SRDP Database & API

Release 0.1

Evan Jones

Oct 26, 2022

CONTENTS

CHAPTER

ONE

WHAT IS SRDP?

Strategies of Resistance Data Project (SRDP) is a global dataset on organizational behavior in self-determination disputes. It is actor-focused and spans periods of relative peace and violence in self-determination conflicts. By linking tactics to specific actors in broader campaigns for political change, we can better understand how these struggles unfold over time, and the conditions under which organizations use conventional politics, violent tactics, nonviolent tactics, or some combination of these.

The previous SRDP comprises 1,124 organizations participating in movements for greater national self-determination around the world, from 1960 to 2005. An update of the dataset is currently underway and will bring all data up to 2020 as well as expand the type of colletions available.

CHAPTER

TWO

WHY DO WE PROVIDE A DATABASE API?

The goal of storing the project data in a relational database and providing an application programming interface (API) on top of that database is threefold:

- Create a single source of truth for all data - previously data has been stored across various providers (dropbox, googlesheets, local devices) and updated in multiple places, leading to increased prevalance of errors and conflicting versions.
- Make it easier for project maintainers and end users to retrieve the data they need, in the desired format and scope, in a programming language of their choice.
- Provide a service layer for project developers to build additional tools on top of (e.g. web UI for labeling, data ingestion pipelines, user-facing website, etc.)

The first version of this API provides just enough functionality such as basic Create, Retrieve, Update, Delete (CRUD) actions to achieve all three of these, but certainly is not a full-fledged API. For example, most endpoints do not accept query parameters to customize requests.

The hope is that these documents serve as a jumping off point for future developers to further expand the service.

Check out the directory section for information on the project file structure. Information on configuring, launching, administrating, and updating source code is available in the *Developer Manual* section.

CHAPTER

THREE

HOW ARE THE DATABASE AND API IMPLEMENTED?

Tech Stack

The relational database uses an MySQL 5.7 image and is containerized. The API app is also containerized and built using Python Flask for routing plus *Flask-SQLAlchemy* to build the database models. The API endpoints are fully documented in a Swagger UI according to OpenAPI specification with the help of the *flask-apispec*. Docker Compose handles container orchestration and deployment. The database itself is persisted to volume (data persists even after starting and stopping of the container)

Table 1: Tech Stack

Purpose	Tool	Documentation
Containerization	Docker	https://docs.docker.com/
Container Orchestration and Deployment	Docker Compose	https://docs.docker.com/compose/
RDBMS	MySQL 5.7	https://dev.mysql.com/doc/refman/5.7/en/
Web Dev	Flask	https://flask.palletsprojects.com/en/2.1.x/
ORM	Flask-SQLAlchemy, SQLAlchemy	https://flask-sqlalchemy.palletsprojects.com/en/2.x/
SQLAlchemy Database Migrations	flask-migrate	https://flask-migrate.readthedocs.io/en/latest/
API Schema	flask-marshmallow	https://flask-marshmallow.readthedocs.io/en/latest/
API Docs	flask-apispec	https://flask-apispec.readthedocs.io/en/latest/usage.html
API Schema	flask-marshmallow	https://flask-marshmallow.readthedocs.io/en/latest/
Server Hosting	Linode	https://www.linode.com/

To gain a better sense of web development in flask and many of the frameworks/modules used in this project, I highly recommend either read or working through in its entirety the Flask Mega Tutorial by Miguel Grinberg. It is highly informative. Much of this project close resembles the code in that tutorial. In particular, chapters 1-4, 17-19, and 23 are must reads.

What is a REST API?

To steal language from Redhat linux's documentation:

> A REST API (also known as RESTful API) is an application programming interface (API or web API) that conforms to the constraints of REST architectural style and allows for interaction with RESTful web services. REST stands for representational state transfer and was created by computer scientist Roy Fielding.
>
> An API is a set of definitions and protocols for building and integrating application software. It's sometimes referred to as a contract between an information provider and an information user—establishing the content required from the consumer (the call) and the content required by the producer (the response). For

> example, the API design for a weather service could specify that the user supply a zip code and that the producer reply with a 2-part answer, the first being the high temperature, and the second being the low.
>
> In other words, if you want to interact with a computer or system to retrieve information or perform a function, an API helps you communicate what you want to that system so it can understand and fulfill the request.
>
> You can think of an API as a mediator between the users or clients and the resources or web services they want to get. It's also a way for an organization to share resources and information while maintaining security, control, and authentication—determining who gets access to what.

So what the hell does that mean in layman's terms? This blog provides a nice explanation.

In this project, the API is a web service that relies on HTTP protocol to send and receive requests and data. However, an API need to necessarily be a web service.

Note: This project is under active development.

CHAPTER

FOUR

CONTENTS

4.1 Codebase Overview

Here you will find a high-level overview of the codebase, the directory structure, and each of the constituent pieces here. Even if you do not understand everything at first, I encourage you to keep reading. At minimum you will gain a sense of what each part of the codebase is functionally responsible for and how they pieces fit together. This is the essential for proper debugging.

4.1.1 Directory Structure

The project has the directory structure shown below.

```
srdp-database
├── app
│   ├── administrator
│   ├── api
│   ├── auth
│   ├── errors
│   ├── main
│   └── template_filters
│   ├── __init__.py
│   ├── api_spec.py
│   ├── email.py
│   └── models.py
├── db
├── deployment
├── docs
│   └── source
├── examples
│   └── api
├── migrations
│   └── versions
├── tests
└── venv
    ├── bin
    ├── include
```

```
    ├── lib
    └── lib64 -> lib
├── .env.example
├── babel.cfg
├── boot.sh
├── config.py
├── docker-compose.yml
├── Dockerfile
├── launch.sh
├── LICENSE
├── Procfile
├── pyproject.toml
├── README.rst
├── requirements.txt
├── runtime.txt
├── srdp.py
└── Vagrantfile
```

4.1.2 Docker

The codebase relies on docker to containerize the web app and MySQL database. Containerizing each componenet allows for painless deployment across a broad array of servers / cloud providers with minimal server-side configuration (apart from setting up a reverse proxy). Each is treated as an independent microservice and their network topology is defined in *docker-compose.yml*. This should be a first point of reference to gain a sense of how the pieces of the application work together at a high-level.

docker-compose.yml spins up two containers. The first is an image of a MySQL database version 5.7 and uses two volumnes: the config settings in *db/custom.cnf* and */var/lib/mysql* which is where the database is mounted on the local machine so that data can persist beyond sessions (in case the container is stopped, the data will still exist when it is spun back up). The second is a containerized version of the Flask API web app which is launched by calling *Dockerfile* in the parent directory. This custom docker file configures the container environment (OS flavor, pre-installed dependencies, etc.) and finishes by calling *boot.sh* which actually launches the web app. Both containers live inside a private networks entitled 'dbnet'. This means that only they can see and communicate with each other apart from a single port, 5000, which is mapped to port 5000 on the localhost. All requests to the API occur through this gateway.

Docker provides great documentation on Docker Compose for learning how to write (and understand) orchestration files for different services.

4.1.3 Database

There are handful pieces that make the database tick. The first is the database itself which is a virtual instance running in a docker container. As noted above, this container is linked with a volume on the local device so that data persists across launch and tear down (whenever the app is updated). The second piece is the .env file which specifies configuration variables for the database, including the database name, the user, the password, the root password, and the connection URI that points to the containerized database. This information is read into *config.py*, which stores all configruation parameters for the Flask app, and then passed to Flask-SQLAlchemy on app launch to create the actual connection between the app and database. The final piece is Flask-SQLAlchemy which is a wrapper around the SQLAlchemy a python ORM module.

There is *zero* SQL involved in creating and defining the database and tables. All of this is handled via SQLAlchemy in *app/models.py*. Each class in the file corresponds to a table in the database and each class variable corresponds to

a field in that table. Classes may also have methods for performing CRUD actions on the table, but these are pure python functions and are completely independent of the database itself. All new tables or modifications to existing tables happen in *app/models.py*.

4.1.4 Flask Logic

The remainder of the codebase consists flask app logic. The flask application follows an "application factory" approach. An app factory approach relies on modular design, where multiple versions of the app can be instantiated with varying configuration parameters. In addition, the application is broken down into modular parts called "blueprints" which groups together similar logic.

The application factory design can be seen in *app/__init__.py*. Each of the flask extensions are first created and then initialized with the app in the *create_app()* function. Thereafter, each blueprint is registered with the app, creating a unique namespace for all the routes within that blueprint. Finally, logging is set up to write to a local log files or a mail server if one is set up (see *here <https://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-x-email-support/page/4> `_*). The appliation can easily work with a gmail account with SMTP and third-party app access enabled. This email address is set in *.env* and the functions for sending emails are located in *app/email.py*.

Finally, we turn to the API routes and specs, the meat and potatoes of the app. The specification for the API is located in *app/api_specs.py*. This file calls the apispec library with marshmallow and flask plugins to generate the website's Swagger docs page for the api routes and database table schemas.

The *api/* directory contains a file for each of the database tables as well as a few utility files for things such authentication/authorization and error handling. Each of these files contains routes for CRUD (create, retrieve, update, delete) actions via their respective HTTP methods (POST, GET, PUT, DELETE). Whenever a new table is added to the database, a new file should be added with the appropriate routes/methods. A new schemas will also need to be defined in *app/apispec.py*

You will notice there are other blueprint directories such as auth, administrator and main. These are currently unused but contain some initial logic if a future developer wants to add a full-fledged front-end UI to the website. Administrator stores logic for an flask-admin administrator portal. Auth contains routes for user authentication, and main serves as a catch all for all other front-end routes.

4.2 Developer Manual

Here you will find a "how to" developer's manual for doing a number of key tasks ranging from installing the application on your local machine as well as on a production server all the way to modifying and update the database and API as the project needs change and grow.

4.2.1 Installation

Installation steps vary depending on whether you're installing the app on a local machine for development and testing or trying to put it into production. However, regardless of which of these two situations you're in, you'll need to install docker and docker compose on the machine. Installation instructions for each of these can be found below:

- Get Docker
- Install Docker Compose

Local Machine

Production Server

4.2.2 Configuration

4.2.3 Launching

4.2.4 Admin

4.2.5 Working with Database Models

4.2.6 Working with API routes

4.2.7 Updating API Documentation

4.3 API Documentation

The end user API route documentation can be found at *https://srdp.ea-jones.com*.