

SPARQL2GraphQL

Bringing RDF datasets to all developers

Daniel Crha

Charles University

Table of contents

1. SPARQL2GrahpQL	3
2. User Guide	4
2.1 Usage guide	4
3. How it works	8
3.1 Overview of how it all works	8
3.2 Observation	10
3.3 Parsing	12
3.4 Postprocessing	13
3.5 Schema Creation	15
3.6 Querying	16
3.7 Hot reloading	17
4. Developer Guide	18
4.1 Development	18
4.2 Editing documentation	20

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 Library interface

The main functionality is encapsulated by the `SPARQL2GraphQL` class in `src/api/index.ts`. This class contains functions for the main tasks users may want to perform.

Most notably, the `buildSchemaAndRunEndpoint` function is provided for users who want the least configuration necessary. It is enough to simply set up a `Config` as shown in the next section, and run SPARQL2GraphQL like so:

```
const config = { ... }; const sparql2graphql = new SPARQL2GraphQL(); // This method returns a Promise, so you should await it //
```

The `SPARQL2GraphQL` class also contains other functions which perform smaller parts of the whole algorithm, which you may want to use if your task is more complex than *convert this SPARQL endpoint into a GraphQL endpoint*. For example, the `observeAndBuildSchema` function performs all the steps up to and including schema building, meaning one could use it to simply save the GraphQL schema into a file and do something else with it, rather than starting a GraphQL endpoint.

2.1.3 Edit configuration

There is one required configuration step before you run the library - 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 known online and working 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.

A very basic configuration can look like this:

```
import { Config, SIMPLE_LOGGER } from './api/config'; const config: Config = { endpoint: { url: 'https://dev.nkod.opendata.cz/sp' }
```

The endpoint `name` can be whatever you want, it's just an easily readable identifier used in logs. The `logger` property is not mandatory, but it is very helpful to specify a logger in order to see what's going on. Any `winston` logger will do, but the `SIMPLE_LOGGER` defined in `src/api/config.ts` defines a sane default logger which logs the most important messages to the console.

There are other configuration values in `Config` 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, they are documented in the configuration file itself.

2.1.4 Run it

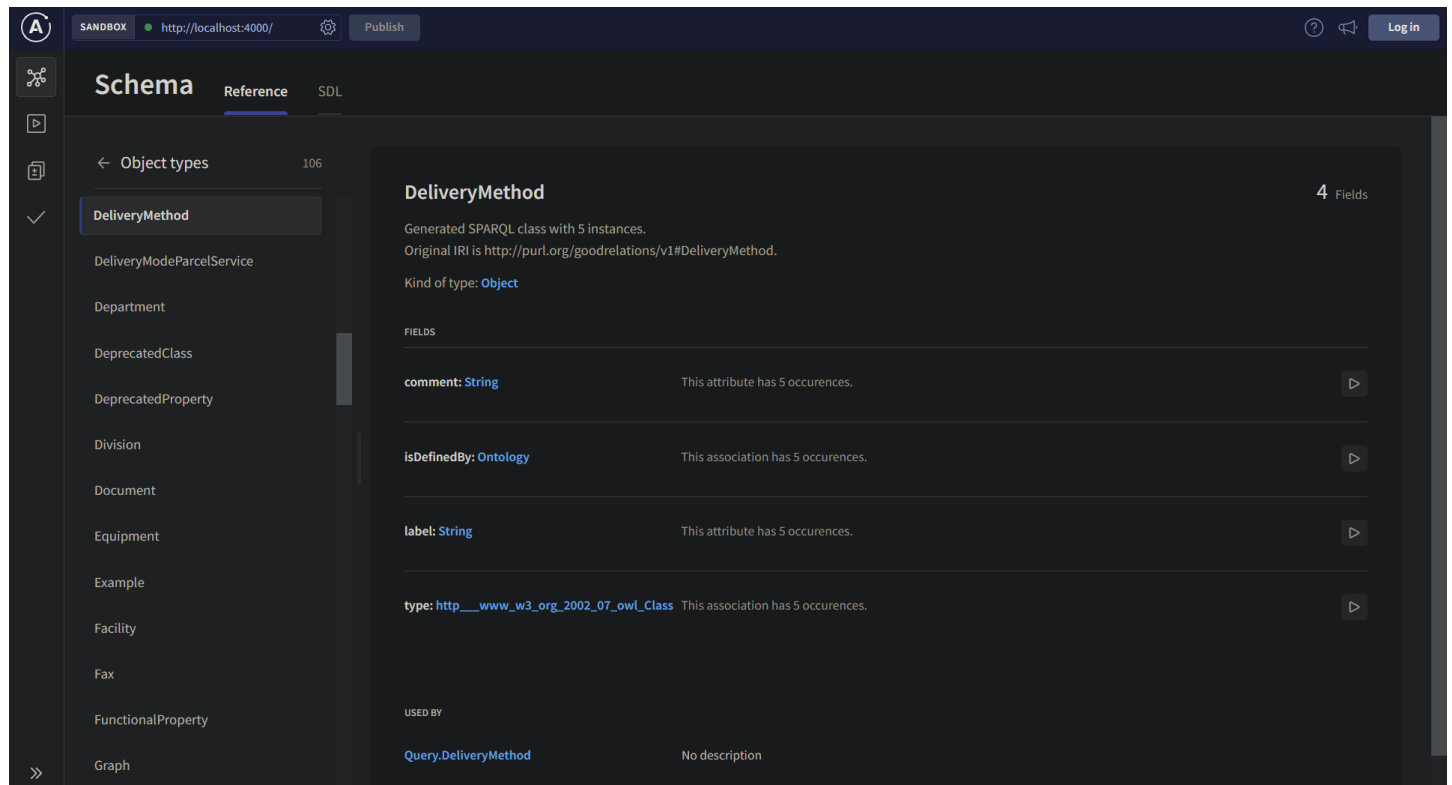
Putting together the above code examples is enough to start a SPARQL2GraphQL instance, which will observe the configured SPARQL endpoint and convert it to a GraphQL one. Alternatively, if you are running SPARQL2GraphQL from this repository, you can run `npm start`, which will run `src/main.ts`, which is an entrypoint meant for starting the server from within this repository.

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. As an example, the `NKOD` endpoint shown in the configuration above takes about 2 minutes to have a fully functional GraphQL instance running.

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

2.1.5 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.

2.1.6 Hot reloading

You may notice that with the default configuration, another round of observation will start right after the GraphQL server is created.

This is **hot reloading** in action. Because fully observing the entire dataset in the endpoint may take a long time, the initial observations are limited by the default configuration - observation will only count up to 1000 occurrences for each property in the dataset.

After the server has started, it's fully available and functional for purposes of schema exploration and querying. In the background, SPARQL2GraphQL continues to make more detailed observations, and seamlessly updating the GraphQL schema in the GraphQL endpoint.

You can read more about hot reloading [here](#).

2.1.7 Query the endpoint

If you are not yet familiar with how to compose GraphQL queries, you can learn more about GraphQL [here](#).

You can use the interface to query the data, either writing the query yourself, or using the graphical query editor to compose the query.

The screenshot displays the SPARQL2GraphQL web interface. On the left, the 'Documentation' panel shows the schema for the 'Dataset' type, including fields like 'limit', 'offset', 'title', and 'theme'. The 'Query' editor in the center contains a GraphQL query that uses the 'limit' and 'offset' variables. The 'Response' panel on the right shows the JSON output of the query, which includes three dataset entries with titles like 'Přehled o změnách vlastního kapitálu - období 12/2019' and 'Přehled převedených výnosů z daní do místních rozpočtů'.

NOTE: be careful running queries which have an unlimited result size! If your SPARQL endpoint contains a very large dataset, the query could either run for a very long time, or it might just fail due to a timeout. You should use the `limit` and `offset` parameters provided in each query field as shown in the image above.

3. How it works

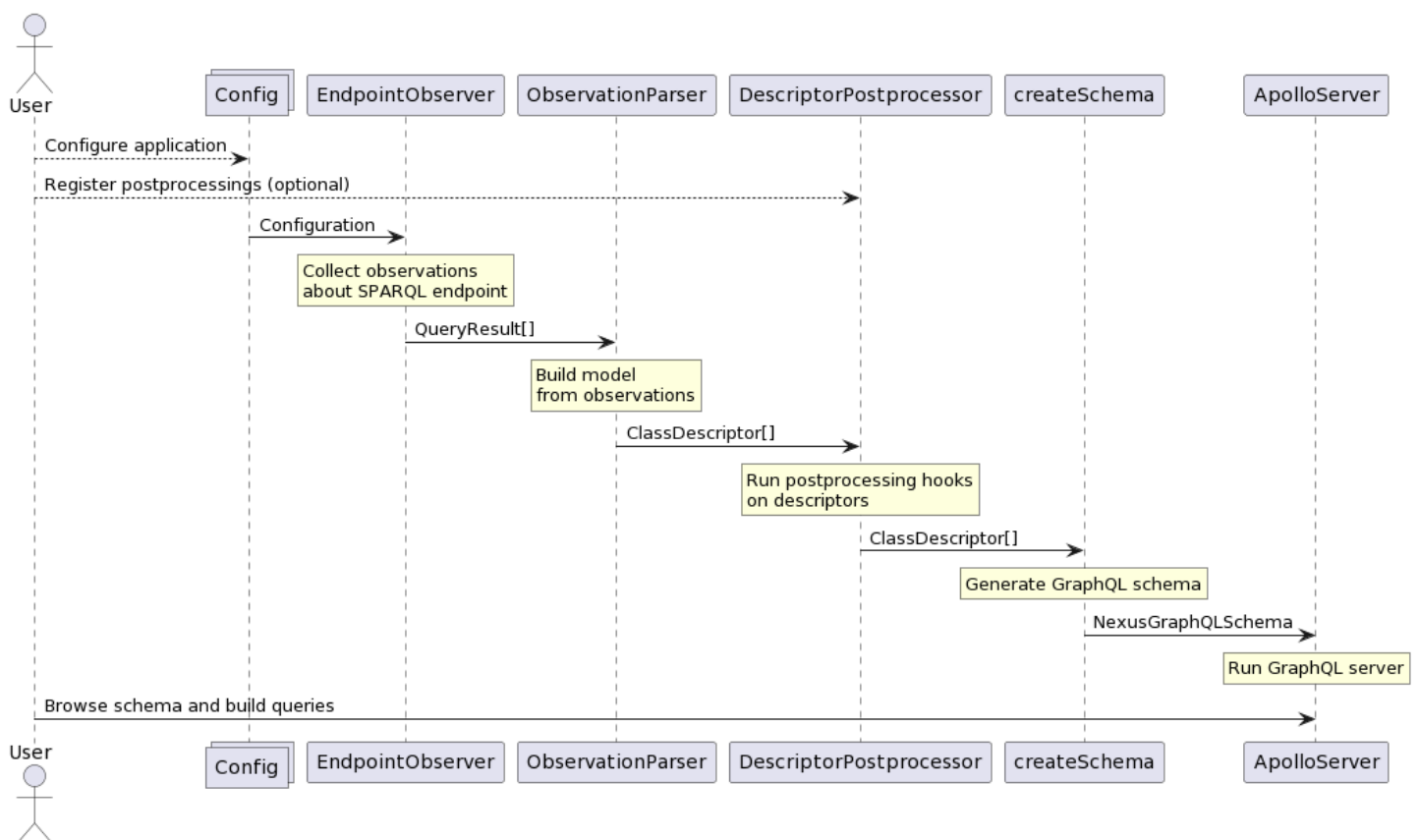
3.1 Overview of how it all works

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](#)
- [Hot Reloading](#)

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.

3.2.1 Ontology

All observations conform to an ontology described in the `ontology.ttl` file, which contains a [RDFS](#) definition of the observations in the [Turtle](#) format.

3.2.2 Observer system

Observation is essentially just a process where we need to execute many SPARQL queries, where some queries have dependencies on outputs of previous queries. As an example, we cannot observe the existence of properties on some class, before we even know about the class' existence.

Therefore the observation logic is split up into observer classes, namely `EndpointObserver` and `InitEndpointObserver`. These observers each collect a specific set of observations. `EndpointObserver`s may declare dependencies on observations produced by other observers. `InitEndpointObserver`s do not required any previous observations to do their observations, and they produce the initial observations which are in turn used by other observers.

The flow of data between observers and their invokation is handled by the `ObserverManager` class. It collects the outputs of its registered observers, and triggers observers which are subscribed to new observations accordingly.

Its usage looks as follows:

```
const observerManager = new ObserverManager(config); observerManager.subscribeInit(new ClassObserver()); observerManager.subscri
```

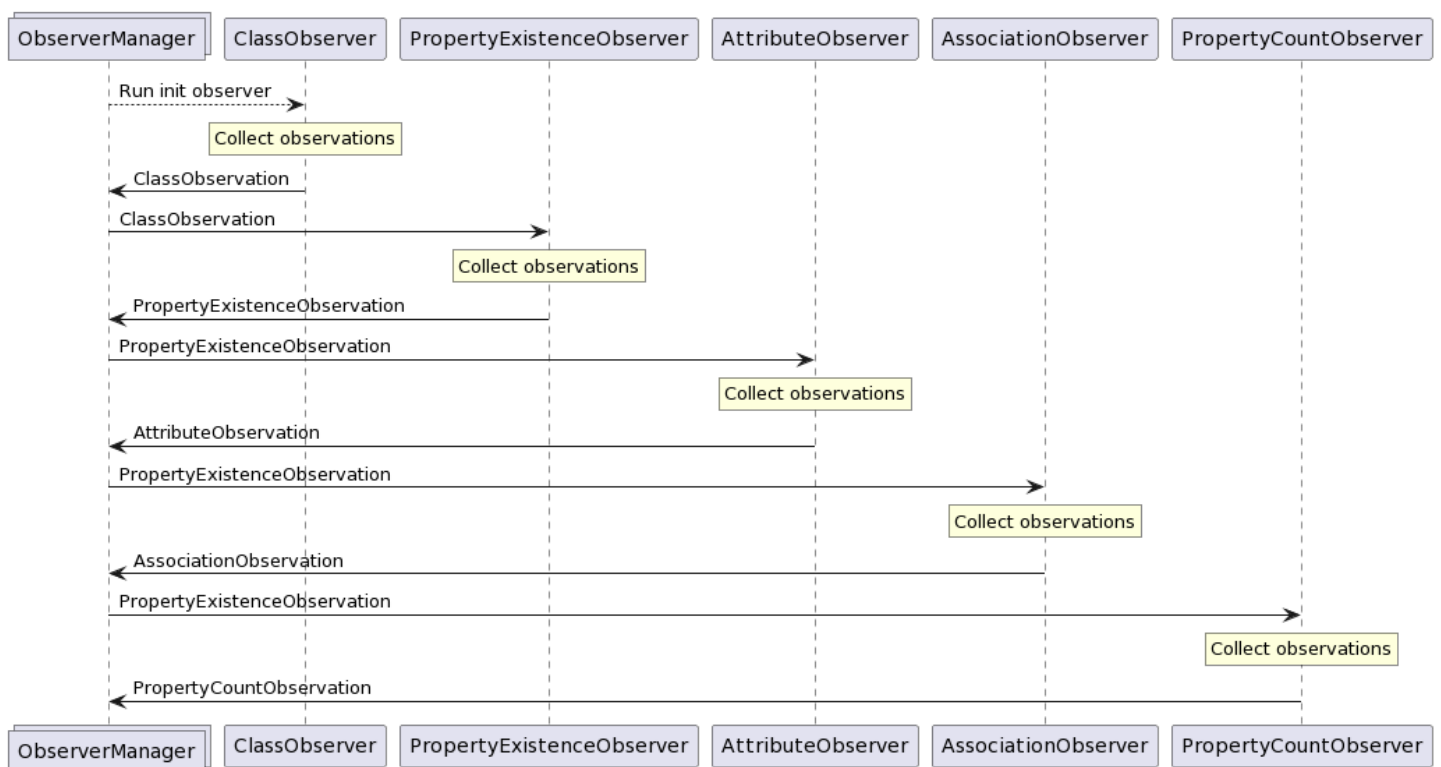
3.2.3 Implemented observers

While it is possible for library users to implement and use their own observers, some observers are pre-implemented to facilitate the main flow of SPARQL2GraphQL.

They each collect a single kind of observation from the ontology:

- `ClassObserver`: collects `ClassObservation`s
- `PropertyObserver`: collects `PropertyExistenceObservation`s
- `AttributeObserver`: collects `AttributeObservation`s
- `AssociationObserver`: collects `AssociationObservation`s
- `PropertyCountObserver`: collects `PropertyCountObservation`s
- `InstanceObserver`: collects `InstanceObservation`s (*currently unused*)

The data flow between these observers looks like this:



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 define 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

3.7 Hot reloading

Hot reloading of the GraphQL schema means updating the running GraphQL server's schema while it is already running. The reason for why you may want to do this is that observing very large datasets can take a very long time, and users may not want to wait for 2 hours before they can even look at the schema or query the endpoint.

Therefore, only the required observations for actually building a schema are performed at startup when using hot reloading. More in-depth observations (i.e. accurate property counts) are then performed in the background, once the server is already running, and the user is able to query it.

The schema is then seamlessly updated, and if the user is using the default playground which is started with SPARQL2GraphQL, they don't even need to refresh their browser window to see the new schema - it refreshes itself automatically.

3.7.1 How it works

After the server starts, the `hotReloadServerSchema` function of the `SPARQL2GraphQL` class can be used to run hot reloading. Firstly, hot reloading must be configured in the `Config`. It is enabled by default, and the default configuration looks like this:

```
hotReload: { configIterator: (config, _oldModel) => { // Examine 10 times more properties than in the last iteration. const newC
```

The hot reload algorithm can be described as the following:

1. Get a new `ObservationConfig` for the next round of observations. This config is obtained using the `configIterator` function from the hot reload configuration.
2. Conduct a new round of observations, which are more detailed and likely take more time than the previous round. The default behavior is to count up to 10 times as many property instances than the previous round of observation.
3. Hot-reload the running GraphQL endpoint's schema
4. Poll the `shouldIterate` function from the hot reload configuration about whether or not the hot reloading should keep iterating. If it returns `true`, the algorithm loops. If `false`, hot reloading stops for good.

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 `src/main.ts` 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.yml` 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).