Naming in deployment pipelines for SaaS

Thomas Kurpick[1], Sebastian Melchior[2]

Abstract: In the race to minimize operational costs, Software as a Service (SaaS) platforms has become increasingly popular. The development of SaaS, however, introduces several aspects that must be considered. In this paper, we present the continuous deployment pipeline of our eTrusted Enterprise SaaS Platform. Thereby, we focus on lessons learned during the evolution of our Continuous Deployment Pipeline with regards to naming and build step order.

Keywords: Engineering of Deployment Pipelines, Provisioning of Software & Infrastructure, Test Automation

1 Introduction

Trusted Shops is a supplier of feedback solutions in the ecommerce sector and offers several services for offline [Lo15] and online shops, such as customer reviews, product reviews and insurance of purchases. As digitalization requires enterprises to put the customer in the center of their organization, Trusted Shops founded the enterprise unit in 2016 offering a transactional feedback platform as a solution, not only for ecommerce but also for a wider variety of business areas, like insurance, mobility and banking.

Experience in scalable and robust service design for over a decade resulted in a microservice architecture that decouples the different aspects that are necessary for a transactional feedback platform. To minimize operational effort for operation and maintenance we choose a continuous deployment approach which allows changes to become visible as fast as possible [Ha10]. Furthermore, to maximize efficiency of our workforce and reduce human error we have minimized manual validation tasks.

Continuous deployment approach [Fi09] automatically releases every change that passes all the stages of the pipeline. We choose this approach to minimize the actual risk of a single change, and to add value to our platform as efficiently as possible.

Developing a SaaS platform together with a continuous deployment pipeline approach introduced the following aspects that needed further attention [HF10]: The general architecture of the platform that allows us to use independent deployment of systems and subsystems, source code management of these components and the actual deployment steps necessary to build, test and deploy the system and subsystems.

[1] Trusted Shops, Trusted Enterprise, Subbelrather Str. 15c, 50823 Köln, thomas.kurpick@etrusted.com
[2] Trusted Shops, Trusted Enterprise, Subbelrather Str. 15c, 50823 Köln, sebastian.melchior@etrusted.com

2 Enterprise Continuous Deployment Pipeline

The platform was developed as a greenfield approach. The team decided to structure the new platform as a microservice architecture. Each microservice is developed independently from each other. No compiled sources are shared between the microservices. Runtime dependencies resulting from requests made to other services must be robust in case of failures. Deployments of a new version are always done with a rolling deployment.

2.1 Revision Source Control with git

For every microservice, we create a standalone git repository that holds all source code and build scripts that are necessary to build the final deployable artifact. For the git branching model we decided not to choose the popular GitFlow [Dr10] branching model, instead we used a simplified version in favor of a quality strategy that uses the pipeline as a quality gate and not the merges of the GitFlow branching model.

Development of new features, bug fixes and changes are developed in their own feature branch. Each commit is checked by our continuous integration server with unit testing and static code analysis. The results are added as commit comments. Once the changes are completed, the developer submits a merge request via our git repository server. The team member, who is assigned to the merge request, does the code review and can comment on critical parts of the changes. When the merge request is accepted, the merge is pushed automatically to the master branch.

This branching model allows us to minimize merges; guarantees that only the master branch is the source of the following pipeline. On the other hand, it is possible to reject changes before they are merged to the master branch. It is easier to reject a commit that has a typo or small errors, than to create an additional bug report or change request just for small changes.

The additional branches that are used by GitFlow are not necessary for SaaS, because we do not deliver or support multiple service versions, as suggested in [Ha07]. We only support the current deployed version x and the new version x+1.

2.2 Pipeline Steps with Jenkins

For each microservice we have two Jenkins jobs [Je11], a simple check job and the continuous deployment job. The check job is triggered for each non-master branch commit. It executes the unit and integration test. Additionally, a static code analysis with Sonarqube is triggered [So07]. The result of the code analysis is not recorded in the global Sonarqube instance, but only posted as comments to the responsible commit in Gitlab. This way potential technical debt is visible to everyone before the merge to master is approved. It can be discussed whether this should be fixed before the merge is approved, or if the issues found by Sonarqube can be accepted.

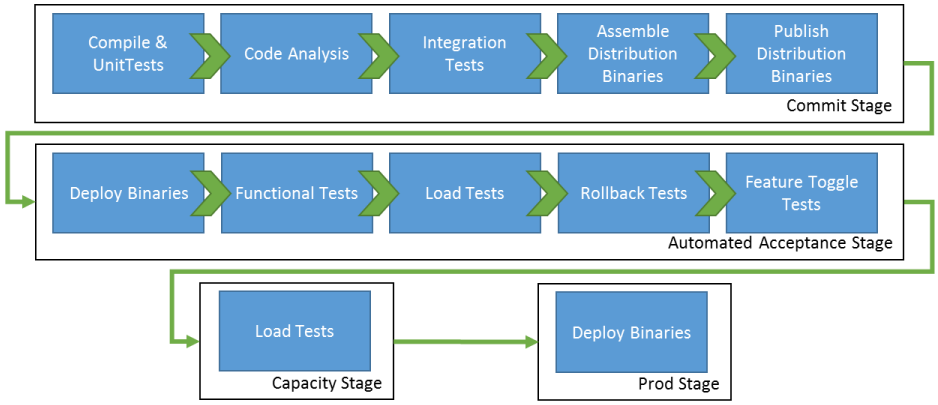

Figure 1 Overview of the deployment pipeline steps used for the eTrusted Enterprise Platform

The continuous deployment pipeline job is triggered with each commit to master. The master branch is checked out and built. After the build the unit tests are executed with the code coverage tool JaCoCo. After the test, the code is analyzed with Sonarqube. The results are transmitted to the Sonarqube server. After the analysis, the integration tests start. The difference between the unit tests and the integration tests is the intention of the tests. The unit tests are purely white box tests for functional tests of internal components without communication with other external components (database, other services, third party services). Whereas the integration tests focus on consumer driven tests that are determined from other teams.

After the tests, the microservice is assembled into a Docker container and published into the container repository. We use AWS as our cloud provider with the ECR service [Ec15].

After the publication of the new container, the pipeline triggers the deployment to the Automated Acceptance stage. The deployment is completed via helm on our Kubernetes Cluster. The deployment is configured as a rolling deployment, with no downtime of the services. After a successful deployment, automated functional end-to-end tests are initiated. These tests focus on user functionality that is visible to users of the platform. Load test for various parts of the platform are started after the automated end-to-end tests are finished.

Additionally, we execute service roll back tests. This test deploys the new version of the service, executes all functional tests, rolls the service back to the previous version and executes the tests again. This procedure guarantees that we are able to roll back to the previous version of a service, if there are problems during or after the deployment of the service. Another test step is used to validate our feature toggles for our platform. We use the feature toggle concept to release changes only for specific tenants or users.

After the Automated Acceptance Test stage, we deploy the service to the capacity stage. Here we stress test our service in a production like environment with load that simulates the production load. Test results are collected and stored in our monitoring system.

After successful performance tests, we deploy the service to the production environment. Besides the automated smoke test, there are no automated tests on this stage but monitoring of system properties. The production environment is monitored by Prometheus.

Errors that where not caught through the automated tests can break our system. But through the automated rollback test, we have the possibility to revert the change or fix forward. We have established an incident review meeting for these errors in order to improve our deployment pipeline. After the actual error is fixed, we have a meeting with the involved team to understand which automated tests were missing in our pipeline.

3 Lessons learned in naming in deployment pipelines

Existing naming guidelines for components focus on descriptive names [Jc99] or [Vs03]. At the beginning, we always used long descriptive names for our components like FeedbackRequestTriggerService, as suggested by the common naming guidelines.

As we developed our pipelines and development setup for our microservice architecture, we encountered different problems with names used to identify services and single page applications. In each system we added to the development and deployment setup, we required names, e.g.: git repository name, Jenkins job name, Kubernetes Helm chart name, etc... Each system has its constraints for the concepts we used, e.g.: Kubernetes [Ku14] in combination with Helm [He16] has a length constraint for its container names of 12 characters; otherwise the name will be trimmed. This resulted in naming our components with two names, a long descriptive name and also a short name with max 8 characters.

In addition, Kubernetes' service names are limited to only 63 characters, because the service names are used by Kubernetes as DNS name segments. This constraint is derived from RFC1035. Jenkins has a best practice of naming its job without the space character. Although it is possible, the job name is used as directory name in the underlying filesystem. Depending on the OS and filesystem format, directory names with spaces can lead to errors in build scripts. At the moment there are no other constraints known in our deployment pipeline. In figure 2 you can see the different systems and how they are interconnected with the names on the logical component.

With each new system, we integrate in our deployment pipeline we must check if new constraints are introduced to our component names, which makes it difficult to choose tools. Violated constraints are usually noticed during integration of these systems. This causes additional work to reassess the integration type and/or tool choice. In listing 1 is a selection of our current naming guidelines, that connect the different systems with each other.

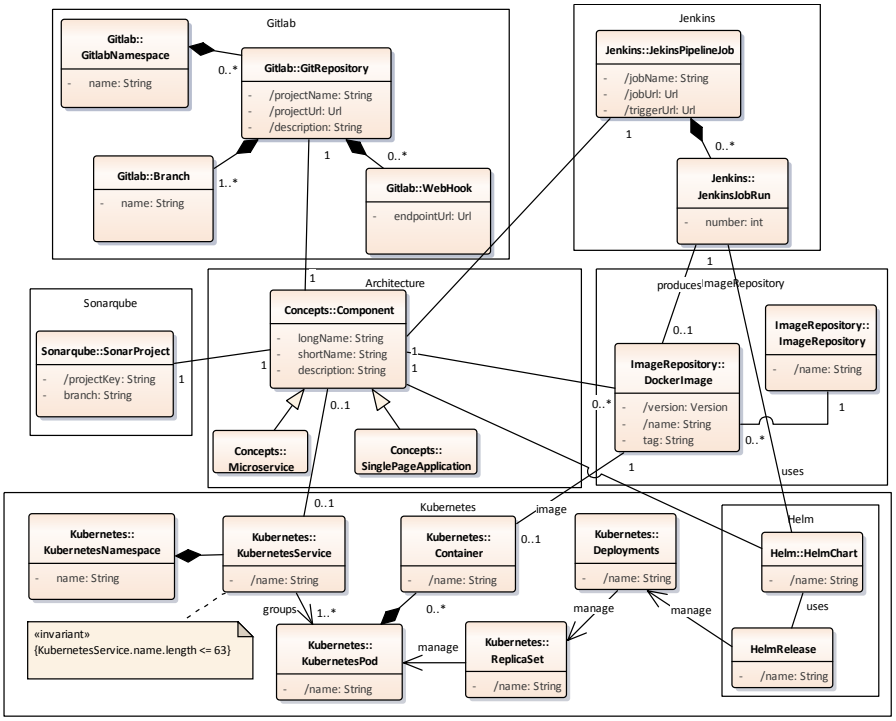

Figure 2 Connected naming concepts of the deployment pipeline systems

Because we integrate new systems in an incremental way, the risk that new systems are not well integrated exists. It could lead to the renaming of all other objects in existing systems, because we change the guidelines of an existing name. Or we introduce an additional name for our components, and then we have to name our existing components according to the new name. Both possibilities are not favorable.

```
KubernetesService.name = component.shortName
ReplicationSet.name = component.shortName + “-“ + <randomeNr>
KubernetesPod.name = ReplicationSet.name + “-“ + <randomeString>
HelmChart.name = component.shortName
HelmRelease.name = component.shortName
HelmFullname = HelmChart.name + “-“ + HelmRelease.name
Depoyments.name = HelmFullname
JenkinsPipelineJob.jobName = component.longName + “-pipeline-master”
GitRepository.projectName = component.longName
GitRepository.projectUrl = GitlabNamespace.name + “/” +
GitRepository.projectName
SonarProject.projectKey = OrganizationName + “:” + component.longName +
“:” componenent.GitRepository.Branch.name
```

Listing 1 Selection of naming guidelines and mappings used in our deployment setup

Our current naming guidelines for our components are:

1. **Long descriptive name**, use only lowercase words separated with “-“
2. **Short abbreviation name**, use only lowercase letters without any special characters. The maximum length is 8 characters.

3. **Additional description**, describe the business function of the component for a new or external person.
4. **Use the long descriptive name whenever it is possible in the deployment pipeline!**

At the moment the naming guidelines are manually enforced by our review process for new components. The other naming mappings of derived names are enforced by our deployment pipeline with scripts. We differ from the previous mentioned guidelines [Vs03, Jc99], because the names are part of URLs in our systems. This imposes constraints on the names that we have to address.

4 Summary & Conclusion

The current setup of our continuous deployment pipeline is working. We are confident that errors are detected in the early stages of our pipeline. Though our agile approach of adding new systems to our pipeline, we run into naming problems that forced us to reassess our naming guidelines and patterns more than once. In the future, we will automate the process for new components, so that we can create and deploy new components in a consistent way.

References

[An10] Anderson, David J: Kanban. Blue Hole Press, 2010.

[Dr10] Driessen, Vincent: A successful Git branching model, http://nvie.com/posts/a-successful-git-branching-model, retrieved: 8.1.2017

[Ec15] Amazon EC2 Container Registry, https://aws.amazon.com/ecr, retrieved: 8.1.2017

[Fi09] Fitz, Timothy: Continuous Deployment, http://timothyfitz.com/2009/02/08/continuous-deployment, retrieved: 8.1.2017

[Ha07] Hamilton, James R: On Designing and Deploying Internet-Scale Services. In: LISA'07 Proceedings of the 21st conference on Large Installation System Administration Conference. p. 1-18, 2007.

[He16] Helm: The Kubernetes Package Manager, https://helm.sh, retrieved: 8.1.2017

[HF10] Humble, Jez; Farley, David: Continuous delivery: reliable software releases through build, test, and deployment automation. Pearson Education, 2010.

[Jc99] Java Naming Conventions, http://www.oracle.com/technetwork/java/codeconventions-135099.html, retrieved: 8.1.2017

[Je11] Jenkins, https://jenkins.io, retrieved: 8.1.2017

[Ku14] Kubernetes: A Container Cluster Manager, http://kubernetes.io, retrieved: 8.1.2017

[Lo15] Locatrust, Entwicklung von Vor-Ort-Trusted-Shop-Zertifikaten für Handel und Kleinunternehmen, Pressemitteilung Trusted Shops, Köln

[So07] Sonarqube: Continuous Code Quality, https://www.sonarqube.org, retrieved: 8.1.2017

[Vs03] Component Naming Recommendations, https://msdn.microsoft.com/en-us/library/4867dawt(v=vs.71).aspx, retrieved: 8.1.2017