Jokster – Media Sharing App

Project Engineering

Year 4

Vladimir Mauer

Bachelor of Engineering (Honours) in Software and Electronic Engineering

Galway-Mayo Institute of Technology

2020/2021

Declaration

This project is presented in partial fulfilment of the requirements for the degree of Bachelor of Engineering (Honours) in Software and Electronic Engineering at Galway-Mayo Institute of Technology.

This project is my own work, except where otherwise accredited. Where the work of others has been used or incorporated during this project, this is acknowledged and referenced.

_____Vladimir Mauer_____

Table of Contents

1 Summary

This project is the base for a web application that can share media in a network of different users. The core aspect of the application is a ranking system based on ratings of funny posts from the community. Posts at the lower rankings get deleted in a cycle.

Jokster is a web application that aims to provide a learning milestone for the development of full stack applications and modern systems. This includes different technologies and tools that interact with each other, from development to deployment in terms of systems thinking. The goal is to have a deeper understanding of how the cycle is defined in industry standards.

The scope of this project entails a base foundation of creating a robust backend that handles all necessary requests, authentication, and database manipulation. It should provide a well-functioning backend API to allow any frontend framework to consume the API endpoints and bring the users a fast application with a good user experience.

The project uses Django for the backend with the Django Rest Framework (DRF) to build the Rest API. PostgreSQL serves as the database. Celery is used to for asynchronous task queues and Redis serves Celery as the message broker and results backend. These celery workers are monitored with Flower. Nginx is used as the web server to serve static and media files. The whole backend is packaged in Docker containers with different shell scripts specified in the compose file. For the frontend ReactJS is used.

2 Poster

GMIT

Department of
Electronic and
Electrical Engineering

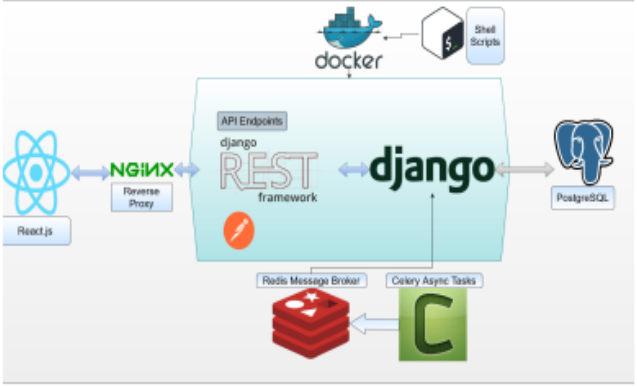
JOKSTER

MEDIA SHARING APP

BY VLADIMIR MAUER

SUMMARY

Jokster is a full-stack media sharing web application with an emphasis on the back-end API. The python framework Django is used to handle requests , models, the PostgreSQL database and authentication. The API endpoints are exposed through Django Rest Framework and consumed by the ReactJS front-end.
The chosen technology stack enables scalable and rapid development for various platforms whether it is a browser or native client.



BACKEND API

The back-end consists of multiple technologies that are packaged and containerized in Docker containers. Celery workers are used for asynchronous tasks queues and Redis acts as the message broker to handle the queues.
Nginx serves as a load balancer and reverse proxy to serve static and media files and also provides an extra layer of security.

CONCLUSION

Working on a full-stack system provided me with the opportunity to apply technologies I learned from different modules and learned new concepts from the base knowledge I gained. This gave me insights into how modern and complex applications are designed and build with the industry standard.

Further development on the project will extend my tool repertoire in the coming years.

3 Introduction

As an active user of various social media platforms my inspiration came from the fasciation for the simplicity on the surface and complexity on the fundamental level of those applications. The goal is to build a functioning prototype that can be further developed and maintained to provide a small community within my social circle. This project validates the acquired skills in the last four years and allows for further improvements in new areas.

The status of the project is a fully functional backend and API endpoints with the groundwork done to implement new models where needed. For the frontend the project contains a skeleton that needs further improvement.

This report contains the sections in the development of the backend.

4 Background

Since my interests lean more towards backend development, I had to make a choice for the right technology stack.

One of my strengths is development in Python, from my work placement and previous projects I have worked on. So, I decided to use Django since it enables rapid development with an easier way of managing models, views and enables the application to scale. PostgreSQL for the database is more commonly used with Django and with ReactJS as one of the main frontend frameworks I have decided to use the Django/React stack.

5 Project Architecture

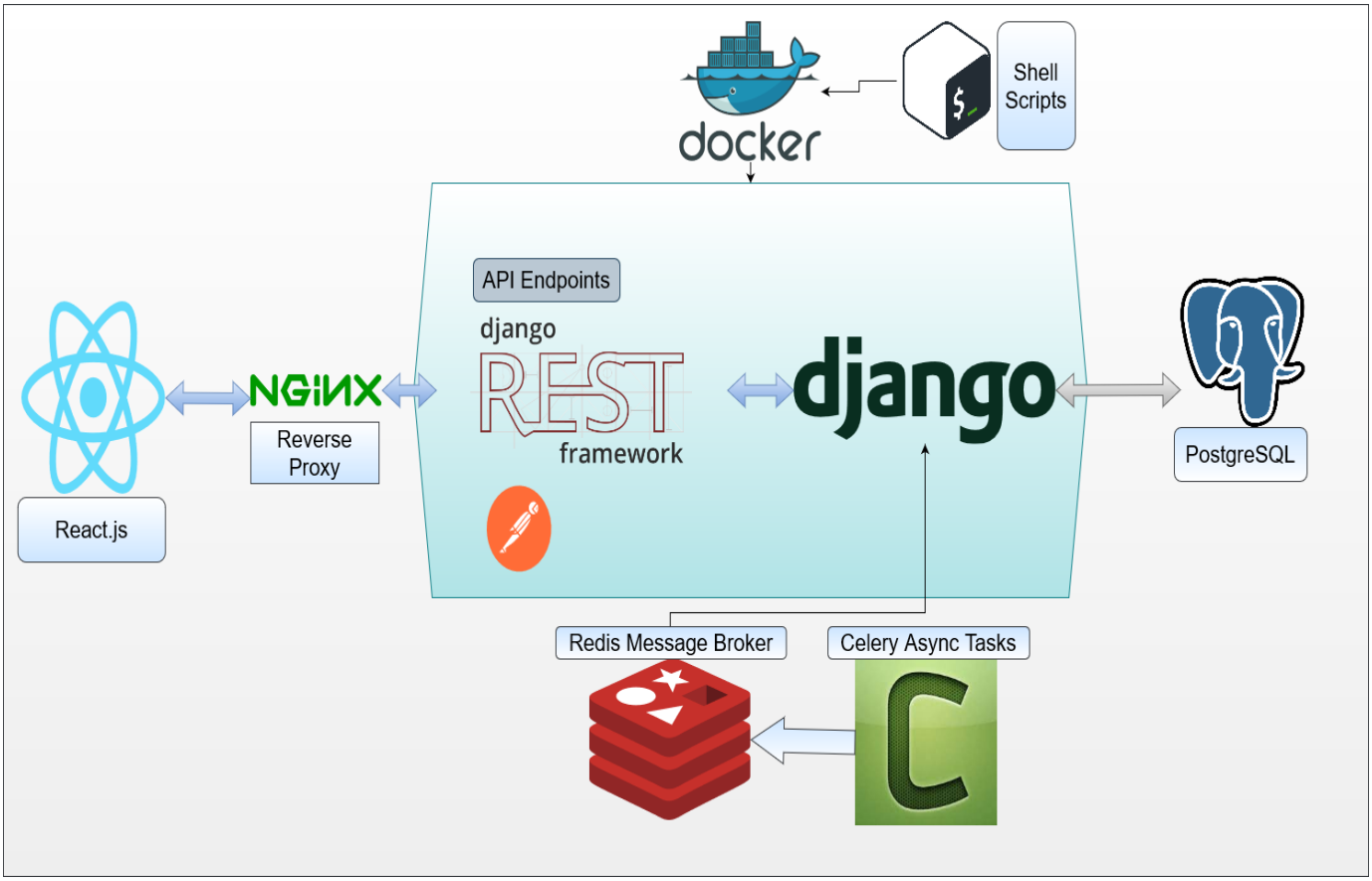


Figure 5-1 Architecture Diagram

6 Docker

Docker is a stack tool that enables applications to run in isolated environment called containers (*Docker Overview | Docker Documentation*, n.d.). While working in a team the containers ensure that every developer gets the same container when they want to change something in the application. The separation makes changes more secure since every piece of a system is separated in a container and doesn't get access to other pieces is the system. This makes deployment easier since you don't need to worry about current configurations in the host.

For multiple containers on one host the docker compose is required. The project contains several Dockerfiles with the necessary commands and requirements instructions to build a docker image.

The configurations for the containers are defined in the docker-compose.yaml (local.yaml for local development) file which then can be run with a single command.

```yaml
services:
  api:
    build:
      context: .
      dockerfile: ./docker/local/django/Dockerfile
    command: /start
    container_name: django-api
    volumes:
      - .:/app
      - static_volume:/app/staticfiles
      - media_volume:/app/mediafiles
    expose:
      - "8000"
    env_file:
      - ./.envs/.local/.django
      - ./.envs/.local/.postgres
    depends_on:
      - postgres
      - mailhog
      - redis
    networks:
      - backend_api
```

- This configuration file defines the file that needs to be sent to Dockers daemon with the build context.

- The volumes define files and directories that can be shared between containers.
- The command is specified in a shell script /start which runs Django commands like migrations and starting the server
- Network builds a single network for the containers in this case it's called backend_api

The Dockerfile for this particular service runs a script /entrypoints which exports the environment variable for the database url and makes sure the postgres service is ready and connected.

```
postgres_ready() {
python << END
import sys
import psycopg2
try:
    psycopg2.connect(
        dbname="${POSTGRES_DB}",
        user="${POSTGRES_USER}",
        password="${POSTGRES_PASSWORD}",
        host="${POSTGRES_HOST}",
        port="${POSTGRES_PORT}",
    )
except psycopg2.OperationalError:
    sys.exit(-1)
sys.exit(0)
END
}

until postgres_ready; do
>&2 echo "Waiting for connection to postgres...."
sleep 1
done
>&2 echo "PosgreSQL ready!!...."
```

Other services are defined in the YAML file as well (nginx, celery etc.).

When it comes to debugging going through logs was helpful but sometimes, they are noisy and it's not easy to spot the issue.

7 Signals/Receivers

Django Signals are like Signals/Slots in QT/C++ which enable an observer pattern functionality.

In the project the signals.py file defines a receiver function which links a ‘user’ to a ‘profile’.

```
@receiver(post_save, sender=AUTH_USER_MODEL)
def create_user_profile(sender,instance, created, **kwargs):
    if created:
        Profile.objects.create(user=instance)
```

This function will automatically run when a user gets created.

8 Celery with Flower

To improve the efficiency and increase the speed of the application Django can be used with celery for asynchronous task queues. These processes can run in the background and can be used to run certain processes periodically as well. This is the equivalent to multi-threading.

To monitor the processes and the progress of tasks from celery flower is a tool that can be used in real time. Both are integrated in a container with their corresponding Dockerfiles and /start shell scripts which run the commands to start a celery worker and restart the workers locally when files get changed.

9 Nginx

To serve static and media files directly, the Nginx web server provides a faster way and a more direct way by cutting out Django from handling static assets. The server is containerized with the volumes set to the staticfiles and mediafiles directories. Nginx is a reverse proxy which sits in front of web servers and forwards client’s requests to the server. This enables further stability and security. (*What Is a Reverse Proxy? | Proxy Servers Explained | Cloudflare*, n.d.)

The nginx configuration also defines the path /redoc on a different port which enables the Redoc tool that autogenerated API docs and JSON objects to use for specific endpoints.

These endpoints were successfully tested with Postman.

10 Serializers/Views

To handle more complex data like querysets or model instances, serializers convert those datatypes to native python types which can then can be rendered to JSON(*Serializers - Django REST Framework*, n.d.).

```
class ProfileSerializer(serializers.ModelSerializer):
    username = serializers.CharField(source='user.username')
    first_name = serializers.CharField(source='user.first_name')
    last_name = serializers.CharField(source='user.last_name')
    email = serializers.EmailField(source='user.email')
    full_name = serializers.SerializerMethodField(read_only=True)
    profile_photo = serializers.SerializerMethodField()
    country = CountryField(name_only=True)
    following = serializers.SerializerMethodField()
```

This class serializes the different database fields from the user model and generates the fields.

This is followed by an inner meta class that validates the fields.

The views handle different requests and response.

```
class ProfileListAPIView(generics.ListAPIView):
    serializer_class = ProfileSerializer
    permission_classes = [permissions.IsAuthenticated]
    queryset = Profile.objects.all()
    renderer_class = (ProfilesJSONRenderer,)
    pagination_class = ProfilePagination
```

This view handles the request to retrieve all Profiles available.

11 Conclusion

The project is a fully functioning backend API handling most basic requests like follow, reactions etc., handling the database and the authentication managed with multiple containers and

different ports. It incorporates most the functionality and configuration to be deployed and to handle different requests from the frontend. It can be used as a boilerplate code for further models and views.

12 References

Docker overview | Docker Documentation. (n.d.). Retrieved April 21, 2022, from https://docs.docker.com/get-started/overview/

What is a reverse proxy? | Proxy servers explained | Cloudflare. (n.d.). Retrieved April 21, 2022, from https://www.cloudflare.com/learning/cdn/glossary/reverse-proxy/

Serializers - Django REST framework. (n.d.). Retrieved April 21, 2022, from https://www.django-rest-framework.org/api-guide/serializers/