UCBoulder Big Data final project

This report documents the design, development and deployment of a complete web application. The application collects data from a public API, transforms the data (using a text to speech algorithm), and presents this data to users over the Internet in the form of a website or through a REST API as a JSON document.

The application provides an audio version of recent research papers. It watches for new research papers on https://arxiv.org/, generates audio of the summary of the paper, and then stores the information about the paper and the generated audio version. This data is accessible using a public website at the following URL.

Public URL

https://ucboulder-bigdata-final-da5143962440.herokuapp.com/

Repository URL

https://github.com/tinybeachthor/ucboulder-bigdata-final

Requirements

The app will transform research papers into an audio format and make it accessible on a website. The requirements for the system are as follows.

- Functional
- Non-functional

Functional

1. The app must be accessible through a website.
2. The app must pull new research papers from the source publication.
3. The app should transform the data to an audio format using a text to speech algorithm.
4. The app should present the recent research papers in text and audio format.
5. The app should present the most recent papers first.
6. The app could provide a REST API endpoint for programmatic access.

Non-functional

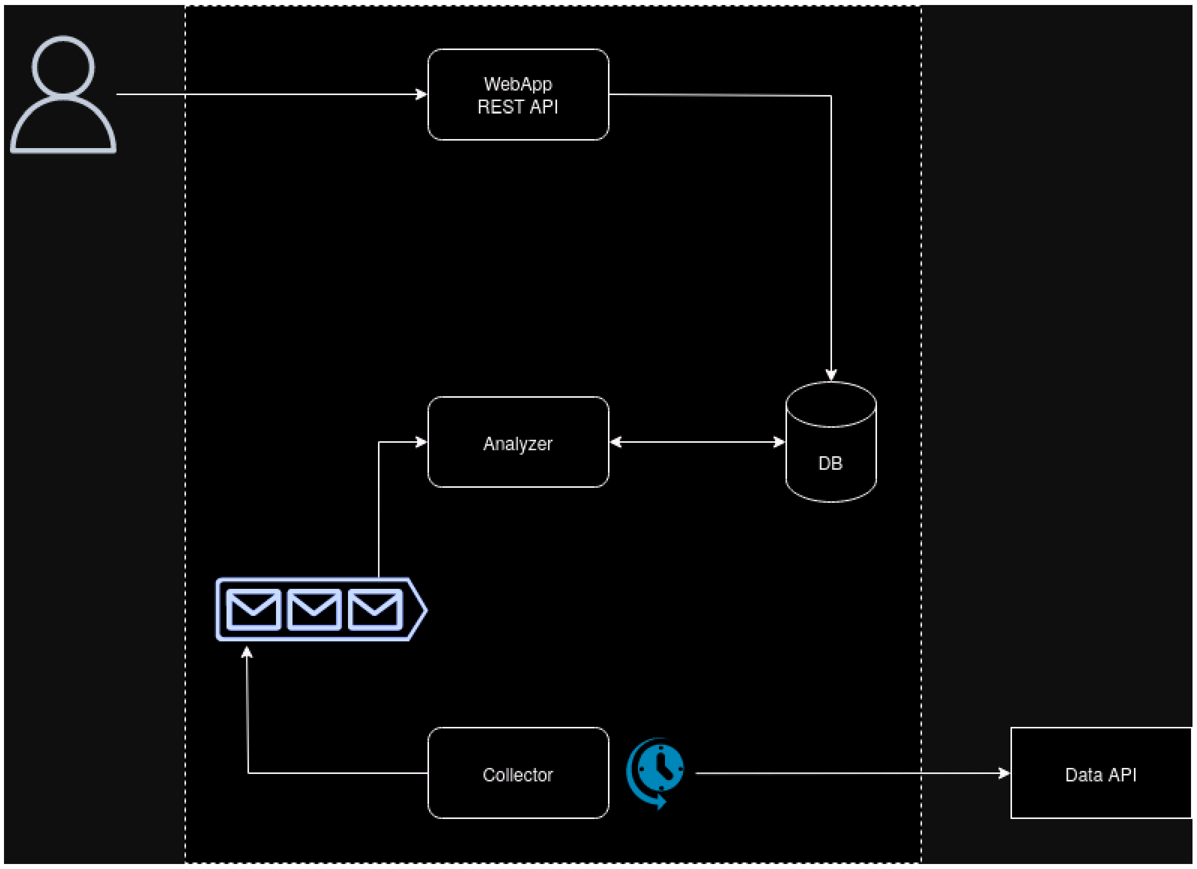
1. The app must not collect any personal user data.
2. The app should respond with fresh data within a reasonable time.
3. The app should be able to scale up and down dynamically to handle spikes in traffic as needed.

Architecture

Architecture diagrams and rationale of the services used for the app.

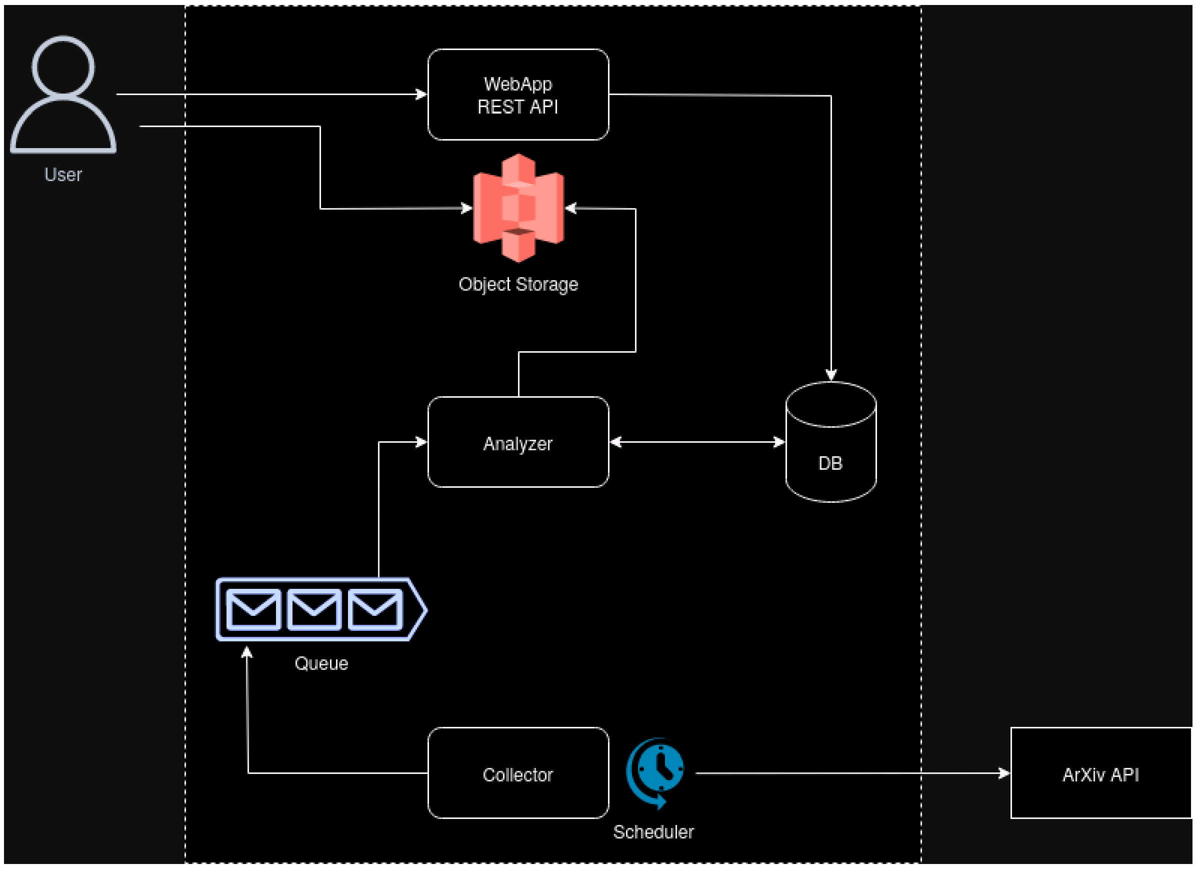
Whiteboard

Here is a diagram of the planned architecture of the app at the beginning.



Final

Diagram of the final architecture of the app.



Components

User

The user interacting with the app over the Internet.

Web app / REST API

The presentation component of the app. Handles all interactions from the users. Presents the most recent articles to the user. Handles requests for older articles through the REST API.

Database

Durable storage of information about the ingested articles. Keeps an index of the timestamps of the articles. Used by the web app to retrieve the most recent articles.

Data API / ArXiv API

External service. Used to load data into the app.

Collector

A job process to load the most recent data from data API.

Scheduler

Invokes the collector job periodically to load recent data into the queue.

Queue

Message queue used to provide loose coupling between the collector and analyzer jobs. Collector pushes messages about the new articles available. Analyzer(s) pull messages from the queue and processes them.

Analyzer

The main processing service. Read data from the queue. Executes text2speech transformation on the data. Stores data in the datastores: audio in the object storage, metadata in the database.

Object Storage

S3-like data storage. Used to store audio files which are too big for the database. Users get the audio data directly from the object storage at a URL generated by the web server.

Changes

A quick rundown of the changes between the planned and realized architecture.

Object Storage

Object storage was added to store audio files. Has the added benefit of an integrated CDN. Users can request data files directly from the object storage. This keeps the work load on the web app to a minimum as all it needs to do is return URLs to the audio files in the object storage.

arXiv podcast - UCBoulder big data final project

Audio arXiv - papers as a podcast

Overview

1. Pull new computer science papers from arXiv
2. Synthesize speech from the text
3. Present as a website with the option to play the audio

Setup

Setup python environment

Create python `venv`. Install runtime dependencies. Install test dependencies.

```
python -m venv venv
pip install -r requirements.txt
pip install pytest
```

Setup development infrastructure

Spin up the dockerized infrastructure. Setup database tables using `flyway`.

```
docker compose up
make setup
```

Run the app

Run the data collector. This is a job that will pull the latest articles from arxiv API and push them to a work queue, then exit.

```
python main.py collect
```

Run the data processor worker. Get item from work queue, run text2speech, push the audio to object storage, insert article info to database.

```
python main.py process
```

Run the web app + API server.

```
flask --app main.py run
```

Testing

All tests are run using `pytest`. Tests are automatically discovered by the test runner.

```
pytest
```

Unit

Unit tests are co-located next to the file being tested. Test files have a `_test.py` suffix. For example: `database.py` has unit tests in `database_test.py` in the same directory.

Integration

They are located in the `/integration` directory under the root of the repository. All integration tests are in this directory.

Design decisions

Service architecture

The flow of data is as follows:

- `collect` job
- `queue`
- `process` worker
- `database` & `object storage`
- `web` server
- user through a website

Using the queue to decouple `collect` job and `process` workers allows extra flexibility with scheduling the collect job and scaling the number of workers as necessary.

All the processing is done asynchronously in the `process` worker and the `web` server only returns data in a presentation format.

Data stores

PostgreSQL

This is a well tested production SQL database. We chose it for the ease of deployment and good availability of documentation.

SQL database provides a very good access to indexed data, we use it to retrieve the most recent articles by date.

Object Storage

To store the generated audio files, we use an S3-like object storage. Specifically Cloudflare R2, it has an integrated CDN and no fees for egress traffic, making it a great economical choice to serve large amount of large files to users.

Deployment

Heroku provides a flexible easy to use environment with built-in system metrics and auto scaling, as well as continuous delivery integration with GitHub repositories.

Rubric

A quick overview of where to find all the graded items. Explanation of how all the graded items were addressed.

Web app

Web server handling the presentation of the data to users. It is located in `apps/web.py` . It is implemented using `flask` web framework and in production served through `gunicorn` .

Data collection

The job collecting new data. Located in `apps/collect.py` . It is invoked every hour using the `heroku scheduler` .

Data analyzer

The worker processing collected data. The main part is running text to speech and storing data and audio in database and object storage, respectively. The code is in `apps/process.py` .

Unit tests

Unit tests are co-located with the files they test in the same directory. For example, `components/database.py` has unit tests in `components/database_test.py` . Unit tests are discovered and run using `pytest` .

Data persistence

Data about the articles is stored in a PostgreSQL database. The data is indexed by publication date and allows retrieval ordered by the date.

The audio files are stored in an S3-like block storage, specifically Cloudflare R2 which allows direct public access through a CDN network.

Rest collaboration or API endpoint

The web server (`apps/web.py`) also exposes a REST API endpoint. This API is used to handle a 'load more' request from the website to load older articles. The API is also publicly accessible and can be used for programmatic access to the service.

Product environment

Production deployment in Heroku. Using `docker-compose` to create a local development environment with all the services running locally in docker: database, S3, rabbit queue, prometheus, grafana.

Integration tests

Integration tests are in the `integration` directory. They are discovered and run using `pytest`.

Mocks or test doubles

Mocks and Spies are used extensively throughout unit tests. For example, in `components/database_test.py` a database `MockCursor` is created to mimic the data returned from the database. Database access functions are then tested against this mock object.

Continuous Integration

Using `GitHub Actions`. The configuration is in `.github/workflows/test.yml`. All the unit and integration tests are run on every push to the repository.

Production monitoring

To collect application metrics (processing time, request counts, ...) we utilize prometheus metrics and expose them on the `/metrics` endpoint.

For system metrics (CPU usage, RAM use, restarts, ...), we utilize the built in Heroku monitoring dashboard.

Event collaboration messaging

The `collect` job places new articles found into a rabbit message queue. The `process` worker is pulling items from this queue and handling them. This way we can easily scale up the number of workers as needed.

Continuous Delivery

Using `Heroku <-> GitHub` integration. After every new push, if continuous integration passes, a new deployment is created.