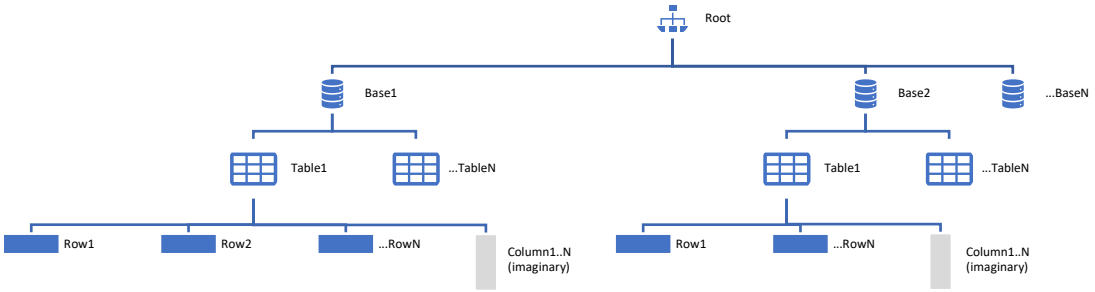
КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ТАРАСА ШЕВЧЕНКА
Спеціальність «Інформатика»

Звіт

З дисципліни «Інформаційні технології»

Виконав
студент групи ТТП-4
Винник Дмитро



Для створення такої структури були описані базові класи елементів дерева:

1) Вершини з/без ідентифікатора:

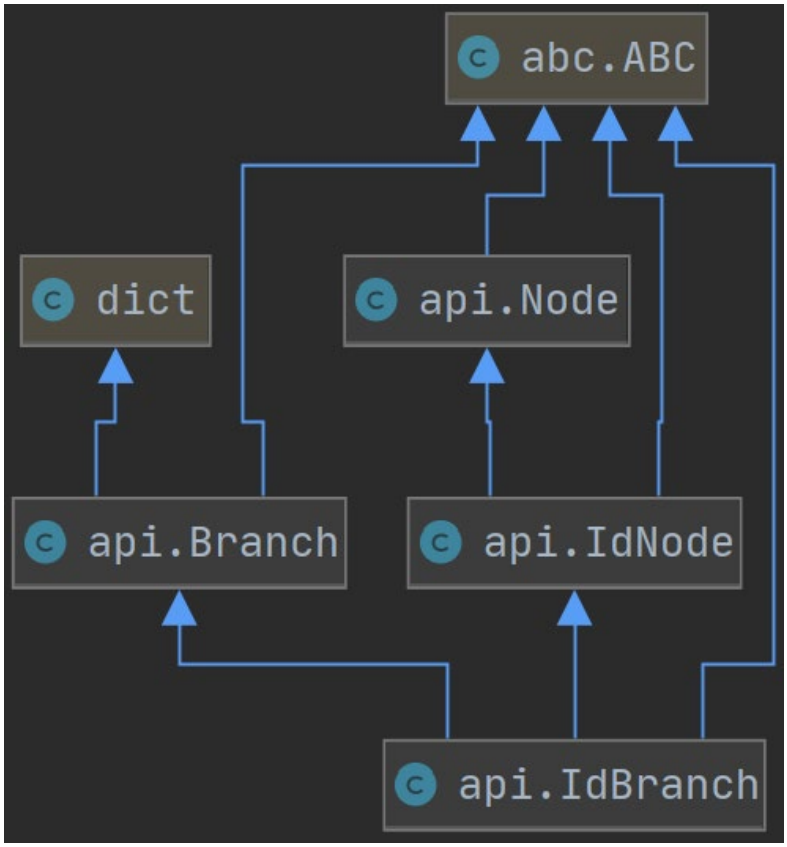
```
13  class Node(ABC):
14      pass
15
16
17  class IdNode(Node, ABC):
18      def __init__(self, id_):
19          self.id_ = id_
```

2) Гілки:

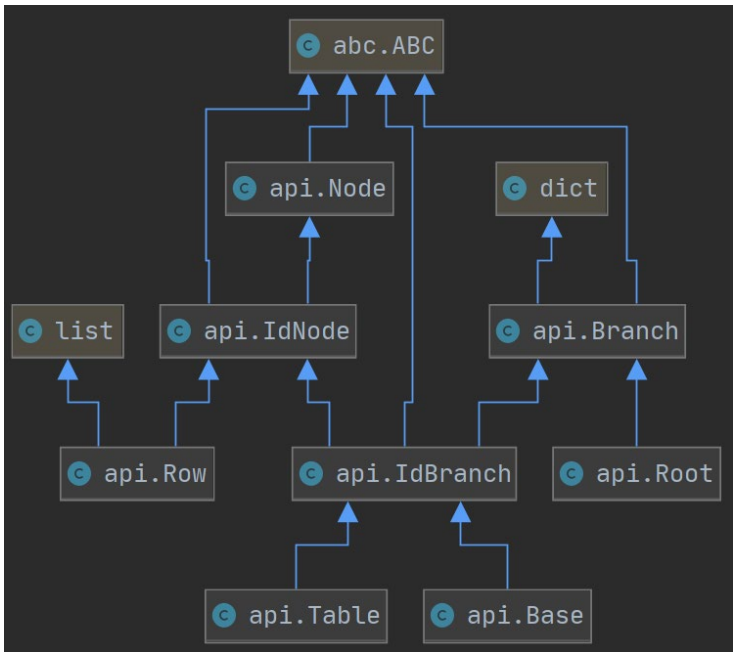
```
22  class Branch(dict, ABC):
23      @property
24      @abstractmethod
25      def children_type(self):
26          raise
27
28      def __init__(self, children: List):
29          super().__init__()
30          for child in children:
31              self.add(child)
32
33      def add(self, child: IdNode, id_=None):
34          assert type(child) == self.children_type, (type(child), self.children_type)
35          if id_ is None:
36              id_ = child.id_
37          if id_ in self:
38              raise Exception('Already exist')
39          self[id_] = child
40
41      def __getitem__(self, item):
42          if isinstance(item, (tuple, list)):
43              child = reduce(getitem, item, self)
44          else:
45              child = dict.__getitem__(self, item)
46
47          if isinstance(child, Branch):
48              child.children = [child[child_id] for child_id in child.keys()]
49          return child
50
51      def __setitem__(self, key, value):
52          call_func = inspect.stack()[1][3]
53          assert call_func == 'add'
54          dict.__setitem__(self, key, value)
```

(Рядок 34 перевіряє приналежність об’єкта до типу дітей цієї гілки. Наприклад при виклику add у Base аргумент перевіриться на приналежність до Table.
Рядки 42-43 дозволяють індексування через кому, наприклад tree[‘base_id’, ‘table_id’]
Рядки 47-48 потрібні для примусового виклику перевантаженого гетера у вкладених об’єктах)

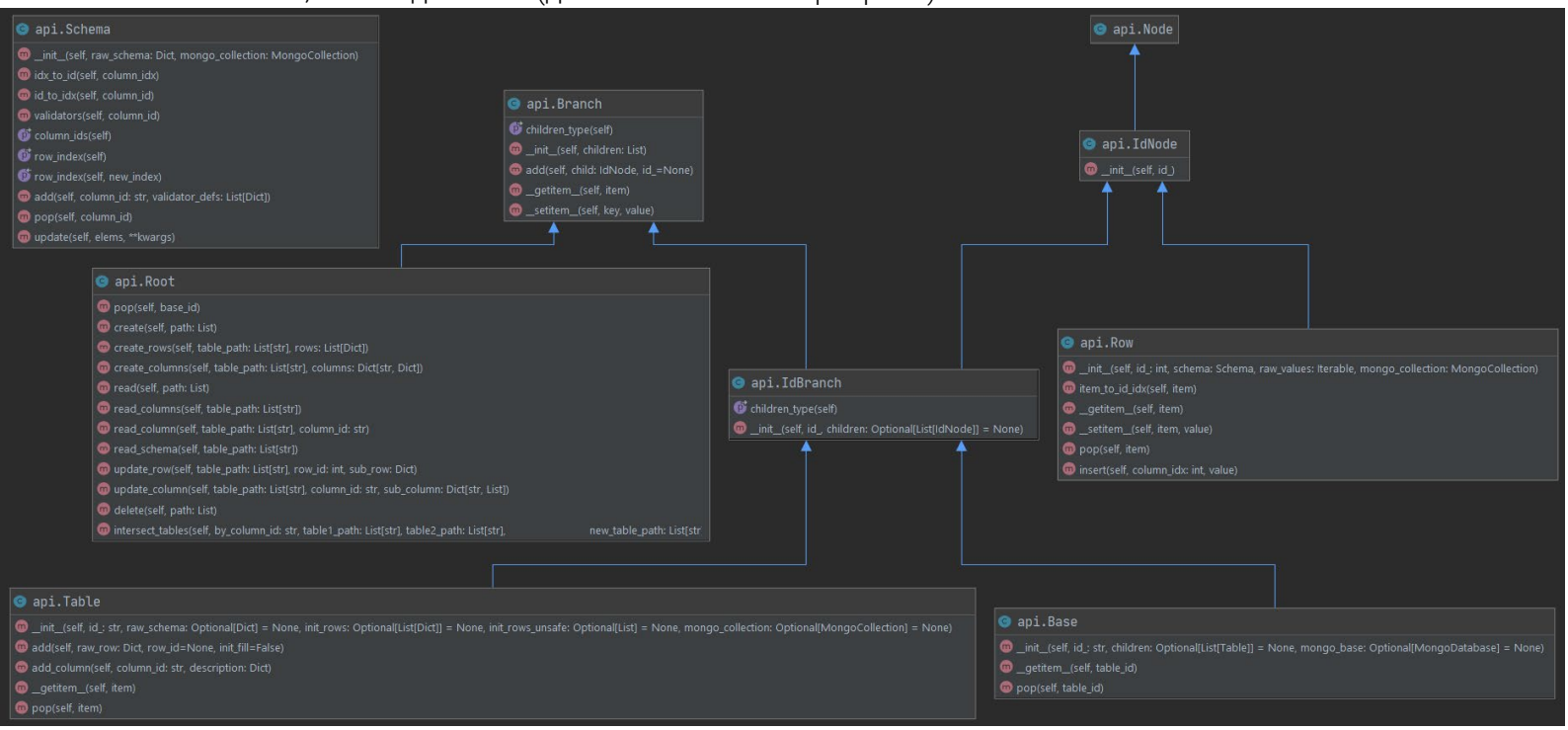
Ось так виглядає діаграма наведених вище класів:



На основі цих класів побудовані класи Root, Base, Table, Row:



Класи, їх методи і поля (деякі базові класи прибрано):



Row наслідується від list, через зручність вставляння нового елемента у певну позицію, чого не можна легко зробити у випадку з dict. Тому окремо від рядків в об'єкті таблиці зберігається схема наступного виду:

{'columns': {column_id1: column_description1, column_id2: column_description2, ...}, ...}

У column_description міститься опис колонки, наприклад її валідатори.

Через глобальну змінну у модулі *api* можна вмикати і вимикати mongo:

```
70        mongo_client: Optional[MongoClient] = None
```

-- Якщо клієнт заданий, то зміни до бази, що в оперативній пам'яті, будуть дублюватися на mongo-базу, що на диску.

Схема створюється з, і зберігається у звичайному dict:

```
73      class Schema(dict):
74          def __init__(self, raw_schema: Dict, mongo_collection: MongoCollection):
75              super().__init__()
76              if mongo_client:
77                  self.mongo_collection = mongo_collection
78              self.update(raw_schema)
79              if not raw_schema:
80                  self['columns'] = {}
81                  self['row_index'] = -1
82
83                  if mongo_client:
84                      self.mongo_collection.update_one(
85                          {'id': 'schema'},
86                          {'$set':
87                              {'columns': self['columns'],
88                               'row_index': self['row_index']}
89                           },
90                          upsert=True
91                      )
```

Перетворення цілочисельного індексу list-рядка на str назву колонки й навпаки:

```
93          def idx_to_id(self, column_idx):
94              return self.column_ids[column_idx]
95
96          def id_to_idx(self, column_id):
97              return self.column_ids.index(column_id)
```

Створення валідаторів колонки з їх описів (validator_def):

```
99      def validators(self, column_id):
100         validators = []
101         for validator_def in self['columns'][column_id]['validator_defs']:
102             # the validators module contains safe classes
103             validator_type = locate(f'validators.{validator_def["name"]}')
104             assert isinstance(validator_type, type)
105
106             validators.append(validator_type(**validator_def['params']))
107         return validators
```

(у рядку 103, у модулі validators шукається клас з ім’ям validator_def[“name”])

Приклади опису валідаторів:

```
{'name': 'TypeValidator', 'params': {'type_descr': 'int'}}
```

```
{'name': 'EmailValidator', 'params': {}}
```

То як виглядають їх класи:

```
class Validator(ABC):
    @abstractmethod
    def __call__(self, value):
        pass


class TypeValidator(Validator):
    def __init__(self, type_descr: str):
        self.type_ = locate(type_descr)

    def __call__(self, value):
        assert isinstance(self.type_, type)
        return isinstance(value, self.type_)


# https://html.spec.whatwg.org/multipage/input.html#email-state-(type=email)
class EmailValidator(Validator):
    def __call__(self, value):
        return re.match(r"^[a-zA-Z0-9.!#$%&'*+\\/=?^_`{|}~-]+@[a-zA-Z0-9](?:[a-zA-Z0-9-]{0,61}[a-zA-Z0-9])?"
                        r"(?:\.[a-zA-Z0-9](?:[a-zA-Z0-9-]{0,61}[a-zA-Z0-9])?)*$", value)
```

Також схема зберігає номер поточного рядка:

```
113         @property
114         def row_index(self):
115             return self['row_index']
116
117         @row_index.setter
118         def row_index(self, new_index):
119             assert 'row_index' not in self or abs(new_index - self['row_index']) == 1
120             self['row_index'] = new_index
121
122             if mongo_client:
123                 self.mongo_collection.update_one(
124                     {'id': 'schema'},
125                     {'$set': {'row_index': self['row_index']}},
126                     upsert=True
127                 )
```

Додавання, видалення, масове додавання колонок у схему:

```python
    def add(self, column_id: str, validator_defs: List[Dict]):
        assert column_id not in self['columns']
        self['columns'][column_id] = {'validator_defs': validator_defs}

        if mongo_client:
            self.mongo_collection.update_one(
                {'id': 'schema'},
                {'$set': {f'columns.{column_id}': self['columns'][column_id]}},
                upsert=True
            )

    def pop(self, column_id):
        self['columns'].pop(column_id)

        if mongo_client:
            self.mongo_collection.update_one(
                {'id': 'schema'},
                {'$unset': {f'columns.{column_id}': 1}},
                upsert=True
            )

    def update(self, elems, **kwargs):
        super().update(elems)
        if mongo_client and elems:
            self.mongo_collection.update_one({'id': 'schema'}, {'$set': self}, upsert=True)
```

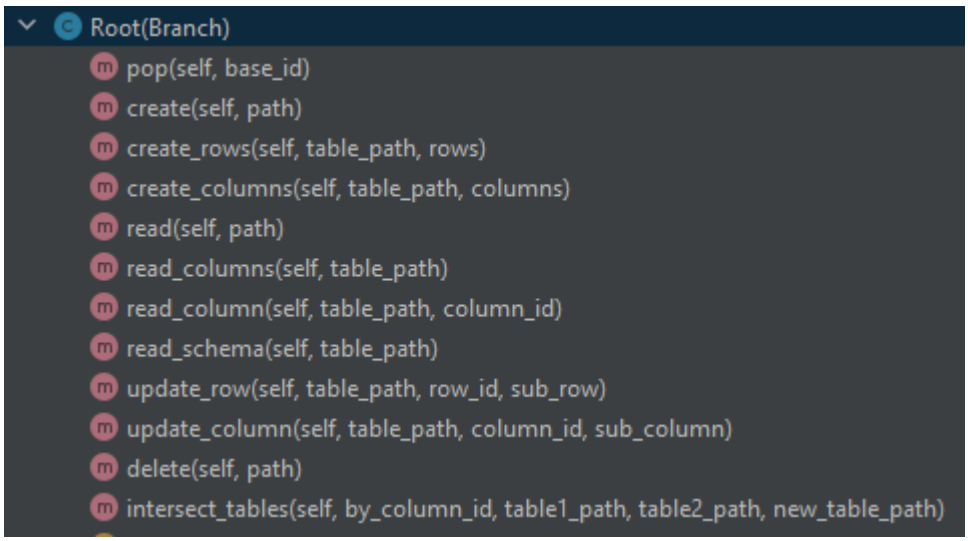
Приклад документів у mongo:

```
_id: ObjectId("5fc6e0981100cb951f9edd41")
id: 0
co1: 1
co2: 2
co3: 5
```

```
_id: ObjectId("5fc6e0981100cb951f9edd42")
id: 1
co1: 3
co2: 4
co3: 6
```

```
_id: ObjectId("5fc6e098cc58ff87dd01483c")
id: "schema"
columns: Object
  co1: Object
    validator_defs: Array
      0: Object
          name: "TypeValidator"
        params: Object
            type_descr: "int"
  co2: Object
  co3: Object
row_index: 1
```

Root є керівним класом, у ньому реалізовані всі методи до яких звертаються gRPC/Flask REST/GraphQL:



Наприклад REST звертається так:

```
3   import api
4
5   # from pymongo import MongoClient
6   # api.mongo_client = MongoClient()
7
8   tree = api.create_tree()
9
10
11  def create(**kwargs):
12      return jsonify(success=tree.create(path=list(kwargs.values())))
13
14
15  def create_rows(**kwargs):
16      return jsonify(success=tree.create_rows(table_path=list(kwargs.values()), rows=request.get_json()))
17
18
19  def create_columns(**kwargs):
20      return jsonify(success=tree.create_columns(table_path=list(kwargs.values()), columns=request.get_json()))
21
22
23  def read(**kwargs):
24      return jsonify(tree.read(list(kwargs.values())))
25
26
27  def read_columns(**kwargs):
28      return jsonify(tree.read_columns(table_path=list(kwargs.values())))
29
30
31  def read_column(**kwargs):
32      column_id = kwargs.pop('column_id')
33      return jsonify(tree.read_column(table_path=list(kwargs.values()), column_id=column_id))
```

api.create_tree з 8-го рядка відповідно до встановленого чи відсутнього клієнта завантажує дерево з mongo чи заповнює встановленими вручну значеннями:

```python
def create_tree() -> Root:
    if mongo_client:
        return Root(children=[
            Base(
                base_id,
                children=[Table(table_id, mongo_collection=mongo_client[base_id][table_id])
                          for table_id in mongo_client[base_id].list_collection_names()],
                mongo_base=mongo_client[base_id],
            ) for base_id in set(mongo_client.list_database_names()) - {'admin', 'config', 'local'}
        ])
    else:
        return Root(children=[
            Base('db1', children=[
                Table(
                    'tb1',
                    raw_schema={'column_index': 1, 'row_index': 2,
                                'columns': {'co1': {'validator_defs': []},
                                            'co2': {'validator_defs': []}}},
                    init_rows_unsafe=[(1, 4), (2, 5), (3, 6)]
                ),
                Table(
                    'tb2',
                    raw_schema={'column_index': 1, 'row_index': 2,
                                'columns': {'co1': {'validator_defs': []},
                                            'co2': {'validator_defs': []}}},
                    init_rows_unsafe=[(1, 4), (2, 5), (3, 6)]
                )
            ]),
            Base('db2', children=[
                Table('tb3',
                      raw_schema={'column_index': 0, 'row_index': -1,
                                  'columns': {'co3': {'validator_defs': []}}})
            ])
        ])
```

Роутинг Flask:

```python
app = Flask(__name__)

app.add_url_rule(rule='/tree/', view_func=read, methods=['GET'])
app.add_url_rule(rule='/tree/<base_id>/', view_func=create, methods=['POST'])
app.add_url_rule(rule='/tree/<base_id>/', view_func=read, methods=['GET'])
app.add_url_rule(rule='/tree/<base_id>/', view_func=delete, methods=['DELETE'])
app.add_url_rule(rule='/tree/<base_id>/<table_id>/', view_func=create, methods=['POST'])
app.add_url_rule(rule='/tree/<base_id>/<table_id>/', view_func=delete, methods=['DELETE'])
app.add_url_rule(rule='/tree/<base_id>/<table_id>/rows/', view_func=create_rows, methods=['POST'])
app.add_url_rule(rule='/tree/<base_id>/<table_id>/rows/', view_func=read, methods=['GET'])
app.add_url_rule(rule='/tree/<base_id>/<table_id>/rows/<int:row_id>/', view_func=read, methods=['GET'])
app.add_url_rule(rule='/tree/<base_id>/<table_id>/rows/<int:row_id>/', view_func=update_row, methods=['PUT'])
app.add_url_rule(rule='/tree/<base_id>/<table_id>/rows/<int:row_id>/', view_func=delete, methods=['DELETE'])
app.add_url_rule(rule='/tree/<base_id>/<table_id>/rows/<int:row_id>/<column_id>/', view_func=read_value, methods=['GET'])
app.add_url_rule(rule='/tree/<base_id>/<table_id>/columns/', view_func=create_columns, methods=['POST'])
app.add_url_rule(rule='/tree/<base_id>/<table_id>/columns/', view_func=read_columns, methods=['GET'])
app.add_url_rule(rule='/tree/<base_id>/<table_id>/columns/<column_id>/', view_func=read_column, methods=['GET'])
app.add_url_rule(rule='/tree/<base_id>/<table_id>/columns/<column_id>/', view_func=update_column, methods=['PUT'])
app.add_url_rule(rule='/tree/<base_id>/<table_id>/columns/<column_id>/', view_func=delete, methods=['DELETE'])
app.add_url_rule(rule='/tree/<base_id>/<table_id>/columns/<column_id>/<int:row_id>/', view_func=read_value, methods=['GET'])
app.add_url_rule(rule='/tree/<base_id>/<table_id>/schema/', view_func=read_schema, methods=['GET'])
app.add_url_rule(rule='/tree/intersect_tables/', view_func=intersect_tables, methods=['POST'])
```

Опис сервісу у gRPC:

```proto
service Tree {
  // Create
  rpc CreateBase(PathRequest) returns (SuccessResponse) {};
  rpc CreateTable(PathRequest) returns (SuccessResponse) {};
  rpc CreateRows(CreateRowsRequest) returns (SuccessResponse) {};
  rpc CreateColumns(CreateColumnsRequest) returns (SuccessResponse) {};

  // Read
  rpc ReadTree(PathRequest) returns (google.protobuf.Struct) {};
  rpc ReadBase(PathRequest) returns (google.protobuf.Struct) {};
  rpc ReadTable(PathRequest) returns (google.protobuf.Struct) {};
  rpc ReadRows(PathRequest) returns (google.protobuf.Struct) {};
  rpc ReadColumns(PathRequest) returns (google.protobuf.Struct) {};
  rpc ReadRow(PathRequest) returns (google.protobuf.ListValue) {};
  rpc ReadColumn(PathRequest) returns (google.protobuf.ListValue) {};
  rpc ReadValue(PathRequest) returns (google.protobuf.ListValue) {};
  rpc ReadSchema(PathRequest) returns (google.protobuf.Struct) {};

  // Update
  rpc UpdateRow(UpdateRowRequest) returns (SuccessResponse) {};
  rpc UpdateColumn(UpdateColumnRequest) returns (SuccessResponse) {};

  // Delete
  rpc DeleteBase(PathRequest) returns (SuccessResponse) {};
  rpc DeleteTable(PathRequest) returns (SuccessResponse) {};
  rpc DeleteRow(PathRequest) returns (SuccessResponse) {};
  rpc DeleteColumn(PathRequest) returns (SuccessResponse) {};

  rpc IntersectTables(IntersectTablesRequest) returns (SuccessResponse) {};
}
```

У gRPC повідомлення (message), що відповідають JSON об’єктам (Struct, ListValue, Value), реалізовані відповідно до JSON стандарту rfc7159, тому цілі числа, під час заповнення цих повідомлень перетворюються на double. Тому перед надсиланням дані конвертуються у такі, що зберігають тип значення поряд із самим значенням.

```python
def to_type_pair(value):
    return [type(value).__name__, value]


def to_type_pairs(values):
    return list(map(to_type_pair, values))


def from_type_pair(pair):
    type_, value = pair
    # noinspection PyCallingNonCallable
    return locate(type_)(value)


def from_type_pairs(pairs):
    return list(map(from_type_pair, pairs))


def to_jsonable_rows(rows):
    return {str(row_id): to_type_pairs(row) for row_id, row in rows.items()}


def from_jsonable_rows(rows):
    return {int(row_id): from_type_pairs(pairs) for row_id, pairs in rows.items()}
```

...

Нижче наведений повний шлях процедури gRPC від її опису до реалізації на сервері, клієнті:

tree.proto:

```
message UpdateColumnRequest {
  repeated string table_path = 1;
  string column_id = 2;
  google.protobuf.Struct sub_column = 3;
}
```

```
message SuccessResponse {
  bool success = 1;
}
```

```
service Tree {
  rpc UpdateColumn(UpdateColumnRequest) returns (SuccessResponse) {};
```

server.py:

```
from concurrent import futures

import grpc

import api
import grpc_.messages.tree_pb2_grpc as tree_service


def serve():
    from pymongo import MongoClient
    api.mongo_client = MongoClient()
    from grpc_.services.tree import TreeServicer

    server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))
    # noinspection PyTypeChecker
    tree_service.add_TreeServicer_to_server(TreeServicer(api), server)
    server.add_insecure_port('[::]:50051')
    server.start()
    server.wait_for_termination()


if __name__ == '__main__':
    serve()
```

grpc_.services.tree.py:

```
class TreeServicer(tree_service.TreeServicer):
    def __init__(self, api: ModuleType):
        self.tree = api.create_tree()

    def UpdateColumn(self, request, context):
        resp = tree_messages.SuccessResponse()
        sub_column = from_jsonable_column(MessageToDict(request.sub_column))
        resp.success = self.tree.update_column(list(request.table_path), request.column_id, sub_column)
        return resp

    def create(self, path):
        resp = tree_messages.SuccessResponse()
        resp.success = self.tree.create(MessageToDict(path))
        return resp
```

client.py:

```
def update_column(base_id: str, table_id: str, column_id: str, sub_column: Dict):
    request = tree_messages.UpdateColumnRequest()
    request.table_path.extend([base_id, table_id])
    request.column_id = column_id
    request.sub_column.update(to_jsonable_column(sub_column))
    response = client.UpdateColumn(request)
    return response
```

Нижче наведений приклад того як працює GraphQL сервер:

```
class SuccessResponse(graphene.ObjectType):
    success = graphene.Boolean()
```

```
class Query(graphene.ObjectType):
    create_node = graphene.Field(SuccessResponse, path=graphene.List(graphene.String, required=True))
```

```
@staticmethod
def resolve_create_node(root, info, path):
    return SuccessResponse(tree.create(path))
```

```
schema = graphene.Schema(query=Query)
app = Flask(__name__)
app.add_url_rule(
    '/graphql',
    view_func=GraphQLView.as_view(
        'graphql',
        schema=schema,
        graphiql=True
    )
)
```

Тести REST:

```python
class TestRest(TestCase):
    @classmethod
    def setUpClass(cls):
        requests.delete('http://localhost:5000/tree/db_test/')
        requests.delete('http://localhost:5000/tree/db_test1/')

    def test1_create(self):
        assert requests.post('http://localhost:5000/tree/db_test/').json()['success']
        assert requests.get('http://localhost:5000/tree/db_test/').ok

        assert requests.post('http://localhost:5000/tree/db_test/tb_test/').json()['success']
        assert requests.get('http://localhost:5000/tree/db_test/tb_test/rows/').ok

        type_validator_def = {'name': 'TypeValidator', 'params': {'type_descr': 'int'}}
        assert requests.post('http://localhost:5000/tree/db_test/tb_test/columns/',
                             json={'co1': {'validator_defs': [type_validator_def]}, 'co2': {}}).json()['success']
        r0 = requests.get('http://localhost:5000/tree/db_test/tb_test/schema/')
        assert r0.json() == {'columns': {'co1': {'validator_defs': [type_validator_def]},
                                         'co2': {'validator_defs': []}}, 'row_index': -1}, (r0.ok, r0.json())

        assert requests.post('http://localhost:5000/tree/db_test/tb_test/rows/',
                             json=[{'co1': 1, 'co2': 2}, {'co1': 3, 'co2': 4}]).json()['success']
        r1 = requests.get('http://localhost:5000/tree/db_test/tb_test/rows/')
        assert list(r1.json().values())[-1] == [3, 4], (r1.ok, r1.json())

        assert requests.post('http://localhost:5000/tree/db_test/tb_test/columns/',
                             json={'co3': {'values': [5, 6]}}).json()['success']
        r2 = requests.get('http://localhost:5000/tree/db_test/tb_test/rows/')
        assert list(r2.json().values()) == [[1, 2, 5], [3, 4, 6]], (r2.ok, r2.json())
        r3 = requests.get('http://localhost:5000/tree/db_test/tb_test/columns/')
        assert list(r3.json().values()) == [[1, 3], [2, 4], [5, 6]], (r3.ok, r3.json())

        assert requests.post('http://localhost:5000/tree/db_test1/').json()['success']
        assert requests.post('http://localhost:5000/tree/db_test1/tb_test1/').json()['success']
        assert requests.post('http://localhost:5000/tree/db_test1/tb_test1/columns/',
                             json={'co1': {}, 'co4': {}}).json()['success']
        assert requests.post('http://localhost:5000/tree/db_test1/tb_test1/rows/',
                             json=[{'co1': 3, 'co4': 4}, {'co1': 5, 'co4': 6}]).json()['success']
        assert requests.post('http://localhost:5000/tree/intersect_tables/',
                             json={'by_column_id': 'co1',
                                   'table1_path': ['db_test', 'tb_test'],
                                   'table2_path': ['db_test1', 'tb_test1'],
                                   'new_table_path': ['db_test1', 'tb_test2']}).json()['success']
        r4 = requests.get('http://localhost:5000/tree/db_test1/tb_test2/columns/')
        assert r4.json() == {'co1': [3], 'co2': [4], 'co3': [6], 'co4': [4]}, (r4.ok, r4.json())

        assert not requests.post('http://localhost:5000/tree/db_test/tb_test/columns/',
                                 json={'co4': {'validator_defs': [{'name': 'EmailValidator', 'params': {}}],
                                               'values': ['vinnik.dmitry07 gmail.com'] * 2}}).json()['success']
```

```
    def test2_read(self):
        assert requests.get('http://localhost:5000/tree/').ok
        assert requests.get('http://localhost:5000/tree/db_test/').ok
        assert requests.get('http://localhost:5000/tree/db_test/tb_test/rows/').ok
        assert requests.get('http://localhost:5000/tree/db_test/tb_test/rows/0/').ok
        assert requests.get('http://localhost:5000/tree/db_test/tb_test/rows/0/co1/').ok
        assert requests.get('http://localhost:5000/tree/db_test/tb_test/columns/').ok
        assert requests.get('http://localhost:5000/tree/db_test/tb_test/columns/co1/').ok
        assert requests.get('http://localhost:5000/tree/db_test/tb_test/columns/co1/0/').ok

    def test3_update(self):
        assert requests.put('http://localhost:5000/tree/db_test/tb_test/rows/0/', json={'co1': 7}).json()['success']
        assert requests.get('http://localhost:5000/tree/db_test/tb_test/rows/0/co1/').json() == 7
        assert requests.put('http://localhost:5000/tree/db_test/tb_test/columns/co1/', json={0: 8}).json()['success']
        assert requests.get('http://localhost:5000/tree/db_test/tb_test/columns/co1/0/').json() == 8

    def test4_delete(self):
        assert requests.delete('http://localhost:5000/tree/db_test/tb_test/columns/co1/').json()['success']
        assert requests.delete('http://localhost:5000/tree/db_test/tb_test/rows/0/').json()['success']
        assert requests.delete('http://localhost:5000/tree/db_test/tb_test/').json()['success']
        assert requests.delete('http://localhost:5000/tree/db_test/').json()['success']
```

Такі ж тести gRPC:

```
class TestGRPC(TestCase):
    @classmethod
    def setUpClass(cls):
        client.delete_base('db_test')
        client.delete_base('db_test1')

    def test1_post(self):
        assert client.create_base('db_test').success
        client.read_base('db_test')

        assert client.create_table('db_test', 'tb_test').success
        client.read_rows('db_test', 'tb_test')

        type_validator_def = {'name': 'TypeValidator', 'params': {'type_descr': 'int'}}
        assert client.create_columns('db_test', 'tb_test',
                                     {'co1': {'validator_defs': [type_validator_def]}, 'co2': {}}).success
        r0 = client.read_schema('db_test', 'tb_test')
        assert r0 == {'columns': {'co1': {'validator_defs': [type_validator_def]},
                                  'co2': {'validator_defs': []}}, 'row_index': -1}, r0

        assert client.create_rows('db_test', 'tb_test', [{'co1': 1, 'co2': 2}, {'co1': 3, 'co2': 4}]).success
        r1 = client.read_rows('db_test', 'tb_test')
        assert list(r1.values())[-1] == [3, 4], r1

        assert client.create_columns('db_test', 'tb_test', {'co3': {'values': [5, 6]}}).success
        r2 = client.read_rows('db_test', 'tb_test')
        assert list(r2.values()) == [[1, 2, 5], [3, 4, 6]], r2
        r3 = client.read_columns('db_test', 'tb_test')
        assert list(r3.values()) == [[1, 3], [2, 4], [5, 6]], r3

        assert client.create_base('db_test1').success
        assert client.create_table('db_test1', 'tb_test1').success
        assert client.create_columns('db_test1', 'tb_test1', {'co1': {}, 'co4': {}}).success
        assert client.create_rows('db_test1', 'tb_test1', [{'co1': 3, 'co4': 4}, {'co1': 5, 'co4': 6}]).success
        assert client.intersect_tables(by_column_id='co1',
                                       table1_path=['db_test', 'tb_test'],
                                       table2_path=['db_test1', 'tb_test1'],
                                       new_table_path=['db_test1', 'tb_test2']).success
        r4 = client.read_columns('db_test1', 'tb_test2')
        assert r4 == {'co2': [4], 'co3': [6], 'co1': [3], 'co4': [4]}, r4

        assert not client.create_columns('db_test', 'tb_test',
                                         {'co4': {'validator_defs': [{'name': 'EmailValidator', 'params': {}}],
                                                  'values': ['vinnik.dmitry07 gmail.com'] * 2}}).success
```

```python
    def test2_get(self):
        client.read_tree()
        client.read_base('db_test')
        client.read_rows('db_test', 'tb_test')
        client.read_row('db_test', 'tb_test', row_id=0)
        client.read_columns('db_test', 'tb_test')
        client.read_column('db_test', 'tb_test', 'co1')
        client.read_value('db_test', 'tb_test', row_id=0, column_id='co1')

    def test3_put(self):
        assert client.update_row('db_test', 'tb_test', row_id=0, sub_row={'co1': 7}).success
        assert client.read_value('db_test', 'tb_test', row_id=0, column_id='co1') == 7
        assert client.update_column('db_test', 'tb_test', column_id='co1', sub_column={0: 8}).success
        assert client.read_value('db_test', 'tb_test', row_id=0, column_id='co1') == 8

    def test4_delete(self):
        assert client.delete_column('db_test', 'tb_test', 'co1').success
        assert client.delete_row('db_test', 'tb_test', row_id=0).success
        assert client.delete_table('db_test', 'tb_test').success
        assert client.delete_base('db_test').success
```