SPARQL2GraphQL

Bringing RDF datasets to all developers

Daniel Crha

Charles University

Table of contents

1. SPARQL2GrahpQL

Many government and non-government organizations today publish various datasets using Linked Data through SPARQL endpoints. These datasets are often also available as JSON or RDF dumps, but having to write code to explore them can be cumbersome. SPARQL2GraphQL aims to make it easier to use these datasets by providing a tool which will give developers a more friendly interface which they likely already know how to use - GraphQL.

Simply configure SPARQL2GraphQL with the URL of a SPARQL endpoint which has some data which interests you, and let it do its magic. Soon, you will have a GraphQL instance which you can use to painlessly explore and query the data.

If you want to get started right away, check out the usage guide.

If you want to find out how it works, find that out here.

If you wish to tinker with the project's code, or even contribute to it, here is a development handbook.

2. User Guide

2.1 Usage guide

This page will explain how to set up SPARQL2GraphQL for a given SPARQL endpoint, step-by-step.

2.1.1 Installing dependencies

The project is written in TypeScript and uses Node.js as its runtime.

You will need to have the following installed before you proceed with usage:

- Node 16.13.0 (Gallium LTS) - easily managed with nvm
- npm

Once you have installed them, run `npm install` in the project root directory to install all required dependencies with npm.

2.1.2 Edit configuration

There is one required configuration step before you run the project - configuring the SPARQL endpoint you want to run. In `src/api/config.ts`, set the value `ENDPOINT_TO_RUN` to refer to your endpoint. There is a pre-defined list of endpoints in `src/observation/endpoints.ts` in case you just want to try the project without having a specific SPARQL endpoint in mind, but you can easily define your own like so:

```
export const ENDPOINT_TO_RUN = { url: 'https://data.europa.eu/sparql', name: 'European Data', };
```

The `name` can be whatever you want, it's just an easily readable identifier used in logs.

There are other configuration values which you are free to modify, but they have sensible defaults in case you just want to get started. If you want to find out more about additional configuration options, variables are documented in the configuration files itself. More advanced configuration options like postprocessing hooks are also available, you can read more about them here.

2.1.3 Run it

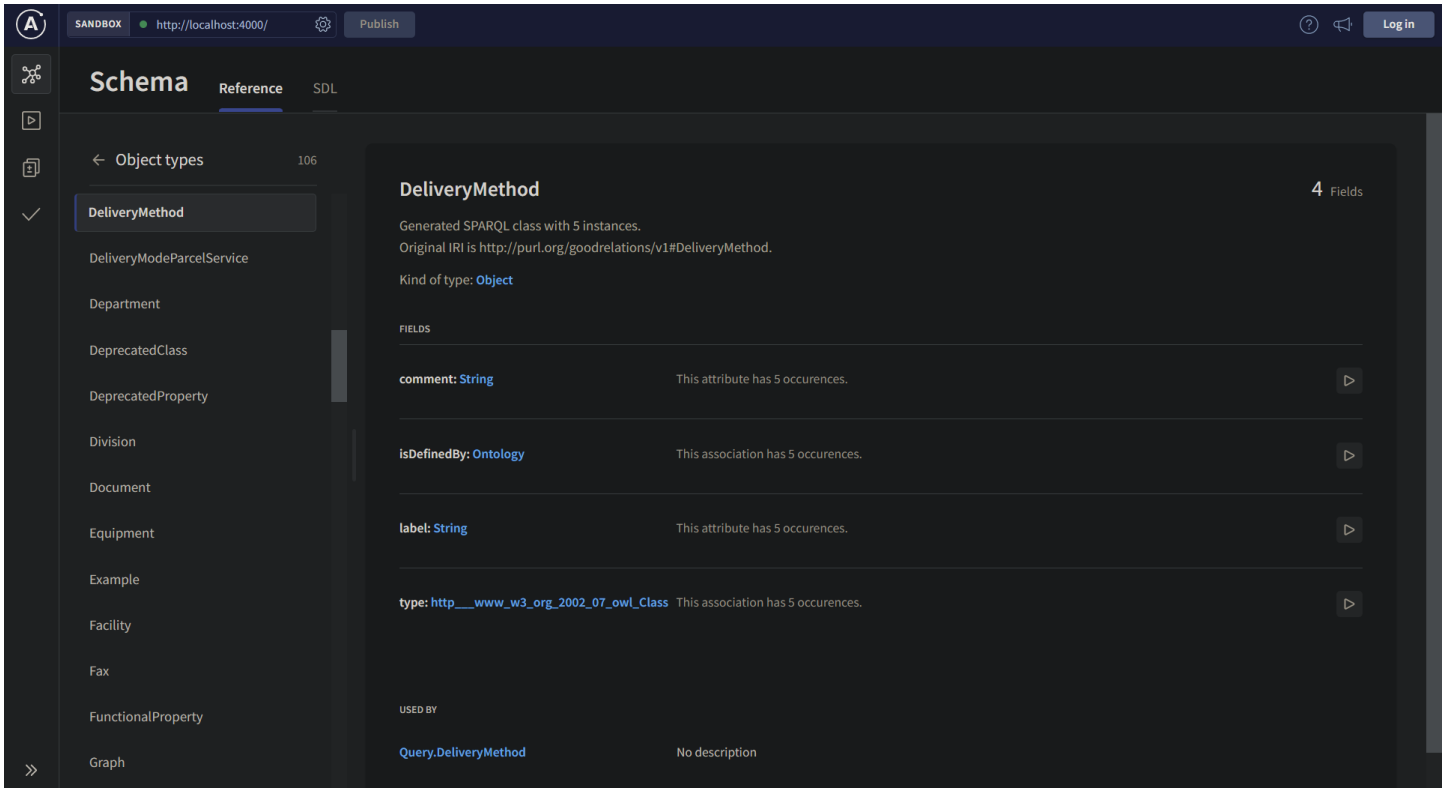
After everything is configured, run `npm start` to start SPARQL2GraphQL.

In the logs, you will see that it will first run some observations on the target endpoint to collect information about its schema and the data contained within. This may take a while, especially for large datasets, depending on the configuration parameters used.

After observation is finished, you will see that a fully functional GraphQL interface is available for you to explore at the configured port (localhost:4000 by default).

2.1.4 Explore the schema

When you open the GraphQL interface in your browser, you will see an interface provided by Apollo Server. It allows you to explore the schema including all of the available classes, as well as their properties, relations, how many times they occur in the dataset and other metadata.



The interface also allows you to interactively build and execute queries against the endpoint.

2.1.5 Query the endpoint

TODO: WRITE THIS SECTION AFTER IMPLEMENTING QUERYING

3. How it works

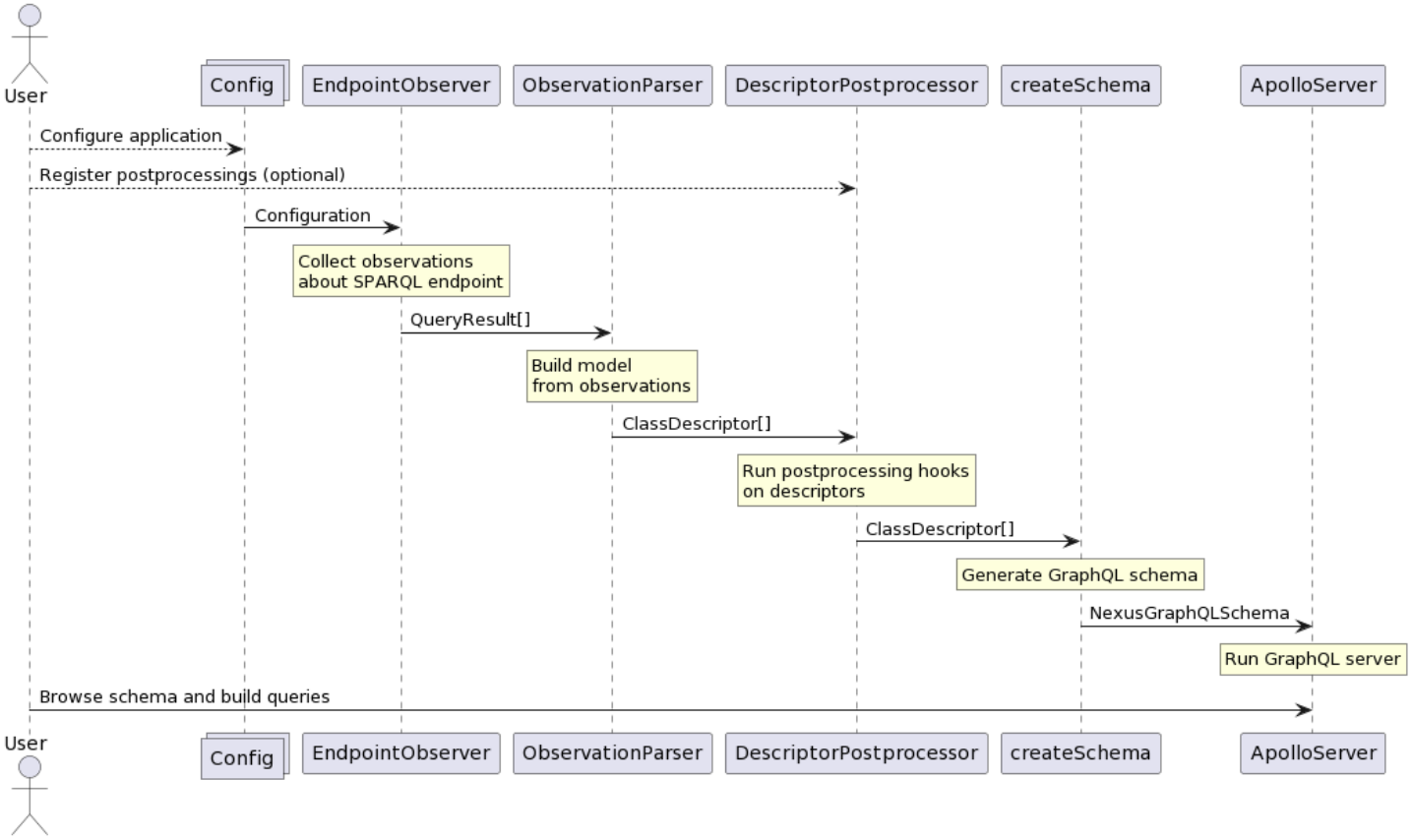
3.1 Overview of how it all works

This page will explain the process used by SPARQL2GraphQL when creating a GraphQL endpoint from the SPARQL endpoint.

3.1.1 General flow

First of all, the user should configure the application. The most basic configuration step is to configure the SPARQL endpoint to observe. However, besides other configuration values, the user may also write custom postprocessing hooks for parts of the extracted data model. You can read more about those here.

When the user configures and runs the application, the setup flow looks like this:



As the diagram shows, at the end of the setup, there is a fully functional GraphQL endpoint active, in which the user can explore and query the endpoint.

TODO: ADD QUERY RESOLVERS INFORMATION HERE WHEN THAT FEATURE IS FINISHED (INCLUDING A GRAPH SHOWING HOW THE GraphQL QUERY IS EXECUTED)

3.1.2 Components

The following is a list of components you can see in the diagrams above, with more detailed descriptions of what they do and how they do it:

- Observation
- Parsing
- Postprocessing
- Schema Creation
- Querying

3.2 Observation

When the application is configured, the user runs the app. Then the application starts observing the SPARQL endpoint, and collecting observations in the form of RDF quads (stored in the `QueryResult` type). These quads are generally constructed to have a blank node subject representing a single observation, and its properties then describe that observation. The following is an example of quads describing the existence of a class in the dataset (`_:observation` denotes a blank node):

```
_:observation se:class foaf:Agent; se:numberOfInstances 42.
```

TODO: make this more ontology-oriented

This particular observation says that the dataset contains a class called `foaf:Agent` which has 42 instances.

3.2.1 Collected observations

The queries used to collect observations are stored in the `QueryBuilder` class. Currently, there are 5 of them:

- `CLASSES_AND_INSTANCE_NUMBERS`: Find classes and how many of them there are.
- `CLASS_INSTANCES`: Find instances of a class for further examination.
- `INSTANCE_ATTRIBUTES`: Get attributes of an instance (attributes are properties which have simple types as their domain, like `boolean` or `number`).
- `INSTANCE_ASSOCIATIONS`: Get associations of an instance (associations are properties which relate to another class).
- `NUMBER_OF_INSTANCE_PROPERTIES`: Collect information about how many instances there are of various class properties.

This list is bound to change in the future, but the goal is to not modify the existing queries if possible, but rather add new ones.

3.3 Parsing

Observations made about a SPARQL endpoint are then parsed by the `ObservationParser`. Its method `buildEndpointModel` receives a list of observations as a parameter (in the form of `QueryResult[]`), and it parses those observations into an object model describing the data.

3.3.1 Descriptors

The model is expressed in the form of **descriptors**, which are objects storing metadata about various entities present in the dataset. The descriptors are located in the `src/models/` folder.

These are the descriptors currently used:

- `EntityDescriptor`: Describes any entity with an IRI, which is basically everything in the RDF world. Other descriptors inherit from this descriptor.
- `NamedEntityDescriptor`: Describes any entity which should also have a human-readable and short name. It can be a bit cumbersome for humans to constantly use IRIs to refer to things, so SPARQL2GraphQL makes this easier by providing simplified names to display in the user interface.
- `ClassDescriptor`: Describes a single class present in the dataset.
- `InstanceDescriptor`: Describes a single instance of a class.
- `PropertyDescriptor`: Describes a property of a class (association or attribute),
- `AssociationDescriptor`: Describes an association of a class, i.e. a property whose range is another class.
- `AttributeDescriptor`: Describes an attribute of a class, i.e. a property whose range is a primitive type.

3.3.2 Parsing algorithm

For each type of observation, there is a method in the `ObservationParser` which parses this observation into a descriptor object.

Therefore the parsing algorithm essentially consists of iterating over the collected observations, choosing the correct parsing function for each observation, and then executing the function.

3.4 Postprocessing

The postprocessing phase occurs after the model (i.e. a set of descriptors) has been built based on observations of the SPARQL endpoint, but before the GraphQL schema is built.

It allows easy additions of various postprocessing activities on the model, like modifying entities in ways which are not connected to model parsing.

An example of this is naming - simplified names are used for all named entities to promote usability of the generated GraphQL endpoint. Since the names are calculated from IRIs, this logic is decoupled from parsing, and it can be easily modified or extended. You can even easily add your own implementation as a new postprocessing hook, and switch it with the default implementation as needed.

3.4.1 Hooks

Postprocessing hooks are functions which are assignable to the `PostprocessingHook` interface:

```
export type PostprocessingHook<TDescriptor extends EntityDescriptor> = ( descriptors: TDescriptor[], ) => void;
```

In other words, it's a function which takes a single parameter - the descriptors of a particular type, and it returns nothing. For example, `PostprocessingHook<ClassDescriptor>` would be run with the list of all `ClassDescriptor`s in the model. You can specify more generic hook constraints, for example using `NamedEntityDescriptor` would run your hook against all descriptors which have a display name property.

The contents of a hook will usually do one or more of the following things:

- Modify the given descriptors in some way
- Run some side effects which depend on the descriptor data
- Log something about the descriptors

You can add your hooks in the `src/postprocessing/hooks` directory, preferably putting one hook per each code file.

Including your hook in the code is then as easy as editing `src/postprocessing/registered_hooks.ts`, and registering your hook in the `getRegisteredPostprocessingHooks()` function. And that's it! Your hook will now automatically be run the next time you run the app.

3.5 Schema Creation

Schema creation occurs after postprocessing, at which point the data model has been finalized in the form of descriptors.

This creation is the purpose of the `createSchema` function in `src/api/schema.ts`. It uses the descriptors to produce a complete GraphQL schema in the form of a `NexusGraphQLSchema` (from the GraphQL Nexus library).

Normally, a schema-first approach is used for modeling GraphQL APIs, meaning developers would first write the GraphQL schema using a schema declaration language, and they would then build their data model to fit that schema. However, automatically generating a schema in the GraphQL schema language is quite cumbersome and involves a lot of string manipulation.

Therefore SPARQL2GraphQL uses GraphQL Nexus as a way to define the GraphQL schema. GraphQL Nexus is a library which facilitates code-first, declarative schema declarations. This means that defining the GraphQL schema is a matter of converting each `ClassDescriptor` into a type definition in the GraphQL schema, and using its related descriptors to definte the properties of that type.

Information like the number of instances is stored in the type and property descriptions to aid developers in exploring the dataset.

3.6 Querying

TODO: WRITE CONTENT FOR THIS AFTER QUERYING IS FINISHED

4. Developer Guide

4.1 Development

Build, lint and test are automatically run in GitHub Actions. They are required to pass for every PR and commit to `master`.

Contributions should be in the form of PRs opened against `master`, containing a comprehensive description of what was changed and why. Bonus points if you include a video or screenshot showcasing the functionality. Tests should also be included with contributions.

Development should follow GitHub Flow and commit messages should follow the Conventional Commits spec.

4.1.1 Useful commands

The following commands will likely be useful to you during development.

Install dependencies

`npm ci` or `npm install` - both will install the dependencies, but `npm ci` will make sure that they match `package-lock.json` exactly, and may take longer to run.

Compile everything

`npm run build` will compile the TypeScript files.

Linting

`npm run lint` will list all linting issues without fixing them automatically.

Formatting (lint w/ autofix)

`npm run format` will run linting and auto-fix all possible issues.

Running

`npm start` will start the app.

Alternatively, if you use VSCode for development, a `Debug API` debug config is included in the repository. Running this debugging config will automatically compile the required files and run the app in debug mode.

Testing

`npm test` will run tests with Jest and generate a testing + coverage report.

4.2 Editing documentation

This documentation is generated using mkdocs from the Markdown files in the `docs` directory, and manually deployed to GitHub Pages with `mkdocs gh-deploy`. The URL for the deployed documentation is https://yawnston.github.io/sparql2graphql/.

If you want to modify the documentation, simply modify the Markdown source of whichever documentation you wish. If you create a new documentation page, you will need to add it to the `nav` in `mkdocs.yml`:

```
nav: - Home: 'index.md' - 'User Guide': - Usage: 'usage.md' - 'How it works': - Overview: 'overview.md' - Observation: 'observat
```

4.2.1 Installing requirements

In order to build, serve or deploy the documentation, you will need to have Python 3 installed, since it is required to run mkdocs.

To install mkdocs and other required plugins, run `pip install -r requirements.txt`.

4.2.2 Viewing documentation locally

Run `mkdocs serve` to build the documentation and serve it on localhost. In the terminal, you will see a link to open this documentation in your web browser.

4.2.3 Generating PDF documentation

If for any reason you want to have a PDF version of the documentation, run `mkdocs build`. This will generate a PDF containing the documentation in `site/pdf/documentation.pdf`.

Note that you may see an error about a missing theme during build - this is unavoidable when using the same `mkdocs.yaml` for both the website and the PDF documentation, and you may safely ignore this error.

4.2.4 Deploying documentation

If you want to deploy the documentation to GitHub Pages, run `mkdocs gh-deploy`, and the updated documentation should be available at https://yawnston.github.io/sparql2graphql/ within a few minutes. Note that you may need to refresh the page without caching in order to see the changes (`Ctrl` + `F5` or `Shift` + `F5` in Google Chrome).