be secured via OAuth 2.0, basic auth, or other form of security.

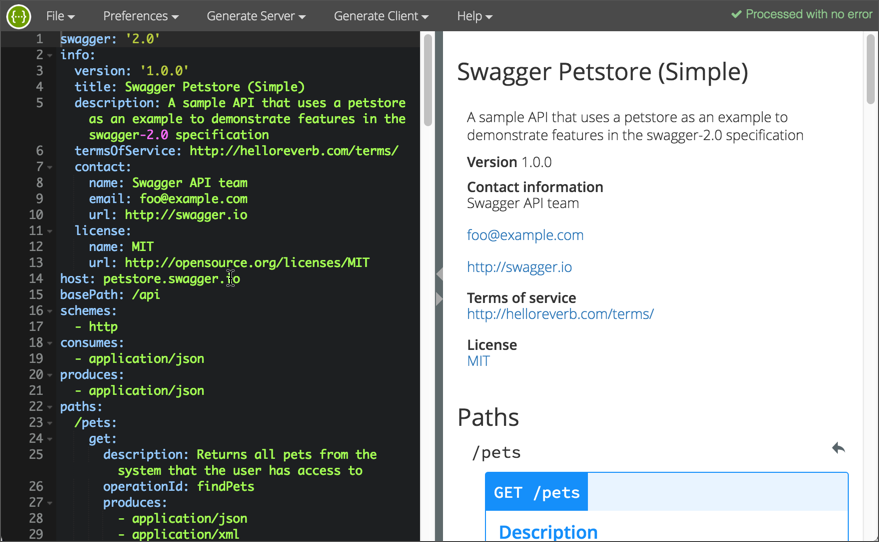
An example below shows the definition of a Basic Auth security scheme for a given API.



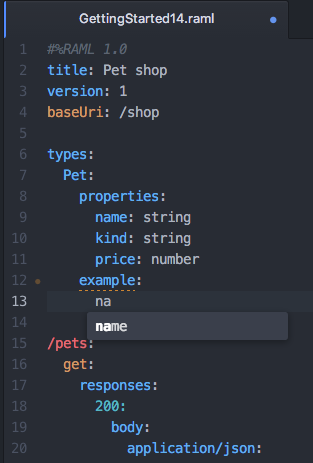
### Swagger and RAML

As stated in Section 4.2.1, both Swagger and RAML are widely used to define API contracts for REST APIs.

An example of an OpenAPI Specification based API contract is provided below:



An example RAML contract is provided below:



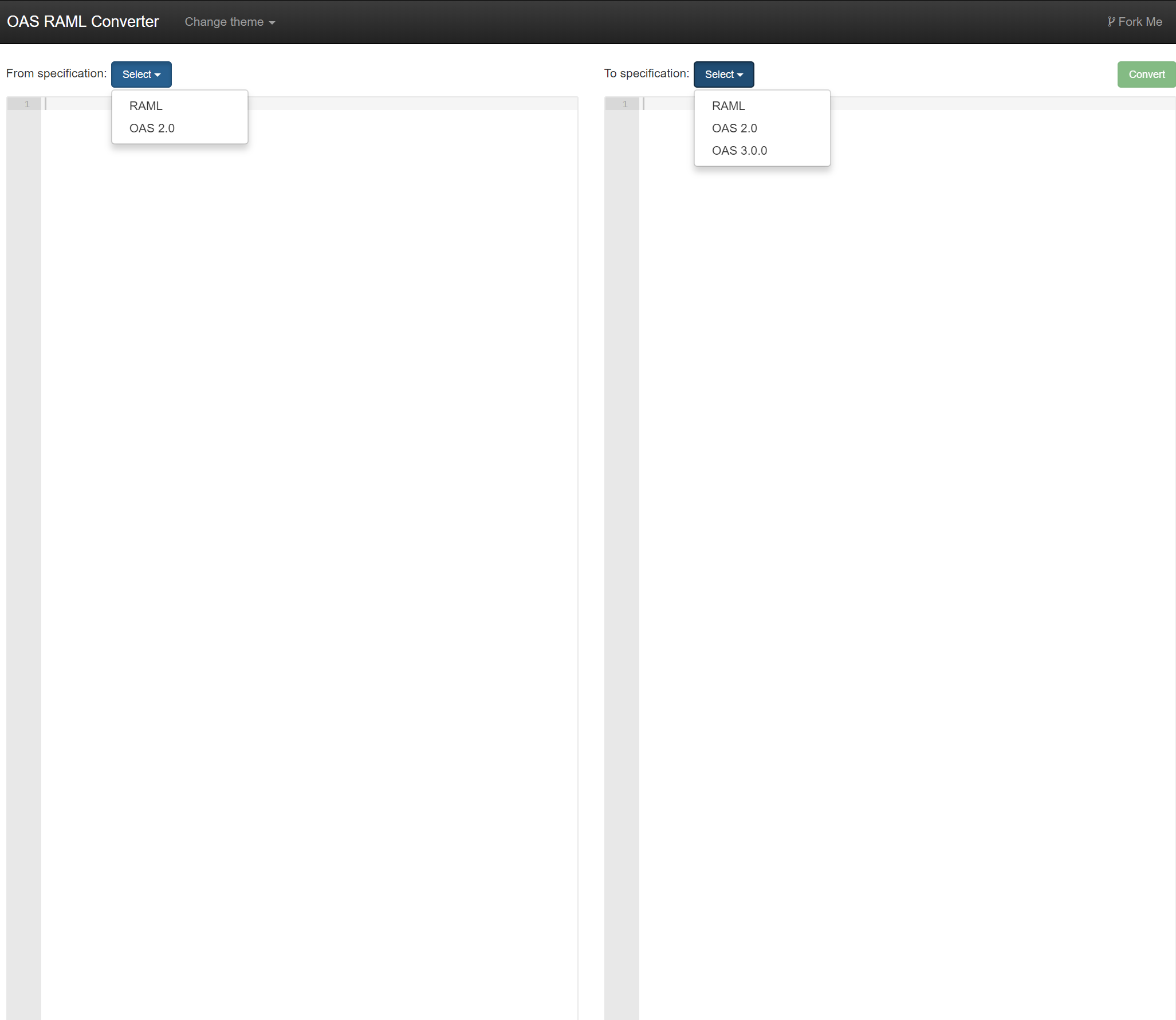
Both Swagger and RAML are very similar, especially if using YAML to define both. There are some slight differences between the two contract definition languages but they are very minor.

As a result, Mulesoft has worked on a conversion utility that allows to convert from OAS Swagger to RAML, to help migrate any non-Mulesoft APIs onto the Mulesoft platform. The utility also allows converting from RAML into OAS Swagger.

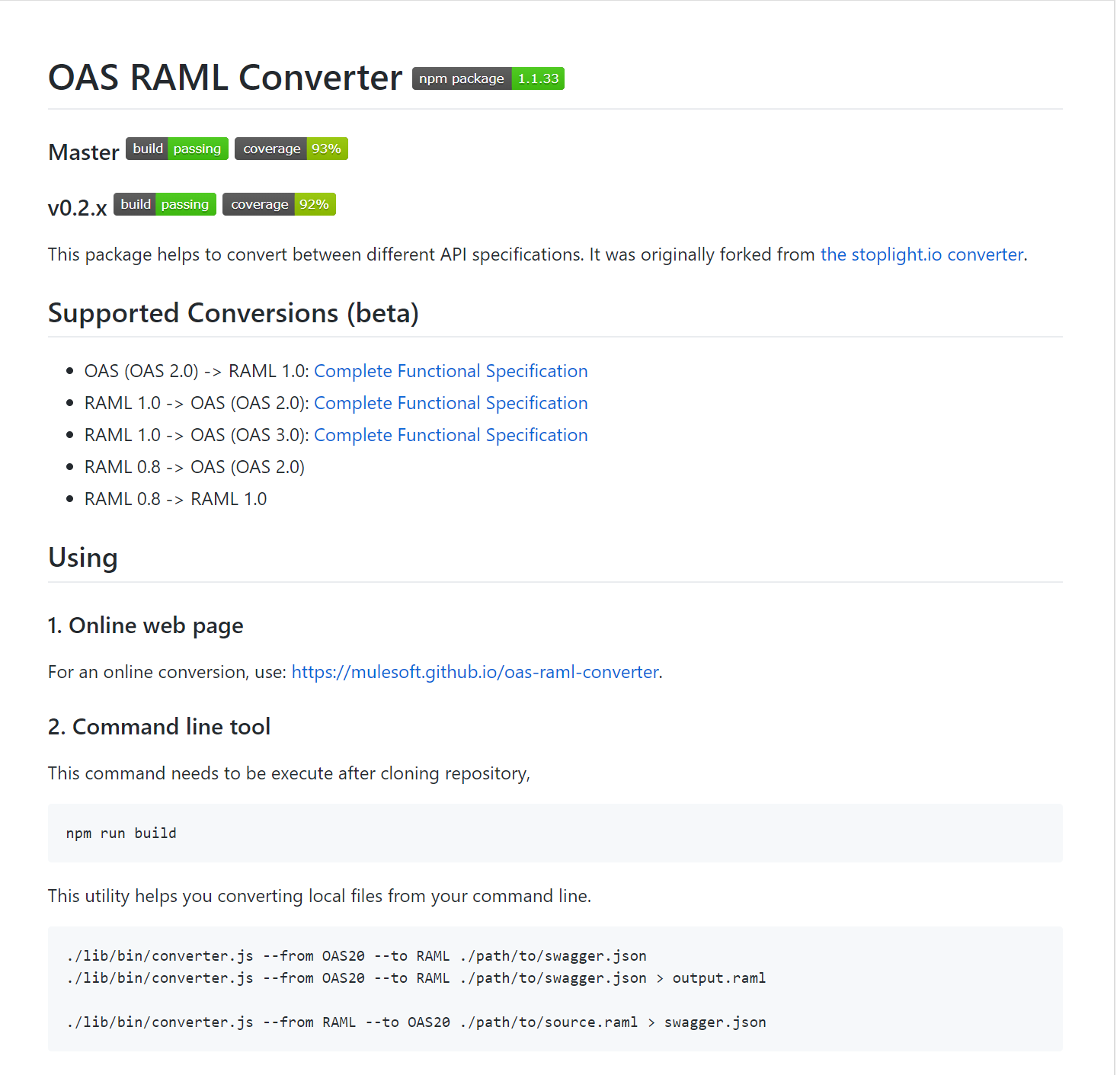
#### Conversions

Mulesoft has developed the OAS RAML Converter, which is a NodeJS based tool that was originally created by an open-source foundation called Spotlight.io. Mulesoft has forked the codebase from that original NodeJS module to create their own. Current version of the utility, v0.2, allows converting from OAS Swagger 2.0 to RAML 1.0, or from RAML 0.8/1.0 into OAS Swagger 2.0/3.0.

While the command-line utility is based on NodeJS and requires Node to be pre-installed, Mulesoft has also launched a web-based converter, which can be used by passing one API contract in Swagger or RAML format and receiving the same API Contract in a different format:

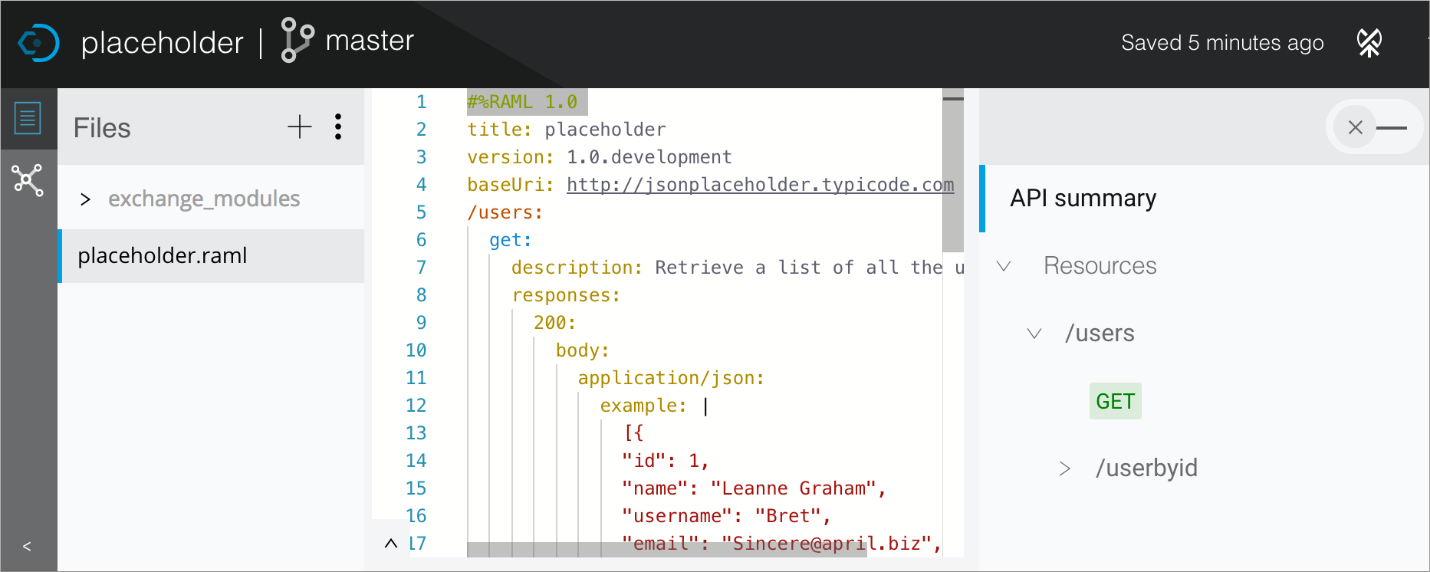


Here is a screenshot of the GitHub web page for this Mulesoft converter project:



## MuleSoft Design Center

Mulesoft has created an API Design Center (screenshot below), which allows a developer to create an API Contract online, within Anypoint Platform Design Center. The contract is auto-validated online, immediately showing the developer whether they are missing anything or misspelled any keywords. The Design Center also provides auto-suggestions on which appropriate keywords are to be used in each line of the contract, as the developer is typing the contract definition.



Once the contract is finished and saved, the developer can immediately see the API Console on the right pane, under “API Summary” section, as shown in the above screenshot. This allows the developer to test the API contract by sending a sample request and getting a response back.

The completed RAML contract can be saved and published to the Exchange from the Design Center.

The developer can also switch the tab on the top-right hand corner of the Design Center to convert the API to a mock API. Once that toggle is switched to “mock” mode, Mulesoft assigns a mock URI endpoint to this API, which can then be accessed via browser, SoapUI or Postman, to return mock responses from this API.

As a developer, you should start creating all of your API contracts on Mulesoft platform within the Design Center. Once the API contract is created, save the contract and publish the RAML to the Exchange of your organizational unit within VA parent organization. Then, you should convert the API to a mock API initially, following the Design First strategy to expose the API contract and mock to your API consumers. Once the mock API is up and running, inform your API consumers to start testing their client applications and microservices to consume your mock API. Only after you have completed your “live” microservice development and validated your API live requests and responses, you can publish your API to Exchange and Mulesoft runtimes, and update the API Contract definition inside the Design Center to no longer return mock responses.

Once any new API contract changes are made to the RAML contract, make sure to version your contract by iterating the version number on the RAML contract. Start with 0.1 version initially and iterate minor versions for minor defects or hotfixes. Iterate major contract versions for significant operation changes. For example, if a new field needs to be added to the response, but the operation remains the same, iterate a minor version (e.g., from 0.1 to 0.2). If there is a change in the operation signature itself (e.g. the URI now has a new path parameter), iterate a major version (from 0.2 to 1.0).