## Standards API Development Lifecycle

There are multiple API development methodologies used by various vendor platforms and development teams. Example methodologies include:

* Waterfall, where every stage of the development is done in a sequence to a previous stage (shown on the below diagram)
* Develop First, where a small team picks up a pilot project, develops a prototype and showcases the prototype to the potential customers before planning to do a full-blown development of all possible Use Cases.
* Design First, where the developers first draft the design of a solution, present the design in a well-documented format (e.g., an API contract) to the target users before a development phase starts

**Example Methodology – Waterfall**



A traditional methodology applied to any 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 hinder development of microservices based architectures which can be delivered iteratively in smaller and quicker releases.
* API contracts, and hence API specifications, 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, and fail to leverage the opportunity to test early, often, and iteratively

As a result, the recommendation is to follow these key guidelines for VA API development:

* Design First instead of Build First
* Follow Test Driven Development
* Create API Contracts and Mockups Early (commonly called “Mocking an API”)
* Iterate Frequently and Often
* Engage Your API Consumers

Each of the following key guidelines mentioned above is elaborated further in the sections below.

### Design First instead of Build First

An API Contract is a common term to describe how API services are brokered among API providers and consumers. It is a standard specification agreement between the provider and consumer. API providers and consumers can enter into an agreement (i.e. the “contract”) for services around a set of resources. Additionally, a provider can publish a codified and machine readable API contract, which a consumer can consume in a self-service way. Examples of an API Contract are a Swagger JSON or YAML file, or an API RAML definition file.

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

1. **Design First**: Create a well-defined API contract first, following Open API or RESTful API Modelling standards, such as Swagger or RAML, from which the code is then developed and tested. The API Contract is distributed to service consumers before the API is developed, so the consumers can begin developing their consumption design and are not waiting for the full API to be ready.



* Benefits of this approach include:
  + Identifying bugs and issues with the API architecture and design becomes easier once a formal API contract is documented in the form of Swagger or RAML.
  + 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 in the API operation responses.
* A primary driver for choosing a Design First approach is when the API consumers are external third-party partner platforms, or web/mobile applications. In such cases, 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 can act as the standard or central alignment that keeps all the developers in synch with your API’s objectives and how the API resources are exposed.

1. **Code First**: Based on the business plan, the API is directly coded first, from which the Swagger or RAML API contract is then generated and distributed to consumers.

* Code First approach becomes important when the project has very strict and fast-paced timelines to deliver the software component or service and the project plan doesn’t allow for building API contracts and mocks first.
* When choosing which API contract standard to leverage (RAMLvs. Swagger) at first the development team should decide who would be the target consumers or users of that contract and which platform the API is to be deployed within. For example, will the API be deployed in the Mulesoft Anypoint Platform, Apigee, or another platform? In most cases, it is recommended to use Open API standards and design a Swagger contract. A Swagger API contract can be expressed in a form of a JSON or a YAML file. As YAML format is more widely used among technology teams, it is recommended.

**MuleSoft Practices**

In the case where an API is closely tied to Mulesoft platform, RAML is a natural choice over Open API Swagger

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.

### Follow Test Driven Development

Test-Driven Development (TDD) is an API development process that relies on iterative, short development lifecycles immediately followed by testing what was developed in each iteration. The core process of TDD is:

1. Requirements or User Stories are gathered.
2. Requirements are turned into specific test cases.
3. The API is designed, mocked, developed and tested to pass the test cases defined above in one complete iteration.

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

An example of a microservice is a User Authentication (Login/Logout) service that handles all user login and logout features for a given application. An API developer or analyst would first gather requirements related to User Authentication, and express them as functional user stories. For each user story, the API developer or analyst will create corresponding test cases with steps to test and validate each user story. For example, a user story will cover “As a user, I should be able to login into an application by providing my username and password.” The corresponding test case will cover user login. Test steps for validating the user login will include testing a user entering username, masked password, submitting the form, and triggering a microservice operation to authenticate the user into the backend LDAP via binding credentials to the credentials store. The test steps will cover all User Interface (UI) steps to login and submit the login form, as well as the test steps to invoke a microservice operation to login the user. If there is a future need for a headless client without the UI, the test cases should be able to cover validating the microservice login operation executed via a headless client such as SoapUI or Postman.

Finally, after the user stories and test cases and steps are documented, an API developer works on an API contract. In this above scenario, the developer will create an API contract for login operation as a starting point, and later will add a new operation for logout.

VA API Implementation projects should follow the TDD approach for all of the APIs developed, as this approach has strong benefits over some of the traditional, Design-Develop-Test, approaches, such as:

* Direct traceability from requirements to test cases to development components and service operations.
* Strong focus on short, iterative development lifecycles, with a strong fit for microservice development.
* Testing is now an integrated function of all API development, as opposed to 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.

In the above TDD development process, an API developer is responsible for documenting the key user stories and corresponding test cases and steps. This upfront test planning will help projects to accelerate their timelines, especially those with dedicated centralized QA teams managing the test plan with test cases and steps. With TDD there is an elevated responsibility put on each API developer to maintain a list of the API user stories and test cases with steps for each API the developer is responsible for.

### Create API Contracts and Mocks Early

Based on the guidelines mentioned above, to follow TDD, 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 is required to be defined and delivered to the development team and to the identified target API consumers.

Mocks of each API are required to be either created manually or, like with some API management platforms, auto-generated based on the created API contract. Since creating “live” API operations requires comprehensive development effort, an API developer would first create or auto-generate a mock API. That mock API generation is trivial compared to creating an actual “live” API operation. Target API consumers would be able to immediately start building their clients to integrate with the backend API, once they are provided with the mock API endpoints.

Mocking API requests and responses is an important next step, after the API contract is designed. In many API management platforms, this is achieved either automatically using an API Console, or manually, by developing and deploying a mock API based on the defined contract. In other API platforms, it can be also an automated function using SwaggerUI toolset, for 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.

### Iterate Frequently and Often

API development should be an iterative process, complete with initial requirements and user stories 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. For more details on API versioning, please see section ”API Version Control”

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 a new API version as a Maven artifact version Id.

With proper API major and minor versioning, an API developer can iterate API development frequently and publish updated API contract versions to the target consumers. For example, a new change to an API contract requires a minor version update. To minimize the impact on API consumers, a developer would update the API packaging and contract definition files (e.g. pom.xml and Swagger or RAML contracts) to iterate the version of an API, and then publish the updates to the API Design Center and API Portal. The target API consumers will review the changes and decide whether they would want to stay on an existing API version or upgrade to the latest version, based on the release notes documented by the API developer.

### Publish an API and Operate

Each new iteration of the code and API contract needs to be published to an online API catalog or API portal to be accessible by the target API consumers. The API consumers can subscribe to new notifications of a particular API update and get notified when a new API version is published to the online catalog. A provider API developer should create well-documented release notes for each new API version developed , explaining what changes have been made with this new version and why API consumers should be using this version instead of an older one.

Once an API version is published, it is the responsibility of the API consumer developer to operate and maintain the consumer codebase and managed to the API versions that are actively used by the consumers. API consumers should be made aware of any downtimes or API availability interruptions, as well as any defects discovered with a particular API version.

### Engage Your API Consumers

Engaging the target API service consumers early avoids requirements miscommunication, promotes consumers starting their integration development to consume your API, and allows your team to focus on development and testing.

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.

To engage an API consumer, a developer would publish the API to the API Portal (such as Mulesoft Exchange, Apigee API Portal, or a custom API online catalog). The catalog or portal will contain all API versions, with the most recent version showing on top. The portal will have an ability for an API consumer to review the API contract for each API version and review the release notes per version, including key changes delivered with each API version. Finally, the portal or catalog will provide an ability for an API consumer to quickly test the API by sending a sample request and receiving a sample response for a given API version.

Once an API is published, an API developer should ask the API consumers to provide feedback on a particular API version. Certain online API portal tools, such as SwaggerHub, provide the ability for an API consumer to provide feedback and comments on an API contract.

### Development Lifecycle Stages

Following the above API development guidelines, below are the key stages of the API development. An API developer should follow the above guidelines and complete all stages in the below lifecycle.



1. **Design Phase** includes the following:
   * Analyze existing API requirements and user stories 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, including 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. Key Use Cases to support the requirements and user stories
4. **Mock Phase** includes creating the 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. The mock APIs are only available in Dev and Integration and QA environments, and should never be promoted to Production.

**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 TDD 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 or similar.
3. Functional tests to demonstrate 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 (i.e. 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. Typically, there should be a peer review checklist that each code reviewer follows when reviewing the code of another developer. Part of such checklists should include API contract formatting, user story traceability to API operations, code comments, unit test cases and test coverage, logging and exception handling, including proper use of shared components and reusable artifacts.
3. Deployment and validation into non-production environment of the live microservice (i.e. no longer mock).
4. **Publish Phase** requires publishing the API manually or via the configured CI/CD pipeline (e.g. Cloudbees, Jenkins) to automate the deploy and publish process of the API. In this phase, all required dependencies and components for this API must 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.