

Лабораторная работа №3

REST API

Теоретическая часть

API Design Guide

Общие правила

1. Для реализации API используется архитектурный стиль [REST](#).
2. Форматом передачи данных ДОЛЖЕН быть json.
3. Название ресурса в эндпоинте ДОЛЖНО быть во множественном числе (POST **/v1/users**, а не POST **/v1/user**), за исключением тех случаев когда ресурс может существовать только в единственном числе (/v1/profile).
4. Название ресурса в эндпоинте ДОЛЖНО быть в **kebab-case**.
5. Параметры в query и поля в body ДОЛЖНЫ быть в **snake_case**.
6. Идентификатор версии API всегда ДОЛЖЕН присутствовать в урле, например POST /v1/users.
7. Несуществующие страницы API ДОЛЖНЫ отдавать 404 ошибку и json ответ соответствующий формату описанному в разделе “Формат ответа”.
8. Все эндпоинты API ДОЛЖНЫ быть задокументированы через [Open Api Specification 3.0](#).
9. Документация ДОЛЖНА быть доступна через браузер используя [Swagger UI](#).
10. API СЛЕДУЕТ реализовывать используя **Design-first** подход.

Формат полей

1. Все целочисленные идентификаторы сущностей ДОЛЖНЫ иметь тип **integer**.
2. Datetime поля должны передаваться строкой в формате ISO-8601 в UTC.
Например "updated_at": "2020-01-01T15:47:21.000000Z"
В OpenApi такое поле описывается как type: string, format: date-time
3. Даты (без времени) должны передаваться строкой в формате [ISO-8601 full date](#)
Например "birthday": "1990-01-25"
В OpenApi такое поле описывается как type: string, format: date
4. Цены должны передаваться в копейках, с типом integer

Формат ответа

Тело ответа ДОЛЖНО содержать только следующие поля:

data – основной объект ответа, может иметь следующие типы:

- null;
- object – в случае, если запрашивается одна сущность, например, запрос по id;
- array – в случае, если запрашивается список сущностей, каждый элемент массива представляет собой отдельную конкретную сущность.

errors – необязательный массив ошибок запроса, каждый элемент в массиве содержит:

- code - обязательный строковый код ошибки, скорее всего берется из фиксированного списка. В документации к api ДОЛЖЕН присутствовать enum, в котором перечислены все коды ошибок;
- message - обязательное строковое описание ошибки;
- meta - опциональный объект с дополнительными метаданными ошибки.

meta – необязательный объект с дополнительной информацией о запросе, например для передачи отладочной информации или информации для пагинации. Каждый компонент, который хочет добавить данные в этот объект, должен добавлять их через дополнительное поле, например: meta.pagination.* В корень объекта meta нельзя

добавлять информацию для исключения конфликта в названии данных разных компонентов.

Стандартные методы

Стандартные методы позволяют реализовать CRUD функциональность для ресурса, которой достаточно в большом числе случаев. API не обязано реализовывать все стандартные методы для всех ресурсов.

Метод	HTTP реализация	Цель
Get	GET <resource-url>/<id>	Получение объекта по id
Create	POST <resource-url>	Создание объекта
Replace	PUT <resource-url>/<id>	Обновление всех полей объекта
Patch	PATCH <resource-url>/<id>	Обновление указанных полей объекта
Delete	DELETE <resource-url>/<id>	Удаление объекта
Search	POST <resource-url>:search	Поиск объектов по фильтрам
SearchOne	POST <resource-url>:search-one	Быстрый поиск одного объекта по фильтру

где <resource-url>, например, /api/v1/users.

Стандартные методы: Get

Формат запроса:

GET /api/v1/users/17?include=addresses,loyalty_cards

параметр **include** является опциональным

Формат ответа:

```
{
  "data": {
    "id": 17,
    "name": "John Doe",
    "last_login_at": "2020-01-01T15:47:21.000000Z",
    "addresses": [
      {
        "id": 1,
        ...
      }
    ],
    "loyalty_cards": [
      {
        "id": 1,
        ...
      }
    ],
  }
}
```

Стандартные методы: Create

Формат запроса:

POST /api/v1/users

```
{
  "name": "John Doe",
  "last_login_at": "2020-01-01T15:47:21.000000Z"
}
```

Формат ответа:

```
{
  "data": {
    "id": 1006779,
    "name": "John Doe",
    "last_login_at": "2020-01-01T15:47:21.000000Z"
  }
}
```

Объект в data полностью соответствует объекту в методе Get без параметров.

Стандартные методы: Replace

Формат запроса:

PUT /api/v1/users/1006779

```
{  
  "name": "John Doe",  
  "last_login_at": null  
}
```

Запрос ДОЛЖЕН являться идемпотентным. Таким образом в data могут отсутствовать только необязательные поля, которые в этом случае будут сброшены до значения по умолчанию. Поле id из data ДОЛЖНО игнорироваться.

Формат ответа:

```
{  
  "data": {  
    "id": 1006779,  
    "name": "John Doe",  
    "last_login_at": null  
  }  
}
```

Объект в data полностью соответствует объекту в методе Get без параметров.

Стандартные методы: Patch

Формат запроса:

PATCH /api/v1/users/1006779

```
{  
  "name": "John Doe"  
}
```

Только поля указанные в data ДОЛЖНЫ быть изменены.
Поле id из data ДОЛЖНО игнорироваться.

Формат ответа:

```
{
  "data": {
    "id": 1006779,
    "name": "John Doe",
    "last_login_at": "2020-01-01T15:47:21.000000Z"
  }
}
```

Объект в data полностью соответствует объекту в методе Get без параметров.

Стандартные методы: Delete

Формат запроса:

DELETE /api/v1/users/1006779

Формат ответа:

```
{
  "data": null
}
```

Если объект уже был удален до этого это не должно приводить к 404 ошибке. Аналогичным образом должны вести себя дополнительные “удаляющие методы”. Например, удаление файла привязанного к объекту.

Стандартные методы: Search

Формат запроса:

POST /api/v1/users:search

```
{
  "sort": [
    "-last_login_at", // по убыванию last_login_at
    "id"
  ],
  "filter": {
    "id": [
      12125,
      1006779
    ],
    "last_login_gte": "2020-01-01T15:47:21.000000Z"
  },
  "include": [
    "roles"
  ],
  "pagination": {
    ...
  }
}
```

Все поля являются опциональными.

Подмножество экземпляров ресурса отфильтрованных сложным образом можно также выносить в отдельный ресурс и отдельный endpoint для него соответственно.

Запрос использует POST чтобы избежать некоторых ограничений, связанных с реализацией параметров в GET запросах как в самом протоколе HTTP, так и в OpenApi генераторах.

Стандартные методы: SearchOne

Формат запроса: полностью совпадает с форматом метода Search

Формат ответа:

```
{
  "data": {
    "id": 1006779,
    "name": "John Doe",
    "last_login_at": "2020-01-01T15:47:21.000000Z"
  }
}
```

Объект в data полностью соответствует объекту в методе Get без параметров.

Дополнительные методы

При проектировании API СЛЕДУЕТ стараться ограничиваться стандартными методами, которых обычно достаточно для большинства задач, однако при необходимости можно добавлять и дополнительные.

Пользовательские методы всегда ДОЛЖНЫ использовать POST и образуются добавлением названия метода к <resource url> через двоеточие.

Форматы тела запроса и ответа не регламентируются за исключением общих правил для формата ответа, описанных в начале этого руководства.

В любом случае СЛЕДУЕТ проектировать их похожими на стандартные методы.

Пример запроса ко всему ресурсу:

POST /api/v1/users:mass-delete

```
{
  "id": [1, 2, 3]
}
```

Пример ответа:

```
{
  "data": null
}
```


HTTP status codes

API ДОЛЖНО использовать только следующие HTTP коды ответа:

- **200 OK** во всех ситуациях, когда запрос не заканчивается ошибкой
- **201 Created** если его генерирует ваш фреймворк автоматически. В противном случае РЕКОМЕНДУЕТСЯ использовать 200
- **401 Unauthorized**
- **403 Forbidden**
- **404 Not found** при запросе несуществующего ресурса или экземпляра ресурса
- **400 Bad Request** при любых других ошибках, причиной которых является клиент
- **500 Internal Server Error** – при любой ошибке приложения, причиной которой является проблемы в самом приложении, а не в запросе.

В документации API у каждого эндпоинта ДОЛЖНЫ быть указаны все возможные коды ответа.

Практическая часть

Задание 1

Напишите спецификацию, используя OpenAPI и Swagger UI.

Для вывода Swagger UI используйте пакет [ensi-platform/laravel-serve-swagger](#).

В спецификации должны присутствовать следующие стандартные методы: Get, Create, Replace, Patch, Delete.

Документация должна быть доступна по ссылке: /docs/swagger

Используйте материалы лекции №9 и официальные документации (с примерами) [openapi](#) и [swagger-ui](#).

Перед реализацией задания предложите и согласуйте с преподавателем предметную область, доменный слой и сами домены.

Задание 2

Используя доменно-ориентированный подход, разработайте небольшое REST API на Laravel.

Требования:

1. Вся логика действий (action) в домене должна быть реализована через классы Action (1 действие – 1 класс).
Пример класса: [PatchUserAction](#)
2. Для входных данных должны быть реализованы классы с запросами (Request) с валидацией.
Пример класса: [PatchUserRequest](#)
3. Для выходных данных должны быть реализованы классы с ресурсами (Resource).
Пример класса: [UsersResource](#)

Пример реализации контроллера [UsersController](#).

Используйте материалы лекций №6 – 8, №11.

Задание 3

Покрыть все эндпоинты API (из задания №1) автотестами. Продемонстрируйте значение Code Coverage. Используйте материалы лекции №10 и официальную документацию фреймворка **PEST**.

Примеры

Пример реализации PatchUserAction

```
<?php
namespace App\Domain\Users\Actions;
use App\Domain\Users\Models\User;
class PatchUserAction
{
    public function execute(int $userId, array $fields): User
    {
        $user = User::findOrFail($userId);
        $user->update($fields);
        return $user;
    }
}
```

Пример реализации PatchUserRequest

```
<?php
namespace App\Http\ApiV1\Modules\Users\Requests;
use App\Http\ApiV1\Support\Requests\BaseFormRequest;
use Illuminate\Validation\Rule;
class PatchUserRequest extends BaseFormRequest
{
    public function rules(): array
    {
        $id = (int) $this->route('userId');
        return [
            'login' => [Rule::unique('users')->ignore($id)],
            'active' => ['boolean'],
            'password' => ['nullable'],
        ];
    }
}
```

Пример реализации UsersResource

```
<?php

namespace App\Http\ApiV1\Modules\Users\Resources;

use App\Http\ApiV1\Support\Resources\BaseJsonResource;

/** @mixin \App\Domain\Users\Models\User */

class UsersResource extends BaseJsonResource
{
    public function toArray($request): array
    {
        return [
            'id' => $this->id,
            'login' => $this->login,
            'active' => $this->active,
            'created_at' => $this->created_at,
            'updated_at' => $this->updated_at,
        ];
    }
}
```

Пример реализации контроллера UsersController

```
<?php

namespace App\Http\ApiV1\Modules\Users\Controllers;

use App\Domain\Users\Actions\CreateUserAction;
use App\Domain\Users\Actions\DeleteUserAction;
use App\Domain\Users\Actions\PatchUserAction;
use App\Http\ApiV1\Modules\Users\Queries\UsersQuery;
use App\Http\ApiV1\Modules\Users\Requests\CreateUserRequest;
use App\Http\ApiV1\Modules\Users\Requests\PatchUserRequest;
use App\Http\ApiV1\Modules\Users\Resources\UsersResource;
use App\Http\ApiV1\Support\Resources\EmptyResource;
use Illuminate\Http\Request;

class UsersController
{
    public function create(CreateUserRequest $request,
                          CreateUserAction $action
    ){
        return new UsersResource($action->execute($request->validated()));
    }

    public function patch(int $userId,
                          PatchUserRequest $request,
                          PatchUserAction $action
    ) {
        return new UsersResource(
            $action->execute($userId, $request->validated())
        );
    }

    public function delete(int $userId, DeleteUserAction $action)
    {
        $action->execute($userId);
        return new EmptyResource();
    }

    public function get(int $userId, UsersQuery $query)
    {
        return new UsersResource($query->findOrFail($userId));
    }
}
```