# Getting Started with Versioning

In this lesson we will discuss which versioning schema to choose considering the naming conventions already being used and recognized by the tools.

**We'll cover the following**



Versioning is one of those things that can be done in many different ways.

## Choosing a versioning schema #

Almost every team I've worked on came up with their own versioning schema. When starting a new project, we would often spend time debating how we were going to version our releases. And yet, coming up with our versioning schema was usually a waste of time. The goals of versioning are simple; we need a unique identifier of a release as well as an indication of whether a change breaks backward compatibility. Given that others already agreed on the format that fulfills those objectives, the best we can do is to use the convention. Otherwise, we are wasting our time reinventing the wheel without understanding that a few things are likely going to go wrong down the line.

### Expectation of users #

First of all, our users might need to know whether our release is production-ready and whether it breaks compatibility with the previous releases. To be honest, most users don't even want to know that, and they will merely expect our applications to always work. For now, we'll assume that they do care about those things, somebody always does. If the end-users don't, the internal ones do. They will assume that you follow one of the few commonly accepted naming schemes. If you do, your users should be able to deduce the readiness and backward compatibility

as, your users should be able to deduce the readiness and backward compatibility

through a simple glance at the release ID without any explanation of how your versioning works.

### Assumptions by tools

Just as users expect a specific versioning scheme, many tools expect it as well. If we focus on the out-of-the-box experience, tools can implement only a limited number of variations, and their integration expectations cannot be infinite. If we take Jenkins X as an example, we can see that it contains a lot of assumptions. It assumes that every Go project uses `Makefile`, that versioning control system is one of the Git flavors, that **Helm** is used for packaging Kubernetes applications, and so on. Over time, the assumptions are growing. For example, Jenkins X initially assumed that **Docker** was the only way to build container images, and **kaniko** was added to the list later. In some other cases, none of the assumptions will fit your use case, and you will have to extend the solution to suit your needs. That's OK. What is not a good idea is to confuse specific needs with those generated by us, not following a commonly adopted standard for no particular reason. Versioning is often one of those cases.

## Naming convention

By now, you can safely assume that I am in favor of naming conventions. They help us gain a common understanding. When I see a release identifier, I assume that it is using semantic versioning simply because that is the industry standard. If most of the applications use it, I will assume that yours use it as well, unless you have a good reason not to. If that's the case, make sure that it is indeed justified to change a convention that is good enough for most of the others in the industry.

So, *what are the commonly used and widely accepted versioning schemes?* The answer to that question depends on who the audience is.

1. Are we creating a release for humans or machines?
2. If it's the former, are they engineers or end-users?

## Versions of my software

We'll take a quick look at some of the software I'm running. That might give us a bit of insight into how others treat versioning, and it might help us answer a few of

the questions.

- I'm writing this on my MacBook running macOS *Mojave*.
- My Pixel 3 phone is running *Android Pie*.
- My "playground" cluster is running Jenkins X *Next Generation* (we did not explore it yet).

The list of software releases with strange names goes on and on, and the only thing that they all have in common is that they have exciting and intriguing names that are easy to remember. Mojave, Pie, and Next Generation are good names for marketing purposes because they are easy to remember. For example, macOS Mojave is catchier and more natural to memorize than macOS 10.14.4. Those are releases for end-users that probably do not care about details like a specific build they're running. Those aren't even releases, but rather significant milestones.

For technical users, we need something more specific and less random, so let's take a look at the release identifiers of some of the software we are already using.

- I am currently running `kubectl` version `v1.13.2`.
- My Kubernetes cluster is `1.12.6-gke.10`.
- My **Helm** client is at the version `v2.12.1` (I'm not using **tiller**).
- I have `jx` CLI version `1.3.1074` and Git `2.17.2`.

> 📝 **What do all those versions have in common?**
> They all contain three numbers, with an optional prefix (e.g., `v`) or a suffix (e.g., `-gke.10`). All those tools are using semantic versioning. That's the versioning engineers should care about. That's the versioning most of us should use.

---

Next, let's explore semantic versioning in detail.