Министерство образования и науки Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

УНИВЕРСИТЕТ ИТМО

Факультет инфокоммуникационных технологий

Образовательная программа 09.03.02

Направление подготовки (специальность) Мобильные сетевые технологии

О Т Ч Е Т

о курсовой работе

Тема задания: реализация клиентской части приложения посредством VueJs

Обучающийся Сулейманов Руслан Имранович, К33402

Руководитель: Добряков Д. И., преподаватель

Оценка за курсовую работу ____

Подписи членов комиссии:

_____ (Добряков Д. И.)
(подпись)

Дата ____

Санкт-Петербург
2021

ВВЕДЕНИЕ

Актуальность

Каждый день миллионы людей смотрят на прогноз погоды на своих гаджетах. И чаще всего, прогноз погоды ограничивается малым объемом информации, например только 1 страной, или даже только 1 городом, иногда хочется узнать погоду где нибудь в глуши, куда собираешься поехать. Для этого и создано данное приложение, можно узнать погоду в любом населенном пункте мира.

Цели и задачи

1. Определение средств разработки
2. Определение функциональных требований
3. Проектирование и реализация серверной части
4. Проектирование и реализация клиентской части

ГЛАВА 1. СРЕДСТВА РАЗРАБОТКИ И ФУНКЦИОНАЛЬНЫЕ ТРЕБОВАНИЯ

1 Средства разработки

Для реализации данного инструмента использовался такой стек технологий: python, Django, sqllite, vue. Выбор Django обусловлен тем, что данный фреймворк позволяет максимально быстро и просто написать серверную часть, даже такому новичку как я. Python использовался как самое простое решение, для реализации доп функций парсинга города, и максимально совместимого с Django, vue - основное функциональное требование.

2 Функциональные требования

Функциональные требования по этому проекту заключались в наличие 9 компонентов в

1) Разработка одностраничного веб-приложения (SPA) с использованием фреймворка Vue.JS
2) Использование миксинов (необязательно, но желательно)
3) Использование Vuex (необязательно, но желательно)
4) В проекте должно быть, как минимум, 10 страниц (обязательно)

3 Проектирование и реализация серверной части

Данный сервер написан на Django rest framework, что позволяет наиболее легким способом соединить бекенд с фронтом. В моей реализации есть два приложения, одно – все про юзера, другое – все про погоду, а точнее города необходимые для поиска данных.
Рассмотрим города:

```python
class CityList(models.Model):
    city_cod = models.IntegerField('код города')
    name = models.CharField(max_length=255)
    state = models.CharField(max_length=255)
    country = models.CharField(max_length=255)
    coord_lon = models.FloatField()
    coord_lat = models.FloatField()

    def __str__(self):
        return self.name
```

Модель города повторяет модель openweather api, откуда и был взят city.list.json
Это файл размером в 2000000 строк.

Для парса данного файла был написан такой код:

```python
import os


def main():
    os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'djangoProject.settings')
    import django
    django.setup()
    from city_app.models import CityList

    with open('city.list.min.json', 'r', encoding='utf-8') as f:
        data = f.read()
    cities = json.loads(data)
    objects_list = []
    for city in cities:
        objects_list.append(
            CityList(city_cod=city['id'], name=city['name'], state=city['state'], country=city['country'],
                     coord_lon=city['coord']['lon'], coord_lat=city['coord']['lat']))
    CityList.objects.bulk_create(objects_list)


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

Данный код парсит файл за 2секунды, с помощью проб и ошибок, было установлено, что функция bulk_create наиболее быстрая, по сравнению с аналогами.

Есть простенький сериалайзер:

```python
class CitySerializer(serializers.ModelSerializer):
    class Meta:
        model = CityList
        fields = "__all__"
```

И простенькая вьюшка на получение данных, с настроенной фильтрацией по городам и пагинацией по 20 элементов:

```python
from rest_framework.response import Response
from rest_framework.views import APIView
from city_app.models import CityList
from city_app.serializers import CitySerializer


class CityView(APIView):
    def get(self, request):
        name = self.request.query_params.get('name')
        if name:
            cities = CityList.objects.filter(name__istartswith=name)[:20]
        else:
            cities = CityList.objects.all()[:20]
        serializer = CitySerializer(cities, many=True)
        return Response(serializer.data)
```

Рассмотрим юзера:

```python
from city_app.models import CityList

class User(AbstractUser):
    first_name = models.CharField(max_length=30, null=False, blank=False)
    last_name = models.CharField(max_length=30, null=False, blank=False)
    birthday = models.DateField(null=True)
    cities = models.ManyToManyField(CityList, through='Membership')

    def __str__(self):
        return f'{self.username}'

class Membership(models.Model):
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    city = models.ForeignKey(CityList, on_delete=models.CASCADE)
    date_joined = models.DateTimeField(auto_now_add=True)

    def __str__(self):
        return f'{self.date_joined}'
```

Переделан абстрактный юзер и созданная связанная таблица, для получения городов юзера. С возможностью расширять добавление города к юзеру, например, если пользователь захочет сохранять заметку о городу. Пока что автоматически добавляется дата добавления города.

Реализовано три пути, получения данных о юзере, получение и создание города, и удаление и получение 1 города

```python
urlpatterns = [
    path('me/', UserViewAPI.as_view()),
    path('me/city/', UserCitytList.as_view({'post': 'create', 'get': 'list'})),
    path('me/city/<int:pk>/', UserCitytDetail.as_view({'delete': 'destroy', 'get': 'retrieve'}))
]
```

Реализовано три сериалайзера:

```python
class CitySerialzer(serializers.ModelSerializer):
    class Meta:
        model = CityList
        fields = "__all__"


class UserSerializer(serializers.ModelSerializer):
    cities = CitySerialzer(many=True)

    class Meta:
        model = User
        fields = [
            "id",
            "username",
            "first_name",
            "last_name",
            "birthday",
            "cities"
        ]


class CityUserSerializer(serializers.ModelSerializer):
    class Meta:
        model = Membership
        fields = ["id", "city", "user", "date_joined"]
        read_only_fields = ["user", "date_joined"]
```

Рассмотрим теперь вьюшки:
Первая реализовано посредством APIView – гет запрос на получение данных юзера.

```python
class UserViewAPI(APIView):
    def get(self, request):
        serializer = UserSerializer(request.user)
        return Response(serializer.data)
```

Вторая вьюшка – получение городов юзера, и создание новых, для чего использовался метод GenericViewSet, и готовые модели cteate и list

```python
class UserCitytList(CreateModelMixin, ListModelMixin, GenericViewSet):
    permission_classes = [IsAuthenticated]
    serializer_class = CityUserSerializer

    def get_queryset(self):
        return Membership.objects.filter(user=self.request.user)

    def perform_create(self, serializer):
        serializer.save(user=self.request.user)
```

Последняя – основная функция - удаление

```python
class UserCitytDetail(DestroyModelMixin, RetrieveModelMixin, GenericViewSet):
    queryset = Membership.objects.all()
    permission_classes = [IsAuthenticated]
    serializer_class = CityUserSerializer

    def perform_create(self, serializer):
        serializer.save(user=self.request.user)
```

Теперь рассмотрим основные настройки сервера:

```python
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',

    'rest_framework',
    'rest_framework_simplejwt',
    'djoser',
    'accounts',
    'city_app',
    'drf_yasg',
    'corsheaders'
]
```

Подключены корсы для возможности получения данных с урла клиента:

```
CORS_ALLOWED_ORIGINS = [
    "http://192.168.1.9:8080",
    "http://localhost:8080"
]
```

Использован djoser для регистрации юзера и simple jwt, для авторизации по jwt токенам.
URL:

```
urlpatterns = [
    path('admin/', admin.site.urls),
    # path to our account's app endpoints
    path('auth/', include('djoser.urls')),
    path('auth/', include('djoser.urls.jwt')),
    path('token/', TokenObtainPairView.as_view(), name='token_obtain_pair'),
    path('token/refresh', TokenRefreshView.as_view(), name='token_refresh'),
    path('api/', include("accounts.urls")),
    path('api/', include("city_app.urls")),
    path('doc/swagger/', schema_view.with_ui('swagger', cache_timeout=0), name='schema-swagger-ui'),
    path('doc/redoc', schema_view.with_ui('redoc', cache_timeout=0), name='schema-redoc')
]
```

Подключен swagger и redoc ждя документации

4 Проектирование и реализация клиентской части

Используемый стек технологий: vuex, router-vue, vue

Реализовано 5 страниц, страница логина, регистрации, главная и профиль, и example – страница показывающая выбранную вами тему (создано для расширения в сторонку показывания погоды и темы по гео-позиции юзера). Настроен роутер по всем этим путям:

Также настроена аутентификация юзера, при входе в систему записывается токен в локал стор, и если там есть данные о токене, то пользователь считается авторизированным, иначе, нет.

```
const router = new VueRouter( options: {
    mode: 'history',
    base: process.env.BASE_URL,
    routes
})
router.beforeEach( guard: (to : Route , from : Route , next : NavigationGuardNext<Vue> ) => {
    let isAuthenticated = localStorage.getItem( key: 'token');
    if (to.name !== 'SignIn'&& to.name !== 'SignUp' && !isAuthenticated) next( to: { name: 'SignIn' })
    else next()
})
export default router
```

Пути:

```
const routes = [
    {
        path: '/',
        name: 'HomePage',
        component: HomePage
    },
    {
        path: '/example',
        name: 'Example',
        component: ThemePage
    },
    {
        path: '/profile',
        name: 'Profile',
        component: ProfilePage
    },
    {
        path: '/signin',
        name: 'SignIn',
        component: SignInPage
    },
    {
        path: '/signup',
        name: 'SignUp',
        component: SignUpPage
    },
]
```

Реализован метод, при котором обновление страницы сохраняет даныне юзера, метод написано в корневом файле app.vue, где проверяется наличие токена:

```
mounted: () => {
    localStorage.theme?
            store.commit( type: 'changeTheme', payload: {theme: localStorage.theme}):null
    localStorage.token?
            store.commit( type: 'addToken', localStorage.token):null
    localStorage.token?
            store.dispatch( type: 'addUser'):null
}
```

Если токен есть, мы записываем токен, и вызываем action getUser,

```
addUser ({ commit }) {
    /**
     * GET user,
     * @constructor
     * @param none
     * @return {object} data - обьект, с данными юзера.
     */
    fetch( input: `${URL}api/me/`, init: {
        headers: {
            'Content-Type': 'application/json',
            'Authorization': `Bearer ${JSON.parse(localStorage.getItem( key: 'token'))?.access}`
        },
        method: 'GET',
    })
        .then(checkResponse)
        .then(data => {
            commit('addUser', data);
            this.dispatch('addMembership');

            commit('clearWeather');
            data.cities.map((element :T )=> {
                this.dispatch('addWeather', {"id": element.id, "city_cod": element.city_cod})
            })
        })
        .catch(err => console.log(err))
},
```

При успешном выполнении которого, получаются данные юзера, и перезаписывается стор с погодой.

addWeather – метод обращающийся к открытой апи для получения данных

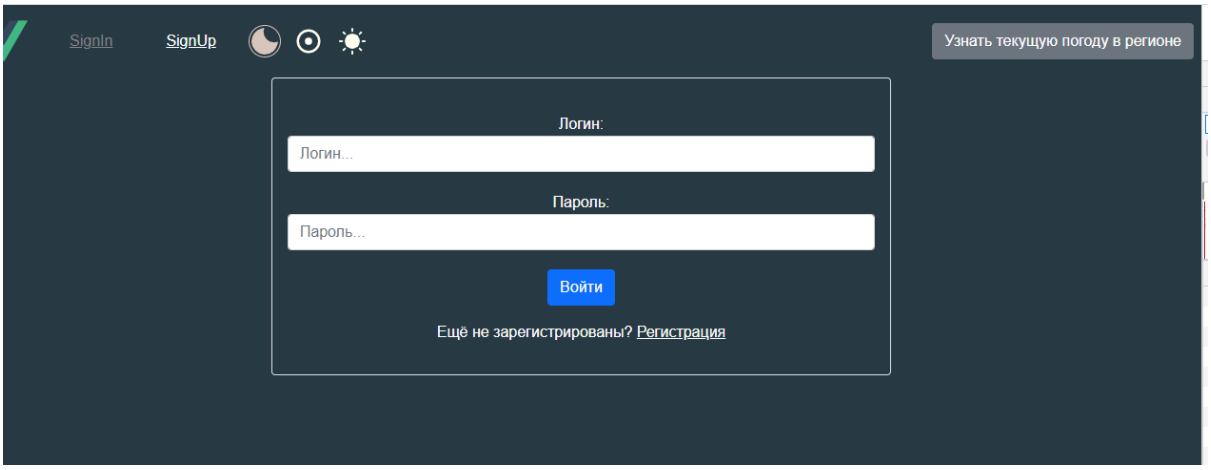
```
addWeather({ commit }, payload :{city_cod: string} ) {
    fetch( input: `https://api.openweathermap.org/data/2.5/weather?id=${payload.city_cod}&appid=${API}&units=metric`, init: {
        method: 'GET',
    })
        .then(checkResponse)
        .then(response => {
            response['city_id'] = payload.id;
            commit('addWeather', response)
        })
        .catch(err => console.log(err))
},
/**
```

Рассмторим детальнее store:

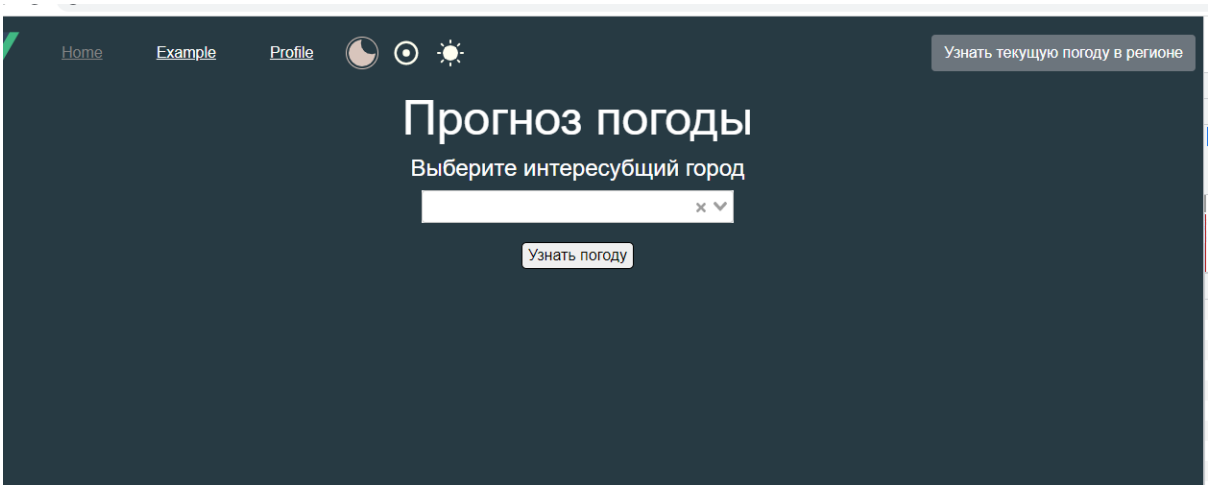
```
const store = new Vuex.Store( options: {
    modules: {
        theme: moduleTheme,
        city: moduleCity,
        metcast : moduleMetcast,
        user: moduleUser
    }
})
export default store;
```

Он состоит из 3 модулей, отвечающих за тему приложений, города, погоду, юзера. В каждом написан set и get функции а также actions для обращения к апи, или более сложной логике чем set.

Рассмотрим страницы:
Страница логина, аналогично странице регистрации, реализовано посредством v-model для хранения данных формы, отправка формы вызывает метод получения токена и запись его в локал стор. Неавторизированный пользователь может сменить тему, и узнать погоду в своем регионе.



При входе в систему открывается главная странице:



Она состоит из 2 компонент, добавление нового города и текущая погода:

```
<template>
    <div class="home">
        <h1>Прогноз погоды</h1>
        <AddNewCity />
        <MetcastCurrent />
    </div>
</template>
```

Так как еще, ничего не выбрано – вторая компонента ничего не рендерит. Рассмотрим метод добавление города:

```
<v-select
        @search="onSearch"
        @input="setSelected"

        :options="cities"
        :value="city"
        label="name"
        placeholder="Город"></v-select>
<button @click="someClick">Узнать погоду</button>
```

Для реализации данного поля используется дополнительная компонента v-select которая дает нам нужный функционал: есть два метода,

```
onSearch(search) {
    if(search.length) {
        getCityList(search)
    }
},
```

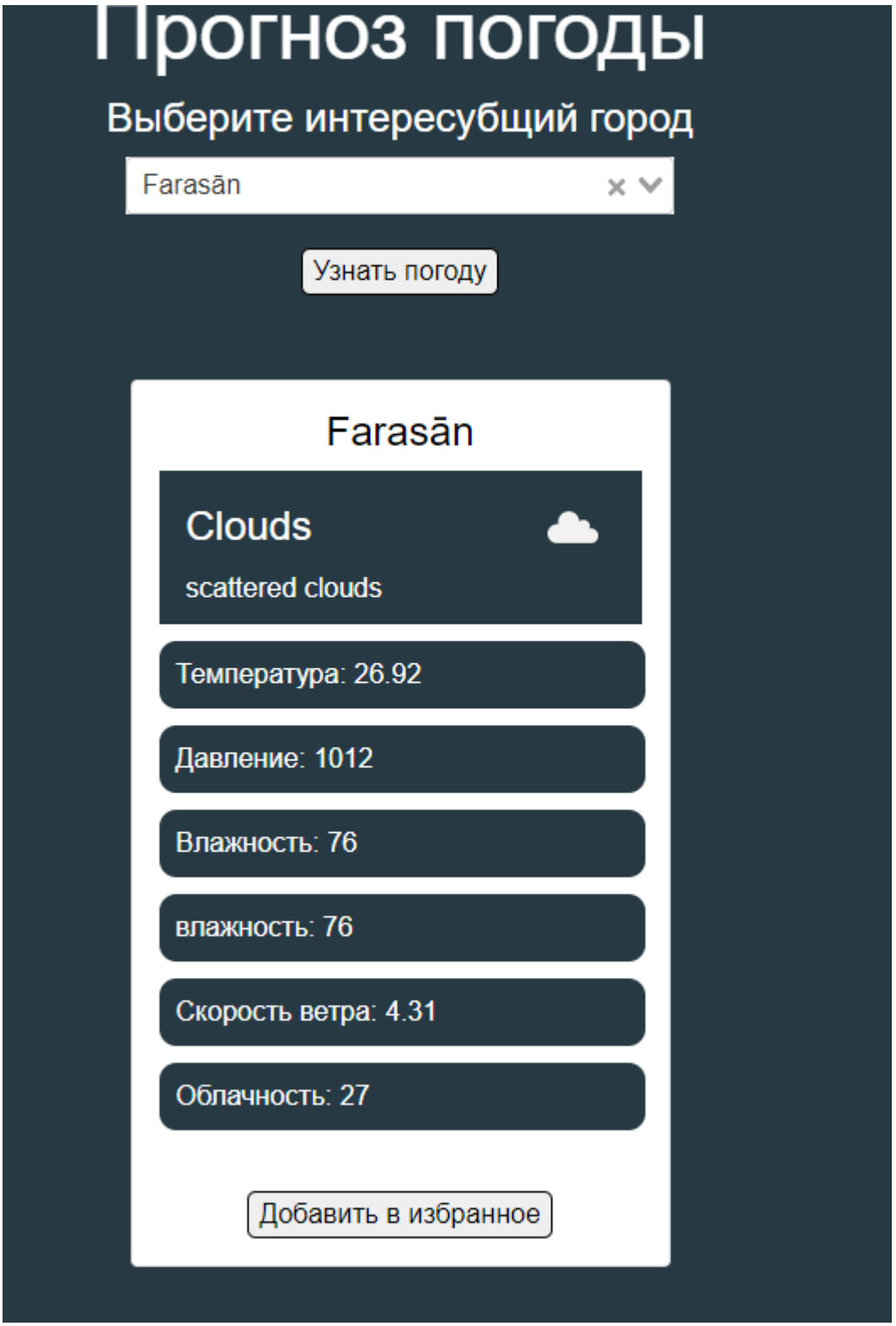
```
setSelected (val) {
    this.$store.commit( type: 'addCity', val)
}
```

Первый – запрос к бд,

```
export const getCityList = async (text) => {
    fetch( input: `${URL}api/city/?name=${text}`, init: {
        headers: {
            'Content-Type': 'application/json',
        },
        method: 'GET',
    })
        .then(checkResponse)
        .then(data => {
            store.commit( type: 'addCityList', data)
        })
        .catch(err => console.log(err))
}
```

Второе – добавление текущего города в стор.

При клике на кнопку, по значению выбранного города отрпавляется запрос к апи погоды, и как итог:

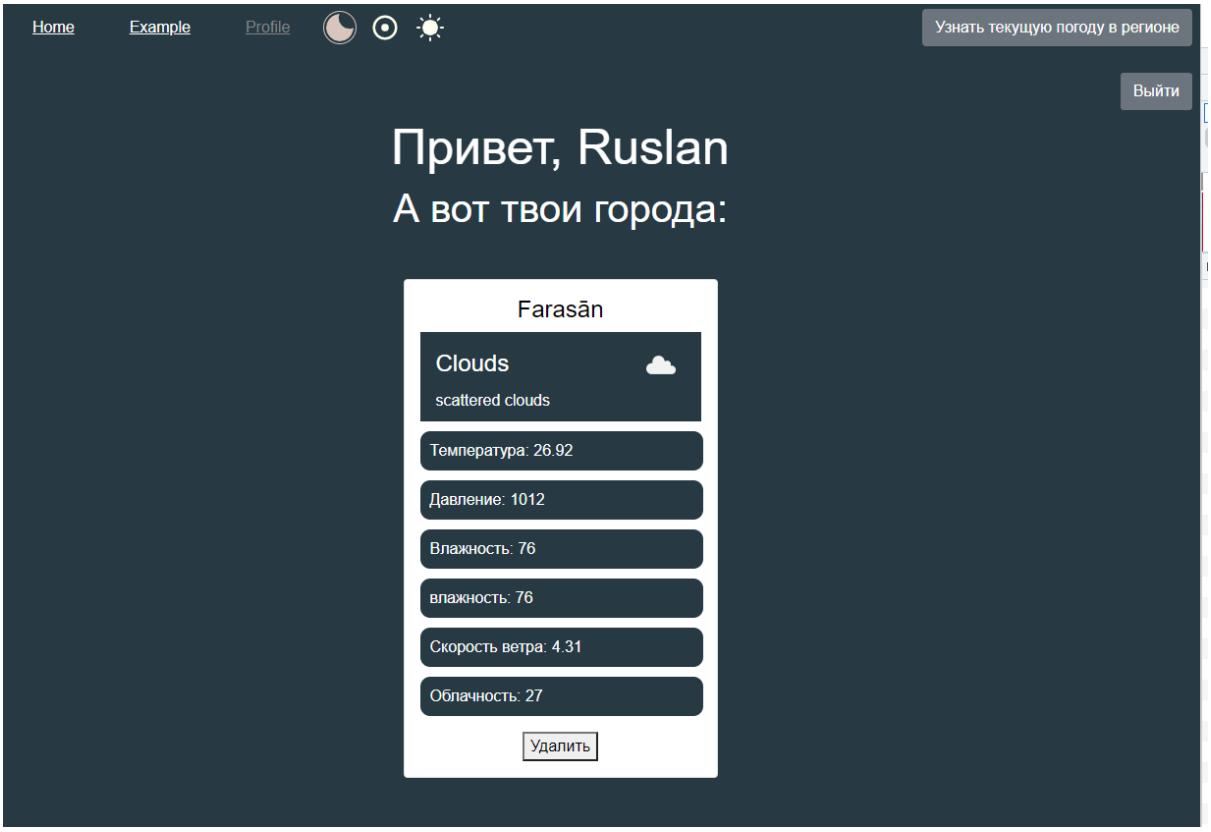

Прогноз погоды
Выберите интересубщий город
Farasān
Узнать погоду
Farasān
Clouds
scattered clouds
Температура: 26.92
Давление: 1012
Влажность: 76
влажность: 76
Скорость ветра: 4.31
Облачность: 27
Добавить в избранное

Как мы можем видеть, можно добавить город в избранное:

```
            <button v-on:click="addClick">Добавить в избранное</button>
        </b-card>
    </div>
/template>

script>
    import {mapGetters} from "vuex";

    export default {
        name: "MetcastCurrent",
        computed: {
            ...mapGetters({
                metcastCurrent: 'metcastCurrent'
            })
        },
        methods: {
            addClick: function () {
                this.$store.dispatch( type: 'postCity')
            }
        }
```

```
    postCity() {
        /**
         * POST api/me/city - отправление нового города пользователя на сервер по айди,
         * @constructor
         * @return none.
         */
        fetch( input: `${URL}api/me/city/`, init: {
            headers: {
                'Content-Type': 'application/json',
                'Authorization': `Bearer ${JSON.parse(localStorage.getItem( key: 'token'))?.access}`
            },
            method: 'POST',
            body: JSON.stringify( value: {"city": this.state.city.city.id})
        })
            .then(checkResponse)
            .then(() => {
                this.dispatch('addUser')
            })
            .catch(err => console.log(err))
    }
```

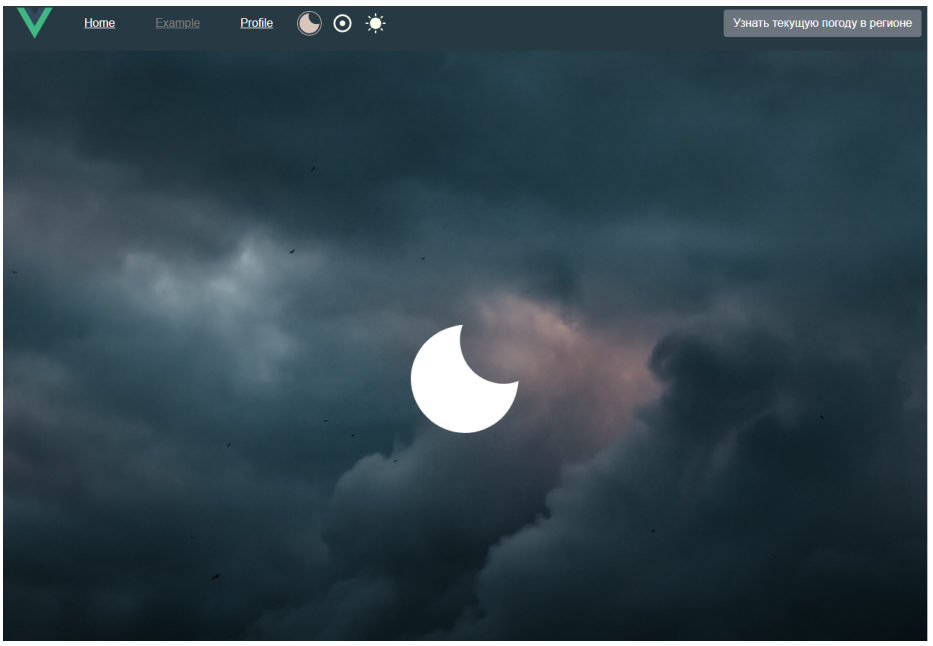
По итогу данного метода, на сервер отправится запрос на добавление города, и при успешном ответе, обновляются данные юзера.

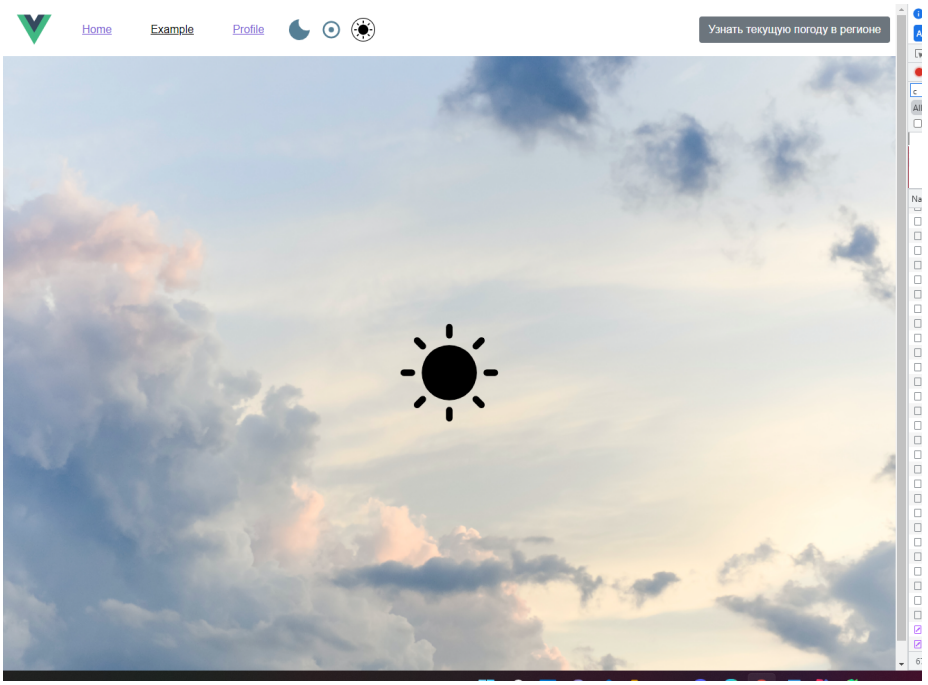


Как мы можем видеть на странице юзера, теперь есть новый закрепленный город, также есть метод выхода из аккаунта, и удаления города, что вызывает обращение к серверу на удаление города.

/example
На данной странице можно посмотреть реализацию смены темы, которая настроена с помощью данной функции:

```
changeTheme (state, payload) {
    state.theme = payload.theme;
    localStorage.setItem('theme', payload.theme)
    if (state.theme==='auto') {
        state.activeTheme = window.matchMedia && window.matchMedia( query: '(prefers-color-scheme: dark)').matches? 'dark':'light';
    }

    if (state) {
        let lightMedia;
        let darkMedia;

        if (state.theme === 'auto') {
            lightMedia = '(prefers-color-scheme: light)';
            darkMedia = '(prefers-color-scheme: dark)';
        } else {
            lightMedia = (state.theme === 'light') ? 'all' : 'not all';
            darkMedia = (state.theme === 'dark') ? 'all' : 'not all';
        }

        [...lightStyles].forEach((link) => {
            link.media = lightMedia;
        });

        [...darkStyles].forEach((link) => {
            link.media = darkMedia;
        });
    }
```





ЗАКЛЮЧЕНИЕ

Выводы по работе

Благодаря данной работе, я освоил новый для себя стек технологий vue, vuex, vue-router. Реализовал свой собственный прогноз погоды, с обработкой 200000 регионов. Оставил методы для улучшения данного функционала. Django в связке с vue предоставляет возможность быстрой реализации готового приложения, за короткие сроки.

СПИСОК ЛИТЕРАТУРЫ

1. Документация Django https://docs.djangoproject.com/en/3.0/.
2. Документация Django REST Framework https://www.django-rest-framework.org.
3. Документация Vuex https://vuex.vuejs.org/ru/
4. Документация Vue-router https://router.vuejs.org/ru/api/
5. Документация VueJs https://vuejs.org/