Karolis Giačas

# FULL STACK SOFTWARE APPLICATION

Bachelor's thesis

Double Degree

2021

South-Eastern Finland
University of Applied Sciences

South-Eastern Finland University of Applied Sciences

| Author (authors) | Degree title | Time |
|---|---|---|
| Karolis Giačas | Bachelor of Engineering | April 2021 |
| **Thesis title** | | 43 pages |
| Full stack software application | | 0 pages of appendices |
| **Commissioned by** | | |
| **Supervisor** | | |
| Timo Mynttinen | | |


**Abstract**

The goal of this thesis is to develop a full stack application with a system architecture similar to the current market standards. The project consists of a few major steps which will be analyzed.

Most of the theoretical background of this study is based on documentation provided by almost every service and package maintainer and includes multiple service comparisons. Also, a lot of information was gathered via developer blogs. The project itself consists of setting up a web framework and a server that would scrape data from the internet, store it and expose it as JSON string in an API endpoint. To accomplish this, multiple additional services will be used: database, task queue and cache storage. All of the services have to be containerized to fit the current market standards. It also increases developing speed, makes hosting less complex and helps to manage services as isolated containers. The next step is the creation of a simple interactive GUI that connects to the open API endpoint and fetches the data from the server. Finally, the server is hosted to any hosting service provider and the application is fully deployed and accessed from anywhere.

The conclusion of the study consists of testing the software. The application is fully deployed and working as intended.



**Keywords**

Docker, django, aws, sql, shell

**CONTENTS**

# 1 INTRODUCTION

The demand for multiple different development services is increasing toe to toe with the digitalisation of the globe. Currently there are hundreds of choices when deciding the technology stack for every application idea. Most of the current marketable applications consist of backend (server-side) and frontend (client-side) services which are developed by backend and frontend engineers. In the past few years, the concept of containerization is becoming more and more popular because it makes handling full stack development more standardized. Containerized applications are perfect for enterprise use because they can be managed more efficiently by controlling every service in an isolated container.

Full stack engineering consists of managing both the development and the operations phases (DevOps). Moderate programming skills are required for the development part, and system administration knowledge is necessary in the operations field.

The purpose of the study is to develop an application prototype which would be similar to the current application market standards. Multiple different technology reviews are included in the theory analysis. The core of the project is actually the development of the application’s backend. The first service developed is a web scrape. The information scraped will be related with some coronavirus statistics. The same application architecture approach can be applied to any types of statistics or information. The next services required are a task queue and cache storage. The web scraping function in the backend must run on a given time interval, so these services are crucial to accomplish that.
The last service in the backend is a web framework capable of opening API (Application Programming Interface) endpoints, modifying data, saving data and much more. For that, an MVC (Model-View-Controller) framework is ideal because it allows to control the routing of a web page, deals with database implementation and supports HTTP requests and responses.

# 2 FULL STACK DEVELOPMENT METHODS

The full stack topic has already become a new job trend, but what is it actually? A full stack engineer is responsible for setting up a project from start to finish or maintaining an already existing project and guaranteeing >99% service availability. There are five main services of a full stack project:

- Frontend
- Backend
- Database
- DevOps
- Mobile App (optional)

In web applications, the frontend is an actual representation of a page in the browser which is generated by HTML, CSS and JavaScript. HTML (HyperText Markup Language) controls the actual creation of the document and marks most of the components on the page. CSS (Cascading Style Sheets) is used to style the HTML components. And JavaScript makes the page interactive with various functions and events. This also includes mobile web frontend development. The most popular current frameworks are React, Angular and Vue.js. For software applications, the frontend is GUI (Graphical User Interface). There are a lot of ways to make a GUI, since most of the programming languages support canvas drawing modules such as “Tkinter” for Python, “Razor” for C#, “JFC/Swing” for Java.

Backend is the actual brains of the project. Here it comes to pure coding and to the idea of the project. This is a complete reverse of the frontend service and the user or a client sees almost nothing of how the backend actually works, and they do not see the code. When a tech-savvy person can meddle with frontend by using a browser inspect tool and actually read all of the code in plain text. But this is a case only for web applications (although fetching the code for GUI is not as simple as finding out the source code of a web page). Backend can be created with a variety of different technologies and frameworks, but some of the more known are “Django” (Python), “Laravel” (PHP), Spring (Java) and .NET Core (C#).

Databases are a straight-forward services which are used to store all kinds of data. Current standard language for relational database management systems are SQL (Structured Query Language). Two of the most common types of databases are Relational and Document. Relational databases are categorized by a set of tables in which every one of them has rows and columns while document databases are more like a subclass of a key-value store. Relational databases work well when the information has to be precisely connected with some other kind of information, for example, a user's shopping history in an e-commerce store. Document databases work better when there is a lot of data and it needs to be stored as fast as possible, these databases are popular for video games and mobile games.

DevOps (development & operations) is the bridge between developers and system administrators. It is a combination of tools and practices designed to increase the ability to deliver applications and services faster than traditional software development implementations. Probably the most important DevOps practice is CI/CD (Continuous Integration & Continuous Delivery). The technical aim of CI is to establish a consistent and automated way to compile, build, package and test applications. While CD picks up where CI ends, it is responsible for pushing code changes to all of the environments in an automated way (some possible environments - production, development, testing). DevOps is also responsible for containerization of the project. It helps to maintain a stable architecture of the applications and makes managing of multiple services easier because each service is created in an isolated container. Most popular container services – Docker and Kubernetes.

Mobile app development skills are a nice thing to have for a full stack engineer, but this is not required since not all of the web pages and not all of the software applications have a separate mobile app. But the demand for mobile apps is increasing every year and we might reach a point when every new business will want to have a mobile app as well as a web page / software application. The current flagmen languages for mobile apps are iOS and Java (Android development).

## 2.1 MVC frameworks

The backend for the project will have to consist of a web server that opens an API endpoint and returns HTTP (Hyper Text Transfer Protocol) responses in a JSON (JavaScript Object Notation) string which is basically a dictionary with key-value store. The backend will also have to scrape some information from the internet via a web scraper function. This function will also run on a given time interval so we will need task queue and cache storage services (2.3). Also, it will store the information in a dedicated database. To develop such a backend server, MVC (Model-View-Controller) framework is the best choice.

An MVC framework makes the life of a developer easier by separating an application's architectural pattern into three main logical components – Model, View and Controller. MVC separates the business logic and presentation layer from each other. MVC architecture is a must in the current web development job trends.

Model in MVC is responsible for declaring how the shape of the data looks and sends the shape for execution to the dedicated database. Also for changes to take effect, database has to be migrated. For example, model declaration in “Django” web framework looks like this -

```
Class User(models.Model):
    username = models.CharField(max_length=64)
    password = models.CharField(max_length=64)
    full_name = models.CharField(max_length=64)
```

Code 1. Django model

This is equivalent to creating a “User” table with three columns (username, password and full_name) in SQL. Also, here we add a max_length constraint to the columns.

Controller in MVC is responsible for handling user interaction with the page and less commonly software application. Controller sends commands to the model and a view to change their states, for example, saving a model instance after a user has successfully registered. Also it can change the associated view's representation. Basically, controller is the engine of a MVC framework.

```
send_context(request):
    user_list = User.objects.all()
    return render(request, 'index.html', { user_list: user_list})
```

Code 2. Django controller

In this example a function "send_context" is created which can be called whenever needed. A variable "user_list" is created which holds the information for all of the users in the database. Then it returns the "index.html" file to the same request with the user list embedded in a view.

Views (or templates in some MVC frameworks) are the actual representation of the page document to the end user. A GUI for a software application is equivalent to a view as well. Most of the web page views consist of three main parts – HTML, CSS and JavaScript. Most of the modern MVC frameworks do not use vanilla HTML, CSS and JavaScript, because there are many developers that are working on new frontend frameworks which would be more optimized and more powerful, but with a drawback of making the code more complex and more difficult to learn. There are three major stacks that the current job market finds very valuable – React, Angular and Vue.js. These frameworks are very similar when compared on a technical level, because all of them use the JavaScript programming language (or a branch of JS called TypeScript). Some of the key differences are that Angular is the heaviest of the frameworks (by page document size) and it has the steepest learning curve. All three have a component functionality which helps to build a more customizable UI. In Angular, components are more like references to HTML objects, while in React and Vue.js, they are combined with the UI which allows more customizability.

Table 1. Frontend framework comparision

| | React | Angular | Vue.js |
|---|---|---|---|
| Initial release | 2013 | 2010 | 2014 |
| Popularity | High | Medium | High |
| Job offers | High | High | Low |
| Community stars | 164k | 71k | 200k |
| Used by | Facebook, Uber | Google, Wix | Alibaba, GitLab |

For this project the actual view will be a GUI for the end user which can be created via any programming language. Since I'm most fluent in Python language, there are two main options – simple GUI creation module called "Tkinter" and an advanced variant called "Kivy". The latter has a lot of different advanced functionalities and is also the go-to module for Python mobile app development. But for this project we will just fetch data from an API and display it to the user so more basic "Tkinter" module will work flawlessly. There will be no view for the backend API endpoint because the response of an API request can be sent back by just using a controller instead of a separate view.

Choosing a backend MVC for a new project can be tricky because there is a lot of things to take into consideration. For example, do you need a minimalistic approach for the application/web or it will have a lot of traffic? Also it depends a lot on what programming language suits the developer best. There's six main frameworks that the job market asks for – Django, Express, Laravel, Rails, Spring and ASP.NET Core. The popularity of all the compared frameworks are pretty similar by the number of community members.

**Django** is a big framework which was created back in 2005. This might look old and outdated, but it is actually one of the main pros for this MVC. By having a really dedicated community over the years Django polished most of itself and became arguably the most secure backend service available. Python is getting a huge acknowledgment from the developer community over the past few years, so it results to developers migrating to this framework. Another key thing to mention

is that Django actually is not a MVC framework but an MVT (Model-View-Template), although practically it follows the same logic. Developers choose Django over other frameworks because it has a really well detailed documentation, is an open-source MVC and is not as difficult as other choices. One of the main downsides for Django is that it is completely based on an ORM (Object-Relational Mapping) which has a tendency of being slower, and it actually fails to compete with vanilla SQL for complex queries.

**Express js** is a very new framework released in 2019. This service is run on JavaScript and can actually compete with other MVCs in API development or a complete application development. The main pros for Express js is that it's using a minimalistic approach which makes it extremely fast. Also JavaScript's popularity is skyrocketing so this MVC might become the next default framework for backend development. They key con for Express js is that it's not very detailed in documentation which might be really challenging for beginners.

**Laravel** is a light-weight framework with a lot of functionality built in. It runs on PHP language which was extremely popular a decade ago. Because of that a lot of current web applications are constructed by Laravel. The framework itself saw the light of the day in 2011. This MVC has a built-in blade templating engine which allows to create simple but effective layouts. It also ensures high security and has a well detailed documentation. Although it's actually not the best MVC for huge websites because it can become too dependent on third-party tools. Also Laravel is still in the development phase and its package composer is not as great as other MVC composers (npm, ruby gems, pip).

**Rails** is considered a beginner-friendly framework because it is quite easy to modify and migrate. Also it has a great package composer called ruby gems which supports a lot of different packages. Rails runs on a Ruby programming language; the framework was released in 2004. This MVC has a active community behind itself as well as supporting a superior testing environment compared to any other MVCs. But it lacks some in quality and standardization in its documentation. Another few cons would be that it lacks in runtime speed and it

can get quite tricky to create a simple API. While it is quite simple for beginners, the learning curve can get extremely steep when trying to delve deeper into the framework trying to unfold its full potential.

**Spring** is run by the Java programming language which in the current years is referenced to as a *dead language.* Probably the biggest advantage is that Spring can be deployed by just assembling a jar artefact and it already uses an embedded web server (Tomact, Jetty, Undertow), but it also arguably makes it more exploitable, so security might be a big concern when choosing Spring. Another advantage in question is its own version control that is arguably better than other version control services. But the current development market environment is standardizing towards “git” version control, because it’s proven to be good over the years. Spring was released in 2002.

**ASP.NET Core** is another possible MVC framework to choose from. It is written in C# language, which currently is the go-to language for game development. Simple .NET applications can get quite complex and it’s not very beginner friendly. But most of the benchmarks shows that Core is an extremely fast and reliable framework. It also supports Visual Studio IDE which makes the development process faster, and it is backed by Microsoft which will remain the elite tech company for a while. Core was released in 2016.

**Phoenix** is an MVC framework which I believe requires an honourable mention. It is run on Elixir programming language and was released in February 2021. It’s currently in an infant stage compared to other MVCs, but it has already proven that it’s the fastest thing out there bypassing other MVC benchmarks sometimes even up to ten times. But since it’s very new and uses not-so-popular programming language only time will show if it has the potential to become the new leader in the framework race.

## 2.2 Containerization services

Containerization is a form of operating system visualization in which applications or services are run in isolated spaces called containers, all using the same shared operating system. A container is basically a fully packaged and portable computing environment. The key advantage of it is isolation because everything an application needs to work (binaries, libraries, configuration files, dependencies) is encapsulated in a single instance, as well the container is abstracted away from the host OS. As a result, containerized applications can be run on most types of computer infrastructure (bare metal, in virtual machines or in the cloud).

Since each container is an executable package that is running on top of a host OS it may support many containers (from one to thousands). For example, micro services architectures use numerous containerized applications. This works because all of the containers run minimally with resource-isolated processes that other containers cannot access. A containerized application is similar to a multi-tier cake: first there is a layer of hardware infrastructure including the CPU(s), disk storage and network interfaces. Secondly it is the host OS and the kernel which bridges the container's OS with the hardware of the underlying system. Thirdly a container engine sitting on top of the host OS is used, and lastly there are binaries and libraries for each application that is running in their isolated containers.

One of the most distinctive features of containerization is that it occurs at the OS level, with all of the containers sharing one kernel. While it is not the same with virtualization. VM (Virtual Machine) runs on top of a hypervisor, which is specialized hardware, software or firmware. Via the hypervisor every VM has not only the essential binaries and or libraries assigned but also a virtualized hardware stack that includes storage, network adapters and CPUs. Biggest problem with that is that each VM requires separate virtualized kernels for every application and a heavy extra layer (hypervisor). The additional layer increases the risk of performance issues. A single VM can support way less virtualized containers (around 10).

Two main containerization services are Docker and Kubernetes. The question about the difference of these two often comes down to which one you should use. But this is a bad view because they actually work the best when used in conjuncture. Docker is a container file format for automating the deployment of applications making it as portable and as self-sufficient as possible.

The idea of containerization and environment isolation isn't new and there's actually multiple different service providers for that, but Docker is current the default container format. It features Docker Engine which is a runtime environment that allows building and running containers on any machine. Docker also maintains Docker Hub to store or share already developed template containers. It also can use Azure Container Registry. The problem here becomes clear when the containerized application requires maintaining a large amount of containers which can make coordination of the instances very complex.

Kubernetes is an orchestration software that provides an API endpoint to control how, when and or where the containers will run. This tackles some of the complexities when scaling multiple containers or deploying across multiple servers. Kubernetes orchestrates a cluster of virtual machines and schedules containers to run on the VMs based on their available resources and resource requirements of each container. In Kubernetes the basic operational unit is called a pod. Pods consist of multiple grouped containers allowing management of their lifecycle and the desired state.

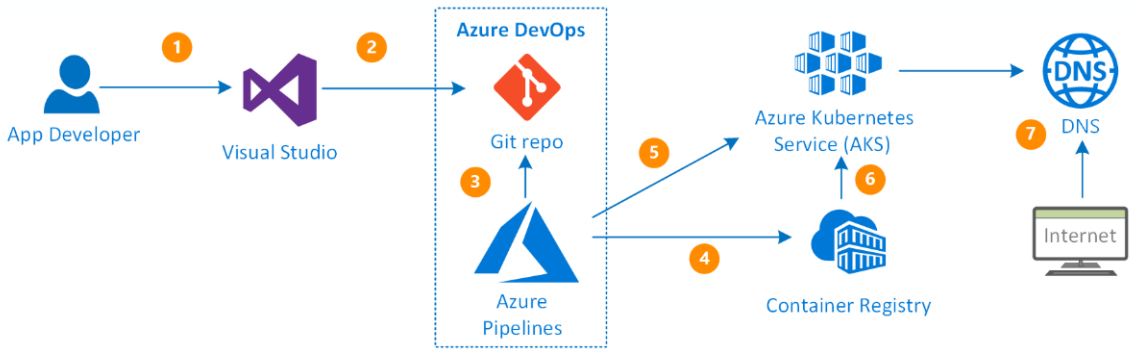


Figure 1. Kubernetes deployment via Azure

In the figure above we have a project maintainer (developer) that make code changes and push it with any version control service (like git). Azure pipelines detect the code changes and push them to update the container in Azure Container Registry, while also keeping AKS (Azure Kubernetes Service) up to date. AKS then synchronizes with the registry and the end user can access the application via internet.

### 2.3 Other services

First service that this project requires is a web scraper. Most of the programming languages support some sort of a web scraping, but Python has a lot of different scraping modules. Some of the more popular ones are requests, beautifulSoup4, lxml, selenium and scrapy. **Requests** is the simplest of all of them. It takes an argument which is an URL, then sends a GET HTTP request to the given URL and returns the source code of the page. Most basic usage looks similar to this -

```
Import requests

url = “http://google.com”
response = requests.get(url)
print(response.text)
```

Code 3. Requests

**BeautifulSoup4** extends on the requests module by adding additional functionality. BS4 transforms a complex HTML document into an equally complex tree of Python objects. Also it allows to filter the raw source code by multiple HTML selectors, for example, by class, ID, name and others.

**Lxml** is quite similar to BS4, but the main difference between these two modules is that lxml also supports scraping XML documents more efficiently and doesn’t require a requests module. Lxml works by using two C libraries, which are libxml2 and libxslt. This makes scraping data extremely fast.

**Selenium** is quite different from other scrapers. Actually, it's not a scraper – it's a crawler. First of all, it works by using a web driver to simulate a browser's functionality. Basically selenium simulates actually opening the page and interacting with it. This allows selenium to automate almost anything in the internet. These automation scripts are called bots and they can cause problems for web pages and that's why developers implement some kind of a captcha barricade to prevent bots from doing anything malicious. Selenium supports almost everything from the above web scraping modules, but is comparably slow.

**Scrapy** is the heaviest module. It is also a web crawler. The main principle is that scrapy enables having multiple nodes called spiders. Each spider can do multiple things and can be easily managed by the main controller. This web crawler supports almost all of the features of the modules mentioned above and is surprisingly fast. Scrapy is an ideal option for huge projects.

The idea for the project is that the web scraper would gather data in a given time period for example, every 12 hours. Let's presume that I will use Django MVC framework as a backend for the project. For the idea mentioned to work, a couple of additional services are required – task queue and cache storage. The default or go-to module for tasks in Python is Celery. The default cache storage that is used in most of the applications is Redis. Celery detects tasks in the code by a special notation that you can see below (@task).

```
@task
some_function():
        # code here
```

Code 4. Celery notation

Celery puts the notated tasks into Redis, which lets backend to continue working on other things. On a separate container Celery runs workers that can pick up tasks. Those workers listen to the Redis container. When a new task arrives, one worker picks it up and executes it while also logging the result back to Celery.

There are two main types of databases to choose from – relational and document (also known as SQL and NoSQL). Relational databases use a set structure that allows them to link information from different tables using indexes. These data pools could then be linked thorough a relationship. While NoSQL databases do not use any kind of relational enforcement. The architect of the database determines what relationships, if any, are necessary for their data, and creates them. Both relational and document databases are better for different things. For example, relational ones would be better suited for an application that requires a lot of complex and intensive queries between different databases, which is required in an e-commerce store. While document databases are typically better for applications that require horizontal scaling and more flexible architecture, such as big data analytics and real-time web applications. Choosing a correct database infrastructure is crucial for a project to succeed. A system architect has to think how the data will be handled, what kind of data will be received and quite a few other things to take into consideration. Since this project won't require big data analytics, nor will it require extensive queries, I will use a PostgreSQL database which is a relational type, works really well with Django backend framework and is quite familiar to me. Another choice could be MongoDB which is a document database, open-source and quite simple to use as well, but since I have more experience with PostgreSQL, I will use a relational database.

To achieve this project four containers will be used, one for each of the services –

- Django
- Celery
- Redis
- PostgreSQL

Since we will not be using hundreds of them, we don't actually need to use Kubernetes containerization service, because Docker supports a service called Docker Compose. It is a tool for defining and running multi-container applications. Compose uses a YAML file to configure all of the application's services. Then by a single command all of the services can be built and ran.

```
version: “3”
services:
    django:
        build: .
        ports:
            - “8000:8000”
        depends_on:
            - db
    db:
        ports:
            - “5432:5432”
```

Code 5. Docker Compose

Above we can see a simple Docker Compose configuration with two services, backend (Django) and a database. Here those ports are defined on which the services should be listening, and a build path is defined as well. Of course additional parameters are required to make this work (like images), but this is how the core concept of Docker Compose YAML file looks like. For the backend we will also have to create a Dockerfile that will define how our service looks like.

Version control sometimes also known as source control is a tool for developers to track and manage changes in the code. Version control systems accelerates software development and deployment. This is a must-have tool for anyone who works with DevOps. Version control keeps track of all of the modifications to the code in a special kind of database. If something goes wrong, developers can turn back the clock and compare earlier versions of the code to identify and help fix the mistake made, while minimizing disruption to all of the team. Two main version control services are GitHub and GitLab. GitHub has higher availability and is more focused on infrastructure performance while GitLab is focused on offering more of a wholesome system with centralized and integrated platform for web developers. For this project GitHub will be used since it’s not actually a web application, but it can be done on the GitLab without any problems as well.

Also another very important thing to mention is that this project is fully done on a Linux machine, because a lot of services have a really great compatibility with Linux kernel and shell is a great command interpreter for all of the development and operations challenges. Shell in Linux operating system takes input from the terminal window in the form of commands, then processes them and shows the output of the command. It's the default interface through which a user works on the programs, commands and scripts in a Unix-based operating systems. Although Docker supports development in Windows, it actually overcomplicates a lot of simple things and sometimes Windows-exclusive issues might occur which is really difficult to debug.

# 3 APPLICATION DEVELOPMENT

The next step of the thesis consists of two main parts – setting up an API endpoint to the world wide web and fetching the data API results via graphical user interface (GUI).

## 3.1 Setting up an API endpoint

Before hosting the application on a world wide web, we need to make sure things work correctly in a local environment. There are some prerequisites before starting to write the backend. The most important thing to install is Python programming language module. This step is required when developing a solution in Windows operating system, but my advice would be to use a Unix-based operating system, because a lot of incompatibilities occur when using Windows for development. Also Python comes pre-installed in an operating system like Ubuntu. The next prerequisite is to install Docker Engine and Docker Compose. Following the Docker documentation, first thing is to update the repositories on the system

```
$ sudo apt-get update

$ sudo apt-get install \
    apt-transport-https \
    ca-certificates \
    curl \
    gnupg \
    lsb-release
```

Figure 2. Update repos

Next step is to add Docker's official GPG key via command

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg
```

Code 6. Add GPG key

Initialize Docker Engine installation

```
sudo apt-get install docker-ce docker-ce-cli containerd.io
```

Code 7. Docker Engine installation

Confirm the successful installation by validating Docker version

```
xakingas@Ubuntu:~$ docker -v
Docker version 20.10.5, build 55c4c88
```

Figure 3. Docker Engine installed

The next step is to setup Docker Compose. The stable version can be downloaded by using this command

```
sudo curl -L "https://github.com/docker/compose/releases/download/1.29.1/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
```

Code 8. Docker Compose stable version download

Apply executable permissions to the binary

```
sudo chmod +x /usr/local/bin/docker-compose
```

Figure 4. Chmod command

Confirm the successful installation by validating Docker-Compose version

```
xakingas@Ubuntu:~$ docker-compose -v
docker-compose version 1.27.1, build 509cfb99
```

Figure 5. Docker Compose version

After dealing with prerequisites, the process of backend server containerization begins. Firstly, the project has to have default Django directories (an empty project). Referencing to the Django documentation, it has to look like this

```
mysite/
    manage.py
    mysite/
        __init__.py
        settings.py
        urls.py
        asgi.py
        wsgi.py
```

Figure 6. Django directories

Top directory (mysite/) is where the project is placed. Then there is the manage.py file which controls some of Django's functionality. The second mysite/ directory is a so-called root folder of a project. Two important files to mention here is settings.py and urls.py. Settings file is responsible for a lot of the server's customization. Urls file handles the routing for the server. Easiest way to create such directory template is installing Django and creating a project in some directory.

```
$ python -m pip install Django
```

Figure 7. Install Django

```
$ django-admin startproject mysite
```

Figure 8. Setup an empty Django project

After settings up the directories, the server is almost ready for containerization. To implement additional modules for the server's container, modules are defined in a separate file called "requirements.txt", which has to be located in the project root directory.

```
1  Django>=3.0,<4.0
2  psycopg2-binary>=2.8
3  requests>=2.25.1
4  beautifulsoup4>=4.9.3
5  celery==4.4.1
6  redis==3.4.1
7  django-celery-beat==2.2.0
```

Figure 9. Requirements.txt

First line refers to Django framework which has to be between version 3 and 4 (current stable version). Line 2 is a binary requirement for PostgreSQL database adapter. Lines 3 & 4 are modules required for web scraping. Line 6 is cache storage service Redis, lines 5 and 7 are modules for task queuing (Celery and its worker Celery Beat).

After finishing with module requirements file, a Docker containerization file needs to be created. Referencing Docker documentation, the file needs to be called "Dockerfile". This file is like the instructions on how the Docker container will look like.

```
1  FROM python:3
2  ARG WORK_DIR
3  ENV VIRTUAL_ENV=$WORK_DIR/.venv
4  COPY manage.py $WORK_DIR/
5  ENV PYTHONUNBUFFERED=1
6  WORKDIR $WORK_DIR/
7  COPY .venv/ $VIRTUAL_ENV
8  COPY requirements.txt $WORK_DIR/
9  RUN pip install -r requirements.txt
10 COPY . $WORK_DIR/
11 ENV PATH="$VIRTUAL_ENV/bin:$PATH"
```

Figure 10. Dockerfile

First of all, the core language (Python) is declared. After that an argument WORK_DIR is created, in line 6 this variable is set to the root directory of a container. Line 3 and 7 deals with locally installed modules by adding them to a virtual environment in the container.  Line 4 is the reason why setting up an empty project is required beforehand. Command COPY takes the manage.py file which is created locally and puts it into the container. Line 5 ensures that any Python output would be sent straight to the server's terminal. Line 8 puts the file requirements.txt into the container and line 9 executes installation of the file's contents. Line 10 puts everything else (core project files) into the container. Line 11 is not really required, but it helps to manage the virtual environment. Django framework's Dockerfile is ready. Since this project consists of multiple containers and not just a single one, setup of docker-compose.yml file is required.

```yaml
version: "3"

services:
  django:
    image: bakalauras_backend
    build:
      context: .
      dockerfile: ./Dockerfile
    env_file: .env
    command: python manage.py runserver 0.0.0.0:8000
    volumes:
      - .:/app
    networks:
      - xakingas-backend-tier
    ports:
      - "8000:8000"
    restart: unless-stopped
    depends_on:
      - db
      - redis

  db:
    image: library/postgres:11.1-alpine
    ports:
      - 5432:5432
    restart: unless-stopped
    networks:
      - xakingas-backend-tier
    volumes:
      - ./.volumes/pg/bachelors:/var/lib/postgresql/data
    environment:
      - POSTGRES_DB=bachelors
      - POSTGRES_USER=xakingas
      - POSTGRES_PASSWORD=xakingas

  redis:
    image: library/redis:5.0-alpine
    restart: unless-stopped
    networks:
      - xakingas-backend-tier
    volumes:
      - db-redis:/data
    ports:
      - 6379:6379

  celery:
    image: bakalauras_backend
    command: python3 -m celery -A xakingas worker --beat --scheduler django --app=xakingas.celeryconf:app --loglevel=debug
    volumes:
      - .:/app
    env_file: .env
    networks:
      - xakingas-backend-tier
    depends_on:
      - django

volumes:
  db-bach:
    driver: local
  db-redis:
    driver: local

networks:
  xakingas-backend-tier:
    driver: bridge
```

Figure 11. Docker Compose file

Starting with the compose file in the very top version of the file is declared, in this case version 3. At the very bottom, the volumes and networks are declared. Volumes is basically a directory where the data of storage services will be kept (one of database and one for cache). Both of the volumes are local to the project root directory. Networks basically stitches up the services on a bridged driver so the containers could detect each other. For that case one network is defined (”xakingas-backend-tier”). Most of the declarations is used for images, so let’s go through each of them one by one. The first service in the file is called ”django”. It takes an image parameter which can be fetched from Docker’s Image Registry called Docker Hub. Next there is a build parameter, with context argument. A .

(”dot”) notation is left there, which in Unix-based systems means ”current directory”.
Another argument is called a ”dockerfile”, which basically connects the previously created Dockefile with the Django service. Next up there’s ”.env” parameter. Every project has some secret keys or any other sensitive information, so for each project a separate .env file should be created. This parameter in the docker-compose.yml configuration declares path to the .env file.  An example of how such a file could look

```
1  CACHE_URL=redis://redis:6379/0
2  CELERY_BROKER_URL=redis://redis:6379/1
3  SECRET_KEY=ck4qa+zz9645$jld@!onhm!0v7p8)#4_jlctr=-6_&rj%5yj5f
4  DEBUG=True
5  ALLOWED_HOSTS=*
```

Figure 12. .env file

It is IMPORTANT that this file does not get into public repositories, because otherwise multiple of different exploits can be used against the server. This can be easily prevented by just creating ”.gitignore” file and appending the file with text ”.env”.

After that a command is given to the Django service. This command will run only after containerization of server’s Dockerfile. The default command for Django server to start is ”python manage.py runserver 0.0.0.0:8000”. This is not fully secure by current market standards, because some sort of web server gateway interface needs to be added, two main choices would be a web server or a gunicorn service (both would need additional containers), but since this project is created for educational purposes and for simplicity’s case – I will not set them up.

Then the volumes and networks are linked with the service. ”restart: unless-stopped” parameter is also given to make the server restart every time something unexpected happens. Service dependencies (”depends_on”) for the Django service is defined (database and cache storage). And lastly ports are defined. First port is called a ”HOST_PORT” which can be accessed externally and the

second port is ”CONTAINER_PORT”, which is basically used by Docker internally. In a production environment, ports should not be the same, because of the security concerns.

Next up there’s three services called ”db”, ”celery” and ”redis”. Each of these services follow almost the same exact template as a previously example, but with some adjustments. Docker Hub supports multiple of free pre-set images for some of the services. Database and Redis images are pulled from Docker Image Registry. Both of these services are run on an ”Alpine” linux distribution which is probably the most lightweight Unix-based distro therefore increasing the performance of the project. Also different ports for each of the service is declared. The last thing I want to mention is the command that is passed to the ”celery” service. The syntax for that can be found in the celery documentation, but basically the worker runs on a specific schedule for an application ”xakingas” and the configuration for celery is included in project’s root directory by a file called ”celeryconf.py”. Other services do not require additional setup, so let’s have a look at celery configuration.

```
1   import os
2
3   from celery import Celery
4
5   os.environ.setdefault("DJANGO_SETTINGS_MODULE", "xakingas.settings")
6
7   app = Celery("xakingas")
8
9   CELERY_TIMEZONE = "UTC"
10
11  app.config_from_object("django.conf:settings", namespace="CELERY")
12  app.autodiscover_tasks()
```

Figure 13. Celery config

Please ignore the red underlined code, because I don’t have celery module installed locally, but it will work in the container swarm. Line 5 sets the settings to the django project’s settings. Line 7 initializes celery instance for application ”xakingas”. Line 9 sets the timezone for celery. Line 11 creates configurations

from the settings (which is defined in line 5) and line 12 declares that celery itself should look for available tasks that would be notated with previously mentioned celery notation (”@task”).

At this point all of the core backend services are ready for deployment, but this project is still empty, so let’s undo that.

Information that will be scraped from the internet can be about anything, but since the world currently is going through a pandemic I want to address that. Webpage scraped will be ”https://worldmeters.info/coronavirus/”. It has the statistics for total cases, total deaths and total recovered for the whole world and for specific countries like Lithuania and Finland.

First of all, models that define how data will shaped in PostgreSQL database has to be created.

```
1   from django.db import models
2
3
4   class WorldData(models.Model):
5       cases = models.CharField(max_length=64,blank=True, null=True)
6       deaths = models.CharField(max_length=64,blank=True, null=True)
7       recovered = models.CharField(max_length=64,blank=True, null=True)
8       fetch_time = models.DateTimeField(auto_now_add=True)
9
10
11  class LithuaniaData(models.Model):
12      cases = models.CharField(max_length=64, blank=True, null=True)
13      deaths = models.CharField(max_length=64, blank=True, null=True)
14      recovered = models.CharField(max_length=64, blank=True, null=True)
15      fetch_time = models.DateTimeField(auto_now_add=True)
16
17
18  class FinlandData(models.Model):
19      cases = models.CharField(max_length=64, blank=True, null=True)
20      deaths = models.CharField(max_length=64, blank=True, null=True)
21      recovered = models.CharField(max_length=64, blank=True, null=True)
22      fetch_time = models.DateTimeField(auto_now_add=True)
```

Figure 14. Django models

This is equivalent to creating three database tables (”worlddata”, ”lithuaniadata” and ”finlanddata”) and each of the tables has four columns that are pretty self-explanatory (cases, deaths, recovered and fetch_time). CharFields with max length attribute is the same as ”VARCHAR (64)” in SQL. Blank and null values are allowed just for the debug reasons (optional). ”DateTimeField” row value is updated when a new record is created. To confirm the model changes to the database it has to be migrated first. Migration is basically Django creating a separate file which will be passed to the database interpreter and raw SQL commands will be pushed to the PostgreSQL service.

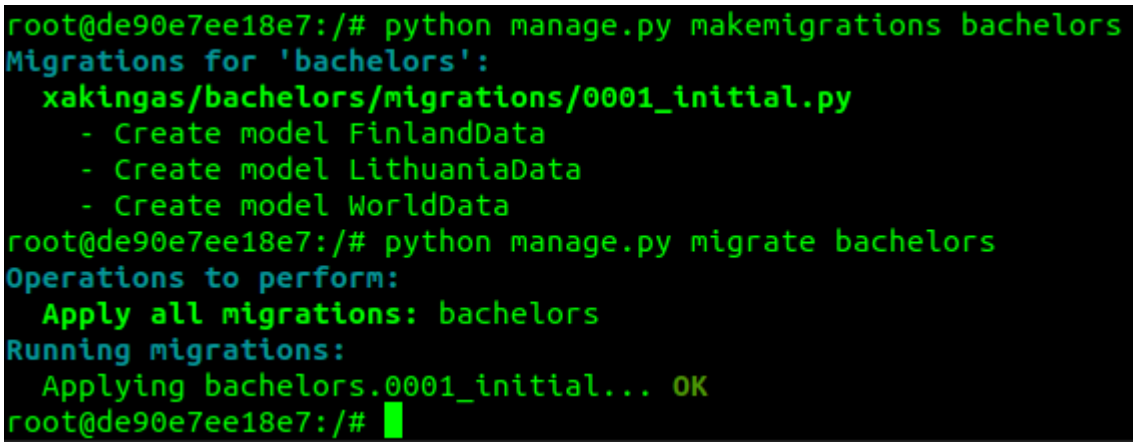


Figure 15. Migration

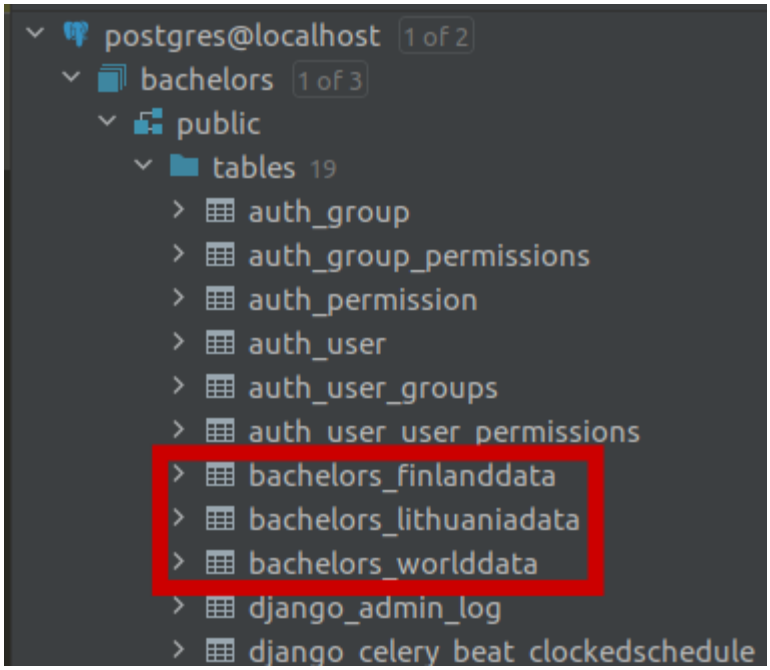


Figure 16. Newly created database tables

The next step is to actually fill these databases with some information. For that a web scraper will be built using Python Requests and BeautifulSoup4 modules. For the scraper to work, the first thing is getting to know how the targeted website's source code looks like. This can be achieved in two main ways. The first one is fetching the source code directly to the terminal via requests, showcasing all kinds of code that website consists of. But a more efficient way would be to use browser's inspect tool.


Figure 17. Inspect tool

In the example above I've inspected the element which includes the numerical value which I'm trying to scrape. Luckily for me, the counter is embedded to a "div" element with unique ID – "maincounter-number".

```
1   import requests
2   from bs4 import BeautifulSoup
3
4   def scrape_data(location):
5       raw_data = requests.get(location)
6       souped = BeautifulSoup(raw_data.content, 'html.parser')
7       result = souped.find_all("div", class_="maincounter-number")
8       cases = result[0].get_text(strip=True)
9       deaths = result[1].get_text(strip=True)
10      recovered = result[2].get_text(strip=True)
11      return cases, deaths, recovered
```

Figure 18. Scraper logic

A simple scraper function is created. This function takes location as an argument and fetches the source code for the location provided. In line 6 the raw_data is converted into a BeautifulSoup object. Then the result is filtered by a BS4

function ”find_all”. Also, the declaration of required element and a unique ID is passed. Finally, 3 different variables with the according information is returned.

```python
from xakingas.bachelors.models import WorldData, LithuaniaData, FinlandData
from xakingas.bachelors.utils import scrape_data
from celery import task

@task
def fetch_data():
    world = "https://www.worldometers.info/coronavirus/"
    world_data = scrape_data(world)
    WorldData.objects.create(
        cases=world_data[0],
        deaths=world_data[1],
        recovered=world_data[2]
    )

    lithuania = "https://www.worldometers.info/coronavirus/country/lithuania/"
    lithuania_data = scrape_data(lithuania)
    LithuaniaData.objects.create(
        cases=lithuania_data[0],
        deaths=lithuania_data[1],
        recovered=lithuania_data[2]
    )

    finland = "https://www.worldometers.info/coronavirus/country/finland/"
    finland_data = scrape_data(finland)
    FinlandData.objects.create(
        cases=finland_data[0],
        deaths=finland_data[1],
        recovered=finland_data[2]
    )
```

Figure 19. Task logic

In the figure above is the main logic for a task that is passed to celery task queuing service. Notice that ”fetch_data” function has a @task annotation. Basically it’s a task that runs on a set interval of time. It calls a previously mentioned ”scrape_data” function and creates new rows in the according database tables.

Another important thing that needs configuring is Django settings.py file.

```
89  DATABASES = {
90      'default': {
91          'ENGINE': 'django.db.backends.postgresql',
92          'NAME': 'bachelors',
93          'USER': 'xakingas',
94          'PASSWORD': 'xakingas',
95          'HOST': 'db',
96          'PORT': 5432,
97      }
98  }
```

Figure 20. Connect database to Django

```
139 CELERY_BROKER_URL = (os.environ.get("CELERY_BROKER_URL"))
140 CELERY_RESULT_BACKEND = "redis://redis:6379"
141 CELERY_IMPORTS = ("xakingas.tasks",)
142
143 CELERY_BEAT_SCHEDULE = {
144     "fetch_data": {
145         "task": "xakingas.tasks.fetch_data",
146         "schedule": crontab(minute='0', hour="*/12"),
147     },
148 }
```

Figure 21. Connect Celery service to Django

One thing to mention here is line 146. Using crontab schedule expressions ”minute=’0’, hour=’*/12’” is equivalent to every 12 hours. One of the last things left is to make the server act as API endpoint, which means returning HTTP (Hyper Text Transfer Protocol) responses. For that I will add a file named views.py (an equivalent of a controller in other MVCs).

```python
import json

from django.core.serializers import serialize
from django.http import HttpResponse

from xakingas.bachelors.models import WorldData, LithuaniaData, FinlandData


def send_world(request):
    obj = WorldData.objects.latest('fetch_time')
    serialized = serialize("json", [obj], fields=('cases', 'deaths', 'recovered'))
    data = json.loads(serialized)
    for d in data:
        data = d['fields']
    finalized = json.dumps(data)

    return HttpResponse(finalized, content_type='application/json')
```

Figure the default command for Django server to start. Fragment of views.py file

This file consists of three main functions. ”send_world”, ”send_lithuania” and ”send_finland”. I will only detail what’s happening here in ”send_world” function, because all of them are quite identical, only the data returned changes. In line 10 the latest scraped data row is queried. After that the ”QuerySet” object is serialized into the JSON string and some data manipulation is used to remove unnecessary information (like a primary key of a row). Finally in line 17 HttpResponse is returned with finalized data.

To test if everything works as expected one more thing is required to do – creating the endpoints themselves in the urls.py file.

```python
from django.conf.urls import url

from . import views


urlpatterns = [
    url(r"^api/world/?$", views.send_world, name="world"),
    url(r"^api/lithuania/?$", views.send_lithuania, name="lithuania"),
    url(r"^api/finland/?$", views.send_finland, name="finland"),
]
```

Figure 22. Urls.py file

For the backend there is three url patterns, one for each of the views.py file functions. ”api/world/” request will return ”send_world” function and so on. Let’s test if everything works as expected.

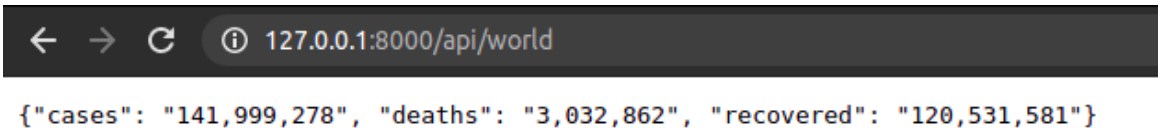


Figure 23. Successful response

The application seems to be fully working locally, it’s time to host it on the world wide web. For that I will use AWS (Amazon Web Services) because it offers a great free tier with a quite a powerful host machine. The process of setting up a default server on AWS is actually quite simple because the dashboard is really user friendly and there is quite a lot of beginner friendly tutorials on YouTube. One important thing is that specific inbound rules need to be set to be able to reach the server.


Figure 24. Inbound rules

Two rules are included – Custom TCP and SSH. Custom TCP rule is used to allow access on the 8000 port of the machine, where the API is exposed. There are two separate custom TCP rules because of Ipv4 and Ipv6. The SSH source is my home IP address, so I could actually get into the remote host from the terminal. But before that let’s push the code to a private remote depository so it could be pulled to the AWS server host.

```
git init

git add .

git commit –m ‘first commit’

git push origin master
```

Code 9. Git code

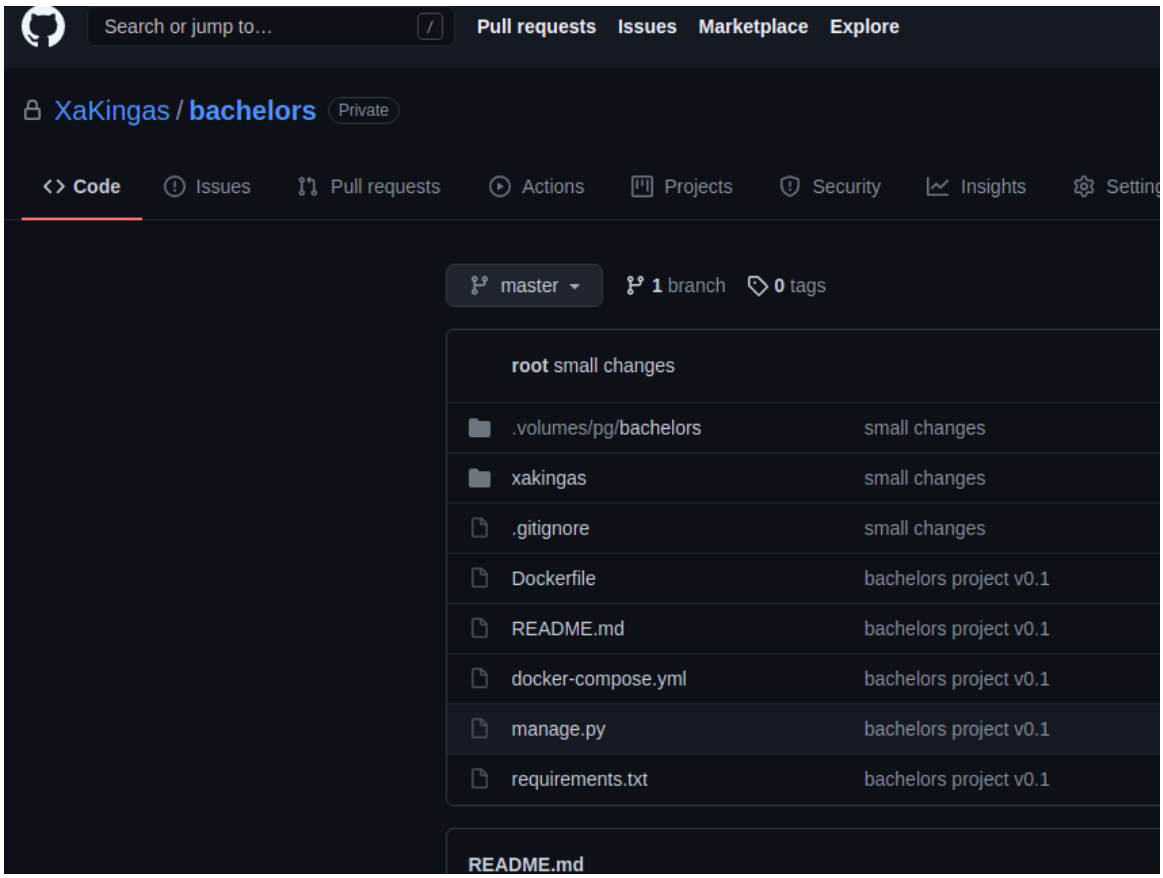


Figure 25. Github repository

Now everything is ready to be hosted on the internet. SSH remotely to the AWS machine.

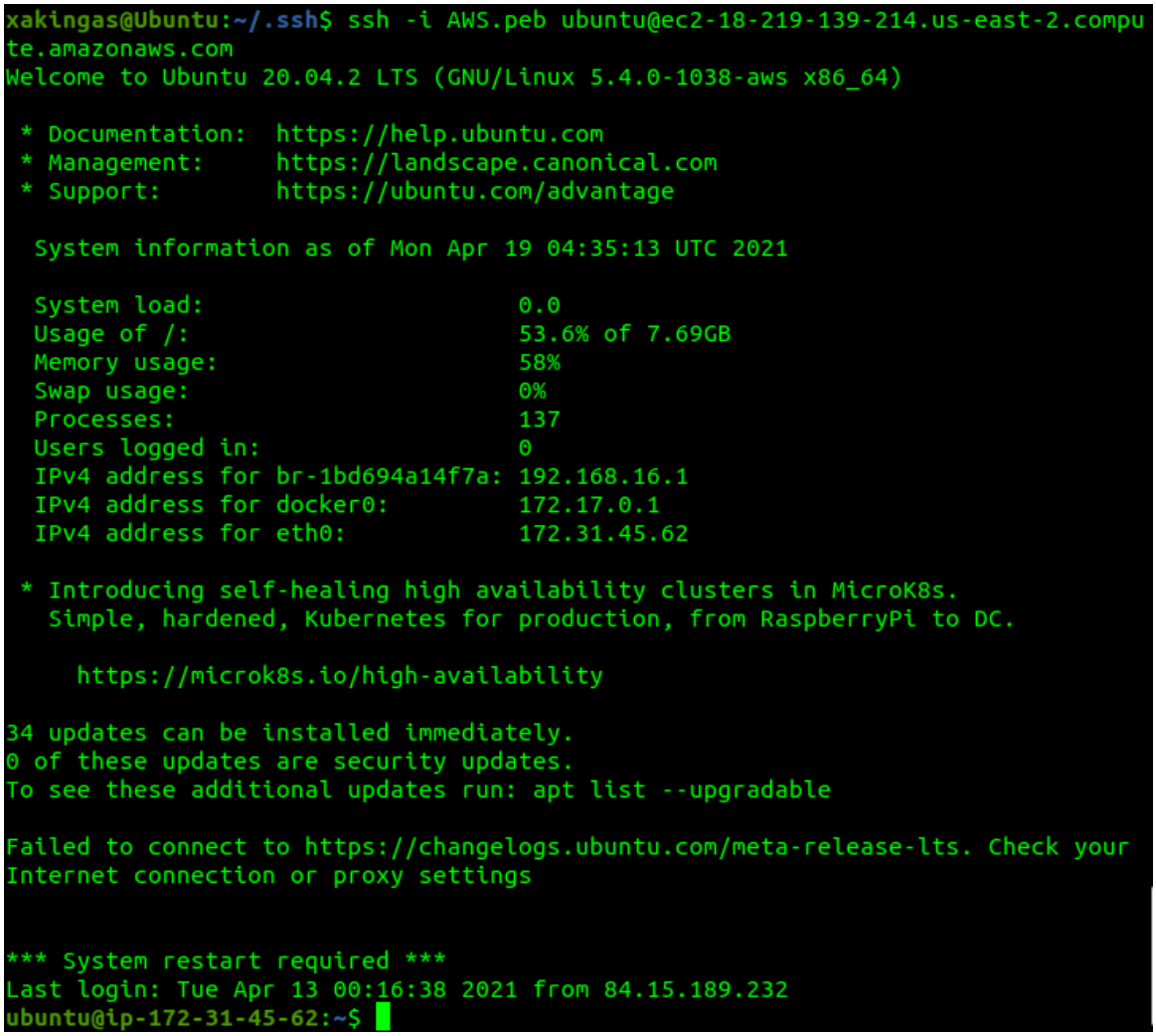


Figure 26. SSH connection

Lastly these commands are run to build the project on an AWS host.

```
git clone < HTTPS url of the repo >

docker-compose build

docker-compose up
```

Code 10. Commands on a AWS host

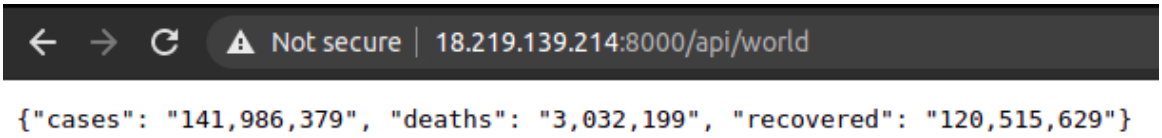


Figure 27. Success

The API endpoint is successfully hosted and deployed to the world wide web.

### 3.2 Fetching data from an API endpoint

To fetch the data from the API I will use requests module. To create a GUI, I will use “Tkinter” module which is really simple to use if you have any experience with Python. To install “Tkinter”, following command must be executed

```
sudo apt-get install python3-tk
```

Code 11. Install Tkinter module

```
1   import tkinter as tk
2   from tkinter import *
3   from PIL import Image, ImageTk
4   import requests
5
6
7   BASE_URL="http://18.219.139.214:8000/api/"
8
9   world_link = BASE_URL + "world"
10  lithu_link = BASE_URL + "lithuania"
11  finla_link = BASE_URL + "finland"
12
```

Figure 28. Base links

In the first part of the code links are created to the API endpoint.

```
13      def dialog(cases, deaths, recovered):
14          dialog = Tk()
15          dialog.title("Information")
16          T = tk.Text(dialog, height=3, width=25)
17          T.pack()
18          T.insert(tk.END, "Cases: ")
19          T.insert(tk.END, cases)
20          T.insert(tk.END, "\n")
21          T.insert(tk.END, "Deaths: ")
22          T.insert(tk.END, deaths)
23          T.insert(tk.END, "\n")
24          T.insert(tk.END, "Recovered: ")
25          T.insert(tk.END, recovered)
26          T.insert(tk.END, "\n")
27
28          tk.mainloop()
```

Figure 29. Dialog

Next a simple dialog box is created which will show the fetched results. This function accepts three arguments – cases, deaths and recovered.

```
31  def clicked_world():
32      world_data = requests.get(world_link)
33      world_dict = world_data.json()
34      dialog(cases=world_dict['cases'], deaths=world_dict['deaths'], recovered=world_dict['recovered'])
35
36
37  def clicked_lithu():
38      lithu_data = requests.get(lithu_link)
39      lithu_dict = lithu_data.json()
40      dialog(cases=lithu_dict['cases'], deaths=lithu_dict['deaths'], recovered=lithu_dict['recovered'])
41
42  def clicked_finla():
43      finla_data = requests.get(finla_link)
44      finla_dict = finla_data.json()
45      dialog(cases=finla_dict['cases'], deaths=finla_dict['deaths'], recovered=finla_dict['recovered'])
```

Figure 30. Functions

Functions are created that will be called on according to button clicks. After the click requests module will fetch the proper url, jsonifies the returned information and pass it to the dialog function which was covered above.

```
main_window = Tk()
main_window.title("Client software")
main_window.geometry('600x314')

background_image = ImageTk.PhotoImage(Image.open("bg.jpg").resize((390,310), Image.ANTIALIAS))
world_image = ImageTk.PhotoImage(Image.open("world.png").resize((200,100), Image.ANTIALIAS))
lithu_image = ImageTk.PhotoImage(Image.open("lt.png").resize((200,100), Image.ANTIALIAS))
finla_image = ImageTk.PhotoImage(Image.open("fi.jpg").resize((200,100), Image.ANTIALIAS))

bg_label = tk.Label(main_window, image=background_image)
bg_label.grid(rowspan=4, column=1)


world_btn = tk.Button(main_window, text="Show World Data", command=clicked_world, image=world_image,
 font=('Calibri', 10, 'bold'), height = 90, width = 180, compound=tk.CENTER)
lithu_btn = tk.Button(main_window, text="Show Lithuania Data", command=clicked_lithu, image=lithu_image,
 font=('Calibri', 10, 'bold'), height = 90, width = 180, compound=tk.CENTER)
finla_btn = tk.Button(main_window, text="Show Finland Data", command=clicked_finla, image=finla_image,
 font=('Calibri', 10, 'bold'), height = 90, width = 180, compound=tk.CENTER)

world_btn.grid(column=2, row=1)
lithu_btn.grid(column=2, row=2)
finla_btn.grid(column=2, row=3)

main_window.mainloop()
```

Figure 31. GUI

In the example above is the creation of the GUI itself. First of all, some images are opened and resized to fit with the GUI. Afterwards buttons are created, gridded and images set. Finally GUI is ran with ”mainloop” function.


Figure 32. Final GUI design

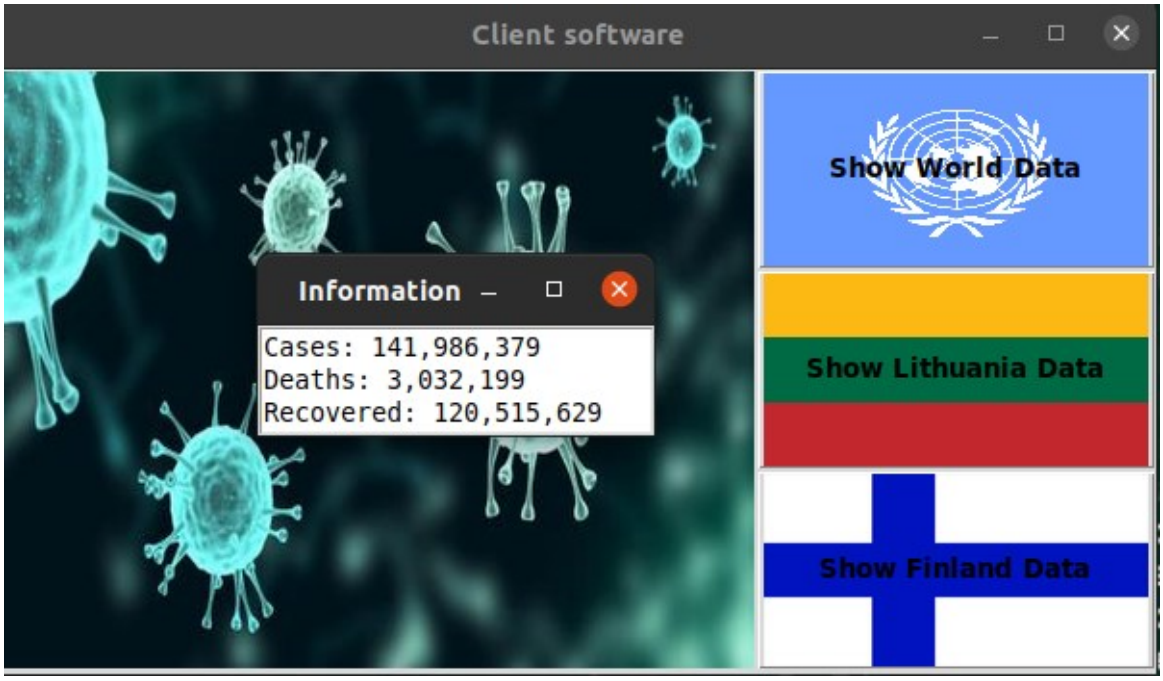


Figure 33. GUI with world data dialog box open

The full stack application has been successfully created and works as intended. The server backend can be accessed from anywhere on the Internet and the client software can run on any machine that supports Python language.

## 4 CONCLUSION

The aim of this study was to develop a full stack application with such infrastructure that would be as similar as possible to the current market standards. Multiple different options have been analyzed showcasing most of the pros and cons of each major framework. The technology stack can vary quite a lot between different goals, so similar analysis has to be done before starting developing any bigger projects to prevent possible compatibility issues.

In conclusion, MVC frameworks are the current way to go with any web and software application development. Although they are more compatible with web applications, I had no problems creating this software by using MVC as the backend for the project. Also containerization is now a required skill to have for any aspiring developer, because the applications created every year need

multiple different services which would be semi-impossible to manage without tools like Docker or Kubernetes.

**REFERENCES**

AWS documentation. Updated at 2021. https://docs.aws.amazon.com [Accessed 17 April 2021]

Burwood. 2019. Containerization vs. Virtualization: What's the Difference? https://www.burwood.com/blog-archive/containerization-vs-virtualization [Accessed 17 April 2021]

Celery documentation. Updated at 2021. https://docs.celeryproject.org/en/stable/ [Accessed 17 April 2021]

Codeinwp. 2021. Angular vs React vs Vue: Which Framework to Choose in 2021 https://www.codeinwp.com/blog/angular-vs-vue-vs-react/ [Accessed 17 April 2021]

Django documentation. Updated at 2021. https://docs.djangoproject.com/en/3.2/ [Accessed 17 April 2021]

Docker documentation. Updated at 2021. https://docs.docker.com/ [Accessed 17 April 2021]

FreeCodeCamp. 2017. What is MVC, and how is it like a sandwich shop? https://www.freecodecamp.org/news/simplified-explanation-to-mvc-5d307796df30/ [Accessed 17 April 2021]

Hazelcast. 2020. Caching. https://hazelcast.com/use-cases/caching/ [Accessed 17 April 2021]

Python Tkinter documentation. Updated at 2021. https://docs.python.org/3/library/tk.html [Accessed 17 April 2021]

LIST OF FIGURES

LIST OF CODES



LIST OF TABLES