Creating a Microservices Starter Kit

Author: Laura Stone

Claps: 185

Date: May 7, 2019

This blog post is about how we began to adopt microservices at Klaviyo. Now, when I say the word â€œmicroservice,â€� itâ€™s not uncommon for people to disregard it as a buzzword. Microservices are often associated with companies like Netflix and Amazon. But Klaviyo isnâ€™t Netflix or Amazon. We donâ€™t have huge teams of developers hyper-focused on optimizing how we build and scale microservices. What we do have are several small teams. Each of these teams is focused on their area of the Klaviyo platform, tasked with how to change their areaâ€™s architecture for greater scale and stability while continuing to deliver value to customers week after week.

Our first step began with implementing a few simple standards and tools. This â€œmicroservices starter kitâ€� enables our engineers to create new services quickly without compromising quality or needing to reinvent the wheel operationally.

Components of the Standardized Starter Kit

The idea behind the toolkit was simple: it should be incredibly easy to create a new service at Klaviyo. As such, the toolkitâ€™s goals are to provide direction on which programming language and framework engineers should use, as well as how to use the frameworkâ€™s tooling to automate common tasks such as linting and testing. Additionally, the toolkit makes it easy to create and operationalize applications and their artifacts by standardizing how services are deployed and monitored.

So, the pieces of application development we standardized were the following:

- Language
- Framework
- Build Tool
- Continuous Integration/Deployment Platform
- Artifact Type
- Infrastructure Platform (with built-in monitoring and centralized logging)

At Klaviyo, we chose the following tools to standardize on:

Kubernetes, Python, Docker, Django, and Jenkins

Language: *Python*

Python is used across all teams at Klaviyo. We use Django to power the Klaviyo web application, as well as a large fleet of celery workers to execute a dizzying number of asynchronous tasks. Additionally, my team (Site Reliability Engineering) uses Python to orchestrate and analyze the state of our platform, for configuration management, and for many other tasks that fall under the category of â€œglue code.â€� Python is our primary language here at Klaviyo, so it made sense for us to standardize on it as the default and best-supported language for new microservices.

Framework: ***Django***

As mentioned above, the engineer team is most familiar with the Django framework due to its use powering Klaviyoâ€™s main web application. Again, it seemed logical to choose this framework to standardize on.

Build Tool: ***Make***

A generic automation tool can be extremely useful for developers to help automatically perform frequent tasks, such as unit testing, linting, and compilation. We decided to use make as our generic build tool at Klaviyo due to its ubiquity and simplicity.

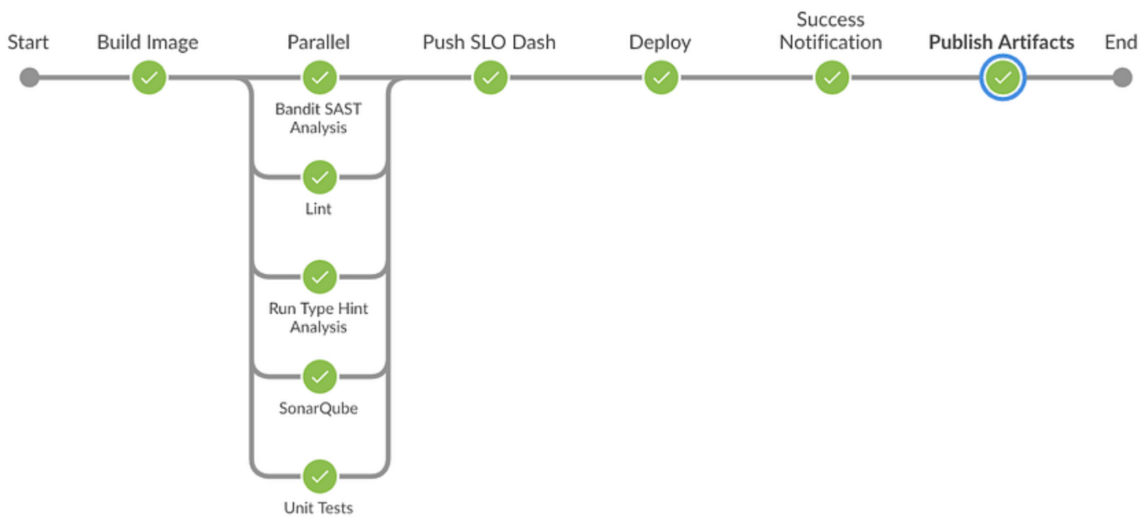
When engineers choose to create a new service via our standardized tooling, they automatically receive a Makefile complete with command shortcuts for:

- Creating a new Django application
- Installing application dependencies
- Running all manner of tests and checks (outlined in the next section)

A developer can switch between projects that have been created with this tooling and know that the way to run unit tests, regardless of language, application framework, or testing framework used, will be â€œmake test.â€� This is very simple but also very powerful because it is another variable the engineer does not need to think about. Itâ€™s another choice they donâ€™t need to make, one more thing they donâ€™t need to remember. Automating these tasks away behind a simple command means a massive increase in developer productivity.

Having one local build system also enables pre-defined continuous integration and continuous deployment pipelines. This means that each code check-in is verified by an automated build to detect problems before they are deployed. If all of those checks pass, a new code artifact is generated and deployed directly to our production environment.

Continuous Integration/Deployment Platform: ***Jenkins***



An example continuous integration pipeline

We use Jenkins as our platform for continuous integration and continuous deployment. As such, we incorporate a Jenkinsfile into our standardized microservices toolkit. By using this pre-created Jenkinsfile template, our engineers automatically receive the following continuous integration benefits:

- Unit tests
- Integration tests
- Linting
- Type checking
- Code quality analysis (code coverage, cyclomatic complexity, code duplication, etc.) via SonarQube
- Static security analysis via bandit

The template also includes artifact creation and storage in our artifact repository as well as deployment of that artifact to production for pushes to the release branch. If any of these checks or steps fails, the build itself fails and engineers are notified of the failure in Slack.

Very little customization is required to get this template to work, as it utilizes functionality we created by extending Jenkins Pipeline via Shared Libraries. That said, engineers are free to modify the Jenkinsfile to add or remove build stages as they see fit.

Artifact: *Docker Image*

Remember those artifacts being generated during our build pipelines? At Klaviyo, we use Docker images. Docker (a lightweight, OS-based virtualization platform) makes it easy for engineers to run the same artifact locally as exists in production or any other environment. This allows engineers to, in the event of a regression, quickly verify when releases cause regressions and identify bugs faster.

As part of the standardized toolkit, each project comes with a Dockerfile template that prescribes a base Docker image (we use the Docker Hub python:3 base image), installs OS and application dependencies to the image and is configured to run a Django application. There are also a number of base Docker images available in our Jenkins Shared Library that engineers can choose from to solve specific needs (e.g. an image with the SonarQube scanner installed).

Infrastructure Platform: *Kubernetes*

We deploy the Docker images generated during the build pipeline to our Kubernetes infrastructure. By choosing to deploy on this platform, newly created services logs are automatically sent to a centralized logging solution (via logstash, AWS S3, and AWS Athena), and pod-level data is automatically monitored (via Prometheus). Additionally, a serviceâ€™s pods are automatically subject to default resource limits.

This pattern provides a centralized location for both logging and monitoring data, making it easier for developers to troubleshoot and debug their applications in production.

Why is the toolkit so powerful?

- Itâ€™s flexible enough to allow teams to do what they want.
- Itâ€™s accessible enough that people can pick it up and use it without a lot of retraining.
- It enables creativity and experimentation by lowering the cost of releasing new services.
- It has taken service launch times from weeks to hours.

There are exceptions to these standards, particularly which languages and frameworks are used. Software engineers at Klaviyo are afforded high degrees of autonomy, enabling them to choose the right tool for the job whenever possible. To that end, we have several services written in Node.js because the JavaScript ecosystem had specific libraries with fantastic support for things we needed to incorporate into our platform. Additionally, we have a service being built in Java because it has performance requirements more stringent than we think Python can provide, better compatibility with frameworks like gRPC, and integrates nicely with Apache Flink.

The outlined standards are not meant as dogma, but as a way to speed up developer velocity. Standardized tools allow them to focus on what matters (the functionality they are building) and ignore what doesnâ€™t (the specific mechanics of how to set up unit testing or automated deployment). It also serves to minimize context switching between teams and projects. By having standards that are supported by the Site Reliability Engineering team, application-focused engineers can focus on getting stuff done and delivering value to customers.

What does the future look like?

While the existence of these tools has increased developer velocity tenfold, this project is far from complete. In an ideal world, everything about new service creation would be automated: from the YAML required to build a service in Kubernetes, to the specific repository values required by the Jenkinsfile. Additionally, the best-case developer workflow would be the ability to go to a UI, add application code and some other small amounts of metadata to some form, and hit Enter to receive a new service (or do something similar on the command line). This blog post describes the necessary first step toward this goal, and the next steps are on the roadmap.

If this sounds like something youâ€™d be interested in, check us out â€” weâ€™re hiring!