Comparing the Creation of Scalable 12 Factor Apps on Heroku's Natively Supported Web Frameworks

Thesis submitted to the Faculty of Computer Science and Business Information Systems at the University of Applied Sciences Wuerzburg Schweinfurt for the degree of Bachelor of Engineering of Computer Science

by

Christian Paling

Primary Examiner: Prof. Dr. Steffen Heinzl

Secondary Examiner: Prof. Dr. Peter Braun

Date of Submission: 01.03.2017

Statutory Declaration

I assure that this thesis is a result of my personal work and that no other than the indicated aids have been used for its completion. Furthermore I assure that all quotations and statements that have been inferred literally or in a general manner from published or unpublished writings are marked as such. Beyond this I assure that the work has not been used, neither completely nor in parts, to pass any previous examination.

DATE SIGNATURE

Abstract

Though cloud hosting platforms like Amazon Web Services, Google App Engine or Heroku are being frequently employed, it is not clear to many developers how these platforms actually work and differentiate from traditional hosting, and how applications should be developed to benefit from these platforms. To provide some answers, professionals of Heroku released the twelve-factor app methodology, a manifesto with twelve guidelines which each application should follow to make full usage of the benefits of cloud hosting.

Since web applications are generally not built from scratch but rely on a framework, three web frameworks, Play, Ruby on Rails, and Laravel, were selected which all run natively on the cloud platform Heroku, and were individually analyzed for each guideline of twelve-factor applications. Moreover, during the course of this thesis, one twelve-factor application was implemented for each framework so development speed and performance could be further evaluated through load testing.

On the basis of the results, it can be concluded that each framework provides strengths and weaknesses regarding the support of the methodology as well as the outcoming application. The Play framework pushes the responsibility to the developer to implement solutions for problems that exceed basic HTTP request and response requirements. Missing documentation has proven to make this difficult, yet rewarding by granting a high performance. Ruby on Rails provides good support to

develop twelve-factor apps because of a wide range of third-party solutions and documentation. Still developers have to adapt some internal structures of the framework to ensure it behaves in a twelve-factor way. Laravel excels with high development speed, and a wide range of built-in solutions for the guidelines of the manifesto. The resulting performance however, is inferior compared to the other offerings.

Contents

1 Introduction

Since the widespread cognizance of cloud hosting among developers and companies the usage of services like Infrastructure as a Service or Platform as a Service has been rising dramatically. While the market share of cloud platforms in Europe was 36% in 2013, the employment of cloud hosting has been increasing every year by 42% until then.[1]

There are multiple reasons for this growth. One of the biggest being the high flexibility of cloud hosting for scaling. When using a more traditional service like a virtual private server then requesting new instances is a more or less tedious process while being only a few clicks or commands on the cloud hosting side. While acquiring new production machines using cloud hosting is being made relatively fast, easy, and cost-efficient by the cloud service providers, the development and deployment side, and the architecture of the app may hinder the full potential of this solution. Some examples of the hindrance are required dependencies which developers have to set up manually or fixed configuration files which have to be changed for each machine. To offer guidelines for these problems the twelve-factor app principles where created and are therefore interesting for highly scalable applications.[2]

The thesis at hand offers a description and elaboration of the twelve-factor app methodology. Furthermore the principles will be practically implemented using a

popular cloud hosting site called Heroku and several frameworks which are natively supported by this platform. At the end of this thesis the reader should know if the methodology is suitable for an own project as well as how to apply it, by having learned about it through the implementation examples outlined in this thesis.

2 Frameworks

Heroku can officially run any web application written in Ruby, Node.js, Java, Python, Clojure, Scala, Go and PHP[3] even though applications written in other languages have already been deployed on Heroku as well.[4] The frameworks used in this thesis differ from the others because they can be deployed immediately after project generation and will run right out of the box without having to provide configuration or third-party functionalities.[3]

2.1 Play Framework

In 2007 a company called *Zenexity* developed the early versions of the Play framework as an internal project. The motivation behind it was to use a new approach for developing web applications compared to the back then usual way of using Java Enterprise Edition. In 2009 the Play framework was open sourced and was used by other companies and people for their projects as well. Because of the fast changing web development techniques the maintainers of Play decided to create a second version to make Play a framework which is up to date with standard web development practices.[5]

Play has many characteristics which are worth mentioning. First of all it enables programmers to develop their web API using asynchronous processing. To provide

asynchronous HTTP programming, the framework is built on top of the Netty network application framework[1]. This leads every endpoint in Play to be non-blocking and asynchronous by default.[5]

Furthermore, Play relies on type safety not only for the business logic and controllers but for templates and routes as well. A compiler can thereby detect mistakes early and help developers working on a larger project.[5]

Since the early development of Play, the maintainers of the framework were experimenting with Scala as a programming language to develop Play applications. In version 2 the Scala language was moved to the core of the project, giving developers the possibility to use Scala or Java to create Play applications while fully leveraging the features of Play and of the programming languages.[5]

Lastly Play emphasizes on a container-less deployment approach which made it differ from the standard Java Enterprise Edition conventions and enabled it to run natively on Heroku.[5]

In this thesis version 2.5.10 of the Play framework as well as Scala 2.12.1 will be used.

2.2 Ruby on Rails

In 2004 David Heinemeier Hansson created and released the Ruby on Rails web framework, often shortened to "Rails", while working on a new online project management tool called Basecamp[2]. The framework was built with the, during this time unknown, Ruby programming language.[6] Through the years the Rails framework

1 `http://netty.io/`

2 `https://basecamp.com/`

became more and more popular, especially among startups, because of its ease of use and high development speed.

The Rails framework excels in the high amount of available packages called *gems*, and in the amount of functionalities it offers like an ORM, security implementations, integrated testing, and more. Over the time the Rails framework has influenced many web frameworks in other programming languages like Grails[3], Sails.js[4] or Swifton[5].

Since January 2016 the Rails framework has its own doctrine which specifies key characteristics of the Ruby on Rails framework.[6] An example is the *Convention over Configuration* paradigm. A Rails developer can switch projects and feel at home because of a similar project structure caused by this methodology.

While Rails offers many solutions to implement an app using the twelve-factor app methodology it does not follow all principles out of the box as we will see in the next chapter.

The code samples of this thesis were written using Ruby 2.3.1 and Ruby on Rails 5.0.0.1.

2.3 Laravel

After the release of Ruby on Rails in 2004 it was clear for developers that using a web framework was the way to create web applications in the future. Shortly afterwards in 2005 the first PHP frameworks started to appear, among them a framework called *CodeIgniter*. CodeIgniter was one of the most popular PHP frameworks but

[3] `https://grails.org/`

[4] `http://sailsjs.com/`

[5] `https://github.com/necolt/Swifton`

[6] `http://rubyonrails.org/doctrine/`

its speed of adoption of modern web technologies was too slow. It happenend then that a frustrated CodeIgniter developer called Taylor Otwell created a framework of his own and called it *Laravel*. Through the years new versions of Laravel have been released until version 5 which is the latest at the date of this thesis.[7]

Offering tools like all previously mentioned frameworks for database management, authentication or security, there are four core concepts in Laravel: service containers, service providers, facades and contracts. A *service container* is a tool in Laravel to manage class dependencies as well as to make use of dependency injection. This concept enables a developer to inject a certain functionality into a class where the injected implementation can be easily swapped. As an example, an object to retrieve user data from a source can be injected into a users endpoint controller. During development and production this source can be an implementation using the database which can easily be swapped to a memory based source for testing purposes.[8] *Service providers* are the central places where a Laravel application gets configured as well as the service containers get registered using the required implementations.[9] To have an easy syntax, Laravel makes use of *facades*. These facades offer expressive and memorable static methods to underlying classes. The framework itself is being shipped with many facades e.g. for the filesystem, caching or logging.[10] Lastly Laravel defines interfaces for core functionalities the framework offers called *contracts*. By using the contracts provided by Laravel a developer can use implementations offered by Laravel itself or create an implementation on his own while sharing a common API.[8]

The code samples in this thesis were written in PHP 7.1.0 and Laravel 5.3.28.

3 Heroku

Initially Heroku started out in 2007 as a platform to easily deploy applications using the uprising Ruby on Rails framework which was not easy to configure on a production environment during this time. Starting from this point Heroku emerged to become a cloud platform provider focusing on simple deployment and scaling mechanisms, supporting multiple programming languages, and enabling developers to focus on their code and not care about the infrastructure of their production system anymore.[11]

Heroku provides so called *dynos* as lightweight Linux containers where developers can execute their code. Each dyno runs a single command and belongs to a certain process type.[12] The term process type is connected to twelve-factor applications and will be explained in detail in the concurrency section.

There are three different types of dynos:

- Web dynos are special because they belong to the *web* process type. Dynos of this process type are run automatically by Heroku. Furthermore they are the only dynos which can receive HTTP traffic.[12]

- Worker dynos handle all other process types which are declared in the configuration file for Heroku called *Procfile*. Examples for these are background jobs or scheduled jobs.[12]

- One-off dynos are being used for temporary tasks like database migrations or administrative tasks.[12]

The already mentioned Procfile configures the dynos by specifying the command a dyno should execute. In each line the developer can declare one of his processes which he wants to run on Heroku. A declaration of a process type looks as follows:[13]

```
1 <process type>: <command>
```

Except for the web process type, its name can be chosen arbitrarily. Heroku will automatically run the web process in a web dyno while dynos for other process types have to be invoked manually by the developer.[13]

In order to create, manage, and scale applications on Heroku, the developer needs the Heroku CLI also widely known as the Heroku toolbelt[1]. By using the Heroku CLI a lot of the features which Heroku provides is accessible using the command line. Examples of these are viewing logs, executing tasks, managing the database, or scaling.

Lastly the platform provides a vast set of add-ons which can be acquired for an application. These add-ons are either provided by Heroku itself or by third-party developers. Examples of available add-ons include Elasticsearch, Redis or RabbitMQ.[14]

[1] https://devcenter.heroku.com/articles/heroku-command-line

4 Twelve-Factor Apps

This chapter is an analysis of all factors of the twelve-factor app methodology. Each factor will be explained and the advantages it offers will be outlined. Furthermore for each guideline examples will be shown on how to apply the factor using the frameworks which were listed two chapters earlier as well as how Heroku supports these factors. If a certain guideline concerns a general development strategy and is in the area of responsibility of a web framework, the specific implementations will be omitted.

4.1 Codebase

> ***"One codebase tracked in revision control, many deploys"*** [1]

A twelve-factor app has to be tracked in a version control system where the *codebase* is any single repository, either the single shared repository when using a centralized system or one of the repositories on the developers machines or a remote machine when using a decentralized system.[15]

[1] `https://12factor.net/codebase`

In recent years a decentralized system, called *Git*, has surpassed *Apache Subversion* in popularity. Subversion was the most popular version control the years before which uses a centralized approach. Today only three version control systems remain relevant: Mercurial, Git and Apache Subversion.[16]

Furthermore two important principles apply to twelve-factor apps when using a source control system:

- Different apps are not allowed to share the same code. If the identical code is being employed across multiple apps, then this code should be extracted as a library and used as a dependency for these applications.[15]
- One codebase correlates to one app. Multiple different codebases are not a single app but a distributed system.[15]

On the other side, a single app might run in multiple environments where each running instance is called a *deploy*. Deploys might be located on the developers machine, on a production server or in an isolated testing environment. While the codebase is always the same across all deploys, the version for each running instance might differ. The production instance can run an older version of the app than on a staging environment or on the developer's machine.[15]

The reasons for this guideline is that having a single codebase for each app makes an automated build and deployment much easier than managing and building multiple codebases into an artifact and deploying it. Furthermore when having multiple applications in a single codebase then often multiple teams are working on the codebase. Having a different organisation of teams and architecture of the

system is prone to dysfunction and creating problems with the resulting system in respect to Conway's law:[17, p. 2]

"Organizations which design systems are constrained to produce systems which are copies of the communication structures of these organizations."[18]

In these cases the applications in the codebase should be split among the teams into separate applications or microservices which use different codebases.[17, p. 2]

Heroku enables the developer to deploy his app by pushing his repository to a Heroku remote. As a version control system Heroku only supports Git. To create the Heroku remote, the CLI offers a command called `heroku create`. After execution, the application can be deployed to Heroku by running `git push heroku <branch-name>`. For other systems like Apache Subversion, a Git repository has to be created where all Subversion configuration files and folders have to be ignored to be able to be deployed to Heroku.[19]

4.2 Dependencies

"Explicitly declare and isolate dependencies" [2]

Through the years developers and operations have been used to server environments which provide all dependencies that an application needs in order to run, a concept known as the *mommy server*.[17, p. 9]

[2] `https://12factor.net/dependencies`

A twelve-factor app never expects the instance on which it runs to offer any dependencies which the app requires. The developer has to vendor all dependencies which are needed into the app. In order to achieve this correctly an application needs two tools:[20]

- All dependencies have to be declared using a *dependency declaration manifest* which a tool can read and on which it can act accordingly.[20]
- During execution the dependencies have to be isolated from the surrounding system to ensure that no implicit dependencies from the system "leak in", a concept called *dependency isolation*.[20] Dependency isolation tools do this by bundling the dependencies into directories of the app or into the artifact itself.[17, p. 10]

When relying on the system to offer dependencies, problems will arise when these dependencies are not met. Either they could not be installed at all or the version between different deploys could differ which might raise exceptions and errors during the runtime of the application.[20]

The benefit gained by managing dependencies this way is a simplified setup. Creating a new runtime instance of the application and installing the dependencies is as easy as executing the command of the tool which reads the dependency declaration manifest and installs the specified dependencies.[20]

4.2.1 Play

Traditionally the Play framework offered a command called *play* to create new applications or run tests. Since version 2.3.0 this tool is discontinued and replaced with the *activator* tool to manage Play applications.[21]

Activator is capable of much more than the former *play* command. It is built upon the *Scala Built Tool*, widely known as *sbt*, and provides all of the commands which sbt defines, as well as a few more features.[22]

Both sbt and activator use the *build.sbt* file to declare its dependencies.

```
1  name := """reminder-app-play"""
2
3  version := "1.0-SNAPSHOT"
4
5  lazy val root = (project in file("."))
6      .enablePlugins(PlayScala)
7
8  scalaVersion := "2.11.7"
9
10 libraryDependencies ++= Seq(
11   jdbc,
12   cache,
13   ws,
14   "org.scalatestplus.play" %% "scalatestplus-play" % "1.5.1"
15 )
```

Listing 4.1: *build.sbt* file of a new Play application scaffold

The file defines certain attributes like the name of the application or the currently used scala version with which it will compile. Furthermore it enables Play framework functionalities and already adds a few dependencies which the Play framework uses.

To fetch dependencies, activator uses the Maven 2 repository[3] and the Typesafe

[3] https://repo.maven.apache.org/maven2/

releases repository[4] by default. Further repositories can be added by including a name and location to the resolvers field using the following syntax: `resolvers += <name> at <location>`[23]

```
resolvers += "sonatype snapshots" at "https://oss.sonatype.org
    /content/repositories/snapshots/"
```

Listing 4.2: Adding the sonatype repository

To declare a dependency, activator uses the *libraryDependencies* field. Dependencies have to be comma separated and can either by declared in values and then added or can be included directly using a valid declaration syntax.[23]

```
jquery = "org.webjars" % "jquery" % "3.1.1-1"

libraryDependencies ++= Seq(
  // other dependencies
  jquery
)
```

Listing 4.3: Add a new dependency using a value

```
libraryDependencies ++= Seq(
  // other dependencies
  "org.webjars" % "jquery" % "3.1.1-1"
)
```

Listing 4.4: Add a new dependency directly

[4] https://dl.bintray.com/typesafe/maven-releases/

For the declaration syntax the *group*, *artifact* and *version* are separated by a % sign resulting in `<group> % <artifact> % <version>`. Furthermore the currently used Scala version can be added to the artifact declaration by using a double % sign after the group declaration. This becomes relevant when a dependency was compiled using multiple Scala versions and the one relevant for the project should be used.

```
1 libraryDependencies ++= Seq(
2   // other dependencies
3   libraryDependencies += "org.scala-tools" %% "scala-stm" % "
     0.3"
4 )
5
6 // equals to
7
8 libraryDependencies ++= Seq(
9   // other dependencies
10  libraryDependencies += "org.scala-tools" % "scala-stm_<
     current_version>" % "0.3"
11 )
```

Listing 4.5: Add the currently used Scala version to the artifact

When fetching its dependencies activator downloads them to a *.ivy2* folder in the home directory. Since all projects share these dependencies Heroku recommends to use the sbt native packer plugin[5] which is enabled in Play since version 2.3 by default.[24]

The plugin provides a *stage* command for activator which compiles the application, downloads and bundles all dependencies and saves the output files in the

5 https://github.com/sbt/sbt-native-packager

target/universal/stage folder of the project.

```
$ activator stage
$ ls target/universal/stage/lib
com.google.guava.guava-19.0.jar
com.google.inject.guice-4.0.jar
...
```

Listing 4.6: The stage command

By using this plugin all dependencies are isolated for each project which makes it thereby compliant to the dependencies guideline of twelve-factor apps.

4.2.2 Ruby on Rails

For Ruby and the Ruby on Rails framework a popular dependency management tool exists, called *bundler*[6]. Bundler expects a file called *Gemfile* in the project's root directory that acts as the dependency declaration manifest and can be created using the `bundle init` command.

[6] `https://bundler.io/`

A typical Gemfile looks as follows:

```
source 'https://rubygems.org'

gem 'rails', '~> 5.0.0'
gem 'pg', '0.18'
gem 'puma', '>= 3.0'

group :development, :test do
  gem 'byebug', platform: :mri
end

group :development do
  gem 'web-console'
end
```

Listing 4.7: Example of a Gemfile

At the top of the Gemfile bundler expects the source for its dependencies which is RubyGems[7] by default. Companies can change the source or add additional sources which can point to their private gem hosting site where custom gems or closed source gems are located.[25]

It is possible to declare gems using various options. The version of the gem can be specified using a static string like the *pg* gem in listing 4.7, a tilde which allows the gem to be updated by only minor versions but not major versions, or with greater (than) and less (than) signs.[26] Furthermore options to add git repositories from which the gem should be cloned or the Ruby platform which the gem supports, are available as well. Since multiple runtime environments for Ruby exist like the

[7] `https://rubygems.org`

default MRI or JRuby which runs Ruby on the JVM, not all gems are compatible for all runtime environments. The platform option ensures that a gem is only being installed when using the right platform.[25]

When initially installing dependencies with the `bundle install` command, bundler loads all gems of the Gemfile with the newest possible versions which are allowed by the declaration, and creates a *Gemfile.lock* where all gems with their concrete versions are defined. This way another `bundle install` command on a different machine can use the exact same versions for these dependencies when a project already contains a *Gemfile.lock*.[27]

When running `bundle update` bundler updates all gems if a new version exist for a gem, which is still allowed by the Gemfile specification, and updates the Gemfile.lock as well. Since certain dependencies like build or debugging tools are only used during development, they can be specified in groups and excluded by running e.g. `bundle install --without development` when they are not required to be installed.[25]

Heroku supports the specification of the Ruby version and platform in the Gemfile. When not specifying the Ruby environment at all, Heroku will use one version below the currently latest version of the MRI platform. The Ruby version specification might look as follows:[28]

```
1 ruby "2.2.2", :engine => "jruby", :engine_version => "9.0.0.0"
```

Listing 4.8: Example of a Ruby specification in a Gemfile

When deploying the application, the platform automatically installs all dependencies declared in *Gemfile.lock* so the environment uses the same gems as during

development. Furthermore through an environment variable certain Gemfile groups can be excluded from being installed. To exclude the *development* and *test* groups, the variable must be set to the following value:

```
1 BUNDLE_WITHOUT="development:test"
```

Listing 4.9: Excluding development and test groups on Heroku

4.2.3 Laravel

To manage its dependencies, Laravel uses the *composer* tool which is, according to its documentation, "strongly inspired by node's npm and ruby's bundler". It therefore does not install dependencies system-wide but manages them per-project using a folder typically called *vendor*.[29]

As its dependency declaration, composer expects a file in the project root named *composer.json*. This file contains metadata as well as all dependencies using the JavaScript Object Notation.

```
{
    "name": "bakku/reminder-app-laravel",
    "description": "The Laravel version of the 12factor thesis
    application.",
    "keywords": ["twelve", "factor"],
    "license": "MIT",
    "type": "project",
    "require": {
        "php": ">=5.6.4",
        "laravel/framework": "5.3.*"
    },
    "require-dev": {
        "fzaninotto/faker": "~1.4",
        "mockery/mockery": "0.9.*",
        "phpunit/phpunit": "~5.0",
        "symfony/css-selector": "3.1.*",
        "symfony/dom-crawler": "3.1.*"
    }
}
```

Listing 4.10: Example of a composer.json

Other than certain self-explanatory metadata, the file contains a *require* and a *require-dev* field. The former contains all libraries which are generally needed while the latter only defines dependencies for development. With `composer install` or `composer install --no-dev` the dependencies will be downloaded and unpacked into the *vendor* folder of the project.[30]

As with bundler, the composer tool supports certain version specifiers like fixed versions, wildcards or a tilde which has the same meaning as with bundler. Furthermore after the initial installation it generates a *composer.lock* file which contains

all used dependencies with their fixed versions, so the exact same libraries will be used across deploys. The *composer.lock* file can be updated using the `composer update` command which will only update libraries where a new version exists which is compliant with the *composer.json* file.[30]

Composer uses the Packagist[8] repository as a source for its dependencies. Packages are distributed using a vendor name and a package name which results to the following declaration syntax: `<vendor>/<package>`.[30]

As with Rails, when pushing a Laravel application to Heroku, the platform uses the *composer.lock* file to resolve all dependencies and install them.[31]

Composer provides a way to declare dependencies and install them, as well as a dependency isolation mechanism by downloading all libraries only project wide. Therefore the way Laravel chose to manage its dependencies is generally compliant with the twelve-factor app methodology. Yet the framework differentiates itself compared to the others of this thesis regarding the application server. Laravel ships with a development server that can be started by executing `php artisan serve`. Since this server is intended for only development usage it contradicts against keeping tools across environments the same which is further elaborated in the dev/prod parity section.[32] Since Heroku uses the Apache web server by default for PHP applications, an instance of the server should be bundled with the codebase on the side of its libraries as well.[31]

In order to achieve this, the application of the thesis defines an image for development using the Docker platform. Docker can be described as a lightweight virtualization platform where developers can define images which bundle code and

[8] `https://packagist.org/`

necessary binaries that can then be shipped across different environments and can be run by Docker in containers.[33] The Docker image is declared in a so called *Dockerfile*:

```
1 FROM php:7.1-apache
2 COPY ./docker-apache2.conf /etc/apache2/apache2.conf
3 RUN a2enmod rewrite
4 RUN apt-get update
5 RUN apt-get install -y postgresql-server-dev-9.4
6 RUN docker-php-ext-install pdo_pgsql
```

Listing 4.11: Example of a Dockerfile for a Laravel application

At the beginning the base image is defined. The Docker Hub provides a wide range of base images which developers can use to build their own specific image. The base image in listing 4.11 contains an Apache web server with PHP version 7.1 installed. After copying a configuration file for the Apache web server into the image, the PDO extension for PostgreSQL is being installed to be able to use the PostgreSQL database with the Laravel application.[34]

By being part of the codebase, the Dockerfile enables developers to distribute more heavy-weight environments, as needed for Laravel applications.

4.3 Config

"Store config in the environment" [9]

A typical application has many configuration values defined which can vary between deploys. Configuration values might be:

- URLs to external services like databases or mail servers.[35]
- Credentials like API keys, usernames or passwords.[35]
- Per-deploy values like hostname, code reloading or caching strategies.[35]

There are three popular strategies to handle configuration values in an application. They can be part of the code, extracted into separate files like XML or JSON, or be defined as environment variables.[35]

The twelve-factor app methodology makes a clear distinction between configuration values which might be different, and configuration values which are the same across deploys. When talking about the former type of configuration values, the first two strategies which were previously described, are not compliant with this guideline.[35]

The reason why configuration values should not be part of the code is that code should be the same across deploys while configuration values might not.[35] A possibility to achieve this on code basis would be to use conditionals and choose configuration values specific for the current deploy. This approach would require

[9] `https://12factor.net/config`

developers to change the code every time a new deploy with different configuration values is planned and is therefore not recommended.

Having separate environment files improves the former approach so the code does not have to be changed. Still for every new deploy a different file has to be created which is not easy to manage when the amount of deploys reaches a certain level. The approach many frameworks like the Ruby on Rails framework uses, where configuration values are grouped to certain named environments, like development, test and production, is in the same category as the former strategy. For every new environment, new files have to be created.[35]

A twelve-factor app stores its configuration in environment variables. Environment variables are easily changeable across deploys, without having to touch any code or files which are part of the codebase.[35]

Kevin Hoffmann proposes a suitable test to check whether an app is compliant to this factor or not:

"A litmus test to see if you have properly externalized your credentials and configuration is to imagine the consequences of your applications source code being pushed to GitHub."[17, p. 18]

Managing environment variables on Heroku is made convenient by the CLI, using a sub-command called `heroku config`. On execution the command returns all current environment variables set for this app. `heroku config:set AWS_S3_USERNAME=heroku` will set a new key-value pair with `AWS_S3_USERNAME` as key and `heroku` as value which can again be retrieved with `heroku config:get AWS_S3_USERNAME`. Deleting the environment variable is performed by executing `heroku config:unset AWS_S3_USERNAME`. Both when unsetting and setting variables, Heroku restarts the application.

While managing environment variables on Heroku is comfortable, a simple mech-

anism has to be found for the development side as well. The problem with using environment variables during development is that they should be scoped for each application and not be globally accessible. Certain variable names as `DATABASE_URL` tend to occur nearly in every application which results in a risk of conflicting configuration variables.

A possible solution would be to move this complication to the user level. More specifically, each app gets a personal user account to have its own set of configuration which are truely part of the environment. An example on how to achieve this using a Unix/Linux system follows:

```
1 $ sudo useradd -m <app-name>
```

Listing 4.12: Creation of a new user account for an application

By passing the *-m* option the account gets an own home directory where shell initialization files are created which can be utilized to configure the environment.

```
1 $ echo "export DATABASE_URL=postgres://user:password@localhost
    :5432/database" >> .bashrc
2 $ exec $SHELL # reload .bashrc
```

Listing 4.13: Add a new environment variable using the .bashrc file

When working on several applications multiple shells can be opened to change the user for each shell depending on the application that is currently under development.

```
$ su <app-name>
```

Listing 4.14: Switch to the account specific by app-name

After having found a way to manage the configuration during development the question arises how each framework supports the usage of environment variables and how they can be integrated into the code. To illustrate this, solutions for configuring the database as well as other backing services will be outlined.

4.3.1 Play

Generally Play uses configuration files to assign its settings. By default this file is called *application.conf* and is located in the *conf* folder of the project. According to the methodology, its key-value pairs should be changed to use environment variables with the following syntax:

```
play.crypto.secret = ${APPLICATION_SECRET}
```

Listing 4.15: Example of an env var setting

The Play framework is mostly unopinionated when it comes to the database layer of the application. It provides configuration settings which can be assigned to specify the location as well as the credentials of the database. The framework will then create connections to this database and provide an API to access it.[36]

The default configuration file contains a *db* section which is responsible for configuring data sources. Since Heroku uses the PostgreSQL database by default, the examples of all frameworks in this thesis will use the same database system as well.

The underlying technology of Play for the database employs JDBC, so the driver which the framework should use must be configured as well.[36]

The resulting *db* section looks as follows:

```
db {
    default.driver = org.postgresql.Driver
    default.url    = ${DATABASE_URL} # environment variable
}
```

Listing 4.16: Configuring the database

Because it is possible to provide multiple data sources to the framework, it uses names to differentiate between these sources. When instantiating a connection to the database in the code however, the framework provides, if not specified otherwise, the source called *default*.[36]

Because the database in this case should generally be PostgreSQL and not differ across deploys, the driver can be provided without an environment variable in contrast to the database URL.

A possible solution to manage the testing database is providing another data source called *test* and using dependency injection to inject the correct database configuration. A different approach uses the *sbt* tool and a separate configuration file to decouple this variation for the testing environment from the code.

Configuration files of Play applications can import other configuration files as well as override their values.[37] This functionality leads to the solution to create a separate configuration file for testing, e.g. *conf/application.test.conf*, to import the default *application.conf* and to override its database values using a environment variable with the URL of the testing database.

```
include "application.conf"

db.default.driver = "org.postgresql.Driver"
db.default.url    = ${TEST_DATABASE_URL}
```

Listing 4.17: *conf/application.test.conf*

Since *sbt* knows if tests are being run, the new configuration file can be explicitly used in the test environment.

```
javaOptions in Test += "-Dconfig.file=conf/application.test.
    conf"
```

Listing 4.18: *build.sbt*

This solution is not only applicable when configuring the database but for any backing service or setting. To gain access to configuration settings, an instance of the Configuration class provided by the framework can be obtained by using dependency injection.

```
// application.conf
mail.user = ${MAIL_USER}

// application.test.conf
mail.user = ${TEST_MAIL_USER}
```

Listing 4.19: A custom setting in both configuration files

```
class UserMailer @Inject() (configuration: Configuration) {
  def sendMail() {
    val from = configuration.underlying.getString("mail.user")
    ...
  }
}
```

Listing 4.20: Read from the configuration

Each configuration setting can be assigned in the default configuration file to a general environment variable and can be overriden by the separate test file with a test environment variable if it should differ in the testing environment.

4.3.2 Ruby on Rails

Ruby on Rails provides multiple ways for configuration. Because of the Convention over Configuration approach of the framework, the location of these configuration files are predefined.

- **Application Configuration**: The file contains configuration across all environments and is located at *config/application.rb*.[38]
- **Environment Configuration**: A file for each environment can be created using the name of the environment as filename. When generating a new Ruby on Rails project, three environment configuration files will be created: *development*, *test* and *production*. These files should be located in the *config/environment* folder.[38]
- **Initializers**: These configuration files are usually modified to configure libraries of the project. They will be parsed after the whole application with all

libraries has finished loading. The files can be arbitrarily named and should be located in the *config/initializers* directory.[38]

- **YAML files**: When generating a project Rails automatically creates two YAML files, *database.yml* and *secrets.yml*. While the former specifies the location and credentials of the database, the latter can be used for secret data like backing service credentials.[38]

By analyzing the way a Ruby on Rails application is being configured, two important concepts can be noticed. Apart from the YAML files, all Ruby on Rails configuration is written itself in Ruby. This will introduce a problem when extracting settings to environment variables which will be outlined further on. Secondly the framework groups its configuration heavily by using the concept of environments. This is apparent because of the environment specific configuration files as well as the YAML files where groups exist for each environment.

```
1 development:
2   secret_key_base: 42342fdfd3fs...
3
4 test:
5   secret_key_base: 34fds34ndsf3fsd...
```

Listing 4.21: Example of a YAML file: *secrets.yml*

Ruby on Rails uses an environment variable named `RAILS_ENV` which specifies the current environment and therefore the configuration which the framework should load. This environment defaults to *development* if not specified otherwise and will be automatically set to *test* when running tests.[38]

Since the environment based approach is not compliant with the twelve-factor app methodology, the way the framework groups its configuration has to be adapted. On the other hand, extracting all configuration out of the environment specific files would create a lot of environment variables. A more sophisticated approach would be to use the three generated files, *development.rb*, *test.rb* and *production.rb* and use them as common groups for development, testing and running the application. Heroku supports this approach by setting the `RAILS_ENV` variable to *production* by default when deploying a Ruby on Rails application. Practically this means to run e.g. a *staging* environment using the `RAILS_ENV` variable set to *production* as well. In case certain settings should differ between *staging* and *production* environment then they should be extracted to environment variables and be configured in the *config/application.rb* file. The following example illustrates this approach by extracting two settings which are likely to differ between *staging* and *production* environment: the log level and the behavior to hide/show full error reports.

```
1 module ReminderAppRails
2   class Application < Rails::Application
3     config.consider_all_requests_local = (ENV['
    FULL_ERROR_REPORTS'] == 'true')
4     config.log_level = ENV['LOG_LEVEL'].to_sym
5   end
6 end
```

Listing 4.22: The application wide configuration: *config/application.rb*

In Ruby the ENV constant contains all environment variables. Since the values are strings they have to be transformed to the specific data type which the configuration setting expects. The log level for example expects a Ruby symbol like `:debug`,

therefore the environment variable has to be converted to a Ruby symbol.

These two configuration settings can be deleted in the *production.rb* file but can stay untouched in *test.rb* and *development.rb* since full error reports and log level set to debug is usually desirable during development and testing.

On the contrary, database credentials usually differ between all machines on which the application is run. They should therefore be extracted to an environment variable. The Ruby on Rails framework automatically uses an environment variable called `DATABASE_URL` if no location and credentials are specified in the *database.yml* file.[38]

The generated *database.yml* file can therefore be truncated to the following lines:

```
1 test:
2   url: <%= ENV["TEST_DATABASE_URL"] %>
```

This will employ the database behind the `DATABASE_URL` environment variable by default and for testing the database behind the `TEST_DATABASE_URL` variable. Since the framework will set `RAILS_ENV` to *test* by its own when running a test, the correct database will be used.

Lastly, for the configuration of backing services or other credentials a technique is to bind keys of the *secrets.yml* file to environment variables. The YAML file is changed to use a default set of key value pairs and injects them into every environment group.

```
default: &default
  secret_key_base: <%= ENV["SECRET_KEY_BASE"] %>

test:
  <<: *default
  backing_service: <%= ENV["TEST_SOME_SERVICE"] %>

development:
  <<: *default
  backing_service: <%= ENV["SOME_SERVICE"] %>

production:
  <<: *default
  backing_service: <%= ENV["SOME_SERVICE"] %>
```

Listing 4.23: The resulting *secrets.yml* file

New data where testing and development can be equal will be added to the default group otherwise the *test* environment gets a separate environment variable. The values are available globally in the application through `Rails.application.secrets.<key>`[10].

This way no secret configuration data is exposed in the repository and the application embraces the high flexibility which comes with this factor of the methodology.

4.3.3 Laravel

Laravel stores its configuration files in the *config* directory of the application. The framework divides configuration settings into different files like *database.php* for database configuration or *mail.php* for mailer settings. By default Laravel makes

[10] `http://apidock.com/rails/v4.2.7/Rails/Application/secrets`

heavy use of environment variables which are generally retrieved using the `env` helper: `env(ENV_VAR, DEFAULT_VALUE)`.

```
'host' => env('MAIL_HOST', 'smtp.gmail.com')
```

Listing 4.24: Example of a Laravel configuration setting

This enables developers to use a general default setting and override it in case it differs on a different machine using an environment variable.

Serving environment variables to the Laravel application while staying align with the methodology proves to be not straight forward. A web server like Apache should not be managed separately but be vendored with the application, yet defining environment variables in e.g. a Docker image where the server is being run, would expose them in the code repository. An acceptable solution would be to utilize Laravel's *.env* files which are ignored by the Git source control system because of the generated *.gitignore*. Environment variables defined in these files will be loaded upon start up and will appear to the application as if the values came from the environment itself.[39]

```
APP_KEY=base64:5FpPlZk0kCfA2PahRe9TMG+Iyl6QR2wSntrq6YZ8NXQ=
DATABASE_URL=postgres://user:pass@localhost:5432/db
```

Listing 4.25: Example of *.env*

Since Heroku generally uses a DATABASE_URL variable to provide the application the location of the database, the Laravel configuration has to be adapted since it expects distinct variables for all available database credentials like the user, the

password or the port.

```
<?php

$db_url = parse_url(env('DATABASE_URL'));

$host = $db_url['host'];
$port = $db_url['port'];
$user = $db_url['user'];
$pass = $db_url['pass'];
$database = substr($db_url['path'], 1);
...
'default' => env('DB_CONNECTION', 'pgsql'),
...
'connections' => [
  'pgsql' => [
    'driver' => 'pgsql',
    'host' => $host,
    'port' => $port,
    'database' => $database,
    'username' => $user,
    'password' => $pass,
    'charset' => 'utf8',
    'prefix' => '',
    'schema' => 'public',
    'sslmode' => 'prefer',
  ]
]
```

Listing 4.26: Extract of the *config/database.php* file

In listing 4.26 the provided URL is split into its subcomponents and saved into

variables. Furthermore a new connection is being defined using the subcomponents of the URL and is set to be utilized by default.

While the database URL for development can be saved in the *.env* file, a separate database URL for the testing database can be written to a *.env.testing* file. When running tests Laravel will automatically read the testing file and use the testing database. Generally *.env.testing* should contain all environment variables which should override variables of *.env* for the testing environment.[39]

Although being not completely compliant to the methodology because of its language specificity and risk of being checked into code repositories, the *.env* files provide a simple and convenient way to configure Laravel applications.

4.4 Backing Services

"Treat backing services as attached resources" [11]

When developing a more complex system, an application generally relies on multiple *backing services*. Examples of backing services are databases like MySQL or MongoDB, caching systems like Redis, or storage services like Amazon S3. For a twelve-factor app these backing services are also called *resources*.[40]

There is an important principle for a twelve-factor app when using backing services:

An application can have local backing services, like a database running on the same machine and backing services which the app accesses over the network like an

[11] `https://12factor.net/backing-services`

external API. The app itself should not make a difference between local backing services and external backing services. Both should be accessed using a certain locator mechanism like a URL.[40]

Kevin Hoffmann further elaborates how severe a twelve-factor app should embrace this principle: "[...] the filesystem must be considered ephemeral, you also need to treat file storage or disk as a backing service."[17, p. 25]

Combined with the configuration approach of the methodology, this strategy offers two advantages:

- The backing services are loosely coupled to the application. This enables a developer to easily switch e.g. between a local PostgreSQL database to an Amazon RDS instance without having to change the codebase.[40]
- Because of the loose coupling the application does not have to be redeployed after the detachment and attachment of a new resource.[17, p. 26]

Heroku offers backing services under the term of *add-ons*. These add-ons are offered either by the platform itself or by third-party providers. By default each app on Heroku is created with already one backing service as an add-on, a PostgreSQL database.[14]

To manage add-ons the CLI offers the `heroku addons` command which will be further illustrated in the following part by showing the usage of the *Heroku Redis*[12] add-on.

When talking about addons on Heroku, it is important to distinguish between a *resource* and an *attachment*. A resource is an instance of an addon while an attach-

[12] `https://elements.heroku.com/addons/heroku-redis`

ment can be compared to a connection between this resource and the application.[14]

```
$ heroku addons:create heroku-redis
```

Listing 4.27: Create a new addon

When creating a new addon as in the previous listing, the platform will choose a random name for the service and attach it to the application using the default value, in this case `REDIS`.[14]

```
$ heroku addons
...
heroku-redis (redis-solid-49153)  hobby-dev  free  created
 |- as REDIS
```

Listing 4.28: List all addons

The name of the resource, here *redis-solid-49153*, can be modified to help the developer to structure his resources when developing larger applications. The name of the attachment however, is used as a prefix for all environment variables of the addon.[14]

```
$ heroku config
...
REDIS_URL:  redis://user:pass@server:port
```

Listing 4.29: Configuration values

In case of the Heroku Redis addon, only one environment variable will be set, the `URL` variable. Since the resource is attached to the application as `REDIS`, the

resulting environment variable is `REDIS_URL`. The name of the attachment can be modified when using a library which expects the environment variable by a certain name. Since the underlying resource may change its location, the addon manages the environment variables and changes them if needed. Further functionalities of the `heroku addons` command include deleting resources, detaching them, or changing performance/cost plans.[14]

4.5 Build, release, run

> ***"Strictly separate build and run stages"*** [13]

From development to deploy, a twelve-factor app walks through three stages: the *build*, the *release*, and the *run* stage. These stages should only direct to to next stage without the ability to go from one stage to the previous one. More specifically this means that the build stage leads to the release stage and the release stage leads to the run stage.[41]

During the build stage, the code base is being transformed into an artifact often done by a continuous integration tool.[41] An important characteristic of this artifact is that it is immutable. Combined with the tenth factor of the methodology, this attribute minimizes the probability of a complication which Kevin Hoffmann calls the "works on my machine" problem.[17, p. 15]

In the release stage the artifact is combined with the configuration for a specific deploy and is afterwards ready to be executed. This release should be identified by some versioning mechanism like a timestamp or an incrementing number as well

[13] `https://12factor.net/build-release-run`

as being immutable like the build. This enables the developer to better be able to rollback a release to a previous release.[17, p. 15]

Lastly, in the run stage the environment executes some of the app's processes of a certain release.[41]

It is recommended for the initiation of the run stage to be kept as simple as possible. The reason for this is, that generally the run stage is initiated by the cloud environment. Since the app might abort, the cloud platform has to be able to restart the app as well. Because a problem during the run time can occur at every point in time, a developer might not be at hand when problems occur during the run stage.[41]

Whenever the git repository is pushed to Heroku, configuration variables are changed or a new add-on resource like a Redis or Elasticsearch server are added or removed, Heroku creates a new release and restarts the app. Releases are named using the `v<number>` scheme e.g. *v1* or *v13*. The history of releases can be viewed by executing `heroku releases`.[42]

```
1 $ heroku releases -n 3
2 === fierce-thicket-96188 Releases - Current: v8
3 v8  Deploy 35abbce            email@example.com  (~ 1h ago)
4 v7  Attach DATABASE           email@example.com  (~ 1h ago)
5 v6  Set LANG, RACK_ENV, ...   email@example.com  (~ 1h ago)
```

Listing 4.30: The last three releases of an app developed for this thesis

The Heroku CLI is also able to offer more detailed information on a specific release:

```
$ heroku releases:info v8
=== Release v8
Add-ons: heroku-postgresql:hobby-dev
By:      email@example.com
Change:  Deploy 35abbce
When:    2016-12-19T12:27:07Z

=== v8 Config vars
DATABASE_URL:             postgres://user:pass@url:5432/db
LANG:                     en_US.UTF-8
RACK_ENV:                 production
RAILS_ENV:                production
RAILS_LOG_TO_STDOUT:      enabled
RAILS_SERVE_STATIC_FILES: enabled
SECRET_KEY_BASE:          secret
```

Listing 4.31: Detailed information of a specific release

The example shows that each release contains information like the time or the reason of its release. Furthermore as stated in this section, a release is made out of the artifact and the specific configuration for this deploy. Heroku perfectly implements this by saving all environment variables for a specific release which will also change when doing a rollback. A rollback is performed by using the `heroku rollback [version]` command. When no version is specified then Heroku will use the previous release otherwise the version which was specified with the command.[42]

4.6 Processes

"Execute the app as one or more stateless processes"[14]

The twelve-factor app methodology structures the different parts of the execution of an application into processes. These processes can be categorized to certain process types of which none or multiple can run at every point of time. Process types are further explained in the concurrency section.[43]

Processes of a twelve-factor app have certain important characteristics. First of all each process should be completely stateless. A stateless process does not have any stored memory about a certain request both before the request was issued as well as after the request was served. The process is only allowed to use its memory during the serving of the request. After the processing of the request has finished all data should be gone. Consequently this means that any state which has to be kept across requests should be stored in a backing service, traditionally a database.[17, p. 39-40.]

Furthermore processes have to follow the *share-nothing* pattern. An example for applications which share data across processes, are systems which rely on high-availability like certain databases and therefore use concepts like clusters and replications. The reason for a process not to share data is that processes in the cloud tend to be short-living. They can be terminated, started and scaled at will. Therefore shared data at process level has the same short lifecycle as processes and might be lost, causing errors during runtime.[17, p. 40]

[14] `https://12factor.net/processes`

Since these characteristics affect the app's caching strategies as well, it is important to choose a sophisticated caching solution. File system and in-memory caching should be avoided and extracted to a backing service. Popular backing services for caching which are also avaible on the Heroku platform are Memcached[15] or the Redis tool.[16][17, p. 41]

After having learnt about how a twelve factor app should treat state, the behaviour of the frameworks should be analyzed concerning this topic. Especially caching and session strategies will be outlined.

4.6.1 Play

To keep data across HTTP requests, the framework provides two concepts: sessions and flash scopes. While sessions are available until the user closes his browser, flash scopes are only present up to the next request. For both types, the framework achieves statelessness by storing a cookie on the client side which will be transferred back to the application for each upcoming request. While the flash scope uses a clear text cookie, the session cookie is signed using the key of the *play.crypto.secret* setting.[44]

As its caching strategy, Play uses the Ehcache library[17]. The framework utilizes Ehcache as an in-memory cache by default. As already elaborated, in-memory caches do not really work well together with the idea of a twelve-factor app, which should be small and elastically scalable. Heavy usage of an in-memory cache would bloat the application and its memory footprint unnecessarily and should be better extracted

[15] `https://memcached.org/`

[16] `https://redis.io/`

[17] `http://www.ehcache.org/`

to a backing service.[45]

Though Play uses Ehcache by default, developers are not bound to it but can utilize other third party caching solutions with Play like play-redis.[18]

4.6.2 Ruby on Rails

The Rails framework enables the developer to easily make use of sessions with different storage strategies that can be configured in the *config/initializers/session_store.rb* file. By default the framework stores the session as a encrypted cookie on the client side. To encrypt the cookie, Rails uses a key provided to the application using the `secret_key_base` field located in the *secrets.yml* file.[46]

Consequently this means that each instance of the application needs the same key otherwise a different instance will not be able to decrypt the cookie correctly. Since the cookie will be sent to the user after processing the request and no session data will be stored on the server side, this approach does not violate a stateless request. While the cookie store is being provided out of the box, storage strategies for the database[19] or for a caching service like redis[20] exist as separate libraries.

Additionally to sessions the framework offers solutions for caching like page caching where a whole web page gets cached, action caching where certain logic of a controller actions like authentication will be run but the rest of the action is cached, or fragment caching where parts of the view logic are cached and served out of the cache for new incoming requests.[47]

By default the framework uses the *tmp/cache* directory as storage for its key-

[18] `https://github.com/karelcemus/play-redis`

[19] `https://github.com/rails/activerecord-session_store`

[20] `https://github.com/redis-store/redis-rails`

value pairs. For a twelve-factor app, this strategy must be changed since it should not use its underlying file system for saving state across requests. The only storage strategy the framework offers which aligns to the methodology is the `mem_cache_store` implementation. This requires a Memcached server as a backing service. Otherwise third-party implementations like the redis-store[21] are suitable as well.

4.6.3 Laravel

Like Rails, Laravel provides multiple session backends which all use the same API and can be configured in the *config/session.php* file. Since the default backend used by the framework is a file based storage, the configuration file has to be adapted according to the methodology. The other session backends which the framework offers right out of the box are saving sessions in an encrypted cookie, using the database, or storing sessions in a PHP array in memory. While Laravel supports backing services like Redis or Memcached as its session backend as well, other libraries have to be added as with Rails or Play.[48]

When using the database as a session backend, the framework provides a prepared migration for the session table. The database can be set up for sessions by executing the two following commands:[48]

```
1 $ php artisan session:table # generate the migration
2 $ php artisan migrate       # migrate
```

Listing 4.32: Set up the database as a session backend

[21] `https://github.com/redis-store/redis-rails`

After setting the `SESSION_DRIVER` environment variable to `database` all session data will be stored in the database system behind the `DATABASE_URL` variable.[48]

The framework handles caching in a similar way. Again the default cache backend is a file storage while the same caching backends are supported as for sessions. Setting up the database for caching takes the same steps as with sessions, but the generation of the cache table requires the `cache:table` artisan command.[49]

Although by default Laravel does not handle sessions or caching in a way accepted by the methodology, it provides all tools necessary to do so in a straight forward manner.

4.7 Port binding

> ***"Export services via port binding"*** [22]

Web applications, especially in the Java environment, are often run in containers. Examples include Tomcat, Microsoft Internet Information Server or JBoss. A common approach is to deploy a set of applications into one container and separate them either using URI schemes or different ports.[17, p. 37]

A twelve-factor app on the other side is never injected into an external application server or a container. It is self-contained and creates a web-facing service on its own.[50]

In case of HTTP this means that the application has to bring its own HTTP server vendored into the application. However, this guideline does not only apply to HTTP but to all services which the application provides. Each of them should

[22] `https://12factor.net/port-binding`

be exported using a certain port on which the application listens for incoming requests.[50]

An application developed in this way can be executed more easily since one does not rely on external applications which have to be set up and configured. Furthermore by using a port binding every application can become a backing service for another application by using the URI with which this application can be accessed.[17, p. 38]

In the following sections, possible ways of exporting the HTTP service of an app using a configurable port, is explained.

4.7.1 Play

As already stated in the introductory part of Play, the framework emphasizes on a container-less deployment which is a precondition for this factor. The internal web server which Play uses is the Netty server[23] which the framework runs right out of the box and can be used for all environments.

With default configuration a Play application exposes its HTTP service using the port number 9000. To adjust the HTTP port, Play does not provide an option in a configuration file like *conf/application.conf* but uses a system property `http.port` which can be passed during start up. This system property can be mapped to a *PORT* environment variable to set the port for the HTTP service.[51]

```
1 $ export PORT=8080 # define PORT as an env var
2 $ activator run -Dhttp.port=$PORT
```

[23] `http://netty.io/`

In the example the Play application will export its HTTP service using the port number 8080 specified by using an environment variable passed to the system property.

4.7.2 Ruby on Rails

The standard way of running an application which is created with Ruby on Rails is generally compliant with this factor. The following presents an example of how to use the *puma* web server[24] which is the default web server since Ruby on Rails version 5.

Heroku recommends to use a configuration file for puma which should be placed at *config/puma.rb*.[52] With Rails 5 this file already exists and provides comments explaining certain configurations. The configuration file is written in standard Ruby meaning that the programming features of the language can be used to adjust the settings of puma. While most parts of the configuration file for puma will be elaborated in the concurrency section, one configuration entry is responsible to declare the port.

```
1 port     ENV.fetch("PORT") { 3000 }
```

Listing 4.33: Set the port as a environment variable

Including this line into the *config/puma.rb* file will tell puma to use the port number given by the *PORT* environment variable. If none is provided then the port number will default to 3000. The Rails application can now be run by executing

[24] https://github.com/puma/puma

```
$ export PORT=8080 # define PORT as an env var
$ bundle exec puma -C config/puma.rb
```

Listing 4.34: Run puma using the config/puma.rb file

which will export its HTTP service using the port 8080 in this example.

4.7.3 Laravel

Since PHP applications generally run in web servers like Apache or Nginx, this factor shows to be problematic for a Laravel application. With a locally installed Apache server in the development environment, distinct applications might use various virtual hosts exposed by individual ports, or utilize URI paths to be separated from each other. With both ways the port on which the application should listen for incoming requests can not be easily changed by using an environment variable.

When using Docker however, as already introduced in the dependency chapter, the port on which the Apache server in the Docker container listens for incoming requests can be exposed to the host machine and arbitrarily set to any number.

When running a single container the ports can be passed as options to the `run` command[53]

```
$ docker run -p 8080:80 php:7.1-apache
```

Listing 4.35: Expose port 80 of the container to port 8080 on the local machine

or by the *ports* field in a *docker-compose.yml*[54]

```
1 version: '2'
2
3 services:
4   web:
5     ports:
6       - "8000:80"
```

Listing 4.36: Expose port 80 of the container to port 8080 on the local machine

Either way the application served through the server in the docker container will be accessible in these examples through `localhost:8080`.

Through Docker, an app using an application server like Apache or Nginx, can be built in a way to support the methodology which would be hardly possible without it, and is therefore an excellent tool to develop twelve-factor apps.

4.8 Concurrency

> ***"Scale out via the process model"*** [25]

In the past, the common practice of scaling was usually done by increasing the size of an application. Adding resources like RAM or CPUs to the application is a process called *vertical scaling*. Contrary to increasing the size of one big process, a twelve-factor app distributes its work across process types. This way the work of the application can be shared across these processes and be individually scaled by creating more instances of a certain process type.[17, p. 43]

[25] `https://12factor.net/concurrency`

Internally a process might introduce an own multiplexing, like managing multiple threads, or implementing an event-based model, but the important point for a twelve-factor app to embrace, is the ability to be spanned across multiple running processes on different machines, instead of a single big one on one machine. This means an important distinction has to be made between the scaling of a process type internally, like having multiple threads which process HTTP requests, and the scaling of a process type through acquiring new machines that run this process.[43]

As already stated in the introductory part of Heroku, the platform uses the *Procfile* to define its process types. A typical Procfile might look like this:

```
1 web: bundle exec puma -C config/puma.rb
2 worker: COUNT=${AMOUNT_WORKERS} QUEUE=* rake environment
    resque:work
3 clock: ruby clock.rb
```

While the meaning of this file will be elaborated in the Ruby on Rails part of this section, it defines three process types: *web*, *worker* and *clock*. When initially deploying to Heroku, the platform parses the Procfile and starts by default one dyno for the web type and zero dynos for all other types. To scale as well as to view the current amount of dynos for each process type, the `heroku ps` command provides these functionalities.[13]

```
1 $ heroku ps
2 === web (Free): bundle exec puma -C config/puma.rb (1)
3 web.1: up 2017/01/18 22:51:16 +0100 (~ 2m ago)
```

Listing 4.37: List all running dynos

Other process types than *web* have to be scaled and therefore started manually.

```
$ heroku ps:scale worker=2 clock=1
```

Listing 4.38: Scaling process types

In this listing, Heroku will start two dynos executing the worker process while it will start one dyno for the clock process type. With the same command, dynos can be scaled down or a certain process type can be stopped by setting the amount of dynos to zero. The potential of this factor on Heroku lies in the possibility to instantly spawn new instances of a certain process type to spread the load of the application.

For development purposes, the Heroku CLI offers a `heroku local` command. It parses the Procfile in the current directory and executes each process type once. Since this behavior is similar to the production behavior on Heroku, the command is recommended for development usage.[26] Another possible tool for the development environment is the foreman[27] which is a possible candidate for a non Heroku platform.

After explaining how to separate an application into distinct process types, concrete implementations of twelve-factor apps will be outlined for each framework. These applications were developed as a part of this thesis and can be found at `https://github.com/bakku/bachelors-thesis`. The applications consist out of a web API which lets users create reminders, background workers which send emails containing the reminders to the users, and a scheduler which retrieves all reminders

[26] `https://devcenter.heroku.com/articles/heroku-local`

[27] `https://github.com/ddollar/foreman`

that are due and delegates jobs to the workers. While the size of the applications does not compare to a real enterprise system, they are sufficient to elaborate how different process types can be implemented.

4.8.1 Play

Being the only framework using a compiled language, the Play application gets built to one artifact but is internally separated using dependency injection modules which are enabled in separate configuration files.

```
1 web: target/universal/stage/bin/reminder-app-play -Dhttp.port=
    ${PORT}
2 worker: target/universal/stage/bin/reminder-app-play -Dconfig.
    file=conf/worker.conf
3 clock: target/universal/stage/bin/reminder-app-play -Dconfig.
    file=conf/scheduler.conf
```

Listing 4.39: Procfile for the Play application

But before explaining the separation with dependency injection, the web process type will be illustrated first.

4.8.1.1 Web process type

As already elaborated in the dependencies chapter, Heroku uses *sbt-native-packager* to build the artifact when deploying a Play application. Building the artifact will also generate an executable script to run the app, named after the application, and located in the *target/universal/stage/bin* folder.[24]

Although all process types will actually include the code to process API requests

because they share the same artifact, only dynos of the web type are able to receive HTTP traffic as mentioned in the introduction of Heroku, therefore the other process types won't interfere with the web process.

While the implementation of the API is not really of relevance for a twelve-factor app, the way to bring concurrency to the process is important to scale the application. Since Scala uses the Java Virtual Machine, concurrency is generally managed by running many threads in one Java Virtual Machine process.[55]

To handle HTTP requests, Play uses a thread pool that is configured in *application.conf* named `default-dispatcher`.[56]

```
akka {
  default-dispatcher {
    fork-join-executor {
      parallelism-max = ${HTTP_THREADS}
      parallelism-min = ${HTTP_THREADS}
    }
  }
}
```

Listing 4.40: Example of a Thread configuration for Play

Different implementations for an executor service are possible for Play to utilize, by default a *ForkJoinPool*. It is possible to fine tune the thread pool of the Fork Join Executor service, though when setting *parallelism-max* and *parallelism-min* both to the same fixed value, the size of the thread pool can be statically configured. A point to take in account when configuring the thread pool is the amount of available database connections because a thread won't be able to quickly process a request that requires the database if no connection is available. Since Heroku offers only

twenty database connections for database instances of the lower tier which were employed for this thesis[57], the application refrains from thread fine tuning since the thread pool won't be able to reach a size where a single dyno won't have the capacity to run these threads. In case of more connections, a fine tuning of the thread pool should be possible to make the application more scalable on multiple machines with different capacities.[56]

4.8.1.2 Worker process type

Varying from the other frameworks, Play does not offer a simple solution out of the box to develop background workers with queues. With the underlying Akka library it is possible to implement background workers, though getting started with the library has proven to be rather difficult with multiple concepts which have to be learned first, like actors or streams. Therefore, for this thesis the background workers are based on a custom solution using a thread pool and RabbitMQ subscribers listening to a messaging queue.

To not start workers on other process types as well, distinct dependency injection modules separate workers from the other process types.

```
1 package modules
2 ...
3 class WorkerModule extends AbstractModule {
4   override def configure() = {
5     bind(classOf[ReminderMailWorkerPool]).asEagerSingleton()
6   }
7 }
```

Listing 4.41: A module for the background workers

When creating separate modules, Play expects them to be explicitly enabled if the dependencies are wished to be registered. The idea is to enable the worker module in a separate configuration file like *worker.conf* which is only used for the worker process type, as shown in the Procfile of listing 4.39. In other process types the module won't be enabled and therefore the workers won't be started.[58]

```
1 include "application.conf"
2
3 play.modules.enabled += modules.WorkerModule
```

Listing 4.42: *worker.conf*

To separately set the amount of workers using environment variables, the thread configuration in the *application.conf* can be expanded with an additional thread pool.

```
1 akka {
2     // ... default-dispatcher
3     worker {
4         fork-join-executor {
5           parallelism-max = ${WORKER_THREADS}
6           parallelism-min = ${WORKER_THREADS}
7         }
8     }
9 }
```

Listing 4.43: Define another thread pool next to default dispatcher

This defined executor service can then be passed as the context in which the workers should be executed as futures.

```
1  class ReminderMailWorkerPool @Inject() (akkaSystem:
       ActorSystem, ...) {
2    ...
3    val workerContext = akkaSystem.dispatchers.lookup("worker")
4    val amountThreads = sys.env("WORKER_THREADS").toInt
5
6    (1 to amountThreads) foreach { _ =>
7      Future {
8        initializeNewWorker
9      }(workerContext)
10   }
11   ...
12 }
```

Listing 4.44: Start workers in the worker thread pool

All workers connect to a RabbitMQ queue to receive tasks by the clock process type. RabbitMQ is basically a message broker using the AMQP protocol that allows an application to send and receive messages and thereby create distributed systems.[59]

```
1  private def initializeNewWorker {
2      val channel = conn.createChannel
3
4      val durable = true
5      channel.queueDeclare(queueName, durable,
6          false, false, null)
7
8      val consumer = new DefaultConsumer(channel) {
9        override def handleDelivery(
10                       consumerTag: String,
11                       envelope: Envelope,
12                       properties: AMQP.BasicProperties,
13                       body: Array[Byte]) {
14         // send reminder mail
15         channel.basicAck(envelope.getDeliveryTag, false)
16       }
17     }
18
19     val autoAck = false
20     channel.basicConsume(queueName, autoAck, consumer)
21 }
```

Listing 4.45: Creating a worker

When using a RabbitMQ consumer as a worker a few things are important to note. Firstly the queue that is being used to send and receive the messages should be durable i.e persist its messages. This ensures that a crashed RabbitMQ server does not loose all jobs that should have been processed. Furthermore a consumer should not acknowledge messages immediately when receiving them but send an acknowledge after finishing his task. This way a job failure that might occur when

a worker crashes, does not appear to be processed and can be handled by another worker.[60]

4.8.1.3 Clock process type

The clock process type is responsible for checking the database for due reminders and delegate them as tasks to the workers. It is intended that the process checks the database once in a fixed time span like one minute. For this requirement the Akka library offers a scheduler which accepts a time span and a code block that it should execute.[61]

```
class ReminderMailJobScheduler @Inject() (...) {
  val queueName = "reminder"

  val connection = RabbitMQHelper
    .setupConnection(config.getString("cloudamqp_url"))

  val channel = connection.createChannel

  val durable = true
  channel.queueDeclare(queueName, durable, false, false, null)

  actorSystem.scheduler.schedule(0 seconds, 60 seconds) {
    val currentTime = Instant.now

    reminderRepo.allBefore(currentTime).foreach { reminder =>
      val message = reminder.userId.toString +
        "," + reminder.id.get.toString

      channel.basicPublish("", queueName,
        MessageProperties.PERSISTENT_TEXT_PLAIN,
        message.getBytes)
    }
  }
}
```

Listing 4.46: Implementation of the scheduler

To achieve a common understanding between consumer and publisher, the developer needs to think of a message format or some kind of serialization. In this example the more or less naive implementation concatenates two ids together separated by a comma.

The clock process type is special among the others since it is the only one that should run on a single machine and thread only. If multiple instances or threads were run then the schedulers would interfere with each other and possibly enqueue the same tasks.

As with the workers, the scheduler gets a separate injection module that is enabled in a new configuration file which gets passed to the executable script as shown in the Procfile.

4.8.2 Ruby on Rails

As already mentioned in the elaboration of this factor, the Procfile for the Ruby on Rails application is constructed in the following way:

```
1 web: bundle exec puma -C config/puma.rb
2 worker: COUNT=${AMOUNT_WORKERS} QUEUE=* rake environment
    resque:work
3 clock: ruby clock.rb
```

Listing 4.47: Procfile for the Rails application

4.8.2.1 Web process type

It is noticable, that the web API is powered by the concurrent web server *puma*[28] which was already introduced previously in this thesis. A characteristic of puma is the ability to add concurrency using multiple processes as well as threads. This gives the possibility to adapt the level of concurrency for a running instance. To

[28] https://github.com/puma/puma

configure the server, the frameworks's generated file is used which is located at *config/puma.rb*.[52]

```
1 threads_count = ENV.fetch("RAILS_MAX_THREADS") { 5 }.to_i
2 threads threads_count, threads_count
3
4 port        ENV.fetch("PORT") { 3000 }
5
6 environment ENV.fetch("RAILS_ENV") { "development" }
7
8 workers     ENV.fetch("WEB_CONCURRENCY") { 2 }
```

Listing 4.48: *config/puma.rb*

For each configuration, environment variables are read, which default to the value between the curly braces. For the *threads* configuration, it is possible to define a minimum and a maximum amount of threads, which are both set to the same value in this example.

For individual processes, puma uses the term *worker* which should not be confused with worker processes doing background jobs. The amount of workers is should be set relatively to the amount of RAM provided by the instance on which the application is running. Heroku recommends for its lowest dyno types, which all provide 512 MB RAM, two to four workers. The amount varies however, depending on the size of the application. Since the application used for this thesis is rather small, it is possible to run up to eight workers on the smallest dyno type. A way to find the right amount of workers, is to set them to a high amount and slowly decreasing them while being aware of the system log of Heroku. When exceeding memory limit, Heroku will immediately print a log event which means the amount of workers is

too high.[52]

The amount of threads depends first of all on whether the system is thread-safe or not. If the application's code is not thread-safe then the amounts of threads should be set to only one and concurrency should be handled exclusively at process level. If on the other hand the code is safe to be shared across threads, the amount of database connections limit the amount of concurrent requests.[52]

Since processes and threads can both be used, the level of concurrency can be calculated using the following formula:

*amount processes * amount threads = amount possible concurrent requests*

As an example, the *free* or *hobby* PostgreSQL instance of Heroku offers 20 connections to the database.[57] When using e.g. four puma processes each process should only have five threads since the amount of possible concurrent requests will otherwise exceed the amount of database connections. If a request is being processed where no connection is available, an error will be thrown, which should be prevented.[52]

Additionally, to further optimize the puma configuration it is recommended to do load testing using tools like *apache benchmark* and comparing different results with different amounts of processes and threads.

4.8.2.2 Worker process type

The worker process type of the shown Procfile is handled by a Redis-backed background processing library called *Resque*. With Resque, tasks are basically put into certain queues while Resque workers take these jobs out of these queues and process

them.[29]

```
class ReminderMailJob
  @queue = :mailing

  def self.perform(reminder_id)
    reminder = Reminder.find(reminder_id)
    ReminderMailer.reminder_email(reminder: reminder)
        .deliver_now
    Rails.logger.info "sent mail to #{reminder.user.email}"
    reminder.delete
  end
end
```

Listing 4.49: Example of a Resque job

A Resque job has to respond to the *perform* method and define its queue, in this example called *mailing*.[30] After its definition, a job can then be enqueued to be processed by a Resque worker.

```
Resque.enqueue(ReminderMailJob, 2)
```

Listing 4.50: Add one job to the queue

Since workers take their job information from the Redis service, they can be distributed across different machines. The Resque library adds a task to the *rake* tool called `resque:work` which starts worker processes. By specifiying the `QUEUE`

[29] https://github.com/resque/resque

[30] https://github.com/resque/resque

parameter it is possible to define which queues these workers will handle. In the shown Procfile the asterisk tells the workers to handle all available queues. The `COUNT` parameter defines the amount of workers to be created. The parameter can be bound to an environment variable to be easily changeable this way.

Other background processing libraries similar to Resque are Sidekiq[31] which uses the Redis-service as well or Delayed Job[32] which uses the database to store its queues.

4.8.2.3 Clock process type

The clock process is a Ruby script which defines a task that is run every minute and enqueues jobs if needed.

[31] `https://github.com/mperham/sidekiq`

[32] `https://github.com/tobi/delayed_job`

```ruby
require_relative 'config/environment'

task = Concurrent::TimerTask.new(execution_interval: 60) do
  reminders = Reminder.all

  reminders.each do |r|
    if r.reminder_date < DateTime.now
      Resque.enqueue(ReminderMailJob, r.id)
    end
  end
end

task.execute

while 1
  sleep 1
end
```

Listing 4.51: Clock process script

By requiring the *config/environment.rb* file, the complete Rails application is loaded so the script has access to all Ruby on Rails functionalities and business logic of the application. For the implementation of scheduled tasks the *concurrent-ruby* library[33] provides concurrency tools inspired by other programming languages like futures, promises or thread pools. In this example a `TimerTask` of the library, similar to the `TimerTask` implementation of Java, will run each minute and let Resque enqueue jobs if needed.

[33] https://github.com/ruby-concurrency/concurrent-ruby

4.8.3 Laravel

As the last example serves the application written with Laravel with the following Procfile.

```
1 web: vendor/bin/heroku-php-apache2 public/
2 worker: php artisan queue:work
3 clock: php schedule.php
```

4.8.3.1 Web process type

Since PHP generally runs in application servers the web process type definition is strongly linked to the way Heroku allows to execute PHP applications. The platform lets developers choose between two PHP engines, PHP-FPM and HHVM, and two web servers, Apache and Nginx.[62]

PHP-FPM is the standard runtime of PHP that normally gets shipped with the language while HHVM is an alternate implementation created by Facebook. Though HHVM has proven to be faster than PHP-FPM, the latter has catched up in performance since the release of PHP version 7.[63]

While the application in this thesis employs a combination of the apache server and the PHP-FPM runtime, both options can generally be changed by replacing the placeholders for the web process type definition: `vendor/bin/heroku-<runtime>-<server>`. The folder after the Heroku server script specifies the document root of the application, for Laravel apps generally the *public* folder.[62]

Out of all three frameworks, Laravel is the only one where Heroku provides concurrency by its own out of the box. The platform grants one PHP process 128

MB RAM and calculates the amount of processes by dividing the memory size of the dyno with the memory limit of one PHP process. As an example for the lower tier dynos, which grant 512 MB RAM, the application is run with four PHP processes.[64]

In case an application's process does not require 128 MB of RAM the value can be adapted using a *.user.ini* file in the document root folder.[64]

```
memory_limit = 64M
```

Listing 4.52: Setting the memory limit to 64 MB

After deploying the application with the *.user.ini* file, Heroku will change the memory limit and provide automatically more processes, again for lower tier dynos eight processes in this case.[64]

Additionally when wishing to set the amount of processes manually, a environment variable called `WEB_CONCURRENCY` can be set to the required value.[64]

```
$ heroku config:set WEB_CONCURRENCY=8
```

Listing 4.53: Manually instruct Heroku to spawn eight processes

4.8.3.2 Worker process type

Laravel ships with a variety of queuing backends like Beanstalkd, Amazon SQS or Redis. In case of not expecting a high amount of jobs, the database can also be a suitable backend without requiring an additional backing service. In order to prepare the database as a queuing backend, the framework provides a ready-made migration

that can be added to the project with `php artisan queue:table`. After migrating the database, and setting the queue driver in *config/queue.php* to `database`, the application is set up for adding background jobs.[65]

```
1 php artisan make:job SendReminderEmail --queued
```

Listing 4.54: Generate a new queued job

The `--queued` option instructs Laravel to generate a job which implements the `ShouldQueue` interface so the job is being pushed on to the queue instead of being run synchronously.[65]

```
class SendReminderEmail implements ShouldQueue
{
    ...
    protected $reminder;

    public function __construct(Reminder $reminder)
    {
        $this->reminder = $reminder;
    }

    public function handle()
    {
        $user = User::find($this->reminder->user_id);
        Mail::to($user->email)
          ->send(new ReminderMail($this->reminder->message));
        $this->reminder->delete();
    }
}
```

Listing 4.55: Structure of a background job

A job may take data through its constructor which gets serialized when being pushed on the queue. The `handle` function of a job defines the code that will be executed by the worker. After the implementation, a job can enqueued using the `dispatch` method which Laravel provides.[65]

```
dispatch(new SendReminderEmail($reminder));
```

Listing 4.56: Enqueueing a job

Lastly, as already defined in the Procfile, the framework offers an artisan com-

mand for starting up a worker: `php artisan queue:work`. The command however does not provide any possibilites to configure an amount of worker processes to spawn but creates a single worker process. This results in an inefficiency of the worker process type on Heroku since only one worker will be run for each dyno.[65] For a more efficient background job solution third party libraries may be used like *php-resque*[34] which is a port of the background job library *Resque* for Ruby.

4.8.3.3 Clock process type

Though Laravel offers a solution for task scheduling, the implementation relies on a crontab to issue an artisan command which is responsible for executing the scheduled tasks when they are due. Since a twelve-factor app can not rely on the availability of cron which is likewise not available on Heroku, the provided functionality can not be employed.[66]

Though other libraries exist like *crunz*[35], they rely on cron as Laravel does. Other possibilites as using an external service that calls an URI every minute do not work as well since dynos other than the web dyno are not able to receive HTTP traffic.[12]

In case the task scheduling is critical, Heroku provides add-ons like *Iron Worker*[36] to fine tune the scheduling time span of a task, otherwise an endless loop with a sleep call might also be sufficient.

[34] `https://github.com/chrisboulton/php-resque`

[35] `https://github.com/lavary/crunz`

[36] `https://elements.heroku.com/addons/iron_worker`

```php
<?php

while (true) {
    echo exec("php artisan schedule:reminders"), "\n";
    sleep(60);
}
```

Listing 4.57: A simple scheduling solution

The script calls a custom artisan command, as further elaborated in the admin tasks section, which contains the scheduling logic and sleeps afterwards for a fixed amount of time. Though the command won't be issued every minute since the time of the artisan command is not taken into consideration, the solution satisfies basic scheduling needs. The small script can then be saved in the project root and be bound to the clock process type as the Procfile specifies it.

4.9 Disposability

> *"Maximize robustness with fast startup and graceful shutdown"* [37]

Processes of applications running on cloud platforms are ephemeral. They can be started or stopped anytime, either by the hosting provider or by the developer. On Heroku multiple events issue a reboot as well as the platform itself, which will restart each dyno at least once a day.[12]

[37] https://12factor.net/disposability

As a consequence of this behavior a twelve-factor app should be able to start up and shut down rapidly. This improves the ability to scale the application and to ship new releases much more quickly.[67]

But not only because of the behavior of a cloud platform is a fast startup and shutdown highly preferred. A slow startup might deny requests to the application which could have been served if the application could boot faster especially combined with a long shutdown possibly because of a previous failure.[17, p. 23-24]

Adam Wiggens and Kevin Hofmann give some examples on how to develop a disposable application:

- An application should not deal with start up activities like populating a cache. A better approach is to externalize these responsibilities to backing services like a caching service.[17, p. 24]

- Processes should shut down gracefully when they receive the unix `SIGTERM` signal. This involves different behavior for the specific process which should shut down. A web process should finish serving his current requests and stop listening on its port while worker processes should return jobs to the queue. This requires HTTP requests to be short and jobs to be reentrant.[67]

Heroku gives processes 30 seconds to shut down gracefully after sending a `SIGTERM`. If they are not able to shut down until then, the platform will terminate these processes.[12]

Because of the disposable nature of a twelve-factor app the microservice architecture is arguable the more suitable software design for larger applications. Big monolithic systems tend to have long startup and shutdown times which makes them unsuitable for cloud platforms like Heroku.[68]

4.10 Dev/prod parity

> ***"Keep development, staging, and production as similar as possible"*** [38]

During the lifetime of an application which is being run on multiple environments, there can be substantial differences across deploys. The methodology categorizes these differences into three types of gaps:

- **Time:** Traditionally the time between code being checked into the repository and a deploy was rather large, creating a gap between development having the newest code and production having older code. A twelve-factor app should keep this gap small, reducing the time between code check-in and deploy to minutes or hours. This gives developers the ability to know what code is currently running on production which enables them to solve problems more quickly.[17, p. 30]
- **People:** When responsibilities of development and deployment are divided between separate people then the whole deployment pipeline relies on good communication between the two parties. Instead of separating developers and operations, the development and deployment process should embrace the DevOps approach. Developers of a twelve-factor app should be responsible for its operation as well.[69]

[38] `https://12factor.net/dev-prod-parity`

- **Tools:** Often times developers use lightweight backing services during development compared to those on production, e.g. SQLite during development and PostgreSQL on production. Increasing the tools gap between deploys makes an app more prone to the problem that the application runs on one environment while it does not on another. By aligning backing services across deploys, developers will gain confidence that the application will run everywhere. The twelve-factor app methodology does not only recommend to use the same backing service but to utilize the same version of the service as well.[69]

Contrary to the time gap as well as the people gap which are part of the organisational structure of a team and not in the area of responsibility of a web framework, it can still substantially influence the tools gap by providing a different set of tools for each environment by default.

While the Play framework does not employ inconsistencies, a gap between tools can be found for the Rails and Laravel framework.

4.10.1 Ruby on Rails

Prior to version 5, the default web server which the framework used was *WEBrick* which is being distributed with the standard Ruby library. WEBrick excels in its ease of use but is poor for production environments since it runs in a single process and in a single thread. Therefore no concurrent requests can be served and each request has to wait for the former to finish. To get the best of both, developers used WEBrick for development purposes and a concurrent web server for production. This approach introduced a gap between web servers across environments by default which is not compliant with the methodology. Since WEBrick is single threaded, errors with non

thread safe code can not be detected but might occur when using a multi-threaded production server.[70]

When using Ruby on Rails prior to version 5 then the same web server should be used across all environments. Since the newest major version 5, the framework ships with the puma web server by default for all environments, which is suitable for production and thereby closes this gap.

4.10.2 Laravel

Though not being the fault of the framework, PHP applications generelly run in application servers. These are often not practical in a development environment since they require to install dependencies like PHP extensions or need configuration for each application. Therefore Laravel ships with a development server accessible with `php artisan serve` to easily run the application.[71]

An alternative to the the already mentioned Docker tool to solve this problem is Laravel Homestead. Homestead is based on the Vagrant tool, which enables developers to provide a definition of a so called Vagrant box that can be transformed to a virtual machine running on e.g. VirtualBox. This way the virtual machine can be kept identical across development environments, providing the same set of tools for every developer. Homestead comes pre-packaged with all tools needed to develop Laravel applications. Utilized correctly[39], Homestead is a solution to the tools gap problem for Laravel applications which is compliant with the dev/prod parity factor.[72]

[39] `https://laravel.com/docs/5.3/homestead#per-project-installation`

4.11 Logs

"Treat logs as event streams" [40]

To analyze the behaviour of an application, logs are highly necessary. Most commonly, applications have a defined location for a log file to which its log events will be saved. Instead of writing its log output to a file, a twelve-factor app should write its log events to *stdout*.[73]

During development, log events will be printed out to the terminal in which the developer is running the application, while on server environments external tools can use these output streams to route them to a specific destination. The important point of this guideline is that it is not disapproved to use a log file as a destination but the application should not be responsible to route streams or manage these log files.[73]

A few reasons support this approach. Firstly, applications hosted on cloud platforms like Heroku can not make assumptions about its underlying file system. Storing a log file at a specified location is therefore a contradiction. Furthermore when extracting log management out of the application it becomes simpler and can be modified without changing the application. Lastly, when scaling on a cloud platform like Heroku, it is often unknown where the application instances are running and therefore hard to manage log files across instances. Aggregation of log events managed by the cloud provider or a tool therefore simplifies the scalability of the application.[17, p. 21-22]

[40] `https://12factor.net/logs`

On Heroku, the platform manages log events across all dynos with a tool called *Logplex*. Logplex routes all log events which were printed to stdout into a single channel and provides the outcome to the developer.[74]

Since Logplex does not provide access to more than 1500 lines of log data a more storage-focused solution has to be used if log files should be analyzed. Heroku offers additional logging add-ons like Papertrail[41].[74]

In the following the usage of Logplex with the Heroku CLI will be further elaborated.

The CLI offers a *logs* command with which log events of the application can be shown and filtered.[74]

```
1 $ heroku logs
2 2017-01-06T16:41:20.084296+00:00 heroku[web.1]: Unidling
3 2017-01-06T16:41:26.603527+00:00 app[web.1]: => Booting Puma
```

Listing 4.58: Example output of the *heroku logs* command

Each log event line has the following structure: `<timestamp> <source>[<dyno>]: <message>` where the source specifies whether the log line was printed by the system (heroku) or by the application.[74]

Log events can easily be filtered by source or dyno:

```
1 $ heroku logs --source app --dyno web.1
2 2017-01-06T16:41:26.603527+00:00 app[web.1]: => Booting Puma
```

Listing 4.59: Filtering by source and dyno

[41] `https://papertrailapp.com/`

By filtering with a dyno the developer can see log events for a specific instance of the application.

To use the logging solution offered by Heroku the application has to print its log events to stdout. Here the question arises whether the covered frameworks support this behaviour or not.

4.11.1 Play

The Play Framework uses Simple Logging Facade for Java[42] combined with Logback[43] for its logging functionalities.[75] As its configuration, Logback expects a *logback.xml* file which is located in the *conf* folder of a Play application.[76] The default *logback.xml* which the framework generates when deriving a project from the *play-scala* template will configure Logback to write to a file as well as to stdout. Since the file appender is not needed it can be removed resulting in a configuration which looks similar to the one that follows.

[42] `http://slf4j.org/`

[43] `http://logback.qos.ch/`

```
<configuration>

  <conversionRule conversionWord="coloredLevel" converterClass
   ="play.api.libs.logback.ColoredLevel" />

  <appender name="STDOUT" class="ch.qos.logback.core.
   ConsoleAppender">
    <encoder>
      <pattern>%coloredLevel %logger{15} - %message%n%
   xException{10}</pattern>
    </encoder>
  </appender>

  <appender name="ASYNCSTDOUT" class="ch.qos.logback.classic.
   AsyncAppender">
    <appender-ref ref="STDOUT" />
  </appender>

  <logger name="play" level="INFO" />
  <logger name="application" level="DEBUG" />

  <root level="WARN">
    <appender-ref ref="ASYNCSTDOUT" />
  </root>

</configuration>
```

Listing 4.60: A modified logback.xml for writing to stdout

The XML defines a colored output using an appender with the given pattern to write to stdout. Furthermore this appender will be wrapped by an asynchronous implementation to improve the performance. On generation the framework defines

two loggers, one called *play* which the framework uses itself and another one called *application*. To work with the application logger, the framework offers a predefined *Logger* object.[75]

```
1 import play.api.Logger
2 ...
3 Logger.info("Informative log event")
```

Listing 4.61: Usage of the Logger object

For better filtering different loggers can be utilized, too, e.g. one for each class.

```
1 import play.api.Logger
2 ...
3 val logger = Logger(this.getClass())
4 logger.info("Informative log event")
```

Listing 4.62: Usage of a custom Logger object

When using custom loggers it is important to note that if a logger is not defined in the *logback.xml*, Logback will fall back to the root logger which currently has the loglevel set to *WARN*. This will result in log events being ignored when being printed using a lower level than *WARN*. The solution is to either define more loggers in the *logback.xml* or to decrease the log level of the root logger.[76]

4.11.2 Ruby on Rails

When creating a new application using the framework's project generator, the *config/environments/production.rb* file contains the following section:

```
1 if ENV["RAILS_LOG_TO_STDOUT"].present?
2   logger           = ActiveSupport::Logger.new(STDOUT)
3   logger.formatter = config.log_formatter
4   config.logger    = ActiveSupport::TaggedLogging.new(logger)
5 end
```

Listing 4.63: Extract from *config/environments/production.rb*

It is apparent that the logger is configured to use *stdout* to stream its log events when the `RAILS_LOG_TO_STDOUT` environment variable is present. `present?` is a method added by the Rails framework which returns true for a string that is non-empty. Thus, developers can arbitrarily set the environment variable to any string.[44]

Since a twelve-factor app does not log to *stdout* only in production environments but everywhere, the piece of code can be extracted from *config/environments/production.rb* and added to *config/application.rb*. Furthermore the behavior to log to *stdout* should not be dependent on an environment variable but the default behavior, therefore the if clause can be removed.

Since controllers and models in Ruby on Rails inherit the frameworks base classes, a *logger* object is provided as well. The logger object responds to the methods *debug*, *info*, *warn*, *error*, and *fatal* which also specify the log level of the log event that will be emitted.[45]

[44] `http://apidock.com/rails/Object/blank%3F`

[45] `http://guides.rubyonrails.org/debugging_rails_applications.html`

```
class PagesController < ApplicationController
  def home
    logger.info "Accessed home action"
  end
end
```

Listing 4.64: Logging in a Ruby on Rails controller

When not inheriting from one of framework's base classes then the logger object is available using the global `Rails` module.

```
Rails.logger.info "Log event outside"
```

Listing 4.65: Logging outside of Rails base classes

4.11.3 Laravel

Laravel makes changing the logging behavior straight-forward. The *config/app.php* file contains two settings for the logging configuration that can be set by using environment variables: one being the logging strategy and the other the log level.[46]

If no logging strategy by the `APP_LOG` environment variable, Laravel enables the *single* strategy, meaning it will log to a single log file located at *storage/log/laravel.log*. To log to *stdout* the framework provides a logging strategy called *errorlog*.[47]

Therefore a twelve-factor app has two possibilities:

1. Set the environment variable `APP_LOG` to *errorlog*.

[46] `https://laravel.com/docs/5.3/errors`

[47] `https://laravel.com/docs/5.3/errors`

2. Don't set the environment variable `APP_LOG` but instead set the default value for the log setting to *errorlog*.

```
'log' => env('APP_LOG', 'errorlog')
```

Listing 4.66: Setting the default value to *errorlog*

Since the second solution solves this problem without having to add a new environment variable for a new deploy, it is recommended instead of the first approach.

Behind Laravel's logging functionalities lies the Monolog[48] library. To not force developers to deal with this tool, the framework provides a facade which handles its calls using Monolog under the hood. According to RFC 5424[49] the logger offers eight log levels: *emergency*, *alert*, *critical*, *error*, *warning*, *notice*, *info* and *debug*.

```
use Illuminate\Support\Facades\Log;

Route::get('/home', function() {
  Log::debug('Accessed Home');
  return 'home';
});
```

Listing 4.67: Example of a debug log event

[48] https://github.com/seldaek/monolog

[49] https://tools.ietf.org/html/rfc5424

4.12 Admin processes

"Run admin/management tasks as one-off processes"[50]

The final factor of the methodology concerns administrative tasks and how to execute them. On the side of all long running processes like the web process serving requests, an application often has to provide administrative tasks. These tasks include database migrations, scripts that can for example clean unused references in a database, or consoles, often called *REPLs*, to inspect the application.[77]

These administrative tasks should be executed using one-off processes, meaning they should start, do their job, and terminate afterwards. Furthermore these tasks should run in the same environment as the other process types of the application. This includes using the same codebase as well as configuration.[77]

On Heroku one-off processes are invoked using the `heroku run` command followed by the executable that should be started.[78]

```
1 $ heroku run bash
2 Running bash on fierce-thicket-96188... up, run.9459 (Free)
```

Listing 4.68: Opening a bash console in the application environment

The `run` command aquires an *one-off dyno* which is loaded using the same codebase and configuration as all other dynos.[78]

Using one-off dynos, administrative tasks, provided either by the framework or created by the developer, can be executed at will. Examples of administrative tasks

[50] `https://12factor.net/admin-processes`

as well as how to implement them, will be illustrated for each framework in the following sections.

4.12.1 Play

The Play framework uses the, in the Scala community widely known, Scala Built Tool, short *sbt*. While *sbt* is used to built a Play application, it is suitable to run tasks as well. Tasks can be run directly from the command line or via the sbt shell.[79]

```
$ sbt compile
```

Listing 4.69: Run a task from the command line

```
$ sbt
[reminder-app-play] $ compile
[success] Total time: 0 s, completed Feb 22, 2017 5:13:04 PM
[reminder-app-play] $
```

Listing 4.70: Start the sbt shell and run a task

Running tasks using the sbt shell is substantially faster than from the command line directly, since the latter requires spinning up a JVM while the former does only when initially starting the sbt shell.[79]

Play extends sbt with further tasks like `playGenerateSecret`, yet the framework does not provide any documentation or information on how a developer can extend sbt with custom tasks by himself. Generally custom sbt tasks can be added by providing a name for the task as well as a code block.[80]

```
1 val helloTask = taskKey[Unit]("Say Hello World")
2
3 helloTask := { println("Hello World" }
```

Listing 4.71: Define a custom task in *build.sbt*

This solution however, is not sufficient since accessing the business logic of the application will be needed for many administrative tasks like cleaning up expired database records. Since the documentation of Play or solutions on the Internet do not state on how to achieve this, especially with components of the application being employed with dependency injection, it can be said, that implementing a proper solution for administrative tasks need a deep understanding of the framework and sbt which were not possible to be gained during the development of the Play application for this thesis.

4.12.2 Ruby on Rails

For administrative tasks Rails uses the *rake* tool[51] which describes itself as "a Make-like program". It enables developers to create tasks written in the Ruby language.[81] Since historically some functionalities of the framework were provided using the *rake* and some by the *rails* executable, the team behind Rails decided to bring more consistency behind the framework's command line interface and wraps all rake tasks since version 5.0+ with the *rails* command.[82]

By default the framework ships with a set of predefined tasks which can be viewed by executing `rails -T`. Examples include `db:migrate` for running the new-

[51] `https://github.com/ruby/rake`

est migrations, `routes` to see the binding of every endpoint with its corresponding controller, or `stats` which presents some code metrics about the application.[83]

```
$ heroku run rails db:migrate
```

Listing 4.72: Run the newest migrations on Heroku

Custom tasks are expected to be located in the *lib/tasks* folder of the project. While tasks can be grouped in arbitrarily named files, the file name should end with *.rake*.[83]

```
desc 'Deletes all reminders that are older than one day'
task delete_old_reminders: :environment do
    Reminder.where('created_at < ?', 1.day.ago).destroy_all
end
```

Listing 4.73: *lib/tasks/delete_old_reminders.rake*

The first line in listing 4.73 gives the task a description that can be read when invoking `rails -T`. The second line defines the name of the newly created task as well as all tasks which should be invoked before running the code block between `do...end`. In listing 4.48 the name of the created task is *delete_old_reminders* while another task called *environment* should be invoked beforehand. The *environment* task in this case is another predefined task by the framework which loads all code of the application. Therefore it is possible to use the whole business logic in the custom task like in this case a model class.[83]

To invoke the task on Heroku, the syntax is similar to the one presented in listing 4.72:

```
$ heroku run rails delete_old_reminders
```

Listing 4.74: Running the newly created task

4.12.3 Laravel

Upon generation of a new Laravel project, an executable named *artisan* is located in the application's root directory. This tool provides all kind of commands needed for managing the Laravel application. The most frequest utilized commands include `tinker` which boots up a REPL, `migrate` to migrate the database, or a set of `make` commands to generate new components for the application.[84]

`make` offers one option to generate a custom artisan command as well.[84]

```
$ php artisan make:command CleanupReminders
```

Listing 4.75: Create a new custom artisan command

On execution artisan will create a new file in the *app/Console/Commands* folder. This file contains a base template for a new artisan command that can be customized to one's needs. An example of a finished artisan command can look as follows.[84]

```php
<?php

namespace App\Console\Commands;

use Illuminate\Console\Command;

class CleanupReminders extends Command
{
    protected $signature = 'cleanup:reminder';
    protected $description = 'Delete reminders that are older
    than a day';

    public function __construct()
    {
        parent::__construct();
    }

    public function handle()
    {
        App\Reminder::where(
          'reminder_date',
          '<',
          Carbon\Carbon::now()->subDay()
        )->delete();
    }
}
```

Listing 4.76: Example of a custom artisan command

The `signature` and `description` variables define how the command will be presented with the artisan executable while the `handle` function defines the code that will be executed. After its definition the command has to be registered in the

app/Console/Kernel.php to be known to artisan.[84]

```
class Kernel extends ConsoleKernel
{
    protected $commands = [
        Commands\CleanupReminders::class
    ];
    ...
}
```

Listing 4.77: Register the task in the *Kernel.php*

```
$ php artisan cleanup:reminder
```

Listing 4.78: When registered the command can be run

5 Evaluation

After having learned about the complete methodology and how to create twelve-factor apps with three different frameworks, the question arises for which type of applications the methodology is applicable and for which not? Do some factors of twelve-factor apps have short comings, and are there certain aspects of an application for which the methodology does not provide a solution?

Furthermore the implementation of a twelve-factor app across the utilized frameworks should be compared by testing the native support of the methodology, development speed and performance.

5.1 Twelve-Factor Apps

Adam Wiggens describes the twelve-factor app manifesto as "a methodology for building software-as-a-service apps"[2] which already limits the type of applications which are suitable to be developed as twelve-factor applications. Moreover, Kevin Hoffmann, equates twelve-factor apps with applications written specifically for cloud environments, so called *cloud-native applications.*[17, Preface VII-XI] Consequently applications that are not suitable for the cloud are generally not suitable or possible to be developed as a fully compliant twelve-factor app as well.

Noah Slater gives applications which are not created for the cloud and there-

fore no twelve-factor apps the term of *legacy apps*. These include applications that base on full-stack platforms like Wordpress, Magento or Drupal which were originally not written for the cloud as well. While it's possible to simply host a legacy application on a cloud hosting platform, this approach makes it similar to using a traditional virtual private server.[85] A cloud-native application however, should be elastically scalable, have ephemeral filesystems, be stateless, and treat everything as a service.[17, Preface VII-XI]. Applications which have to violate these characteristics are not cloud-native and therefore generally not suitable as candidates for twelve-factor apps.

The twelve-factor app methodology is generally a great way to develop applications for the cloud because of the way it harmonizes with it. Running `heroku ps:scale worker=20` to increase the amount of background workers in case of a high expected load, displays one of the strong points of a cloud-native application. Still some of the factors can be impractical for production uses.

Not expecting any dependent executable on the instance of the application can be viewed as too strict and too specific to Heroku. While a developer might not have the possibility to add a dependency to a dyno, when using a cloud hosting platform like Amazon Web Services, possibilities exist to create a custom image of a machine and deploy applications only on instances of this image.[86] Furthermore an automated deploy with a tool like *capistrano*[1] can be configured to install dependencies on the machine if they don't exist already. Though generally preferred to be bundled with the application, binaries like *imagemagick* or *curl* might also be externally expected as long as they are being installed as part of an automatic procedure.

[1] `http://capistranorb.com/`

Furthermore, logging to stdout and expecting a tool to aggregate the log events and manage them is unnecessary when libraries exist that can accomplish the same as well. Tools like Simple Logging Facade For Java[2] are capable of achieving the same results which the twelve-factor app methodology expects. Using tools as previously mentioned, and not logging to stdout should therefore not be unaccepted, but an alternative solution.[87]

While mostly agreeing with the factors of twelve-factor apps, Kevin Hoffman explains in his book "Beyond the Twelve-Factor App" some problems of cloud native applications for which the methodology does not provide a solution for.

Firstly, since being distributed across multiple machines and having the characteristic of being elastically scalable, developers are not in the clear about how the application and the instances behave and whether there are problems or not. Therefore it is essential to provide guidelines on how to properly monitor applications running on cloud platforms which is according to Hoffman not easily achieved.[17, p. 45-47]

Secondly, Hoffman critises, that the methodology does not provide information on how to sufficiently secure a twelve-factor app. While being executed in different data centers and accessed by many clients, malicious usage of the application should not be underestimated. He proposes the concept of security by design, where the security of the application is a central part of the development since day one, and concretely urges developers to at least secure every endpoint of the application using role-based access control to be able to trace every request back to a consumer.[17, p. 49-50]

[2] `https://www.slf4j.org/`

Lastly, Hoffman discusses the idea of giving the API of the application more attention than the twelve-factor app methodology is doing. Since applications running in the cloud, will often act as services used by clients, the API becomes the medium of communication for which every functional component of the application will be developed. By defining the API of the application as the first step in the development process, developers create a common understanding with other stakeholders of the application. Kevin Hoffman calls this approach *API First*, derived from *Mobile First*, which should be made part of the development process of a twelve-factor app.[17, p.5-8]

5.2 Implementation across the Frameworks

The following sections base on the experiences that were made during the development process of this thesis. For a performance evaluation the worker and web process types were individually benchmarked.

The web process type was executed on two dynos with ten threads each. The API was then load tested with apache benchmark and 1000 requests with 20 concurrent users.

The worker process type was tested by preparing 100 jobs which had to be processed. Again two dynos were used with 10 threads on each dyno, execpt Laravel, where setting the amount of workers was not possible.

5.2.1 Play

In the comparison of the three, the Play framework holds a special place, because it provides much less out of the box than Laravel and Ruby on Rails. The developer has

therefore the freedom and duty to implement solutions of his own to build a cloud-native application. Generally the solution implemented with the Play framework seems not as natural for a cloud environment as the other applications. A different process type should not be separated using a concept as dependency injection, but should be separately invoked and not implicitly include the whole API of the web process type.

Though the unsatisfying implementation could be caused from an insufficient knowledge of the framework, the documentation of Play does not provide enough information to tackle problems like these. Though the creators of Play worked together with Heroku to provide a good support of the platform for the framework[88], no official documentation exist on how to write a Play application that is tailored to Heroku. Furthermore, third-party solutions found on the Internet often refers to older versions of Play which can not be utilized anymore since of breaking changes in the framework. These problems lead Play to be the framework with the slowest development speed and the longest time to initially learn among the three of this thesis.

However when using Play, the performance will be higher than with Ruby on Rails or Laravel. Though being nearly even with the Rails framework with 34 requests per second, the workers were able to process all 100 jobs in 36 seconds which is by far the fastest processing.

5.2.2 Ruby on Rails

Rails has been the framework for which Heroku was originally built, therefore the usage and the availability of solutions to implement cloud-native applications is wide-spread. The core of the framework has evolved in a more twelve-factor way as

well e.g. providing secrets for configuration or using puma across all environments, yet the move to a fully compatible framework to build a twelve-factor app has not occured.

The Ruby language that powers Rails is developer friendly and can be quickly picked up with previous programming knowledge, the documentation of Rails is thorough and understandable. Experienced Rails programmers however state that beginners of the framework will benefit from an early high development speed but will suffer from a so called code mess later on. This is caused by the framework not offering enough best practices on how to organize the application and basically only providing models, views and controllers.[89]

Among the three frameworks, Rails was the second most performant, being able to process 33 requests per second and processing 100 jobs in 51 seconds.

5.2.3 Laravel

Laravel brings good support for twelve-factor apps from ground up like using environment variables by default and providing solutions out of the box to develop stateless, share-nothing processes. Yet, a missing multi threading ability e.g. for workers or the need of being run in application servers hurts the flexibility of Laravel.

It excels, furthermore with its clear documentation and simple solutions for widespread best practices in web development like queues, caching, sessions or mailing. Among all three framework it grants developers the highest productivity and development speed and can moreover be quickly learned.

Compared with Rails and Play, Laravel is the slowest framework, being able to process 20 requests per second and taking 65 seconds to process all 100 tasks.

6 Summary

The twelve-factor app methodology provides valuable guidelines and principles to follow, when developing an application for the cloud. It enables to make usage of the benefits which cloud hosting grants compared to traditional hosting. Though it is arguable that the manifesto leans towards the way Heroku handles applications, regardless of the employed cloud hosting platform, important lessons can be learned and applied for every project.

A perfect framework for the methodology however, does not exist. Each one has distinct strengths and weaknesses and supports or contradicts the methodology differently. The underlying technology should be chosen by analyzing the requirements of the project.

By using Play, newer developers will initially have it harder to pick up the framework and its concepts, the application will have to rely on more in-house built solutions, therefore the development speed will generally suffer. Yet when in the need of high performance, the time and energy should be invested.

Laravel provides many solutions out of the box, an easy and understandable documentation so developers will be able to implement an application very rapidly. Yet the flexibility and performance of the framework has proven to be less in the comparison of the three.

Lastly Rails provides a stable and performant platform, that can be acquired quickly by developers with previous programming knowledge. It has a long history of being deployed to Heroku, so developers can built on experience made by other developers. For the average application, it can be said that choosing the Rails framework as underlying technology is a safe pick.

Bibliography

[1] Jelle Frank van der Zwet, Vincent int Veld and Jürgen Sprenzinger. *Traditionelles Hosting oder Cloud?* 19th Nov. 2013. URL: http://www.datacenter-insider.de/traditionelles-hosting-oder-cloud-a-425478/ (visited on 31/10/2016).

[2] Adam Wiggins. *The Twelve-Factor App*. 30th Jan. 2012. URL: https://12factor.net/ (visited on 31/10/2016).

[3] Heroku. *How Heroku works*. URL: https://devcenter.heroku.com/articles/how-heroku-works (visited on 10/12/2016).

[4] Jan Ahrens. *Heroku with C*. 17th June 2014. URL: http://blog.jan-ahrens.eu/2014/06/17/heroku-with-c.html (visited on 10/12/2016).

[5] Play. *Introducting Play 2*. URL: https://www.playframework.com/documentation/2.5.x/Philosophy (visited on 25/12/2016).

[6] Sayanee Basu. *Ruby on Rails Study Guide: The History of Rails*. 22nd Jan. 2013. URL: https://code.tutsplus.com/articles/ruby-on-rails-study-guide-the-history-of-rails--net-29439 (visited on 28/11/2016).

[7] Matt Stauffer. *Laravel: Up and Running*. O'Reilly Media, Inc., 2016.

[8] Laravel. *Service Container*. URL: https://laravel.com/docs/5.3/container (visited on 26/12/2016).

[9] Laravel. *Service Provider*. URL: https://laravel.com/docs/5.3/providers (visited on 26/12/2016).

[10] Laravel. *Facades*. URL: https://laravel.com/docs/5.3/facades (visited on 26/12/2016).

[11] Francis Irving. *Herokus early history: 4 home pages that made $212 million*. 5th May 2012. URL: https://www.flourish.org/2012/05/herokus-early-history-4-home-pages-that-made-212-million/ (visited on 24/11/2016).

[12] Heroku. *Dynos and the Dyno Manager*. URL: https://devcenter.heroku.com/articles/dynos (visited on 05/12/2016).

[13] Heroku. *Process Types and the Procfile*. URL: https://devcenter.heroku.com/articles/procfile (visited on 05/12/2016).

[14] Heroku. *Add-ons*. URL: https://devcenter.heroku.com/articles/add-ons (visited on 16/02/2017).

[15] Adam Wiggins. *I. Codebase*. 30th Jan. 2012. URL: https://12factor.net/dependencies (visited on 12/12/2016).

[16] RhodeCode. *Version Control Systems Popularity in 2016*. URL: https://rhodecode.com/insights/version-control-systems-2016 (visited on 29/11/2016).

[17] Kevin Hoffman. *Beyond the Twelve-Factor App*. O'Reilly Media, Inc., 2016.

[18] Melvin E. Conway. *How Do Committees Invent?* URL: http://www.melconway.com/research/committees.html (visited on 12/12/2016).

[19] Heroku. *Deploying with Git*. URL: https://devcenter.heroku.com/articles/git (visited on 05/12/2016).

[20] Adam Wiggins. *II. Dependencies*. 30th Jan. 2012. URL: https://12factor.net/codebase (visited on 12/12/2016).

[21] Play. *Downloads*. URL: https://www.playframework.com/download (visited on 04/01/2017).

[22] Dick Wall. *Typesafe Activator, What *is* it? (and why should I care?)* URL: https://www.lightbend.com/blog/typesafe-activator-what-is-it (visited on 04/01/2017).

[23] Play. *Managing library dependencies*. URL: https://www.playframework.com/documentation/2.5.x/SBTDependencies (visited on 04/01/2017).

[24] Heroku. *Deploying Scala*. URL: https://devcenter.heroku.com/articles/deploying-scala (visited on 04/01/2017).

[25] Bundler. *Docs*. URL: https://bundler.io/v1.13/man/gemfile.5.html (visited on 12/12/2016).

[26] Daniel Kehoe. *Managing Rails Versions and Gems*. URL: https://railsapps.github.io/managing-rails-versions-gems.html (visited on 12/12/2016).

[27] Bundler. *Bundler's Purpose and Rationale*. URL: https://bundler.io/v1.13/rationale.html (visited on 20/12/2016).

[28] Heroku. *Specifying a Ruby Version*. URL: https://devcenter.heroku.com/articles/ruby-versions (visited on 12/12/2016).

[29] Composer. *Introduction*. URL: https://getcomposer.org/doc/00-intro.md (visited on 04/01/2017).

[30] Composer. *Basic Usage*. URL: https://getcomposer.org/doc/01-basic-usage.md (visited on 04/01/2017).

[31] Heroku. *Getting Started with Laravel on Heroku*. URL: https://devcenter.heroku.com/articles/getting-started-with-laravel (visited on 04/01/2017).

[32] Laravel. *Installation*. URL: https://laravel.com/docs/5.4 (visited on 15/02/2017).

[33] Docker Inc. *What is Docker?* URL: https://www.docker.com/what-docker (visited on 16/02/2017).

[34] Docker Inc. *Dockerfile reference*. URL: https://docs.docker.com/engine/reference/builder/ (visited on 16/02/2017).

[35] Adam Wiggins. *III. Config*. 30th Jan. 2012. URL: https://12factor.net/config (visited on 12/12/2016).

[36] Play. *Accessing an SQL database*. URL: https://www.playframework.com/documentation/2.5.x/ScalaDatabase (visited on 12/01/2017).

[37] Play. *Configuration file syntax and features*. URL: https://www.playframework.com/documentation/2.5.x/ConfigFile (visited on 12/01/2017).

[38] Rails. *Configuring Rails Applications*. URL: http://guides.rubyonrails.org/configuring.html (visited on 17/12/2016).

[39] Laravel. *Configuration*. URL: https://laravel.com/docs/5.3/configuration (visited on 16/02/2017).

[40] Adam Wiggins. *IV. Backing Services*. 30th Jan. 2012. URL: https://12factor.net/backing-services (visited on 12/12/2016).

[41] Adam Wiggins. *V. Build, release, run*. 30th Jan. 2012. URL: https://12factor.net/build-release-run (visited on 14/12/2016).

[42] Heroku. *Releases*. URL: https://devcenter.heroku.com/articles/releases (visited on 20/12/2016).

[43] Adam Wiggins. *VI. Processes*. 30th Jan. 2012. URL: https://12factor.net/processes (visited on 14/12/2016).

[44] Play. *Session and Flash scopes*. URL: https://www.playframework.com/documentation/2.5.x/ScalaSessionFlash (visited on 16/02/2017).

[45] Play. *The Play cache API*. URL: https://www.playframework.com/documentation/2.5.x/ScalaCache (visited on 16/02/2017).

[46] Rails. *Ruby on Rails Security Guide*. URL: http://edgeguides.rubyonrails.org/security.html (visited on 16/01/2017).

[47] Rails. *Caching with Rails: An Overview*. URL: http://guides.rubyonrails.org/caching_with_rails.html (visited on 16/01/2017).

[48] Laravel. *HTTP Session*. URL: https://laravel.com/docs/5.3/session (visited on 17/02/2017).

[49] Laravel. *Cache*. URL: https://laravel.com/docs/5.3/cache (visited on 17/02/2017).

[50] Adam Wiggins. *VII. Port binding*. 30th Jan. 2012. URL: https://12factor.net/port-binding (visited on 14/12/2016).

[51] Play. *Setting up a front end HTTP server*. URL: https://www.playframework.com/documentation/2.5.x/HTTPServer (visited on 30/12/2016).

[52] Heroku. *Deploying Rails Applications with the Puma Web Server*. URL: https://devcenter.heroku.com/articles/deploying-rails-applications-with-the-puma-web-server (visited on 30/12/2016).

[53] Docker Inc. *docker run*. URL: https://docs.docker.com/engine/reference/commandline/run/ (visited on 17/02/2017).

[54] Docker Inc. *Compose file version 2 reference*. URL: https://docs.docker.com/compose/compose-file/compose-file-v2/ (visited on 17/02/2017).

[55] Adam Wiggins. *VIII. Concurrency*. 30th Jan. 2012. URL: https://12factor.net/concurrency (visited on 14/12/2016).

[56] Play. *Understanding Play thread pools*. URL: https://www.playframework.com/documentation/2.5.x/ThreadPools (visited on 20/02/2017).

[57] Heroku. *Heroku Postgres*. URL: https://elements.heroku.com/addons/heroku-postgresql (visited on 20/02/2017).

[58] Play. *Dependency Injection*. URL: https://www.playframework.com/documentation/2.5.x/ScalaDependencyInjection (visited on 20/02/2017).

[59] Pivotal Software Inc. *What can RabbitMQ do for you?* URL: https://www.rabbitmq.com/features.html (visited on 20/02/2017).

[60] Pivotal Software Inc. *Work Queues*. URL: https://www.rabbitmq.com/tutorials/tutorial-two-java.html (visited on 20/02/2017).

[61] Play. *Integrating with Akka*. URL: https://www.playframework.com/documentation/2.5.x/ScalaAkka (visited on 20/02/2017).

[62] Heroku. *Deploying PHP Apps on Heroku*. URL: https://devcenter.heroku.com/articles/deploying-php (visited on 20/02/2017).

[63] Vikrant Datta. *HHVM vs PHP 7 Performance Showdown (WordPress, Nginx)*. URL: http://blog.wpoven.com/2016/04/14/hhvm-vs-php-7-performance-showdown-wordpress-nginx/ (visited on 20/02/2017).

[64] Heroku. *Optimizing PHP Application Concurrency*. URL: https://devcenter.heroku.com/articles/php-concurrency (visited on 20/02/2017).

[65] Laravel. *Queues*. URL: https://laravel.com/docs/5.3/queues (visited on 21/02/2017).

[66] Laravel. *Task Scheduling*. URL: https://laravel.com/docs/5.3/scheduling (visited on 21/02/2017).

[67] Adam Wiggins. *IX. Disposability*. 30th Jan. 2012. URL: https://12factor.net/disposability (visited on 14/12/2016).

[68] Chris Richardson. *Pattern: Monolithic Architecture*. URL: http://microservices.io/patterns/monolithic.html (visited on 26/02/2017).

[69] Adam Wiggins. *X. Dev/prod parity*. 30th Jan. 2012. URL: https://12factor.net/dev-prod-parity (visited on 14/12/2016).

[70] Heroku. *Ruby Default Web Server*. URL: https://devcenter.heroku.com/articles/ruby-default-web-server (visited on 04/01/2017).

[71] Laravel. *Laravel Quickstart*. URL: https://laravel.com/docs/4.2/quick (visited on 21/02/2017).

[72] Laravel. *Laravel Homestead*. URL: https://laravel.com/docs/5.3/homestead (visited on 21/02/2017).

[73] Adam Wiggins. *XI. Logs*. 30th Jan. 2012. URL: https://12factor.net/logs (visited on 14/12/2016).

[74] Heroku. *Logging*. URL: https://devcenter.heroku.com/articles/logging (visited on 06/01/2017).

[75] Play. *The Logging API*. URL: https://www.playframework.com/documentation/2.5.x/ScalaLogging (visited on 07/01/2017).

[76] Qos. *Chapter 3: Logback configuration*. URL: http://logback.qos.ch/manual/configuration.html (visited on 07/01/2017).

[77] Adam Wiggins. *XII. Admin processes*. 30th Jan. 2012. URL: https://12factor.net/admin-processes (visited on 14/12/2016).

[78] Heroku. *One-Off Dynos*. URL: https://devcenter.heroku.com/articles/one-off-dynos (visited on 26/02/2017).

[79] Lightbend Inc. *Running*. URL: http://www.scala-sbt.org/0.13/docs/Running.html (visited on 22/02/2017).

[80] Lightbend Inc. *Custom settings and tasks*. URL: http://www.scala-sbt.org/0.13/docs/Custom-Settings.html (visited on 22/02/2017).

[81] Jim Weirich. *RAKE Ruby Make*. URL: https://ruby.github.io/rake/ (visited on 02/02/2017).

[82] Mohit Natoo. *Rails 5 brings consistency by wrapping all rake commands using rails*. URL: http://blog.bigbinary.com/2016/01/14/rails-5-supports-rake-commands-using-rails.html (visited on 02/02/2017).

[83] Rails. *The Rails Command Line*. URL: http://guides.rubyonrails.org/command_line.html (visited on 02/02/2017).

[84] Laravel. *Console Commands*. URL: https://laravel.com/docs/5.3/artisan (visited on 17/02/2017).

[85] Noah Slater. *Why Your App Wont Work In The Cloud*. 24th Apr. 2014. URL: https://blog.engineyard.com/2014/why-your-app-wont-work-in-the-cloud (visited on 22/02/2017).

[86] Inc Amazon Web Services. *Creating a Custom Amazon Machine Image (AMI)*. URL: https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/using-features.customenv.html (visited on 22/02/2017).

[87] Bozhidar Bozhanov. *The 12-Factor App: A Java Developer's Perspective*. 25th Aug. 2015. URL: https://dzone.com/articles/the-12-factor-app-a-java-developers-perspective (visited on 22/02/2017).

[88] Havoc Pennington. *Scala on Heroku*. URL: http://www.lightbend.com/blog/scala-on-heroku (visited on 23/02/2017).

[89] Rails Hurts. *Why it happens to your code again and again?* URL: http://railshurts.com/mess/ (visited on 23/02/2017).