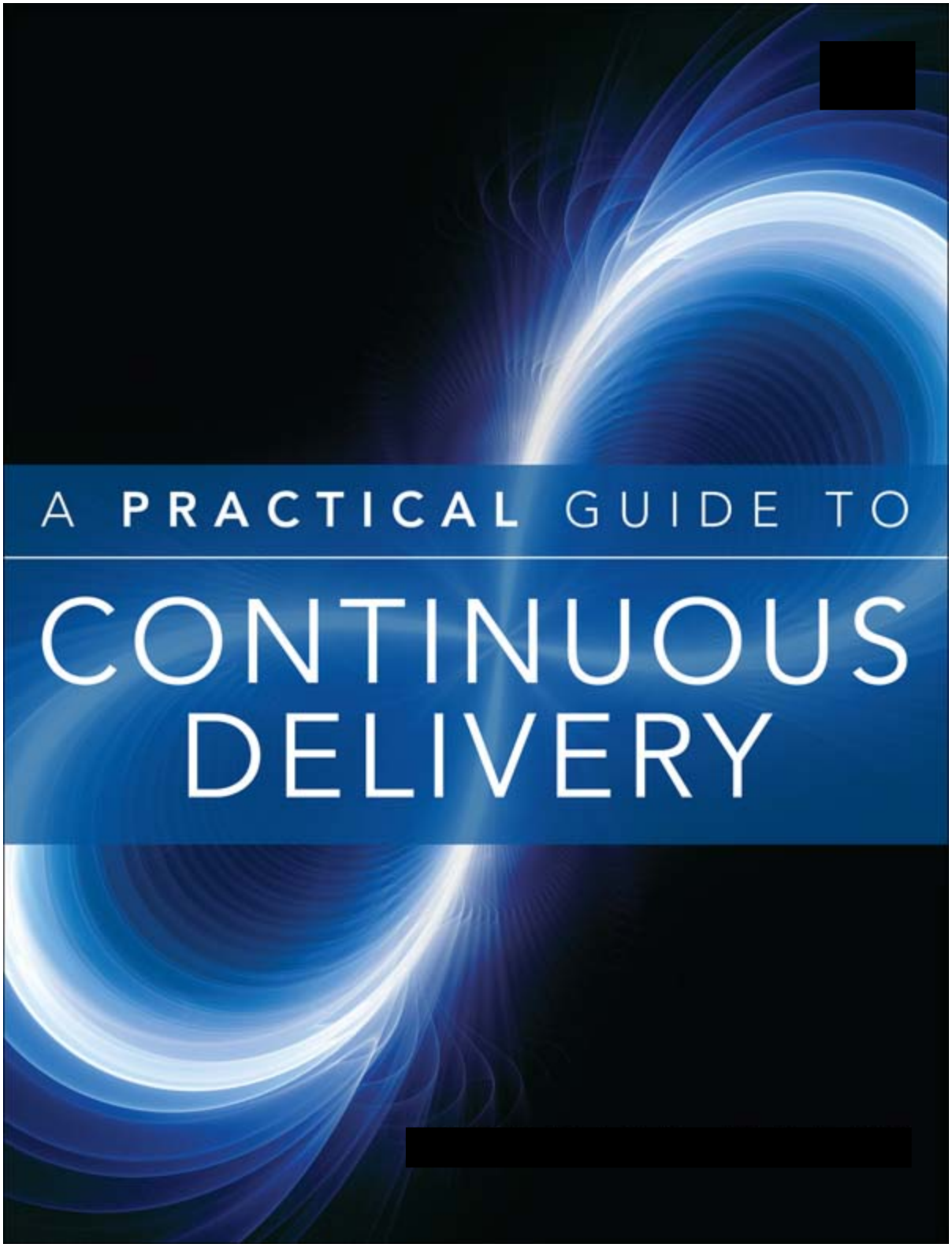
A PRACTICAL GUIDE TO
CONTINUOUS
DELIVERY

Contents at a Glance

Contents

PART I

Foundations

In this part Chapter 1 provides the basics to understand Continuous Delivery. Chapter 2 explains the technical foundations of Continuous Delivery: It discusses the automated deployment of infrastructure and the automated installation of software without which Continuous Delivery would be impossible.

Chapter 1. Continuous Delivery: What and How?

1.1 Introduction: What Is Continuous Delivery?

This question is not so easy to answer. The inventors of the term do not provide a real definition.[1] Martin Fowler focuses in his discussion[2] of Continuous Delivery on the fact that software can be brought into production at any time. This requires an automation of the processes necessary for the installation of software and feedback about software quality. Wikipedia[3] on the other hand defines Continuous Delivery as an optimization and automation of the software release process.

In the end, the main objective of Continuous Delivery is to analyze and optimize the process leading up to the release of software. Exactly speaking this process is often blended out during development.

1.2 Why Software Releases are So Complicated

Software releases are a challenge—very likely every IT department has already worked during a weekend to bring a software release into production. Such events often end with bringing the software somehow into production—because from a certain point the path back to the old version is even more dangerous and difficult than the path ahead. However, the installation of the release is then often followed by a long phase in which the release has to be stabilized.

1.2.1 Continuous Integration Creates Hope

Nowadays it is the release into production that represents a challenge. Not so long ago the problems started much earlier: Individual teams worked independently on their modules, and prior to the release the different versions first had to be integrated. When the modules were put together for the first time, the system frequently did not even compile. Often it took days or even weeks until all changes were integrated and compiled successfully. Only then could the deployments commence. These problems have mostly been solved by now: All teams work on a shared version of the code that is permanently automatically integrated, compiled, and tested. This approach is called Continuous Integration. The required infrastructure for Continuous

Integration is detailed in Chapter 3, "Build Automation and Continuous Integration." The fact that the former problems associated with this phase have been solved raises hopes that there will also be solutions for the problems arising during the other phases leading up to production.

1.2.2 Slow and Risky Processes

The processes in the later phases are often highly complex and elaborate. In addition, manual steps render them very tedious and error-prone. This is true for the release into production, but also for the preceding phases—for example, during testing. Especially during a manual process, which, to make things worse, is only performed a few times per year, errors are likely to occur. This of course contributes to the risk associated with the overall procedure.

Because of the high risk and complexity, releases are not very frequently brought into production. In the end this causes the processes to take even longer due to lack of practice. In addition, this makes it difficult to optimize the processes.

1.2.3 It's Possible to be Fast

On the other hand, there are always possibilities to bring a release rapidly into production in an emergency—for instance, when an error has to be urgently repaired. However, in such a case all the tests and therefore all the safety nets, which are an integral part of the standard process, are omitted. This is of course a pretty high risk—there are good reasons to normally run those tests.

Therefore, the normal path into production is slow and risky—and in emergencies the path can be faster, but at the expense of even more risk.

1.3 Values of Continuous Delivery

Using the motivation and the approaches of Continuous Integration, we want to optimize the way for releases into production.

A fundamental principle of Continuous Integration is: "If it hurts, do it more often and bring the pain forward." What sounds like masochism is in reality an approach for problem solving. Instead of avoiding problems with releases by bringing as few releases as possible into production, these processes should be performed as often and as early as possible in order to

optimize them as quickly as possible—with regards to speed and with regards to reliability. Consequently, Continuous Delivery forces the organization to change and to adopt a new way of working.

In the end this approach is not really surprising: As mentioned, every IT organization is able to rapidly bring a fix into production—and in such a scenario it is common practice to perform only a fraction of the usual tests and security checks. This is possible because the change is small and therefore represents only a small risk. Here, another approach for minimizing risk becomes apparent: Instead of trying to safeguard against failures via complex processes and rare releases, it is also possible to more frequently bring small changes into production. This approach is in essence identical to the Continuous Integration strategy: Continuous Integration means that even small software changes by the individual developers and the team are permanently integrated instead of having the teams and developers work independently for days or weeks and integrating all the accumulated changes only at the end—a strategy that frequently causes substantial problems; in some cases the problems are so large that the software cannot be compiled at all.

However, Continuous Delivery is more than just "fast and small." Continuous Delivery is based on different values. From these values, concrete technical measures can be deduced.

1.3.1 Regularity

Regularity means to execute processes more frequently. All processes that are necessary to bring software into production should regularly be performed—and not only when a release has to be brought into production. For example, it is necessary to build test and staging environments. The test environments can be used for acceptance or technical tests. The staging environments can be used by the final customer for testing and evaluating the features of a new release. By providing these environments, the process for the generation of an environment can turn into a regular process that is not merely performed when the production environment has to be created. To generate this multitude of environments without too much effort the processes have to become largely automated. Regularity usually leads to automation. Similar rules apply to tests: It does not make sense to postpone the necessary tests until right before the release—instead they should rather be performed regularly. Also in this case this approach basically forces

automation in order to limit the necessary effort. Regularity also leads to a high degree of reliability—processes that are frequently performed can be reliably repeated and executed.

1.3.2 Traceability/Confirmability

All changes to the software that is supposed to be brought into production and to the infrastructure that is required for the release have to be traceable. It has to be possible to reconstruct each state of the software and of the infrastructure. This leads to versioning that does not only comprise the software, but also the necessary environments. Ideally, it is possible to generate each state of the software together with the environment required for the operation in the right configuration. Thereby all changes to the software and the environments can be traced. Likewise it is very easy to generate a suitable system for error analyses. And finally, changes can be documented or audited in this manner.

One possible solution for the problem is that production and staging environments are only accessible for certain persons. This is supposed to avoid “quick fixes” that are not documented and cannot be traced anymore. Besides, security requirements and data security argue against accessing production environments.

With Continuous Delivery, interventions into an environment are only possible when an installation script is changed. The changes to the scripts are traceable when they are deposited in a version control system. The developers of the scripts also do not have access to the production data so that there are also no problems with data security.

1.3.3 Regression

To minimize the risk associated with bringing software into production, the software has to be tested. Of course, the correct functioning of new features has to be assured during the testing. However, a lot of effort arises from the attempt to avoid regressions—that is, errors that are introduced by modifications in already tested software parts. This would in effect require that all tests be rerun in case of a modification since in the end a modification at one site of the system might cause an error somewhere else. This necessitates automated tests. Otherwise the required effort for the execution becomes much too high. Should an error nevertheless make its way into production, it can still be discovered by monitoring. Ideally, there is

the possibility to install as simply as possible an older version on the production system without the error (rollback) or to bring a fix quickly into production (roll forward). In the end the idea is to have a kind of early warning system that takes measures throughout different phases of the project, like test and production, to discover and solve regressions.

1.4 Benefits of Continuous Delivery

Continuous Delivery offers numerous benefits. Depending on the scenario the different advantages can be of varying importance—consequently this will influence how Continuous Delivery is implemented.

1.4.1 Continuous Delivery for Time to Market

Continuous Delivery decreases the time required for bringing changes into production. This generates a substantial benefit on the business end: It becomes much easier to react to changes of the market.

However, the advantages extend beyond a faster time to market: Modern approaches like Lean Startup[4] advocate a strategy that benefits even more from the increased speed. The focus of Lean Startup is to position products on the market and to evaluate their chances at the market while investing as little effort as possible in doing so. Just like with scientific experiments, it is defined beforehand how the success of a product on the market can be measured. Then the experiment is performed, and afterwards the success or failure is measured.

1.4.2 One Example

Let us look at a concrete example. In a web shop a new feature is supposed to be created: Orders can be delivered on a defined day. As a first experiment the new feature can be advertised. Here the number of clicks on a link within the advertisement can be used as an indication for the success of this experiment. At this point no software has been developed yet—that is, the feature is not yet implemented. If the experiment did not lead to a promising result, the feature does not appear to be beneficial and other features can be prioritized instead—without much effort having been invested.

1.4.3 Implementing a Feature and Bringing It into Production

If the experiment was successful, the feature will be implemented and brought into production. Even this step can be conducted like an experiment: Metrics can help to control the success of the feature. For example, the number of orders with a fixed delivery date can be measured.

1.4.4 On to the Next Feature

The analysis of the metrics reveals that the number of orders is high enough —interestingly most orders are not sent directly to the customer, but to a third person. Additional measurements show that the ordered items are obviously birthday presents. Based on this information the feature can be expanded—for example with a birthday calendar and recommendations for suitable presents. This requires of course that such additional features are designed, implemented, brought into production and finally evaluated with regards to their success. Alternatively, there might also be options to evaluate the potential market success of these features without any implementation—via advertisements, customer interviews, surveys, or other approaches.

1.4.5 Continuous Delivery Generates Competitive Advantages

Continuous Delivery makes it possible to more rapidly bring required software changes into production. This allows enterprises to more quickly test different ideas and to develop their business model further. This creates a competitive advantage: Since more ideas can be evaluated, it is easier to filter out the right ones—and this is not based on subjective estimations of market chances, but on the basis of objectively measured data (Figure 1.1).

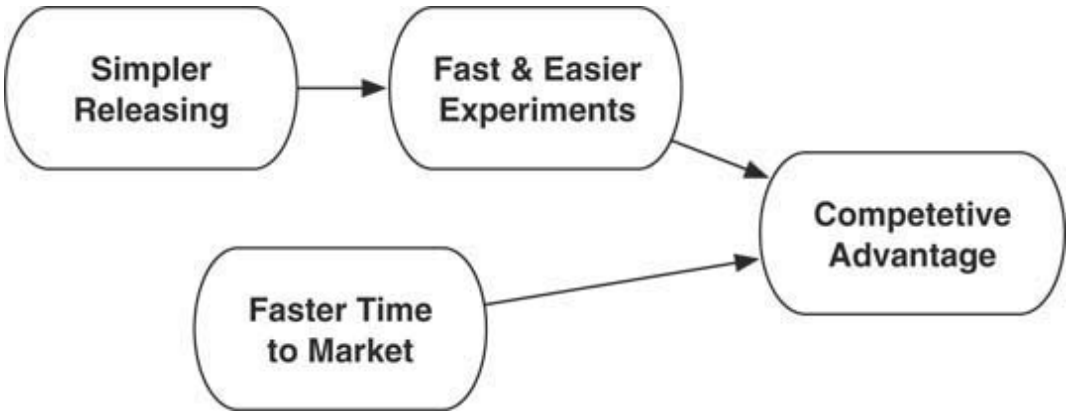

Figure 1.1 *Reasons for Continuous Delivery in a startup*

1.4.6 Without Continuous Delivery

Without Continuous Delivery the feature for the fixed delivery dates would have been planned and brought into production during the next release—this would likely have taken a number of months. Before the release, marketing would hardly have dared to advertise the feature since the long time up to the next release would render such advertisements futile. If the feature had not proven successful in the end, there would have been high costs caused by its implementation without creating any benefit. Evaluating the success of a new feature would certainly also be possible in the classical approach; however, the reaction would be drastically slower. Further developments such as the features supporting the buying of birthday presents would reach the market much later since they would require that the software be brought into production again and the time-consuming release process be run through a second time. Besides, it remains doubtful whether the success of the feature would have been analyzed in enough detail to recognize the potential for additional features supporting the shopping for birthday presents.

1.4.7 Continuous Delivery and Lean Startup

Therefore, optimization cycles can be passed through much faster thanks to Continuous Delivery because each feature can be brought into production practically at any time. This makes approaches like Lean Startup possible. This influences how the business end is working: It has to more rapidly define new features and does not have to focus on long range planning anymore, but can immediately react to the outcome of the current experiments. This is especially easy in startups, but such structures can also be built in classical organizations. The Lean Startup approach has, unfortunately, a misleading name: It is an approach where new products are positioned on the market via a series of experiments, and this approach can of course also be implemented in classical enterprises, not only in startups. It can also be used when products have to be delivered classically—for instance, on media such as CDs, with other complex installation procedures, or as part of another product such as a machine. In such a case the installation of the software has to be simplified or ideally automated. Besides a range of customers has to be identified who would like to test new software versions and be willing to provide feedback on them—that is, classical beta testers or power users.

1.4.8 Effects on the Development Process

Continuous Delivery influences the software development process: When individual features are supposed to be brought into production, the process has to support this. Some processes use iterations of one or several weeks' length. At the end of each iteration a new release with several features is brought into production. This is not an ideal approach for Continuous Delivery because in this way individual features cannot pass through the pipeline on their own. This also poses obstacles for the Lean Startup approach: When several features are rolled out at the same time, it is not obvious which change influences the measured values. Let us assume that the option for delivery on a fixed date is introduced in parallel with a change of the shipment costs—it will not be possible to distinguish which of the two changes had a greater influence on the higher number of sold items.

Therefore, processes like Scrum, XP (Extreme Programming), and of course the waterfall are impedimentary since they always bring several features together into production. Kanban,[5] on the other hand, focuses on bringing a single feature through the different phases into production. This fits ideally with Continuous Delivery. Of course, the other processes can also be modified in ways that allow them to support the delivery of individual features. However, in such a case the processes have been adapted and are not implemented according to the textbook anymore. Another possibility is to initially deactivate the additional features in order to bring several features together in one release into production, but still be able to measure their effects separately.

In the end this approach also means that teams include multiple different roles. In addition to development and operation of the features, business roles such as marketing are conceivable. Thanks to the decreased organizational hurdles, the feedback from the business end can be translated into experiments even faster.

Try and Experiment

- Gather information about Lean Startup and Kanban. Where did Kanban come from originally?

Choose a project you know or a feature in a project:

- What could a minimal product look like? The minimal product should give an idea about the market chances of the planned complete product.
- Is it also possible to evaluate the product without software? Is it, for instance, possible to advertise it? Are interviews of potential users an option?
- How can the success of the feature be measured? Is there, for instance, an influence on sales, a number of clicks, or another value that could be measured?
- How much time in advance do marketing and sales typically have for planning a product or feature? How does that fit to the Lean Startup idea?

1.4.9 Continuous Delivery to Minimize Risk

The use of Continuous Delivery as described in the last section goes together with a certain business model. However, for classical enterprises the business often depends on long-range planning. In such a case an approach like Lean Startup cannot be implemented. In addition, there are many enterprises for which time to market is not a decisive factor. Not all markets are very competitive in this regard. This can of course change when such companies are suddenly confronted with competitors that are able to enter the market with a Lean Startup model.

In many scenarios time to market cannot motivate the introduction of Continuous Delivery. Still the techniques can be useful since Continuous Delivery offers additional benefits:

- Manual release processes require a lot of effort. It is no rare event that entire IT departments are blocked for a whole weekend for a release. And after a release there is frequently still extensive follow-up work to do.
- Also the risk is high: The software rollout depends on many manual modifications, which easily leads to mistakes. If the errors are not

discovered and fixed in time, this can have far-reaching consequences for the enterprise.

The sufferers are found in the IT departments: Developers and system administrators who work through weekends and nights to bring releases into production and to fix errors. In addition to working long hours they are subjected to high stress because of the high risk. And the risks should not be underestimated: Knight Capital, for instance, lost $440M because of a failed software rollout.[6] As a consequence the company went into insolvency. A number of questions[7] arise from such scenarios—in particular why the problem occurred, why it wasn't noticed in a timely manner, and ultimately how such events can be prevented in other environments.

Continuous Delivery can be a solution for such situations: Fundamental aspects of Continuous Delivery are the higher reliability and the quality of the release process. This allows developers and system administrators to sleep calmly in the true sense of the word. Different factors are relevant for this:

- Due to the higher level of automation of the release processes, the results become easier to reproduce. Thus, when the software has been deployed and tested in a test or staging environment, the exact same result will be obtained in production because the environment is completely identical. This allows largely eliminating sources of error, and consequently the risk decreases.
- In addition, testing software becomes much easier since the tests are largely automated. This increases the quality further as the tests can be performed more frequently.
- When there are more frequent deployments, the risk decreases likewise since fewer changes are brought into production per deployment. Fewer changes translate into a smaller risk that an error has crept in.

In a way, the situation is paradoxical: The classical IT tries to bring releases as seldom as possible into production since they are associated with a high risk. During each release an error with potentially disastrous consequences can creep in. Fewer releases should therefore result in fewer problems.

Continuous Delivery on the other hand advocates frequent releases. In that case fewer changes go live at each release, which also decreases the probability for the occurrence of errors. Automated and reliable processes are a prerequisite for this strategy. Otherwise the frequent releases lead to an overload of the IT personnel performing manual processes, and in addition the risk increases since errors are more prone to occur during manual processes. Instead of aiming at a low release frequency the relevant process are automated to decrease the release-associated risk. It is of course an added advantage that in case of a high release frequency the individual releases comprise fewer changes so that the inherent risk of errors is lower.

Here, the motivation for Continuous Delivery (Figure 1.2) thus profoundly differs from that of the Lean Startup idea: The focus is on reliability and a better technical quality of the releases—not on time to market. And the beneficiaries are the IT departments—not only the business domains.

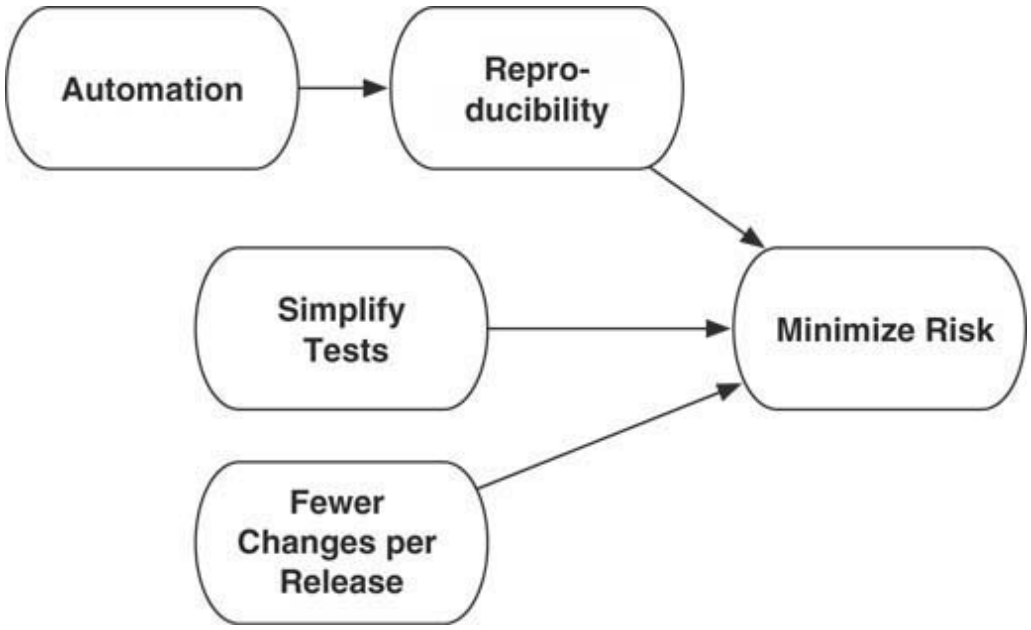

Figure 1.2 *Reasons for Continuous Delivery in an enterprise*

Since the benefits are different, other compromises can be made: For example, investing in a Continuous Delivery pipeline is often worthwhile even if it does not extend all the way up to production—that is, when the production environment still has to be built manually. In the end the production has only to be built once for each release, but multiple environments are required for the different tests. However, if time to market

is the main motivation for Continuous Delivery, it is essential that the pipeline include production.

Try and Experiment

Look at your current project:

- Where do problems typically arise during installation?
- Could these problems be solved by automation?
- Where should the current approaches be simplified in order to facilitate automation and optimization? Evaluate the required effort and the expected benefit.
- How are production systems and test systems currently built? By the same team? Would it be conceivable to apply automation to both areas or only to one of the two?
- For which systems would automation be useful? How often are the systems built?

1.4.10 Faster Feedback and Lean

When a developer modifies the code, she receives feedback from her own tests, integration tests, performance tests, and finally from production. If changes are brought into production only once per quarter, several months can pass between the code modifications and the feedback from production. The same can hold true for acceptance or performance tests. If an error occurs then, the developer has to think back to what it was she had implemented months ago and what the problem might be.

With Continuous Delivery the feedback cycles become much faster: Every time the code passes through the pipeline the developer and his/her entire team receive feedback. Automated acceptance and capacity tests can be run after each change. This enables the developer and the development team to recognize and fix errors much more rapidly. The speed of feedback can be further increased by preferring fast tests, such as unit tests, and by first testing broadly and only afterwards testing deeply. This ensures from the start that all features function at least for easy cases—the so-called “happy path.” This makes spotting basic errors easier and faster. In addition, tests

that are known from experience to fail more often should be executed at the start.

Continuous Delivery is also in line with Lean thinking. Lean regards everything that is not paid for by the customer as waste. Any change to the code is waste until it is brought into production since only then will the customer be willing to pay for the modifications. Besides, Continuous Delivery implements shorter cycle times for faster feedback—another Lean concept.

Try and Experiment

Have a look at your current project:

- How much time passes between a code change and
  - feedback from a Continuous Integration server?
  - feedback from an acceptance test?
  - feedback from a performance/capacity test?
  - bringing it into production?

1.5 Generations and Structure of a Continuous Delivery Pipeline

As already mentioned, Continuous Delivery extends the approach of Continuous Integration to additional phases. Figure 1.3 offers an overview of the phases.

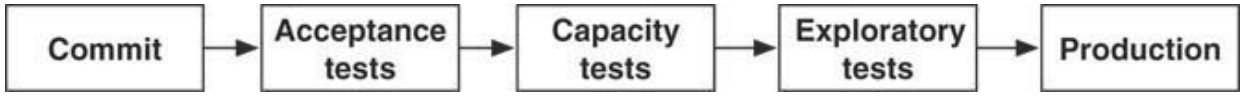

Figure 1.3 *Phases of a Continuous Delivery pipeline*

This section introduces the structure of a Continuous Delivery environment. It is oriented along Humble et al. (see footnote 1) and consists of the following phases:

- The commit phase comprises the activities that are typically covered by a Continuous Integration infrastructure such as the build process, unit

tests, and static code analysis. Chapter 3 discusses this part of the pipeline in detail.

- The next step is acceptance testing, which is the focus of Chapter 4, "Acceptance Tests." Strictly speaking, the topic is automated tests: Either the interactions with the GUI are automated to test the system or the requirements are described in natural language in a manner that allows them to be used as automated tests. From this phase on, if not before it is necessary to generate environments on which the applications can run. Therefore, Chapter 2, "Providing Infrastructure," deals with the question of how such environments can be generated.
- Capacity tests (Chapter 5, "Capacity Tests") ensure that the software can cope with the expected load. For this purpose an automated test should be used that unambiguously indicates whether the software is sufficiently fast. The relevant point is not only performance, but also scalability. Therefore, the test can also take place in an environment that does not correspond to the production environment. However, the environment has to be able to deliver reliable results about the behavior in production. Depending on the concrete use case other non-functional requirements, such as security, can also be tested in an automated fashion.
- During explorative tests (Chapter 6, "Exploratory Testing") the application is not examined based on a strict test plan. Instead, domain experts test the application with a focus on new features and unanticipated behaviors. Thus, even in Continuous Delivery not all tests have to be automated. In fact, by having a large number of automated tests, capacity is freed for explorative testing since routine tests do not have to be manually worked off anymore.
- The deployment into production (Chapter 7, "Deploy—The Rollout in Production") merely comprises the installation of the application in another environment and is therefore relatively low risk. There are different approaches to further minimize the risks associated with the introduction into production.
- During operation of the application, challenges arise—especially in the areas of monitoring and surveillance of log files. These challenges are discussed in Chapter 8, "Operations."

In principle, releases are promoted into the individual phases. It is conceivable that a release manages to reach the acceptance test phase and successfully passes the tests there, but shows too low a performance during the capacity tests. In such a case the release is never going to be promoted into the following phases, like explorative testing or production. In this manner, the software has to show that it fulfills increasing requirements before it finally goes into production.

Let us assume for example that the software contains an error in the logic. Such an error would at the latest be discovered during the acceptance tests, since those check the correct implementation of the application. As a consequence, the pipeline would be broken off (Figure 1.4). Additional tests are not needed anymore at this point.

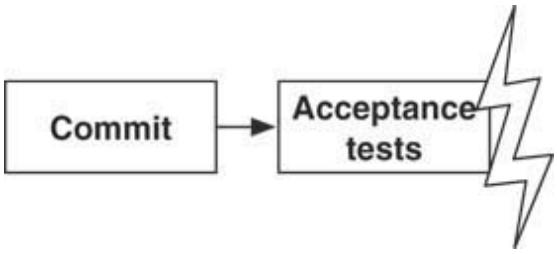

Figure 1.4 *Continuous Delivery pipeline stops at acceptance tests*

The developers will then fix the error, and the software is built anew. This time it also passes the acceptance test. However, there is still an error in a new function for which there is no automated acceptance test. This error can only be discovered during the explorative tests. Consequently, this time the pipeline is interrupted at the explorative tests, and the software does not go into production (Figure 1.5). This prevents testers wasting time with software that does not fulfill the requirements with regards to load handling, or that contains errors that can be detected by automated tests.

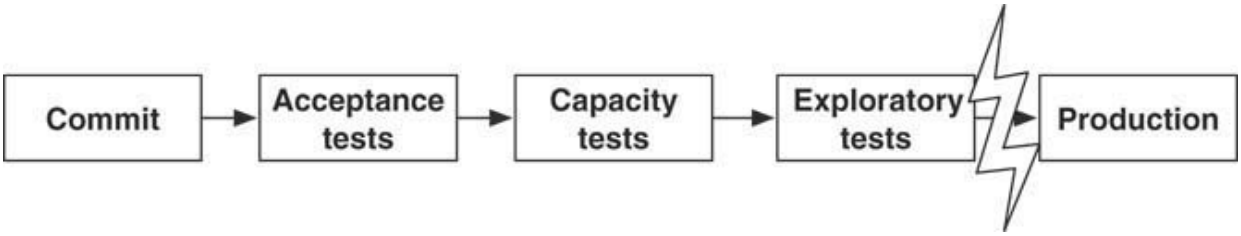

Figure 1.5 *Continuous Delivery pipeline stops at explorative tests*

In principle, several releases can be processed in the pipeline in parallel. Of course this requires that the pipeline support multiple releases in parallel —if the tests are running in fixed environments, this is not possible since the environment will be occupied by a test so that a parallel test of a second release cannot run at the same time.

However, it is very rare that releases are processed in parallel by Continuous Delivery. A project should have exactly one state in the version administration, which is then promoted through the pipeline. At the most it might happen that modifications to the software occur with such a speed that a new release is already sent into the pipeline before the previous release has left the pipeline. Maybe there are exceptions for hotfixes—but one objective of Continuous Delivery is just to treat all releases equally.

1.5.1 The Example

Throughout the book an example application is used—a customer registration inspired by the example of Big Money Online Commerce Inc. (see section P.2). This example is intentionally kept very simple concerning the domain logic. Essentially the first name, name, and email address of a customer are registered. The registrations are validated. The email address has to be syntactically correct, and there is only one registration allowed per email address. In addition, a registration can be searched based on the email address, and can be deleted.

Since the application is not very complex, it is relatively easy to understand so that the reader can concentrate on the different aspects of Continuous Delivery that are illustrated by the example application.

Technically the application is implemented with Java and the framework Spring Boot. This makes it possible to start the application, including web interface, without installing a web or application server. Thus the testing becomes easier since no infrastructure has to be installed. However, the application can also be run in an application or web server like Apache Tomcat if that is necessary. The data are stored in HSQLDB. This is an in-memory database that runs inside the Java process. This measure also reduces the technical complexity of the application.

The source code of the example can be downloaded at http://github.com/ewolff/user-registration-V2. An important note: The example code contains services that run under root rights and can be accessed via the net. This is not acceptable for production environments

because of the resulting security problems. However, the example code is only meant for experimenting. For that the easy structure of the examples is very useful.

1.6 Conclusion

Putting software into production is slow and risky. Optimizing this process has the potential to make software development overall more effective and efficient. Continuous Delivery might therefore be one of the best options to improve software projects.

Continuous Delivery aims at regular, reproducible processes to deliver software—much like Continuous Integration does to integrate all changes. While Continuous Delivery seems like a great option to decrease time to market it actually has much more to offer: It is an approach to minimizing risk in a software development project because it ensures that software can actually be deployed and run in production. So any project can gain some advantage—even if it is not in a very competitive market where time to market is not that important after all.

Endnotes

1. Jez Humble, David Farley: *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*, Addison-Wesley, 2010, ISBN 978-0-32160-191-9.
2. https://martinfowler.com/bliki/ContinuousDelivery.html
3. https://en.wikipedia.org/wiki/Continuous_delivery
4. Eric Ries. Kanban: *The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses*, Crown Business, 2010, ISBN 978-0-67092-160-7.
5. David J. Anderson: *Kanban: Successful Evolutionary Change for Your Technology Business*, Blue Hole Press, 2010, ISBN 978-0-98452-140-1.
6. https://www.sec.gov/litigation/admin/2013/34-70694.pdf
7. http://www.kitchensoap.com/2013/10/29/counterfactuals-knight-capital/

Chapter 2. Providing Infrastructure

2.1 Introduction

This chapter focuses on an essential foundation for Continuous Delivery: provisioning infrastructure. In the different phases of the Continuous Delivery pipeline, software has to be installed on computers to perform acceptance and capacity tests or explorative tests. This approach requires automation since a manual installation would be much too laborious. In addition, automation will decrease the risk associated with bringing a new release into production since it guarantees that the necessary environments are always installed identically.

Section 2.2 shows how simple installation scripts can support automation. However, such scripts are often not sufficient. Therefore, section 2.3 discusses a tool for infrastructure automation: Chef. Section 2.3.1 first describes the technical basics. Chef can either be used as a simple command line tool (Chef Solo, section 2.3.2) or as a client/server solution (section 2.3.4 and section 2.3.5).

Vagrant (section 2.4) is an excellent tool for installing virtual machines on developers' computers. Section 2.4.1 provides a concrete example for using Vagrant and Chef. Section 2.4.2 finishes this topic with a discussion of the usefulness of Vagrant.

Another alternative is described in section 2.5: Docker is not only a lightweight alternative for virtualization, but also uses a very simple approach for the installation of software in Docker containers. The section discusses the use of Docker with Vagrant (section 2.5.4), as well as the installation of Docker on servers with Docker Machine (section 2.5.5), more complex Docker setups (section 2.5.6) and the coordination of multiple Docker containers with Docker Compose (section 2.5.7). It is easiest to reproduce the installation of software when the installation on a server is never changed—in line with the concept of the "immutable server" (section 2.6).

The use of tools like Chef and Docker results in fundamental differences with regards to how infrastructure is handled. Section 2.7 explains the effects of "infrastructure as code."

Section 2.8 shows how Platform as a Service Clouds (PaaS Clouds) can be utilized for Continuous Delivery.

Finally, section 2.9 focuses on the specific challenges with regards to data and databases. Databases are especially difficult to install and to update since they contain a lot of data which might have to be migrated depending on the circumstances. Moreover, it is not easy to generate and provide suitable test data sets for the application. The chapter then finishes with a summary and conclusion in section 2.10.

2.1.1 Infrastructure Automation: An Example

Section P.2 described the scenario of Big Money Online Commerce Inc., which did not use Continuous Delivery. However, Big Money Online Commerce Inc. did learn from its errors. One measure the company introduced was infrastructure automation. This makes it substantially faster to generate environments with the necessary software and renders the process more reproducible. In fact, the infrastructure automation solved the following problems for the team:

- Generating the testing environment had always been very laborious. Since this process is now automated, the effort to generate such an environment has become much smaller. This was a prerequisite for generating multiple testing environments which now allow for more comprehensive tests.
- In the scenario that was presented at the start of the book an error had happened in production because the production environment had not been built in the same way as the testing environment. Since the infrastructure is now automated, it can be reproduced exactly—this is true for the testing as well as for the production environment. Therefore, this error source is now abolished, and such errors indeed did not occur again.
- In addition, nightly releases into production are also easier: It is just an automated process that has to be initiated. Manual errors are impossible. And most important: The release process has already been used for the generation of the other environments. Consequently, it has been thoroughly tested.

To do so the team had to use technologies which are also able to generate complex environments—this is exactly the topic of this chapter.

2.2 Installation Scripts

System configuration and software installation have been automated for a long time already: Windows software uses, for instance, Windows installers to do so. However, the situation is different when the software is written in-house and has to be brought into production: In this case configuration and installation often have to be performed manually. In some cases the necessary steps are described in a handbook. These steps have to be manually executed. Of course, even if checklists are used for verification, it is rather difficult to follow complex guidelines absolutely correctly so that this process is quite error prone and hardly reproducible.

2.2.1 Problems of Classical Installation Scripts

Some projects have turned to automating software installation with the help of scripts. The scripts implement the necessary sequence of steps to create the required files and the correct configurations. For the example application in this book—the user registration—it is, for instance, first necessary to install a Tomcat server. If Linux is used, a possible installation script could look like Listing 2.1: `apt-get` is a service program for the Ubuntu Linux distribution. Initially the script updates the index of available packages and installs all available updates with `apt-get`. Then it installs OpenJDK and Tomcat. Finally the application is copied into the directory in which Tomcat expects the executable applications. Prior to running the script the application is copied onto the server. It is also possible to download the application via HTTP onto the server. This allows installation of the application directly out of a repository.

Listing 2.1 *Installation script for user registration*


```
#!/bin/sh
apt-get
apt-get dist-upgrade -y
apt-get install -y openjdk-8-jre-headless
apt-get install -y tomcat7
cp /vagrant/user-registration.war /var/lib/tomcat7/webapps/
```

At first this approach looks convincing due to its simplicity. However, many of the typical challenges are not solved:

- Changes to the port that the Tomcat server uses or to the memory configuration for Tomcat are not possible with this approach. They would require that Tomcat configuration files be modified.
- In the case of a new deployment, such scripts might cause problems. In the given example the application will be copied again into the directory of Tomcat. This will cause Tomcat to restart the application, which will lead to a short downtime of the application. In the case of a complex application this could also result in users being logged out of the application. Of course, the script could be modified such that the file is only overwritten if it is not yet present or has been changed. However, this would make the script more complicated and would also have to be tested.
- When the server configuration is modified, this script is not able to return the server back to the correct state in all cases. When, for example, one of the Tomcat configurations is changed, this installation will not be able to undo this modification.

Apart from these problems, which are associated even with such a simple script, there are additional fundamental challenges:

- It is laborious to install a real application including all components. Of course, it is even more laborious to automate this process. Therefore, many attempts to automate never reach completion since the costs for a full automation are too high. Consequently, a handbook and manual interventions are still required. It can be very difficult to perform the manual steps and interventions in the right place—this often requires comprehensive knowledge about the script. The tools presented in this chapter facilitate the implementation of such automation and scripts and therefore make it easier to really achieve full automation.
- When an installation crashes, it has to be restarted. In such a scenario the system is in a state where some parts have already been installed. Just to start the installation again can create problems. The script usually expects to find the system in a state without any installed software. If the script then tries to create folders and files, these

operations can fail and might thereby interrupt the entire installation. To fix such problems the installation script would have to be able to deal with these special cases and would have to be tested in different environments.

This problem is also the reason why updates to a new software version are problematic. In such a case there is already an old version of the software installed on the system that has to be updated. This means that files might already be present and have to be overwritten. The script has to implement the necessary logic for this. In addition, superfluous elements that are not required anymore for the new version have to be deleted. If this does not happen, problems can arise with the new installation because old values might still be read out and used. However, it is very laborious to cover and automate all update paths that occur in practice. On the other hand, one can circumvent this problem by just building the server from scratch for each new release.

In the end such simple scripts are often sufficient. But when changes to the configuration are necessary or when, in addition to an installation, an update of an old version also has to be supported, the scripts quickly get very complicated. Thus it can be sufficient to implement installation scripts if software just has to be completely newly installed. Docker (section 2.5) is well suited for building servers completely anew for each new software release and therefore often uses installation scripts.

Try and Experiment

Section 2.4 will still deal in detail with Vagrant, a tool for the management of virtual machines. To gain experience with the installation script we will use this tool here. This will allow us to very easily start a virtual machine and the installation script.

First of all, Vagrant has to be installed:

1. Start by installing a virtualization solution—for example, Virtual Box.[1]
2. Download Vagrant[2] and install it by following the instructions.
3. Install Git.[3]
4. Fetch the GitHub repository with the command:

```
git clone https://github.com/ewolff/user-registration-V2.git
```

Now you can start a machine in the sub-directory `user-registration/shell` and have it provisioned via `vagrant up`. During the provisioning the installation script is executed.

The installation script can also be executed without starting the machine again by using `vagrant provision`.

- Is it possible to access the application during provisioning?
- What happens with registered users if the provisioning is repeated? Note that the application uses an in-memory database and does not store the data about the users on the hard disk.

2.3 Chef

Installation scripts describe which steps are necessary for the installation of software. Another approach is to describe how a system is supposed to look after the software installation instead of describing which actions are necessary to reach this state.

Let us assume for example that a configuration file is required to run the software. In the classical approach the configuration file is generated. If the file already exists, this can cause problems; for one thing, it is not possible to overwrite the file. If the installation script overwrites the application it

can cause a restart of the application—including data loss and a short downtime of the application. Therefore, the overwriting has to be implemented in a proper manner. Problems can also arise if the rights for accessing the file have not been set correctly, so that the file is present but cannot be read. Of course, the required directories have to exist, too. Besides, overwriting a file in a running system can lead to additional problems. For example, applications that are accessing the file can run into trouble during write access to the file.

An alternative would be to describe which content the file is supposed to have and which access rights should be set. In that case the desired state could be compared with the current state during the installation and the file could be adjusted accordingly. Such an approach has a number of advantages:

- The installation can be repeated as often as required—the result should always be the same. Such an installation procedure is called idempotent.
- Updates to a new version are relatively easy since it is not assumed that the system has to be built entirely anew. While a system where an older software version is installed is a special case, it can nevertheless be dealt with by the scripts. In that case the file might already be present, but is then overwritten with a new version if necessary.
- The changes to the system can be documented. Each change to a file or installation of specific software packages can be recorded in a log file. When, for instance, a file with new content is written, the old content can be automatically backed up so that the change is traceable. A regular installation script again would have to implement special cases to do so.
- Finally, if the file is already present and the access rights are set correctly, there are no changes necessary at all. This can profoundly speed up the process.
- Often a process has to be restarted to make sure that the new file is really read. The restart can now likewise be ensured—this, however, is only necessary if the file content really changed.

Ideally, the system has to be described completely—that is, including the information about which version and which updates of the operating system have to be installed. This is the only way to ensure that the installation is really completely repeatable.

Chef, which will be introduced in this section, belongs to the software systems following this approach.

2.3.1 Chef versus Puppet

Another system with a similar approach like Chef is Puppet.[4] The Continuous Integration infrastructure from the next chapter is generated with a Puppet script. However, the implementation differs in important points: Puppet pursues a declarative approach with a Ruby DSL (domain-specific language) and JSON data structures. The user defines dependencies, and Puppet installs the software based on a plan that it constructs based on the dependencies and components. While Chef also uses a Ruby DSL, the user writes a kind of installation script instead of the declarations used by Puppet. Since a domain-specific language is used in both cases, it is not strictly required that one be proficient in Ruby in order to use Chef or Puppet. However, Chef and Puppet can easily be supplemented with all the power of Ruby. This approach is also easier to understand for developers.

Another advantage of both approaches is that the technologies are relatively widespread. Therefore, installation scripts exist for many typical software packages, which can be used for your own system installation. However, these scripts have to be approached with some caution: Most of the time they have to be adapted to your own needs for your own infrastructure, because, for instance, additional modifications in the configuration files are necessary in the context of the project. In some cases the project can require installation of plug-ins for software, and the installation is not contained in the generic script. It can also be necessary to adjust specific settings—for example, the network port the process uses—but these options are not part of the regular scripts. In addition, not all variants of operating systems are necessarily supported. Linux is not just Linux—different distributions store configuration files in different places and software packages have different names. Even if the scripts seem to take different Linux variants into consideration, there might be problems in

real life because the scripts have not been tested on each version of the operating system.

In addition to different Linux derivatives, Puppet and Chef also support Windows and Mac OS X. Therefore, the approaches described here are also adaptable to these platforms—however, there are of course differences when it comes to details. For example, neither Windows nor Mac OS X comprises a package manager as part of the basic installation.

Chef and Puppet are open source projects under an Apache 2.0 license. This is a very liberal license so that from this perspective nothing argues against their use. In addition there are commercial versions available from the companies Chef and Puppet Labs. Therefore, it is either possible to use the products for free or to buy greater security by paying for commercial support.

2.3.2 Other Alternatives

In addition to Puppet and Chef there are many other tools. Some examples are:

- Ansible[5] focuses on having its own syntax for installation scripts, which it calls playbooks. YAML files are used for defining the servers. Servers are installed remotely via SSH. This approach is simple and secure via SSH—and, most of all, it hardly requires any additional software on the systems which are to be installed. However, it can be necessary to install Python libraries on the machines because Ansible is implemented in Python. It supports, in addition to Linux, Windows, and Mac OS X. There is a document, "how to get started with Ansible"[6] that is very useful. An example of the installation of Tomcat[7] is provided.
- Salt—or more specifically the SaltStack[8]—are likewise implemented in Python. Salt is based on a Master Server and Minions which run as Daemons on the administered systems. These components communicate via ZeroMQ—a messaging system. This allows Salt to communicate quickly and effectively and therefore to scale on a large number of systems. Linux, Mac OS X, and Windows are supported. There is also a tutorial.[9]

The implementation of Ansible started in 2012, and Salt has been on the market since 2011. So these technologies are substantially newer than Chef (2009) and Puppet (2005). Consequently, they do not have such a large community yet, and there is not as much experience using them in practice. On the other hand, they solve a number of problems which are encountered in the use of the older technologies.

2.3.3 Technical Foundations

In general Chef can be used in three different ways:

- *Chef Solo* is the simplest variant. Here, Chef is used as normal command line tool—a kind of universal installation script. However, the entire configuration has to be present on the computer. This can be sensible for generating systems for developers who get by without a large infrastructure. For example, Vagrant (section 2.4) uses this approach. With the same approach servers can be installed as well. This requires, for example, that the configuration be read from a version control like Git and then the installation process is triggered. This allows the installation of large environments without the need for a central server. However, these configurations have to be especially secured since they can, for instance, contain the passwords of the production databases. Likewise the rollout of changes to the different servers should be automated. This requires Chef Solo to be executed again on the servers.
- *Chef Server* configures and manages a number of Chef clients. The client itself possesses only a very small core. This approach is especially useful when several computers have to be installed. Besides, this approach makes it possible for the server to keep track of all computers. In this way, a suitable entry for each computer can be generated on a monitoring system. In addition, it is possible to execute the requests via the computers and the installed software. This allows, for instance, automatic supplementing of the configuration of a load balancer when a new web server has been installed.
- *Chef Zero* is a variant of the Chef server which runs in-memory. This makes it small and very fast. Chef Zero is especially useful for tests where a Chef server would be too slow or the setup too laborious.

A commercial variant is *Enterprise Chef*. This version has a number of additional features—for instance, support for tenants or authentication with LDAP or Active Directory. This version is also available as a hosted system. In that case it is the company Chef Inc. that runs the Chef server. This is a useful approach when an infrastructure is supposed to be operated in a Cloud environment like Amazon Web Services. In that case it is not necessary to care anymore about how the Chef server is installed, how backups are made, or how sufficient availability can be guaranteed. These tasks are taken care of by Chef Inc.

Basic Chef Terms

Three terms are central for the configuration of systems with the help of Chef (Figure 2.1):

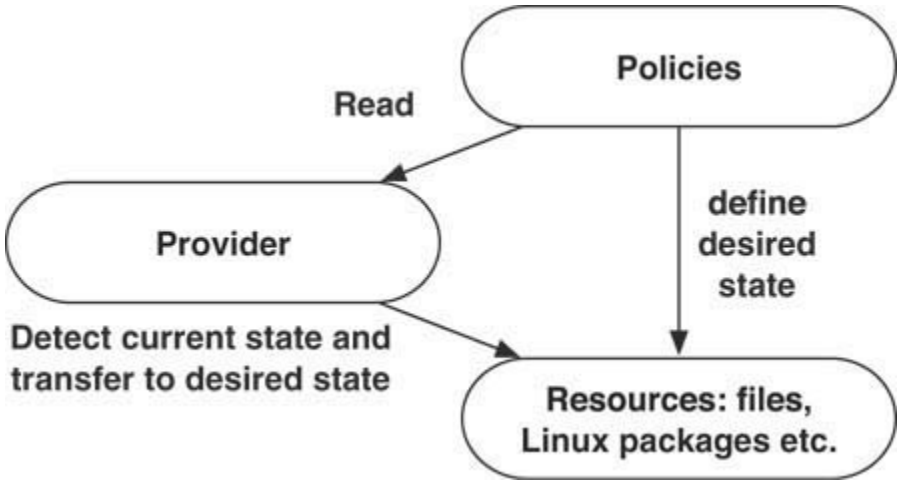

Figure 2.1 *Resources, policies, and providers*

- *Resources* comprise everything that can be configured or installed: for example, files or Linux packages. Chef supports a multitude of resources, such as code repositories, network adaptors, and file systems.
- *Policies* define what resources should look like. For example, a policy can state that the user "wolff" should be there or that the package "tomcat" should be installed. Developing automation with Chef consists mainly of generating such policies.
- *Providers* are responsible for determining the current state of a resource and to modify it so that it is in line with a policy. The built-in

providers of Chef support typical operating system resources, such as files.

Thus, the policies define what resources should look like after the Chef run. The necessary steps to get there are executed by the providers. This ensures the mentioned idempotency: Independently of how often the Chef run is repeated the final result is always the same.

Chefs, Cooking Books, and Recipes

The Chef DSL is written in Ruby. Therefore Chef can be extended with Ruby. The DSL makes it possible to configure simple systems also without knowing Ruby. The users do not notice at all that they are in reality programming and not configuring. On the other hand, power users can use Ruby to extend the system. For the configuration the user writes so-called recipes—which is quite consistent with the solution being called "Chef."

Listing 2.2 shows part of a recipe—in this case for Tomcat. The entire project can be found at https://github.com/ewolff/user-registration-V2. First of all the recipe determines to which user or group the file should belong and which rights should be set for the file. If the file is changed, the Tomcat service is supposed to be restarted. This service is an additional Chef resource that is defined at another place within the file.

Listing 2.2 *Part of the Tomcat recipe*

Click here to view code image

```
template "/etc/tomcat7/server.xml" do
  owner "root"
  group "root"
  mode "0644"
  notifies :restart, resources(:service => "tomcat")
  source "server.xml.erb"
end
```

Afterwards a template is referenced that serves to generate an XML file with a Tomcat configuration. By using a template, values like network ports, which are specific for a certain server, can be inserted at the right positions in the file. Chef will not simply overwrite the file with a newly generated file. Instead Chef will first check whether the file is consistent with the template. Only when this is not the case will the file be replaced.

The old version of the file will be archived to allow tracing of which change was performed when. Subsequently, checks are performed to determine whether the owner of the file and the access rights are correct. Only when a change of the files is necessary is the Tomcat service started anew.

Obviously one recipe on its own makes little sense. Different additional elements are required, such as the referenced templates. Therefore, a recipe is usually stored in a directory structure: a so-called cookbook. It consists of a directory with different sub-directories (Figure 2.2). Typically, a cookbook contains the following components:

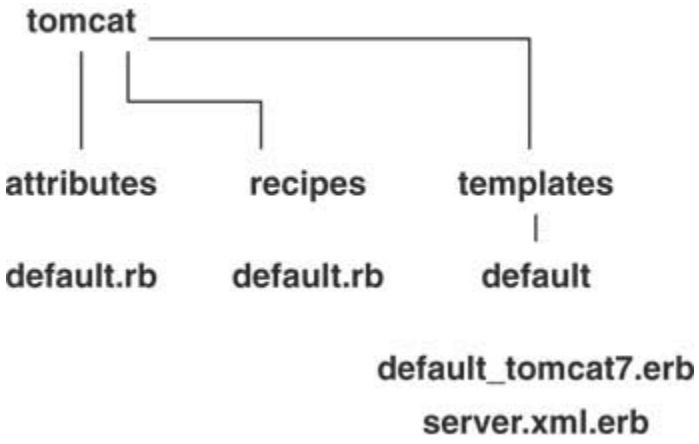

Figure 2.2 *Directories of a Chef recipe*

- In the directory `recipes` the discussed recipes are found; they define the policies for resources. Usually, there is a recipe in the file `default.rb`. For complex installations additional recipes can be defined, which can be stored in other files.
- Within the recipes values can be adjusted that are specific for a certain server. This requires that attributes be defined in the recipes. The possible attributes and their default values are defined in the directory `attributes`. As will be discussed later, these attributes can be overwritten outside of the recipes so that the recipes can be used in different contexts. Consistently, “Convention over Configuration” is implemented. This approach means that sensible defaults are used if no specific configuration is provided.
- Additional files can be stored in the directory `files`. These are files that can be copied onto the servers without additional changes. However, it

is often more sensible to load files; for instance, from a repository server.

- Finally the directory `templates` contains the necessary templates, for instance for configuration files. There the file `server.xml.erb` would be stored which was referenced in Listing 2.2.

For the management of complex cookbooks and of the dependencies of the cookbooks and recipes Berkshelf[10] is useful.

There are a number of additional possible components of cookbooks, but the ones described above usually suffice. Let us take as an example the already mentioned Tomcat cookbook [8]. Listing 2.2 references the template `server.xml.erb`, which can be found in the directory `templates/default`. If other templates are supposed to be used for computers with certain host names or operating systems, these are stored in special templates in sub-directories that derive their names from the host name or the operating system.

Listing 2.3 shows a part of the template `server.xml.erb`. As you can see, there are terms now and then that are bracketed by `<%=` and `%>`. These terms are used to insert values into the templates. At other places some values are fixed. For example, the timeout is set to 20000. Thus this value cannot be configured from the outside. The template would need to be changed at this location to add a configurable value. This is a good example for a scenario where a cookbook provides only limited flexibility that has to be extended if necessary. Most cookbooks are hosted on websites like GitHub.[11] These websites make it very easy for users to provide modifications to the original developers to be incorporated into their projects. This allows the development of cookbooks collaboratively.

Listing 2.3 *Part of the Tomcat recipe*

Click here to view code image

```
...
<Connector port="<%= node["tomcat"]["port"] %>"
protocol="HTTP/1.1"
  connectionTimeout="20000"
  URIEncoding="UTF-8"
  redirectPort="<%= node["tomcat"]["ssl_port"] %>" />
<Connector executor="tomcatThreadPool"
  port="<%= node["tomcat"]["port"] %>" protocol="HTTP/1.1"
```

```
    connectionTimeout="20000"
    redirectPort="<%= node["tomcat"]["ssl_port"] %>" />
...
```

The cookbook can also contain defaults for values, which can differ in every environment. For this purpose a file, `default.rb`, can be created in the sub-directory `attributes`. Listing 2.4 shows an excerpt. As you can see, sensible values are set.

Listing 2.4 *Excerpt from* `default.rb` *that defines default values*

Click here to view code image

```
default["tomcat"]["port"] = 8080
default["tomcat"]["ssl_port"] = 8443
default["tomcat"]["ajp_port"] = 8009
default["tomcat"]["java_options"] = "-Xmx128M -
Djava.awt.headless=true"
default["tomcat"]["use_security_manager"] = false

set["tomcat"]["user"] = "tomcat"
set["tomcat"]["group"] = "tomcat"
```

Of course, it is possible to write cookbooks on your own. However, in general it is not necessary to start from scratch. There are different collections of cookbooks[12] on the Internet. Therefore, it is worthwhile to first check out these collections before setting to work. As mentioned, the existing cookbooks are often adapted to the specific demands of the project. Likewise existing cookbooks can serve as examples and inspiration for how to create your own cookbooks.

When using cookbooks from the Internet for your own project, adaptations are normally necessary to obtain good results. Often they have to be supplemented to support a certain operating system or the configuration has to be rendered more flexible in certain places—for example, by the introduction of new attributes. Often cookbooks for a special use-case are very simple and can be written rapidly while the already prepared cookbooks contain general solutions and therefore are rather complex—and in the end still have to be extended for the special use-case. Thus in the end, writing your own cookbook can still be the more straightforward approach.

Roles

To really install a server you must define which recipes are supposed to be applied on the server. In addition, the attributes within the recipes have to be adjusted. Theoretically, it would be possible to hand over the list of recipes and the values for the attributes to Chef upon each installation. However, this does not make a lot of sense. In the end it is always the same kind of servers that have to be installed in each phase of the Continuous Delivery pipeline. Besides, there can be several identical servers; for instance, when multiple web servers have to be provided behind a load balancer.

Therefore, Chef uses the concept of "roles": A role defines which recipes are executed and which values the attributes are supposed to have. Each type of server requires the definition of a role. Subsequently, it is very easy to install an arbitrary number of such servers. In addition, the Chef server can be used to execute queries, such as how many servers are there with a certain role, and which servers they are. In this manner, a load balancer whose configuration files contain, for example, all web servers can be installed.

The information for a Chef role is stored in a JSON file; alternatively, a Ruby DSL can be used. An example is shown in Listing 2.5. For a role definition the `json_class` and the `chef_type` have to be set to the values in the listing. Afterwards information follows that specifies which cookbooks are executed and which values the attributes should have. Initially the role is denoted as "tomcatserver," and this role is described. Then the individual attributes of the cookbook tomcat and `webapp` are adjusted. Finally, the `run_list` defines which recipes are to be executed.

Listing 2.5 *JSON configuration of a Chef role*

Click here to view code image

```
{
  "json_class": "Chef::Role",
  "chef_type": "role",
  "name": "tomcatserver",
  "description": "Install Tomcat and a web application inside
Tomcat",
  "default_attributes": {
    "webapp": {
      "webapp": "demo.war"
    },
    "tomcat": {
```

```
        "port" : 8080
      }
    },
    "run_list": [
      "recipe[apt]",
      "recipe[webapp]"
    ]

}
```

Each server—denoted as a node in the terminology of Chef—can have one or multiple roles. Nodes can also overwrite attributes to further adjust each server individually.

2.3.4 Chef Solo

The easiest approach for using Chef is Chef Solo. Here, the provisioning of software on the system is triggered by the command line. This means that the configurations and installation scripts have to be transferred onto the system. Therefore, the use of Chef Solo makes sense for test scenarios and local developer machines where this limitation is not critical. In the case of a distributed installation on multiple computers, version control tools and other tools for the transfer of files can also be used to support these scenarios. The infrastructure for such an approach is often easier than the installation of a central Chef server. In addition, this strategy helps to avoid a central bottleneck.

Try and Experiment

This task is meant to help the reader to gain experience with Chef Solo. The objective is to install a simple Java application completely with a Tomcat web server and JDK.

1. This task is best solved on a virtual machine. Therefore, install software like VirtualBox.[13]
2. Now install a VM with Ubuntu 15.04. The CD image can be downloaded at http://releases.ubuntu.com/15.04/. The server version is sufficient. Please make sure to install version 15.04—with other Ubuntu versions there can be problems with the scripts.
3. Install only a simple server. You do not need any of the special packages that are offered during the installation. However, it might be easier to install an SSH server and to perform the subsequent steps per `ssh`—in that case you can use, for instance, copy/paste from other windows.
4. Start the VM with the installed operating system.
5. Update the software. The following commands are used for updating: `sudo apt-get update` and `sudo apt-get upgrade`.

Now you have a VM with a simple Ubuntu installation without special software. The next step will be to install Chef.

1. Install Chef. During the installation a server is asked for—however, it is not necessary to answer this question. The command for installing is: `sudo apt-get install chef`.
2. In addition, install the version control Git with: `sudo apt-get install git-core`.
3. Fetch the GitHub repository using the command: `git clone https://github.com/ewolff/user-registration-V2.git`.
4. Adjust `solo.rb` in the directory `chef`: In the first line the variable root is defined. It has to contain the directory in which the Git repository is contained—for example: `root = '/home/ubuntu/user-registration-V2/chef/'`.

Advice: Nano is an editor that can be quite useful and is also already installed on an Ubuntu system. You can start it with the command `nano`.

1. Now all required files are together, and Chef Solo can be started with the command: `sudo chef-solo -j node.json -c solo.rb`.
2. Now Tomcat should be available. With `curl localhost:8080` this can be tested, and with `curl localhost:8080/demo/` the application can be started.

However, this will only result in HTML code. In principle, the application can also be opened in the browser, but this will of course only function if a network connection to the VM is possible.

At the end it is worthwhile to check out the configuration files `node.json` and `solo.rb`. In addition, it is of course interesting to have a look at the used Chef role and the different recipes.

For those who are motivated for additional experimenting:

- Let the installation run a second time—what happens?
- Change the port on which Tomcat runs. There are different possibilities: either change the default value in the cookbook, or have a look at the file where the role is defined.
- Of course, you can also try out letting the software run on another Ubuntu or Linux version. However, depending on the system, this requires comprehensive changes.
- It is also possible to expand the Chef scripts and in this manner to install additional software. To do so you can “borrow” cookbooks from [2] and integrate them into the configuration.

2.3.5 Chef Solo: Conclusion

This task should have illustrated how systems can be configured with Chef. However, it should also have become apparent that this approach is laborious. First, Chef needs to be installed. Afterwards, individual files have to be adjusted—exactly the type of work that should in principle be avoided by using Chef.

In addition, it would now of course be possible to fetch the current version of the configuration from a Git repository at regular intervals and

thereby roll out changes; however, with Chef Solo nobody keeps track of the installed servers. But such an overview would certainly be very helpful.

2.3.6 Knife and Chef Server

With Chef Server, information pertaining to roles or cookbooks is kept centrally on a server. Such a Chef server can of course be installed.[14] This is necessary if you want to keep your cookbooks and roles exclusively in your own computing center. In this case backups and disaster recovery are mandatory since roles and recipes should not get lost. However, a high availability does not necessarily have to be ensured since a server failure only means that no new computers can be installed and that roles and recipes cannot be modified.

The alternative is to use Chef Enterprise, where the Chef server is supplied by Chef Inc. and can be used. Here, there is no additional expenditure for operation.

Knife represents a kind of "remote control" for a Chef server. It can administer roles, cookbooks, and so on on the Chef server. However, a lot more is possible: Knife can interact with a virtualization solution. Knife can also install a Chef client on a computer. When this setup is properly configured, a new computer can be started and the software installed on the computer without any manual intervention.

Try and Experiment

Let us have a look at Chef Server and Knife. By using the Amazon EC2 Cloud infrastructure and the Hosted Enterprise Chef Server we can get an impression of the possibilities offered by Chef Server and Knife without large investments in infrastructure.

This example can be run on any computer. In contrast to the previous example it does not require the use of a virtual machine. However, you can of course reuse your virtual machine to install the necessary software separately.

1. Install Git (see http://git-scm.com/book/en/Getting-Started-Installing-Git).
2. As before, we will employ the example of the Tomcat server. Therefore, you have to check out the project with `git clone https://github.com/ewolff/user-registration-V2.git`
3. Then you have to install Knife, which is part of the Chef Development Kit. Instructions for the installation can be found at https://downloads.chef.io/chef-dk/.
4. Finally, the Knife EC2 plug-in has to be installed—see https://github.com/chef/knife-ec2#installation.

Now we have assembled all software required for this example. The next step is to open an account at Amazon and to transfer the information into the configuration:

1. You need an account at Amazon which can be set up at Amazon at http://aws.amazon.com/, which provides new users with a considerable amount of free usage of virtual machines. However, this comprises only micro instances, the least powerful virtual computers in the Amazon Cloud. Nevertheless, it makes sense for tests to use micro instances, if possible, since they are very cheap and still normally sufficiently powerful for tests.
2. At https://console.aws.amazon.com/iam/home?#security_credential an access key can be generated. It consists of an access key ID and a secret key. The environment variables `AWS_ACCESS_KEY_ID` and

`AWS_SECRET_ACCESS_KEY`, respectively, have to be set to these values in order to be used by Knife later on.

3. Create a security group at https://console.aws.amazon.com/ec2/home?region=eu-west-1#s=SecurityGroups. This serves to configure the firewall of the virtual machine. The security group should open up ports 22 (SSH) and 8080/8081 (HTTP, with a "Custom TCP Rule"). Please ensure that the security group is not assigned to any VPC. Enter the name of the security group in the line `knife[:groups]` in `knife.rb` in the directory `user-registration-V2/chef/.chef`.
4. At https://console.aws.amazon.com/ec2/v2/home?region=eu-west-1#KeyPairs you can generate a key pair. The result is a `.pem` file. Save it in the `.chef` directory and enter the file name, without the `.pem` extension, in the line `knife[:aws_ssh_keyname]` in `knife.rb`. Pay attention to the correct file rights (`chmod 400 *.pem`).

Costs at Amazon

Having running instances at Amazon creates costs. Therefore, it is important to have a look at https://console.aws.amazon.com/ec2/v2/home?region=eu-west-1 to determine whether servers are still running and to terminate them if they are. To do so the server has to be selected and "terminate" has to be chosen as the action. Amazon has a worldwide Cloud offer that is divided into different regions. For each of these regions there are multiple availability zones—in practice, these are separate computing centers. For this task we will use the region EU-West-1, which is located in Europe. The console displays each region separately. Therefore, it is important to ensure that the correct region is in the console. However, the given links above lead directly to the region EU-West-1.

With the security group we have taken care that the proper ports on the machine and on the firewall are open. The SSH key is used by Knife to log into the computer and to install Chef there. The other settings in `knife.rb` make sure that the machine runs in the region EU-West-1—in a computing center in Ireland. A micro instance will be used which does not have a lot of power, but is at least not very costly. Finally, the AMI image ensures that the instance boots with Ubuntu 15.04 as the operating system.

Now Knife can start a virtual machine with Ubuntu. A hosted Chef server is responsible for the cookbooks. This server has to be set up next:

1. We need a Hosted Enterprise Chef account, which can be obtained at https://manage.chef.io/. Hosted Enterprise Chef is free of charge up to a certain number of computers.
2. At https://manage.chef.io/organizations/ you can create an organization.
3. Download the validation key for the organization and save it in the `.chef` directory.
4. In `knife.rb` the file name has to be set in `validation_key`.
5. Now the organization name has to be entered in `knife.rb`—at the definition of `validation_client_name` and `chef_server_url`.
6. Click on the link "Users." There you can have a key created for your account—a file with the extension `.pem`. Download the key, save it in the directory `.chef`, and change the line with `client_key` in `knife.rb` to the name of this file.
7. The name of this file—without extension—has to be set as the value for the `node_name`.

Now everything is configured. When we now simply execute `knife node list` in the directory with the sub-directory `.chef`, we should obtain a sensible response—namely that there are no nodes yet, since we have not configured any servers yet.

As the next step we will feed the necessary information into the Chef server:

1. Enter the command `knife cookbook upload -a` into the directory `user-registration/chef`. This will cause all cookbooks to be uploaded onto the server. Uploading the web applications can take some time.
2. Do the same with the role `tomcatserver` via `knife role from file roles/tomcatserver.json`.
3. Now you can use `knife ec2 server create -r 'role[tomcatserver]' -i .chef/<Amazon PEM>.pem -r 'role[tomcatserver]'` to start a new server onto which Chef and Tomcat will be installed. The file name,

which is passed in as an argument, is the file belonging to your key pair at Amazon.

Using a single command we have now started a server that is also accessible from the internet. When the server started, the public DNS name was stated under which the server is now accessible. A curl can be used to contact the application, for instance via `curl http://:8080/demo/`.

Let us have a look at the newly started node by using Knife—this can be done with `knife node list`. This command returns all running nodes. `knife node show <instance-id>` shows more detailed information about the nodes. These instances should now be deleted with `knife ec2 server delete <instance-id> --purge`—otherwise costs are incurred. If there were difficulties during the setup of the machines, it might happen that they do not appear in the list of nodes—in such a case please use the EC2 console mentioned in the boxed text to also delete these servers.

One more comment: The virtual machines that are used in this example are very slow. Real machines for production are considerably faster. For readers who feel like experimenting: It is possible to generate AMI images of running instances in the EC2 console, and also of the instances that we started with Knife. In that case the entire content of the hard disk is saved. Via this new AMI machines can be started. They contain the entire software that was previously installed, but without an installation process—that is, practically immediately.

Additional ideas for experimenting:

- It is also possible to view and monitor the running nodes in the AWS console at https://console.aws.amazon.com/, in the EC2 dashboard of the EU-West-1 region.
- The software could be started with another version of Ubuntu. The names of the AMIs that have to be entered in `knife.rb` can be found at http://uec-images.ubuntu.com/.
- The machines can also be started in another region. To do so another image has to be used—a list can be found at http://uec-images.ubuntu.com/vivid/current/. This image and the region have to be entered in `knife.rb`. In addition, a new SSH key pair has to be generated at Amazon for each region.

2.3.7 Chef Server: Conclusion

An approach based on Knife and Chef Server is ideal for the installation of complex environments comprising many servers. A new server can be started with a single command. In addition, the existing types of servers, recipes and roles are tracked in the background. Thus an entire assembly of machines can be generated simply via a Chef server, the required Chef configurations, and Knife. This also allows rapid creation of environments for software testing.

Try and Experiment

This chapter was mainly meant to provide an introduction into the fundamental technologies for script-based tools. Possible extensions of these topics are:

- An alternative to the previously presented Chef is Puppet. At https://puppetlabs.com/download-learning-vm a virtual machine is provided to get to know Puppet.
- Of course, it is also valuable to get to know Chef even better—for this purpose a comprehensive introduction can be found at https://learn.chef.io/.
- Finally, Chef recipes are provided.[15] It is worthwhile to have a closer look at them. How can you automate an existing application with these tools? Which cookbooks are missing? Are the cookbooks flexible enough?

2.4 Vagrant

To use Chef there has to be a suitable computer which has a minimal operating system and a client for Chef installed. This is exactly what Knife offers (see section 2.3.4); however, then a Chef server has to be installed or Hosted Chef has to be used. In addition, an appropriate infrastructure for virtualization has to be provided or a Cloud has to be used.

Such an approach is too laborious for cases where developers just quickly want to test something. For such scenarios Vagrant[16] provides an ideal solution: It uses virtualization for provisioning a computer. It supports

Virtual Box,[17] which is under the GPL open source license and can be used for free. In addition, Vagrant's commercial version supports VMware Workstation and VMware Fusion as virtualization on Windows/Linux and Mac OS X, respectively. Vagrant is modular and can be extended via plug-ins[18]—among them numerous providers for other virtualization solutions. In this way, Vagrant can be enlarged and turned into a general "remote control" for virtualization and provisioning—similar to Knife (section 2.3.4). In contrast to Knife, Vagrant is not limited to Chef for the provisioning.

The virtual machine contains, in addition to the operating system, the respective parts of Chef or Puppet that are necessary for the installation of additional software. An SSH server with an authentication via certificates is also installed. This is done via a certificate that is known to Vagrant. This allows Vagrant to log into the virtual machine and to trigger actions there. In addition, parts of the host file system can be blended into the file system of the virtual machine. As already mentioned, this requires the installation of extensions on the virtual machine. Since the exact configuration is complex, it does not make much sense to do the configuration by yourself. Either one of the ready-made VMs[19] should be used or tools like veewee[20] or Packer[21] should be employed for generating such machines. Packer is the more modern solution, which also offers more features. These tools support different operating systems for the VM in addition to the different Linux distributions—Windows, for instance.

Vagrant can then do the following (Figure 2.3):

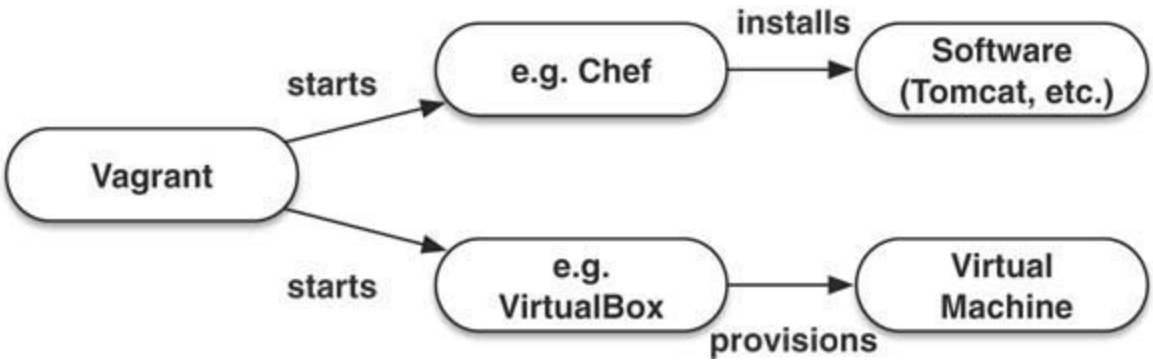

Figure 2.3 *Vagrant starts a virtual machine using e.g. Virtualbox and installs software on it using, for example, Chef*

- Import an empty virtual machine.

- Configure a virtual machine in such a way that it can use files of the host. This allows the virtual machine, for instance, to access Chef recipes that are stored on the host.
- Now Vagrant can log in on the virtual machine via SSH and install the machine with Chef Solo (see section 2.3.2). In addition to Chef Solo, Puppet or simple shell scripts can be used.

Vagrant can also be used to generate and provision multiple virtual machines.

For which purposes is it worthwhile to use Vagrant? Vagrant is very easy to install on a laptop or a developer machine. In addition, virtualization software is needed. Once both are in place, software can automatically be installed on the local computer on one or multiple VMs. Since the same tools can be used that are also employed for installations in production, the developers can try out the software in an environment that is similar to production.

2.4.1 An Example with Chef and Vagrant

To better illustrate Vagrant let us return to the example that was introduced in section 2.3.2. We will again install Java, a Tomcat server, and a Java web application on a computer.

Central for Vagrant is the Vagrantfile. This is actually in Ruby-Code—even if it maybe does not look like it at first glance. In fact, a Ruby DSL is used here—like previously in the case of Chef.

Listing 2.6 shows the Vagrantfile for the example. Here, it is defined which base image is supposed to be used for the system. This image is stored locally. This is followed by port forwards: Certain ports of the computer are redirected to ports of the virtual machine. If for instance the URL http://localhost:18080/ is opened, it is redirected to port 8080 of the virtual machine—where Tomcat should be running. Finally, the provisioning of the software is configured with Chef. It contains the paths to the cookbooks and the roles including the role the server is supposed to assume. Since the virtual machine can access the directories of the host, it is very easy in this manner to provide the VM with cookbooks and roles.

Listing 2.6 *Vagrantfile*

Click here to view code image

```
Vagrant.configure(2) do |config|
  config.vm.box = "ubuntu/vivid64"

  config.vm.network "forwarded_port", guest: 8080, host: 18080
  config.vm.network "forwarded_port", guest: 8081, host: 18081

  config.vm.provision :chef_solo do |chef|
    chef.cookbooks_path = ["cookbooks"]
    chef.roles_path=["roles"]
    chef.add_role("tomcatserver")
  end

end
```

With the command `vagrant up` the system can be started. This requires only that the Vagrantfile be in the current directory. The base image is downloaded, and the software is installed. `vagrant help` can be used to list additional commands. `vagrant provision`, for instance, can be employed to trigger a new Chef run. Thus, it is not necessary to build the VM completely anew—with `vagrant provision` only the necessary updating is performed, which can save a lot of time. `vagrant halt` can be used to stop the environment—only the VM is terminated. `vagrant destroy`, on the other hand, completely destroys a VM—including all files. Finally, `vagrant ssh` provides a shell on the virtual machine.

Try and Experiment

First Vagrant has to be installed:

1. Start off by installing virtualization software—for instance, Virtual Box.[22]
2. Download Vagrant[23] and install it according to instructions.
3. Fetch the GitHub repository with the command: `git clone https://github.com/ewolff/user-registration-V2.git`
4. Now you can start a machine and have it provisioned in the sub-directory `user-registration/chef` by using `vagrant up`.

There are different options for further experiments:

- A number of other Vagrant commands were already mentioned. If you start Vagrant without a command, all available commands are shown. Try them out.
- vagrant-cachier[24] allows buffering the Linux packages for the installation—this speeds the creation of Vagrant VMs since it is no longer necessary to download large amounts of software from the internet. Modify the Vagrantfile accordingly.
- Of course, the port for Tomcat can be changed. To do so the role has to be modified accordingly. Port 8081 is a good choice since it is already forwarded in the Vagrantfile.
- The list of plug-ins[25] for Vagrant is very comprehensive. It is useful to get more familiar with the options and to try out some of the plug-ins.
- In addition, you can use Packer[26] to assemble your own VM, which can be used with Vagrant.
- It is also possible with Vagrant to set up multiple virtual machines: see http://docs.vagrantup.com/v2/multi-machine/. Why not an additional Tomcat or a database server? With vagrant-hostmanager[27] the different VMs in Vagrant can find each other via their names.

2.4.2 Vagrant: Conclusion

At least for developer machines Vagrant is an excellent tool for installing servers and keeping the environments of all developers identical. In contrast to a manual setup for Virtual Box there are profound simplifications. However, with the plug-ins Vagrant gets even more interesting: It can be viewed as a general tool for starting and setting up VMs. Therefore, Vagrant is more or less an indispensable tool for developers working on infrastructure automation.

2.5 Docker

Continuous Delivery is based on virtual machines. It is practically impossible to use physical machines for the different phases of the Continuous Delivery pipeline. The virtual machines are entirely separate from each other—each has its own virtual hard drives with their own installation of the operating system. However, to keep the effort for the operation of the application minimal, installed operating systems and other components of the environments, like monitoring, are mostly identical on all machines. An application mostly consists of multiple machines—for instance, a database server and an application server.

With this approach resources are squandered: The virtual machines are largely identical, but they are set up completely separately. It would be much better if the parts they have in common were stored only once.

Besides, an increased virtualization efficiency is also advantageous in other respects: While tools like Chef make it possible to bring virtual machines up to the current software status, this creates a larger expenditure — the installation not only has to function on a fresh machine, but also on a machine where the software is already installed. However, if it was possible to rapidly install and start up a virtual machine anew, the virtual machine with the old software version could be deleted and a completely new one could be set up. Thereby the installation of software should become profoundly easier. In this context Phoenix servers[28] are discussed: Servers which at any time can be newly created like the mystical bird Phoenix that rises from its own ashes.

2.5.1 Docker's Solution

Docker[29] offers a substantially more efficient alternative to virtualization. It does so by employing a number of different technologies:

- Instead of a complete virtualization, Docker uses the so-called Linux Containers (LXC—LinuX Container) which in turn employ the cgroups of the Linux kernel. This allows implementation of a lightweight alternative to virtual machines: All containers use the same underlying operating system. Therefore, only one instance of the kernel is in memory. However, processes, networks, file systems, and users are kept separate. In comparison to a virtual machine (VM), a container has a profoundly smaller overhead—it is easily possible to run hundreds of containers on a simple laptop. In addition, it is much faster to start a container than a virtual machine since no operating system has to be booted—just a new process is started. The container is easy to manage since it only requires an individual configuration of the operating system resources. In addition to LXC as a basis technology, Docker is supposed to also support technologies like BSD Jails or Solaris Zones in the future.
- The file system is optimized as well: Basis images can be used from which only reading is possible. At the same time additional file systems can be blended into the container onto which it is also possible to write. One file system can overlay another file system. This allows, for instance, generation of a basis image that contains an operating system. If software is installed in a running container or if files are changed, only the modified data have to be saved. This substantially reduces the requirement for memory for the container on the hard drive.
- There are interesting additional options: For instance, a basis image can be started with an operating system, and afterwards software can be installed. As mentioned, only those changes to the file system are saved that were done during the installation of the software. Based on this delta an image can be generated. Then a container can be started that blends in this delta in addition to the basis image with the operating system—of course, subsequently additional software can be installed. In this way every "layer" in the file system can contain changes. The real file system at run time can be composed of such layers. This allows efficient reuse of software installations.
- Figure 2.4 shows an example for the file system of the running container. On the lowest level there is the Ubuntu installation. On top is the installation of Java. This is followed by the example application.

For the running container to be able to still write changes into the file system, there is a file system on top into which the new and updated files can be written. If the container reads a file, it will go through the layers starting from the top until it finds the desired file.

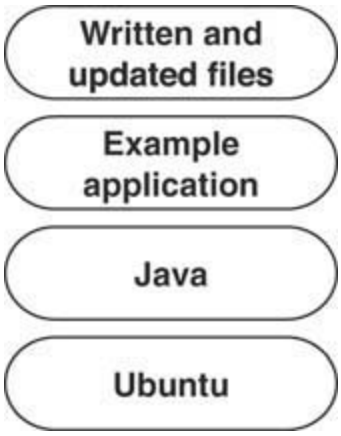

Figure 2.4 *Docker's file system*

Docker Container versus Virtualization

Thus, Docker containers offer a very efficient alternative to virtualization. However, in the end it is not a "real" virtualization since each container has only some separate resources, its own memory and its own file system; all share, for instance, the kernel. Therefore, this approach also has a number of disadvantages. A Docker container can only use Linux and only the same kernel as the host operating system—accordingly Windows applications, for instance, cannot be run. In addition, the separation between the containers is not as strict as in the case of real virtual machines. An error in the kernel would affect all Docker containers. Furthermore, Docker does not run on Mac OS X or on Windows systems—however, Vagrant (see section 2.5.4) or Docker Machine (see section 2.5.5) can help to circumvent this problem. Also on these platforms Docker can be installed directly. Behind the scenes a virtual machine with Linux runs the containers.

The Objective of Docker

The main objective of Docker is to use containers for the distribution of software. In comparison to normal processes Linux containers are much better at separating individual applications. Besides, when software has to be installed on another system, it is very easy to just use an image of a file

system for this. Docker offers a repository for images so that numerous servers can be run with identical images.

Communication between Docker Containers

Docker containers somehow have to communicate with each other. For example, an application on a web server has to be able to communicate with a database. To do so containers can expose network ports which other containers can use. In addition, a file system can be used in a shared manner. There data can be written, which other containers can read.

In the end, this results in a component model where software is distributed into individual Docker containers that communicate via ports and file systems. This allows distribution of the software very well. The containers are in general more fine-grained than typical virtual machines. In section 8.3 we will add analysis of the log files to the example application. We will use numerous Docker containers for this. In the case of a pure virtualization approach all these components would likely run on a single virtual machine in order not to waste resources.

2.5.2 Creating Docker Containers

Of course, in principle Docker containers can be installed with solutions like Chef or Puppet. The fundamental advantage of these tools is that the installations are reproducible and it is strictly defined what the system is supposed to look like after the installation—this makes updates very simple. This leads to scripts that are easy to adapt to different requirements by using parameters—they are therefore reusable. However, writing these scripts can be very laborious and time-consuming in practice.

When using Docker, the update of an installation is not so important anymore: Starting a new container has a substantially smaller overhead than starting a virtual machine. The same is true for the installation of containers. Besides, Docker containers are more fine-grained. Therefore, much less has to be installed in an individual container. That makes the installation easier and requires fewer resources.

Therefore, for Docker containers, in addition to the installation via Chef or Puppet an alternative is convenient where the containers are completely newly installed when a new release of the software or a component has to be delivered. Creating a completely new image is not that much work anymore. This profoundly reduces the effort for generating the installation

automation as the starting point can be a completely new container. So there is no need to be able to update an old version of the system.

If you nevertheless want to use Puppet or Chef in combination with Docker, Packer (see Endnote 26) can generate suitable Docker images which already contain Chef or Puppet.

Dockerfiles

Docker images can be generated in an interactive manner. In this case, a container is simply started, and the user can install the necessary software with the right commands. In the end, an image of the generated container is created so that as many instances of the container as needed can be generated and the state of the container can be exactly reproduced. However, the entire installation has to be repeated for a new release of the software. An alternative approach is a Dockerfile. Dockerfiles make it possible to automatically generate images that can serve as the basis for the file system of a Docker container. Writing a Dockerfile is very simple.

An example is shown in Listing 2.7. With this Dockerfile, Java can be installed in an image for Docker containers. This image is the basis for the installation of additional software like a Tomcat webserver or a Java application.

Listing 2.7 *Dockerfile for the installation of Java*

Click here to view code image

```
FROM ubuntu:15.04
RUN apt-get update ; apt-get dist-upgrade -y
RUN apt-get install -y openjdk-8-jre-headless
```

With the `FROM` directive another image is imported. In this case, a basic Ubuntu installation is used. Before the colon the image name is given and after it the tag—in this case the version number of Ubuntu. This image is downloaded from the public Docker repository. Such installations are profoundly reduced in comparison to typical Ubuntu installations. Most network services and many of the usual tools are not available in order to keep the size of the container image as low as possible. The installation itself is done with `RUN` entries. They are executed in the container. The first `RUN` updates the Ubuntu installation. Depending on the point in time when the script is executed, different updates are made so that the installation is

not reproducible in the strict sense. Therefore, it can make more sense in production to start from a fixed image—like Ubuntu 15.04, which is used in the `FROM` in this Dockerfile. This renders the installation completely reproducible. The second `RUN` command Docker installs Java in the image.

Behind the scenes each line in the Dockerfile results in a commit: The changes to the file system of the container are written into a separate layer within the file systems (see Figure 2.4). Therefore, this Dockerfile results in two images:

- An image with an updated Ubuntu installation
- An image with the Java installation

When another Dockerfile also updates the Ubuntu installation with the same `RUN` entry, this does not lead to an additional image, but the already existing intermediate image is reused—this means that no new image at all is created in the end. Also in this area Docker is optimized for efficiency.

In summary, Docker offers a very simple approach for the generation of images and containers that exclusively utilizes simple scripts.

Building and Starting Docker Images

To generate an image with Docker, the Docker command line tool can be used. For instance, the command for generating the Java image is:

```
docker build -t java java
```

Docker looks in the sub-directory `java` for a Dockerfile and creates the image based on this Dockerfile. It is tagged as `java` because of the parameter `-t`. Generating the image can take some time since first the Ubuntu basis image and the different packages have to be downloaded.

When you want to start a new container with this image, this can also be done with the command line:

```
docker run java echo "hello"
```

With this command a container is started with the generated image. On this container the command `echo "hello"` is executed. Afterwards the container shuts itself down again.

The command `docker run -i -t java /bin/bash` starts a container with the same image, but with a shell that waits for the entry of commands. The option `-i` prevents the container from being immediately stopped, and with `-t` a virtual terminal is connected to the container. Any further commands are therefore immediately directed to the container.

Try and Experiment

Docker is going to be introduced in still more detail within this chapter, and it is also going to be used later in Chapter 8, "Operations," in an example for the generation of a complex environment. However, some basic tasks are already worth practicing now:

- Find the Docker online documentation.
- Based on the information in this chapter, create a Docker image with Java. Find out which images are generated in addition to the Ubuntu basis image and the image with installed Java. Hint: When you start `docker` without parameters, you will get a list of all possible commands —one of them serves to list the images, and this command has to be appropriately parameterized. When you add the option `-h` to a Docker command, Docker lists the possible parameters.

2.5.3 Running the Example Application with Docker

By using the images from section 2.5.2 for the installation of Docker it is quite easy to run the example application with Docker. The necessary Dockerfile is shown in Listing 2.8. `COPY` copies the application out of the file system of the host into the Docker image. `CMD` defines the command that is to be executed when starting the container. With this command line the application is started. The application log file is saved in the directory `/log`. `EXPOSE` makes port 8080 of this container available to the outside.

Listing 2.8 *Dockerfile for the example application*

Click here to view code image

```
FROM java
COPY user-registration-application-0.0.1-SNAPSHOT.war user-
registration-application-0.0.1-SNAPSHOT.war
```

```
CMD /usr/bin/java -Dlogging.path=/log/ -jar user-registration-
application-0.0.1-SNAPSHOT.war
EXPOSE 8080
```

Generating the image works exactly as described in the previous section. Starting the image is more interesting:

```
docker run -p 8080:8080 -v /log:/log user-registration
```

The parameter `-p` connects port 8080 of the container to port 8080 of the host so that the application is also available at the host under this port. Likewise, the directory `/log` of the host is linked to the directory `/log` of the container. In this manner the container can access the directories of the host.

For tests, it can make sense not to store the executable file within the Docker image, but to read it from the host file system. In that case, every change to the application is immediately available in the Docker container. It just has to be ensured that in the container the directory '/target' with the result of the build is mounted and that the command line for starting the Java process is modified in such a way that the executable program is read from this directory.

Ideally the result of the build should be loaded from a repository server and subsequently copied into the image. In addition, it is also possible to generate a Docker image directly as result of the build process.

Additional Docker Commands

`docker ps` shows the currently running container, and with `docker ps -a` also the already stopped containers can be displayed. `docker logs` shows the log output of a container—the container ID has to be handed over as parameter. Images can also be stored and loaded from a repository—this can be done with `docker push` and `docker pull`.

Try and Experiment

Clone the example project.[30] With the Git command line tool[31] the necessary command is

```
git clone https://github.com/ewolff/user-registration-V2.git
```

The described installation of the application can be found in the sub-directory `docker`. ker (see: https://docs.docker.com/installation/).

1. First, compile the application with `mvn install` in the sub-directory `user-registration-application`.
2. Then, generate the Docker images for Java and for the user registration in the sub-directory `docker`. This can be done with commands like `docker build -t java java`. Remember that you will need to create two images.
3. Use `docker run -p 8080:8080 -v /log:/log user-registration` to start the application as Docker container. Maybe you have to use a different directory than `/log` for saving the log files. This directory will contain the output of the process as log file which facilitates the search for errors.
4. Generate the Docker image as part of the build process rather than by a separate call of Docker. There are Maven plug-ins which directly generate Docker images. Such a plug-in could for instance be integrated into the build process in order to generate the Docker image upon each build.

2.5.4 Docker and Vagrant

Vagrant (see also section 2.4) can use Docker as provider. In that case not virtual machines with VirtualBox, but Linux containers with Docker are used. This allows to easily generate a system consisting of multiple Docker containers since the creation of images and the starting of containers with the correct parameters is defined in a single Vagrantfile. In addition, accessing the file system of the host gets easier.

Nevertheless, the approach with Docker as provider differs substantially from using VirtualBox: Chef and Puppet are not available in most Docker images. Besides, the user often cannot connect to the machine via SSH

since no SSH server is available on the container. Therefore, simply replacing VirtualBox with Docker is not possible.

Vagrant as Provisioner

Dockerfiles represent an alternative to provisioning with Chef or Puppet. Instead of Chef or Puppet configurations containers with Dockerfiles are defined. Vagrant will then start a virtual machine in which it installs Docker and with the help of Dockerfiles generates and starts the necessary containers.

For the example application with the Dockerfiles which were described in sections 2.5.3 and 2.5.2 the Vagrantfile from Listing 2.9 can be used. An Ubuntu image into which Vagrant installs Docker serves as the basis. The `d.build_image` entries generate the appropriate images. This happens during the provisioning of the machine, i.e. only upon the first call of vagrant up. `run: "always"` ensures that the Docker containers are started `d.run` upon each call of `vagrant up`—i.e. not only upon the first call.

Listing 2.9 *Dockerfile for the example application*

Click here to view code image

```
Vagrant.configure("2") do |config|
  config.vm.box = "ubuntu/vivid64"
  config.vm.synced_folder "log", "/log", create: true
  config.vm.network "forwarded_port", guest: 8080, host: 8090

  config.vm.provision "docker" do |d|
    d.build_image "--tag=java /vagrant/java"
    d.build_image "--tag=user-registration /vagrant/user-
registration"
  end
  config.vm.provision "docker", run: "always" do |d|
    d.run "user-registration",
      args: "-p 8080:8080 -v /log:/log"
  end

end
```

The remainder of the configuration first of all connects directories and network ports. First, the ports and directories of the host on which Vagrant runs are connected to the virtual machine in which Docker runs and finally to the Docker container.

Figure 2.5 shows the different levels:

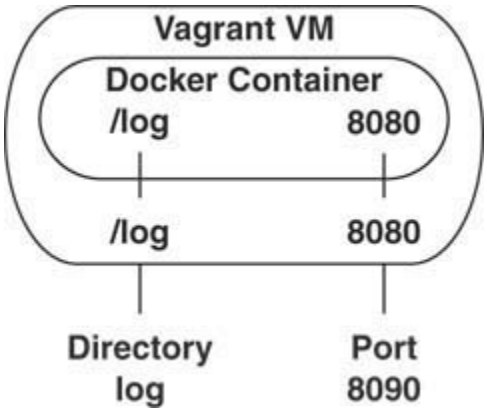

Figure 2.5 *Binding directories and ports between Docker, Vagrant, and the host*

- The `Vagrantfile` connects the directory `log` on the host with the directory `/log` on the Vagrant VM via the entry `config.vm.synced_folder`. Via the parameter `-v` for `d.run` it is mapped to `/log` in the Docker container.
- Port 8090 on the host is mapped onto port 8080 on the Vagrant VM and onto port 8080 on the Docker container. Therefore, the application is also available under the URL http://localhost:8090.
- `vagrant up` starts the Vagrant VM. The Docker images are generated, and the containers are started. `vagrant provision` generates only the Docker images and starts the containers. The Docker command line tool can also be used. However, to do so a session in the Vagrant VM has to be started via `vagrant ssh`.

Try and Experiment

1. Install Vagrant (see section 2.4.1).
2. Clone the example project.[32] When using the Git command line tool[33] the command is `git clone https://github.com/ewolff/user-registration-V2.git`.
3. In the example project there is a `Vagrantfile` in the directory `docker`. It deploys the example application as a Docker container in Vagrant. Start this environment.

The solution copies the application into the Docker image. Alternatively, the application could directly mount the directory of the Maven build into the host and into the Docker image. This requires that first the sub-directory `target` from the host be mounted into the Vagrant VM. This can be done via an entry in `config.vm.synced_folder` in the Vagrantfile. Subsequently, this directory has also to be mounted into the Docker container. In the Vagrantfile the log directory is mounted into the Docker container. This has to happen with the directory `target` as well. Finally, the command line for the start has to be changed in such a manner that this directory is used.

- As mentioned already, Vagrant can also use Docker as provider. Generate an appropriate environment based on the example.

2.5.5 Docker Machine

Vagrant serves to install environments on a developer laptop. In addition to Docker, Vagrant can also use simple shell scripts for deployment. However, this solution is ill-suited for production environments. Docker Machine[34] is specialized for Docker. It supports many more virtualization solutions and also a number of Cloud providers.

Figure 2.6 shows how Docker Machine can create a Docker environment: First a virtual machine is installed based on a virtualization solution like VirtualBox. It is based on Boot2Docker, a very light-weight Linux, which is meant as a run time environment for Docker containers. Docker Machine installs a current version of Docker on this virtual machine. A command like `docker-machine create --driver virtualbox dev` for instance

generates a new environment with the name dev, which runs on a VirtualBox machine.

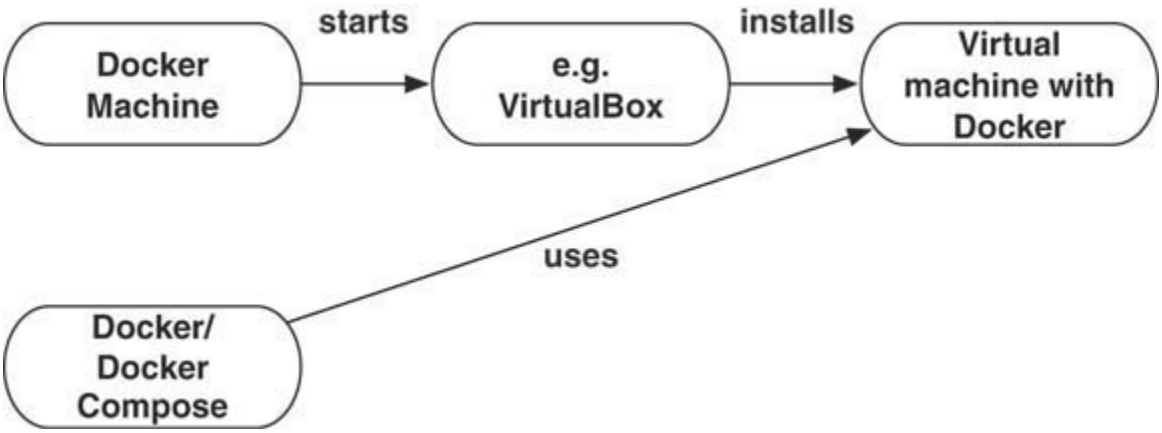

Figure 2.6 *Docker Machine*

Now the Docker tool can communicate with this computer. The Docker command line tools use a REST interface for communicating with the Docker server. Therefore, the command line tool just has to be configured in a way that it appropriately communicates with the server. Under Linux or Mac OS X the command `eval "$(docker-machine env dev)"`, which appropriately configures Docker, is sufficient for this. For Windows PowerShell the analogous command is `docker-machine.exe env --shell powershell dev` and for Windows command shell cmd `docker-machine.exe env --shell cmd dev`.

Thus, Docker Machine makes it very easy to install one or multiple Docker environments. All these environments can be administered by Docker Machine, and the Docker command line tool can access them. Since Docker Machine also supports technologies such as the Amazon Cloud or VMware vSphere, it can be used to create production environments.

Try and Experiment

1. The installation of Docker Machine is described at https://docs.docker.com/machine/#installation.
2. Docker Machine requires a virtualization technology like VirtualBox. The Installation of VirtualBox is described at https://www.virtualbox.org/wiki/Downloads.
3. Now a Docker environment can be created on a VirtualBox with the command `docker-machine create --driver virtualbox dev`.
4. `docker-machine env dev` explains how the environment can be accessed. To do so a command like `eval "$(docker-machine env dev)"` (for Linux / Max OS X), `docker-machine.exe env --shell powershell dev` (for Windows PowerShell), or `docker-machine.exe env --shell cmd dev` (cmd Windows command shell) has to be entered. You might need to provide the `--shell` parameter in other cases, too, if the shell is not correctly autodetected.
5. Clone the example project. With the Git command line tool the required command is `git clone https://github.com/ewolff/user-registration-V2.git`.
6. Now the Java Docker image can be created with `docker build -t java java` in the `docker` directory.
7. Likewise, the Docker image containing the application can be generated in the Docker directory with `docker build -t user-registration user-registration`.
8. Now the application can be started with `docker run -p 8080:8080 -d user-registration`. This connects port 8080 of the Docker container to port 8080 of the server.
9. `docker-machine ip dev` returns the IP address of the environment. The application is available on port 8080 and can be called in the browser under a URL like http://<ip>:8080/.
10. https://docs.docker.com/machine/get-started-cloud/ shows how Docker Machine can also be used with a Cloud. Accordingly, the example application can very easily also be started on a Cloud environment. The only thing that changes is the installation of the environment with `docker-machine create`.

11. At the end of your experiments `docker-machine rm` deletes the environment. Especially in the case of a Cloud environment this is important to reduce costs.

2.5.6 Complex Configurations with Docker

The example application in this chapter is a special case. It is a very simple application that does not use any database and consists of a single container. More complex environments comprise multiple containers that also have to communicate with each other. In the end a Docker container is a component —for instance, a database or a web application—that is able to deliver a service together with other components.

There are different options to enable Docker containers to communicate with each other:

- A container can expose ports. In the example application this is the port that the web application uses to listen for requests. Such ports can be used not only by the host, but also by other containers. This requires that so-called links between containers are established.
- Of course, containers can use directories of the host. Likewise containers can use specific data volumes that allow part of the file system of a container to be used by other containers in parallel.

Therefore, Docker containers can be used to implement individual components that exchange information via ports or shared directories. In Chapter 8 this approach is used to generate a more complex system consisting of the example application and a log analysis for the application.

Docker Registry

Docker images comprise the data of a virtual hard disk. Docker Registry allows for saving and downloading Docker images. This makes it possible to save Docker images as result of a build process and subsequently to roll them out on servers. Because of the efficient storage of images it is easy to also distribute complex installations in a well-performing manner. In addition, many Cloud offers are arising that are able to directly run Docker containers.

Docker in a Cluster

In the scenarios described so far we have utilized Docker to deploy containers on a server. In the end this turns Docker into a tool for the automation of software installations. With this approach only the software above the level of the operating system changes: Instead of individual processes and an installation with scripts or tools like Chef, Docker is now used. Underneath the long-known operating system Linux still runs.

However, it would be desirable to be able to directly deploy a system of Docker containers on a cluster. In that case it is for instance possible to start multiple instances of the containers depending on the load and the availability requirements. However, here Docker solves problems which in most companies are already solved by virtualization.

For such scenarios there are different technologies:

- Apache Mesos[35] is a scheduler. It administers a cluster of servers and assigns jobs to specific servers. Mesosphere[36] allows the running of Docker containers with the help of the Mesos scheduler. Mesos also supports many other kinds of jobs.
- Kubernetes[37] likewise supports the execution of Docker containers in a cluster. However, its approach is different than the one taken by Mesos. Kubernetes offers a service that distributes pods in the cluster. Pods are linked Docker containers that are supposed to run on a physical server. As a basis Kubernetes requires only a simple installation of the operating system—the cluster management is implemented by Kubernetes. Kubernetes is based on the internal Google system for the administration of Linux containers.
- CoreOS[38] is a very lightweight server operating system. Via etcd it supports the cluster-wide distribution of configurations. fleetd allows the deployment of services in the cluster—up to redundant installation, fail-safety, dependencies, and shared deployment on one node. All services have to be deployed as Docker containers, while the operating system itself remains largely unchanged. CoreOS can be used as a foundation for Kubernetes.
- Docker Machine[39] allows for the installation of Docker on different virtualization and Cloud systems (see section 2.5.5). Docker Compose (see next section) can configure a larger number of Docker containers

and the links between the containers. Docker Swarm[40] can combine servers that were generated with Docker Machine into a cluster. The Docker Compose configuration of the system can define which parts of the system should be distributed in the cluster and how they should be distributed.

2.5.7 Docker Compose

Docker Compose[41] allows the definition of Docker containers that together deliver a service. YAML is the format of the Docker Compose files.

As an example let us look at the configuration of the example application and the monitoring solution Graphite, which is going to be discussed in more detail in section 8.8.

Listing 2.10 shows the configuration of the example application for the Graphite monitoring. It consists of the different services:

Listing 2.10 *Docker Compose configuration for the example application and monitoring*

Click here to view code image

```
carbon:
  build: carbon
  ports:
   —”2003:2003”
graphite-web:
  build: graphite-web
  ports:
   —”8082:80”
  volumes_from:
  —carbon
user-registration:
  build: user-registration
  ports:
   —”8083:8080”
  links:
 —carbon
```

- `carbon` is the server, which saves the monitoring values of the application. The build entry defines that there is a `Dockerfile` in the sub-directory carbon via which the Docker image for the service can be generated. The port entry exposes port 2003 as port 2003 on the host on

which the container runs. This port can be used by an application to save values in the database.

- `graphite-web` is the web application that allows the user to analyze the values. It is available under port 8082 on the host, which is redirected to port 80 of the Docker container. The `volumes_from` entry takes care that a hard drive with data from the whisper database of the carbon container is also accessible in this container. Whisper is a library that stores and retrieves the metrics.
- Finally, the `user-registration` container contains the application itself. Via the port of the carbon containers the application delivers the monitoring data—therefore the two containers have a link. Thereby the container `user-registration` can access the container `carbon` under the host name carbon.

In contrast to the Vagrant configuration there is no Java container, that is, a container that only contains a Java installation. The reason is that Docker Compose only supports containers that indeed offer a service. Therefore, this basis image is now loaded from the internet.

This creates a system where three containers communicate with each other (Figure 2.7) via network connections or shared file systems.

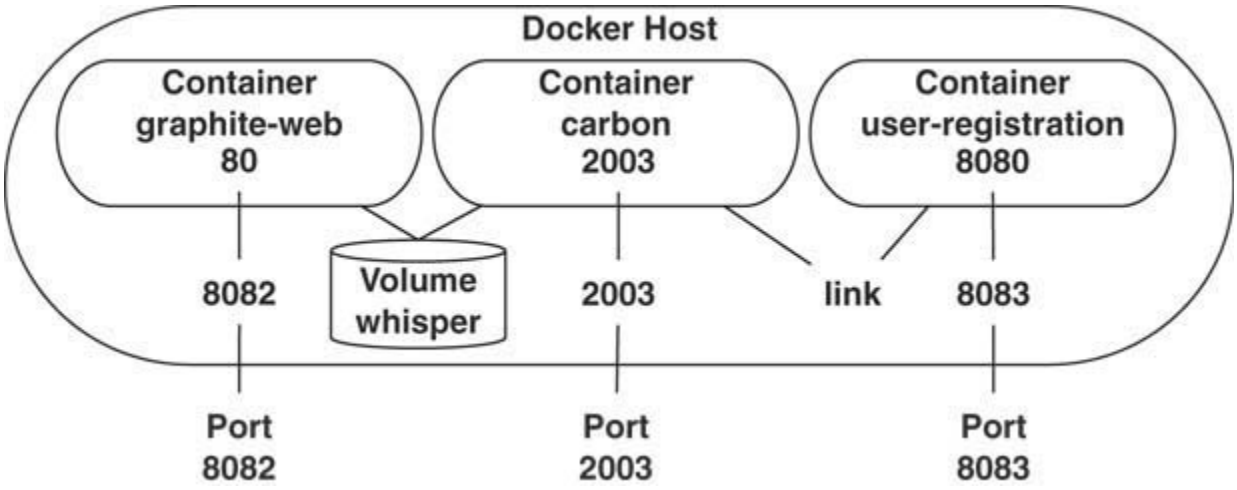

Figure 2.7 *Docker Compose setup*

Try and Experiment

1. First generate a Docker environment, for instance with Docker Machine (see section 2.5.5). At the end the `docker` command should work.
2. Install Docker Compose—see https://docs.docker.com/compose/install/.
3. Clone the example project. With the Git command line tool the required command for this is `git clone https://github.com/ewolff/user-registration-V2.git`.
4. Change into the sub-directory `graphite`.
5. Via `docker-compose` the system is created based on the Docker Compose configuration. The appropriate images are generated.
6. `docker-compose up` will then start the system. Docker Compose works with the same settings as the Docker command line tool. It can also work together with Docker Machine. Therefore, it is transparent whether the system is created on a local VM or somewhere in the Cloud.

Now the system should be available under the indicated ports.

Suggestions for additional interesting experiments:

- Recapitulate exactly how the volumes between the `user-registration` and the `graphite-web` container work. To do so have a look at the Dockerfiles. Where exactly is the volume defined?
- Find out how Docker Registry works. Download the example application into a Docker repository and start it from there.
- Extend the setup comprising Docker Machine and Docker Compose by using Docker Swarm (see Endnote 40) to run the application in a cluster. Since Docker Swarm is able to run on a Docker Machine infrastructure and since the Docker Compose configuration can include Docker Swarm settings, this should not be hard to do.
- Based on the Docker Compose configuration, the system can also be run on Mesos/Mesosphere, Kubernetes or CoreOS (see Endnotes 35, 36, 37, and 38). However, this infrastructure differs profoundly from a simple Docker infrastructure so that the required effort is larger.

- Solutions like the Amazon Cloud allow, via the EC2 Container Service,[42] the running of Docker containers. This can also serve as a basis to run a Docker system.

2.6 Immutable Server

Solutions like Chef focus on idempotency (section 2.3): No matter how often an installation script is called, the result should always be the same. Therefore, scripts describe the desired state of the servers. During an installation run the necessary steps to reach this target state are executed.

2.6.1 Disadvantages of Idempotency

However, this approach also has disadvantages:

- Describing the target state of a server can be more complex than defining the required installation steps.
- The description is not without loopholes. When a script does not contain information regarding a resource such as a file or a package, this resource cannot be adapted to a desired state. Further, when this resource is relevant but not in the necessary state, this can result in errors.
- Often servers are permanently updated during run time to the current configuration, but never completely installed anew. In this case, it can happen that it is no longer possible to create a completely new server since the permanent updates have introduced changes that are not performed upon a run of the current installation—for instance, because they have been deleted from the scripts in the meanwhile. This is a dangerous situation since it is not entirely clear anymore what the configuration in production actually looks like and because it is not easy anymore to install a new server with all changes.

Therefore, it is better when a server is always entirely newly created. In this case, it can be ensured that the server is in line with all requirements. Immutable servers are based exactly on this idea. A server is always created in one piece by installing the software on a basis image. When a change in the configuration or installation becomes necessary, a completely new

image is created. The installation of a server is never adjusted or modified. This ensures that servers can really be reproduced at any time and that they always have exactly the right configuration.

However, the effort for creating immutable servers appears very high at first glance—in the end the server has to be completely newly installed. But in comparison to the other phases in a Continuous Delivery pipeline this effort can still be rather small—in general the effort for the test phases is the largest.

2.6.2 Immutable Server and Docker

In addition, the optimizations achieved with Docker are especially effective when using immutable servers. For instance, when only a configuration file has been changed and copying this file into the image is the last step in Dockerfile, the creation of the new Docker images will be very fast. Also, it will need hardly any storage space on the hard drive since Docker uses always one image for each step of the installation. Therefore, Docker can reuse the basis images of the previous installations. Only for the last step—copying of the configuration file—is a new image really necessary. However, this image will not be very large and can be rapidly generated.

In this manner, Docker can be combined with an immutable server to guarantee a specific state of a software installation and at the same time to facilitate the installation compared to idempotent installation approaches. Of course, it is also possible to implement an immutable server without Docker or to combine Docker with Chef.

2.7 Infrastructure as Code

The tools described so far will change the character of the infrastructure compared to classical approaches. In fact, infrastructure is turned into code—exactly like the production code of the application. Consequently, the term “infrastructure as code” is generally used. The infrastructure is not generated in a complex manual process, but by an automated process. This has a number of benefits:

- Errors occurring during the creation of environments are largely avoided. Each environment is the result of the same software.

Therefore, all environments should look the same. This creates additional safety for the delivery in production.

- In addition, it can be guaranteed that test environments are exactly like the production environments—up to the level of firewall rules and network topology. Especially in this area most test environments differ from production environments. When infrastructure as code is consequently used, such differences can be avoided and thus the predictive value of the tests can be increased. Consequently, fewer errors are found at the stage of production.
- When problems with an environment arise during a test or when the environment is altered unintentionally, only a manual repair of the environment can solve the problem. In the case of infrastructure as code the environment can be simply deleted and replaced by a fresh one. Alternatively, the environment can be repaired with the help of automation.
- Infrastructure can be versioned together with software. This ensures that software and infrastructure fit together. When a certain software version requires a specific infrastructure change, such a mechanism can ensure that this change is introduced.
- Finally, processes for modifying the infrastructure can be established: This allows the review of all infrastructure changes. In addition, this mechanism ensures that all infrastructure changes are documented and traceable.
- In addition, infrastructure as code helps to keep track of the installed software. Each installation and each component can be found in the installation rules. This facilitates an inventory—the installation of each individual component can be found in the configuration. Consequently, this approach can tackle problems for which normally a Configuration Management System is necessary. Necessary updates—for example because of security issues—can also be rolled out centrally.
- Normally, the number of available environments is limited. Besides, the environments have to be managed: New software has to be installed or test data have to be provided. The automation allows the installation of environments without much effort. When, in addition, servers can be provided flexibly thanks to virtualization, an unlimited (in principle) number of environments can be generated. Especially during a test

phase this is of course very helpful since numerous releases and problems can thereby be tested in parallel. Ideally, the environments can be selected simply via a portal and are automatically provided. Likewise, it is possible to start environments in times of high load. For instance, for end-of-year business, extra server farms can be installed that can then be deleted afterwards.

- Operations is subject to permanent cost pressure. At the same time, the number of applications and therefore the number of systems is constantly rising—in part also due to the fact that virtualization has lowered the costs of hardware. In order to be able to keep the ever larger and more complex zoo running with the existing operations team, operations has to become much more productive. Automation and infrastructure as code are the required tools to make this possible.
- To really profit from these advantages, changes of the infrastructure should only be introduced as changes of the infrastructure code. To attain this goal in reality can be quite difficult. Therefore, it can make sense to reduce the complexity of the environment.

2.7.1 Testing Infrastructure as Code

Since infrastructure is turned into code, the usual rules for code apply: For example, it makes sense to write tests for the code. This is common practice for business logic and should of course also apply to infrastructure automation. Accordingly, there are special tools to meet this challenge.[43] The key term is test-driven infrastructure.

Serverspec

An option to practically implement tests is provided by Serverspec.[44] This technology allows the development of tests that define in which state the server should be after an installation run. The tests are executed via SSH so it is not necessary to install complex software on the servers to be tested. In addition, the state of servers can be evaluated independently of the employed infrastructure automation.

Test Kitchen

Test Kitchen[45] is a solution specifically for Chef. It enables the test of Chef scripts with the help of different virtualization solutions and also supports

the test-driven development of Chef code—that is, the test can already be generated prior to the infrastructure automation.

ChefSpec

In addition, unit tests can be performed with ChefSpec.[46] In this case, the automations are not really executed, but only simulated. This allows for a faster execution of the tests and therefore provides more rapid feedback. However, since no real servers are installed, the predictive value of these tests is somewhat limited. Besides, it can be helpful when the application at least briefly examines at the start whether all necessary resources are available and accessible. Furthermore, it should ensure that the configuration is syntactically correct. Thereby possible problems can be identified and solved early on.

2.8 Platform as a Service (PaaS)

Solutions like Docker, Chef, or Puppet all have one thing in common: They automate the installation of complex applications and build up an individual technology stack. For a Java application, for instance, a Java run time environment is initially installed on the operating system and then an application or web server might be generated if necessary. Afterwards the application is deployed on this server. Additional services—databases, for example—likewise have to be installed and provided to the application.

In principle, another approach would also be possible: When the environment is standardized—that is, the exact same application server and database are always used—dealing with the application is much easier: Only the standardized environment has to be provided, and then the application has to be installed on this environment. This is exactly the approach PaaS (Platform as a Service) takes: A standardized platform is provided on which the application can be deployed. This renders automatic deployment much easier. Often it is sufficient to just call a command line tool. On the other hand, flexibility is reduced. Introducing changes to the platform is not easy anymore.

There are different PaaS offerings:

- Cloud Foundry[47] is an open source project, which can be installed in your own data center or can be used in a public Cloud. An entire

ecosystem has grown around Cloud Foundry—there are numerous extensions and operators of public Cloud Foundry installations. Cloud Foundry uses so-called Buildpacks in order to support different programming languages. Currently, there are Buildpacks for Java, Node.js, Ruby, and Go. For other languages you can use your own Buildpacks or try out Heroku Buildpacks. In addition, databases can be integrated into the platform as services that are available to the applications.

- Heroku[48] is only available as PaaS in the public Cloud. Via the already mentioned Buildpacks it supports numerous languages—apart from Ruby, Node.js, PHP, Python, Go, Scala, Clojure, and Java there are also Buildpacks for exotic languages like EMACSlisp that are supported by the community. Add-ons allow the inclusion of databases or other support—for instance, for the analysis of log files. Since Heroku runs in the Amazon Cloud, it is also possible to install other functionalities on servers in the Amazon Cloud and to use them via Heroku applications.
- Google App Engine[49] is a public Cloud PaaS that supports PHP, Python, Java, and Go. In addition, there is an SQL database that is based on MySQL, and a simple NoSQL database.
- Amazon Elastic Beanstalk[50] uses the Apache HTTP server for Node.js, PHP, Go, Python, Ruby, .NET, and Apache Tomcat for Java; it also supports Docker. The service runs in the Amazon Cloud and therefore can access a large variety of databases and other software components. In addition, it is possible to install servers by yourself and to run them in the Amazon Cloud. This allows the implementation of practically any solution.
- Microsoft's Azure App Service[51] supports .NET, Node.js, Java, PHP, Python, and Ruby. The range of options is extended by different databases, like Oracle or Microsoft SQL Server. Since it is in Azure, it is likewise possible to run any server, and this technology allows the running of nearly any application.
- OpenShift[52] from Red Hat offers, in addition to an installation in the public Cloud, a product that can be installed in your own data center. OpenShift supports Haskell, Java, PHP, Node.js, Ruby, Python, and Perl. For Java there are, in addition to Tomcat, JBoss and Vert.x

available. Databases like MongoDB, PostgreSQL, and MySQL are likewise on offer.

Especially in the case of applications that have to be accessible from the web, these approaches have profound advantages: It is quite easy to scale the applications, and many services also have world-wide distributed data centers so that a high degree of availability can be achieved. This means that it is not necessary to run your own data center and internet connections. Frequently this leads also to a cost reduction. The case is different when the application has to run in a data center on-site—for instance, because of data security. Most solutions are not meant to be installed in your own data center—in the list above, OpenShift and Cloud Foundry are the exceptions. Besides, the installation of such an environment is complex and laborious. Therefore, it often makes more sense in such scenarios to employ solutions like Puppet, Chef, or Docker. In addition there are some solutions like Flynn[53] or Dokku,[54] which implement simple PaaS Cloud solutions based on Docker.

Try and Experiment

The solutions mentioned here offer cost-free access for testing, and tutorials to facilitate the first steps. The example application is a simple Java application—however, it requires Servlet 3.0 support. Register with a Cloud provider of your choice and try to get the application to run there.

2.9 Handling Data and Databases

Databases pose a special challenge during the provision of infrastructure—therefore, an entire section is dedicated to this topic.

Relational databases have a schema. It determines how the data that should be stored in the database have to be structured. Therefore, the database has to be provided with a schema after the installation. This could of course be done via a script that provides the empty database with a suitable schema. However, this approach has a profound disadvantage: Its starting point is an empty database. Accordingly, this strategy cannot work when there are already data stored in the database during production. The problem is analogous to the installation scripts that were described in

section 2.2: It does not make sense to just install software blindly. An approach has to be chosen that is able to deal with different starting states, in this case of a database, and to transfer them all into the desired final state. When there is an old version of a database in production, the database has to be updated properly. This might require migrating data—for instance, a new column could be provided with default values. However, if the version of the database is already up-to-date, such a migration should not take place.

2.9.1 Handling Schemas

To be at all able to modify the schemas of the data in the database, all applications that access the data have to be modified. Therefore, when several applications access the same database and some of these applications are not developed any further, changing the schemas becomes practically impossible. Thus, shared access to a database should be avoided, especially in scenarios where software often has to be modified and Continuous Delivery is employed.

For such scenarios there are tools available that all work in a similar manner:

- Within the database itself a version number is saved. Via this version number it is transparent which version the schema in the database belongs to.
- To bring a database from one version to the next one, suitable scripts are developed. These scripts can introduce the necessary changes to the schema and also adjust the data itself appropriately.
- If necessary, scripts can also be developed that regenerate an older database version from the current one—for example, to undo changes. This can make sense, for instance, when a test database has a too new version of the database schema for testing a bug fix of an old software version.

With this approach, a tool can determine which operations are necessary for the migration. When version 38 of the schema is found in the database and version 42 is required, the scripts necessary for migrating from version 38 to 39 and then on to version 40, 41 and finally 42 are executed. In the

case of a completely empty database all scripts can be run one after the other.

There are many tools supporting this approach:

- Active Record Migrations[55] as a part of Ruby on Rails offers support for this approach. The changes are written in a Ruby DSL. This tool deserves special mention since it was one of the first ones based on this approach.
- Flyway[56] implements a similar approach using Java. The scripts can be implemented in SQL, but also in Java.
- Liquibase[57] is likewise implemented in Java, but typically uses its own DSL instead of SQL.

In addition to these tools there are many similar tools that can be integrated into popular software development environments.

However, schemas are a problem that is limited to relational databases. An alternative option is the use of NoSQL databases. They are much more flexible with regards to schemas—in fact, arbitrary data can be stored in them. Thus, there is no rigid schema definition—consequently, the schema definition never has to be modified. This benefit should not be underestimated: There are in fact projects which do not use relational databases for exactly this reason, but instead have changed to NoSQL databases. In reality, migrating data and changing schemas is a real problem in relational databases—especially in the case of large data sets.

2.9.2 Test and Master Data

In addition to handling schemas there is a further problem with regards to databases: An empty database does not make a lot of sense. Often databases have to contain certain master data, and for tests, of course, test data, and in production, naturally, the production data.

Master data can be generated with a script—these scripts then also have to be able to handle situations where databases already contain data. Therefore, the master data can, for example, be generated via scripts of the respective schema migration tools. The same holds true for test data.

Alternatively, backups from production can be used as data for tests. Of course, this requires that no data security guidelines oppose this approach.

In addition, the testing cases have to be adapted to the data, and have to be consistent across the stages. When the data have been altered in production, this can lead to failures during testing.

Therefore, depending on the actual situation, it can make more sense to define a specific test data set that is adapted to the testing scenarios. In that case, it will be ensured that the test data fit the testing scenarios. However, when the task is to migrate old data, a test with as large an amount as possible of old data should be run as early as possible, because such data sets have many surprises that are not found in synthetically created test data. For instance, it is not unusual that old data contain data sets twice or comprise data sets that are not in line with the schema—that are, in fact, invalid. These data sets are especially important for testing since they can cause problems in the software. Therefore, it should be the goal in such a scenario to run the tests with real production data early on.

Finally, an additional problem is that databases can behave differently depending on the amount of data that is stored within. Thus, while a schema migration might run without problems with a reduced test data set, the migration on the production system might fail, just because the amount of data is profoundly larger. If this is only noticed shortly before the scheduled delivery of a release, it will be too late. Therefore, the migration of data, especially, has to be tested in time with a realistic amount of data.

In addition to these rather technical approaches for handling databases there are also some other options of a more conceptual nature to address these issues. Thus, section 11.5 discusses approaches for handling databases from an architectural viewpoint, and explains once more the relevance of databases for Continuous Delivery.

2.10 Conclusion

Different technologies for infrastructures have been presented in this chapter:

- *Chef* makes it possible to build up environments. In this process only the desired state of the environment is described. Chef then transfers the environment into this state. This approach avoids many problems associated with normal installation scripts. In addition, there are already many prepared scripts.

- *Knife* and *Chef Server* can be used to easily start and install a server just with the call of a command line tool.
- *Vagrant,* on the other hand, is excellently suited to build up environments on developer computers.
- *Docker* represents an interesting alternative option: In addition to being a very efficient alternative to virtualization, Docker containers also allow for the installation of software with easy scripts. Even the installation of Docker containers in a cluster can be done relatively simply.
- *Docker Machine* allows the installation and execution of Docker containers on practically any server.
- With *Docker Compose*, multiple linked Docker containers can be installed and run.
- All of these tools turn infrastructure into code—which has far-reaching consequences.
- Just relational databases are somewhat more difficult to handle.

With this the basis of Continuous Delivery is set: Environments for each phase of the deployment pipeline can be generated—and essentially by pushing a button. Only on this basis can we in fact implement the pipeline.

Endnotes

1. https://www.virtualbox.org/
2. https://www.vagrantup.com/
3. https://git-scm.com/book/en/v2/Getting-Started-Installing-Git
4. https://puppet.com/
5. https://www.ansible.com/
6. https://www.ansible.com/get-started
7. https://github.com/ansible/ansible-examples/tree/master/tomcat-standalone
8. https://saltstack.com/
9. https://docs.saltstack.com/en/latest/topics/tutorials/walkthrough.html
10. https://docs.chef.io/berkshelf.html

11. https://github.com/
12. https://supermarket.chef.io/cookbooks
13. https://www.virtualbox.org/
14. https://docs.chef.io/install_server.html
15. https://learn.chef.io/
16. https://www.vagrantup.com/
17. https://www.virtualbox.org/
18. https://github.com/mitchellh/vagrant/wiki/Available-Vagrant-Plugins
19. http://www.vagrantbox.es/
20. https://github.com/jedi4ever/veewee
21. https://www.packer.io/
22. https://www.virtualbox.org/
23. https://www.vagrantup.com/
24. https://github.com/fgrehm/vagrant-cachier
25. https://github.com/mitchellh/vagrant/wiki/Available-Vagrant-Plugins
26. https://www.packer.io/
27. https://github.com/devopsgroup-io/vagrant-hostmanager
28. https://martinfowler.com/bliki/PhoenixServer.html
29. https://www.docker.com/
30. https://github.com/ewolff/user-registration-V2
31. https://git-scm.com/
32. https://github.com/ewolff/user-registration-V2
33. https://git-scm.com/
34. https://docs.docker.com/machine/
35. http://mesos.apache.org/
36. https://mesosphere.com/
37. https://kubernetes.io/
38. https://coreos.com/
39. https://docs.docker.com/machine/
40. https://docs.docker.com/swarm/

41. https://docs.docker.com/compose/
42. https://aws.amazon.com/documentation/ecs/
43. Stephen Nelson-Smith: *Test-Driven Infrastructure with Chef: Bring Behavior-Driven Development to Infrastructure as Code*, O'Reilly, 2nd Edition 2013, ISBN 978-1-44937-220-0.
44. http://serverspec.org/
45. http://kitchen.ci/
46. http://docs.chef.io/chefspec.html
47. https://www.cloudfoundry.org/
48. https://www.heroku.com/
49. https://cloud.google.com/appengine/docs
50. https://aws.amazon.com/elasticbeanstalk/
51. https://azure.microsoft.com/en-us/?b=16.47
52. https://www.openshift.com/
53. https://flynn.io/
54. https://github.com/dokku/dokku
55. http://edgeguides.rubyonrails.org/active_record_migrations.html
56. https://flywaydb.org/
57. http://www.liquibase.org/

PART II

The Continuous Delivery Pipeline

The following chapters each explain a phase in the Continuous Delivery Pipeline in detail:

- **Chapter 3** explains the commit phase with unit tests, automated code reviews, build tools and repositories.
- **Chapter 4** provides more information about automated acceptance tests.
- In **Chapter 5** the focus is on automated capacity tests.
- **Chapter 6** describes explorative and manual tests.
- **Chapter 7** details techniques and technologies for deployment.
- What is important for operations is detailed in **Chapter 8**.

Chapter 3. Build Automation and Continuous Integration

This chapter was written by Bastian Spanneberg. He is a Senior Engineer at Instana where he works on platform architecture and automation. His Twitter handle is @spanneberg.

3.1 Introduction

On the technical level Continuous Delivery is completely focused on automation. The first prerequisite for a Continuous Delivery pipeline that has to be met is the automation of the build process and the creation and automation of a suitable infrastructure for the Continuous Delivery pipeline. These topics are covered in this chapter.

The automated build is the basis for the commit stage and is therefore the first step of a Continuous Delivery pipeline. Thus, section 3.2 deals with the topic of build tools: their features and use. Afterwards, unit tests are discussed in an extra section (section 3.3). Unit tests are an important fundamental to gather reliable information about software functionality and stability in the course of the build. Section 3.4 addresses Continuous Integration; that is, the continuous building, integration, and testing of software. The widely used Continuous Integration server Jenkins is introduced as concrete technology. Static code quality analysis is the focus of section 3.5. Here, the widely used tool SonarQube is explained in detail. The section demonstrates how the tool can be integrated into the build and into the Continuous Integration environment. Using Artifactory as an example, the last section of this chapter (section 3.6) introduces the topic of artifact repositories and their role in the context of Continuous Integration and Continuous Delivery.

3.1.1 Build Automation: An Example

Section P.2 introduced the example of Big Money Online Commerce Inc., which did not use Continuous Delivery. However, the company had employed an automated build as well as a Continuous Integration server. In addition, upon the introduction of Continuous Delivery, a static code analysis was introduced to identify problems with regards to code

complexity and test coverage early on. This analysis yielded a number of hints of how project quality and especially maintainability could be improved. The results of the build are now stored in a repository to make them available to the entire pipeline. This creates a solid technological basis for the Continuous Delivery pipeline and ensures the necessary quality for the later steps.

3.2 Build Automation and Build Tools

Build tools automate software builds. Usually, a software build consists of a number of steps that are dependent on each other. In detail the sequence of events depends, for instance, on which programming language is used and for which target platform software is developed. Nevertheless, many builds share individual phases. The following phases are generally part of a build:

- Compiling the source code to binary code.
- Running and evaluating unit tests.
- Processing existing resource files (for example, configurations).
- Generating artifacts that can be used later (for example, WAR or EAR files, Debian packages).

Additional steps that are often executed in the course of a build and that can likewise be automated with the employed build tools are:

- Administering dependencies of, for instance the libraries that are used in the project.
- Running additional tests, such as acceptance and load tests.
- Analyzing code quality and examining defined conventions in the source code (static code analysis).
- Archiving generated artifacts and packages in a central repository.

For developers many of these tasks are taken care of by the respective development environment without the need to automate them with the help of a tool. The essential reason to nevertheless implement these tasks via specialized tools is to provide and ensure a reproducible and isolated build that is independent of the used development tools. Ultimately, developers can use different tools; however, there still has to be a uniform and

reproducible build. This build represents the basis for the application of Continuous Integration and, in the end, Continuous Delivery.

3.2.1 Build Tools in the Java World

In the Java world build automation is currently dominated by three tools: Ant, Maven, and Gradle. These tools exhibit profound differences with regards to the approaches they pursue.

- Ant[1] follows an imperative approach. Developers have to take care of all build aspects by themselves. For example, they have to generate directories for the compiled code and explicitly handle dependencies between the individual parts of the build. In essence, the developers have to implement the build completely by themselves in a step-by-step manner.
- Maven[2] takes the opposite route and uses a declarative approach. For many aspects of the build it predefines conventions that Maven projects have to abide by. Examples are the locations where source code and tests are stored, the directories for the build results, and the sequence of the build phases. This renders the configuration of a build very easy. However, wherever a build deviates from the convention or where configurations have to be changed, the developer has to specifically declare this. So in practice Maven builds are often even more complex than builds with other tools. In any case the sequence of phases of a build has to fit to the life cycle model of Maven, which defines the different build phases.
- Gradle[3] represents the newest approach with regards to Java build tools. It attempts to combine the best aspects of the imperative and the declarative approaches. As in the case of Maven, with Gradle many aspects of a build are defined by conventions. However, if the conventions do not fit, Gradle, like Ant, allows the developer to completely deviate from them and to implement independent processes with the help of the Gradle DSL or the programming language Groovy on which Gradle is based.

The following sections discuss these three tools in more detail.

3.2.2 Ant

Ant is the oldest of the mentioned tools. It resembles the well-known Unix build tool Make and follows an imperative approach. This means it is completely up to the developer to implement all phases of the build with Ant scripts in XML files and to define and administer the dependencies between the phases. Build goals are called targets in Ant and consist of a number of tasks. Ant innately brings a large number of tasks along—for instance, for processing files and directories, for compiling Java code, or for generating different types of archives. In addition, the developer can implement individual tasks in Java when the ones contained in Ant are not sufficient. Within the Ant contrib project[4] a number of additional tasks are available which are easy to embed. Ant does not administer dependencies. However, there is a separate solution for this, called Ivy,[5] which is very easy to integrate with Ant.

In many companies a lot of very complex Ant builds can still be found. In newer projects Ant is usually not taken into consideration anymore since the complete implementation of a build in XML normally appears to be too complex. Therefore, this chapter concentrates on Maven and Gradle instead. However, in the context of Continuous Delivery, the concrete selection of a build tool does not play an important role. Ant still allows the implementation of all necessary steps without problems, even if it is in some ways profoundly more cumbersome to use than a more modern tool.

The most important point is to properly and intelligibly automate all necessary steps and to continuously maintain and refactor the generated build logic—as well as the program code itself—in order to ensure a long-lasting high code quality for build scripts.

3.2.3 Maven

Currently, Maven is likely the most widely used build tool for Java—especially in large enterprises. In contrast to the earlier de facto standard Ant, Maven does not intend that the developers write all necessary tasks for the compilation, testing and packaging themselves, nor define the dependencies between the phases on their own. Instead, Maven is geared to the model “convention over configuration” by working with a preset, universally reasonable life cycle. Of course, this approach inherently limits the types of builds that Maven can support. The Maven default life cycle comprises a number of predefined and fixed phases. Within those it represents all activities of a classical software build. Figure 3.1 shows the

sequence of events as it is implemented in Maven: First, the source files are verified, and the build is initialized. Subsequently, the sources and resources are compiled and processed. The same happens with test sources and resources. Afterwards, the result is packaged, and optionally an integration test is run. Finally, the result can be verified, installed, and deployed in an artifact repository. Not all actions are necessarily executed in each phase. However, in this way Maven offers a skeleton along which your own build logic can be oriented.

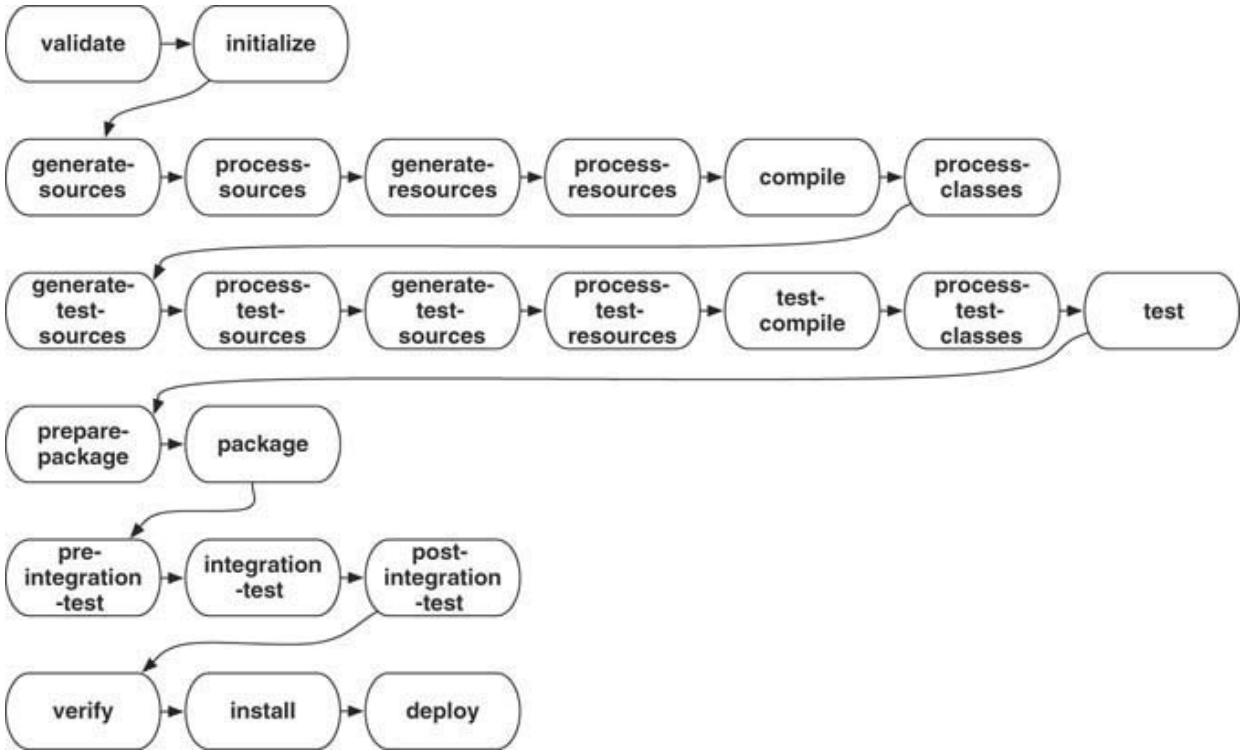

Figure 3.1 *Phases of a standard Maven build*

Due to the standardized phases, the commands for the compilation and testing of software are the same in each Maven project: `mvn package` executes all phases up to packaging and creates a build. `mvn test`, on the other hand stops at the phase "test."

Maven plug-ins implement the concrete tasks of the different phases. The execution of the different plug-ins is mapped onto the appropriate phases. There are predefined plug-ins for each phase. Via the phases a Maven build can also be extended beyond the standard. For instance, in the phase "test" a plug-in for running load tests can be configured in addition to the standard

test. Alternatively, certain actions of plug-ins, called goals in the Maven jargon, can be executed singly on the command line.

Maven's approach follows conventions. This has the advantage that the build files are more compact and maintainable than Ant build files, since it is not necessary in Maven to explicitly determine all details. Nevertheless, in reality Maven build files also have a substantial size. One reason for their large size is that many of the default settings of Maven are very conservative. For example, the compiler plug-in still assumes Java 1.5 as a setting, although Java 1.8 has become the current version in the meantime and 1.5 is not even supported anymore. Such settings generally have to be overwritten. In addition, the use of XML quickly results in large files. However, the structure of the build files is always the same. This makes it easy for developers to start working on Maven projects. They only need to become familiar with the predefined structure.

Listing 3.1 shows a minimal example of a Maven Project Object Model (POM) of a Java 8 project. POMs define how Maven is supposed to perform a build. If settings are supposed to be shared across multiple Maven projects, a good solution is to generate a so-called parent POM. Parent POMs allow the developer to define settings for multiple builds and to embed the settings in each build. Then the concrete builds only have to define the specific settings that are unique to them in the POM. The user registration example project, for instance, uses a parent POM of the Spring Boot Framework.

Listing 3.1 *A basic Maven POM (pom.xml)*

Click here to view code image

```
<project xmlns="http://maven.apache.org/POM/4.0.0" … >
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.example</groupId>
  <artifactId>my-project</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>
  <dependencies>
    <dependency>
      <groupId>org.junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <build>
```

```
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-compiler-plugin</artifactId>
          <version>3.1</version>
          <configuration>
            <source>1.8</source>
            <target>1.8</target>
          </configuration>
        </plugin>
      </plugins>
    </build>
  </project>
```

It contains the name of the project, consisting of a group and an artifact ID as well as the version. Via this information the software can be referenced by other components. Dependencies on other components are indicated in the same manner. The declaration regarding the packaging determines which type of artifact is created by the build. In addition to JAR (Java Archive) there are also, for instance, WAR (Web Archive) and EAR (Enterprise Archive). Also, in this case, the concrete implementation of the packaging is hidden in a plug-in, to be specific in the maven-jar-plugin[6] in the case of the above example.

The handling of dependencies is directly integrated in Maven, as can be seen in the example. Here, a dependency on the test framework JUnit is declared. During the build Maven takes care that the dependencies, if necessary, are downloaded from a central repository on the Internet or from a company-internal repository and are added to the class path during compilation. Transitive dependencies, that is, dependencies of the dependencies, are also handled by Maven. The Maven documentation[7] covers this topic in more detail.

Versioning and Snapshots

Another core concept of Maven is the use of so-called snapshot versions. They are employed to provide intermediate versions of the software in between the formal releases for other team members or for components that have defined the software as a dependency. These intermediate versions are saved in specific artifact repositories from which all interested parties can obtain the current state. Upon each build the snapshot version—in the example 0.0.1-SNAPSHOT—is overwritten. In this way the most current snapshot is always provided under this version. This means that the software

changes, but not its version number. Consequently, the different versions cannot be distinguished by the build tool anymore. This does not fit the objective of Continuous Delivery: that each change and therefore each build represents a potential, unambiguously identifiable candidate for a release. Each build is propagated through the Continuous Delivery pipeline and finally can even be brought into production if its quality is sufficient. In the end, snapshots only make sense for builds that are generated outside of the Continuous Delivery pipeline and are never propagated through the Continuous Delivery pipeline—for example, local builds of an individual developer.

However, in the Maven world it has never been the goal to roll out snapshots on production environments. They are just meant as integration artifacts for other teams and components. In this respect Maven contradicts a fundamental concept of the Continuous Delivery area.

Releases with Maven

The classical way to generate a Maven release is the use of the Maven release plug-in.[8] It takes care of a number of activities that the developers would have to do themselves during the manual build of a release:

- Within the POM the -SNAPSHOT postfix of the name is removed from the version, and afterwards the software is built and tested. In this manner a version of the software with a proper version number is generated.
- After a successful build the modified POM is checked into version control, and the version is tagged. The tag is typically the release version number. Subsequently, the version number is incremented and supplemented with a -SNAPSHOT. This change is also checked into version control.
- The previously tagged release state is checked out, built, and tested.
- The generated artifact (for example, JAR, WAR, or EAR) is stored in an artifact repository.

Although there are no changes in the software except for the version number, the software is completely built and tested multiple times. In addition, the plug-in commits two new code versions into version control.

This approach is not feasible anymore when each change is supposed to represent a potential release in line with the Continuous Delivery idea. Firstly, two new versions of the software are created which do not differ from the original version except for the version number in the POM. Secondly, the software is built and tested more often than required, which unnecessarily wastes resources. Still, a procedure has to be found for unambiguously identifying each build without putting up with the overhead produced by the release plug-in.

One option is to set the version from the outside at build time. This can be done with the Maven version plug-in.[9] It can set the version of a project as a parameter:

```
> mvn versions:set -DnewVersion=0.0.2
> mvn clean deploy
```

A build server can use this feature, for instance, to make a consecutively numbered build number a part of the version number. Tagging the version in version control can be done with the Maven SCM plug-in:[10]

```
> mvn scm:tag
```

If these two elements are combined into a central build, an unambiguously identifiable version can be built for each change. Even if the POM contains a snapshot version, in this scenario it only serves the developers for the purpose of integration. However, one still has to decide whether in the context of Continuous Delivery it makes sense to generate an SCM tag since each build is a potential release and therefore would have to be tagged. Alternatively, the commit hash can be turned into a part of the final version number or can be made available within the application by other means to be able to identify the version of the source code at run time.

An illuminating discussion of snapshots, releases, and Continuous Delivery can be found in Axel Fontaine's blog article Maven Releases on Steroids[11] and Maven Release Plug-In: The Final Nail in the Coffin.[12]

3.2.4 Gradle

Gradle[13] is an alternative to Maven that is more and more popular lately. The aim of the tool according to its makers is to combine the flexibility of Ant with the convention-over-configuration approach of Maven to offer the best of both worlds.

Gradle is implemented as a DSL that is based on the programming language Groovy.[14] Consequently, each Gradle script is at the same time a Groovy script that allows the developers to embed Groovy code directly into the build files and to implement practically any functionality in this manner. Groovy resembles Java. Every Java program is also a valid Groovy program. However, Groovy offers options for writing programs with substantially easier syntax.

The core concepts of Gradle are tasks and the dependencies between them. Based on these, Gradle calculates a directed, acyclic graph to determine which tasks have to be executed in which order. This is necessary since this graph can change through custom tasks, additional plug-ins, or the modification of existing dependencies.

Another advantage of Gradle is the support of incremental builds: Gradle only executes a build task when the source of the build has changed or when the task has never been executed before. Otherwise the task is left out and Gradle directly executes the next step in the execution graph. Especially in the case of large, complex builds this can save a lot of time. As mentioned above Gradle also offers the option to extend its functionality by embedding plug-ins. This allows the developer to embed support of other programming languages like Groovy, C++, or Objective-C in Gradle.

The syntax of Gradle is very compact in comparison to the two other "competitors," Ant and Maven, due to the absence of XML. The example from the section about Maven looks like Listing 3.2 in Gradle.

Listing 3.2 *A minimal Gradle build file*

Click here to view code image

```
apply plugin: 'java'

archivesBaseName = 'my-project'
version = '1.0.0-SNAPSHOT'

sourceCompatibility = '1.8'
targetCompatibility = '1.8'

repositories {
  mavenCentral()
}

dependencies {
```

```
    testCompile "org.junit:junit:4.11"
}
```

Concerning default settings—for example, with regards to which Java version is used—Gradle also behaves more intelligently than Maven. It uses as a standard setting the Java version with which Gradle is also executed. If the above-mentioned build script is executed with a Java 8 Virtual Machine, the information regarding the language version can be omitted. However, in this case the build is dependent on the installed Java version. Therefore, the version should be explicitly set to ensure the reproducibility of the build. In addition, the property `archivesBaseName`, which determines the name of the generated archive, can be omitted. As a default setting Gradle uses the name of the directory in which the build manuscript is localized.

Since the available build commands differ depending on the employed plug-in and the tasks contained in the build, Gradle offers its own task in order to display all tasks currently available in a project:

Click here to view code image

```
> gradle tasks
…
Build tasks
-----------
assemble - Assembles the outputs of this project.
build - Assembles and tests this project.
buildDependents - Assembles and tests this project and all
projects that depend on it.
buildNeeded - Assembles and tests this project and all projects
it depends on.
classes - Assembles classes 'main'.
clean - Deletes the build directory.
jar - Assembles a jar archive containing the main classes.
testClasses - Assembles classes 'test'.
…
```

The command returns an overview of all defined tasks. The tasks can originate from plug-ins or can be defined directly in the build script or in other included scripts. This makes it easier for developers to become familiar with the build of a project.

The Gradle Wrapper

With regards to build automation, another feature of Gradle is very useful: the Gradle Wrapper. It allows the developer to configure the employed Gradle version in the build file, and thus to turn this and the associated

binary into a part of the project and to put it under version control, too. This offers the benefit that after checking out a project—regardless of whether on a build server or on the computer of a new developer—everything necessary to build the project is already present: the missing parts are loaded from the Gradle website or from an internal repository, as appropriate. Therefore, the tool does not have to be separately installed or maintained. Only the person setting up the Wrapper at the very beginning needs a local Gradle installation. Afterwards everybody else can just use the Wrapper. This way it can be ensured that everybody uses exactly the same tool for the builds. This guarantees that the build is fully reproducible.

To set up the Wrapper for a project, only the Wrapper task that is built into Gradle has to be executed in the project directory:

```
> gradle wrapper
```

After the call there are two scripts, `gradlew` and `gradlew.bat`, in the project directory to run the Wrapper on Windows as well as on Linux or Mac OS X. In addition, a new directory called `gradle` is created in which the actual Wrapper binary is located. These files can be included in version control. In that case a Gradle installation is immediately available to all developers. If the version is supposed to be explicitly administered in the project, the Wrapper task can be configured directly in the build file:

```
task wrapper(type: Wrapper) {
gradleVersion = '2.10'
}
```

This also makes a version update of the tool substantially easier since the version is configured directly in the build script. An update of the installed tool is not necessary, nor is it necessary to take extra care of an installation of Gradle on the Continuous Integration servers; the build there can also use the checked in Wrapper. The makers of Gradle recommend using Wrapper instead of a manual local installation.

3.2.5 Additional Build Tools

The examples in this chapter focus on established build tools in the Java field. However, there are of course many other tools for other programming languages. Some of the more widely used ones are, for instance:

- Rake[15] is written in Ruby and focuses on builds for this programming language. It follows the tradition of Make and defines rules for how to generate a certain target file from a specific source file—for instance, how to generate a compiled byte code file from a file with Java code. Since Rake can be extended with Ruby, it can in principle be used for all kinds of projects.
- Buildr[16] uses Rake and extends it so that it supports the typical requirements of Java projects. In line with this it provides definitions for the typical tasks found in the context of Java.
- Grunt[17] is a tool for JavaScript. It is basically just a task runner. However, Grunt is primarily used to run the typical tasks for the build of JavaScript projects such as code compression or testing. Grunt is written in JavaScript and therefore can also be extended by JavaScript.
- sbt[18] is meant for Scala, but can also be used for Java projects. As in the case of Gradle the definition of the build is effected in a special DSL that, however, is written in Scala. sbt enables builds and tests to permanently run in the background to get rapid feedback when the compilation or the tests do not work anymore.
- Leiningen[19] is written in Clojure and uses a DSL for defining builds. This DSL uses a declarative approach like Maven.

3.2.6 Choosing the Right Tool

The choice of the build tools plays a decisive role when it comes to implementing Continuous Delivery. Nevertheless, this decision has, of course, effects with regards to which challenges are solved in certain areas. Especially for newly starting projects one should consider in advance what the selected tool is supposed to achieve and whether it is possible to delegate tasks like the deployment to other tools. In principle, such tasks can be performed by a build tool—however, a tool specialized in deployment might still be the better choice. In any case, clear interfaces should be defined between different tasks—for instance, the generation of a well-defined artifact that then can be employed for the deployment.

The tools presented in this chapter have shown how differently builds can be implemented. Gradle provides a lot of freedom without forgoing the conventions that are known from Maven. This is offset by a higher risk of chaotic code because of the large degree of freedom. Therefore, if Gradle is

chosen, regular reviews and refactorings of the build scripts are very important. In the case of Maven, on the other hand, simple extensions and modifications become extremely difficult if they do not comply with the conventions implemented by Maven.

No matter which tool is selected in the end, it makes a lot of sense to research the existing options and extensions in detail to find out how development and build can be further improved.

Try and Experiment

1. Install the Git command line tool (http://git-scm.com/).
2. Port the module `user-registration-application` from the example project from Maven to Gradle. To start, check out the source code of the project on GitHub if you did not do so before:

```
> git checkout https://github.com/ewolff/user-registration-V2.git
```

Afterwards, download the current version of Gradle and install it on your computer. As a first step for porting the build to Gradle you can use the Build Init[20] feature. This allows you to rapidly generate a first version of the Gradle build:

```
> gradle init
```

Upon the call Gradle determines that a Maven POM is present in the current directory and attempts to read out all necessary information to the project from there. This works quite well for the example project. It can already be compiled without any errors via the generated Gradle script `build.gradle`.

```
> gradle clean build
```

The line about the use of the Maven plug-in can be deleted from the Gradle build file `build.gradle`. The Java version should still be adapted with the two parameters `sourceCompatibility` and `targetCompatibility`. The generated repository snippet can be abbreviated with the more handy `mavenCentral()`, as already shown in the example listing in this chapter. Upon a new build without a previous clean it is easy to see the incremental build. During its execution there will be the message “UP-TO-DATE” written next to each build task, and Gradle does not run the respective task again.

As next steps you can embed the Gradle plug-in of the Spring Boot project[21] into your build and install the Gradle Wrapper for the project. Try also to port one of the other modules of the example project to Gradle and to combine the two ported modules into one multi-module build.[22] Pay special

attention to which information you can transfer into the Gradle cross-project configuration (as in the case of Maven with the use of Parent POMs).

3.3 Unit Tests

Unit tests evaluate the functionality of a small part of software—a unit. Normally, they only address individual methods of a class or at maximum the functionality of an entire class. Dependencies on other classes are ignored in unit tests or are replaced for the test. Accordingly, unit tests are in clear and deliberate contrast to integration and acceptance tests which check software in its entirety and run within a complete environment.

Fortunately, creating unit tests is routine for most developers nowadays. Nevertheless, there are still projects which have only a very low test coverage – especially legacy projects. To successfully implement Continuous Delivery unit tests is an important fundamental. They constitute —in addition to the types of tests covered in the following chapters—a first safety net to ensure that code changes do not impair existing functionalities. The quality of unit tests is determined by the test coverage. The test coverage is a measure of the percentage of code lines a test runs through upon its execution and therefore of the percentage of code explicitly or implicitly evaluated by the test. When the test coverage is high, the unit tests can ensure that there are hardly any errors left in the application.

In comparison to other test types, unit tests offer a number of advantages for obtaining rapid initial feedback:

- They test a small, clearly defined functionality, and therefore are easy to understand for developers who have newly joined the project. Thus, they can serve as a form of documentation for the developer team.
- They can be run in complete isolation and without additional dependencies. There is no need for a specifically adapted runtime environment. Thus, unit tests can be run on a developer computer or, in the context of Continuous Integration, on the Continuous Integration server.
- By simulating (“mocking”) external dependencies they run very quickly.
- The tests focus on a clear functionality. Therefore, they are well suited for delivering first rapid feedback in the context of a delivery pipeline.

- Mocks are also the reason why unit tests evaluate only one unit in isolation: The failure of a test can only be due to a problem in one single small unit.

For Continuous Delivery Unit tests are excellently suited as fundamentals for the test portfolio. This idea is further discussed in the context of the test pyramid (section 4.2). In this context, runtime and stability are especially important points arguing for unit tests. Unit tests are normally executed in the range of tenths of seconds up to a few seconds. Thus, they are very useful to developers for ensuring that none of the existing functionalities are broken during development. The developer can simply quickly run the unit tests after each change. In particular, as automated tests in the commit phase of a Continuous Delivery pipeline they provide very fast feedback.

Running unit tests during a normal build is activated by default by both Maven and Gradle. This ensures that for each artifact at least the unit tests have been run in order to guarantee quality.

3.3.1 Writing Good Unit Tests

Aside from testing, unit tests can fulfill an additional purpose by serving as documentation for developers. When they are written clearly and with a strong focus, the functionality of a tested class will be easy to figure out just by reading the unit test.

To ensure the necessary good readability of the test code, many teams have adopted the “arrange-act-assert” convention. This structure divides the tests in three areas:

- **Arrange**
  First, the test code defines the behavior of the mocked dependencies and prepares the test data.
- **Act**
  Next, the function to be tested is executed.
- **Assert**
  Finally, the test evaluates whether the expected result was returned with respect to whether the expected interactions with the dependencies took place.

The schematic structure of the tests is similar to the Given-When-Then schema that has been established in the context of Behavior-Driven Development (BDD). In section 4.6 an example using the BDD framework JBehave is depicted.

The work for the developers can be facilitated not only by following this simple sequence schema but also by providing the test methods with names that immediately give an idea of which kind of behavior is expected in the test. A simple unit test for the example application can look, for instance, like that shown in Listing 3.3.

Listing 3.3 *Unit test with Arrange-Act-Assert convention*

Click here to view code image

```
@RunWith(MockitoJUnitRunner.class)
public class RegistrationServiceUnitTest {

    @Mock
    private JdbcTemplate jdbcTemplateMock;

    @InjectMocks
    private RegistrationService service;

    @Test
    public void registerNewUser() {
        // arrange
        User user = new User(
          "Bastian","Spanneberg",
"bastian.spanneberg@codecentric.de");
        // act
        boolean registered = service.register(user);
        // assert
        assertThat(registered, is(true));
        verify(jdbcTemplateMock).update(
           anyString(),
           eq(user.getFirstname()),
           eq(user.getName()),
           eq(user.getEmail()));
    }

}
```

However, this test uses the normal `RegistrationService`. `JdbcTemplate` normally serves to access the database and is used by `RegistrationService`. But in the example a mock is used instead of `JdbcTemplate`—recognizable

due to the annotation `@Mock`. With the annotation `@InjectMocks` the mock is injected into the `RegistrationService`. Initially, a customer is created as a test object. This is the Arrange phase of the test. Then the customer is registered (Act), and subsequently it is evaluated whether the customer was successfully registered. Finally, during the Assert it is also examined whether the expected calls to `JdbcTemplate` have occurred.

As a unit test framework JUnit[23] is used in the example; this framework is used in numerous Java-based projects. A widely used alternative is the TestNG[24] framework. For generating the mocks—that is, for the simulation of dependencies that are not part of the test—Mockito[25] is used. As can be seen in the example, in the current version of Mockito the mocks do not have to be generated programmatically by hand; the developer can take care of this via an annotation. The object to be tested is likewise annotated. Mockito injects the generated mocks via reflections into the tested object.

Testing calls on generated mocks, like the call to `JdbcTemplate` in the example, should be omitted for most unit tests. Instead it is sufficient to check the return value of the method. The test should know as little as possible about the internal matter of the method. The way the test is implemented in the example, it will fail if the database is accessed in a different manner than with `JdbcTemplate`. Therefore, it should only be examined whether the user was successfully registered. This can be ensured by the return value of the register method—or by reading out the new user from the database. In some cases—for example when testing and refactoring legacy code or when testing methods without return values—the use of `verify()` can in fact make sense. Then the correct behavior of the logic can be ensured in this manner.

3.3.2 TDD—Test-Driven Development

In the optimal case developers have already written the tests before they start implementing the function. This approach is known as Test-Driven Development, abbreviated TDD.

In the case of TDD the developers usually proceed in an iterative manner. Initially, they express in a test the expected behavior they want to implement. Since at this point the implementation has not been done, the test fails of course. It is “red.” Subsequently, the developers begin with implementing the function. The goal is that the test runs successfully and thus turns “green.” Afterwards they refactor the test as well as the

implemented application code if necessary. If these cleanup efforts cause errors, the associated test turns “red” and the game starts over again. Accordingly, the test should be run over and over again. As a handy term for this cycle the expression “red—green—refactor” has been coined (Figure 3.2).

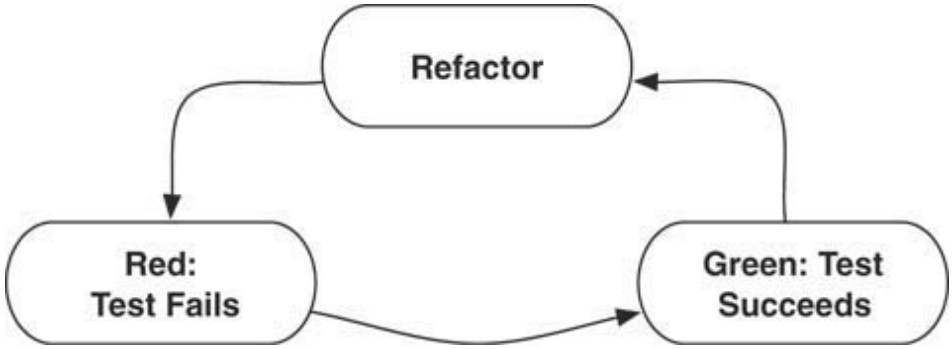

Figure 3.2 *Red—Green—Refactor*

Initially, this approach is not that easy for many developers because, for example, they do not know yet how they are going to implement a function, which classes are concerned, and how all parts should interact. They only want to write tests when all this information is complete. However, this is one of the great advantages of TDD. TDD trains developers over time to perform changes in ever smaller increments. Since the developers always only want to get the next test green, they do not even have another option than to advance in small increments. In the end this approach is often simpler and more productive since the developers do not get lost in complex considerations and designs, but only make the next simple step.

Since tests are always written prior to each implementation, a high test coverage is guaranteed and therefore a high level of safety against regressions, which can also be examined in the course of the Continuous Integration build. By using IDE plug-ins such as Infinitest,[26] which, upon a code change, immediately executes the associated tests, the benefit for the development process can be increased even further. The developers no longer need to keep in mind that they should test all the time. The plug-in is available for Eclipse and IntelliJ.

3.3.3 Clean Code and Software Craftsmanship

In the past few years the Clean Code and Software Craftsmanship movements have gained considerable popularity. Both have in common that

their members intensively deal with topics concerning clean software design as well as different test types and approaches. Code Retreats[27] and Coding Dojos allow participants to learn and try out TDD or other techniques.

Try and Experiment

- Assemble some of your colleagues and take some time (about one to two hours) together to hold a TDD Coding Dojo. To do so you need a Kata. This is a small contained task where a certain functionality is implemented. An example is the String Calculator Kata[28] by Roy Osherove.
- Perform the exercises in groups of two on one computer and take turns with writing the code. Try to host such exercises in your company more often to increase the use of Test-Driven Development. You can contrive your own Katas or do more demanding exercises, for example, by using the Taking Baby Steps[29] approach.
- Inform yourself about the SOLID principles.[30] Consider how their application affects the testability and readability of code and whether their use favors TDD.

3.4 Continuous Integration

Once the fundamentals of the build are implemented, it is imperative to execute the build in an automated manner continuously upon each code change or at least shortly after important changes. In the process, changes introduced by different developers are constantly integrated. This task is done by a build server, which checks out the current software version and triggers the build with the build tool.

This approach—known under the name Continuous Integration (CI), which was coined by Kent Beck and Martin Fowler—makes it possible to recognize errors in the software much faster than in cases where the software is built and integrated only nightly or even less frequently. Especially if the business logic is distributed over multiple components—potentially developed by different teams—not just building and testing but also integrating the current code of the different components does play a big role

in detecting incompatibilities early on. The use of Continuous Integration is nowadays well established in most companies and teams.

For implementing Continuous Integration a number of build servers are available. In the open source area the CI server Jenkins[31] is very popular and widely used. It originated as a fork from the Hudson[32] project. Go,[33] by the company ThoughtWorks, is also worth mentioning. Go was originally sold as a product, but has since been released as open source. In the commercial field Atlassian Bamboo[34] and TeamCity,[35] by JetBrains, are among the better known build servers. Most of the mentioned tools are usually operated on-site in a private data center. Of course, this means that employees must have the time and know-how to handle operations and maintenance. If a company does not want to make this investment, there are now a number of Cloud offerings, like Travis CI[36] and drone.io.[37] Continuous Integration then takes place in the Cloud so that extra software installations are unnecessary and time and effort for operations can be avoided. Both products offer free-of-charge versions for the creation of code from publicly accessible repositories—for example, on GitHub. For the creation of private projects, costs depend on the number of projects. Jenkins and Bamboo are likewise available as Cloud versions. Bamboo is offered directly by the manufacturer.

Prerequisite for the use of these offers is always a Cloud source code repository such as GitHub or Bitbucket to allow access for the Continuous Integration servers. For many medium and large companies this is no valid option for instance due to data security reasons so that on premise versions are the rule.

3.4.1 Jenkins

As mentioned already, Jenkins is the most widely used CI server. Accordingly, it will be discussed here in more detail. For Jenkins a build is a *job,* which runs in a *workspace*. This is a directory in the file system into which the build saves its files. In the standard installation, Jenkins offers freestyle jobs in which arbitrary steps can be combined in a build, and Maven jobs, which expect a Maven call as a central build step. Additional job types can be offered by plug-ins. A number of settings for determining the environment and the job behavior are common to all jobs. They can be modified via the Jenkins web surface:

- Parametrization

If necessary, build jobs can be parameterized. Modifiable data of a build can be easily passed in from outside without having to be held directly in the build script or in the build configuration. There the parameters can be read out as variables.

- Source Code Management (SCM)

  Here, the Source Code Management repository is configured from where the source code is checked out. The Jenkins standard installation offers support for CVS and SVN (Subversion). Support for other versioning tools like Git can easily be added via plug-ins. Depending on the used SCM here, specific configurations can still be performed.

- Build Triggers

  This defines the point in time at which the build is supposed to run:

  - Time-directed and independent of changes in the Source Code Repository.
  - Upon changes in the Source Code Repository.
  - After a successful completion of other build jobs.
  - When SNAPSHOT dependencies of the project were created on the same build server.

  Thereby the successful integration of multiple components can be tested. However, this variant is only available for Maven builds.

- Build Environment

  In this section, if necessary, the build environment can be modified. For instance, it is possible via plug-ins to ensure that the build always takes place on an empty workspace or that additional environment variables are read from files.

- Build

  This is a central part since the build commands are configured here. For Maven jobs this is the respective call, if necessary with a reference to the POM to be used. For freestyle jobs a number of build steps can be indicated and thus several tools can be combined in a build.

- Post-Build Actions
- Here tasks are defined which are supposed to be executed after the build. Examples are the initiation of additional jobs, the archiving of

built artifacts or test results, or an email notification in case of unsuccessful builds. Via plug-ins Jenkins can trigger many different actions.

Extension via Plug-ins

In spite of the standard functions Jenkins brings its full benefit only in combination with the numerous available plug-ins[38] that the Jenkins community offers. With their help Jenkins can offer many more features. Of course, with an increasing number of plug-ins the danger of incompatibilities between them increases. Therefore, the interplay of new plug-ins with already used ones should be carefully tested prior to installing. For example, a separate Jenkins instance can be used for testing plug-ins before they are rolled out on the production build server. As in the case of the code and the build scripts the configuration of Jenkins and of the used plug-ins as well as of the build jobs should be regularly reviewed to assure their quality. A number of useful plug-ins are mentioned in the following paragraphs.

SCM Sync Configuration Plug-in

This plug-in allows the developer to synchronize the configuration of Jenkins and of all jobs and views into version control. At the moment Subversion and Git are supported. In the standard settings the Jenkins system configuration, and all jobs, are placed under version control. Additional files in the Jenkins file system can be added manually. If the plug-in is active, a commit message is asked for upon changes to a versioned configuration and the change, together with the commit message, is checked into version control. In case of data loss the Jenkins configuration can be regenerated via version control.

Environment Injector Plug-in

The Environment Injector plug-in allows the developer to manipulate the environment variables of the build in various ways and at different phases of the build life cycle—for example before or after a check-out from the source repository or prior to the build. In addition, it will export the trigger of the build (manual, per SCM, via a user, or via an upstream build) as environment variables. These can be used, for instance, in scripts within a build to trigger specific actions depending on the different triggers.

Job Configuration History Plug-in

Since job configurations can rapidly become very complex, the use of this plug-in is strongly advised. It versions changes to the configuration of the build and allows the developer to compare versions in the Jenkins user interface in order to trace the changes over time. If errors have crept into the configuration, this plug-in is extremely useful in reverting to the last stable configuration.

Clone Workspace SCM Plug-in

This plug-in makes it possible to archive the current workspace after the build and to continue to use it in other builds as a starting point. Especially in combination with incremental build tools like Gradle, it can be very useful when the subsequent steps build on output of a previous build. This speeds up the build since now only those steps of the build are executed where the input in fact has been changed. Also, in environments with frequent changes, this plug-in can be helpful to ensure that all steps in the pipeline build on the same version state.

Build Pipeline Plug-in

A plug-in that deserves special mention is the Build Pipeline plug-in. As the name suggests, the plug-in serves to build up and visualize a build pipeline. The option that jobs trigger each other is the foundation for these pipeline views. Starting from the first job, Jenkins visualizes the chain of subsequent jobs. These might be, for instance, the execution of acceptance or load tests, or a deployment. In addition, the plug-in allows manually triggering subsequent builds as a new type of post-build action. Parts of a pipeline can thereby be started selectively upon request rather than in a fully automated manner. For example, a deployment onto a QA environment should not be executed for every build. This plug-in makes it possible to implement the coordination that is required for a Continuous Delivery pipeline (Figure 3.3).

Figure 3.3 *Screenshot of the Build Pipeline plug-in*

Amazon EC2 Plug-in

Amazon's Elastic Compute Cloud (EC2) is a Cloud service and offers servers for rent. The Jenkins plug-in allows, if necessary, the generation of new Jenkins slaves in the EC2 Cloud and the transferring of build jobs there. If the instances are not used over a longer time, the plug-in will destroy them again so that unnecessary costs are avoided. Of course, an Amazon Web Services (AWS) account via which the instances can be billed is a prerequisite for the use of this plug-in. If the company guidelines allow it, this plug-in offers a very simple possibility for scaling Jenkins when the load increases without having to build up the necessary infrastructure by yourself. In addition, the Cloud offerings allow for the processing of exceptionally high loads.

Job DSL Plug-in

In the case of an increasing number of jobs the manual configuration in the Jenkins web interface rapidly becomes difficult to understand and to change. There are often also jobs that are logically connected, and all have to be

adjusted together in the case of a modification such as a change of the repository. In addition, in large enterprises there is the question of how easily jobs can be restored in case of data loss (see also the SCM Sync Configuration plug-in). For these questions the Job DSL plug-in provides a solution. As can be expected from the name, it offers a DSL to create Jenkins jobs and views. DSL scripts with the descriptions of one or multiple jobs can be put in so-called seed jobs together with associated views. There, a DSL script from version control can also be referenced. If the job is executed, all elements described there are created or updated. Listing 3.4 shows a Job DSL script for the generation of two interdependent jobs and an associated pipeline view.

Listing 3.4 *Job DSL script for the example application*

Click here to view code image

```
job('user-registration-parent-build') {
  scm {
    git {
      remote {
        url('https://github.com/ewolff/user-registration-V2')
      }
    }
  }
  triggers {
    scm('H/15 * * * *')
  }
  steps {
    maven {
      goals('-e clean install -pl .')
      rootPOM('pom.xml')
      mavenInstallation('Maven 3')
    }
  }
  publishers {
    publishCloneWorkspace('')
  }
}

job('user-registration-build') {
  scm {
    cloneWorkspace('user-registration-parent-build')
  }
  triggers {
    upstream('user-registration-parent-build', 'SUCCESS')
  }
```

```
  steps {
    maven {
      goals('-e clean install')
      rootPOM('user-registration-application/pom.xml')
      mavenInstallation('Maven 3')
}
  }
  publishers {
    publishCloneWorkspace('')
  }
}

buildPipelineView('user-registration-pipeline') {
    selectedJob('user-registration-parent-build')
}
```

As you can see, the plug-in does not only support default features of Jenkins, but also widely used plug-ins like the already mentioned Clone Workspace or the Build Pipeline plug-in. The exact API of the individual elements can be comfortably researched online in the Job DSL API Viewer.[39]

Writing Your Own Plug-ins

Although the number of plug-ins available to the community is immense and most requirements are covered by one or the other plug-in, it might still be necessary to write your own plug-ins. Jenkins offers numerous entry points for developing individual extensions. A tutorial for the development of plug-ins can be found in the documentation.[40] In addition, it is a good starting point to study the source code of an already existing plug-in in order to get familiar with the development model.

3.4.2 Continuous Integration Infrastructure

When the use of Cloud variants or running the Continuous Integration server in a public Cloud infrastructure is not a possible option, there are some questions that have to be addressed when creating your own internally used Continuous Integration infrastructure:

- How do you organize the infrastructure?
- How is the infrastructure supposed to be maintained?
- Which requirements do you have for the infrastructure?

Often these questions are not asked or not considered when Continuous Integration is introduced, although they play an important role, especially in the long run. Wrong decisions can lead to a lot of extra work and trouble in the end.

It is unfortunately a quite frequent anti-pattern to set up the CI server on hardware that is "left over" at that moment. Often this hardware is already a bit older and not very powerful. Therefore, it is not of any use for other purposes. However, in doing so the developers misjudge the importance of the Continuous Integration server, which plays a central role in the delivery chain of the software. Failures or errors of this server will bring down the entire process. It is a central task of the CI server to deliver rapid feedback regarding the correctness and quality of the code. If its hard drive is smaller and slower than the hard drives in the developers' computers or the network connection is too slow, the server easily turns into a bottleneck and stalls the entire project – causing significant follow-up costs.

Another question to ask concerns the structure and the scaling of the CI infrastructure: Is there a single master for the coordination of the jobs that delegates the builds to the associated slaves?

Such a structure (Figure 3.4) has the benefit that all configurations take place on the master. This facilitates the backup of configurations and jobs. On the other hand, an increasing number of jobs and users will also increase the load and therefore the risk of a breakdown of the master. When the master fails, the entire build infrastructure becomes unavailable. Also, the administration and coordination of the used plug-ins play a role in this scenario. Different teams want to use different plug-ins, and thus the risk of mutual incompatibilities rises.

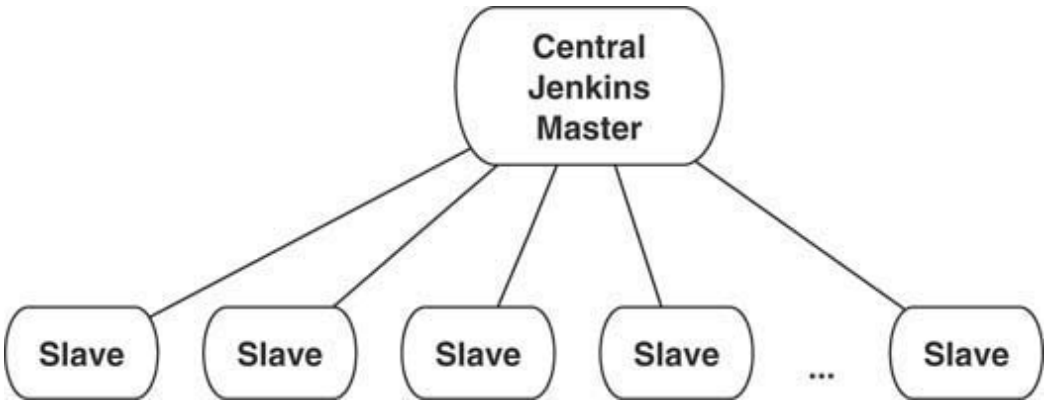

Figure 3.4 *Centralized CI setup*

A different approach is to provide each team with its own master and to supply the associated slaves when necessary (Figure 3.5).

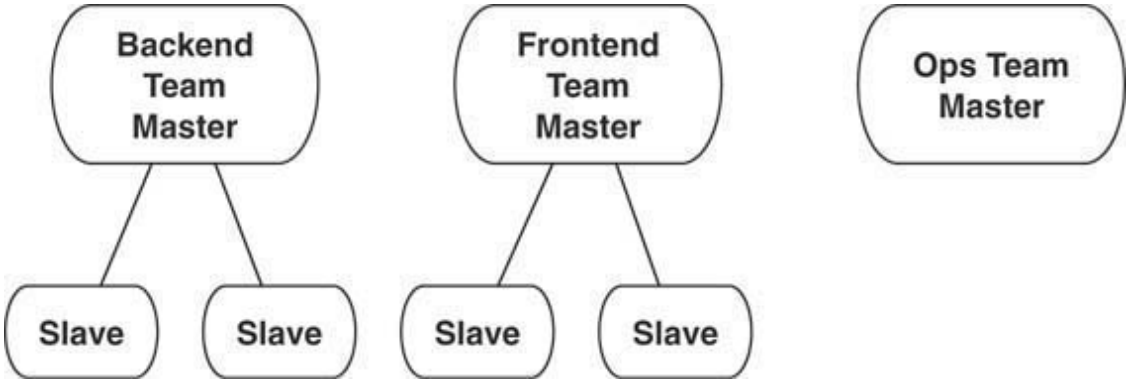

Figure 3.5 *Centralized CI setup*

This renders the teams independent of each other and makes them free in their choice of plug-ins. On the other hand, every master has to be administered and maintained. In addition, in this scenario there is no longer an easy way to perform a central backup.

Again it has to be decided which variant entails more benefits in a concrete scenario and which disadvantages just have to be dealt with later on.

3.4.3 Conclusion

When building up a Continuous Integration infrastructure there are many aspects to consider. Cloud solutions provide very low initial hurdles and costs and offer a good basis for open source projects. In combination with non-public repositories they might also be an option for enterprises. However, for many companies the transfer of business-sensitive data—such as software source code—into a Cloud is no option. In those cases the knowledge and infrastructure for Continuous Integration has to be built up within the company, and the infrastructure needs to be operated locally. In such a scenario, it is very advisable to think about the future early on and to devise the approach and the actual infrastructure in such a way that it will be able to cope with future growth and changing requirements In any case, quick and dirty solutions should be avoided. The Continuous Integration server represents the first central component for Continuous Delivery and therefore it should be handled with the necessary attention and diligence.

Try and Experiment

Set up a Continuous Integration job for the example project. You can use the Vagrant box that exists there for it.

1. Within the example project change into the directory `ci-setup`.[41] Execute the command `vagrant up` there. When you are starting this box for the first time, it might take some minutes for the process to complete. After the start has been completed, the Jenkins server that was set up there can be accessed under http://localhost:9191 on your local computer.
2. To be able to execute the Maven build of the project, you first have to install a Maven version in Jenkins. To do so follow the link "Manage Jenkins" on the start page and then the link "Configure System." In the section "Maven" you can configure a Maven version, which Jenkins will download directly from the Apache download site, if necessary.
3. Now you can set up the actual build job. As a starting point for a build you need a source code repository from which Jenkins can download the code. Initially, you can use the repository address of the example project for this (https://github.com/ewolff/user-registration-V2.git). If you later want to create your own changes in Jenkins, you should create a fork of the example project (see https://help.github.com/articles/fork-a-repo). On this fork you can perform changes which will then trigger builds in Jenkins. To create a fork you will need your own GitHub account. Since the example project is built in Maven, you should set up a Maven job for it. To do so you can click on the point "New Item" at the left upper corner in Jenkins and there select the option "Build a Maven Project."
4. In the job configuration under "Source Code Management" you first have to state the address of the underlying repository from which Jenkins should check out the code. To do so you have to select Git for version control and enter the repository URL that was given above. Under "Build Triggers" you have to indicate when Jenkins is supposed to build the project. The best option here is "Poll SCM." In that case Jenkins examines at regular intervals whether there are changes and triggers the job if it finds changes. In the associated configuration you have to indicate a Cron-expression, which determines the examination

interval. "*/10 * * *" checks every ten minutes whether there are changes in the version control.

5. Finally, under "Build" you configure the path to the Maven POM and the Maven goals that are supposed to be executed during the build. A sensible option is "clean install." In that case all artifacts of an old build are removed, and the project is built entirely anew.
6. Save the job and then click on "Build now" to trigger a build in Jenkins.
7. For the same project also set up a freestyle job. This is an alternative to the Maven job that you have already implemented. For this you first have to add the new step "Invoke top-level Maven targets" under "Build" to be able to build a Maven project. Analogously, you could also embed additional steps there, such as the execution of shell scripts or of an Ant target.
8. The Build Pipeline plug-in that was mentioned in the text is already installed in the example Jenkins server. As a starting point for additional experiments, create a pipeline view. To do so use the "+" tab on the main page and choose the job you just created as the starting point. In the course of the book you can extend this pipeline with new steps, such as the execution of load tests, by adding new jobs that are triggered by your initial build job.

Once you have become familiar with the described features, try to create the jobs and views you generated with the help of a Job DSL script.

- To do so, create a freestyle job and add a step "Process Job DSLs" under Build. There, select the option "Use the provided DSL script." You will obtain a text field where you can enter your script. Use the Job DSL API Viewer that was mentioned in the text to become familiar with the possible commands.
- Experiment also with the Cloud offering TravisCI. You can register there with your GitHub account. If you already created a fork of the example project in the course of the previous exercises, the project should already be built in Travis, since the TravisCI configuration `.travis.yml` is already provided. Find out which options Travis offers and set up your own project, which you then build with Travis. Travis offers tutorials for

different programming languages. The tutorial for Java can be found at http://docs.travis-ci.com/user/languages/java/.

3.5 Measuring Code Quality

How small functional units can be tested with unit tests has already been discussed. Other types of tests such as acceptance and performance tests are the topic of later chapters. However, there is also a whole other dimension of software quality which can likewise be measured in an automated manner: the code quality.

Bad software mostly arises in a slow process. The follow-up costs associated with bad software accumulate little by little. In particular, the necessary effort for software changes will increase over time, and the speed with which new features can be delivered will decrease. Therefore, the code quality should also be continuously monitored when establishing Continuous Delivery and—as in the case of other types of tests—stop the Continuous Delivery pipeline if the software does not meet the requirements. Different standard metrics can be used to judge the quality of the code.

For example, the complexity of a class or of a method can be determined by evaluating how many possible execution paths there are. The more branches exist, the greater will be the number of possible execution paths, and consequently the complexity for understanding or testing the code goes up. Increasing complexity translates at the same time into worse maintainability of the code and a larger risk due to hidden errors.

In addition, in most cases the code coverage, that is, the coverage of the code by unit tests, is also measured. Each code line that is executed in a test is considered covered. In terms of code coverage a distinction is made between line coverage and branch coverage. For line coverage the number of executed lines in relation to the total number of code lines in the software matters. For calculating the branch coverage the number of executed branches is determined. In the following code example, for instance, the line coverage would be 80% if `myCondition` is `true` in the relevant test. However, the branch coverage would only be 50% since only one of the two possible paths is executed.

```
If ( myCondition ) {
  statement1;
  statement2;
```

```
    statement3;
    statement4;
} else {
    statement1;
}
```

Therefore, to be able to make reliable statements concerning the code coverage, both values should always be considered.

In addition to these classical metrics it is also common practice to use static code analysis to ensure that certain guidelines are complied with regards to the development style. For instance, it can be verified that there are no `// TODO` comments left in the code, and it can be checked whether conventions for the naming of classes or packages were adhered to or whether the stack trace is conserved upon the rethrowing of exceptions.

In the Java field a number of tools have been established that help to determine and examine these values and rules. Some of the more widely used tools are: Checkstyle,[42] FindBugs,[43] and PMD[44] for static code analysis and Emma,[45] Cobertura,[46] and JaCoCo[47] for measuring code coverage. These tools can either be triggered via the build script of the software or by using plug-ins within the Continuous Integration server.

3.5.1 SonarQube

Thus, to really measure software quality, multiple tools have to be installed and used simultaneously. As an alternative to the use of the tools that were just introduced, the software SonarQube[48] (formerly known just as Sonar) has been established in many companies—primarily in the Java field, but increasingly for other languages as well. SonarQube combines numerous metrics with some of the above-mentioned tools as well as with a graphical interface for administering rules and for evaluating the results (Figure 3.6). For individual projects individual dashboards and sets of rules can be configured. Also, in the case of SonarQube, the support of additional languages such as JavaScript or PHP can be supplemented via additional plug-ins, and the software can be extended to encompass other additional functions.

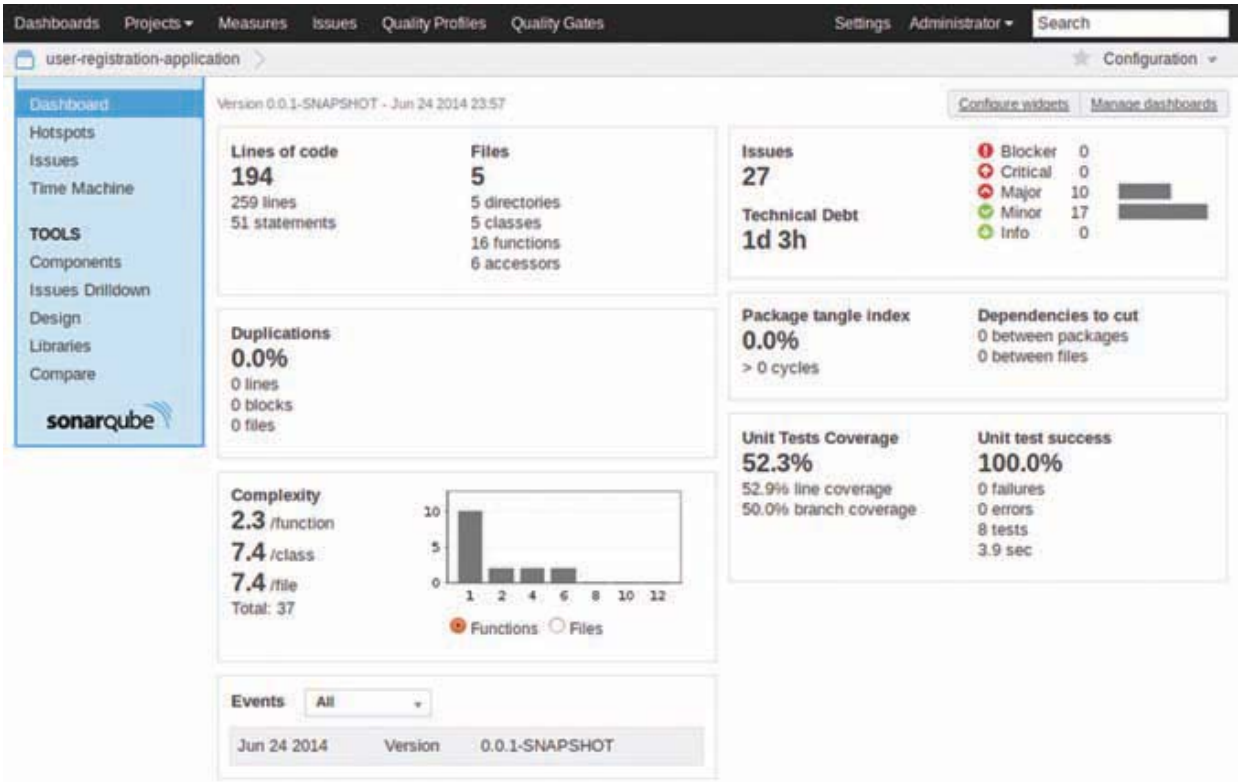

Figure 3.6 *SonarQube dashboard for the example application*

Integration in the Pipeline

SonarQube can either be integrated into the build, configured, and called via appropriate plug-ins, or its execution can be integrated via a plug-in in the Continuous Integration server.

For Maven[49] as well as for Gradle[50] plug-ins are available that make the integration very easy. In both cases a number of properties have to be set in the build file, which provides the necessary credentials to the plug-in and tells it at which address the SonarQube instance and its underlying database can be found (Listing 3.5). Figure 3.6 shows which properties have to be set for the Maven plug-in. User name and password should rather be taken care of in the Maven `settings.xml` file on the host on which the analysis is executed, and not directly in the POM. `settings.xml` contains settings that are specific for a server. Otherwise the POM cannot be checked into version control anymore without all developers knowing the password for the SonarQube server. Additional information for configuring the SonarQube plug-ins for Jenkins can be found in the documentation.

Listing 3.5 *Configuration for SonarQube with Maven*

Click here to view code image

```
<properties>
  <sonar.jdbc.url>
jdbc:mysql://localhost:3306/sonar?
autoReconnect=true&useUnicode=true&characterEncoding=utf8
  </sonar.jdbc.url>
  <sonar.jdbc.username>sonar</sonar.jdbc.username>
  <sonar.jdbc.password>sonar</sonar.jdbc.password>
  <sonar.host.url>http://localhost:9000</sonar.host.url>
</properties>
```

Now the code analysis can be triggered with a simple Maven command:

```
> mvn clean install sonar:sonar
```

From the SonarQube instance the plug-in fetches the rules that have been configured there, runs the code analysis, and saves the results in the database. The integration in a Gradle build follows the same schema as in Maven.

Alternatively to embedding into the build script, the integration can also happen directly on the CI server via a plug-in. For Jenkins and Bamboo suitable plug-ins are available. For the Jenkins plug-in[51] the same configuration as described above has to be performed at a central location in the configuration menu of the CI server. It is also possible to maintain multiple instances there in parallel. The plug-in extends Jenkins by an additional SonarQube build task and by a SonarQube post-build action by which the analysis can be triggered.

Which variant is chosen in the end also depends on the other tooling. If a plug-in for the CI server is available, the data and configuration will be better stored there. In addition, in this manner it can be ensured that up-to-date information about quality is always available. Otherwise the approach that has to be taken is an integration into the build—however, in principle, every project should have a CI server nowadays. In the case of integration into the build it is important to make sure to maintain access data outside of version control in a well-secured location.

In any case the employment of SonarQube or similar tools for the continuous measurement of code quality is a decisive factor since this is an additional important basis for ensuring the speedy development of features.

Try and Experiment

- Experiment with the different ways of integrating SonarQube into your build pipeline. On the Continuous Integration Vagrant Box from the example project a SonarQube server is already installed. You can access it under http://localhost:9393 after a successful start. On the Jenkins server the SonarQube plug-in for the integration is already installed. You can access Jenkins at http://localhost:9191. With the access "data admin/admin" you can log into the SonarQube server as administrator and under "Quality Profiles" you can change the existing rule set or create a new one.
- Try first to integrate the analysis via the build (either Maven or Gradle) and to execute it in the course of your build job on the Jenkins server.
- Afterwards change to the integration via a Jenkins plug-in. You can enter the access data to your SonarQube server in Jenkins under "Manage Jenkins > Configure System." Subsequently, you can let the analysis run as part of your build or separately within an extra job on Jenkins. This approach allows you to execute the analysis in parallel to acceptance tests in order to shorten feedback times.
- To further increase the usefulness of SonarQube, the analysis should fail when certain thresholds are exceeded. For these thresholds SonarQube offers the definition of so-called *quality gates*. To also let the associated build job fail, you have to install the Build Breaker plug-in in SonarQube. To do so log in as administrator ("admin/admin") and then click in the right upper corner on the link "Settings." In the menu that appears go into the update center and from there into the tab "Available Plug-Ins." There you can select and install the Build Breaker plug-in. Afterwards, you have to link your project to a quality gate in the section "Quality Gates." To start out, you can use the already existing quality gate "SonarQube way." Later on, think about sensible values for your own quality gate that fits to a project you are currently working on.

3.6 Managing Artifacts

An aspect that should not be underestimated—neither for professional and smooth development nor for a well-functioning Continuous Delivery pipeline—is the administration and handling of the generated artifacts.

Concomitant with the widespread use of Maven, the term “artifact repository” has been established. These play a central role in Maven. Dependencies are downloaded from repositories, while created artifacts, on the other hand, are provided via repositories. The repository is defined in the POM. There are public repositories. However, repositories can also be installed on-site. The repositories that are installed on-site in companies contain the build results of local projects and often serve additionally as proxies for external artifacts from public repositories. In this manner the build process can be decoupled from the availability of external repositories —because in the end it is not guaranteed that the public repositories will still contain the libraries in the right versions a few years down the road. It can also be determined which external sources are allowed to be used at all. Therefore, the internal repositories play a central role for the management of internal as well as external artifacts. This holds true for the developers as well as for the central build process.

Especially when setting up a Continuous Delivery pipeline the artifact repository plays an important role since it serves as a central instance for all artifacts that are required in the delivery process.

As Figure 3.7 shows, the artifact repository is at the center of all activities of the pipeline. In the commit phase the artifact to be delivered is generated. In doing so dependencies to other internal projects via snapshot or release repositories are dissolved. Dependencies on external projects are dissolved via the repository proxy during building. In the course of the build all necessary quality criteria are examined: successful tests, coverage, and coding guidelines. Afterwards the artifact is published on the repository.

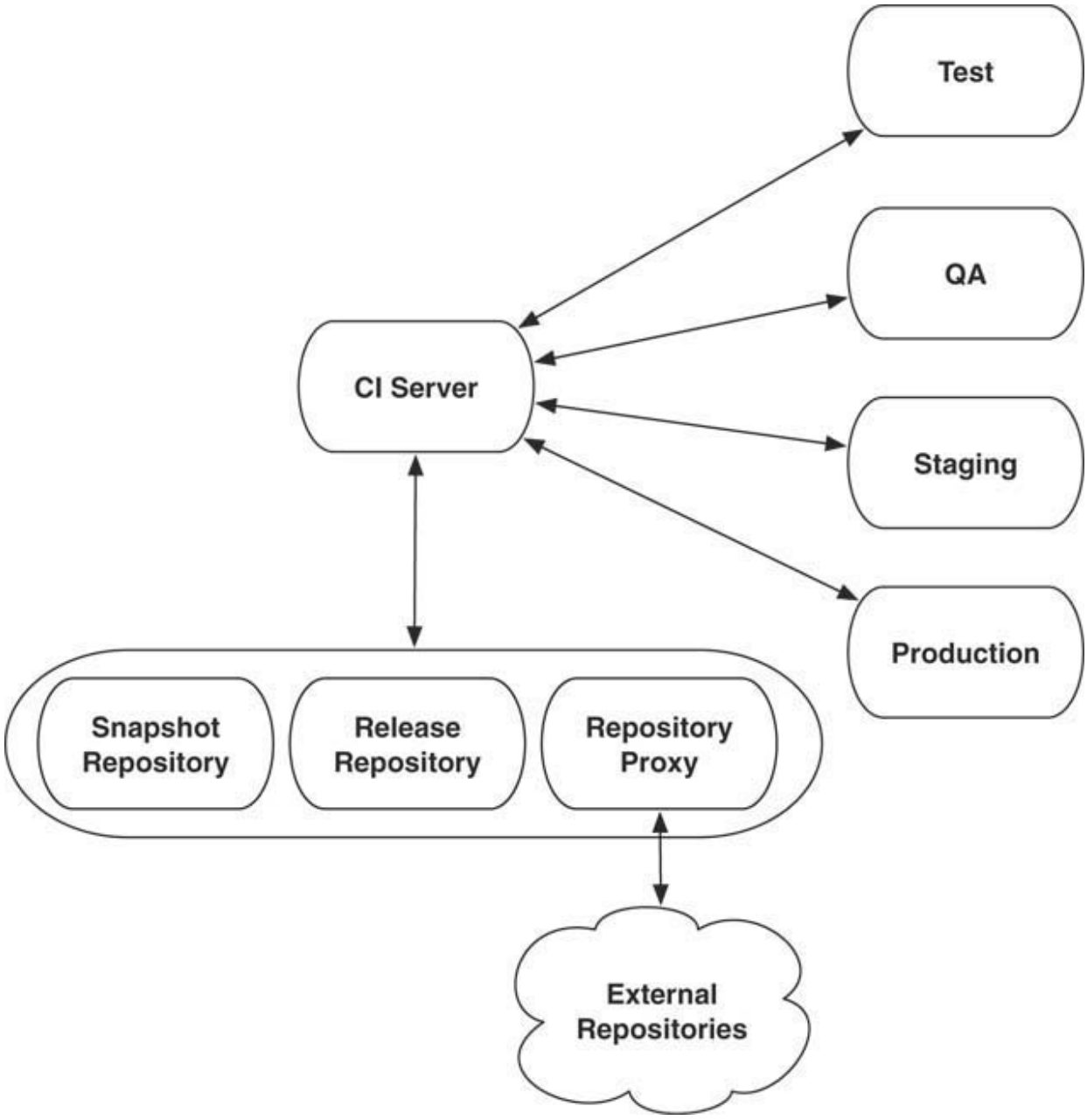

Figure 3.7 *A repository is the heart of the delivery infrastructure*

Subsequent steps, like the deployment on a test or QA environment or, finally, the deployment on staging and production environments, use the artifacts that have been published there. Therefore, the artifact repository serves as the interface between the different steps of the process.

To also represent in the repository how the software is promoted through the pipeline, one can use either metadata of the release, or several repositories that individually represent one of the phases of the Continuous

Delivery pipeline. Then the software can be transferred into a new repository when it has successfully passed a test. This is also called staging.[52] However, this functionality is often only available in the commercial versions of the repository software. But at least for Jenkins and the repository server Nexus there is a Jenkins plug-in[53] that represents staging by transfer into other repositories.

The artifacts can be JAR, WAR, or EAR files, or artifact types that are closer to operations such as Debian or RPM packages, which likewise can be created in the pipeline. Many of the popular repository servers support such artifact types, however, in part only as commercial features. Such artifacts have the advantage that they are very familiar to system administrators and can be very easily integrated into automated rollout processes. Also—and especially for such artifacts—there are alternative solutions with which complex Linux setups can be provided with Debian or RPM packages.

Whether a tool coordinates all necessary steps like the CI server in Figure 3.7 or whether the pipeline ranges across multiple tools does not matter in the end. Also, commercial solutions in the area of release and deployment automation offer adaptors for interacting with the widely used artifact repositories.

3.6.1 Integration into the Build

Integrating the repositories into the software build is very easy with Maven as well as with Gradle. In both cases, there are two different types of repositories:

- From one of the repositories dependencies will be resolved. Therefore, it contains the libraries from external sources and the results of all internal projects.
- The other repository serves to publish the generated artifacts. So the result of the build is stored in this repository.

Listing 3.6 *Definition of a repository in a Maven POM*

Click here to view code image

```
...
<repositories>
  <repository>
    <id>spring-milestones</id>
```

```
      <url>http://repo.springsource.org/milestone</url>
    </repository>
  </repositories>
  …
  <distributionManagement>
    <repository>
      <id>internal-release</id>
      <name>Internal Release Repository</name>
      <url>http://localhost:9292/artifactory/libs-release-
  local</url>
    </repository>
  </distributionManagement>
```

With the information given in Listing 3.6, a call of

```
> mvn deploy
```

will execute a build of the artifact, generate the Java archives (JARs) and then upload them into the repository, which is defined in the section `distributionManagement`. Normally, account credentials also have to be configured since not everybody should be able to upload artifacts. Classically, the access data are not maintained in the project POM, but in a central location identified, for instance, in Maven's `settings.xml` on the Continuous Integration server. Details regarding the configuration of access credentials can be found in the Maven documentation.

Gradle borrowed a number of ideas from Maven with regards to handling repositories so that the comparable configurations look similar as shown in Listing 3.7.

Listing 3.7 *Definition of a Gradle repository*

Click here to view code image

```
  repositories {
    mavenLocal()
    mavenCentral()
    maven { url 'http://repo.springsource.org/milestone' }
  }
  …
  uploadArchives {
    repositories {
      maven {
        url = 'http:// 192.168.33.22:8081/artifactory/libs-release-
  local
      }
    }
  }
```

The appropriate Gradle command for uploading the created archives is then:

```
> gradle uploadArchives
```

Gradle can not only handle Maven repositories, but also the somewhat older Ivy repositories, as well as simple network drives or similar things. It is by far more flexible and also easier to extend with regards to repositories than Maven. The local Maven repository and the central public repository Maven Central are both implicitly configured in Maven. In the case of Gradle they have to be explicitly configured since other repositories could also be used in principle.

3.6.2 Advanced Features of Repositories

Popular repository solutions like Artifactory[54] and Nexus[55] offer substantially more features than just storing and providing Java artifacts. Among the advanced features are:

- REST interfaces for calling and manipulating the data.
- A sophisticated user and right management.
- The option to examine published artifacts with regards to license questions.

Often additional artifact types such as Ruby Gems or Debian and RPM packages can be administered in the same repository. Therefore, they are also interesting for multi-language projects and offer a central site to administer all artifacts generated in the company, no matter whether they are packages from operations or artifacts from software development. All these artifacts can be distributed to the relevant systems.

Try and Experiment

- In the Vagrant Box for the Continuous Integration setup, the artifact repository Artifactory is also installed. Once the box has started successfully, you can access it at http://localhost:9292. Extend the build script from the example project so that the generated artifacts are published into the repository. The code snippets in the text provide a good starting point for this. Experiment with defining separate accounts for the upload of artifacts. To do so log in as administrator with admin/password and setup the necessary accounts. Store the account credentials for the Continuous Integration server for use with Maven.
- Find out whether an artifact repository is used in your company. If yes, who is using it? Only the teams in software development or also in operations? Are guidelines for the use of external artifacts represented via this repository?
- When you are experimenting with your own project, which, for example, has to be delivered on an applications server, you can try out the integration of Artifactory and Jenkins. For instance, you can use the Repository Connector Plug-in[56] to load a WAR or EAR file from your repository in an extra Jenkins deployment job and then to transfer it—for example, with the SCP plug-in[57]—to the host where it is supposed to run in the end.

3.7 Conclusion

This chapter should have provided you with an idea of which topics in the fields of development, build, and Continuous Integration play a role in the context of Continuous Delivery. The choice of tools, techniques, and methods is important for setting up a good basis for facilitating later automation.

Especially, it should have become clear that—exactly like the actual program code—code and logic with regards to build and integration have to be checked again and again and have to be updated if they no longer comply with current demands. Also, in this field legacy code can pose a problem. In fact, it becomes especially harmful when problems occur because in the worst case the entire software delivery chain stands still and thus all teams are blocked.

It is important to make sure that the knowledge about build and Continuous Integration setup is not confined to a single person or a small circle. Every developer should understand what happens during the build and should also keep this knowledge in mind during development. Tools like SonarQube can be very useful for all team members and can bring additional transparency if they are used in a well-planned and continuous manner and are accessible for all team members.

In any case the efforts undertaken in this area lay the foundation for a successful Continuous Delivery pipeline. Therefore, quality, maintainability, and regular maintenance of the build logic and the associated infrastructure are essential.

Endnotes

1. http://ant.apache.org/
2. http://maven.apache.org/
3. https://gradle.org/
4. http://ant-contrib.sourceforge.net/tasks/tasks/index.html
5. http://ant.apache.org/ivy/
6. http://maven.apache.org/components/plugins/maven-jar-plugin/
7. http://maven.apache.org/
8. http://maven.apache.org/maven-release/maven-release-plugin/
9. http://www.mojohaus.org/versions-maven-plugin/
10. http://maven.apache.org/scm/maven-scm-plugin/
11. https://axelfontaine.com/blog/maven-releases-steroids.html
12. https://axelfontaine.com/blog/final-nail.html
13. https://gradle.org/
14. http://www.groovy-lang.org/
15. https://github.com/ruby/rake
16. http://buildr.apache.org/
17. http://gruntjs.com/
18. http://www.scala-sbt.org/
19. https://leiningen.org/

20. https://docs.gradle.org/current/userguide/build_init_plugin.html
21. http://docs.spring.io/spring-boot/docs/current/reference/html/build-tool-plugins-gradle-plugin.html
22. https://docs.gradle.org/current/userguide/multi_project_builds.html
23. http://junit.org/junit4/
24. http://testng.org/doc/index.html
25. http://site.mockito.org/
26. http://infinitest.github.io/
27. http://coderetreat.org/
28. http://osherove.com/tdd-kata-1
29. http://blog.adrianbolboaca.ro/2013/03/taking-baby-steps/
30. https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)
31. https://jenkins.io/index.html
32. http://hudson-ci.org/
33. https://www.gocd.io/
34. https://www.atlassian.com/software/bamboo
35. http://www.jetbrains.com/teamcity/
36. https://travis-ci.org/
37. https://github.com/drone/drone
38. https://wiki.jenkins-ci.org/display/JENKINS/Plugins
39. http://jenkinsci.github.io/job-dsl-plugin/
40. https://wiki.jenkins-ci.org/display/JENKINS/Plugin+tutorial
41. https://github.com/ewolff/user-registration-V2/tree/master/ci-setup
42. http://checkstyle.sourceforge.net/
43. http://findbugs.sourceforge.net/
44. https://pmd.github.io/
45. http://emma.sourceforge.net/
46. http://cobertura.github.io/cobertura/
47. http://www.eclemma.org/jacoco/
48. https://www.sonarqube.org/

49. https://docs.sonarqube.org/display/SCAN/Analyzing+with+SonarQube+Scanner+for+maven
50. https://docs.sonarqube.org/display/SCAN/Analyzing+with+SonarQube+Scanner+for+Gradle
51. https://docs.sonarqube.org/display/SONAR/Analyzing+with+SonarQube+Scanner+for+Jenkins
52. https://www.jfrog.com/confluence/display/RTF/Build+Integration#BuildIntegration-ReleaseManagement
53. https://github.com/jenkinsci/artifact-promotion-plugin/tree/master
54. https://www.jfrog.com/artifactory/
55. http://www.sonatype.org/nexus/
56. https://wiki.jenkins-ci.org/display/JENKINS/Repository+Connector+Plugin
57. https://wiki.jenkins-ci.org/display/JENKINS/SCP+plugin

Chapter 4. Acceptance Tests

4.1 Introduction

The ultimate requirement is an automated acceptance test—because then it can be clearly decided whether the requirement is correctly implemented or not. However, acceptance tests are just one type of test. Section 4.2 discusses the test pyramid. This pyramid is a model showing which kinds of tests are supposed to be present, and to what extent, in a project. Section 4.3 elaborates on the advantages of acceptance tests. One possibility for the implementation of acceptance tests is the simulation of user activities via the GUI to test whether the system is actually behaving as expected. This approach is the focus of section 4.4. The tests are based on the tool Selenium. Alternative tools are the topic of section 4.5.

Section 4.6 describes an approach where requirements can simply be written in natural language. Thus, they look like entirely normal requirements, but can also run as automated tests. For that the Java framework JBehave[1] is used. Alternative technologies are introduced in section 4.7. Section 4.8 explains with which strategies automated acceptance tests can be successfully implemented in projects. Finally, section 4.9 provides a conclusion.

4.1.1 Acceptance Tests: An Example

The special feature of the acceptance tests in this chapter is their automation. This reduces the time and effort for performing the tests. Big Money Online Commerce Inc. from section P.2 has also adapted this Continuous Delivery technique.

Without automated acceptance testing, errors that would have been noticed easily with proper testing can slip into production. As you might remember, that was what had previously happened in the company as reported in section P.2: A test was not rerun after a bug fix because the effort was deemed too great so that an error, in the end, went unnoticed.

In addition, automated tests are reproducible. In former times problems occurred because test results were not reproducible or because the people running the tests made mistakes. Both these sources of problems are addressed by automated acceptance testing.

And finally, after the initial investment in automated tests the expenditure for testing has gone down considerably. The investment has more than paid for itself. Now the tests allow implementation and rollout of new features with less effort.

4.2 The Test Pyramid

Chapter 3, "Build Automation and Continuous Integration," discussed unit tests. This chapter now focuses on automated acceptance tests. Chapter 6, "Exploratory Testing," will show how manual exploratory tests fit into the overall picture. One question is how much focus should be given to the different test types relative to each other. The test pyramid[2] (Figure 4.1) helps to address this question. The size of the different sections of the pyramid indicates how many tests should be there in these areas relative to the other areas.

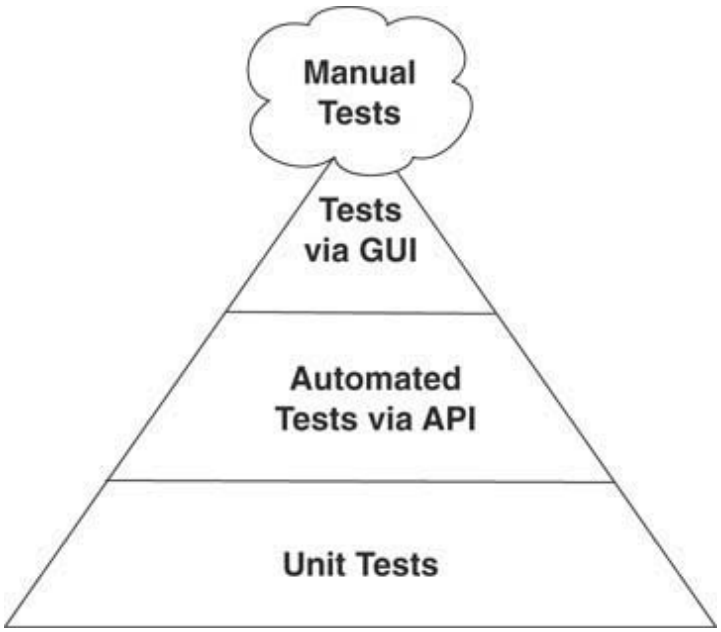

Figure 4.1 *Test pyramid*

At the base of the pyramid are unit tests—this type of test should be the most numerous. They are easy to implement, do not have dependencies outside of the tested class, and run comparably fast. Therefore, they should be the preferred test type.

Tests for an API—for instance, acceptance tests with BDD tools, as shown in this chapter—represent the next layer. The final layer is made up

by acceptance tests, which are based on GUI automation. Tests with an API are preferred; they are faster than GUI tests because they do not use the GUI layer and therefore can be more efficient. Besides, changes to the GUI do not break these tests. Tests via the GUI can fail because a GUI element was renamed or other small changes were introduced into the GUI.

The tests via the GUI or an API evaluate the required functionalities—in the end they define the requirements for the system. Therefore, they represent an important supplement for unit tests.

Nevertheless, when a problem occurs in a GUI test or in a test that utilizes an API, a unit test should be written as well that mimics the situation in which the error happened, because such errors should in principle already be detected by the unit tests.

In the test pyramid, manual tests are only used in specific scenarios. They are laborious and hard to reproduce. Especially when the pipeline has to be run frequently, the effort of manual tests is just not manageable anymore.

However, in reality, one hardly ever finds a proper test pyramid, but rather sees the opposite picture: There are numerous manual tests (Figure 4.2). Accordingly, the test pyramid turns into an ice cream cone. Automation is limited to automating the GUI interactions. There are quite a number of GUI tests because they are easy to implement based on the manual tests. If there are tests for the API, it is only a small number and they are only used for special cases. Finally, there are some unit tests. Usually projects implement at least comprehensive unit tests. However, quite a few of them are of low quality. The focus on manual tests creates high manual time investment and therefore expenditure. While the GUI tests are automated, they are not only slow but also fragile when it comes to changes. Merely changing the name of a GUI element might break a test. A test setup like this ice cream cone is the most frequent problem with regards to test strategies.

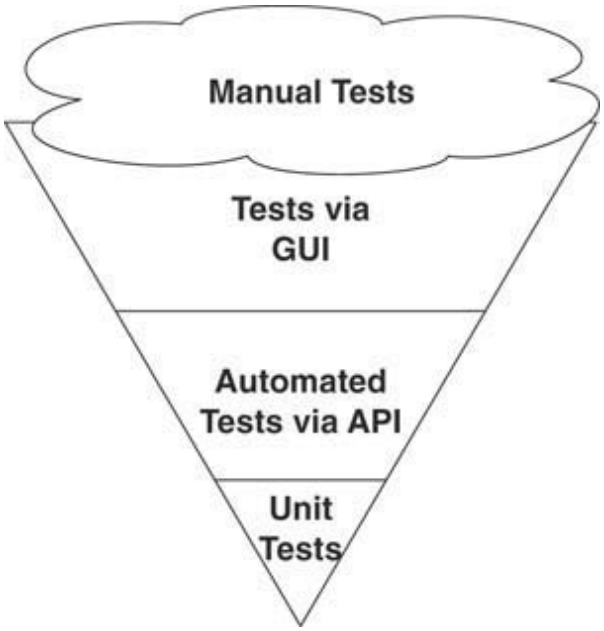

Figure 4.2 *In reality tests often look more like an ice cream cone*

Instead, developers should always aim for a test pyramid as depicted in Figure 4.1. This holds true independently of Continuous Delivery. It is a general best practice.

Continuous Delivery focuses on rapid feedback. Therefore, apart from devising a test strategy that is in line with the concept of the test pyramid, it matters most how fast errors are detected. Tests should therefore first address basic functionalities, such as simple business processes like creating a customer or generating a new order. The initial tests should cover many processes: Initially, the objective is to run through all typical business cases based on a simple case. Only afterwards does it make sense to test more in depth, that is, to devise tests that evaluate whether all corner cases and potential error situations are sensibly dealt with.

Therefore, the tests should be structured in such a way that first simple tests covering the basic business processes are performed and only afterwards are more complex cases tested. If the software contains a critical error, it will be found quickly—at least if it involves basic features. Such an error will already be identified when testing just broadly. In that case tests going into more depth are not needed anymore. If simple cases do not work, complex scenarios are even less likely to work. If every feature is tested in depth right from the start, it might happen that a critical error is only identified late during testing. When the error affects the feature that is tested

last, all other features will have been tested in depth before—which will have taken quite some time. This of course delays the feedback.

Therefore, it makes sense not to necessarily sort tests in capacity tests and acceptance tests but according to the likelihood with which they will find an error, and first test broadly rather than deeply. In addition, in the first test the so-called “happy path” should be tested—i.e. the successful execution of a business process with simple parameters. Corner cases and complex scenarios should be examined only later.

Try and Experiment

Analyze the testing strategy of a project you are familiar with:

- Which fraction of the current tests is explorative—that is, explores new features?
- Which fraction of the current tests is automated? Consider unit tests, capacity tests, tests with regards to other non-functional requirements (usability, security), and functional acceptance tests.
- Does the testing strategy resemble the test pyramid, or an ice cream cone?
- How is the project safeguarded against regressions—that is, errors that in principle have already been removed, but occur again due to changes in the code? Are the regression tests automated? Which benefits would be associated with automation?
- Are non-functional requirements such as usability or “look and feel” tested?
- Where are most errors found? Are they identified by one of the tests? By which one? Or are they only found in production? A good source for this information is the bug tracker.

Based on this information you can devise a plan for which additional tests might be good to have and which tests should be the first ones to be automated.

4.3 What Are Acceptance Tests?

The objective of acceptance tests can already be guessed from their name: The users respectively customers “accept” the software. This indicates also the special challenge with regards to this type of tests: Testers, requirements engineers, developers, and finally customers have to understand the acceptance tests and need to make sure that the tests are really covering the acceptance criteria. This requires communication between all parties involved—therefore acceptance tests strengthen the exchange within the project.

4.3.1 Automated Acceptance Tests

For Continuous Delivery the acceptance tests should be automated. When new releases are supposed to be delivered frequently and therefore have to be tested often, the testing costs are much too high without automation. Besides, the test results of manual tests are hard to reproduce since an error detected by such a test is not necessarily due to an error in the software, but can also be caused by a mistake during the execution of the test.

4.3.2 More Than Just an Increase in Efficiency

Automation not only makes the execution of the tests more efficient: Automation allows running the tests more frequently. Thus feedback is available faster. When the acceptance tests are performed upon each commit, the team gets to know within a few hours or even minutes whether there are errors in the software. Consequently, the search for errors can be limited to changes that have been introduced during the last few hours.

However, manual tests are often only performed shortly before the release —that is, at a profoundly later point in time. Therefore, it is much more difficult to figure out which changes have caused an error. In addition, the tests occur in a phase in which the team is already stressed due to the imminent release that will happen in the near future. In conclusion, test automation improves the process of error elimination and can also improve software quality.

The effort necessary for automating is often overestimated: Frequently, manual tests are structured in such a manner that they are nearly entirely formal—but only nearly entirely. If they were entirely formalized, it would be very easy to automate them. Concretely, an Excel file can contain the information about which actions have to be performed at the surface to obtain a certain result, and this can be done at the following level of detail:

- Please enter the following text into the text field: “Test customer.”
- Press the OK button.
- Examine whether the result is in line with expectations.

From an economic viewpoint this type of test is the worst possible. The test is hard to generate because it is very detailed. It is not simply the expected behavior that is described, but the specific use of GUI elements. The testing is very laborious since it is still a human who has to perform the interactions and check the results. A tester who understands the application from the domain perspective is thereby degraded to a test automaton. The description hardly allows him to perceive the domain aspect that is being tested. Indeed, concrete values are tested—but what is the significance of these values from a domain perspective? Which special cases are being tested right now? Since the deeper meaning of the test is not clear, the results of failed tests remain unspecific—something just “went wrong,” but it is not easy to deduce which domain-based problem is the underlying cause for the failed test.

Section 4.4 shows that it takes little additional effort to automate such tests. Subsequently, the necessary effort for executing these tests decreases dramatically. At the same time testers can concentrate on tasks for which their domain knowledge is useful. Therefore, automated acceptance tests make a lot of sense in such situations from an economic perspective and they pay for themselves rapidly.

4.3.3 Manual Tests

Continuous Delivery does not demand the complete automation of all acceptance tests. This is a difference from Extreme Programming where each requirement is safeguarded by an automated test. Continuous Delivery includes a phase with manual explorative tests (see Chapter 6). The automated acceptance tests are meant to help avoid regressions—errors creeping in due to changes in software modules that have already been tested. The explorative tests are first of all supposed to find errors in new modules and test for instance usability, which is hard to automate. In addition, the team is unencumbered by the automated acceptance tests. Thus, it can better focus on the explorative tests instead of having to test the same components over and over again.

4.3.4 What about the Customer?

In principle, acceptance tests ensure the acceptance of the software by the customer. When the customer has outsourced the development to an IT service provider, the acceptance tests are crucial, since only after the tests and the successful acceptance by the customer is the IT service provider paid for its work. The team does not always manage to convince the customer that the automated acceptance tests in fact reflect the requirements of the customer. This underlines the fact that acceptance tests are highly relevant for communication. When the customers themselves can understand and follow the automated acceptance tests, they will be more likely to accept them as acceptance criterion.

Nothing argues against combining the formal acceptance with a manual acceptance test. However, this test is only performed once, namely, during the acceptance of a release by the customer. The automation of acceptance tests is still necessary to ensure the correct functioning of the system during the running of the Continuous Delivery pipeline. The manual acceptance tests of the customer can also build the basis for automated acceptance tests. Errors that are only noticed during the acceptance test by the customer should lead to a new automated acceptance test to ensure that the error is in fact removed and also does not occur again as regression.

4.3.5 Acceptance versus Unit Tests

Finally, it is important to distinguish unit tests from acceptance tests. The main criterion for distinguishing between the two is not the technologies being used. Acceptance tests could also be implemented with unit test frameworks like JUnit.

First of all, an important criterion to differentiate between the two test types is the different stakeholders. Unit tests are a tool for developers. They are written by developers to ensure correct implementation. A user might not even know that there are unit tests. Acceptance tests on the other hand are used by developers and users to ensure the correct implementation of requirements with regards to domain logic. Users have to understand the acceptance tests to be able to ensure that the tests really evaluate the acceptance criteria of the software.

In addition, the layers the tests address are different. As the name already states, unit tests evaluate individual units of an application in isolation; for example, individual classes. Consequently, they do not cover the interplay of

the units. Acceptance tests evaluate larger parts of the system or even the entire system. However, this is only a result of the focus of the tests: The objective of acceptance tests is to safeguard domain logic—after all, this is only possible when more than just individual classes are tested. In this regard acceptance tests also differ from integration tests. The integration tests focus only on the correct interplay of the components and do not necessarily test the acceptance criteria of the software.

At the technical level unit tests are mostly white box tests. “White” means, in this context, transparent: These tests refer to the internals of the implementation and the classes. Therefore, unit tests are rather fragile: When the same functionality is implemented with other classes or when the structure of the classes has been changed by a refactoring, the unit tests also have to be adapted.

Acceptance tests are black box tests: They test the correctness of the implementation with the help of an API or user interface and do not depend on internal matters. Therefore, they are also less fragile: They should only report an error when the functionality is not correctly implemented. It should not matter in which classes the functionality is implemented and what these classes look like.

In the end, acceptance tests are primarily valuable because they test what the actual user is interested in and what therefore is really important—namely, the correct implementation of the features.

4.3.6 Test Environments

Continuous Delivery makes it easier to generate test environments due to infrastructure automation. However, one problem remains: Third-party systems which are not part of the infrastructure have to be present for test environments. There are multiple solutions for this problem:

- Often third-party systems can use, in addition to the production systems, test systems that behave like production systems.
- The third-party systems can be replaced by stubs that simulate the behavior of the production systems.

In both cases there is a difference between the test environment and the production environment. This limits the reliability of the tests—errors might not be detected. However, this problem cannot be perfectly solved; in the

end, production environments and test environment are never entirely identical. But Continuous Delivery is still helpful in such scenarios for yet another reason: Since the deployment in production is automated and additional approaches for risk minimization are used (see Chapter 7, "Deploy—The Rollout in Production"), the risk arising from the limited reliability of the tests can be further decreased. If an error still remains undetected during testing, a fix can be relatively quickly brought into production.

4.4 GUI-Based Acceptance Tests

Via the GUI, users will later interact with the application. Therefore when users test an application, they will use the application via the GUI and observe whether it behaves as desired. Instead of performing each test manually every time, it is obviously beneficial to automate the tests—this leads us exactly to the automated GUI-based tests which are the topic of this section. In the end the interactions with the GUI are recorded in these tests, and it is examined whether the right results are displayed in the GUI.

4.4.1 Problems of GUI Tests

However, this approach is not perfect: Potentially, semantics are lost during the tests. In the end the functions of the GUI are just one way to use the features of the application—and the tests should ensure the correct functioning of the features. This difference becomes apparent upon changes to the GUI: The GUI tests might not work anymore because the GUI has changed its appearance. For example, it is possible that values are not entered anymore as text, but are selected from a list. In that case the test will fail because it is not anticipating the selection of the values from a list. However, the features of the application have in essence remained the same —therefore the test should in principle still be passed successfully. Accordingly, GUI tests are fragile. Changes to the GUI can break the tests even if there are no real domain-based errors in the application. This is especially true when the tester records the interactions with the GUI and later replays them as an automated test. The recording is based on concrete GUI elements. When those change, it is quite possible that the test will fail.

4.4.2 Abstractions against Fragile GUI Tests

To circumvent this problem, abstractions can be built into the GUI tests that abstract from the concrete GUI. If a button is renamed or moved, only the abstraction layer has to be adapted—the test remains unchanged. In this way, the approach gets closer to the procedure approach in section 4.6 where the tests are described entirely textually. However, when the approach becomes too complex, it is possible that users and requirement engineers will not understand the tests anymore. This contradicts the goal of ensuring the correct functioning of the application through collaborative acceptance testing by developers, users, and requirement engineers . Simple GUI tests are normally easy to understand since they just automate the interaction with the GUI. Additional abstractions put a veil over them and therefore make the tests harder to figure out.

In addition, the generation of test environments is more complicated since the entire application including the GUI has to run and a production-like database has to be installed. Moreover, the infrastructure for automating the tests has to be generated. This usually comprises clients with appropriate web browsers and automation tools.

In spite of these limitations GUI tests can be sensible tools because how the tests work is immediately understandable. Laborious, manual work is automated—and such an automation is essential for Continuous Delivery. At any time during the tests users can be shown exactly what is happening in the familiar GUI.

4.4.3 Automation with Selenium

Concretely, in this chapter we will use Selenium[3] as a tool. Selenium offers the option of executing tests via a remote-controlled web browser. Selenium simulates interactions of a user with the system and controls whether the desired results are displayed.

4.4.4 Web Driver API

The tests can be implemented as program code—the web driver API serves this purpose. The API remote controls a web browser. There is support for practically all popular browsers such as Chrome, Safari, Internet Explorer, and Firefox. To avoid providing each browser on each single computer, tests can also run in a cluster where on each computer certain browsers are installed. The Selenium server can coordinate the cluster. In this manner the tests can be executed with different browsers, and the differences between

the behavior with the different browsers can be identified. In addition, this allows parallelizing the tests so that the test results are available more quickly. With the help of Selenium Grid the test can be distributed onto multiple computers in the network. In that case it is not necessary that all browsers be available on one computer, and for laborious tests sufficient hardware for rapid test execution can be provided.

4.4.5 Tests without Web Browser: HtmlUnit

Selenium does not necessarily need to use a web browser. An alternative is the use of HtmlUnit. With this Java library the necessary interactions with the web application are implemented entirely in Java so that no browser has to be installed. In that case tests can be executed on any computer without having to install additional software. Therefore, the example employs exactly this approach.

4.4.6 Selenium Web Driver API

The interaction with the application via a browser or HtmlUnit takes place with the help of the Web Driver API. The Web Driver API is available for programming languages like Java, Ruby, C#, Python, and JavaScript—many other programming languages are supported by community projects. With the API a programmer can implement the tests. However, acceptance tests are also supposed to be understood and maybe even implemented by users and testers.

4.4.7 Selenium IDE

An alternative is the use of the Selenium IDE. It provides an interface within a web browser for recording the interactions with the web application. Selenium IDE is implemented as an extension for the web browser Firefox. Test expressions can be manually added to the recorded interactions to examine whether the application behaves as expected.

Ideally, acceptance tests are supposed to represent the contract between the customer and the team. Via the acceptance tests the customer expresses which functionality is supposed to be implemented. Therefore, it is necessary that both sides be able to understand and evaluate the acceptance tests. This renders Selenium IDE especially attractive: Via the tool the customers are able to use the application and to define and record the desired behavior. The Selenium IDE allows them to do so largely on their own.

Figure 4.3 shows a test for an example application in the Selenium IDE. To be precise it is a test suite, which consists of individual test cases. The first test case is called UserDoesNotExist and ensures that the test user does not exist at the start of the test. To ensure this the search function is used for searching for the user. The expected result is that the user is not found. In that case a user is registered. The next test case, RegisterUserAgain, registers the user again. This test case is currently selected so that the details of this test case are visible on the right site of the IDE. As the first step the homepage is opened under the URL "/en." Afterwards the link "Register User" is clicked. Finally, the data of the customer are entered, and the form is submitted. The result should be that somewhere on the page the response "already in use" appears— and this is exactly what happens. This error message indicates that the user registration was not successful because there is already a data set with this email address.

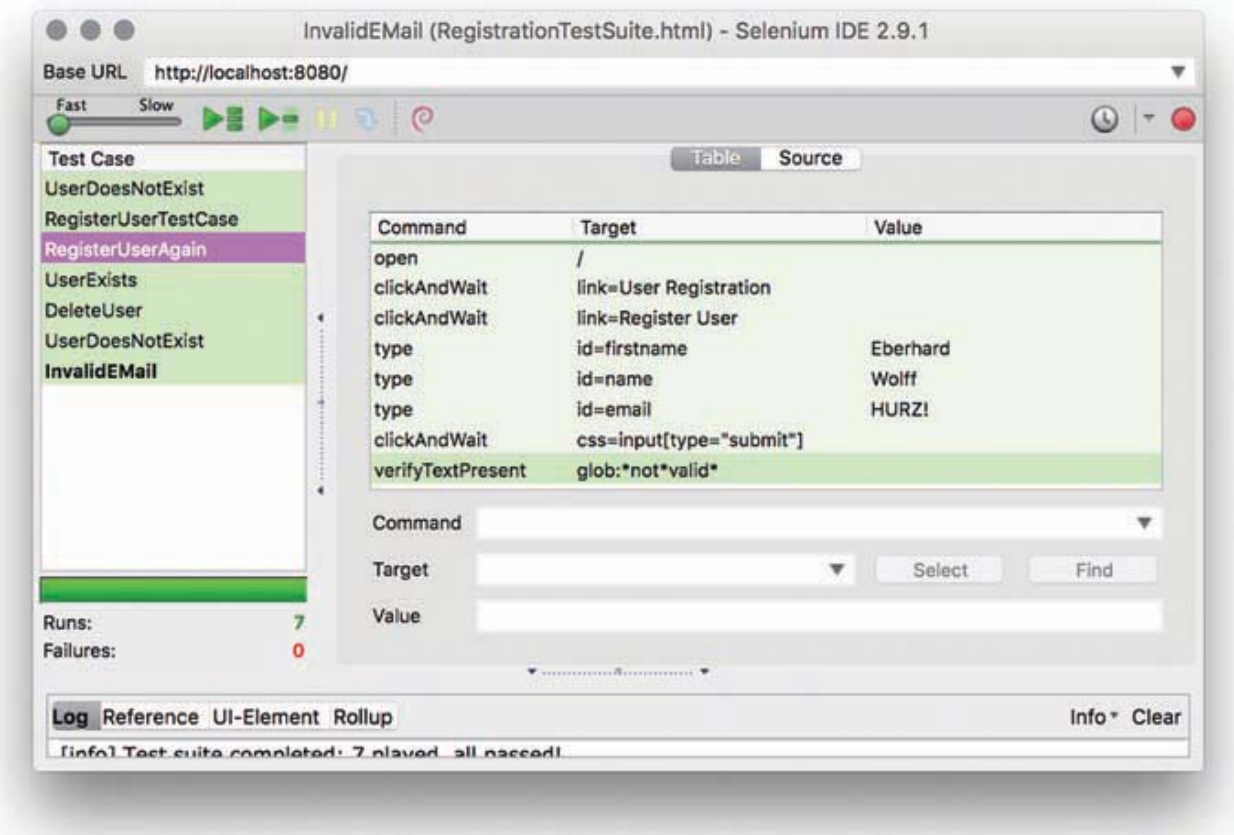

Figure 4.3 *Screenshot of the Selenium IDE.*

Additional test cases follow:

- UserExists tests whether the user now really exists.
- DeleteUser deletes the user.
- UserDoesNotExist tests whether the user does not exist anymore.
- InvalidEMail examines whether an invalid email address causes the rejection of a registration.

4.4.8 Problems with Automated GUI Tests

However, this test does have one problem: Many of the steps build on each other. If the user is not successfully registered in step one, neither searching for the user nor deleting her will work. Although it is only one functionality that is not working anymore, a large part of the test suite will fail. Such unspecific errors make it hard to spot the cause of a problem.

Upon a new run of a test there can also be problems: If certain datasets have been set up during the first execution of the test, they cannot be set up again during a second test run. Therefore, tests have to clean up at the end. However, when the test does not complete due to an error, the clean-up is not executed and the system remains in a state in which the datasets are not cleaned up.

4.4.9 Executing GUI Tests

The recorded tests can be executed in the Selenium IDE. In that case the test case is colored green upon success, and red upon failure. The test cases and test suites are stored as HTML files. The HTML files contain the Selenium commands. It is possible to run these tests in an automated manner as part of a build. To do so it is just necessary to install the Selenium server. In that case the tests can be started with a command line like the following:

```
java -jar selenium-server-standalone-2.48.2.jar -html4Suite *firefox
http://localhost:8080/ RegistrationTestSuite.html result.html.
```

The option `htmlSuite` sets the server into the right mode. The identification string of the browser, the basis URL, the test data, and finally the file in which the result is supposed to be stored follow. This allows the execution of a test recorded with Selenium IDE in the context of a build.

4.4.10 Exporting the Tests as Code

In addition, the tests can be exported as program code. This has the benefit that the tests can be optimized, for instance by a one-time implementation and reuse of functionalities that are used in numerous tests. The tests can also be modified in such a manner that they are safeguarded against changes to the GUI. A test framework is required for the execution of the tests. Ruby with RSpec or Test::Unit can be used as a test framework, or Python with unittest, or finally C# with NUnit. For Java JUnit 4, JUnit 3 or TestNG can be used. Although these frameworks were originally meant for unit tests, they are used in this case for acceptance tests. The tests employ the Selenium Web Driver to run the interactions of the test cases with the website.

4.4.11 Manual Modifications of the Test Cases

Here also, input from the developers is required: In some cases, the generated test cases do not function properly and have to be modified. This concerns for instance checking whether the right results are displayed. In addition, the developers can in some places unify and simplify the code. For instance, in the example a customer is searched for or set up multiple times. Ideally, such functionalities are implemented only once and then called in the right places.

Modifications can also render the tests more stable. Tests have to check the right results. To do so they can examine whether certain elements are present—for instance, a certain error text. However, when the error text changes, the test breaks, even though the functionality is still implemented correctly. However, this can be avoided: For instance, a convention can be introduced that a certain CSS class is only used when an error is supposed to be returned. In the script in Figure 4.3 such an approach is employed by searching for a `div` element with the CSS class `alert` and `alert-error`. In addition, whether the expected error message is returned is checked.

4.4.12 Test Data

In addition, tests have to generate test data. In the example, the script itself generates the test data. However, this is not always possible, or it might not be feasible because it is too complicated or takes too long. In this case a developer can help as well by implementing appropriate functionalities.

Thus, in the end abstractions and facilitations for implementing tests will be realized. As in the case of the implementation of the functionalities, GUI-based acceptance tests also require the collaboration of users and developers.

The modifications and the implementation in code also mean that the customers can no longer easily understand what exactly is being tested. However, this is precisely the objective of acceptance tests. Besides, the approach for the code generation creates a number of problems: Since the generated code is reworked, changing the test results in difficulties. The code cannot just be generated one more time; instead the test has to be modified in the Selenium IDE and also in the code.

Nevertheless, Selenium IDE is an important tool: It is possible to run HTML scripts. This can be very useful for explorative testing (see Chapter 6) and the recording of scripts. Thereby, testers can more easily regenerate test situations and automate procedures at least partially. Besides, the implementation of tests that are entirely automated is ideally prepared by this approach.

Another advantage of this test approach is that the interface is tested, and thus layout problems or browser incompatibilities can also be evaluated.

4.4.13 Page Object

It might happen that the GUI tests stop working if the GUI is modified—for example, because the naming of web GUI elements has been changed. This can be counteracted—as so often during software development—by an abstraction layer. In the concrete case the elements on the web page can be hidden behind Page Objects.[4] They offer an abstraction that orients along the importance of the web GUI from the point of view of a business approach. For example, methods such as `registerCustomerWithEmailFirstnameName()` can be offered that internally provide the respective web GUI elements with data and trigger a request. When the GUI changes, only this method has to be adjusted instead of every code line, and it is not necessary to individually change the elements.

4.5 Alternative Tools for GUI Tests

Selenium is widely used for implementing GUI tests and has been around for a long time. It offers numerous features and can meet complex demands. Still, there are some alternatives to Selenium.

4.5.1 PhantomJS

Especially in the JavaScript community, PhantomJS[5] is often used. In this tool the tests are written with JavaScript. Apart from that PhantomJS is very similar to Selenium. It can also use WebDriver for remote controlling the browser—like Selenium.

4.5.2 Windmill

At first glance Windmill[6] is a very interesting solution: Tests can be recorded with the browser. Python or JavaScript can also be used to define test scripts. However, the Windmill project is largely inactive at the moment.

Try and Experiment

To become familiar with web-based acceptance tests, the following steps are worthwhile:

1. First set up the Selenium IDE. To do so have a look at http://docs.seleniumhq.org/download/.
2. The example project can be found at http://github.com/ewolff/user-registration-V2. To execute:
3. Install Maven (see http://maven.apache.org/download.cgi).
4. Execute `mvn install` in the directory `user-registration-V2`.
5. Execute `mvn spring-boot:run` in the sub-directory `user-registration-application`.
6. Now you can try out the application at http://localhost:8080.
7. What happens if you register two times with the same email address?
8. What happens if you use an invalid email address?
9. At http://docs.seleniumhq.org/docs/02_selenium_ide.jsp#building-test-cases you can find information about how to record and run tests with the Selenium IDE.
10. In the sub-directory `user-registration/user-registration-acceptancetest-selenium/Selenium/en` the test suite `RegistrationTestSuite.html`can be found. It comprises essential components of the registration. Load it into the Selenium IDE and let it run.
11. Download the Selenium Stand Alone Server; start the test suite using it. The script `runTest.sh` can be useful for this.
12. In the Selenium IDE you can have code generated from the test cases. Do so and have a look at the code.
13. Optional: With the help of this code implement a test suite for the application in your preferred programming language.
14. In the project `user-acceptancetests-selenium` you can find Java tests that have been created based on the test that was generated by the Selenium IDE.
15. Have a look at the tests.
16. With the command `mvn test` you can execute the tests.

17. Compare them with the tests generated by IDE.
18. Where has the code been unified?
19. How does the code for checking the results differ between the generated code and the reworked code?
20. Rewrite the test in such a way that it uses Page Objects.

4.6 Textual Acceptance Tests

Normally, requirements are recorded textually—therefore acceptance tests should be depicted like a requirement.

4.6.1 Behavior-Driven Development

In this context Behavior-Driven Development (BDD) has established itself. In this approach the written representation of the acceptance tests is formalized to such an extent that it can be executed automatically. BDD combines techniques of test-driven development with the idea of *Ubiquitous Language* from Domain-Driven Design (DDD).[7] Test-driven development is a technique where tests are written prior to the actual implementation. The test will of course fail without an appropriate implementation. When the test finally successfully runs through, the features have in fact been implemented.

Ubiquitous Language in DDD represents a common language for users, requirements engineers, and developers that can be found back in the code. It comprises the central technical terminology of the domain. With BBD, an acceptance test is a common language for everybody in the team—users, testers, and developers.

Listing 4.1 *A BDD Narrative*

```
Narrative:
In order to use the website
As a customer
I want to register
So that I can login
```

For example, there are stories. They define features and put them into the context of the domain. For the user registration Listing 4.1 shows the narrative. It is defined that the using role is the customer and what is

supposed to be achieved is the feature. The discrimination of the narrative in "in order to" (aim), "as a" (role), "I want to" (feature) and "so that" (benefit) is determined. However, this description allows "only" the definition of features and therefore can strengthen the understanding of the application between developers and customers. It cannot be executed as an automated acceptance test. However, the description can become a part of the JBehave test.

Listing 4.2 *BDD Acceptance Test*

Click here to view code image

```
Scenario: User registers successfully

Given a new user with email eberhard.wolff@gmail.com firstname
Eberhard name Wolff
When the user registers
Then a user with email eberhard.wolff@gmail.com should exist
And no error should be reported
```

Let us have a look at Listing 4.2. There a concrete scenario is described. The scenario belongs to a story and defines the possible sequence of events in the context of the story. There are three components:

- "Given" describes a context in which the scenario takes place.
- "When" is an event that occurs.
- "Then" defines what the expected result is.

If one of these types of components appears more than once, the components can be connected with "and." Concretely, two expected results are defined in Listing 4.2: The customer is supposed to exist, and no error should be reported.

The scenario is composed in natural language; nevertheless it conforms to a formal structure so that the test is automated. In this manner it is possible that a domain expert describes an expected behavior which can be evaluated by an automated test.

By the way, it is of course possible to write this test prior to the actual implementation. The tool will just indicate that the code that is necessary for the test has not yet been implemented. Therefore, the test-driven approach can also be used for acceptance tests.

Listing 4.3 *Code for the BDD Acceptance Test*

Click here to view code image

```
public class UserRegistrationSteps {
  private RegistrationService registrationService;
  private User customer;
  private boolean error = false;

  // Initialization of the RegistrationService omitted


@Given("a new user with email $email firstname $firstname"+ "
name $name")
  public void givenUser(String email, String firstname, String
name) {
    user = new User(firstname, name, email);
  }

  @When("the user registers")
  public void registerUser() {
    try {
      registrationService.register(user);
    } catch (IllegalArgumentException ex) {
      error = true;
    }
  }

  @Then("a user with email $email should exist")
  public void exists(String email) {
    assertNotNull(registrationService.getByEMail(email));
  }

  @Then("no error should be reported")
  public void noError() {
    assertFalse(error);
  }

}
```

Listing 4.3 shows the necessary code for running the test. It uses the framework JBehave.[8] Within the annotations `@Given`, `@When`, and `@Then` the texts indicate the respective code that is supposed to be executed. Initially a user is created as an object with the method `givenUser()` during the test from listing. Then the user is registered in `registerUser()`. Finally, `exists()` and `noError()` are called to ensure that the expected result was achieved.

4.6.2 Different Adaptors

For implementing the tests an adapter is used in the example, which calls the application logic via the textual acceptance tests (see Figure 4.4). This approach has the advantage that it is not necessary to start the entire application, but just a simple application that only supports the tests. For example, no GUI is needed. Consequently, the tests are quite rapid. In addition, the tests are relatively easy to start on a developer computer and often even within the IDE. When implementing the adapter, developers can implement exactly the functionalities necessary for the tests. Therefore, the test implementation is simpler (Figure 4.4).

Figure 4.4 *Textual acceptance tests with adapter for business logic*

An alternative is shown in Figure 4.5: In this case the acceptance tests work with the help of an adapter via the web surface. The tests can still be composed in natural language—however web GUI-based. An example is shown in Listing 4.4. As you can see, in this case the testing happens at the level of the web GUI so that errors in the web GUI are found. However, the tests are on a higher abstraction level than the Selenium tests (Figure 4.5).

Listing 4.4 *Example for a textual acceptance test through the Web GUI*

```
Given user is on the homepage
When the user enters eberhard.wolff@gmail.com as email
And submits the search form
4 user should be found
```

Figure 4.5 *Textual acceptance tests with adapter for the web GUI*

The main difference is that the test does not name the concrete GUI elements and also does not define what the website looks like when no user has been found. This logic is implemented in the adapter layer. Therefore the implementation of the tests is easier, and the tests are more stable compared to pure Selenium tests. Changes to the GUI can be compensated for by changes in the adapter layer so that the tests do not break.

In addition, the tests evaluate the web GUI while the other textual acceptance tests operate directly on the logic. However, this also means that an appropriate running environment with a web server has to be available.

4.7 Alternative Frameworks

In addition to JBehave there are a number of other frameworks that likewise can be used for textual tests:

- Cucumber[9] supports different programming languages. The tests can be written in natural language, as in the case of JBehave.
- For Ruby, RSpec[10] offers support for a similar approach. However, here the tests are written in normal Ruby and not in natural language.
- Jasmine[11] is a JavaScript framework that allows one to write requirements in JavaScript code. The code is very easy to read, but it is not as easy to understand for non-developers as the frameworks that use natural language.

With JGiven[12] tests can be implemented as Java code, but the output of the tests occurs as plain text. Thus programmers can write the tests relatively easily, and the domain experts can still understand the functionalities. However, here the domain experts don't realistically have the option to write the tests by themselves.

Tools like RSpec or Jasmine claim to support BDD, but they do not support natural language. Instead they focus on tests that are implemented in a programming language. These approaches have the disadvantage that the customers cannot make much of the program code. Thus the original goal of acceptance tests is not achieved—namely to communicate about the requirements and to work together on automation.

Try and Experiment

The example project can be found at http://github.com/ewolff/user-registration-V2. To execute it:

1. Install Maven (see http://maven.apache.org/download.cgi).
2. Afterwards execute `mvn install` in the directory `user-registration-V2`.

Now you can run the textual acceptance tests:

- In the sub-directory `user-registration-acceptancetest-jbehave` you can execute the tests with `mvn integration-test`.
- Try to break the test by introducing changes into it. You can find the code for the tests at `src/main/resources/com/ewolff/user_registration/user_registration_scenarios.story`.
- Have a look at http://jbehave.org/reference/web/stable/using-selenium.html and try to implement the test from Listing 4.4 with the web driver. The existing test can serve as an example.
- Have a look at Cucumber.[13] Introductory articles can be found at https://github.com/cucumber/cucumber/wiki/tutorials-and-related-blog-posts. Since Cucumber supports many programming languages, Java applications as well as applications written in different programming languages can be tested.

4.8 Strategies for Acceptance Tests

To benefit from acceptance tests it is important to select the right tool. The approaches introduced in this chapter are based either on GUI tests or on textual descriptions of acceptance criteria. It is not so much the technical quality of the approaches that is important here: Acceptance tests serve first of all for communication between users, requirement engineers, and the developer team. When the users or requirement engineers cannot handle the tool, it will not be used and the approach will not be successful in the end. Therefore, the main goal when selecting a tool for the acceptance tests has to be that the selected technology is easy to use and to understand, especially for users and testers. In the end the customers have to accept the software based on the acceptance tests.

4.8.1 The Right Tool

Some customers and requirements engineers are used to write test protocols with Excel. They contain a description of the individual steps of the tests. Likewise there are projects where Excel is used to determine the desired results for certain input values. In such a scenario it can make sense to use the Excel files as input for acceptance tests. This allows one to keep using the same tools. The data in the Excel spreadsheets just have to be analyzable in an automated manner and therefore have to comply with some formal rules. However, this is a much lower barrier than having to additionally accustom the users to a new tool.

The technical implementation of this approach is also not very laborious: There is for example the Framework for Integrated Tests (Fit),[14] which follows exactly this approach. It is available for different platforms. Fit accepts HTML files for defining the tests. And most Office tools like Excel or Word allow one to export documents as HTML. These can be used as acceptance tests for these tools.

An alternative is concordion.[15] This tool likewise uses HTML—however, the HTML code has to be reworked. Exports out of Excel or Word are not directly usable.

Thus, these tools can facilitate communication with the customers in regards to acceptance tests. Still, it is not always possible that customer and team really work together on the acceptance tests. However, even in such cases automated acceptance tests make sense because they ensure the correct

functioning of the domain logic. The joint work of developers and customers on the acceptance tests makes it easier to generate these tests and offers a possibility for aligning functionalities. When this is not possible, the work will be more difficult, though not impossible. Developers can record the demands of the customer in the tests. Finally, the developers also implement the production code based on customer demands.

4.8.2 Rapid Feedback

An important objective of Continuous Delivery is rapid feedback. When a developer checks in erroneous code, the Continuous Delivery pipeline should indicate the error as fast as possible. Therefore, with regards to acceptance tests, it is best to initially test the simple cases of all features and only afterwards to test the features in all their depth. When a feature basically does not work, this problem is detected earlier. In that case the more detailed tests of the feature would fail anyway so that one can do without running them. This approach can for instance be implemented across different steps in the Continuous Delivery pipeline.

4.8.3 Test Coverage

The goal of acceptance tests is not really a complete or high test coverage. Ideal acceptance tests are structured in such a manner that the successful execution of the tests is equivalent to an acceptance of the software by the customer. Therefore, the focus should not be on corner cases, but on functionalities that are especially important for the customer.

It is also important to view the acceptance tests together with the explorative tests (Chapter 6). The automated acceptance tests are meant to free the testers from routine tasks so that they can concentrate on the explorative tests. Therefore, a complete automation of all tests is not obligatory—a manual test phase is intended anyway.

Try and Experiment

Have a look at your current project.

- Which tests are there?
- How are they currently represented? With the help of an Excel-based test manual? How far away are they from a complete automation?
- Which tool (Selenium, JBehave, concordion, Fit) is therefore best suited? For which part of the tests?

4.9 Conclusion

A fundamental test approach for acceptance tests are the GUI-based tests. At first glance, these tests appear to be a very good strategy: In the end, manual tests also use the GUI—therefore what could be better than automating these interactions? With Selenium IDE the procedures can be written down and then automated. However, in the end it is not that easy: GUI tests are fragile—they can break even upon small changes to the GUI. When the GUI tests are transformed into code, they can be optimized—however, in that case the recorded interactions and the tests are quite different so users might not understand whether the tests actually cover the relevant business logic. Besides, it is not possible to define acceptance criteria before the code is implemented. Finally, the automation of manual tests can ensure that the tests will work. However, an error can result in a multitude of failed tests. An error during the setup of the test data can for instance cause numerous ensuing errors because the data is not available. Besides, setting up large amounts of test data is still difficult with this approach.

Textual acceptance tests do not possess these weaknesses since they are based on formulating the test scenarios in natural language. This does not require functioning code. The test can be written without any code having to be implemented. Each of the tests usually covers exactly one scenario—if an error in the domain logic has crept in, only one scenario should fail.

A good compromise is the use of textual GUI acceptance tests. In the end acceptance tests serve to facilitate communication between customers and developers. When the customers only trust a GUI test that they can understand and execute on their own, if necessary, this is an important argument. Maybe simple Excel tables are preferred as specification? In that

case concordion or Fit might be an alternative, because in the end the customer is nearly always right....

However, even when the communication with the customer cannot be entirely implemented via acceptance tests, automated acceptance tests make a lot of sense to safeguard the correct implementation of the domain logic in the software.

Endnotes

1. http://jbehave.org/
2. https://martinfowler.com/bliki/TestPyramid.html
3. http://docs.seleniumhq.org/
4. https://martinfowler.com/bliki/PageObject.html
5. http://phantomjs.org/
6. https://github.com/windmill
7. Eric Evans: *Domain-Driven Design: Tackling Complexity in the Heart of Software*, Addison-Wesley, 2003, ISBN 978-0-32112-521-7
8. http://jbehave.org/
9. https://cucumber.io/
10. http://rspec.info/
11. https://jasmine.github.io/
12. http://jgiven.org/
13. https://cucumber.io/
14. http://fit.c2.com/
15. http://concordion.org/

Chapter 5. Capacity Tests

5.1 Introduction

Capacity tests ensure that an application provides the necessary performance for a certain number of users. Section 5.2 discusses the motivation for capacity tests and the fundamental challenges associated with the implementation of capacity tests. The concrete implementation of capacity tests is the focus of section 5.3. Gatling is a tool that is especially well suited for capacity tests—section 5.4 demonstrates how this tool can be used to implement capacity tests for the example application. Since no tool is the optimal solution for all problems, section 5.5 introduces some alternatives.

5.1.1 Capacity Tests: an Example

A critical feature of the capacity tests described in this chapter is their automation. Before introducing automated capacity tests, Big Money Online Commerce Inc. from section P.2 had the problem that for some releases the performance in production suddenly decreased. Thanks to the introduction of automated capacity tests each change is now tested first. Performance problems can be traced to certain code changes, and it is also known which use cases are problematic. Previously, hints regarding possible problems could only be obtained after manual tests—and often these tests were irregular and not part of the standard test suite. Therefore, it was hardly possible to derive conclusive information about the performance.

5.2 Capacity Tests—How?

At the outset, it is important to clarify some terms:

- Performance
  A measure of the speed with which a system can process a certain request. Bad performance means a user has to wait for a long time for a reaction from the system.
- Throughput
  How many requests a system can process in a certain time. Bad throughput will result in very few users being able to use the system in

parallel.

A fundamental problem in this area is nonlinear effects. When for instance 200 users create 25% load on the server, 400 users should result in 50% load if the system behaves linearly. However, since the systems are not linear, in reality the results might look completely different. This profoundly limits the informational value of capacity tests: The tests take place in an environment that differs from the production environment, and they are performed with data and data volumes that do not correspond to the data volume during production. Therefore, only a part of the real load is considered.

When non-linear effects occur, problems might arise during production in spite of successful capacity tests. Concretely, it is for instance possible that no index is used in the database. This does not cause problems in small test data sets, but might provide very low performance with the data volumes present during production.

5.2.1 Objectives of Capacity Tests

Within the bounds of possibility capacity tests serve to safeguard performance and throughput of applications. Performance and throughput of an application are important non-functional requirements: When an application is not sufficiently fast, this is only acceptable to a certain degree. From a certain point onward, the application will not be accepted anymore.

5.2.2 Data Volumes and Environments

It would be optimal if capacity tests could be performed with the full data volume from production and on an environment like the one used in production. However, this is often not possible because the required hardware is not available.

Besides, the data have to correspond to the real production data—otherwise the application will behave completely unrealistically. Therefore, it can be helpful to implement a test data generator. This allows one to generate different, large test data sets relatively simply.

5.2.3 Performance Tests Only at the End of the Implementation?

Reaching the necessary performance is a fundamental problem: In the end the application can only be properly evaluated for performance if it is complete with regards to domain functionality, and correctly implemented.

Only then does the application do what is required of it. As long as the features have not yet been completely implemented, the gaps can cause unrealistic results in performance tests. However, when the performance tests finally commence and the application in fact does not reach the necessary performance, it is often too late for fundamental changes—and besides, during optimizations domain-based errors might be introduced into the application. To find such errors, acceptance tests would have to be run again. Afterwards, capacity tests would have to be repeated to evaluate whether the performance is now improved.

5.2.4 Capacity Tests = Risk Management

In essence, capacity tests are a form of risk management. Failure of the software to reach the necessary performance for the client must be avoided. A fundamental basis for this is model building: The real behavior of the application in production and the behavior of the users have to be modeled. Often this is not so easy since each user might use the application differently. When a comparable application is already in production, metrics from production can be integrated into the capacity tests. However, if an entirely new application is being implemented, capacity tests can only be based on predictions and educated guesses.

5.2.5 Simulating Users

With regards to user behavior, not only does the number of users matter, but also how often users might use the application and which specific functionality they use. Depending on the application small differences can have profound effects on performance and therefore on capacity tests.

Often it makes sense to record the performance in scenarios similar to use cases: For an e-commerce website different scenarios could be, for example, searching the product catalog, selecting different products, and checking out. This renders the performance requirements more concrete:

- How long may the search last?
- How long may it take for the purchase to be finalized? Maybe in this scenario longer waiting times are acceptable than for the initial search.
- How many users are going to use the e-commerce website in parallel?

5.2.6 Documenting Performance Requirements

A prerequisite for performance tests is that there be concrete and measurable requirements for the performance of the individual scenarios. Statements like “fast,” “about 5 ms” or “without perceivable delay” have to be replaced by concrete requirements like “less than 5 ms.” Based on such values the simulation of the scenario can be implemented with the help of an appropriate tool.

Of course, it is important that the requirement be realistic. If the requirements are too aggressive, the software is optimized too much. This will increase implementation costs and might also necessitate the purchase of expensive hardware. If the requirements are not aggressive enough, the software will not be usable in the end. Still, there should be a certain flexibility. For extreme cases a longer processing time might be necessary and also acceptable.

5.2.7 Hardware for Capacity Tests

Model building is also key for the hardware requirement of capacity tests: When a production environment is not available, an environment should be used that at least allows inferences about the expected performance on the production environment. For example, if the application will later be running in a cluster, it might be possible that a cluster node is sufficient for the capacity tests. In addition, the number of simulated users has to be selected to match the number of users per node in production. At the same time, it has to be ensured by tests that the application can also be scaled horizontally —that is, that employing more nodes will in fact lead to a homogeneous load distribution. Still, non-linear effects can render the entire approach futile.

The hardware employed for the capacity tests should resemble the production hardware as much as possible to obtain reliable results. This holds true not only for the actual server hardware, which has to allow inferences about production, but also for simulated clients and the structure of the network. The clients have to run on a separate system. Otherwise the server and the load generator on the computers will compete with each other for resources. This precludes inferences about production since now the conditions during the capacity test are completely different.

In addition, the network infrastructure should be as close to production as possible: If in production different routers, switches or firewalls are between server and clients, this should also be the case for the capacity tests. If this

cannot be realized, at least a similar degree of delay should be built into the network communication with the help of simulators.

When a large load is supposed to be simulated, multiple load generators have to be installed on different computers in the network and coordinated. This allows one to create the load by multiple computers so that the possible load is not limited by the power of an individual computer.

5.2.8 Cloud and Virtualization

Especially in the case of capacity tests, the deployment automation of Continuous Delivery is advantageous: Setting up an environment for a capacity test does not take much time due to the automation of the deployment. It is also possible to use a Cloud solution as infrastructure. In a Cloud an environment can be rented for exactly the time that is necessary for the test. This decreases costs for a capacity test environment that resembles the production environment. Thus it gets more feasible to test on a production-like environment since it does not have to be bought but can be rented. The influence of differently dimensioned servers on the application can thereby also be tested simply: In the Cloud a computer can be booted anew and can be provided with more CPU capacity or memory.

Virtualization approaches in your own computing center provide similar options. However, in this case sufficient resources have to be available in order to set up such a production-like environment—that is, computers have to be bought—while in a public Cloud they only need to be rented.

Of course, the performance of a virtualized environment is not comparable to a physical environment, and a Cloud infrastructure is used by many customers whose activities might influence the capacity tests. However, production environments nowadays are also frequently virtualized environments and therefore have the same effects.

5.2.9 Minimizing Risk by Continuous Testing

As mentioned already, a fundamental risk with capacity tests is that a test at the end of the project leaves too little time to still implement changes. However, only at the end of the project the capacity of the application can really be tested since only then the complete functionality is correctly implemented—and only in this case the performance can be reliably evaluated.

Of course, an approach with such a test phase does not fit at all with Continuous Delivery. Instead of relying on a one-time test phase, in a Continuous Delivery pipeline a capacity test is started after each commit. In this manner the performance of the application can be ensured at all times. When new features are implemented, new capacity tests can be implemented at the same time in order to safeguard the new features with regards to performance. Thus, it will be clear relatively soon after a commit whether the performance of the application has suffered. The necessary optimizations can be started much faster and can focus on areas that have only recently been modified and that are therefore the likely cause of the problem identified in the capacity test, since the capacity test was still successful prior to the last changes. Likewise it is ensured that the application achieves the right performance at all times; otherwise this can only be known after the respective final test phase. This makes handling performance risks much easier.

5.2.10 Capacity Tests—Sensible or Not?

The fundamental challenge with capacity tests is to ensure that the performance and the behavior in production are modeled well. However, the model can hardly be perfect, either in regards to hardware or in regards to test data. Real users will always have surprising ideas, and the test environment in the end also has to be economical. Therefore, capacity tests always have to be connected to comprehensive monitoring in production. This allows one to align capacity tests progressively with the behavior in production. In addition, the performance in production can be evaluated. This knowledge can be used to deduce optimization strategies for the system.

The Continuous Delivery pipeline provides a possibility to bring changes faster and more reliably into production. The risk of performance problems decreases, since the team can react much faster. Therefore, it is not necessarily required to invest a lot of effort in capacity tests when performance and capacity are monitored in production and the team can react sufficiently fast to potential problems. The risk is further reduced since the changes in a release are smaller and can be rolled back. Thus, when a performance problem occurs in production upon a new software version, measures like the ones described in Chapter 7, "Deploy—The Rollout in Production," can be used to recognize the effects during the roll-out in

production and to stop the delivery of the software. This is also a possible answer to the limited predictive value of capacity tests due to non-linear effects.

5.3 Implementing Capacity Tests

For capacity tests there are approaches similar to those for acceptance tests:

- The tests can be implemented via an individual API. This API provides the functionalities of the application specifically for the capacity test (Figure 5.1 and can be implemented in an adapter. The load is created by a load generator, which calls the API. This allows one to implement functionalities that are specialized for capacity tests—for example, the initialization of test data. A disadvantage of this approach is that the API does not correspond to actual use. Therefore, the determined performance is not directly identical to production—another security factor for capacity tests. Certain parts of the application, such as the GUI, are not tested at all.

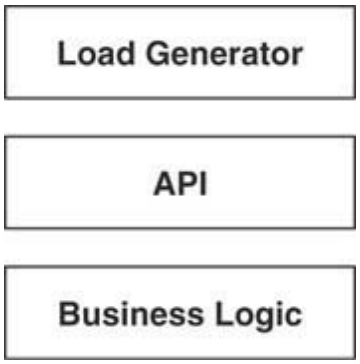

Figure 5.1 *Capacity tests with an API*

- The alternative is to use the interface with which the application is also used in other cases. This can be a web interface, a GUI, or a web services API. The load generator uses this interface and thereby creates the necessary load (Figure 5.2). Thus the application is used exactly as it would be in production. However, the implementation of the capacity tests becomes more complicated this way since the respective interface needs to be used even though it was not specifically designed for capacity tests. The test can even comprise measuring the performance within the browser for rendering and loading of the JavaScript.

Figure 5.2 *Capacity tests through the Web GUI*

The capacity tests need to have an unambiguous result. The application either fulfills the requirements or it does not. If it does not fulfill the requirements, the capacity test has to fail. This ensures that a potential performance problem is really noted and fixed. When only a report is delivered, it has to be manually analyzed to determine whether the values are sufficient or not. This step will surely be forgotten at some point. Then it can happen that a performance problem is not recognized in time. Thus, there is a danger that the results of the capacity tests are ignored.

However, detailed results are nevertheless good to have to be able to identify the cause of the problem. Therefore, reports can be useful to indicate exactly where a performance problem was created.

This topic will be discussed in detail in Chapter 8, "Operations."

5.4 Capacity Tests with Gatling

For the example application we will use Gatling[1] as a tool for capacity tests. This tool can serve as a load generator for simulating the behavior of real users and thus for implementing capacity tests. Gatling offers a number of advantages:

- It is implemented in the programming language Scala. Thus, it can use the Java Virtual Machine (JVM) and at the same time profit from Scala advantages with regards to the implementation of concurrent systems. Especially for a load generator, it is important that many parallel users can be simulated in a performant manner to allow for a realistic load.
- Gatling capacity tests are written in an expressive DSL. This makes it easy to formulate and extend scenarios on your own. In the end the

entire power of Scala as a programming language can be used for this.

The registration of customers will again serve as our example. In the first step the Gatling recorder has to be configured (Figure 5.3). It serves as a proxy for a web browser and records all requests of the web browser. The configuration mostly determines under which port the proxy can be reached and where the recorded script is stored.

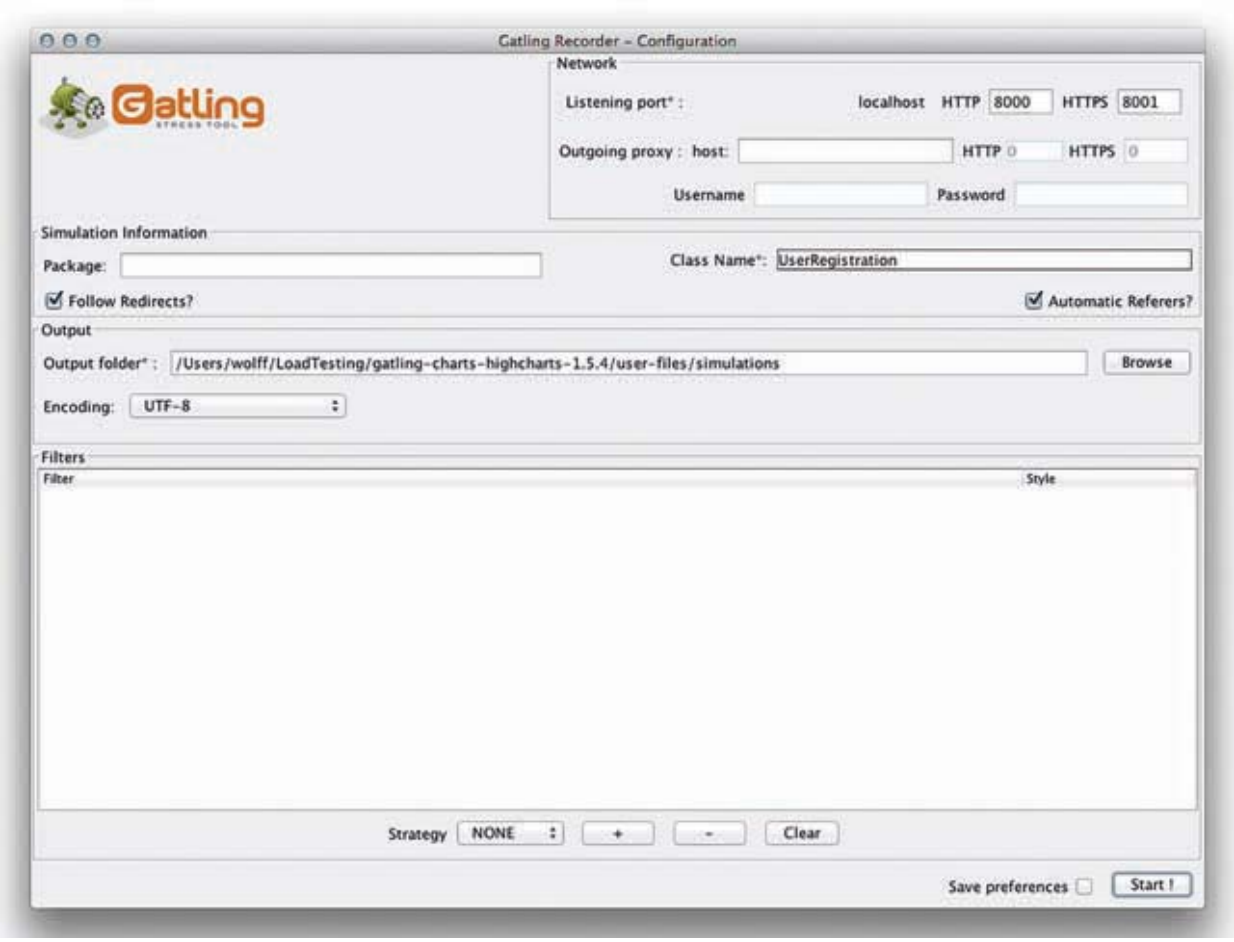

Figure 5.3 *Configuration of the Gatling Recorder*

Figure 5.4 shows an example of the recorded communication. In the example the recorder recorded the call of the initial page of the example application and the registration of a new user. For the current step the data for the registration of a user are recognizable as parameters. The response of the server is an HTML page. However, this page is packed and therefore not readable. The recorder saves the interactions as Scala source code.

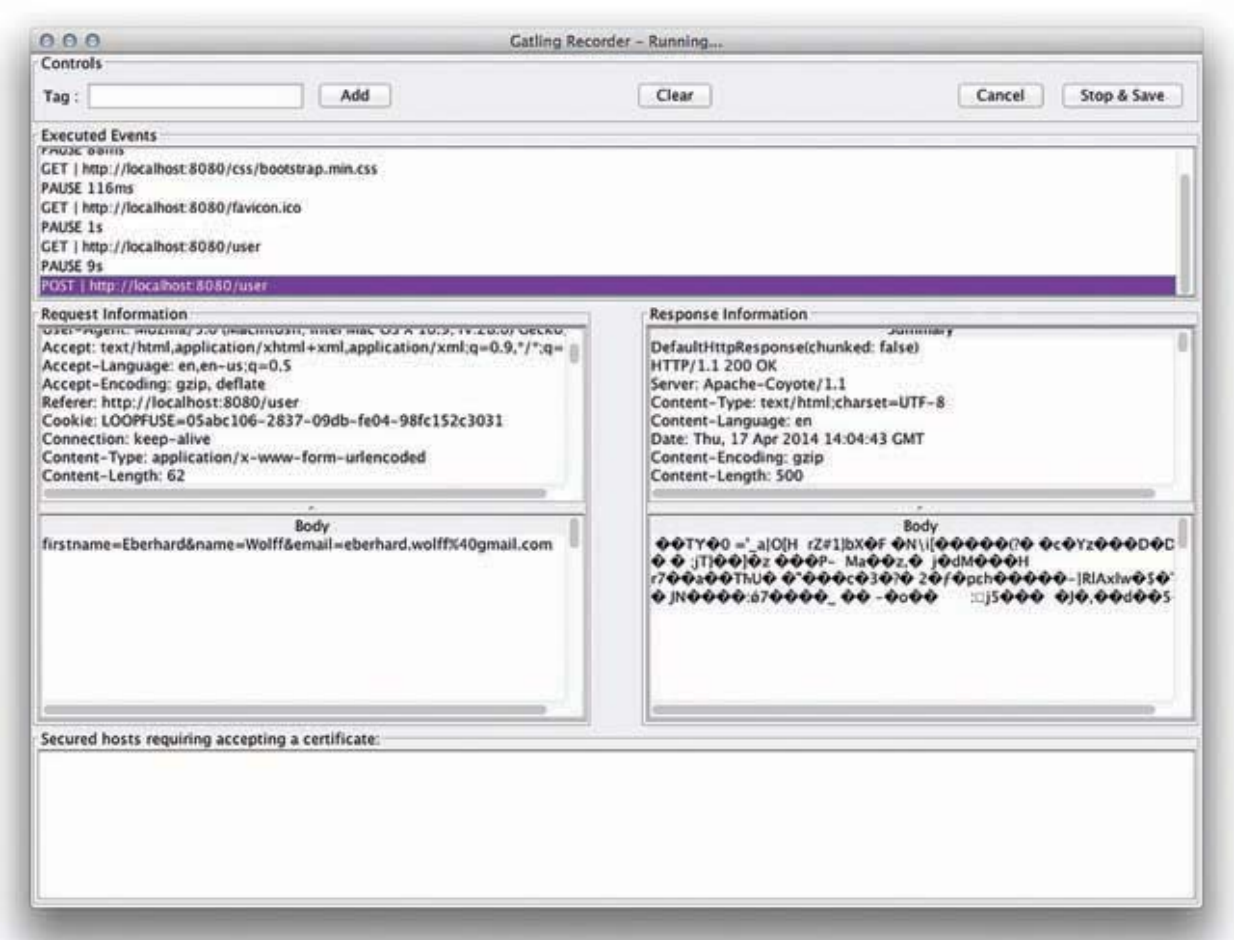

Figure 5.4 *Configuration of the Gatling Recorder*

Listing 5.1 shows the code after it has been manually reworked. Now there is the `emailFeeder`: This object allows the generation of a new email address upon each test run. In the concrete example this is achieved with the help of a UUID; that is, a globally unique string of numbers. While a fixed email address is used for the recorded interaction, this test code uses an individual email address for each run. Thus, appropriate test data are injected into the test with the help of the `emailFeeder`. If the tests were all performed with the same email address, problems would arise since two users are not allowed to have the same email address. When two users simultaneously register with the same email address, an error occurs. During a capacity test the simultaneous use of the application by several users is simulated so that each simulated user needs to use a different email address. Such parameterizations of test data are often necessary to simulate a larger number of users.

Listing 5.1 *Gatling Scala DSL code for a capacity test*

Click here to view code image

```
class UserRegistration extends Simulation {
  val emailFeeder = new Feeder[String] {
    override def hasNext = true
    override def next: Map[String, String] = {
      val email =
        scala.math.abs(java.util.UUID.randomUUID.
getMostSignificantBits)
         + "_gatling@dontsend.com"
      Map("email" -> email)
    }
  }
  val httpProtocol = http
    .baseURL("http://127.0.0.1:8080")
    .acceptHeader(
     "text/html,application/xhtml+xml,"+
"application/xml;q=0.9,*/*;q=0.8")
    .acceptEncodingHeader("gzip, deflate")
    .acceptLanguageHeader("en,en-us;q=0.5")
    .connection("keep-alive")
    .header("Cache-Control", "max-age=0")
  val formHeader = Map(
    "Content-Type" -> "application/x-www-form-urlencoded")
  val scn = scenario("Registration")
    .repeat(10) {
      (
        exec(http("GET index")
.get("/"))
        .pause(88 milliseconds)
        .exec(http("GET css")
          .get("/css/bootstrap.min.css"))
        .pause(1)
        .exec(http("GET form")
          .get("/user"))
        .pause(7)
        .feed(emailFeeder)
        .exec(http("POST user data")
          .post("/user")
          .headers(formHeader)
          .formParam("firstname", "Eberhard")
          .formParam("name", "Wolff")
          .formParam("email", "${email}"))
        .pause(4)
        .exec(http("POST delete user")
          .post("/userdelete")
          .headers(formHeader)
          .formParam("email", "${email}")))
    }
```

```
    setUp(scn.inject(rampUsers(5) over (10 seconds))).
     protocols(httpProtocol)
}
```

Afterwards `httpProtocol` sets the options for the HTTP protocol in the code—options like different headers and the URL of the application. The variable `formHeader` contains additional headers for the transfer of form data.

The actual scenario is quite simple: The test is performed ten times. First, the test opens the homepage of the application (`"GET index"`) and reads the used CSS (`"GET css"`). Afterwards the form is requested (`"GET form"`), and submitted (`"POST user data"`). Finally the customer is deleted again (`"POST delete user"`). In between there are pauses in which real users would, for instance, enter data. These pauses are also called "user think times." They serve to realistically simulate human users who need some time to enter data and consider what they want to do next.

Finally, the test is started in the code. In the test five users are simulated. The load is increased over 10 seconds—that is, additional simulated users are started step by step. This helps to avoid all simulations being started at once, thereby causing the application to suddenly experience a very high load. Such a scenario is unrealistic and can cause a breakdown of the application, which would never occur in production.

While the test runs, the response times and all other behaviors are measured for the different steps. The result for `"POST user data"` is depicted in Figure 5.5. At the beginning the application still needs to warm up. Afterwards the response times remain constantly low apart from one outlier.

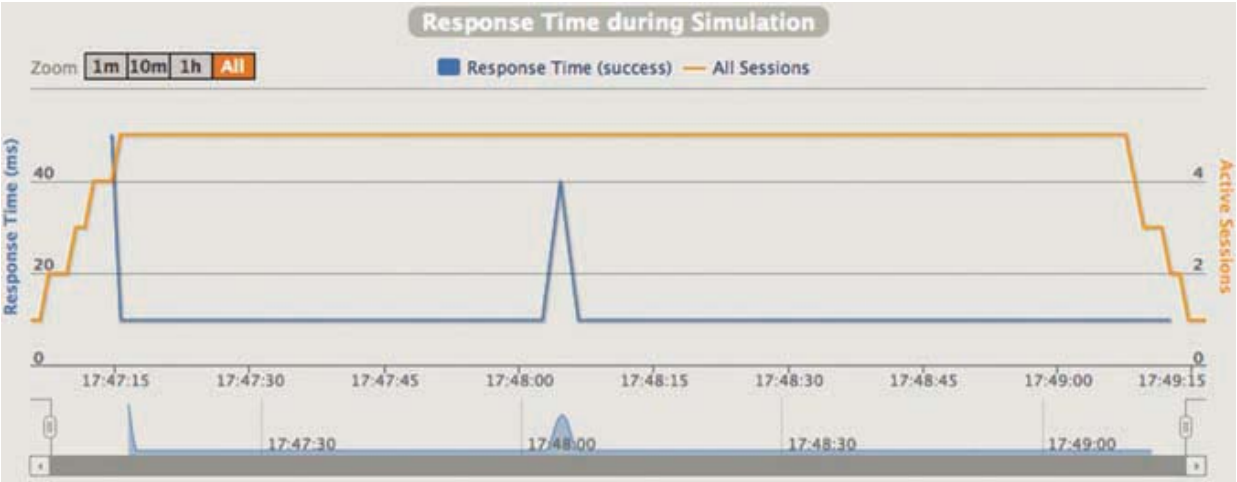

Figure 5.5 *Results of a capacity test*

5.4.1 Demo versus Real Life

To make the demo easy to understand and use, it unfortunately violates established best practices. For one thing, the application and the load generator run on the same computer. Whenever realistic scenarios have to be tested, the load generator and the application have to run on different computers. Otherwise they will compete for resources, which is an unrealistic scenario compared to production. In addition, the capacity test delivers only a report. Thus, the conditions that will make the test unambiguously fail if the application is not sufficiently fast have not been defined.

Try and Experiment

The example project can be found at http://github.com/ewolff/user-registration-V2. To execute it:

- Install Maven (see http://maven.apache.org/download.cgi).
- Afterwards execute `mvn install` in the directory `user-registration`.
- Now you can run the capacity tests:
  - In the subdirectory `user-registration-capacitytest-gatling` you can execute the capacity tests with `mvn test`.
  - Afterwards the results can be found as HTML files in the directory `target/gatling/results`.
  - Search for the Gatling Cheat Sheet in the internet.
  - Search information about assertions in Gatling. How do they work in Gatling 2.0?
- In `src/test/scala` you can find the test.
  - Modify the test so that it fails due to bad performance.
  - To do so insert an assertion. Searching the documentation for this feature was one of the earlier tasks.
  - The assertion has to be defined upon calling `setUp` just like the `httpProtocol`.
  - For example, you can check an average time of 1 ms for `"POST user data."` In that case the test should fail.
- Look for the documentation of the Gatling Recorder.
  - Download Gatling and let the recorder run.
  - Configure a web browser with the proxy.
  - Now record the search for a customer with the help of the recorder. It is OK when no customer is found, since the search will be executed anyhow.
  - Run this test also as part of the capacity test. To do so you have to copy the code for the test into the project.
- Run the test with different test data—for example, use a data set with 0, 1, and a large number of results. To do so use a feeder like the one used previously for the email addresses for the registration.

5.5 Alternatives to Gatling

Currently, Gatling only offers support for the HTTP protocol. However, it is possible to write extensions for other protocols on your own. Due to the limitation on HTTP, Gatling cannot easily be used for implementing a capacity test for an application via an API.

5.5.1 Grinder

A possible alternative to Gatling is Grinder.[2] With this tool capacity tests are implemented in Jython. This is a variant of the programming language Python which is implemented in Java. An alternative programming language is Clojure which resembles LISP, but also runs on a JVM. Therefore, all APIs that can be called with Java can be used with Grinder for the implementation of capacity tests. Thus, implementing a capacity test against an API is relatively easy. In addition, there are prepared classes—among others, for HTTP or JMS to test an application on a server with these protocols.

5.5.2 Apache JMeter

Another option is Apache JMeter.[3] This tool has a graphical editor for the tests that supports, in addition to HTTP, different protocols like JMS, SMTP, and FTP. It is also possible to initially record the interactions with a website and then to replay them. JMeter can also visualize the results. JMeter is surely the most popular and most widely used tool at the moment. There is a commercial offering for JMeter called Blazemeter with which computers are provided in the Cloud for executing the load test. Of course, this works only with public websites.

5.5.3 Tsung

Another interesting tool is Tsung,[4] which is written in Erlang. This programming language is specialized for distributed systems. In addition, with Erlang's programming paradigm a lot of network traffic can be generated with only few threads. Clusters can also be operated relatively simply. Therefore, Erlang offers ideal preconditions for implementing load test tools. Tsung supports protocols such as HTTP, WebDAV, SOAP,

PostgreSQL, MySQL, LDAP, and Jabber/XMPP. Different user behaviors can be simulated, and the performance of the systems can also be monitored.

5.5.4 Commercial Solutions

Commercial solutions include HP LoadRunner[5] and Rational Performance Tester.[6] In addition, there are commercial providers which offer online load tests. In this case the test runs—as with Blazemeter—in the Cloud, so you need neither your own servers nor a laborious installation. In addition, the resources for the capacity tests are only paid for the time they are actually used. Such offerings make it possible to use a large number of computers and thus to create a large amount of load. Often the providers also offer very simple and efficient possibilities to evaluate the tests and make inferences. Moreover, these tests use the website exactly as a customer would use it—that is, on production hardware with the complete network topology.

Spirent Blitz[7] is another example of a provider that allows the development and execution of complex load tests with a simple website. LoadStorm[8] provides similar possibilities. Here, the load test can be implemented by recording interactions in the web browser.

5.6 Conclusion

Capacity tests ensure that an application supports the expected number of users and that it is also sufficiently fast in doing so. Continuous Delivery aims at performing capacity tests if possible upon each commit, to get continuous feedback about the current performance of the application. An important prerequisite is to determine the correct performance requirements for the application and to set up an environment that in fact allows inferences about the behavior in production.

The tests can use the application either via a special API or via the interface that is also used by the users—for instance, the web interface.

The example with Gatling shows a typical procedure for capacity tests: The interaction with a website is recorded. Based on this a suitable test is implemented with a DSL that evaluates the performance of the application. The test can still be fine-tuned in the code. It can become part of the Continuous Delivery pipeline so that the current performance can be measured upon each pipeline run.

Alternatives to Gatling are Grinder, where the tests are written as Clojure or Jython programs, and Apache JMeter, which capitalizes on a graphical implementation of capacity tests.

Endnotes

1. http://gatling.io/
2. http://grinder.sourceforge.net/
3. https://jmeter.apache.org/
4. http://tsung.erlang-projects.org/
5. http://www8.hp.com/us/en/software-solutions/loadrunner-load-testing/
6. http://www-03.ibm.com/software/products/en/performance
7. https://www.blitz.io/
8. http://loadstorm.com/

Chapter 6. Exploratory Testing

6.1 Introduction

So far, the focus has been on automated tests. In this chapter the spotlight now is on manual testing, and especially on exploratory tests. Section 6.2 discusses why and where exploratory tests make sense. Section 6.3 shows a concrete procedure for performing these tests. A comprehensive introduction into this topic can also be found in the book "Explore It!"[1] that explains the concepts in much more detail.

6.1.1 Exploratory Tests: An Example

Big Money Online Commerce Inc. from section P.2 had a lot of experience with manual tests—in the end there was a comprehensive manual testing phase for each release. However, due to the introduction of automated acceptance tests the focus of the manual tests changed: Now their main objective is to thoroughly test the functionality of the new features. In this context the most important point is to understand the software from the domain perspective. This is the characteristic feature of manual exploratory tests. In addition, the exploratory tests are used to evaluate and improve select software characteristics: Is the usability sufficient? Is the security good enough? Are there perhaps extreme cases nobody has thought about so far?

6.2 Why Exploratory Tests?

Continuous Delivery focuses mainly on automated tests. Manual tests are expensive, especially when the application is tested frequently. Every test run has to be performed and analyzed by testers. Automated tests on the other hand have to be automated only once. The subsequent runs of automated tests are practically free of cost. Especially with Continuous Delivery, applications are very frequently deployed again and consequently also tested often. Therefore, tests should be automated since costs are otherwise too high.

6.2.1 Sometimes Manual Testing Is Still Better

Nevertheless, in some cases manual tests can be the better approach: When new domain features are dealt with, experts should initially test the application manually. To do so the testers must be aware of the domain contexts so that they can more reliably evaluate the features. Requirements or user stories can serve as the basis for these tests. In addition, the manual tests are also an option to once more critically scrutinize the requirements and thus to identify possible errors. Naturally, this cannot be achieved with automated tests since those need to be based on defined requirements.

Exploratory tests exploit the fact that testers will often identify problems only when they deviate from scripts and independently examine and explore the application. In doing so they are guided by their experience and the known error-prone aspects of the application. Of course, such an approach cannot be automated, but only experienced testers can perform such tests.

In this manner exploratory tests can be a good tool to have a closer look at known error-prone features or certain error behaviors. When for instance a certain functionality or a certain module more frequently causes problems in production, exploratory tests can serve to dissect this area more carefully to identify additional errors.

6.2.2 Test by the Customers

Sometimes customers might want to test a release on their own prior to its acceptance. Again, in this case the focus is on the new functionalities. If the customer also tests older features, the team should use the manual tests to improve the trust in automation. For example, the team can demonstrate how the features the customer is evaluating manually are tested in an automated fashion. In addition, the team can implement, as automated tests, acceptance tests of the customer for new features.

Tests for regressions should never be performed manually since the costs are too high when each release has to be manually tested for them. On the other hand, not each and every requirement has to result in an automated test. It is sufficient when the automated tests provide the team with sufficient confidence for rolling out the application. To achieve this the application usually has to be tested at least to such an extent that all use cases pass the tests once without errors.

6.2.3 Manual Tests for Non-Functional Requirements

Exploratory Tests focus on the newly implemented functionalities of the software. However, manual tests can also be superior to automated tests when it comes to non-functional requirements:

- *Usability tests* evaluate how easy it is to use an application. They are very simple to perform for humans, while it is very difficult to test usability in an automated fashion.
- The same holds true for *look and feel* or the adherence to *design guidelines*. Humans can spot errors frequently at first glance, while automated processes that do this are very difficult to implement.
- Finally, tests for *security problems* are often easier to implement by experts than by automated processes. In this area code reviews or penetration tests are also frequently employed, which are too complex for automation.

6.3 How to Go About It?

Exploratory tests primarily target new business functionalities. Thus, the testers have to evaluate the application based on the requirements. In this process tools like Selenium (see Chapter 4, "Acceptance Tests") can support the automation of routine activities. They can record interactions with the web GUI of the application and play them back. Subsequently, the testers can go on working manually with the application. Such automation scripts can also be the basis for a later complete test automation.

6.3.1 Missions Guide the Tests

Test plans describing which functionalities are supposed to be tested in which manner can be helpful. However, they can also be an indication that the tests have reached a level of maturity and complexity where automation would be beneficial. Therefore exploratory tests don't use test plans but rather charters (see section 6.3.6).

6.3.2 Automated Environment

An environment that is set up in an automated manner is a good basis for exploratory tests. In this environment the software and suitable test data have to be installed. Only those releases that have previously successfully passed the other stages—that is, the tests of the commit phase, the

automated acceptance tests, and the capacity tests—should be evaluated with exploratory tests. Otherwise the effort for manual testing is wasted on releases that are of doubtful quality.

6.3.3 Showcases as a Basis

Showcases can serve as a concrete basis for the exploratory tests. For agile projects at the end of an iteration the customer is often shown in the context of a showcase how the newly implemented features work and what can be achieved with these features. It is precisely these features that are also supposed to be addressed by exploratory tests. Therefore, one result of the exploratory tests could be a successful run through such a showcase.

6.3.4 Example: An E-Commerce Application

Let us assume for example that in an e-commerce application the possibility for express orders was implemented. In the context of exploratory tests an expert can start by creating an express order and testing whether the order is correctly processed. In addition to this obvious test case, the tester can evaluate what happens when customers order, via express order, items that are not deliverable at the moment or for which the delivery time is too long for an express order. Likewise, the cancellation of express orders can be tested. In doing so special attention can be paid to known error-prone processes—for example, updates on the status of the order. Due to the knowledge of the tester about the business processes and about the current system and its error-prone areas, manual tests can be highly effective. Of course, in this way a good basis for automated tests is also generated, which will later safeguard the application against regressions.

In addition, exploratory tests can examine usability and look and feel with established techniques from this area. Here, user interviews can also be helpful, or the interactions with the software can be recorded and subsequently evaluated.

Another component of manual tests can also be security tests—for instance, penetration tests or code reviews with a focus on security problems. Likewise, manual capacity or performance tests are an option. However, such tests often require a load driver, so they can hardly be implemented without automation.

6.3.5 Beta Tests

Another option for exploratory tests are beta tests. Here, the software is distributed to a limited circle of users who try it out and are supposed to report back errors and suggestions for improvements. In some cases, this allows, at the same time, testing of whether the new software version generates, for instance, a higher volume of sales.

6.3.6 Session-Based Tests

For structuring exploratory tests the method of session-based tests[2] is useful. Here, the tests are divided into sessions. Each session has a mission that defines which kind of errors or which problem is the focus of the session. The session is, so to speak, an expedition into the application. The target of the expedition is determined by a charter. For a geographical expedition the charter can determine, for instance, that the goal is to explore a certain landscape. For exploratory tests a charter likewise determines the objective of the session. A charter has the following form:

Explore ... (target)
with ... (tool)
to find … (information).

These three parts determine the following aspects:

- The *target* is the part of the application that is supposed to be the focus of the test—that is, a certain function, requirement, or module.
- The *tool* defines which means the test shall employ. Possible tools are specific data sets or software tools.
- Finally, the *information* determines what is supposed to be the result of the test. Potential results are insights about security, speed, or reliability. Thus, exploratory tests do not necessarily only have to deal with domain requirements and errors.

In the context of the customer registration the following charter for an exploratory test would be conceivable:

Explore the customer registration

with suitable data sets

to ensure correct internationalization.

As a result of this charter the customer registration could be tested with names containing unusual letters—or even composed of an entirely different letter set such as Korean, Japanese, or Chinese characters. Maybe the division into first and last name does not fit for all countries? And which letters are actually allowed as part of email addresses?

A completely different charter could be:

Explore the customer registration

with OWASP attacks

to identify security gaps.

OWASP[3] is a collection of the most common security gaps in web applications. Here, the exploratory test evaluates security—a non-functional requirement.

Both charters share the characteristic that they are formulated in a general manner. In the end manual tests should not be entirely predefined; only a goal and a rough idea of the approach should be given. The testers themselves are responsible for the concrete realization of the test session.

Charters can be derived from requirements or can serve to more carefully explore certain risks—for instance, to identify security gaps early on.

The formal result of an exploratory test is the report of the session that contains, in addition to the charter, information about how the test was performed, which errors were identified, and how long the session lasted. Finally, a retrospective of the session is supplied. A session lasts only a few hours. In this manner the manual testing can be divided into different sessions with specific aims and thereby be structured.

Testers are relatively free in their choice of tools. In addition to tools for test automation, testers can evaluate log files or system metrics. Of course, the tools for monitoring or log analysis presented in this book are very helpful for this (Chapter 8, “Operations”). In addition, exploratory tests can build on interfaces, APIs, or web services. Naturally, this requires a solid

technical understanding on the part of the testers. Such tests are only possible when the testers are familiar with the necessary tools. In extreme cases the testers might even have to write the required software on their own.

During the test sessions the testers explore the software with different techniques:

- Applications have *variables*. Depending on the type they can contain different values. In this context it is worthwhile to test different variations—for instance, the use of Chinese characters in a first name.
- *Interactions or processes* can be modified. This allows one to examine what happens when the business processes are not run through in the manner that was originally expected.
- In addition, testers can explore *entities and relationships between entities*. This might allow one to create invalid entities.
- Many applications possess certain *states and transitions between the states*. In this context testers can test whether the states are sensibly implemented or whether surprising states can occur.

These techniques allow the testers to realize the different charters and thus to explore potential weaknesses of the software.

Try and Experiment

- Implement one of the two test charters mentioned in the text for the customer registration.
- Which additional test charters would make sense in the context of the customer registration?

Have a look at a project of your choice:

- Select a functionality. What might a sensible charter look like for testing this functionality?
- Which non-functional requirements (for example, security or performance) are especially important for the application? Design a charter to test them.

6.4 Conclusion

To clearly state it one more time: In a Continuous Delivery project not too many manual tests should be implemented. Continuous Delivery means that deployment and testing occur more often; in fact, very frequently. Therefore, it is nearly always worthwhile to automate tests. However, there is no rule without exception: Exploratory Tests of new features are in fact only possible in a manual manner. This allows one to get a better idea of error-prone aspects of the application and to better employ tests for safeguarding against regressions. The same holds true for tests of usability or look and feel—and of course also for tests concerning other non-functional requirements, like penetration tests. Exploratory tests for new features should result, in the end, in automated tests. In addition, exploratory tests can be a good option for exploring certain aspects of the application more closely. In the end not all errors can be detected by automated tests.

Endnotes

1. Elisabeth Hendrickson: *Explore It!—Reduce Risk and Increase Confidence with Exploratory Testing,* Pragmatic Bookshelf, 2013, ISBN 978-1-93778-502-4.
2. http://www.satisfice.com/sbtm/
3. https://www.owasp.org/index.php/Main_Page

Chapter 7. Deploy—The Rollout in Production

7.1 Introduction

In the end the deployment in production is just another deployment—and Chapter 2, "Providing Infrastructure," already discussed tools for deployment in detail. However, during the rollout in production the consequences of a failure are far greater than during a rollout in a test environment. Therefore, this chapter introduces approaches for further minimizing the risk associated with rollouts in production. Section 7.2 discusses the idea that it is desirable to be able to undo a software rollout. In case of a problem a new version that removes the error can immediately be rolled out as an alternative: a roll forward (section 7.3). Another possibility is introduced in section 7.4: For a blue/green deployment an entirely new environment is set up for a new release. This allows one to run the new release in parallel to the old one and to then switch from the old to the new one when everything works fine. With the approach of canary releasing, on the other hand (section 7.5), a release is initially only rolled out on a limited number of servers before the software is deployed on all computers. Section 7.6 shows how continuous deployment can be used to deploy every code change into production. Most approaches presented in this chapter deal with web applications. Section 7.8 explains how the rollout also functions for other types of applications.

Databases are a special challenge for deployments in production. This topic is going to be covered in detail in section 11.5 so it will not play a prominent role in this chapter.

7.1.1 Deployment: an Example

Initially Big Money Online Commerce Inc. from section P.2 assumed that due to the infrastructure automation a deployment in production would be quite easy and that the associated risks would be significantly reduced now. And indeed, this was the case. However, the deployment still required that the website be shut down, and consequently it was not available. Further, one time the deployment still failed because the production environment differed from the test environment. This caused a production breakdown.

Therefore, additional measures had to be implemented to bring the software safely into production. To just update the production environment to the current version of the software was not sufficient. There had to be an option to fall back to the old version or to avoid production outages by other means. Such measures will be introduced in this chapter.

7.2 Rollout and Rollback

During a rollout in production the risk management is particularly important. The most obvious approach to minimize the risk is to provide the option for a rollback. When the new software version causes a problem, it can be replaced by the old version via a rollback.

When this alternative is resorted to, something has indeed gone wrong and the application is not available at the moment. Therefore, it is very important that this process really work. Consequently, it has to be tested to ensure that the procedures in fact function in an emergency.

7.2.1 Benefits

The process itself should not be very complex—after all, the old version of the application was successfully brought into production and the required processes should not differ fundamentally from the processes for the new version. Only the handling of the database might be problematic (see sections 2.9 and 11.5), since changing database schemas in relational databases, especially in the case of larger amounts of data, can be complicated and unreliable. For a rollback there is no special additional expenditure except that the old version has to be held available and a process has to be established to bring the old version back into production.

7.2.2 Disadvantages

The process for the deployment of the old version on a production environment cannot be entirely tested since it can never be run on the production environment. Therefore, a rollback is always associated with a certain degree of risk. The risk is even higher if changes to the database occurred, since data loss must be avoided.

In a rollback the old version has to be brought back into production. This takes some time so that it is inevitable that the application will be down for a certain time. Depending on the needed availability this might mean that a

rollback is essentially impossible. Another problem is database changes: It is very difficult to undo such changes. This increases the risk that a rollback will not work in the end, renders the process more complex, and leads to a longer outage. Thus rollbacks are in many scenarios only a theoretical option, unless special approaches are taken as described in sections 2.9 and 11.5 to handle the databases in a smarter manner.

Another problem is that, while after a rollback the application is available again, the search for the cause of the problems with the new version is difficult. The data and log files on the production computers are often crucial in identifying the source of the problem—however, it is exactly this information that has likely been deleted during the rollback. Moreover, since a rollback mostly has to be done as fast as possible, it is likely that important information got deleted by mistake in the rush. In this case the only approach left to dissect the problem is to simulate the situation in production. However, this strategy has only limited hopes for success: The application has already been comprehensively tested in a production-like environment, such as staging. When the error only occurs during the introduction in production, then it is quite likely that it will be hard to reproduce this error outside of production.

7.3 Roll Forward

Alternatively to a rollback, a roll forward—also known as a patch forward—can be used. In this case a new version of the software is deployed upon an error. This version fixes the error. Of course, this change also has to be tested. However, this does not create much expenditure due to the Continuous Delivery pipeline. In the end this approach trusts in the ability of the Continuous Delivery pipeline to deliver the changes sufficiently fast to remove the error quickly.

7.3.1 Benefits

Since the change is relatively small, the deployment of this version can be performed very rapidly. Typically, it is as laborious as a rollback. However, this approach is less complex than a rollback: A release can do without a rollback process. In the case of database changes a rollback of the changes can be very difficult, whereas upon a roll forward, the changes in the database often remain; this facilitates this approach.

In addition, with this approach there are no corner cases: The procedure for solving a problem is identical with the approach for rolling out a new release—and ideally, this process is performed multiple times per day. Thus no corner cases have to be tested any longer.

7.3.2 Disadvantages

With roll forward there is no easy procedure to fall back to a release that works for sure. Of course, code changes can be quickly undone with version control; however, it is not always obvious which part of the code is causing the problem. And if the database has been changed, the old code might not be able to work with the new database. If the error is more complicated in the end, a solution might have to be found under high time pressure.

However, these risks can be reduced with additional measures. Chapter 11, "Continuous Delivery, DevOps and Software Architecture," will discuss how good architecture can reduce the adverse effects of a component failure and how changes to the database can be handled in a better way. The techniques that are to be introduced in this chapter are also helpful in reducing the risk. In the end, a roll forward is also an indication that Continuous Delivery has been very well implemented by the team. The team has sufficient trust in its ability to release and in its Continuous Delivery pipeline that in an emergency it would rather choose to deliver a new release than go back to an old release. At the same time the complexity of the Continuous Delivery pipeline can be reduced since no rollback has to be provided for anymore.

7.4 Blue/Green Deployment

With blue/green deployment the new software version is installed on an entirely separate system. To finally really bring the new version into production only a switch from the current environment to the new environment has to be performed—for instance by reconfiguring a router (Figure 7.1).

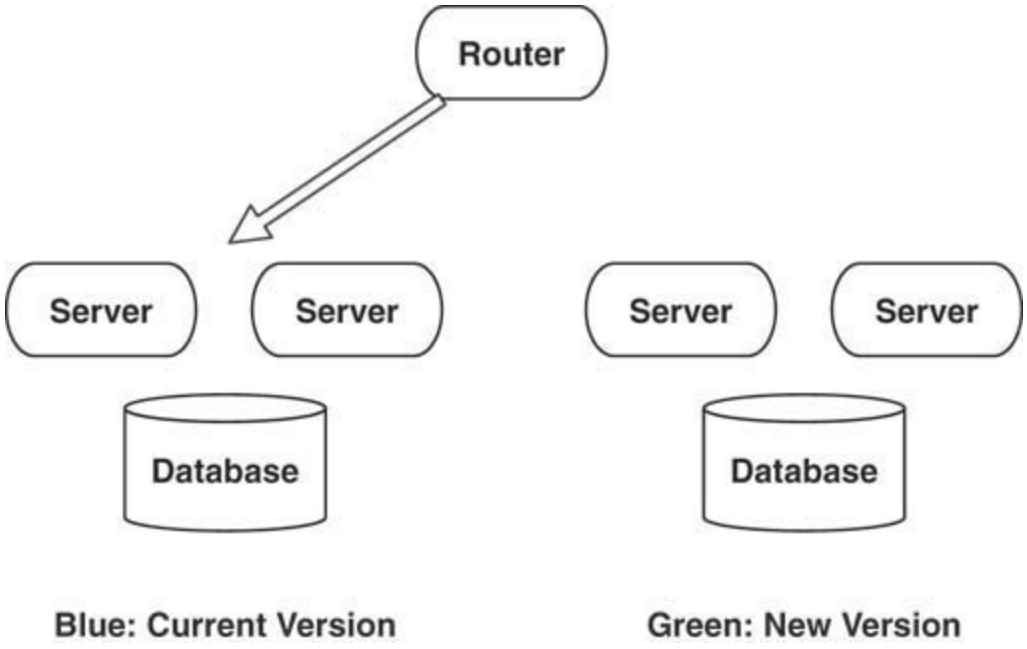

Figure 7.1 *Blue/green deployment*

7.4.1 Benefits

With this approach a new software version can be brought into production without risking downtime of the application. In addition, it is possible to first test the application comprehensively on the new environment—with regards to performance as well as function. If a problem should occur nevertheless, the fallback is also very simple; the router only has to switch back to the old environment.

7.4.2 Disadvantages

An obvious problem with this approach is the high consumption of resources: Two environments have to be provided at the same time, and both need to be dimensioned sufficiently large to handle production load. This means that every machine necessary for production needs to be provided twice. This problem can be addressed by different strategies: With a public Cloud the problem is far smaller. The second environment only has to be provided for a limited amount of time while the new is deployed and tested. When the new version runs reliably, the old one can be shut down. Since in a public Cloud resources are only paid for while they are being used, much less cost arises than would be the case in a company data center, which would have to provide the resources permanently.

However, in a non-Cloud environment the resource consumption can also be dealt with. For example, a staging environment can serve as a second production environment. The last tests prior to the release into production are performed on such an environment. Consequently, this environment has to match the production environment as closely as possible. When going from staging to production, the application has only to be reconfigured. For instance, test data have to be replaced by actual production data, and the third-party systems from production have to be called instead of the mocks. Of course, the process for turning a staging version into a production version has to be automated to prevent errors.

In addition, it is also possible to install the new software version next to the production version on each computer. Both versions have to employ different ports to be separate from each other. Therefore, the setup becomes more complex in this case. Further, the individual servers have to be dimensioned in such a way that they can support two versions of the software at the same time.

In this scenario it is again the databases that pose a problem, since they have to be kept synchronously on both environments. One option to circumvent this problem is to set the database on read-only and to thereby disallow any changes to the data for a certain time. Additional alternatives are described in sections 2.9 and 11.5. But if it is not possible to establish a secure way to handle the databases, then blue/green deployment is no more viable than the rollback approach.

7.5 Canary Releasing

Canary releasing (Figure 7.2) is another option to reduce the risks associated with a release into production. Here, the new software version is initially deployed only on one or a few servers in the cluster. These servers can be set up in such a way that they initially cannot answer requests from users. This allows one to test them first and to operate them under reduced load at the beginning. If the software withstands the test, it can gradually be deployed on more servers until finally all servers are switched to the new version.

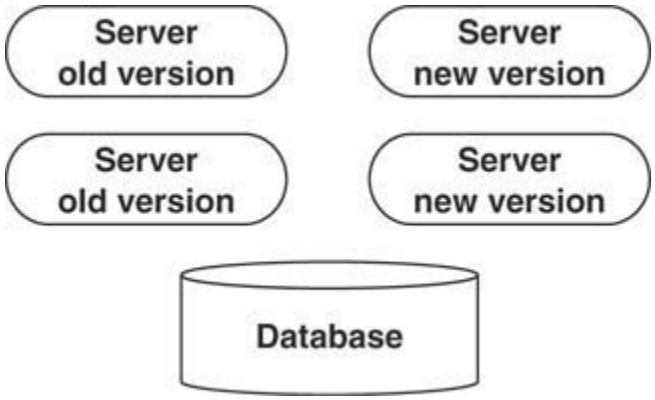

Figure 7.2 *Canary releasing*

This approach has been named after a strategy used in coal mining. Coal miners took canaries down with them into the mine because these birds react very sensitively to toxic gases: When the canaries get nervous or pass out, this indicates that it is probably better to leave the mine. The strategy behind canary releasing is similar: When the new software does not function on the first servers, it is not deployed any further. Thus, this approach installs a kind of early warning system.

7.5.1 Benefits

This approach allows one to deploy new software versions without any application outage (zero downtime deployment). At the same time this strategy does not require many additional resources. As long as the rollout does not take place during a time of high load, it should be possible to use some servers for the deployment of the new software without them having to take over load immediately. This also opens up the opportunity to test the application in the production environment without causing potential problems for users. During these tests the application can be screened for problems in the business logic, and its performance in the production environment can be evaluated.

Naturally, rolling back to the old version is very simple in this context: It only requires reinstalling the old software version on the server. This should not pose any problems, especially in the case of automated installations. Alternatively, the servers with the new version can be shut down, and servers with the old version can be set up again in an automated fashion.

7.5.2 Disadvantages

This approach absolutely requires that database and third-party systems support both versions of the software. This increases complexity. Therefore, no more than two software versions should be in production at any time. Having more versions simultaneously in production leads to very high complexity and is also of no use. The only objective should be to switch the entire production environment step-by-step to a new software version. The only reason to have two software versions simultaneously in production is to minimize the risk associated with a release into production. When two different software versions are permanently simultaneously in production, numerous problems arise—for example, errors would always have to be fixed in both versions, and Continuous Delivery pipelines would have to be established for both versions as well. Therefore, this state should only last for a few hours. If the new version proves not yet suitable for production, it should be rapidly removed from production; on the other hand, if the version performs well, it can simply be rolled out to all servers.

7.6 Continuous Deployment

If the deployment pipeline is largely automated, the application can be deployed in production in a fully automated manner for each code change. This is termed continuous deployment. This approach is a radical renunciation of classical approaches, and at first glance seems very hard to implement. In the end it is for a good reason that blue/green deployments or canary releasing take comprehensive precautions to ensure that there are no problems in production, or at least to ensure that potential problems can be rapidly fixed—preferably without the users noticing any problems at all.

However, continuous deployment can of course also employ strategies like canary releasing to first deploy the application on a limited number of servers before it is actually used by all users. When each change goes into production, the changes are smaller so that the risk decreases. In the end, with a small change there is of course less potential for problems than with a large modification. Therefore such a deployment cannot be compared to a classical release with regards to its extent. If the architectural approaches described in Chapter 11 are realized, it is even possible that large parts of the application are not changed at all so that the risk decreases even further. In addition, organizational measures can help to decrease the risk still further. For example, deployments can only be allowed during office hours

and information from monitoring can immediately be passed on to the developers to identify potential problems quickly.

In the end, continuous deployment can only be sensibly implemented with feature toggles (section 11.7): Initially, new features are not activated in production. Only when they are fully implemented, tested, and cleared are they activated in production. Without this decoupling of new features from the releases, new releases cannot reliably be constantly brought into production. Since manual testing mostly concerns such new releases, this stage can be omitted if applicable.

7.6.1 Benefits

Another advantage of continuous deployment is that the time required for fixing a problem in production further decreases: If there is a problem, a new software version merely has to be generated (roll forward, see section 7.3). This new version automatically goes into production after successfully passing the Continuous Delivery pipeline. In contrast to the frequently used hot fixes which often pass by the official process, here the usual process is fully complied with. Since each change is small, such a fix is relatively quickly available in production. Therefore, special treatment of hot fixes is not necessary anymore in this model. In addition, it might be possible to save on the options for a rollback: When each release is sufficiently small, it is possible just to deliver a new release with a fix in case of a problem rather than rolling back. This especially holds true when additional measures, such as canary releasing, are used to safeguard quality.

With continuous deployment the responsibility of the team increases: Each change ends up in production. This can have positive effects for quality: The team has to be very sure that a new release in fact does not cause any problems in production. Therefore, the team will invest in the deployment pipeline and integrate safeguards to avoid errors in production as far as possible. This leads to a very clean approach. In addition, the software architecture will be optimized in such a way that whenever possible only small parts have to be exchanged upon each release, which will further improve the quality. However, this also means that an architecture that is not optimized for Continuous Delivery can render this approach practically impossible.

Since it is easier to bring changes into production, this approach also allows for greater flexibility. A new feature can rapidly be brought into

production; subsequently, how users react to the change and how the feature behaves in production can be evaluated. Finally, the feature can be quickly changed or turned off with the next release.

7.6.2 Disadvantages

To responsibly implement this approach a highly optimized Continuous Delivery pipeline is essential. This is a significant investment. In addition, the software architecture has to be appropriately optimized. Therefore, this approach can only be realized with little effort if the project has been designed from the start to support this strategy. Of course, continuous deployment can only be implemented if the quality of the Continuous Delivery pipeline is sufficient. Otherwise the approach can easily lead to profound problems.

In addition, the approach relies on trust: In principle, every developer can just bring code into production. If there is a manual testing phase, at least a manual clearance is necessary. Omitting this step can be problematic in some environments with restrictive compliance requirements. Often releases have to be performed by people who were not responsible for the code modification or who even work in a separate organizational unit. However, this requirement can in the end be combined with continuous deployment by integrating the “four eyes” principles into the pipeline. This will decrease the responsibility of the developers, since they will feel that there is still a test which should find potential errors. To deploy directly in production means for the developers that they themselves have to answer for the quality, and this feeling of responsibility can lead to higher quality.

7.7 Virtualization

Of course, it has to be possible to install servers in an automated manner in the production environment. To do so new virtual machines have to be started on a server, and software has to be installed on them. Examples for technologies that facilitate this process are:

VMware[1] offers comprehensive commercial solutions for operating even large server infrastructures. Many enterprises have been employing these technologies, which have proven their worth in practice, for a long time. In addition to server virtualization with ESX, vSphere, for managing complex infrastructures, is in the center of the VMware product line-up. Tools for

disaster recovery or virtual networks extend the array of products. Also, VMware offers ready-to-use solutions for a private Cloud.

OpenNebula[2] is an open source solution with similar options. Among others it offers the same APIs as the Amazon Cloud so that resources in your own data center can be supplemented with resources from a public Cloud. OpenNebula is developed by OpenNebula Systems and aims to provide a simple-to-use environment for enterprises.

OpenStack[3] is also an open source solution. Many enterprise vendors support the further development of OpenStack. These vendors offer their own distributions of OpenStack. Consequently, this technology is used relatively widely. OpenStack consists of many individual services for the administration of storage, computing resources, and networks. Due to the many services the installation can be a challenge. Like OpenNebula, OpenStack can deal with different virtualization hypervisors like Xen, KVM, or ESX.

Additional solutions in the Cloud area are Eucalyptus[4] and CloudStack.[5]

At the moment solutions are also created that are specialized for Docker (see section 2.5) and can directly execute Docker containers. The other already described virtualization solutions also have rudiments of support for Docker directly.

Thus virtual machines are available in production. As already described in Chapter 2, "Providing Infrastructure," there are different technologies for the installation of the application on these virtual machines. Ideally, upon starting a VM the software is also immediately installed on the virtual environment. How to set up such an environment is not described in this book in detail. The book just introduces possible technology stacks for automating the installation of an application.

7.7.1 Physical Hosts

In principle the presented approaches can be applied to production environments that use physical hosts. In the end for most technologies it does not matter whether the application is supposed to be installed or monitored on virtual or physical hosts. In addition, in production the flexibility offered by virtualization is not such a decisive advantage anymore: While it is very helpful when building up test environments—for instance, when several environments can be installed with different releases

or for different tasks—in production there is typically only one environment. However, approaches like canary releasing, blue/green deployment, and continuous deployment are much easier to implement with virtual machines.

7.8 Beyond Web Applications

The approaches that have been presented so far require that software be rolled out on servers. This has the benefit that the servers are under the control of the organization that also develops the software. This renders a direct deployment very simple. However, there is also software that cannot be operated in this manner. It is typically handed over to the client and is then installed by the client. In this case some of the approaches described in this chapter are difficult to employ. One example is mobile apps that are only sold through an online store.

In such scenarios the following approaches are conceivable:

- Deployment on the client's computer can be automated—for instance via a suitable update server. This allows in principle the rollout of new features at any time. App stores for mobile applications already offer this option. However, it can be implemented as well for desktop or server applications. In this case any release can be rolled out to clients. To avoid annoying users, care should be taken not to roll out too many updates. Further, the update process has to be as easy and trouble-free as possible. Of course, this requires that the process has been thoroughly tested.
- It is also possible to implement a variant of canary releasing: Most applications have power users who work intensively with the application and like to give feedback. These users can more frequently be provided with new releases—for example via an automated channel—and can be asked for feedback. This allows one to achieve a high release frequency with at least some of the users. These users benefit by being able to direct the development of the product and by benefiting earlier from new features.

Try and Experiment

Choose a project you are familiar with.

- Is it possible to roll back deployments?
- Are the necessary processes automated?
- Are the processes tested?
- Has there already been a case where a rollback was successfully performed?
- Is a rollback a realistic way to fix problems with a release?
- Would it be possible to employ blue/green deployment?
- How can the necessary infrastructure be provided?
- How can you deal with databases and third-party systems upon switching to a new release?
- Would canary releasing be an option?
- How do you deal with databases and third-party systems? They have to support both software versions.
- Under which conditions is Continuous Deployment conceivable?
- Does the architecture need to be modified? See also Chapter 11.
- Which improvements of the Continuous Delivery pipeline are necessary?

7.9 Conclusion

With Continuous Delivery the process for bringing software into production is technically just another execution of the same process that was already used for setting up the different test environments. But since the risk in production is greater, at least a rollback to a stable version of the application has to be possible. However, this option is often difficult to implement. Therefore, canary releasing or blue/green deployment is frequently the better alternative. Continuous deployment is the most radical approach: Each change enters production largely in automated fashion. These approaches can also be implemented with software that has to be installed locally on the client's computers. However, there are some hurdles to overcome in this case and, if necessary, the processes have to be performed

less frequently so as not to force the client to perform updates too often. This objective competes with the wish for frequent releases as a means of risk minimization.

Endnotes

1. http://www.vmware.com/
2. https://opennebula.org/
3. https://www.openstack.org/
4. http://www8.hp.com/us/en/cloud/helion-eucalyptus.html
5. http://cloudstack.apache.org/

Chapter 8. Operations

8.1 Introduction

The focus of this chapter is how applications can successfully run in production. Section 8.2 discusses the challenges of operations. Log files are an important source of feedback from production. Section 8.3 shows how log files can be used to record and analyze information about the application. Section 8.4 demonstrates practically, with the help of the example application, how log files can be analyzed. As a tool for this, the ELK stack is used—ELK stands for Elasticsearch, Logstash, and Kibana. Elasticsearch stores the information from the log files, Logstash collects and parses the information, and Kibana provides the surface for the analysis. A suitable example environment for your own experiments is also provided. This environment is generated in a fully automated manner with the help of Docker and Vagrant. Of course, there are also alternatives to this technology stack (section 8.5). Advanced log techniques are the topic of section 8.6.

In addition to logging, monitoring is also a way to obtain feedback from operations (section 8.7)—here the focus is on recording numerical metrics. Section 8.8 demonstrates how metrics can be visualized with Graphite. Again with the help of the example application, section 8.9 shows concretely how monitoring works. In this case an installation of Graphite based on Docker and Vagrant is provided for you to experiment with. Alternative monitoring solutions are introduced in section 8.10. In conclusion, section 8.11 describes some additional challenges associated with operating an application.

8.1.1 Operate—An Example

In section P.2 you got to know Big Money Online Commerce Inc. and learned that a problem occurred during a release that was brought into production: The registration of new customers had an outage. Initially, nobody noticed this outage although it of course directly and strongly influenced the success of the application on the market. The measures described in this chapter serve to obtain a better view into the application during production and thereby to avoid such situations.

8.2 Challenges in Operations

For Continuous Delivery it is not only the installation of the software in production which is important. At least as important is the feedback from production. In principle, applications should function in production without any errors. However, to be able to still intervene in case of unavoidable errors and to understand how the application can be improved, data from production is essential. Therefore, the application has to be monitored. Here, different aspects have to be dealt with:

- The infrastructure—servers, network components, and other hardware—has to be provided with monitoring. This is necessary to identify technical problems like outages or an overload of technical components, and to trigger an alarm if necessary so that the problems get fixed. This area is the focus of classical operations. Continuous Delivery has little effect on classical monitoring so this aspect is not discussed further here.
- The application can also deliver data about technical problems and errors. This allows a response to specific errors of an application. This aspect can be covered by special tools, or the application can be integrated into the existing infrastructure for operations. The application has to deliver the relevant information in such a manner that the tools can display and interpret it.
- Finally, the application can also supply data with regards to what is happening from a business perspective. The tools used for technical problems can also process such information and thus offer interesting insights into the events around the application. However, it might also be sensible to use specialized tools—for example, to analyze which links users click.

In the end, monitoring during production serves not only to record and process information concerning technical problems of the application, but also to obtain business data. In this area in particular, Continuous Delivery and the associated tools are very interesting. Therefore, this area will be the focus of this chapter.

In general it is not new to export business data from an application to analyze it from the business perspective. In addition to monitoring, data can for instance also be exported into data warehouses for analysis. These are relational databases specialized for large data volumes that are able to create

statistics about this data. There are also suitable front-ends that support creating and analyzing statistics. Big Data is a similar approach: This is an umbrella term for relatively new technologies that likewise can handle large data volumes, such as NoSQL databases. They are not based on the relational model and thereby achieve price-efficient scalability: Instead of buying large servers many cheap servers of smaller dimensions can be used. The technical term for this is horizontal scalability. Section 8.3 introduces Elasticsearch for saving and analyzing log files. While Elasticsearch is a search engine, it scales horizontally like NoSQL databases and also saves the raw data in addition to search indices. In these respects it is very similar to a NoSQL database.

However, the strategies behind these tools differ from the ones used by the tools presented in this chapter. They are mostly based on an export of data from the database. For this purpose extract transform load tools (ETL tools) are used. The data is extracted from the application and transformed into a data structure that is suitable for the subsequent analysis. In this process data might be aggregated—maybe the sales volume of an individual customer is not important for the statistics, but the sums for certain regions or products per day are. Finally, this information is loaded into a database for analysis.

There are numerous alternatives to the tools for monitoring and logging presented in this chapter that are especially interesting for business data and their analysis. However, this chapter introduces tools that record data from the perspective of operations, but can also be used for business data.

8.3 Log Files

Log files are a very simple and at first glance unimpressive way to record information about applications. The information is written into a file and provided with additional information such as a time stamp. This approach has a number of advantages:

- It is very simple to write log files. Writing log files can be implemented with every technology. For most languages there is already a large choice of libraries that facilitate the handling of log files.
- The data are persistent without having to go to great effort.
- Since information only has to be added to the log files, it is very quick to write them, in line with the fact that adding is the most efficient

operation on files.

- Even large amounts of information can be processed, for example by log file rotation: If a certain size threshold for the log file has been surpassed, a new file is created into which log information is written. After some time the data can be compressed or deleted.
- It is easy to analyze and search the files. Tools like Grep filter relevant information from the files. Scripts can be written rapidly to extract the essential information and thereby analyze the application step-by-step.
- The log information can very easily be read and interpreted by humans —in fact, with very simple tools. Therefore, it can be analyzed with little effort. However, this also means that the information from the log files is interpreted by a human. Thus, it should also be as easy to understand as possible. When a problem causes the production environment to fail at night, the person taking care of it cannot be expected to have the necessary time and concentration to unravel complex messages and relationships.
- Log files are independent of a concrete technology. When the application writes logs, the information in these files can be analyzed with many different technologies—there is no dependence on a certain monitoring solution.

8.3.1 What Should Be Logged?

Many different types of information can be written into a log file. However, not all of them are equally important. Therefore, in general logging systems offer different levels for the saved information. This allows one to filter certain messages so that they do not end up in a log file. In this way, the data volume and the influence on performance can be minimized in a production system, while on a test system all necessary information is immediately available to identify and remove problems.

A rough discrimination is *Error* for problems, *Info* for information and *Debug* for detailed messages to developers. If the log system is configured to write information marked “Info” into the log file, the data with higher prioritization—for example, “Error”—is also written into the file.

There is the following rule of thumb for the use of log levels:

- If an error occurs, this always has to be recorded in a log file. Ideally, the entry must comprise all information needed to analyze the error situation in detail. Errors should be saved with a log level like *Error*. This log level should be reserved for real errors—in the case of a system that functions normally no entries of this type should end up in a log file. This ensures that a log entry of the Error type attracts sufficient attention. If events with the level Error are constantly logged when there are no real errors, nobody will notice if a real error is recorded and thus the error will not be rapidly fixed. What's more, in every environment information with this level has to be saved in a log file so that errors do not go unnoticed. The handling of the logging of errors is very critical. Only when every error is logged, and sufficient information regarding the error scenario is saved, is there a chance to spot errors in production and to remove them quickly. In the end it is not possible to use a debugger or other tools for analyzing errors in production—and even if it were possible, the error first would have to be reproduced.
- Log files can also be exploited to generate statistics during the normal operation of the application. This can be done via entries with a log level like *Info*, which only contain information about successfully executed business processes. Production systems have to write this information into a log file when such information is supposed to be evaluated later on.
- Finally, detailed information with a log level like *Debug* can be written into the log file to facilitate the identification of errors. This log level should only be active during an error investigation and ideally only in that part of the system that is investigated at the time. Consequently, this log level is typically deactivated in production systems.

In this manner log files can very easily be used for error investigations and monitoring.

It is also possible to supply each log message with an unambiguous code. This makes it easier to track where the log message is coming from. Besides, additional information—for instance, measures for fixing errors—can be saved to each code in handbooks or other documentation, such as a Wiki, or certain employees can be informed. Suitable tools can also automate such measures.

However, when a large number of servers is in use, it is not that simple anymore to evaluate log files since the information is distributed across different servers. Still they have to be accessible in a coordinated manner. In addition, simple tools for the analysis of log files reach their limit upon a certain data volume. At some point it will not be sufficient anymore to just read in the data from start to end and thereby to search for information the way tools like Grep do.

8.3.2 Tools for Processing Log Files

Luckily there are tools that are designed especially for such challenges. In essence, there are three different tasks they have to deal with:

- The data has to be collected from the different log files, which can be stored on different servers within the network.
- The files also have to be parsed: Log files contain information like the name of the server, the log level, or the writing component. When the data are supposed to be searched, a search for a specific field should be possible in addition to a full text search.
- The suitably processed log information has to be saved in such a manner that searches can be performed efficiently. It also has to be possible to store and search large data volumes.
- Finally, there has to be a way to analyze the data and thereby to identify errors or to analyze customer behavior.

ELK: Elasticsearch, Logstash, Kibana

Elasticsearch, Logstash, and Kibana can serve as concrete tools.

- Logstash[1] allows one to parse log files and to collect them from servers within the network. Logstash is a very powerful tool. It can read data from an input, modify or filter the data, and finally write it to an output. In addition to reading log files and saving them in Elasticsearch, other inputs and outputs are supported. For instance, data from message queues or databases can be read or written. Finally, the data can also be parsed or supplemented—for example a time stamp can be added to each log entry, or individual fields can be cut out and processed further.

- Elasticsearch[2] stores the data and provides them for analysis. Elasticsearch can not only search the data with a full text search, but it can also search within structured data and store the data permanently like a database. Finally, Elasticsearch offers statistics functionalities and thereby can analyze the data. As a search engine Elasticsearch is optimized for short response times so that the data can be analyzed nearly interactively.
- Kibana[3] is a web application for analyzing data from Elasticsearch. In addition to simple queries, statistical analyses are also possible.

The combined tools are also known as the ELK stack (Elasticsearch, Logstash, Kibana) as shown in Figure 8.1.

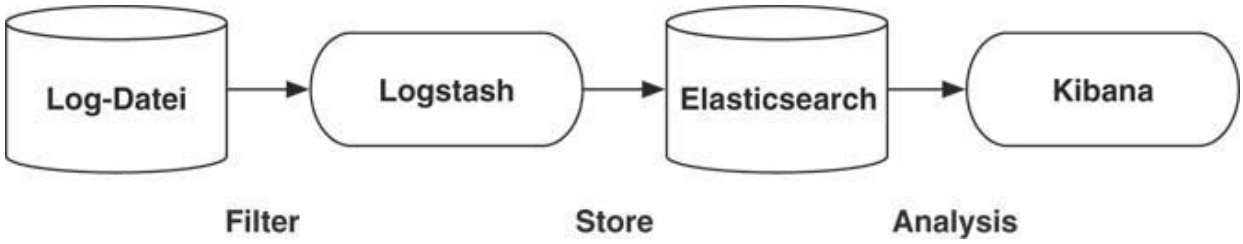

Figure 8.1 *Processing logs with the ELK stack*

8.3.3 Logging in the Example Application

First, the code of the example application has to be modified in such a way that the essential information in fact ends up in a log file. For Java there are different log libraries—the example application uses Apache Commons Logging[4] as the API. Behind the scenes Logback[5] writes the files.

The application writes two different types of information into the log files:

- In an error situation the essential information about the problem has to be recorded with the log level Error in the log file. This does not require any modifications in the application: When an error occurs, an exception is thrown—either automatically or triggered by written code. The infrastructure automatically takes care that exceptions are logged. This does not hold true for exceptions which are successfully caught and handled—however, such exceptions do not represent real errors, but a normal sequence of events in the program. The exception is caught and therefore the error is handled.

- For all business processes information with the log level Info are written. One example is the successful registration of a user. The most important corner cases have to be recorded as well—for example when the email address of a user is invalid or has already been used for a registration. With these log entries statistics can be generated regarding the number of registrations. In addition it is possible to check later whether a user attempted to register and for which reasons this did not work out. This allows one to scrutinize errors more closely. For example, in response to questions by the support team, what actually happened can be investigated in more detail.
- It is also possible to integrate log messages into the code with a level such as Debug. However, this only makes sense when specific errors are searched for. Since this is not the case for the example application, its code remains without such outputs.

With this information, errors in the application can be analyzed. In addition it is possible to evaluate, with the help of the log files, which business processes have been executed and how often. In addition, some solutions for analyzing log files can automatically extract key/value pairs. When for instance the string `firstname="Eberhard"` in the log output appears, this is parsed and interpreted. The variable `firstname` with the value `Eberhard` is added to the log entry. This makes it very easy to filter all entries that concern a user with a certain first name.

8.4 Analyzing Logs of the Example Application

In the case of the example application the log files can also be analyzed. This is done by a system of Docker containers, in each of which a part of the infrastructure is installed.

Figure 8.2 shows how exactly this happens:

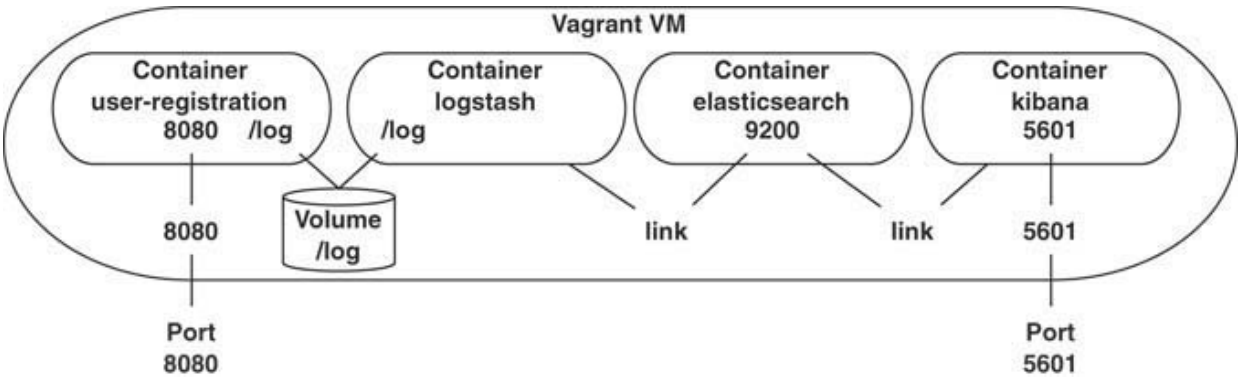

Figure 8.2 *Log setup for the example application*

- The application is installed in the container `user-registration`. The web UI is available at port 8080, which is mapped to port 8080 of the Vagrant VM and port 8080 of the host. Consequently, the application can be found at the URL http://localhost:8080/. The application writes the log files into the directory `/log`. This directory maps to a data volume.
- This volume is likewise mounted by the container `logstash`. The container parses files that are stored in this data volume and saves the results in Elasticsearch. To do so the container `logstash` uses a link to the container `elasticsearch` so that communication between the two containers is possible via a shared port.
- The container `kibana` offers a web server with a Node.js application to deliver HTML and JavaScript to the browser. This web server runs in the container under port 5601, and this very port is mapped to the Vagrant VM and the host. Accessing the data that is stored in the Elasticsearch server occurs via a link between the containers.

Thus, this setup uses individual Docker containers as components that communicate via ports or shared file systems, as described in Chapter 3, "Build Automation and Continuous Integration."

Listing 8.1 *Logstash configuration*

Click here to view code image

```
input {
  file {
    path => ["/log/spring.log"]
    start_position => beginning
```

```
  }
}

filter {
  multiline {
    pattern => "((^\s*)[a-z\$\.A-Z\.]*Exception.+)|((^\s*)at .+)"
    what => "previous"
  }
  grok {
    match => [ "message",
      "^(?<date>[0-9] \-[0-9] \-[0-9] )
       %{TIME:time}
       (?:\s*)
       (?<level>[A-Z]+)
       %{NUMBER:pid}
       (?:\-\-\-)
       (?<thread>\[.*\])
       (?<class>[0-9a-z\$A-Z\[\]/\.]*)
       (?:\s*:\s)
       (?<logmessage>(.|\s)+)"]
  }
  kv {}
}

output {
  elasticsearch {
    hosts => ["elasticsearch"]
  }
}
```

In this system the configuration of Logstash (Listing 8.1) is especially interesting. Logstash uses a pipeline: Data is read from an input, filtered, and finally written to an output. The `input` defines that Logstash is supposed to read the log file under the indicated path starting from the beginning of the file. The `multiline` entry in the section `filter` takes care that entries running across multiple lines are summarized into a single message. This is necessary for Java exceptions: They contain information on exactly where an error occurred distributed across multiple lines. Therefore, the entry contains a regular expression that matches lines starting with the name of a Java exception or the string “at.” The `grok` filter cuts the log message from the field `message` into different fields that each have to correspond to a regular term.

Figure 8.3 shows exactly how a log line is processed: First the date is saved in the field `date` and the time in the field `time`. Afterwards the log level is stored in `level`, the process ID in `pid`, the name of the current thread

in `thread` and the Java class that generated the log message in `class`. Finally, there is the `logmessage`—the actual information from the log message.

Figure 8.3 *Example for handling a log message*

The last filter is `kv`—an abbreviation for key value. When the log message contains terms like email=eberhard.wolff@gmail.com (see Figure 8.3), a field with the name `email` and the value `eberhard.wolff@gmail.com` is created. This greatly facilitates the search for log messages for specific customers.

Finally, `output` defines in the Logstash configuration that the data are supposed to be exported into Elasticsearch, specifically into the index `logs`. Logstash generates from the fields a JSON document since this format is supported by Elasticsearch. Elasticsearch can search for arbitrary values for arbitrary fields and can also generate statistics.

The host name `elasticsearch` is defined for the Elasticsearch server. The Docker link ensures that the Docker container with the Elasticsearch installation can be accessed under this name.

8.4.1 Analyses with Kibana

With the help of Kibana users can analyze the collected data from Elasticsearch directly in the browser. To do so there are different tabs in the Kibana user interface. As a starting point the Discover tab is very useful. There all fields are displayed so that users can first have a look at all messages and interpret them. Afterwards, the users can determine the distribution of the values, limit the output to selected fields, or filter the output in such a way that only log messages with certain values are displayed. Figure 8.4 shows a screenshot. Here, you can see the distribution of the different log levels.

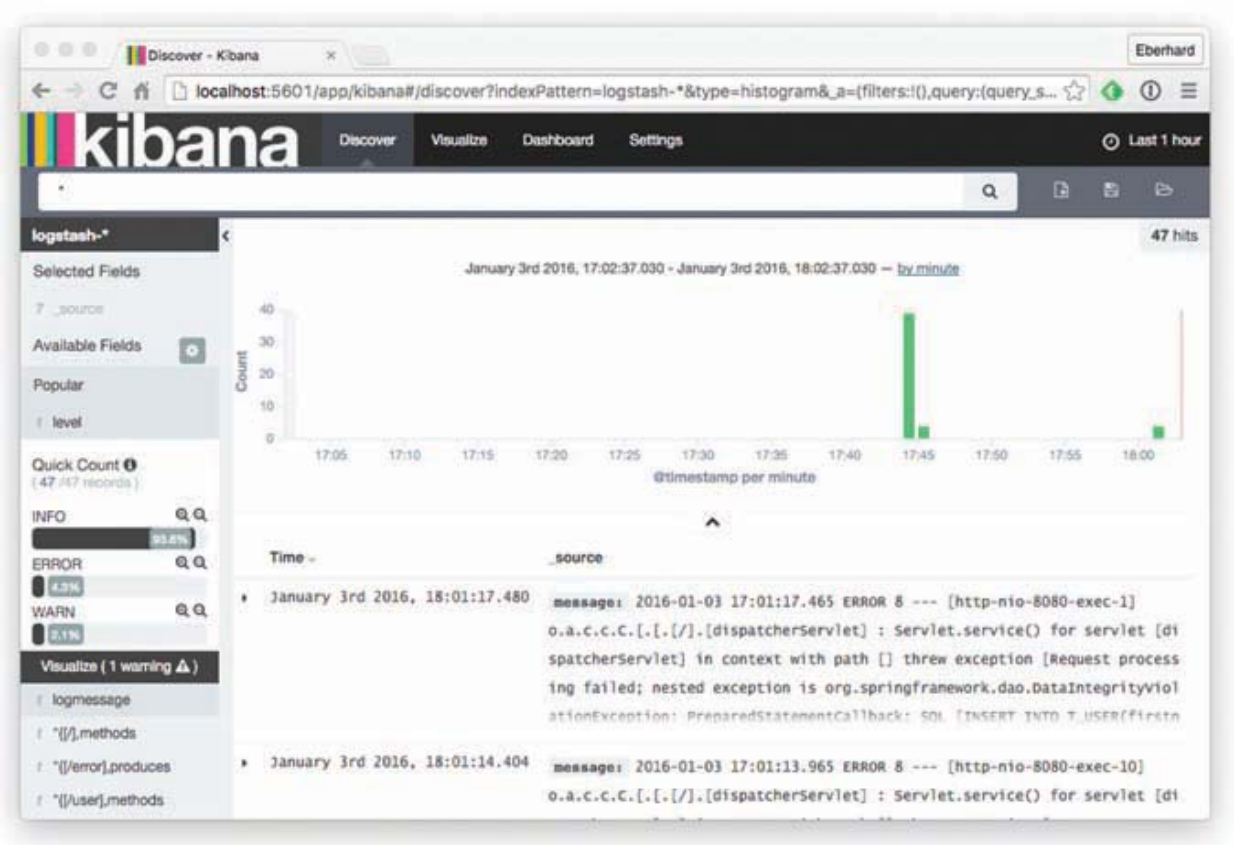

Figure 8.4 *Analyzing data with Kibana*

In addition, graphical analyses can be performed—for instance, based on the domain of the registered email addresses (see Figure 8.5). Elasticsearch and Kibana can do much more than just search data—for example, they can also generate statistics. Elasticsearch is optimized for rapid response times so that analyses can be performed very quickly.

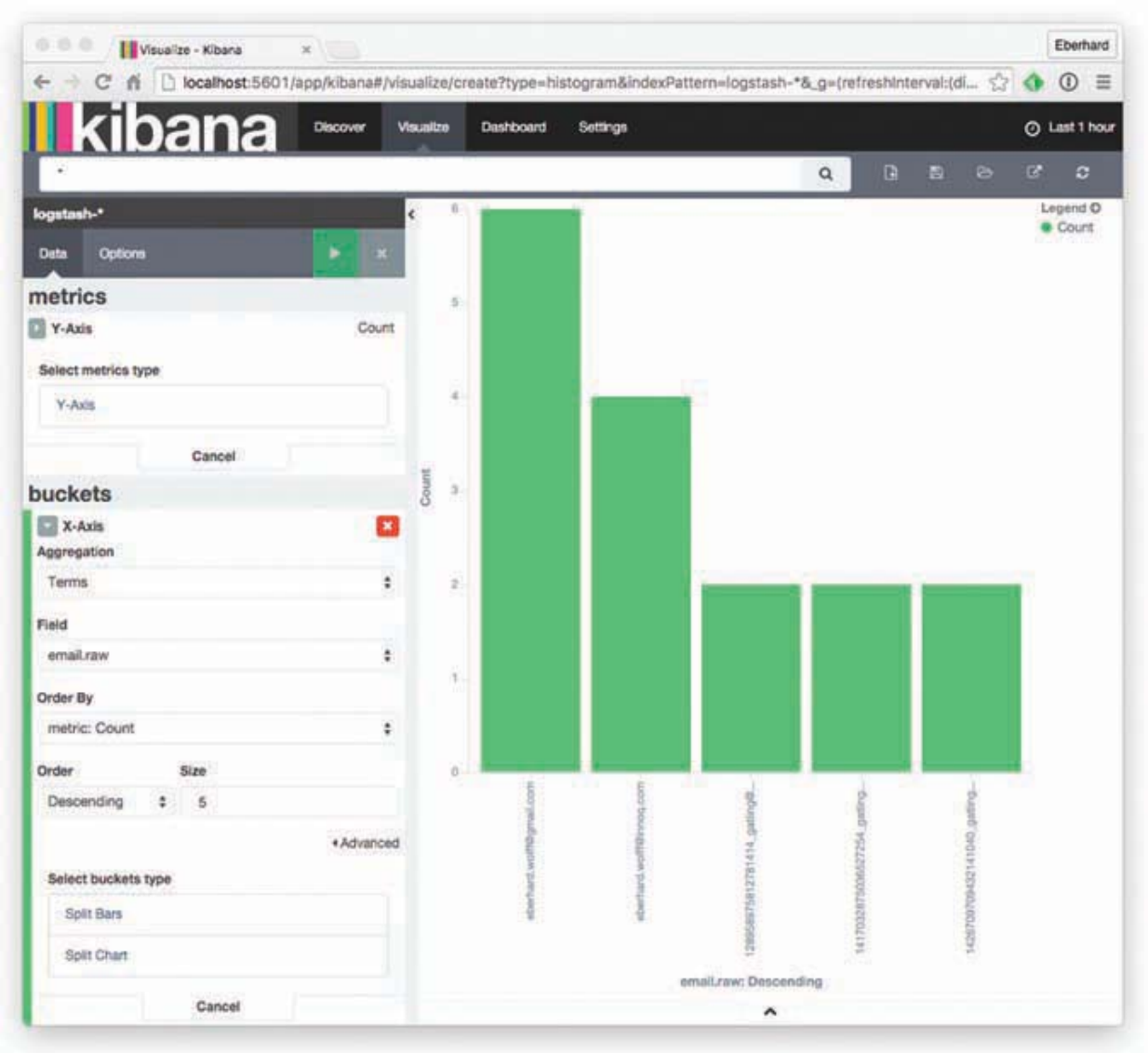

Figure 8.5 *Graphical analysis with Kibana*

8.4.2 ELK—Scalability

For the example application the accumulating data volumes are easily manageable. However, one of the strengths of the ELK stack is its easy scalability:

- Elasticsearch can divide indices into shards. Each entry is allocated to a shard. Since the shards can be stored on different servers, this allows distribution of the load. Shards can also be replicated across multiple servers to ensure fail-safeness. In addition, reads can target an arbitrary

replicate of the data. Accordingly, the reads can also be scaled with replicates.

- In the absence of additional configurations Logstash writes the data of each day in another index. Since the current data are typically read more often, this can reduce the data volume that has to be searched for typical requests and thus improve performance. And there are still other possibilities to distribute the data to indices—for example, according to the origin of the users.
- With the presented configuration Logstash has to parse each line of the log. It would be more efficient if the application sent the log information as a JSON document because the parsing could be completely avoided. In the case of large data volumes this leads to significantly less load for Logstash.
- Finally, Logstash can be scaled. In the presented configuration Logstash takes over different functions—collecting data from the server and parsing. For scaling, these functions are separated and executed by different technologies. The data is read by a shipper, buffered in a broker, and processed with Logstash. This allows the installation of Logstash on a server other than the actual application. The server is therefore unburdened from the CPU's intensive filtering and processing of log information. The broker serves as a buffer in case so many log messages accumulate that they cannot be processed immediately. Redis[6] —a fast in-memory database—is often used as the broker. Logstash can serve as the shipper. It will simply take over the logs from a local file and write them into the broker. This is possible since Logstash can connect numerous different data sources and data sinks. The Logstash Book[7] discusses in detail how to set up complex Logstash installations and how to scale Logstash.
- Other options are Filebeat,[8] Beaver,[9] and Woodchuck,[10] which represent special shippers. Usually Filebeat is preferred because it is supported by Elastic, the company behind the ELK stack. In addition, it is possible to use standard UNIX tools like syslog.

Try and Experiment

Clone the example project (https://github.com/ewolff/user-registration-V2). With the Git command line tool (http://git-scm.com/) the necessary command is `git clone https://github.com/ewolff/user-registration-V2.git`

- An environment for the analysis of log files with the ELK stack can be found in the sub-directory `log-analysis`.
- Install Vagrant (http://www.vagrantup.com/downloads.html).
- With `vagrant up` you can start the environment.
- The following URLs are available:
  - http://localhost:8080/ for the application
  - http://localhost:5601/ for Kibana
- Execute some interactions with the application. Alternatively, modify the load or acceptance test in such a manner that it communicates with the application and that it does not start the application on its own. Then use the load test to create load for the application.
  - Hint: When you enter a very long name (>30 characters), an error will occur in the application.
- Open the Kibana homepage under http://localhost:5601/.
- There you have to configure the index.
  - Accept the default logstash-* as “Index name or pattern.”
  - Time-field name” should be “@timestamp.”
- Have a look at the analysis in the discovery tab of Kibana.
  - In the upper right corner you can set the period of time for which log messages are displayed by the user interface. The default value is 15 min—if necessary, modify this value.
  - On the left you can select a field. Then all the values of the field will be displayed.
- Click on the magnifying glass next to one of the values, so you can activate a filter.
- Search for certain entries. To do so you only have to enter a search term into the entry line on the top of the website.

- Search for errors (level is ERROR). To find errors you first have to create some with the help of suitable entries (for instance with first names with > 30 characters).

More options can be found at https://www.elastic.co/guide/en/kibana/4.3/getting-started.html. For example, you can create a graphical analysis in the Visualize tab or a dashboard in the Dashboard tab.

- At the moment the setup writes the log data into a file and Logstash parses them and sends them to Elasticsearch. To facilitate the work for Logstash the data can be delivered directly as JSON.
- Information for changing the log configuration can be found at https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-logging.html.
- To do so a file `logback-spring.xml` has to be created in the application within the directory `src/main/resource`. For more information see https://github.com/spring-projects/spring-boot/tree/master/spring-boot-samples/spring-boot-sample-logback/.
- Afterwards, a JSON encoder can be used; see https://github.com/logstash/logstash-logback-encoder. The data is still supposed to be written in the file; therefore only an encoder entry for the `LogstashAccessEncoder` is required—see https://github.com/logstash/logstash-logback-encoder#encoder.
- Finally, the Logstash configuration has to be modified so that it does not parse the data anymore.
- In the end the log data is only required for the analysis in the ELK stack. To save the data into a file as well does not make much sense. Change the setup so that the data is sent directly via the network. To do so you can extend the JSON setup in such a manner that Logstash receives the information directly via the network per TCP; see also https://github.com/logstash/logstash-logback-encoder#tcp.
- Modify the setup so that the information from the application is written by Logstash into the broker Redis and from there filtered and analyzed. This task is laborious and requires, among other things, the installation of additional Docker containers.

- Create a new image and a new container with a Redis installation. Hint: There is an Ubuntu package called `redis-server`.
- Modify the Logstash installation in such a manner that it only reads in the log file and transfers it to Redis.
- Generate a new Logstash installation that reads data from Redis, parses the data, and writes it into Elasticsearch. This requires a new Docker container.
- Use Docker Compose instead of Vagrant for starting the ELK environment. A suitable configuration can be found in the file `docker-compose.yml`. Hints as to how to use Docker Compose can be found in section 2.5.7.

8.5 Other Technologies for Logs

Of course, the ELK stack is not the only option for analyzing log files. Graylog[11] is also an open source solution and likewise uses Elasticsearch for saving the log files. In addition, it uses MongoDB for metadata. However, Graylog defines its own format for the log messages: GELF (Graylog Extended Log Format), which also standardizes the transferred data. Extensions for GELF are available for many log libraries and programming languages. In addition, the respective information can be extracted from different log data or collected with the Unix tool syslog. Logstash also supports GELF as an input and output format. Graylog2 has a web interface with which the analysis of the information from the logs is possible.

Splunk[12] is a commercial solution and has been on the market for a long time. It can be extended by numerous plug-ins. There are also apps that immediately deliver a turn-key solution for certain infrastructures like Microsoft Windows Server. The software does not necessarily have to be installed in your own data center; it is also available as a Cloud solution. Splunk aims at being a solution that can not only analyze log files, but can serve in general for the analysis of machine data. In line with this there is a version of Splunk called Splunk Hunk that can be integrated into the Hadoop Big Data platform. For processing logs Splunk can first record the data with a forwarder and then provide them with an indexer for searching. Search heads then take over the processing of search requests. The aim to be an

enterprise solution is underlined by the security concept. Analyses as well as alerts in case of certain problems are possible.

To avoid the installation of an infrastructure for the analysis of log files it is also possible to use a Cloud offering. Such solutions offer turn-key analysis and data storage. In that case neither scaling nor installation is an issue. However, the logs have to be sent to the cloud, and, of course, this requires bandwidth. Furthermore, such a solution is just unsuitable for some applications—for instance, when there are data security issues.

Splunk Cloud as a possible solution has already been introduced. Alternatives are:

- Loggly[13] is rather simple so that logs can be analyzed without much effort.
- Sumo Logic[14] on the other hand has numerous features. Among others it is able to recognize anomalies independently. With its security features it is meant more for use in the enterprise area.
- Papertrail[15] in contrast is very simple—it mainly serves to consolidate and analyze multiple logs. The user interface is text-based. Papertrail does not focus on complex applications, but searching is of course possible.

Most offerings can be tested free of charge so that the way is free for experimenting. The example application can serve as a basis for experiments.

Some of the monitoring solutions mentioned in section 8.10 can deal with log files. This is especially true for commercial solutions.

8.6 Advanced Log Techniques

In the area of logging there are numerous tools and approaches available. The example application uses a Java library to write into a log file. A usual approach, at least with Linux, is to use a service like syslog-ng[16] as a central logging system that can receive information from different sources and write them into log files. This allows the use of a common logging infrastructure for all applications.

8.6.1 Anonymization

Another problem when dealing with logging, especially with regards to production, is data security. Production data often contains personal data that must be protected and therefore cannot simply be passed on to developers. This data has to be anonymized before it goes on to analysis. For this purpose Logstash offers a filter called `anonymize`. This makes use of a hash. Hashes are unambiguous—that is, whenever a given hash value appears in the log for a user name, it always corresponds to the same user name. Yet it is not possible to determine the user name from the hash value.

8.6.2 Performance

Most systems use a multitude of servers and different applications to implement the necessary functionalities. To analyze the performance of such a system, the connected processing steps on the different servers have to be examined. To do so an unambiguous string can be used for each request; this string is passed on in each log message. When the log messages are collected on a central server and all requests with the same value are searched, this allows one to check which system components have used up how much time for processing the request. There are specialized solutions like Zipkin[17] that not only support the collection of data, but also the extraction and display of data.

8.6.3 Time

During logging it is often a problem that the clocks on the different computers are not synchronized. This makes it difficult to exactly analyze relationships. Logstash provides each entry with a time stamp. Therefore the synchronization problem can be circumvented when at least the clocks on the servers on which Logstash runs are synchronized.

8.6.4 Ops Database

Step-by-step more and more data end up via logging in a central storage like Elasticsearch. This allows one to collect configuration settings, performance data, or start time, stop time, and statistics of batch processes. Of course, the data from monitoring can also be collected and saved. In the end this leads to an Ops database that contains all information regarding the application. The data from this database can, for instance, be used to plan capacities or to provide hints for optimization—for example, error-prone areas or performance bottlenecks can be identified. Thus this database can be an

important source for the feedback that Continuous Delivery especially relies on as a basis for software optimization.

8.7 Monitoring

In addition to the analysis of log files the metrics of an application are of interest. By means of numerical values the metrics shed light on the current state of the application and how it changes over time. This allows the identification of problems early on—for example, a decrease in the free storage space of the hard drive. An alarm can be triggered and the problem resolved before it becomes critical—for example, by deleting files. Therefore, metrics are a fundamental part of monitoring of systems and applications. Every operations team will have installed a solution to cover this area.

The monitoring has to take place via an external process. Only an external process can determine whether the production process is down and can still trigger an alarm in such a situation. In addition, this is the only way to ensure that the monitoring is also available during times when the production process runs under high load. In such situations monitoring within the production process would no longer react.

In the context of Continuous Delivery, monitoring and metrics are as important as the analysis of log files. They provide feedback from the application and thereby can influence the further development of the application. For example, these technologies allow one to collect not only technical values but also data about sales revenue, and to align them with technical metrics. When sales revenue suddenly collapses, this can be caused by a problem within the application. Such correlations then become obvious. Therefore the focus is not only on technical data but also on data that have business importance. In the end the software is supposed to support business processes. It is only logical to measure the application with regards to which business values it achieves. If the system, for instance, functions at the technical level, but does not lead to any sales volume, this should stand out in the monitoring and should also trigger an alarm—in the end this is a total failure of the system from the business perspective.

8.8 Metrics with Graphite

Most IT environments already have a monitoring system into which all servers are integrated and that is able to inform administrators about problems, even at night. Therefore, new applications mostly just have to integrate into an existing monitoring system. To collect some experience and impressions with a monitoring tool this section will introduce Graphite.[18] Graphite is a tool for handling metrics. It represents a complete solution:

- Graphite can store numerical data. The limitation on this type of data is acceptable: Typical monitoring values like utilization or response times can be represented numerically.
- Graphite is specialized for the processing of time series since production data frequently are only interesting with regards to their development over time.
- The data can be displayed by a web application so that the analysis is quite simple for users.

Graphite is composed of three components:

- *Carbon* receives data and buffers it in an internal cache.
- *Whisper* is a simple data library for time series. It only contains data from a certain time interval since at some point old data become irrelevant for monitoring.
- Finally, the *Graphite web app* allows one to access the collected data.

Figure 8.6 shows an example of the user interface of the Graphite web app. In addition to response times, the number of requests to certain URLs of the example application is displayed. The application can thereby not only be monitored from a classical operations perspective, but its behavior can be evaluated.

Figure 8.6 *Graphite dashboard*

In the dashboard two different data types can be distinguished:

- *Gauges* are numerical values that change over time. Thus average values can be determined. This is shown in the access times on the dashboard in Figure 8.6.
- *Counters* can be increased or decreased. In the dashboard this is used for representing the accesses to the URLs, for deleting, and for displaying a user.

Reporting values to Graphite is very simple: Only the value, the name of the metric, and the time stamp have to be handed over via a network socket.

In production Graphite can also be supplemented by other tools:

- StatsD[19] can collect values and can pass them on to Graphite. This allows one to reduce the data volume before data is handed over to Graphite to unburden Graphite itself.
- Grafana[20] extends Graphite by alternative dashboards and other graphical elements.
- collectd[21] collects statistics about a system, like the CPU utilization. These data can be analyzed with specific front-ends or stored in Graphite.
- Seyren[22] extends Graphite by a functionality for triggering alarms.

For monitoring, especially, there is a multitude of solutions. This list can only provide a first impression. The sensible use of monitoring is also important from a business perspective. Thus, when, for instance, an A/B test is meant to determine which of two alternatives is the better one, the decisive metrics should first be defined and recorded.

8.9 Metrics in the Example Application

In the example application the metrics are recorded with Metrics.[23] This is a Java library for recording metrics. Without additional code modifications HTTP requests are measured first of all. This allows one to record how often a certain URL is called and how long this takes. Self-defined metrics can of course also be recorded. Metrics has an extension for the support of Graphite so that the values can be directly passed on to Graphite, where the user can evaluate them.

8.9.1 Structure of the Example

For Graphite there is likewise an example installation in three Docker containers. In the process some files of the Docker installation are overwritten. Figure 8.7 shows the setup in an overview. The example application runs in its own container. The application can be accessed via port 8083.

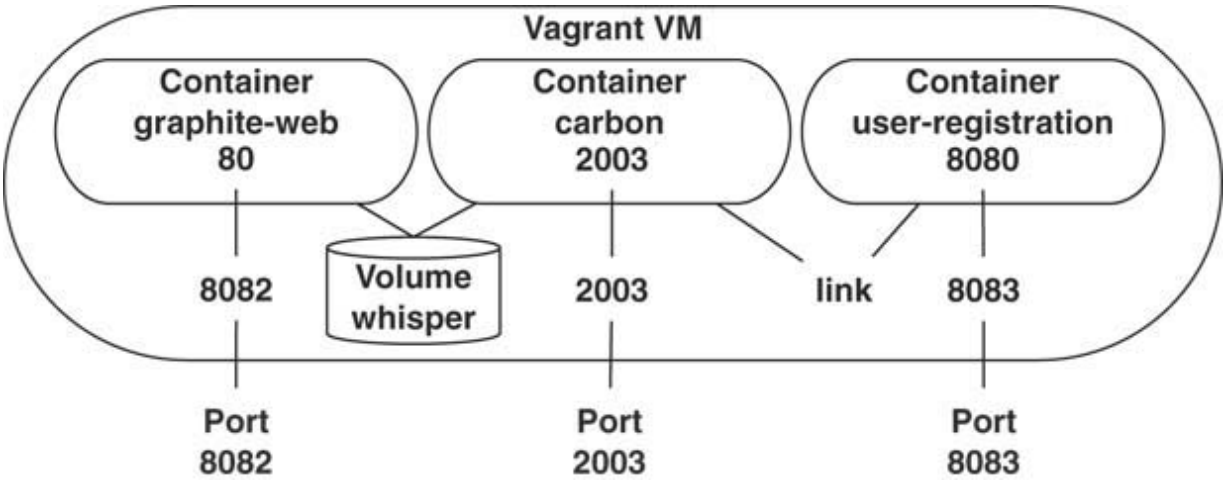

Figure 8.7 *Graphite setup*

Carbon accepts the monitoring data and runs in another container. To transfer data to Carbon, port 2003 is used. This port is provided and exported to the outside via a link between the containers of the application. Thereby the host can also hand over data to Graphite. Thus, the Graphite installation can very easily be reused for other contexts. Carbon can also be scaled—the data storage can be distributed to multiple servers so that even very large data volumes can be handled. Internally Carbon uses Whisper for saving data, and writes them onto a volume.

The Graphite web app is available on the host via the URL http://localhost:8082/. It reads from a data volume, into which the Carbon container writes the data, and provides them for analysis.

Try and Experiment

Have a look at a project you are familiar with:

- Which tool is used for monitoring?
- Which information does the application pass on to monitoring?
- Is only technical or also business information recorded?
- When functions of central importance for the business fail, as for instance the charging of customer accounts, would monitoring identify the problem? For example, when would it be noted that the system is not achieving any sales volume anymore?
- Clone the example project (https://github.com/ewolff/user-registration-V2). With the Git command line tool (http://git-scm.com/) the required command is: `git clone https://github.com/ewolff/user-registration-V2.git`
- An environment for monitoring with Graphite can be found in the sub-directory `graphite`.
- Install Vagrant (http://www.vagrantup.com/downloads.html).
- With `vagrant up` you can start the environment.
- The following URLs are available:
  - http://localhost:8083/ for the application
  - http://localhost:8082/ for Graphite
- Execute some interactions with the application. Alternatively, modify the load or acceptance test in such a manner that it communicates with the application and does not start the application on its own. Then use the load test to set the application under load.
- Generate a dashboard in Graphite like the one depicted in Figure 8.6.
- Find out how the links and the access to the ports are implemented in the Docker configuration.
- The installation of Graphite has to modify some files of the Graphite installation. How is this implemented? Which benefits and disadvantages does the approach have in comparison to the approach of the templates of Chef?
- Supplement the setup by another component, like StatsD.[24]

- Which additional information can be read from the example project? Is the number of registrations a meaningful information?
- Modify the code in such a way that the number of registrations is saved in Graphite. Since the application uses Spring Boot, this can be done with little effort.
- Use Docker Compose instead of Vagrant for starting the Graphite environment. A suitable configuration can be found in the file `docker-compose.yml`. Hints on how to use Docker Compose are given in section 2.5.7.

8.10 Other Monitoring Solutions

Graphite is not the only solution for monitoring applications and IT environments:

- Nagios[25] is a comprehensive solution for monitoring and can be an alternative to Graphite.
- Incinga[26] originally was a fork of Nagios and has become a rewrite. However, it still covers a very similar context.
- There are also different commercial solutions such as HP Operations Manager,[27] IBM Tivoli,[28] CA Opscenter[29] and BMC Remedy.[30] These tools are very comprehensive, have been on the market for a very long time, and offer support for numerous software and hardware products. Such platforms are often introduced enterprise-wide—and such introductions are in fact very complex projects. Applications have to integrate into the solution in order to comply with enterprise standards. Some of these solutions can also analyze and monitor log files.
- Riemann[31] is an open source tool. It is based on the processing of event streams. To do so it uses functional programming logic to define logic for the response to certain events. A dashboard can also be configured for this, or messages can be sent via SMS or email.
- The monitoring can also be transferred into the Cloud, so it is not necessary to install a comprehensive infrastructure. This facilitates the introduction of tools and the monitoring of applications. An example is NewRelic.[32]

- With TICK stack[33] a complete open source stack for monitoring is available—it is based on InfluxDB. This time services database is, like Whisper, specialized for saving time-dependent data and can be combined, for instance, with Grafana or Graphite. Additional tools that belong to the stack are: Telegraf for collecting data, Chronograf for visualization, and Kapacitor for alerts and the automated search for anomalies.
- Packetbeat[34] uses the ELK stack Elasticsearch for saving data and Kibana for analysis. This makes it relatively easy to combine it with the ELK stack. Packetbeat monitors the network traffic, but can also check the content of network packets and thereby provide information regarding, for instance, specific SQL requests. This approach is especially helpful for distributed applications since it facilitates obtaining an overview and a precise analysis of the entire system.

8.11 Additional Challenges When Operating an Application

When operating an application, not only logging and monitoring are of interest. An additional point is the auditing of changes. For Continuous Delivery this topic is especially important since changes to the production environment are ideally only possible by changing scripts. With version control the changes can be tracked—it is always clear who performed which change. Thus, the auditing of changes to the production environment can easily take place by means of version control.

In addition, it is no longer necessary to have direct access to the servers: Information from production can be obtained via log files and metrics. The deployment of new versions can take place via automated processes. This has different advantages: For instance there are benefits with regards to security since nobody—including hackers—can log into the system anymore. And in addition to people from the operations team, developers also can directly read out information from production, for instance via a system like the ELK stack. This speeds up the analysis and fixing of errors.

8.11.1 Scripts

While operating an application it can be necessary to change configurations or to interact in other ways with the application. Such operations should be automatable. When manual changes to the application are necessary, this

leads to a higher expenditure and potentially to errors. When the processes are automated, they can be exactly reproduced without much effort—this is exactly the aim of Continuous Delivery. However, to achieve this it has to be possible to automate such interventions by a script. Although an admin user interface for manual interventions might be useful, it is still better when the interventions that are possible are also available via a scriptable interface.

8.11.2 Applications in a Client's Data Center

The approaches that were presented here are optimized for the operation of in-house applications. However, in many cases applications are installed in, for instance, the data center of the client as part of a machine or on a mobile device. This makes the rollout of new releases and monitoring much more difficult. However, the principles introduced here can also be used in such cases:

- The installation of new versions should be automated. This is mostly already the case for mobile applications since app stores offer such functionalities. Software that is installed in the data center of a client should be kept up-to-date via similar mechanisms. Updates should be installed in an automated manner, so fewer versions have to be supported, errors can be fixed faster, and it is easier to react to security gaps.
- To continuously obtain log files from all clients and installations is unrealistic, due to data security among other things. Nevertheless it should be possible in case of an error to send all relevant information in an email or to make the information available via other means. This should largely be automated so that just a click is required. This way it can be ensured that the developers quickly get the decisive information.

In the end the objective remains the same: if possible to offer direct access to the necessary information and to deliver new software as easily and automatically as possible. Since nowadays the distribution of software mostly occurs via the internet and, therefore, access to the information from the production systems is also possible, similar approaches can be used for systems that are installed at the client site as for systems running in your own data center.

8.12 Conclusion

Technically the main focus when operating an application is to have access to the information from production and to analyze it. This chapter has shown two options for this. The application can write log files that can be analyzed by an infrastructure like the ELK stack. Another option is monitoring where primarily numerical values are evaluated—Graphite was presented as example. Thereby operations obtains the necessary information out of the application. At the same time developers and product managers can go on to develop the application further in a sensible way with the help of this data. Without this feedback the purposeful ongoing development of an application would hardly be possible.

Endnotes

1. https://www.elastic.co/products/logstash
2. https://www.elastic.co/products/elasticsearch
3. https://www.elastic.co/products/kibana
4. http://commons.apache.org/proper/commons-logging/
5. http://logback.qos.ch/
6. https://redis.io/
7. https://www.logstashbook.com/
8. https://www.elastic.co/products/beats/filebeat
9. https://github.com/python-beaver/python-beaver
10. https://github.com/danryan/woodchuck
11. https://www.graylog.org/
12. https://www.splunk.com/
13. https://www.loggly.com/
14. https://www.sumologic.com/
15. https://papertrailapp.com/
16. https://www.balabit.com/network-security/syslog-ng
17. https://github.com/openzipkin/zipkin
18. http://graphiteapp.org/
19. https://github.com/etsy/statsd/

20. http://grafana.org/
21. https://collectd.org/
22. https://github.com/scobal/seyren
23. http://metrics.codahale.com/ or https://github.com/dropwizard/metrics
24. https://github.com/etsy/statsd/
25. https://www.nagios.org/
26. https://www.icinga.com/
27. http://www8.hp.com/us/en/software-solutions/operations-manager-infrastructure-monitoring/
28. https://www.ibm.com/software/tivoli
29. http://www3.ca.com/au/opscenter.aspx
30. http://www.bmc.com/it-solutions/remedy-itsm.html
31. http://riemann.io/
32. https://newrelic.com/
33. https://www.influxdata.com/
34. https://www.elastic.co/products/beats/packetbeat

PART III

Management, Organization, and Architecture for Continuous Delivery

Continuous Delivery is not just a technology but reaches much further. This is the focus of the last chapters of the book:

- **Chapter 9** explains how Continuous Delivery can be introduced into an enterprise.
- **Chapter 10** shows the connection between the technical Continuous Delivery approach and DevOps—an organizational approach.
- Continuous Delivery and DevOps also influence software architecture. **Chapter 11** details this.
- **Chapter 12** finally provides a conclusion of the book.

Chapter 9. Introducing Continuous Delivery into Your Enterprise

9.1 Introduction

Introducing Continuous Delivery essentially means to build up a Continuous Delivery pipeline. Section 9.2 shows how this can be done right from the start for new projects. Introducing Continuous Delivery into an already existing project is mostly an optimization task since there is already a pipeline for bringing releases into production. The main question here is where and how to start with optimizing the pipeline. Section 9.3 describes Value Stream Mapping, a popular approach for introducing Continuous Delivery, while section 9.4 discusses other approaches like quality investments, Stop the Line, and the 5 Whys.

9.2 Continuous Delivery Right from the Start

Introducing Continuous Delivery is easiest when it is done right from the beginning of a new project. When the implementation of the Continuous Delivery pipeline is started right away, the pipeline can be further developed step-by-step along with the project. First, the work on certain phases can be started—for example, commit (see Chapter 3, "Build Automation and Continuous Integration") and deploy (Chapter 7, "Deploy—The Rollout in Production"), since at least these two phases have to exist to bring the application into production. Subsequently, other phases can be added step-by-step—for instance, acceptance tests or other automated tests. Depending on the non-functional requirements it may be possible to do without capacity or penetration tests. This allows one to gradually invest effort, which is in general easier to realize.

Besides, introducing Continuous Delivery at the start of a project has the benefit that Continuous Delivery can be kept in mind right from the beginning when choosing technologies and architectures. When a certain database or a certain technology for another component of the infrastructure is especially easy to automate, it can be used preferably. Likewise the complexity for configuring the application and setting up the infrastructure is kept in check right from the start, since otherwise automation will be

difficult to achieve. Consequently, the application is immediately optimized to possess a lower complexity regarding its installation and operation.

The choice of technology influences the Continuous Delivery pipeline in many regards: For example the selected programming language influences the speed of the compilers. This in turn affects the time used by the Continuous Delivery pipeline. Decisions regarding the programming language can also influence the quality of the Continuous Delivery pipeline.

In the end it will pay off to implement Continuous Delivery right from the start in order to set up and optimize the Continuous Delivery pipeline step-by-step. In addition, this allows one to align technology and architecture decisions with the demands of Continuous Delivery. This makes it easier to set up the pipeline. In summary, this means that introducing Continuous Delivery is easier for new projects.

9.3 Value Stream Mapping

Most projects built on an existing codebase were designed without taking Continuous Delivery into consideration. To introduce Continuous Delivery successfully in an existing project, it is best to start by thinking about the fundamentals of Continuous Delivery: Continuous Delivery is based on rapid feedback and Lean (see section 1.4.10). The objective of Lean is to reduce the amount of waste. Waste in this context means everything that does not create immediate value for the customer. An example is for instance code that is not in production. Although it might contain new features, these features cannot yet be used by the customer. Therefore, Continuous Delivery focuses not on waiting till a larger number of changes can be delivered together, but instead on rapidly delivering small changes. Features and code changes are meant to continuously flow, so to speak, through the pipeline, and deployments should be performed regularly. This corresponds to the ideal of Lean to have a constant flow of value through the system.

In a certain respect there is always a type of Continuous Delivery pipeline because in the end there is always a process to bring a software release into production. However, this process can be very complex or it can take a long time so that it is very far from resembling a continuous flow.

9.3.1 Value Stream Mapping Describes the Sequence of Events

To approach the ideal state of Continuous Delivery with its rapid feedback, Value Stream Mapping is helpful. It describes the current sequence of events up to the release. For the individual steps the processing time (Value Add Time) and the waiting time (Waste Time) are determined. The sum is the total time needed (Cycle Time). The efficiency of the process derives from the Value Add Time divided by the Cycle Time. Therefore the ideal can be approached by minimizing Waste Time and by implementing a constant flow of features through the pipeline. With regards to these times, what happens in case of errors or other problems? This should also be considered since the fixing of errors can cost a lot of time. Therefore, in the test phases, what happens when an error occurs during a test also has to be determined. In addition, contact partners and necessary resources for an optimization can be determined in the Value Stream. Between the steps queues result when there is a bottleneck in the process. A queue occurs when changes do not pass sufficiently fast through a certain phase. If, for instance, many code changes are waiting for manual testing, this indicates a bottleneck.

9.3.2 Optimizations

One possibility for optimizing the Value Stream is to introduce restrictions for queues. In this case it becomes obvious where bottlenecks are—namely in those places where the limits of the queues are reached. This allows one to identify the bottlenecks and to eliminate them. For instance, it could be defined that only a certain number of changes may wait for manual testing. When the queue is filled up, the preceding steps may not accept any more new tasks—in the most extreme case this leads to an interruption of the implementation of new features. This creates pressure to remove the bottleneck at the manual testing phase.

The determination of cycle times can also highlight optimization potential. When certain phases take a long time or long waiting times occur, it becomes obvious where the problems in the pipeline are currently located.

Figure 9.1 shows an example. Commit, build, and unit tests, as well as code analysis, run on a Continuous Integration server. The server processes these phases one after the other. There is hardly any waiting time—the Continuous Integration server starts almost immediately upon each commit by a developer. The passage through the phases is also relatively fast. However, the acceptance and capacity tests take very long so that the changes have to wait for some time before being able to enter these phases.

The reason is that these tests are performed manually, and only once per release—for example, only once every two weeks. Therefore a change has to wait on average five days before being able to run through these tests.

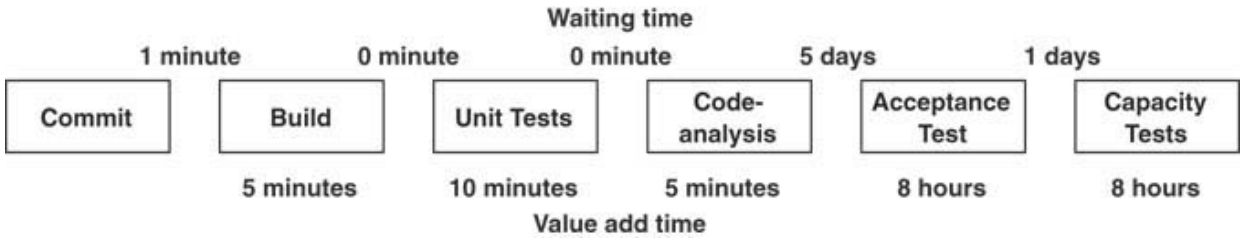

Figure 9.1 *Value Stream Mapping before optimization*

Value Stream Mapping is also sensible when the pipeline is not yet fully automated. It can be used to first identify optimization potential and derive suitable optimization measures before introducing Continuous Delivery.

Step-by-step the bottlenecks in the pipeline are removed. In Figure 9.2 the pipeline is already largely automated after several optimization steps—especially with regards to acceptance and capacity tests. They are performed on an extra system separate from the Continuous Integration server. However, waiting times still occur during these tests. To reduce the waiting times further the processing time for the tests can be decreased. When the processing time decreases, the waiting times will also get shorter since the necessary computers are more rapidly available again.

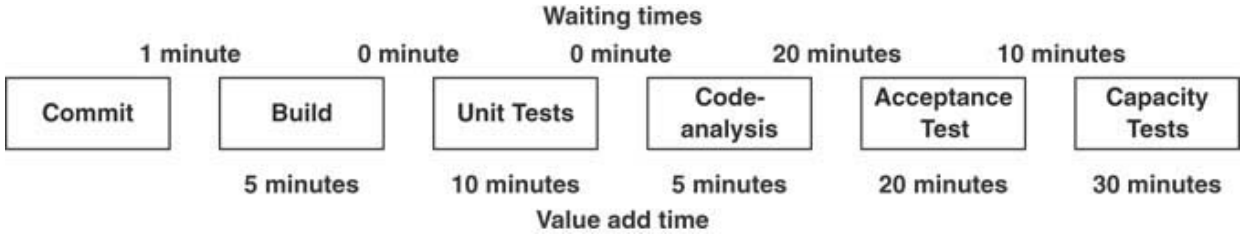

Figure 9.2 *Value Stream Mapping after optimization*

There are different approaches for optimizing the processing times of the acceptance and capacity tests, such as better hardware, test improvements, or providing multiple environments for the tests. In addition, the tests can be divided so that first a reduced test version, which tests rather broadly, runs, and only later does a more comprehensive test version, which evaluates in depth, run. This allows one to provide more rapid feedback to the developers when a feature is not working at all.

Therefore Value Stream Mapping can be a valuable tool for optimizing an existing pipeline step-by-step.

9.4 Additional Measures for Optimization

Value Stream Mapping provides indications where the Continuous Delivery pipeline can be optimized. However, in general, finding steps to be improved is not the main problem—it is deciding what and where to start optimizing.

9.4.1 Quality Investments

The metaphor of the quality investment[1] is helpful for coming to this decision. The central idea is to evaluate costs and benefits of every possible measure and to go for the one that provides the best trade-off between costs and benefits. The costs are the effort that is necessary to implement an optimization. With regards to the automation of tests this would comprise the effort of writing the tests and integrating them into the Continuous Delivery pipeline.

9.4.2 Costs

Evaluating costs is especially important when Continuous Delivery is supposed to be implemented into an already existing project, because the costs of certain measures can be extremely high when certain software characteristics have not been considered previously. Often, developers show little interest in how easy or difficult the installation of a software package is. It may happen that the developers allow for a very flexible configuration, which, however, creates high expenditure during operation and therefore profoundly raises costs in the end. Due to this complex configuration it becomes very laborious to arrive at an automated installation of the software. In the case of Continuous Delivery the software has to be installed in the different phases in order to be tested. Therefore, automated software installation is really desirable. However, the effort to achieve this may be so large that it is in the end still better to do without full automation.

9.4.3 Benefits

Costs are offset by benefits. With Continuous Delivery benefits fall into two different dimensions: reduction in effort and increase in reliability. For example, the automation of acceptance tests reduces the effort for running

these tests. At the same time the tests become more reliable since now results can be exactly reproduced due to automation.

In addition to this direct benefit there is also an indirect benefit: The test automation allows for more frequent testing. This increases the probability that errors are detected earlier and therefore raises the quality in the end. When the tests are performed more frequently, the developers more rapidly receive feedback if an error occurs. This facilitates the removal of the error because the developers will have introduced fewer code changes until the failure of the test. In addition, they remember their recent concrete code changes in more detail and therefore can more easily identify the error.

Thus, the determination of costs and benefits seems to be an easy task at first glance—after all, it is only an evaluation of the expenditure associated with implementing the measure and the productivity / reliability benefit that comes with it. However, a detailed analysis is very complex since there are direct and indirect benefits that are in fact hard to predict and judge.

In the end the most important thing is the mindset. No matter how beautiful a Continuous Delivery pipeline can become after observing certain measures, all that really matters is whether a change results in a benefit and whether the benefit obtained is in a reasonable relation to the previous expenditure. Measures that do not hold up according to such considerations should not be implemented.

This is exactly the idea behind quality investments. For each investment costs and benefits have to be clear. The concept of quality investments brings this relationship between costs and benefits into the center of the discussion.

9.4.4 Do not Check in on a Red Build!

A relatively easy way to improve the adaptation of Continuous Delivery is to follow a simple guideline:

> When the Continuous Delivery pipeline has failed, nobody may check in.

First, there are very practical reasons for this rule. When the system is currently not working, all additional changes will render the search for errors more complicated. Furthermore, errors in the new code will not immediately be detected since the Continuous Delivery pipeline is failing. This will cause errors that occur due to new code changes being overlooked.

This simple rule translates for the teams into the necessity of keeping the pipeline green as often as possible—otherwise it is not possible to bring code into production. Therefore, the teams will invest effort into improving the pipeline so that they will not be delayed by a red Continuous Delivery pipeline. How to improve the pipeline is up to the teams—alternatively, a person with a specific role can coordinate the activities for the removal of the current problem.

9.4.5 Stop the Line

Another option for optimizing the Continuous Delivery pipeline is "Stop the Line." This idea comes from industry: There every worker on an assembly line can stop the conveyor belt if a problem arises. The problem has to be solved before production can continue. This principle can also be employed in the context of Continuous Delivery: When a problem occurs in the Continuous Delivery pipeline, all team members immediately assemble, prioritize the problem, and determine how to deal with it. This leads to a rapid removal of the problem. The benefit of "Stop the Line" is that problems are immediately eliminated. They have a high priority and are definitively attended to. This allows them to ensure the quality of the pipeline. However, while the solution might solve the immediate problem, the underlying cause might still not have been identified and removed.

9.4.6 5 Whys

To deal with this the 5 Whys[2] can be used in the context of a post mortem. Here, the cause of a problem is determined by asking *Why* five times in a row in order to approach the underlying problem.

An example:

- *Why* is the build red?
- Due to the failed test.
- *Why* did the test fail?
- Because the test data for the test was not correct.
- *Why* was the test data not correct?
- Because an error occurred during the import of the test data.
- *Why* did this error happen?

- Because the test data was generated manually.
- *Why* was the test data generated manually?
- Because there is no automation.

Therefore, in this example the measure to be taken would be to automate the generation of test data. In that case in the future the test data will be provided identically for each run of the pipeline. Thereby the underlying cause of the original problem is eliminated.

In the context of "Stop the Line" the focus is different: Here, the main point is to remove the problem as fast as possible. To generate the test data from now on in an automated fashion can hardly be implemented without some time investment. Therefore, in the context of "Stop the Line" the solution is rather to correct the test data without addressing the underlying problem. This switches the pipeline quickly to green again. However, it has the drawback that the problem might arise again and again during future runs.

9.4.7 DevOps

When introducing Continuous Delivery, operations and development should work together. Each of these groups masters a part of Continuous Delivery particularly well. Operations is familiar with aspects like monitoring, security, and network infrastructures, which are essential for installing and running the application. The developers on the other hand know the code, the development infrastructure, and the middleware like the application server very well. When both work together a Continuous Delivery pipeline can quite easily be set up. Therefore the introduction of DevOps should at least be considered when starting with Continuous Delivery (see Chapter 10, "Continuous Delivery and DevOps").

Try and Experiment

- Sketch the current release process in a project you know with the help of a Value Stream Map (see Figures 9.1 and 9.2).
- For how long does a code change remain in which step?
- How many modifications wait in which step of the Value Stream?
- If this information is not known, how could it be determined?
- Derive some optimization measures from this knowledge. There is always room for improvement....
- Sketch two or three measures for optimizing the Continuous Delivery pipeline including a cost/benefit analysis in line with the concept of quality investments. Which measures would you implement first?
- Define a team capable of implementing these optimizations.
- Which skills are necessary?
- Can developers or operations implement the measure on their own or is their cooperation essential?
- Do some research into the 5 Whys method. Try to apply it to the last production failure you were confronted with.

9.5 Conclusion

Continuous Delivery can and, if possible, should be used right from the start of a project. This allows building up the necessary pipeline step-by-step. For projects that have been running for some time there is already a pipeline to bring code changes into production. This pipeline then "merely" has to be optimized for Continuous Delivery. Different approaches can be employed for this:

- Value Stream Mapping helps to analyze the pipeline with regards to time and throughput. This information can be used to devise optimizations.
- Quality investments represent a mind set for evaluating such optimizations.
- "Stop the Line" serves as security net for removing problems that occur during the run through the pipeline.

- To eliminate problems thoroughly for the future, the 5 Whys method can be used.

In practice, most of the time a combination of these techniques will be used for setting up and optimizing the Continuous Delivery pipeline.

Endnotes

1. https://www.infoq.com/articles/no-more-technical-debt
2. https://en.wikipedia.org/wiki/5_Whys

Chapter 10. Continuous Delivery and DevOps

10.1 Introduction

Continuous Delivery is primarily a technical method. In this chapter the effects of Continuous Delivery on the organization are discussed. Section 10.2 describes DevOps: This is an organizational form which is optimally suited for implementing Continuous Delivery. DevOps centers on the collaboration of operations (Ops) and development (Dev). Section 10.3 shows concrete possibilities for the interplay of DevOps and Continuous Delivery. Section 10.4 asks the question whether Continuous Delivery without DevOps is even possible. In conclusion, section 10.5 highlights additional organizational approaches that extend DevOps.

10.2 What Is DevOps?

Classically, development and operations are separate entities within an organization—in general, they are two different departments that only meet at a very high level of the IT management hierarchy. They also have different objectives: Operations is supposed to work in a very cost-efficient way, and its performance is evaluated based on the costs. The development team implements new features and is evaluated based on how rapidly and efficiently it can deliver these features. The reason for the sharp separation is the idea of division of labor and the associated higher efficiency that is usually achieved by specialization, standardization, and industrialization. When operations, for instance, only has to support one type of platform it can specialize in this platform and run it particularly efficiently by means of automation.

10.2.1 Problems

However, this approach can also create problems. Operations will aim for achieving a high degree of software stability and might view each change and every new release as a potential danger to stability. The development team on the other hand in essence is paid for changing the software. This can easily lead to a confrontation between the two departments. Meanwhile the common goal of both departments can easily get buried: They are

supposed to offer an optimal service for the clients, regardless of whether these clients are within their own enterprise or actual customers in the marketplace.

Also, in day-to-day business the differences are significant: Development and operations have entirely divergent perspectives regarding the application. Operations views the application as an operating system process and can monitor it with suitable tools. In addition to CPU utilization and I/O load, the behavior with regards to the kernel can also be evaluated. With such tools, operations can analyze the behavior of the applications as well as errors that arise. In doing so the application often remains a black box whose internal processes are not known. Developers on the other hand know the business processes and work with, for instance, exceptions and log files. Operations should also acquire this knowledge. Frequently, operations does not know certain infrastructures like Java Virtual Machine and features like Garbage Collection at all although they are often highly relevant for the analysis of problems. Developers on the other hand are often not familiar with typical tools and approaches used by operations since they typically only administer and install their own computers or testing servers, on which these tools are not available or are not able to fully express their impact. Besides, these systems are hardly comparable to production systems.

Thus, since operations and development both possess only a part of the necessary knowledge and the required tools, in practice a sensible operation of applications is only possible with the combined know-how of operations and development. This holds even more true for deployment where the application has to find its way from operations to development—this is exactly the area Continuous Delivery deals with.

10.2.2 The Client Perspective

For clients the strong division into operations and development is also not helpful: When there is an error or a problem, operations or development can contribute to its solution depending on the underlying cause. However, for the client it is often hard to figure out which department can help in which way to fix the problem. In the worst case operations refers to development as the one who has to remove the problem while the developers will say that it is the responsibility of operations to deal with the problem. The client has a hard time deciding who is really responsible in the end.

10.2.3 Pioneer: Amazon

The separation of operations and development is basically common to all large IT organizations. However, in 2006 it became known that one of the really big IT companies does it differently: Amazon. Since 2006 Amazon has had teams that handle development as well as operations. Each team is responsible for a certain service with some business-relevant logic. Therefore, each team has the freedom to optimize the service on its own in such a manner that it can be well operated as well as easily developed further. For example, without any further coordination it is possible to introduce enhancements for the monitoring of the application. In addition, the team can also decide independently which technology stack it wants to use—in the end it is its own responsibility to run the technologies and to remove potential errors and problems. Therefore, enterprise-wide standards are largely avoided with the exceptions that all applications have to be able to run on the Amazon Cloud infrastructure. This seems to be somewhat chaotic but it really offers a lot of freedom and provides a foundation for self-organization.

There is only a small part of operations left which works independently of the individual teams and is responsible for providing and maintaining the hardware and the Amazon Cloud. On the virtual machines the different teams can then install the operating system and all the other software.

10.2.4 DevOps

The term DevOps was coined in 2009 by the Devopsdays conference in Belgium. It is composed of development (Dev) and operations (Ops). The term indicates the focus of the concept: Development and operations are meant to grow together into one team, split into sub-teams based on components and domain-based responsibilities.

For clients, such an IT organization is more transparent: When there is a problem with a certain service, it is immediately clear who is the relevant contact person and can therefore fix the problem. In addition, it is clear who is able to extend a certain service by additional functionalities.

Figure 10.1 *Classic IT organization with a split into Dev and Ops*

Therefore, DevOps focuses on changing the organizational model (Figure 10.1). Separate units like operations and development—often called silos—are dissolved to implement an integrated approach where each team takes over responsibility for both areas.

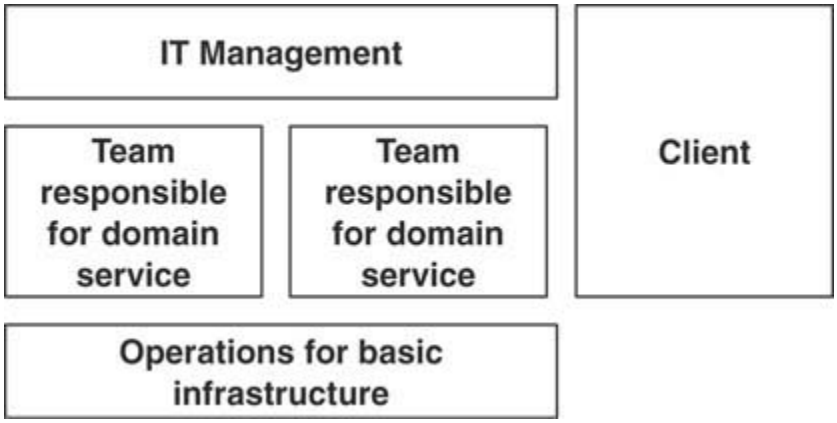

Figure 10.2 *DevOps teams*

Thus, when introducing DevOps it does not make much sense in addition to the existing operations and development teams to establish a new DevOps team that is somehow supposed to moderate the cooperation between development and operations. This only creates an additional silo since now there is a third team in addition to operations and development. This runs counter to the aim of reducing the number of silos.

It makes much more sense to start by first running and developing one service with some business logic according to the DevOps concept. This service is then taken care of by one team, which deals with its development and operation in parallel. This would constitute a first DevOps seed from which to go on to change the organization (Figure 10.2).

In practice it is often difficult to change an entire organization, especially in large enterprises. However, a collaboration can also be achieved differently: In the end there is always informal collaboration going on in

addition to the formal organizational structure. This trend can be encouraged—for instance by moving the actual workplaces of the employees from operations into one room, with those from development. This makes it much easier for operations and development to communicate with each other and therefore facilitates their collaboration. In addition, employees from operations can join the development team for an internship and the other way round. This is also a means to strengthen the exchange of knowledge and the collaboration without formally changing the organizational structure.

In the end DevOps represents a different concept of development and operations. Both units have to work closely together for IT to be able to create the optimal value for the client. So although a change in the organization is not necessarily required, it is certainly helpful.

10.3 Continuous Delivery and DevOps

Continuous Delivery is especially easy to implement in the context of DevOps since it requires the know-how of both departments: Development knows the application and knows exactly how the application has to be configured, how the application is structured internally, and which metrics are especially interesting for monitoring from the perspective of the application logic. Operations on the other hand knows the surrounding conditions and can, for instance, provide tools for monitoring or log analysis. By means of a DevOps team both areas can be covered at once during software development and operation. Furthermore, the feedback from production can be used directly to optimize the further development of the application, since all necessary roles are united in one team. This allows the team to reach the main goal of Continuous Delivery, namely fast feedback.

10.3.1 DevOps: More Than Continuous Delivery

Often Continuous Delivery is even seen as synonymous with DevOps. However, this misses the point. While Continuous Delivery is greatly facilitated by DevOps and represents a fundamental method in the DevOps area, there are many more areas than just Continuous Delivery where DevOps is a beneficial organizational approach:

- For example, development and operations can work together for monitoring. Operations can monitor processes and the entire server at the level of the operating system. Development can draw conclusions based on monitoring and optimize the application and increase its reliability accordingly. In addition, development can offer additional metrics that operations can include in the monitoring. This allows the team to learn more about the behavior of the application. This monitoring can also comprise business metrics. When a new release leads to, for example, a decrease in the volume of sales, this is immediately visible so that the problem can quickly be fixed.
- In addition, operations and development can work together for troubleshooting. Although operations is strong in troubleshooting and problem analysis, it is frequently limited to tools for system analysis like tcpdump or strace. The team only looks at the software from the outside. Development knows the internal structure of the application and from this perspective can offer substantially more information for troubleshooting. Therefore, the software development can be extended in such a way that it returns additional information for troubleshooting, or the development team can even provide specialized tools for the analysis of applications. Examples are profiling and tracing across the layers of the application. The implementation can be very simple—for example, in production with an additional parameter in the URL that makes the application display additional data. Moreover, additional controls for operations can be provided—for example the deactivation of parts of the application.
- Finally, the development team can provide operations with the possibility of an intervention into the application. Short of restarting the application, operations normally has almost no ability to intervene. However, it can be desirable to be able to deactivate certain functionalities—for instance when another system that is needed for the functionality is not available at the time or has to be serviced. This can help to avoid a crash of the entire application. In addition, an administration tool for modifying data can be useful for repairing errors in data sets. The final result could be an administration tool for the application that allows an administrator to remove problems occurring in the application.

Thus, it is quite obvious that DevOps is a lot more than just Continuous Delivery. It is a special mind set and an organizational form that can result in numerous technical measures of which Continuous Delivery is just one.

10.3.2 Individual Responsibility and Self-Organization

In essence, DevOps is about individual responsibility. The teams take over the entire responsibility for a component—for its development as well as its operation. Thus the teams have much less need to coordinate their work with other teams. Everything associated with the specific component can be worked on by a single team. This comprises the development, the operation, and especially the rollout into production. This allows the teams to work faster, to develop software more rapidly, and to bring it into production quickly. For instance it is no longer necessary to write a comprehensive operations manual since operation and development of the component are working closely anyway. Therefore, written communication can in many places be replaced by direct communication. In addition, due to automation far fewer manual processes, which would have to be documented, are necessary.

10.3.3 Technology Decisions

Moreover, this approach allows the teams to make many more decisions on their own. They can decide to use a new technology—without any influence from management. However, this also means that the team takes over the responsibility for this technology even in production. If the chosen technology causes problems in production, the team has to find ways to deal with these problems. That is why the team can take over the responsibility for the decision in the first place. Consequently, it is the team that has to organize on-call duties. Therefore, it is in the interest of the entire team—development and operations—to avoid problems that might result in calls in the middle of the night.

10.3.4 Less Central Control

Central control is less important in such an environment. Of course, certain general rules have to be established and these should also be controlled. However, these general rules should concern fundamental characteristics: Each team will have a Continuous Delivery pipeline and therefore will use an automated deployment of the components in different environments.

This encompasses components that can be deployed and tested in an automated fashion. However, the decision on which concrete technologies to use for this can be delegated to the individual teams. The choice of technologies for implementation, such as frameworks, programming languages, or application servers can be entirely left to the teams since they have the responsibility for development and operation. When they decide on the technology that best fits their requirements, there is no reason not to accept this decision. In the end it is also the team and not an independent operations department that has to solve the problem if the technology does not work in production. Likewise, the team has to deal with the problem if the technology causes high expenditure during the implementation.

10.3.5 Technology Pluralism

This allows one to use many different technologies within one organization. A classical organization will aim at limiting the number of technologies and controlling their use in order to minimize risk and to realize synergy potentials. When all teams use the same programming languages and infrastructure, each developer will know the required technologies and therefore can support every project to a certain extent. Operations can also focus on these technologies and thus become very familiar with the type of problems occurring specifically with this technology.

Such synergies are not so much in the focus of DevOps. The main point is here the freedom of the teams to make their own decisions. Thus every team can choose the technology stack best suited for the respective demands. This allows every team to work with optimal productivity.

10.3.6 Exchange Between Teams

To share best practices and self-developed tools or frameworks with other teams the exchange between teams can be strengthened. Up to a point it is also possible to establish a standard technology stack. However, the compulsion to use this technology stack is replaced by the benefits such an established technology stack can offer: Other teams have already gained experience with those technologies and thus can provide support. Besides, there is already a self-developed tool set. This renders a technology used by some teams attractive for other teams as well. In the end it is quite likely that in spite of the freedom of choice certain technologies get established across the individual teams.

The total freedom of choice can also be restricted in some regards: When there is only a single person in a team who masters a certain technology, problems can arise when this person is on vacation or leaves the company. To counteract such problems guidelines can be established which only allow the use of a certain technology when there are multiple people within a team who know how to use this technology. Of course, this problem exists also at the level of the team: When a team employs certain technologies that nobody else in the organization is familiar with it has to be ensured that this team will exist in the long term. Finally, it can be difficult to shift people from one team to another to provide extra support in times of high work load since it might be that members of one team do not know the technology stack of the other.

10.3.7 Architecture

For the architecture teams, a high degree of independence poses a challenge: A central instance that forces the architecture into a certain direction is at odds with the independence of the teams. It has to be replaced by processes that allow the teams to develop the system further in a coordinated manner. This process can arise spontaneously, but experience teaches that usually it at least has to be moderated—meetings have to be defined where the teams can coordinate their work. It can also be necessary that a central architect derives the relevant architecture topics—for instance from the teams' plans for further development—and designs a process for coordination between the teams. However, it is important that the teams decide about the all-encompassing topics and that the central instance only moderates this process. In the end the teams have to live with the consequences of the decisions and the architecture and should be able to develop software efficiently.

Thus, DevOps influences processes, teams and approaches far beyond Continuous Delivery. At the same time DevOps can be used to realize many additional benefits. Continuous Delivery can certainly be implemented without the organizational changes associated with DevOps, but for a complete Continuous Delivery pipeline abilities from operations are well as from development are crucial. Therefore, a Continuous Delivery pipeline cannot be completely implemented without a certain level of cooperation and coordination.

Try and Experiment

- If you are working as developer in a classical organization: Get into contact with a colleague from operations who runs your application.
- How does the monitoring work?
- What does the colleague typically do with the application?
- Which tools does she use for this?
- Which features could help her with troubleshooting?
- Which incidents occurred and how often was operations called at night? For which reasons?
- What are the top ten errors in the log files?
- What are the largest problems during installation and operation of the application?
- If you are working as part of the operations team in a classical organization: Identify a developer of one of your applications and discuss with him the points above.
- Choose an application as an example of the monitoring.
- Which values of the application can currently be monitored?
- In which categories do they fall—for instance, technical (DB, VM), business (sales volume, new registrations)?
- For which values are alarms defined?

10.4 Continuous Delivery without DevOps?

As discussed in the previous sections Continuous Delivery makes most sense together with DevOps since the Continuous Delivery pipeline requires skills from development (Dev) and operations (Ops) in the different phases. On the other hand, for most organizations DevOps is only possible when the existing structures are dramatically changed. Especially in large enterprises the areas of development and operations are frequently separated directly under the CIO. To really introduce DevOps the organizations would have to be fundamentally changed since the teams then would have to comprise employees from development as well as from

operations. However, such a fundamental restructuring of the organization is laborious and often causes substantial opposition.

Therefore, the question is whether Continuous Delivery can also be introduced without DevOps. For Continuous Delivery mixed teams are not mandatory, but collaborative work on the Continuous Delivery pipeline is necessary. In concrete terms operations should work on automating the deployment, and the developers should focus on implementing the different test phases. Reciprocal support going beyond that is of course also conceivable and helpful. Such a joined work on the pipeline does not necessarily require restructuring the organization.

However, organizational boundaries are often reflected in the pipeline: For example, development can set up test systems and establish an automation for them. However, when operations is not willing to take over the automation, but wants to set up its own automation, the pipeline breaks into two parts. The systems and the automation in development will be structured differently from those in operations. This will make it difficult to reproduce problems in production on the systems from development. This causes double expenditures since changes have to be tracked in both systems. This even holds true when development and operations use the same tools and operations want to take over the changes from development manually. The two automation approaches will develop apart and a later unification will cost a lot of effort. This technical problem is caused by an organizational problem: Operations does not trust the changes introduced by development and therefore does not want to adopt them. However, this problem can also be solved without restructuring the organization, simply by better collaboration. When the initial mistrust is removed, a technical solution will be found. On the other hand no technical solution will be able to remove an underlying psychological problem.

10.4.1 Terminating the Continuous Delivery Pipeline

If there is no support from operations, the question is whether Continuous Delivery can be implemented at all. In the end the objective of Continuous Delivery is to bring software more rapidly into production. This is hardly possible if operations does not support the Continuous Delivery efforts and therefore the pipeline cannot go into production. However, it would still be possible to implement the Continuous Delivery pipeline with its different

test phases and to terminate it prior to production. Of course, it is not possible to achieve a faster deployment in production this way.

However, time-to-market is not the only objective of Continuous Delivery. Another aim is reproducibility and repeatability. Both can still be achieved by a shortened Continuous Delivery pipeline. All tests are always reproducible. Further, the tests are repeated upon each change, so that they are performed much more frequently. This results in higher software quality and faster feedback—a fundamental goal of Continuous Delivery. In addition, the software at least has to be installed on test systems so that the installation is also in principle reproducible.

The higher software quality and the automated and therefore reproducible deployment of the application, at least on test systems, is already reason enough to introduce Continuous Delivery. Thus it is not only possible to introduce Continuous Delivery without DevOps but a reduced version can even be implemented entirely without support from operations.

Unfortunately, it is not possible to realize all the benefits of Continuous Delivery this way, but it is still a useful step. Besides, operations might still continue the Continuous Delivery pipeline later in production.

A Continuous Delivery pipeline that is only implemented by operations would only comprise the automated rollout in production. Automation should be implemented by operations anyway, to limit expenditures. However, only by means of the test phases does a rapid deployment in production become realistic, since only in that case is the software of demonstrably sufficient quality for production. Even if the deployment itself is entirely automated and functions reliably, nobody will bring new software into production without prior testing since only the tests ensure the necessary quality.

All in all, it is conceivable to implement Continuous Delivery without DevOps, but these approaches of course entail fewer benefits than an introduction in conjunction with DevOps.

10.5 Conclusion

DevOps extends Continuous Delivery by an organizational model where development and operations cooperate more closely. Continuous Delivery focuses on the delivery of software. For that the two departments have to work closely together so that DevOps is particularly helpful for Continuous

Delivery. However, the collaboration can go far beyond that—monitoring and troubleshooting can also be facilitated by the improved collaboration. The next step could be teams that are positioned even more broadly, as suggested by Design Thinking.[1] Lean Startup[2] focuses on the evaluation of features in order to improve and refine the product step-by-step. Since Continuous Delivery allows the delivery and evaluation of individual features, Lean Startup can be particularly well implemented with Continuous Delivery.

Endnotes

1. https://en.wikipedia.org/wiki/Design_thinking
2. Eric Ries: Kanban: *The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses*, Crown Business, 2010, ISBN 978-0-67092-160-7.

Chapter 11. Continuous Delivery, DevOps, and Software Architecture

11.1 Introduction

Continuous Delivery and DevOps are approaches that allow one to operate software more efficiently as well as more effectively and to bring it more easily into production. They mainly affect deployment and operating processes. However, this chapter focuses on an entirely different aspect: Continuous Delivery also influences the architecture of the applications. At first glance this connection is not so obvious—why should deployment and software operations influence software architecture?

First of all, section 11.2 defines the term "software architecture." Section 11.3 discusses how the split into components can be optimized for Continuous Delivery. Section 11.4 focuses on the interfaces between the components, and section 11.5 shows how databases can be handled, which can pose quite a challenge in the context of Continuous Delivery. Continuous Delivery is focused on the delivery of new features; section 11.6 centers on the delivery of these new features. Section 11.7 explains how to handle new features in the software architecture. Finally, section 11.8 provides a conclusion.

11.2 Software Architecture

Software architecture[1] can be defined as the structures that are necessary to understand a software system. This comprises individual software components, the relations between them, and the characteristics of the components and relations. This is a very general definition. Depending on the level at which software architecture is viewed there are different examples for components and connections:

- For an object-oriented system the components could be classes. In this case the relations are the use of a class by another class or the inheritance between the classes.

- Programming languages like Java, C#, or C++ offer the possibility of structuring multiple related classes in a package or namespace. These can also be used for the implementation of a component. The relations between these components are the use of a class from another package or namespace.
- On a coarser granularity level the components can be deployment units —and the relations can be dependencies between the individual units. Examples for such deployment units are WAR or EAR files for Java, DLLs for .NET, and libraries in other systems.
- At the level of enterprise software architecture the individual components can be software systems. In this case the relations are calls of one system by another.

A fundamental decision for software architecture is the technical implementation of components—that is, how software architecture ultimately manifests in the system.

11.2.1 Why Software Architecture?

Software architecture is relevant since a complete system is too complex to understand. Therefore, the system is divided into components. Developers can understand and change these components one by one. The relations between the components are also relatively simple to fathom. Only when the developers understand these aspects can they modify the components or the relations between them and thus further change the system.

Besides, the architecture defines the technical fundamentals of a system. In the end the individual components of the architecture have to be implemented, and this requires technical decisions. For example, object-oriented systems can be implemented in different languages, and different libraries can be used for typical problems like persistence. These decisions form a technology stack and influence the nonfunctional requirements like performance, security, and scalability.

As an example let us have a look at a very simple system that is supposed to represent orders in an enterprise (Figure 11.1). This requires three components:

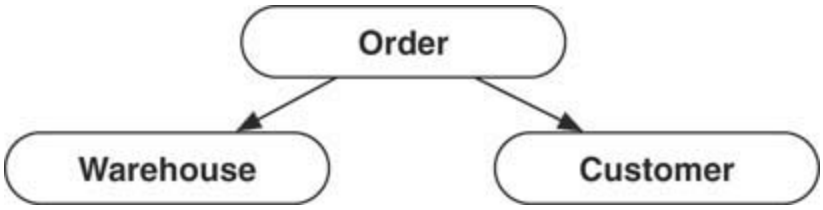

Figure 11.1 *Splitting a system into components*

- The order component is responsible for implementing the order process, and in addition allows for accessing the orders processed so far.
- The warehouse component keeps track of which goods are in stock, and in which quantities. It can also trigger the delivery of items to customers.
- The customer component knows the customer data and provides them for the processing of orders.

The division into components is solely driven by domain responsibilities and entirely technology independent. This approach is very important since each software system has to fulfill a certain functionality. To align the architecture along these functionalities supports the actual goal: the development of the software system to implement a certain functionality. Often, this goal gets lost during the definition of the architecture, and technologies creep into the foreground. An initial technology-independent design helps to avoid this problem.

Of course, a technical implementation of the components is necessary nevertheless: The components can be classes, packages, or deployment units. However, this decision is independent from the decision about the split of the domain functionality. Thus, one option would be to implement the entire system as a single deployment unit and to implement the individual components as classes or packages. This approach has a number of benefits: It is technically simple and offers high performance, since the components can communicate directly via method calls with each other; therefore, there is no overhead for distributed communication. If each component were a deployment unit, the project would be profoundly more complex, especially with regards to the build process, since each deployment unit is usually a separate project that has to be compiled separately and, afterwards, also has to be installed in the runtime

environment by itself. In addition, the components often have to use more complex communication mechanisms—for example SOAP or REST. This not only renders the implementation more complex, but also negatively affects performance.

In a first step all components can be merged into a single deployment unit to reduce complexity and to implement a high-performance solution. The communication between the components is implemented by method calls (Figure 11.2).

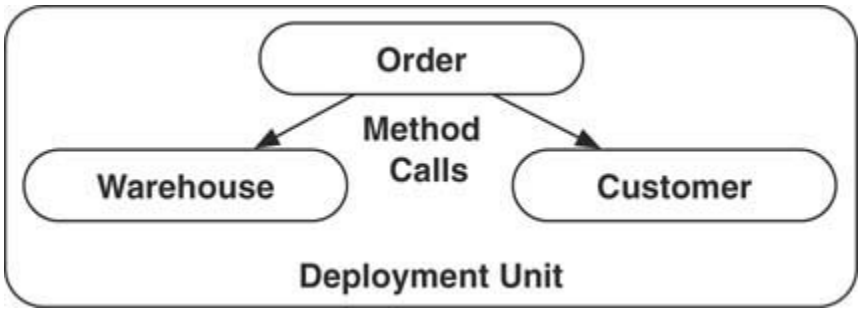

Figure 11.2 *Implementation of the component*

11.3 Optimizing Architecture for Continuous Delivery

However, when just a single component is modified—for example the one for handling orders—the entire Continuous Delivery pipeline has to be run. In the process the Continuous Delivery pipeline builds and tests not just the modified component, but all components. This causes a lot of unnecessary work and, more importantly, delays the feedback from the tests. Before the acceptance tests for the order component can be performed, the commit phase for all components is executed; thus all unit tests of all components are run, even though only one of these components has changed. Therefore, an error causing the acceptance test to fail will only be discovered after all unit tests have been finished.

Nevertheless, this procedure does make sense to a degree: When the pipeline is run through, all components should be tested. After all, it is possible that an error in one component only manifests by a dependent component showing an incorrect behavior. However, in the case of the orders this can be excluded since no other component uses them, as the architecture diagram in Figure 11.3 shows.

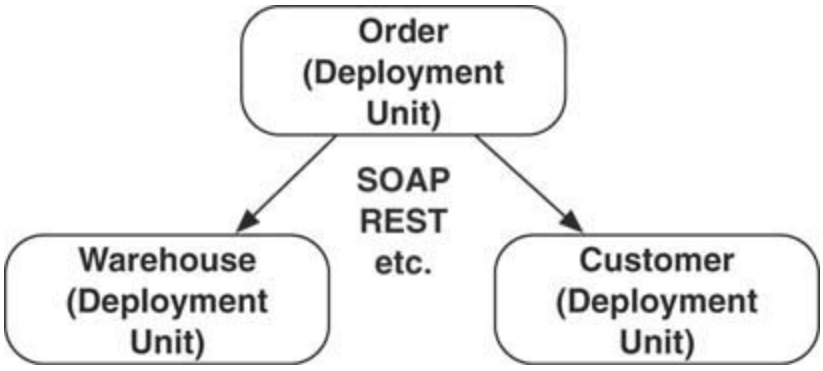

Figure 11.3 *Splitting the system into many deployment units*

11.3.1 Smaller Deployment Units

If this influencing factor is factored into the architecture and the architecture is optimized for Continuous Delivery and rapid feedback, another technical implementation of the components becomes sensible. Each component can be implemented as an individual deployment unit, such as a library, a Java JAR file, a Java WAR file, or an executable application in an arbitrary programming language.

In that case there will be individual Continuous Delivery pipelines for the components order, customers, and warehouse. Therefore, a change to the order component will then only trigger the execution of one pipeline so that feedback will be available much faster.

In addition, this approach minimizes risk: Instead of newly deploying the entire system only a part is deployed anew. Thus, less is changed, which translates into a smaller risk that an error might creep in somewhere. Additionally, the deployment is simpler and faster since the deployment units are smaller. For the same reason a change to a deployment unit can be more rapidly undone if a problem should still arise somewhere.

This approach can also affect the communication between the components: When the deployment units are libraries, method calls can still be used for this communication. However, in that case the deployment of a component usually requires at least a restart of other components for loading the library in a new version. But when the deployment units turn into individual services, which run as individual processes on the servers, the communication of the components also has to change. In this area there are different options:

- REST[2] can be used as the protocol. It uses HTTP not only for reading out data, but also for creating new data records or for modifying data records. JSON,[3] for example, can be used as the data format. But ProtoBuf[4] is more efficient with regards to the size of the messages and the performance. Another alternative is XML.[5] While being less efficient, it provides a very powerful type system and is mostly used in the enterprise context.
- An alternative to REST is the asynchronous communication via Message Oriented Middleware (MOM), which is used for many integration solutions, such as Enterprise Service Buses. This decouples the components even more.

In the end it is very advantageous for Continuous Delivery when the individual deployment units are as small as possible.

11.4 Interfaces

When a component provides interfaces—such as the interface of the warehouse component or the customer component for the order component—there is an additional problem during deployment: In the case of a change to the interface all dependent components also have to be newly deployed. The dependent components use the interface and have to track the changes to be able to appropriately use the interface in the new version. Now the problem is this: It is not a single component that has to be newly deployed, but numerous components. Thus the same problem that was meant to be avoided by deploying small components is back: The deployment becomes complex and risky.

To solve this problem the old version of the interface also still has to be supported after a modification of the interface. In this manner a dependent component can still use the old interface. Therefore, it is not necessary to deploy all components anew at once; they can be deployed step-by-step. First, the component is deployed anew, which offers the new interface. Subsequently, all components that use the interface are shifted to the new version and deployed anew. Afterwards the support for the old interface can be terminated. To support an additional version of an interface creates, of course, an increased effort. However, compared to a complex and extensive deployment this is surely the better alternative.

In distributed systems the versioning of interfaces is not unusual at all. To do so it is, for instance, possible to use the version number as part of the name—for example as a part of the URL of a REST resource. The support of old interface versions can of course only be stopped when there is no other component using the old version anymore. Normally, this can turn into a big challenge: In some cases it is not even known which components still use a certain interface. However, in a Continuous Deployment scenario this problem can be solved: Usually, all components belong to one application, and one team is responsible for the application in its entirety. Therefore, the team controls all components and thus can limit the support of old interfaces in a sensible manner—for example, to the current and the previous version. Since the effort and the risk of deployment of components decreases with Continuous Deployment, it is easily possible to deploy components anew when they just use the new interface of another component. Usually such a modification is not very comprehensive, which decreases the risk further. In fact, in distributed systems the old interface is still offered just because nobody wants to take the risk of a new deployment.

Of course, this also means that modifications of interfaces for external components that are implemented by other teams, or of public interfaces that are provided via the internet, are still only possible with great care: The users of these interfaces are outside the realm of influence of the team, and therefore it is far more difficult to just stop providing a version of an interface.

11.4.1 Postel's Law or the Robustness Principle

The presented concept for interface versioning is actually a special variant of Postel's Law, which is also known as the Robustness Principle.[6] It states that components should be strict in what they do and liberal in what they accept from others. Phrased differently: Each component should be strict when using other components—that is, adhere as exactly as possible to guidelines—but whenever possible accommodate errors upon its use. When each component behaves in line with the Robustness Principle, interoperability will improve: If each component complied exactly with the requirements of the other components, interoperability would already be ensured, but should there nevertheless be a deviation, the employed component will try to accommodate it.

Applying this principle to the versioning of interfaces means that each component should accept calls to the old version of the interface, but should use only the most current version of other interfaces.

11.4.2 Design for Failure

However, even if the interfaces are versioned, there is still a number of problems to solve. During the update of a component, it can be down for a certain time. If the component is called during this outage, the other components have to appropriately deal with this situation.

There are different approaches for dealing with the outage of components:

- Default values can be used. If for instance the component that normally displays recommendations for the customer is down, generic recommendations might be shown when this component is not available.
- Instead of the original algorithm, a simplified version can be used. For example, if the credit rating of a customer cannot be determined, a simpler algorithm can be used that accepts a credit up to a fixed upper limit. This is in fact primarily a business decision: Is the enterprise willing to forego sales in case of an outage, or to accept a certain degree of risk?
- Another approach is the Circuit Breaker.[7] When the call of another component is not successful, other calls are not forwarded anymore to that component—that is, the Circuit Breaker prevents calls to the component and immediately responds with a suitable error message. In this way it that a system wait for the response of an unavailable component can be avoided. In addition, this approach prevents calls to the component piling up. If the accumulation of calls is not prevented, the component has to deal directly with a high load once it is available again, which might cause it to fail again immediately. The Circuit Breaker can be combined with the other measures so that it not only sends an error message, but compensates for the outage of the component as far as possible.

In summary, the concept is to revert to a simpler alternative and on no account to “pass on” the outage of the component. This allows isolation of

the outage of a component, for example during deployment.

This makes the system overall more resilient. In the end the assumption with regards to calling other components is that the components are not always available and that sometimes they just do not respond. Especially for distributed systems this assumption makes a lot of sense, since the network can fail or other problems with the hardware can result in the outage of components.

11.4.3 State

Another problem during the deployment of a new version of a component is the state that the component keeps in its memory. This state usually gets lost during the update of a component. Therefore, for Continuous Delivery, components should not have any state if possible. More precisely, they should not have a state that is kept in the memory of the component. In the end handling state is the focus of most systems, so that a system entirely without state does not make much sense. However, this state should be kept in an external system such as a database or a cache. A cache is volatile of course—but that is also the case of the state of a component is only stored in memory.

By the way, it is also possible to solve this problem by means of the infrastructure. For example, some web servers can store the state. In the Java system the term HTTP session is used. However, the information from an HTTP session can also be stored in databases or caches, like memcached, without any code modifications. Of course, a new software version has to be able to deal with the information regarding state that was written by an old version.

11.5 Databases

Suppose that the system state is saved in a database. Dealing with databases is a challenge in a Continuous Deployment scenario. If changes to the data structures in the software are made, the schemas in the database also have to be modified—for example by adding or deleting columns. To do so, data already present in the database has to be appropriately changed. Such changes can potentially affect a large amount of data. This makes changes very hard to implement. In addition it is also difficult to roll back any changes. This is due to the fact that changes of databases usually involve

very large amounts of data and that any change that affects all of the data takes a long time, or is not even feasible under certain conditions. It is in the nature of things that changes take a relatively long time. The changes are also hard to test because this requires a database that is comparable to the production database. Moreover, providing such a database for tests is laborious, and the required hardware and software are expensive.

Even if the software can easily be changed and newly deployed by the measures described in this chapter, the changes to the database are still a hurdle that has to be overcome first.

11.5.1 Keeping Databases Stable

To solve this problem we must first find a solution to keep the databases stable. When a component requires a change to the database schema, the respective modification of the database is first performed. This is done independently of the rollout of the actual component. For example, the database schema is only changed on a certain day of the week, whereas changes to individual components can happen on any day. This change can be safeguarded by appropriate tests and security measures.

When a change to the database is necessary, this change is first introduced and then the components that are supposed to work with the modified database are redeployed. Often it is not possible to roll back changes to the database because this requires a complex process like the one which was necessary for the original change to the database. To securely implement and test this process is relatively laborious so this step is often omitted. This distinguishes a change to the database schema from a change to a component, for which a rollback is usually relatively easy. This makes it, of course, harder to bring changes quickly into production.

In the end a pragmatic approach is the best way to deal with the problem. Rather than really solving the problem it should be avoided as much as possible by changing the database as rarely as possible. A solution to the problem would surely be more laborious so although this approach might not appear very elegant, it is not very complicated to implement. In fact, it is successfully used in many projects.

However, to a certain degree this approach disagrees with the philosophy of Continuous Delivery since it is a fundamental goal of Continuous Delivery to automate processes as much as possible and to run through

them more frequently. Therefore, release cycles that often take months are replaced by frequent releases that in many cases can even be performed several times per day. The fact that release cycles without Continuous Delivery take so long is due to the risk associated with classical releases, which have to be cushioned by complex manual security nets—exactly as in this approach for dealing with databases.

11.5.2 Database = Component

Treating the database simply as a component is another option to deal with the problem. When a component changes its interface, it should also still support the old version of the interface. The interface of the database is the schema. Therefore, the schema also has to be versioned, and the change to the schema has to be performed in such a way that the previous version is still supported. When for example a column is supposed to be added to the schema, any read access could just ignore the extra data. For write access there have to be appropriate default values in the database. When a data record is inserted without a value for the new column, the database will provide a default value instead. This ensures that components that were designed to deal with the old version of the database schema also work correctly with the new schema. When a column is removed, initially the schema can be changed so that values for this column become optional. Subsequently, a new version of the software can be deployed that no longer writes or reads this column. Finally, upon the next database update the column can really be removed from the database.

11.5.3 Views and Stored Procedures

There are still other options to ensure compatibility. For example, views can be used to simultaneously offer the old and the new version of the schema to the components.

In addition, it is possible to completely hide the schema and instead to allow access only via Stored Procedures. In that case the schema can be modified without restriction as long as the interface in the Stored Procedures remains. In the end the database is really a component with its own API, which is offered via the Stored Procedures. The persistence layer is implemented, so to speak, in the database. However, it is often difficult to implement such a solution.

While this approach is more complex to implement, it offers backwards compatibility, which might be even more important for databases than for other components since database changes are much more difficult than changes to other components.

11.5.4 A Database per Component

By the way, the problem can be reduced by each component having its own database or at least database schema. In that case a database change only has to be coordinated with a single component—this is much easier and further reduces the risk associated with a database change.

11.5.5 NoSQL Databases

Problems with schemas are specific for relational databases. NoSQL databases are much more flexible with regards to schemas and can handle data records of nearly any structure. Thus, with NoSQL databases it is quite possible to simultaneously process data records with completely different structures. Of course, it is still necessary to convert the data records in a suitable manner. And the rules for compatibility still apply—as long as components still expect that certain columns are present, they may not yet be deleted from the datasets. Nevertheless, NoSQL databases have significant benefits in Continuous Deployment scenarios due to their greater flexibility. These benefits by themselves can be sufficient motivation to use this type of database.

11.6 Microservices

Microservices[8] are not only an interesting approach for software architecture that have received quite a lot of attention lately, but they also have comprehensive synergies with Continuous Delivery.

Microservices modularize systems. What is special about this modularization is that each Microservice can be independently deployed. Concretely, Microservices can, for example, be Docker containers in which the software that constitutes the Microservice is installed. Together the Microservices form the overall software system. To do so they can each provide a part of the web interface. Thus, a system can consist of Microservices that have HTML links to each other. Alternatively, parts of the HTML pages of the Microservices are reloaded with JavaScript and in

this manner are displayed together. This approach is the basis for Self-Contained Systems (SCS).[9]

Alternatively, Microservices can offer services via, for example, REST services that can be used by other Microservices, other external systems, or mobile clients. In addition, there are Microservices that offer a user interface, and other Microservices, that are called and implement the business logic.

11.6.1 Microservices and Continuous Delivery

This architectural approach has a number of benefits, particularly for Continuous Delivery:

- As described in section 11.3, the division into multiple independently deployable units can simplify the creation of a Continuous Delivery Pipeline since the pipeline is less complex. The risk of a deployment is also reduced because each deployment only changes one Microservice and not the entire system.
- The Microservices can communicate with each other via suitable interfaces (section 11.3). Versioning of interfaces enables independent deployment of Microservices. If a change to an interface is necessary, first a Microservice is deployed that simultaneously offers a new and an old version of the interface. Only afterwards are all Microservices that use the new interface deployed. Finally, the old interface is removed. When the Microservices each implement a part of the web user interface, such interfaces will be an exception, because the Microservices do not need to communicate that much with each other but just represent a part of the web interface. Consequently, the implementation of the system is easier since there is no need to version interfaces and therefore no problem with it.
- Microservices are supposed to be resilient. This means they still have to work when other Microservices are down. Therefore, they implement "Design for Failure" and resilience as discussed in section 11.4. This further reduces the risk associated with deployment because the breakdown of an individual Microservice does not affect the other Microservices.

- With regards to databases (section 11.5), each Microservice should have its own data. Thus, a Microservice may not share schemas with other Microservices. This at least reduces the problem associated with updating the databases: Just one database has to be updated for one Microservice. Likewise, each Microservice can use its own individual database. For instance, a Microservice can use NoSQL databases if that makes sense in the specific context. Of course, using a NoSQL database in just one Microservice is naturally much less risky than the conversion of an entire application.

11.6.2 Introducing Continuous Delivery with Microservices

Thus, Microservices support the architectural demands of a Continuous Delivery system quite nicely. In addition, Microservices can be used quite easily for supplementing a legacy system. Therefore, when implementing a Continuous Delivery pipeline is too laborious and complex for a system, the system can instead be supplemented with Microservices for which Continuous Delivery is much easier to implement. The aim to achieve Continuous Delivery in conjunction with the possibility of supplementing legacy systems by Microservices is, for many projects, the main reason for replacing legacy systems step-by-step with Microservices. Thus, the introduction of Continuous Delivery is often in reality an introduction of Microservices.

11.6.3 Microservices Entail Continuous Delivery

However, Microservices represent not only a good means for introducing Continuous Delivery. A Microservice-based system consists of numerous, independently deployable artifacts. This can only work when each Microservice has a largely automated Continuous Delivery pipeline. Due to the multitude of deployable artifacts manual interventions are not feasible anymore; they are just too laborious. Particularly in the areas of deployment and operations, Microservices pose the largest challenges. Therefore, Continuous Delivery is essential to be able to work with Microservices. In fact, Microservices cannot be implemented without Continuous Delivery.

11.6.4 Organization

The organizational structures presented in section 11.2 not only apply extremely well to Continuous Delivery, but also to Microservices: They

also support individual responsibility and self-organization. Microservices allow for technological freedom. Each Microservice can be implemented in a different programming language and on a different platform. Thus, a team that is responsible for a specific Microservice has little need to coordinate its work with other teams. This saves coordination effort since each team can make its own technological decisions and can work more easily on different domain aspects. Of course, the individual Continuous Delivery pipelines also allow for complete independence with regards to introducing software in production. The individual feature teams will also profit from this fact because they can not only develop features independently, but also bring them independently into production.

In summary, Microservices can be an excellent complement for Continuous Delivery.

11.7 Handling New Features

A new feature is usually implemented in the code and is available immediately once the implementation is complete and deployed in production.

11.7.1 Feature Branches

To decouple the work on features, in some projects feature branches are set up in the version control system. An individual branch is set up in which the work on the feature takes place. Work on other features or fixing of bugs takes place in other branches so that these changes do not influence each other. This allows one to completely decouple the implementation of different features so that different teams can work in parallel on the different tasks. Furthermore, in this manner the implementation of a feature is separated from the other features so that the features can also be brought individually into production.

In principle, this approach is not compatible with a Continuous Deployment strategy. In fact, for each branch an individual Continuous Deployment pipeline would have to be set up. However, this is laborious. Besides, in this manner a fundamental goal of Continuous Delivery is not achieved: The branches are developed further in parallel and at a certain time have to be integrated. Problems that arise due to the fact that changes in the branches are not compatible with each other only manifest during the

integration of the branches. However, the Continuous Delivery pipeline is actually meant to provide feedback all the time and as fast as possible. In the case of feature branches the feedback for such integration problems is delayed until the branches are integrated. Therefore, feature branches do not fit very well with Continuous Delivery.

Consequently, all developers should work on a single branch. However, there is still a problem with the implementation of new features: There has to be a mechanism for activating new features in production. In the end the features are not immediately implemented in full—and in this state they should not yet be employed by the users. Activating a feature is often also a business decision and can be supported by, for instance, a marketing campaign so that it has to be done at a specific time.

11.7.2 Feature Toggles

Using the approach of feature toggles it is possible to develop features together and to continuously integrate them, but to only activate them at a defined point in time. They represent a switch that activates or deactivates a certain feature. Thereby, the new feature can be implemented and code in production can immediately contain the feature—it is just not activated yet.

11.7.3 Benefits

This approach has a number of benefits:

- The implementation is decoupled from the deployment. The code for the new feature can be brought into production before the feature is activated. This allows the team to handle deadlines in a more relaxed manner: Frequently, a feature has to be available to the user by a certain deadline. In the conventional way of doing things the code has to be brought into production exactly at this time point—not earlier or later. With feature toggles the code can be brought into production before the assigned date. At the end of the deadline the new feature just has to be activated—a relatively low-risk endeavor.
- The feature can also already have been tested. This only requires its activation for certain users. With the help of these test users it can be examined whether the feature really behaves as intended. This does not require a specific testing environment, so it can be ensured that the feature actually also runs on the production environment as desired. Of

course, precautions have to be taken for separating the testing from the production operations to avoid, for instance, test orders actually being delivered. On the other hand, it is not necessarily required to provide a testing environment that is identical to the production environment. Usually, such a requirement could not be fulfilled anyhow since the costs would be much too high and because certain external systems would not be present in multiple versions.

- Feature toggles also allow one to test how a new feature is liked by the customers or to selectively implement features for certain user groups. To do so the feature has to be activated for a certain circle of users and stay inactive for other users. Afterwards one can examine whether the features really had the intended positive effects on, for example, the sales volume or other business numbers. The technical term for this is A/B testing. From the business perspective A/B testing is very important since experience shows that a lot of software changes do not have the intended effect or even have a negative effect. A/B testing allows one to recognize potential negative effects of software changes early on and to focus on those changes promising to improve the business—for example, by raising the volume of sales.
- In addition, it is possible to activate a new feature first on one or a few servers. If the implementation contains an error that, for instance, causes the system to crash, the effects can be limited to a small number of servers. Step-by-step the feature can then be activated on more and more servers. This is an implementation of Canary Releasing (section 7.5).
- Finally, feature toggles can also have useful applications outside of Continuous Delivery. For example, in the case of an outage of a certain system a feature can be deactivated. Then this specific feature is not usable, but this might in fact prevent the breakdown of the entire system. However, this relates more to operating the application and not so much to Continuous Delivery.

11.7.4 Use Cases for Feature Toggles

Thus, feature toggles[10] have many positive effects. Three different areas can be identified for feature toggles:

- Release toggles serve to decouple the activation of a feature from the release date for the code changes in the feature. First the code is deployed and the feature deactivated. When the feature is really completed and tested, the toggle is activated.
- Business toggles have the purpose of testing features for customers or selectively offering features only for defined groups of customers.
- Operations toggles allow one to deactivate features to avoid the breakdown of the entire application.

Each of these scenarios has specific differences and therefore different demands on the toggles.

11.7.5 Disadvantages

Feature toggles also have disadvantages: They increase the complexity of the software, since there has to be a distinction of cases that activate or deactivate the features. Moreover, in addition to the implementation there have to be tests.

To reduce complexity it makes sense to implement feature toggles as simply as possible. It can, for instance, be sufficient not to display a link on an HTML page so that the feature cannot be reached.

In addition, the number of feature toggles should be limited. In corner cases it can be sufficient to implement a single switch that activates or deactivates all new features. Alternatively, the feature toggles can be implemented in a more fine-grained manner. Which approach fits best depends on the objective that is supposed to be achieved with the feature toggles. If, for instance, certain features are supposed to be evaluated by A/B testing, there has to be a feature toggle for each of the features. The same holds true for the use of feature toggles in compensating for the outage of a system. Depending on how the feature toggles are actually used, different combinations of feature toggles also have to be tested. When certain features have in principle already been tested, but have not yet been activated in production, the configuration of the feature toggles from production still has to be checked. Otherwise the risk is too high that changes to the code create problems that only manifest in production with a given configuration of the feature toggles.

Feature toggles that are not needed anymore should be deleted from the code to reduce the complexity of the code.

11.8 Conclusion

Continuous Delivery has effects on software architecture—even if it does not look like it at first glance. Therefore, the software architecture might also have to be adapted when introducing Continuous Delivery. Just changing the processes is not sufficient. This aspect is often ignored and can be an obstacle to the successful implementation of Continuous Delivery. In the end, in addition to the classical nonfunctional requirements (like performance or scalability), the handling of new features and the deployment also constitute influence factors for architecture, the design of components, and the technology choice.

Try and Experiment

- Do some research into the topic of "Microservices" and their effects on Continuous Delivery.
- Which libraries for implementing feature toggles are available in your preferred programming language? Which benefits does the use of these libraries have? For Java there is, for instance, Togglz.[11]

Choose a project you know well:

- Which components can be identified in the project? Often architecture documents or overview presentations can be helpful for identifying components.
- Which component (for instance REST) can be used to decouple the components for Continuous Deployment appropriately? Which advantages does this technology have compared to other approaches?
- Which components should be decoupled in this manner? Pay attention to the increased expenditure for the uncoupling and dependencies during deployment.
- How are changes to the database schema supposed to be handled? Select one of the defined strategies.

Chapter 12. Conclusion: What Are the Benefits?

This book should have made clear that Continuous Delivery is much more than just an automation of infrastructure and an optimization of deployment. In fact, the central objectives of Continuous Delivery are:

- **Feedback**
  Faster and more frequent deployment results in faster feedback from production. The different test phases also focus on providing feedback with regards to software problems as fast as possible.
- **Automation**
  Tests and deployment are automated for Continuous Delivery to allow for rapid feedback.
- **Reproducibility**
  The extensive automation allows one to exactly reproduce test results and software installations.
- **Easier Error Search**
  All changes to servers and software are versioned. A change to a firewall configuration can, for instance, be rapidly undone by bringing back the previous software version of the configuration. Further, it is always transparent exactly what was changed with regards not only to software, but also to infrastructure.

In the end, Continuous Delivery is a continuation of Continuous Integration. Not only are all software changes continuously integrated, but comprehensive tests are also executed, and the software is brought into production if it is of sufficient quality.

The higher speed with which changes can be implemented translates into a profound advantage for the business. The higher reliability is mainly attractive for the IT department since it means that there is less stress.

Continuous Integration is, meanwhile, the standard in industry. Very likely Continuous Delivery will be as successful. In the future it will just be the routine process for developing software. However, Continuous Delivery affects the entire process up to production and therefore represents a much more fundamental change than Continuous Integration.

The next step after adapting Continuous Delivery is probably going to be a change of software architecture as discussed in Chapter 11, "Continuous Delivery, DevOps and Software Architecture," to facilitate the handling of Continuous Delivery. A Microservices-based architecture[1] fits very well with Continuous Delivery because it allows one to reduce the size of components, which in turn facilitates the pipelines.

From an organizational viewpoint Continuous Delivery has very strong ties to DevOps. However, feedback from production and the definition of new features is also interesting for business areas and departments such as marketing and sales. Here, Lean Startup[2] and Design Thinking[3] come into play (see sections 1.4.7 and 10.5). Thus, Continuous Delivery is an optimization of software development and results in profound benefits. However, the next steps are already on the horizon.

Index

B

C

D

E

F

Q

R

S

T