NetPlanner-BackEnd

Farzad Mohammadi

Oct 20, 2020

TABLE OF CONTENTS:

This documentation purpose is to explain NetPlanner BackEnd architecture and modules and help future developers to contribute on this code more easily.

NetPlanner different parts are:

- **Web Server** This is front gate of backend and we are using **Nginx** here
- **WSGI Server** This one is a interface that connects *web server* to *Service* and here we are using **Gunicorn**.
- **Service** This is where our actual logic (algorithms, API, ..) implemented and here we are using **Flask**.
- **Database** This is where we are storing algorithms results, users information and etc. here we are using **Postgresql**.

We are going to discus all mentioned parts of backend in separate section

CHAPTER

ONE

IN SERVICE PART WE HAVE BELOW TOPICS:

- Restful API and it’s specification
- Routing of API
- User Management
- Security in API

CHAPTER TWO

IN DATABASE PART WE HAVE BELOW TOPICS:

- ORM system
- Models

2.1 Modules in NetPlanner BackEnd

2.2 Main Objects Data Structure

2.3 Restful API with OpenAPI 3 and Routing

In this section we are going deep to our Restful API.

specification link: OpenAPI 3

We strongly suggest that after reading this documentation (which its purpose isn't covering all of specification aspects) visit above web page and learn more about OpenAPI 3

In order to simplify above specification, we can say our API is consist of several **paths** and each path has several **verbs**.

An example of path: `/PhysicalTopology/read_all`

by verbs, we mean **HTTP method**.

After specifying path we have to define verbs of the path. An example of **get** method for above path:

```
/PhysicalTopology/read_all/{UserId}:
  summary: Reading all PT's
  get:
    tags:
      - Physical Topology
    summary: Reading all PT's stored in Database
    security:
      - jwt: []
    operationId: PhysicalTopology.read_all_PT
    parameters:
      - $ref: '#/components/parameters/UserId'
    responses:
      200:
        description: Returning back all PT's
        content:
          application/json:
```

(continues on next page)

(continued from previous page)

```
            schema:
              type: array
              items:
                type: object
                properties:
                  name:
                    type: string
                  id:
                    type: integer
                    format: int64
                  create_date:
                    type: string
                    format: date-time
                required:
                  - name
                  - id
                  - create_date
      default:
        description: error handling
        content:
          application/json:
            schema:
              type: object
              properties:
                error_msg:
                  type: string
              required:
                - error_msg
```

2.3.1 Usage of different components in our example code

- **tags** Tags used for grouping different paths that are related in some manner
- **security** This one will be discussed in its own section (see Security section)
- **operationId** This one tells that which function in which module is responsible of handling this **(path, verb) pair**. from above code we can tell that function with name *read_all_PT* and module *PhysicalTopology* is responsible for this (path, verb).

  example of (path, verb) pair: *(/PhysicalTopology/read_all/{UserId}, GET)*
- **parameters** As its name speaks, this component defines this (path, verb) parameters. As you can see in this code we have used *Reference* to prevent copying same exact code in multiple places.

  References are defined at the top of specification file. In this case definition is:

  ```
  UserId:
    name: UserId
    in: path
    required: true
    schema:
      type: integer
  ```

  As you can guess *name* is name of this parameter and we will use this in our handler function. after that we have **in** keyword which specifies where is the place of this parameter in HTTP request, *in* can be **query**, **path** or **header**. After that we have *schema* section which defines parameter structure, in this example *Id* is simple integer type and there is no need for advanced schema.

- **responses** In this section we define different possible respond status codes (HTTP status codes) and for each one them we have to describe respond body structure(schema). In our example code we can see that for status code 200 this (path, verb) will return an array of *JSON* object and each JSON object has three mandatory properties with the name of name, ida and create_date.

  Also there is an **default** responses. its usage in our application - *NOT IN GENERAL* - is that we can return any status code other than 200 (in this (path, verb)) with an **error_msg**.

2.3.2 Brief look at handling function

This handling function corresponds to above api specification:

```
def read_all_PT(UserId):
  PTs = PhysicalTopologyModel.query.filter_by(user_id= UserId).all()
  if not PTs:
      return {"error_msg": "no Physical Topology found"}, 404
  else:
      schema = PhysicalTopologySchema(only=("id", "name", "create_date"), many= True)
      return schema.dump(PTs), 200
```

First thing is that we can simply receive our parameter in function. Then we can see application of default responses here as we returned an *error_msg* with 404 status code and at the last line you can see that we are returning a JSON object with status code of 200 (its not clear how *schema.dump(PTs)* is a JSON object but leave it for now we will explain this later in **Models** section).

2.3.3 request body in POST and PUT method

we started with GET method because its the easiest method to explain also its doesn't require request body.

Request Body is much like a responses body but its sent from user of the API to server

An example of request body definition:

```
requestBody:
  description: providing information for creating new Physical Topology
  content:
    application/json:
    schema:
      $ref: '#/components/schemas/PhysicalTopology'
```

First of all this example proves our claim that request body is so much like response body, Second we have used another Reference here for **PhysicalTopology** object.

PhysicalTopology and **TrafficMatrix** objects will be discussed in their own section

An example of extracting request body in handler function is as follow:

```
def create_PhysicalTopology(name, UserId):
  PT = json.loads(request.get_data())
  PT_object = PhysicalTopologyModel(name= name, data= PT)
  db.session.add(PT_object)
  db.session.commit()

  return {"Id": PT_object.id}, 201
```

with *json.loads(request.get_data())* we can extract request body. (you have to import **json** library).

2.3.4 Connection between Flask and OpenAPI

This is though if don't use **connexion** library. In connexion this can be done with this line of code:

```
connex_app = connexion.App(__name__,
                           specification_dir=os.path.join(basedir, "openapi"))
```

In above code we simple gave **openapi.yaml** file path to app initializer.

We will explore more about this in **Modules** section.

2.4 Routing

In this section we are going to explain Routing in NetPlanner Backend

2.5 User Management

In this section we are going to explain user management in NetPlanner BackEnd

2.6 Security in Backend

In this section we are going to discus Security in different modules of NetPlanner

2.7 Object Relational mapper

Here we are going to explore NetPlanner ORM

2.8 Models and Tables in Database

Here we are going to dive into our Database and its structure

2.9 Contact to Developers

- **Farzad Mohammadi**

  email: farzad.mohammadi87@gmail.com

  linkedin: https://www.linkedin.com/in/farzad-mohammadi-119067159/