The Misinformation Game

Technical project documentation

Authors (Alphabetical order)

Baris Ertugrul

Celina Huseby Botten

Gaël Garnier

Grigory Provodin

Hamza El Haouti

Kevin Inkoom

Pedro Parente Fonte Santa

Course

Minor Cloud Solutions

Semester 2 of 23-24

Lecturers

Peter Odenhoven x Michel Vorenhout

Class

CS2

Creation Date

14th of June 2024

Last Modification Date

1st of July 2024

Version

1

Table of Contents

Introduction

This project is conducted by students at the Amsterdam University of Applied Sciences for the client Wouter van den Bos. Bos is a researcher at the Universiteit van Amsterdam (UvA) with a broad background in neuroscience and developmental psychology. His aim is to undertake a 5-year ERC project focusing on social media use among young people. The research will utilize a social media simulator, incorporating "The Misinformation Game", a free and open-source application built to study how people interact with information on social media. The application simulates a fake social media platform (either showing the posts one at a time or in a feed format) where participants can interact with both posts and their comments. The foundation of this student project relies on the primary objective to deploy this application on cloud infrastructure provided by Microsoft Azure, as the UvA's servers currently do not support Google Firebase at this point, which the application is originally built on. Additional information about the participants will be collected separately through Qualtrics.

1. The Misinformation Game

"The Misinformation Game" application is an open-source social media simulator for research (based on https://misinfogame.com/), powered with Google Firebase components for the backend with several functionalities provided out of the box. The Universiteit van Amsterdam servers do not support Google Firebase and the focus of the project has therefore been to migrate it to Microsoft Azure services and self-made components to deploy it to the Azure cloud (which in contrast is supported by the UvA).

1.1. The Misinformation Game Workflow

The original application consisted of various Firebase components including Firebase Firestore (database), Firebase Authentication, Firebase Storage, as well as Firebase Hosting. These four areas have been the main focus for this student project, and the new solution relies on Microsoft Azure services and self-made components (see figure 1.1 for the workflow chart of the original application).

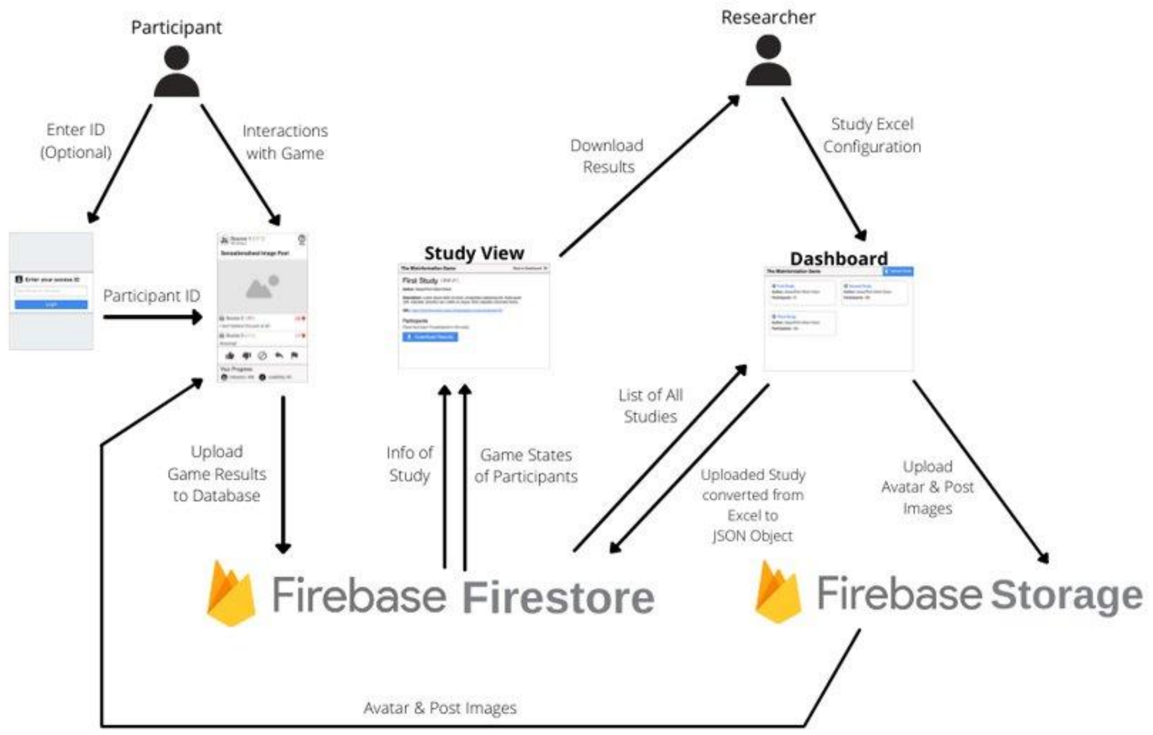


Figure 1.1 Original workflow of the application with Firebase.

The modified application consists of four main parts: one app for the game itself (where participants interact with the posts), a dashboard for the researchers (where they "control" the studies), an API (Application Programming Interface – which facilitates the communication between the frontend and backend of the application), a blob storage (which stores images used in the game), and a database (storing the studies, game results, and authenticated users). (see figure 1.2 for an overview of the workflow). With the new solution, the game and the dashboard for the researchers are separated into two applications and have their own Web Apps deployed in

Azure. They are both connected to the same API in order to communicate with the database in the backend. The results from the studies are processed by the API, modified and stored in the database, and the researchers can get this data from the database with the help from the API.

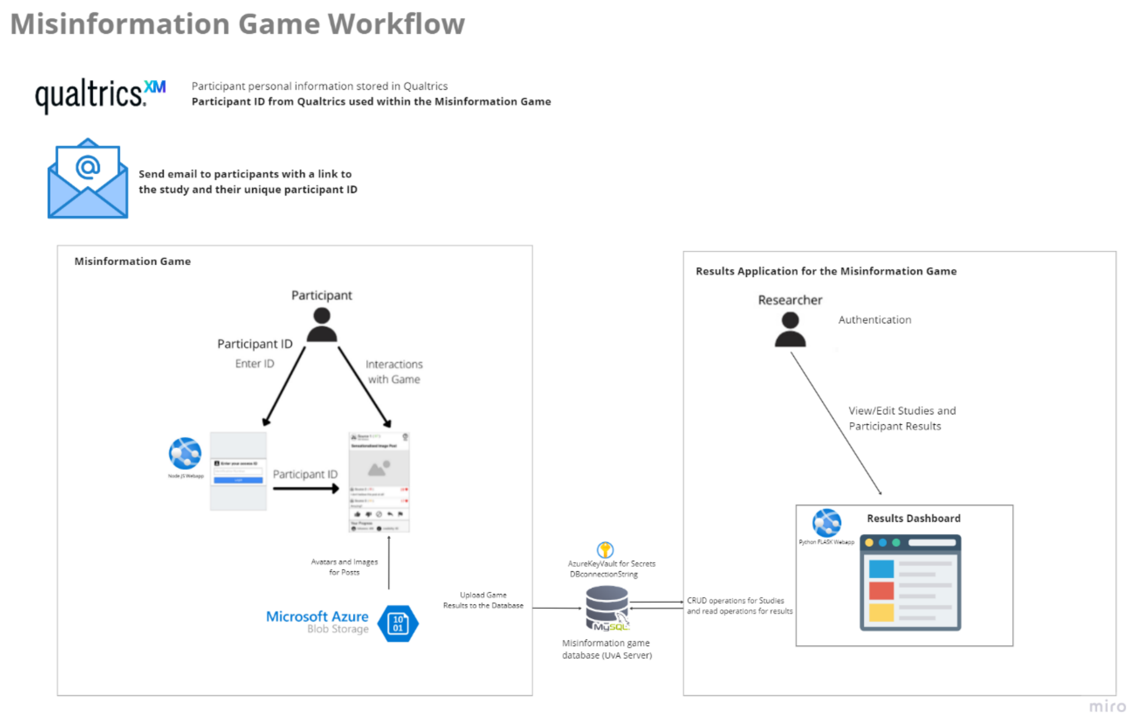

Misinformation Game Workflow
qualtrics XM
Participant personal information stored in Qualtrics
Participant ID from Qualtrics used within the Misinformation Game
Send email to participants with a link to the study and their unique participant ID
Misinformation Game
Participant
Participant ID
Enter ID
Interactions with Game
Node.JS Webapp
Participant ID
Avatars and Images for Posts
Microsoft Azure Blob Storage
Upload Game Results to the Database
AzureKeyVault for Secrets DBconnectionString
MySQL
Misinformation game database (UvA Server)
CRUD operations for Studies and read operations for results
Results Application for the Misinformation Game
Researcher
Authentication
View/Edit Studies and Participant Results
Results Dashboard
Python FLASK Webapp
miro

Figure 1.2 Basic workflow for the modified application with the new solution.

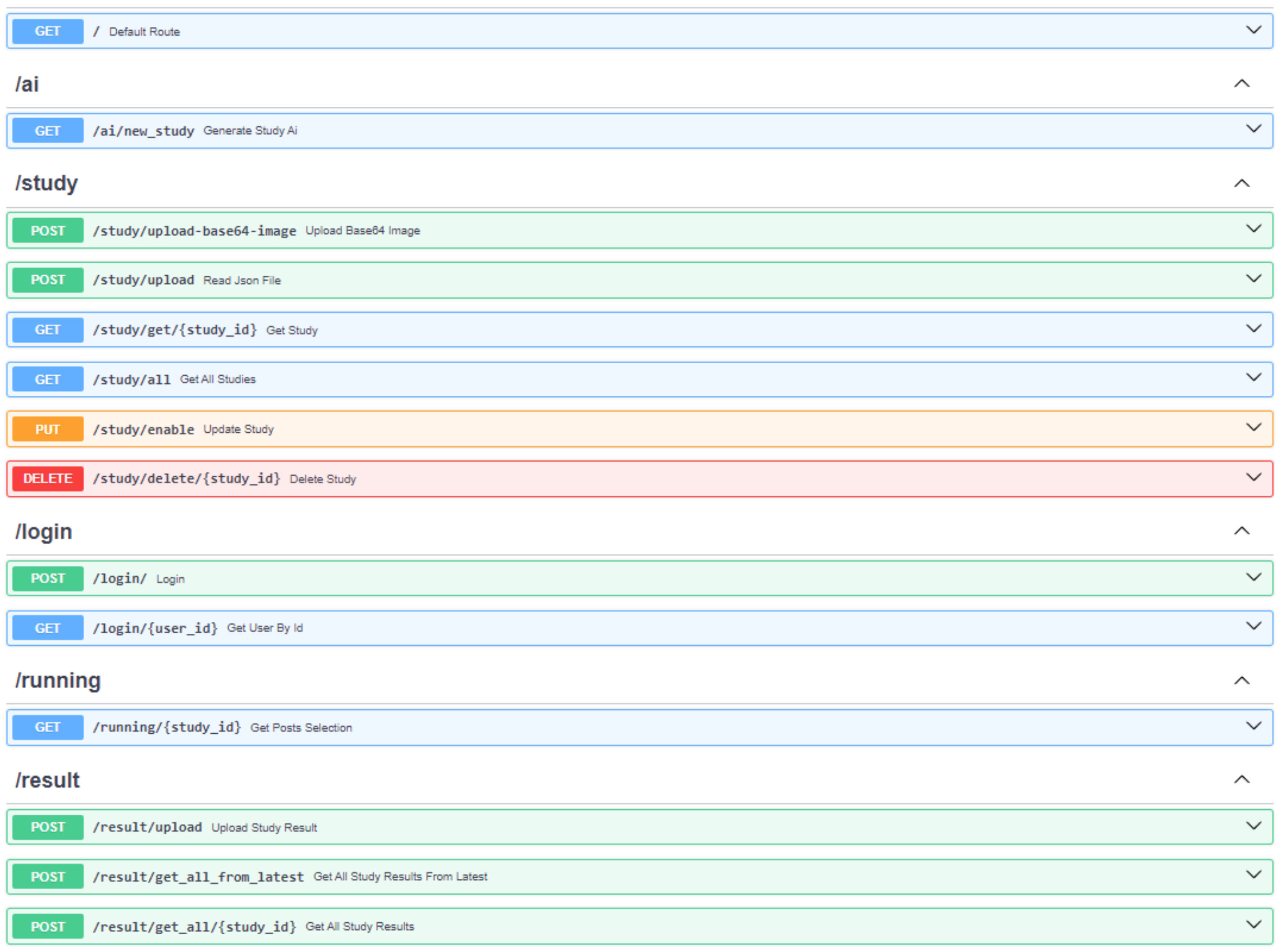
A new API has been developed to facilitate communication between the frontend and backend components of the application. The game's original code (the *legacy app*) has been mostly remained untouched, with only necessary modifications made to redirect endpoints from the previous Firebase solution to the new API. As a result, the game retains its original appearance as the original open-source application, but now operates using the new API infrastructure instead of Firebase.

The application is now hosted on Microsoft Azure instead of Google Firebase, utilizing three Azure Web App Services: two B1 tier services for the frontend (one for the researcher dashboard and one for the game itself) and one B2 tier service for the API backend. The deployment is managed through containerized hosting using Azure Container Registry, employing Docker instead of the standard Web App deployment.

All three App Services will remain live at all times. However, the game frontend should only be accessible during test periods for security reasons. This is because participants only need an ID/number to enter the test, without needing proper login. This means that anyone who can guess or obtain an ID could gain access. Therefore, to maximize security, it is important to limit the exposure of the tests to necessary periods only. During periods without active tests, the Web App and URL will still be live, but the website will display no content. Once a study is active, the website will display the relevant content again with the test for participants interaction.

The original relational database schema has also been modified, to a more normalized database. This means that the various data is organized in several related tables, to minimize redundancy and give a clearer overview. However, with the current solution, results are still stored denormalized due to time restraints and for a simpler overview of the results for the researchers. The results are therefore displayed in fewer tables to optimize data retrieval. The database schema consists of the tables: study_ui_settings, study_basic_settings, study_advanced_settings, study_pages_settings, admin_users, post_selection_methods, studies, avatar, source_style, sources, participants, posts, comments, posts_interactions, comments_interactions, study_results_file (see "db_model.py" file in the project code for more details).

However, as of now, the results are stored in one single table. Though for data retrieval (see chapter "4.3 How to Download Results"), you will get a downloaded spreadsheet with the results of the test.



GET / Default Route

/ai

GET /ai/new_study Generate Study Ai

/study

POST /study/upload-base64-image Upload Base64 Image

POST /study/upload Read Json File

GET /study/get/{study_id} Get Study

GET /study/all Get All Studies

PUT /study/enable Update Study

DELETE /study/delete/{study_id} Delete Study

/login

POST /login/ Login

GET /login/{user_id} Get User By Id

/running

GET /running/{study_id} Get Posts Selection

/result

POST /result/upload Upload Study Result

POST /result/get_all_from_latest Get All Study Results From Latest

POST /result/get_all/{study_id} Get All Study Results

Figure 1.3 Current working endpoints in the application ("FastAPI Swagger UI").

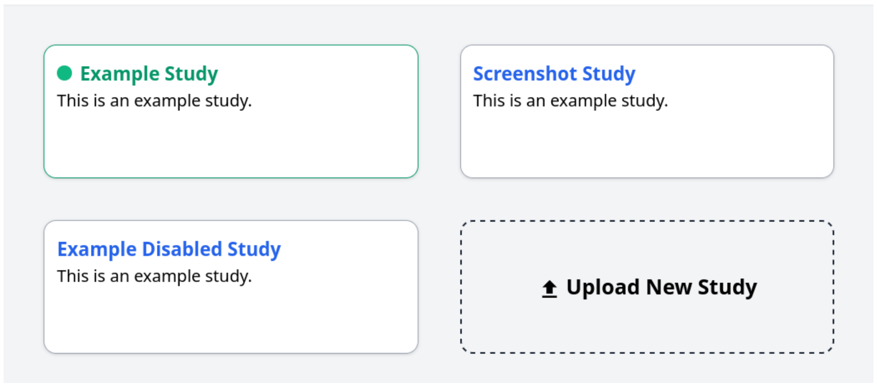
These endpoints will be further explained in later chapters, including "Researcher login", "Participant login", "Upload new studies", "Enable and disable studies", "Download results", and "Delete study".

1.2. Researcher Dashboard Interaction

In the researcher dashboard, authorized users can "control" the studies. This includes getting an overview of the uploaded studies, which one is live, and for each study you have the choice to enable/disable it, downloading results, and deleting studies. ("Editing Studies" is still shown but doesn't work – could perhaps be considered to implement later).

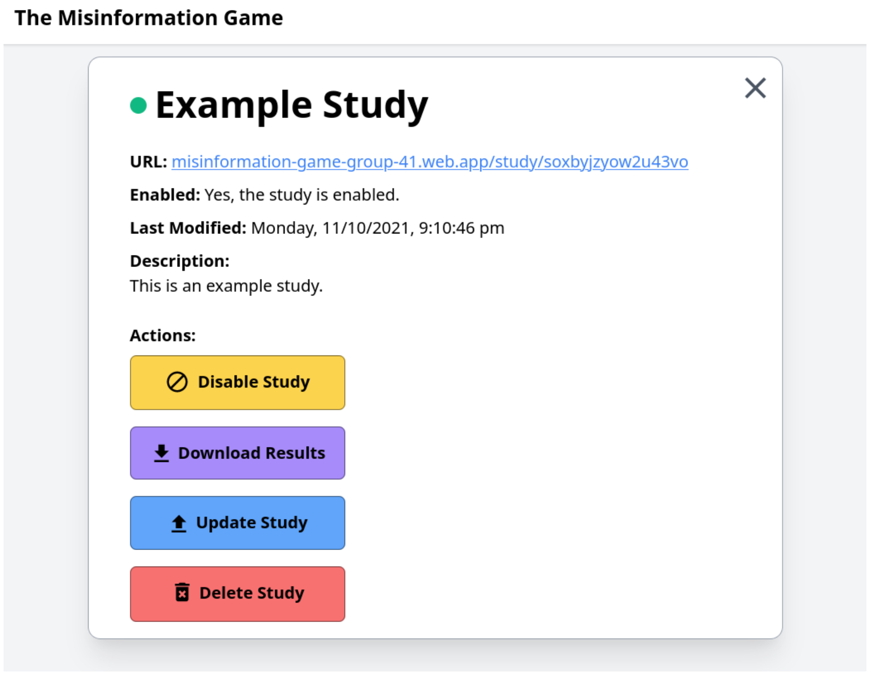
Figure 1.4 shows the overview of all studies, which one is live (shown with a green border) and the possibility to "Upload New Studies". The logged in researcher will be shown in the top right corner.

Figure 1.5 shows the different available actions for a study. When a study is live, the action for "Enable Study" is instead shown as a yellow button with "Disable Study".



● Example Study
This is an example study.
Screenshot Study
This is an example study.
Example Disabled Study
This is an example study.
Upload New Study

Figure 1.4 Researcher dashboard with an overview of studies.


The Misinformation Game

● Example Study

URL: misinformation-game-group-41.web.app/study/soxbyjzyow2u43vo

Enabled: Yes, the study is enabled.

Last Modified: Monday, 11/10/2021, 9:10:46 pm

Description:
This is an example study.

Actions:

Figure 1.5 Available functionalities for a study ("Update Study" is not available).

1.3. Participant Study Interaction

After getting access to the (see chapter "3 Login: Participants" for how participants will log in) participants will first be presented with a couple of pages with information. Afterwards, they will enter the study and see the posts.

The participants have various ways of interacting with the posts, and the related comments. On each post, the participants can like or dislike the post, share it, or flag it. It is also possible to skip the post by clicking "Skip Post" (see figure 1.6). The participant can also interact with comments on the posts, by liking or disliking (see figure 1.7). If you click on the "thumbs up" button, clicking an additional time will remove the "like".

The time they use on each post will also be registered, to get as much data from the studies as possible.

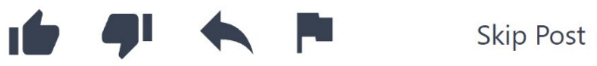


Figure 1.6 Possible reactions on posts.

Figure 1.7 Possible reactions on comments.

It is also possible for participants to add their own comments to posts (see figure 1.8).

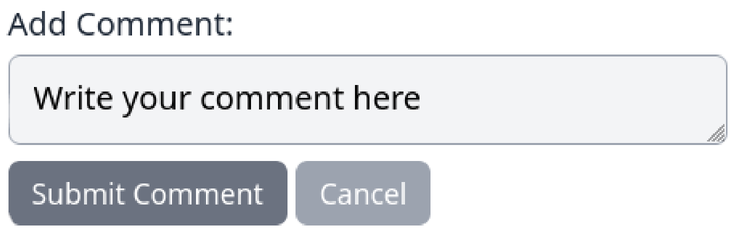


Figure 1.8 Adding comments to posts.

During the study, participants' progress will be tracked, with the imagined number of followers and credibility score always visible. Based on the participants' interactions, these metrics will change accordingly – starting with 0 followers and credibility score of 50.

To proceed to the next post, participants can press "Continue to Next Post" if the posts are displayed on at a time. In a continuous-feed format, they can simply scroll down to view the next post.

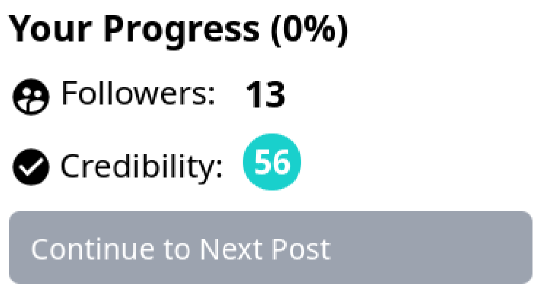


Figure 1.9 The progress of the participant.

1.4. In code: Backend

For an overview of the backend directory see figure 1.10. The directory includes an *app* directory, *.dockerignore* (specifies which files and directories should be excluded from the build context when creating a Docker image) and *.gitignore* files (specifies which files and directories Git should ignore, these won't be tracked and included in version control), a Dockerfile, two README files (one for Docker deployment and one for a general overview), and a requirements.txt file listing the necessary packages for running the backend.

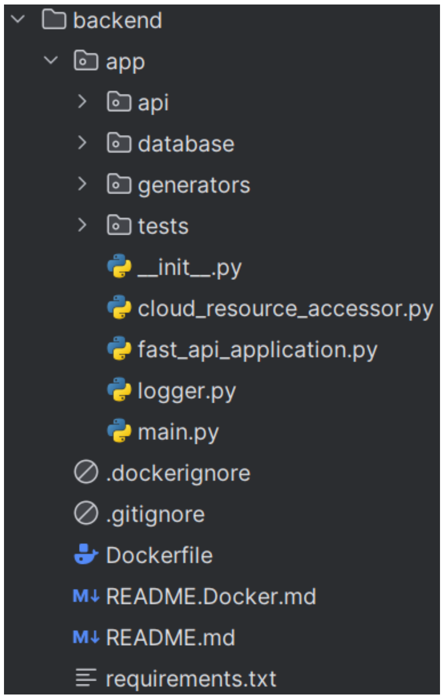
Within the app directory, there are several subdirectories and files:

- api: Contains the different API routers and response types.
- database: Includes the database models, blob storage, conversion functions, and related components.
- generators: Implements an AI generator for creating simulated data for posts and similar purposes.
- tests: Contains JSON filed for testing the API.

Additionally, there are five Python files:

- __init__.py: An empty initialization file.
- cloud_resource_accessor.py: A file accessing cloud resources.
- fast_api_aplication.py: Contains the main class for the FastAPI application.
- logger.py: Handles logging and formatting.
- main.py: The main file for running the backend.

For a more detailed description of the application's structure and functions, including documentation and comments, refer to the codebase.



```
backend
  app
    api
    database
    generators
    tests
    __init__.py
    cloud_resource_accessor.py
    fast_api_application.py
    logger.py
    main.py
  .dockerignore
  .gitignore
  Dockerfile
  README.Docker.md
  README.md
  requirements.txt
```

Figure 1.10 Backend directory.

2. Login: Researchers

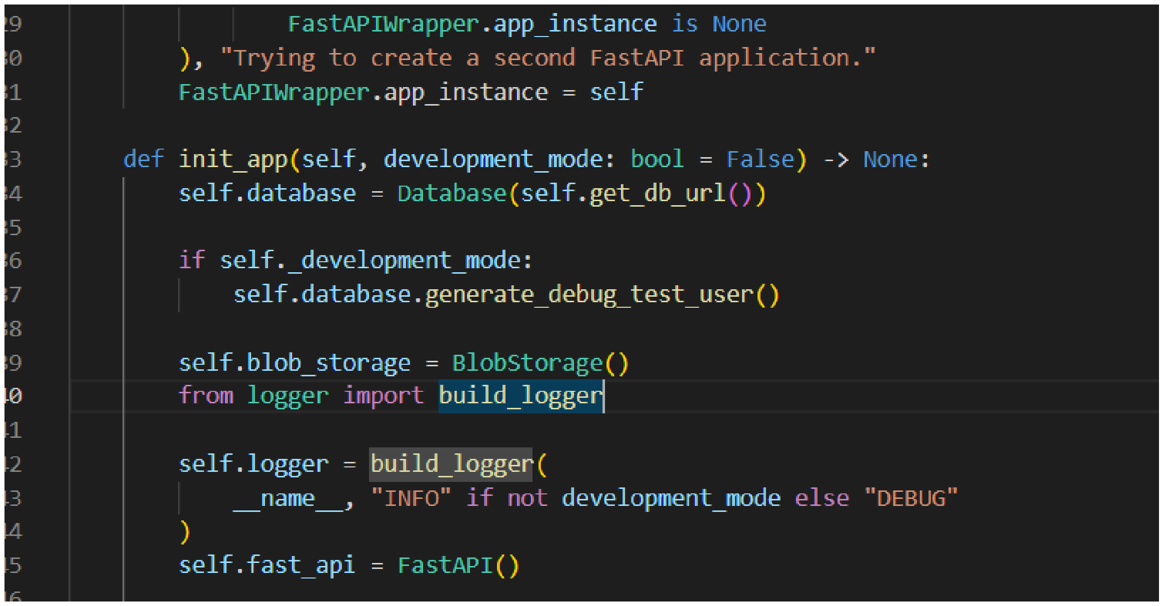
For the researchers we developed a whole new application using Flask-framework. This application provides researchers with a centralized platform to manage their studies. In this application we implemented a new login system. In the previous setup, researchers had to log in using their registered Google accounts, which were stored in a database. However, in the new application we use the authentication method username and password combination. The decision to move away from the Single Sign-On (SSO) approach with Google accounts was driven by the restriction imposed by ICTS (Hva/Uva IT department) to use Entra-ID. To ensure the security of the new authentication system we have implemented security measures like password hashing, input validation and token-based Authentication.

2.1. In code: Researcher Login

When running the application in debug mode there will be a test user created in the backend with the following credentials:

Username: test

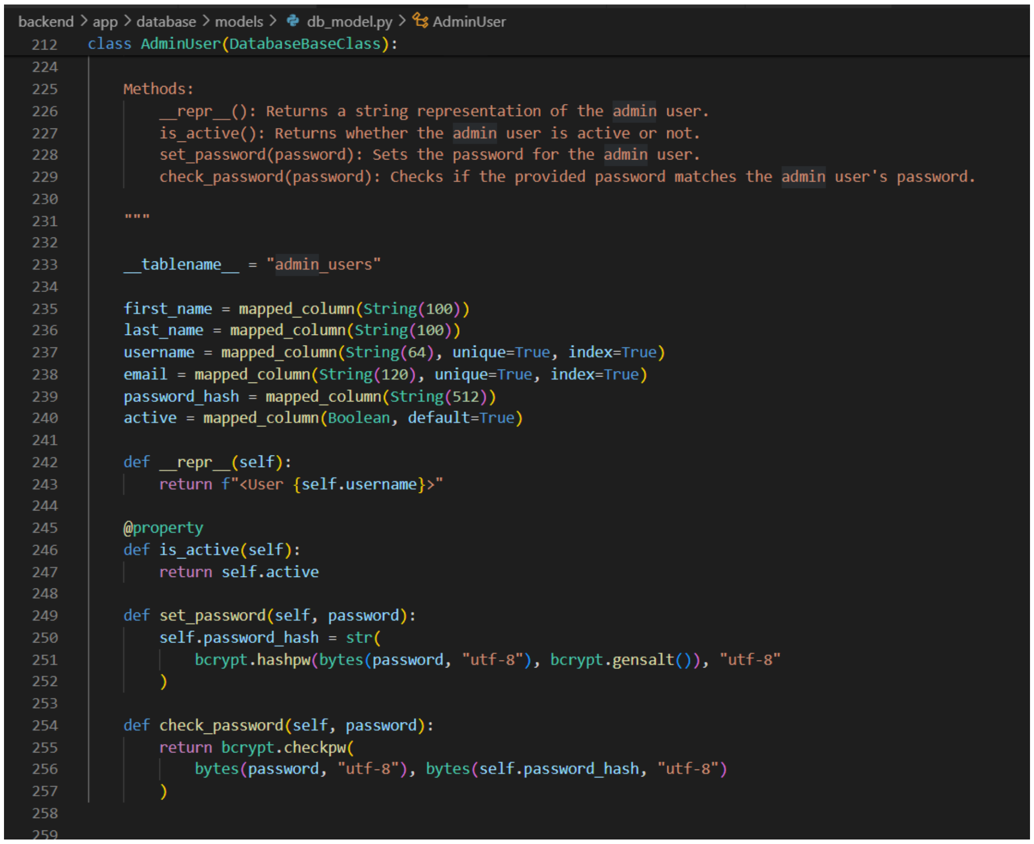
Password: test



```
            FastAPIWrapper.app_instance is None
        ), "Trying to create a second FastAPI application."
        FastAPIWrapper.app_instance = self

    def init_app(self, development_mode: bool = False) -> None:
        self.database = Database(self.get_db_url())

        if self._development_mode:
            self.database.generate_debug_test_user()

        self.blob_storage = BlobStorage()
        from logger import build_logger

        self.logger = build_logger(
            __name__, "INFO" if not development_mode else "DEBUG"
        )
        self.fast_api = FastAPI()
```

Figure 2.1 Debug mode.

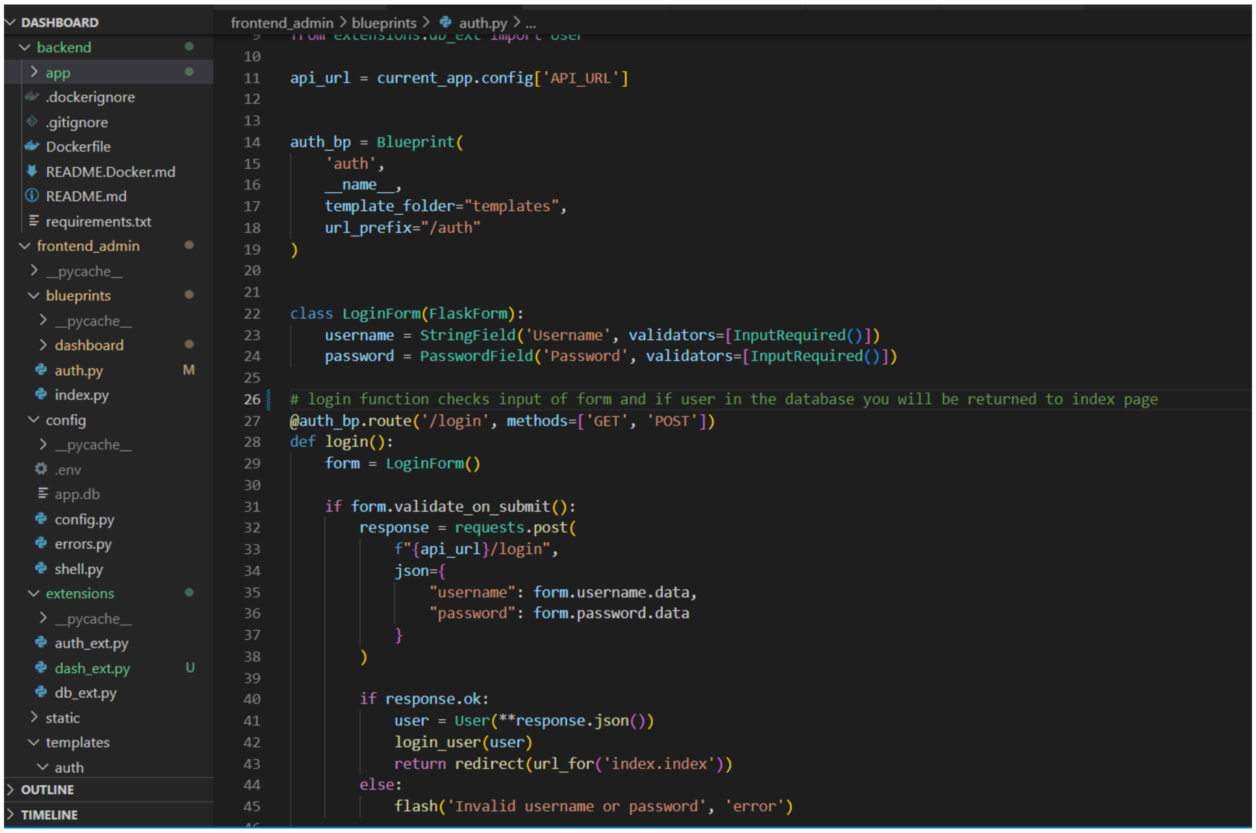
There are multiple approaches to create new users. One way is to add users directly to the database. Alternatively, you can create a new API endpoint and combine it with a registration form. The approach of API endpoint and registration form is the best practice because accessing the database directly is less secure and scalable.



```
class AdminUser(DatabaseBaseClass):

    Methods:
        __repr__(): Returns a string representation of the admin user.
        is_active(): Returns whether the admin user is active or not.
        set_password(password): Sets the password for the admin user.
        check_password(password): Checks if the provided password matches the admin user's password.

    """

    __tablename__ = "admin_users"

    first_name = mapped_column(String(100))
    last_name = mapped_column(String(100))
    username = mapped_column(String(64), unique=True, index=True)
    email = mapped_column(String(120), unique=True, index=True)
    password_hash = mapped_column(String(512))
    active = mapped_column(Boolean, default=True)

    def __repr__(self):
        return f"<User {self.username}>"

    @property
    def is_active(self):
        return self.active

    def set_password(self, password):
        self.password_hash = str(
            bcrypt.hashpw(bytes(password, "utf-8"), bcrypt.gensalt()), "utf-8"
        )

    def check_password(self, password):
        return bcrypt.checkpw(
            bytes(password, "utf-8"), bytes(self.password_hash, "utf-8")
        )

```

Figure 2.2 Admin model.

In the db_model.py we create the table “admin_users” for the researchers with the needed attributes. In the table we save the password in an hash with help from bcrypt. For more information you can check the comments at db_model.py.



```python
from extensions.db_ext import User

api_url = current_app.config['API_URL']


auth_bp = Blueprint(
    'auth',
    __name__,
    template_folder="templates",
    url_prefix="/auth"
)


class LoginForm(FlaskForm):
    username = StringField('Username', validators=[InputRequired()])
    password = PasswordField('Password', validators=[InputRequired()])

# login function checks input of form and if user in the database you will be returned to index page
@auth_bp.route('/login', methods=['GET', 'POST'])
def login():
    form = LoginForm()

    if form.validate_on_submit():
        response = requests.post(
            f"{api_url}/login",
            json={
                "username": form.username.data,
                "password": form.password.data
            }
        )

        if response.ok:
            user = User(**response.json())
            login_user(user)
            return redirect(url_for('index.index'))
        else:
            flash('Invalid username or password', 'error')
```

Figure 2.3 Login system.

In the flask app we work with blueprints. The reason we use this is because it's easier to organize the components in the app. We have now a better overview of the different components.

the auth.py file defines a Blueprint for authentication, a login form, and two view functions: login and logout. The login function handles user login by sending a request to the API and logging the user in if the response is OK. The logout function logs the user out and redirects them to the login page.

3. Login: Participants

The participants that will partake in the studies will be able to access the game via a designated URL, which may be distributed for instance via email. This URL is dynamically linked to the currently enabled study/game. To enter, the participants will need an access ID. This ID will be connected to their corresponding ID in Qualtrics, a software used for survey creation and reporting. This linkage ensures that each participant’s actions and results in the game are accurately connected to their profile in Qualtrics, facilitating comprehensive research analysis and reporting.

4. Navigate the Dashboard

The dashboard is designed for authorized users (the researchers) to login and "control" the studies/games in the application. From this interface, you can upload new studies, enable and disable studies (to choose which one is currently live and available for participants to interact with), as well as download the results from various studies.

4.1. How to Upload New Studies

To upload a new study, researchers have to login to the application dashboard, navigate to studies and click on the "Upload New Study" button. A pop-up window will ask for a file and you must navigate to the correct file in your computer. It will need a spreadsheet with a specific format (see chapter "4.1.1 Spreadsheets" for additional information). After selecting the wanted file, click "Submit File". From this, the application will convert the spreadsheet to a playable study.

The study does not become live right away, and another step will be necessary to make it the current study (see chapter "5.2 How to Enable and Disable Studies").

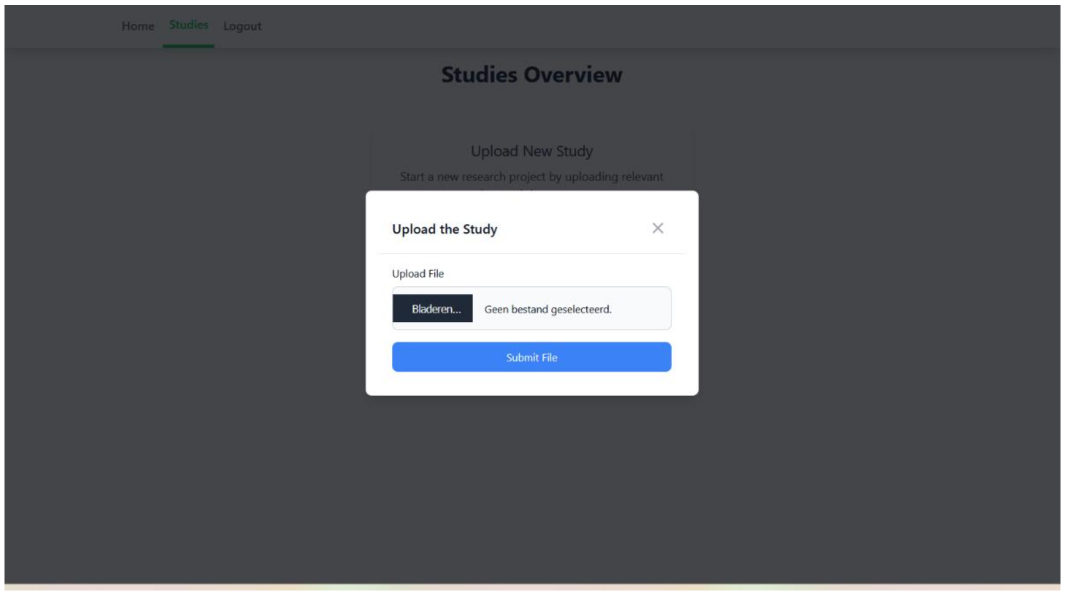


Figure 4.1 Upload new study.

4.1.1. Spreadsheets

To successfully upload a study, the spreadsheet must follow a specific format (links to the required format are provided below). The spreadsheet consists of two pages that cannot be edited, providing details on its functionalities, formatting guidelines, a general overview, and status information on the current condition of the test/spreadsheet. The remaining pages contain customizable variables to tailor the study as needed. For clarification, each element/row in the spreadsheet includes a descriptive blue box explaining its meaning and purpose.

A study can be presented in either a single-post format (displaying one post at a time) or continuous-feed format (showing all posts in a vertical scroll). This setting can be adjusted in the spreadsheet in the "General" page under the "User Interface Settings" table, along with several other settings such as name of the study, enabling/disabling reaction features, advanced settings, and such.

In the spreadsheets, various functionalities can be customized to uniquely tailor the studies. For example, you can edit the study name, description, and the prompt shown to participants. More technical functionalities include adjusting the study length (number of posts) and whether it is required for participants to react or comment on posts. You can also enable or disable different reactions, allow multiple reactions or just some, and control the visibility of followers and credibility. Advanced settings include specifying the number of characters required for comments and the wait time before participants can proceed from the prompt page. All functionalities and details can be seen in the spreadsheets.

The following links show the format for the spreadsheets to use when making and uploading new studies. When the legacy app was designed, Google Sheets were used specifically. Using Microsoft Excel will lead to dysfunction, and therefore only Google Sheet should be used when editing the spreadsheet. A copy of the Study Configuration Template Spreadsheet must be made before you can configure your own study (when opening the link, navigate to *File --> Make a copy*).

- Study Configuration Template Spreadsheet
- Example Study (Single Post Mode)
- Example Study (Feed Mode)

4.2. How to Enable and Disable Studies

To enable a study that is uploaded to the application (see chapter "5.1 How to Upload New Studies"), click on the uploaded study and locate the green "Enable Study" button. A pop-up window will prompt you to confirm your intention to enable the study. Click "Enable" to make the game live. This action will trigger processes in the backend and the study will be available using the URL (which now can be sent to the participants to partake in it).

To disable a study, navigate to the currently active study and find the yellow button for "Disable study". When clicking this, a pop-up window will appear and ask if you are sure you want to disable the study. If you want to proceed, click "Disable". This will deactivate the study, and the URL will not contain any data for ongoing tests.

4.3. How to Download Results

After participants have engaged in the study, and the results of their interactions in the game have been stored in the database, this can be downloaded from the researcher dashboard. To do so, navigate to the study you want to download the results from. Click on "Download results". You will receive a spreadsheet containing the results of the selected test.

4.4. How to Delete Studies

There is also an option to delete studies. Although studies should generally not be deleted, but rather archived for research purposes, this feature provides reassurance in case an incorrect study is uploaded, or a similar issue occurs.

To delete a study, navigate to the study in the dashboard, click on it, and a pop-up window will appear. Click the "Delete Study" button. A confirmation window will then appear to ensure you want to proceed with the deletion. This second step provides reassurance, as deleting a study is irreversible and will permanently remove the possibility of retrieving results associated with this study.

4.5. Future Enhancements and Actions

Regarding the future of this application, there are some further improvements and changes that can be considered:

1. Implementing a statistics dashboard for the researchers.
2. Storing data in a normalized format in the database (seeing that the database is normalized, but the results are stored denormalized).
3. Enable editing of studies.
4. Ensure images are properly displayed (they are stored as blobs, but it is not working in the studies).
   4.1. Consider adding an additional entry in the database for each Post to identify images by their ID in the container. Then clear the JSON model in the Python API code to handle this change.
5. Consolidate access to one single URL, with studies pushed to this address (instead of having new URLs for each study).
   5.1. Also, when enabling a new study, override the currently active study (include a confirmation pop-up window with "Are you sure you want to override the current study **study name**?)
6. Consider additional client-requested features, such as mobile scaling and a Likert scale for each post.
7. Enhance the legacy application (frontend client – completely convert to the Flask app), seeing that the original, open-source app consists of deprecated code, challenging architecture, and the lack of sufficient documentation.
8. Implementing a logging system, for when someone has done something (such as downloaded results from a study or enabled/disabled a study) -> who has done *what when*.
9. Migrating from spreadsheets to CSV or JSON.
10. Results when a user posts a comment show a "Comment's Metadata is missing". Can not comment a study right now. (Error might be located in the legacy JavaScript application)
11. Additional security measures.

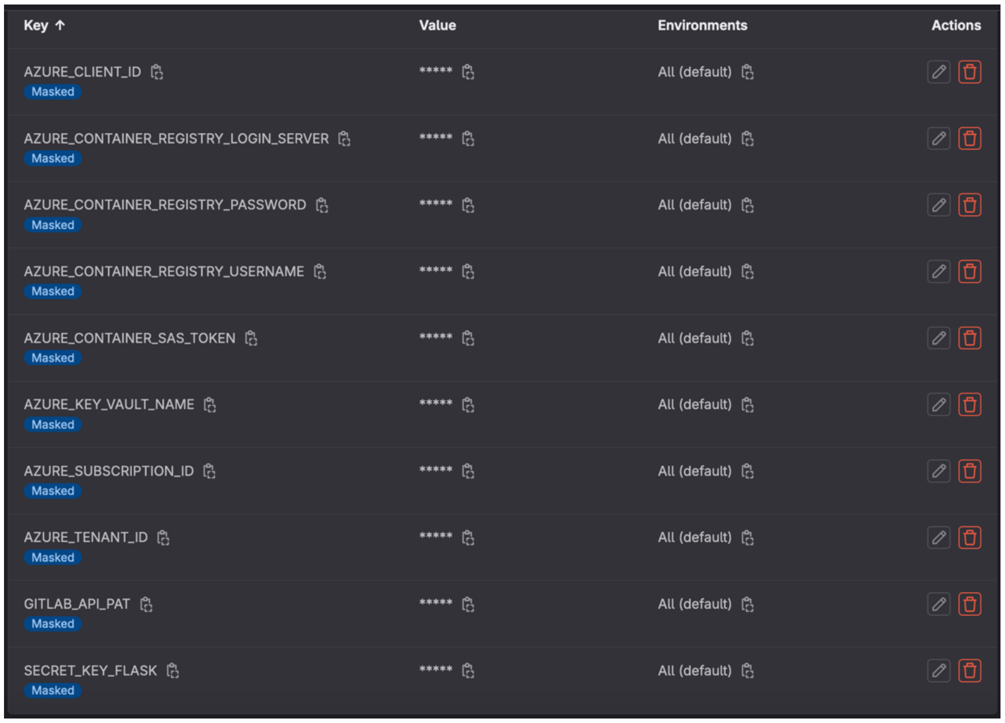
Additionally, please review the backlog in GitLab.

5. Deployment Script

We deploy the application using CI/CD pipeline in our Repository. In the root folder there is a file called "*.gitlab-ci.yml*" which is the configuration file for the pipeline. It defines a set of jobs that are executed in a specific order to automate various tasks. The pipeline is pushing the terraform script(IaC) and docker images of the application to create the needed resources in Azure to run the misinfogame. See the folder Workflow for more details on code.

5.1. GitLab Variables

In the GitLab settings we have stored several variables for the pipeline. The variables are used to store sensitive information such as Azure credentials and other configuration values. All these variables are masked which means that the values of the variables don't appear in the job logs. Here is an overview of the variables stored (see Figure 5.1):



Key ↑	Value	Environments	Actions
AZURE_CLIENT_ID Masked	*****	All (default)	
AZURE_CONTAINER_REGISTRY_LOGIN_SERVER Masked	*****	All (default)	
AZURE_CONTAINER_REGISTRY_PASSWORD Masked	*****	All (default)	
AZURE_CONTAINER_REGISTRY_USERNAME Masked	*****	All (default)	
AZURE_CONTAINER_SAS_TOKEN Masked	*****	All (default)	
AZURE_KEY_VAULT_NAME Masked	*****	All (default)	
AZURE_SUBSCRIPTION_ID Masked	*****	All (default)	
AZURE_TENANT_ID Masked	*****	All (default)	
GITLAB_API_PAT Masked	*****	All (default)	
SECRET_KEY_FLASK Masked	*****	All (default)	

Figure 5.1 Gitlab Variables

The variables "AZURE_TENANT_ID","AZURE_CLIENT_ID" and "AZURE_SUBSCRIPTION_ID" are used in the terraform_deploy, terraform_destroy, and azure_container_apps_update stages to authenticate with Azure. AZURE_SUBSCRIPTION_ID is used to set the subscription.

The AZURE_CONTAINER_SAS_TOKEN variable configures the Terraform backend in the terraform_deploy and terraform_destroy stages. The variable is passed as a run argument to prevent its reveal in the source code.

AZURE_CONTAINER_REGISTRY_USERNAME, AZURE_CONTAINER_REGISTRY_PASSWORD, AZURE_CONTAINER_REGISTRY_LOGIN_SERVER are used in the docker_build_and_push_backend, docker_build_and_push_frontend_admin, and docker_build_and_push_frontend_client stages to authenticate with the Azure Container Registry.

Variables AZURE_KEY_VAULT_NAME and SECRET_KEY_FLASK are used to configure as Terraform variables in the terraform_deploy stage and the terraform_destroy stage. These variables are declared as sensitive and passed as run arguments to prevent their reveal in the source code. AZURE_KEY_VAULT_NAME is the name of the Azure KeyVault that is being accessed by the frontend_admin to retrieve an authentication token for accessing the backend. SECRET_KEY_FLASK is the name (key) of the secret that is stored in the aforementioned KeyVault (since the backend authentication is still in development the dummy secret is stored there now).

The variable GITLAB_API_PAT is the GitLab personal access that has the scope of 'api'. It is used to access two GitLab APIs in the docker_build_and_push_frontend_client job. Note: The token should be generated again if the project will be migrated to another repository. Read more on this in the APIs section.

5.2. Pipeline Stages

The pipeline is divided into six stages:

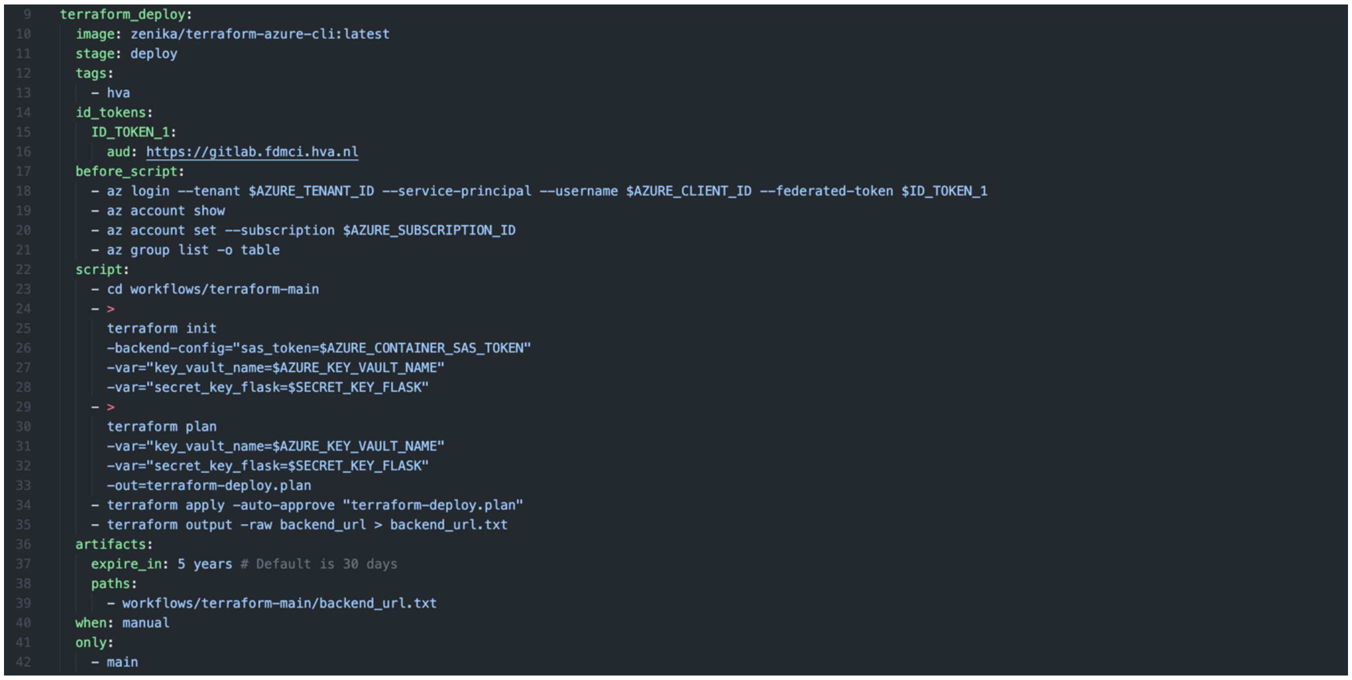
- Deploy
- Backend_build_and_push
- Frontend_admin_build_and_push
- Frontend_client_build_and_push
- Update_container_apps
- Destroy

The ` terraform_deploy` stage in the GitLab pipeline uses the `zenika/terraform-azure-cli:latest` Docker image image that combines Azure CLI and Terraform and operates within the `deploy` stage. It is tagged with `hva` to use HVA's shared runners for this job.

The stage includes:

- Authentication using Azure CLI with a federated token (`ID_TOKEN_1`), (`AZURE_TENANT_ID`), and (`AZURE_CLIENT_ID`).
- Configuration of Azure subscription and retrieval of resource group details.
- Terraform commands executed within the `workflows/terraform-main` directory:

  - Initialization (`terraform init`) with backend configuration using SAS token and variable assignments (`$AZURE_KEY_VAULT_NAME`, `$SECRET_KEY_FLASK`).
  - Planning (`terraform plan`) to generate an execution plan (`terraform-deploy.plan`).
  - Deployment (`terraform apply`) using the generated plan file.
  - Extraction of `backend_url` output to `backend_url.txt`.
- Artifacts from the stage, specifically `backend_url.txt`, are preserved for 5 years.
- A manual trigger (`when: manual`) is required.
- Restricting execution to changes in the `main` branch only.



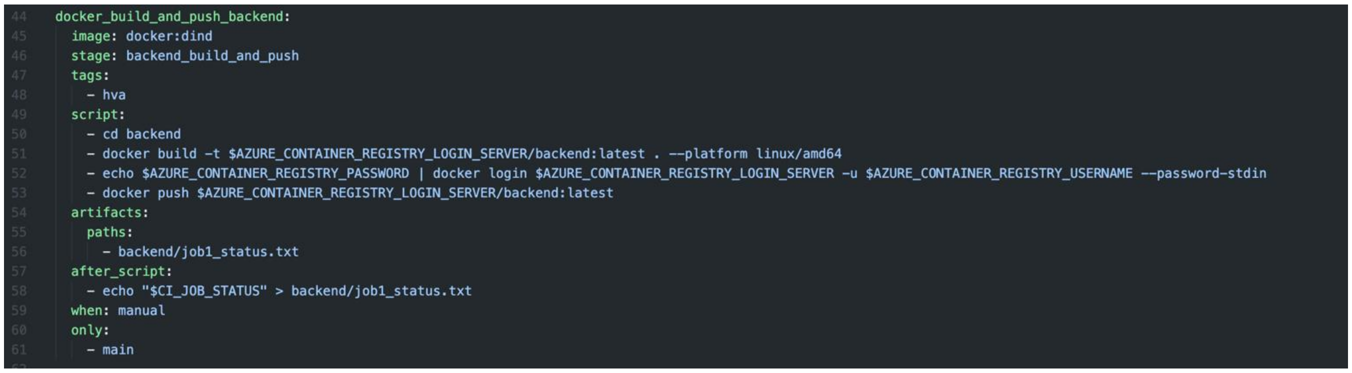
```
terraform_deploy:
  image: zenika/terraform-azure-cli:latest
  stage: deploy
  tags:
    - hva
  id_tokens:
    ID_TOKEN_1:
      aud: https://gitlab.fdmci.hva.nl
  before_script:
    - az login --tenant $AZURE_TENANT_ID --service-principal --username $AZURE_CLIENT_ID --federated-token $ID_TOKEN_1
    - az account show
    - az account set --subscription $AZURE_SUBSCRIPTION_ID
    - az group list -o table
  script:
    - cd workflows/terraform-main
    - >
      terraform init
      -backend-config="sas_token=$AZURE_CONTAINER_SAS_TOKEN"
      -var="key_vault_name=$AZURE_KEY_VAULT_NAME"
      -var="secret_key_flask=$SECRET_KEY_FLASK"
    - >
      terraform plan
      -var="key_vault_name=$AZURE_KEY_VAULT_NAME"
      -var="secret_key_flask=$SECRET_KEY_FLASK"
      -out=terraform-deploy.plan
    - terraform apply -auto-approve "terraform-deploy.plan"
    - terraform output -raw backend_url > backend_url.txt
  artifacts:
    expire_in: 5 years # Default is 30 days
    paths:
      - workflows/terraform-main/backend_url.txt
  when: manual
  only:
    - main
```

Figure 5.2.1 deploy stage.

The `docker_build_and_push_backend` stage in the GitLab pipeline utilizes the `docker:dind` Docker image and operates within the `backend_build_and_push` stage. It is tagged with `hva` to use HVA's shared runners for this job.

The stage executes the following steps:

- Changes directory to `backend`.
- Builds a Docker image (`docker build`) tagged as `$AZURE_CONTAINER_REGISTRY_LOGIN_SERVER/backend:latest` for the `linux/amd64` platform.
- Logs into Azure Container Registry (`docker login`) using credentials from environment variables.
- Pushes the Docker image (`docker push`) to `$AZURE_CONTAINER_REGISTRY_LOGIN_SERVER/backend:latest`.
- Artifacts include `backend/job1_status.txt` capturing the status of the CI job (`$CI_JOB_STATUS`).
- This stage is set to manual (`when: manual`).
- This stage is triggered exclusively by changes in the `main` branch (`only: - main`).



```
docker_build_and_push_backend:
  image: docker:dind
  stage: backend_build_and_push
  tags:
    - hva
  script:
    - cd backend
    - docker build -t $AZURE_CONTAINER_REGISTRY_LOGIN_SERVER/backend:latest . --platform linux/amd64
    - echo $AZURE_CONTAINER_REGISTRY_PASSWORD | docker login $AZURE_CONTAINER_REGISTRY_LOGIN_SERVER -u $AZURE_CONTAINER_REGISTRY_USERNAME --password-stdin
    - docker push $AZURE_CONTAINER_REGISTRY_LOGIN_SERVER/backend:latest
  artifacts:
    paths:
      - backend/job1_status.txt
  after_script:
    - echo "$CI_JOB_STATUS" > backend/job1_status.txt
  when: manual
  only:
    - main
```

Figure 5.2.2 backend_build_and_push stage.

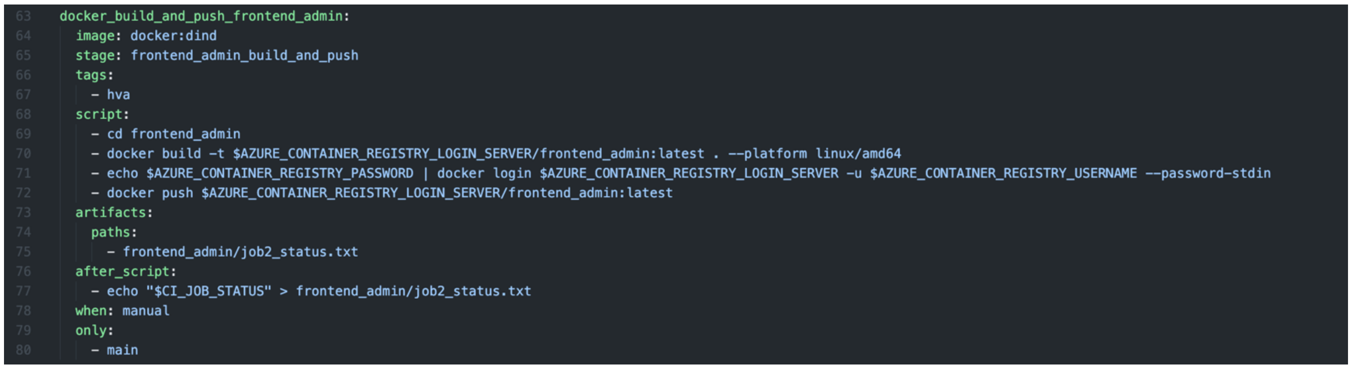
The `docker_build_and_push_frontend_admin` stage in the GitLab pipeline utilizes the `docker:dind` Docker image and operates within the `frontend_admin_build_and_push` stage. It is tagged with `hva` to use HVA's shared runners for this job.

The stage executes the following steps:

- Changes directory to `frontend_admin`.
- Builds a Docker image (`docker build`) tagged as `$AZURE_CONTAINER_REGISTRY_LOGIN_SERVER/frontend_admin:latest` for the `linux/amd64` platform.
- Logs into Azure Container Registry (`docker login`) using credentials from environment variables.
- Pushes the Docker image (`docker push`) to `$AZURE_CONTAINER_REGISTRY_LOGIN_SERVER/frontend_admin:latest`.

- Artifacts include `frontend_admin/job2_status.txt` capturing the status of the CI job (`$CI_JOB_STATUS`).

- This stage is set to manual (`when: manual`).
- This stage is triggered exclusively by changes in the `main` branch (`only: - main`).



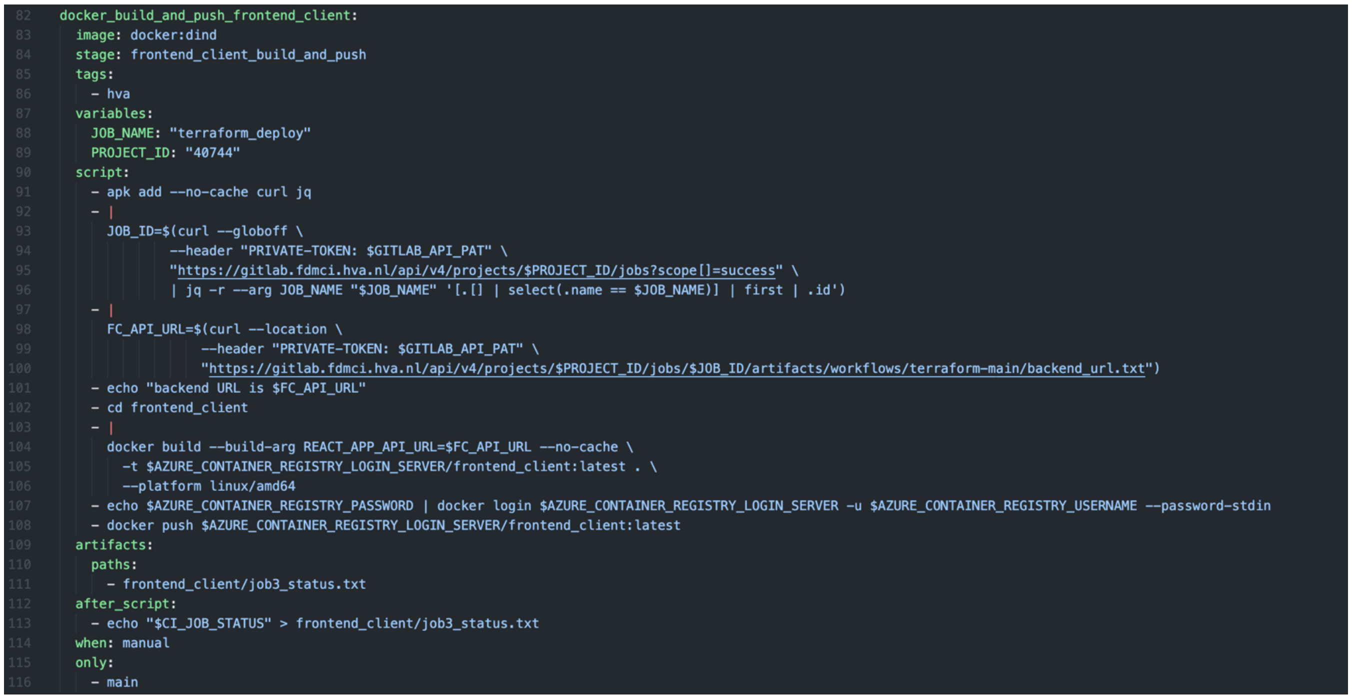
```
docker_build_and_push_frontend_admin:
  image: docker:dind
  stage: frontend_admin_build_and_push
  tags:
    - hva
  script:
    - cd frontend_admin
    - docker build -t $AZURE_CONTAINER_REGISTRY_LOGIN_SERVER/frontend_admin:latest . --platform linux/amd64
    - echo $AZURE_CONTAINER_REGISTRY_PASSWORD | docker login $AZURE_CONTAINER_REGISTRY_LOGIN_SERVER -u $AZURE_CONTAINER_REGISTRY_USERNAME --password-stdin
    - docker push $AZURE_CONTAINER_REGISTRY_LOGIN_SERVER/frontend_admin:latest
  artifacts:
    paths:
      - frontend_admin/job2_status.txt
  after_script:
    - echo "$CI_JOB_STATUS" > frontend_admin/job2_status.txt
  when: manual
  only:
    - main
```

Figure 5.2.3 frontend_admin_build_and_push stage.

The `docker_build_and_push_frontend_client` stage in the GitLab pipeline uses the `docker:dind` Docker image and operates within the `frontend_client_build_and_push` stage. It is tagged with `hva` to use HVA's shared runners for this job. It includes specific variables for job name and project ID.

The stage performs the following steps:

- Installs `curl` and `jq` using `apk` (these packages are needed for manipulations with APIs)
- Retrieves the job ID of the successful `terraform_deploy` job using the GitLab API.
- Extracts the `backend_url.txt` artifact from the `terraform_deploy` job to get the backend URL.
- Logs the backend URL.
- Changes directory to `frontend_client`.
- Builds a Docker image (`docker build`) with the backend URL as a build argument, tagged as `$AZURE_CONTAINER_REGISTRY_LOGIN_SERVER/frontend_client:latest` for the `linux/amd64` platform, without using cache.
- Logs into Azure Container Registry (`docker login`) using credentials from environment variables.
- Pushes the Docker image (`docker push`) to `$AZURE_CONTAINER_REGISTRY_LOGIN_SERVER/frontend_client:latest`.
- Artifacts include `frontend_client/job3_status.txt` capturing the status of the CI job (`$CI_JOB_STATUS`).
- This stage is set to manual (`when: manual`).
- This stage is triggered exclusively by changes in the `main` branch (`only: - main`).



```yaml
docker_build_and_push_frontend_client:
  image: docker:dind
  stage: frontend_client_build_and_push
  tags:
    - hva
  variables:
    JOB_NAME: "terraform_deploy"
    PROJECT_ID: "40744"
  script:
    - apk add --no-cache curl jq
    - |
      JOB_ID=$(curl --globoff \
              --header "PRIVATE-TOKEN: $GITLAB_API_PAT" \
              "https://gitlab.fdmci.hva.nl/api/v4/projects/$PROJECT_ID/jobs?scope[]=success" \
              | jq -r --arg JOB_NAME "$JOB_NAME" '[.[] | select(.name == $JOB_NAME)] | first | .id')
    - |
      FC_API_URL=$(curl --location \
                  --header "PRIVATE-TOKEN: $GITLAB_API_PAT" \
                  "https://gitlab.fdmci.hva.nl/api/v4/projects/$PROJECT_ID/jobs/$JOB_ID/artifacts/workflows/terraform-main/backend_url.txt")
    - echo "backend URL is $FC_API_URL"
    - cd frontend_client
    - |
      docker build --build-arg REACT_APP_API_URL=$FC_API_URL --no-cache \
        -t $AZURE_CONTAINER_REGISTRY_LOGIN_SERVER/frontend_client:latest . \
        --platform linux/amd64
    - echo $AZURE_CONTAINER_REGISTRY_PASSWORD | docker login $AZURE_CONTAINER_REGISTRY_LOGIN_SERVER -u $AZURE_CONTAINER_REGISTRY_USERNAME --password-stdin
    - docker push $AZURE_CONTAINER_REGISTRY_LOGIN_SERVER/frontend_client:latest
  artifacts:
    paths:
      - frontend_client/job3_status.txt
  after_script:
    - echo "$CI_JOB_STATUS" > frontend_client/job3_status.txt
  when: manual
  only:
    - main
```

Figure 5.2.4 frontend_client_build_and_push stage.

The `azure_container_apps_update` stage in the GitLab pipeline uses the `mcr.microsoft.com/azure-cli:latest` Docker image and operates within the `update_container_apps` stage. It is tagged with `hva` to use HVA's shared runners for this job. It also sets a specific Azure resource group via a local variable.

This stage is a workaround to enable the CI/CD for container apps because the default option requiring authentication with a GitHub account was not suitable. This stage will be obsolete if you configure CI/CD in settings of Azure container apps.

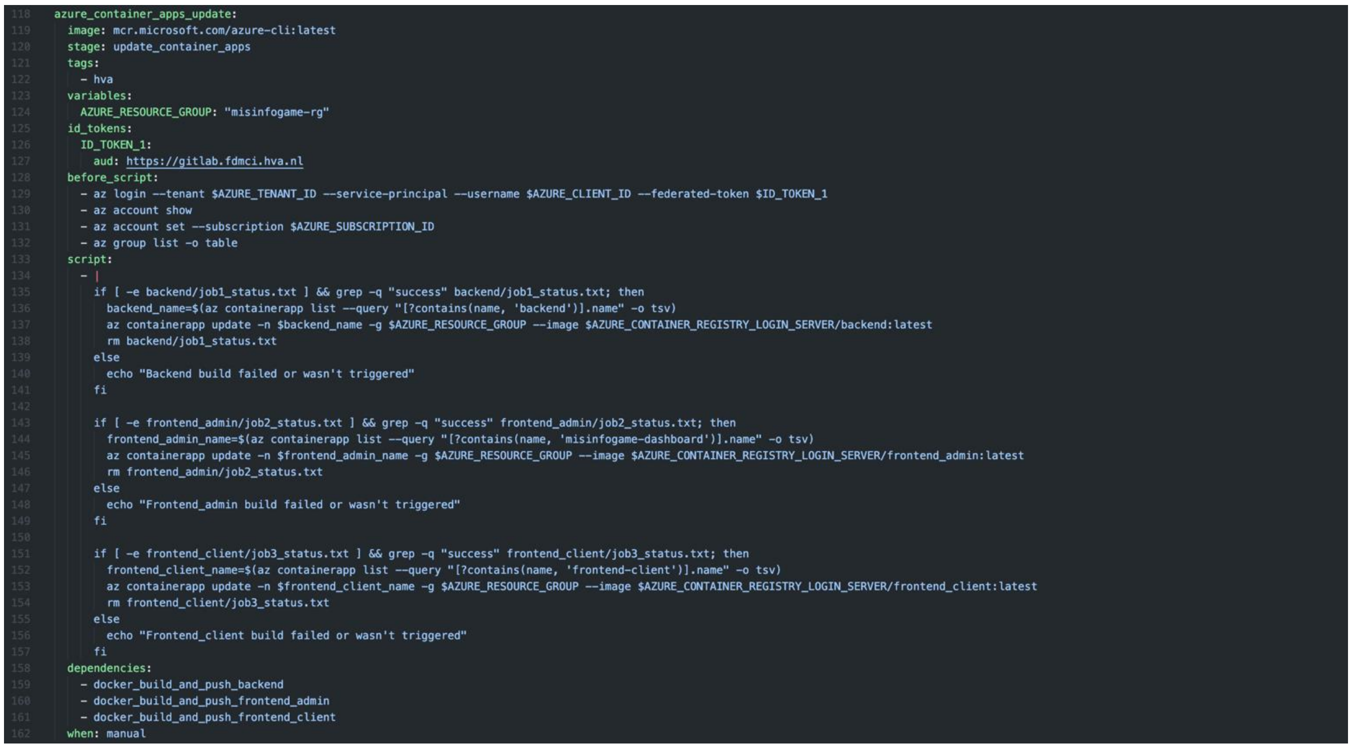
This stage performs the following steps:

- Authenticate using Azure CLI with a federated token (`ID_TOKEN_1`), (`AZURE_TENANT_ID`), and (`AZURE_CLIENT_ID`).
- Sets the subscription.
- Lists Azure resource groups to confirm successful authentication.

The main script updates container apps based on the success of previous build stages:

- Backend:
    - o Checks if `backend/job1_status.txt` exists and contains "success".
    - o Retrieves the name of the backend container app.
    - o Updates the backend container app with the latest image.
    - o Deletes `backend/job1_status.txt`.
    - o Logs a failure message if the backend build failed or wasn't triggered.

- Frontend Admin:
    - Checks if `frontend_admin/job2_status.txt` exists and contains "success".
    - Retrieves the name of the frontend admin container app.
    - Updates the frontend admin container app with the latest image.
    - Deletes `frontend_admin/job2_status.txt`.
    - Logs a failure message if the frontend admin build failed or wasn't triggered.
- Frontend Client:
    - Checks if `frontend_client/job3_status.txt` exists and contains "success".
    - Retrieves the name of the frontend client container app.
    - Updates the frontend client container app with the latest image.
    - Deletes `frontend_client/job3_status.txt`.
    - Logs a failure message if the frontend client build failed or wasn't triggered.
- This stage has dependencies on the `docker_build_and_push_backend`, `docker_build_and_push_frontend_admin`, and `docker_build_and_push_frontend_client` stages.
- It is set to manual (`when: manual`).



```
azure_container_apps_update:
  image: mcr.microsoft.com/azure-cli:latest
  stage: update_container_apps
  tags:
    - hva
  variables:
    AZURE_RESOURCE_GROUP: "misinfogame-rg"
  id_tokens:
    ID_TOKEN_1:
      aud: https://gitlab.fdmci.hva.nl
  before_script:
    - az login --tenant $AZURE_TENANT_ID --service-principal --username $AZURE_CLIENT_ID --federated-token $ID_TOKEN_1
    - az account show
    - az account set --subscription $AZURE_SUBSCRIPTION_ID
    - az group list -o table
  script:
    - |
      if [ -e backend/job1_status.txt ] && grep -q "success" backend/job1_status.txt; then
        backend_name=$(az containerapp list --query "[?contains(name, 'backend')].name" -o tsv)
        az containerapp update -n $backend_name -g $AZURE_RESOURCE_GROUP --image $AZURE_CONTAINER_REGISTRY_LOGIN_SERVER/backend:latest
        rm backend/job1_status.txt
      else
        echo "Backend build failed or wasn't triggered"
      fi

      if [ -e frontend_admin/job2_status.txt ] && grep -q "success" frontend_admin/job2_status.txt; then
        frontend_admin_name=$(az containerapp list --query "[?contains(name, 'misinfogame-dashboard')].name" -o tsv)
        az containerapp update -n $frontend_admin_name -g $AZURE_RESOURCE_GROUP --image $AZURE_CONTAINER_REGISTRY_LOGIN_SERVER/frontend_admin:latest
        rm frontend_admin/job2_status.txt
      else
        echo "Frontend_admin build failed or wasn't triggered"
      fi

      if [ -e frontend_client/job3_status.txt ] && grep -q "success" frontend_client/job3_status.txt; then
        frontend_client_name=$(az containerapp list --query "[?contains(name, 'frontend-client')].name" -o tsv)
        az containerapp update -n $frontend_client_name -g $AZURE_RESOURCE_GROUP --image $AZURE_CONTAINER_REGISTRY_LOGIN_SERVER/frontend_client:latest
        rm frontend_client/job3_status.txt
      else
        echo "Frontend_client build failed or wasn't triggered"
      fi
  dependencies:
    - docker_build_and_push_backend
    - docker_build_and_push_frontend_admin
    - docker_build_and_push_frontend_client
  when: manual
```

Figure 5.2.5 update_container_apps stage.

The `terraform_destroy` stage in the GitLab pipeline uses the `zenika/terraform-azure-cli:latest` Docker image which is the image that combines Azure CLI and Terraform and operates within the `destroy` stage. It is tagged with `hva` to use HVA's shared runners for this job.

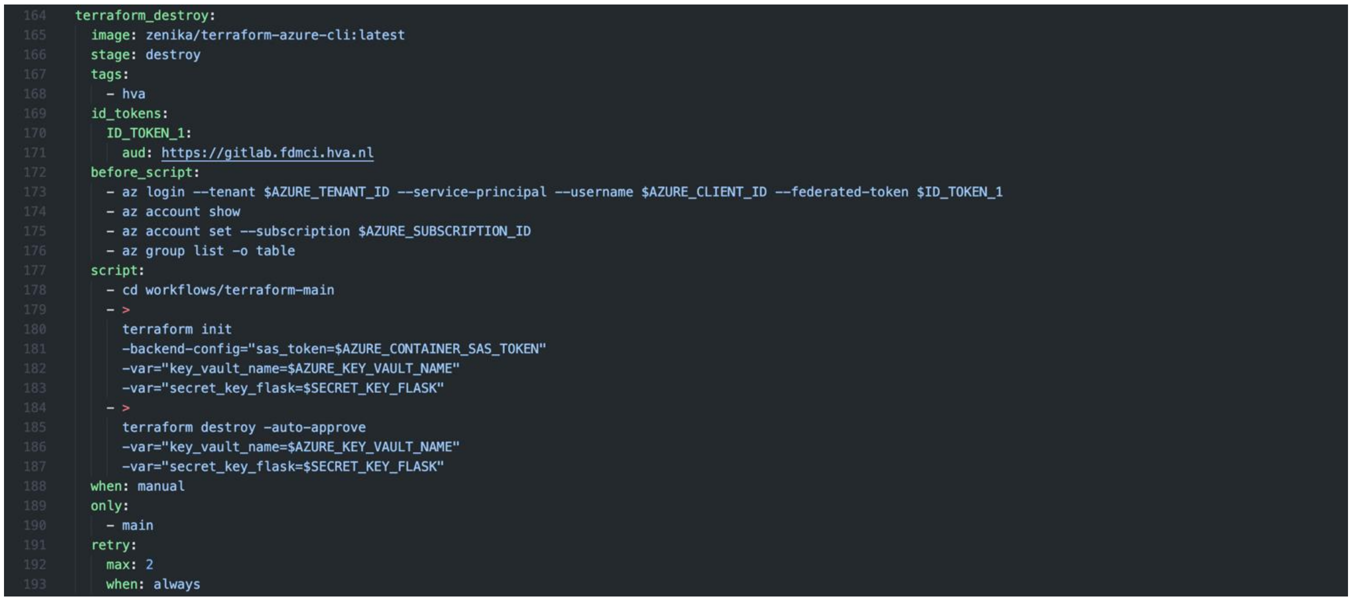
This stage includes a retry policy because the default timeout was sometimes insufficient for this job. Configuring the timeout in the pipeline did not work properly, so a retry was added. This issue can be investigated and potentially solved by properly configuring the timeout.

This stage performs the following steps:

- Authenticate using Azure CLI with a federated token (`ID_TOKEN_1`), (`AZURE_TENANT_ID`), and (`AZURE_CLIENT_ID`).
- Sets the subscription.
- Lists Azure resource groups to confirm successful authentication.

The main script executes Terraform commands within the `workflows/terraform-main` directory:

- Initialization (`terraform init`)
  - Configures the backend using the SAS token and sets variables for the key vault name and secret key.
- Destruction (`terraform destroy`):
  - Destroys the infrastructure, automatically approving the action and using the specified variables.
- The stage is set to manual (`when: manual`) .
- This stage triggers only on changes to the `main` branch (`only: - main`).
- Retry Policy
  - The stage includes a retry policy to handle cases where the default timeout is insufficient. The job will retry up to 2 times (`max: 2`) regardless of the failure reason (`when: always`).

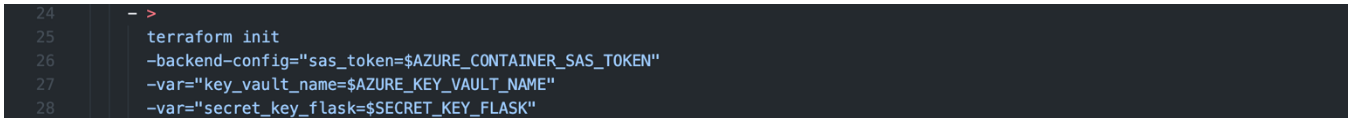


```yaml
terraform_destroy:
  image: zenika/terraform-azure-cli:latest
  stage: destroy
  tags:
    - hva
  id_tokens:
    ID_TOKEN_1:
      aud: https://gitlab.fdmci.hva.nl
  before_script:
    - az login --tenant $AZURE_TENANT_ID --service-principal --username $AZURE_CLIENT_ID --federated-token $ID_TOKEN_1
    - az account show
    - az account set --subscription $AZURE_SUBSCRIPTION_ID
    - az group list -o table
  script:
    - cd workflows/terraform-main
    - >
      terraform init
      -backend-config="sas_token=$AZURE_CONTAINER_SAS_TOKEN"
      -var="key_vault_name=$AZURE_KEY_VAULT_NAME"
      -var="secret_key_flask=$SECRET_KEY_FLASK"
    - >
      terraform destroy -auto-approve
      -var="key_vault_name=$AZURE_KEY_VAULT_NAME"
      -var="secret_key_flask=$SECRET_KEY_FLASK"
  when: manual
  only:
    - main
  retry:
    max: 2
    when: always
```

Figure 5.2.6 destroy stage.

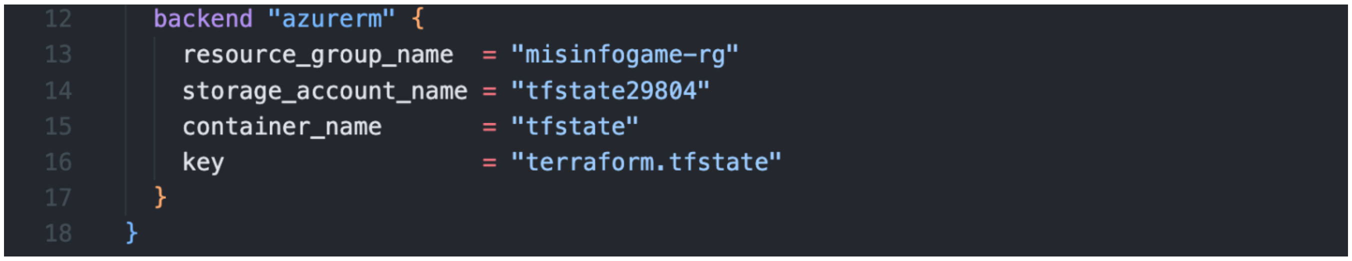
5.3. Terraform Backend configuration

Terraform uses a backend to manage the state file, which keeps track of the infrastructure resources it manages. In this .gitlab-ci.yml file, the backend is configured to use Azure Storage . Here's a snippet of how the Terraform backend is initialized:



```yaml
    - >
      terraform init
      -backend-config="sas_token=$AZURE_CONTAINER_SAS_TOKEN"
      -var="key_vault_name=$AZURE_KEY_VAULT_NAME"
      -var="secret_key_flask=$SECRET_KEY_FLASK"
```

This is how backend is defined in the Terraform script (see /workflows/terraform-main/providers.tf)



```hcl
  backend "azurerm" {
    resource_group_name  = "misinfogame-rg"
    storage_account_name = "tfstate29804"
    container_name       = "tfstate"
    key                  = "terraform.tfstate"
  }
}
```

The terraform init command initializes the working directory containing Terraform configuration files and sets up the backend configuration with a Shared Access Signature (SAS) token for Azure Storage. It uses the Shared Access Signature (SAS) token to authenticate and authorize access to the Azure Blob storage allowing

Terraform to read and write to the container. The token is generated with an expire date. In this case this date is 27.06.2029. To generate the token, you will need the following permissions:

- Read
- Write
- List

Using SAS tokens to authenticate Blob Storage is best practice because by using a SAS token, you can grant limited access to your storage account without exposing your account keys. However, SAS tokens have an expiration date and time, and they can be scoped to specific permissions, services, and resources. This makes them a secure option for temporary access, especially in automated CI/CD pipelines.

Here is an example of how to generate a SAS token for a blob container in Azure CLI:

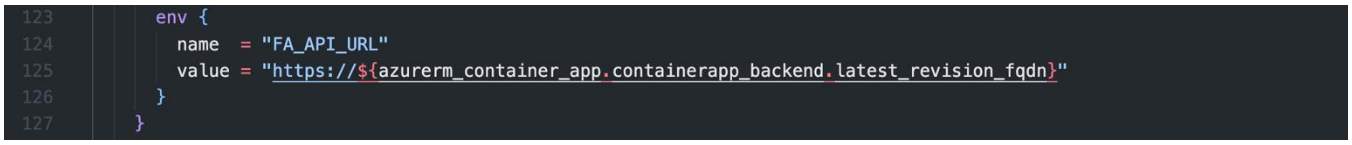
```
az storage container generate-sas \
   --account-name <your-storage-account-name> \
   --name <your-container-name> \
   --permissions rwl \
   --expiry 2024-07-01\
   --output tsv
```

- --account-name: The name of your storage account.
- --name: The name of the container.
- --permissions: The permissions to grant (r for read, w for write, l for list).
- --expiry: The expiration date and time of the SAS token.

5.4. GitLab APIs

The names and URLs of the Azure resources, three container apps in our case, will be slightly different every time deploy stage is executed. This is done because they should be globally unique, and we can't guarantee it by having a hardcoded names because at some point, they may be taken. Thus, we append a randomly generated number to the name of every container app.

Both frontend_admin and frontend_client require the URL of the backend. Frontend_admin can receive this URL straight in the Terraform and it will be passed as an environmental variable to the Docker container on a run.

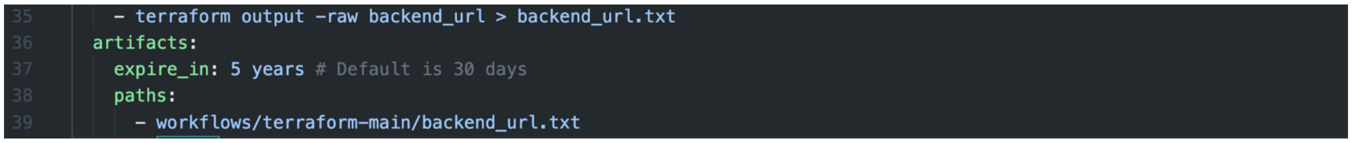


```
    env {
      name  = "FA_API_URL"
      value = "https://${azurerm_container_app.containerapp_backend.latest_revision_fqdn}"
    }
  }
```

Meanwhile, the same approach cannot be used for frontend_client because the React app is transpiled into regular JavaScript, which is compatible with the browser but does not allow passing environmental variables directly. Thus, the API logic was implemented.

Workflow of the URL retrieval:

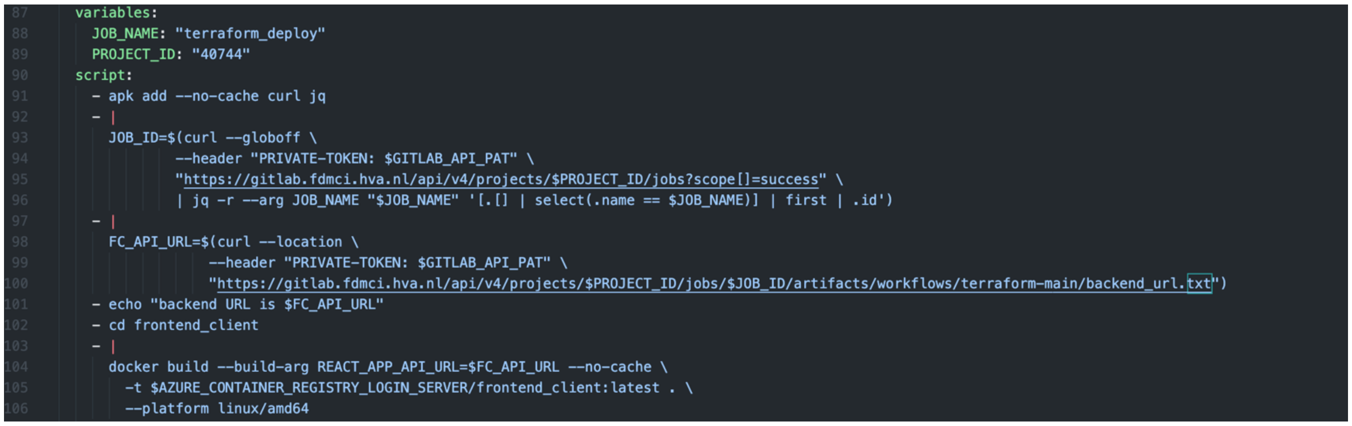
1. The URL is retrieved with the 'terraform output' command and stored in the backend_url.txt artifact.



```
      - terraform output -raw backend_url > backend_url.txt
    artifacts:
      expire_in: 5 years # Default is 30 days
      paths:
        - workflows/terraform-main/backend_url.txt
```

Since the pipeline is modular and it doesn't enforce to run all the stages on some change, different stages of the pipeline can be run on different commits. For example, on the first commit you run the 'deploy' stage and on the second commit you run the 'frontend_client_build_and_push'. Since the runner executes the cleanup script it will not be possible to just store this artifact and access it from different commits. This artifact could only be accessed from the commit where it was generated i.e., 'deploy' stage. Therefore, we had a need in the GitLab Job and Job artifacts APIs.

2. Then the GitLab Job API is used to list all of the jobs that resulted in success, and the jq is used to parse through the json and retrieve the id of the latest successful 'terraform_deploy' job.
3. GitLab Job artifact API is then used to get the URL out of the artifact using the id obtained by the previous API as one of the parameters.
4. The URL is then used as a build argument on a 'docker build'



```
  variables:
    JOB_NAME: "terraform_deploy"
    PROJECT_ID: "40744"
  script:
    - apk add --no-cache curl jq
    - |
      JOB_ID=$(curl --globoff \
              --header "PRIVATE-TOKEN: $GITLAB_API_PAT" \
              "https://gitlab.fdmci.hva.nl/api/v4/projects/$PROJECT_ID/jobs?scope[]=success" \
              | jq -r --arg JOB_NAME "$JOB_NAME" '[.[] | select(.name == $JOB_NAME)] | first | .id')
    - |
      FC_API_URL=$(curl --location \
                  --header "PRIVATE-TOKEN: $GITLAB_API_PAT" \
                  "https://gitlab.fdmci.hva.nl/api/v4/projects/$PROJECT_ID/jobs/$JOB_ID/artifacts/workflows/terraform-main/backend_url.txt")
    - echo "backend URL is $FC_API_URL"
    - cd frontend_client
    - |
      docker build --build-arg REACT_APP_API_URL=$FC_API_URL --no-cache \
        -t $AZURE_CONTAINER_REGISTRY_LOGIN_SERVER/frontend_client:latest . \
        --platform linux/amd64
```

When migrating to another repository, you must keep in mind that these APIs will require a few changes to work in your project.

1. Generate a new GitLab private access token. Note: Assigning 'api' scope for this token will be adequate if it will be used only for accessing the APIs. Also keep in mind, that the maximum life of token is 12 months.
2. Change the PROJECT_ID variable. To retrieve this variable, go to: Your repository → Settings → General, and on the right you will see you project id.

5.5. Miscellaneous

5.5.1. KeyVault

In the folder terraform-keyvault we use Terraform to create an Azure Key Vault. An access policy is setup for the current tenant granting permissions to manage secrets.

The permissions include:

- Get: Retrieve secrets.
- Set: Create or update secrets.
- Delete: Remove secrets.
- List: List all secrets in the Key Vault.

5.5.2. Depends_on

The depends_on meta-argument in Terraform is used to specify explicit dependencies between resources. It ensures that Terraform creates resources in a specific order, which is essential when resources rely on each other to function correctly. In our case we are creating docker containers that communicate with each other which makes it crucial to create them in the right order to prevent errors.

5.5.3. Terraform .gitignore

If you plan to run the Terraform scripts locally, ensure that you do not remove the Terraform .gitignore file stored in the root of the workflows directory. Removing this file could result in versioning the tfstate files, which store all credentials in plain text.

5.5.4. Runner error

HVA's runners are not perfect and can sometimes run out of memory, resulting in the following error:

```
/bin/bash: line 49: printf: write error: No space left on device
```

However, this issue only occurred once throughout the entire project, so it should not be a major concern.

5.5.5. Pipeline usage

The pipeline is meant to be used in the following way:

1. First you deploy the Azure resources with the 'deploy' stage.
2. Then you start working on the development of frontend_admin, frontend_client, and backend.
3. Then you build a respective Docker image/s.
4. Then you update the container app/s via 'update_container_apps' with a newly build Docker image/s.
5. Then when you finished with the development or hosting the test you run the 'Destroy'.

Few things to note:

- Build the frontend_admin container after you run the 'deploy'.
- If you want your container app to serve the latest image you pushed to the ACR, run 'update_container_apps' after you ran container build jobs.

5.6. Exclusion of ACR and KV

The ACR and Key Vault form the foundational components of our project infrastructure. They are essential for the operation and security of our application, the Misinformation Game. The contents of these resources need to be persistent to ensure consistent deployments. By keeping them outside the pipeline, we prevent unintended changes or loss of critical data.

Our Key Vault stores the secret key for Flask, which is crucial for the functionality of our Flask application. Without this key, the Flask app cannot function properly. By maintaining the Key Vault outside the pipeline, we minimize the risk of unauthorized access to this sensitive information.

Our ACR contains the latest Docker images for various components of our application:

- Backend
- Frontend Admin
- Frontend Client

By excluding these resources from the pipeline, we accelerate the deployment process and reduce the complexity of our CI/CD workflow. It also helps in cost management by avoiding unnecessary recreation of these resources in each pipeline run.