Solveig Ranøien Engen
Martin Heggholmen
Guro Eline Karlsen

Assignment 1 - IDG2001

File processing web service - Report Group 3

What choices we made during development

We chose an asynchronous file handling system “aiofiles” for faster handling, and preventing the program from having to wait for the finished processing of files.

For the ASGI server we chose Hypercorn, also because of its asynchronous. But we might as well have gone with Uvicorn, they both serve similar purposes, and it doesn’t seem to matter which we chose for this specific project.

For the database we chose mongoDB just because of its simplicity and json-like formatting, we found it an easier database to work with than for example postgreSQL and SQL.
We could have hosted mongoDB in railway, but we chose to host it on atlas instead, and link it to fastAPI in railway, because you seem to have more control over who can connect and other factors in atlas, even if it provides the same service.

For the hosting we chose railway, because of its ease of use and setup, and the fact that you can deploy straight from the github repo, and receive live updates when the git is pushed. The

We chose FastAPI just because it's fast, asynchronous, and handles uploads and processing efficiently. While an alternative to this would be Flask, we found that FastAPI works better for async. Also it seems that FastAPI has better documentation.

How it works

The system works by creating a folder to put the files in first, if these does not exist already, by using the os.makedirs command
Then the process_csv function reads the uploaded csv file asynchronously, processes its content, generates .md files, and saves them to the processed file folder.
the .md files will be created from the template_content variable.
After processing the function creates a zip file with all the md files in it, and saves it to the zip_files folder

when a get request is made to /download-zip the server responds with the corresponding zip file if it exists, and downloads the file to the computer.

First we import the dependencies, here we used fastAPI for ASGI, jinja for template handling, we used uuid for generating names for the zipfile, but we ended up just calling the zipped files output.zip, so we don't really need it.

```python
# import dependencies
from fastapi import FastAPI, UploadFile, File, HTTPException
from fastapi.responses import FileResponse
from fastapi.staticfiles import StaticFiles
from jinja2 import Environment, FileSystemLoader, select_autoescape
import os
import csv
import aiofiles
import zipfile
from uuid import uuid4
import shutil
```

Afterwards we defined the directory paths.

```python
# Initialize the FastAPI application
app = FastAPI()

# Define the directory paths for various stages of file handling
UPLOAD_FOLDER = 'uploaded_files'
PROCESSED_FOLDER = 'processed_files'
ZIP_FOLDER = 'zip_files'
TEMP_FOLDER = 'temp_files'
```

create the folders if they do not exist

```python
# Ensure the directories exist, or create them if they do not
os.makedirs(UPLOAD_FOLDER, exist_ok=True)
os.makedirs(PROCESSED_FOLDER, exist_ok=True)
os.makedirs(ZIP_FOLDER, exist_ok=True)
os.makedirs(TEMP_FOLDER, exist_ok=True)
```

This makes sure that the files that are being processed is temporarily stored in the temp folder for processing.

```
# Set up the Jinja2 environment for template rendering
# Templates are loaded from TEMP_FOLDER where uploaded ZIP contents are extracted
env = Environment(
    loader=FileSystemLoader(TEMP_FOLDER),
    autoescape=select_autoescape()
)
```

this transforms the csv to markdown files with the template

```
# Process uploaded CSV and template to generate markdown files and zip them
async def process_csv_and_template(csv_file_path: str, template_name: str, file_id: str) -> str:
    processed_files = []
```

we used jinja for template handling here

```
    # Load the Jinja2 template by name
    template = env.get_template(template_name)
```

this generates the markdown file, opens and processes each row of the csv file,

generates the name for the .md files

renders the content.

```
# Process uploaded CSV and template to generate markdown files and zip them
async def process_csv_and_template(csv_file_path: str, template_name: str, file_id: str) -> str:
    processed_files = []

    # Load the Jinja2 template by name
    template = env.get_template(template_name)

    # Open and read the CSV file
    async with aiofiles.open(csv_file_path, 'r', encoding='utf-8') as csvfile:
        content = await csvfile.read()
        reader = csv.DictReader(content.splitlines())

        # Process each row in the CSV file
        for row in reader:
            # Generate the filename and file path for the markdown file
            md_filename = f"{row['FirstName']}_{row['LastName']}.md"
            md_file_path = os.path.join(PROCESSED_FOLDER, md_filename)
            # Render the markdown content from the template with row data
            markdown_content = template.render(row)

            # Write the rendered content to the markdown file
            async with aiofiles.open(md_file_path, 'w', encoding='utf-8') as md_file:
                await md_file.write(markdown_content)
            processed_files.append(md_file_path)
```

This creates a zip for all the markdown files that just have been processed.

```
    # Create a ZIP file of all processed markdown files
    zip_filename = f"{file_id}.zip"
    zip_file_path = os.path.join(ZIP_FOLDER, zip_filename)
    with zipfile.ZipFile(zip_file_path, 'w') as zipf:
        for file_path in processed_files:
            zipf.write(file_path, os.path.basename(file_path))

    return zip_file_path
```

Again we dont use the unique ID, but , this is the endpoint we use to upload a zip that contains a markdown and a csv. This will only take .zip and no other format.
here we used aiofiles for input/output operations, so the pipeline will not get blocked.

We use zipfile to extract the template and the csv file, process the template and post the results

```
# Endpoint to upload a ZIP file containing a markdown template and a CSV file
@app.post("/upload-zip/")
async def upload_zip(file: UploadFile = File(...)):
    # Ensure the uploaded file is a ZIP
    if not file.filename.endswith('.zip'):
        raise HTTPException(status_code=400, detail="Invalid file extension.")

    # Generate a unique ID for the file and define its path
    file_id = str(uuid4())
    zip_path = os.path.join(UPLOAD_FOLDER, f"{file_id}.zip")

    # Save the uploaded ZIP file
    async with aiofiles.open(zip_path, 'wb') as out_file:
        content = await file.read()
        await out_file.write(content)

    # Extract the ZIP to get the template and CSV file
    with zipfile.ZipFile(zip_path, 'r') as zipf:
        zipf.extractall(TEMP_FOLDER)
        for file_name in zipf.namelist():
            if file_name.endswith('.md'):
                template_name = file_name
            elif file_name.endswith('.csv'):
                csv_file_path = os.path.join(TEMP_FOLDER, file_name)

    # Process the template and CSV, then zip the results
    zip_file_path = await process_csv_and_template(csv_file_path, template_name, file_id)
```

```python
    # Process the template and CSV, then zip the results
    zip_file_path = await process_csv_and_template(csv_file_path, template_name, file_id)

    # Clean up the temporary folder
    shutil.rmtree(TEMP_FOLDER, ignore_errors=True)

    return {"file_id": file_id}

# Endpoint to download the processed ZIP file named 'output.zip'
@app.get("/download-zip/{file_id}")
async def download_zip(file_id: str):
    # Construct the file path and serve it if it exists
    zip_file_path = os.path.join(ZIP_FOLDER, f"{file_id}.zip")
    if os.path.exists(zip_file_path):
        return FileResponse(path=zip_file_path, media_type='application/zip', filename="output.zip")
    raise HTTPException(status_code=404, detail="File not found")

# Serve static files, useful for frontend applications
app.mount("/", StaticFiles(directory="static", html=True), name="static")
```

we also have an endpoint to download the finished output.zip, if the file exists, if it doesn't exist we get a 404 not found error.

finally we serve the static files to / so we can use it all from one static page.

How to use

To run the website you can simply upload a .zip file to the railway server hosted on:
the zip file must contain a .md file and a .csv file
https://fastapi-production-c667.up.railway.app/
(if you don't have a file of your own, we have provided different zipped folders within the assets folder for testing.)
after you select a file press upload, and a zipped file with the personas you have uploaded in your csv file will be added to .md files, one for each persona, zipped up and automatically downloaded in your browser, congratulations you now have a zip file with .md files in it.
if you don't want to run it on railway you can run it locally with these commands

Install the requirements:
pip install -r requirements.txt

Run the app:
uvicorn main:app --reload

Solveig Ranøien Engen
Martin Heggholmen
Guro Eline Karlsen

How to setup a system like this?

First you need python installed on your system, you can use FastAPI like we did in this case, but you may also use Django, Flask, Tornado or other options, you can chose the system that works best for you, there are many alternatives to suit your needs in the Python universe. You can decide if you want to host the application online with Railway, Heroku, Vercel, and many more.

For the ASGI servers to server web applications you can use Daphne, Gunicorn, Uvicorn, Hypercorn or many others. For this project we hosted a railway server with fastAPI from a github repo, so everything was close to ready to go from the beginning. We entered the dependencies we needed in requirements.txt, and used main.js for all the file handling, and index.html with html and css to server the static website, and style.css for styling the webpage.

We made sure to install all the dependencies with
pip install -r requirements.txt
and tested the app with
uvicorn main:app --reload
since the railway app is hosted directly from the github folder what happen is that when we push the main branch to github it relaunches the railway project automatically to display the latest code, it uses the dependencies declared in the dependencies.txt

Other use cases for a system like this could be to for example swap out the handling of the csv files with image handling and manipulating. You can get it to process images with tasks such as resizing, cropping or applying different filters to the images. You have to change what process the file is supposed to go through as well as what is being uploaded depending on what content you want. The easiest replacing code for a new use case would be generating reports, invoices, or certificates automatically from structured data. You could generate all kinds of file formats such as pdf, word and convert from one file type to another with some changes to the code. A code like this could also be used for data visualization and analysis. Instead of generating documents, the system could analyze data provided in the uploaded files and generate visualizations such as charts, graphs, or interactive dashboards. This could help users gain insights from their data and make informed decisions.