

department of veteran’s affairs

api developer playbook

A picture containing clipart

Description generated with very high confidence

**API Developer Playbook Blueprint Documentation**

**Maps to Framework Set #1**

**Version 0.2**

**June 22, 2018**

**Prepared by:**

**Jordan Braunstein**

**Max Girin**

****

****

Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Name** | **Version** | **Description** |
| 05/30/2018 | Jordan Braunstein  Max Girin | 0.1 | DRAFT version of Outline distributed for ASG Review |
| 06/12/2018 | Max Girin | 0.2 | DRAFT of Section 2.1 - Developer Lifecycle |
| 06/22/2018 | Max Girin | 0.3 | DRAFT of Section 4 – API Contracts |
|  |  |  |  |

[1. Introduction 6](#_Toc517560659)

[1.1 Purpose and Use 6](#_Toc517560660)

[1.2 VA API Vision 6](#_Toc517560661)

[1.3 Vendor Agnostic 6](#_Toc517560662)

[1.4 MuleSoft Applicability 6](#_Toc517560663)

[2. Standards 7](#_Toc517560664)

[2.1 Development Lifecycle 7](#_Toc517560665)

[2.1.1 Design First instead of Build First 7](#_Toc517560666)

[2.1.2 Follow Test Driven Development 8](#_Toc517560667)

[2.1.3 Create API Contracts and Mocks Early 8](#_Toc517560668)

[2.1.4 Iterate Frequently and Often 8](#_Toc517560669)

[2.1.5 Engage Your API Consumers 9](#_Toc517560670)

[2.1.6 Development Lifecycle Stages 9](#_Toc517560671)

[2.2.1 FHIR 11](#_Toc517560672)

[2.2.1.1 Argonaut 11](#_Toc517560673)

[3. API Layers 12](#_Toc517560674)

[3.1 Loose Coupling 12](#_Toc517560675)

[3.2 Policy Impacts 12](#_Toc517560676)

[3.3 API Management Impacts 12](#_Toc517560677)

[3.4 Layer Responsibilities 12](#_Toc517560678)

[3.4.1 Experience 12](#_Toc517560679)

[3.4.2 Process 12](#_Toc517560680)

[3.4.3 System 12](#_Toc517560681)

[3.4.3.1 Native System API 12](#_Toc517560682)

[3.4.4 Decision Tree 12](#_Toc517560683)

[4. API Contracts 13](#_Toc517560684)

[4.1 Design First Methodology 13](#_Toc517560685)

[4.2 API Contract Standards 15](#_Toc517560686)

[4.2.1 Contract Standards 15](#_Toc517560687)

[4.2.2 API Specifications 15](#_Toc517560688)

[4.2.3 Swagger and RAML 18](#_Toc517560689)

[4.2.3.1 Conversions 19](#_Toc517560690)

[4.3 MuleSoft Design Center 21](#_Toc517560691)

[5. Naming Conventions 22](#_Toc517560692)

[5.1 API Naming Standards 22](#_Toc517560693)

[5.1.1 Alignment with FHIR and Argonaut 22](#_Toc517560694)

[5.1.2 Alignment with Version Control 22](#_Toc517560695)

[5.2 Environment Properties 22](#_Toc517560696)

[5.3 Configuration Parameters 22](#_Toc517560697)

[5.4 API REST and SOAP Standards 22](#_Toc517560698)

[5.4.1 RESTful URI Naming Standards 22](#_Toc517560699)

[5.4.2 SOAP Message XML Standards 22](#_Toc517560700)

[5.4.3 JSON Schema Naming Standards 22](#_Toc517560701)

[5.4 Objects and Methods 22](#_Toc517560702)

[5.4.1 Canonical Naming 22](#_Toc517560703)

[5.5 Middleware Artifacts (flows, sub-flows, transformations, variables) 22](#_Toc517560704)

[6. Configuration Management 22](#_Toc517560705)

[6.1 API Configuration Management 22](#_Toc517560706)

[6.1.1 Version Numbers 22](#_Toc517560707)

[6.1.2 GitHub Integration 22](#_Toc517560708)

[6.1.3 Branching and Merging 22](#_Toc517560709)

[6.2 Environment Configuration 22](#_Toc517560710)

[7. Unit Testing 22](#_Toc517560711)

[7.1 Unit Testing Approach 22](#_Toc517560712)

[7.2 Agnostic Unit Testing 22](#_Toc517560713)

[7.3 MuleSoft MUnit 22](#_Toc517560714)

[7.3.1 Mocking Features 22](#_Toc517560715)

[7.4 Integration with Continuous Integration Continuous Deployment 22](#_Toc517560716)

[8. Developer Setup 23](#_Toc517560717)

[8.1 MuleSoft Developer Setup 23](#_Toc517560718)

[8.1.1 Maven 23](#_Toc517560719)

[8.1.2 GitHub Plugin 23](#_Toc517560720)

[8.1.3 Munit Setup 23](#_Toc517560721)

[9. Exception Handling 23](#_Toc517560722)

[9.1 Exception Handling Approach 23](#_Toc517560723)

[9.1.1 Transactions vs. non-Transactions 23](#_Toc517560724)

[9.2 Global Exceptions 23](#_Toc517560725)

[9.3 API-Level Exceptions 23](#_Toc517560726)

[9.4 Middleware-Level Exceptions 23](#_Toc517560727)

[9.5 Message Process Failures and Retries 23](#_Toc517560728)

[9.5.1 Message Retries 23](#_Toc517560729)

[9.5.2 Handling Message Failures 23](#_Toc517560730)

[9.6 Exception Logging 23](#_Toc517560731)

[9.7 Notification and Alerting 23](#_Toc517560732)

[9.8 Integration with Queuing 23](#_Toc517560733)

[10. Baseline Logging 23](#_Toc517560734)

[10.1 Logging Framework Used within Each API 23](#_Toc517560735)

[10.2 Logged API Messages 23](#_Toc517560736)

[10.3 API Audit-Level Logging 23](#_Toc517560737)

[10.4 Integration with Third-Party Log Aggregators 23](#_Toc517560738)

[11. Baseline API Security 23](#_Toc517560739)

[11.1 VA Standards 23](#_Toc517560740)

[11.2 OAuth 2.0 and OpenID Connect Security 23](#_Toc517560741)

[11.3 API Security Guidelines and Best Practices 23](#_Toc517560742)

[12. Re-Usability and Shared Components 24](#_Toc517560743)

[12.1 What Constitutes Reusable Artifacts 24](#_Toc517560744)

[12.2 How to Create Shared Components 24](#_Toc517560745)

[12.3 Reuse Guidelines 24](#_Toc517560746)

[12.4 Deploying and Integrating Shared Components into an API 24](#_Toc517560747)

[13. API Documentation Blueprint 24](#_Toc517560748)

[13.1 Core API Design Document 24](#_Toc517560749)

[14. References 25](#_Toc517560750)

[15. Acronym 25](#_Toc517560751)

Table of Figures

Table of Tables

[Table 1: Reference Table 8](#_Toc515341204)

[Table 1: Acronym Table 8](#_Toc515341205)

# 1. Introduction

## 1.1 Purpose and Use

## 1.2 VA API Vision

## 1.3 Vendor Agnostic

## 1.4 MuleSoft Applicability

# 2. Standards

## 2.1 Development Lifecycle

There are multiple API development methodologies used by various vendor platforms and development teams. Traditional methodology applied to any general software development is to plan, design, develop, test, and deploy a software component or platform, following a basic waterfall approach to all phases of the development lifecycle. However, there are many flaws with this traditional lifecycle approach to development, especially when focusing on microservice and API development. The key flaws identified with the traditional, waterfall, development lifecycle, are:

* Poor fit with the new Agile project methodologies. Waterfall project methodologies are inflexible to requirements and design changes and are typically geared towards large, monolithic projects, rather than towards microservices based architectures.
* API contracts are not well defined up front. This would cause delays for microservice consumers to start the integration with your microservice or API, until after the Development phase of the project is completed.
* Testing is performed only at later stages of the development lifecycle, post design and development stages.

As a result of our analysis of multiple development lifecycle methodologies, the recommendation is to follow these key guidelines in all of the API development:

* Design First instead of Build First
* Follow Test Driven Development
* Create API Contracts and Mocks Early
* Iterate Frequently and Often
* Engage Your API Consumers

### 2.1.1 Design First instead of Build First

There are two well-established approaches when it comes to API development:

1. **Design First**: Create a well-defined API contract, following Open API or RESTful API Modelling standards, such as Swagger or RAML, from which the code is then developed and tested. Mulesoft platform has been designed specifically with the Design First approach in mind.
2. **Code First**: Based on the business plan, API is directly coded, from which the Swagger or RAML API contract is then generated. Certain middleware platforms, such as Boomi, are built around this approach.

The biggest reason to go with the Design First approach is when the API audience are external consumers and partner platforms. In such a case, the API is a key distribution channel that your API consumers can use to consume your services provided, and a solid design plays a key role in your service adoption. The API contract is the cornerstone of the API design, and, as such, can act as the central draft the keeps all of the developers aligned on what your API’s objectives are, and how the API resources are exposed. Identifying bugs and issues with the API architecture and design becomes easier once a formal API contract is documented in a form of Swagger or RAML. Furthermore, mocking an API response becomes as straight-forward as taking the API contract and adding sample data values for each response object and attribute returned back in the API operation responses.

Code First approach becomes important when speedy code delivery is of utmost importance and the team has very strict and fast-paced timelines to deliver the software component or service.

When using Mulesoft Anypoint Platform, Design First approach is the best practice approach to all API development. A RAML contract is first created within the Design Center or using Anypoint Studio, published into Anypoint Platform and advertised to API consumers via Exchange. The consumers at this point can review the RAML contract and use the API Console to test out the sample inputs and outputs into the API, using Mock API capabilities.

### 2.1.2 Follow Test Driven Development

Test Driven Development (TDD) is an API development process that relies on the iterative short development lifecycle:

1. Requirements are gathered
2. Requirements are turned into specific test cases
3. API is designed, mocked, developed and tested to pass the test cases defined above

This approach to API development is targeted primarily for small, component-based APIs that fit easily into microservice architecture. Microservice APIs are typically small functional components that are independent of the rest of the API platform

We will follow the Test Driven Development approach for all of the APIs developed, as this approach has strong benefits over some of the traditional, Design-Develop-Test, approaches:

* Direct traceability from requirements to test cases to development components and service operations
* Strong focus on short iterative development lifecycle, with a strong fit for microservice development
* Testing is now an integrated function of all API development, as opposed to being a function of a central QA team isolated from the development lifecycle
* Testing frameworks such as MUnit, JUnit, Mocha, Jasmine, and Cucumber, have a strong fit into the overall microservices architecture

### 2.1.3 Create API Contracts and Mocks Early

Based on the guidelines mentioned above, to follow Design First, Test Driven Development, creating API contracts and mocks is an essential first step after the API requirements are delivered to your team to create an API. Either an Open API Standards contract such as Swagger 1.x or 2.x, or a RESTful API Modelling Language (RAML) contract are required to be defined and delivered to the development team and to the identified target API consumers.

Mocking API requests and responses is an important next step, after the API contract is designed. In the Mulesoft Anypoint Platform, this is achieved either automatically using an API Console inside the Design Center, or manually, by developing and deploying a mock API based on the contract defined. In other API platforms, it can be also an automated function using SwaggerUI, as an example, or a manually developed mock API solution. For manually developed mock APIs, the developer can choose to hard-code a JSON or SOAP-based response for a given API operation, or leverage a mock framework such as Mockito, for example.

### 2.1.4 Iterate Frequently and Often

API development should be an iterative process, with initial requirements collected, test cases defined, an API contract designed, and mock responses created and published as the initial version of the API. Following this mock API deployment, an actual API should then be developed by replacing mock operation responses with “live” service operations. Each API development iteration should be clearly labelled and versioned in the source code repository and the API Management console.

Mulesoft API Manager provides ability to tag APIs with versions, which should be used during each successive API iteration deployment.

Furthermore, many packaging and build frameworks such as Maven, have clearly identified version tags for each new API iteration. In Maven, pom.xml should be modified to iterate new API version as a Maven artifact version Id.

### 2.1.5 Engage Your API Consumers

Engage your API consumer early, as soon as the requirements are converted into test cases and the API contract is defined in a form of a Swagger or RAML contract. At that point, the target API consumer can review the contract. Next, develop the mock service operations and deploy as the early API iteration, for the API consumers to start integrating with your API mock service. Engaging the target service consumers early avoids requirements miscommunication, unblocks the consumers to start their integration development to consume your API, and allows your team to focus on development and testing, rather than spending time clarifying requirements and changing developed code based on missing information.

### 2.1.6 Development Lifecycle Stages

Following the above API development guidelines, below are the key stages of the API development. As an API developer, follow the above guidelines and make sure to complete all stages in the below lifecycle.



1. **Design Phase** includes the following:
   * Analyze existing API requirements created by business stakeholders and your target API consumers.
   * Create an overall sequence flow diagram for the API service and its operations, covering all service integration points and interactions with other services, as well as parallel/sequential/rollback requirements.
   * Identify and document key use cases to be supported by this API.

**Deliverables** produced by each developer in this phase:

1. Sequence Flow Diagram.
2. RAML or Swagger API Specification for the API.
3. **Mock Phase** includes creating Mock service to provide sample request and response in the service endpoint. The service at this point should be deployed in its initial iteration with mock inputs and outputs, for target users to be able to consume that service to integrate with their end user platforms.

**Deliverables** produced by each developer in this phase:

1. Mock service operations returning sample output for successful service requests, following the RAML or Swagger API contract defined in the Design phase.
2. Mock service operations returning sample error output for negative service requests.
3. Deployment of the mock service.
4. **Test Phase** follows the test-driven development approach and requires the developers to produce test cases and steps (required for all services developed). Initial unit and functional tests should also be developed after the API initial iteration has been developed. The tests have to cover use cases identified in the Design phase.

**Deliverables** produced by each developer in this phase:

1. Test Cases and Steps as part of the overall Test Plan. Each Test Case should have traceability to the requirements.
2. Unit tests covering mock API operations (using MUnit, Mocha/Chai, JUnit, etc).
3. Functional tests to test how the APIs are to be consumed (using SoapUI, Postman, or automated Gerkin/Cucumber functional test framework).
4. **Build Phase** requires the full development of the API, replacing mock operations with “live” ones, with the complete code review and baseline check-in into the Master branch of the source code repository. At the end of this phase, the “live” API is ready for deployment into non-production environment, with the live operations (not mocks).

**Deliverables** produced by each developer in this phase:

1. Development of the source code.
2. Peer Code Review by the development lead/designated peer.
3. Deployment and validation into non-production environment of the live microservice (no longer mock).
4. **Publish Phase** requires publishing the API manually and via the configured CI/CD pipeline (e.g., Cloudbees, Jenkins, etc) to automate deploy and publish process of the API. In this phase, all required dependencies and components for this API have to also be published into the internal dependency management system such as Artifactory.

**Deliverables** produced by each developer in this phase:

1. Externalizing all test data and configurations to support multi-environment deployments.
2. Deployment and validation into non-production environments manually and via the CI/CD pipeline.
3. Execution of all the functional tests in each of the non-production environments (via SoapUI, Postman and/or automated functional test suite such as Cucumber).
4. **Operate and Engage Phases** require target API consumers and representatives of the business teams to validate the published services in the QA and UAT environments to provide feedback and identify defects.

**Deliverables** produced by each developer in this phase:

1. Resolve defects identified by others.
2. Redeploy the APIs with fixed defects across environments.

2.2 Industry Standards

### 2.2.1 FHIR

#### 2.2.1.1 Argonaut

# 3. API Layers

## 3.1 Loose Coupling

## 3.2 Policy Impacts

## 3.3 API Management Impacts

## 3.4 Layer Responsibilities

### 3.4.1 Experience

### 3.4.2 Process

### 3.4.3 System

#### 3.4.3.1 Native System API

### 3.4.4 Decision Tree

# 4. 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

## 4.1 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.

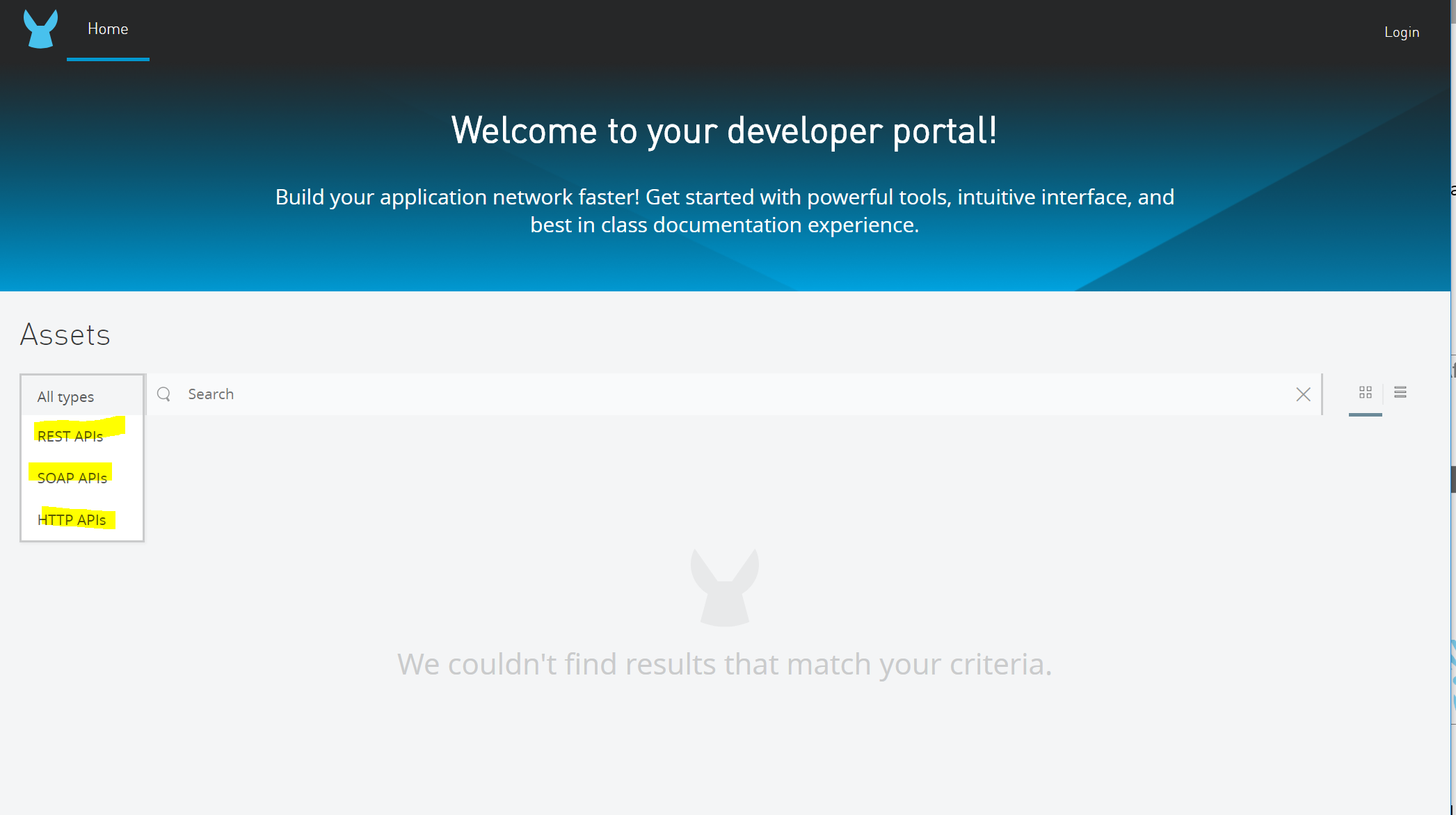
Example

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.

## 4.2 API Contract Standards

### 4.2.1 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.

### 4.2.2 API Specifications

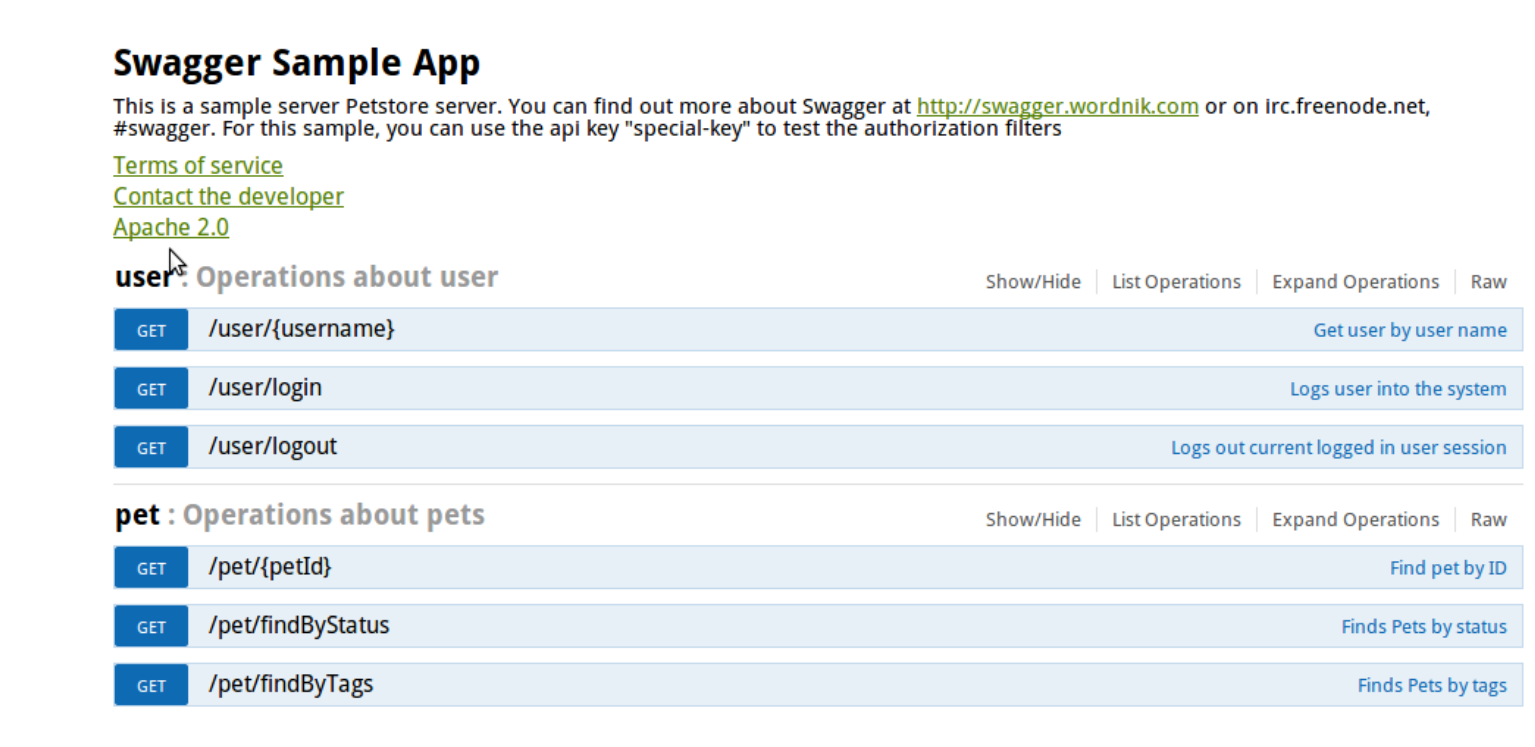
When developing an API Contract, follow these key guidelines to make sure to include all pertinent service definition information is included in the contract:

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

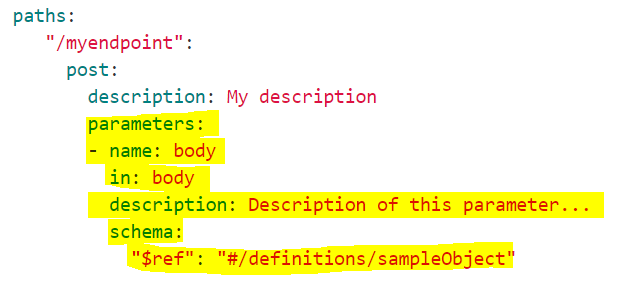
An example of a service operation is provided below for a RESTful operation:



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:



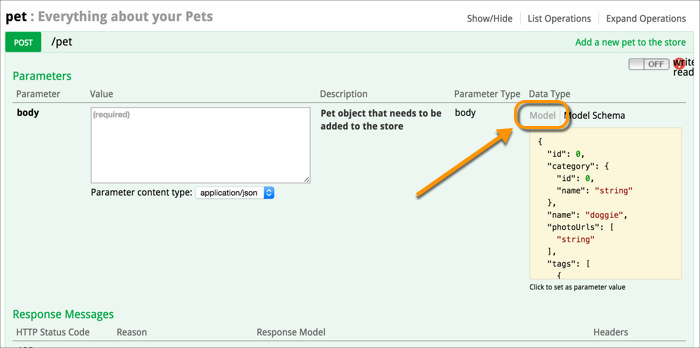
Response Requirements

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)

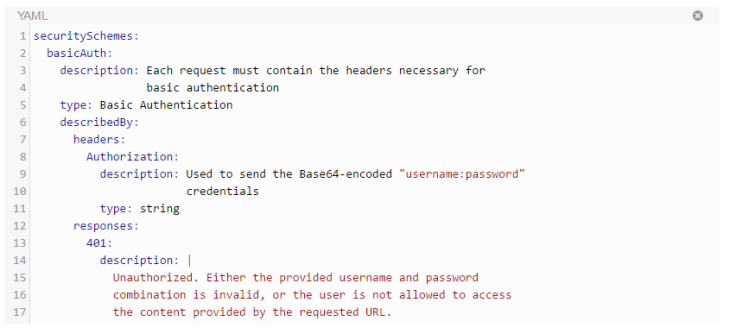
Example of a response model for a Swagger sample pet store is provided below:



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.

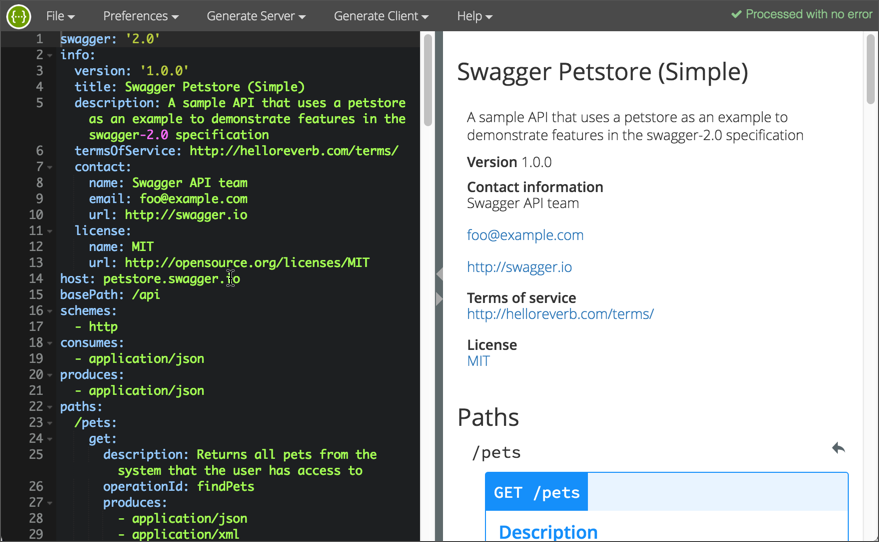
An example below shows the definition of a Basic Auth security scheme for a given API.



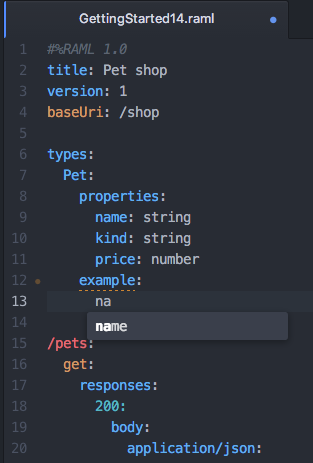
### 4.2.3 Swagger and RAML

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:



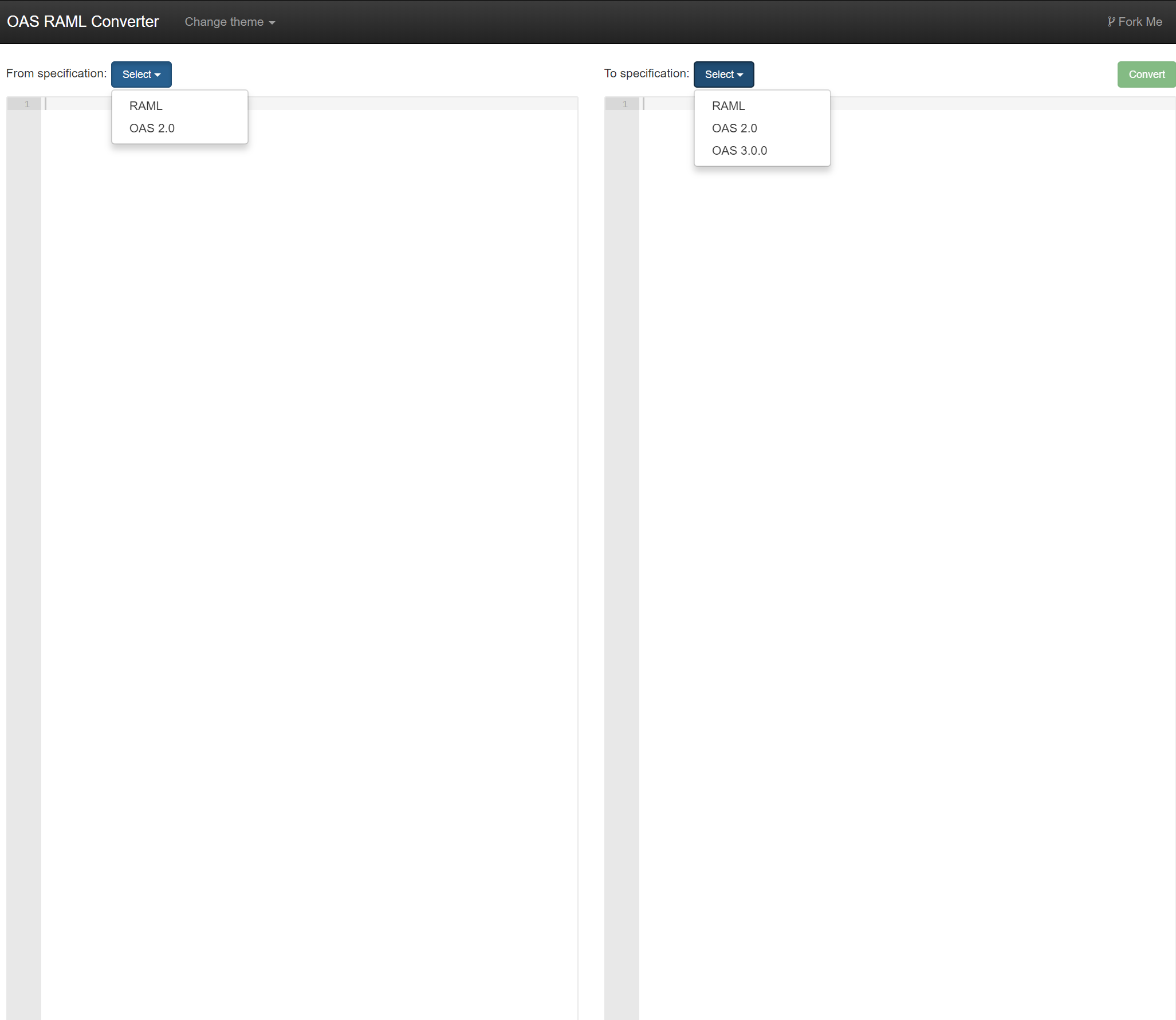
Both Swagger and RAML are very similar, especially if using YAML to define both. There are some slight differences between the two contract definition languages but they are very minor.

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.

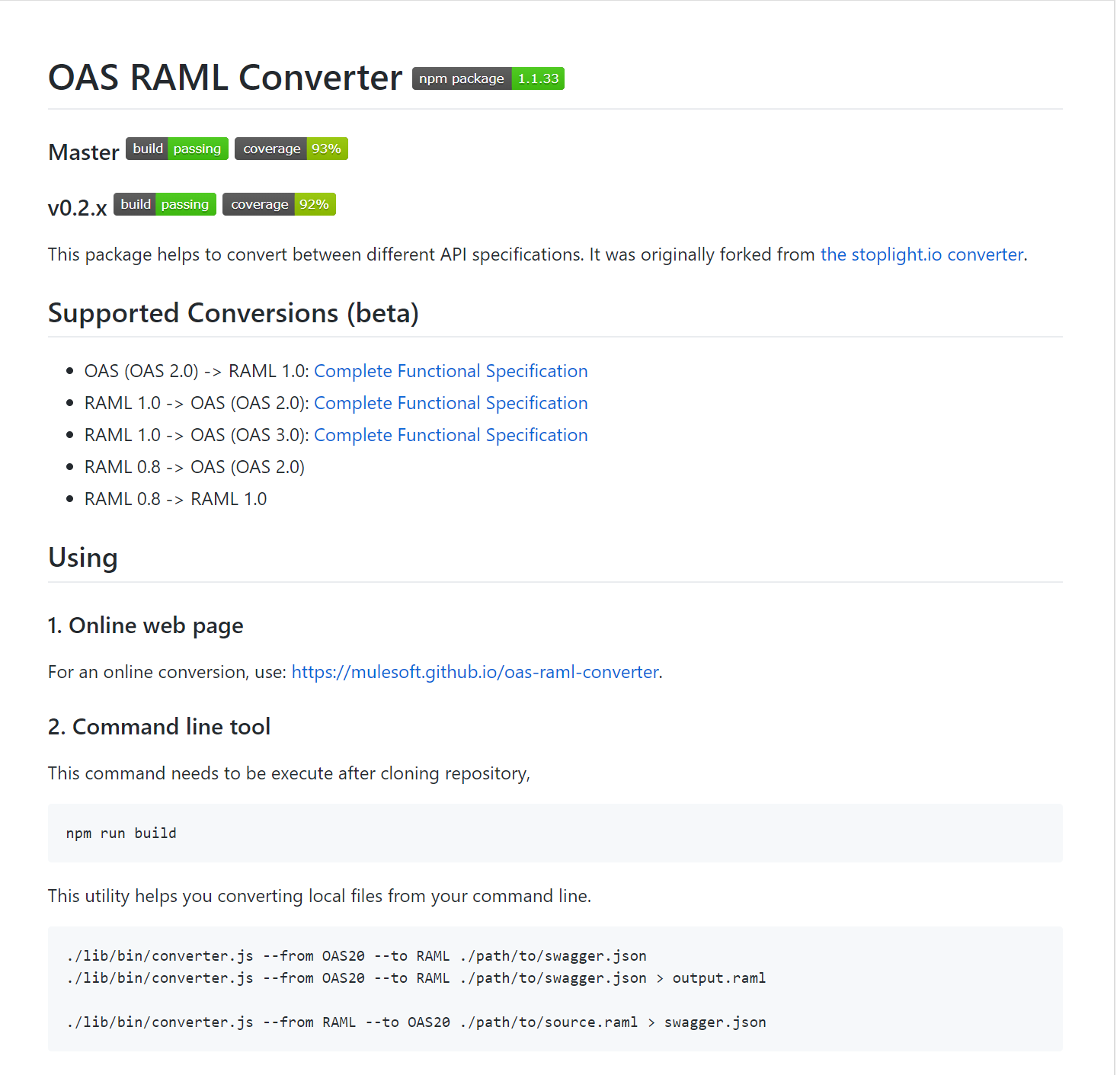
#### 4.2.3.1 Conversions

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:

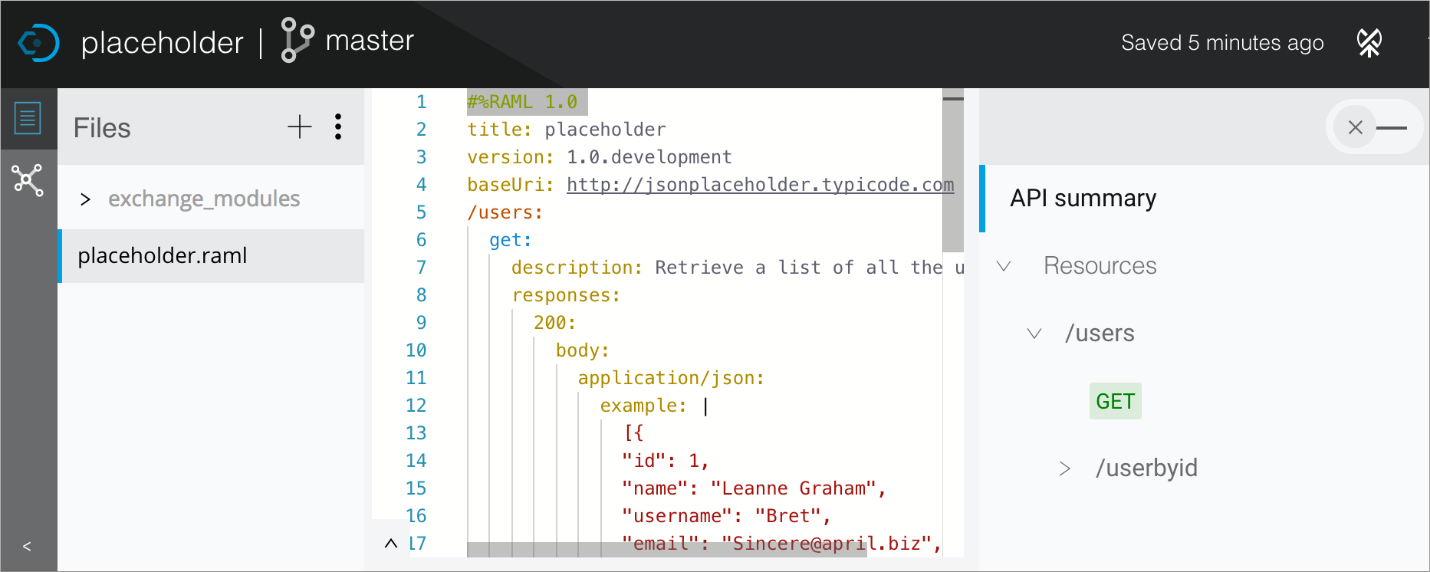


Here is a screenshot of the GitHub web page for this Mulesoft converter project:



## 4.3 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).

# 5. Naming Conventions

## 5.1 API Naming Standards

### 5.1.1 Alignment with FHIR and Argonaut

## 5.1.2 Alignment with Version Control

## 5.2 Environment Properties

## 5.3 Configuration Parameters

## 5.4 API REST and SOAP Standards

### 5.4.1 RESTful URI Naming Standards

### 5.4.2 SOAP Message XML Standards

### 5.4.3 JSON Schema Naming Standards

## 5.4 Objects and Methods

### 5.4.1 Canonical Naming

## 5.5 Middleware Artifacts (flows, sub-flows, transformations, variables)

# 6. Configuration Management

## 6.1 API Configuration Management

### 6.1.1 Version Numbers

### 6.1.2 GitHub Integration

#### 6.1.3 Branching and Merging

## 6.2 Environment Configuration

# 7. Unit Testing

## 7.1 Unit Testing Approach

## 7.2 Agnostic Unit Testing

## 7.3 MuleSoft MUnit

### 7.3.1 Mocking Features

## 7.4 Integration with Continuous Integration Continuous Deployment

# 8. Developer Setup

## 8.1 MuleSoft Developer Setup

### 8.1.1 Maven

### 8.1.2 GitHub Plugin

### 8.1.3 Munit Setup

# 9. Exception Handling

## 9.1 Exception Handling Approach

### 9.1.1 Transactions vs. non-Transactions

## 9.2 Global Exceptions

## 9.3 API-Level Exceptions

## 9.4 Middleware-Level Exceptions

## 9.5 Message Process Failures and Retries

## 9.5.1 Message Retries

## 9.5.2 Handling Message Failures

## 9.6 Exception Logging

## 9.7 Notification and Alerting

## 9.8 Integration with Queuing

# 10. Baseline Logging

## 10.1 Logging Framework Used within Each API

## 10.2 Logged API Messages

## 10.3 API Audit-Level Logging

## 10.4 Integration with Third-Party Log Aggregators

# 11. Baseline API Security

## 11.1 VA Standards

## 11.2 OAuth 2.0 and OpenID Connect Security

## 11.3 API Security Guidelines and Best Practices

# 12. Re-Usability and Shared Components

## 12.1 What Constitutes Reusable Artifacts

## 12.2 How to Create Shared Components

## 12.3 Reuse Guidelines

## 12.4 Deploying and Integrating Shared Components into an API

# 13. API Documentation Blueprint

## 13.1 Core API Design Document

# 14. References

Table : Reference Table

|  |  |  |  |
| --- | --- | --- | --- |
| Ref. ID | Reference Name | Reference | Definition |
| R1 | FHIR Standard | http://www.fhir.org/ |  |
| R2 | HTTPS Standard | <https://https.cio.gov/everything/> | All web traffic shall be HTTPS and none shall be HTTP |
|  |  |  |  |
|  |  |  |  |

# 15. Acronym

Table 2: Acronym Table

|  |  |  |
| --- | --- | --- |
| Acronym | Elaboration | Definition |
| API | Application Programming Interface | A computer function that performs a logical operation on rules or data. |
| SDK | Software Development Kit | A collection of software modules and assets that allows a consumer to apply the modules for runtime functionality. |
| ROI | Return on Investment | The benefit or payoff received from investing cost into an initiative |
| RAML | RESTful API Modeling Language | A language and approach to building API soft contracts the describe and specify API design details. |
| JSON | Javascript Object Notation |  |
| SOAP |  |  |
| XML | Xtensible Markup Language |  |
| FHIR |  |  |
|  |  |  |