ceiba Documentation

Release 1.0.0

Felipe Zapata

Mar 22, 2021

CONTENTS:

CHAPTER

ONE

CEIBA

Scientific simulations generate large volume of data that needs to be stored and processed by multidisciplinary teams across different geographical locations. Distributing computational expensive simulations among the available resources, avoiding duplication and keeping the data safe are challenges that scientists face every day.

Ceiba and its command line interface Ceiba-cli. Ceiba solves the problem of computing, storing and securely sharing computationally expensive simulation results. Researchers can save significant time and resources by easily computing new data and reusing existing simulation data to answer their questions.

See documentation and blog post.

1.1 Installation

The [provisioning folder](https://github.com/nlesc-nano/ceiba/tree/main/provisioning) contains the instructions to deploy the web service using docker. Alternatively, you can deploy the web service locally using the following instructions:

1. Install Docker
2. Define a environment variable *MONGO_PASSWORD* with the database password. Now you can run the following command to start both the server and the mongodb services:

```
provisioning/start_app.sh
```

1.2 Contributing

If you want to contribute to the development of ceiba, have a look at the contribution guidelines.

1.3 License

1.4 Credits

This package was created with Cookiecutter and the NLeSC/python-template.

CHAPTER TWO

THE CEIBA WEB SERVICE

Most of the scientific simulations are usually performed in supercomputer or high tech facilities. Usually the data is kept on those facilities stored in a raw format, in contradiction with the scientific FAIR principles for data.

This repo contains a library to create a web service to interact with a database containing a set of numerical properties. All the interactions with the database are defined by a GraphQL API and the service is developed using tartiflette

CHAPTER

THREE

INTERACTIONS WITH THE DATABASE

Using the GraphQL query language the service define a set of rules to interact with the services: **queries** and **mutations**.

The queries are just read only actions against the database, while the mutations, involved some change in the database state.

CHAPTER

FOUR

SCHEMA QUERIES

See the available queries schema definitions.

CHAPTER

FIVE

SCHEMA MUTATIONS

See the available queries schema definitions.

CHAPTER

SIX

QUERIES

Queries are defined using the schema definition language. You can find the mutations definition ceiba/sdl/Query.graphql

Queries involved *read-only* interactions between the client and the database.

CHAPTER

SEVEN

MUTATIONS

Mutations are defined using the schema definition language. You can find the mutations definition at ceiba/sdl/Mutation.graphql

Mutations involved a change in the database state, requested by the client. **All the mutations required authentication**.

CHAPTER

EIGHT

DATA LAYOUT

The data layouts specifies how is the data and its metadata store in the database. Since we use Mongodb for the database, we will also explain the data layout using Mongobd's terminology.

8.1 Properties collections

The following schema defines how the properties are stored:

```
# Unique identifier
_id: Integer

# Name to which the property belongs. e.g. Theory level
collection_name: String

# Metadata associated with the given property
metadata: String

# Properties values as JSON
data: Optional[String]

# Input with which the property was computed encoded as JSON
input: Optional[String]
```

Notice that the previous schema mirros the GraphQL definition of Property in the server.

8.2 Jobs collections

The following schema defines how jobs are defined:

```
# Unique identifier
id: Integer

# compute Properties
property: Ref[Property]

# Input to perform the computation
settings: String

# Job status
status: Status
```

(continues on next page)

(continued from previous page)

```
# User who es executing the job
user: Optional[String]

# Timestamp = datatime.timestamp()
schedule_time: Optional[Float]

# Timestamp = datatime.timestamp()
report_time: Optional[Float]

# platform where the job was run: platform.platform()
platform: Optional[String]
```

Notice that the previous schema mirros the GraphQL definition of Job in the server.

Note:

- Optional[T] is a type that could be either `None` or some `T`.
- References `ref` are implemented as Mongodb DBRefs.
- Jobs are stored in a collections named like `jobs_<property_collection_name>`.

CHAPTER

NINE

INDICES AND TABLES

- genindex
- modindex
- search