НЕУДОБНЫЕ ВОПРОСЫ:

разработка REST API

Макиевский Станислав Евгеньевич,

Преподаватель ИПТИП, fullstack-разработчик (C#)

МИРЭА
Российский технологический университет

Кто я?

- # Выпускник ФГБОУ ВПО «Сибирский государственный индустриальный университет».
- # Разработчик (Java - с 2010 по 2015 г, PHP – с 2013 г, C# - с 2014 г), руководитель разработки.
- # Автор курсов и программ обучения, ментор WorldSkills Russia, тренер сборной России по направлению «Программные решения для бизнеса», тренер сборной Москвы по программированию, чемпион по направлению «Бизнес-программирование» в чемпионатах профессионального мастерства по г. Москве с 2017 по 2022 гг.
- # Сертифицированный разработчик Microsoft Corp., fullstack-программист.

...

НЕУДОБНЫЕ ВОПРОСЫ: разработка REST API

Макиевский Станислав Евгеньевич

План мастер-класса

- Что такое RESTful API?
- Как применять RESTful:
  - что такое представление, как оно связано с доменной сущностью;
  - как CRUD ложится на HTTP verbs;
  - Практики написания;
  - Технологии;
- Пример: проектируем RESTful API для магазина комиксов.
- Микросервисы и монолиты по нагрузке (простое);
- Математическое описание расчета эффективности применения SOLID;
- Описание API с помощью протокола OpenAPI.
- Версионирование API.

ЧТО ТАКОЕ RESTFUL?

REST — это аббревиатура от **Representational State Transfer** (Передача Состояния Представления).

ИЛИ это согласованный набор архитектурных принципов для создания более масштабируемой и гибкой сети.

ЧТО ТАКОЕ RESTFUL?

> Сервер содержит ресурсы и определяет операции над ними: БД, JSON, XML и другие.

> Клиент работает с представлениями ресурсов, для изменения состояния сервера он передает желаемое представление ресурса на сервер.

И? Зачем? ТЕХНОЛОГИЯ.

НЕУДОБНЫЕ ВОПРОСЫ: разработка REST API

Макиевский Станислав Евгеньевич

ЧТО ТАКОЕ ПРЕДСТАВЛЕНИЕ?

В терминологии REST API ресурс – это:

1. Информация, которую различные приложения предоставляют своим клиентам;
2. Он должен иметь постоянный уникальный идентификатор;
3. Клиент управляет ресурсами, направляя серверу данные в виде JSON, содержащие представление;
4. Использует методы (глаголы): добавить, удалить или изменить.

CRUD МЕТОДЫ НА БАЗЕ HTTP

НАЗВАНИЕ		Описание
C	POST	Создание новых ресурсов. При успешном создании ресурса возвращается HTTP код 201, а также в заголовке «Location» передается адрес созданного ресурса.
R	GET	Получение представления ресурса. Запрос используется только для чтения данных, менять с помощью него состояние нельзя.
U	PUT	Полное обновление или создание ресурса. В отличие от POST, PUT метод передает ID ресурса, выбранного клиентом.
	PATCH	Изменение представления существующего ресурса по ID. Возможна передача не полного представления, а только полей, которые требуется изменить.
D	DELETE	Используется для удаления ресурса, идентифицированного конкретным ID.

ХОРОШИЙ, ПЛОХОЙ, ЗЛОЙ RESTFUL

Получение данных о журнале:

ПЛОХО GET: /comics?id=1

ХОРОШО GET: /comics/{id}

Макиевский Станислав Евгеньевич

ХОРОШИЙ, ПЛОХОЙ, ЗЛОЙ RESTFUL

Получение данных о всех комиксах:

ПЛОХО GET: /comics/all

ХОРОШО GET: /comics/

ХОРОШИЙ, ПЛОХОЙ, ЗЛОЙ RESTFUL

Создание журнала:

ПЛОХО GET /comics?action=create&name=...

ХОРОШО POST: /comics

RAW body: {"name": "..."}

Макиевский Станислав Евгеньевич

ХОРОШИЙ, ПЛОХОЙ, ЗЛОЙ RESTFUL

Полностью обновить комикс по ID:

ПЛОХО PUT /comics?action=put&name=...

ХОРОШО PUT: /comics/{id}

RAW body: {"name": "..."}

Макиевский Станислав Евгеньевич

ХОРОШИЙ, ПЛОХОЙ, ЗЛОЙ RESTFUL

Частично обновить комикс по ID:

ПЛОХО PATCH /comics?action=patch&name=...

ХОРОШО PATCH: /comics/{id}

RAW body: {"name": "..."}

ХОРОШИЙ, ПЛОХОЙ, ЗЛОЙ RESTFUL

Удалить комикс по ID:

ПЛОХО DELETE /comics?id ={id}

ХОРОШО DELETE: /comics/{id}

БАЗА ДАННЫХ

Ссылка на файл SQL

МЕХАНИКА ЗАПРОСА

REQUEST:

Метод:

GET
POST
PUT (PATCH)
DELETE

+ URL

RESPONCE:

Код состояния:

200 (OK)
302 (Found)
404 (Not found)
502 (Bad Gateway)


```
[
  {
    "id": 0,
    "image_url": "string",
    "name": "string",
    "price": 0,
    "description": "string",
    "author_id": 0,
    "genre_id": 0,
    "published_date": "2024-10-21",
    "stock": 0
  }
]
```

ПИШЕМ? А НА ЧЁМ?

- PHP (MVC);
- ASP.Net 8.0 (SOLID);
- RUST (WA);
- PHP (easy-and-fasty);

Что такое MVC?

Model
(Service)
View
Controller

- это архитектурный шаблон, который разделяет приложение на три (четыре) основные части:

Что такое MVC?

Model (Модель) — это та часть, которая отвечает за данные.

Она **описывает логику работы** с данными и их структуру, но **не отвечает за то, как они отображаются**.

Что такое MVC?

Service (слой бизнес-логики, сервис) — это компонент архитектуры, который отвечает за выполнение бизнес-логики приложения, обрабатывая данные между контроллером и моделью.

Он инкапсулирует сложные операции, обеспечивая разделение ответственности и повторное использование логики.

Что такое MVC?

View (Представление) — это то, что видит пользователь.

МОДЕЛЬ

- Хранит;
- Обрабатывает;

ИНФОРМАЦИЮ

Представление отвечает за отображение данных на экране. Оно получает данные от модели и показывает их пользователю в удобном виде (например, как веб-страница).

Что такое MVC?

Что такое MVC?

Controller (Контроллер) — это "связующее звено" между моделью и представлением.

МОДЕЛЬ

•Хранит;
•Обрабатывает;

ИНФОРМАЦИЮ

Контроллер принимает запросы от пользователя (например, через нажатие кнопок на сайте), обрабатывает их, взаимодействует с моделью для получения или изменения данных, и затем выбирает, какое представление показать пользователю.

Что такое MVC?

Просто о понятном

 # Модель – это как **база данных** или **структура данных**. Например, "комиксы" в магазине — это данные, которые модель обрабатывает.

 # Представление - это то, как **информация о комиксах отображается на экране** для пользователя (например, список комиксов на сайте).

 # Контроллер – это когда вы **нажимаете** кнопку "Показать комиксы", контроллер **получает** этот запрос, **обращается к модели за данными** (например, из базы данных), а затем **выбирает**, как эти данные будут **показаны** (представление).

НА ПЕРЕФЕРИИ ТЕХНОЛОГИЙ

В примерах разработки использовались:

OPENAPI – Swager;

API-Test – Postman;

DB – MySQL;

Системы управления пакетами: NuGet, composer, cargo;

REST API PHP: по шагам

Шаг 1. Подготовим БД и реализуем её в рамках СУБД.

REST API PHP: по шагам

Шаг 2. Определимся с архитектурой проекта.

1. public # Публичная папка для API
2. src # Исходники приложения
3. Controller # Контроллеры для обработки запросов
4. Service # Сервисы для логики приложения
5. Config # Конфигурации приложения
6. Model # Модели данных
7. vendor # Внешние библиотеки (например, Composer)

REST API PHP: по шагам

Шаг 3. Создаем файл src/Config/Database.php конфигурации базы данных.

```php
<?php
namespace Config;

use PDO;
use PDOException;

class Database {
    private $host = 'localhost';        // Хост базы данных
    private $db_name = 'burnfeniks_opendoor'; // Имя базы данных
    private $username = '046351469_open';// Имя пользователя
    private $password = '046351469_open';// Пароль пользователя
    private $conn;                      // Соединение с базой данных

```

Перейдем к контенту

REST API PHP: по шагам

Перейдем к контенту

```php
<?php
require_once __DIR__ . '/../vendor/autoload.php';

use Config\Database;
use Service\ComicsService;
use Controller\ComicsController;

// Создаем объект базы данных и подключаемся
$database = new Database();
$db = $database->getConnection();

// Создаем сервис и контроллер для работы с комиксами
$comicsService = new ComicsService($db);
$comicsController = new ComicsController($comicsService);

// Получаем метод запроса и маршрут
$method = $_SERVER['REQUEST_METHOD'];
$uri = parse_url($_SERVER['REQUEST_URI'], PHP_URL_PATH);
```

Шаг 4. Создаем файл public/index.php конфигурации базы данных.

REST API PHP: по шагам

```php
<?php
namespace Service;

use PDO;

class ComicsService {
    private $conn;

    public function __construct($db) {
        $this->conn = $db;
    }

    // Получение всех комиксов
    public function getAllComics() {
        $query = "
            SELECT
                comics.id, comics.image_url, comics.name, comics.price, comics.description,
                comics.published_date, comics.stock,
                authors.id AS author_id, authors.name AS author_name,
                genres.id AS genre_id, genres.genre_name AS genre_name
            FROM
                comics
            JOIN
                authors ON comics.author_id = authors.id
            JOIN
                genres ON comics.genre_id = genres.id
        ";

```

Шаг 5.
Создаем файл src/Service/ComicsService.php и описываем запросы к данным.

Перейдем к контенту

REST API PHP: по шагам

Перейдем к контенту

Шаг 6. Создаем контроллер src/Controller/ComicsController.php для обработки запросов.

```php
<?php
namespace Controller;

use Service\ComicsService;

class ComicsController {
    private $comicsService;

    public function __construct(ComicsService $comicsService) {
        $this->comicsService = $comicsService;
    }
```

REST API PHP: вывод

- Папка `public/` содержит публично доступные файлы, которые обрабатывают HTTP-запросы.
- Папка `src/` включает все классы, структурированные по принципам ООП:
  - Контроллеры `Controller/` отвечают за обработку запросов.
  - Сервисы `Service/` содержат логику приложения, такую как работа с JWT и пользователями.
  - Модели `Model/` представляют данные и взаимодействие с базой данных.
  - Конфигурация `Config/` отвечает за подключение к базе данных.
- Маршрутизация (например, через параметр action в URL) направляет запросы к соответствующим методам контроллеров.

Таким образом, данная структура проекта помогает разделить логику на отдельные слои, что облегчает поддержку, тестирование и расширение кода.

REST API PHP: вывод

- Директория vendor/ содержит файлы autoload.php и composer.json .
  - Файл autoload.php содержит директивы по проектам (точки доступа);
  - Файл composer.json информацию по библиотекам;
- Директория swagger/ содержит файл swagger.json для настройки визуализации ваших запросов к API.
  - JSON: объекты, массивы, перечисления;
  - Схема данных;
  - Код состояния;

REST API PHP: по шагам

Шаг 7. Приступаем к работе с Swagger и создаем файл public/swagger.php и инициализируем точку входа.

```php
<?php
header('Content-Type: application/json');
// Убедитесь, что путь к файлу swagger.json корректен
echo file_get_contents(__DIR__ . '/../swagger/swagger.json');
```

Перейдем к контенту

REST API PHP: по шагам

Шаг 8. Там же создаем файл с интерфейсом Swagger UI public/swagger.html.
Например: https://opendoor.easy4.team/public/swagger.html

```php
<?php
header('Content-Type: application/json');
// Убедитесь, что путь к файлу swagger.json корректен
echo file_get_contents(__DIR__ . '/../swagger/swagger.json');
```

Перейдем к контенту

REST API PHP: по шагам

Шаг 9. Конфигурируем сервер и создаем файл public/.htaccess

```
RewriteEngine On
RewriteBase /public

# Перенаправляем все запросы на index.php, кроме существующих файлов и директорий
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d
RewriteRule ^(.*)$ index.php [QSA,L]

# Разрешаем CORS для OPTIONS запросов
<IfModule mod_headers.c>
    Header always set Access-Control-Allow-Origin "*"
    Header always set Access-Control-Allow-Methods "GET, POST, OPTIONS"
    Header always set Access-Control-Allow-Headers "Content-Type, Authorization"
</IfModule>
```

Нам он нужен для:

1. Настройки маршрутизации REST API для передачи на единственную точку входа (index.php);
2. Управления CORS (если требуется доступ из других доменов).

Перейдем к контенту

REST API PHP: вывод

Анализ всех моих решений:

Структурирование проекта:

- Создана корректная иерархия директорий и файлов для REST API;
- Придерживаясь принципов ООП;
- Иерархия создана с разделением на классы для контроллеров, сервисов, моделей и конфигурации базы данных;
- Отличная поддерживаемость проекта;
- Расширяемый проект;
- Четкое разделение ответственности между классами.

REST API PHP: вывод

Анализ всех моих решений:

Создание файлов с исходным кодом:

- Все файлы: index.php, конфигурации базы данных, модели, контроллеры и сервисы, - были правильно созданы и размещены в нужных местах.
- Каждый файл соответствует своей функциональной ответственности. Контроллеры отвечают за запросы, сервисы за бизнес-логику, а модели работают с базой данных.

PHP. M(S)VC. ПРОСТО.

Жизненный цикл по улучшению вашей API:

НЕУДОБНЫЕ ВОПРОСЫ: разработка REST API

Макиевский Станислав Евгеньевич

PHP (api): авторский подход

- Отсутствие Swagger;
- Простая обработка;
- Расширяемый проект;
- Простая поддержка;
- Без ООП;
- Без классов;
- Без сервисов;

Ссылка на идею

PHP (api): авторский подход

Шаг 1. Обращаемся к файлу connect.php и вносим свои данные для подключения к СУБД.

Code | Blame | 3 lines (3 loc) · 174 Bytes

```php
<?php
    $connect = mysqli_connect("localhost", "твой логин", "твой пароль", "твоя БД") or die("Подключение отсутствует.");
?>
```

Перейдем к контенту

PHP (api): авторский подход

Шаг 2. Настраиваем файл .htaccess для того чтобы все запросы шли от файла index.php.

```
RewriteEngine on
RewriteCond %{REQUEST_FILENAME} !-f
RewriteRule ^(.*)$ index.php\?q\=$1 [L,QSA]
```

Перейдем к контенту

PHP (api): авторский подход

Перейдем к контенту

```php
<?php
    header("Content-Type:application/json");
    header('Access-Control-Allow-Origin: *');

    require_once 'connect.php';
    require_once 'function_post.php';
    require_once 'function_get.php';
    require_once 'function_responce.php';

    $actionMethod = $_SERVER['REQUEST_METHOD'];

    $paramUrl = explode("/", $_GET['q']);
    $typeUrl = $paramUrl[0];
    $idUrl = $paramUrl[1];
            //GET
    switch ($actionMethod) {
        case 'GET':

                switch ($typeUrl) {
                    case 'users':
                        viewallusers($connect);
                    break;
```

Шаг 3.
Настраиваем index.php как файл-маршрутизатор и передаем данные в запросы.

PHP (api): авторский подход

```php
<?php
    require_once "function_responce.php";
    ///
    /// Вывести всех пользователей
    ///
    function viewallusers($connect){
        $users = mysqli_query($connect, "SELECT `user`.`id` AS `id`, `user`.`login` A
        if(mysqli_num_rows($users) == 0){
            http_response_code(404);
            $responce = [
                "status" => false,
                "description" => "Таблица пуста."
            ];

            echo json_encode($responce);
        }
        else{

            $userList = array();
            while($user = mysqli_fetch_assoc($users)){
                $userList[] = $user;
            }

            echo json_encode($userList);
        }
    }
```

Шаг 3.
Настраиваем файлы на исполнение, в зависимости от методов GET, POST, PUT, PATCH, DELETE.

Перейдем к контенту

REST API PHP: вывод

Анализ всех моих действий:

Отсутствие Swagger;
Кошмарная обработка;
Отчасти расширяемый проект;
Простая поддержка и никакой защиты;
Без ООП;

SOLID vs MVC

 # SOLID — это набор принципов объектно-ориентированного программирования, которые помогают делать код более поддерживаемым, гибким и расширяемым.

 # MVC — это архитектурный паттерн, который разделяет приложение на три компонента, он помогает структурировать код и отделить бизнес-логику от представления данных.

А теперь про бизнес

Почему SOLID используют чаще в бизнес-разработке:

- **#Гибкость и поддерживаемость** на всех уровнях архитектуры системы, особенно с быстро меняющимися требованиями, что делает их более подходящими для сложных бизнес-приложений.
- **#Работа с бизнес-логикой,** где SOLID нацелен на решение задач управления бизнес-логикой, где важна возможность адаптации к изменениям.
- **#Тестируемость и модульность** значительно упрощают написание и поддержку тестов.

SOLID: немного про эффективность

- Single Responsibility — *делай модули меньше*
- Open Closed — *делай модули расширяемыми*
- Liskov Substitution — *наследуйся правильно*
- Interface Segregation — *дроби интерфейсы*
- Dependency Inversion — *используй интерфейсы*

SOLID: ещё проще

S — один класс или модуль должен отвечать только за одну задачу.

O — код должен быть открыт для добавления нового функционала, но закрыт для изменений.

L — объекты дочерних классов должны спокойно заменять объекты родительских классов, не ломая программу.

I — лучше несколько маленьких интерфейсов, которые делают что-то одно, чем один большой интерфейс с кучей методов, которыми никто не пользуется.

D — высокоуровневый код не должен зависеть от низкоуровневого. Вместо этого, оба уровня должны зависеть от абстракций — общих правил или интерфейсов.

SOLID: не понятно, можно проще?

S — двигатель отвечает за движение, фары — за свет, а кондиционер — за охлаждение. Каждая часть делает своё дело.

O — ты можешь поставить новую магнитолу в машину, не разбирая весь двигатель. Машина остаётся такой же, а новые функции добавляются.

L — представь, что у тебя было кресло водителя, и ты его заменил на кресло с подогревом. Кресло подошло и ничего не сломалось.

I — если бы в машине был один пульт для всего — для управления двигателем, фарами, кондиционером — это было бы неудобно. Отдельные кнопки для каждой функции.

D — машина может работать с любыми шинами, если они подходят по стандарту. Ты не зависишь от конкретной марки, а можешь выбрать любую шину, которая соответствует требованиям.

ASP.Net: технологии и зависимости

1. **ASP.NET Core 8.0**;
2. **MySQL**;
3. **Entity Framework Core** - ORM для работы с базой данных;
4. **Pomelo.EntityFrameworkCore.MySql** - провайдер для подключения к MySQL;
5. **Swagger/OpenAPI** для генерации документации API и взаимодействия с ним.

ASP.Net: немного про эффективность

Разделение логики контроллера по интерфейсам делает код:

1. Более чистым;
2. Легче тестируемым;
3. Легко расширяемым.

Шаги для реализации CRUD с интерфейсами:

1. Создадим интерфейс для CRUD операций (**IComicService**).
2. Создадим сервис, который реализует этот интерфейс (**ComicService**).
3. Инъекция зависимости (Dependency Injection) сервиса в контроллер (**ComicsController**).

ASP.Net : ready…stady…csharp!

Шаг 1. Создаем необходимый интерфейс IComicService;

```
5   public interface IComicService
6   {
7       Task<IEnumerable<Comic>> GetAllComicsAsync();
8       Task<Comic> GetComicByIdAsync(int id);
9       Task<Comic> CreateComicAsync(Comic comic);
10      Task UpdateComicAsync(int id, Comic comic);
11      Task DeleteComicAsync(int id);
12      Task<Comic> PatchComicAsync(int id, Comic comic);
13  }
```

Ссылка на проект

ASP.Net : ready…stady…csharp!

Шаг 2. Создание реализации сервиса ComicService;

```
6   public class ComicService : IComicService
7   {
8       private readonly ApplicationDbContext _context;
9
10      public ComicService(ApplicationDbContext context)
11      {
12          _context = context;
13      }
14
15      public async Task<IEnumerable<Comic>> GetAllComicsAsync()
16      {
17          return await _context.Comics.ToListAsync();
18      }
```

Ссылка на проект

ASP.Net : ready…stady…csharp!

Шаг 3. Внедрение интерфейса и сервиса в контроллер ComicsController;

```csharp
public class ComicsController : ControllerBase
{
    private readonly IComicService _comicService;

    public ComicsController(IComicService comicService)
    {
        _comicService = comicService;
    }

    // GET: api/Comics
    [HttpGet]
    public async Task<ActionResult<IEnumerable<Comic>>> GetAllComics()
    {
        var comics = await _comicService.GetAllComicsAsync();
        return Ok(comics);
    }
```

Ссылка на проект

НЕУДОБНЫЕ ВОПРОСЫ: разработка REST API

Макиевский Станислав Евгеньевич

ASP.Net : ready…stady…csharp!

Ссылка на проект

Шаг 4. Регистрация сервиса в DI-контейнере*;

```
builder.Services.AddScoped<IComicService, ComicService>();
builder.Services.AddScoped<IUserService, UserService>();
builder.Services.AddScoped<IJwtService, JwtService>();
```

***DI контейнер (Dependency Injection контейнер)** — это программный компонент, который управляет зависимостями объектов в приложении.
В контексте внедрения зависимостей (Dependency Injection, DI) контейнер отвечает за создание и управление жизненным циклом объектов и за предоставление их в другие объекты, которым эти зависимости нужны.

ASP.Net : ready…stady…csharp!

Ссылка на проект

```
builder.Services.AddScoped<IComicService, ComicService>();
builder.Services.AddScoped<IUserService, UserService>();
builder.Services.AddScoped<IJwtService, JwtService>();
```

Пример: если объект класса Controller зависит от объекта Service, то DI контейнер создаст объект Service и передаст его в Controller, избавляя вас от необходимости явно управлять созданием и передачей объектов.

Зачем?

- Упрощается изменение и замена зависимостей (модульность).
- Легче создавать заглушки (моки) для тестирования компонентов (тестирование).
- Компоненты меньше зависят друг от друга и более независимы в реализации (clean code – чистый код).

ASP.Net : JSON WEB TOKEN

Шаг 1. Настройка JWT-аутентификации и установка Microsoft.AspNetCore.Authentication.JwtBearer, настраиваем appsettings.json.

```
11      "Jwt": {
12        "Key": "3wE5r8Vt9pJ2uN7gXzQ1Lk6C4nZmA0YvPsR3HcB5JfO2M8dFgT5PqB1X1E6U7Wn",
13        "Issuer": "Makievskiy_SE",
14        "Audience": "Users"
```

Key – секретный ключ, для подписи и проверки JWT-токенов;
Issuer – издатель, идентификатор сервиса, кто сгенерировал его;
Audience – получатель, кому предназначался.

```
17  builder.Services.AddAuthentication(options =>
18  {
19      options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
20      options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
21  })
22  .AddJwtBearer(options =>
23  {
24      options.TokenValidationParameters = new TokenValidationParameters
25      {
26          ValidateIssuerSigningKey = true,
27          IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(builder.Configuration["Jwt:Key"])),
28          ValidateIssuer = true,
29          ValidateAudience = true,
30          ValidIssuer = builder.Configuration["Jwt:Issuer"],
31          ValidAudience = builder.Configuration["Jwt:Audience"],
32          ValidateLifetime = true
33      };
34  });
```

```
93  app.UseAuthentication();
```

ASP.Net : JSON WEB TOKEN

Шаг 2. Создание сервиса для работы с JWT IJwtService;

```
using OpenDoor.Models;

namespace OpenDoor.Services
{
    public interface IJwtService
    {
        string GenerateToken(User user);
        bool ValidateToken(string token);
        Task<string> Authenticate(LoginModel loginModel);

    }
}
```

Ссылка на проект

ASP.Net : JSON WEB TOKEN

Шаг 3. Реализуем в сервисе JwtService все действия интерфейсов по работе с данными.

```
using Microsoft.IdentityModel.Tokens;
using OpenDoor.Models;
using OpenDoor.Services;
using System.IdentityModel.Tokens.Jwt;
using System.Security.Claims;
using System.Text;

public class JwtService : IJwtService
{
    private readonly IConfiguration _configuration;
    private readonly IUserService _userService; // Сервис для работы с пользователями

    public JwtService(IConfiguration configuration, IUserService userService)
    {
        _configuration = configuration;
        _userService = userService;  // Инициализируем сервис работы с пользователями
    }
```

Ссылка на проект

ASP.Net : JSON WEB TOKEN

Шаг 4. Реализовал логику авторизации на отдельном контроллере AuthController ;

```
using Microsoft.AspNetCore.Mvc;
using OpenDoor.Services;
using System.Threading.Tasks;

[Route("api/[controller]")]
[ApiController]
public class AuthController : ControllerBase
{
    private readonly IJwtService _jwtService;

    public AuthController(IJwtService jwtService)
    {
        _jwtService = jwtService;
    }

    // POST: api/Auth/login
    [HttpPost("login")]
    public async Task<IActionResult> Login([FromBody] LoginModel loginModel)
    {
```

Ссылка на проект

Тестирование REST

POSTMAN

— мощный инструмент, который используется для работы с API на уровне интерфейсов прикладного программирования.

Ссылка на коллекцию POSTMAN

Тестирование REST

Что можно тестировать?

- Тестируй не только успешные запросы (200 OK), но и ошибки (400, 401, 404 и т.д.).
- Проверка времени выполнения
- Проверка структуры данных, все ключевые поля присутствуют и имеют правильный тип
- **Обработка динамических данных**, которые могут изменяться, используй динамическую обработку и проверки.

#OPENAPI

> Спецификация OpenAPI определяет стандарт независимого от языка описания API, который **позволяет людям и машинам понимать возможности службы без доступа к исходному коду**, документации или путем перехвата сетевого траффика.

#OPENAPI

OpenAPI — это описание методов API, для которых описываются:

- query-параметры;
- заголовки и авторизация;
- схема входных и выходных сообщений;
- и т.п.

#OPENAPI

```
{
  "openapi": "3.0.0",
  "info": {
    "title": "День открытых дверей ИПТИП",
    "version": "3.2.7",
    "description": "API для интернет-магазина комиксов"
  },
  "servers": [
    {
      "url": "https://opendoor.easy4.team/public",
      "description": "Основной сервер"
    }
  ],
  "paths": {
    "/comics": {
      "get": {
        "summary": "Получить все комиксы",
        "description": "Показать все комиксы",
        "responses": {
          "200": {
            "description": "Список комиксов",
```

...

Монолит или микросервисы?

НЕУДОБНЫЕ ВОПРОСЫ: разработка REST API

Макиевский Станислав Евгеньевич

Микросервисы. Просто и понятно?

НЕУДОБНЫЕ ВОПРОСЫ: разработка REST API

Макиевский Станислав Евгеньевич

Микросервисы.

При простое.

Сервис	Память, MB	CPU usage
Сервис 1	281	0-1
Сервис 2	290	0-1
Сервис 3	269	0-1
Сервис 4	296	0-2
Сервис 5	272	0-1
Сервис 6	259	0-1
Сервис 7	100	0-1
Сервис 8	281	4-8
Сервис 9	271	0-1
Сервис 10	270	0-1
Сервис 11	266	0-1
СУММА :	2855	4-19

Монолит.

При простое.

Сервис	Память, MB	CPU usage
Сервис	410	3-10
	x6.9 меньше	X2 меньше

Микросервисы.

При нагрузке.

Сервис	Память, MB	CPU usage
Сервис 1	345	10
Сервис 2	330	20
Сервис 3	315	1
Сервис 4	308	1
Сервис 5	375	107
Сервис 6	311	3
Сервис 7	307	10
Сервис 8	110	14
Сервис 9	375	11
Сервис 10	445	90
Сервис 11	430	80
СУММА :	3651	347

Монолит.

При нагрузке.

Сервис	Память, MB	CPU usage
Сервис	640	170
	x5.5 меньше	x4 меньше

Монолит.

При нагрузке.

Сервис	Память, MB	CPU usage
Сервис	640	170
	x5.5 меньше	x4 меньше

А как добиться результата???

Константы для идеальной работы:

- Мощность сервера (ЦПУ и ОЗУ);
- Пропускная способность сети;
- Количество доступных серверов или микросервисов;
- Количество подключений;
- Время отклика;

Формула расчета эффективности (с целью) применения SOLID:

$$E_{SOLID} = \frac{\overbrace{(P_{before} - P_{after})}^{\text{ОЗУ}} + \overbrace{(C_{before} - C_{after})}^{\text{ЦПУ}} + \overbrace{(N_{users_after} - N_{users_before})}^{\text{Прирост юзеров}}}{\text{Дни для оптимизации } T_{opt}} \times (1 - B(E, m))$$

Формула Эрланга

$$B(E, m) = \frac{\frac{E^m}{m!}}{\sum_{i=0}^{m} \frac{E^i}{i!}}$$

- рассчитывает вероятность отказа в обслуживании. Это доля запросов, которые система не сможет обслужить из-за нехватки ресурсов (например, недостаточного количества потоков или серверов).

$$\overbrace{(N_{users_after} - N_{users_before})}^{\text{Юзеры}}$$

- влияние улучшений (прирост) на способность системы обрабатывать большее количество пользователей

$$E_{SOLID} = 164.6$$

$$\frac{E^m}{m!} = \frac{10^5}{5!} = \frac{100000}{120} = 833.33$$

$$\sum_{i=0}^{m} \frac{E^i}{i!} = \frac{10^0}{0!} + \frac{10^1}{1!} + \frac{10^2}{2!} + \frac{10^3}{3!} + \frac{10^4}{4!} + \frac{10^5}{5!} = 1477.67$$

$$B(E, m) = \frac{833.33}{1477.67} = 0.564$$

А как добиться результата???

Формула расчета эффективности (с целью) применения SOLID:

$$E_{SOLID} = \frac{\overset{\text{ОЗУ}}{(3500 - 2855)} + \overset{\text{ЦПУ}}{(20 - 19)} + \overset{\text{Юзеры}}{(2000 - 1000)}}{\underset{\text{Дни для оптимизации}}{10}} \times (1 - B(E, m))$$

Формула Эрланга

$$E_{SOLID} = 164.6$$

$$\frac{E^m}{m!} = \frac{10^5}{5!} = \frac{100000}{120} = 833.33$$

$$\sum_{i=0}^{m} \frac{E^i}{i!} = \frac{10^0}{0!} + \frac{10^1}{1!} + \frac{10^2}{2!} + \frac{10^3}{3!} + \frac{10^4}{4!} + \frac{10^5}{5!} = 1477.67$$

$$= 1477.67$$

$$B(E, m) = \frac{833.33}{1477.67} = \boxed{0.564}$$

Нужно улучшить

$$E_{SOLID} = 164.6 \times (1 - 0.564) = 164.6 \times 0.436 = 71.77$$

164.6 – единицы улучшения состояния системы по отношению к изначальным.
1 - это идеальное состояние, чтобы найти вероятность, что система справляется с нагрузкой.
71.77 % – единиц процента, насколько хорошо повлиял SOLID на разрабатываемый проект.
0,564 (56.4 %) – вероятность отказа, **относительно хороший показатель**, что при текущей нагрузке 40% система обрабатывает корректно.

Выводы:

- REST – это набор рекомендаций, которые помогут построить выразительный и расширяемый API.
- Представления не обязательно должны соответствовать 1 к 1 доменным сущностям.
- Каждый ресурс на сервере имеет свой неизменяемый идентификатор, операции над сущностью выполняются в привязке этому идентификатору.
- API строится вокруг HTTP verbs, CRUD операции ложатся на них 1 к 1, иные операции выполняются через POST (или PUT) методы.

В заключении:

«Человек молод до тех пор, пока делает глупости».

Пётр Леонидович Капица,

советский физик, инженер и инноватор, Нобелевский лауреат (1978)

Всем спасибо!