fBPMN-API Documentation

Paul Malinvaud @paulmalinvaud

2021-08-31

Flask-Project

The application is mainly based on the Flask package. It is possible to run an application with this simple package, but adding features makes it easier to use. The main aggregations are initialized in the _*init*_.py file of the app directory.

*init*.py

The few lines in this file are essential to understand how the Python backend of the application works.

```
app = Flask(__name__)
app.config.from_object(Config)
db = SQLAlchemy(app)
migrate = Migrate(app, db)
cors = CORS(app, resources={r"/*": {"origins": "http://localhost:3000"}})
ma = Marshmallow()
```

The first line is found in all Flask projects and initializes the Flask object. A configuration is specified in the config.py file. db is the instance of SQLAlchemy (documentation here https://flask-restplus.readthedocs.io/en/stable/) associated with app. SQLAlchemy is an ORM and allows to build a database from Python objects. This will allow us to think of the Flask app in terms of objects and not in terms of data. Flask-Migrate is an extension that handles SQLAlchemy database migrations for Flask applications using Alembic. The database operations are made available through the Flask command-line interface(not really useful as long as we don't want to centralise all the data). CORS (documentation here https://flask-cors.readthedocs.io/en/latest/) is a a Flask extension for handling Cross Origin Resource Sharing. Marshmallow (documentation here https://marshmallow.readthedocs.io/en/stable/) is an ORM/ODM/framework-agnostic library for converting complex datatypes, such as objects, to and from native Python datatypes.

models.py

The models.py file declares the classes that will be used for fbpmn checks via the web-application. Here is a class diagram:

App class diagram

```
@startuml fBPMN_backend

enum Communication {
    Network01Bag
    Network02FifoPair
    Network03Causal
    Network04Inbox
    Network05Outbox
    Network06Fifo
    Network07RSC
}
enum Value {
    FAIL
    SUCCESS
    INCONCLUSIVE
}
enum Status  {
    PENDING
    DONE
    ABORTED
}

class Model {
    +id : int
    +name : string
    +content : string
    +verification : Verification
    +Model()
}

class UserNets {
    +id : int
    +name : string
    +content : string
    +verification : Verification
    +UserNets()
}

class UserDefs {
    +id : int
    +name : string
    +content : string
    +verification : Verification
    +UserDefs()
}

class UserProps {
    +id : int
    +name : string
    +content : string
    +verification : Verification
    +UserProps()
}

class Constraints {
    +id : int
    +name : string
    +content : string
    +verification : Verification
    +Constraints()
}
```

```
class Verification {
    +id : int
    +status : Status
    +value : Value
    +pub_date : Date
    +duration: int
    +output : string
    +model_id : int
    +usernets_content : string
    +userdefs_content : string
    +userprops_content : string
    +constraints_content : string
    +results : Result
    +Verification()
    +aborted()
    +all_ok()
    +create_model(in content : string)
    +create_file(in type : string, in content_request : string, in model_name :
string)
    +create_properties_files(in def_content : string, in prop_content : string, in
model_name : string)
    +launch_check(in model_name : string)
    +create_results_list(in workdir : string, in model_name : string)
    +create_results_list(in workdir : string, in model_name : string, in results_list
: [Result])
}

class Result {
    +id : int
    +communication : Communication
    +property : string
    +value : boolean
    +counter_example: CounterExample
    +verification_id : int
    +Result()
    +create_counter_example(in workdir : string, in model_name : string)
    +is_ok()
}

class CounterExample {
    +id : int
    +lcex : string
    +lstatus : string
    +lname : string
    +lmodel : string
    +results_id : int
}

class Application {
    +Application()
    +create_verification()
    +get_all_elements(in type : string)
    +get_element_by_id(in type : string, in id : int)
    +get_latest_verification()
    +is_ok_verif(in verification_id : int)
    +is_ok_result(in result_id : int)
}

Result "*" --* Verification
Model "1" --* Verification
UserDefs "1" --* Verification
UserNets "0,1" --* Verification
UserProps "1" --* Verification
Constraints "1" --* Verification
CounterExample "0,1" --* Result
Result -- Communication
Verification -- Status
Verification -- Value
Application *-- "1" Verification

@enduml
```

schemas.py

In schemas.py, our classes seen above are serialized to present the API information in JSON. Here is an example of a Schema created for the Result class:

```
class ResultSchema(ma.SQLAlchemyAutoSchema):
    class Meta:
        model = Result
        include_relationships = True

    communication = EnumField(Communication)
    verification = ma.HyperlinkRelated(VERIFICATION_BY_ID)
```

Our class inherits from ma.SQLAlchemyAutoSchema which is useful for serializing classes created with SQLAlchemy. ma.SQALAlchemySchema is also an alternative but is not automatic. In our case, we only need to specify some information in Meta like the Model (here Result). We also modify the verification field to make it a link to the resource in the API instead of a simple number, we make that for respect the HATEOAS principle. HATEOAS stands for Hypermedia As The Engine Of Application State. It is a part of REST application architecture and is a constraint for REST.

resources.py

Flask-RESTPlus (documentation here https://flask-restplus.readthedocs.io/en/stable/#welcome-to-flask-restplus-s-documentation) is an extension for Flask that adds support for quickly building REST APIs. Flask-RESTPlus encourages best practices with minimal setup. If you are familiar with

Flask, Flask-RESTPlus should be easy to pick up. It provides a coherent collection of decorators and tools to describe your API and expose its documentation properly (using Swagger).

```
@verifications_ns.route(f'{URL_ID}')
class VerificationById(Resource):
    def get(self, id):
        v = a.get_element_by_id(Verification, id)
        if v:
            return (create_schema(VerificationSchema, False)).jsonify(v)
        return {'message': VERIFICATION_NOT_FOUND}, 404

    def delete(self, id):
        v = Verification.query.get(id)
        db.session.delete(v)
        db.session.commit()
        return "Verification was successfully deleted"
```

For each resource we can declare the four methods get, put, post, delete. In the case of get, here, we use a business method of Application (with a.get_element_by_id(Verification, id)) and also the schemas declared before (with VerificationSchema) to serialize

routes.py

Use of all objects instantiated in previous files. Similar to a main class.