Министерство науки и высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»
(Университет ИТМО)

Факультет Программной инженерии и компьютерной техники

Образовательная программа Программно-информационные системы

Направление подготовки (специальность) 09.03.04 – Программная инженерия

О Т Ч Е Т

о производственной практике, производственно-технологической

Тема задания: Разработка внутреннего ресурса для фармацевтов аптеки

Обучающийся Ореховский Антон, Р3317

Руководитель практики от профильной организации:
Руководитель практики от университета: Маркина Татьяна Анатольевна, Университет ИТМО, старший преподаватель

Практика пройдена с оценкой _____

Подписи членов комиссии:

_____ _____
(подпись)

_____ _____
(подпись)

_____ _____
(подпись)

Дата _____

Санкт-Петербург
2020

Оглавление

Этап 1. Подготовка к выполнению практической части

Ознакомившись с данной предметной областью, было составлено следующее техническое задание:

Назначение разработки

Данное приложение является инструментом для фармацевтов аптеки, которое поможет им с легкостью просматривать доступные лекарства, фильтровать их, искать по ключевым словам, продавать их и изменять в случае необходимости.

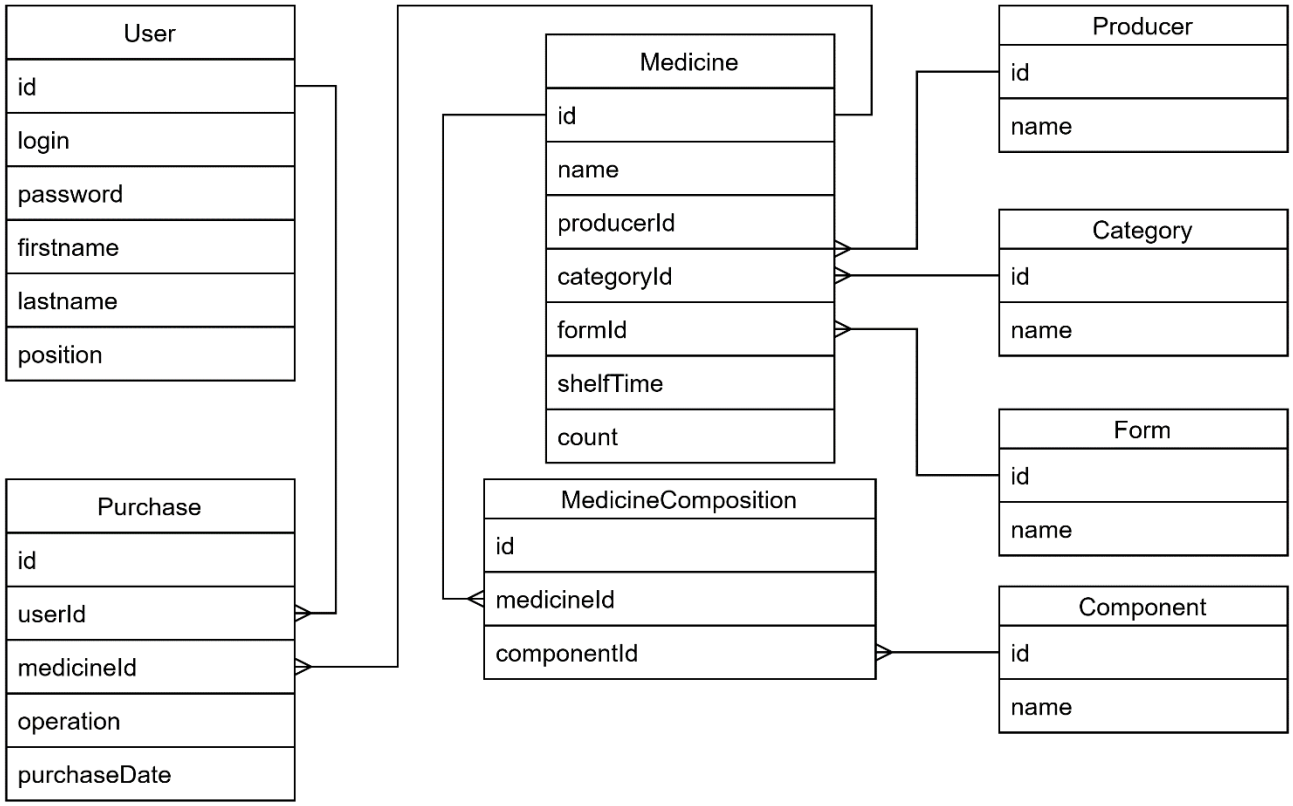
Технические условия

- Система хранения данных
  - СУБД MS SQL Server
  - Язык программирования SQL, TSQL
- Сервер
  - Протокол передачи данных REST (JSON)
  - Протокол сеансов HTTPS
  - Протокол защиты информации SSL
  - Среда разработки ASP.NET Core
- Клиент
  - Платформа для разработки Angular
  - Язык разработки TypeScript

Логика работы

1. Вход с авторизацией по логину-паролю;
2. Просмотр списка лекарств аптеки с пагинацией;
3. Возможность фильтрации списка лекарств по нескольким полям;
4. Пополнение и редактирование списка лекарств;
5. Параметры лекарств для ввода и редактирования:
   - Название
   - Производитель
   - Тип лекарства из фиксированного набора. Например: витамины, антибиотики, косметика, детские лекарства и т.п.
   - Срок годности
6. Поиск по полям: Название, производитель

Далее мною была составлена база данных со следующей моделью:



Этап 2. Реализация серверной части

Реализацию серверной части я начал с создания объектно-реляционных моделей данных, так как они являются основополагающим элементом связи приложения с базой данных. Данные модели можно было создать вручную, а можно было использовать специальную утилиту Scaffold, которую предоставляет EntityFramework – наиболее популярный фреймворк связи приложения с БД. Данная утилита автоматически создает шаблоны ОРМ и класс DbContext по существующей БД. Пример такой ОРМ, которая отражает таблицу Medicine:

```
public partial class Medicine
{
    public Medicine()
    {
        MedicineComposition = new HashSet<MedicineComposition>();
        Purchase = new HashSet<Purchase>();
    }
    public int Id { get; set; }
    public string Name { get; set; }
    public int ProducerId { get; set; }
    public int CategoryId { get; set; }
    public int FormId { get; set; }
    public int ShelfTime { get; set; }
    public int Count { get; set; }

    public virtual Category Category { get; set; }
    public virtual Form Form { get; set; }
    public virtual Producer Producer { get; set; }
    public virtual ICollection<MedicineComposition> MedicineComposition { get; set; }
    public virtual ICollection<Purchase> Purchase { get; set; }
}
```

Далее мною были написаны контроллеры, которые предоставляют методы, реализующие бизнес логику данного приложения. В своей первоначальной версии эти контроллеры содержали логику работы с базой данных, что не является хорошей практикой, так как это ухудшает масштабируемость кода и «захламляет» логику приложения, поэтому для улучшения качества структуры приложения, мною были написаны специальные сервисы для доступа к БД, которые подключались к контроллерам через Dependency Injection. В целях независимости методов от их реализации, в контроллере подключается интерфейс сервиса, а не его конкретная реализация. Пример интерфейса:

```
public interface IUserService
{
        public User GetUser(string login, string password);
        public int GetUserPositionId(int id);
        public string GetUserPosition(int id);
        public IEnumerable<User> GetAllUsers();
        public void AddUser(UserViewModel user);
}
```

Так как то, что возвращают контроллеры зачастую отличается от ОРМ, имеет смысл создать отдельные модели, которые будут возвращать контроллеры. Что бы не пришлось прообразовывать данные из одной модели в другую вручную, можно использовать специальную библиотеку AutoMapper, что я и сделал. Подключить ее к проекту не составило труда. Для корректной работы необходимо было лишь создать так называемые профили для автоматического преобразования. Пример такого профиля:

```
public class UserProfile : Profile
{
        public UserProfile()
        {
            CreateMap<User, UserViewModel>()
                .ForMember(dest => dest.Position, opt => opt.MapFrom(src =>
src.PositionNavigation.Name))
                .ForMember(dest => dest.PositionId, opt => opt.MapFrom(src =>
src.Position));
        }
}
```

Пример контроллера, который использует Mapper и Service:

```
[Route("api/[controller]/[action]")]
[ApiController]
public class UserController : ControllerBase
{
        private readonly IUserService _userService;
        private readonly IMapper _mapper;
        public UserController(IUserService userService, IMapper mapper)
        {
            _userService = userService;
            _mapper = mapper;
        }

        [HttpGet]
        [ActionName("GetUser")]
        public UserViewModel GetUser([FromQuery] string login, [FromQuery] string password)
=> _mapper.Map<UserViewModel>(_userService.GetUser(login, password));

        [HttpGet]
        [ActionName("ValidateUser")]
        public bool ValidateUser([FromQuery] string login, [FromQuery] string password)
```

```
        {
            return (_userService.GetUser(login, password) != null);
        }

        [HttpGet]
        [ActionName("GetUserPosition")]
        public string GetUserPosition([FromQuery] int id) =>
_userService.GetUserPosition(id);

        [HttpGet]
        [ActionName("GetUserPositionId")]
        public int GetUserPositionId([FromQuery] int id) =>
_userService.GetUserPositionId(id);

        [HttpGet]
        [ActionName("GetAllUsers")]
        public IEnumerable<UserViewModel> GetAllUsers()
            => _userService.GetAllUsers().Select(element =>
_mapper.Map<UserViewModel>(element));

        [HttpPost]
        [ActionName("AddUser")]
        public void AddUser([FromBody] UserViewModel user)
            => _userService.AddUser(user);
}
```

Этап 3. Реализация клиентской части

Клиентские приложения написанные на Angular состоят из модулей и компонентов. Компоненты — это своего рода страницы, которые видит пользователь. Они имеют свою бизнес логику, свое отображение и стиль. Компоненты могут также состоять из других компонентов. Модули – это механизм объединения компонентов, сервисов и т. п. в единое целое. Мое приложение состоит из одного модуля и четырех компонентов.

Пример компонента:

```
<header>
  <nav class="navbar navbar-dark navbar-expand-md bg-dark">
    <div class="navbar-brand">Аптека</div>
    </nav>
</header>
<div class="login-form">
    <h1>Войдите в систему</h1>
    <div>
      <input class="form-control" type="text" id="login" placeholder="Логин">
      <input class="form-control" type="password" id="password" placeholder="Пароль">
      <button class="btn btn-outline-success float-right login-btn" (click)="login()"
onclick="return false;">Войти</button>
      <small id="error-msg" class="text-danger"> </small>
    </div>
</div>
```

```
import { Component, OnInit } from '@angular/core';
import { HttpClient, HttpErrorResponse } from '@angular/common/http';
import { Injectable } from '@angular/core';
import { Router } from '@angular/router';
import { LoginService } from './login.service';

@Component({
  selector: 'app-login',
  templateUrl: './login.component.html',
```

```
  styleUrls: ['./login.component.css'],
  providers: [LoginService]
})
export class LoginComponent implements OnInit {

  constructor(private loginService: LoginService, private router: Router) { }

  ngOnInit() {
    if ($(document).height() <= $(window).height())
      $("#footer").addClass("fixed-bottom");
  }


  login() {
    var login = <string > $("#login").val();
    var password = <string> $("#password").val();
    this.loginService.validateUser(login, password)
      .subscribe( (data:any) => {
        if (data) {
          localStorage.setItem('access-token', data.access_token);
          localStorage.setItem('username', data.username);
          localStorage.setItem('role', data.role);
          this.router.navigateByUrl('/medicine-list');
        }
      }, (error: HttpErrorResponse) => {
        $('#error-msg').text('Введённые логин и пароль неверны');

      });
  }
}
```

Так как компоненты отвечают только за определенную часть отображения, необходимо было создать навигацию между ними. Модуль навигации имеет следующий вид:

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { LoginComponent } from './login/login.component';
import { MedicineComponent } from './medicine/medicine.component';
import { MedicinelistComponent } from './medicinelist/medicinelist.component';
import { ErrorComponent } from './error/error.component';


const routes: Routes = [
  { path:'', component: LoginComponent },
  { path:'medicine-list', component: MedicinelistComponent },
  { path:'medicine/:id', component: MedicineComponent},
  { path:'error', component: ErrorComponent},
  { path:'**', redirectTo: '/error?status=404&message=Страница не найдена'}
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
       export class AppRoutingModule { }
```

Этап 4. Связывание клиентской и серверной частей

Основной способ взаимодействия клиента с сервером при данном наборе технологий основан на подходе REST. Сервер предоставляет API с доступными методами (в моем случае только Get и Post). Клиент-Angular же обращается посредством модуля HttpClient. Хорошей практикой является написание взаимодействия клиент-сервер в отдельном файле, так как это

структурирует код и упрощает его переиспользование. Следуя этому принципу, я создал специальный сервис для каждого компонента. Пример одного из них:

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';

import { Router, ActivatedRoute } from '@angular/router';
import { Observable } from 'rxjs';

@Injectable()
export class LoginService {
    constructor (private http:HttpClient, private router: Router, private activatedRoute:
ActivatedRoute) { }

    apiUrl = '/api/Login/';

    validateUser(login:string, password:string) {
        return this.http.get('/api/Auth/ValidateUser?login=' + login + "&password=" +
password)
    }
}
```

Для осуществления авторизации, по запросу ValidateUser в контроллере Auth генерируется JWT токен, который является ключом авторизации. На стороне клиента этот ключ записывается во все заголовки запросов к серверу при помощи модуля HttpInterceptor

```
import { Injectable } from '@angular/core';
import { HttpRequest, HttpHandler, HttpEvent, HttpInterceptor, HttpErrorResponse } from
'@angular/common/http';
import { Observable, throwError, of } from 'rxjs';
import { catchError } from 'rxjs/operators';
import { Router } from '@angular/router';

@Injectable()
export class AuthInterceptor implements HttpInterceptor {

  constructor(private router: Router) {}

  intercept(request: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    request = request.clone({
      setHeaders: {
        Authorization: `Bearer ${localStorage.getItem('access-token') }`
      }
    });
    return next.handle(request).pipe(catchError(x => this.handleAuthError(x)));
  }

  private handleAuthError(err: HttpErrorResponse):Observable<any> {
    if (err.status === 401 ) {
      localStorage.clear();
      this.router.navigateByUrl('/error?status=401&message=Ошибка авторизации');
      return of(err.message);
    }
    if (err.status === 403) {
      this.router.navigateByUrl('/error?status=403&message=Отказано в доступе');
      return of(err.message);
    }
    return throwError(err);
  }
}
```

В этом модуле я написал логику обработки ошибок типа 401 и 403.

На стороне сервера авторизация происходит автоматически у тех методов, которые помечены специальным атрибутом Authorize. Так же в этом атрибуте можно указывать роли, которые имеет доступ к этому методу. Так, например, приведенный ниже метод доступен только авторизованным пользователям, роль у которых admin или manager:

```
[Authorize(Roles = "admin, manager"), HttpGet]
[ActionName("AlterMedicine")]
public async Task<bool> AlterMedicine([FromQuery] int id, [FromQuery] string name,
    [FromQuery] string producer, [FromQuery] string category, [FromQuery] string form,
    [FromQuery] string[] component, [FromQuery] int shelfTime, [FromQuery] int count)
{
    MedicineViewModel medicine = new MedicineViewModel
    {
        Id = id,
        Name = name,
        Producer = producer,
        Category = category,
        Form = form,
        Components = component,
        ShelfTime = shelfTime,
        Count = count
    };
    var login = GetLogin(Request.Headers["Authorization"].First());
    return await _medicineApiProvider.AlterMedicine(medicine, login);
}
```

Этап 5. Выполнение дополнительных заданий

Любое веб-приложение должно иметь централизованную систему ошибок. Для этой роли я написал компонент, который будет отображать все ошибки.

```
import { Component, OnInit } from '@angular/core';
import { Router, ActivatedRoute } from '@angular/router';

@Component({
  selector: 'app-error',
  templateUrl: './error.component.html',
  styleUrls: ['./error.component.css']
})
export class ErrorComponent implements OnInit {

  constructor(private router: Router, private activatedRoute: ActivatedRoute) { }

  status:string;
  message:string="Произошла непредвиденная ошибка";
  isAuthorized:boolean;

  ngOnInit() {
    this.isAuthorized = localStorage.getItem('access-token') ? true : false;
    if ($(document).height() <= $(window).height())
      $("#footer").addClass("fixed-bottom");
    this.activatedRoute.queryParamMap.subscribe(params => {
      this.status = params.get("status");
      if (params.get("message")) {
        this.message = params.get("message");
      }
    })
  }
}
```

Также для более четкого разделения логики имеет смысл вынести в отдельный API бизнес логику взаимодействия с БД. Таким образом я создал новые API и подключил из к

главному посредством ApiProvider-ов – специальных классов, которые будут делать запросы к новым API.

Пример Provider-а:

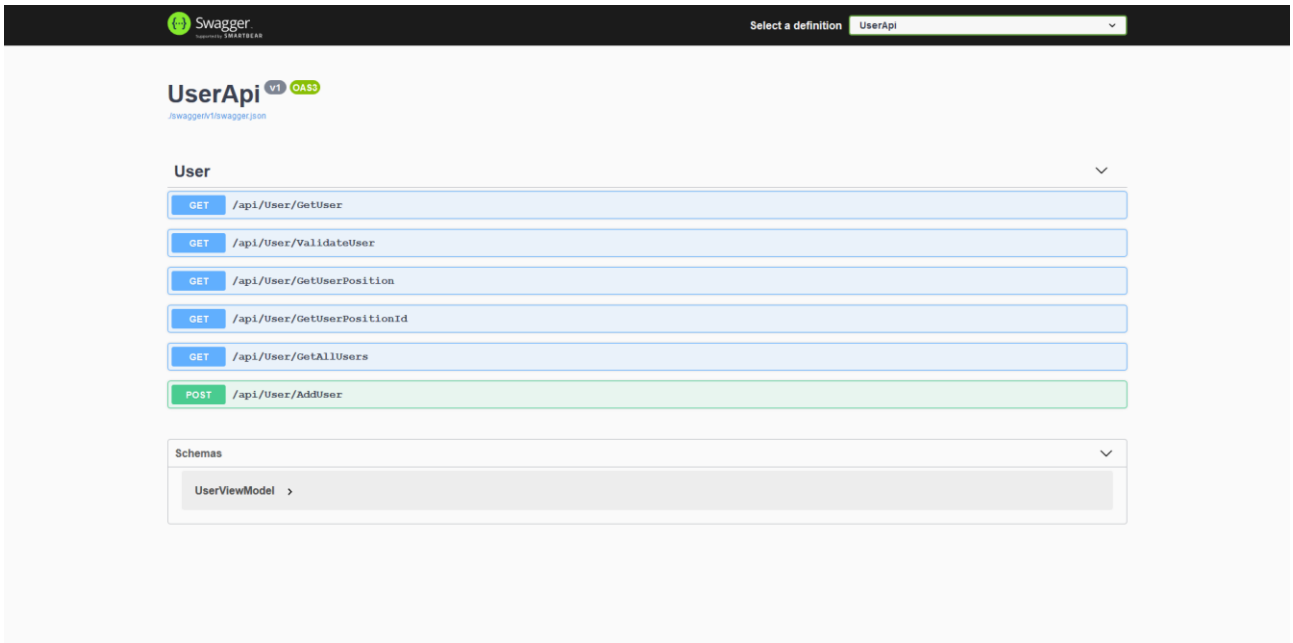
```
public class UserApiProvider : IUserApiProvider
    {
        private readonly string URL;
        static HttpClient client = new HttpClient();
        public UserApiProvider(IConfiguration configuration)
        {
            URL = configuration.GetValue<string>("UserApiUrl");
        }

        private async Task<T> GetRequest<T>(string requestString)
        {
            HttpResponseMessage responce = await client.GetAsync(URL + requestString);
            if (responce.IsSuccessStatusCode)
            {
                return await responce.Content.ReadAsAsync<T>();
            }
            return default;
        }
        private async Task<bool> PostRequest(string requestString, HttpContent content)
        {
            var responce = await client.PostAsync(URL + requestString, content);
            return responce.IsSuccessStatusCode;
        }

        public async Task<UserViewModel> GetUser(string login, string password)
            =>await ПetRequest<UserViewModel>($"GetUser?login={login}&password={password}");
       . . .
```

Также, для удобства, я добавил Swagger для каждого нового API – пакет, который показывает все доступные методы у данного API.

Пример такого Swagger-а:



В данном Swagger-е можно с легкостью выполнить запрос к API, просмотреть список методов и способов обращения к ним.

Последним дополнительным заданием было логирование каждой операции с количеством лекарств. Для этого из каждого запроса от клиента на изменение количества медикаментов необходимо было разбирать JWT токен из заголовка Authorization и запоминать пользователя, который совершает это действие.

Пример такого метода:

```
[Authorize(Roles = "seller"), HttpGet]
[ActionName("SellMedicine")]
public async Task<bool> AlterMedicine([FromQuery] int id, [FromQuery] int count)
{
       var login = GetLogin(Request.Headers["Authorization"].First());
       return await _medicineApiProvider.SellMedicine(id, count, login);
}
```

Приватный метод GetLogin:

```
private string GetLogin(string token)
{
            var handler = new JwtSecurityTokenHandler();
            var tokenS = handler.ReadToken(token.Substring(7)) as JwtSecurityToken;
            return tokenS.Claims.Where(claim => claim.Type ==
                     ClaimsIdentity.DefaultNameClaimType).First().Value;
}
```