Software Architecture Report

Team Members

- Beşliu Radu-Ştefan
- Florea Mădălin-Alexandru
- Gheorghe Robert-Mihai
- Huțan Mihai-Alexandru

Purpose

Vision Canvas is a web application written in React, which uses a backend made in Python, that allows its users to edit images. Within the registration process, anyone can create a personal account or use Google to sign in, upload a photo and edit it.

The editor has a lot of complex and useful features, such as: cropping, adding shapes and text, color filtering, masking, resizing etc. Premium features include special algorithms that allow an user to remove objects, background and/or blur the image.

To get access to Vision Canvas Premium, a user can buy a lifetime subscription worth €9.99. The purchase is processed through Stripe, thus being very easy to track the amount of purchases, which user decided to purchase a subscription etc., also allowing the buyer to use multiple payment methods, such as, but not limited to, PayPal, Credit/Debit Card, Google Pay, Apple Pay.

Subscribing also gives users the option to save and delete images, thus allowing users to store and retrieve photos safely and securely. The profile and the images are stored in Firestore, part of Firebase, which is easy, intuitive and free to use for the purposes of our project.

Everything but the "Share" functionality has been implemented successfully and is ready to use.

Run the project locally

To run the project locally, the user needs to have Node.js installed on their system. The second step is to install all the required packages (which are listed in the **Libraries** section of this document) using ***npm install --force*** (the *--force* argument is required, since there is a package that is not compatible with the version of React we are using).

To gain access to most of the features, the *.env* file has to be present, since all the configuration variables are set there. Without it, the app is unusable.

If there is access to the list mentioned before, anyone can run the project using the ***npm run dev*** command.

Build the project

Building the project is rather easy. Running the command ***npm run build*** should suffice.

Deploy the project

Both parts of our project (frontend and backend) are deployed on Google Cloud, thus making it really easy to access from anywhere. The services are hosted on a European server.

For a successful deployment, there were several steps required:

- Firstly, we needed to ensure that the application was built correctly, running the commands above.
- The next step required us to access the variables stored in the *.env* file. This allows the application to communicate with Firebase and the backend, granting access to all of its features.

For local use, there are only 3 commands required, all of them mentioned above:

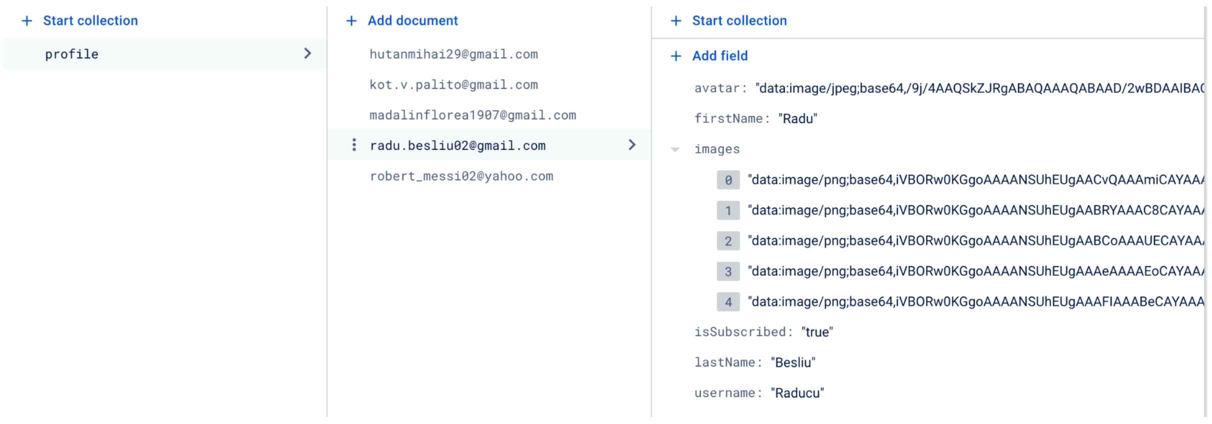
- ***npm install --force***
- ***npm run build***
- ***npm run preview***

Data sources

The data used in our application originates from multiple sources.

Firstly, we use Firebase to store all the required information about each user, their profile, authentication information and images stored.

An example of how the data is structured inside Firestore can be visualized in the following image:



```
profile

hutanmihai29@gmail.com
kot.v.palito@gmail.com
madalinflorea1907@gmail.com
radu.besliu02@gmail.com
robert_messi02@yahoo.com

avatar: "data:image/jpeg;base64,/9j/4AAQSkZJRgABAQAAAQABAAD/2wBDAAIBA
firstName: "Radu"
images
    0 "data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAACvQAAAmiCAYAA
    1 "data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAABRYAAAC8CAYAA
    2 "data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAABCoAAAUECAYAA
    3 "data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAAeAAAAEoCAYAA
    4 "data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAAFIAAABeCAYAAA
isSubscribed: "true"
lastName: "Besliu"
username: "Raducu"
```

We have one collection, named **profile**, which contains all information about each user. The key for each document in this collection is represented by the **email** of a user, thus allowing a person to keep all their information, even if they login using email & password or through Google.

All images (user avatar and saved images) are stored in base64 format, public/private buckets not being needed. The problem using this way of storing all of the photos is represented by the limit of 1MB for each key.

Other fields are represented by the: username, first name, last name, and a boolean that shows if the user is subscribed to the premium version or not.

To access all the data from Firestore, there are a couple of needed variables. These include: API key, auth domain, database URL, project id, messaging sender and the app id. Having all of these variables set in an environment file grants full access to the features.

Secondly, data is also returned by the backend. We have 3 functions available: object removal, background removal and blur. Each of them takes in a common argument: the image, in base64, and other specific ones (e.g. blur value, blur type, object removal coordinates). Each function returns the same type of data: either an error, or the modified image, in base64.

Data inputs

The application presents various data inputs. Most of the data is given by the user.

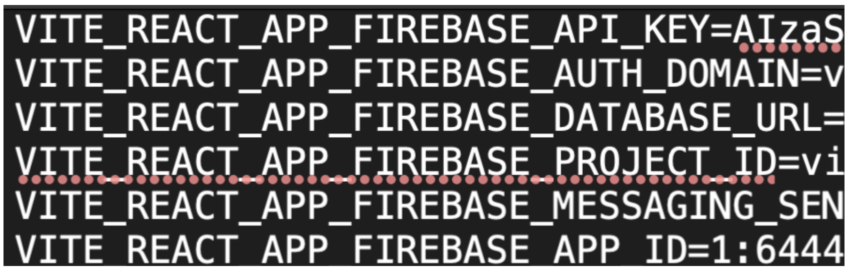
Firstly, we have a couple of forms represented by register, login, forgot password, and account details. Each of these forms has specific validation implemented. For example, registering requires a password of at least 6 characters long and a valid email. Logging in with no data and/or invalid data would trigger specific errors, thus ensuring the application cannot be accessed with incorrect information. The account details form has multiple fields, such as: first name, last name and username. Users are required to complete all of these fields. If a username is not provided, it will be set to a concatenation between the first name and last name.

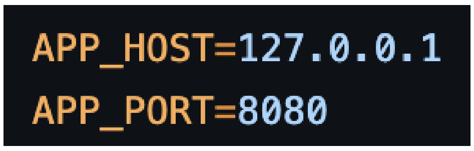
Secondly, users can upload images. There are only two places where this is possible: “Upload Avatar” and “Upload Image” (within the Dashboard). The “Upload Avatar” functionality is self-explanatory and has no limits set by us; however, there is a constraint imposed by Firebase for one field, as mentioned earlier. This limitation is due to the conversion of the image into base64 format and subsequent storage. For the “Upload Image” functionality, there is a validation that checks if the image is less than 5MB. We allow users to upload images larger than 1MB, since they are not saved automatically, only when the “Save” button is clicked explicitly. Until then, images remain editable, and users can download their edited photos at their convenience. This approach serves as a practical workaround for the limitation imposed by Firestore, as most users tend to choose to download rather than save their images.

The last 2 forms of data input are represented by Firestore and the API, respectively (both presented in the previous section - Data Sources).

Configuration files

Our application uses a couple of configuration files, each serving specific purposes. Below, they are presented for both frontend and backend, respectively:


```
VITE_REACT_APP_FIREBASE_API_KEY=AIzaS
VITE_REACT_APP_FIREBASE_AUTH_DOMAIN=v
VITE_REACT_APP_FIREBASE_DATABASE_URL=
VITE_REACT_APP_FIREBASE_PROJECT_ID=vi
VITE_REACT_APP_FIREBASE_MESSAGING_SEN
VITE_REACT_APP_FIREBASE_APP_ID=1:6444
```


```
APP_HOST=127.0.0.1
APP_PORT=8080
```

All of the variables used for the frontend have been mentioned in a previous section. All of them are related to Firebase.

The backend configuration file includes two basic variables: the host and port, which allow this service to run.

User journey

A user journey map can be accessed at this link.

Most valuable outputs

The most valuable outputs of our application are best demonstrated through our user personas, accessible via the links provided here and here. Therefore, the true value lies in what users can create using the features we've developed. The user experience, coupled with a range of editing features, has the potential to drive subscription purchases. Subscribers gain access to premium features, enabling them to enhance their photo editing capabilities. This, in turn, supports our ongoing development efforts, allowing us to create new features and further improve existing ones.

Deployment plan

The frontend pipeline has two stages: build and deploy. The build stage, using code from the pull request (PR), includes two types of jobs: *build_pr* and *build_main*. Additionally, it loads necessary variables for Google Cloud login and secret variables for the frontend, as presented earlier. It builds the image and pushes it to Google Artifact Registry, with the specific tags (commit SHA for PR and *latest* for main branch). If the build is successful, the app is deployed to Google Cloud Run.

On the other hand, the backend pipeline consists of four stages: lint, test, build and deploy. The linting stage ensures that the code respects the PEP8 standard, promoting uniformity in code writing across developers. The testing stage validates that code reaching production passes all available tests, preventing unexpected bugs from reaching our users. The remaining stages are similar to the corresponding ones in the frontend pipeline.

QA process

The QA process was divided into two categories: manual and automated testing.

Manual testing involved each team member navigating through all possible flows and notifying the team when a bug was found. The next step included identifying a solution to the problem and prioritizing it based on its impact on the entire application. For instance, we encountered a bug during one of our reviews that constantly redirected the user to the editing page, even if they wanted to quit. This bug had the highest priority, since the main feature of the application was not stable. The bug was solved changing the logic of how images are stored in the context of the app and how they are removed when the page (component) is unmounted.

Automated testing consists of end-to-end testing and unit testing. **End-to-end** testing simulates calling endpoints from the perspective of a user, with different payloads. We tested all possible inputs and outputs for our endpoints, ensuring everything works as expected. **Unit testing** ensures that each of the endpoints calls its utility and service functions, correctly and in order.

External APIs used

The external APIs used in our project consist of:

- Firebase API
- Stripe API.

Libraries

The project makes use of multiple libraries, essential for both the frontend and backend. Below, we'll describe some of these key libraries that contribute to the project's functionality and performance.

For the frontend part of the application, multiple packages were used to help us achieve our goal:

- **Material UI**: introduces pre-designed reusable components, great for creating a quick layout.
- **Styled Components**: another useful styling library which does exactly what it says: it allows us to create styled components by writing CSS, offering flexibility in usage.
- **React Google Button**: a small yet useful package which creates a component that includes the Google logo, used for logging in.
- **Stripe**: handles payments and utilizes dependencies to access functions from the Stripe API for payment processing.
- **Firebase**: used to access the Firebase API, as mentioned previously.

- **ToastUI Image Editor**: the core feature of our application, providing various editing capabilities described in the application overview.
- Dev-related dependencies include: **Vite**, **TypeScript**, **ESLint**, **Prettier**.

For the backend part of the application, we rely on libraries across four categories:

1. Serving as our web framework is **FastAPI**, chosen for its lightweight nature and high performance. While Django and Flask were considered, neither offered the capability to create asynchronous endpoints, a feature that FastAPI provides, resulting in significant performance improvements.
2. For image processing we used the following computer vision specific libraries: **NumPy**, **OpenCV-Python**, **Pillow** and **Rembg**.
3. For testing we used **pytest**, **pytest-asyncio** (for asynchronous testing), **pytest-cov** (for coverage reports) and **HTTPX** (for simulating a client in end-to-end testing).
4. Dev-related dependencies are **Poetry**, **Poe The Poet** and optionally **pre-commit** (which runs our preconfigured checkers for the staged code).

Vulnerabilities

- **Injection Vulnerabilities**: Packages that handle user input, such as **express** for handling HTTP requests, could be vulnerable to injection attacks if not used properly or if they contain vulnerabilities themselves.
- **Cross-Origin Resource Sharing (CORS)**: The **CORS** package, when improperly configured, can lead to security issues, especially if it's too permissive.
- **Outdated packages**: The presence of outdated packages is a potential concern, as known vulnerabilities, that are patched in updated versions, may affect our website.