LAMK
Lahden ammattikorkeakoulu
Lahti University of Applied Sciences

DESIGNING SCALABLE MICROSERVICES

Case: AWS with Python

LAHTI UNIVERSITY OF APPLIED SCIENCES
Bachelor of Engineering, Information and Communications Technology
Bachelor’s Thesis
Spring 2019

Abstract

Author(s) Uotila, Timsa	Type of publication Bachelor’s thesis	Published Spring 2019
	Number of pages 40	
Title of publication **Designing Scalable Microservices** Case: AWS with Python		
Name of Degree Bachelor of Engineering, Information and Communications Technology		

Abstract

The object of the thesis was to design and develop a prototype of a backend application used to serve data for a feed-based social media application. The requirements for the application design were to incorporate authentication, a process for storing photos, effective enough reads and writes of the feed data, as well as to retain modularity for both future scaling and testing.

One of the technical requirements was running the application on Amazon Web Services (AWS), with an emphasis on utilizing its Lambdas for serverless architecture. Other AWS resources used were S3, RDS, EC2 and SQS.

The application logic was written using Python-based frameworks. Flask was used for API and general server logic, while SQLAlchemy's engine was utilized for Object Relational Mapping on PostgreSQL databases. Marshmallow handled any serialization needed for outgoing and incoming data. Application Lambda's access to the AWS resources was managed by Boto3. For deployment it was necessary to have an easily integrated and lightweight deployment tool and Zappa was chosen for this task.

The result of the work was a working prototype, which supports scalability and has served as a basis for client’s application as well as in another project the designer has worked in.

Keywords

Amazon Web Services, serverless, microservices, Python, Flask

Tiivistelmä

Tekijä(t) Uotila, Timsa	Julkaisun laji Opinnäytetyö, AMK	Valmistumisaika Kevät 2019
	Sivumäärä 40	

Työn nimi

Skaalautuvien mikropalvelujen suunnittelu
Case: Python ja AWS

Tutkinto

Tieto- ja viestintätekniikan koulutus

Tiivistelmä

Opinnäytetyön tavoitteena oli suunnitella ja kehittää tekninen prototyyppi sosiaalisen median applikaation käyttämälle taustajärjestelmälle. Järjestelmän vaatimuksena oli autentikointi, prosessi kuvien käsittelylle sekä riittävän suorituskykyinen luku- ja kirjoitus syötteen datalle. Myös modulaarisuutta tulevaa skaalautumista ja testausta ajatellen pidettiin toimeksiantajan kannalta tärkeänä.

Yksi teknisistä vaatimuksista oli applikaation ajaminen Amazon Web Servicessä (AWS) hyödyntäen tämän Lambdoja Serverless -arkkitehtuurin toteuttamiseksi. Muut käytetyt AWS-resurssit olivat S3, RDS, EC2 ja SQS.

Applikaation logiikka kirjoitettiin Python-pohjaisilla rajapinnoilla. Flask ohjelmistokehystä sovellettiin API -rajapintaan ja yleiseen palvelinlogiikkaan, kun taas SQLAlchemyn moottoria käytettiin ORM -pohjaisiin rakenteisiin PostgreSQL-tietokantojen yhteydessä. Marshmallow hoiti serialisoinnin sekä sisään- että ulosliikkuvassa datassa. Applikaatio-Lambdan yhteydestä muihin AWS-resursseihin vastasi Boto3. Pystyttämisen kannalta oli tarve käyttää helposti integroitavaa ja kevyttä työkalua ja tämän osalta päädyttiin Zappaan.

Työn lopputuloksena oli toimiva tekninen prototyyppi, joka on skaalautuva ja on toiminut pohjana asiakkaan applikaatiossa, kuten myös toisissa alkavissa projekteissa.

Asiasanat

Amazon Web Services, serverless, microservices, Python, Flask

CONTENTS

1 INTRODUCTION

This thesis was commissioned by a startup company that specializes in providing an ecosystem designed to improve the lives of families with small children. The thesis deals with the design and development of prototype for the business logic of the said ecosystem - a server application mainly interfacing a client-side mobile application that enables simple storing and browsing of childhood memories for families and extended families.

Two of the most important considerations for this environment is maintaining scalability while having a cost-effective way of running the service. The application needs to survive the multiple iterations it goes through from a beta version with the small userbase to a highly scaling one on the released version. To fulfil these demands, the application is to be developed in the style of serverless architecture.

The application itself has several requirements for it to be considered a minimum viable product. As it is marketed as a private storage for a family's media, the user authentication should be handled with a reasonable care. The media should be easily accessible and uploadable, but only by authenticated parties with correct access rights to the given data. A user should be able to fetch paginated feed elements quickly enough to ensure a good user experience. A feed element must be able to contain media and must allow comments from users with correct access rights. The application needs to be testable to ensure it fulfils the given quality standards.

2 OPERATIONAL ENVIRONMENT

The operational environment can be described in terms of end users with multiple different mobile applications as well as a backend that works as a mediator for uploading and fetching for uploaded media. The userbase consists of families with children and their closest relatives and friends who they wish to share their children's moments with.

A user can operate the application with various user rights. The parent or the guardian of one or multiple children has rights to manage and add data to the said children's feed. There also exists rights for publishing new data to feed, without added privileges for its management. The latter is often given to close relatives like grandparents who spend a significant portion of time with the child and have their own input to give into their feed. More distant relatives and the family's friends can also be given access rights to read the feed. The figure below (Figure 1) elaborates the different use cases.

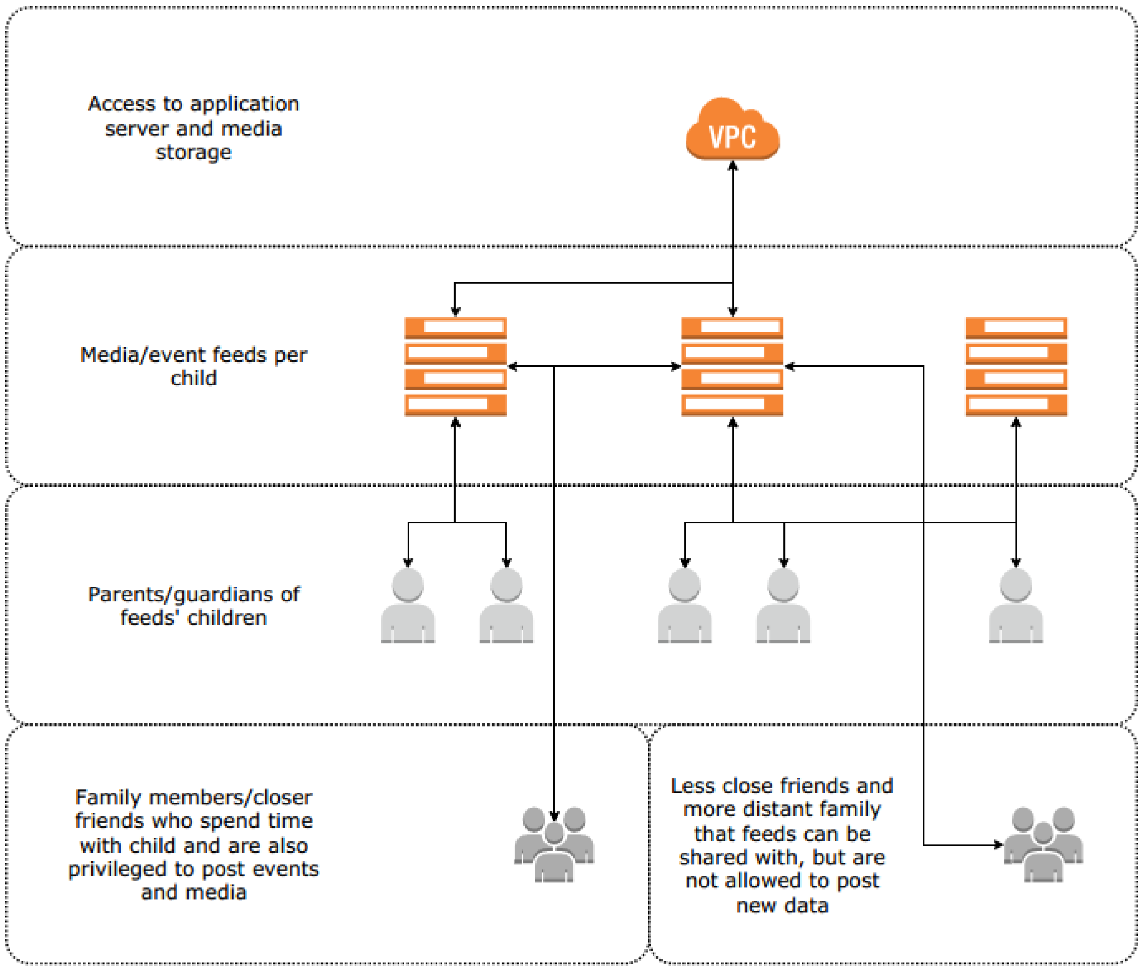

Access to application server and media storage
VPC
Media/event feeds per child
Parents/guardians of feeds' children
Family members/closer friends who spend time with child and are also privileged to post events and media
Less close friends and more distant family that feeds can be shared with, but are not allowed to post new data

Figure 1. Operational environment

As is common in social media applications, the storage needs to be able to contain high volumes of data that is easily fetchable while a large intersecting userbase navigates and alters different content of their choosing. Another important consideration is the accessibility from the point of view of the locality. The internet is a global marketplace and a product should be reasonably easy to expand into a new geographical location.

These requirements can become overwhelming for small businesses. As the amount of data increases over time, so does both the cost and the surface to be maintained. One of the ways to overcome the limitations in workforce and resources is to adopt a Serverless Architecture model and host the server application in the cloud.

3 CLOUD COMPUTING AND SERVERLESS ARCHITECTURE

3.1 General concepts

Cloud technologies have gained popularity with mid-size and small businesses as they can provide more cost-effective and flexible solutions than setting up and maintaining an in-house server. A cloud-hosted solution abstracts away the hardware issues and outsources some of the maintenance burden. (AWS 2019r)

A serverless architecture refers to a specific execution model in cloud computing. This enables running your application logic on isolated containers that are billed by their active runtime. In other words, this way a user pays only for consumed resources, rather than some pre-purchased maximum capacity. (AWS 2019q)

It is worth noting that the term "serverless" is somewhat a misnomer as the application is still run on a server that is run by the cloud provider. Some of the more notable providers are Amazon Web Services, Microsoft Azure and Oracle Cloud Services. The provider utilized for the documented application is Amazon Web Services.

3.2 Amazon Web Services

Amazon Web Services (AWS) is a cloud computing and web service platform developed by Amazon.com Inc. The initial launch of AWS dates to March 2006 and since then has provided an increasingly vast set of building blocks for distributed computing (AWS 2019b). Currently AWS is available in 190 countries and provides over 130 services for computing, storage, networking, analytics and monitoring, among many others (AWS 2019h; AWS 2019n).

Many of the services are divided into georedundant areas where the data center that hosts a customer’s stack is run. These are named AWS Regions. The provider’s catalogue currently spans 20 regions with more to come. This design is aimed for replicability and availability world-wide and offers additional safety for services that could suffer from environmental disturbances, such as earthquakes. (AWS 2019h)

A region is also divided into Availability Zones (AZ), providing a way to apply horizontal scaling. The number of Availability Zones is dependent on the region they exist in, with the largest count of AZs per Region being 6 in US East (North Virginia) and the smallest in Asia Pacific (Osaka) with only one. Utilizing Availability Zones provides benefits from parallelism to stability, which is achieved by having backup machines available on one Availability Zone when the service in another is compromised. (AWS 2019h)

The services in AWS are maintained and deployed from within AWS Console or handled by a scripted deployment tool, like CloudFormation or Terraform. The benefits from the latter are improved versioning and self-documenting, common to a more infrastructure as code (IaC) way of deployment.

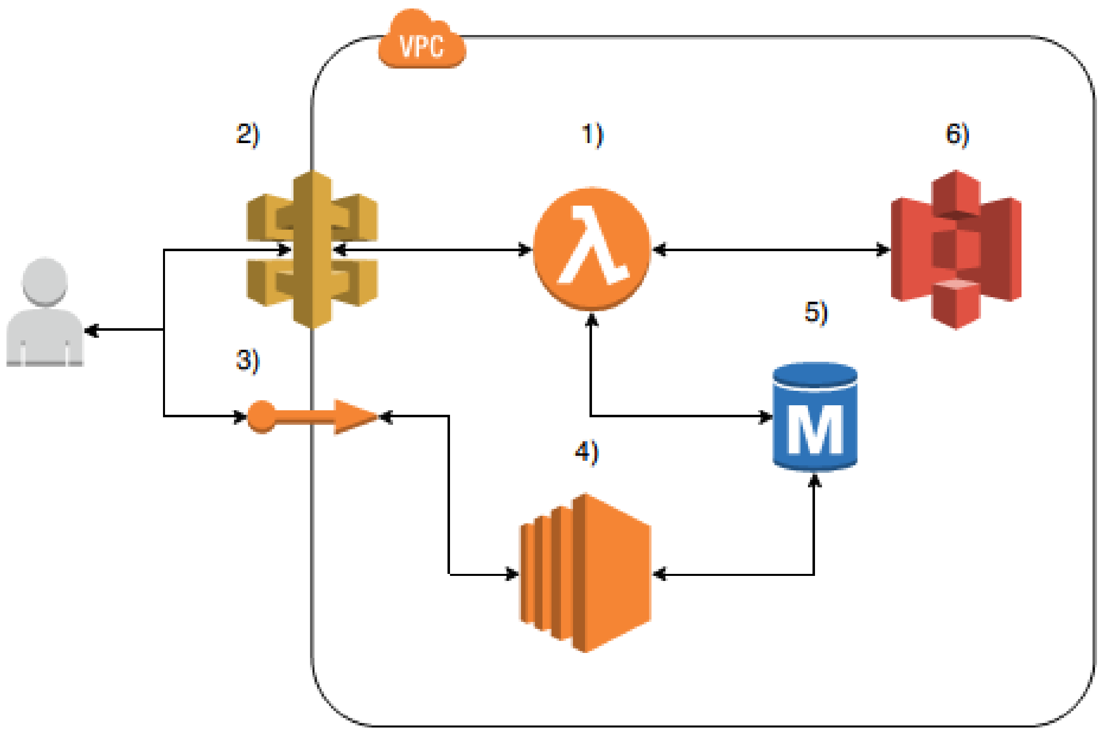


Figure 2. An example of AWS components in relation to each other

The figure above (figure 2) showcases a somewhat typical component stack, including components that are examined in the following sections.

3.2.1 Lambda

The component that enables running serverless routines on AWS is called a Lambda. These can be thought of as processes responding to certain events from other services or mandated by user-supplied code that is written in one of the languages AWS Lambda supports. As of now, application code for Lambda can be written in NodeJS, Java, C#, Go or Python. Common uses for Lambdas are running processors for uploaded data, handling analytic streams or providing API for web and mobile applications (AWS 2019p). Lambda can be seen in the general overview diagram (Figure 2) under number 1.

Lambdas can be invoked via calls to AWS API Gateway or by specified events from other AWS resources (AWS 2019p). The figure below (Figure 3) elaborates different invoke scenarios that set Lambdas running.

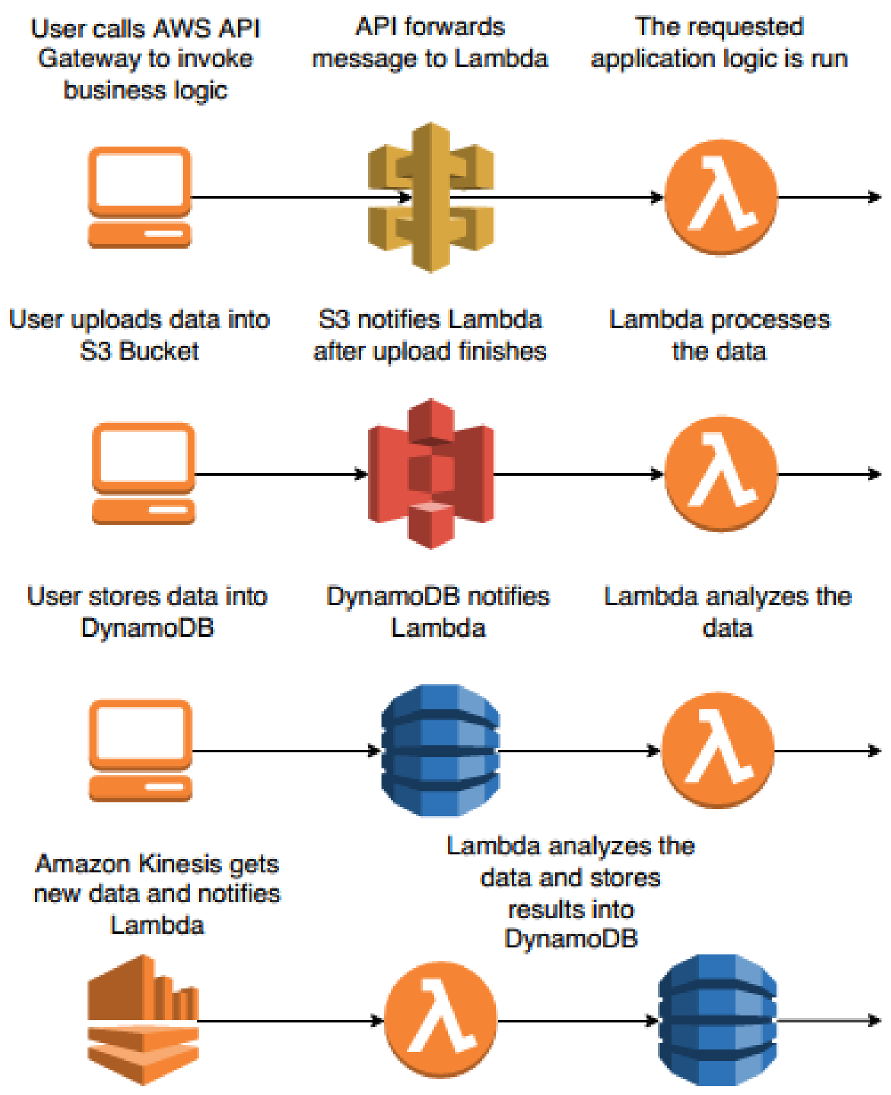

User calls AWS API Gateway to invoke business logic
API forwards message to Lambda
The requested application logic is run
User uploads data into S3 Bucket
S3 notifies Lambda after upload finishes
Lambda processes the data
User stores data into DynamoDB
DynamoDB notifies Lambda
Lambda analyzes the data
Amazon Kinesis gets new data and notifies Lambda
Lambda analyzes the data and stores results into DynamoDB

Figure 3. Different Lambda function invocations

Lambdas provide scalability and affordability, as the program logic is only run on demand and charged respectively. Charging is based on the factors of request count and used computing time. (AWS 2019p)

A request is counted each time a response executes to an event notification or invoke call, including any test invokes from AWS Console. Duration is calculated from the beginning of the executing code to its return or an untimely termination if it fails to finish before specified timeout. The duration is measured with a precision of 100ms. A default timeout for Lambda is 3 seconds, which can be configured up to 900 seconds. (AWS 2019i; AWS 2019j; AWS 2019m)

By default, AWS provides users with a free-tier plan that has one million free requests per month and 3,200,000 seconds of compute time for Lambdas with memory of 128MB. The amount of memory can be increased at the expense of free compute time. (AWS 2019j)

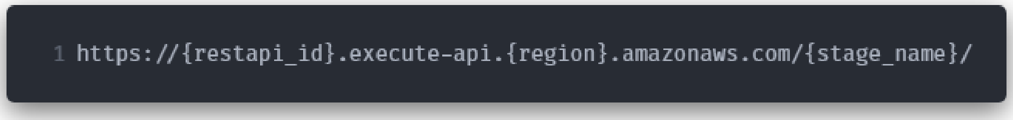
3.2.2 AWS API Gateway

API Gateway is a service designed to provide easily deployable and maintainable APIs into the managed cloud infrastructure. An API can conform to a HTTP/REST or WebSocket -based schemas. API Gateway offers a centralized way for approaching monitoring and endpoint authorization. (AWS 2019c)

The service allows executed API calls to modify resources, notify other AWS-based services and invoke calls from application lambdas residing in S3 Buckets. A closer examination of the said S3 buckets will follow in chapter 3.1.2.

AWS differentiates three different configurable endpoint types: Private, which is only available within the managed Virtual Private Cloud (VPC); Regional, which is optimized for API access from one specified region; and Edge Optimized API endpoints, which enable optimized access from multiple regions.

The figure below (Figure 4) describes the anatomy of invoke URL created by API Gateway. A unique Rest API Id is generated on creation. The region of the said API is also visible in the said URL alongside the given stage name.



```
https://{restapi_id}.execute-api.{region}.amazonaws.com/{stage_name}/
```

Figure 4. An example of an API gateway generated invoke URL

Commonly, API Gateway is deployed via AWS Console, IaC-script or in some cases a specific deployment tool designed to hoist Application Lambdas and their respective API endpoints into the cloud. One of such tools is Zappa, which will be introduced in chapter 4.2. AWS API Gateway can be seen in the general overview diagram (Figure 2) under number 2.

3.2.3 Elastic IP

Elastic IP addresses are public IPv4 Addresses associated with computational instances, namely with different versions of EC2 and the infrastructure’s network interface. Any cloud stack running in an internal private network requires at least one public IP to communicate over the internet. Elastic IPs are often configured to permit access to a public API or create private tunnels for developers and maintainers. (AWS 2019o)

Since public addresses are a limited resource, AWS limits their Elastic IPs to five IPs per region. In a case where a larger amount is needed, a customer must submit an additional request form to Amazon (AWS 2019o). Elastic IP is located in the general overview diagram (Figure 2) under number 3.

3.2.4 EC2

Amazon Elastic Compute Cloud, or EC2 for short, supplies AWS customers with computing capacity in the form of hosted machine instances. A user can select from a variety of hardware setups with different statistics in terms of CPU, memory and storage. Instances with both Linux and Windows platforms are available. (AWS 2019d)

Another selling point for EC2, besides cutting off costs in hardware, is its significant speed in deployment and scaling compared to traditional server management. Where obtaining the servers themselves would be time-consuming, normally several EC2 instances can be deployed from a few minutes to hours. Instances can also be configured to automatically scale on demand, which cuts costs by not running extra instances until they are needed.

For new users AWS offers an affordable 750 hours per month of compute time on EC2 instances for the first 12 months included in its Free Tier plan (AWS 2019e). This allows considerable cuts to expenses while the initial version of the application is developed.

There are four different payment models according to the selected type of instance. On-Demand charges for computing capacity per hour or second, depending on the selected instance, making this a good choice for environments with unpredictable workloads. Spot Instances allow additional control with interrupts that reduce running time. Reserved Instances allow for reduced costs in EC2 with environments that commit to usage over 1- or 3-year terms. Dedicated Hosts allow discounts for customers that are dependent on existing server-bound software licenses. A payment plan should reflect the needs of the application workload and customer’s organization (AWS 2019e). EC2 can be seen in the general overview diagram (Figure 2) under number 4.

3.2.5 RDS

Amazon Relational Database Service (RDS) provides customers with easily scalable database instances, offering a choice of multiple relational database engines from Amazon Aurora to PostgreSQL, MySQL, MariaDB, Oracle Database or SQL Server (AWS 2019g).

RDS can be run in multi-AZ mode, which enables running replicas in different Availability Zones. These replicas can take over if a used database instance goes down. Other stability-increasing features are automated backups, database snapshots and automatic host replacement. With some engines, the read traffic can be offloaded by utilizing a deployment of Read Replicas. In these cases, writes happen to Master Write database, while reads are only done from Read Replica. This can speed up both operations if database is under a heavy load. (AWS 2019g)

RDS offers two payment modes, on-demand pricing and reserved-instance pricing with variable upfront costs. As a rule of thumb, on-demand pricing would be beneficial for a development and staging databases, with inconsistent state and varied need for availability, while production that is usually meant to be available 24/7 could benefit from the reserved-instance plan (AWS 2019f; AWS 2019g). RDS is located under number 5 in the general overview diagram (Figure 2).

3.2.6 S3 Buckets

S3, or Simple Storage Service is a data storage designed for file-level data, such as photos, videos and documents. The data is stored as keyed objects inside an S3 bucket, which several different operations can be performed on. AWS has a specific API, which can be accessed manually via AWS Console or programmable instances like Lambdas. Accessing, creating, updating or deleting a bucket can also invoke calls to other AWS hosted services (AWS 2019k). S3 Bucket can be seen in the general overview diagram (Figure 2) under number 6.

Apart from storing static assets, it’s a common pattern in AWS to store application code inside a bucket. This can either be the client website that is being served to outside world or the Lambda code that is read during specific API Gateway calls. Some frameworks utilize buckets in storing application-level configurations and environment variables.

Considering costs, S3 charges are based on used storage as well as types of API calls made to the buckets. The calls are divided into PUT, COPY, POST and LIST, which cost $0.005/1000 requests, while GET -requests run for mere $0.0004/1000 requests. The basic pricing for storage starts from $0.023 for the first 50gigabytes, $0.022 for the next 450 down to $0.021 for storage that grows over 500gigabytes. Additional cheaper plans are also available if more infrequent access to the storage is needed. AWS Free Tier plan covers 2000 PUTS, 20 000 GETs and 15gigabytes of outward data transfer for the first year. (AWS 2019l)

3.2.7 SQS

Amazon Simple Queue Service refers to a service that enables users with message queues, which can be easily scaled and managed within AWS ecosystem. Queues can provide control over message-oriented middleware and ensure reliable delivery of their contents. AWS also provides encryption when the delivered data is sensitive.

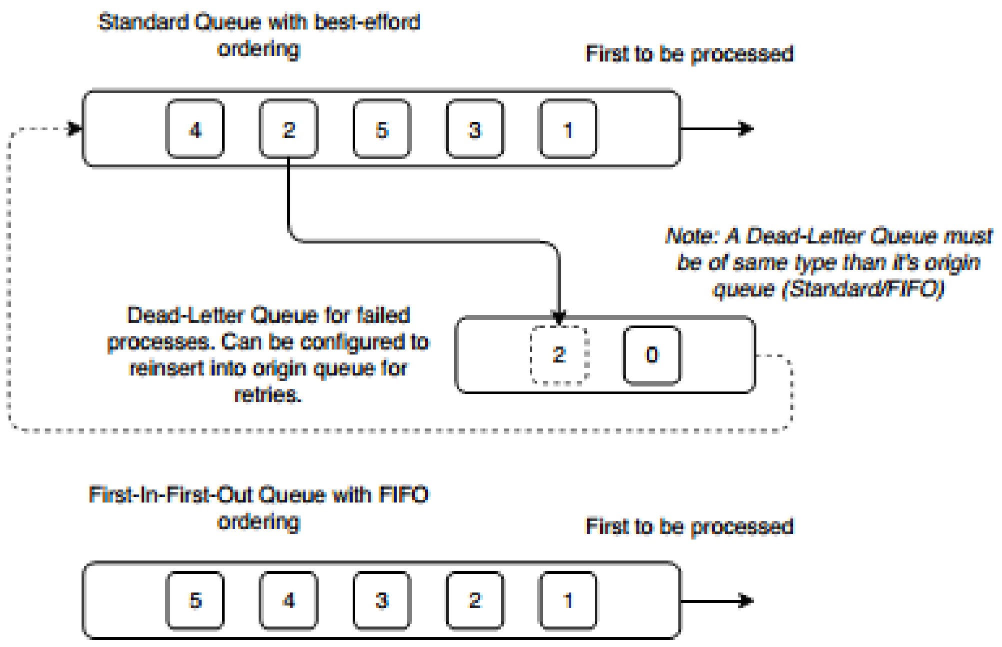

Standard Queue with best-efford ordering
First to be processed
4
2
5
3
1
Note: A Dead-Letter Queue must be of same type than it's origin queue (Standard/FIFO)
Dead-Letter Queue for failed processes. Can be configured to reinsert into origin queue for retries.
2
0
First-In-First-Out Queue with FIFO ordering
First to be processed
5
4
3
2
1

Figure 5. AWS Standard Queues and FIFO queues

AWS offers two distinct queue types: Standard queues and FIFO queues, where standard queues offer maximum throughput and reliable at-least-once delivery, while FIFO (First In, First Out) queues enable messages to be processed in exact order and only once. Both queues can be configured to utilize AWS Dead Letter queues, which handle erroneous attempts at processes. With these it is possible to employ schemes for retries and error logging. The figure above (Figure 5) illustrates both queue variants.

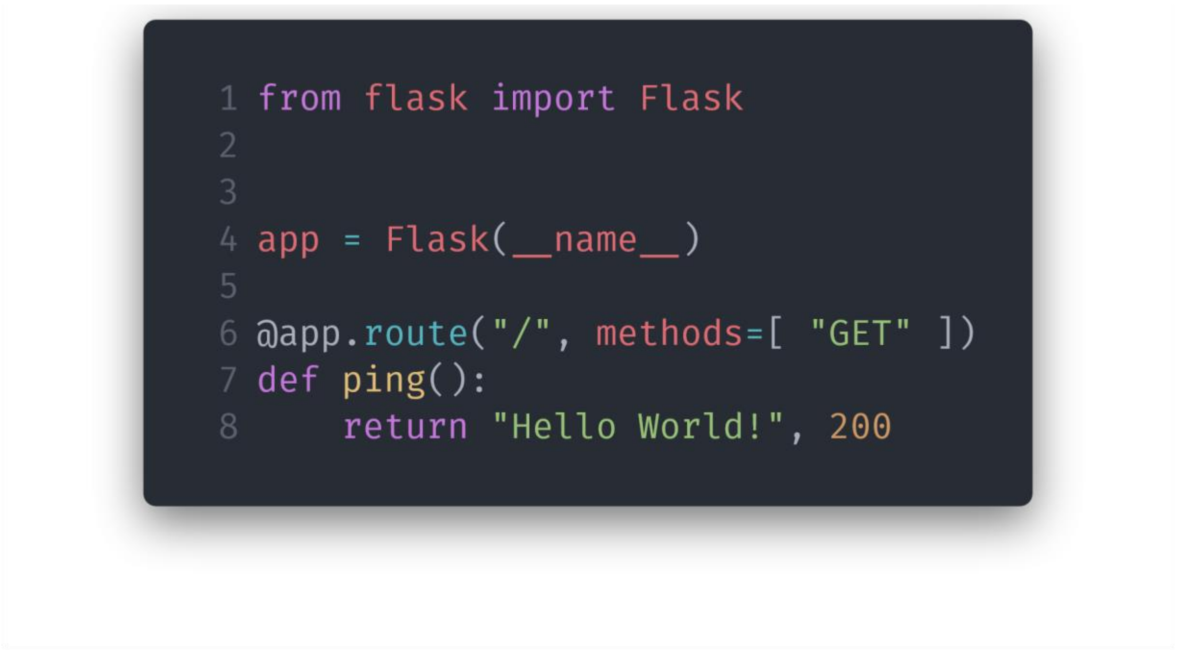
4 OVERVIEW OF SERVER-SIDE TECHNOLOGIES

With AWS serving as the cloud infrastructure and Lambdas running the application logic, an AWS compatible run-time environment is needed. Python is one of the languages supported by AWS and has well-evolved SDK in the form of Boto3. If the application is lightweight and doesn’t require heavy modelling of database objects, using this approach alone is perfectly sufficient. However, if the developed microservices are intended to function with more complex relational data and response schemas, a right framework might offer some benefits in the form of effective management of database bindings and well-tested network handling. Python has some strong AWS compatible contenders in the form of Flask and Django, of which the former is examined.

4.1 Flask

Flask is WSGI (Web Server Gateway Interface) based microframework for Python with built-in development server and debugger, good extensibility and documentation to name a few features. Flask can be both used to serve templated html -pages, as well as a basis for REST/CRUD APIs. (Flask 2019)

In terms of application architecture Flask is unopinionated. At the core it’s a minimalistic framework that has capability to be structured according to MVW (Model-View-Whatever) paradigms but doesn’t force developer’s hand either way. For developers that have been heavily into MVC -applications, it’s noteworthy to point out that Flask community and documentation tends to call “views” what traditional MVC calls “controllers”, while traditional “views” are referred as “templates”. In the snippet below (Figure 6), a naive Flask application with an example view is being initialized.



```python
from flask import Flask


app = Flask(__name__)

@app.route("/", methods=[ "GET" ])
def ping():
    return "Hello World!", 200
```

Figure 6. A naive Flask application initialization

Writing a Serverless application with a framework like Flask can utilize best from both microservice and more traditional monolithic architectures. Many of commonly used patterns and tested utilities can be reused, while breaking the application into smaller pieces or multiple Flask instances make the solution more operable in serverless context. An application or a microservice component written in Flask can be deployed to AWS with a relative ease using a tool like Zappa.

4.2 Zappa

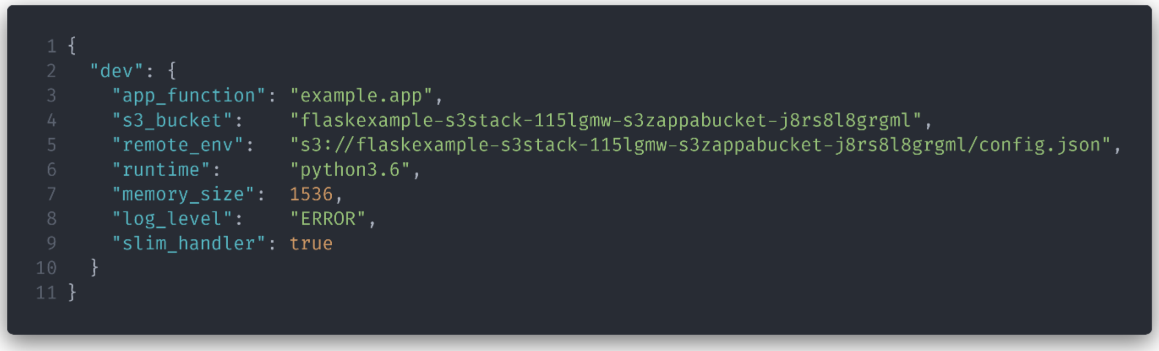
Zappa is a tool aimed to help deployment of Serverless Python Web Services. It is written on top of Boto3 and provides bindings for both the AWS Lambda and API Gateway and is capable of handling applications written in Django and Flask. With Zappa, it’s possible to omit predefined API Gateway bindings from the uploading phase of cloud environment and defer them to be established when the first deployment of application logic is done. (Zappa 2019a; Zappa 2019b)

The semantics of Zappa’s behaviour can be divided into two phases. First the application is packaged into an archive that contains all the required code for the application to run. During this process Zappa will replace any local dependencies with AWS compatible versions and skip unnecessary files. Such are like Boto, as it is already available in the Lambda execution environment. After this the application is uploaded into an S3 -bucket, that the lambda can be invoked from. (Zappa 2019b)

A common issue in Serverless applications is perceived overhead when a new Lambda container is started at the beginning of each request. This is because an application environment needs to set up the environment each time. Zappa enables a feature that keeps AWS Lambda in a so-called warm state. This means that the application itself stays initialized, but paused, waiting for new requests to come in, in which case it wakes up and runs the needed logic. This effectively removes the unnecessary overhead while still reaping the cost benefits from the on-demand computing, while the only downside is forcing a new deployment each time some of the application initialization configurations change. (Zappa 2019b)

Zappa requires the developer's machine to be set up with AWS CLI to enable a connection to the targeted AWS environment. After this the details of Zappa deployment are defined with a configuration file in either JSON or YML formats. Below is a naive example snippet (Figure 7) of said zappa_settings.json.

The second line on the snippet designates a stage name that serves as an identifier in multi-stage configurations. The parameter "app_function" on line 3 holds the modular path to the application from the point of the view of the Zappa's configuration file. In example the application would be found from the file named "example.py" residing in the same directory with the configuration file. The application would be initialized into variable called "app". The following line (line 4) in the configuration states the S3 bucket where the application is to be uploaded. Apart from this, Zappa can be given remote environment variables (line 5) that reside inside an S3 bucket. In this example the variables are defined in a json-file located in a same bucket where the application is to be uploaded into. Another common configuration is the used Python runtime (line 6) allowing the use of python2.7, python3.6 or python3.7. If not given, the runtime defaults to whatever is being used. "memory_size" designates how many MBs of memory is allocated for the deployed Lambda function defaulting to 512, while "log_level" adjusts the Python debuggers output level on AWS. The last property in the example (line 9) is beneficial for projects larger than 50MB, allowing the actual Zappa handler to be condensed into a smaller handler which in turn loads the actual application from S3 at runtime. (Zappa 2019b)



```json
{
  "dev": {
    "app_function": "example.app",
    "s3_bucket":    "flaskexample-s3stack-115lgmw-s3zappabucket-j8rs8l8grgml",
    "remote_env":   "s3://flaskexample-s3stack-115lgmw-s3zappabucket-j8rs8l8grgml/config.json",
    "runtime":      "python3.6",
    "memory_size":  1536,
    "log_level":    "ERROR",
    "slim_handler": true
  }
}
```

Figure 7. A naive example of zappa configuration file in json -format

4.3 SQLAlchemy

SQLAlchemy is a powerful data-binding library designed to bridge Python code with different SQL engines, attempting to enable a development experience where the Python-level code is as agnostic as possible towards the underlying databases semantics (Myers & Copeland 2016). Currently SQLAlchemy provides support for Relational Database engines such as PostgreSQL, MySQL, SQLite, Oracle, Microsoft SQL Server, Firebird and Sybase (SQLAlchemy 2019a).

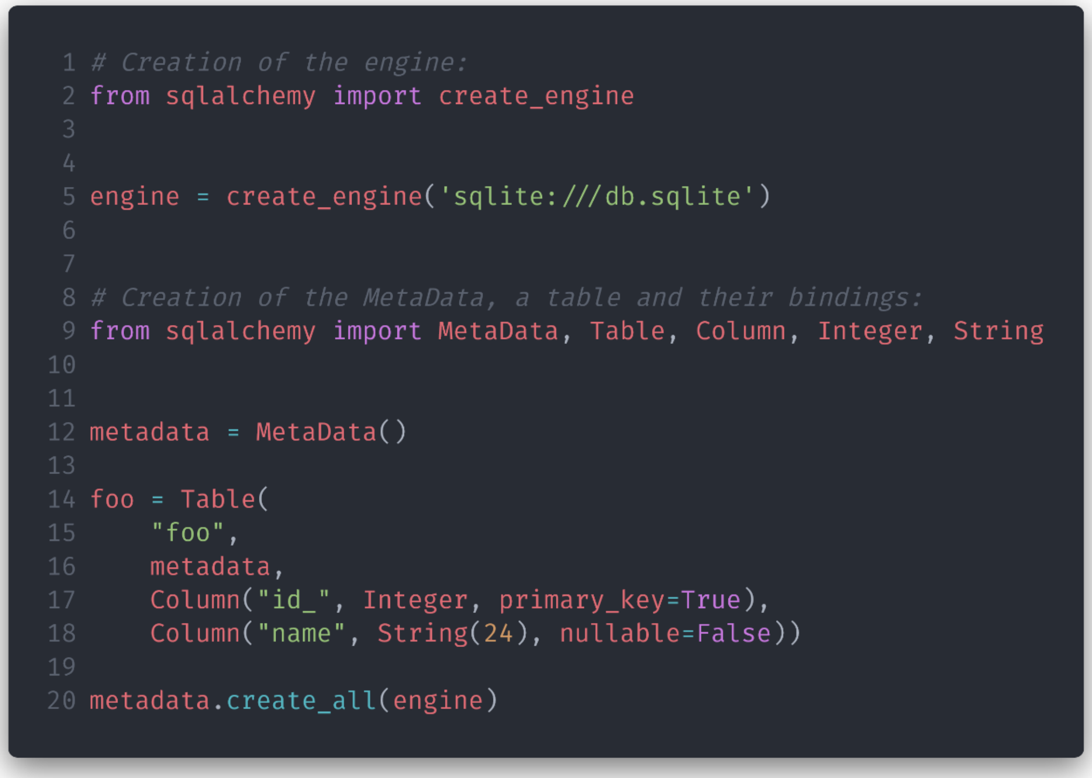
SQLAlchemy makes the data access layer code highly migratable across different languages and is a good fit for environments that interface multiple different database schemas. Another benefit is the security of the library's properly sanitized and escaped functions. The library can be run in two combinable modes, making it very flexible in multiple situations (Myers & Copeland 2016: xiii-xv).

The two distinguishable modes of SQLAlchemy are called Core and ORM. Core is the low-level API that exposes library's own query syntax, while ORM is written on top of Core and enables the developer to create the data layer by the principles of Object Relational Mapping. When needed, a developer using an ORM can drop down to Core-level syntax to combine both approaches. (Myers & Copeland 2016: xiii-xv).

A suitable mode should be selected according to the domain's needs. For example, the Core is fitting for times when a more schema-centric view is required or when working with data for which business objects are not needed. It's also a convenient solution when working with a framework that already has its own ORM present. ORM could be utilized when the data is primarily viewed as business objects or when a quick prototype is required. It's also completely fine to mix approaches, such as having the business application conform into ORM bindings and handle analytics with Core's more low-level functions. (Myers & Copeland 2016: xiii-xv)

To understand both modes' behaviour, it's necessary to have a look how a database connection and mapping is created. For SQLAlchemy to function, two components, Engine and MetaData, require to be configured. The snippet below (Figure 8) elaborates an example configuration where a working database connection is achieved.

First the engine is created on line 5 by passing it a connection string to the underlying database server. Then, a MetaData object needs to be created to act as a book-keeper between the database session and given table configurations. This instance can be bound to a table when the given table is configured (lines 14-18). To finish the bindings to all of the created tables, the metadata needs to invoke "create_all" -function (line 20), that only applies creation to instances that are not yet reflected by SQLAlchemy to avoid attempting redundant creation of existing tables.



```
 1 # Creation of the engine:
 2 from sqlalchemy import create_engine
 3
 4
 5 engine = create_engine('sqlite:///db.sqlite')
 6
 7
 8 # Creation of the MetaData, a table and their bindings:
 9 from sqlalchemy import MetaData, Table, Column, Integer, String
10
11
12 metadata = MetaData()
13
14 foo = Table(
15     "foo",
16     metadata,
17     Column("id_", Integer, primary_key=True),
18     Column("name", String(24), nullable=False))
19
20 metadata.create_all(engine)
```

Figure 8. Steps to an example connection with SQLAlchemy

4.3.1 Object Relational Mapping

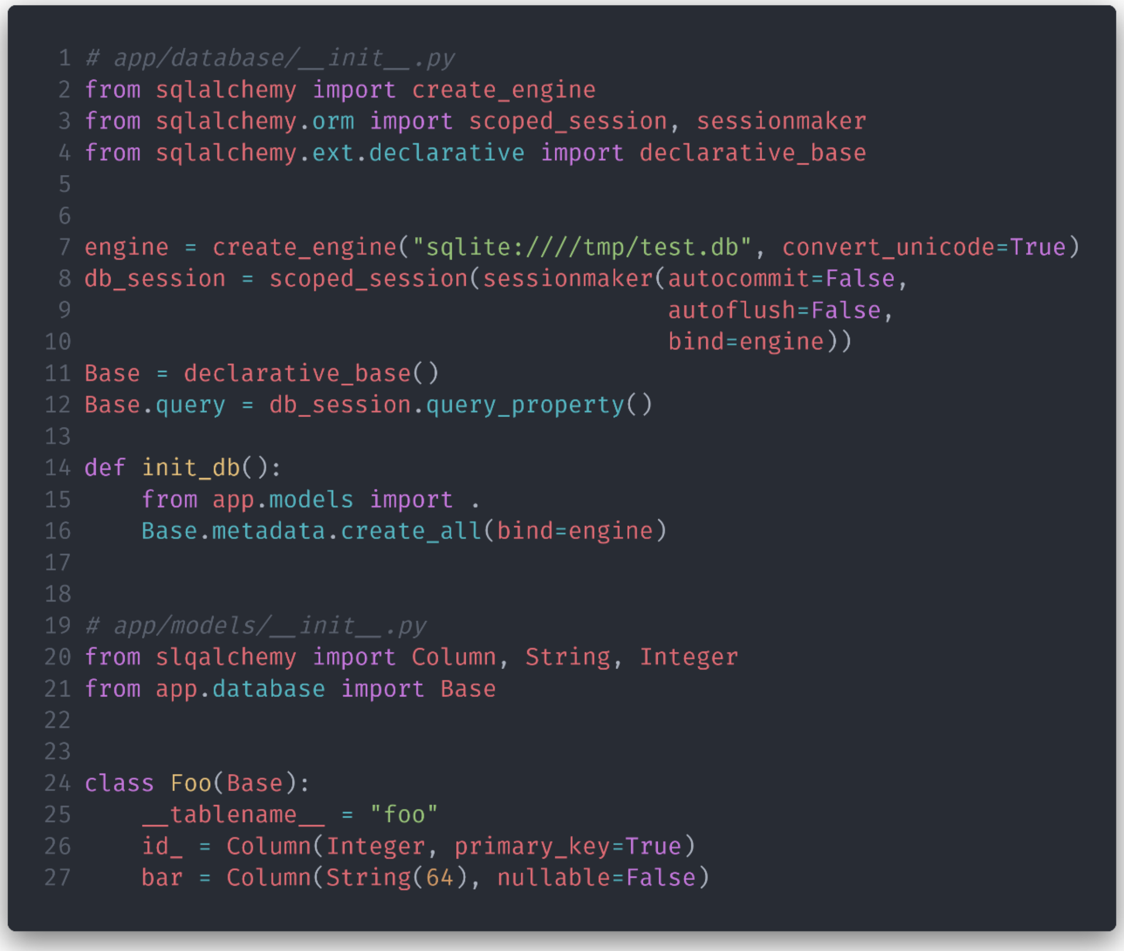
The utilities enabling ORM bindings are located in SQLAlchemy's orm -module. Models are utilized by creating a declarative base that serves as the parent object to be extended by the

ORM -models. The declarative base object contains a similar MetaData that is used by the Core -bindings. It is also possible to create Models without using declarative base, via manual object relational mapping. However, this approach adds little to no benefits while introducing extra verbosity. As a predecessor for declarative approach, manual mapping for ORM is mostly retained for backwards compatibility.

An important concept in ORM based operations is the notion of Session. This essentially holds the current database connection and works as a holding zone for Python-level models representing data that is being either queried or under a transaction.

Any ORM-level object can have one of five different states from the perspective of the session. “Transient” refers to a state where object has yet to be written into the database but has no existing database identity. Once an object is added into a session, either directly or via parent object, its state turns into “Pending”. This means the object has not yet been flushed into the database but will be during the next occurring flush. “Persistent” state refers to an object that has just been flushed or is queried from database. “Deleted” is essentially a state where delete transaction has been called, but not yet flushed. After a flush, the object enters “Detached” state, rendering it inaccessible.

The snippet below (Figure 9) presents variations to the previous example (Figure 8) that implement an Object Relational Mapping. The function “init_db” (line 13) is intended to be called once at the beginning of the application’s initialization.



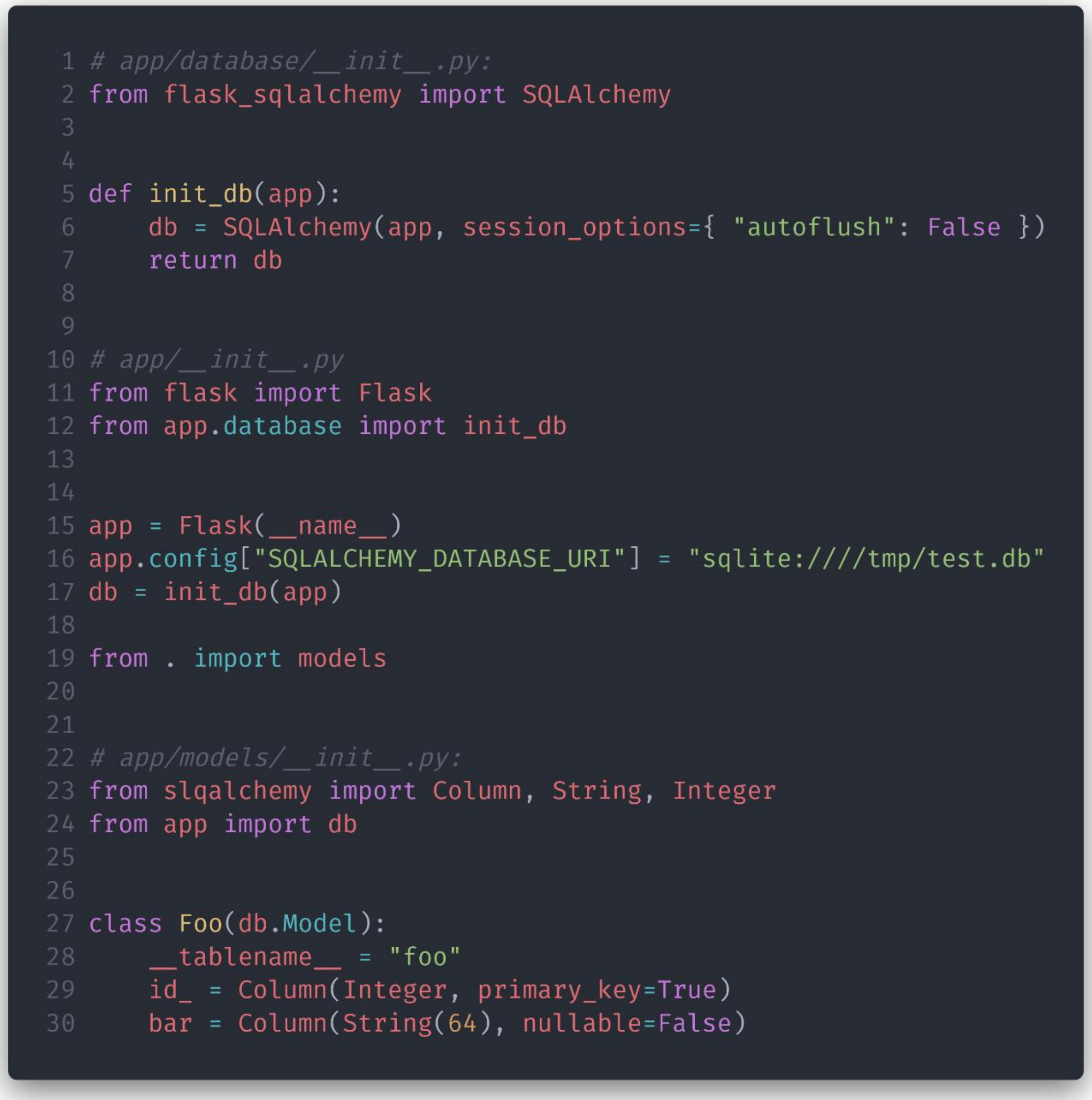
```python
# app/database/__init__.py
from sqlalchemy import create_engine
from sqlalchemy.orm import scoped_session, sessionmaker
from sqlalchemy.ext.declarative import declarative_base


engine = create_engine("sqlite:////tmp/test.db", convert_unicode=True)
db_session = scoped_session(sessionmaker(autocommit=False,
                                         autoflush=False,
                                         bind=engine))
Base = declarative_base()
Base.query = db_session.query_property()

def init_db():
    from app.models import .
    Base.metadata.create_all(bind=engine)


# app/models/__init__.py
from slqalchemy import Column, String, Integer
from app.database import Base


class Foo(Base):
    __tablename__ = "foo"
    id_ = Column(Integer, primary_key=True)
    bar = Column(String(64), nullable=False)
```

Figure 9. Binding an engine and models into a declarative base class

4.3.2 Flask-SQLAlchemy

Flask-SQLAlchemy is an optional extension for Flask applications using SQLAlchemy. It provides ease of configuration by abstracting away a lot of manual work involved with session's book-keeping. Flask-SQLAlchemy handles removing all the database sessions automatically, that previously had to be written explicitly. Besides this, a developer commonly needs very little access to the internals that the extension hides, making the improvement of signal-to-noise ratio between useful business logic and unnecessary technical cruft a welcome one (Flask-SQLAlchemy 2019).

The initialization with Flask-SQLAlchemy is elaborated in the snippet below (Figure 10). One of the notable differences with more inlined approach is that this way the whole configuration step can be made at once making the configured database object returnable. Unlike in the previous example (Figure 9) where the session is global to the module it is defined in and can be accidentally imported before necessary tie-ins have been finished, the latter enforces a structure that doesn't abuse Python's import resolution.



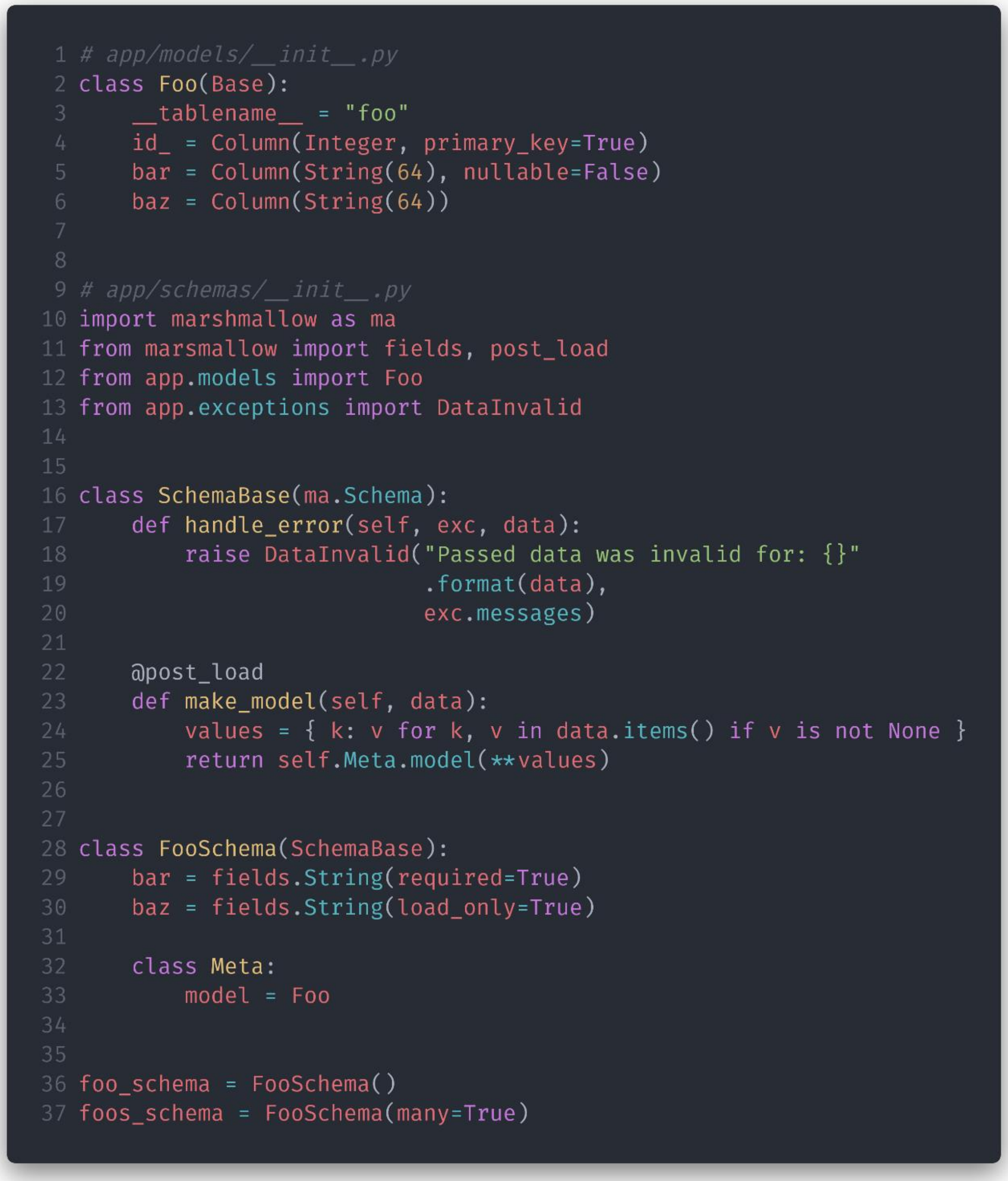
```python
# app/database/__init__.py:
from flask_sqlalchemy import SQLAlchemy


def init_db(app):
    db = SQLAlchemy(app, session_options={ "autoflush": False })
    return db


# app/__init__.py
from flask import Flask
from app.database import init_db


app = Flask(__name__)
app.config["SQLALCHEMY_DATABASE_URI"] = "sqlite:////tmp/test.db"
db = init_db(app)

from . import models


# app/models/__init__.py:
from slqalchemy import Column, String, Integer
from app import db


class Foo(db.Model):
    __tablename__ = "foo"
    id_ = Column(Integer, primary_key=True)
    bar = Column(String(64), nullable=False)
```

Figure 10. A database configuration made with Flask-SQLAlchemy extension

4.4 Marshmallow

Marshmallow is framework-agnostic library designed to help the conversion of different datatypes, to and from native Python datatypes. The benefits from using such library are unified API for deserialization and serialization and well-tested validation utilities for the data to be converted. As most APIs deal with JSON, some form of conversion is needed. The more security-critical the application is the stricter and more careful the validation needs to be. Another compelling argument for Marshmallow is its easy integration with SQLAlchemy's ORM-level.

Marshmallow calls data transforms to python "loading", and from python "dumping". In the code below (Figure 11) provides an example usage of configuration for an arbitrary model that can be loaded and dumped both as a single model or within a collection (lines 36-37). A Schema can be configured using a base class that can be bound with application specifics, such as error handling (lines 17-20) and utility method that automatically calls corresponding model's constructor after a load event (22-25).



```python
# app/models/__init__.py
class Foo(Base):
    __tablename__ = "foo"
    id_ = Column(Integer, primary_key=True)
    bar = Column(String(64), nullable=False)
    baz = Column(String(64))


# app/schemas/__init__.py
import marshmallow as ma
from marsmallow import fields, post_load
from app.models import Foo
from app.exceptions import DataInvalid


class SchemaBase(ma.Schema):
    def handle_error(self, exc, data):
        raise DataInvalid("Passed data was invalid for: {}"
                          .format(data),
                          exc.messages)

    @post_load
    def make_model(self, data):
        values = { k: v for k, v in data.items() if v is not None }
        return self.Meta.model(**values)


class FooSchema(SchemaBase):
    bar = fields.String(required=True)
    baz = fields.String(load_only=True)

    class Meta:
        model = Foo


foo_schema = FooSchema()
foos_schema = FooSchema(many=True)
```

Figure 11. An example of serialization and deserialization configurations with Marshmallow

5 DESIGN AND DEVELOPMENT

The implementation for the prototype server, is required to be extensible, reliable and provide clear separation of concerns. Architecturally, maintainability and high-level of code reuse is prioritized, while reasonable performance serves as a secondary aim.

5.1 Cloud Infrastructure

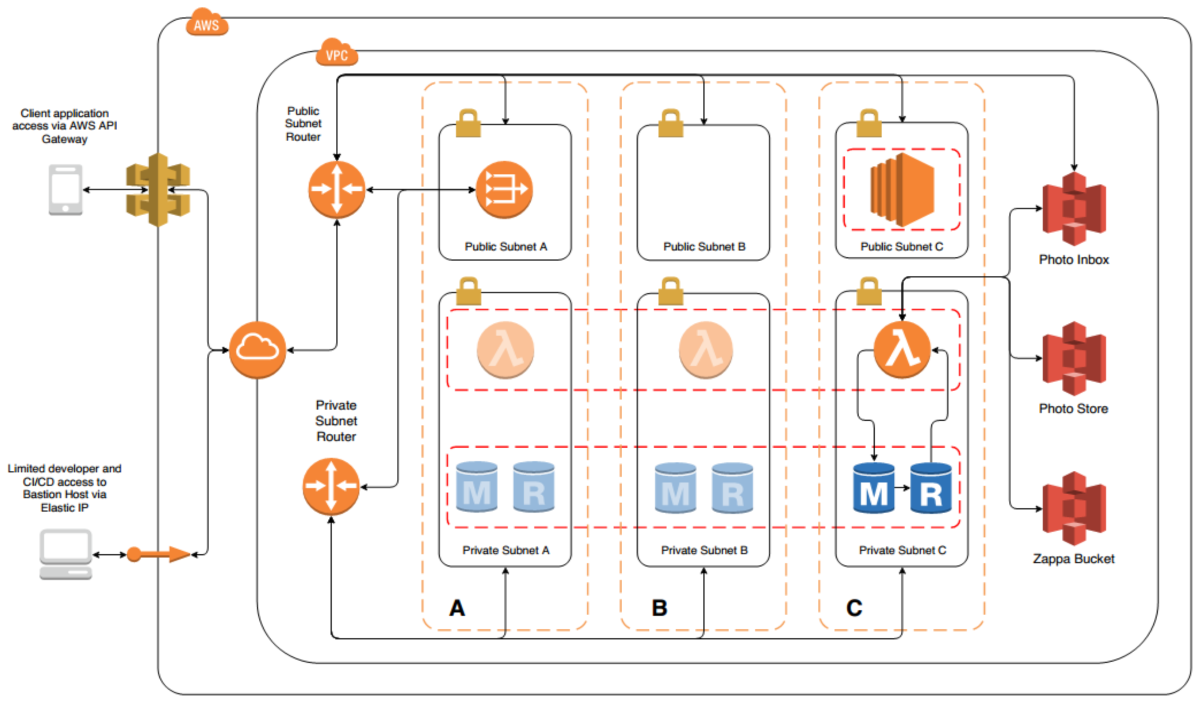


Figure 12. The cloud infrastructure of the server application

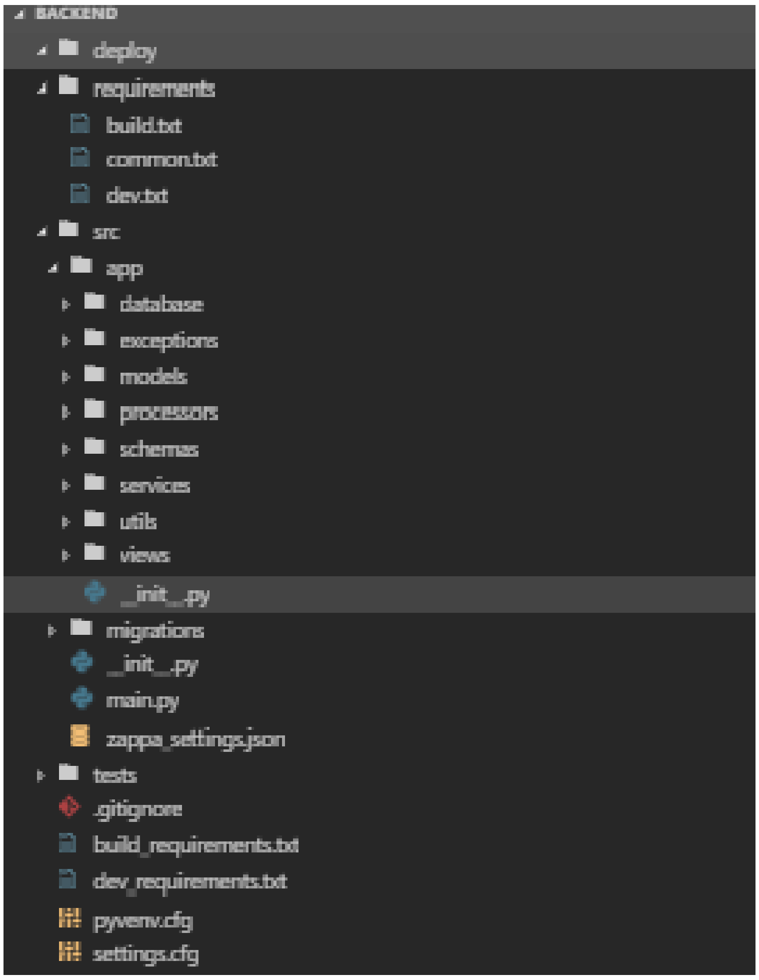
The structure for the Virtual Private Cloud (VPC) the application is meant to run in, is shown in the diagraph above (Figure 12). Starting from the upper left corner, an AWS API Gateway is employed to be the primary gateway into the cloud environment. The API Gateway is connected to the Internet Gateway interfacing a public subnet router, enabling all connections with internet to and from the cloud. The Infrastructure is divided into three Availability Zones, starting from C, meant to be extended via duplication when needed. All these AZ's have their corresponding public and private subnets. A NAT Gateway residing in public subnet A enables the address translation for public connections.

Critical resources have been secured with security groups according to their type. These are described with dashed red lines in the diagram. The Application Lambda and possible duplicates reside in private subnets. They have been given their own specific security group, allowing them to be accessed only by specific parties, mainly the Zappa's deployment and other AWS resources that it has bound onto the Lambda. Another private subnet resident is the Database cluster, consisting of Master and Read Replica instances, enjoying from the security of their own security group. The access to this security group is reserved to residents in the Lambda-level security group.

Besides limiting the database access to Lambdas, another backdoor is required for developer and migration access. This originates from the lower left corner's dedicated Elastic IP, which is connected to the EC2 instance on the public subnet C. This instance serves as a reverse proxy, forwarding ports to the database instances. The access is limited only to specific users, governed by a security group of their own. A reverse proxy such as this is commonly referred to as a Bastion Host and shall be referred to as such for the remainder of this thesis.

5.2 Project Structure

As Flask applications are very flexible about project structure, a developer has the freedom to configure the application as he/she pleases. In this work the application structure aims to provide clear separation of concerns with the limits of Python's own import system in mind. The folder tree below (Figure 13) describes the structure to be used.



```
BACKEND
  deploy
  requirements
    build.txt
    common.txt
    dev.txt
  src
    app
      database
      exceptions
      models
      processors
      schemas
      services
      utils
      views
      __init__.py
    migrations
    __init__.py
    main.py
    zappa_settings.json
  tests
  .gitignore
  build_requirements.txt
  dev_requirements.txt
  pyvenv.cfg
  settings.cfg
```

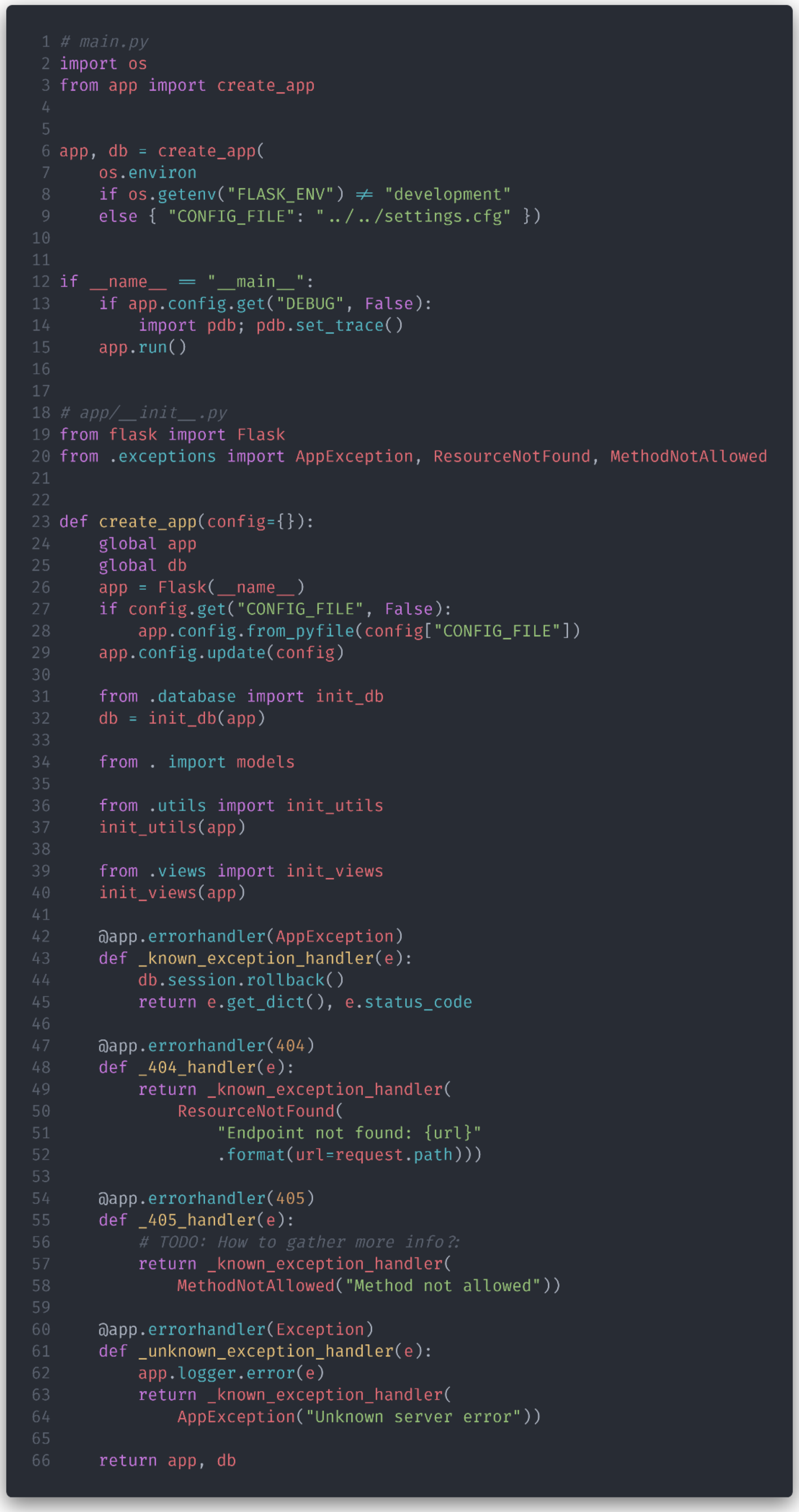
Figure 13. General project structure of the work

At the core, the root of the application project has default configurations for the local development environment as well as files separating the dependency requirements for different builds. Local application settings are to be found in “settings.cfg” while “pyvenv.cfg” holds the configuration about the current Python environment. In the folder, the dependencies are divided into development and build level dependencies that point into the requirements folder containing clear separation for build and development only libraries, while the file “common.txt” contains all libraries required by both build variants. A development-level variant is only to be used locally, whereas build one is reserved for deployment.

Besides the requirements folder, the root contains specific directories for deployment and tests. The actual application is in the src folder defining the structure of the application with a Python module called “app”. The previously mentioned configuration file for Zappa resides here, as well as the main module that holds the initialized app.

The application module is separated into contextual submodules for different levels of business logic as well as to ease the initialization process that is dependent on Python’s import resolution order. As a quick rundown, the database module contains the necessary code for setting up the database connection and semantics of its configuration. The intention is that no code in this folder is run after the app has started up. The module for exceptions is a simple utility module containing the definition of app-level exceptions and their semantics. “Models” contain the SQLAlchemy ORM-level definitions for the business objects of the application. “Processors” is reserved to functionality intended to be invoked by events received by Lambda, like in the case of photo uploads, which is examined later in more detail. “Schemas” contain the configuration for serialization, while “services” is a module handling the responsibilities of the Data Access Layer. “Utils” handles the setup and use of external resources, located in AWS or otherwise. Finally, the “views” module holds the code configuring different routes for the API.

The initialization of the app is divided into two parts. Flask provides an App Factory pattern for clear separation of the configuration phase and start-up phase of the application. As Zappa itself can omit configuration from the overhead cost of the served application and a developer often needs different configuration between different builds, this lends itself well for the situation at hand. Below is a snippet (Figure 14) comprised of multiple files, describing the configuration steps.



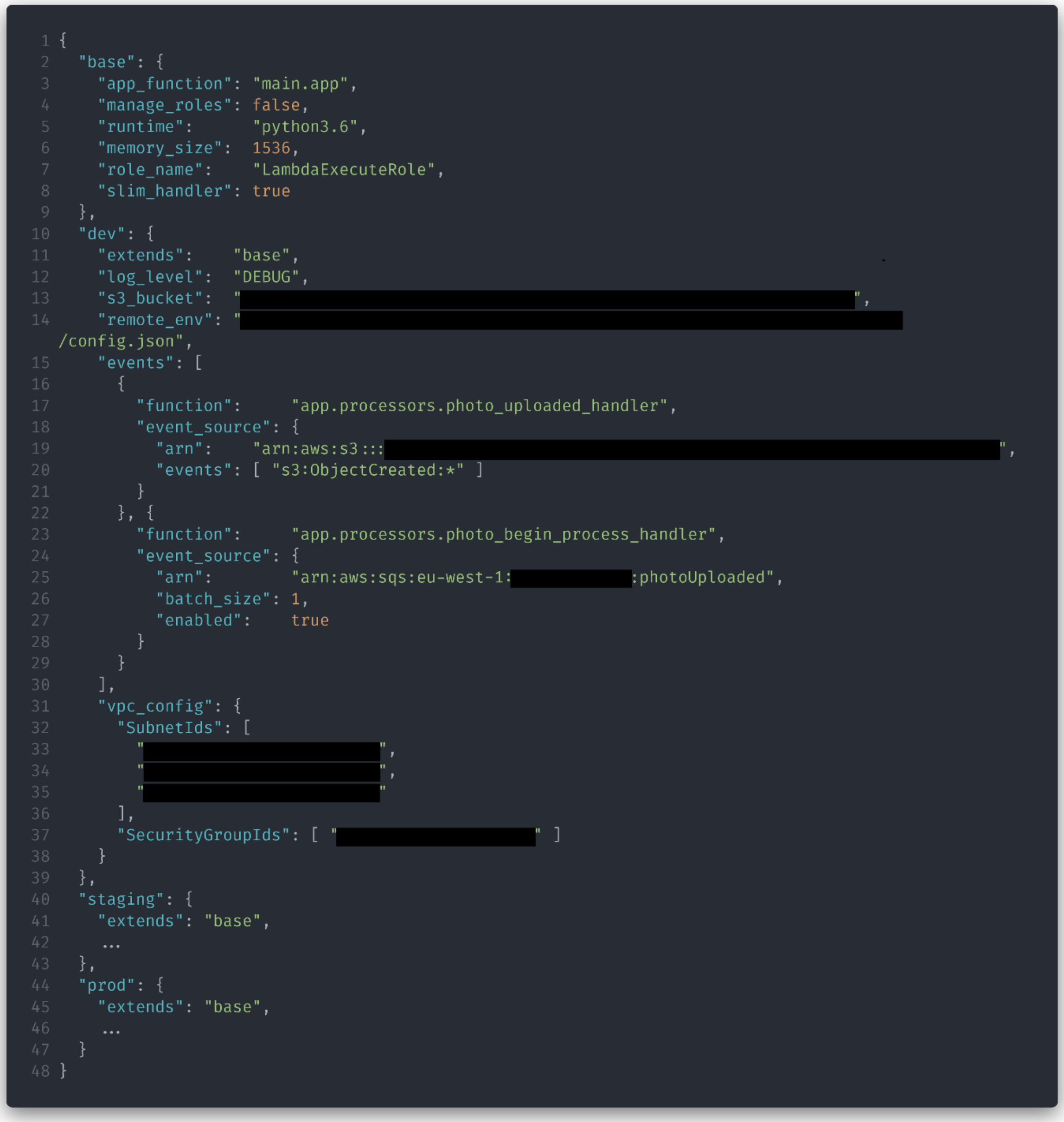
```python
# main.py
import os
from app import create_app


app, db = create_app(
    os.environ
    if os.getenv("FLASK_ENV") != "development"
    else { "CONFIG_FILE": "../../settings.cfg" })


if __name__ == "__main__":
    if app.config.get("DEBUG", False):
        import pdb; pdb.set_trace()
    app.run()


# app/__init__.py
from flask import Flask
from .exceptions import AppException, ResourceNotFound, MethodNotAllowed


def create_app(config={}):
    global app
    global db
    app = Flask(__name__)
    if config.get("CONFIG_FILE", False):
        app.config.from_pyfile(config["CONFIG_FILE"])
    app.config.update(config)

    from .database import init_db
    db = init_db(app)

    from . import models

    from .utils import init_utils
    init_utils(app)

    from .views import init_views
    init_views(app)

    @app.errorhandler(AppException)
    def _known_exception_handler(e):
        db.session.rollback()
        return e.get_dict(), e.status_code

    @app.errorhandler(404)
    def _404_handler(e):
        return _known_exception_handler(
            ResourceNotFound(
                "Endpoint not found: {url}"
                .format(url=request.path)))

    @app.errorhandler(405)
    def _405_handler(e):
        # TODO: How to gather more info?:
        return _known_exception_handler(
            MethodNotAllowed("Method not allowed"))

    @app.errorhandler(Exception)
    def _unknown_exception_handler(e):
        app.logger.error(e)
        return _known_exception_handler(
            AppException("Unknown server error"))

    return app, db
```

FIGURE 14. Setting up Flask application utilizing an App Factory

The application is created inside main module (lines 6-9), which can either be invoked locally with a debugger (12-15) or served to Zappa for deployment. The function returning the app and database is configured to be called with local configuration or accept one from environment variables pointed to Zappa. The other part of the snippet describes the application packages root module that contains the App Factory function “create_app” (lines 23-66). Since the function resides in the root of the module and has all the application module imports inlined inside of it, sequencing the correct import order becomes trivial, ensuring that everything has been initialized if an app has been created. After the initialization is completed, both the application and database objects are being put to the module’s global context, ready to be served to any submodule sharing the application namespace. App Factory is also a good place to define general handlers such as how Flask treats specific errors (lines 42-64).

5.2.1 Zappa Configuration

As seen before, the configuration of Zappa is handled via zappa_settings.json -file. The example below (Figure 15) clarifies the configuration used for this work.



```
 1 {
 2   "base": {
 3     "app_function": "main.app",
 4     "manage_roles": false,
 5     "runtime":      "python3.6",
 6     "memory_size":  1536,
 7     "role_name":    "LambdaExecuteRole",
 8     "slim_handler": true
 9   },
10   "dev": {
11     "extends":    "base",
12     "log_level":  "DEBUG",
13     "s3_bucket":  "[redacted]",
14     "remote_env": "[redacted]
/config.json",
15     "events": [
16       {
17         "function":     "app.processors.photo_uploaded_handler",
18         "event_source": {
19           "arn":    "arn:aws:s3:::[redacted]",
20           "events": [ "s3:ObjectCreated:*" ]
21         }
22       }, {
23         "function":     "app.processors.photo_begin_process_handler",
24         "event_source": {
25           "arn":        "arn:aws:sqs:eu-west-1:[redacted]:photoUploaded",
26           "batch_size": 1,
27           "enabled":    true
28         }
29       }
30     ],
31     "vpc_config": {
32       "SubnetIds": [
33         "[redacted]",
34         "[redacted]",
35         "[redacted]"
36       ],
37       "SecurityGroupIds": [ "[redacted]" ]
38     }
39   },
40   "staging": {
41     "extends": "base",
42     ...
43   },
44   "prod": {
45     "extends": "base",
46     ...
47   }
48 }
```

Figure 15. The project's configuration file for Zappa

Since any professional project needs a separation for at least between the development and production environments, and sometimes even a third one for staging, it's a good idea to separate shared base configurations, which can be extended by each of the used stages. This way a developer avoids having multiple places of change when adjusting configurations that are common to each environment as the most common differences between stage variants are only about the AWS URIs or the desired log-level.

Besides being triggered by the calls to the AWS API Gateway, the configuration contains events that are tied to application logic intended to be invoked by changes in AWS resources. The lines 15-27 contain semantics for events after photo upload. The private subnets mentioned earlier are pointed to

Zappa to allow working network connections within the cloud, as well as any security groups that govern Lambda's behaviour.

5.3 Database Solutions

The database schema for the work is presented in the following diagram (Figure 16).

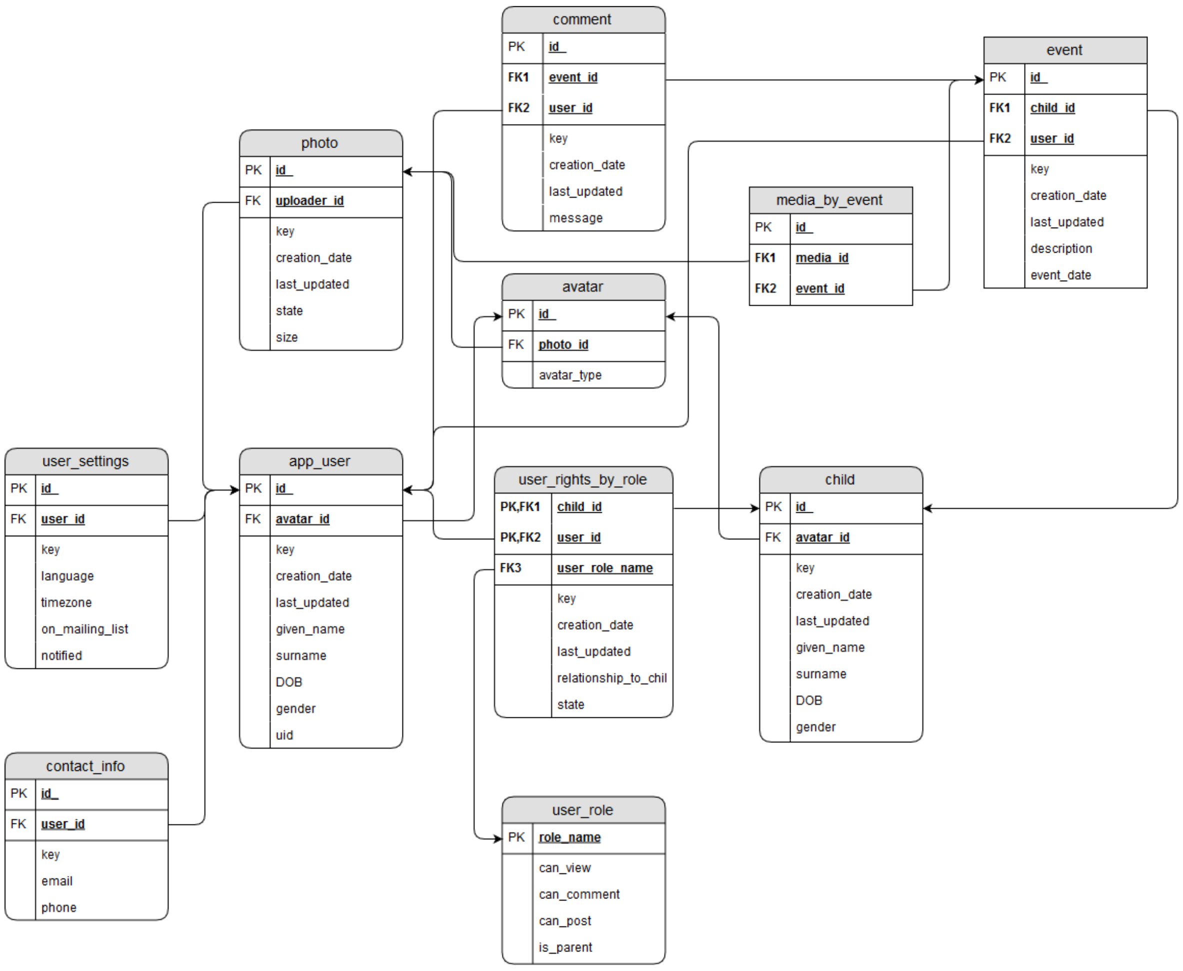
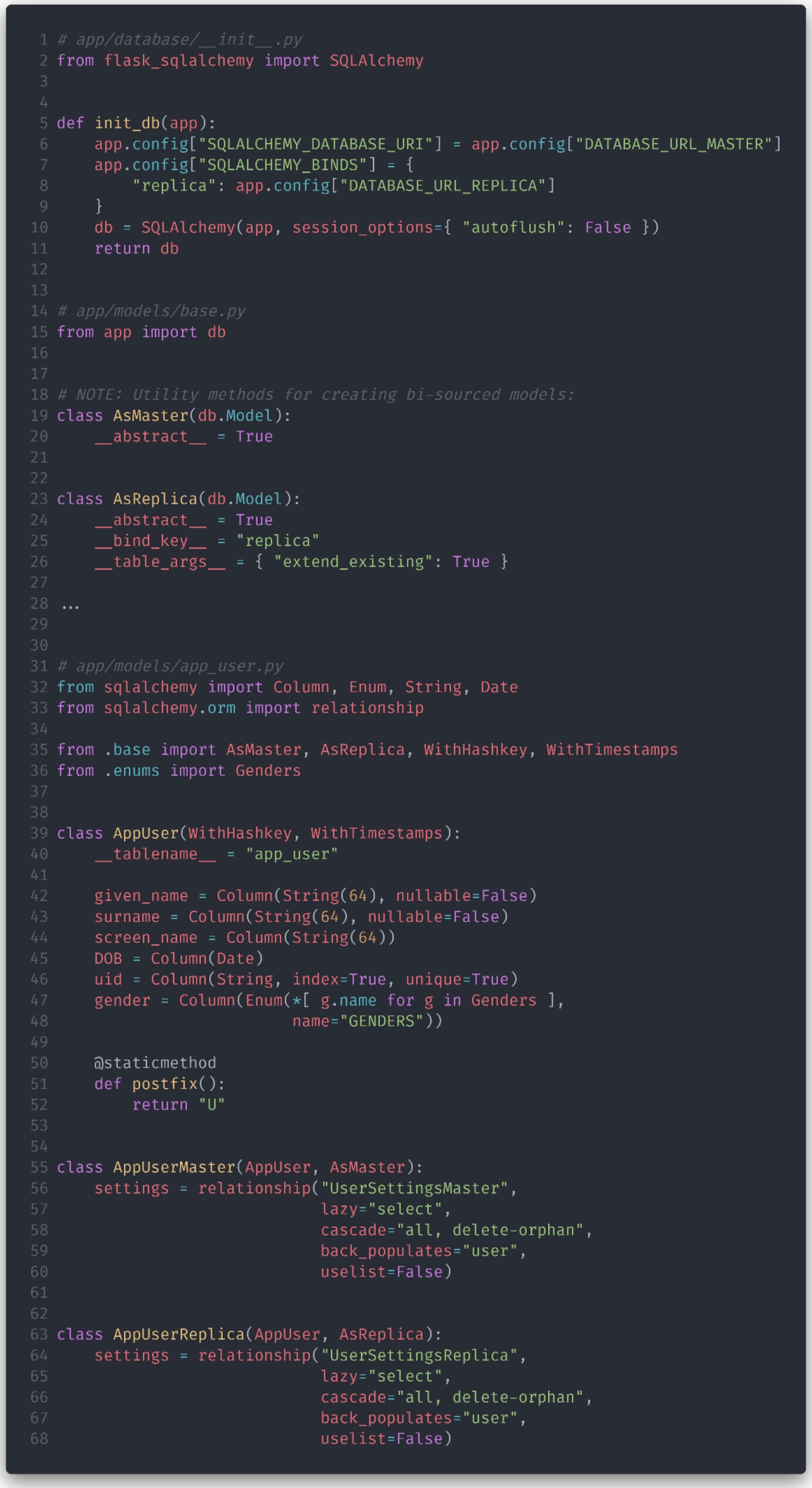


Figure 16. The database schema for the project

The connections to be emphasized in the above schema are the association table between users and children "user_rights_by_role", tied to a template table concerning configured privileges, named as "user role", as well as the bindings of photos used both in avatars and event media.

5.3.1 Master and Read Replica

The database traffic is designed to be offloaded with Master-Replica schema. This essentially means that all the transactions are done on Master and all reads, apart from few exceptions, are performed on Read Replica. To allow such behaviour, Flask exposes a configuration value to set multiple database targets, while SQLAlchemy allows configuring binds on models that specify which target is desired. The following snippet (Figure 17) elaborates this behaviour.



```python
# app/database/__init__.py
from flask_sqlalchemy import SQLAlchemy


def init_db(app):
    app.config["SQLALCHEMY_DATABASE_URI"] = app.config["DATABASE_URL_MASTER"]
    app.config["SQLALCHEMY_BINDS"] = {
        "replica": app.config["DATABASE_URL_REPLICA"]
    }
    db = SQLAlchemy(app, session_options={ "autoflush": False })
    return db


# app/models/base.py
from app import db


# NOTE: Utility methods for creating bi-sourced models:
class AsMaster(db.Model):
    __abstract__ = True


class AsReplica(db.Model):
    __abstract__ = True
    __bind_key__ = "replica"
    __table_args__ = { "extend_existing": True }

...


# app/models/app_user.py
from sqlalchemy import Column, Enum, String, Date
from sqlalchemy.orm import relationship

from .base import AsMaster, AsReplica, WithHashkey, WithTimestamps
from .enums import Genders


class AppUser(WithHashkey, WithTimestamps):
    __tablename__ = "app_user"

    given_name = Column(String(64), nullable=False)
    surname = Column(String(64), nullable=False)
    screen_name = Column(String(64))
    DOB = Column(Date)
    uid = Column(String, index=True, unique=True)
    gender = Column(Enum(*[ g.name for g in Genders ],
                     name="GENDERS"))

    @staticmethod
    def postfix():
        return "U"


class AppUserMaster(AppUser, AsMaster):
    settings = relationship("UserSettingsMaster",
                            lazy="select",
                            cascade="all, delete-orphan",
                            back_populates="user",
                            uselist=False)


class AppUserReplica(AppUser, AsReplica):
    settings = relationship("UserSettingsReplica",
                            lazy="select",
                            cascade="all, delete-orphan",
                            back_populates="user",
                            uselist=False)
```

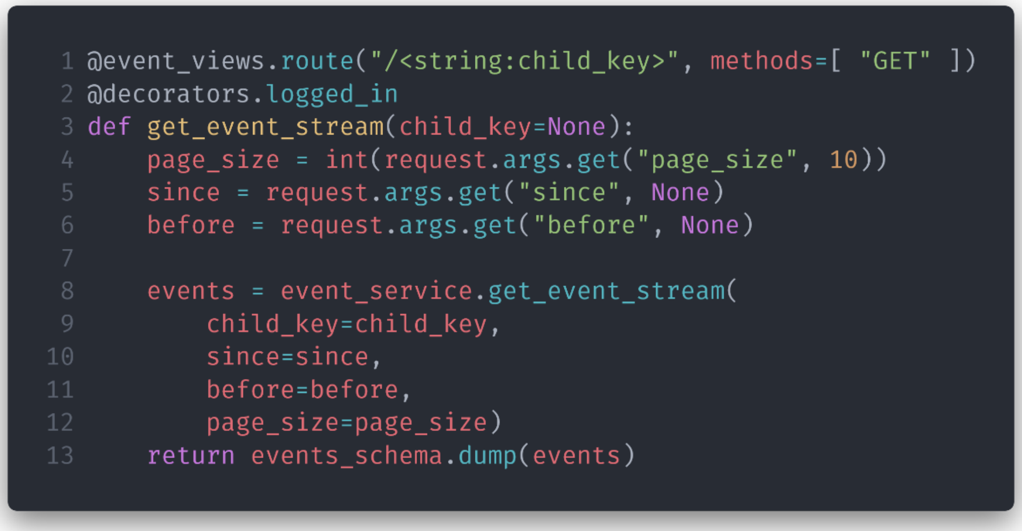
Figure 17. Structuring models for multiple target databases

At the beginning of snippet (lines 5-11) Flask application is instructed to point to the master with default database binding while the connection string to replica utilizes “SQLALCHEMY_BINDS” dictionary, which stores any alternative connections as key-value pairs. Then the module containing the models of the database instance is used for setting up class mixins that distinguish the target database at a model-level (lines 15-26). The same file contains definitions for more utility mixins, that are left out for brevity. Finally, in the file defining an actual model for a database table, the previously mentioned mixins are used. The code is structured in a manner, where the shared info such as table name and column names are defined once as a template model, which does not invoke a creation of an actual table (lines 39-52). This is then used by both models to avoid unnecessary repetition. Both models need to be configured separately for their Python level relationships, as residents in master are only connected to other entities in master database while the ones in replica are connected to their own (lines 55-63).

5.4 Paginating Resources

The pagination refers to chunking requested data into limited and easily digestible slices. Assuming a feed or any requested resource has business objects reaching up to hundreds of items, it’s not palatable to expect a client to fetch them at once. This is where pagination can assist and since most of the fetchable resources are hash-keyed, a schema operating in chunks is matter of generating right kind of combination of endpoints and queries.

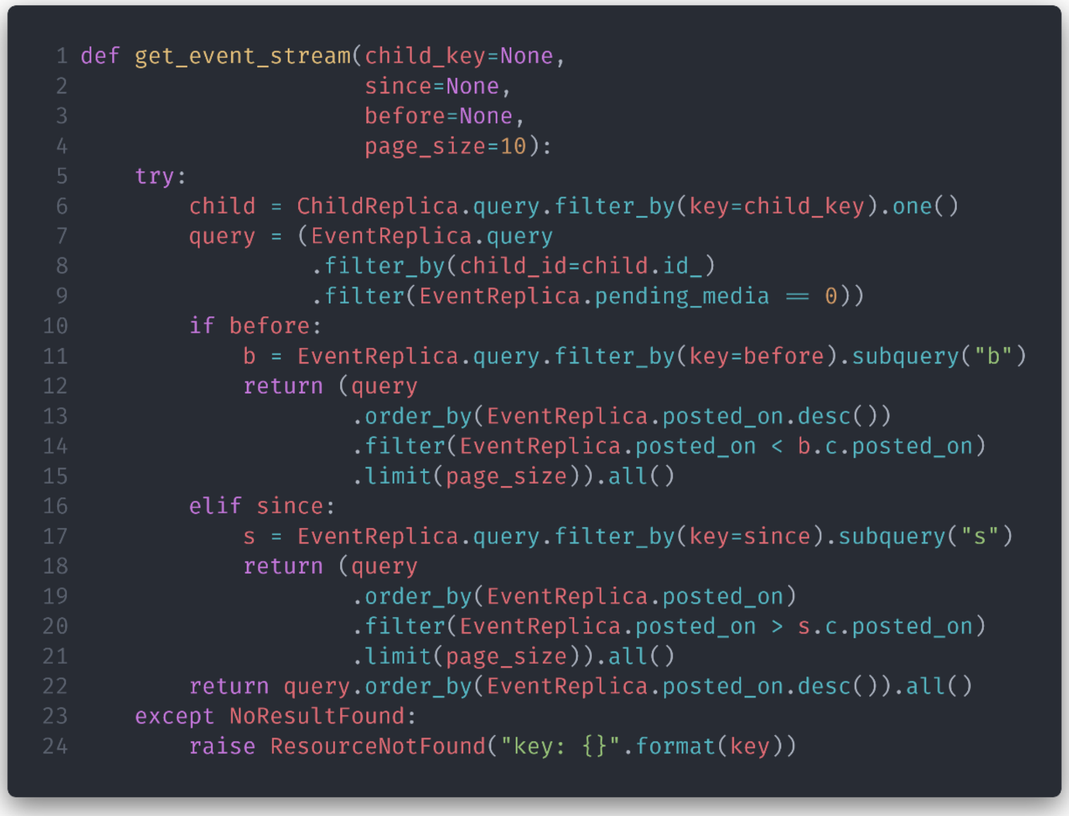
Below is a snippet (Figure 18) describing how the endpoint for fetching the event feed is structured to enable pagination. The API route expects the requester to add child’s public key to the GET-request’s URL path (lines 1-3). This will be used to identify which of children’s events the requester is interested in. The request also accepts query parameters controlling the fetched page size and takes additional event keys to decipher the current location in feed, as well as the direction of the requested fetch (lines 4-5). After parsing the needed parameters, these are forwarded to the corresponding function in the service-layer (lines 8-13).



```python
@event_views.route("/<string:child_key>", methods=[ "GET" ])
@decorators.logged_in
def get_event_stream(child_key=None):
    page_size = int(request.args.get("page_size", 10))
    since = request.args.get("since", None)
    before = request.args.get("before", None)

    events = event_service.get_event_stream(
        child_key=child_key,
        since=since,
        before=before,
        page_size=page_size)
    return events_schema.dump(events)
```

Figure 18. The endpoint structure using pagination

Besides endpoint passing along the needed parameters, a query needs to be built using specific keys. An example of such query is shown in the snippet below (Figure 19). Since the routine only handles reading data and doesn't modify anything, only replica -versions of the models can be used. At the beginning of the routine a corresponding child is queried to enable failing fast if no child with given key is present at the database (line 6). Utilizing a try catch a proper error response can be forwarded from Flask. If the query succeeds, however, a routine continues parsing a first layer of the query to be used. It is to be noted that no database queries are run at this point, as only proper statements are constructed (lines 7-9). The next steps are dependent on the given query parameters. If "before" argument was given it will attempt to construct a query comparing the timestamps to an event found with the given key, and only events published before the compared are to be fetched with the amount of results limited to given page size (lines 11-15). In the used structure the "before" key takes precedence over "since" key. If "since" is detected without "before" key, the constructed query will have the opposite effect, attempting to query events preceding the given event key (lines 17-21). If neither is given the query still supports loading all the events for the rare cases this would be desired (lines 22).



```python
def get_event_stream(child_key=None,
                     since=None,
                     before=None,
                     page_size=10):
    try:
        child = ChildReplica.query.filter_by(key=child_key).one()
        query = (EventReplica.query
                 .filter_by(child_id=child.id_)
                 .filter(EventReplica.pending_media == 0))
        if before:
            b = EventReplica.query.filter_by(key=before).subquery("b")
            return (query
                    .order_by(EventReplica.posted_on.desc())
                    .filter(EventReplica.posted_on < b.c.posted_on)
                    .limit(page_size)).all()
        elif since:
            s = EventReplica.query.filter_by(key=since).subquery("s")
            return (query
                    .order_by(EventReplica.posted_on)
                    .filter(EventReplica.posted_on > s.c.posted_on)
                    .limit(page_size)).all()
        return query.order_by(EventReplica.posted_on.desc()).all()
    except NoResultFound:
        raise ResourceNotFound("key: {}".format(key))
```

Figure 19. Querying information with pagination

5.5 Handling Media

The uploading of media is implemented with S3, explicit lambda handlers and SQS for handling retries and providing queue to balance and synchronize the ongoing processing. The sequence diagram below (Figure 20) explains the semantics of an image upload process with all related components including the client application.

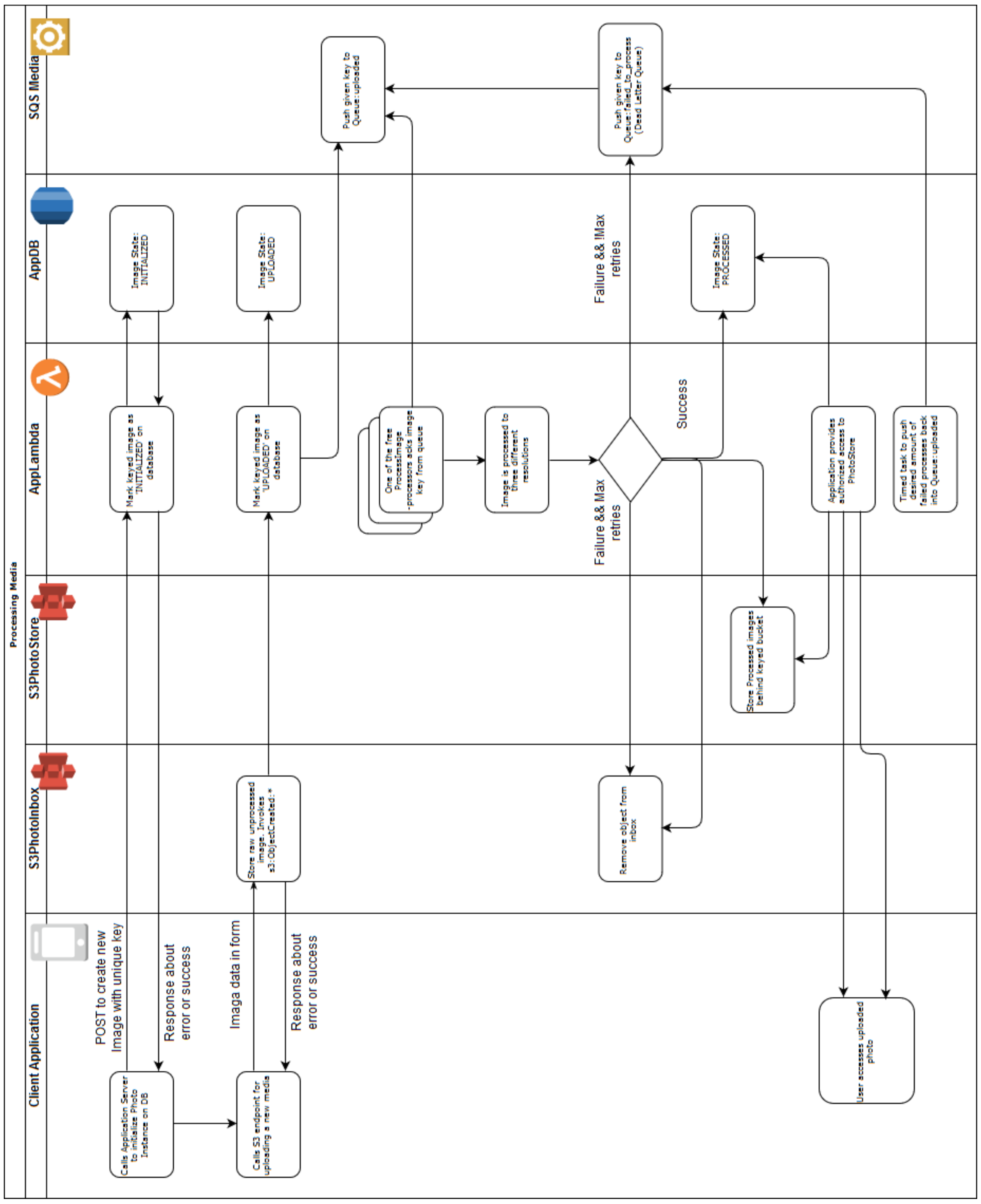

Processing Media
Client Application
S3PhotoInbox
S3PhotoStore
AppLambda
AppDB
SQS Media
Calls Application Server to initialize Photo Instance on DB
POST to create new Image with unique key
Response about error or success
Mark keyed image as "INITIALIZED" on database
Image State: INITIALIZED
Calls S3 endpoint for uploading a new media
Imaga data in form
Response about error or success
Store raw unprocessed image. Invokes s3:ObjectCreated:*
Mark keyed image as "UPLOADED" on database
Image State: UPLOADED
Push given key to Queue:uploaded
One of the free ProcessImage -processors acks image key from queue
Image is processed to three different resolutions
Failure && Max retries
Failure && !Max retries
Success
Push given key to Queue:failed_to_process (Dead Letter Queue)
Image State: PROCESSED
Store Processed images behind keyed bucket
Remove object from inbox
Application provides authorized access to PhotoStore
Timed task to push desired amount of failed processes back into Queue:uploaded
User accesses uploaded photo

Figure 20. The image upload process

At the beginning of the process the client application needs to instantiate a photo to the database and create a unique hash-key to be used both by the client application and the S3 buckets. This also sets the database instances state to “INITIALIZED”. After a successful initialization the client gets the upload URI and the generated key for the inbox bucket. If the upload succeeds the inbox bucket signals an event about new object, which in turn notifies the App Lambda to mark the photo to be “UPLOADED” and pushes this to the AWS Standard Queue to be processed. The processor handlers acknowledge the messages pushed in to the queue and begin processing the required versions of the picture. At the time of failure, the message is pushed to the dead letter queue to await a retry and a count for retries is incremented. If the count exceeds the specified limit, the process is discarded, and the object is to be purged both from the S3 inbox and database. On success the image state is marked as “PROCESSED” and the corresponding picture is stored into a photo store bucket, where the application can forward the requested images from.

6 CONCLUSIONS

The aim of the work was to provide an initial design and development for proof of concept backend service. The service was to be used in conjunction with a social media app primarily used in mobile phones. The design needed to accommodate the needs of a growing userbase and provide an easily scalable and affordable architectural basis for the solution.

The process was implemented on top of Amazon Web Services (AWS) due to its well-documented nature and wide scale of integrable solutions. The on-demand nature of AWS costs was a good fit for a starting company's needs. The source code was largely dominated by Python-based libraries, which proved mostly mature and well adjusted to their tasks.

The result of the work is a working proof of concept that serves as basis for further backend application development. Due to its adjustable and modular nature, the work served well both in the assignee's case, as well as a template in future endeavours with adjustments required by their problem-domain. The application structure has proven to be easily testable and integrable into automated deployment. The future versions will see changes into a database model supporting feeds with multiple children, as well as videos added into the possible media types. Increasing user acquisition and responsivity is to be achieved by integrating notifications and email services.

The biggest challenges in the work were due to the nature of open source libraries themselves. Some parts of the used dependencies had been left undocumented while some of the necessary updates introduced breaking changes. In general, these were usually fixed and taken care of within one week.

REFERENCES

AWS 2019a. Amazon Simple Queue Service [accessed 28.3.2019]. Available at: https://aws.amazon.com/sqs/

AWS 2019b. About AWS [accessed 23.3.2019]. Available at: https://aws.amazon.com/about-aws/

AWS 2019c. Amazon API Gateway [accessed 24.3.2019]. Available at: https://aws.amazon.com/api-gateway/

AWS 2019d. EC2 [accessed 24.3.2019]. Available at: https://aws.amazon.com/ec2/

AWS 2019e. Amazon EC2 Pricing [accessed 24.3.2019]. Available at: https://aws.amazon.com/ec2/pricing/

AWS 2019f. Amazon RDS Reserved Instances [accessed 25.3.2019]. Available at: https://aws.amazon.com/rds/reserved-instances/

AWS 2019g. Amazon Relational Database Service (RDS) [accessed 25.3.2019]. Available at: https://aws.amazon.com/rds/

AWS 2019h. AWS Global Infrastructure [accessed 23.3.2019]. Available at: https://aws.amazon.com/about-aws/global-infrastructure/

AWS 2019i. AWS Lambda Limits [accessed 23.3.2019]. Available at: https://docs.aws.amazon.com/lambda/latest/dg/limits.html

AWS 2019j. AWS Lambda Pricing [accessed 23.3.2019]. Available at: https://aws.amazon.com/lambda/pricing/

AWS 2019k. AWS S3 [accessed 24.3.2019]. Available at: https://aws.amazon.com/s3/

AWS 2019l. AWS S3 Pricing [accessed 24.3.2019]. Available at: https://aws.amazon.com/s3/pricing/

AWS 2019m. Basic AWS Lambda Function Configuration [accessed 23.3.2019]. Available at: https://docs.aws.amazon.com/lambda/latest/dg/resource-model.html

AWS 2019n. Cloud Products [accessed 23.3.2019]. Available at: https://aws.amazon.com/products/

AWS 2019o. Elastic IP Addresses [accessed 24.3.2019]. Available at: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/elastic-ip-addresses-eip.html

AWS 2019p. Lambda [accessed 23.3.2019]. Available at: https://aws.amazon.com/lambda/

AWS 2019q. Serverless [accessed 23.3.2019]. Available at: https://aws.amazon.com/serverless/

AWS 2019r. What is Cloud Computing [accessed 23.3.2019]. Available at: https://aws.amazon.com/what-is-cloud-computing/

Flask 2019. Flask [accessed 28.3.2019]. Available at: http://flask.pocoo.org/

Flask-SQLAlchemy 2019. Flask [accessed 29.3.2019]. Available at: http://flask-sqlalchemy.pocoo.org/2.3/

Marshmallow 2019. Marshmallow [accessed 29.3.2019]. Available at: https://marshmallow.readthedocs.io/en/3.0/

Myers, J. & Copeland, R. 2016. Essential SQLAlchemy. United States of America: O'Reilly Media

SQLAlchemy 2019a. Dialects [accessed 28.3.2019]. Available at: https://docs.sqlalchemy.org/en/latest/dialects/index.html

Zappa 2019a Zappa [accessed 28.3.2019]. Available at: https://www.zappa.io/

Zappa 2019b Zappa GitHub [accessed 28.3.2019]. Available at: https://github.com/Miserlou/Zappa