Department of Veterans Affairs

API Developer Playbook

API Developer Playbook Blueprint Documentation

Maps to Framework Set #1



Document Version Draft 01.00.01

Revision History

| Date | Version | Description | Author |
| --- | --- | --- | --- |
|  |  |  |  |

Table of Contents

[1. Configuration Management 1](#_Toc519761308)

[1.1. API Configuration Management 1](#_Toc519761309)

[1.1.1. Configuration Management in the Cloud 1](#_Toc519761310)

[1.1.2. Configuration Management Software Tools 1](#_Toc519761311)

[1.1.3. Future of Configuration Management Software 2](#_Toc519761312)

[1.2. Changes to RESTful APIs 2](#_Toc519761313)

[1.3. When to Version 3](#_Toc519761314)

[1.4. Versioning Guidelines 3](#_Toc519761315)

[1.4.1. URI Versioning 3](#_Toc519761316)

[1.4.2. Media Type Versioning 4](#_Toc519761317)

[1.4.2.1. HTTP Request Accept Examples: 4](#_Toc519761318)

[1.4.2.2. HTTP Response Content-Type Examples: 4](#_Toc519761319)

[1.4.3. Summary 5](#_Toc519761320)

[1.5. GitHub 6](#_Toc519761321)

[1.5.1. The GIT Command Line 6](#_Toc519761322)

[1.5.2. GitHub Branching and Merging 6](#_Toc519761323)

[1.5.3. GitHub Branching 6](#_Toc519761324)

[1.5.3.1. GitHub Branching and Related Command Lines 7](#_Toc519761325)

[1.5.4. GitHub Merging 7](#_Toc519761326)

[1.5.4.1. GitHub Merging and Related Command Lines 8](#_Toc519761327)

[1.5.4.2. Resolving GitHub Merging Conflicts 8](#_Toc519761328)

[1.5.5. GitHub Integration 9](#_Toc519761329)

[1.5.5.1. Testing 9](#_Toc519761330)

[1.5.5.2. Code Quality 9](#_Toc519761331)

[1.5.5.3. Code Review 9](#_Toc519761332)

[1.5.5.4. Continuous Integration 9](#_Toc519761333)

[1.5.5.5. Dependency Management 9](#_Toc519761334)

[1.5.5.6. Ddeployment 10](#_Toc519761335)

Table of Tables

**Table of Figures**

# Configuration Management

**The section of the document is dedicated to the generic topic of Configuration Management, albeit with a more detailed discussion of Configuration Management topics. DevOps, which is a set of tools, processes and procedures that modern software development teams put in place to ensure that source code is deployed in a stable and repeatable way, will not be discussed in this section.**

**So, for the purposed of this section, the definition of API Configuration Management is the set of practices aiming to manage the runtime state of the software applications.**

## API Configuration Management

Applications typically consists of one or more databases, web servers, application servers, reverse proxies, load balancers, and other moving parts that need work together at runtime to implement a working system.

Environment configuration management is the process of describing the ways in which all these inputs, typically referred to artifacts, interact and how they leverage the underlying infrastructure. Specifically, it addresses the installation, configuration and execution of configuration items on servers.

Traditional configuration management required the writing of items such as scripts, hand-maintain files listing technical endpoints, port numbers, and namespaces etc. The laborious nature of this process resulted in the increased complexity of a system as well as the configuration management process itself.

Increased infrastructure configuration automation evolved because of the needs of processes such as artifact versioning, environment changes and the need for coordination between developers and system engineers.

### Configuration Management in the Cloud

The cloud facilitated the migration of server infrastructure from on-premise to the cloud and while the inherent complexities of running an on-premise infrastructure decreased, new problems arose as well.

Cloud technologies have enabled teams to deploy software to a multitude of servers concurrently to satisfy the demands of modern software usage. Managing many servers requires automation on a different scale with a more systematic approach.

This is where an **API-driven** approach to configuration management can help.

Rather than installing and launching scripts individually on each node, a centralized configuration management server could control all the nodes programmatically and drastically reduce the workload of the team’s system administrators.

### Configuration Management Software Tools

New configuration management software tools have gradually been introduced in the last decade to address the growing needs of automated configuration management. Following is a high-level overview of existing configuration management tools.

* **Ansible**

Ansible, also open source and built in Python. It was created in reaction to the relative complexity of Puppet and Chef, and attempts to offer a simpler, more elegant alternative, with a shorter learning curve. Ansible is based on an agentless architecture, meaning that no agent is required to run on each infrastructure node, which leads to less complexity and less load on the network. Ansible modules, referred to as units of work, can be written with a variety of scripting languages like Python, Perl or Ruby. Ansible lets users define playbooks in YAML for often used system descriptions.

* **Puppet**

It is built in Ruby but offers its own JSON-like declarative language to create ‘manifests’, the modules in which configuration items are described, using high-level concepts like users, services and packages. Puppet is “Model driven”, which means that not much programming is usually required. This makes Puppet more desirable for system engineers with little experience in programming.

* **Chef**

Is generally developer-friendly and very popular with teams already using Ruby. It lets developers write ‘cookbooks’ in Ruby and stores the resulting instructions in PostgreSQL.

* **SaltStack**

It is built in Python and open sourced in 2011. Salt focuses on low-latency architecture and fault tolerance. It features a decentralized setup, small messaging payloads, no single point of failure, and parallel execution of commands for optimal performance.

### Future of Configuration Management Software

While the configuration management technological landscape has greatly matured, the consensus that things will keep evolving in the future. The current mainstream solutions are often viewed as too complicated, unforgiving and a difficult to maintain. One alternative to the classic configuration management approach is the emerging immutable container infrastructure. Docker is an example of this evolving technology.

Whereas regular configuration management aims to define and manage state at run time, containerized applications require that all the configuration be defined at build time. The resulting portable containers can then be moved from one host to the next without any changes in state. These states can then be saved in Docker images and used later to re-spawn a new instance of the complete environment. Images therefore offer an alternative means of configuration management, arguably superior to runtime configuration management.

## Changes to RESTful APIs

The following is a paraphrased excerpt from an interview with Roy Fielding and his discussions involving RESTful API changes over time.

<https://www.infoq.com/articles/roy-fielding-on-versioning>

Anticipating change is one of the central themes of REST. It makes sense that experienced developers are going to think about the ways that their API might change in the future, and to think that versioning the interface is paving the way for those changes. That led to a never-ending debate about where and how-to version the API.

Reality regarding services APIs is that contracts between services and service consumers are bound to change. API Contract versioning facilitates the rolling out of new service features that involve contract changes and at the same time provide backward compatibility for service consumers that are still using prior API versions. Plan for contract versioning from the very start of the design effort, even though the current thinking is that is versioning will not be needed, because eventually it most likely will.

## When to Version

APIs only need to be up-versioned when a breaking change is made. Breaking changes include:

* A change in the format of the response data for one or more calls
* A change in the response type (i.e. changing an integer to a float)
* Removing any part of the API.

Breaking changes should always result in a change to the major version number for an API or content response type.

Non-breaking changes, such as adding new endpoints or new response parameters, do not require a change to the major version number. However, it can be helpful to track the minor versions of APIs when changes are made to support customers who may be receiving cached versions of data or may be experiencing other API issues.

## Versioning Guidelines

There is substantial documentation on the web regarding RESTful versioning. This document will detail **two** options that for the most part have been overwhelmingly referenced in these various discussions, more so then what seems to be less viable options:

* URI Versioning – version the URI space using version indicators
* Media Type Versioning – version the Representation of the Resource

Pros and cons of each option will be noted.

### URI Versioning

The first recommendation addresses how to handle versioning if the development community views the URL as the most important facet which make the URL the center of the versioning efforts. Developers who are familiar with versioning with WS-\* Web services tend to be more comfortable with changing the URL when versions change.

Like most other problems with REST, a common flawed approach to API versioning stems from misunderstanding of the URI. This popular way simply adds a version number as a segment in the resource identifiers

Adding, removing, reordering, and renaming fields constitutes a breaking change for a client which philosophically is a departure from REST. Creating a new version in this scenario, changes the URL structure. The resource object version for 1.0 and 2.0 looks like this:

* http://www. mycompany.com/myapp /resource1.0
* http://www. mycompany.com/myapp /resource2.0

In the above examples it is most likely the behavior of the resource that changes and not the actual resource. In other words, it is the contract that changed. The two identifiers are used to access the same resource. This is the most common argument why this approach is somewhat problematic but it is simple and will never break any cache constraints. But there is one scenario when the version in the URL somewhat makes sense. That is when the resource changes and it each individual version must be accessible

By convention, the default value for format is json. Examples:

* Request the resource as XML: http://www. mycompany.com/app/1.0/resource ?format=xml
* Request the resource as JSON: http://www. mycompany.com/app/1.0/resource ?format=json

Some developers tend to prefer versioning by URL which allows them to determine which version of the service is in use at a glance. From a purest standpoint the URI in REST should represent the resource structure only. It could be argued that a version is not a resource but an attribute of the resource.

### Media Type Versioning

The REST API would make use of custom vendor Multipurpose Internet Mail Extensions or MIME media types instead of generic media types such as application/json. It is these media types that are going to be versioned instead of the URIs. The media type versioning philosophy makes use of the MIME sub type of vnd which is vendor-specific. The Internet Assigned Numbers Authority (IANA) is the official authority for the standardization and publication of these vendor classifications.

If the API is to be used internally only, the vnd MINE type should not be an issue. If an external presence is desired, thought should be given with registering it with the IANA This is a standard way for sharing out-of-band information with potential clients.

#### HTTP Request Accept Examples:

* **Accept**: application/vnd. mycompany.myapp.resource-v1+json

or using a v=1 qualifier:

* **Accept**: application/vnd. mycompany.myapp.resource+json;v=1

#### HTTP Response Content-Type Examples:

* **Content-Type**: application/vnd. mycompany.v1+json

What is important to understand here is that the client makes no assumptions about the structure of the response beyond what is defined in the media type. Therefore, generic media types are not ideal as they do not provide enough semantic information and force the client to use additional hints to process the actual representation of the resource.

The Accept header in the request tells the server the content types that the client is expecting in the response body.

Content negotiation preserves clean set of URLs but there is still the task of having to deal with the complexity of serving different versions of content somewhere. This burden tends to be moved up the stack to the API controller infrastructure which become responsible for figuring out which version of a resource to send. The result tends to be a more complex API as clients must know which headers to specify before requesting a resource.

### Summary

If the responsible development community is more RESTful architecturally minded and aware, REST API versioning should be accommodated by using version data representations in the MIME types in the HTTP Application Accepts. If development community views the URL as the most important facet then changing the URL there when versions change.

Both mechanisms are valid. You need to know your consumer to know which path to follow. In general, working in an enterprises minded environment tends to point developers towards REST versioning using the MIME philosophy.

## GitHub

Git and GitHub are not the same thing. Git by itself, is a distributed revision control and software configuration management (SCM) system. GitHub is a web-based software as a service for software development projects that uses Git

GitHub is a publicly available, free service which requires all code, unless there is a paid account, be available in the public domain. GitHub currently hosts the source code for numerous open source projects. An in-depth discussion on the pros and cons of Git compared to centralized source code control systems will not be a part of this discussion as Git and GitHub is the VA’s recommended SCM alternative.

### The GIT Command Line

There are a lot of different ways to use Git. There are the original command-line tools, and there are many graphical user interfaces of varying capabilities. The command line is the only place where all Git commands can be executed. Most of the GUIs (e.g. Bit Bucket) implement only a subset of Git functionality for simplicity and the choice of a graphical client is a matter of personal taste. For purposes of clarity only GIT command lines features will be discussed in this section of the document.

### GitHub Branching and Merging

It can be argued that Git changed the way developers think of merging and branching. From the classic CVS/Subversion world where merging/branching has always been considered a bit daunting and something you only do when it was necessary.

With Git, merging and branching actions are simple, and they are considered one of the core parts of a daily development workflow, for example, in the world of CVS/Subversion, branching and merging is treated as an advanced concept, while in Git they are considered basic concepts.

This section of the document will cover GitHub Branching and Merging commands at a somewhat low level using Git command lines as part of the discussion. A detailed philosophical discussion of Branching and Merging is better left to the configuration management strategies of individual development teams.

### GitHub Branching

Software development generally occurs using multiple branches in parallel. For example, a software artifact may be under development/modification by multiple developers on different branches.

Typically, there are multiple types of branches utilizing various naming conventions:

* **Master Branch -**  Publicly released production-ready code.
* **Release Branch -** Branches that are being prepared for deployment. but still internal to the deployment effort.
* **Project Branches** - Branches in development for pre-release projects, such as partner deployments. Branched that are at a lower level then a Release Branch artifact.
* **Development Branch**: Branches that contain in-progress artifacts that are being worked by development staff.

These branches are arranged in a liner sequence where the Development Branch migrates to the Project Branch and so on upwards. This linear sequence simplifies the *merge/release strategy* (i.e. Develop to Project to Release to Master)

The *merge/release strategy* alluded to above means that when code is ready to be moved up the branch food chain a "forwards merge" along the direction of this chain is performed. Conversely a "backwards merges" are performed on a regular basis to bring changes (e.g. hotfixes) from the higher level branched back down to the lower level branches. Backwards merges can happen any time, but should be regularly scheduled.

Choosing where to branch off from and target a pull request depends on an assessment of three factors: how urgent the fix/feature is, how risky (in terms of introducing other bugs) the required changes are, and how long it will take to develop. A good heuristic is to see how urgent the code that is being committed versus how risky or potentially breaking it is:

#### GitHub Branching and Related Command Lines

Simply stated a branch is required when a feature hosted in the high-level branch must be modified and/or added. A branch which will host the development of the new feature, must be created from an upper level branch, which is typically the Master. This section will focus solely on the branch creation process only. Subsequent process flow commands such as commits and check-ins are not part of this discussion. The following are typical steps used in creating and removing a branch:

1. Creating a new branch:

Execute the Git ***branch*** <new branch name**>** followed by Git ***checkout*** <new branch name***>.*** These two commands can be combined into one by using the checkout command with the -b switch - git ***checkout -b*** <new branch name**>.**

1. Removing a branch

Branches are typical removed after their contents have been merged to a higher-level branch and is good practice. Execute the Git branch command with the -d switch - Git ***branch -d*** <branch name**>.**

### GitHub Merging

GIT merging facilitates consolidates the migration of a forked branch or branches back to a common branch. The Git ***merge*** command accomplishes this task.

The current branch will be updated to reflect the merge, but the target branch will be completely unaffected. GIT ***merge*** is often used in conjunction with Git ***checkout*** for selecting the current branch and Git ***branch -d*** for deleting the merged branch target branch

Typically, Git ***merge*** is used to combine two branches. The following examples will focus on this branch merging pattern. Git ***merge*** takes two commit pointers, usually the branch tips which are the last commit or most recent commit on a branch, and will find a common base commit between them. Once Git finds a common base commit it will create a new "merge commit" that combines the changes of each queued merge commit sequence

Here is the scenario: a branch containing a new feature that is based off the master branch. This feature branch needs to be merged back into the Master Branch. Invoking this command will merge the specified branch feature into the current branch, which in this case is Master. Git will determine the merge algorithm automatically.

#### GitHub Merging and Related Command Lines

Before performing a merge, there are several preparation steps needed to ensure the merge is performed correctly.

1. Confirm the receiving branch of the merge:

Execute Git ***status*** to ensure that HEAD (think of the HEAD as the "current branch") is pointing to the correct merge-receiving branch. If needed, execute Git ***checkout*** <receiving branch> to switch to the receiving branch. In this case execute git ***checkout*** ***master***

1. Fetch latest remote commits:

Insure that the receiving branch and the merging branch are up-to-date with the latest remote changes. Pull the latest remote commits by executing Git ***fetch***. Execute Git ***pull*** once the fetch is completed to ensure the master branch has the latest updates

1. Perform the merge:

Once steps 1 and 2 have been taken place, a merge can be initiated by executing Git ***merge*** <branch name> where <branch name> is the name of the branch that will be merged into the receiving branch which in this scenario is Master.

#### Resolving GitHub Merging Conflicts

If the two branches both changed the same part of the same file, Git won't be able to determine which version to use. When this a situation occurs, it stops right before the merge commit so the conflict can be resolved manually. When a merge conflict is encountered, running the Git ***status*** command shows which files need to be resolved.

### GitHub Integration

There are numerous products that integrate with the GitHub Platform. The following GitHub Market Place Link list in more many integrating products: <https://github.com/marketplace>.

The following is a small sub set by grouped by product functionality.

#### Testing

* TestQuality

TestQuality's GitHub Test Management extends Github to provide powerful and modern test plan management helping developers and testers create, run, coordinate and monitor software testing tasks.

#### Code Quality

* Sider

Sider supports over 20 analysis tools to empower development teams by increasing code review efficiency and quality. It also offers quick and simple extended functionality with custom tools to tailor-fit your team, enabling them to save time and improve the development process.

#### Code Review

* AccessLint

AccessLint brings automated web accessibility testing into the development workflow. When a pull request is opened, AccessLint reviews the changes and comments with any new accessibility issues, giving you quick, timely, and targeted feedback, before code goes live.

#### Continuous Integration

* Jenkins

With the help of the Git plugin Jenkins can easily pull source code from any Git repository that the Jenkins build node can access. The GitHub plugin extends upon that integration further by providing improved bi-directional integration with GitHub. Allowing the setting up of a Service Hook which will hit the Jenkins instance every time a change is pushed to GitHub.

#### Dependency Management

* Dependabot

Dependabot helps to keep Ruby, JavaScript, Python, PHP, Elixir, Rust, Java and .NET dependencies up to date. Every day, Dependabot checks the dependency files for outdated or insecure requirements and opens individual pull requests for any it finds. Review the PRs, merge them, and get to work on the latest, most secure releases

#### Ddeployment

* Codefresh

Codefresh is designed specifically for teams deploying containers to production. Move fast and deploy with confidence thanks to:

* + Docker Optimized CI
  + Kubernetes Optimized CD
  + Insanely fast build flow optimized for Docker
  + Easy Image Management from an embedded registry
  + Quick Setup and Powerful Pipelines
  + Streamlined deployment to Kubernetes, Swarm, ECS, and more.