Degree Thesis, Åland University of Applied Sciences, Bachelor of Information Technology

SOFTWARE LIFECYCLE

- A hitchhiker's journey to software lifecycles of today and yesteryear

Lucas Pentinsaari

2021:16

Datum för godkännande: 18.05.2021
Handledare: Björn-Erik Zetterman

DEGREE THESIS
Åland University of Applied Sciences

Study Program:	Bachelor of Information Technology
Author:	Lucas Pentinsaari
Title:	Software Lifecycle - A hitchhiker's journey to software lifecycles of today and yesteryear
Academic Supervisor:	Björn-Erik Zetterman
Technical Supervisor:	

Abstract

The thesis dives into the software life cycles used today and looks back on how it all used to function without all the automated tooling we have now.

In order to understand the software life cycles further, I created a proof of concept using Spring Boot, OpenAPI, Express, React, MongoDB, Docker, Github Actions and Heroku.

The result of the proof of concept grew into a three-layered system with a short software life cycle that uses continuous integration to do automated testing and deployment to a production environment hosted on Heroku.

Keywords

Java, Spring Boot, GitHub Actions, Heroku, JavaScript

Serial number:	ISSN:	Language:	Number of pages:
2021:16	1458-1531	English	35 pages

Handed in:	Date of presentation:	Approved on:
10.05.2021	12.05.2021	18.05.2021

EXAMENSARBETE
Högskolan på Åland

Utbildningsprogram:	Informationsteknik
Författare:	Lucas Pentinsaari
Arbetets namn:	Mjukvaru Livscykel - En vandrares färd till mjukvaru livscykler idag och förr
Handledare	Björn-Erik Zetterman
Uppdragsgivare:	

Abstrakt

Examensarbetet dyker in i mjukvaru-livscykler som används idag och tittar tillbaks på hur det funktionerade utan alla automatiseringsverktyg som vi har idag.

För att kunna förstå mjukvaru-livscykler bättre, har jag skapat ett koncept-projekt med hjälp av Spring Boot, OpenAPI, Express, React, MongoDB, Docker, GitHub Actions och Heroku.

Resultatet av koncept-projektet växte till ett tre-lager-system med en kort mjukvaru-livscykel som använder sig av kontinuerlig integration för att kunna göra automatiserad testning av koden, samt driftsättning till en produktionsmiljö med Heroku som värdsystem.

Nyckelord (sökord)
Java, Spring Boot, GitHub Actions, Heroku, JavaScript

Högskolans serienummer:	**ISSN:**	**Språk:**	**Sidantal:**
2021:16	1458-1531	Engelska	35 sidor

Inlämningsdatum:	**Presentationsdatum:**	**Datum för godkännande:**
10.05.2021	12.05.2021	18.05.2021

TABLE OF CONTENTS

1. INTRODUCTION

1.1. Purpose

This thesis studies the life cycles of modern software, whilst looking back at the past to see what might be a factor in the massive leap of wanting to automate the procedure of releasing software to the world.

The thesis has presented an opportunity to create a proof-of-concept pipeline which takes the software from the developer's own machine and forwards it straight into production, all while having the software become tested and deployed through automation.

1.2. Background

The software development life cycle is planning, developing, testing, and deployment of applications, which is taught early on during software engineering studies, where an example of this is at Åland University of Applied Sciences, where such a course helps the student become familiar with going through the entire life cycle.

The life cycle follows a set of pillars as illustrated by Figure 1, of which developing, testing, deployment, and maintenance are pillars where software takes the stage.

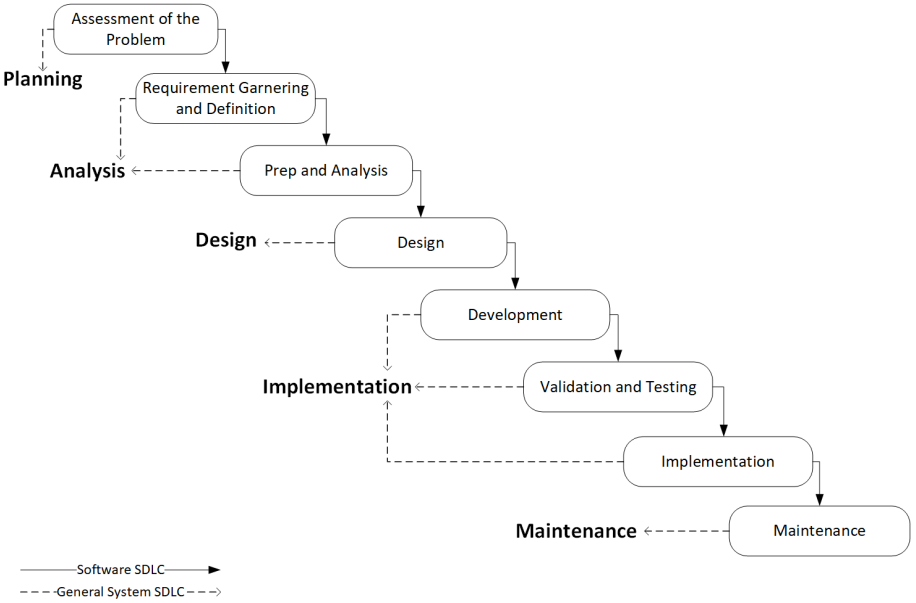


Figure 1.Software Development Life Cycle (Offor, 2020)

The implementation stages are also where a lot of software development and IT operations pipelines are set in order to make sure that the transition between having the software on the local machine and the production machine goes smoothly with automation for testing and validation of the software.

Having little experience with the software development and IT operations pipelines, it was essential to learn more about what they are like right now and how they might have looked in the past. Not only that, but how I would want it to look like in a small proof of concept.

1.3 Hypothesis

The rise of various software development tools over the past two decades has given a path for potential automation and quality assurance to be more relevant than ever before.

2. METHODOLOGY

2.1. Survey

To begin the glorious quest for knowledge, it had to start from a place close to heart, aptly named the programmer's hangout[1], where an online community with a staggering community of eighty thousand members resides, which is where the entire thesis idea originated from in the first place.

Building up knowledge requires people from all walks of life. Having already been a long-time member and moderator at the programmer's hangout, It was bound to have selected well-known members that could answer at least some of the questions regarding how the software life cycle works today.

A survey had to take place to get a solid grasp of the specific software development environments and their software life cycles.

The survey itself was anonymous and followed two simple rules:

- No personal information.
- No Non-Disclosure Agreement[2]-related matters should ever be disclosed.

These rules were a helpful hand in getting people to take the survey in the first place.

Google Forms have been used to conduct the survey, which helped to create a straightforward and nicely formatted survey with minimal effort, which then, later on, could be exported to an excel spreadsheet, a CSV[3] file, or be read directly off the survey platform.

[1] https://theprogrammershangout.com/
[2] Also known as NDA
[3] Comma-Separated Values | https://en.wikipedia.org/wiki/Comma-separated_values

2.2. Email interview

To continue the glorious quest for knowledge as we now have the current data, past data was needed of what was going on whenever people were taking on the software development life cycle into their professional field.

Much like the survey in the previous section, this time, the same concept was planned like this:

- No personal information.
- No NDA-related matters should ever be disclosed.

The same pattern would also be used where two specific questions would be the main point and having the other questions would be syntactic sugar to compose an interesting survey.

The data and research gathered on the topic through old papers on software development life cycles became crucial in seeing how the current life cycle has been developed into a highly collaborative system.

2.3. Proof of Concept

In order to solidify the lessons learned with the survey and email interview, a Proof of Concept has been created. The basic idea is to create a three-layered system consisting of an API, Client, and Database, where the API talks to the client and database, the database only does what the API asks for and the client asks for information from the API.

The product will pass through stages before appearing in the production build, which is where more tools are introduced in order to make sure that everything is tested and ready for release. Compared to some workflows however, it will not go through a release candidacy, but simply be released as stable once it has passed all requirements.

3. SURVEY OF THE CURRENT SOFTWARE LIFE CYCLE

3.1. Introduction

Ever since the first computer came into existence, humans have always wanted to make them even more effective, and they have done it now for a long time. Instead of having computers be as big as an entire room, they are everywhere in all electronics possible.

Much like the computer, programmers worldwide have tried, and succeeded in, making the world more efficient through software, especially in the ability to collaborate online through the power of the internet. One of the most incredible collaborative efforts today is the Linux kernel, which has had over 8000 collaborators create 15 million lines of code (*How Linux Is Built*, 2012).

Native software development, where software is built natively on a machine and used on the same machine, has been and is still the de facto way of building applications. Still, much of the world is now moving towards containerized, serverless solutions and more.

However, the way we develop applications is not the only thing that has seen a significant change compared to the past. The Agile methodology has taken over the corporate structure for the dev teams, which has led to rapid development led by sprints, planning, and a lot of communication between companies and their customers.

By combining Agile methodology and faster deployments, you can see yourself having what the world currently looks like, with a lot more to come once more companies start to make their pipelines even more effective.

3.2. Background of respondents

As described in the methodology, no personal information was ever to be revealed. However, we need to establish the background to dig through the data in an orderly fashion.

The generalized information is as follows:

- All participants are either students who contribute to open-source or have at least one year of working experience in the IT industry.
- All know at least one programming language.
- All use at least one or more operating systems.

Using this as the minimum baseline, the software life cycle can be as simple as using a local development environment natively and then uploading the produced content through FTP (*File Transfer Protocol*), or it can go well beyond that and be an entire pipeline that has been well established since years ago.

22 people participated in the survey in total and a supplementary five people who answered my emails regarding the life cycles decades ago, so let's go through the survey results to see today's software life cycles and what interesting new technologies might be pursued in the future.

Check the appendix at the bottom of the thesis in order to see the questions presented to the various groups.

3.3. Can you describe your current software lifecycle?

The question itself has given a mixed bag of answers, which will need to be summarized.

Containerized software life cycles were quite common and followed a flow where the developer could use the container locally, followed by being pushed to a version control repository. It goes through the source with a continuous integration tool that would build an image out of the container document, gets tested using various means of testing such as unit

tests, integration tests, and more. The staging environment was quite common to where a newly built image that has passed all the automated testing would end up in. This stage usually has some other team test the code through interaction, as an end-user would. Later on, when the other team completed all the testing, a final build would be made and released into production.

Unit testing is the means of testing individual units of source code (Wikipedia contributors, 2021e). It can, in some sense, create predictable source code. The predictable part is that the unit test itself proves that the tested code could be used in the way that the unit test intended it to be used.

Unit tests are often automated tests that the software developer can run. In the case of the software life cycle, it can also run during the continuous integration part, creating a failed build if the unit tests fail for whatever reason.

Integration testing is the phase in software testing that takes the previously unit tested modules and groups them together, tested as a group (Wikipedia contributors, 2020).

Serverless architecture seems to have gotten into projects and used in full swing with a slew of services from AWS[4] (*Amazon Web Services*), Microsoft Azure[5], and GCP[6] (*Google Cloud Platform*), where the entire point of using these services is not to be reliant on managing servers themselves (*What Is Serverless?*, n.d.), hence the word serverless.

Depending on how reliant a company might be towards a particular serverless service provider can create a lock-in of sorts, which means that the pipeline created only caters to a specific service provider.

The route to production for the serverless architecture follows a similar road as the containerized life cycle did, where external services do automated builds, various tests, and

[4] https://aws.amazon.com/
[5] https://azure.microsoft.com/en-us/
[6] https://cloud.google.com/

other different things before it goes into production, With this booming age of technology that we currently live in, one software life cycle will most likely never stop existing: the native development environment.

When native development environments are spoken of, there has been a joke going around since the dawn of personal computing that goes "it works on my machine" (seen in Figure 2) as the prime excuse for when software does not work as intended on another machine. This is due to non-predictable behavior in computers, which containers can help with. It will set up a development environment that looks similar to what the production environment looks like with the same software available. However, it is not always fool-proof.

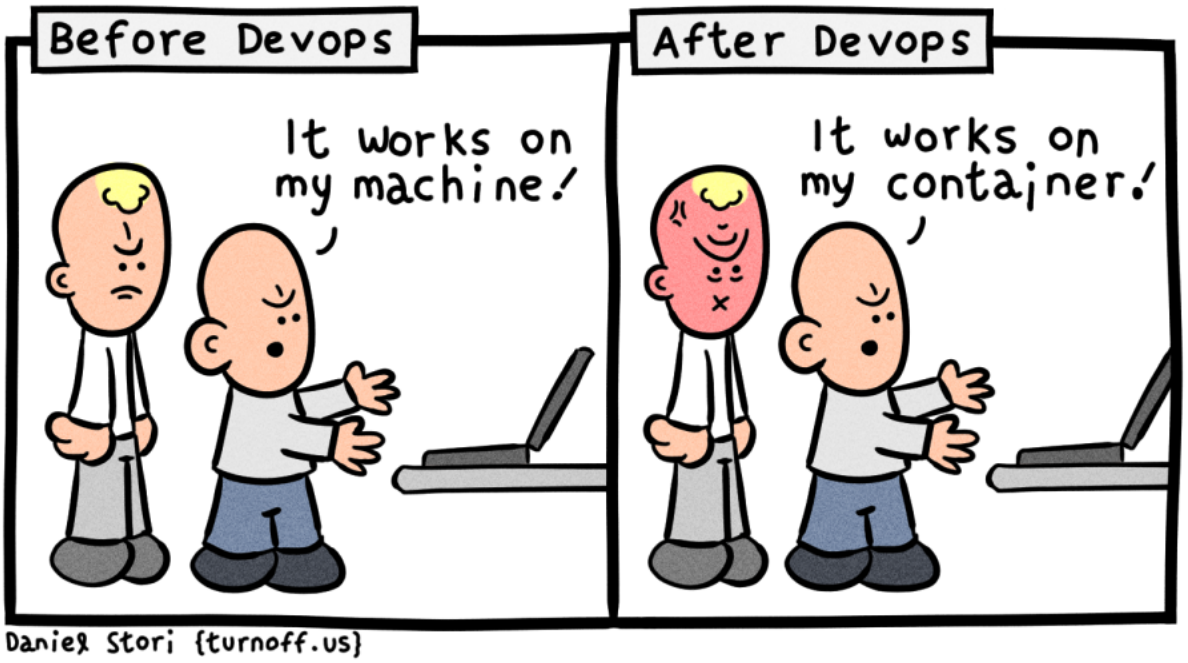


Figure 2. It works on my machine (Before Devops / after Devops, n.d.)

Even with the native environment, the same kind of structure and life cycle can be expected. Still, instead of having the benefit of containers or serverless services, the same software tool can be used whenever there's a need for a build. Still, now it's most likely contained within a server that takes complete control of the resources.

Like the container and serverless setup, a product is often released in different stages, having other teams test the software before releasing it to production.

In figure 3 is a pie chart over the 22 participants of what type of environment their software life cycle used, seeing a clear winner of the native environment, containers coming in second place and serverless coming third.

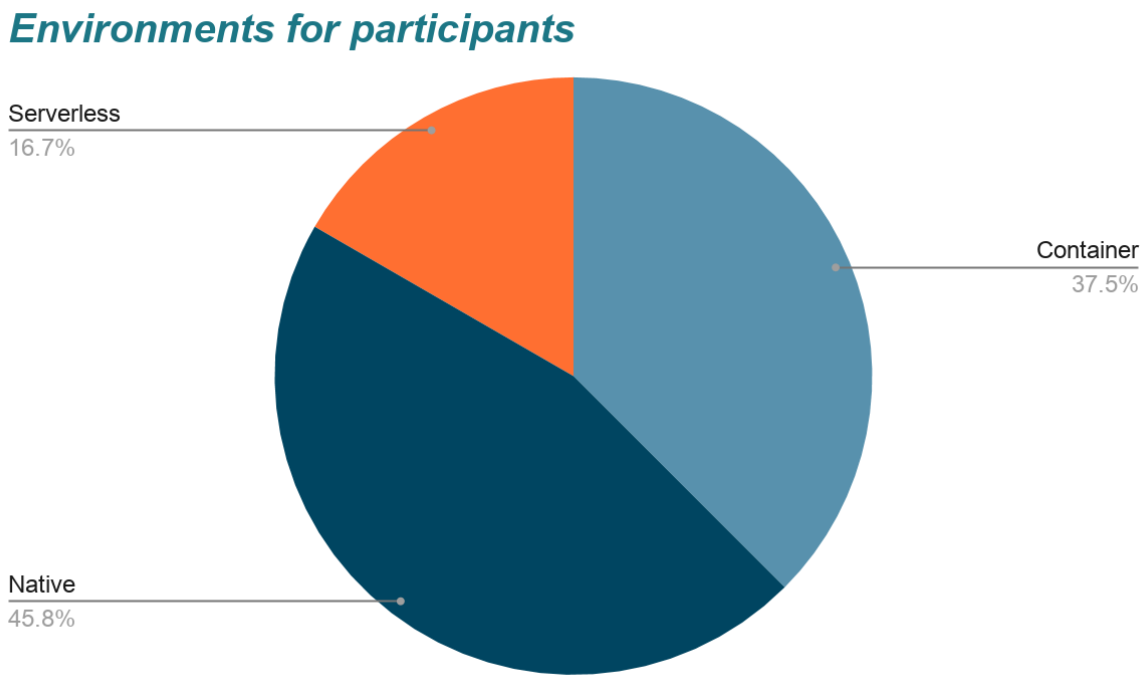


Figure 3. Survey participants environment usage.

3.4. Comparing past and current software life cycles

Considering what the previous section provided to what exists today in software development, with what pipelines have been explored, how much automation is done today compared to the automation in the past.

It can be said that the IT industry today focuses a lot more on fast deliveries, rapid development, and high automation to the point that the pipeline between local and staging areas should not have any use of an actual person to manage anything.

Depending on how far back we would like to go, we can see a lot of things that are second nature to us today, become a prototype or perhaps be the initial release of a specific tool.

Only going back a decade, and we can see the early stages of Docker adoption (Wikipedia contributors, 2021c), Kubernetes only being a pipe dream (Wikipedia contributors, 2021d), Jenkins released only months prior (Wikipedia contributors, 2021a), and Java SE 7 releasing in a few months (Wikipedia contributors, 2021g). Bitbucket had already garnered a few years on its neck, the very same as the other version control systems, which had sprung up at about the same year as Bitbucket. Something blasphemous today is that it was not second nature for all programmers to have your code in a repository or that you needed to use a version control system to manage the state of the project.

Going back even further, say two decades ago, we would see that seventeen software developers met at a resort, talking about lightweight development methods, which then came to be called the Manifesto for Agile Software Development (*Manifesto for Agile Software Development*, n.d.), Eclipse(Eclipse Foundation, Inc, n.d.), a popular Integrated Development Environment used by programmers, was to see an initial release in a few months. Git had not yet been born, and the computer would see the release of the AMD Athlon processor just two years prior.

So through the questions asked to those who have been in the industry for some time, it has become clear that development was not nearly as rapid as it was today. However, the pipeline itself might not be as changed as we would like to think.

Version control systems had existed for a long time now, where CVS (C*oncurrent Versions System*) was very early in the cycle in 1990 when the first release was made (Wikipedia contributors, 2021f), followed by SVN (*Apache Subversion*) released in 2000 (Wikipedia contributors, 2021h).

The big selling point now, compared to one or two decades ago, is automation and the way we work as software developers. As a surprise was how well it seems companies have taken

the Agile methodology and formed it around the companies to how it could best work for them.

3.5. The future additions to the life cycles

With the second question being, "Is there something you would like to use in the future for your software life cycle?", the participants got the chance to express their curiosity about what they wanted their software life cycle to include in the future.

Many participants reported that they either wanted or predicted that Kubernetes would be in their software life cycle in the future. A minority saw an opportunity of wanting to expand their automation process further.

Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications (*Production-Grade Container Orchestration*, n.d.).

To simplify the explanation, Kubernetes manages the state of containerized application through scaling, health checks, load balancing, and more, and to simplify it even further would be:

You are a kindergarten teacher, the load of work you will need to do during a day is massive as handling children can be a tough job depending on the children. While on recess, your job might be to look out for the children so that they might not get hurt during playtime, or if someone is fighting, it is your job to put an end to that. Managing the children is your priority, and likewise, for Kubernetes managing the containers is its priority making sure that all of them works as predicted through manifests and automation.

4. PROOF OF CONCEPT

When a TV show wants to air on a particular television network, a pilot episode[7] is frequently used to make the TV network interested in the show. The pilot episode and a proof of concept go hand in hand with the premise of wanting to sell something to someone without showing off the finished product.

In this scenario, the proof of concept is a three-layered software stack where information is a linear road from start to finish as shown in figure 4. However, it is simply not the software that is the big focus, but instead the pipeline from which the software stack will travel.

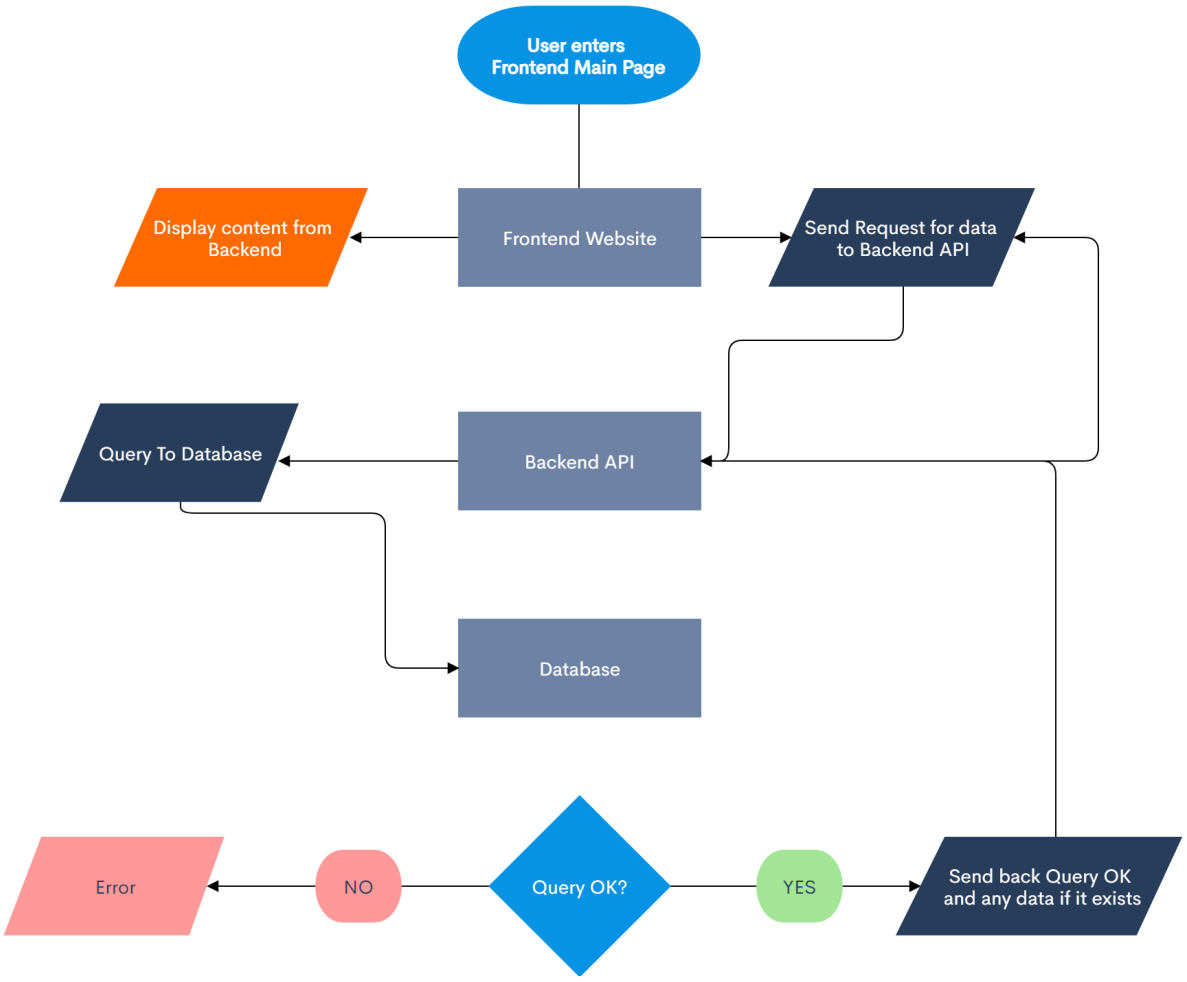


Figure 4. Three-layered system flowchart

[7] https://en.wikipedia.org/wiki/Television_pilot

4.1. Technologies

4.1.1. Application Programming Interface

4.1.1.1. Java

Java is one of the most used programming languages to this date, it is so popular that it came in second place on the TIOBE list of popular programming languages, Before April 2, 2020, it was number one on the list (*TIOBE Index*, n.d.), which only says that people like to use the language, but not really why they like to use it, and the reasons are many, such as:

1. Platform Independent, as it can run on anything as long as the JVM (*Java Virtual Machine*) is close by
2. Simplicity, as it is a very simple language to pick up and learn.
3. Robust, as it features garbage collection and excellent exception handling which helps developers procure good programming habits which create reliable applications.

These three points alone make up for what makes Java popular (with more available), and what is probably even more probable for the booming popularity is the rise of the software frameworks, such as Spring Boot[8] with its core framework called Spring.

A software framework is like a library of generic functionality that can be selectively changed by additional user-written code and changed but extended and removed for the user to fit the framework to the project at hand (Wikipedia contributors, 2021b).

4.1.1.2. Spring Boot

Spring Boot (see Figure 5) is a Spring-based configuration. A solution for creating stand-alone, production-grade Spring-based applications (*Spring Boot*, n.d.) and was used to make a REST API (Representational state transfer Application Programming Interface) that would talk to the client website and send data that were acquired from the database.

[8] https://spring.io/projects/spring-boot

Figure 5. Spring framework logo (File:Spring Framework Logo 2018.svg, 2018)

A REST API uses HTTP to create interactive applications that respond to HTTP requests, a specific kind of request is a GET request, which only fetches data upon what instructions the REST API has received (IBM Cloud Education, n.d.).
An example can be:

```
GET /projects
```

It can be translated to http://your-website.com/projects.

When the web browser or application make this request, it will go to the Spring Boot application, which checks for the available projects from the database, which then sends back a response, and what you receive is the result of that request and a 200 HTTP status code (as seen in Figure 6), which is the HTTP status code that the current request was OK.

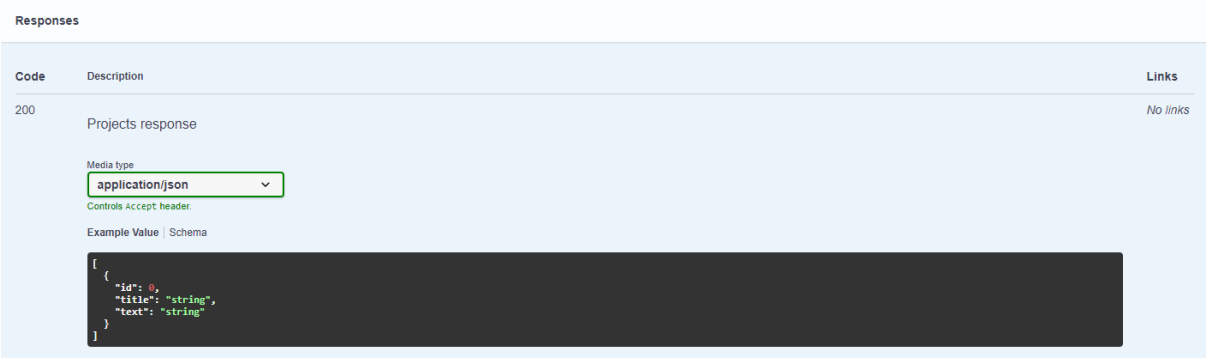


Figure 6. OpenAPI displays what kind of response can be received off of the project's endpoint.

4.1.1.3. Gradle

Gradle is an incredible build automation tool for a range of programming languages (*Gradle Build Tool*, n.d.). It introduces Groovy & Kotlin-based DSL (domain-specific language) for creating project configuration files, compared to Maven, another build automation tool for a

range of programming languages that use XML as their go-to language for the same task (*Gradle Build Tool*, n.d.; van Zyl Vincent Siveton, 2006).

Maven:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <version>2.4.5</version>
</dependency>
```

Gradle:

```
dependencies {
implementation group: 'org.springframework.boot',
 name: 'spring-boot-starter-web',
 version: '2.4.5'
}
```

Above displays the difference in what both project configuration languages look like whenever a project needs a particular dependency.

In the proof of concept, the idea was to use Gradle to generate classes through the OpenAPI specification, which defines a specification for describing, producing, consuming, and visualizing RESTful web services (*OpenAPI Specification*, n.d.).

4.1.2. Client (Website)

4.1.2.1. ReactJS

React is a JavaScript library for building user interfaces (*Getting Started*, 2020). The website used it to create a straightforward user interface for reading the API's response, formatting the data, and displaying it.

React uses states to change data if it's available or not. Using that proved helpful in showing whenever the web application was loading in the data or if it had already done it and displayed it directly.

```
const App = () => {
const ListLoading = loadingCards(List);
const [ appState, setAppState ] = useState({
        loading: true,
        projects: null,
});
useEffect(() => {
        setAppState({ loading: true });
        fetch (API_URL)
                .then(res => res.json())
                .then(projects => {
                        setAppState({ loading: false, projects: projects });
                })
}, [setAppState]);
```

4.1.2.2. ExpressJS

Express[9] is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications (*Express - Node.js Web Application Framework*, n.d.).

In order for the web application to not use the development server that ReactJS provides, Express was used in order to handle all the server-side requests that might have arrived, such as requesting certain routes and handling all of the things which might be sorely needed, since React builds as a static web application and it would then be up to ExpressJS to send the correct web page.

```
const express = require('express')
const path = require('path')
const app = express()
const cors = require('cors')
const port = process.env.PORT || 3000;
```

[9] https://expressjs.com/

```
app.use(express.static(path.join(__dirname, 'build')))

// load the value in the server
const { API_URL } = process.env;

const corsOptions = {
  origin: API_URL,
  optionsSuccessStatus: 200
}

app.get('/*', cors(corsOptions), function (req, res) {
  res.render(path.join(__dirname, 'build', 'index.html'), {API_URL})
})

// default Heroku PORT
app.listen(port, () => {
  console.log("Server has started on: " + port)
})
```

We first set up the necessary packages required to run the express application, adding a Cross-Origin Resource Sharing (CORS)[10] helper to accept requests or responses from the API. This is then set so that whenever something or someone sends a GET request to the root path, it should render index.html from the build folder and pass along the API_URL, which has been defined as an environment variable previously.

4.1.3 Database

4.1.3.1. MongoDB

One of the databases that previous experience existed for was MongoDB, an open-source document-oriented NoSQL database. What does a document-oriented NoSQL database mean?

NoSQL is referring to non-SQL or non-relational. This inherently refers to not using SQL (Structured Query Language), a domain-specific language used in programming and designed to manage data held in a relational database.

Any type of database that does not follow the typical structure of using SQL or is stored in a relational database is referred to as NoSQL, which is valid for MongoDB. It uses JSON-like

[10] https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS

documents that any programmer used to JSON documents would also be used to MongoDB documents per default.

a MongoDB document can look like this:

```
{
      "_id": {"$numberLong":"1"},
      "title": "API",
      "text": "Uses MongoDB, Spring Boot, OpenAPI & JUnit",
      "_class":"dev.dreameh.backend.service.domain.Project"
}
```

To make use of the MongoDB database, MongoDB Atlas was used, as it is MongoDB's own on-demand fully managed service (as seen in Figure 7), which runs on AWS, Microsoft Azure, and Google Cloud Platform, and the plan which was used for this project was a free tier plan which gives a single cluster of 3 nodes, ready to be used at any point.

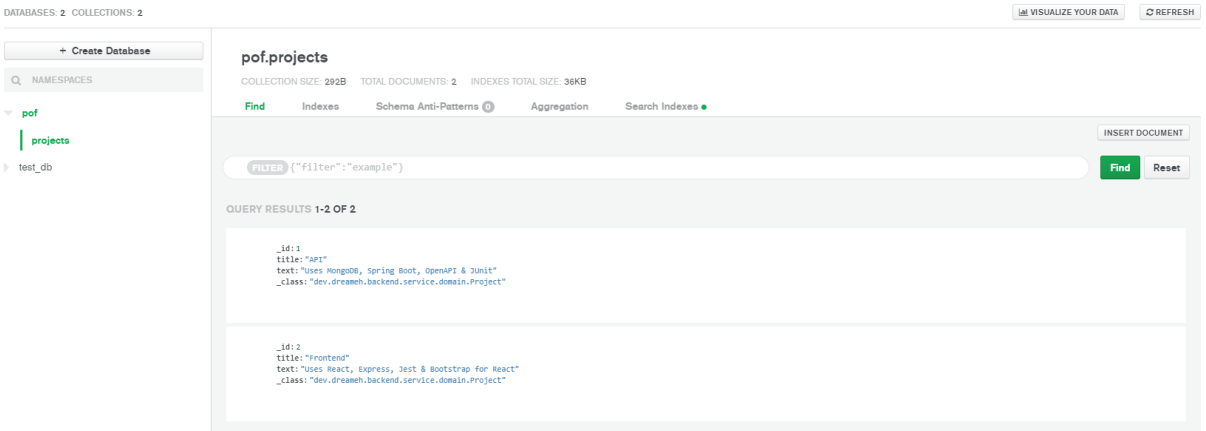


Figure 7. The MongoDB Atlas Collections with available documents

4.2. Local Environment

With Docker, Docker-compose, and Git, an excellent local development environment can be achieved. Docker was used to create container images and write Dockerfiles, which was then used by Docker-compose, which is a side project of Docker to connect different container images to work together as services for a bigger whole.

Using Docker-compose, it was possible to create all container images, connect a network between them, and then run them in one command.

```
$ docker-compose -d up --build
```

This helps to check the health of all images, seeing if they are all alive and running. If not, then no service will be started, and if a change is done to just one of the projects, then another command can be used to rebuild the image in question, having the rest still up and running.

```
$ docker-compose -d up --build service_name
```

These two commands were the only thing needed during the development as the Dockerfiles included all the information to build and run the applications.

4.2.1. Docker

Docker is a platform for developers and system administrators to develop, deploy, and run applications with containers. Containers allow for total application and environment isolation which makes it much easier to ensure an application runs with any necessary dependencies.

This means that with Docker you are OS-independent, no need to check for whether a specific dependency can work on your system, or whether the application you are trying to run, actually runs as expected. In figure 9 shown below you can see the creation of the services used in the proof of concept.

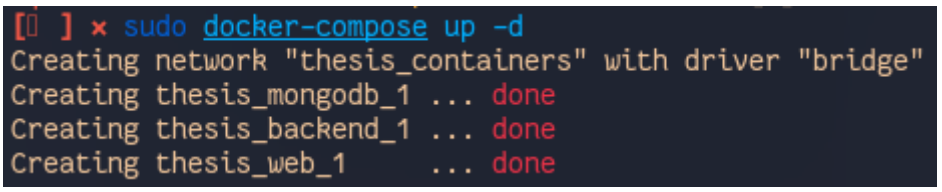


Figure 8. Docker running through docker-compose to create docker images to run the local environment

4.2.2. Git

Git[11] is a free and open-source distributed version control system; this tells us that we can handle different revisions of our codebase within a repository.

The repository consists of branches. These branches allow people to work on the same codebase without locking a specific file for whenever two people are working on the same file, as changes are treated as different revisions of the same file.

There are a wide variety of git server services out there, and the bigger ones are:

- GitHub[12]
- GitLab[13]
- Atlassian (using Bitbucket)[14]

However, these services are much more than just Git servers and instead incorporate collaboration features, making it quite a lot easier to work together to reach a common goal.

4.3. Continuous Integration/Deployment

Continuous Integration is automating the integration of the code change from multiple contributors into a single project (Atlassian, n.d.).

It takes new code changes and goes through it with the help of automation tools that test the correctness of the code itself through different varieties of testing suites, building the project, and checking whether it follows code standards if any of such things are set up.

Continuous Deployments is automating the process of automatically deploying new builds of different software. In the proof of concept, the continuous deployment is a part of GitHub Actions whole process, with deploying the code to Heroku at the very end.

[11] https://git-scm.com/
[12] https://github.com/
[13] https://gitlab.com/
[14] https://www.atlassian.com/software/bitbucket

4.3.1. Github Actions

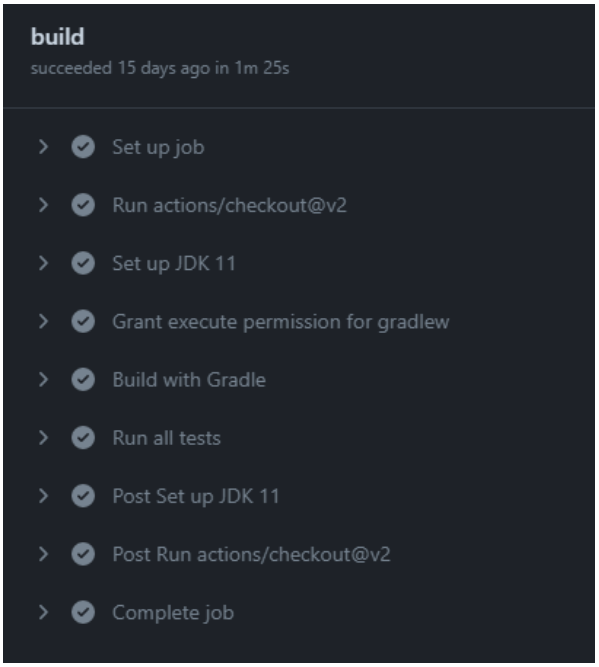
GitHub Actions is a new CI/CD tool that builds, tests, and deploys code directly from Github repositories. It is pretty similar to GitLab CI in that you need a workflow document that describes the job you should perform whenever a commit has reached the repository.

In the process of creating the proof of concept, it became clear that a large number of survey users use continuous integration tools, much similar to Github Actions. Since it is a pretty new tool, it was only fitting to test it in the proof of concept to do the process of building, testing, and deploying the code.

For the API following process runs each time a new push or pull request reaches the repository:

- Check out the revision
- Set up Java JDK[15] 11
- Build the project using Gradle[16]
- Test the project using Gradle

If these job steps are cleared as shown in Figure 10, it will proceed to the next stage to be deployed to Heroku using a different workflow document as shown in Figure 11.



[15] Java Development Kit
[16] https://gradle.org/

Figure 9. All steps which Github goes through in the workflow document

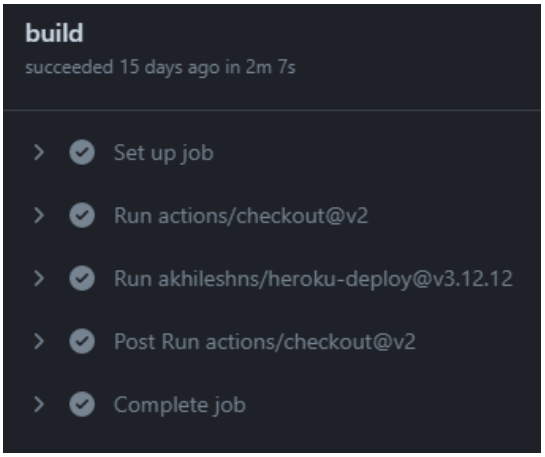


Figure 10. When the first workflow document is done, the deploy document will run.

A similar workflow is done for the Client, which is as follows:

- Check out the revision.
- Set up NodeJS 15.12.0
- Install dependencies using Yarn
- Build the project using Yarn

Since the frontend is relatively small, it felt unnecessary to use any testing other than a simple build test, as React is quite good at finding out what works and what does not during build time.

4.4. Production

4.4.1. Getting to Production

Getting to production is quite an easy task for what has been set up as a pipeline as shown in figure 12 below.

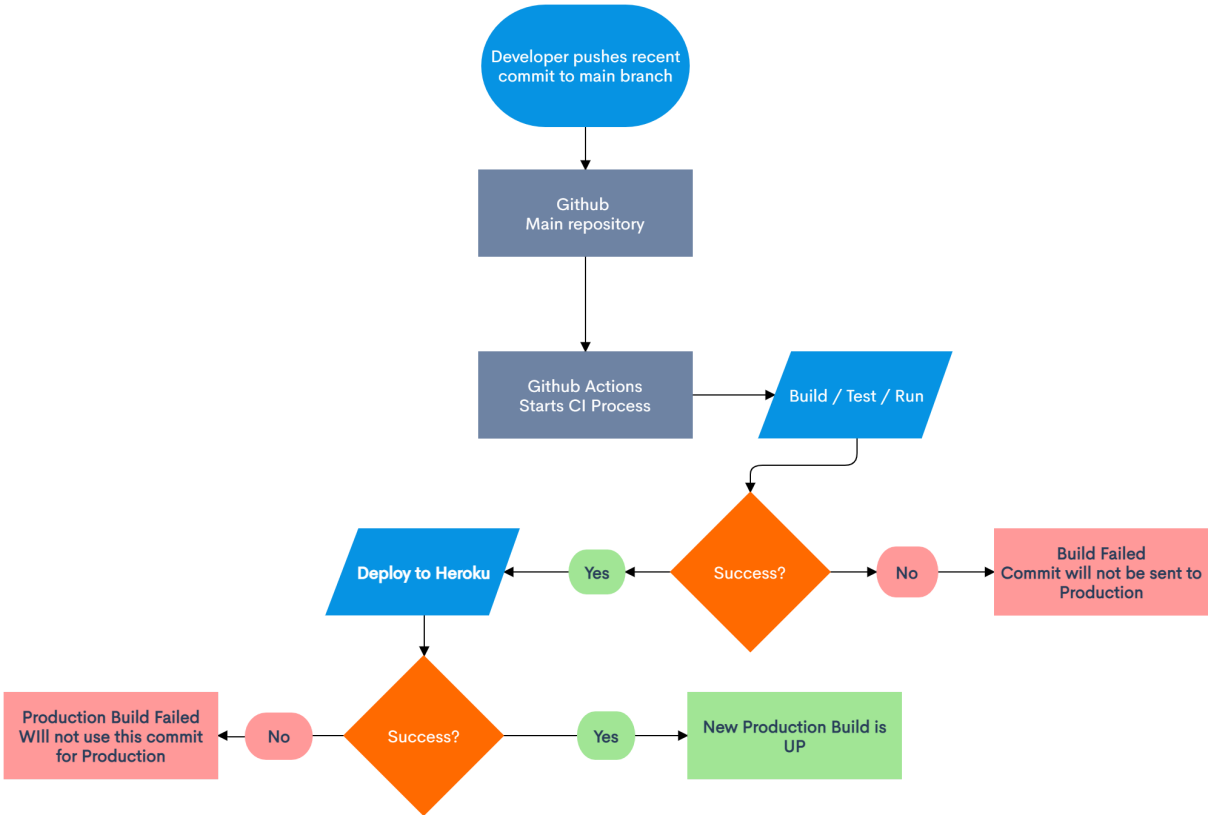


Figure 11. Flowchart of the entire process from when the developer pushes something to the main branch.

- We start by pushing the latest git commit to the main GitHub repository.
- When it arrives at the GitHub repository, GitHub actions kick in and goes through the workflow document steps, which have been defined.
- If the first workflow document is successful, it will go to the deployment phase of the workflow.
- To deploy to Heroku, Heroku will run a build of the application and then set that version if successful as the current production build. If not, it will not do anything.

The pipeline itself is straightforward, and if all code is properly tested, it will most likely find a culprit. A slight problem can arise, for when there exist no tests overall but the code itself can be built, a faulty build might end up in production, which is why code needs to be tested and be strict for wrong code to not arrive in production.

Another way, which was not used in the proof of concept, was the idea of staging builds, which Heroku has official support for where instead of a development build going straight into production, it would stay in staging until the staging build has been manually tested.

4.4.3. Comparison

The main difference between the local and production environment should always be fault tolerance. The product should be able to be in a state where bugs get caught with tools specialized for such purposes in a local environment. In contrast, the production environment should be like a mountain, sturdy and immovable, with no errors or bugs.

This is the sole reason why GitHub Actions is used, as it builds the project, tests the code. After that is done, it sends it away to production, and because of that, it fails fast, as no non-predictable code will go through, with the help of unit testing and other kinds of tests available for the CI to perform.

However, it can only go as far as testing allows it, as non-predictable behavior can appear outside of the scope of the testing.

To further discuss the difference between the local and the production environment, it can be said that the local environment focuses on being convenient to set up, build, test, and run as many times as possible with the use of Docker and Docker-compose. In contrast, the production environment sits on a PaaS (Platform as a Service), which uses a similar structure as Kubernetes with the pod system, but their version is called dynos. You define it through a Procfile of what build pack you want to use and what command you want to run.

The PaaS is Heroku, a great complete hosting platform where you can take your application, go through the application with a CI, test, build, run and then put the application in a staging area. Once manually tested, transfer the application to the production stage.

The proof of concept only used Heroku as past experiences made it easy to set up and predict how the applications would work together as one defined unit.

It can also be said that even though Heroku nowadays supports using Dockerfiles as their configuration for the dynos, it was unchartered territory. It was quite a new feature, so I didn't dare use that, which meant that the only place where Docker was used was in the local environment, and it would then switch to the dyno in the production environment.

5. DISCUSSION AND CONCLUSION

5.1. Conclusions

It can be arduous to set up an entire project from the local development environment up to the production line. It has been shown that depending on the need and the know-how, you will end up with different results of how it can look like.

It can be said that with the uprise of tools of convenience, automation, and predictability, the software life cycle of today has steadily changed and is likely to change even further as the thirst for effectiveness is ever-growing.

It can also be said that the software life cycle of yesteryear looked widely different from what it looks like today. Many convenience tools started their uprising two decades ago, and with each year since, it has given even more tools and toys for developers to use and play with.

With the rise of the tools, the rise of Agile Methodology and all of its branches rose side-by-side.

The first mini hypothesis was that most probably already had converted to a container pipeline or a serverless, but to see that many today work in a native environment was surprising.

5.2. Limitations

While the entire software development life cycle itself is an interesting topic, some constraints had to be applied. Not only because the thesis would get to be way longer than expected, but it is not entirely relevant to the current stance of what a junior programmer straight out of university will do.

As most things tend to evolve during writing, all information written will most likely have become outdated. This is the sole reason why the latest and greatest pipeline has not been used throughout the proof of concept creation.

The initial pipeline was initially going to be Kubernetes all the way. However, due to a very high learning curve and many failed attempts later, another pipeline was used, as is displayed in the proof of concept section.

Lastly, during the enterprise Java course, the software used was very much similar to what the API looked like in the end. The only difference was the use of OpenAPI in order to document the endpoint.

5.3. The Future

Looking at what the participants of the survey said gave some ideas of what to continue to look into further, seeing as serverless will most likely take over the world completely sooner than later, that would be a good spot to start and research whether it might be feasible for bigger projects to use it, and how much can be used without getting into a lock-in.

More automation is on the list, preferably finding ways of doing it more strictly so less lousy code can escape the continuous integration process.

REFERENCES

Atlassian. (n.d.). *Continuous integration*. https://www.atlassian.com/continuous-delivery/continuous-integration

before devops / after devops. (n.d.). https://turnoff.us/geek/before-devops-after-devops/

Eclipse Foundation, Inc. (n.d.). *The Community for Open Innovation and Collaboration*. https://www.eclipse.org/

Express - Node.js web application framework. (n.d.). Retrieved April 11, 2021, from https://expressjs.com/

File:Spring Framework Logo 2018.svg. (2018, December 10). https://commons.wikimedia.org/wiki/File:Spring_Framework_Logo_2018.svg

Getting Started. (2020, October 26). https://reactjs.org/docs/getting-started.html

Gradle Build Tool. (n.d.). https://gradle.org/

How Linux is Built. (2012). https://www.youtube.com/watch?v=yVpbFMhOAwE

IBM Cloud Education. (n.d.). *What is a REST API?* IBM. https://www.ibm.com/cloud/learn/rest-apis

Manifesto for Agile Software Development. (n.d.). https://agilemanifesto.org/

Offor, P. I. (2020, January 6). *General System SDLC & Software SDLC*. https://commons.wikimedia.org/wiki/File:General_System_SDLC_%26_Software_SDLC.gif

OpenAPI Specification. (n.d.). https://spec.openapis.org/oas/v3.1.0

Production-Grade Container Orchestration. (n.d.). https://kubernetes.io/

Spring Boot. (n.d.). https://spring.io/projects/spring-boot

TIOBE Index. (n.d.). https://www.tiobe.com/tiobe-index/

van Zyl Vincent Siveton, J. (2006, November 1). *Maven Getting Started Guide*. https://maven.apache.org/guides/getting-started/index.html

What is serverless? (n.d.). https://www.redhat.com/en/topics/cloud-native-apps/what-is-serverless

Wikipedia contributors. (2020, December 6). *Integration testing*. Wikipedia, The Free Encyclopedia.

https://en.wikipedia.org/wiki/Integration_testing

Wikipedia contributors. (2021a, February 25). *Jenkins (software)*. Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Jenkins_(software)&oldid=1008909894

Wikipedia contributors. (2021b, April 9). *Software framework*. Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Software_framework

Wikipedia contributors. (2021c, April 12). *Docker (software)*. Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Docker_(software)&oldid=1017343533

Wikipedia contributors. (2021d, April 12). *Kubernetes*. Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Kubernetes&oldid=1017385791

Wikipedia contributors. (2021e, April 14). *Unit testing*. Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Unit_testing

Wikipedia contributors. (2021f, April 16). *Concurrent Versions System*. Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Concurrent_Versions_System

Wikipedia contributors. (2021g, April 16). *Java version history*. Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Java_version_history

Wikipedia contributors. (2021h, April 17). *Apache Subversion*. Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Apache_Subversion

APPENDIX

Questionnaire

1. Can you describe your current software lifecycle?
2. Is there something you would like to use in the future for your software lifecycle?

Email Correspondence Questions

1. How could the process look like to take a software solution from the local environment (your computer) to a customer or an in-house production?
2. How did it look like with testing of the software solution?
3. What’s one thing that’s changed about the IT industry that surprised you the most?
4. What's one thing that's changed about the IT industry that you dislike the most?
5. What do you want to see in the future of the IT industry?