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 and query 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.

SPARQL2GraphQL aims to be both simple to use and extensible. A minimal usage looks like this:

```
const config: Config = { endpoint: { url: 'https://dev.nkod.opendata.cz/sparql', name: 'NKOD', }, logger: SIMPLE_LOGGER, }; cons
```

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` functions 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. This is done by creating a `Config` object, which you will pass to SPARQL2GraphQL functions. 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://data.gov.cz/sparql', na
```

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, you can refer to the configuration guide. Alternatively, you can examine their definition in the code, where they are also documented.

2.1.4 Run it

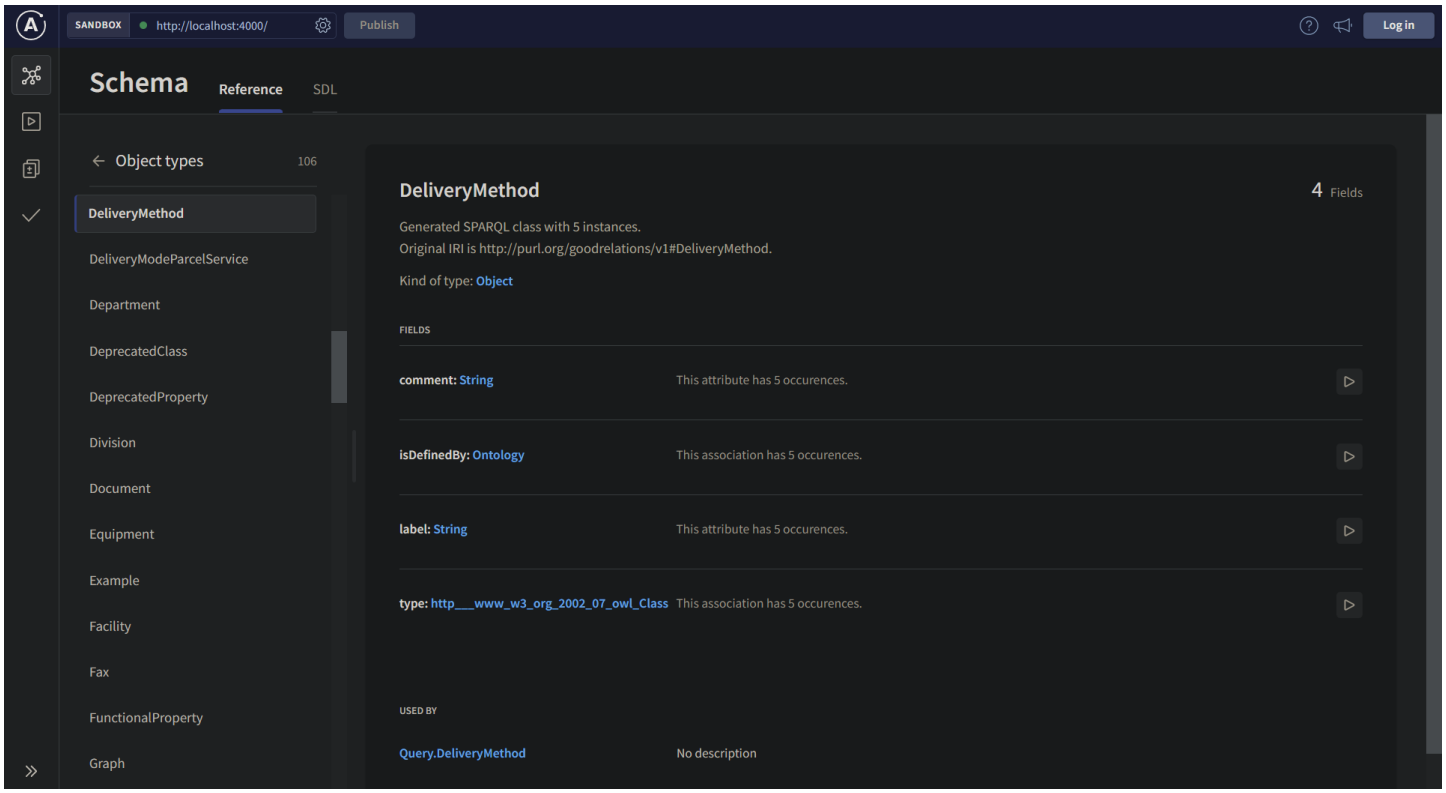
Putting together the above code examples is enough to start a SPARQL2GraphQL instance, which will observe the configured SPARQL endpoint and convert it to a GraphQL one. Alternatively, if you are running SPARQL2GraphQL from this repository, you can run `npm start`, which will run `src/main.ts`, which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

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 CZ Government open data endpoint shown in the configuraiton above takes under 10 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 occurences for each property in the dataset.

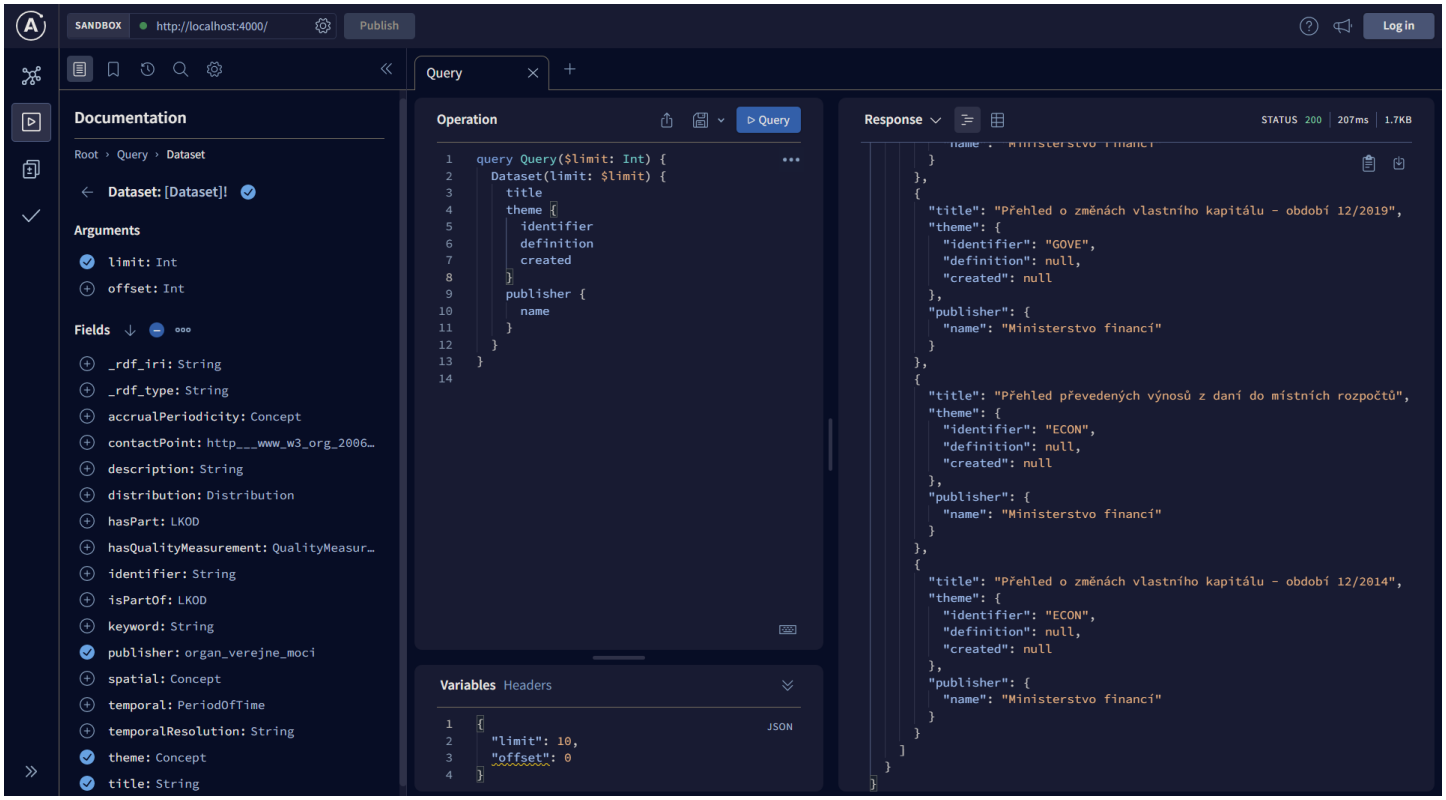
After the server has started, it's fully available and functional for purposes of schema exploration and querying. In the background, SPARQL2GraphQL continues to make more detailed observations, and seamlessly updating the GraphQL schema in the GraphQL endpoint.

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.



NOTE: be careful running queries which have an unlimited result size! If your SPARQL endpoint contains a 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.

2.2 Configuration

This page reviews and explains all available configuration options in SPARQL2GraphQL. Before you explore the individual configuration options, it is recommended that you familiarize yourself with the basic concepts in SPARQL2GraphQL described here.

2.2.1 General configuration

The root configuration type is `Config`, which exposes the following properties:

```
interface Config { endpoint: SPARQLEndpointDefinition; logger?: winston.Logger; observation?: ObservationConfig; postprocessing?
```

Only the `endpoint` property is mandatory, since it contains the SPARQL endpoint which you want SPARQL2GraphQL to run against. An endpoint looks like this:

```
{ url: 'https://dev.nkod.opendata.cz/sparql', name: 'NKOD', }
```

It is also very highly recommended to configure a `logger`, since it is very helpful to know what exactly is happening, since the bootstrapping process can take a very long time. The logging framework of choice is winston, so you are free to pass in any winston logger. However, if you want a sensible default, you can use the `DEFAULT_LOGGER` exposed by SPARQL2GraphQL.

2.2.2 Specialized configuration

The remaining configuration values are more specialized, and they affect individual components of SPARQL2GraphQL.

All of these options have sane default values, so it is recommended to first try not defining them (and thereby using the defaults). If you find that you want to adjust the behavior of SPARQL2GraphQL afterwards, then you can look into modifying these values.

Observation

The `observation` property of `Config` modified the endpoint observation phase, and it can contain the following values:

```
interface ObservationConfig { maxPropertyCount: number | undefined; propertySampleSize: number | undefined; ontologyPrefixIri: s
```

`maxPropertyCount` sets a maximum number of properties to be examined when performing observations which count or enumerate many properties. Unless you *need* to have the most accurate schema in the GraphQL endpoint from the very start, it is recommended to set this value. It will **significantly** speed up observations on large datasets. A good default value is `1000`, and for best results, combine this value with hot reloading, where each iteration of hot reloading increases it by an order of magnitude.

When analyzing the range for each attribute and association, a sample of up to `propertySampleSize` occurences is selected, and their types are used to determine that property's type. Setting `propertySampleSize` is highly recommended, with a reasonable default being `100` or `1000`. While this setting may in some rare cases lead to the generated schema missing some return types for some properties, leaving it unlimited may result in errors during observation for large datasets, where the process is unable to allocate enough memory to hold all of the observations.

`ontologyPrefixIri` sets the IRI for the ontology created during observation. You should not need to modify this value from the default `http://skodapetr.eu/ontology/sparql-endpoint/`, unless you wish to save the observations themselves and use them for other purposes.

By default, all properties in the created GraphQL schema are arrays, since in RDF, any property can be specified multiple times. `shouldDetectNonArrayProperties` allows observations which flag properties as scalars, making sure that properties are not flagged as arrays unless they can really contain multiple values.

The created GraphQL schema contains comments which have additional information about the data, namely for each property, it specifies how many times it occurs in the dataset. This can be helpful when first exploring the schema, and deciding whether a property is important or not. Setting `shouldCountProperties` to `true` will enable the counting of properties, otherwise their counts will be set to `0`.

It is recommended to initially set both `shouldCountProperties` and `shouldDetectNonArrayProperties` to `false` to ensure fast startup time, but to set them to `true` in the hot reloading config. That way, the necessary observations will be carried out in the background while you can already explore and query the dataset.

Postprocessing

Read more about postprocessing here.

Schema

The schema configuration only contains one option:

```
interface SchemaConfig { graphqlSchemaOutputPath: string | undefined; }
```

If you set `graphqlSchemaOutputPath` to a valid file path, the generated GraphQL schema will be saved to this path when it is generated. This can be useful if you want to use some visualization tools to better aid you in exploring the created GraphQL endpoint.

Server

The server configuration contains one option:

```
interface ServerConfig { port: number; }
```

The `port` option will configure the port where the GraphQL endpoint will be available.

Model Checkpointing

Read more about model checkpointing here.

Hot reload

Read more about hot reloading here.

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 Bootstrapping flow

First of all, the user should configure the application. The most basic configuration step is to configure the SPARQL endpoint to observe. You can read more about configuration here.

The bootstrapping phase of SPARQL2GraphQL is divided into four distinct phases:

1. **Observation**
2. **Model Parsing**
3. **Postprocessing**
4. **Schema Creation**

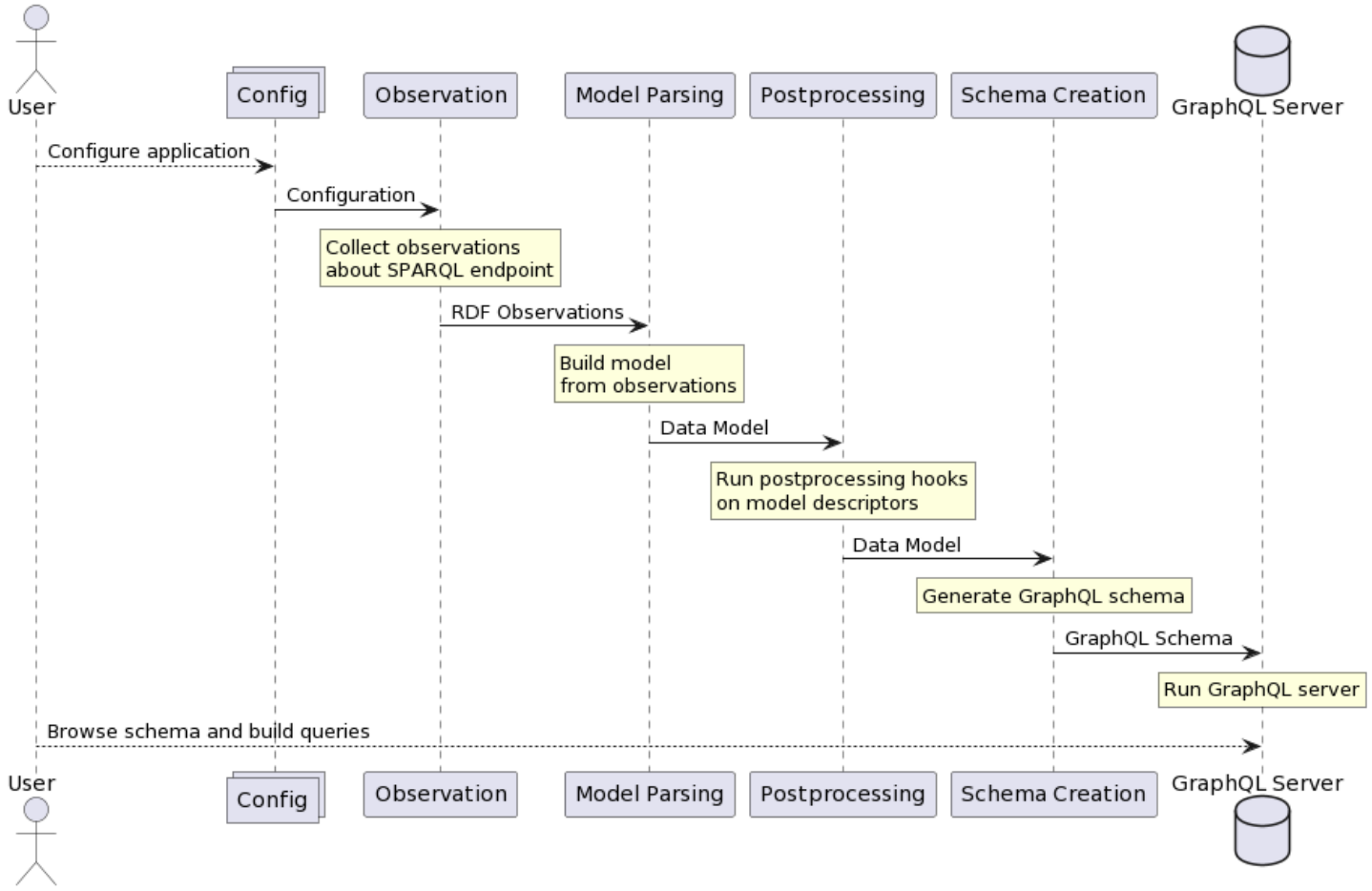
The observation phase uses SPARQL queries to extract some information about the schema of the data in the endpoint. There are many kinds of observations, but the most basic ones are along the lines of `X is a class with N instances` or `Y is a property of class X`. These observations are in the form of RDF data conforming to an ontology defined by SPARQL2GraphQL. This phase can take up to tens of minutes to execute if the SPARQL endpoint contains a large amount of data.

After the observations are collected, the model parsing phase converts these observations into a set of JavaScript objects describing the data schema. These objects are referred to as **descriptors**, since they *describe* the data model.

The postprocessing phase is meant to execute functions called **postprocessing hooks** on descriptors (parts of the model), after the whole model has been built. An example of a postprocessing is giving each property and class in the model a short human-readable name, which it does not necessarily have in the original RDF representation.

The schema creation phase takes the model and its descriptors, and converts it into a GraphQL schema.

After all of these phases are performed, a GraphQL endpoint is started using the generated schema (using Apollo Server). The whole flow we just described looks like this:



As the diagram shows, at the end of the setup, there is a fully functional GraphQL endpoint active, which the user is free to explore and query. Visiting the endpoint URL in the browser will redirect the user to a playground, which provides helpful tools for exploring and querying.

3.1.2 Querying flow

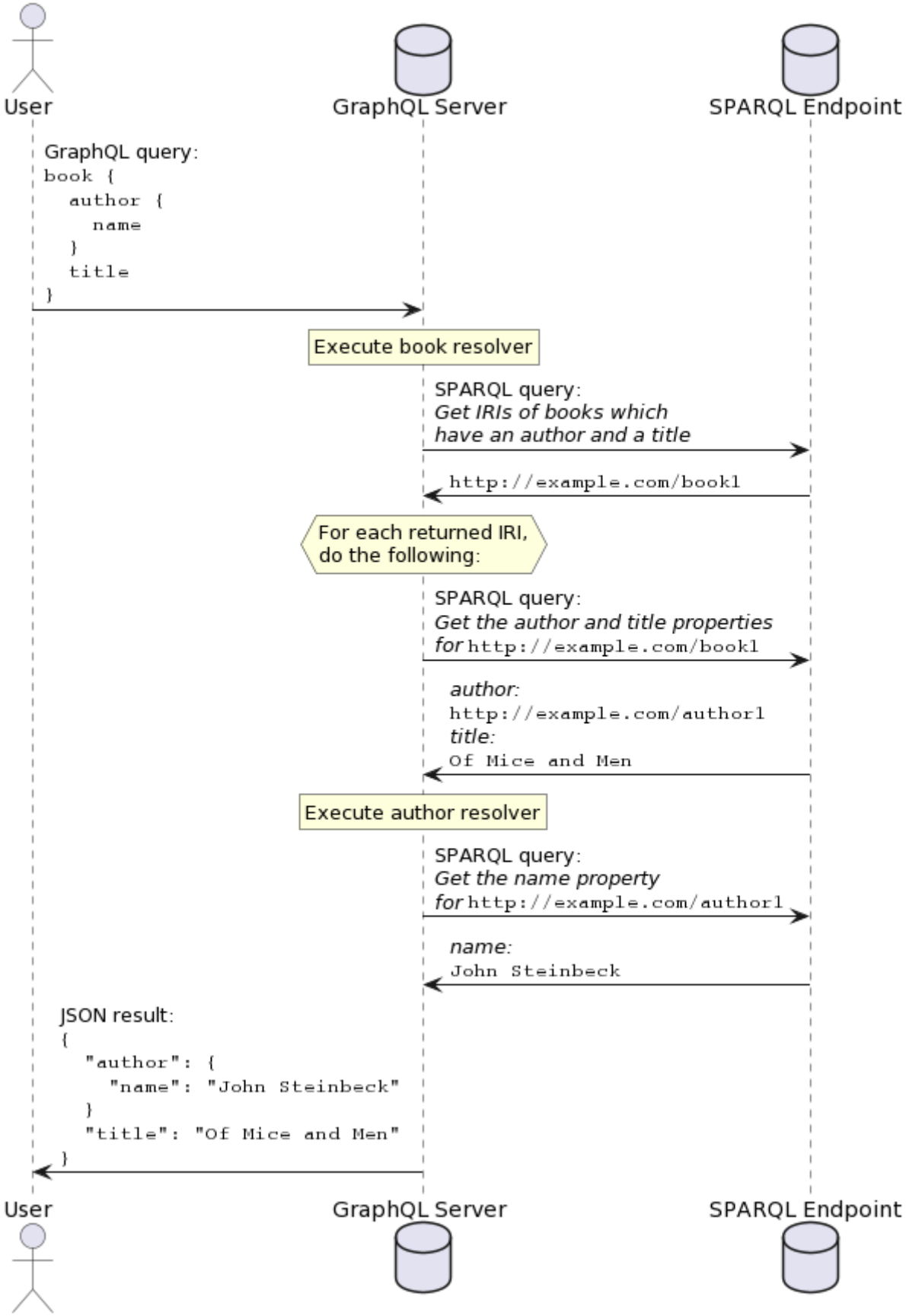
SPARQL2GraphQL utilizes a query translation approach, whereby incoming GraphQL queries are translated into SPARQL queries, executed on the SPARQL endpoint, and their results are aggregated into a JSON response returned by the GraphQL endpoint.

GraphQL querying is based on the concept of resolvers, which are functions, each of which is able to return a single requested property. If the following GraphQL query is sent to the GraphQL endpoint:

```
book { author { name } title }
```

then a resolver will be called for the `books` field, and when that resolver returns an object, resolvers for the `author` and `title` fields will be run. SPARQL2GraphQL resolvers effectively translate each field query into a SPARQL query if the queried field contains a non-scalar value. Scalar values are fetched with the parent object's query.

Execution of this example query would look like this:


User
GraphQL Server
SPARQL Endpoint
GraphQL query:
book {
author {
name
}
title
}
Execute book resolver
SPARQL query:
Get IRIs of books which
have an author and a title
http://example.com/book1
For each returned IRI,
do the following:
SPARQL query:
Get the author and title properties
for http://example.com/book1
author:
http://example.com/author1
title:
Of Mice and Men
Execute author resolver
SPARQL query:
Get the name property
for http://example.com/author1
name:
John Steinbeck
JSON result:
{
"author": {
"name": "John Steinbeck"
}
"title": "Of Mice and Men"
}
User
GraphQL Server
SPARQL Endpoint

3.1.3 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
- Model Parsing
- Postprocessing
- Schema Creation
- Querying
- Model Checkpointing
- 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
```

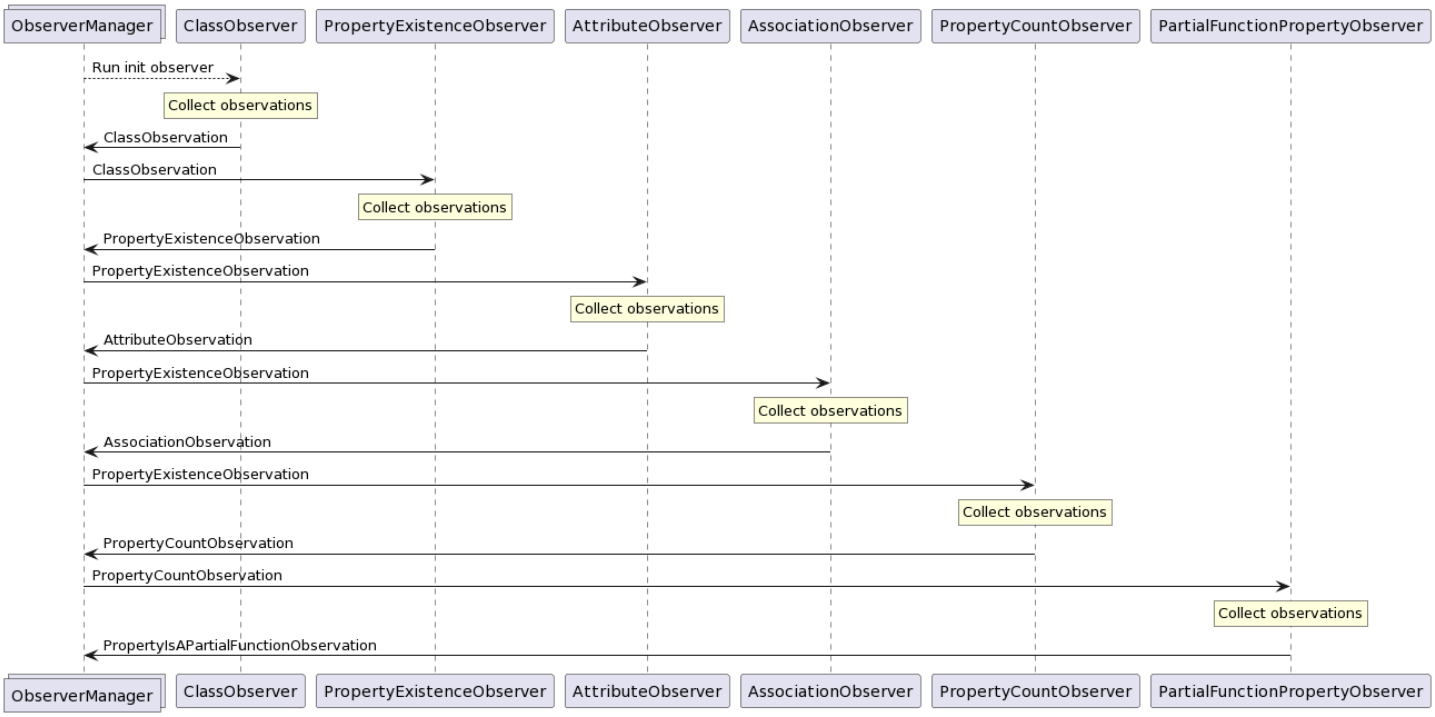
3.2.3 Implemented observers

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. These describe the existence of classes and their number of instances in the dataset.
- `PropertyObserver`: collects `PropertyExistenceObservation`s. These describe the existence of properties on given classes.
- `AttributeObserver`: collects `AttributeObservation`s. These describe attributes, i.e. properties whose ranges contain literals (strings, ints, booleans, ...).
- `AssociationObserver`: collects `AssociationObservation`s. These describe associations whose ranges contain other classes.
- `PropertyCountObserver`: collects `PropertyCountObservation`s. These count the number of occurences of a given property in the dataset. This can be useful for users to determine which classes are important in the data.
- `PartialFunctionObserver`: collects `PartialFunctionObservation`s. These describe properties which are guaranteed not to be array-type properties, i.e. properties which are instantiated no more than once for any given class instance.
- `InstanceObserver`: collects `InstanceObservation`s. These describe the existence of class instances. *(currently unused)*

The data flow between these observers looks like this:



3.2.4 Observation speed

Essential observations for constructing the GraphQL schema are `ClassObservation`, `PropertyExistenceObservation`, `AttributeObservation` and `AssociationObservation`. Other observations are unnecessary, and they just enhance the schema further. If the fastest startup time is desired, you may want to consider disabling non-required observations, and configuring hot reloading to perform these additional observations in the background while the GraphQL endpoint is already functional.

An interesting bit of information about observation query performance is the fact that in SPARQL endpoints with multiple graphs, adding a `GRAPH ?g { ... }` clause around the query body led to significant performance improvements on certain queries. Namely queries with many class instances across multiple graphs which originally took tens of minutes now take a few minutes at maximum. While this change did slightly increase execution times for other queries, the total observation time for endpoints with mulitple graphs has improved dramatically.

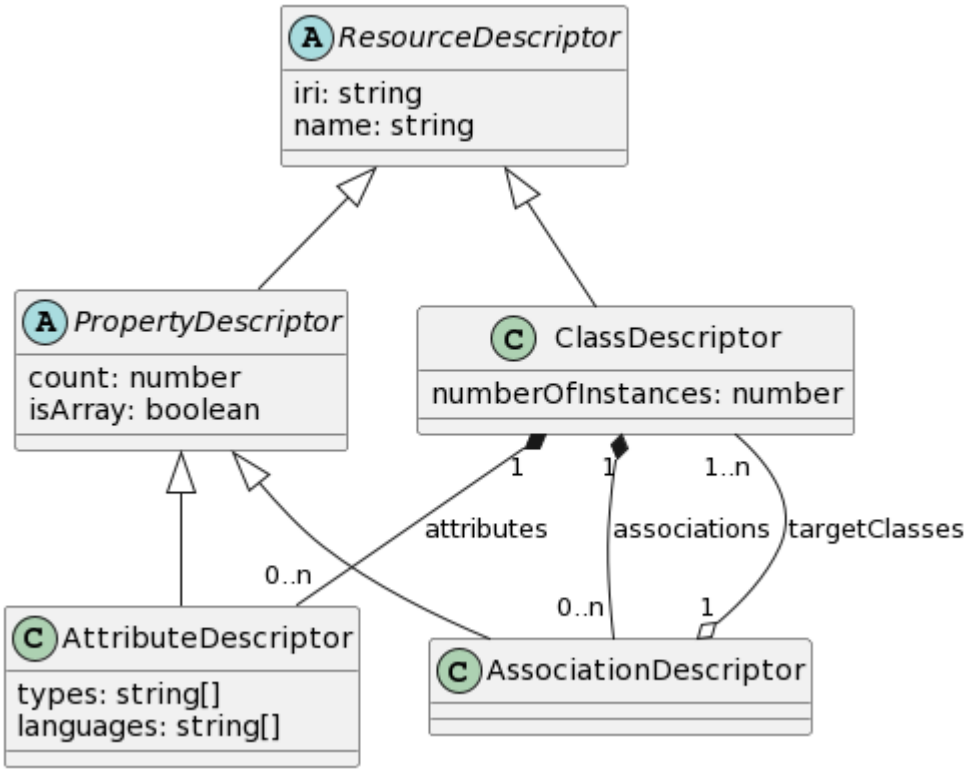
3.3 Parsing

Observations made about a SPARQL endpoint are then parsed by the `ObservationParser` class. It contains an implementation of an algorithm which converts RDF observations into **descriptors**. **Descriptors** are plain JavaScript objects which describe the model of the data in the SPARQL endpoint. They are the stepping stone between the RDF observations, and creation of the GraphQL schema.

This parsing algorithm is not the only possible parsing algorithm for the observations we are collecting. Observations are simply pieces of information to be interpreted, and there are many ways of interpreting that information. The parsing algorithm implemented is simply one which uses all of the observations SPARQL2GraphQL is currently collecting efficiently. There are currently no plans to let library users specify custom parsing algorithms, but should the need arise, the library design is modular enough to allow it.

3.3.1 Descriptors

This section contains a more thorough explanation of what each descriptor means, and what it says about the underlying data. The following is a class diagram showing all of the currently used descriptors:



ResourceDescriptor

Describes any RDF resource which is identified by an IRI, i.e. classes and properties. This descriptor also has a name field, which is meant to contain a GraphQL-compatible, human-readable name, meaning a string only consisting of characters `[_a-zA-Z0-9]`. This name is used as the GraphQL identifier for this resource. After all, it's much easier for humans to read and reason about `Label` than `http://www.w3.org/2008/05/skos-xl#Label`.

```
interface ResourceDescriptor { iri: string; name: string; }
```

This descriptor is extended by all other descriptors.

ClassDescriptor

Describes a single class present in the dataset.

```
interface ClassDescriptor extends ResourceDescriptor { numberOfInstances: number; attributes: AttributeDescriptor[]; association
```

For example, if the dataset contains 5 dogs and 10 cats, we would have the following class descriptors:

```
[ { iri: 'http://example.com/dog', name: 'Dog', numberOfInstances: 5, attributes: [], associations: [], }, { iri: 'http://exampl
```

PropertyDescriptor

Describes a property of a class. A property is any relation in the dataset where the subject is a class. This descriptor is not instantiated on its own, rather it is always as an instance of an `AssociationDescriptor` or an `AttributeDescriptor`.

```
interface PropertyDescriptor extends ResourceDescriptor { count: number; isArray: boolean; }
```

AssociationDescriptor

A special case of a property where its domain is another class. If an association has multiple target classes, its range is their union.

```
interface AssociationDescriptor extends PropertyDescriptor { targetClasses: ClassDescriptor[]; }
```

In our cats and dogs example, let's say each cat has a friend among the dogs. That would be modelled like so:

```
{ iri: 'http://example.com/cat', name: 'Cat', numberOfInstances: 10, attributes: [], associations: [{ count: 10, isArray: false,
```

AttributeDescriptor

A special case of a property where its domain is a literal, i.e. a primitive value like a string or an integer.

```
interface AttributeDescriptor extends PropertyDescriptor { types: string[]; languages: string[]; }
```

The `types` array contains a list of types in this attribute's range. If the `types` array contains a language string type, then the `languages` array will hold a list of available languages for this language string.

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 `ResourceDescriptor` 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 run your own hooks by adding them in the `postprocessing` field of the root configuration object. This field takes a `PostprocessingConfig`, and its `hooks` field contains a dictionary containing the registered hooks for each descriptor type. These hooks are then automatically executed as necessary.

NOTE: it is recommended to use `DEFAULT_POSTPROCESSING_CONFIG` as the basis for adding your new hooks, since it also contains a registered hook for generating GraphQL identifiers for classes and properties. Only create a whole new `PostprocessingHookDict` if you want to replace the functionality of the `buildNamesFromIRIs` postprocessing hook included in this library.

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.

3.5.1 Schema specifics

Classes

All classes observed in the dataset have a corresponding type created in the GraphQL schema, and this type is always queryable from the root query.

Types in the root query support the following arguments:

- sort: takes either `ASC` or `DESC`, will ensure the ordering of results. Class instances are sorted by their IRI.
- limit: do not return more than `limit` results
- offset: skip the first `offset` instances
- filter: given an IRI, only return the instance with this IRI

Attributes

Attributes are represented as fields on the source type. Strings are mapped to string fields, ints are mapped to int fields, booleans are mapped to boolean fields and lang strings (strings which are tagged with a language in the RDF dataset) are mapped to objects containing the respective languages as fields.

Other types are mapped to string fields by default.

Attribute fields except for language strings and booleans support `sort`, `limit` and `offset` arguments. For booleans these arguments do not make sense. They may be implemented for language strings in the future.

Associations

Associations are represented as fields on the source type, with the target type being one of the following:

- The target type if the associaton only has a single target type
- A newly created union type joining all target types of the associaiton in case the association has multiple target types.

Association fields support `sort`, `limit`, `offset` and `filter` arguments.

3.6 Querying

NOTE: This introduction is identical to the introduction given in the overview. If you have already read it, you may skip to the other subsections of this article.

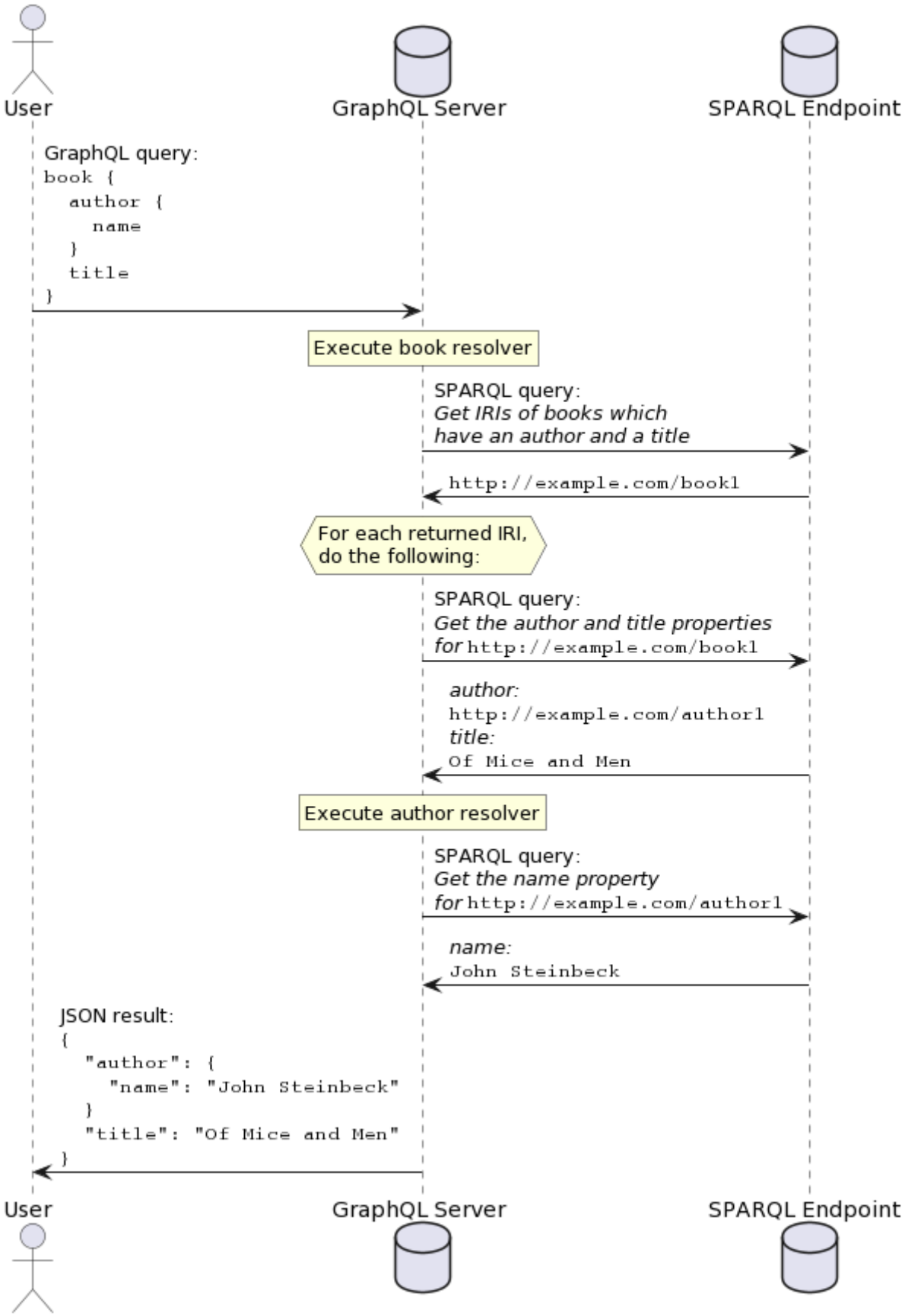
SPARQL2GraphQL utilizes a query translation approach, whereby incoming GraphQL queries are translated into SPARQL queries, executed on the SPARQL endpoint, and their results are aggregated into a JSON response returned by the GraphQL endpoint.

GraphQL querying is based on the concept of resolvers, which are functions, each of which is able to return a single requested property. If the following GraphQL query is sent to the GraphQL endpoint:

```
book { author { name } title }
```

then a resolver will be called for the `books` field, and when that resolver returns an object, resolvers for the `author` and `title` fields will be run. SPARQL2GraphQL resolvers effectively translate each field query into a SPARQL query if the queried field contains a non-scalar value. Scalar values are fetched with the parent object's query.

Execution of this example query would look like this:


User
GraphQL Server
SPARQL Endpoint
GraphQL query:
book {
author {
name
}
title
}
Execute book resolver
SPARQL query:
Get IRIs of books which
have an author and a title
http://example.com/book1
For each returned IRI,
do the following:
SPARQL query:
Get the author and title properties
for http://example.com/book1
author:
http://example.com/author1
title:
Of Mice and Men
Execute author resolver
SPARQL query:
Get the name property
for http://example.com/author1
name:
John Steinbeck
JSON result:
{
"author": {
"name": "John Steinbeck"
}
"title": "Of Mice and Men"
}
User
GraphQL Server
SPARQL Endpoint

3.6.1 Query root execution

NOTE: The following description of query execution functionality describes the way it is implemented in this library. This is not the only possible way of implementing querying, and the design of this library accounts for different possible implementations in the future.

Each GraphQL query has a root `Query` object, whose fields are all object types extracted from the SPARQL endpoint. Each of these fields is a non-null list of objects, which are resolved with a **class resolver** (implemented by the `createClassResolver` factory function).

The **class resolver** is a function which is called by Apollo Server any time class instances are requested as part of a GraphQL query. Its job is to return the requested instances as plain JavaScript objects. It does this by looking at the requested fields for these objects, and executing a SPARQL query which returns a list of IRIs representing the objects to be returned from the resolver.

If we take the following GraphQL query:

```
dcat_Dataset(limit: $limit) { title { cs en } conformsTo }
```

then the corresponding SPARQL IRI query will look like this:

```
SELECT DISTINCT ?instance WHERE { ?instance a <http://www.w3.org/ns/dcat#Dataset> . ?instance <http://purl.org/dc/terms/title> [
```

Afterwards, the **class resolver** takes the received instance IRIs, and for each of them, it executes a SPARQL query asking for that instance's properties which have been requested by the GraphQL query.

Assuming one of the IRIs returned by the previous query was `https://data.gov.cz/zdroj/datové-sady/00297569/3384952`, then the followup query asking for this instance's properties will look like this:

```
SELECT ?property ?value WHERE { VALUES (?property) { ( <http://purl.org/dc/terms/title> ) ( <http://purl.org/dc/terms/conformsTo
```

Finally, the **class resolver** will create a plain JavaScript object containing the returned properties, and it will return it to Apollo Server. Apollo Server will then *execute a resolver for each field on this object*.

These additional resolvers are responsible for taking the data retrieved by the class resolver, and converting it into the appropriate format for return as the GraphQL query response. What this means in practice is that while everything is stored as a string in RDF, these additional resolvers have to convert these strings to the appropriate data types for their respective fields. For example, a boolean field resolver would parse the retrieved string into a boolean.

This chaining of resolvers will occur recursively for each field of objects nested in the query, until all fields have been resolved. The final GraphQL response for our example scenario will look like this:

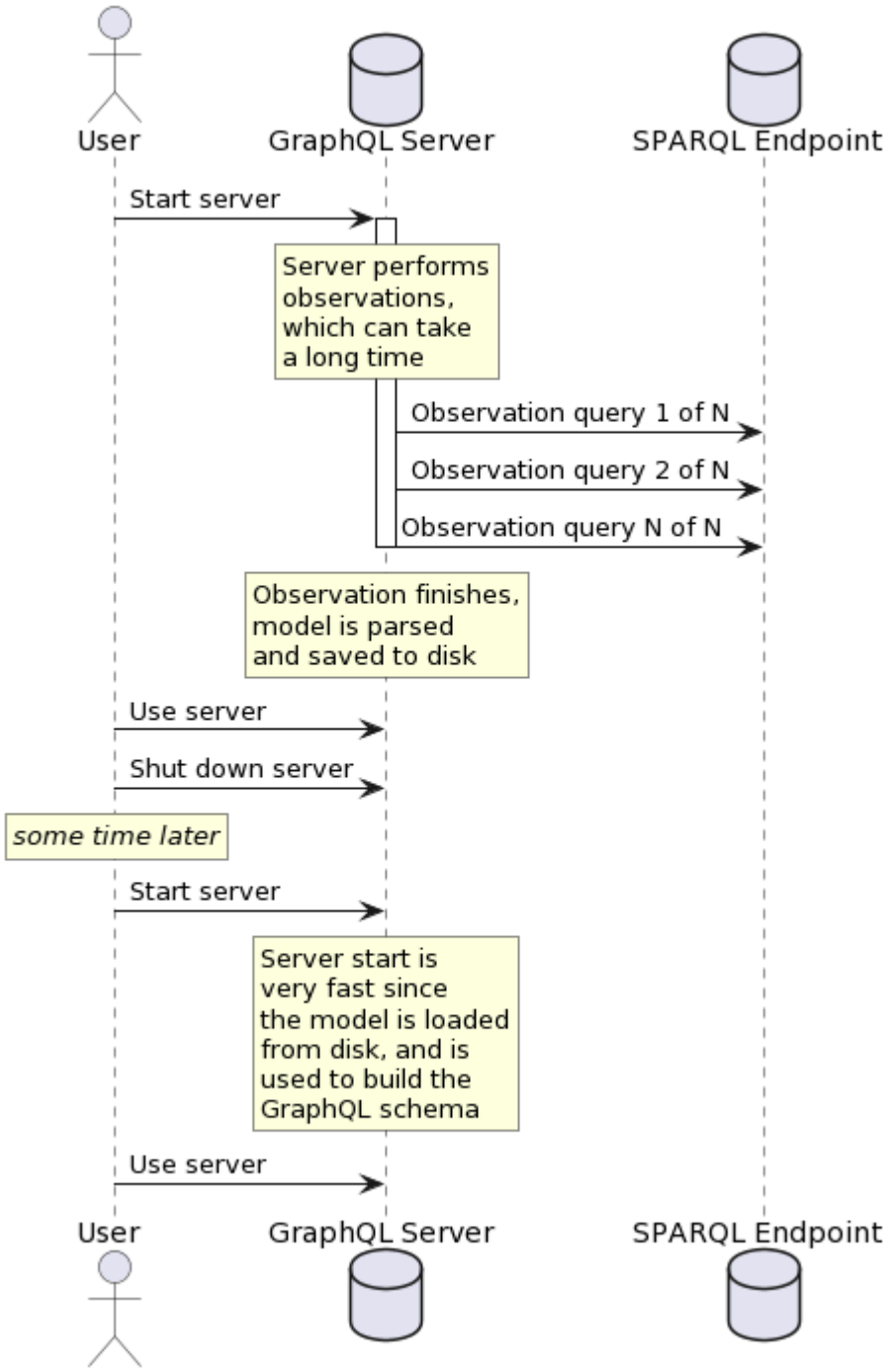

```
"data": { "dcat_Dataset": [ { "title": { "cs": [ "Moravskoslezský kraj - úřední deska" ], "en": [ "Official bulletin board of Mo
```


The concept of resolvers is described in more detail in the official GraphQL documentation in case you wish to learn more.

3.7 Model Checkpointing

The observation phase can take quite a long time, and it is undesirable to have to wait for the entire observation phase to happen again every time you want to start the GraphQL endpoint.

Therefore SPARQL2GraphQL allows you to use the following options to save the generated data model to disk, and then later reuse it instead of carrying out observations again.




```
interface ModelCheckpointConfig { loadModelFromCheckpoint: boolean; saveModelToFile: boolean; overwriteFile: boolean; checkpoint
```

`checkpointFilePath` points to the file which will be used to store/load model checkpoints.

If `saveModelToFile` is set to `true`, the model will be saved to the configured file after it is built from the observations. If the file already exists and `overwriteFile` is set to `false`, the file will not be overwritten by the new checkpoint.

If `loadModelFromCheckpoint` is set to `true`, at startup, SPARQL2GraphQL will check whether the checkpoint exists. If it does, then it will use the model checkpoint rather than carrying out observations again.

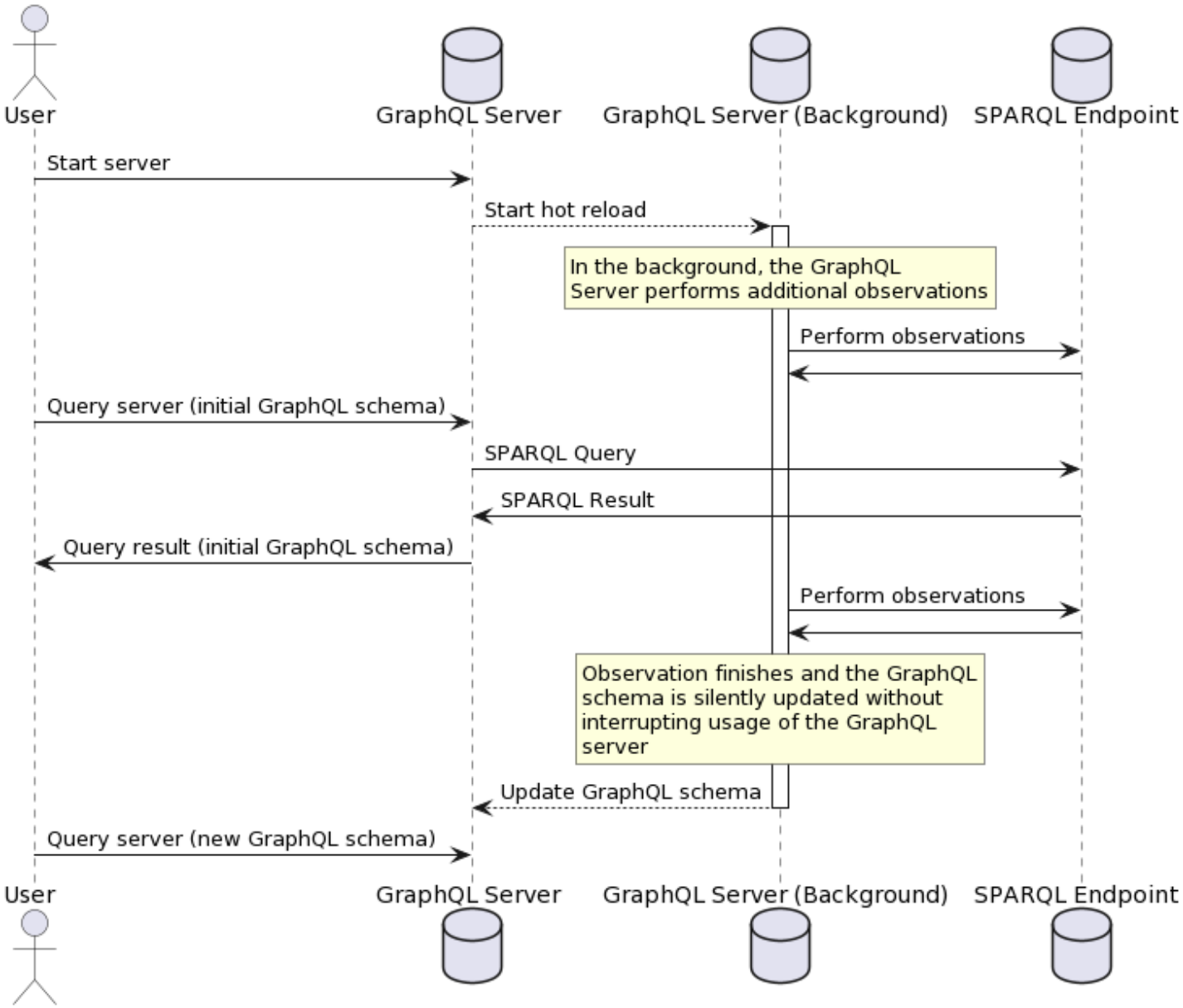
3.8 Hot reloading

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.

In the following diagram, you can see how the GraphQL schema continues to be updated in the background while the user is free to explore and query the endpoint:



3.8.1 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.yaml` for both the website and the PDF documentation, and you may safely ignore this error.

4.2.4 Diagrams

Diagrams are generated with PlantUML, which is a tool for generating UML diagrams out of text descriptions.

The source files for all diagrams contained in this documentation are in the `diagrams` folder.

4.2.5 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).