INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Bachelor's Thesis

Implementing a continuous deployment pipeline for a web application

Roman Anasal

INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Bachelor's Thesis

Implementing a continuous deployment pipeline for a web application

Roman Anasal

Evaluator:	Prof. Dr. Dieter Kranzlmüller
Supervisor:	Dr. Nils gentschen Felde Alexander Kiefer (FlixBus GmbH)
Submission date:	September 15, 2015

I hereby assure the single-handed composition of this bachelor's thesis only supported by the declared resources.

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Munich, September 14, 2015

. . . . . .

Roman Anasal

Abstract

Developing software in a team with several developers using agile software engineering patterns two common challenges are the regular integration of code changes made by one developer with changes made by another developer and the subsequent verification of the success of this by testing the new code with automated tests as well as with manual and user acceptance tests. In cooperation with the Java team of FlixBus who faced these challenges the objective of this bachelor's thesis was the implementation of a continuous deployment pipeline for a regular web application of FlixBus.

Selecting the emergent Docker container management engine as a suitable technology a concept for the continuous deployment pipeline was created and implemented according to the requirements imposed by the team, their agile development workflow and the technology stack of the application. The resulting implementation was put into operation, integrating it into the existing tools and workflows of the team. It is since then providing the team with a fully automated provisioning and deployment of new test instances of the application in a dedicated integration environment and an automated process for deploying successfully tested versions as a new release into production use, all at the push of a button.

Contents

1 Preface

In the course of this thesis we will be accompanying the development process of a web application together with the objective to improve the development processes by implementing a continuous deployment pipeline for it.

Working on this implementation was done in cooperation with FlixBus and its Java development team which is developing the application. Therefore this made this a very practical oriented task, developing and implementing a solution for a real-world problem of a real-world application. Still being a rather young startup company, FlixBus and therefore its developers have to be flexible and adaptive in order to serve an equally young yet highly competitive market which is reflected by the agile development practices put in use by the developers.

Utilizing agile software engineering patterns has its own upsides as well as its own downsides, some of which the practices of continuous integration - on which continuous deployment is based on - try to tackle by emphasizing early feedback over success and failure of submitted code changes and by pushing successful changes to end users as soon as possible. The foundations of all of this will be discussed in more depth in chapter 2 where we will also see that the basis for this is heavy automation and enhanced reproducibility of the development process. Further improving this automation is one goal of the continuous deployment pipeline to be implemented here.

In the subsequent chapter 3 the focus lies on the structured assessment of the requirements which the desired continuous deployment pipeline is expected to fulfill and which then are the basement for a technology independent concept of the pipeline developed during the following chapter 4 after we took a closer look at the status quo of the development process. After both - the examination of the status quo and the development of the concept for the pipeline - a quick evaluation of the coverage of the requirements from chapter 3 is done respectively.

Then the next chapter 5 constitutes the majority of this whole thesis since it describes and explains every single part of the pipeline implemented according to the concept of chapter 4. At the end of the chapter again the coverage of the requirements is checked and this time also an evaluation of the final implementation is made. Although the detailedness of chapter 5 is intended to allow the reader to reproduce the implementation of the pipeline it does not go into extensive depth in order to not blow the scope of this thesis. Also all application and company specific aspects are kept to a minimum wherever possible and abstracted for simplicity most of the time.

Furthermore a fundamental experience of the reader in dealing with Linux-based machines to fully comprehend the descriptions of the implementation in chapter 5 is more than helpful. Likewise advanced understanding of virtualization technologies in general and in combination with Linux in particular is beneficial since these will be considered and partially used in the conception and the implementation phase. Nevertheless the eventually used technology - Linux containers together with Docker as management toolkit - will be adequately illuminated to enable the reader to follow even without previous experience with

this technology.

The final chapter 6 then concludes the experiences made in the course of the implementation by collecting the most important lessons learned and caveats discovered during the conception and implementation of the continuous deployment pipeline of this thesis. At the end of chapter 6 this thesis then finishes with an outlook to possible further improvements of the development process of the application which can now be realized extending the new continuous deployment pipeline and the techniques it introduced. Thoughts on how these can be incorporated and early concept ideas are also given, leaving the reader with a solid starting point for further adventures with Docker and continuous deployment.

2 Foundations

In the course of this paper we will be accompanying the implementation of a continuous deployment pipeline for an internal application of FlixBus.

Launched in Germany in February of 2013 FlixBus[1] is a young but steadily growing intercity bus service provider. Although FlixBus does not itself perform the transportation of passengers which is done by associated bus companies but provides all of the infrastructure around it like the booking and ticketing system, customer service via call center and on site, planning and pricing of the transportation network and of course the brand.

In January 2015 FlixBus then merged with its formerly major rival on the market MeinFernbus which now make up the new brand MeinFernbus FlixBus. By now it has over 700 employees (excluding bus drivers) in two headquarters - one in Munich, one in Berlin; and new headquarters in other European countries are already in construction.

2.1 About planr

The application the pipeline will be for is *planr*, a planning tool for the transportation network of FlixBus, developed in-house and for in-house use. It is developed by the Java development department which consisted four developers at the start of the implementation in the end of 2014 and has grown to a total of eight developers meanwhile.

2.1.1 Purpose of the application

The application is used for managing the transportation network and bus lines of FlixBus. It is supposed to provide the network planning department a reliable and easy to use tool to fulfill their task. The department has about 50 employees and was currently previously using Excel sheets for its work which were shared through network drives. Although sufficient in early days when the planning department was small and the bus lines to manage were few the Excel solution became conceivably inappropriate for the new challenges of the grown network which is guaranteed to grow even further.

Therefore FlixBus decided to task its in-house Java development department with the development of a tool for this which started the development of planr in November 2014 and launched into production use in July 2015. The objective was not only to develop a drop-in replacement for the Excel sheets with a more consistent and validated database but also to provide a tool which eases the work of the planning department and conquers the challenges arising with the expansion of FlixBus to - among others - France and Italy.

2.1.2 Technology stack used

The application is designed as a browser based web application and is written mainly in Java on the server-side and JavaScript on the client-side. Thereby it builds upon existing

[1] `https://www.flixbus.com/company/about-flixbus`

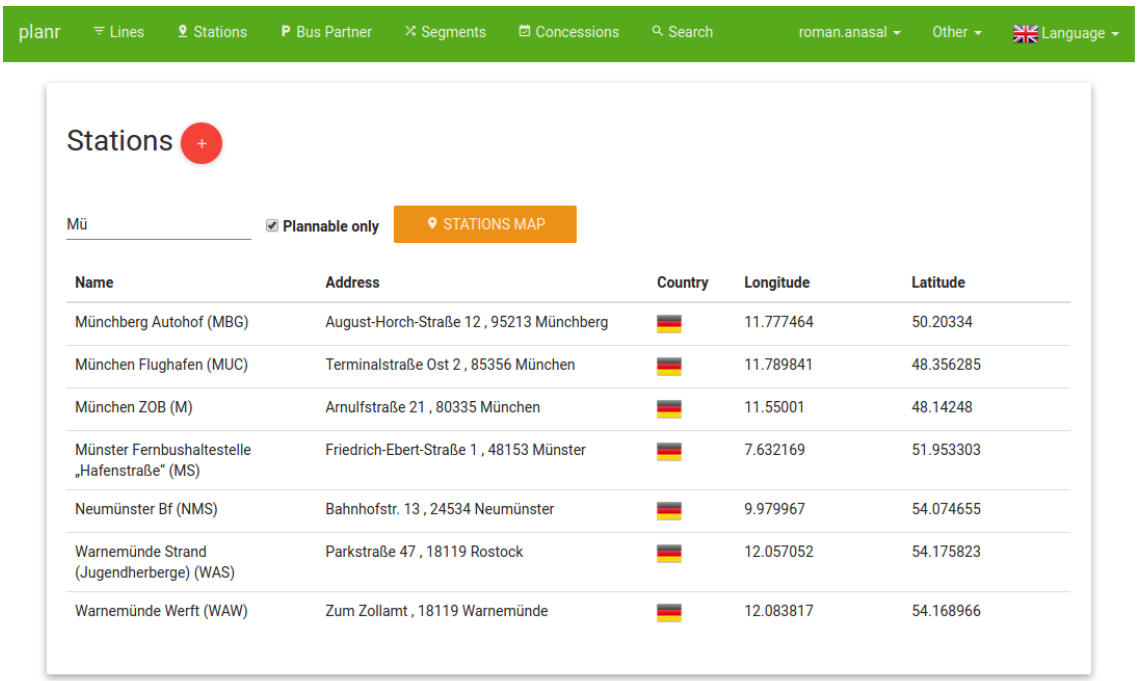

Figure 2.1: a screenshot of the user interface of planr

frameworks such as Spring[2], a Java framework for MVC web applications and RESTful web services, and AngularJS[3], a JavaScript framework for client-side MVC web applications. As its database backend planr uses a current MySQL and it embeds an Apache Tomcat for the handling of HTTP.

Development is done mostly on the local machines of the developers tracking the changes with Git as version control system and using Gradle[4] as build automation system. A private instance of Atlassian Stash[5] is acting as the central remote code repository which is also integrated with a private instance of Atlassian Bamboo[6] acting as the build server and providing automated testing. Bug tracking and reporting is done using JIRA[7], also an Atlassian tool. While most developers use a Mac hardware as well as Mac OS X as operating system on their local machines the final production systems and the integration system are all Linux based VMs run in a private cloud alongside with Stash and Bamboo. As the development methodology the team follows the agile pattern of Scrum.

2.2 Continuous deployment - definitions and motivations

Before we start digging deeper into the subject we first need to define what this is all about and what the motivations for this are. But for a better understanding of the solutions we also

[2] https://projects.spring.io/spring-framework/
[3] https://angularjs.org/
[4] https://gradle.org/
[5] https://www.atlassian.com/software/stash
[6] https://www.atlassian.com/software/bamboo
[7] https://www.atlassian.com/software/jira

need a clear picture of the challenges and common problems they intend to fix. Therefore we will be looking at the principles and motivations of continuous integration, continuous delivery and finally continuous deployment in the following sections, which build on top of one another.

2.2.1 Continuous Integration

In a typical setup a development team consists of multiple developers working simultaneously on the code. In order to develop new features without interfering with each other often a branching scheme is used whereby the developers branch from the mainline code - also often called the "master" branch or "trunk" - commit their changes to this feature branch and merge it back into the mainline at the end of the development cycle of that feature.

With multiple branches of the codebase diverging for too long this scheme yields some serious issues especially when merging back into or from the mainline is only done once at the very end of a lengthy development cycle, because the changes on the various branches may interfere or even conflict. So merging these changes together typically requires some integration work to flatten out these conflicts and harmonize the changes. This causes some teams to schedule a dedicated integration phase at the end of a development cycle.

But without knowing the degree of interference between the branches beforehand it can be very difficult or impossible to predict the time needed for integration. Best case it only needs some or no adjustments when the changes barely overlap, worst case changes conflict in such a way that features have to be reworked or even rewritten completely which consequently makes the integration process expensive. Worsening this can be the delay of testing into the integration phase which means the code is in an inconsistent, non-runnable state most of the time and therefore some newly introduced bugs may only be identified at the very end. In this scenario the code has to be considered "broken until proven to work"[HF11]. But at this point the developer may have already moved on which makes it even more expensive due to the mental "context switch" needed here.

Continuous integration intends to counteract these drawbacks by integrating often and integrating early. This means each branch should be merged back to the mainline at least once a day so issues with integration are identified just when they emerge and therefore can be fixed when they're still cheap to fix. Identifying integration issues this way also implies to automatically build and test the changes merged to give timely feedback to the developers and immediately fixing problems appearing thereby.

This way the opposite of the aforementioned comes true: the code is "proven to work" [HF11] with every change since breakage is identified instantly and also fixed immediately.

2.2.2 Continuous delivery

Building upon continuous integration continuous delivery expands these ideas to the delivery process. Similar to a delayed integration phase placing the delivery only at the very end of development comprises some significant drawbacks. Being able to deliver only every now and then means users/customers have to wait that long until new features are available for them in production use. Considering new features as value added to the product this means the users have to wait for this added value although it might be ready for some time already. This might cohere with a manual packaging process which may be felt as effortful

and cumbersome to the team and therefore only executed reluctantly. In any case a manual packaging conceals some error-proneness.

Moreover such a process prevents the developers from getting valuable feedback from the users. Just like the feedback provided by automated tests in the continuous integration process early user feedback enables the developers to fix bugs just when they emerge and therefore to fix them cheaply. Furthermore in the case of continuous delivery it allows the developers to verify that their understanding of the user requirements is correct more often and so prevents the costly development of features which provide no real value to the user.

Being able to deliver regularly and in a reliable manner of course again requires automation of the process making it reproducible and far less error-prone.

2.2.3 Continuous deployment

Even one step further now goes the idea of continuous deployment. The one extra step it demands is to also automate the deployment of a new release into production. This means the pipeline beginning with the check-in of the code and ending with the deployed product ready for the user in production is fully automated.

To achieve this the environment the testing takes place in needs to be as production-like as possible to mitigate unpleasant surprises when releasing into production. On the other hand this ensures that the product can be reliably deployed at any time at the push of a button.

So in one short sentence as Jez Humble puts it: "Continuous deployment is the practice of releasing every good build to users – a more accurate name might have been "continuous release"" [Hum10].

2.3 Provisioning strategies

The one fundamental principle of continuous integration up to continuous deployment is automation. In order to properly automate the various steps in a reliable and reproducible way - which is a basic requirement for continuous integration - the environments in which these steps take place either need to be in a known state or can be transformed into such on demand. Otherwise the automation may build upon premises which aren't fulfilled by the environment and may therefore have an unexpected and unintended outcome. The "state" of an environment consists of e.g. the software installed on the system (operating system, applications, libraries etc.), configuration or data accessible.

Therefore the management of the different environments like systems for production and integration is an important component of the pipeline for which different approaches exist. I'd like to take a closer look at some of these in this section.

2.3.1 Manual provisioning on bare-metal

Without much automation in place an usual way to go is setting up the systems manually, which is installing and configuring the operating system and software needed by hand. This is often done on demand and without much documentation of what was done and why. In terms of continuous integration this should be considered bad practice because reproducing an environment set up this way can be very difficult up to impossible due to the lack of a

complete documentation. So provisioning a new environment or recreating an existing one is a manual process entailing the risk of errors and incompleteness.

2.3.2 Automated provisioning

To overcome the error-proneness of manual provisioning, tools such as Puppet[8] or Chef[9] emerged to automate the provisioning and configuration management. These tools use some form of a description of the desired state, validate that the system is in the desired state and if necessary apply the required steps to transition to the desired state for example by installing software, changing configuration files or creating new users.

Environments managed with such tools can be reproduced far more easily because in the best case it boils down to a single, simple command required to trigger the fully automated provisioning based on an existing configuration file. Nevertheless there may still occur inconsistencies between environments when there are changes made to them outside of what is tracked and managed by the automatic provisioning tools.

2.3.3 Virtual machines

Another way for reproducibility is utilizing virtualization: instead of installing an environment directly on a bare-metal host machine it is installed into a virtual machine (VM). Hereby a complete machine including all its "hardware" is virtualized by the underlying host system, often complemented with virtualized networks too. This additional abstraction from the underlying/real hardware allows virtual machines to be portable from one host system to another by copying the configuration (e.g. as XML file) and virtual hard drives between them. Using these technologies a VM can also be completely cloned onto another host while the original is still remaining where it is and therefore allows to copy a production VM and use it as a production-like integration system.

However this method is quite resource consuming because a single VM can occupy several GB even for small sized applications because its filesystem has to house a full-blown operating system with all its system services, even if they might not be needed for the target application itself. Furthermore it may make cloning a whole machine easier but recreating a VM from scratch is subject to the same drawbacks as with the manual provisioning as discussed in section 2.3.1. But as with bare-metal machines one can use tools like Puppet or Chef to automate the provisioning of new machines or in the case of VMs shift to something like Vagrant[10] which takes the automation one step further and also automates the provisioning of the VM itself.

This however still does not reduce the resources consumed by running a full-blown operating system and virtualizing all its hardware although only a subset of this is really needed by the application.

2.3.4 Chroot, jails & containers

Then there are some more lightweight virtualization methods. Already since 1979 there exists the chroot program wrapping around the same-named system call on Unix and Unix-like operating systems [Byf14]. It **ch**anges the filesystem **root** of an application before

[8] `https://github.com/puppetlabs/puppet`
[9] `https://www.chef.io/chef/`
[10] `https://www.vagrantup.com/about.html`

starting it so that the filesystem root seen by the application is actually a subdirectory of the real filesystem of the host machine. Because of this the application then can access only files and directories contained in this subtree via normal filesystem system calls. This on the one hand isolates the application into this directory and can on the other hand provide the application with files in an expected path without colliding with paths of the host system. This is especially useful for example when the application uses a library which is (and maybe has to be) installed in another, incompatible version on the host at the same filesystem path. This isolation already allows packaging applications as an archive of a minimal root filesystem containing all and only the files, libraries and other dependencies needed by the application. Also only the application (and its dependencies) itself need to run within the chroot environment, so for everything else - which is especially the kernel - the host is reused.

Extending the isolation capabilities of the chroot call the concepts of jails (BSD, 2000), Solaris Zones (2005) and Linux Containers (LXC, 2008) emerged which added among others isolation of network, process and IPC namespaces [Sub14] which increases the overall isolation and therefore the reproducibility. A notable popularity boost to this technologies was provided since 2013 with the introduction of Docker, a management tool set for Linux containers aiming to help "developers and sysadmins to build, ship, and run distributed applications"[11] in a reproducible way, leveraging the analogy of shipping containers in which the containers have a standardized format allowing to be handled by any tool conforming to this standard, i.e. to be run on any host supporting the Docker engine. Beside easy provisioning of existing containers Docker also provides a build system to reproducibly rebuild containers based on a simple recipe.

2.4 Summary

So to conclude, what we have here is a Java-based web application developed for internal use of a department of about 50 people. Development takes place on local developer machines while the production, integration and bug tracking systems run on self-hosted Linux servers.

To this we now want to apply the principles of continuous * - meaning the common principles of continuous integration, continuous delivery and continuous deployment. These demand that integration/building/deployment should be done as often as possible and therefore be - for a reasonable practicability - as easy as possible to the developers and where applicable to the operations staff. This in turn relies on heavy utilization of automation based on reliable reproducibility. As we just saw, there are two main approaches to providing such reproducibility in the various execution environments: configuration management or some kind of virtualization. In either case this can be done manually or in an automated fashion. But doing it manually - which is in the case of configuration management manually verifying the configuration and changing it as needed and in case of virtualization manually setting up the VM/chroot/container - not only heavily contradicts the required easiness but due to its error-proneness also the required reproducibility, and at last, obviously, the principle of automation. Therefore only the automated patterns qualify for continuous *.

In order to pick an appropriate approach and technology for our case we'll examine in the next chapter the requirements which arise from the application of continuous *, the current technology stack and from the explicit requests of the developers.

[11] `https://www.docker.com/whatisdocker/`

3 Requirements

Before starting the construction of the pipeline the requirements assigned to it have to be examined so their achievement can be correctly evaluated at the end. In this section we will therefore go through the various requirements. First I will be determining the business requirements implicitly given by the existing technology stack but also by the paradigms of continuous *. Further on the requirements and expectations imposed on the pipeline by the developers and the management are collected in cooperation with the team. Subsequently the requirements will be classified as being functional or non-functional and prioritized as mandatory or optional requirements.

The end users themselves will not be involved further in the assessment at this point because the pipeline is mostly out of their scope.

3.1 Implicit requirements

The category of implicit requirements contains all requirements derived from the application of continuous * to the development process of planr, implicitly expected by the developers or given through business requirements. These were gathered by examining each of the mentioned origins and subsequently confirmed by the developer team as being mandatory to the intended deployment pipeline.

Integrate with the existing technology stack The first and an important origin of requirements is, of course, the existing technology stack which is already established and in use. Hence the pipeline should either integrate smoothly with the environment described in section 2.1.2 on page 3 or its specific advantages had to justify the replacement of components not fitting together. So in best case the pipeline should just add on to the existing stack, i.e. Git, Stash, Bamboo, production and integration machines running Linux.

Push-button releases This requirement derives directly from the principles of continuous * and continuous deployment in particular which demand that deploying new releases has to be doable effortlessly for the developers. Therefore deploying to production should require no more than "the push of a button" when the team decides that the new version is ready for release. Deploying to integration may even spare the button push since every successful build is inherently ready for release into integration. So this may be automated completely.

Production-like integration environment Additionally as required by continuous * this has to be reproducible, not only within the same deployment environment but also in between the various environments. To be able to (more or less) guarantee this, the differences between the various environments have to be very small. Not having the integration environment as production-like as possible inheres the risk that tests passing in the former could fail in the latter because of the differences.

Build and deployment to integration in <10min But not only the ease of deployment is implied by the continuous integration paradigms but also the speed. Even having a fully automated deployment pipeline is worth little if it takes hours or even days to deploy a single change because it doesn't provide the timely feedback to the developers about the success of their changes needed for the continuity of integration efforts. Jez Humble and David Farley recommend a timespan of a few minutes not significantly longer than about ten minutes in their book[HF11].

Support upgrades of the database Deploying new application releases is just one use case of the intended deployment pipeline - although the main use case. Since the database is seen as a separate service the other use case of the pipeline was appointed to be the deployment of new versions of the database system and therefore allowing to upgrade to a new MySQL release.

Enable deployment to all environments through the same pipeline Here again, to comply with the mantras of continuous * the difference between the environments have to be as small as possible also in the deployment process. Hence the deployment is then "proven to work" as continuously as the code, i.e. each successful deployment into integration reasserts that the next deployment to production will succeed as well.

3.2 Developer requirements

Additionally to the aforementioned implicit requirements the following ones were explicitly stated or repeated by the developer team. In the following first the mandatory then the optional requirements gathered this way are listed.

3.2.1 Mandatory requirements

Build and deployment to integration in <10min As already required by continuous integration the team also emphasized the necessity of only a short timespan between code check-in and feedback about the success, i.e. finished deployment into integration or failure, with a target maximum of five to ten minutes.

Dedicated instance per development branch on integration Furthermore the integration pipeline should support the current development model based on feature branches, enabling the developers to work on separate branches, one per feature/bug, but also testing them on integration systems. This means the pipeline should be able to have multiple separate instances of the application deployed in the integration environment(s) at the same time, which is currently a single server.

Semi-persistent database instances per development branch on integration Additionally each instance on the integration system should have its own copy of the database to operate on. Although a fresh database for each new instance was desired the team explicitly wished for this database to be persistent during the lifecycle of a development branch i.e. it should remain between deployments of new releases of the same branch.

Recent copy of production database on integration In order to test the application in the integration environment against some real-world data the database instances set up for each branch should be based on a recent copy of the production database.

Easy administration and cleanup of integration instances Lastly the administration/maintenance and cleanup of the various instances throughout and especially at the end of their lifecycle on integration should be easy and/or be done automatically.

Local testing environment can be setup fast and easy/supported cross-platform Starting the integration of new code already on the developer machines it should be possible to do local testing in a production/integration-like environment which in turn has to be easy to setup on local machines by the developers. This again requires some kind of cross-platform support in order to be deployable on at least Mac OS and Linux - which are currently used by the team locally - and Windows - so it may be used by future team members.

3.2.2 Optional requirements

While working out the requirements of the developer team listed above we discussed some requirements that were then given a low priority. For completeness these will be recorded here too but will receive only very few regard during further development of the pipeline.

Zero-downtime deployments One of these possible requirements was a zero-downtime strategy for deployments meaning that there should be no interruption of the service even during deployment of a new release. Regarding the integration environment a downtime affects the development team and maybe some test users only, for whom such a downtime is considered totally acceptable. Regarding the production environment a downtime affects the planning department which would be unable to continue working during the downtime and in worst case could lose some unsaved work in progress. Therefore a downtime should be carefully scheduled and communicated to the users beforehand. But since the user group and thereby their office hours is well known a new release can easily be scheduled outside these timeframes.

Horizontal scalability Associated with zero-downtime is the ability to scale horizontally i.e. deploy multiple instances of the application and run some kind of load balancing between them. This would contribute to overall performance and failure safety/service recovery particularly in production. But for now the need for this scaling isn't anticipated and it's unclear whether the application itself would support it yet.

Separate environment configuration from code Currently the configurations for the various environments are managed as environment specific configuration files within the code repository. To have a clean separation between code and the environments an implementation of the pipeline should be favored in which all environment specific configuration - in this case notably the configuration of the database connection - can be completely moved to the environment itself hence removing everything environment specific from the codebase. This is desirable because the configuration varies between deployment environments, the deployed code does (or at least should) not[Wig12].

3.3 Summary

Having gathered the various requirements I classified them in terms of the two aspects of being functional or non-functional - i.e. describing *what* should be possible versus *how* it is done (e.g. "it should be fast, easy and incredibly good looking") respectively - and being mandatory or optional. The result of which is table 3.1.

	functional	non-functional
mandatory	enable deployment to all environments through the same pipeline	integrate with the existing technology stack
	dedicated instance per development branch on integration	push-button releases
	semi-persistent database instances per development branch on integration	production-like integration environment
	recent copy of production database on integration	build and deployment to integration in <10min
	support upgrades of the database	easy administration and cleanup of integration instances
		local testing environment can be setup fast and easy/supported cross-platform
optional	zero-downtime deployments	separate environment configurations from code
	horizontal scalability	

Table 3.1: overview and classification of the requirements

4 Concept

Equipped with the foundations of continuous * and the requirements for the desired continuous deployment pipeline its conceptional architecture will be designed throughout this chapter. For this we will first examine the status quo and the current deployment process. Subsequently the concept of the new deployment pipeline is developed while trying to already satisfy the requirements best possible. So at the end of this chapter we'll get the final concept for the new deployment pipeline and a list of satisfied and yet unsatisfied requirements which then have to be taken care of in the implementation phase in chapter 5.

4.1 Status quo - analyzing the existing setup

So the first requirement I want to consider is the integration into the existing technology stack because for this we have to first determine the status quo of the current setup. And this is just the right time to do this because on this occasion we can also check which other requirements maybe are already fulfilled by the status quo and which are not. Based on this analysis we then can (hopefully) build upon the existing setup to comply with the remaining requirements. We'll perform this analysis along the steps of the pipeline a new feature has to take, from writing to deployment into production.

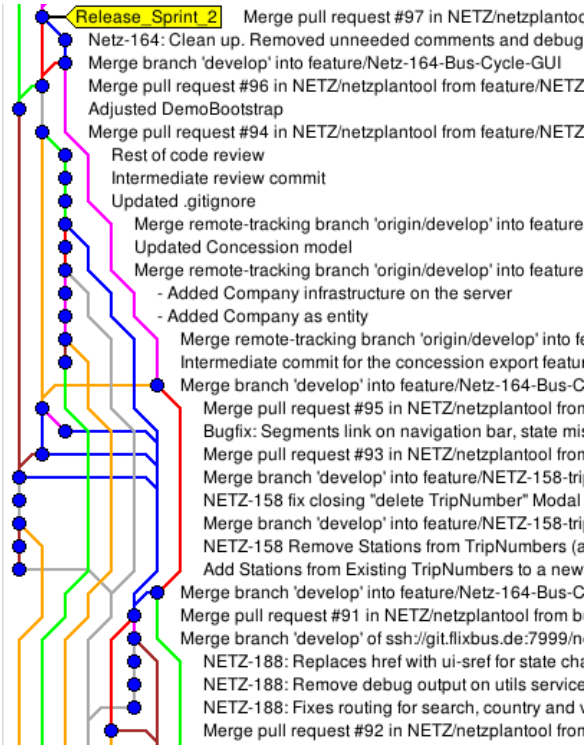

Figure 4.1: screenshot of the current branching scheme

4.1.1 Version control and branching scheme

Our pipeline begins on the local developer machines where a new feature is born. The actual coding of a new feature will be outside the scope of our pipeline which instead begins with the check-in of new code into the version control system - which is Git as we saw in chapter 2.1.2. The branching scheme currently used by the team dedicates a distinct branch to each new feature or bugfix while holding the current development mainline in a branch named "develop" and the version currently in production in a branch named "master". The names of the development branches are by convention constructed from the

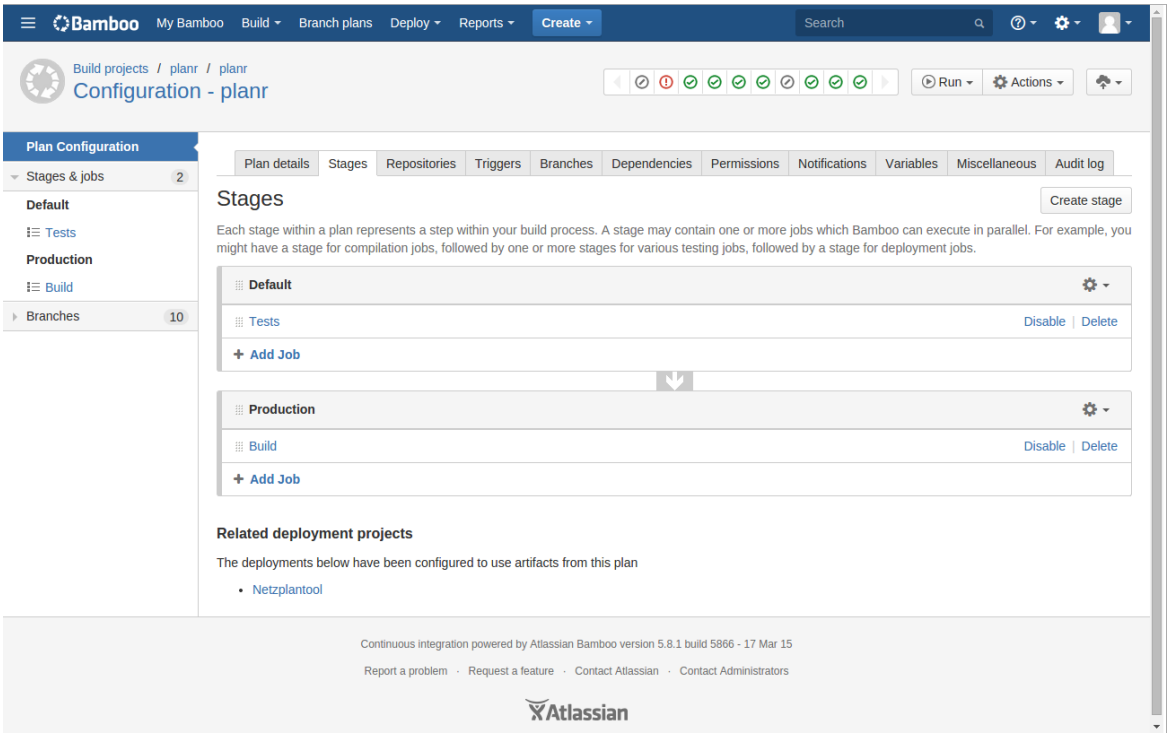

Figure 4.2: screenshot of the current build tasks in Bamboo

ID of the corresponding bug report/feature request in JIRA and a short, descriptive title, mostly also taken from JIRA.

Merging back of a branch to mainline, i.e. the “develop” branch, then requires the creation of a pull request and approval of another developer, both of which are done in Stash, the central Git repository. When a branch is long living and diverges greatly from mainline, “develop” is merged into that branch from time to time. However this workflow is implemented purely by convention and there are no technical measures to enforce it.

Therefore in the current workflow a dedicated mainline branch exists to which every feature and bugfix that has been (partly) completed is merged back. The lifespan of most feature branches amounts to only a few days but may sometimes also extend to a few weeks for more extensive features. This already complies with continuous * thus far.

4.1.2 Automated tests and building

The next step in the pipeline is the automated building and testing of the application. As said earlier build automation is provided by Gradle which takes care of compiling the source code in the correct order and packaging it to a single JAR file as the final build product. Testing is also automated through a corresponding Gradle task. Therefore automated building and testing are already in harmony with continuous *.

Furthermore the build server, Bamboo, is connected with the Stash repository and configured to checkout every push to Stash and automatically test and build it by executing the corresponding Gradle tasks. This means a developer only has to check-in his code changes and push them to Stash, from where it will be automatically checked out, tested and built by

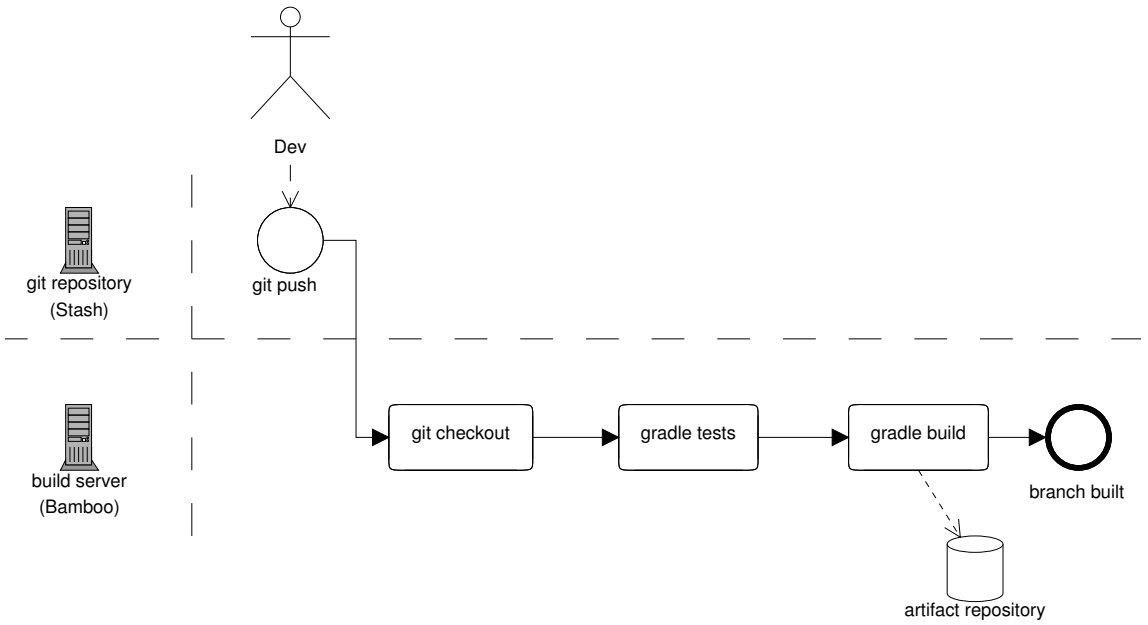

Figure 4.3: diagram of the current build workflow

Bamboo. The success of a build can be tracked in Bamboo and if a build fails the developer who triggered it will be informed by e-mail. The created build artifacts - the application JAR file, configuration files for the different environments and shell scripts for deployment (see next section) - are stored together with the logs and test results inside of Bamboo.

So this too already complies with continuous * because it is highly automated and therefore very easy, almost completely transparent to use for the developers, while still giving instant feedback on breaking changes.

4.1.3 Deployments

After a successful build the application is available for deployment. This can be done through a "deployment project" (see figure 4.4) configured in Bamboo which has to be triggered manually and is mostly identical for the integration and production environments. It then loads the build artifacts of the specified build to be released which are: the application JAR file, a common and an environment specific configuration file containing the credentials for database access etc. and a shell script, `planr.sh`, to manage the starting and stopping of the application. It then connects to the target server via SSH to stop the currently running instance, if any. This is done by executing `planr.sh stop` which makes a HTTP call to the application, asking it to shutdown gracefully, and killing it if it doesn't exit itself after a given time.

Next it uploads the artifacts of the new release to the target server one by one via SCP and finally starts up the new application by executing `planr.sh start $env`, with `$env` being the short name of the environment of which its specific configuration file should be loaded ("int" or "prod"). After starting the JAR file with Java the `planr.sh` script repeatedly calls a REST endpoint of the application via HTTP to see whether it started up successfully.

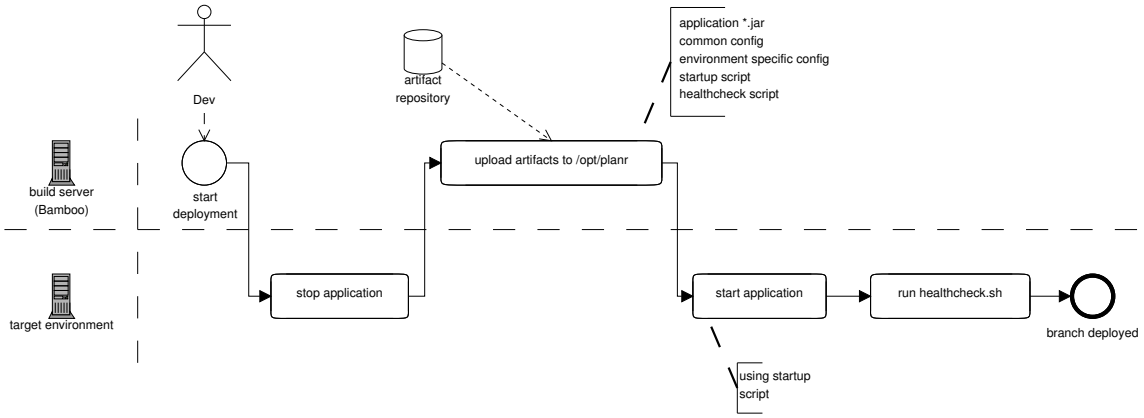

Figure 4.5: diagram of the current deployment workflow

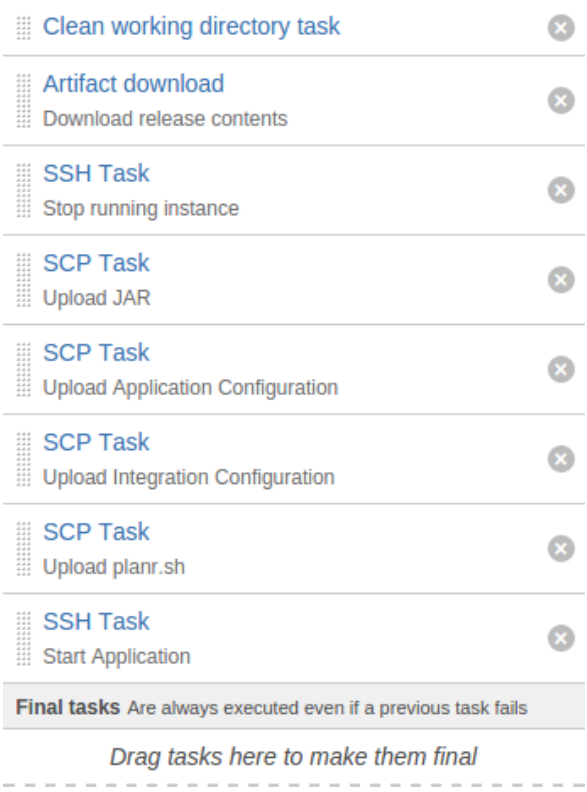

Figure 4.4: screenshot of the current deployment to integration in Bamboo

The script then exits either with an exit code of success, if it detects a successful start up of the application, or with an exit code indicating failure after a timeout. In case of a successful startup the application then listens on TCP port 8080 for incoming web requests and talks to a MySQL database which is configured in the environment specific configuration files. In the integration environment this is a MySQL instance running on the same server, in case of the production environment it is a MySQL instance running on a separate, dedicated host.

Here now we find the first violations of continuous * and the given requirements which we'll look at in detail in the section.

4.1.4 Matching the requirements

Up to the deployment step the pipeline already looks fine and mostly complies with continuous * and the requirements given in chapter 3. The deployment step thou infringes various of these requirements, the most noticeable being:

Production-like integration environment The first and most important one is the reproducibility and similarity of the deployment environments, i.e. integration and production. Currently these are set up manually and there is no explicit, detailed documentation about the steps taken to set them up. Also there is no configuration management in place to track past and future changes to the environments done manually. Therefore the integration environment can not be guaranteed to be production-like.

Although few, the application has some dependencies which have to be provided by the environment. These are a compatible Java runtime environment (JRE), a MySQL database and the correct directory layout to start the application in. Having different versions of the first two installed in the different environments may already cause a build of the application, which successfully passed the tests and build stage and ran successfully in the integration environment, to fail in the production environment due to the differences between the versions.

Local testing environment can be setup fast and easy/supported cross-platform Furthermore setting up local testing environments on the developer machines isn't trivial because it involves manually installing the correct JRE, installing and configuring a MySQL instance with the correct database, user/password, and permissions needed by the application. Despite in this case there indeed is a documentation in place for this it is nevertheless a manual and cumbersome task. Additionally in this case the JRE and MySQL may not only differ in version from the production environment but also in the underlying operating system, i.e. Mac OS X on most developer machines.

Dedicated instance per development branch on integration The next requirement not fulfilled by the current setup is the possibility to run multiple instances of the application in the integration environment in parallel. First of all this is simply by design of the current setup. But even with slight modifications this can't be achieved because multiple instances on the same host would heavily interfere with each other:
In the deployment Bamboo uploads the build artifacts always into the same folder on the target host, overwriting any previous files. To support multiple instances in parallel every instance had to be uploaded into a different folder. But then the startup of a second instance would still fail after all, trying to bind to the same TCP port (8080) as the first one. So every instance had to be running on another, available port.

Semi-persistent database instances per development branch on integration Yet even started in different directories and on different ports the instances would still operate on the same database and thus interfere with each other there.

Furthermore the current state of the remaining requirements is as follows:

Integrate with the existing technology stack With the current implementation just being the existing technology stack, this is tautologically true.

Push-button releases Releases are completely scripted and can be triggered in Bamboo, so this is true.

Build and deployment to integration in <10min Although deployment to integration has to be triggered manually the accumulated time consumed by building and deploying is still under 10 minutes.

Support upgrades of the database Upgrades of the database have to be performed manually since the management of the databases and their installed versions is outside the current pipeline and they were initially provisioned manually.

Enable deployment to all environments through the same pipeline Deployment is done in Bamboo where every environment - which currently are only production and integration - is configured, so this is true.

Recent copy of production database on integration Currently there is no way to get the current data from the production database into integration besides manual insertion.

Semi-persistent database instances per development branch on integration Due to the lack of support for multiple application instances on integration there is also just a single database which is persistent for all branches, so this is not given.

Easy administration and cleanup of integration instances This can be considered as satisfied because there is just a single integration instance which is overwritten every time a new version gets deployed (which kinda makes this requirement obsolete at the moment...)

Zero-downtime deployments These are not given because previous/running instances are stopped before uploading and starting the new release on integration as well as production. The startup of the new release than takes a perceptible time causing a clear downtime.

Horizontal scalability Currently there exists only a single production instance and automated provisioning of additional production instances can't be done easily due to the lack of reproducibility of the environments as mentioned above.

Separate environment configuration from code The configurations for the different environments are currently part of the codebase and therefore this requirement is not satisfied.

Recalling the overview in table 3.1 on page 12 and adding the current status to the fulfilled (✔) and unsatisfied (✗) requirements we get table 4.1.

	functional	non-functional
mandatory	enable deployment to all environments through the same pipeline ✔	integrate with the existing technology stack ✔
	dedicated instance per development branch on integration ✗	push-button releases ✔
	semi-persistent database instances per development branch on integration ✗	production-like integration environment ✗
	recent copy of production database on integration ✗	build and deployment to integration in <10min ✔
	support upgrades of the database ✗	easy administration and cleanup of integration instances ✔
		local testing environment can be setup fast and easy/supported cross-platform ✗
optional	zero-downtime deployments ✗	separate environment configurations from code ✗
	horizontal scalability ✗	

Table 4.1: fulfillment of the requirements by the current implementation

4.2 Drafting the new continuous deployment pipeline

So as we just saw the status quo already complies partly with the principles of continuous * and fulfills some of our requirements. Especially practicing continuous integration is fully supported by the current workflow and setup. Therefore we can build upon this.

In order to complete the realization of continuous * here we have to focus on the continuous delivery and deployment part of the pipeline which currently isn't complying. The main problem here is the insufficient reproducibility of the runtime environments and as a consequence the (possible) differences between the environments, especially between integration and production. Furthermore running multiple instances in the integration environment simultaneously isn't supported by the current implementation in any way.

Therefore the challenges we have to conquer first and for the most part are, making the runtime environments reliably reproducible and sufficiently isolating the application instances running in the integration environment from one another.

4.2.1 Making the environments reproducible

As discussed in section 2.3 there are two main approaches for reproducible environments: (automated) configuration management and virtualization.

Evaluating these two options for our specific case it came clear quickly that utilizing virtualization would be the better one. First this is due to the concern that even with an automated configuration management tool in place there still could be changes made to the servers manually which then were outside the scope of the configuration management. But second and actually more important is the requirement of isolation in order to run multiple instances in the integration environment since this in combination with a configuration management tool still would have additionally required the automated provisioning of a dedicated integration environment for each application instance. Because otherwise multiple instances running on the same machine would interfere and conflict with each other as discussed earlier in section 4.1.4.

Utilizing virtualization however already includes a certain isolation of the running application, i.e. running it inside a VM/chroot/container, and therefore allows running multiple instances on the same host machine more easily. Moreover the automatically provisioned host in the configuration management scenario would have been VMs themselves just like the existing hosts anyway.

Deciding to use virtualization we next had to draw the line between the build product and the environment that is defining what has to be included in the final build product and what has to be provided by the environment.

One possible answer to this is to keep the line where it is now: the final build product is the application jar file, configuration files and the control script, and the runtime environments are just virtualized versions of the current environments, i.e. a VM/chroot/container with the application dependencies - mainly the JRE - preinstalled. Deployment then would be just like in the current implementation by copying the artifacts into the - now virtualized - environment and starting it via SSH. But now additional instances of the environment could be provisioned by copying the base image of the virtualized environment and starting it up as a new instance.

The other approach is to make the virtualized environment itself the product of the build meaning the final build product is a VM/container/chroot image or archive file. In this

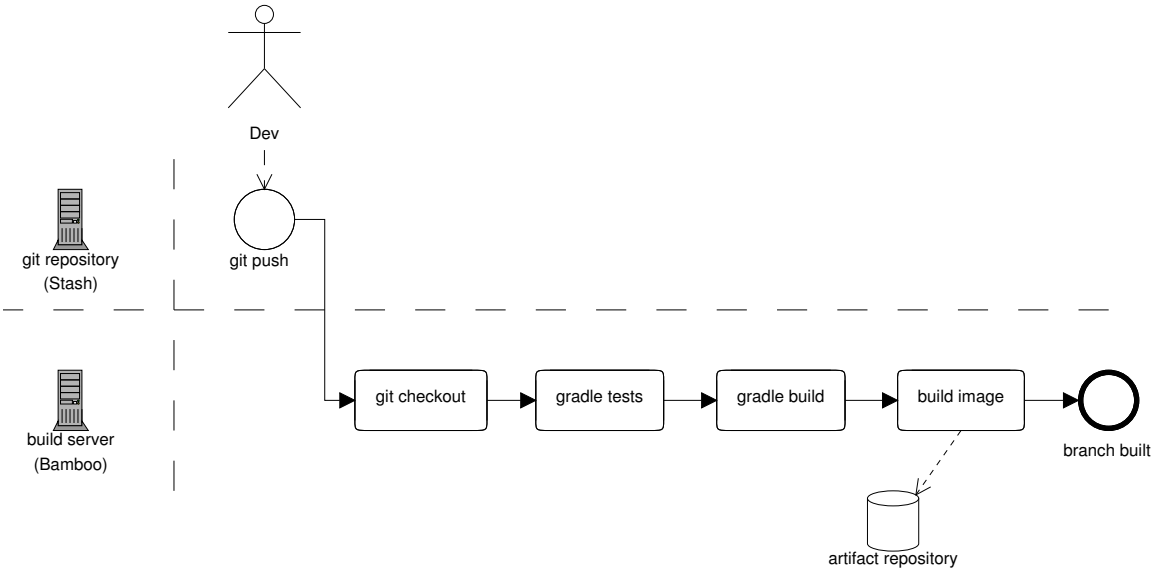

Figure 4.6: diagram of the build pipeline extended by the image build

case the deployment involves just uploading the image to the target host and starting it up. For this the target host has to provide nothing more than the runtime for the chosen virtualization solution, i.e. a virtualization tool like QEMU, VMware, VirtualBox and such like, the chroot syscall/binary or a container management like LXC or Docker.

In the end we chose the latter solution because it greatly reduced the requirements of the application/build product against the environment to only the virtualization runtime by including all other application specific dependencies in the final build product. Additionally this also allows using different dependencies in different builds, i.e. a newer version of the application could include a newer version of the JRE but still run in the same environment.

4.2.2 Enabling multiple simultaneous instances in one environment

Integrating this sort of virtualization conceptionally into the current implementation also is rather easy: you just have to add an additional step to the building process where the final image is created by packaging the build artifacts and their dependencies together, which is show in the updated figure 4.6.

As mentioned earlier this build product then just needs to be uploaded into the target environment and started there as shown in figure 4.7.

Thanks to the isolation inherent to the virtualization multiple instances of the application can now be deployed on the same system. But one question here is still unanswered: where to put the database? Because here again we have multiple options to choose from:

1. Setup a central MySQL instance every application instance communicates with.
2. Provision a dedicated MySQL instance per application instance
   a) by including it in the application image.
   b) as a separate virtualized instance.

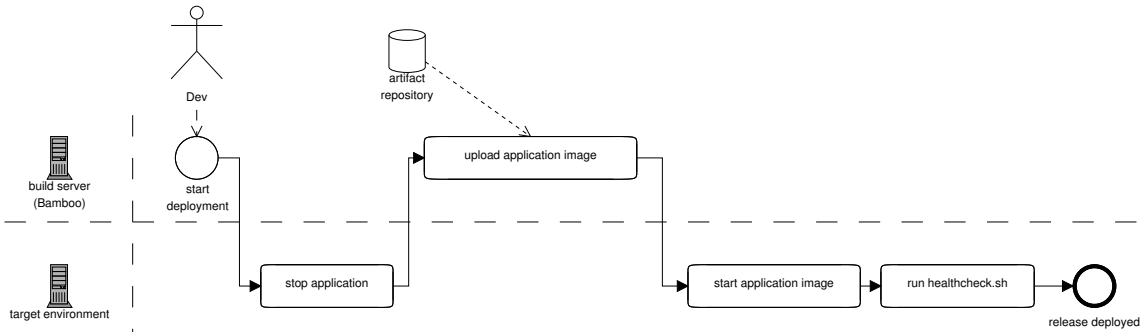

Figure 4.7: diagram of the deployment to production using virtualization

Considering the exact MySQL version and configuration an explicit dependency of the application the first option can be eliminated. This is because the version and configuration the application expects may change over time which would make running two application instances with conflicting dependencies on the database impossible.

With the second option variation a) seems to be the easier one at first glance because coupling the application and database this way always guarantees the database to be in the correct version and configuration the application expects. But at second glance this also brings in new problems when considering the requirement of having semi-persistent database instances per development branch on integration since this implies a certain decoupling of the deployed application release and the database it operates on. Including the database with the application would entail that each new build of the application would operate also on a new database. This would even apply to production deployments therefore resetting the database with every new release which is obviously undesirable. But instead there should be only a separate database instance per branch and not per build in integration and a single, persistent database in production.

Therefore variation b) is the necessary compromise between these two since it allows to have a dedicated MySQL instance for each running application instance while at the same time decoupling the lifecycle of the database from the application instance and therefore enabling the reuse of the database by another application instance of the same branch. The conceptional process of this is shown in figure 4.8. How the application and database instances have to be wired up so each application instance can talk to its own database - and only to that one - depends on the specific implementation and will thus be discussed later in section 5.1.5.

4.2.3 Building the database images

After we just clarified the integration of the database into the pipeline in the previous section we now need to determine when and how the database image is to be built, especially the one used in the integration environment. Since changes to the required database version or configuration are comparatively seldom not every application build needs to trigger a new build of the database image.

But with regards to the requirement that the database instances in the integration environment should provide a reasonably recent copy of the production data, new database images for use in integration should be build regularly. Furthermore these images should in-

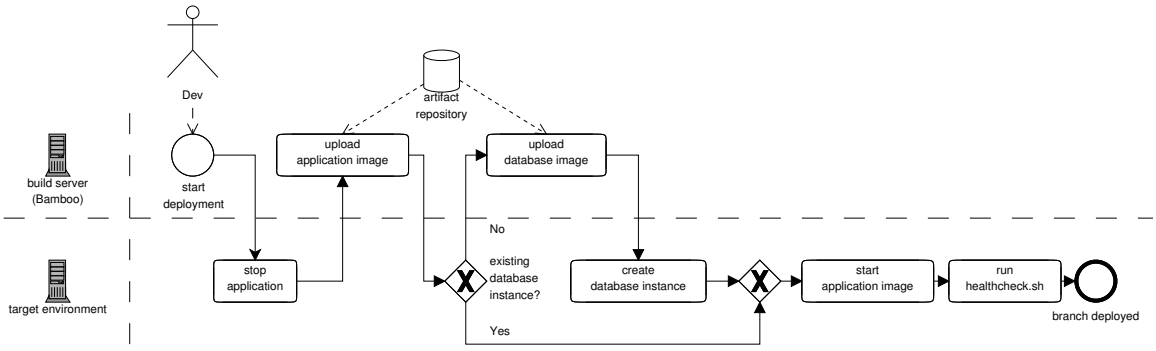

Figure 4.8: diagram of the deployment workflow for integration using virtualization and separate database images

clude a recent copy of production data while the database images used in production should include no data, but besides of that there should be no notable difference. Therefore the production database and the database instances on integration should be based on a common base image. The images used in integration should then be build upon this base image by loading a SQL dump of the production database into it and saving it as a new image including the inserted data like shown in figure 4.9.

In order to upgrade to an newer version of the database system the common base image has to be re-created including the new MySQL version. Creating a new database image for integration based on this new base image the application can be tested against the new MySQL version. When tested successfully the database upgrade can be deployed by uploading and starting the new base image on the production database host like shown in figure 4.10. To not lose the production data in this process the database files of the production instance have to be persistently stored somewhere outside the virtualized database instance, e.g. in a separate virtual hard drive or in a directory on the host system which can then be mounted into the VM/chroot/container filesystem. But since this depends on the implementation of this concept it will be further discussed in section 5.1.7.

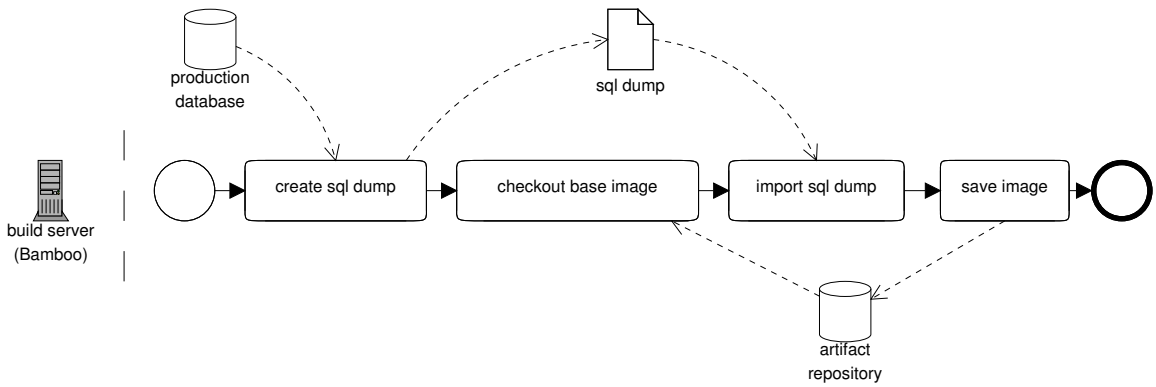

Figure 4.9: diagram of the build pipeline of the database images for integration

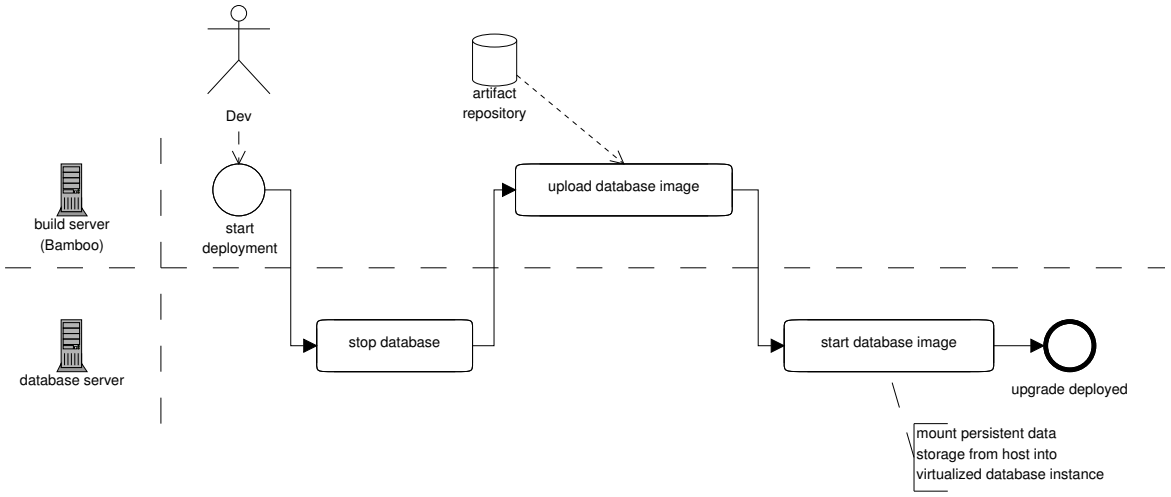

Figure 4.10: diagram of the upgrade workflow of the production database

4.2.4 Summary

So starting from the status quo only a few conceptional changes were necessary to advance towards a functioning continuous deployment pipeline. Utilizing virtualization only one additional was appended to the build process in figure 4.6 on page 20 now packaging the application into an image including all its dependencies designated to run in a virtualized environment. In the deployment procedure in figure 4.7 on page 21 there is even only the conceptional change that instead of uploading and starting the application file(s) now the application image is uploaded and started. This makes the creation of the database image for use in integration and the upgrade procedure of the production database the only conceptionally new processes introduced by our concept.

With this concept we now can again update the requirements in table 4.2 on the following page which should now be satisfied as long as they don't depend on the specific implementation. In that case they're taken care of in chapter 5 on page 25. The changes regarding the status quo in table 4.1 on page 18 are as follows:

Dedicated instance per development branch on integration By introducing virtualization we are now able to avoid the issues mentioned in section 4.1.4 regarding the deployment of multiple parallel instances to the same target system. The isolation provided by the virtualization allows us run multiple application instances at the same time on the same host.

Semi-persistent database instances per development branch on integration This applies also to the database instances allowing us to deploy multiple separate instances in the integration environment. Being slightly decoupled from the respective application instance a new application instance can reuse a database instance of a previous application instance as shown in figure 4.8.

Recent copy of production database on integration The creation of the images of the database instances used in the new deployment to integration is described in section 4.2.3

and includes the copying of the current production data into the integration images, satisfying this requirement.

Support upgrades of the database Upgrading the database is supported by the concept as described in section 4.2.3 through the creation of a new base image and deploying it or the images based on it to production and integration respectively.

Production-like integration environment Since most of the runtime environment and dependencies of the application will be included in the application image which does not change between the deployment environments the execution environment as see by the application will be the same in production and integration (and other environments if applicable).

While the fulfillment of the requirements marked with **"?"** in table 4.2 now depends on the specific implementation the status of the remaining requirements not mentioned here did not change any further.

	functional	non-functional
mandatory	enable deployment to all environments through the same pipeline ✔	integrate with the existing technology stack **?**
	dedicated instance per development branch on integration ✔	push-button releases **?**
	semi-persistent database instances per development branch on integration ✔	production-like integration environment ✔
	recent copy of production database on integration ✔	build and deployment to integration in <10min **?**
	support upgrades of the database ✔	easy administration and cleanup of integration instances **?**
		local testing environment can be setup fast and easy/supported cross-platform **?**
optional	zero-downtime deployments ✗	separate environment configurations from code ✗
	horizontal scalability ✗	

Table 4.2: fulfillment of the requirements by the new concept

5 Implementation

The concept of our continuous deployment pipeline developed in the last chapter will now be put into practice in this chapter. We start with the selection of one of the available technologies to implement the concept. Then the implementations of the various parts of the concept are described throughout this chapter one by one. At the we will again match the new pipeline against the requirements and also evaluate the improvements as well as any additional overhead introduced with the new implementation.

5.1 Chosen technology and its implementation

Now it's time to take a closer look at the technologies available for the implementation of our concept. So far we've always considered three different technology categories for our virtualization solution: virtual machines, chroot and Linux containers. Let's sort out the most appropriate of these three. For that I first want to recall where two application instances on the same environment would possibly conflict with each other in the current implementation as described in section 4.1.4 on page 17, which is the filesystem path the application and configuration files lie in, the TCP port the application binds to and the database it operates on.

For the last one we've already implemented an appropriate isolation into the concept of our new pipeline by providing a dedicated yet separately virtualized database instance to each virtualized application instance. So therefore this requirement should be realizable with any of the three solutions. Similarly the isolation of the filesystem paths is also given by all the three. On the contrary a sufficient isolation of the TCP port is only possible with two of the three solutions: VMs and containers. This is because chroot only isolates the filesystem the application can access but everything else is still shared with the host and therefore with other application instances virtualized with chroot. So binding to the same TCP port would still raise conflicts between the instances. For this reason chroot must be discarded as a possible implementation of our concept.

Since the remaining options now both satisfy our requirements for them, we have to compare their particular advantages and disadvantages. The main difference between VMs and containers is the extent to which the virtualization goes: VMs provide the guest systems with a complete virtual machine by virtualizing the various hardware components while containers only put the virtualized application into an isolated context, separating it from the execution environment of the other applications on the host but still sharing the same (Linux) kernel of the underlying host operating system. With this VMs provide a greater level of virtualization, allowing to even run a whole different operating system on top of the virtual hardware and thus running applications not executable directly on the host operating system itself. This, however, comes at an equally greater cost compared to containers which don't have the overhead of virtualizing all the hardware and having to launch a full blown (virtualized) operating system just to run a single application on top of it, but can instead launch the application directly after just setting up the respective isolated execution context

for it. Certainly though containers can only run applications compatible with the host operating system.

Considering these differences in the context of our application the advantages of the greater virtualization of VMs diminish mainly: our application neither needs access to any special (virtual) hardware nor does it need to run its own, full blown operating system since it already runs natively on Linux. The required isolation of the filesystem and the networking layer can just as well be provided by containers but without the costs of the additional overhead accompanying VMs.

Therefore the final choice is to utilize Linux containers as a lightweight virtualization of our application. The tool set used for this will be Docker since the Docker ecosystem is focused on the simple management and automation of applications in containers thus exactly what we need as we'll see in the following sections.

5.1.1 Docker 101

Prior to the description of the implementation it is necessary to give a basic introduction to Docker, its concepts and its vocabulary up front. If you're already familiar with Docker you may skip this section as it is mostly meant for those who never worked with Docker before.

So first of all: what exactly is Docker and what does it do? Docker is not a virtualization technology by itself but rather builds on the existing container technology of Linux. Instead its notable innovation is the tremendous simplification of the management of such containers by providing a simple commandline tool to start, stop, delete, interconnect and build such containers in just a few steps. Moreover Docker has built up a large community by now which provides many pre-built container images for many different applications. So Docker is a management engine and toolbox for handling Linux containers[1].

The Docker engine consists of several components and uses a sometimes uncommon vocabulary[2]:

Docker daemon The Docker daemon runs as a daemon process in background on the host machine and does all of the heavy lifting, i.e. manages the actual creation, starting, stopping and deletion of containers.

Docker client The user interacts with the Docker daemon only through the Docker client program which is the `docker` binary. It takes the commands entered by the user such as `docker run`, `docker start` and `docker stop`, sends them to the Docker daemon and returns the daemon's responses back to the user.

Container A container is the actual execution environment to run isolated applications in. A container usually has its own filesystem tree, IPC and process namespaces as well as network setup (see figure 5.1 below).

Image Each container is created from an image which is like a snapshot of a container containing an archive of its filesystem. These container templates are the exchange format for Docker containers. But in order to not have to transfer whole images - which can be several hundred megabytes up to gigabytes in size - every time an image changed Docker saves images in layers: every time a container or a new image is created

[1] `https://docs.docker.com/misc/faq/#what-does-docker-add-to-just-plain-lxc`
[2] see also `https://docs.docker.com/reference/glossary/`

based on another image Docker adds a new layer when saving the new image. This layer contains only the differences to its parent layer. That way common layers can be shared between images and only new layers have to be transfered saving bandwidth and storage.

Registry A registry stores Docker images which can be up- and downloaded to/from it with the `docker pull` and `docker push` subcommands. Serving as a central image storage there are public registries everyone may access but also private ones that can be installed in internal only networks behind a firewall and on your own servers.

Repository A repository is like the path to an image and describes where in a registry the image is stored. It is at the same time also used as the name given to an image and may also contain the hostname of the registry it is stored in. Its format is `[<registry.host.name>[:<port>]/][<path-to>/]<imagename>`.

Tag Each image can have a tag allowing to have multiple versions of the same image. It is added to the image name and when omitted defaults to "latest". So the complete format for an image name is `[<registry.host.name>[:<port>]/][<path-to>/]<imagename>[:<tag>]`.

Docker hub The Docker hub[3] is the official public registry of the Docker project and the default when no registry host is given in the image name. It's also the central image exchange point of the Docker community and thus already holds thousands of images created by the community and free to use.

Dockerfile A Dockerfile is like a recipe describing how to build an image and is therefore the primary component for the fully automated creation of new Docker images[4]. When running `docker build` Docker by default looks for a file simply named "Dockerfile" in the given directory, downloads if necessary the base image the new one should be build on, copies files into it and executes commands in it according to the directives in the Dockerfile and finally saves the result as a new image. While doing so Docker creates a new image layer for each directive of the Dockerfile executed allowing to reuse these layers and enable advanced caching of the builds. An example of a Dockerfile can be see in listing 5.1.

Volumes Since Docker containers have their own filesystem tree and are completely isolated from one another and from the host it is difficult to exchange and share files between them. In order to address this issue Docker uses so called volumes which are basically just filesystem directories on the host or inside other containers which are mounted into the filesystem of a container by Docker. So other containers and/or the host can access the same directory thus allowing to easily share files through them.

The Docker engine is written in the Go programming language and uses several web technologies and concepts: it is built around a client/server architecture where the communication between these two is done over a REST-based API and using JSON as the data format. By default the interface for the communication is a Unix socket but it can also be bound to a TCP socket allowing the Docker daemon and the Docker client to run on separate hosts.

[3] `https://hub.docker.com/`
[4] `https://docs.docker.com/reference/builder/`

Listing 5.1: a simple Dockerfile example

```
1  # base this image on the official ubuntu image from Docker hub
2  FROM ubuntu
3
4  # specify the maintainer of this image/Dockerfile to be added as metadata
5  MAINTAINER Roman Anasal <roman.anasal@flixbus.de>
6
7  # install additional packages to the image through the package manager
8  RUN apt-get update -y \
9   && apt-get install -y cowsay
10
11 # change the PATH environment variable to include /usr/games/
12 ENV PATH=/usr/games/:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
13
14 # set the default command to be executed in a new container
15 CMD cowsay "I␣like␣trains"
```

The isolation of applications running inside Docker containers is achieved using the Linux control groups (cgroups) and namespaces [FFRR14]. But for isolating the network Docker creates a virtual network by creating a bridge device on the host and adding a virtual interface to it for each Docker container as well as one for the host thus enabling network communication between the containers and with the host like shown in figure 5.1.

But Docker enables containers not only to communicate within this virtual network but also allows to "link" containers together. This means that when starting up a new container linked to another Docker adds environment variables to the started container pointing to the IP address and to exposed ports of the other container when available. Additionally it adds an entry to the `/etc/hosts` file inside the container so the alias of the linked container is resolved to its IP inside the started container.

When working with Docker one should also always bear in mind that Docker is still under heavy development thus regularly deprecating features and API functions while simultaneously adding new ones.

5.1.2 Preparing the hosts for Docker

In order to use Docker it has to be installed and correctly configured on every host that will be handling Docker containers. According to our concept these are the build server, the integration server, the production application server and production database server. Fortunately all three servers run the same operating system, a Debian Wheezy installation, so the installation procedure is the same for all of them. Installing and configuring Docker on the developer machines needs a little bit more effort and will be described later in section 5.1.8.

The quick and easy way to install Docker and also the recommended one by the Docker homepage is to simply execute

```
1  curl -sSL https://get.docker.com/ | sh
```

i.e. downloading and executing a shell script from the Docker homepage[5]. However, this being a production installation in an enterprise environment I didn't feel comfortable exe-

[5] Docker has since been added to the standard software repositories of many major Linux distributions. But using the install script is still recommended by the homepage for most older releases, see the guides listed at`https://docs.docker.com/installation/`

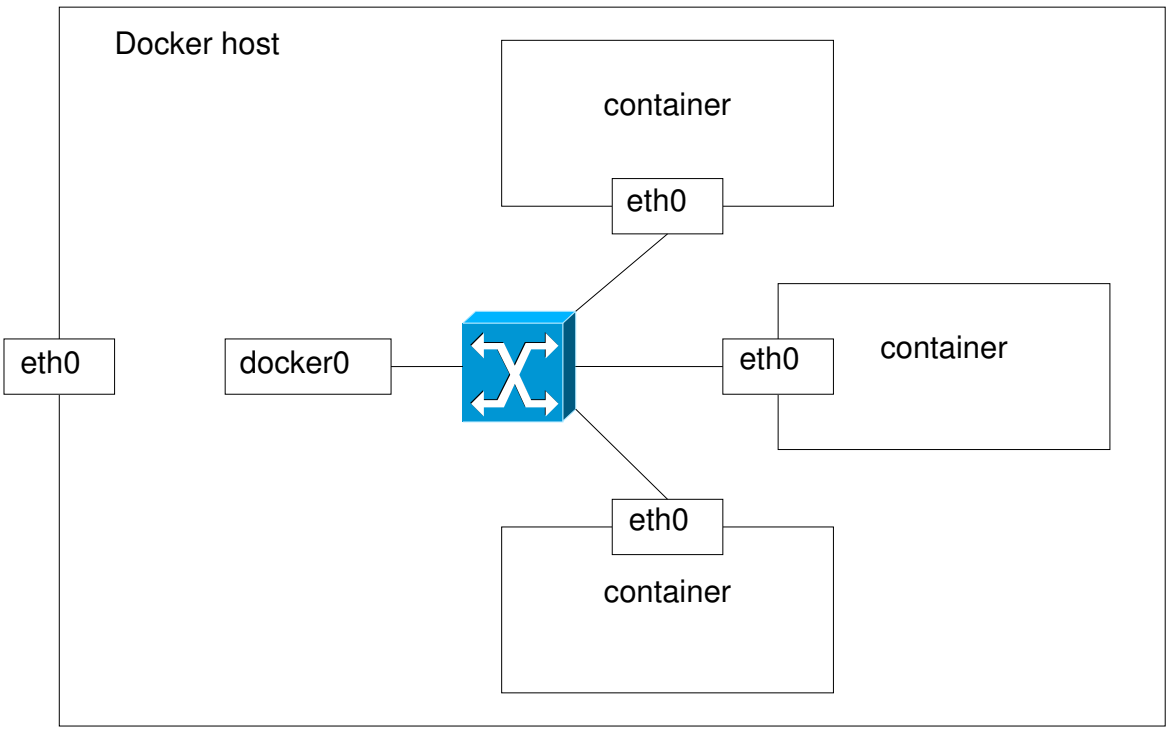

Figure 5.1: schema of Docker's virtual network

cuting a random script from the internet with root privileges on the servers. Instead I first took a look inside the script to figure out what it does and then decided to perform the steps manually since all it basically did, was try to detect the installed Linux distribution and version, add the correct software repository to the system and finally install Docker through the respective system package manager.

The most important dependency of Docker is a current Linux kernel. The one coming with Debian Wheezy by default is version 3.2 which is recent enough to run Docker. But as we discovered later on not all Docker features are supported by this kernel version, namely the `docker exec` command which we used for debugging (see section 6.1.3 for more on this). So we had to install the newer 3.16 kernel which is included in the official wheezy-backports repository:

Listing 5.2: upgrading to a newer kernel on Debian Wheezy

```
1 # enable the wheezy-backports repository
2 echo "deb␣http://http.debian.net/debian␣wheezy-backports␣main" >> \
      /etc/apt/sources.list
3
4 # update the package cache and install the new kernel
5 apt-get update
6 apt-get install -t wheezy-backports linux-image-amd64
7
8 # reboot to load the new kernel
9 reboot
```

Having installed a suitable kernel the next step is to add the Docker package repository and install Docker via `apt-get`:

Listing 5.3: installing Docker on Debian Wheezy

```
# add the Docker repository
echo "deb␣https://apt.dockerproject.org/repo␣debian-wheezy␣main" > \
    /etc/apt/sources.list.d/docker.list
# add the public key of the repository
apt-key adv --keyserver hkp://p80.pool.sks-keyservers.net:80 --recv-keys \
    58118E89F3A912897C070ADBF76221572C52609D

# install Docker
apt-get update
apt-get install docker-engine
```

After this the Docker daemon should be configured as a system service to start up at boot time and already be running. The success of the installation can be tested by executing the official hello-world image[6]:

Listing 5.4: Docker hello world example

```
> docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from hello-world

535020c3e8ad: Pull complete
af340544ed62: Pull complete
hello-world:latest: The image you are pulling has been verified. Important: image \
    verification is a tech preview feature and should not be relied on to provide \
    security.

Digest: sha256:d5fbd996e6562438f7ea5389d7da867fe58e04d581810e230df4cc073271ea52
Status: Downloaded newer image for hello-world:latest

Hello from Docker.
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
 3. The Docker daemon created a new container from that image which runs the \
     executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it to \
     your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker Hub account:
 https://hub.docker.com

For more examples and ideas, visit:
 https://docs.docker.com/userguide/
```

5.1.3 Building the application image with docker

Having successfully installed Docker on the servers we can now begin to integrate Docker into the deplyoment pipeline. The first place Docker appears in the pipeline is in the build step or more precisely: after the successful build of the application JAR file which now has to be packaged into a docker container image according to the concept in figure 4.6.

[6] https://hub.docker.com/_/hello-world/

For this we create a Dockerfile[7] which defines how the image is to be built and place it in the root directory of the project:

Listing 5.5: Dockerfile of the application image

```
1  FROM errordeveloper/oracle-jre
2  MAINTAINER Roman Anasal <roman.anasal@flixbus.de>
3
4  WORKDIR /opt/planr
5  CMD ["-jar", "planr.jar"]
6  EXPOSE 8080
7
8  COPY config/ /opt/planr/config
9  COPY build/libs/netzplantool-0.1.0.jar /opt/planr/planr.jar
10 COPY build/libs/newrelic.jar /opt/planr/newrelic.jar
```

First the image to build upon is specified in line 1, so in this case `errordeveloper/oracle-jre`[8] which is then downloaded from the official Docker hub. It is a minimalistic image containing only a busybox shell environment and Oracle's JRE in version 8 thus everything the application depends on while still being very lightweight.

The next line adds the maintainer of the image defined by the Dockerfile as metadata to the image. Line 4 causes the directory `/opt/planr/` within the image to be the working directory when a new container is started based on that image. Line 5 configures `-jar planr.jar` to be the default command to be executed on container startup. Together with `ENTRYPOINT [ "java"]` which is defined in the Dockerfile of the `errordeveloper/oracle-jre` base image this causes the complete effective command executed on startup to be `java -jar planr.jar` by default. Through line 6 then TCP port 8080 of the container will be exposed to the outside.

The last three lines then copy the artifacts previously build into the working directory of the image which are: the `config/` directory containing all the configuration files and the application file `planr.jar`. The additional `newrelic.jar` is a health monitoring tool used by the development team, but regarding our pipeline we can more or less ignore this since it has no particular impact on it.

Now with the Dockerfile in place everything needed to build the image is to execute `docker \ build .` from the project root directory. This is also the one additional step we identified in the concept to integrate in the automated build process in Bamboo. But just executing `docker build` would create an untagged image only identified by its hash id which is not quite human readable thus tagging the images with a readable name seems like a reasonable idea. Since the tag name/version should be different in every build the tag name is constructed using variables provided by Bamboo at build time. So in the end we added a new script task in the build stage of Bamboo executing the following command:

```
1  docker build -t "${bamboo.DOCKER_REGISTRY}/${bamboo.DOCKER_APP_IMAGE}:${bamboo.\
       planKey}-${bamboo.buildNumber}" .
```

Wherein the variables `${bamboo.DOCKER_REGISTRY}` (="registry.flixbus.com") and `${bamboo.DOCKER_APP_IMAGE}` (="planr/planr") are statically defined by us in the configuration of the build plan while `${bamboo.planKey}` and `${bamboo.buildNumber}` are dynamically provided by Bamboo and contain the planKey, a short key name for the current build, in this case "NETZ-NET", and the buildNumber, an automatically incremented build counter. So this results in a unique image name for each build in the form of e.g. `registry.flixbus.com/planr/planr:NETZ-NET-177`.

[7] `https://docs.docker.com/reference/builder/`
[8] `https://hub.docker.com/r/errordeveloper/oracle-jre/`

The tag name being at the same time its "repository" in the vocabulary of Docker contains also the information where to up- and download this image to/from with `docker pull` and `docker push`, i.e. the repository "planr/planr" on the registry server "registry.flixbus.com". How and why we set this up will be explained in section 5.1.4.

But since the tag of the image is just a name which can be changed easily by re-tagging the image we wanted to include some more metadata in the image itself which then could not get lost accidentally like the tag name. The best way to do this we found to be adding the metadata as environment variables to the image since these can easily be extracted from the image as well as from running containers with the `docker inspect` subcommand. Sadly the current syntax of Dockerfiles doesn't enable passing through environment variables at the runtime of `docker build` which is why the Dockerfile for that had to be generated dynamically containing the variables provided by Bamboo in the build stage. Luckily it is possible to to pass a "Dockerfile" on stdin to the `docker build` command so we could change the build task in Bamboo to the following:

Listing 5.6: building the application image in Bamboo with Docker

```
1  # abort script if any command fails
2  set -e
3
4  docker build -t "${bamboo.DOCKER_REGISTRY}/${bamboo.DOCKER_APP_IMAGE}:${bamboo.\
       planKey}-${bamboo.buildNumber}-interim" .
5
6  # add some env vars to the image
7  docker build -t "${bamboo.DOCKER_REGISTRY}/${bamboo.DOCKER_APP_IMAGE}:${bamboo.\
       planKey}-${bamboo.buildNumber}" - <<-EOT
8      FROM ${bamboo.DOCKER_REGISTRY}/${bamboo.DOCKER_APP_IMAGE}:${bamboo.planKey}-${\
           bamboo.buildNumber}-interim
9
10     ENV BAMBOO_BRANCH ${bamboo.planRepository.branch}
11     ENV BAMBOO_COMMIT ${bamboo.planRepository.revision}
12     ENV BAMBOO_RESULTS_URL ${bamboo.resultsUrl}
13     ENV BAMBOO_PLAN  ${bamboo.planKey}
14     ENV BAMBOO_BUILD ${bamboo.buildNumber}
15 EOT
```

So we basically first execute the same `docker build` command as before using the Dockerfile in the project root but this time tagging the image as interim. Then another `docker build` is executed and passing it a dynamically generated Dockerfile on stdin which generates an image based on the interim one (line 8) and adds the various Bamboo variables as environment variables `BAMBOO_*` in lines 10 to 14 to the final image[9]. We will make further use of these variables later in section 5.1.6.

5.1.4 Setting up a private Docker registry

After a new application image is built it has to be stored in the artifact repository so it can be used later on, for example in the deployments defined. Bamboo provides native support for acting as said artifact repository by saving the build products, defined to be artifacts in the configuration of the build plan and providing them to other build tasks or as downloads to the user. But Docker does not create a simple image file stored in the working directory

[9] Note that after we created this workaround to store metadata in the image Docker introduced so called "labels" for this purpose (see [Fra15] and `https://docs.docker.com/userguide/labels-custom-metadata/`) which should be used instead with newer Docker versions.

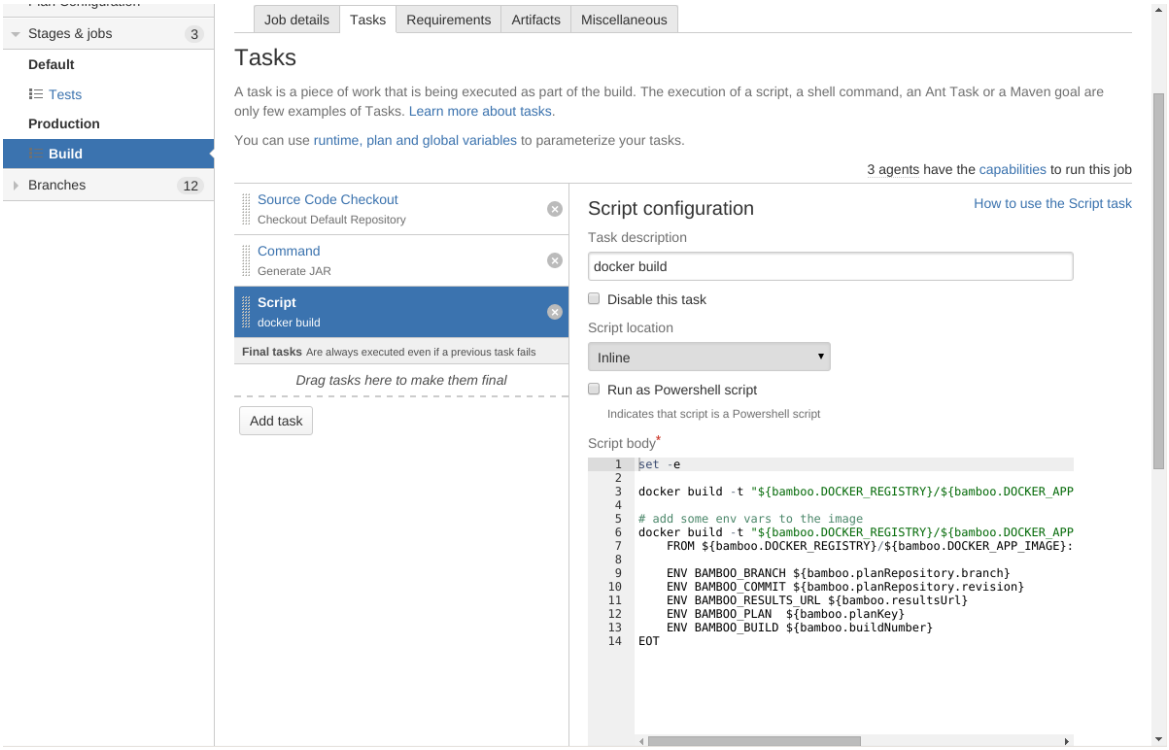

Figure 5.2: screenshot of the Docker build step in in Bamboo

but instead the images are stored by the Docker daemon layer by layer in its own backend (see section 5.1.1).

Now though it would be possible to export the images into a file using `docker save` this would export all layers of the image including the base image shared between all builds of the application image. This would not only consume more space on the hard drive but also bandwidth when deploying the images into the integration and production environments since not only new image layers containing the application file but also the layers of the base image have to be transfered even when already present at the target system. But bringing to mind that the layering concept of Docker images was specifically designed to reduce exactly this resource consumption we wanted to make use of it for a better performance, i.e. use a Docker registry as artifact repository and transfer images to and from it with `docker pull` and `docker push` which would transfer only the missing layers.

For this we had three options: either use the publicly hosted Docker hub, use an on-premise trusted registry or set up a private registry ourself. The first option was discarded quickly because it meant uploading the images containing a crucial business application to a server outside of our control and therefore having to trust the operator and infrastructure of the hub. But we rather wanted to keep this data within the internal network and on our own servers. For this purchasing the on-site installation of a trusted registry was possible. This is basically an on-premise version of the public Docker hub but also associated to a support agreement and therefore additional costs. But since we needed the registry - for now - only for storing and transferring the images between the systems and none of the further features provided by a trusted registry, setting up a simple registry ourselves was the way to go.

5.1.4.1 Installing the registry

Luckily there not only exists an open source version of a simple Docker registry[10] but it's even available as a pre-built Docker image[11] so the first setup can be as easy as one simple command:

```
1 docker run -p 5000:5000 registry
```

This starts a new container in the foreground from the registry image - after downloading it from the Docker hub if necessary - and binds the TCP port 5000 of the container to the host's port 5000, i.e. making it available on that port to the outside. To test it we can now already push our first image to it from the host itself:

Listing 5.7: retagging and uploading a local image to a private registry

```
1  # retag the registry image (or any other) to add the registry address \
       "localhost:5000"
2  > docker tag registry localhost:5000/registry
3
4  # now push the image to our registry
5  > docker push localhost:5000/registry
6  The push refers to a repository [localhost:5000/registry] (len: 1)
7  Sending image list
8  Pushing repository localhost:5000/registry (1 tags)
9  706766fe1019: Image successfully pushed
10 a62a42e77c9c: Image successfully pushed
11 2c014f14d3d9: Image successfully pushed
12 b7cf8f0d9e82: Image successfully pushed
13 d0b170ebeeab: Image successfully pushed
14 c8b8061f16bd: Image successfully pushed
15 4a66d663c62d: Image successfully pushed
16 30e37d2bb8f7: Image successfully pushed
17 56291d0100f7: Image successfully pushed
18 0671ae25286f: Image successfully pushed
19 cb1abf9cc100: Image successfully pushed
20 8599929d3e92: Image successfully pushed
21 1c29ed6cac96: Image successfully pushed
22 11963ef38b8f: Image successfully pushed
23 Pushing tag for rev [11963ef38b8f] on \
       {http://localhost:5000/v1/repositories/registry/tags/latest}
```

Without further modification though any uploaded images would be stored inside the container and therefore get lost with deletion of the container. To enable a permanent storage of the images a data volume has to be mounted into the container where the registry data can survive. This is easily achieved by adding the respective start parameter to mount a directory of the host filesystem, in this case `/opt/docker/registry/data/`, into the container and setting the environment variable `SETTINGS_FLAVOR=local` to configure the registry to use the local filesystem (of the container) as storage backend[12]:

```
1 docker run -p 5000:5000 -e SETTINGS_FLAVOR=local -v \
      /opt/docker/registry/data/:/tmp/registry/ registry
```

But since we needed our own registry in order to transfer images between *different* hosts it had to be available under another hostname than "localhost". For this we had to set a corresponding DNS record for the hostname we chose, registry.flixbus.com, and let it point

[10] `https://github.com/docker/docker-registry`

[11] `https://hub.docker.com/_/registry/`

[12] The registry supports also various cloud storage systems such as Amazon's S3 or Google's cloud storage as alternative backends

to the server running the registry. We decided to run the registry on the same machine as Bamboo thus keeping those two tightly together instead of requesting another VM to be set up to exclusively run the registry on. But this setup does not rely in any way on the registry and Bamboo being on the same machine and actually the registry can be moved easily by copying and starting it on a new host and updating the DNS record accordingly anytime. So we just requested the DNS admin to create the respective DNS record, which he did.

5.1.4.2 Enabling and securing remote access to the registry

Making the registry accessible from other hosts caused two problems. For one the simple registry itself did not provide any access control whatsoever whereby anyone with access to the exposed port of the registry, which serves standard HTTP requests, could also access and even manipulate the images in the registry. Even limited to the internal network only this was too much of a security loophole to us so we wanted to have some kind of simple authentication restricting the access.

Second, the registry is only capable of unencrypted HTTP while the Docker daemon - which acts as the HTTP client when up- or downloading images - by default expects connections to a remote registry to be encrypted over HTTPS and aborts communication with an error message otherwise. Although there are possible start parameters to the Docker daemon to allow insecure connections (`--insecure-registry`) we were not willing to activate these in production use because it would weaken the security which was intended to be there for good reasons.

Fortunately both problems could be solved by the same measure: moving the registry behind a reverse HTTP proxy which adds the HTTPS encryption and basic authentication to the connection between itself and the client, i.e. the Docker daemon. That way only connection attempts with valid authentication and authorization would be forwarded by the proxy to the registry, stripping away the encryption on the connection between the proxy and the registry.

We designated the nginx HTTP server[13] as the reverse proxy for our registry since there is already an instance of it running on the same host in front of Bamboo (funnily for almost the same reason - adding HTTPS to the forwarded connections) and because we will be using nginx also later on as reverse proxy for our applications as described in section 5.1.6. With the Docker daemon as well as nginx supporting standard HTTP basic authentication the nginx configuration shown 5.8 in listing was enough[14].

With this configuration the nginx instance running natively on the machine hosting the registry as Docker container listens on port 443 of the host for incoming HTTPS connections and forwards them via HTTP to port 5000 on localhost which in turn is port-forwarded to the container's port 5000 by the Docker daemon. Before doing so however nginx verifies that the client provided valid credentials via basic authentication which exist in the file `/opt/docker/registry/htpasswd` of the host machine. This file acts as local user database and can be edited with the `htpasswd` tool[15], e.g to add an user named "planr":

[13] `http://nginx.org/en/`
[14] excluding the further configurations for the reverse proxy to Bamboo, since it's not relevant here
[15] On Debian based distributions the `htpasswd` command is provided by the package `apache2-utils`.

Listing 5.8: configuration file for nginx acting as reverse proxy for the registry container

```
server {
        listen 443;

        ssl on;
        ssl_certificate /etc/nginx/certs/registry.crt;
        ssl_certificate_key /etc/nginx/certs/registry.key;
        client_max_body_size 0; # disable any limits to avoid HTTP 413 for large \
            image uploads

        # required to avoid HTTP 411: see Issue #1486 \
            (https://github.com/docker/docker/issues/1486)
        chunked_transfer_encoding on;

        # pass request to the registry on port 5000 of localhost
        # enabling basic authentication
        location / {
                auth_basic                          "Restricted";
                auth_basic_user_file                /opt/docker/registry/htpasswd;
                proxy_pass                          http://localhost:5000;
                proxy_set_header  Host              $http_host;   # required for \
                    docker client's sake
                proxy_set_header  X-Real-IP         $remote_addr; # pass on real \
                    client's IP
                proxy_set_header  Authorization     ""; # see \
                    https://github.com/dotcloud/local-registry/issues/170
                proxy_read_timeout                  900;
        }

        # disbale basic authentication for URIs /_ping and /v1/_ping
        location /_ping {
                auth_basic off;
                proxy_pass                          http://localhost:5000;
                proxy_set_header  Host              $http_host;   # required for \
                    docker client's sake
                proxy_set_header  X-Real-IP         $remote_addr; # pass on real \
                    client's IP
                proxy_set_header  Authorization     ""; # see \
                    https://github.com/dotcloud/local-registry/issues/170
                proxy_read_timeout                  900;
        }
        location /v1/_ping {
                auth_basic off;
                proxy_pass                          http://localhost:5000;
                proxy_set_header  Host              $http_host;   # required for \
                    docker client's sake
                proxy_set_header  X-Real-IP         $remote_addr; # pass on real \
                    client's IP
                proxy_set_header  Authorization     ""; # see \
                    https://github.com/dotcloud/local-registry/issues/170
                proxy_read_timeout                  900;
        }
}
```

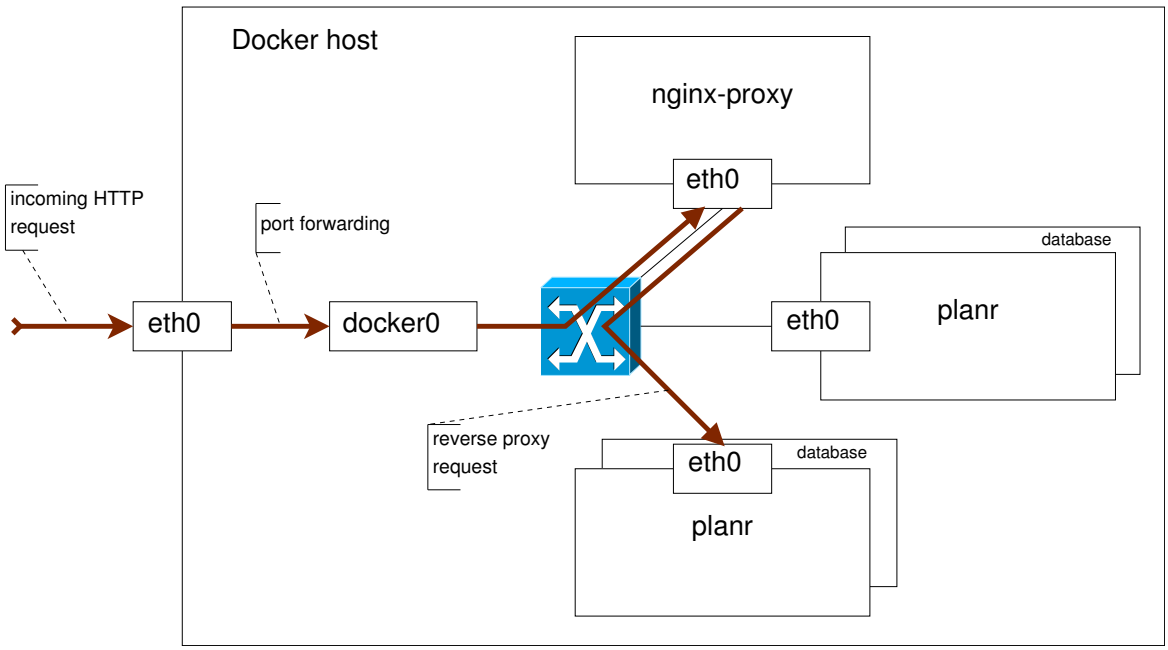

Figure 5.3: the connection flow through nginx-proxy

```
> htpasswd /opt/docker/registry/htpasswd planr
New password:
Re-type new password:
Adding password for user planr
```

The certificate and corresponding private key used for the HTTPS encryption are stored in `/etc/nginx/certs/`. Eventually we used a certificate properly signed by a commercial certificate authority and issued for "registry.fixbus.com" so this can now be successfully validated by clients. Prior to that we used a self-signed certificate though, which is described further in section 5.1.4.3.

Having set up a reverse proxy for our registry it now is no longer necessary to forward port 5000 of the host entirely to the container. On the contrary this would even allow to bypass the proxy by directly connecting to the HTTP port 5000 of the host instead of the HTTPS port 443. Therefore we need to modify the start command to bind only to the loopback address 127.0.0.1 (`-p 127.0.0.1:5000:5000`). At the same time we now want to start the registry as daemon process (`-d`) and that it be automatically restarted should the host reboot which is done by another start parameter (`--restart=always`). So the final command to start our registry is

```
docker run -d --restart=always -p 127.0.0.1:5000:5000  -e SETTINGS_FLAVOR=local -v \
    /opt/docker/registry/data/:/tmp/registry/ registry
```

Finally to use our new and secure private registry we need to initially execute `docker login` on every host that should be able to connect to it. This will ask the user to enter the credentials and save them locally after testing the connection to the registry with them.

```
# using the user "planr" created previously
> docker login --username=planr registry.flixbus.com
Password:
Email:
WARNING: login credentials saved in /home/roman/.dockercfg.
Account created. Please see the documentation of the registry \
    https://registry.flixbus.com/v1/ for instructions how to activate it.
```

Note that the e-mail may be left blank since its not really used anywhere.

5.1.4.3 Using self-signed certificates

As mentioned in the previous section we first used self-signed certificates for HTTPS with nginx since requesting a certificate properly signed by a commercial certificate authority (CA) is associated with additional efforts and costs. Nevertheless I'd like to document the steps taken for that and the caveats here because using a self-signed certificate is still a reasonable thing to do while in the development phase.

The first step is to create our own "CA" which will be signing the certificate for our registry which we will be creating in the next step. Since "create our own CA" eventually just means creating a private/public key pair basically the same way we will create the actual certificate we can do both using `openssl`[16]:

Listing 5.9: creating a self-signed certificate with `openssl`

```
# generate a new private key for our CA and output it in file ca.key
> openssl genrsa -out ca.key 2048
Generating RSA private key, 2048 bit long modulus
.....................................................+++
..............+++
e is 65537 (0x10001)

# create a certificate for our CA valid for 1000 days from key ca.key and output it \
    in file ca.crt
> openssl req -x509 -new -key ca.key -days 1000 -out ca.crt
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:DE
State or Province Name (full name) [Some-State]:Bavaria
Locality Name (eg, city) []:Munich
Organization Name (eg, company) [Internet Widgits Pty Ltd]:FlixBus GmbH
Organizational Unit Name (eg, section) []:Java Dev
Common Name (e.g. server FQDN or YOUR name) []:Docker CA
Email Address []:


# generate a new private key for the registry and output it in file registry.key
> openssl genrsa -out registry.key 2048
Generating RSA private key, 2048 bit long modulus
..............+++
...........+++
e is 65537 (0x10001)

```

[16] Please note that the basics of `openssl` as well as the concepts of a public key infrastructure needed for proper creation of trusted certificates are far outside the scope of this paper. For a more advanced understanding of these concepts please refer to the according literature.

```
# create a certificate signing request (CSR) for the registry in file registry.csr
> openssl req -new -key registry.key -out registry.csr
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:DE
State or Province Name (full name) [Some-State]:Bavaria
Locality Name (eg, city) []:Munich
Organization Name (eg, company) [Internet Widgits Pty Ltd]:FlixBus GmbH
Organizational Unit Name (eg, section) []:Java Dev
Common Name (e.g. server FQDN or YOUR name) []:registry.flixbus.com
Email Address []:
Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:

# sign the csr for the registry with our CA and output the final certificate in \
    registry.crt
> openssl x509 -req -in registry.csr -CA ca.crt -CAkey ca.key -CAcreateserial -out \
    registry.crt -days 1000
Signature ok
subject=/C=DE/ST=Bavaria/L=Munich/O=FlixBus GmbH/OU=Java Dev/CN=registry.flixbus.com
Getting CA Private Key


# list of files created
> ls
ca.crt        # the certificate of the CA
ca.key        # the private key of the CA
ca.srl        # the serial file of the CA containing the serial numbers of all \
    certificates signed by this CA
registry.crt # the final certificate of the registry signed by the CA
registry.csr # the certificate signing request of the registry
registry.key # the private key of the registry
```

Actually when purchasing a certificate signed by a commercial CA one would just skip the first two `openssl` commands and submit the CSR created by the forth `openssl` command to the CA which in turn basically just performs the signing analogous to the last `openssl` command.

In `/etc/nginx/certs/` of the registry host machine we can now place the certificate and private key for our registry we just created. But having this certificate signed with a CA certificate we created ourself it is not automatically trusted by the Docker daemon when trying to connect. In order to persuade the Docker daemon to trust our certificate - or more specifically the CA which signed it - we have to place the `ca.crt` in `/etc/docker/certs.d/registry.flixbus.com/ca.crt` of the machine running the Docker daemon we want to enable to connect to the registry, i.e. on the planr production, integration and database host, the Bamboo host as well as every developer machine.

It is important to keep the difference between the Docker daemon and Docker client in mind here because this can cause some confusion when these two are not running on the same host. This is especially the case when using docker-machine or boot2docker (see also section 5.1.8) which are specifically designed to provide a seamless local integration of a remote Docker host. In that cases the host where the `docker run` command is executed is not the one the actual container is started on.

Instead the locally executed Docker command acts as a client and connects to the remote Docker daemon through a (network) socket sending it the commands over HTTP to actually start the container on the host the Docker daemon is running on. Respectively all required Docker images have to be present on the host running the Docker daemon which therefore has to be able to connect to and download them from the registry. That means the Docker daemon has to find our custom CA certificate under `/etc/docker/certs.d/registry.flixbus.com/ca.crt` on the machine itself is running on and not on the (local) machine the Docker client command is executed.

After having our self-signed certificate generated and installed on the server (nginx) and the (Docker) clients the next step is technically the same as before: login to the registry on the clients with `docker login` and pull or push images with `docker pull`/`docker push`. Sadly we discovered that the `docker login` command didn't completely respect the provided, custom `ca.crt` and therefore failed to login. But since pulling and pushing worked when disabling authentication it seemed that only the initial login was failing which we could work around by logging in "manually":

As we saw on page 38 after successfully verifying the credentials `docker login` saves them into the file `~/.dockercfg` in the user's home directory which then looks something like this:

Listing 5.10: contents of `~/.dockercfg`

```
{
        "registry.flixbus.com": {
                "auth": "cGxhbnI6czNjUmVU",
                "email": ""
        }
}
```

Knowing the technologies and data formats Docker uses it was an easy guess that this is an JSON encoded object with the keys being the registry host (and optional port) and values being objects again with the attributes `auth` and `email`. The `email` is of no further interest for us since it's nowhere really used and can be left blank like said earlier. So the `auth` attribute has to contain the credentials in some encoded form. A quick guess showed it to be the base64-encoded concatenation of the username and password delimited by a colon[17]:

```
> echo "cGxhbnI6czNjUmVU" | base64 -d
planr:s3cReT
```

Equipped with this knowledge we could write our own version of `docker login` as a little shell script which is shown in listing 5.11.

5.1.5 Setting up the deployment with Docker

Finally having all the infrastructure in place and the creation of container images already integrated into the build pipeline we can now begin to automate the deployment of newly built container images to integration as well as to production. Looking back at figure 4.8 on page 22 the conceptional steps still missing in our implementation regarding the deployment to integration are: uploading the application image and a database image to integration,

[17] see also `https://coreos.com/os/docs/latest/registry-authentication.html`

Listing 5.11: docker-login.sh: workaround for manually performing docker login

```
1  #! /bin/bash
2
3  # usage: docker-login.sh > ~/.dockercfg
4
5  # read required variables from user input
6  read -p "Host:␣" hostname
7  read -p "Username:␣" username
8  read -p "Password:␣" -s password; echo # add a new line after password input since \
       it was suppressed by -s
9  read -p "E-Mail:␣" email
10
11 # generate base64 encoded auth token
12 token="$(echo␣-n␣"$username:$password"␣|␣base64)"
13
14 # output the generated json object
15 cat <<EOT
16 {
17         "$hostname": {
18                 "auth": "$token",
19                 "email": "$email"
20         }
21 }
22 EOT
```

setting up and starting a new database instance per branch if required, starting up the application image and verifying its successful startup.

In order to have only one application and one database instance running per development branch we decided to name the containers based on their corresponding branch. For this the name is constructed from the short name of the branch, i.e. the ID of the respective JIRA bug, the hostname of the integration system (`planr-int.flixbus.com`) and in case of the database containers with the prefix "db", which results in e.g. `netz-314.planr-int.flixbus.com` and `db.netz-314.planr-int.flixbus.com` respectively. Since the names of running (and stopped but not yet deleted) containers have to be unique on the Docker host this ensures that only one database/application instance per branch is running. Further more it allows us to replace a running application container with a newer image by name but at the same time retaining the associated database instance. Naming the containers in hostname format is by purpose because these names will be further used as such in section 5.1.6.

The deployment to production is analogous to this but leaving aside the database as shown in figure 4.7 on page 21 and using the hostname of the production host as name.

5.1.5.1 Automating deployment to production

In our final implementation most of the deployment is done by a small shell script shown in listing 5.12. It is part of the codebase and is uploaded by Bamboo to the target system and started there via SSH.

Line 8 to 10 read the parameters passed on command line and set the default values if necessary. The first parameter is the hostname to use for the application container, the second parameter is the configuration file to be used by the application and the third parameter is the image to start as new container.

Listing 5.12: start_production.sh

```
1  #!/bin/bash
2
3  # usage: ./start_production.sh [$VIRTUAL_HOST=localhost [$CONFIG=prod \
       [$IMAGE=registry.flixbus.com/planr/planr]]]
4
5  # immediatly exit if any command fails
6  set -e
7
8  export VIRTUAL_HOST="${1:-localhost}"
9  export SPRING_PROFILES_ACTIVE="${2:-prod}"
10 IMAGE="${3:-registry.flixbus.com/planr/planr}"
11
12 export VIRTUAL_PORT=8080
13 DB_IP="192.168.1.10"
14
15 echo "cleanup␣prior␣instances␣of␣$VIRTUAL_HOST,␣if␣any"
16 docker rm -fv "$VIRTUAL_HOST" || true
17
18 echo "starting␣application␣container..."
19 docker run -d \
20         --add-host=db:"$DB_IP" \
21         -e VIRTUAL_HOST \
22         -e VIRTUAL_PORT \
23         -e SPRING_PROFILES_ACTIVE \
24         --name "$VIRTUAL_HOST" \
25         --hostname "$VIRTUAL_HOST" \
26         "$IMAGE" -jar -javaagent:newrelic.jar -Dnewrelic.environment=prod \
               -Dnewrelic.config.file=config/newrelic.yml planr.jar
27
28 echo "Application␣started,␣waiting␣until␣it's␣running"
29
30 DIR="$(dirname␣"$0")"
31 exec "$DIR/healthcheck.sh" "$VIRTUAL_HOST"
```

An invokation of the script might therefore look something like this:

```
1  ./start_production.sh planr.flixbus.com prod \
       registry.flixbus.com/planr/planr:NETZ-NET-177
```

Next line 16 ensures that any possibly running application instance with the same name is stopped and removed. (Note: the "`|| true`" is necessary due to the `set -e` in line 6 because the script would stop here otherwise if there is no container under that name running and so `docker rm` would return with a non-zero exit code). Finally the main magic happens in the lines 19 to 26 which is one single `docker run` command. Line 20 adds the IP address of the remote database host to the `/etc/hosts` file of the application container so the hostname "db" points to the database server when resolved within the application container. This allows us to specify the database host in every configuration file of the application simply by the hostname "db". Lines 21 to 23 pass the various environment variables through to the processes running inside the container[18]. `SPRING_PROFILES_ACTIVE` is evaluated by the application itself and determines which configuration file to use, `VIRTUAL_HOST` and `VIRTUAL_PORT` will be used by the nginx-proxy as described in section 5.1.6.

Line 24 and 25 set the name of the container (as listed by `docker ps` or the like) and the hostname (as seen by the processes running inside the container).

[18]Note that these variables have to be `exported` for this to work properly.

The last line of the command eventually specifies the image to run and the command line to be executed inside the container which eventually evaluates to

```
1 java -jar -javaagent:newrelic.jar -Dnewrelic.environment=prod \
    -Dnewrelic.config.file=config/newrelic.yml planr.jar
```

Most of which is application specific and of no further interest regarding our deployment pipeline.

The last line of the script then calls the health check script passing the name of the application container. That script periodically tries to call a REST-endpoint of the application that indicates the successful startup of the application. It returns either with an exit code of zero in case of success or non-zero if the application/container exited or the health check timed out. The health check script is mostly application specific but may be found in appendix 1 on page 74.

5.1.5.2 Automating deployment to integration

The shell script as seen in listing 5.13 used to automate the deployment to integration is mostly identical to the one used for production.

The command line parameters to the script are the same but this time the passed hostname should be based on the branch name as described above and therefore the script should be called with something like the following:

```
1 ./start_integration.sh netz-314.planr-int.flixbus.com int \
    registry.flixbus.com/planr/planr:NETZ-NET-177
```

After again initializing the command line parameters and some variables and stopping any application instances running with the same name it differs from the production deployment: In lines 22 to 26 the script checks whether there already exists a container with the name of the according database instance (which is the application hostname prefixed with “db”, i.e. in this example `db.netz-314.planr-int.flixbus.com`) and if so ensures that the container is running. Otherwise a new database container is created and started. This way a fresh database instance is created each time a new branch is deployed the first time while already existing database instances of the same branch are reused thus preserving any data over multiple deployments and updates of the application images of that branch.

This also causes the difference of the `docker run` command in line 30 where instead of manually providing the mapping of the hostname “db” to the database IP the application container is “linked” to the corresponding database container. This has the same effect and enables resolving the hostname “db” within the application container to the IP of the database container.

The last notable difference to production is the debug port 9090 which is also exposed and mapped to a random available port of the host by line 33. The host port mapping to the debug port is then determined in line 41 and outputted in line 43. This is done to allow the developers to directly connect to the application’s debugging interface.

Listing 5.13: start_integration.sh

```
#!/bin/bash

# usage: ./start_integration.sh [$VIRTUAL_HOST=localhost [$CONFIG=int [$IMAGE=\
    registry.flixbus.com/planr/planr]]]

# immediatly exit if any command fails
set -e

export VIRTUAL_HOST="${1:-localhost}"
export SPRING_PROFILES_ACTIVE="${2:-int}"
IMAGE="${3:-registry.flixbus.com/planr/planr}"

export VIRTUAL_PORT=8080
DEBUG_PORT=9090

DB_NAME="db.$VIRTUAL_HOST"
DB_IMAGE="registry.flixbus.com/planr/mysql-integration"

echo "cleanup prior instances of $VIRTUAL_HOST, if any"
docker rm -fv "$VIRTUAL_HOST" || true

echo "starting database if neccessary..."
if docker inspect -f "{{.Id}}" "$DB_NAME"; then
        docker start "$DB_NAME"
else
        docker run --name "$DB_NAME" -d "$DB_IMAGE"
fi

echo "starting application container..."
docker run -d \
        --link "$DB_NAME":db \
        -e VIRTUAL_HOST \
        -e VIRTUAL_PORT \
        --expose "$DEBUG_PORT" -p "$DEBUG_PORT" \
        -e SPRING_PROFILES_ACTIVE \
        --name "$VIRTUAL_HOST" \
        --hostname "$VIRTUAL_HOST" \
        "$IMAGE" -jar -agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=\
            "$DEBUG_PORT" planr.jar

echo "Application started, waiting until it's running"

EXT_DEBUG_PORT="$(docker port "$VIRTUAL_HOST" "$DEBUG_PORT")"

echo "Debug port is $EXT_DEBUG_PORT"

DIR="$(dirname "$0")"
exec "$DIR/healthcheck.sh" "$VIRTUAL_HOST"
```

5.1.5.3 Integrating deployment into Bamboo

Having automated the deployment of a new application - and in case of a deployment to integration the setup of an associated database for each branch - the correct invocation of these scripts needs to be integrated into Bamboo to be part of the pipeline. In the case of integration the deployment should be done automatically after every successful build of the application, in case of production it should be a push-button release for every build approved by tester users. In both cases all Bamboo needs to do for this is to copy the appropriate script(s) from the codebase onto the target system and invoke it there with the correct parameters.

Analogous to a build plan Bamboo has so called "deployment projects" to configure the steps to perform for a deployment. With this we configured the deployment to production with three steps: download the scripts as build artifacts from the build to be deployed, upload the scripts to the target system via SCP and finally invoke the start script on the target system via SSH.

The `start_integration.sh` script is invoked by a small shell script shown in listing 5.14 run by a SSH task in Bamboo which determines the correct parameters to be passed to it.

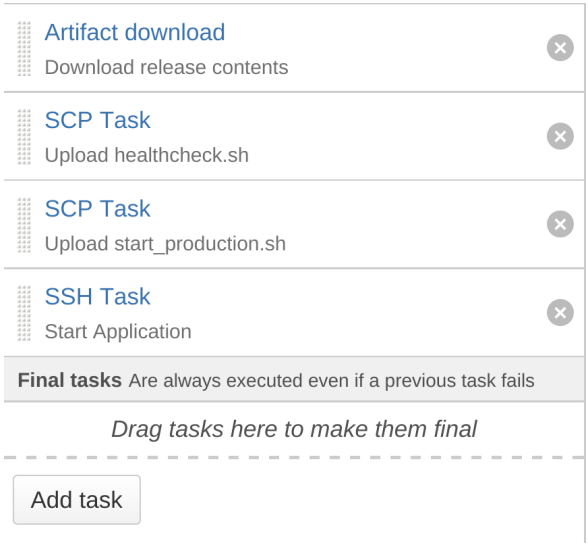

Figure 5.4: screenshot of the deployment task in Bamboo for production

Again all `${bamboo.*}` variables get replaced by Bamboo before executing the script. Except for `${bamboo.planKey}` and `${bamboo.buildNumber}` which are dynamically generated based on the build to be deployed, all this variables are statically configured in the deployment's configuration. With this the name of the application image is constructed in line 5 just like described in section 5.1.3 when building the image. The `docker pull` in line 8 ensures that the image is updated from the registry so in case that there already exists an image with the same name on the target host for whatever reason it will actually be the same image as the one in the registry under that name.

Listing 5.14: Bamboo task performing the actual deployment to production

```
1  # immediatly exit if any command fails
2  set -e
3
4  VIRTUAL_HOST="${bamboo.PRODUCTION_HOST}"
5  image="${bamboo.DOCKER_REGISTRY}/${bamboo.DOCKER_APP_IMAGE}:${bamboo.planKey}-${\
       bamboo.buildNumber}"
6
7  # (re)download image from registry
8  docker pull "$image"
9
10 # ensure scripts are executable
11 chmod +x "${bamboo.PRODUCTION_PATH}/scripts/healthcheck.sh"
12 chmod +x "${bamboo.PRODUCTION_PATH}/scripts/start_production.sh"
13
14 "${bamboo.PRODUCTION_PATH}/scripts/start_production.sh" "$VIRTUAL_HOST" "${bamboo.\
       PRODUCTION_CONFIG}" "$image"
15
16 cat <<EOT
17 ****************************************
18 *
19 * container should now be available here: https://$VIRTUAL_HOST/
20 *
21 ****************************************
22 EOT
```

Before the last command just echoes the URL into the logs of Bamboo where the application should be accessible in the end the start script is invoked in line 14. The effective command executed by this line would look like so:

```
1 /opt/planr/scripts/start_production.sh planr.flixbus.com prod \
      registry.flixbus.com/planr/planr:NETZ-NET-177
```

The commands executed to deploy a freshly built image into integration look almost the same but differ in two major aspects:

Listing 5.15: Bamboo task performing the deployment to integration

```
1  # immediatly exit if any command fails
2  set -e
3
4  normalized_branch=$(echo "${bamboo.planRepository.branch}" | sed 's&^.*/&&;s&[^a-z0\
       -9]&-&gi')
5  subdomain=$(echo "$normalized_branch" | sed 's/^\([a-z]\+-[0-9]\+\)-.*/\1/i')
6
7  VIRTUAL_HOST="${subdomain}.${bamboo.STAGING_HOST}"
8  image="${bamboo.DOCKER_REGISTRY}/${bamboo.DOCKER_APP_IMAGE}:${bamboo.planKey}-${\
       bamboo.buildNumber}"
9
10 # (re)download images from registry
11 docker pull "$image"
12 docker pull "${bamboo.DOCKER_REGISTRY}/planr/mysql-staging"
13
14 chmod +x "${bamboo.STAGING_PATH}/scripts/healthcheck.sh"
15 chmod +x "${bamboo.STAGING_PATH}/scripts/start_staging.sh"
16
17 "${bamboo.STAGING_PATH}/scripts/start_staging.sh" "$VIRTUAL_HOST" "${bamboo.\
       STAGING_CONFIG}" "$image"
18
19 cat <<EOT
20 ****************************************
21 *
22 * container should now be available here: https://$VIRTUAL_HOST/
23 *
24 ****************************************
25 EOT
```

On the one hand it is that the VIRTUAL_HOST is prefixed with a subdomain derived from the branch name. This is done by first normalizing the branch name in line 4 by stripping away all path-like prefixes like e.g. “feature/” or “bug/” and replacing all non alphanumerical characters with a dash so only characters allowed in a hostname are left over. Then line 5 tries to shorten this down to just the name of the corresponding JIRA bug assuming the respective naming convention. Should a branch not be named according to the naming convention its full but normalized name would be used instead. The effective command executed by line 17 would then look like this:

```
1 /opt/planr/scripts/start_integration.sh NETZ-563.planr.flixbus.com int \
      registry.flixbus.com/planr/planr:NETZ-NET-177
```

The other difference to the production deployment is line 12 which also updates the database image from the registry on the target host for the same reason as above. Here this step is even more important since that image is updated regularly but always stored under the same name.

5.1.6 Using nginx-proxy to manage integration instances

So up to this point we automated the building and deployment of new application images and integrated this into the pipeline of Bamboo. Having deployed a new version of the application

into integration the next step is to test it and let some test users evaluate the new features. After passing this testing the new version then may be deployed into production.

But you may have already been wondering how the application instances - especially on integration - should be accessed. Since we did not expose the application's HTTP port via port forwarding to the outside it is currently only accessible on the internal, virtual docker network on the Docker host. As already mentioned earlier we eventually set up a reverse proxy with nginx - or to be more precise we used a Docker image (`jwilder/nginx-proxy`[19]) that combined nginx with docker-gen[20], a small tool that listens on the Docker socket for start and stop events of other containers and then regenerates the nginx configuration files from a template based on the containers still running. We will examine the details of this in the following.

There are two main reasons we did utilize a reverse proxy here: first as before we wanted to encrypt the connection with HTTPS. Secondly without a reverse proxy we would have had to forward the application port of each instance on a separate port of the Docker host. In that case we would have needed some sort of tracking which application instance is listening on which host port. But as we'll see in the following using a reverse proxy instead additionally allowed for an easier management of the integration instances - which is also a mandatory requirement.

5.1.6.1 Making application instances available to the outside through nginx-proxy

The way the `jwilder/nginx-proxy` image works is that it starts a nginx daemon and an instance of docker-gen. Docker-gen then listens on the Docker socket for start/stop events of other containers - the socket has to be mounted from the host into the container for this - (re)generates a new nginx configuration file based on a template file and the currently running containers, and signals nginx to reload the configuration file provided that it actually changed.

The template - which can be found in appendix 2 in its final version - is written using the text/template syntax of the Go language[21] and iterates through an array of all running containers looking for containers having the `VIRTUAL_HOST` environment variable set. It then adds a section for each (non-empty) value of `VIRTUAL_HOST` to the resulting configuration, forwarding all requests for that given hostname to the corresponding container(s) and the port given by `VIRTUAL_PORT`. The values of these environment variables are set when starting the application containers in the deployment scripts (see section 5.1.5).

So in order to enable access to the application containers we simply need to start the `jwilder/nginx-proxy` image and expose its ports on the host:

```
1 docker run -d --restart=always -p 80:80 -p 443:443 -v \
      /opt/docker/nginx/certs.d/:/etc/nginx/certs/:ro -v \
      /var/run/docker.sock:/tmp/docker.sock jwilder/nginx-proxy
```

Having deployed some branches to integration, say e.g. `develop` and `netz-314`, this is already enough to access those instances by opening `https://develop.planr-int.flixbus.com/` and `https://netz-314.planr-int.flixbus.com/` respectively and then let nginx dispatch the requests to the correct container by the hostname used in the request. The activation of HTTPS is done

[19] `https://hub.docker.com/r/jwilder/nginx-proxy/`
[20] `https://github.com/jwilder/docker-gen`
[21] `https://github.com/jwilder/docker-gen#templating`

by providing the certificates and corresponding private keys by mounting the host directory `/opt/docker/nginx/certs.d/` containing them into the container. The docker-gen template then automatically detects the available certificates and changes the nginx configuration to provide HTTPS encryption.

But for this to actually work, we still need to ensure that the hostnames used in `VIRTUAL_HOST` correctly resolve to the host's IP address via DNS. Since these hostnames are constructed from the branch names and we don't know all the possible future branch names in advance we were lucky that it was possible to configure a wildcard domain with our DNS provider, i.e. that every subdomain under `*.planr-int.flixbus.com` now resolves to the Docker host's address while we don't have to explicitly specify each subdomain. In order to avoid warnings in the browsers we also purchased another wildcard certificate issued for `*.planr-int.flixbus.com` and signed by a commercial CA. Important to note here is that wildcard certificates only apply to one level of subdomains as defined on page 4, section 3.1 of RFC2818 ([Res00]), so it was not possible to use a single wildcard certificate issued for `*.flixbus.com` for all HTTPS connections in our pipeline.

For consistencies sake we used the same setup with nginx-proxy on production too although there only one single application instance would be running at a time. Therefore we only need a single DNS record for this (`planr.flixbus.com`) and added `DEFAULT_HOST=planr.flixbus.com` to the environment of the nginx-proxy container so that every request would be forwarded to the one application container even when the hostname used in the request didn't match:

```
1 docker run -d --restart=always -p 80:80 -p 443:443 -v \
      /opt/docker/nginx/certs.d/:/etc/nginx/certs/:ro -v \
      /var/run/docker.sock:/tmp/docker.sock -e DEFAULT_HOST=planr.flixbus.com \
      jwilder/nginx-proxy
```

5.1.6.2 Providing a dynamic index page of integration instances

Although having each application instance on integration accessible on its own subdomain constructed from its branch name is already much more human-friendly to memorize than random port numbers we still need to keep track of which instances are currently running and available. Since the nginx-proxy container already does something very similar while generating a new configuration based on the running containers we found it the easiest to integrate this into the nginx-proxy container.

For this we created our own version of the nginx-proxy image adding an additional docker-gen template file (see `admin.tmpl` in appendix 3) which produces a simple HTML file with a table listing all available application instances together with the URL they're accessible on. Furthermore because we added some build related metadata as environment variables to the application images (see section 5.1.3 on page 32) it was trivial to also display this alongside with the listed containers. It was even possible to examine and list the host port the debug port of each container gets forwarded to. Hereafter it was just a small change in the template of the nginx configuration to make the generated `index.html` available when accessing via an (sub)domain that isn't mapped to an application container; we designated `planr-int.flixbus.com` for this.

Listing all running application instances with the URL the application is accessible on, the name of the Docker image, the application configuration in use, the branch name and commit ID the application was built from, the build plan key used by Bamboo, the build number as a link to the build logs and the debug port on the host we now have a great

URL	Image	Config	Branch
NETZ-563.planr-int.flixbus.com	registry.flixbus.com/planr/planr:NETZ-NET357-8	int	feature/NETZ-563-Efficent-Map-for-planr
NETZ-575.planr-int.flixbus.com	registry.flixbus.com/planr/planr:NETZ-NET372-3	int	feature/NETZ-575-schedule-versioning-fo
NETZ-686.planr-int.flixbus.com	registry.flixbus.com/planr/planr:NETZ-NET364-16	int	feature/NETZ-686_loading_schedule_is_s
NETZ-818.planr-int.flixbus.com	registry.flixbus.com/planr/planr:NETZ-NET354-4	int	bugfix/NETZ-818-fix-broken-versioning
NETZ-824.planr-int.flixbus.com	registry.flixbus.com/planr/planr:NETZ-NET374-2	int	feature/NETZ-824-linevariations
NETZ-825.planr-int.flixbus.com	registry.flixbus.com/planr/planr:NETZ-NET362-19	int	feature/NETZ-825-concession-with-owner
NETZ-866.planr-int.flixbus.com	registry.flixbus.com/planr/planr:NETZ-NET370-4	int	feature/NETZ-866-Create-Bundle
develop.planr-int.flixbus.com	registry.flixbus.com/planr/planr:NETZ-NET-193	int	develop
master.planr-int.flixbus.com	registry.flixbus.com/planr/planr:NETZ-NET235-29	int	master

Branch	Commit	Plan	Build	Debug Port
eature/NETZ-563-Efficent-Map-for-planr	8f19675be5c6e4c43a514d8e6cffe1d5ab8f099e	NETZ-NET357	8	33102
eature/NETZ-575-schedule-versioning-for-merge	c1ed91589ed7fada539aa019a93b2b66bc142ad5	NETZ-NET372	3	33108
eature/NETZ-686_loading_schedule_is_slow	8e7bf6377f301b422ed38651fa7003752fda6e29	NETZ-NET364	16	33105
ugfix/NETZ-818-fix-broken-versioning	aaadab2ec9f145e9e096a128c88dde6a1e56a160	NETZ-NET354	4	33020
eature/NETZ-824-linevariations	999912a96e814573436599c0064ef4220fec0bd5	NETZ-NET374	2	33117
eature/NETZ-825-concession-with-owner	b8c9e58d84e6000ce29b091bd4a0b1e3f9bd090a	NETZ-NET362	19	33118
eature/NETZ-866-Create-Bundle	d38249d8b3fc6666efdbe9384457fb9cdd1e7183	NETZ-NET370	4	33119
evelop	897f38a301d9ea17b6974a4329e68548356c3f90	NETZ-NET	193	33114
aster	b0aa8ca6408e6dcd3209d05795a3d2dcf443e68e	NETZ-NET235	29	33075

Figure 5.5: screenshot of the table on the index page on integration

and detailed overview of our running instances on integration and therefore already satisfied much of the requirement of easy administration of the running instances.

5.1.6.3 Integrating simple container management

But just listing the running instances is only one half of the requirement of easy administration since cleaning up obsolete containers is an explicit part of it. In order to automatically delete containers there was no reliable indicator available to reliably determine the end of a branch's lifecycle. Therefore this should be done through the manual push of a button which then triggers the cleanup of an application instance and its corresponding database.

Adding a button to the index page which would then call some endpoint which in turn implements the actual deletion of the containers was a trivial step. But then it seemed quite challenging to provide the suitable endpoint since it had to translate the HTTP request to the corresponding Docker command and execute it. However the nginx-proxy image itself neither included the Docker client binary nor any CGI module for interactively handling HTTP requests.

But still somewhat unwilling to add a whole new bunch of tools just to handle this one feature we found a neat trick to solve this without much of an overhead: Docker itself already uses HTTP on the socket for its communication between a Docker client and a Docker daemon and the Docker socket of the host is already mounted into the nginx-proxy container. Therefore it was possible to configure nginx to pass some requests directly into the Docker socket so the "translation" of the HTTP request to the corresponding docker command is done by the Docker daemon. For this we only need to drop a suitable HTTP request nginx can forward unmodified to the socket.

Looking at the Docker API documentation, in order to delete a container we have to

issue a request to the endpoint `/containers/<container_id>` with the HTTP method `DELETE` on the Docker socket[22]. Additionally we want to include the query string parameters `v=1` to also delete volumes only used by this container and `force=1` to first stop a running container if necessary. So that nginx passes such a request to the Docker socket we added this paragraph to the configuration of the default host configuration in the `nginx.tmpl` template (see appendix 2 for the complete file):

```
73 location /containers/ {
74         proxy_pass http://docker-socket;
75         auth_basic "Restricted";
76         auth_basic_user_file /app/htpasswd;
77 }
```

This passes all requests under the path `/containers/` to `http://docker-socket` which is mapped to the docker socket file by these lines a bit above:

```
34 upstream docker-socket {
35         server unix:/tmp/docker.sock;
36 }
```

It is vital that the URL to the Docker socket is protected by some kind of authentication - which in this case is via HTTP basic authentication and the file `/app/htpasswd` in the container containing the valid users - because nginx passes any requests on this path directly to the Docker socket. But having access to this socket it is a cakewalk to gain full root access to the host machine since one can start a container from any image and mount any directory of the host into that container. Hence access to the socket can be seen as equal to root access to the host and must therefore not be publicly accessible.

Next we copy the template of the index page and add delete buttons and some JavaScript that generates the correct HTTP requests when a delete button gets clicked (see `admin.tmpl` in appendix 3)

What the `docker-utils.js` script shown in listing 5.16 does is to register an event handler to all the delete buttons. When then a button is clicked the event handler first asks the user to confirm that he wants to delete the selected application and corresponding database container and if confirmed calls `removeApp()` passing the name of the application container. `removeApp()` then calls `deleteContainer()` two times which in turn constructs the respective HTTP request to delete the given container (using the `$.ajax()`[23] helper of the jQuery[24] framework used). On the first call `removeApp()` passes the name of the database container to `deleteContainer()` by simply prefixing the passed name of the application container with "db." and on the second call it passes the name of the application container. Finally it triggers a page reload after a short waiting period so the page should be already updated by docker-gen having removed the deleted application instance from the list.

This small administration page will now be provided as `/containers/admin/index.html` for which we add the following paragraph to the nginx configuration template:

```
79 location /containers/admin/ {
80         root /app/htdocs/;
81         index index.html;
82         auth_basic "Restricted";
83         auth_basic_user_file /app/htpasswd;
84 }
```

[22] https://docs.docker.com/reference/api/docker_remote_api_v1.15/#remove-a-container
[23] https://api.jquery.com/jQuery.ajax/
[24] https://jquery.com/

Listing 5.16: docker-utils.js: JavaScript helper functions used on the administration page

```
jQuery(function($) {
    function deleteContainer(container, success, error) {
        $.ajax({
            url: '/containers/' + container + '?v=1&force=1',
            method: 'DELETE',
            complete: function(jqXHR, textStatus) {
                if ([204, 404].indexOf(jqXHR.status) != -1) {
                    success(container, jqXHR, textStatus);
                } else {
                    error(container, jqXHR, textStatus);
                }
            }
        });
    }

    function removeApp(app_host) {
        var db_host = "db." + app_host;
        var error = function(container, jqXHR, textStatus) {
            alert('Deleting container "' + container + '" failed with code ' + \
                jqXHR.status + ' (' + textStatus + ')');
        }
        deleteContainer(db_host, function() {
            deleteContainer(app_host, function() {
                setTimeout(function() {
                    window.location.reload()
                }, 800);
            }, error);
        }, error);
    }

    $('.delete-button').click(function() {
        var button = $(this);
        var app_host = button.data('app-host');
        if (confirm('Are you sure you want to delete the container for "' + \
            app_host + '" and its corresponding database?')) {
            removeApp(app_host);
        }
    });
});
```

Again we add HTTP basic authentication to the page. This is important to make for a convenient usability of the administration page: when accessing the page for the first time the browser prompts for the credentials to use for HTTP basic authentication and saves them for further request. This way the request submitted by the `$.ajax()` function can succeed since the browser automatically adds the stored credentials to them. This is the reason why the administration page is a separate one instead of simply adding the delete buttons to the unprotected index page because the `$.ajax()` requests then would fail since the browser wouldn't have the credentials cached and would not automatically prompt for them since the request was triggered by JavaScript in the background.

5.1.6.4 Adding convenient error messages for deployment

After some time of using all of this the development team found that the nginx-proxy instance in production needed one final touch, which was to add a custom error message to be shown while a new version of the application gets deployed. Because while a new version is starting up it is not yet accepting incoming HTTP requests which nginx - trying to pass requests

to the application not yet ready to accept them - reports with a HTTP response code of 502 ("bad gateway"). Therefore the team created a custom error page as static HTML file, placed it under `/app/htdocs/error/502.html` in the container and added the following lines to the nginx configuration template:

```
193 location /error/ {
194         root /app/htdocs/;
195 }
196 error_page 502 /error/502.html;
```

5.1.7 Dockerizing the database

Up to this point we only covered the virtualization of the application itself and payed little attention to the virtualization of the database although it is as important as the virtualization of the application in order to provide a production-like database to test against in integration. In section 5.1.5.2 the Docker image used for the database instances of each branch, `registry.flixbus.com/planr/mysql-integration`, was already mentioned but without further explanation when and how is was created. But we'll catch up on this now.

5.1.7.1 Creating a common base image

According to the concept designed in section 4.2.3 the first step for this is to create a common base image on which then the images to be used in production and integration could be build upon respectively. That way the difference between the images can be kept to a minimum making the integration instances as production-like as possible.

We created this base image based on the official `mysql` image on the Docker hub[25] by copying the Dockerfile of its version 5.6[26] and modifying it a little - see appendix 4. Namely we removed the definition of the `VOLUME` storing the database's data files and didn't include the script that handles the initialization of MySQL on first startup.

The `VOLUME` was removed because in production this is also done by the start parameters to the docker container and in integration it would be more disadvantageous than beneficial to have the volume already predefined in the image[27]. Similarly the initialization script was replaced by the initial manual setup of the production MySQL instance (creating a database and user with full access on it) - or to be more exact by migrating the already existing MySQL data into the container setup - and in integration the initialization is handled differently as described below.

With this we are already ready to build the base image and setup the production database with it:

```
1 # build our mysql-base image and upload it to our registry
2 # (on any machine with Docker and with the above Dockerfile in the current directory)
3 docker build -t registry.flixbus.com/planr/mysql-base .
4 docker push registry.flixbus.com/planr/mysql-base
5
6
7 # executed on the host machine of the production database
8 docker run -d --restart=always --name=planr_db -p 3306:3306 -v \
      /opt/planr/database/:/var/lib/mysql/:rw registry.flixbus.com/planr/mysql-base
```

[25] `https://hub.docker.com/_/mysql/`

[26] `https://github.com/docker-library/mysql/blob/1f430aeee538aec3b51554ca9fc66955231b3563/5.6/Dockerfile`

[27] see section 6.1.2 on page 64 for more on this

The last command downloads the image from the registry if necessary, starts the container with the name `planr_db` (`--name=planr_db`) in background (`-d`) and sets the restart policy to always (`--restart=always`) so the container is automatically restarted when the host reboots. The MySQL port 3306 of the container is forwarded to the same port of the host machine (`-p \ 3306:3306`) to make it available to the outside. Furthermore the directory `/opt/planr/database/` of the host's filesystem is mounted into the container in read-write mode to persistently house the database's data files (`-v /opt/planr/database/:/var/lib/mysql/:rw`).

However before the first start of the production database container we had to migrate the existing data of the natively installed MySQL instance into the container setup which was also a MySQL in version 5.6 thus the "migration" went like this:

```
# stop native MySQL instance
service mysql stop

# create directory for container volume and copy MySQL files there
mkdir -p /opt/planr/database/
cp -a /var/lib/mysql/* /opt/planr/database/

# change the owner and group of the volume directory
# where 48 is the numerical id mapped to the mysql user/group within the container
chown 48:48 -R /opt/planr/database/
```

With the actual database data stored on the host filesystem deploying a new version of the mysql-base image available in the registry is quite straightforward as well because it just involves pulling the new image from the repository and restarting the container with it:

```
# pull the latest image from the registry
docker pull registry.flixbus.com/planr/mysql-base

# stop and remove the currently running db container
docker rm -f planr_db

# restart the database
docker run -d --restart=always --name=planr_db -p 3306:3306 -v \
    /opt/planr/database/:/var/lib/mysql/:rw registry.flixbus.com/planr/mysql-base
```

5.1.7.2 Building mysql-integration with recent production snapshots

Now looking back at section 4.2.3 the specified concept for the integration database images gives a quite precise guideline to the implementation of the mysql-integration image.

The first step is to create a dump of the production database that should then be included in the image which we accomplish with the standard `mysqldump` utility. To automate this task with Bamboo we created a new build plan "dump production database" which executed a single short shell script on the build server:

```
# immediatly exit if any command fails
set -e

echo 'Start dump of ${bamboo.DB_NAME} database (production)...'
mysqldump --host="${bamboo.DB_HOST}" --user="${bamboo.DB_USER}" \
    --password="${bamboo.DB_PASSWORD}" --databases "${bamboo.DB_NAME}" > \
    "${bamboo.DB_DUMP_NAME}"
echo '...dump of ${bamboo.DB_NAME} database finished'
```

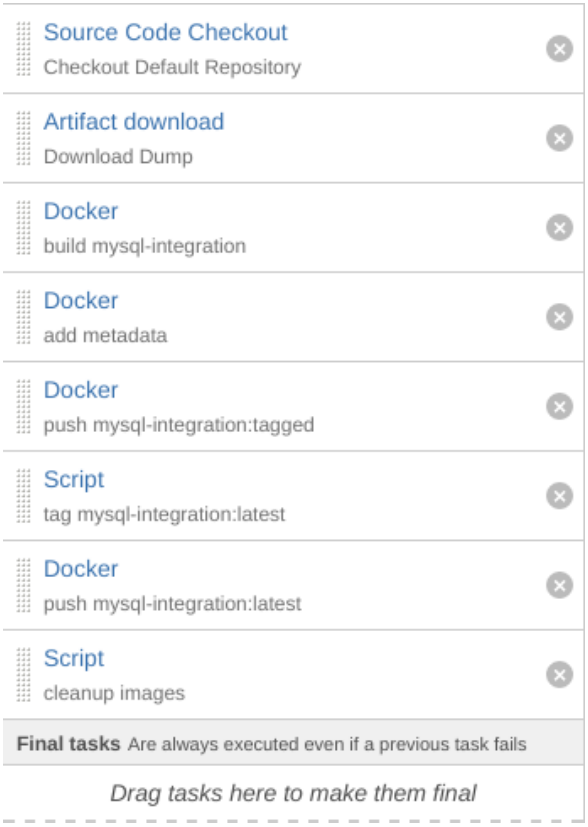

Figure 5.6: screenshot of Bamboo tasks to create the mysql-integration image

The relevant line here is line 5 which executes `mysqldump` on the build server connecting to the production database instance and dumbing the named database. All the `${bamboo.*}` variables are replaced by Bamboo before executing the script and can be configured in the "variables" tab of the build plan in Bamboo. `${bamboo.DB_DUMP_NAME}` is configured to be `dump.sql` and is the file the output of `mysqldump` is piped into. This file is also configured as build artifact of this build plan so it can be accessed by another build plan in the next step which is building the mysql-integration Docker image.

For this we set up another build plan with the following tasks: first checkout the applications source code which also contains the Dockerfile of the mysql-integration image. Then download the `dump.sql` artifact of the last run of the "dump production database" build plan and place it in the same folder as the Dockerfile of mysql-integration.

The next steps then are analogous to the build of the application image as described in section 5.1.3: build an interim image according to the Dockerfile, add some build metadata to the image as environment variables, tag and upload it to the registry as `registry.flixbus.com/planr/mysql-integration:latest` and `registry.flixbus.com/planr/mysql-integration:${bamboo.planKey}-${bamboo.buildNumber}` and finally clean up by deleting all these interim images from the building machine again.

But the most interesting part of this whole process is the Dockerfile which defines how to build the mysql-integration image implementing the concept of figure 4.9:

Listing 5.17: Dockerfile of mysql-integration

```
1  FROM registry.flixbus.de/planr/mysql-base
2  MAINTAINER Roman Anasal <roman.anasal@flixbus.de>
3
4  RUN mysql_install_db
5
6  COPY setup_users.sql /tmp/
7  COPY dump.sql /tmp/
8
9  RUN mysqld & PID=$! && sleep 5 && \
10     cat /tmp/setup_users.sql /tmp/dump.sql | mysql -u root && \
11     rm /tmp/setup_users.sql /tmp/dump.sql && \
12     kill $PID && wait
```

In the first line we can see that this image directly builds upon the common mysql-base image. From there it first runs `mysql_install_db` inside the image which initializes the database files for the first run. In line 6 and 7 it copies two SQL files into the container where the first is part of the source code repository while the second is the database dump which was

downloaded as artifact in the step before. The last `RUN` directive then starts up the MySQL daemon within the image, executes the two SQL files and afterwards deletes them from the final image and at last gracefully shuts down the MySQL daemon. The `setup_users.sql` file contains a simple SQL snippet to create a new database user and grant him access to the application database that in turn is created by the SQL in `dump.sql` which besides the data and schema of the database, also contains the SQL needed to create the database since we used the `--databases` switch with the `mysqldump` command earlier.

So the final mysql-integration image we obtain with this is mostly the same as the mysql-base image - actually it is identical to it except for - containing a current copy of the production database's data. Hence when such an image is started, e.g. in the `start_integration.sh` script from section 5.1.5.2, it directly starts up the MySQL daemon and provides access to a copy of the production database shortly after. All changes to such a database are saved inside the container and are therefore coupled to the lifetime of the container allowing for easy cleanup when not needed anymore by simply deleting the container.

5.1.8 Local usage of Docker

Most of our requirements are mastered at this point. But one requirement we did not take any care of yet which is enabling an easy setup of a local testing environment independent of the operating system of the developer machines. So let's assess what's required to be supported on the developer machines in order to setup a local testing and integration environment. First of all of course Docker needs to run on the local machines since it is the central technology of our pipeline. But as more or less everything part of the various application environments is packaged as Docker container that's basically already it. Docker is the only major requirement for our setup.

5.1.8.1 Local usage on Linux

So regarding a local developer machine with a Linux operating system the setup is rather easy because it is roughly the same as the setup of the servers as described in section 5.1.2 and Docker can be found in the software repositories of every major and current Linux distribution. With Docker installed the images in our private registry can be downloaded and started locally as well after the initial execution of `docker login` as described in section 5.1.4.2 on page 38. Preferably every developer should have his own login credentials for this which can be created with `htpasswd` as described just a little prior in section 5.1.4.2.

After this initial local setup of Docker bootstrapping a local testing environment already is automated thanks to the `start_integration.sh` shell script from section 5.1.5.2: invoking this script automatically downloads the images of the application passed as parameter and the integration database if necessary, sets up (or restarts) and links together a database container and an application container. Shortly after the started application container accepts incoming HTTP requests. These can be done by pointing the local browser directly to the internal IP and port of the application container which can be determined with

```
1 docker inspect -f '{{.NetworkSettings.IPAddress}}' $container_name_or_id
```

Alternatively one can also utilize the nginx-proxy as described in section 5.1.6 for more comfort. Not providing any certificates locally nginx-proxy gracefully falls back to unencrypted HTTP only, so this isn't any problem. However since nginx-proxy dispatches the

requests based on the host name in the request only one instance named "localhost" can be supported without any further messing around with the local DNS.

Furthermore the developer isn't limited to use only images already available in the registry uploaded by an automatic build from Bamboo but can also test docker images built locally. He can do this by invoking `docker build` in the directory containing the corresponding Dockerfile of the image to build and then passing the name or ID of the created image to `start_integration.sh`.

5.1.8.2 Local usage on Mac OS X and Microsoft Windows

So setting up Docker on Linux is straightforward but what about the other two major operating systems we have to support? Docker runs containers using Linux specific kernel features and therefore depends on a recent Linux kernel as operating system like mentioned earlier. Thus it seems to be impossible to run Docker on top of another operating system.

But this being a commonly requested use case for local development fortunately there already was a clever solution for this available: boot2docker[28][29]. What boot2docker is providing is on the one hand an image of a lightweight Linux distribution which can be run as a virtual machine and on the other hand a boot2docker CLI command native to the host operating system of the developer machine. This CLI command enables to provision a new VM running the boot2docker Linux distribution with a few commands by leveraging VirtualBox as hypervisor. Inside the VM then runs a Docker daemon binding its control socket to a TCP port of the VM. The boot2docker CLI can then be used to export the address of this port to the shell environment on the developer machine. This is then used by the also included Docker client program to connect from the host system to the Docker daemon running inside the VM.

```
1  # initialize the boot2docker VM
2  boot2docker init
3  # boot the VM
4  boot2docker up
5
6  # set the appropriate environment variables for the Docker client
7  eval "$(boot2docker␣shellinit)"
8  # export DOCKER_HOST=tcp://<ip of the boot2docker VM>:<port of Docker daemon>
9
10 # run the hello-world container
11 docker run hello-world
```

So boot2docker exploits Docker's client/server architecture to augment a native Docker installation on other operating systems by passing the `DOCKER_HOST` environment variable to the native Docker client on the host so it connects to a Docker daemon which actually runs inside a Linux VM. This also means that the Docker containers run on the Linux VM as well.

Having Docker installed this way on a Mac OS X or Microsoft Windows operating system opens up mostly the same capabilities as with a Linux machine above: the developer can access the registry after an initial `docker login`, start a local integration environment with the `start_integration.sh` shell script - provided the bash shell is available on the developer

[28] `http://boot2docker.io/`

[29] Although by this time boot2docker is deprecated in favor of docker-machine (`https://docs.docker.com/installation/mac/`), the principles behind both are the same.

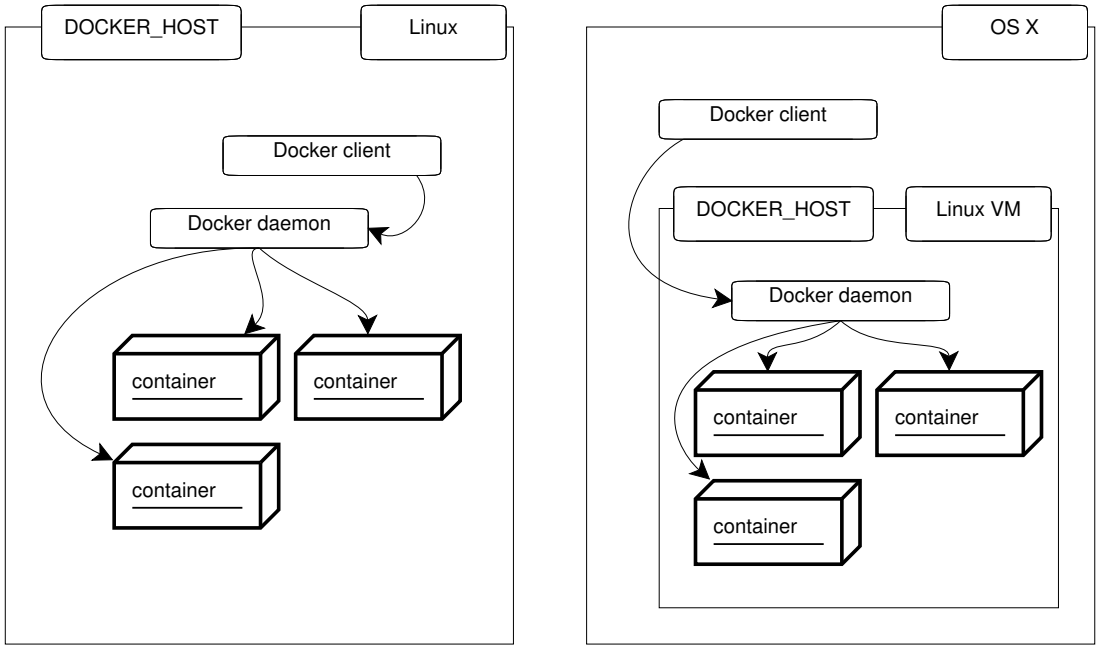

Figure 5.7: running Docker on Linux vs. running Docker on Mac OS X

machine which is the case on Mac OS X and can be installed on Microsoft Windows e.g. with Cygwin[30] - and build new Docker images from source.

5.2 Summary and evaluation

5.2.1 Coverage of requirements

Having finished the implementation of the pipeline the most important question is: did we satisfy all the requirements imposed on the pipeline? To determine this let's check the remaining requirements that were not already satisfied by our concept:

Integrate with the existing technology stack Beginning our pipeline just after the push of code changes to the Stash repository the changes our new implementation introduced were mostly just additions to the existing state: so the building and testing steps of the application in Bamboo remain untouched and are only complemented with the additional `docker build` and `docker push` steps introduced in section 5.1.3. Despite the deployment scripts were completely rewritten/changed they still are just tasks executed inside Bamboo and therefore integrate smoothly. Also the new build plans used to build the database images fully integrated in Bamboo. The target system in production and integration didn't need any change besides the installation of Docker (and maybe an upgrade of the kernel). So this requirement is fully satisfied.

Push-button releases Since the deployments are integrated into Bamboo just as before they can be triggered just as easy as in the previous status quo. Although the deployment to

[30] `https://www.cygwin.com/`

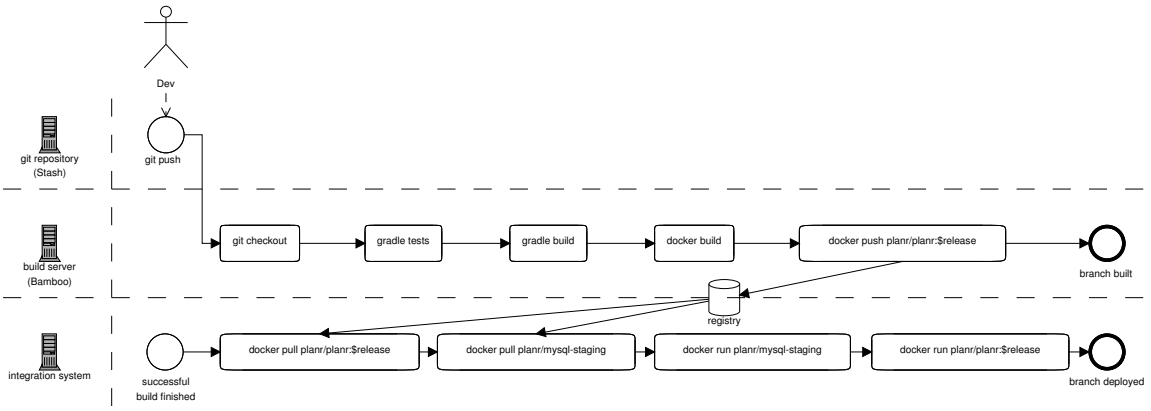

Figure 5.8: diagrams of the full build and the deployment workflow

integration was further improved because it is now automatically triggered for every good build of the application, deprecating the manually interaction for integration deployments.

Build and deployment to integration in <10min After pushing new changes to the Stash repository Bamboo automatically triggers a new build of the application, packages that build into a Docker image and deploys that image to integration, notifying the initiating developer if anything of this fails. Currently the time span from the push until the new application instance on integration is ready for testing varies between 7 and 9 minutes (see section 5.2.3 for details). Although tight this is still in the acceptable range.

Easy administration and cleanup of integration instances This requirement is solved with our nginx-proxy implementation described in section 5.1.6. It not only lists all running integration instances with additional metadata but also allows to delete no longer required instances with nothing more than a click.

Local testing environment can be setup fast and easy/supported cross-platform As we just saw in section 5.1.8 this is given on any platform running Docker or boot2docker respectively.

Zero-downtime deployments Deployments to production or of a new version of the same branch to integration still have a downtime after the previous instances is stopped and deleted and before the new instance finished starting up. Although with about 2 minutes being comparably small this downtime is still too big to comply with this requirement. Nevertheless the ground work for this feature is done as we'll see later in the outlook in section 6.2.

Horizontal scalability When only regarding the horizontal scalability of integration instances this could be counted as satisfied because the number of parallel running integration instances isn't limited conceptionally. Also it would be principally possible to run multiple production application instances at the same time - actually the nginx-proxy then even would automatically perform a simple load balancing between these instances. But on the one hand all instances would still have to run on the same Docker

	functional	non-functional
mandatory	enable deployment to all environments through the same pipeline ✔	integrate with the existing technology stack ✔
	dedicated instance per development branch on integration ✔	push-button releases ✔
	semi-persistent database instances per development branch on integration ✔	production-like integration environment ✔
	recent copy of production database on integration ✔	build and deployment to integration in <10min ✔
	support upgrades of the database ✔	easy administration and cleanup of integration instances ✔
		local testing environment can be setup fast and easy/supported cross-platform ✔
optional	zero-downtime deployments ✗	separate environment configurations from code ✗
	horizontal scalability ✗	

Table 5.1: fulfillment of the requirements by the new implementation

host since we don't have a load balancer/gateway for multiple Docker hosts configured yet and on the other hand the application itself would not support this because the session data is stored on the filesystem, i.e. inside each instance's container.

Separate environment configuration from code In the source code repository are still one common configuration file as well as multiple environment specific configuration files included. We already moved almost everything into the common configuration file but the only things we could not move yet and therefore are the only content of the environment specific configurations are the credentials of the database user. In fact the database name, connection options such as the encoding and even the name of the database host are included in the common configuration since they're identical in every environment - the latter one thanks to the linking and DNS features of Docker, seen in section 5.1.5. We could have eliminated this one last difference by using the same credentials on every database but decided against this as a safety measure so that an integration instance of the application could never accidentally connect to the production database successfully. On the other hand all the configuration files are included into the application images and which one is used is determined by an environment variable passed on container start up.

So as we can see in table 5.1 only the optional requirements are not completely satisfied yet. But since their fulfillment currently would require more effort than the advantages were worth the development team decided that this is OK.

5.2.2 Improvements made with the new implementation

At this point I'd like to make a short recap about what significant changes were introduced by this implementation and the advantages they bring.

Probably the most important and extensive improvement is the enhanced reproducibility entailed by using the container technology. The deployed build artifacts - i.e. the application images - now contain all application specific dependencies and configurations by themselves and the only requirement imposed on the target environment is the Docker engine. That way it is always guaranteed that the runtime environment of the application is exactly the same in production as on the testing and integration systems - no "wrong JRE version installed", no "the filesystem path is different" and no "the production system uses another character encoding" is even possible.

This isolation of the runtime environments into the containers allows us to run multiple instances of the application of the same host system at the same time without any worries that they would interfere with one another. And this does not only hold for the application instances but for the database instances as well as we did see in the integration environment where now every development branch has its own test database coupled to the lifecycle of the branch. Additionally these databases also contain recent copies of the production data and therefore enable testing with real-life data. So probably the only way to test in a more production-like environment is to test directly in production itself (which is quite obviously, a very bad idea).

Thanks to the start scripts included in the codebase and the deployments configured in Bamboo, the only manual action required to deploy is in case of production the push of a single button and in case of integration even just pushing code changes to the central Stash repository - which is done anyway. Furthermore the scripts also allow to quickly bootstrap an integration instance on local developer machines.

But this wouldn't be possible without the registry that not only acts as a central artifact repository but also enables an easy and efficient distribution of the images into the various environments as well as to the developer machines if needed. At the same time these images now also contain extended metadata notably the branch name and commit ID they were created from, their creation time as well as a web link to the complete logs of the build.

5.2.3 Costs and overhead of the new implementation

Only highlighting the positive aspects of our implementation would be quite one-sided, so what are the costs all this awesomeness comes at? As mentioned earlier the total duration of a run of our pipeline takes 7 to 9 minutes which is already quite close to our 10 minute limit. In order to determine the impact of our implementation and separate that from the application specific contribution to the total duration we had a look at some of the build logs in Bamboo. Those showed that durations of the three main build stages were for the tests about 2-3 minutes, the building stage about 1-2 minutes and for the deployment stage again 2-3 minutes.

The test stage is purely application specific and wasn't touched at all on behalf of our implementation. The building stage is divided in the application specific building of the application JAR file and the Docker specific building and uploading of the application image. So we have to subdivide this stage again into the application specific build - which takes mostly about 1 minute, sometimes up to 1:30 minutes - and the Docker specific build -

which varies between 15 and 25 seconds for building the application image, another 20 to 30 seconds for storing the image in the registry and 1-2 seconds for removing the temporary images from the build server, thus totaling to less than 1 minute.

The complete deployment stage was newly added as part of our implementation. But since this stage ends not until the application is ready to accept incoming requests its duration also includes the full application specific startup and initialization phase. Looking at the logs these make up for the majority since the application is already running after round about 25 seconds of copying and starting the start scripts, pulling the image(s) from the registry and setting up the containers. So the total contribution of our Docker specific implementation to the whole pipeline varies between 2 and 2:30 minutes.

But what exactly is the additional overhead imposed by Docker? To measure the time Docker takes to setup a new container, start an application in it and clean it up afterwards the following command is enough:

```
1 > date -u +%H:%M:%S.%N; docker run --rm ubuntu date -u +%H:%M:%S.%N;  date -u \
      +%H:%M:%S.%N
2 13:38:38.206711033
3 13:38:41.108290534
4 13:38:41.483847419
```

This line first echoes the current time (including nanoseconds) and then runs the same command within a Docker container - in this case based on the Ubuntu image - and finally outputs the time once again after the Docker command returned. From this output - which is in this case the result from a run on a developer machine - we can tell the time it took to setup and initialize the new container and start a new process in it - little less than 3 seconds - and how long it took to clean up the container - not even half a second.

Furthermore regarding also memory usage, CPU load, storage and network bandwidth there is little to no overhead since all that Docker does is to setup an isolated execution environment using the container features of the Linux kernel. But the isolated processes then still run directly on the kernel without Docker adding any additional virtualization or emulation layers. So actually the performance difference of processes running inside a Docker container is mostly negligible to running it natively which is also shown by a paper published by IBM: [FFRR14].

But wait, don't we have an increased consumption of storage and network bandwidth since we now have to store and transfer not only the application but also all of its dependencies and execution environment? Well, yes, our final build artifact is now a Docker image, containing not only the application but also the entire Java runtime environment totaling to a size of about 250MB. But due to Dockers layering concept common image layers can be reused and shared between multiple images and therefore only new layers have to be transfered and then occupy additional storage. Since all application images use the same base image, only the layers adding the application files to the image are different between the builds. And since the layers only contain the differences to their parent layers, the size of the layers to transfer and store for each new build is with around 85MB just as big as the application files contained in that layers, which had to be transfered anyways. Therefore also the additional storage and network usage impose only negligible overhead after the first use.

6 Conclusion

In this last chapter I'd like to leave some final thoughts on the experiences made with Docker in the course of the implementation of our continuous deployment pipeline. Therefore the first section of this chapter is devoted to record some of the most important learnings made while getting to known this rather young technology. In the very end of this chapter the outlook then shows up some possibilities and further improvements the container-based implementation can be extended to in its further development.

6.1 Learnings

This section covers the learnings I gathered through the experience of the implementation and by experimenting with Docker. Some of them are just the end result of these experiences, others did also actively influence the implementation. These considerations therefore may also make some of the decisions made in the final implementation more comprehensible to the reader retrospectively.

6.1.1 Docker advantages and disadvantages

The evaluation of the implementation in section 5.2.3 made clear that the overhead introduced with Docker is mostly negligible and that the performance of the virtualization is therefore almost equal to a natively run application. Thus are Docker and Linux containers indeed a very lightweight alternative to virtual machines. Due to the absence of virtual hardware and to the shared kernel the application inside the container has to be compatible with the underlying operating system and can not bring its own as it would be the case with virtual machines. Therefore (currently) only Linux applications can be managed utilizing the Docker engine.

But this may change in the future since Docker is still under heavy development which can be seen as an advantage and disadvantage at the same time: some features especially in the user interface are still missing or incomplete causing a negative user experience which on the other hand may already be fixed in the next release though. Introducing new features and refactoring existing code the Docker developers also don't hesitate too much deprecating APIs and therefore one should always take a look in the release notes before upgrading to a new Docker release.

Nevertheless, with its already existing and field-tested build automation of images and the focus on reproducibility Docker containers were an excellent fit for a continuous deployment pipeline regarding our requirements.

6.1.2 Tips and tricks for reproducible image builds

Speaking of build automation: there are some aspects to keep in mind for efficient and reproducible builds of Docker images. The most import of them is the caching mechanism

which is based on the concept of image layers described in section 5.1.1. When building a new image Docker first loads the base image specified with the FROM directive in a Dockerfile and then executes the subsequent directives starting from this base image. Since each directive implicitly creates a new image layer, Docker first checks if there already exists a layer based on the same parent layer/image and created through executing the same directive in a previous build. If such a layer is found, instead of executing the directive Docker reuses the existing layer created the same way[1].

This build cache is the reason why we preferably ordered the directives in our Dockerfiles that way that steps which are likely to have the same result between multiple builds are executed first and steps likely to produce different results are executed last allowing to share as much layers as possible between the various builds. For example in the Dockerfile of the application image from section 5.1.3 the files of the application are added at the very end of the build, allowing the previous layers to be shared between all application images since these are identical for every build based on the same base image. Therefore only the the layers created by the last three directives differ between the builds (or even only the last two if the application configuration files copied into the image don't change).

Another important aspect for reproducibility is the source of the base image used: when the image referenced in the FROM directive does not exist on the Docker host the Docker daemon tries to download it e.g. from the Docker hub. In subsequent builds this download is skipped since the image then already exists on the host. But therefore it may happen that building the same Dockerfile on different Docker hosts creates images actually based on two different base images in the case that on one Docker host an older version of the base image was already present while the other Docker host downloaded the latest version from the Docker hub. This was one reason why we created our own mysql-base image and stored it in our private registry since there it could not be change without our knowledge.

One other reason though was to remove the VOLUME declared in the original Dockerfile as already mentioned in section 5.1.7.1. Otherwise each container created from that image or any image based on it would automatically create a new data volume mounted into that container. Although these volumes are managed by the Docker daemon, since to prevent accidental data loss they are not automatically deleted this could cause the accumulation of orphaned volumes. Because after the last container referencing such a volume got removed there is no way to delete it through the Docker client[2] and one has to delete the volume manually or using a suitable script[3]. (This is also an example of a feature incomplete in the user interface yet.)

6.1.3 Debugging applications inside containers

Docker containers offer a great isolation of processes running in them. But this isolation can make it also a bit tricky when the need to inspect the application in this isolated environment arises. With planr this was the case a few times when trying to debug the application itself. Two Docker subcommands proofed to be useful in such a case.

The first of this was `docker logs`, especially because all the containers in our pipeline are started in the background (due to the `-d` switch to `docker run`) and therefore all output of the application to stdout and stderr is captured only by the Docker daemon. Thankfully the

[1] for more details, see `https://docs.docker.com/articles/dockerfile_best-practices/#build-cache`
[2] see `https://docs.docker.com/userguide/dockervolumes/#creating-and-mounting-a-data-volume-container`
[3] For example this script: `https://github.com/chadoe/docker-cleanup-volumes`

daemon stores this output into separate log files for each container which can be read with `docker logs` allowing to investigate for example why an application instance abnormally exited during startup. Passing the `-f` switch `docker logs` can also continuously output the logs of a running container.

The second useful command was `docker exec` which executes an arbitrary command within the execution environment of an already running container. This allows e.g. to start a shell in the container and therefore to interactively inspect the container from within. Although only binaries/scripts already existing inside the container can be executed that way.

6.1.4 Limitations of boot2docker

Boot2docker, as described in section 5.1.8.2, is a great tool to run Docker seamlessly integrated on top of a non-Linux operating system. However having the Docker daemon and therefore the containers actually running on "another" host - i.e. the boot2docker VM (see figure 5.7) - adds some limitations and special characteristics to keep in mind:

The internal virtual Docker network is shared by the containers and the Docker host. Because the Docker host is the VM and not the developer machine itself it is not possible (without manually manipulating the developer machine's and the VM's routing tables or tunneling connections e.g. with SSH) to directly access the containers via their internal IPs. Therefore container ports to be accessed from the developer machine have to be forwarded on the Docker host's network interface. Then it is possible to connect to them on the IP address of the VM from where the port is forwarded to the container. That IP can be determined with `boot2docker ip`.

This especially means that when using the `start_integration.sh` script the application container can only be accessed through a nginx-proxy since there is no direct route to the container's internal IP. But because the nginx-proxy can also only be accessed through the IP address of the VM and not by hostname (unless of course you somehow configured the DNS on the developer machine to let some domains resolve to the VM's IP) it has to be started with a `DEFAULT_HOST` environment variable equal to the `VIRTUAL_HOST` environment variable of the (only) running application container (see also section 5.1.6.1 on page 48). Alternatively the exposed port of the application container itself could be forwarded on the Docker host's network interface instead. But this would mean to either edit the `start_integration.sh` script or execute the adapted commands manually.

Similarly to the network the volumes mounted into containers are, too, local to the Docker host and not the developer machine. So all paths passed to the `-v|--volume` switch to be mounted into a container must exist inside the VM. However this is different to the `ADD` and `COPY` directives in a Dockerfile because `docker build` sends its working directory - its "build context" - as a tar archive to the (remote) Docker daemon which can that way access the client-local files while building a new image.

Another pitfall is with connecting to a registry: the `docker login` stores the credentials to a registry in the file `~/.dockercfg` which is to be considered client-side, i.e. this file lies in the home directory on the developer machine. In contrast to this when connecting to a registry with a self-signed certificate (see section 5.1.4.3) this connection is established by the Docker daemon and therefore a custom `ca.crt` file as to be placed on the filesystem of the Docker host, i.e. inside the VM.

6.1.5 Security considerations

While Docker really makes it easy to quickly setup a container-based infrastructure and applications, verifying the integrity of images is currently quite hard: in contrary to most of the software package managers used by Linux distributions, images especially in the public Docker hub are not (yet) cryptographically signed by their creators allowing to check images against malicious manipulation by default. But also besides that the creator of an image could deliberately place malicious code into images published by him. So in order to get a trustable image it should be built by yourself after reviewing the Dockerfile. But since most Dockerfiles start with another base image you would have to go all along the chain to the root image which is often created by copying an archive file into an empty container. Not always is it possible to then verify that archive's integrity without having to also create that from scratch.

Another security risk is caused by missing updates in Docker images. Since every image includes all dependencies required by the applications run in it, a security-critical update of a shared library for example has to be included in every image which basically means rebuilding them all. This is different to the method commonly used on Linux systems where a shared library is installed system-wide and therefore only has to be updated once for the whole system. The additional effort to rebuild all images every time a single package is updated makes it likely that this will not be done in practice leaving the images vulnerable through the outdated packages.

6.2 Outlook

Finally this section is supposed to turn the view towards the future of the implemented pipeline. Conceivable next steps are of course the fulfillment of our optional requirements which were not completely satisfied by the new implementation yet. But also further use cases not considered so far may be incorporated into the pipeline and the pipeline itself may be ported to other development projects.

6.2.1 Zero-downtime deployments and horizontal scalability

As stated in section 5.2.1 the ground work for zero-downtime deployments as well as for horizontal scalability of the application is already made by the container-based implementation. Packaged as Docker image the application can be deployed quickly on any host running the Docker engine allowing for multiple instances on multiple hosts and/or on the same host due to the isolation. A simple load balancing for multiple instances on the same host would already be done by the nginx-proxy. Distributing requests to multiple Docker hosts could be done by an additional nginx container which still is to be created as such a load balancer. So the main challenge for horizontal scalability at this time is the application itself because currently the session data is stored inside a running container and thereby bound to a single instance. For containers on the same host this could be solved by sharing a data volume for the session data between them, for application instances on multiple hosts storing the session data inside the database would be a possible solution. In any case it is in the scope of the application to change this behavior and not in the scope of the deployment pipeline.

Being able to run multiple production instances of the application container then instantly enables zero-downtime deployments because a new instances then can startup while the

previous instance keeps running and serving requests until the new one is ready to take over. This could be additionally supported by adding some more logic to the nginx-proxy container, for example by forwarding requests only to application containers which already passed the health check.

6.2.2 Building and testing in containers

The compilation of the application as well as the execution of the unit tests are still done using the natively installed JRE on the Bamboo server since these steps in the pipeline remain unchanged since the status quo of the beginning. But aiming for perfectly reproducible builds these steps should also be moved into Docker containers in the future adding the "build" and "test" environments to the Docker pipeline and therefore introducing the same benefits as in the other environments: isolated runtime environments containing all required dependencies, therefore allowing to provision additional instances of these environments in an easy and reproducible fashion.

6.2.3 Porting this pipeline to other projects

Since the concept of the pipeline is quite independent of the actual application only few minor changes are required to apply continuous deployment with this pipeline to other (web) applications. Changing the base image used for the application image even an application written in a completely different programming language can be supported easily.

In fact, after the implementation of the continuous deployment pipeline for planr was finished, the development of another internal web application for FlixBus was assigned to the Java team, this time also aiming at external end users. Based on the same technology stack as planr the setup of an additional continuous deployment pipeline for this new project was possible by just copying the one developed in this paper and applying minor adjustments - mostly changing the host- and domain names used.

So at the time of writing this, already two separate development projects are using this container-based continuous deployment pipeline, therefore acknowledging a certain success regarding its implementation.

List of Figures

Bibliography

[Byf14] Byfield, Bruce: *By the Bootstrap – Using debootstrap and schroot to run a chroot jail*. Linux Magazine, August 2014. http://www.linux-magazine.com/Issues/2014/165/Command-Line-Debootstrap.

[FFRR14] Felter, Wes, Alexandre Ferreira, Ram Rajamony and Juan Rubio: *An Updated Performance Comparison of Virtual Machines and Linux Containers*. July 2014. http://domino.research.ibm.com/library/cyberdig.nsf/papers/0929052195DD819C85257D2300681E7B/$File/rc25482.pdf.

[Fra15] Frazelle, Jessie: *Docker 1.6: Engine & Orchestration Updates, Registry 2.0, & Windows Client Preview*. https://blog.docker.com/2015/04/docker-release-1-6/, April 16, 2015.

[HF11] Humble, Jez and David Farley: *Continuous delivery – reliable software releases through build, test, and deployment automation*. Pearson Education, Inc., Boston, 2011.

[Hum10] Humble, Jez: *Continuous Delivery vs Continuous Deployment*. http://continuousdelivery.com/2010/08/continuous-delivery-vs-continuous-deployment/, August 13, 2010.

[Res00] Rescorla, Eric: *Request for Comments: 2818*. The Internet Engineering Task Force, May 2000. https://www.ietf.org/rfc/rfc2818.txt.

[Sub14] Subbarayan, Uday: *Linux Containers - Server Virtualization 2.0 : A Game Changer!* http://uds-web.blogspot.de/2014/09/linux-containers-server-virtualization.html, September 1, 2014.

[Wig12] Wiggins, Adam: *The Twelve-Factor App - Chapter III. Config*. http://12factor.net/config, January 2012.

Appendix

1 healthcheck.sh

This health check script is used to determine whether a new application instance started up successfully. It has to be started directly on the Docker host passing the name of the application container. The script is used by the deployment scripts described in section 5.1.5.

Listing 1: healthcheck.sh

```
#!/bin/bash

set -e

if [ $# -ne 1 ]; then
    echo "usage: $0 \$CONTAINER" >&2
    exit 1
fi

SLEEP=5
MAXWAIT=100

CONTAINER="$1"

APP_IP="$(docker inspect -f '{{.NetworkSettings.IPAddress}}' "$CONTAINER")"
VIRTUAL_PORT="$(docker inspect -f '{{range .Config.Env}}{{println .}}{{end}}' \
    "$CONTAINER" | grep '^VIRTUAL_PORT=' | sed 's/^VIRTUAL_PORT=//')"

healthcheck() {
    CHECK_PATTERNS=(
        '^\{"status":"UP".*\}$'
        '^\{.*"ldap":\{"status":"UP"\}.*\}$'
        '^\{.*"db":\{"status":"UP","database":"MySQL","hello":1\}.*\}$'
        '^\{.*"diskSpace":\{"status":"UP","free":[0-9]+,"threshold":[0-9]+\}.*\}$'
    )
    CHECK_URL="http://$APP_IP:$VIRTUAL_PORT/health"

    HEALTH="$(curl --connect-timeout 2 -timeout -s "$CHECK_URL")"

    for PATTERN in ${CHECK_PATTERNS[@]}; do
        if ! echo "$HEALTH" | grep -qE "$PATTERN"; then
            return 1
        fi
    done
    return 0
}

WAIT=0
while [ "$WAIT" -lt "$MAXWAIT" ] ; do
    running="$(docker inspect -f '{{.State.Running}}' "$CONTAINER")"
    if healthcheck; then
        echo "Application running!"
        exit 0
    elif [ ! "$running" = "true" ]; then
        echo "container $CONTAINER is not running anymore" >&2
        echo "please see \`docker logs $CONTAINER\` for details" >&2
        exit 1
    fi
    echo "Application not running!"
    WAIT=$(($WAIT + 1))
    sleep "$SLEEP"
done

echo "application did not come up after $(($SLEEP * $MAXWAIT))s" >&2
exit 1
```

2 nginx.tmpl

This file is the configuration template of the nginx-proxy explained in section 5.1.6. It is parsed by docker-gen when receiving a container start/stop event from the Docker daemon, generating the configuration file to be used by the nginx daemon inside the nginx-proxy.

Listing 2: docker-gen template of the nginx-proxy configuration

```
# If we receive X-Forwarded-Proto, pass it through; otherwise, pass along the
# scheme used to connect to this server
map $http_x_forwarded_proto $proxy_x_forwarded_proto {
  default $http_x_forwarded_proto;
  ''      $scheme;
}

# If we receive Upgrade, set Connection to "upgrade"; otherwise, delete any
# Connection header that may have been passed to this server
map $http_upgrade $proxy_connection {
  default upgrade;
  ''      '';
}

gzip_types text/plain text/css application/javascript application/json application/x-javascript text/xml application/xml application/xml+rss text/javascript;

log_format vhost '$host $remote_addr - $remote_user [$time_local] '
                 '"$request" $status $body_bytes_sent '
                 '"$http_referer" "$http_user_agent"';

access_log /proc/self/fd/1 vhost;
error_log /proc/self/fd/2;

# HTTP 1.1 support
proxy_http_version 1.1;
proxy_buffering off;
proxy_set_header Host $http_host;
proxy_set_header Upgrade $http_upgrade;
proxy_set_header Connection $proxy_connection;
proxy_set_header X-Real-IP $remote_addr;
proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
proxy_set_header X-Forwarded-Proto $proxy_x_forwarded_proto;

upstream docker-socket {
  server unix:/tmp/docker.sock;
}

{{ if (and (exists "/etc/nginx/certs/default.crt") (exists "/etc/nginx/certs/default.key")) }}

server {
  listen 80;
  {{ if $.Env.DEFAULT_HOST }}
    return 301 https://{{ $.Env.DEFAULT_HOST }}$request_uri;
  {{ else }}
    return 301 https://$host$request_uri;
  {{ end }}
}

server {
  listen 443 ssl;

  ssl_protocols TLSv1 TLSv1.1 TLSv1.2;
  ssl_ciphers ECDHE-RSA-AES128-GCM-SHA256:ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-RSA-AES256-GCM-SHA384:ECDHE-ECDSA-AES256-GCM-SHA384:DHE-RSA-AES128-GCM-SHA256:DHE-DSS-AES128-GCM-SHA256:kEDH+AESGCM:ECDHE-RSA-AES128-SHA256:ECDHE-ECDSA-AES128-
```

```
      SHA256:ECDHE-RSA-AES128-SHA:ECDHE-ECDSA-AES128-SHA:ECDHE-RSA-AES256-SHA384:\
      ECDHE-ECDSA-AES256-SHA384:ECDHE-RSA-AES256-SHA:ECDHE-ECDSA-AES256-SHA:DHE-RSA-\
      AES128-SHA256:DHE-RSA-AES128-SHA:DHE-DSS-AES128-SHA256:DHE-RSA-AES256-SHA256:\
      DHE-DSS-AES256-SHA:DHE-RSA-AES256-SHA:AES128-GCM-SHA256:AES256-GCM-SHA384:\
      AES128-SHA256:AES256-SHA256:AES128-SHA:AES256-SHA:AES:CAMELLIA:DES-CBC3-SHA:!\
      aNULL:!eNULL:!EXPORT:!DES:!RC4:!MD5:!PSK:!aECDH:!EDH-DSS-DES-CBC3-SHA:!EDH-RSA-\
      DES-CBC3-SHA:!KRB5-DES-CBC3-SHA;

  ssl_prefer_server_ciphers on;
  ssl_session_timeout 5m;
  ssl_session_cache shared:SSL:50m;

  ssl_certificate /etc/nginx/certs/default.crt;
  ssl_certificate_key /etc/nginx/certs/default.key;

  add_header Strict-Transport-Security "max-age=31536000";

{{ else }}

server {
  listen 80;

{{ end }}

  root /app/htdocs;

  location /containers/ {
    proxy_pass http://docker-socket;
    auth_basic "Restricted";
    auth_basic_user_file /app/htpasswd;
  }

  location /containers/admin/ {
    root /app/htdocs/;
    index index.html;
    auth_basic "Restricted";
    auth_basic_user_file /app/htpasswd;
  }
}

{{ range $host, $containers := groupByMulti $ "Env.VIRTUAL_HOST" "," }}

upstream {{ $host }} {
{{ range $container := $containers }}
  {{ $addrLen := len $container.Addresses }}
  {{/* If only 1 port exposed, use that */}}
  {{ if eq $addrLen 1 }}
    {{ with $address := index $container.Addresses 0 }}
      # {{$container.Name}}
      server {{ $address.IP }}:{{ $address.Port }};
    {{ end }}
  {{/* If more than one port exposed, use the one matching VIRTUAL_PORT env var */}}
  {{ else if $container.Env.VIRTUAL_PORT }}
    {{ range $address := .Addresses }}
      {{ if eq $address.Port $container.Env.VIRTUAL_PORT }}
      # {{$container.Name}}
      server {{ $address.IP }}:{{ $address.Port }};
      {{ end }}
    {{ end }}
  {{/* Else default to standard web port 80 */}}
  {{ else }}
    {{ range $address := $container.Addresses }}
      {{ if eq $address.Port "80" }}
      # {{$container.Name}}
      server {{ $address.IP }}:{{ $address.Port }};
      {{ end }}
```

```
113     {{ end }}
114   {{ end }}
115 {{ end }}
116 }
117
118 {{/* Get the VIRTUAL_PROTO defined by containers w/ the same vhost, falling back to \
        "http" */}}
119 {{ $proto := or (first (groupByKeys $containers "Env.VIRTUAL_PROTO")) "http" }}
120
121 {{/* Get the first cert name defined by containers w/ the same vhost */}}
122 {{ $certName := (first (groupByKeys $containers "Env.CERT_NAME")) }}
123
124 {{/* Get the best matching cert  by name for the vhost. */}}
125 {{ $vhostCert := (closest (dir "/etc/nginx/certs") (printf "%s.crt" $host))}}
126
127 {{/* vhostCert is actually a filename so remove any suffixes since they are added \
        later */}}
128 {{ $vhostCert := replace $vhostCert ".crt" "" -1 }}
129 {{ $vhostCert := replace $vhostCert ".key" "" -1 }}
130
131 {{/* Use the cert specifid on the container or fallback to the best vhost match */}}
132 {{ $cert := (coalesce $certName $vhostCert) }}
133
134 {{ if (and (ne $cert "") (exists (printf "/etc/nginx/certs/%s.crt" $cert)) (exists (\
        printf "/etc/nginx/certs/%s.key" $cert))) }}
135
136 server {
137   server_name {{ $host }};
138   return 301 https://$host$request_uri;
139 }
140
141 server {
142   server_name {{ $host }};
143   {{ if (and ($.Env.DEFAULT_HOST) (eq $.Env.DEFAULT_HOST $host)) }}
144     listen 443 ssl default_server;
145   {{ else }}
146     listen 443 ssl;
147   {{ end }}
148
149   ssl_protocols TLSv1 TLSv1.1 TLSv1.2;
150   ssl_ciphers ECDHE-RSA-AES128-GCM-SHA256:ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-RSA-\
        AES256-GCM-SHA384:ECDHE-ECDSA-AES256-GCM-SHA384:DHE-RSA-AES128-GCM-SHA256:DHE-\
        DSS-AES128-GCM-SHA256:kEDH+AESGCM:ECDHE-RSA-AES128-SHA256:ECDHE-ECDSA-AES128-\
        SHA256:ECDHE-RSA-AES128-SHA:ECDHE-ECDSA-AES128-SHA:ECDHE-RSA-AES256-SHA384:\
        ECDHE-ECDSA-AES256-SHA384:ECDHE-RSA-AES256-SHA:ECDHE-ECDSA-AES256-SHA:DHE-RSA-\
        AES128-SHA256:DHE-RSA-AES128-SHA:DHE-DSS-AES128-SHA256:DHE-RSA-AES256-SHA256:\
        DHE-DSS-AES256-SHA:DHE-RSA-AES256-SHA:AES128-GCM-SHA256:AES256-GCM-SHA384:\
        AES128-SHA256:AES256-SHA256:AES128-SHA:AES256-SHA:AES:CAMELLIA:DES-CBC3-SHA:!\
        aNULL:!eNULL:!EXPORT:!DES:!RC4:!MD5:!PSK:!aECDH:!EDH-DSS-DES-CBC3-SHA:!EDH-RSA-\
        DES-CBC3-SHA:!KRB5-DES-CBC3-SHA;
151
152   ssl_prefer_server_ciphers on;
153   ssl_session_timeout 5m;
154   ssl_session_cache shared:SSL:50m;
155
156   ssl_certificate /etc/nginx/certs/{{ (printf "%s.crt" $cert) }};
157   ssl_certificate_key /etc/nginx/certs/{{ (printf "%s.key" $cert) }};
158
159   add_header Strict-Transport-Security "max-age=31536000";
160
161   {{ if (exists (printf "/etc/nginx/vhost.d/%s" $host)) }}
162   include {{ printf "/etc/nginx/vhost.d/%s" $host }};
163   {{ end }}
164
165   location / {
166     proxy_pass {{ $proto }}://{{ $host }};
```

```
167     {{ if (exists (printf "/etc/nginx/htpasswd/%s" $host)) }}
168     auth_basic  "Restricted␣{{␣$host␣}}";
169     auth_basic_user_file  {{ (printf "/etc/nginx/htpasswd/%s" $host) }};
170     {{ end }}
171   }
172 {{ else }}
173
174 server {
175   {{ if (and (($.Env.DEFAULT_HOST) (eq $.Env.DEFAULT_HOST $host))) }}
176     listen 80 default_server;
177   {{ end }}
178   server_name {{ $host }};
179
180   {{ if (exists (printf "/etc/nginx/vhost.d/%s" $host)) }}
181   include {{ printf "/etc/nginx/vhost.d/%s" $host }};
182   {{ end }}
183
184   location / {
185     proxy_pass {{ $proto }}://{{ $host }};
186     {{ if (exists (printf "/etc/nginx/htpasswd/%s" $host)) }}
187     auth_basic  "Restricted␣{{␣$host␣}}";
188     auth_basic_user_file  {{ (printf "/etc/nginx/htpasswd/%s" $host) }};
189     {{ end }}
190   }
191 {{ end }}
192
193   location /error/ {
194     root /app/htdocs/;
195   }
196   error_page 502 /error/502.html;
197 }
198
199 {{ end }}
```

3 admin.tmpl

Listing all running application containers the HTML page generated from this template by docker-gen inside the nginx-proxy, the generated administration page additionally allows to stop and remove instances of the application and its respective database container as described in section 5.1.6.3. It differs from the `index.tmpl` template used for the indexing page only by adding the delete button in lines 28-30 and 45 and the required JavaScript in lines 5 and 6.

Listing 3: docker-gen template of the administration page of nginx-proxy

```
1  <?DOCTYPE html>
2  <html>
3    <head>
4      <title>Planr index</title>
5      <script src="jquery-1.11.3.min.js"></script>
6      <script src="docker-utils.js"></script>
7    </head>
8    <body>
9      <table>
10       <tbody>
11         {{ range $host, $containers := groupByMulti $ "Env.VIRTUAL_HOST" "," }}
12           {{ range $container := $containers }}
13             <tr>
14               <td><a href="//{{␣$host␣}}/" title="{{␣$host␣}}">{{ $host }}</a></td>
15               <td>{{ $container.Image }}</td>
16               <td>{{ $container.Env.SPRING_PROFILES_ACTIVE }}</td>
17               <td>{{ $container.Env.BAMBOO_BRANCH }}</td>
18               <td>{{ $container.Env.BAMBOO_COMMIT }}</td>
19               <td>{{ $container.Env.BAMBOO_PLAN }}</td>
20               <td><a href="{{␣$container.Env.BAMBOO_RESULTS_URL␣}}">{{ \
                     $container.Env.BAMBOO_BUILD }}</a></td>
21               <td>
22                 {{ range $address := $container.Addresses }}
23                   {{ if $address.HostPort }}
24                     {{ $address.HostPort }}
25                   {{ end }}
26                 {{ end }}
27               </td>
28               <td>
29                 <button type="button" class="delete-button" data-app-host="{{␣$host␣\
                       }}">Delete</button>
30               </td>
31             </tr>
32           {{ end }}
33         {{ end }}
34       </tbody>
35       <thead>
36         <tr>
37           <th>URL</th>
38           <th>Image</th>
39           <th>Config</th>
40           <th>Branch</th>
41           <th>Commit</th>
42           <th>Plan</th>
43           <th>Build</th>
44           <th>Debug Port</th>
45           <th>Actions</th>
46         </tr>
47       </thead>
48     </table>
49   </body>
50 </html>
```

4 Dockerfile for mysql-base

As explained in section 5.1.7 the mysql-base image serves as image of the production database containers as well as the base image for the mysql-integration images.

Listing 4: Dockerfile for mysql-base

```
1  FROM debian:wheezy
2  MAINTAINER Roman Anasal <roman.anasal@flixbus.de>
3
4  # add our user and group first to make sure their IDs get assigned consistently, \
       regardless of whatever dependencies get added
5  RUN groupadd -r mysql && useradd -r -g mysql mysql
6
7  # FATAL ERROR: please install the following Perl modules before executing \
       /usr/local/mysql/scripts/mysql_install_db:
8  # File::Basename
9  # File::Copy
10 # Sys::Hostname
11 # Data::Dumper
12 RUN apt-get update && apt-get install -y perl --no-install-recommends && rm -rf \
       /var/lib/apt/lists/*
13
14 # gpg: key 5072E1F5: public key "MySQL Release Engineering \
       <mysql-build@oss.oracle.com>" imported
15 RUN apt-key adv --keyserver pool.sks-keyservers.net --recv-keys \
       A4A9406876FCBD3C456770C88C718D3B5072E1F5
16
17 ENV MYSQL_MAJOR 5.6
18 ENV MYSQL_VERSION 5.6.24
19
20 RUN echo "deb␣http://repo.mysql.com/apt/debian/␣wheezy␣mysql-${MYSQL_MAJOR}" > \
       /etc/apt/sources.list.d/mysql.list
21
22 # the "/var/lib/mysql" stuff here is because the mysql-server postinst doesn't have \
       an explicit way to disable the mysql_install_db codepath besides having a \
       database already "configured" (ie, stuff in /var/lib/mysql/mysql)
23 # also, we set debconf keys to make APT a little quieter
24 RUN { \
25     echo mysql-community-server mysql-community-server/data-dir select ''; \
26     echo mysql-community-server mysql-community-server/root-pass password ''; \
27     echo mysql-community-server mysql-community-server/re-root-pass password ''; \
28     echo mysql-community-server mysql-community-server/remove-test-db select false; \
29   } | debconf-set-selections \
30   && apt-get update && apt-get install -y mysql-server="${MYSQL_VERSION}"* && rm \
         -rf /var/lib/apt/lists/* \
31   && rm -rf /var/lib/mysql && mkdir -p /var/lib/mysql && chown mysql:mysql \
         /var/lib/mysql
32
33 # comment out a few problematic configuration values
34 RUN sed -Ei 's/^(bind-address|log)/#&/' /etc/mysql/my.cnf
35
36 EXPOSE 3306
37 CMD ["mysqld"]
```