****Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Реферат**

**Тема: Написание клиент-серверного приложения с использованием Django и Django rest.**

Группа: М80 – 107Б-18

Студент: Син Денис Дмитриевич

Преподаватель: Ридли Александра Николаевна

Оценка: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Дата: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Москва, 2019.

**Содержание.**

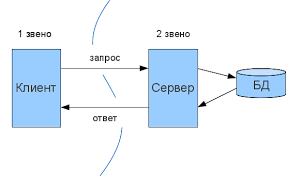
1. Введение
2. Структура приложения
3. RESTful приложения
4. Реализация API
5. Реализация клиентской части
6. Фрагменты кода
7. Заключение

**Введение**.

В современном мире каждый человек пользуется различными приложениями на разных устройствах. Данные пользователей в основном хранятся не локально на их устройствах, а на сервере данного приложения и разработка данных приложений делится на два этапа разработка серверной части и разработка клиентской части. И необходимо продумать взаимодействие между этими частями. В данном реферате я реализовал клиент-серверное приложение.

**Структура приложения.**

Как уже было написано выше разработка клиент-серверных приложений делится на два этапа на разработку клиентской части и разработку серверной части.



**Разработка клиентской части**.

Приложения в основном бывают трех типов:

1. Web приложения
2. Мобильные приложения
3. Приложения на десктоп

Для каждого типа существуют разные языки программирования и технологии. Я реализовал web приложение. Для клиентской части такого типа существует единственный стек технологий html, css, javascript. Язык html предназначен для разметки приложения, css – для стилей, язык javascript используется для интерактивности и взаимодействия с сервером.

Приложение можно написать на чистом javascript, но сейчас для написания клиентской части используют фреймворки и библиотеки. В данной работе я пользовался библиотекой React.js.

**Основные концепции react приложений**.

1. Декларативный подход к описанию интерфейсов
2. Функциональный подход к разработке
3. Компонентный подход
4. Компоненты могут содержать состояния

**Разработка серверной части**.

Для серверной части в отличии от клиентской существуют много технологий и языков программирования: php, ruby (ruby on rails), python (django, flask), java (spring), javascript (express.js), C++.

Для своего приложения я выбрал язык python, фреймворки django и django rest(для написания api).

Также в приложении я использовал базу данных sqlite3. При написании приложения мне не понадобилось работать с sql запросами, так как в Django реализована технология ORM(Object-Relational Mapping), которая связывает базу данных с концепцией ООП.

**RESTful приложения.**

REST (Representational State Transfer) – это архитектурный подход, позволяющий сделать сетевое взаимодействие удобнее, прозрачнее и стандартизированнее.

Основные принципы построения rest API:

1. Структура запросов и ответов MEHOD URI
2. Кодировка запросов и ответов UTF-8
3. Структура URI /*:entity*[/*:id*][/?*:params*]
4. Методы http
   1. GET /:entity/:id — **getById**
   2. GET /:entity[?param1=...&param2=...] — списочный get
   3. HEAD /:entity[/:id] — запрос заголовкок
   4. POST /:entity — создаёт новый объект типа :entity
   5. PUT /:entity/:id — изменяет объект целиком
   6. PATCH /:entity/:id — изменяет отдельные поля объекта
   7. DELETE /:entity/:id — удаляет объект, если он существует.
   8. OPTIONS /:entity[/:id] – получает список методов доступных по данному URI

**Реализация API.**

Для реализации API необходимо:

1. Определить сущности приложения.
2. Реализовать сериализацию данных.
3. Реализовать роутинг запросов.
4. Реализовать представление данных.

**Таблицы в базе данных.**



В базе данных у нас будут две таблицы User и Task и между ними существует отношение many to many.

**Сереализация данных.**

Для сереализации данных необходимо каждую запись таблицы переводить в файл формата json и наоборот при получении данных с клиента переводить json в данные и изменять их в таблице.

**Роутинг запросов**.

Роутинг запросов представляет из себя список URI. И при запросе на данные ресурсы будут вызываться соответствующие им функции, отвечающие за представление данных.

**Представление данных**.

За представление данных отвечают специальные функции, которые проверяют методы, сереализуют данные и затем изменяют данные в базе данных.

**Реализация клиентской части**.

Для реализации клиентской части необходимо:

1. Написать компоненты
2. Реализовать получение данных через API.
3. Реализовать роутинг на клиенте.

**Компоненты**.

При написании компонентов я использовал синтаксис jsx, который. позволяет смешивать javascript и html вместе. Основная задача в этом пункте состоит в написании отдельного компонента со своими состояниями и обработки некоторых событий.

**Получение данных**.

Для получения данных я использовал метод fetch, который позволяет делать асинхронные http запросы и дает возможность работать с промисами в javascript. При получении данных необходимо отправить запрос по необходимому uri, указать метод, и если при запросе необходима авторизация передовать token в заголовке запроса.

**Роутинг на клиенте**.

Чтобы реализовать роутинг нужно подключить библиотеку react-router, которая позволяет при переходе на ссылки отображать нужные компоненты или несколько компонентов.

**Фрагменты кода**.

**Пример реализации компонента Todo**

import React, { Component } from 'react'

import './todo.css';

import TodoField from '../todo-field';

export default class Todo extends Component {

state = {

inputTask: '',

tasks: []

};

getTasks = () => {

fetch('http://localhost:8000/api/task/', {

method: 'GET',

headers: {

"Content-type": "application/json",

"Authorization": `Token ${this.props.token}`

}

})

.then((res) => {

return res.json();

})

.then((body) => {

this.setState(() => {

return {

tasks: [ ...body ]

};

});

})

.catch((err) => {

console.log(err.message);

});

};

createTask = () => {

fetch('http://localhost:8000/api/task/', {

method: 'POST',

headers: {

"Content-type": "application/json",

"Authorization": `Token ${this.props.token}`

},

body: JSON.stringify({

name: this.state.inputTask,

description: '',

difficult: 2

})

})

.then((res) => {

return res.json();

})

.then((body) => {

console.log(body);

this.setState(({ tasks, inputTask }) => {

return {

inputTask: '',

tasks: [...tasks, body]

};

});

})

.catch((err) => {

console.log(2, err.message);

});

};

deleteTask = (id) => {

fetch('http://localhost:8000/api/task/', {

method: 'DELETE',

headers: {

"Content-type": "application/json",

"Authorization": `Token ${this.props.token}`

},

body: JSON.stringify({

id: id

})

})

.then((res) => {

return res.json();

})

.then((body) => {

console.log(body);

this.setState(({ tasks }) => {

let idDel = tasks.findIndex((val) => {

return val.id === id;

});

const newTasks = [...tasks.slice(0, idDel), ...tasks.slice(idDel + 1)]

return {

tasks: newTasks

};

});

})

.catch((err) => {

console.log(err.message);

});

}

onChangeTask = (event) => {

this.setState({ inputTask: event.target.value });

console.log(this.state.inputTask);

};

componentWillMount() {

this.getTasks();

};

render() {

return (

<div className="todo">

<h3 className="todo-title">Todo App</h3>

<ul className="list-group todo-list">

{this.state.tasks.map(({ name, id }) => {return <TodoField

key={ id }

name={ name }

num={ id }

onDelete={ this.deleteTask }

/>

})}

</ul>

<div className="todo-field">

<label htmlFor="exampleInputTask"></label>

<input

type="text"

className="form-control"

id="todo-input"

placeholder="Enter task"

value={ this.state.inputTask }

onChange={ this.onChangeTask }

>

</input>

<button

type="button"

className="btn btn-outline-success"

onClick={ this.createTask }

>Add</button>

</div>

</div>

)

}

}

**Модели приложения**:

from django.db import models

from django.contrib.auth.models import AbstractUser

class User(AbstractUser):

level = models.FloatField(default=0, blank=True)

tasks = models.ManyToManyField('Task', related\_name='users')

class Task(models.Model):

STATUSES = (

(0, 'Not performed'),

(1, 'Done'),

(2, 'Expired')

)

name = models.CharField(max\_length=50)

description = models.CharField(max\_length=255, blank=True)

status = models.IntegerField(choices=STATUSES, default=0)

creation\_date = models.DateTimeField(auto\_now\_add=True)

expiration\_date = models.DateTimeField(blank=True, null=True)

difficult = models.FloatField(default=1)

deadline = models.DateTimeField(blank=True, null=True)

**Представления данных**:

from rest\_framework.authtoken.models import Token

from rest\_framework.response import Response

from rest\_framework.authtoken.views import ObtainAuthToken

from rest\_framework.generics import (

CreateAPIView,

ListAPIView,

DestroyAPIView,

# UpdateAPIView,

)

from rest\_framework.views import APIView

# from rest\_framework.status import (

# HTTP\_200\_OK,

# HTTP\_400\_BAD\_REQUEST

# )

from rest\_framework import (

permissions,

authentication

)

from .serializers import (

UserCreateSerializer,

UserListSerializer,

TaskCreateSerializer,

TaskListSerializer,

TaskDeleteSerializer

)

from .models import (

User,

Task

)

class UserCreateAPIView(CreateAPIView):

serializer\_class = UserCreateSerializer

queryset = User.objects.all()

class UserListAPIView(ListAPIView):

serializer\_class = UserListSerializer

queryset = User.objects.all()

class UserAuthToken(ObtainAuthToken):

def post(self, request, \*args, \*\*kwargs):

serializer = self.serializer\_class(

data=request.data,

context={'request': request}

)

serializer.is\_valid(raise\_exception=True)

user = serializer.validated\_data['user']

token, created = Token.objects.get\_or\_create(user=user)

return Response({

'token': token.key,

'user\_id': user.pk,

'email': user.email

})

class TaskAPIView(APIView):

authentication\_classes = (authentication.TokenAuthentication,)

permission\_classes = (permissions.IsAuthenticated,)

def get(self, request, \*args, \*\*kwargs):

user = User.objects.get(id=request.auth.user\_id)

tasks = user.tasks.all()

response = [TaskListSerializer(task).data for task in tasks]

return Response(response)

def post(self, request, \*args, \*\*kwargs):

user = User.objects.get(id=request.auth.user\_id)

serializer = TaskListSerializer(data=request.data)

serializer.is\_valid(raise\_exception=True)

name = serializer.validated\_data['name']

description = serializer.validated\_data['description']

difficult = serializer.validated\_data['difficult']

task\_obj = Task(

name=name,

description=description,

difficult=difficult

)

task\_obj.save()

user.tasks.add(task\_obj)

user.save()

return Response(TaskListSerializer(task\_obj).data)

def delete(self, request, \*args, \*\*kwargs):

serializer = TaskDeleteSerializer(data=request.data)

serializer.is\_valid(raise\_exception=True)

id = serializer.validated\_data['id']

print(id)

task = Task.objects.get(id=id)

task.delete()

return Response('{"Delete": "Done"}')

**Роутинг на сервере:**

from django.urls import path

from .views import (

UserCreateAPIView,

UserAuthToken,

UserListAPIView,

TaskAPIView

)

urlpatterns = [

path('register/', UserCreateAPIView.as\_view()),

path('login/', UserAuthToken.as\_view()),

path('user/', UserListAPIView.as\_view()),

path('task/', TaskAPIView.as\_view())

]

**Заключение**.

В данной работе я реализовал клиент-серверное приложение. Чтобы написать действительно качественное приложение необходимо заранее продумать архитектуру, как будет происходить хранение данных, как они будут изменяться и еще много других важных вещей. Больших проблем с реализацией серверной части не было, там мы храним данные, реализуем сереализацию данных и изменение данных, к тому же большой функционал дают фреймворки Django и Django rest, на клиенте же были некоторые сложности, связанные с хранением данных. Если мы хотим, чтобы один компонент обменивался данными с другим, то возникают сложности и из-за этого возникают проблемы с масштабированием приложения. Однако существует решение flux архитектура, которую использует facebook. И для react существует реализация данной архитектуры, которая называется redux.