

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 12, 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 |
|  |  |  |  |
|  |  |  |  |

[1. Introduction 6](#_Toc516653923)

[1.1 Purpose and Use 6](#_Toc516653924)

[1.2 VA API Vision 6](#_Toc516653925)

[1.3 Vendor Agnostic 6](#_Toc516653926)

[1.4 MuleSoft Applicability 6](#_Toc516653927)

[2. Standards 7](#_Toc516653928)

[2.1 Development Lifecycle 7](#_Toc516653929)

[2.1.1 Design First instead of Build First 7](#_Toc516653930)

[2.1.2 Follow Test Driven Development 8](#_Toc516653931)

[2.1.3 Create API Contracts and Mocks Early 8](#_Toc516653932)

[2.1.4 Iterate Frequently and Often 8](#_Toc516653933)

[2.1.5 Engage Your API Consumers 9](#_Toc516653934)

[2.1.6 Development Lifecycle Stages 9](#_Toc516653935)

[2.2.1 FHIR 11](#_Toc516653936)

[2.2.1.1 Argonaut 11](#_Toc516653937)

[3. API Layers 12](#_Toc516653938)

[3.1 Loose Coupling 12](#_Toc516653939)

[3.2 Policy Impacts 12](#_Toc516653940)

[3.3 API Management Impacts 12](#_Toc516653941)

[3.4 Layer Responsibilities 12](#_Toc516653942)

[3.4.1 Experience 12](#_Toc516653943)

[3.4.2 Process 12](#_Toc516653944)

[3.4.3 System 12](#_Toc516653945)

[3.4.3.1 Native System API 12](#_Toc516653946)

[3.4.4 Decision Tree 12](#_Toc516653947)

[4. API Contracts 12](#_Toc516653948)

[4.1 Design First Methodology 12](#_Toc516653949)

[4.2 API Contract Standards 12](#_Toc516653950)

[4.2.1. API Specifications 12](#_Toc516653951)

[4.2.2 Swagger and RAML 12](#_Toc516653952)

[4.2.2.1 Conversions 12](#_Toc516653953)

[4.3 MuleSoft Designer Studio 12](#_Toc516653954)

[5. Naming Conventions 12](#_Toc516653955)

[5.1 API Naming Standards 12](#_Toc516653956)

[5.1.1 Alignment with FHIR and Argonaut 12](#_Toc516653957)

[5.1.2 Alignment with Version Control 12](#_Toc516653958)

[5.2 Environment Properties 12](#_Toc516653959)

[5.3 Configuration Parameters 12](#_Toc516653960)

[5.4 API REST and SOAP Standards 12](#_Toc516653961)

[5.4.1 RESTful URI Naming Standards 12](#_Toc516653962)

[5.4.2 SOAP Message XML Standards 12](#_Toc516653963)

[5.4.3 JSON Schema Naming Standards 12](#_Toc516653964)

[5.4 Objects and Methods 13](#_Toc516653965)

[5.4.1 Canonical Naming 13](#_Toc516653966)

[5.5 Middleware Artifacts (flows, sub-flows, transformations, variables) 13](#_Toc516653967)

[6. Configuration Management 13](#_Toc516653968)

[6.1 API Configuration Management 13](#_Toc516653969)

[6.1.1 Version Numbers 13](#_Toc516653970)

[6.1.2 GitHub Integration 13](#_Toc516653971)

[6.1.3 Branching and Merging 13](#_Toc516653972)

[6.2 Environment Configuration 13](#_Toc516653973)

[7. Unit Testing 13](#_Toc516653974)

[7.1 Unit Testing Approach 13](#_Toc516653975)

[7.2 Agnostic Unit Testing 13](#_Toc516653976)

[7.3 MuleSoft MUnit 13](#_Toc516653977)

[7.3.1 Mocking Features 13](#_Toc516653978)

[7.4 Integration with Continuous Integration Continuous Deployment 13](#_Toc516653979)

[8. Developer Setup 13](#_Toc516653980)

[8.1 MuleSoft Developer Setup 13](#_Toc516653981)

[8.1.1 Maven 13](#_Toc516653982)

[8.1.2 GitHub Plugin 13](#_Toc516653983)

[8.1.3 Munit Setup 13](#_Toc516653984)

[9. Exception Handling 14](#_Toc516653985)

[9.1 Exception Handling Approach 14](#_Toc516653986)

[9.1.1 Transactions vs. non-Transactions 14](#_Toc516653987)

[9.2 Global Exceptions 14](#_Toc516653988)

[9.3 API-Level Exceptions 14](#_Toc516653989)

[9.4 Middleware-Level Exceptions 14](#_Toc516653990)

[9.5 Message Process Failures and Retries 14](#_Toc516653991)

[9.5.1 Message Retries 14](#_Toc516653992)

[9.5.2 Handling Message Failures 14](#_Toc516653993)

[9.6 Exception Logging 14](#_Toc516653994)

[9.7 Notification and Alerting 14](#_Toc516653995)

[9.8 Integration with Queuing 14](#_Toc516653996)

[10. Baseline Logging 14](#_Toc516653997)

[10.1 Logging Framework Used within Each API 14](#_Toc516653998)

[10.2 Logged API Messages 14](#_Toc516653999)

[10.3 API Audit-Level Logging 14](#_Toc516654000)

[10.4 Integration with Third-Party Log Aggregators 14](#_Toc516654001)

[11. Baseline API Security 14](#_Toc516654002)

[11.1 VA Standards 14](#_Toc516654003)

[11.2 OAuth 2.0 and OpenID Connect Security 14](#_Toc516654004)

[11.3 API Security Guidelines and Best Practices 14](#_Toc516654005)

[12. Re-Usability and Shared Components 14](#_Toc516654006)

[12.1 What Constitutes Reusable Artifacts 14](#_Toc516654007)

[12.2 How to Create Shared Components 14](#_Toc516654008)

[12.3 Reuse Guidelines 14](#_Toc516654009)

[12.4 Deploying and Integrating Shared Components into an API 14](#_Toc516654010)

[13. API Documentation Blueprint 15](#_Toc516654011)

[13.1 Core API Design Document 15](#_Toc516654012)

[14. References 16](#_Toc516654013)

[15. Acronym 16](#_Toc516654014)

Table of Figures

**No table of figures entries found.**

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

## 4.1 Design First Methodology

## 4.2 API Contract Standards

### 4.2.1. API Specifications

## 4.2.2 Swagger and RAML

#### 4.2.2.1 Conversions

## 4.3 MuleSoft Designer Studio

# 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 1: 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 |  |  |
|  |  |  |