Getting Started with ChatOps

This lesson introduces us to the term "ChatOps" and explains how we can achieve it using Git and Prow.

We'll cover the following

- The role of Git in Jenkins X
- Communication driven actions
- RBAC (Role-Based Authentication)
- Using features of Git for ChatOps
- Slash commands using Prow

> ⚠ The examples in this chapter work only with serverless Jenkins X.

The role of Git in Jenkins X

Jenkins X's main logic is based on applying GitOps principles. Every change must be recorded in Git, and only Git is allowed to initiate events that result in changes to our clusters. That logic is the cornerstone of Jenkins X, and has served us well so far. However, there are actions we might need to perform that do not result in changes to the source code or configurations.

We might need to assign a pull request to someone for review. That someone might have to review the changes and might need to run some manual tests if they are not fully automated. A pull request could need additional changes, and the committer might need to be notified of that. Someone might need to approve and merge a pull request or choose to cancel it altogether. The list of the actions that we might have to perform once a pull request is created can be quite extensive, and many of them do not result in changes to source code. The period starting with the creation of a pull request until it is merged to the master is filled with communication, rather than changes to the project in question. As such, we need an effective way to facilitate that communication.

Communication driven actions

The communication and decision making that surrounds a pull request needs to be recorded. We need to be able to track who said what and who made what decision. Otherwise, we would not be able to capture the events that lead to a merge of a pull request to the master branch. We'd be running blind. That means that verbal communication must be discarded since it would not be recorded.

Given that such communication should be closely related to the pull request, we can discard emails and wiki pages as well, thus leading us back to *Git*. Almost every Git platform has a mechanism to create comments tied to pull requests. We can certainly use those comments to record the communication. But, communication by itself is useless if it does not result in *concrete actions*.

If we do need to document the communication surrounding a pull request, it would be a waste of effort to have to perform related actions separately. Ideally, communication should result in actions.

- When we write a comment that a pull request should be assigned to a reviewer, it should trigger an action that will do the actual *assignment* and *notify* that person that there is a pending action.
- If we comment that a pull request should be labeled as "urgent", that comment should *add* the label.
- If a reviewer writes that a pull request should be canceled, that comment should *close* the pull request.
- Similarly, if a person with sufficient permissions comments that the pull request is approved, it should be *merged* automatically.

RBAC (Role-Based Authentication)

There are a couple of concepts that need to be tied together for our process surrounding pull requests to be effective. We need to be able to communicate, and we already have comments for that. People with sufficient privileges need to be able to perform specific actions (e.g., merge a pull request). Git platforms already implement some form of **RBAC** (Role-Based Authentication), so that part is already solved.

Furthermore, we need to be notified that there is a pending action we should perform as well as the fact that a significant milestone has been reached. This is

solved as well. Every Git flavor provides a notification mechanism. What we're missing is a process that will tie it all together by executing actions based on our comments.

>  We should be able to implement ChatOps principles if we manage to convert comments into actions controlled by RBAC and if we receive notifications when there is a pending task.

The idea behind ChatOps is to unify communication about the work that should be done with the history of what has been done. The expression of a desire (e.g., approve this pull request) becomes an action (execution of the approval process) and is at the same time recorded.

ChatOps is similar to verbal communication, like commands we might give if we had a butler. "Please make me a meal," is an expression of a desire. Your butler would transmit your wish to a cook, wait until the meal is ready, and bring it back to you.

Given that we are obsessed with recording everything we do when developing software, verbal expressions are not good enough, so we need to write them down. Hence the idea of *ChatOps*.

>  ChatOps converts parts of communication into commands that are automatically executed and provides feedback on the results.

Using features of Git for ChatOps

In a ChatOps environment, a chat client is the primary source of communication for ongoing work. However, since we adopted Git as the only source of truth, it should come as no surprise that the role of a chat client is given to Git. After all, it has comments, and that can be considered chat of sorts. Some call it GitChat, but we'll stick with the more general term, ChatOps.

If we assume that only Git should be able to initiate a change in our clusters, it stands to reason that such changes can be started either of the following:

- A change in the source code.

- Writing comments in Git.
- Creating an issue.

We can define ChatOps as conversation-driven development. Communication is essential for all but single-person teams. We need to communicate with others when the feature we're developing is ready. We need to ask others to review our changes. We might need to ask for permission to merge to the master branch. The list of things we need to communicate is infinite. That does not mean that all communication becomes ChatOps, but rather that parts of our communication does. It's up to the system to figure out which parts of communication should result in actions, and what is pure human-to-human messaging without tangible outcomes.

As we already saw, four elements need to be combined into a process. We need communication (comments), permissions (RBAC), notifications (email), and actions. All but the last are already solved in every Git platform. We just need to figure out how to combine comments, permissions, and notifications into concrete actions. We'll do that by introducing **Prow** to our solution.

Slash commands using Prow

Prow is a project created by the team managing continuous delivery processes for Kubernetes projects. It does quite a few things, but we will not use everything it offers because there are better ways to accomplish some of its tasks. The parts of **Prow** we are primarily interested in are those related to communication between Git and processes running in our cluster.

We'll use it to capture Git events created through *slash commands* written in comments. When such events are captured, **Prow** will either forward them to other processes running in the cluster (e.g., execute pipeline builds), or perform Git actions (e.g., merge a pull request).

Git actions (e.g., merge a pull request).

Since this might be the first time you hear the term slash commands, so a short explanation might be in order.

Slash commands act as shortcuts for specific actions. Type a slash command in the Git comment field, click the button, and that's it. You executed a task or a command. Of course, our comments are not limited to slash commands. Instead, they are often combined with "conversational" text. Unfortunately, commands and the rest of the communication must be in separate lines. A command must be at the start of a line, and the whole line is considered a single command. We could, for example, write the following text.

```
This PR looks OK.

/lgtm
```

Prow parses each line and will deduce that there is a slash command `/lgtm`.

Slash commands are by no means specific to Git and are widely used in other tools. Slack, for example, is known for its wide range of supported slash commands and the ability to extend them. But, since we are focused on Git, we'll limit our ChatOps experience with Slash commands to what **Prow** offers as the mechanism adopted by Jenkins X (and by the Kubernetes community).

Prow will be our only entry point to the cluster. Since it only accepts requests from Git webhooks or slash commands, the only way we will be able to change something in our cluster is by changing the source code or by writing commands in Git comments. At the same time, **Prow** is highly available (unlike static Jenkins), so we'll use it to solve yet another problem.

> 🔍 At the time of this writing, **Prow** only supports GitHub. The community is working hard on adding support for other Git platforms. Until that is finished, we are restricted to GitHub. If you do use a different Git platform (e.g., GitLab), I still recommend going through the exercises in this chapter. They will provide a learning experience and chances are by the time you start using Jenkins X in production, support for other Git flavors will be finished.

> ⚠ We cannot use **Prow** with anything but GitHub, and serverless Jenkins X doesn't work without **Prow**. However, that is not an issue. Or, at least, it is not a problem anymore. The team created a new project called **Lighthouse** that performs all the same functions that **Prow** provides while supporting all major Git providers. Everything you read about **Prow** applies equally to **Lighthouse**, which should be your choice if you do not use GitHub.

As always, we need a cluster with Jenkins X to explore things through hands-on exercises.