DevOps Level 1 course

Final project documentation

Parengė: Mantas Andriekus

Kaunas, 2024

Table of Contents

1. Task

To create application useful for users to control their to-do list. Each user can access and modify his own to-do list. Application should be created using new technologies. Application type – web based.

1.1. Application usability

Application will be used for big group of people simultaneously. First application will be used as browser based, but in future will be developed mobile application. For this reason, system must provide data for mobile applications.

1.1.1. Users

User will use system to control to-do list. User actions:

- Register using personal data;
- Login to application;
- Add new to-do;
- Confirm that to-do is done;
- Remove to-do;

1.1.2. To-do list

To-do list should be represented clearly. To-do list should contain information:

- To-do information (description text);
- Creation date;
- Completion date;
- To-do is completed or not;

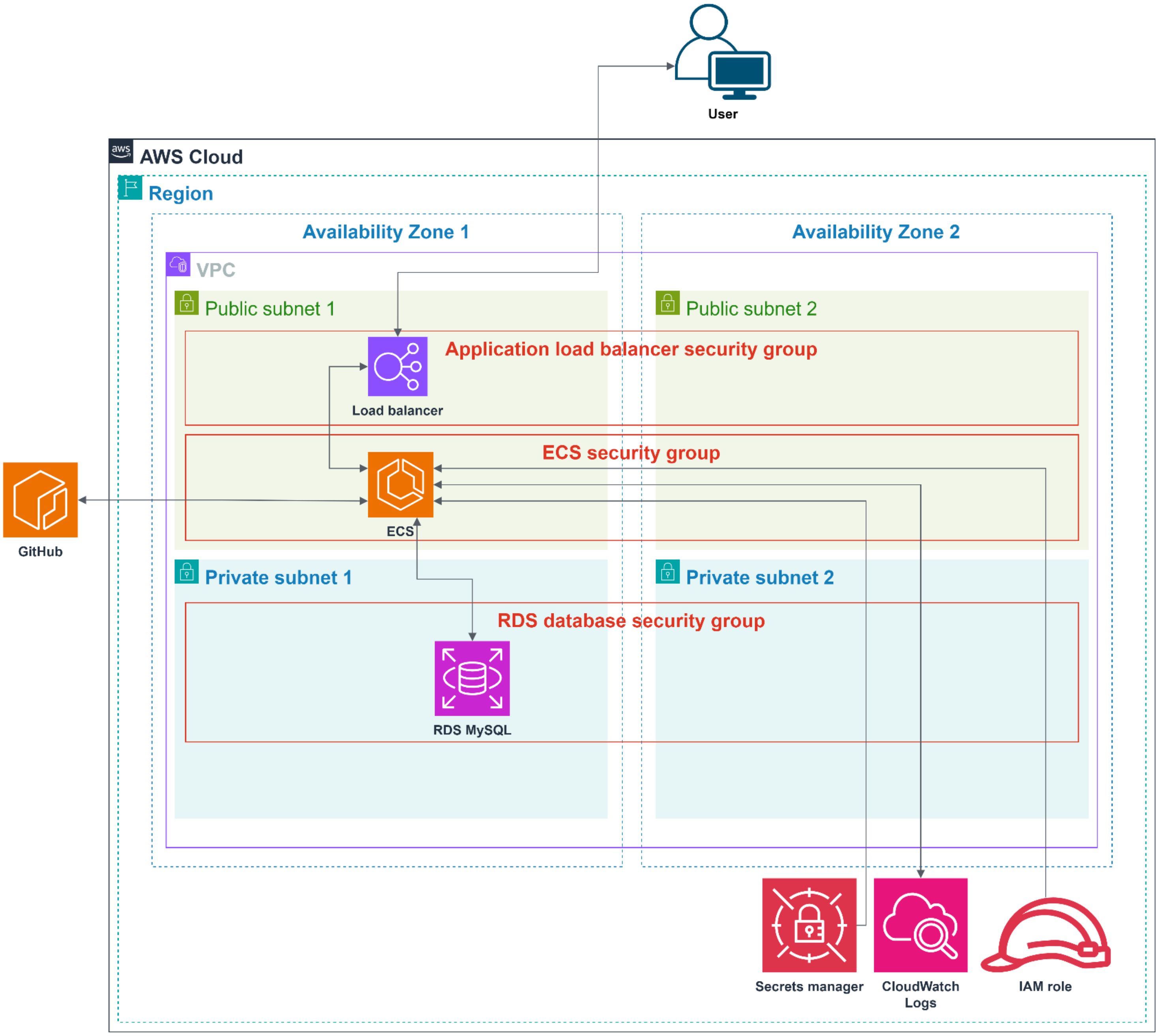
1.2. Technical requirements

Application must be deployable in could environment. Brief list of requirements:

- Deployable in cloud (AWS) environment;
- Infrastructure written in code;
- RDS MySQL database. Database disabled form access from outside;
- Application must be containerized;
- Application must be cost effective;
- Application must be written in Python programming language;
- Application must use Flask framework;
- Application and infrastructure code must have different repositories.
- Code must be versioned using version control system git;

2. Infrastructure

As required, infrastructure will be deployed to AWS environment. Infrastructure will be realized as IaC (infrastructure as a code) using Terraform and Terraform AWS provider library. For code versioning will be used git version control system and github.com repository. Using the given requirements, the architecture of the resources needed to complete the task was created:



According this block diagram resources will be created.

2.1. Virtual private cloud

For virtual cloud (VPC) creation was made a module named: my_vpc. This module automatically creates:

- VPC itself;
- Public subnets;
- Private subnets;
- Assigns subnet to availability zone;
- Internet gateway;
- Route table;
- Subnets (private) group for database;

Module variables:

Variable name	Type	Definition
Inputs		
vpc_name	string	Name of VPC.
vpc_cidr_range	string	CIDR block for VPC.
public_subnets	list(string)	List of subnets to create inside VPC with internet accessibility.
private_subnets	list(string)	List of subnets to create inside VPC without internet accessibility.
azs	list(string)	List of availability zones to be used.
Outputs		
public_ids	-	Public subnet-(s) id-(s).
private_ids	-	Private subnet-(s) id-(s).
vpc_id	-	VPC id.
db_subnet_gr_name	-	Subnets group name. Will be used for database creation.

When using my_vpc module, you need to assign input variables (mentioned above). Module outputs variables for usage in other configuration parts.

Public subnets used for resources with accessibility from outside the cloud. Private subnets used to restrict accessibility from outside.

2.2. Security group

Security group works like firewall. Security group blocks incoming and outgoing traffic. For this project to complete, needs three security groups:

- Security group for application load balancer (ALB);
- Security group for elastic container service (ECS);
- Security group for relational database service (RDS);

Created module for security group. Module variables:

Variable name	Type	Definition
Inputs		
vpc_id	string	VPC id in which security group works.
environment_name	string	Environment name to tag resource.
Outputs		
alb_sg_id	-	Application load balancer security group id. Used in load balancer.
ecs_sg_id	-	Elastic container service security group id. Used in ECS.
db_sg_id	-	Database security group. Used for database.

2.3. Logs

Using native CloudWatch for logging. CloudWatch will log containers stdout. Creating log group to combine logs and log stream. In task definition file we only register log group.

2.4. Secrets manager

Secrets manager is used to protect sensitive data. Secrets managers ensures safe sensitive data provision to resources. Secrets, for this project, will be created manually by AWS cloud administrator:

- **db_creds** – this secrete used to connect to database. Secret properties:
  - db_username;
  - db_password;
  - db_host;
  - db_name;
- **registry_creds** – this secret used to connect to docker hub registry. Secret properties:
  - username;
  - password;

2.5. IAM policies

Changed “ecsTaskExecutionRole” role policies. Appended two policies:

- Read secrets from secrets manager;
- Log stdout to CloudWatch log group.

2.6. Database

Using AWS RDS. Selected database – MySQL. Database uses private subnets group to restrict accessibility from outside. Secrets for database is used from secrets manager. Security group defined by ***db_sg_id*** [1].

2.7. Application load balancer

Application load balancer used to ensure ECS tasks stability and reduce response time. Application load balancer divides incoming traffic to ECS tasks. This way the response time is as minimum as possible.

Load balancer uses security group and public subnets.

Load balancer has target group and listener. Listener are responsible to catch configured traffic and redirect it to target group. Target group using configuration redirects traffic to targets.

2.8. Elastic container service

Elastic container service ensures that tasks (container) will be created and started. ECS monitors task parameters and if needed takes actions to ensure stability. ECS can reload tasks if current state is “unhealthy”, can auto-scale to increase or decrease tasks number. This service decided to use, because it takes a lot of work for us and is cost efficient (pay-as-you-go).

To activate ECS, resources was created:

- Cluster – cluster used for services. One cluster could have more than one service;
- Service – service lives in cluster. Service is used to serve container. In service configuration is set network configuration, assigned load balancer. It is like sand box for tasks. Service could have one or more than one task.
- Task definition – task definitions define the task capabilities (cpu, memory, network mode). In task definition also configures container image properties.

[1] 2.2. Security group

2.9. Pipeline

Using Git Hub repository as version control system. Pipeline is written using Git Hub Actions. Pipeline file location: *./.github/workflows/main.yml,* To accomplish tasks, pipeline has jobs and steps.

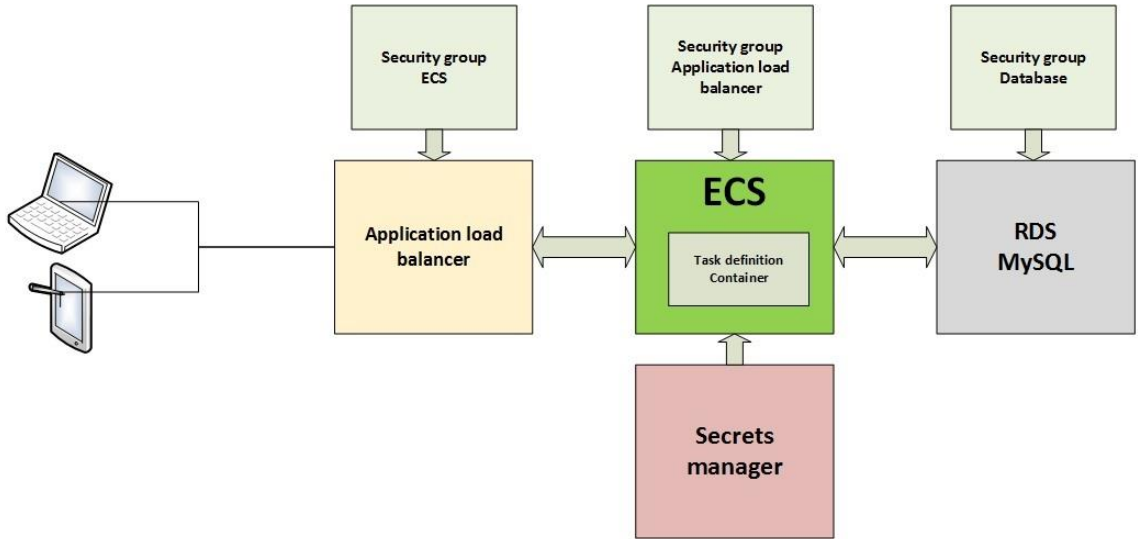
2.9.1. Initialize infrastructure

This job checkouts repository. Initialize Terraform, formats code and apply it to cloud. Job description:

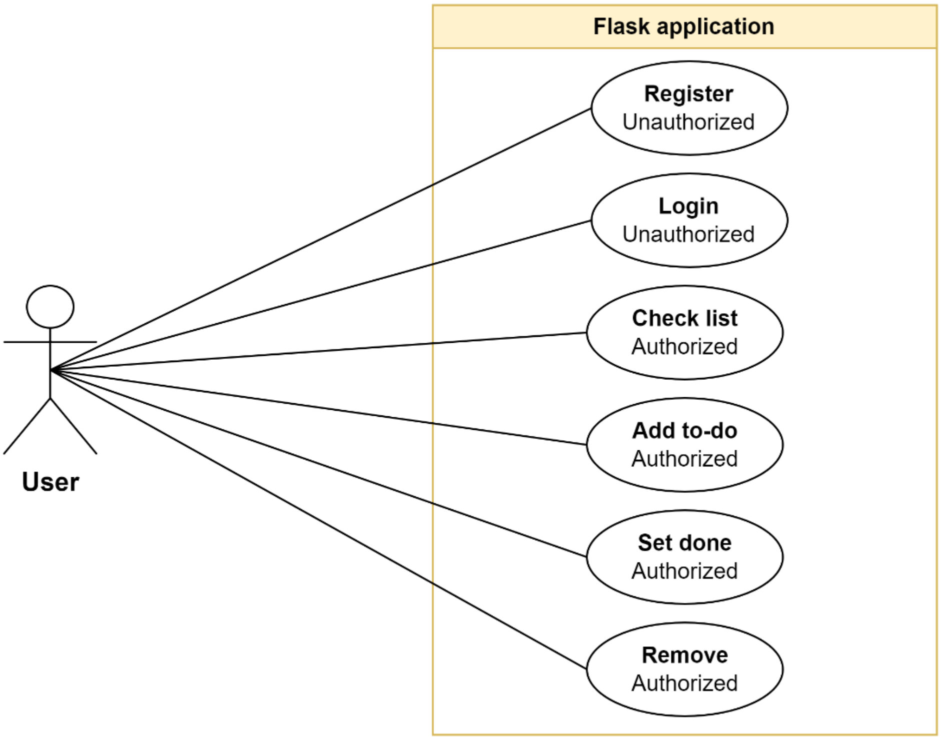
```
initialize-infra:
  name: 'Init. infrastructure'
  runs-on: ubuntu-latest
  env:
    AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY }}
    AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_KEY }}
  steps:
    - name: 'Checkout repository'
      uses: actions/checkout@v4

    - name: 'Setup Terraform'
      uses: hashicorp/setup-terraform@v2

    - name: 'Init. Terraform'
      working-directory: ./environments/staging
      run: |
        terraform fmt
        terraform init

    - name: 'Apply infrastructure'
      working-directory: ./environments/staging
      run: terraform apply-auto-approve
```

3. Web application

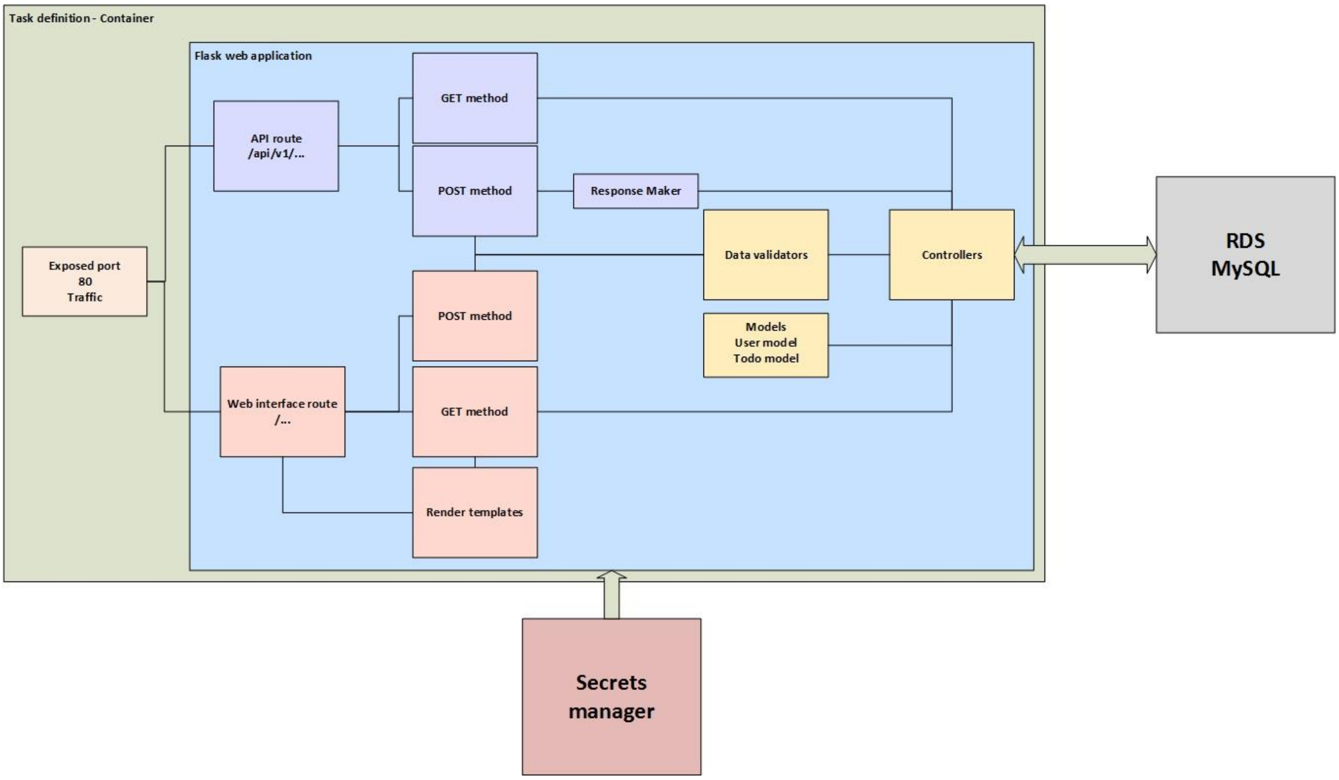
Web application created using Python programming language and native Flask framework. Web application have user interface and API. User interface used for browser usage. API will be used in future for mobile applications. Workflow diagram of application:



Interaction with system described in use-case diagram:



Flask application block diagram:



3.1. Front-end development

Web user interface is used for users to interact with system using browser. Pages user is able to access:

- Index
- Login
- Register

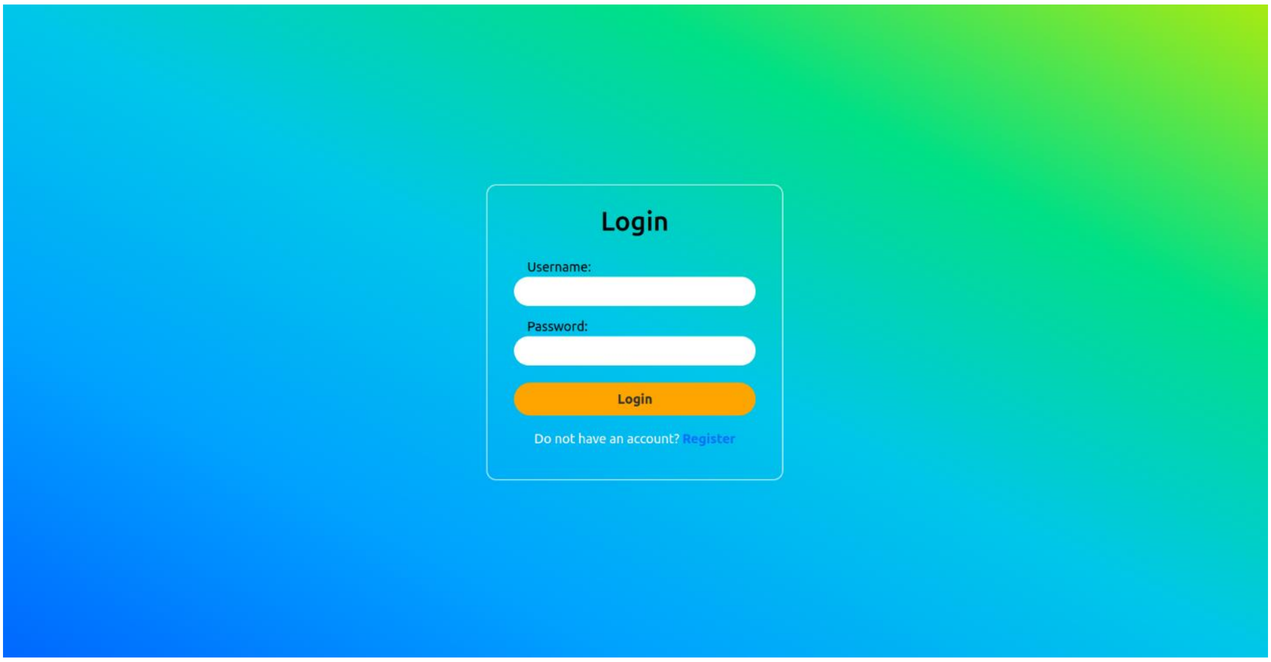
Pages to look similar, created layout.html. This layout used in all pages to not repeat common code. Pages and layout files are located in ***templates*** directory. Style (.css) and JavaScript (.js) files are located in ***static*** directory.

Unregistered users are able to access login and registration pages. If user is not logged in, system always redirects to login page.

3.1.1. Login page

Login page is a root page. This page is used to let user log in to system. All fields are required. After successful data validation user will be redirect to Index page. If user does not have an account, he can register. Registration page activates, when user registration link is activated.

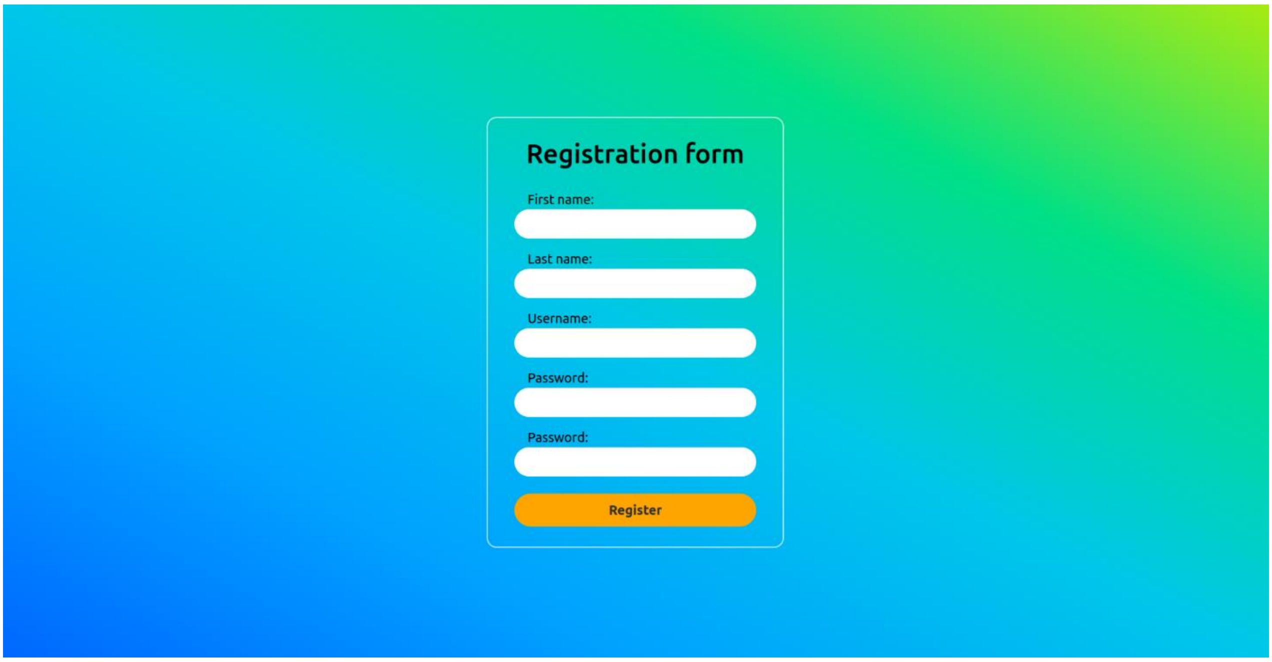
If data is not valid, system will show error messages in top of the page.



3.1.2. Registration page

Registration page lets new user to fill required fields. All fields are required. If given data is valid then user sensitive information is encrypted and saved to database. After successful registration, user will be redirected to login page.

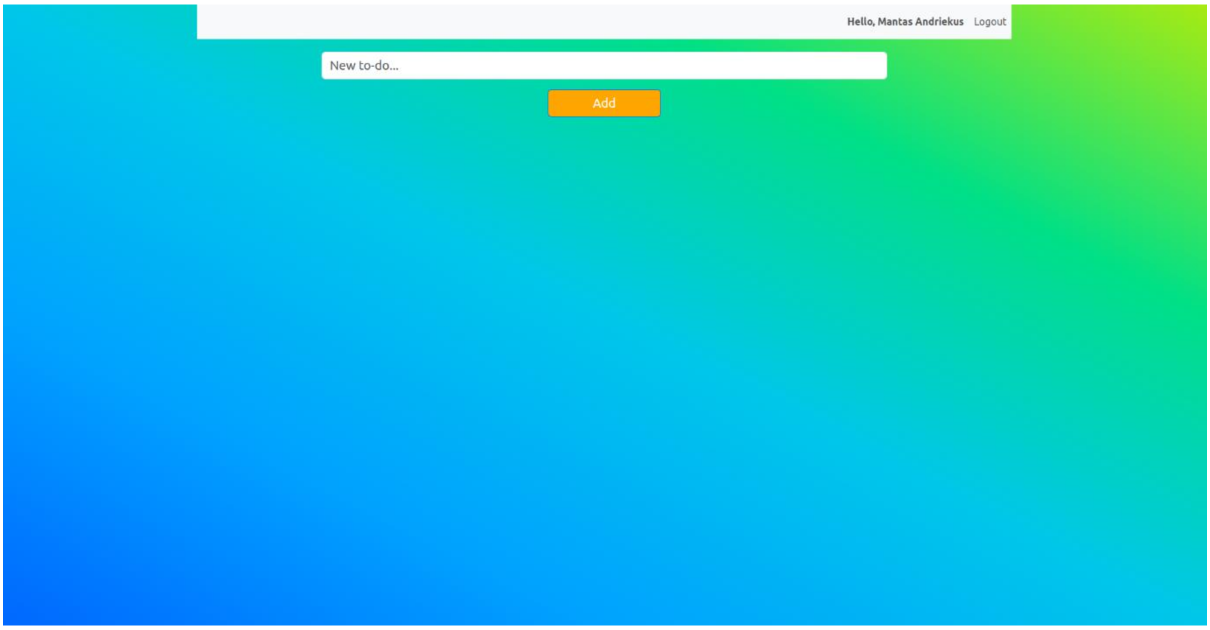
If data is not valid, system will show error messages in top of the page.



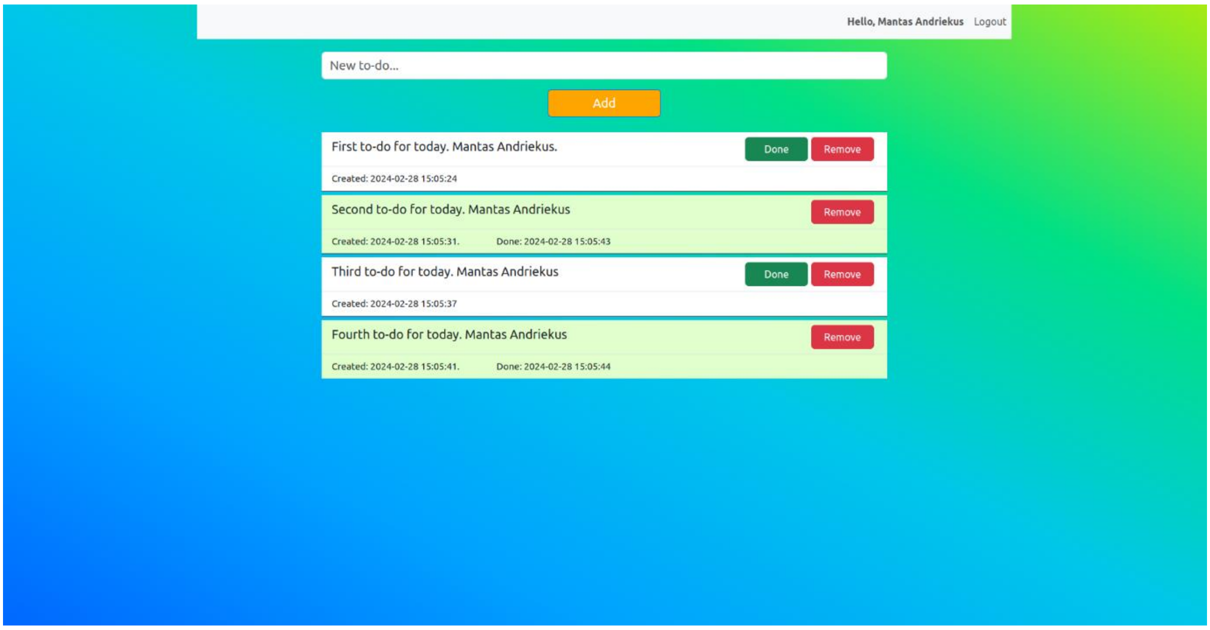
3.1.3. Index page

Index page can access only registered and logged in users. In this page user actions:

- Check to-do list;
- Add new to-do to list;
- Set to-do as done;
- Remove to-do from the list;



Every user registered to the system are able to access only their own to-do list.



First to-do for today. Mantas Andriekus.

Created: 2024-02-28 15:05:24

Second to-do for today. Mantas Andriekus

Created: 2024-02-28 15:05:31. Done: 2024-02-28 15:05:43

Third to-do for today. Mantas Andriekus

Created: 2024-02-28 15:05:37

Fourth to-do for today. Mantas Andriekus

Created: 2024-02-28 15:05:41. Done: 2024-02-28 15:05:44

3.2. Back-end development

Back-end is written in Python programming language. Program written using block diagram (***Flask application block diagram***).

3.2.1. Routes

Routes in flask is used to control traffic flow programmatically. Two routes are set:

- api – used to control request and send respond to api clients;
- views – used to control request and send respond to web user interface;

To separate routes using flask module Blueprints. This way we can easily separate routes using blueprint name. Blueprints registers to flask application. Methods (routes) are decorated with blueprint name and parameters:

```
@views.route(“/”, methods = [“GET”, “POST”])
```

- @views.route – method decorator for routing;
- “/” – URL path;
- methods = [“GET”, “POST”] – methods to response;

Routes have more decorators discussed deeply in 3.2.2., 3.2.3. sections.

3.2.2. Views blueprint

This blueprint redirects traffic to methods requested as views route. Method selection uses URL described in decorator.

3.2.2.1. Index

This route is used to show to-do list and execute action with it. This route renders template ***index.html***. Route description:

```
@views.route("/", methods = ["GET", "POST"])
@login_required
def index():
```

@login_required – decorator to tell flask application, that this method could be executed only with authorized users. Unauthorized users cannot access this page.

3.2.2.2. Log in

This route is used to render page from template ***login.html*** for user to log in. Route description:

```
@views.route("/login", methods = ["GET", "POST"])
def login():
```

3.2.2.3. Log out

This route is used for user to log out. This route does not render any templates. Route description:

```
@views.route("/logout")
@login_required
def logout():
```

This route is accessible for authorized users.

3.2.2.4. User registration

This route is used to render page from template ***register.html*** for user to register. Route description:

```
@views.route("/register", methods = ["GET", "POST"])
def register():
```

3.2.2.5. Remove to-do

This route is used for user to remove to-do from list by to-do id. This route does not render any templates. Route description:

```
@views.route("/remove-todo", methods=["POST"])
def remove_todo():
```

3.2.2.6. Update to-do

This route is used for user to update to-do. Basically, the update means “Set to-do as done”. This route does not render any templates. Route description:

```
@views.route("/update-todo", methods=["POST"])
def update_todo():
```

3.2.3. API blueprint

This blueprint redirects traffic to methods requested as api route. Method selection uses URL described in decorator.

To access data using API, user must be authenticated. Blueprint have method (decorator) to check user identity. Methods decorated with this method, only can be access with JWT token.

3.2.3.1. Token validation

Token validation method used to validate token generated in log in. Using JWT token. This method wraps validation function and return error or function (user tried to access) with user data. Method header:

```
def token_validation(func):
   """ method to wrap/decorate routes """
   @wraps(func)
   def decorated(*args, **kwargs):
```

3.2.3.2. User registration

This route is used to register user using *API*. As data format, method accepts *JSON* data format. Using validators module, data is verified. If user data is acceptable, user ir registered to the system. Using credentials user used to register, user can log in. Method header:

```
@api.route("/api/v1/register_user", methods = ["POST"])
def register_user():
```

3.2.3.3. Log in

This route is used to authenticate user data and generate a token. As data format, method accepts *JSON* data format. If user provided data is valid, method will generate *JWT* token and send it back to application requested. Otherwise, method returns error message and *HTTP* code. Method header:

```
@api.route("/api/v1/login", methods = ["POST"])
def login():
```

3.2.3.4. Add to-do

This route is used add new to-do item to list. As data format, method accepts *JSON* data format. Add to-do method is accesable only for authenticated users. User (application) must have valid *JWT* token. Adding new to-do to database, to-do have had user id. Using *@token_validation*, user information is passed to method. Method header:

```
@api.route("/api/v1/add_todo", methods = ["POST"])
@token_validation
def add_todo(user_by_token):
```

3.2.3.5. Get to-to list

This route is used to get all items in to-do list by user id. Method return data in *JSON* data format. Get to-do list method is accesable only for authenticated users. User (application) must have valid *JWT* token. Method header:

```
@api.route("/api/v1/todo_list", methods = ["GET"])
@token_validation
def get_todo_list(user_by_token):
```

3.2.3.6. Update to-do

This route is used to update to-do item in database by item id. As data format, method accepts *JSON* data format. Method return data in *JSON* data format. Update to-do item method is accesable only for authenticated users. User (application) must have valid *JWT* token. Method header:

```
@api.route("/api/v1/update_todo_status", methods = ["POST"])
@token_validation
def update_todo_status(user_by_token):
```

3.2.3.7. Remove to-do

This route is used to remove to-do item from database by item id. As data format, method accepts *JSON* data format. Method return data in *JSON* data format. Update to-do item method is accesable only for authenticated users. User (application) must have valid *JWT* token. Method header:

```
@api.route("/api/v1/remove_todo", methods = ["POST"])
@token_validation
def remove_todo(user_by_token)
```

3.2.4. Validators

Module *validators* is used to validate data user provides. Custom validator has been written to validate data from web UI and API. Having custom validation module in future will be easier to add validation patterns. Validator validates:

- Registration data;

  ```
  def validate_registration_data(self, user):
  ```

- Log in data;

  ```
  def validate_login_data(self, username, password):
  ```

Validator class have private (encapsulated) methods:

- def __validate_str_input(self, input_data, min, max, message): - validates string inputs. Method parameters:
  - input_data – data to be validated;
  - min – minimum characters set;
  - max – maximum characters set;
  - message – message to be joined to error message;
- __validate_username(self, username, message): - validates user name. In database user name is unique. Method parameters:
  - username – provided user name;
  - messaage - message to be joined to error message;
- def __validate_password(self, password, password_to_match, message): - validates password. Checks if passwords match. Method parameters:
  - password – password provided;
  - password_to_match – password provided to match first password;
  - messaage - message to be joined to error message;

3.2.5. Controllers

This module is used to perform actions between database and API or web UI. Two controllers used:

- User controller – performs actions with database to user model. Module methods:
    - register_user(user_info): - gets data (valid data) from *api* or *views* and saves it in database. User password is hashed using *bcrypt* module;
    - login_user(username, password): - gets data (valid data) from *api* or *views*. Gets user data from database and returns to requester;
    - def get_user_by_username(username): - gets user data (if user exists) from database and returns to requester;
- To-do controller – performs actions with database to todo model. Module methods:
    - def add_todo(description, user_id): - gets data (valid data) from *api* or *views* blueprints and saves it in database;
    - def remove_todo(id): - removes to-do item from database by id (if exists);
    - def update_todo(id): - updates to-do item data in database. To-do item identified by its id;
    - def get_todo_list(user_id): - gets all to-do items saved in database by user (requester) id;

3.2.6. Models

Models is used to execute action with data in objects-oriented programming level. All models have method “to_json(self)”. This method helps to convert Python class object to *JSON* data format. Two models have been created.

3.2.6.1. User

This model has all parameters to hold data about system users. Model description:

```
class User(db.Model, UserMixin):
    """ User model. Object to create and make manipulations with database """
    # table name in database
    __tablename__ = "users"

    # id- user id as primary key
    id = db.Column("id", Integer, primary_key=True, nullable=False)
    # public id- used to hide real users number, or other relevant data about database
    public_id = db.Column("public_id", String(100), unique=True, nullable=False)
    # users first name
    first_name = db.Column("firstname", String(50))
    # users last name
```

```
last_name = db.Column("lastname", String(50))
# users username (login). Must be unique.
username = db.Column("username", String(100), unique=True)
# password- hashed
password = db.Column("password", String(200))
# relationship with Todo table
todo_list = relationship("Todo")

def __init__(self, public_id,firs_name,
            last_name, username, password):
    """ Initialize model """

    self.public_id = public_id
    self.first_name = firs_name
    self.last_name = last_name
    self.username = username
    self.password = password

def to_json(self):
    """ method used to convert model data to json """
    return {
            "id": self.id,
            "public_id": self.public_id,
            "first_name": self.first_name,
            "last_name": self.last_name,
            "user_name": self.username,
            "password": self.password
            }
```

3.2.6.2. Todo

This model has all parameters to hold data about to-do. Model description:

```
            class Todo(db.Model, UserMixin):
                """ Todo model. Object to create and make manipulations with database """
                # table name in database
                __tablename__ = "todos"
                # id- needs a primary key
                id = db.Column("id", Integer, primary_key=True, nullable=False)
                # to-do descriptio- to-do text
                description = db.Column("description", String(200))
                # is to-do done, o only created
                is_done = db.Column("isdone", Boolean)
                # creation date and time
                created_date = db.Column("created_date", DateTime)
                # date, when todo was set as done
                done_date = db.Column("done_date", DateTime)
                # user id to create relationship with User model
                user_id = db.Column("user_id", Integer, ForeignKey("users.id"))

                def __init__(self, description, user_id):
                    """ Initialize model """
                    creation_date   =   datetime.strptime(str(datetime.now().strftime("%Y-%m-%d
%H:%M:%S")),"%Y-%m-%d %H:%M:%S")

                    self.description = description
                    self.is_done = False
```

```
        self.created_date = creation_date
        self.user_id = user_id

    def to_json(self):
        """ method used to convert model data to json """
        return {
                "id": self.id,
                "description": self.description,
                "is_done": self.is_done,
                "created_date": self.created_date,
                "done_date": self.done_date
                }
```

3.3. Docker image

Docker image is an existing python image (python:slim-bullseye) with additional dependencies to run flask application. Flask application has its own working directory. Docker container expose 80 port for HTTP requests. As soon as docker image runs, Flask application starts. Docker file is compiled and image built in pipeline. Docker file:

```
FROM python:slim-bullseye
WORKDIR /final-project
COPY /site /final-project
ENV FLASK_APP="main.py"
ENV FLASK_DEBUG=1
RUN pip install--upgrade pip \
   # flask framework
   pip install flask \
   # flask login module for website
   flask_login \
   # sql_alchemy module to communicate with DB
   flask_sqlalchemy \
   # bcrypt for encryption
   bcrypt \
   # pyJWT using for api to encode and decode token
   pyJWT \
   #
   pymysql
EXPOSE 80
CMD [ "python", "main.py" ]
```

3.4. Pipeline

Using Git Hub repository as version control system. Pipeline is written using Git Hub Actions. Pipeline file location: *./.github/workflows/main.yml.* To accomplish tasks, pipeline has jobs and steps.

3.4.1. Lint Python code

This job checks if code is written using best programming practices. Job description:

```
lint-code:
      name: 'Linting code'
      runs-on: ubuntu-latest
      steps:
        - name: 'Checkout reposiroty'
          uses: actions/checkout@v4
        - name: 'Install dependencies'
          run: |
            python-m pip install--upgrade pip
            pip install pylint flask flask_login flask_sqlalchemy
        - name: 'Test python code with pylint'
          working-directory: ./site
          run: |
            pylint app/*.py
```

3.4.2. Build and publish docker image

This job builds and publishes project image to private registry (Docker Hub). Job description:

```
build-publish-image:
    name: 'Build and publish final project image'
    runs-on: ubuntu-latest
    needs: lint-code
    steps:
     - name: 'Checkout reposiroty'
       uses: actions/checkout@v4

     - name: 'Build image'
       run: |
        docker build-t mantelis900726/final-project-image:latest .
     - name: 'Publish image'
       run: |
        docker login -u ${{ secrets.DOCKERHUB_USERNAME }} -p ${{
secrets.DOCKERHUB_PASSWORD }}
        docker push mantelis900726/final-project-image:latest
```

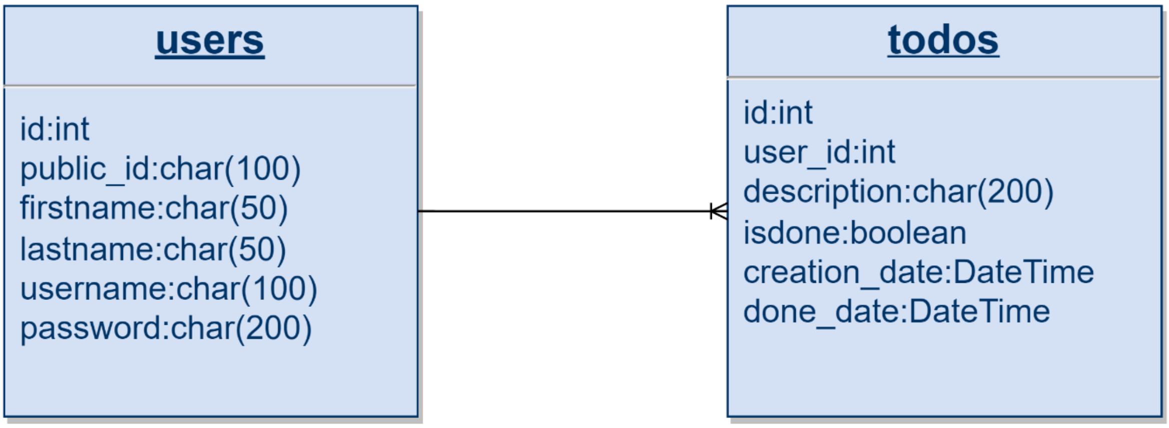
3.4.3. Change task definition

This job changes task definition in AWS ECS service. Connects to AWS and sets new task definition. ECS service automatically starts new container. When container is up and running, ECS service shutdowns old container. Job description:

```
aws_ecs_task_definition:
  name: 'Change task definition'
  runs-on: ubuntu-latest
  needs: build-publish-image
  steps:
   - name: 'Checkout repository'
     uses: actions/checkout@v4

   - name: 'Configure AWS credentials'
     uses: aws-actions/configure-aws-credentials@v1
     with:
      aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY }}
      aws-secret-access-key: ${{ secrets.AWS_SECRET_KEY }}
      aws-region: ${{ env.REGION }}

   - name: 'Change task definition in cloud'
     uses: aws-actions/amazon-ecs-deploy-task-definition@v1
     with:
      task-definition: ${{ env.TASK_DEFINITION }}
      service: ${{ env.ECS_SERVICE_NAME }}
      cluster: ${{ env.ECS_CLUSTER_NAME }}
      wait-for-service-stability: true
```

3.5. Database

Database is used to store application data. Using MySQL database. Database engine MySQL 8.0. Database tables are generated automatically using Flask-SQLAlchemy module. Tables parameters and relationships defined in models. Database diagram:



Flask-SQLAlchemy generates two tables with relationship one-to-many.