# Data Gandalf Developer Guide

This document serves as a repository of knowledge of how a developer would get up and running with the Data Gandalf application. It also serves as a guide for future development and extensions to the project. This document will be divided up in the different systems that the project consists of. Data Gandalf is divided into three large parts. There is the main web application, a data subsystem, and a model subsystem.

## Main Web Application

The main web application is user facing and allows for browsing all the various datasets. It consists of the database, backend, and frontend. For quick and easy deployment, Docker can be used, and a Docker Compose file compose.yaml to spin up all all the containers needed. Assuming Docker and Docker Compose are installed, this can be done with

```
docker compose up -d
```

### Database

Before the backend can run, a database needs to be running for it to connect to. PostgreSQL is the database that is supported by the project. The database should be populated with the information from database. The information for a production deployment is in pg_dump.sql while example.sql is for testing purposes.

### Backend

The backend is used for the business logic of main web application. The code for this resides in the backend folder. All the backend needs to run is a version of Python 3 and its Python dependencies that are listed in backend/requirements.txt. These dependencies include FastAPI and Microsoft Recommenders. Assuming the current working directory is backend, these dependencies can be installed with

```
python3 -m pip install -r requirements.txt
```

The backend can then be run with

```
python3 -m uvicorn backend.main:app --host 0.0.0.0 --port 8080
```

The backend unit tests are implemented using pytest with the pytest-cov plugin for coverage. To run these tests, enter

```
python3 -m pytest --cov=backend --cov-branch tests -rA
```

The following subsections will delve into the details of the files that comprise the backend.

**backend/backend/config.py** This file holds global constants that are used to configure the backend. Some of these configuration values can be overridden using environment variables.

**backend/backend/db.py** This file contains functions that can be called to complete CRUD operations with the PostgreSQL database.

**backend/backend/main.py** This is the entrypoint to the backend and where the application starts off from. Inside this file is a list of FastAPI REST endpoints that can reached by the frontend. New REST endpoints can be added here by simply adding an annotated function.

**backend/backend/models.py** This file contains the Pydantic schemes used by the REST API for the POST endpoints.

**backend/backend/recommender.py** This file contains the logic to load in a recommendation model from a serialized file. The model can be called to perform rankings on datasets.

**backend/models** This folder contains the serialized models used for ranking.

**backend/tests** This folder contains the corresponding unit test code for the backend.

**backend/Dockerfile** This file is used to build a container for the backend.

### Frontend

The frontend uses React and the Next.js frameworks to render a web page using the data from the backend. The code for the frontend resides is the frontend folder. In order to run the frontend, Node 20 or later must be installed. The dependencies are listed in frontend/package.json. Assuming the current working directory is frontend, these dependencies can be installed with

```
npm install
```

The frontend can then be run with

```
npm run dev
```

The frontend end-to-end tests are implemented using Playwright. To run these tests, enter

```
npx playwright test
```

To run the tests in UI mode, the `--ui` flag can be added.

```
npx playwright test --ui
```

The following subsections will delve into the details of the files that comprise the frontend.

**frontend/.github/workflows/playwright.yml** This file contains the GitHub actions to run the Playwright system tests on pushes to main as well as pull requests to main.

**frontend/app** This folder contains all of the React code that makes up the frontend. Adding pages to the web application can be done here. In addition to two subfolders, it includes the following files:

- globals.css: A CSS file defining the global style for the web application.
- layout.tsx: React code applied to all pages in Data Gandalf.
- page.tsx: The homepage for Data Gandalf.
- utilities.tsx: A bundle of useful TypeScript functions.

**frontend/app/components** This folder contains reusable React code that can be used for the various pages. More components can be added here. It includes the following files:

- filterBar.tsx: A component for filtering datasets.
- grid.tsx: A wrapper component for grid items.
- gridItem.tsx: A component for displaying a single dataset in the grid.
- gridItemLarge.tsx: A component for showing a lot of data about a dataset.
- loadingIcon.tsx: A component containing an SVG for loading.
- rating.tsx: A component for users to rate recommendations.

**frontend/app/dataset/[dataset]/page.tsx** This is a dynamic route, a page whose path is determined at runtime, that displays info about a specific dataset and gives recommendations related to that dataset.

**frontend/public** This folder contains public assets for the web application to use such as favicons.

**frontend/tests** This folder contains the Playwright system test cases.

**frontend/Dockerfile** This file is used to build a container for the frontend.

**frontend/next.config.js** This file is used to make any configurations to Next.js.

**frontend/tailwind.config.ts** This file is used to make any configurations to Tailwind CSS, the library used to style the web application.

## Data Subsystem

### Data Workflow

1. Follow instructions in "Kaggle CLI Setup" section to prepare environment.
2. (Optional) Modify `main.py` file with configuration. Configuration in file contains default values that can be overridden with command-line arguments:
   - STAGES: Array containing combination of "FETCH", "EXTRACT", "UPLOAD", defaults to all three.
   - SAVE_CSV: True or False, default False. Whether csv files of downloaded data should be saved.
   - SAVE_METADATA: True or False, default False. Whether metadata files extracted should be saved. Set to True and modify STAGES if you want to manually examine JSON before uploading.
   - TOPICS: Array containing list of topic strings. If "FETCH" is in STAGES, specifies which topics to query kaggle for. If "EXTRACT" is in STAGES, specifies which topics (folders in `datasets/`) to extract metadata from. If "UPLOAD" is in stages, specifies which topics (folders in `metadata/`) to upload.
   - DATASETS_PER_TOPIC: Integer. If "FETCH" is in STAGES, specifies how many datasets to pull from kaggle from each topic.

   The command line arguments below are grouped by their effect. If one of the options under a bullet point is included in the command, default settings above from the script file will be completely overridden by the values from the CLI. Any of these can be combined.

   - STAGES: -f, -e, -u. If any of these are specified, only the listed stages will be run by the system. To run all stages with otherwise default settings, `python main.py -f -e -u`
   - SAVE_CSV: -c. If included, sets SAVE_CSV to True.
   - SAVE_METADATA: -m. If included, sets SAVE_METADATA to True.
   - TOPICS: -t {topic1} -t {topic2} –topic {topic3}. If included, overrides default topics. For example, `python main.py -t sports --topic housing -t academics` will set TOPICS=["sports", "housing", "academics"].
   - DATASETS_PER_TOPIC: -n, –num_datasets. If included,overrides DATASETS_PER_TOPIC. For example, `python main.py -n 3` will pull 3 datasets from each default topic.
3. Run `python main.py` from `data_system`.
   - Data will be pulled from kaggle if "FETCH" is in stages.
   - Data will be analyzed and extracted to JSON if "EXTRACT" is in stages.
   - Database will now populate if "UPLOAD" is in STAGES.
4. (Optional) To serialize existing database, view "DB Dump and Load Process" below.

### Kaggle CLI Setup

These steps are necessary when running the 'data_fetching" component.

1. Go to https://www.kaggle.com/docs/api
2. Register.
3. Run `pip install kaggle` in the virtual environment.
4. Follow steps in https://www.kaggle.com/docs/api?rvi=1 to get authentication token and store on local computer.

**Description/Design**

The data subsystem is meant to pull existing datasets from public sources, structure the data, and upload data to a configured database.

The system is broadly split into three components:

1. Data Fetching
2. Metadata Extraction
3. Data Uploading

All of which are used in sequence by the `main.py` script.

**1. Data Fetching** The data fetching component (found in `data_fetching` folder) uses the Kaggle command line interface to pull a configured number of public datasets with a given topic. Each dataset pulled is stored in its own folder in the `datasets` folder. Datasets typically contain `.csv` files as data and `.json` files containing metadata provided by Kaggle.

Files: * `data_fetching/kaggle.py`: Provides a "fetch" method to be used publicly by the main script as well as helper methods for itself. Calling fetch(topics, num_datasets, output_folder) will automatically fetch num_datasets datasets for each topic in the topics array and save each pulled dataset to the output_folder. The main script is configured with output_folder=`datasets`, so each dataset will be saved as a `datasets/{topic}/{dataset_name}` folder.

**2. Metadata Extraction** The metadata extraction component (found in `data_fetching` folder) uses Pandas data analysis to gain additional insights about the data pulled. It reads from the `datasets` folder (populated by the above component) and outputs to the `metadata` folder.

It combines metadata provided by kaggle (title, description, etc.) with this custom-generated metadata (row count, column names, etc.) and outputs a `.json` file for each dataset to the `metadata/{topic}` folder.

Files:

- `data_fetching/extractor_interface.py`: Contains MetadataExtractor class. Provides extensible interface currently only implemented by KaggleExtractor in `data_fetching/kaggle_extractor.py`. In essence, takes a file path containing datasets (pulled from data fetching component) as input and calls the extending class's `extract_from_dataset` and `output_data` methods to determine how to generate metadata and where to store the metadata.
- `data_fetching/kaggle_extractor.py`: Contains KaggleExtractor class that implements MetadataExtractor. Overrides `extract_from_dataset` to specify how to convert kaggle datasets into json. Overrides `output_data` to specify where and how to store the data (in this case, as JSON in the `metadata` folder).

**3. Data Uploading** The data uploading component (found in `data_uploading` folder) takes metadata generated by the Metadata Extraction component and uploads it to a database configured in `database_connection/db.py`. The component is mostly responsible for loading the `.json` files and converting them to objects of the Dataset class (`database_connection/models.py`) to work with the SQLModel ORM. However, it also does some small data transformation, like extracting licenses from a nested structure to a simple list.

Files:

- `data_uploading/uploader_interface.py`: Contains a MetadataUploader interface-like class for uploading data. Overriding classes must specify how to prepare uploading, how to upload, and how to report issues found uploading.
- `data_uploading/json_to_db_uploader.py`: Contains a JsonToDbUploader class that extends MetadataUploader. Specifies how to read in metadata generated in Metadata Extraction and how to save it to the database.

### DB Dump and Load Process

#### Prerequisites

- PostgreSQL and all related CLI are installed (pg_dump, psql) as well as PGAdmin.

#### Dump Process

1. Run `pg_dump -U <username> <database_to_dump> > <dump_filename>.sql`

#### Load Process

1. Create an empty database with name .
2. Run `psql -U <username> <db_name> < <dump_filename>.sql` The database should now be populated with correct tables and records.

#### Testing

Tests exist for each component: kaggle, the extractor, and the uploader.

**Kaggle Tests** Kaggle tests don't interact with the Kaggle CLI but instead tests the helper methods used within `data_fetching/kaggle.py` to ensure they work. Tested methods:

- clean_dataset() - Ensure only `.csv` and `.json` files are retained after calling clean_dataset.
- get_topic_urlList() - Ensure raw output of CLI (generated from sample) is processed correctly, ignoring first two elements and only reading one element (delimited by spaces) from each url.
- ensure_data_exists() - Ensure that this returns true when `.csv` files are present and false otherwise.
- process_dataset() - Ensure that datasets are not re-pulled. Ensure that new datasets being pulled result in success messages. Ensure that exceptions are being thrown when datasets are invalid and not when they are valid.

**Extractor Tests** Extractor tests use fake data from the `tests/test_files` folder to test the behavior of the extractor.

- extract_topics() - Tested on a variety of different file structures representing different pulled datasets. Ensure exceptions are thrown with broken data and data is extracted properly with valid data.

**Uploader Tests** Uploader tests patch the actual uploading process (to the database), intercepting values to compare them against expected results.

- prepare_upload(), upload() - Tests conversion of .jsonto `Dataset class objects, using the same fake data as extractor tests in`tests/test_files".

**Running Tests** Run All Tests and Generate Coverage Report:

```
coverage run --source=data_fetching,data_uploading --omit=data_fetching/run_extractor.py,data_uploading/
-m pytest
```

View Coverage Report:

```
coverage report -m
```

## Model Subsystem

The model subsystem is used by administrators to train models from the pulled datasets for the web application to use. This subsystem uses the Microsoft Recommenders package to train models. This subsystem has the same dependencies as the backend.

The following section will look at each subfolder of the model subsystem indivudally.

### model_system/training

This folder trains the TF-IDF model. model_training.py is the training script, and it is configured by config.py.

Before running the training script, ensure that there is an active PostgreSQL database running that has the following 2 tables:

1. **Dataset**: This table contains the dataset metadata used to train the model.
2. **Rating**: This table contains the user ratings for recommendations between two datasets. This table is optional, and if it does not exist, the model will be trained on just the metadata.

These tables should follow the schema defined in the backend. The location of this PostgreSQL database and its tables are determined by config.py.

config.py also has the following options:

- COLS_TO_CLEAN: The names of the columns that will be combined into a single column and used as text input TF-IDF.
- [*]_COL: The name of the given column. These four columns are required for model training and tuning.
- TOKENIZATION_METHOD: This is the tokenization method that the model will use to generate tokens for the input text. It has the following options (Source):
  1. "**none**": No tokenization is applied. Each word is considered a token.
  2. "**nltk**": Simple stemming is applied using NLTK.
  3. "**bert**": HuggingFace BERT word tokenization is applied.
  4. "**scibert**": SciBERT word tokenization is applied. This is recommended for scientific journal articles.
- [*]_WEIGHT: The tuning weight for the given column. If the weight is set to 1, the column will be ignored.

To run the training script, run the following command:

```
python -m training.model_training
```

**model_system/tests** This folder contains unit tests for the `model_training.py` script. The tests mostly cover the tuning functionlity, because TF-IDF training is already tested by Microsoft.

To run the tests without coverage:

```
pytest tests/model_training_test.py
```

To run the tests with coverage:

```
pytest --cov="training"  tests/model_training_test.py
```

**model_system/scripts** This folder contains a script to dump all the user ratings for ease of use with the `model_training.py` script. To run the script, have the app running locally, then run:

```
bash scripts/rating_dump.sh
```

**model_system/notebooks** This folder contains Jupyter notebooks for exploration and research of the the TF-IDF model. It contains the following notebooks:

- `preliminary_TFIDF`: Initial TF-IDF API testing. Uses mock metadata to train and examine a model.
- `real_data_TFIDF`: TF-IDF training using real metadata sources from Kaggle.
- `v2_TFIDF`: Testing of model serialization using pickle.
- `naive_recommnder`: Testing of a naive model used to compare to the complex model.
- `custom_weights_TFIDF`: Exploration and testing of a custom weighting loop after model training.

**model_system/mock_data** This folder contains mock metadata used to train some of the prelimanary models.