Съдържание

[1. Увод 3](#_Toc196415319)

[2. Същност на REST архитектурата 4](#_Toc196415320)

[3. Сфера на приложение на REST 5](#_Toc196415321)

[4. Основни HTTP методи в REST API 7](#_Toc196415322)

[5. Дизайн на REST API 8](#_Toc196415323)

[6. Сигурност в REST API 8](#_Toc196415324)

[7. Потенциални проблеми и ограничения при използване на REST API 9](#_Toc196415325)

[8. Тестове на REST API 10](#_Toc196415326)

[9. Описание на разработваното приложение 11](#_Toc196415327)

[10. Използвани технологии и инструменти 12](#_Toc196415328)

[10.1. Python 12](#_Toc196415329)

[10.2. Django 13](#_Toc196415330)

[10.3. Django Rest Framework (DRF) 13](#_Toc196415331)

[11. Архитектура на приложението 14](#_Toc196415332)

[11.1. Модели на данни 14](#_Toc196415333)

[11.2. Схема на база данни (описание) 15](#_Toc196415334)

[11.3. Взаимодействие между компонентите 16](#_Toc196415335)

[12. Реализация на REST API 17](#_Toc196415336)

[12.1. Дефиниране на сериализатори 17](#_Toc196415337)

[12.2. Изгледи (Views) 18](#_Toc196415338)

[12.3. Маршрутизиране на заявки (urls.py) 19](#_Toc196415339)

[12.4. Връзка между компонентите 19](#_Toc196415340)

[12.5. Примерни заявки и отговори 20](#_Toc196415341)

[12.6. Коментар и предимства на реализираната архитектура 20](#_Toc196415342)

[13. Потребителски интерфейс 21](#_Toc196415343)

[13.1. Как е реализиран 21](#_Toc196415344)

[13.2. Основни моменти: 21](#_Toc196415345)

[13.3. Примерен код на потребителския интерфейс 22](#_Toc196415346)

[13.4. Причини за добавянето на интерфейса 23](#_Toc196415347)

[13.5. Примерен вид на интерфейса (screenshot) 24](#_Toc196415348)

[13.6. Примери за работа с приложението 24](#_Toc196415349)

[13.6.1 Примерни заявки и отговори 24](#_Toc196415350)

[13.6.2 Работа с вградения уеб интерфейс на DRF 26](#_Toc196415351)

[14. Тестове и резултати 27](#_Toc196415352)

[14.1. Проведени тестове 27](#_Toc196415353)

[14.2. Констатации 28](#_Toc196415354)

[15. ЗАКЛЮЧЕНИЕ 28](#_Toc196415355)

[16. Литература: 30](#_Toc196415356)

# Увод

В съвременния свят обменът на данни между различни софтуерни системи и платформи е от ключово значение за развитието на дигиталното общество. Независимо дали става дума за уеб приложения, мобилни апликации или интеграция между различни услуги, необходимостта от стандартизиран, сигурен и лесен за използване интерфейс за комуникация е безспорна. Именно тук се появява REST архитектурата – един от най-популярните и широко използвани подходи за изграждане на уеб услуги и програмни интерфейси (API).

Настоящият дипломен проект има за цел да представи на практика основните принципи, добри практики и реализация на REST API чрез съвременните инструменти Python, Django и Django Rest Framework (DRF). За целта е разработено демонстрационно приложение за управление на обучителни курсове за деца, което включва възможност за създаване, редакция, преглед и изтриване на курсове, както и за записване на деца за избран курс. Проектът е структуриран така, че да бъде максимално ясен, четим и подходящ за учебни цели, като всяка част от архитектурата – модели, сериализатори, изгледи, маршрутизация и потребителски интерфейс – е реализирана по начин, който следва съвременните стандарти и улеснява бъдещо разширяване.

Една от основните цели на проекта е да се демонстрира не само как се изгражда REST API на теория, но и как то може да бъде използвано на практика – както от други софтуерни системи, така и от реални потребители. За тази цел към приложението е добавен и примерен потребителски интерфейс, реализиран с помощта на Vue.js и Axios, който работи напълно независимо от бекенда и комуникира с него чрез публичното API. Това позволява да се тества и визуализира целият процес – от въвеждане на данни от потребителя, през обработка на заявките на сървъра, до съхранение и извличане на информация от базата данни.

В хода на разработката са разгледани и практически аспекти като сигурност, CORS ограничения, валидация на данните и автоматично генериране на документация и тестови интерфейс чрез DRF. Проведени са тестове на всички основни функционалности, както през вградения уеб интерфейс на DRF, така и през външния фронтенд, което гарантира надеждността и коректната работа на системата.

Проектът има за цел не само да покаже техническите възможности на REST API и съвременните инструменти за неговата реализация, но и да послужи като учебен модел за бъдещи разработчици, които искат да се запознаят с добрите практики при изграждане на уеб услуги и интеграционни решения.

# Същност на REST архитектурата

REST, или Representational State Transfer, е архитектурен стил за изграждане на разпределени програмни системи и най-вече за уеб услуги, които комуникират по HTTP. REST е дефиниран от Рой Филдинг през 2000 година в неговата докторска дисертация. Основната идея е организиране на комуникацията между компютърни системи така, че всяка информация да бъде представена като ресурс, достъпен чрез уникален адрес (URL).

Исторически погледнато, концепцията за REST възниква като следствие от нуждата за по-опростени, мащабируеми и устойчиви начини за обмен на данни между различни уеб приложения. Преди REST най-популярен е подходът със SOAP (Simple Object Access Protocol), който обаче често се оказва твърде сложен и тежък за внедряване. Филдинг въвежда REST като по-лекия, гъвкав и разбираем вариант за уеб услуги.

Основните принципи на REST са няколко. На първо място стои „статуса“ (stateless) — всяка заявка към сървъра трябва да съдържа цялата необходима информация и не разчита на стари сесии. Второ, данните и ресурсите са достъпни чрез „унифициран интерфейс“ – това означава, че комуникацията между клиент и сървър винаги използва стандартни HTTP методи и адреси. Трето, REST се основава на слоеве – клиентът не трябва да знае подробности за това как точно сървърът обработва заявките или колко сървъра участват във веригата.

Като резултат, REST архитектурата дава възможност за създаване на мащабируеми, лесни за поддръжка и развити системи, които работят бързо и сигурно в интернет среда.

# Сфера на приложение на REST

REST архитектурата е изключително разпространена и се използва навсякъде, където има нужда от обмен на данни между различни приложения по интернет. REST се прилага при създаването на уеб и мобилни приложения, които работят с бази данни, при Интернет на нещата (IoT) и при интеграция между различни софтуерни системи.

Една от причините REST да е толкова широко използван е неговата простота и съвместимост със съвременните уеб технологии. Докато архитектури като SOAP изискват специфични структури и протоколи, REST използва стандартни HTTP методи като GET, POST, PUT и DELETE, което го прави удобен за разработчиците и лесен за тестване.

Интересно е да се разгледат приликите и разликите между REST и други популярни архитектурни стилове за изграждане на уеб услуги и обмен на данни. Един от основните конкуренти на REST е **SOAP** (Simple Object Access Protocol). SOAP е по-стар стандарт, който поставя акцент върху строго определени формати на съобщенията, базирани на XML, както и на сигурността и валидирането на информацията. Точно поради изискванията за формалност, висока сигурност и надеждност, SOAP често остава предпочитан избор в големи банки, държавни институции и корпоративни системи, където се обработват критични данни или се изисква сложна авторизация и интеграция с наследени системи. Въпреки това, SOAP услугите са доста тромави по отношение на разработка и изискват повече време за настройка и поддръжка, докато REST се възприема като по-лесен и бърз за използване, особено когато е нужно да се достави уеб услуга до широк кръг устройства и приложения.

В последните години все по-голяма популярност набира и **GraphQL** – нов подход, разработен от Facebook, който позволява на клиентските приложения сами да избират какви данни искат да получат. За разлика от REST, където когато се поиска даден ресурс, се връща винаги предварително определена структура от данни, при GraphQL клиентът може да „запита“ само за нужните му части от информацията. Това значително намалява ненужния трафик между клиент и сървър, особено при мобилни апликации или сложни уеб интерфейси. От друга страна, внедряването на GraphQL е по-сложно и изисква допълнителен слой от логика, затова REST се използва масово там, където бързината и простотата са от ключово значение.

Въпреки наличието на алтернативи, REST остава **основен стандарт** за изграждане и интеграция на уеб услуги – особено при публични API, които се използват от множество външни разработчици и различни системи. Причината за това е, че REST работи директно върху стандартния HTTP протокол, разбира се лесно от всички езикови платформи и не изисква специални библиотеки при интеграция. REST е особено подходящ и за приложения, които трябва да оперират в реално време и да обслужват голям брой потребители с минимални закъснения.

Примерите за реални, широко използвани приложения, изградени с REST, са безброй. Социалните мрежи, като **Facebook, Instagram и Twitter**, предоставят публични REST API, чрез които се създават външни приложения, анализират се данни, публикуват се статуси или снимки и др. Също така платформи за електронна търговия, като **Amazon** и **eBay**, използват REST API-та за управление на продукти, поръчки и потребители от хиляди външни интегратори.

Една цяла категория са приложенията за карти и локационни услуги – **Google Maps API** и **OpenStreetMap** REST API дават възможност на уеб и мобилни приложения да питат за адреси, да търсят по координати или да визуализират маршрути в реално време.

Разплащателните системи (например **PayPal**, **Stripe** и **Revolut**) също предоставят REST API, които се ползват от онлайн магазини, мобилни портфейли и други услуги за изпращане и приемане на пари.

Голяма част от **IoT устройствата** (Интернет на нещата) – като домашни термостати, алармени системи, умни електромери и сензори – използват REST API за изпращане на отчитания към облака или за получаване на команди и настройки от отдалечени сървъри.

Освен всичко това, REST API e гръбнакът при интеграцията между множество ежедневни приложения – системи за резервации (хотели, събития, коуъркинг пространства), услуги за изпращане на SMS или имейли, облачни хартиени документации (като Google Drive или Dropbox), а също така и най-обикновени мобилни приложения за напомняния, бележки и други.

Накратко, REST се използва толкова масово, че повечето програмисти, още с началото на обучението си по уеб развитие, се сблъскват с него и го използват при първите си проекти – както в училище, така и в професионална среда.

# Основни HTTP методи в REST API

REST API използва основните HTTP методи за управление на ресурси, като прави комуникацията между клиент и сървър ясна и предвидима. Най-важните методи са:

* **GET** се използва, за да се поиска информация от сървъра. Например, ако имаме ресурс „курсове“, един GET към /api/courses/ ще върне списък с всички налични курсове.
* **POST** се използва за създаване на нов ресурс. Ако изпратим POST с данни за нов курс към /api/courses/, на сървъра ще бъде създаден нов запис.
* **PUT** служи за обновяване на съществуващ ресурс. Например, PUT към /api/courses/1/ с нови данни ще промени информацията за курса с номер 1.
* **DELETE** премахва даден ресурс. Когато изпратим DELETE заявка към /api/courses/1/, този курс ще бъде изтрит.

Кодът на една такава заявка с помощта на библиотеката requests в Python може да изглежда така:

# Пример за POST заявка

import requests

data = {"name": "Python за начинаещи", "age\_group": "12-15"}

response = requests.post("http://localhost:8000/api/courses/", data=data)

print(response.status\_code)

Съществуват и други методи. **PATCH** често се прилага за частично обновяване на существо ресурс (например само името на курса), а **OPTIONS** се използва за получаване на информация какви методи са позволени върху даден ресурс.

Тези методи осигуряват яснота в комуникацията между клиент и сървър и позволяват разнообразни операции с ресурсите по единна логика.

# Дизайн на REST API

Когато се планира REST API, е важно да се следват добри практики, така че API-то да е интуитивно, сигурно и лесно за ползване и разширяване. Добре е ресурсите да се именуват ясно и в множествено число – например /api/courses/ за всички курсове и /api/courses/1/ за отделен курс.

Друго важно нещо е използването на подходящи HTTP статус кодове. Например, при успешна заявка за получаване на ресурс сървърът връща код **200 OK**. Ако е създаден нов ресурс с POST-заявка, се връща **201 Created**, а ако има грешка при заявката – **400 Bad Request** или **404 Not Found**.

Добрата документация на API-то е ключова – тя улеснява другите програмисти да разбират как да използват услугата. За документиране често се използват инструменти като Swagger и OpenAPI, които автоматично описват ресурсите и методите.

Пример за добре оформен път в REST API:

GET /api/courses/

POST /api/courses/

GET /api/courses/1/

PUT /api/courses/1/

DELETE /api/courses/1/

Така всички операции по отношение на даден ресурс са подредени логично и разбираемо.

# Сигурност в REST API

Сигурността заема основно място при разработването на REST API, понеже често се обменят чувствителни данни между клиент и сървър. Основният подход е използването на автентикация и авторизация. Най-често се среща „token-based authentication“, където след успешно влизане на потребител, сървърът издава специален токен, който потребителят добавя към всяка бъдеща заявка.

Авторизацията определя какви действия може да извършва даденият потребител, напр. дали има право да изтрива или само да чете ресурси. Задължително е използването на сигурна връзка чрез HTTPS, така че данните да не могат да бъдат прихванати от трети лица.

Често срещани уязвимости са „SQL injection“ и „Cross-site scripting (XSS)“. За да се предпазим, трябва да валидираме входящите данни и да използваме защитени методи за достъп до базата данни.

# Потенциални проблеми и ограничения при използване на REST API

Въпреки многото предимства на REST API, при реална разработка и интеграция понякога се появяват специфични трудности, които могат да затруднят разработчиците. Един от най-често срещаните проблеми особено при уеб приложения е така нареченият **CORS** (Cross-Origin Resource Sharing).

**CORS** е механизъм в браузъра, който ограничава достъпа на JavaScript приложения (работещи на даден домейн) до ресурси и API-та, които се намират на друг домейн. По подразбиране, ако се направи заявка към REST API от уеб страница, заредена от различен адрес (например front-end приложението е на [http://localhost:3000](http://localhost:3000/), а API-то е на [http://localhost:8000](http://localhost:8000/)), браузърът блокира заявката и връща грешка. Това се прави от съображения за сигурност, така че злонамерен код да не може да прави неразрешени заявки към чужди сървъри.

Решаването на CORS проблемите става чрез добавяне на специални HTTP хедъри от страна на сървъра, които изрично позволяват заявките от определени домейни. В Django, например, това може да се направи чрез инсталиране на разширението **django-cors-headers** и добавяне на няколко настройки в settings.py:

# Примерен код за разрешаване на CORS в Django

INSTALLED\_APPS = [

...,

'corsheaders',

...,

]

MIDDLEWARE = [

'corsheaders.middleware.CorsMiddleware',

...,

]

CORS\_ALLOW\_ALL\_ORIGINS = True # За развитие; в продукция добавят се конкретни домейни

# CORS\_ALLOWED\_ORIGINS = [

# "http://localhost:3000",

# ]

Освен CORS ограниченията, понякога REST API може да създава и други трудности:

* Не всички HTTP клиенти обработват правилно статус кодовете и грешките, което води до объркване при интеграция;
* Ограничения в скоростта и честотата на заявките (rate limiting) – понякога сървърът, за защита, ограничава прекалено много заявките;
* Версиониране на API – при промени по API-то може да стане нелесно поддържането на старата функционалност за стари клиенти;
* Разлика между различни реализации – не всички REST API-та спазват едни и същи стандарти и добри практики.

В заключение, познаването на тези ограничения е много важно, за да се проектира и използва REST API без излишни проблеми и с добра сигурност и съвместимост.

# Тестове на REST API

Тестването е важна част от разработката на REST API. Основните видове тестове са unit тестове (тестове на отделни функции), integration тестове (тестване на работата на API-то като цяло) и тестове за сигурност.

Популярни инструменти за тестване на REST API са **Postman** и **Swagger**. В Postman можем лесно да изпращаме заявки като GET, POST, PUT или DELETE и да видим какво връща сървърът. Това спестява много време на разработчика, защото позволява бързо откриване на грешки.

Swagger също е много удобен, тъй като не само документира API-то, а и дава възможност заявките да се пробват директно от браузъра. Освен това много платформи като Django REST Framework имат вградена поддръжка за автоматично генериране на документация и тестване през уеб интерфейс.

# Описание на разработваното приложение

Разработеното от мен приложение представлява малка демонстративна система за управление на резервации за обучителни курсове за деца, реализирана чрез собствено REST API. Основната цел на проекта е да онагледя принципите и добрите практики при създаване на REST API, а не да реализирам завършен продукт, готов за внедряване в реална организация. Приложението е създадено така, че всеки основен елемент да бъде ясен, лесно проследим и да служи като пример за бъдещи проекти и учебна работа.

Чрез този пример постигам две основни задачи: първо, научавам се как се имплементират принципите на REST на практика чрез реален код; второ, създавам лесноразбираемо API, което може да се използва от други ученици за упражнение или като основа за разширяване с нови функции.

В системата са реализирани следните основни функционалности:

* Възможност за създаване, преглед, редакция и изтриване на обучителни курсове
* Възможност за записване (резервация) на дете за избран курс
* Получаване на списък с всички курсове, както и списък с всички записвания по курсове
* Ясно разделяне на моделите (данните), сериализаторите (конвертиране на данни до JSON) и изгледите (организиране на логиката при достъпа до ресурси)
* Примерен прост потребителски интерфейс, който демонстрира как може да се използва API-то през браузър

Избрах да се фокусирам само върху най-важните аспекти на едно REST API — модели, сериализатори, изгледи, маршрути и работа с потребителски интерфейс — за да не натоварвам проекта с излишни детайли и за да е максимално ясен и опростен за разбиране. Всеки елемент е реализиран последователно, с подчертаване на най-ключовите техники при работа с Django Rest Framework.

Проектът има ясно структурирани папки и файлове, следвайки добрите практики в съвременното програмиране. Това дава възможност на други ученици или начинаещи разработчици лесно да стартират приложението и да разберат основните принципи на работа с REST API в Python среда. Освен това структурирането на кода и логическото разделяне на различните части помагат при по-сложни бъдещи разширения.

Целта не е приложение с много функции, а чист и модерен код, който да служи като модел и основа за учене, тестване и надграждане. По този начин моят проект е най-вече учебно пособие, а не готов за внедряване продукт — подход, който според мен е най-полезен в една училищна и образователна среда.

# Използвани технологии и инструменти

При реализирането на проекта избрах да използвам следните съвременни технологии и инструменти, които работят добре за изграждането на REST API и са подходящи за начинаещи и напреднали ученици:

## Python

Python е един от най-популярните програмни езици в света днес. Причините да го избера са няколко. На първо място, синтаксисът му е лесен за разбиране и писане, дори за хора без дълъг опит. Python има огромна общност от програмисти, което означава, че информация, помощ и готови библиотеки винаги са достъпни.

Python е изключително подходящ за сървърно програмиране, разработване на уеб приложения и, разбира се, за работа с бази данни. В образователна среда, той е перфектен избор за запознаване с концепциите на обектно-ориентираното програмиране и за изграждане на реални проекти.

## Django

Django е мощен уеб фреймуърк за Python, предназначен за бърза разработка и добри практики при изграждането на уеб приложения. Django разполага със собствен ORM (Object-Relational Mapping), система за управление на шаблони, администраторска част и редица вградени функции за сигурност.

Изборът на Django в този проект се дължи на следните предимства:

* Позволява ясно и разделно изграждане на моделите (данните), изгледите (логиката), сериализаторите и маршрутите;
* Включва мощни вградени средства за администрация и лесна връзка с бази данни;
* Има солидна документация и голям брой готови разширения за стандартизирани задачи.

Една класическа Django структура съдържа папки за приложенията, настройки, шаблони и миграционни файлове. За нашето учебно приложение това дава голямо улеснение в организацията и последващата поддръжка.

## Django Rest Framework (DRF)

Django Rest Framework е разширение на Django, създадено специално за разработване на REST API. DRF улеснява описването и сериализацията на моделите, обработката на заявки от клиентите и връщането на отговори във формат JSON, подходящ за web и mobile приложения.

Основните предимства на DRF, които използвах в проекта, са:

* Автоматичното и ясно създаване на API интерфейс за ресурсите (курс, записване и т.н.);
* Механизъм за сериализация – превръщане на Python обекти в JSON и обратно;
* Възможност за бързо създаване на CRUD операции (create, read, update, delete) върху ресурсите;
* Вградена документация и тестване през уеб браузър;
* Механизми за сигурност и автентикация/авторизация, ако решим да ги добавим.

Пример за сериализатор в Django Rest Framework:

from rest\_framework import serializers

from .models import Course

class CourseSerializer(serializers.ModelSerializer):

class Meta:

model = Course

fields = '\_\_all\_\_'

DRF отлично се съчетава с всички останали части на Django и осигурява модел, който лесно може да бъде разширен или модифициран според нуждите на приложението.

Използването на тези инструменти ми даде възможност да реализирам REST API, което е едновременно практично, лесно за разбиране и подходящо като модел за обучение – точно както изисква целта на проекта. Всички избрани технологии се използват активно в професионалната практика, а добрият им дизайн улеснява поддръжката дори и за по-начинаещи програмисти.

# Архитектура на приложението

Архитектурата на приложението е създадена с цел да бъде максимално ясна, прегледна и в същото време разширяема. В основата на всяко добро приложение стои правилното моделиране на данните — затова започнах с определянето на двата най-важни модела, които отразяват основните обекти и тяхната връзка в реалния свят. Архитектурата следва класическия MVT (Model-View-Template) подход на Django, като при създаването на API разчитаме на моделите, сериализаторите, изгледите и маршрутизацията.

## Модели на данни

В моето приложение има два основни модела:

1. **Course (Курс)**

Курсът представлява образователна програма, по която децата могат да се обучават. Всеки курс има име, възрастова група, начало и край на курса, както и допълнителна информация за преподавателя и максимален брой участници. С това отразяваме основните характеристики на всяко учебно събитие.

class Course(models.Model):

name = models.CharField(max\_length=100)

age\_group = models.CharField(max\_length=20)

start\_date = models.DateField()

end\_date = models.DateField()

teacher = models.CharField(max\_length=100)

max\_students = models.PositiveIntegerField()

1. **Enrollment (Записване)**

Този модел отразява записването на дете за конкретен курс. Той съдържа информация за името на детето, класа, името на родителя, телефон за връзка, имейл и връзка към съответния курс. Връзката е реализирана чрез ForeignKey към модела Course, което означава, че всяко записване се отнася точно за един курс, а един курс може да има няколко записвания.

class Enrollment(models.Model):

child\_name = models.CharField(max\_length=50)

child\_grade = models.CharField(max\_length=20)

parent\_name = models.CharField(max\_length=100)

parent\_phone = models.CharField(max\_length=20)

parent\_email = models.EmailField()

course = models.ForeignKey(Course, related\_name='enrollments', on\_delete=models.CASCADE)

Тази структура позволява лесното разширяване на системата — може да се добавят нови свойства на курсовете, или записванията да се обогатят с още данни, ако потрябва.

## Схема на база данни (описание)

Може да си представим връзката между моделите така:

Course (Курс)

ID (PK)

|-- name

|-- age\_group

|-- start\_date

|-- end\_date

|-- teacher

|-- max\_students

| 1:M

|

Enrollment (Записване)

ID (PK)

|-- child\_name

|-- child\_grade

|-- parent\_name

|-- parent\_phone

|-- parent\_email

|-- course\_id (FK)

Всяко записване е директно свързано с един курс (чрез course\_id), което е практичен модел за всякакъв тип резервационна система. Тази структура елиминира дублиране на данни, тъй като детайлите за курса са само на едно място, а към всяко записване има препратка към точния курс.

## Взаимодействие между компонентите

Архитектурата на приложението е организирана така, че всеки компонент изпълнява конкретна и самостоятелна роля:

* **Модели**: Описват структурата на базата данни и връзките между курсове и записвания.
* **Сериализатори**: Отговарят за преобразуването на данните от/към JSON формат, така че да могат да се изпращат лесно по мрежата. Например, когато се прави заявка за всички записвания към даден курс, сериализаторът се грижи това да се представи подходящо пред клиента (например браузър или мобилно приложение).
* **Изгледи**: Реализират логиката при получаване и обработване на заявките. Пример — при заявка GET /courses/ се връща списък с всички курсове; при POST /enrollments/ се добавя ново записване.
* **Маршрутизация**: Свързва всички възможни URL пътища със съответните изгледи и контролира „кога, до кого и как“ да се подава бъдещата обработка на заявките.

Всяка част работи самостоятелно, но всички са тясно свързани — когато се създава ново записване, сериализаторът проверява изпратените от клиента данни, моделът съхранява данните в базата, а изгледът връща подходящ отговор.

Използването на Django и Django Rest Framework позволява да се поддържат ясни граници между всеки отделен слой на приложението. Например, ако искам да разширя логиката за курсовете, работя основно в модела и сериализатора за Course; ако трябва да добавя филтри или допълнително поведение при заявките, работя в изгледите.

Така изградената архитектура осигурява:

* Четлив и подреден код
* Лесна поддръжка и възможност за разширяване
* Работещ прототип, който отлично демонстрира принципите на работа с REST API
* Логическа основа, подходяща за използване като учебно пособие или за бъдещо развитие

Благодарение на модулната структура на Django и DRF, системата може лесно да се надгражда – например да се добави администраторски панел, допълнителна валидация на данните, удостоверяване и права на достъп, както и интеграция с други приложения или потребителски интерфейс.

# Реализация на REST API

Реализацията на REST API е централен аспект в този учебен проект. Тя осигурява мост между клиентските приложения и сървърната логика, като позволява лесно и стандартизирано извличане, създаване, редакция и изтриване на информация за курсове и записвания. Логиката е изградена с помощта на Django Rest Framework (DRF), който улеснява и ускорява целия процес чрез своите компоненти – сериализатори, изгледи и маршрутизация чрез Router.

## Дефиниране на сериализатори

Сериализаторите са ключът за преобразуване на Django моделите в JSON формат и обратно. Това е необходимо, за да може клиентът (например фронт-енд приложение, мобилен ап или дори Postman) да комуникира със сървъра по структуриран начин.

**serializers.py:**

from rest\_framework import serializers

from .models import Course, Registration

class CourseSerializer(serializers.ModelSerializer):

class Meta:

model = Course

fields = '\_\_all\_\_'

class RegistrationSerializer(serializers.ModelSerializer):

class Meta:

model = Registration

fields = '\_\_all\_\_'

В кода се вижда, че за всеки от двата модела – **Course** и **Registration** – има отделен сериализатор, базиран на ModelSerializer. Така се серилизират и десерилизират всички полета, описани по моделите. Това прави възможно автоматичното създаване на, преглед на, промяна и изтриване на данните, без да е необходимо да се описват ръчно всяко едно поле.

## Изгледи (Views)

Изгледите (views) управляват логиката на получаваните заявки към API-то. Основното предимство тук е използването на **ViewSets**, което съществено опростява кода и дава предимство при маршрутизацията с DRF Router.

**views.py:**

from rest\_framework import viewsets

from .models import Course, Registration

from .serializers import CourseSerializer, RegistrationSerializer

class CourseViewSet(viewsets.ModelViewSet):

queryset = Course.objects.all()

serializer\_class = CourseSerializer

class RegistrationViewSet(viewsets.ModelViewSet):

queryset = Registration.objects.all()

serializer\_class = RegistrationSerializer

Всяка от тези две ViewSet класове наследява от ModelViewSet – това осигурява пълния набор от CRUD операции (Create, Read, Update, Delete) само с няколко реда код. DRF автоматично предоставя съответните методи за обработка на GET, POST, PUT, PATCH и DELETE заявки.

Например:

* GET към /courses/ връща списък с всички курсове.
* POST към /courses/ създава нов курс.
* GET към /enrollments/2/ връща записването с id=2.
* PUT/PATCH/DELETE към /enrollments/2/ – редактира или изтрива това записване.

## Маршрутизиране на заявки (urls.py)

При DRF, най-ефективен начин да се направи маршрутизация с ViewSet-и е чрез **DefaultRouter**. Това опростява и систематизира структурата на URL адресите за API-то.

**urls.py:**

from django.urls import path, include

from rest\_framework.routers import DefaultRouter

from .views import CourseViewSet, RegistrationViewSet

router = DefaultRouter()

router.register(r'courses', CourseViewSet)

router.register(r'registration', RegistrationViewSet)

urlpatterns = [

path('', include(router.urls)),

]

Този фрагмент регистрира двата ViewSet-а – за курсове и записвания – и автоматично създава URL адреси с нужните HTTP методи. Примерни маршрути, които DRF генерира:

* /courses/ – GET (списък курсове), POST (нов курс)
* /courses/<id>/ – GET, PUT, PATCH, DELETE за конкретен курс
* /enrollments/ – GET (списък записвания), POST (ново записване)
* /enrollments/<id>/ – GET, PUT, PATCH, DELETE за конкретно записване

Поради използването на Router, не е нужно да се пише ръчно път за всяка отделна операция – DRF поема това автоматично, което е много удобно специално в учебни и примерни приложения.

## Връзка между компонентите

* Клиентът изпраща заявка към определен URL (/courses/, /enrollments/).
* Router я препраща към съответния ViewSet.
* ViewSet извиква сериализатора, за да превърне данните за модела в JSON (при GET) или JSON в модел (при POST/PUT).
* Сериализаторът работи с моделите (Course, Enrollment), които пък четат или записват в базата данни.
* На клиента се връща отговор със статуса и резултата (успех или грешка) в JSON формат.

## Примерни заявки и отговори

**Пример за курс (Course):**

{

"id": 1,

"name": "Летен курс по Python",

"age\_group": "12-15",

"start\_date": "2024-07-01",

"end\_date": "2024-07-30",

"teacher": "Иван Петров",

"max\_students": 12

}

**Пример за записване (Registration):**

{

"id": 1,

"child\_name": "Петър Иванов",

"child\_grade": "6 клас",

"parent\_name": "Мария Иванова",

"parent\_phone": "+359888778899",

"parent\_email": "petar.parents@mail.bg",

"course": 1

}

## Коментар и предимства на реализираната архитектура

Този подход – с DRF ViewSets, сериализатори и DefaultRouter – позволява бързо, ясно и достъпно изграждане на стандартизирано REST API.

**Основни предимства:**

* Ясна и компактна логика;
* Малко ръчно писане на код за маршрутизация;
* Всички стандартни операции са налични по подразбиране;
* Ясен и удобен за разширяване код.

В учебна среда това е чудесен пример, защото с минимум конфигурация се виждат всички основни части на REST API, а проектът лесно може да се надгради с повече модели, валидации, автентикация, или по-специални релации между данните.

Реализацията на REST API в проекта е изцяло в духа на съвременната практика и преподаване – кратка, ясна и готова за надграждане както от ученици, така и от всеки, който иска да се учи от модерен пример за проектиране на уеб услуги.

# Потребителски интерфейс

## Как е реализиран

Въпреки че основната цел на проекта е да демонстрира изграждането на REST API с Django Rest Framework, за да се „затвори цикълът“ и да се визуализира пълният процес от frontend до backend, беше реализирана проста уеб форма за регистрация на дете за избран курс. Това демонстрира работещата интеграция между frontend и backend, тестване на API-то „в реални условия“ и практическия аспект на отворената архитектура на разработеното API.

Потребителският интерфейс е реализиран като отделна HTML страница извън Django приложението – в собствена папка, за да се подчертае, че е изцяло отделен компонент и достъпът му до API-то става, както би било при реално потребление на външен API. За реализацията е използван модерен JavaScript фреймуърк **Vue.js** (чрез CDN) и библиотеката **Axios** за изпращане на HTTP заявки. Създадена е компактна, интуитивна и напълно функционална форма за регистрация, която получава наличните курсове по API, позволява избор и изпраща данните за записване към backend-а.

## Основни моменти:

* **Пълна отделеност на интерфейса** – HTML файлът е в самостоятелна папка, няма нищо общо със static файловете на Django. Това е важно, защото:
  + Демонстрира се как работят CORS HTTP хедърите – за да може формата да комуникира с REST API-то на различен адрес/порт (примерно front-end на file:// или през web server, а бекенд на http://localhost:8000).
  + Тества се структурата, която най-често се използва при реални фронтенд/бекенд разделения.
* **Използване на Vue 3** – лесно двупосочно обвързване на данните (data binding), минимален и разбираем код.
* **Axios за AJAX** – просто, универсално изпращане на GET и POST заявки.
* **Потребителски опит** – формата извежда съобщения за успех и грешка, автоматично зарежда списъка с курсовете през API.

## Примерен код на потребителския интерфейс

HTML файлът съдържа целия необходим код – саморазгъващ се Vue компонент, управляващ формата, и стилове за по-добро потребителско усещане.

<!DOCTYPE html>

<html lang="bg">

<head>

<meta charset="UTF-8">

<title>Записване за курсове</title>

<script src="https://unpkg.com/vue@3/dist/vue.global.prod.js"></script>

<script src="https://cdn.jsdelivr.net/npm/axios/dist/axios.min.js"></script>

<style>

body { font-family: Arial; max-width: 420px; margin: 30px auto;}

form label { display: block; margin: 12px 0 4px }

input, select { width: 100%; padding: 6px; margin-bottom: 10px; }

button { padding: 8px 14px; background: #48c; color: #fff; border: none; border-radius: 3px; cursor: pointer; }

button:disabled { background: #ccc; }

.message { margin-top: 15px; font-weight: bold; }

</style>

</head>

<body>

<div id="app"></div>

<script>

const server\_addr = 'http://localhost:8000'

const { createApp, ref, onMounted } = Vue;

createApp({

setup() {

const form = ref({

child\_name: "",

child\_age: "",

phone: "",

email: "",

course: ""

});

const courses = ref([]);

const message = ref("");

const loading = ref(false);

onMounted(() => {

axios.get(server\_addr + "/courses/")

.then(res => { courses.value = res.data; })

.catch(() => { message.value = "Грешка при зареждане на курсовете!"; });

});

function submitForm() {

loading.value = true;

message.value = "";

axios.post(server\_addr + "/enrollments/", form.value)

.then(() => {

message.value = "Успешна регистрация!";

form.value = { child\_name: "", child\_age: "", phone: "", email: "", course: "" };

})

.catch(err => {

if (err.response && err.response.data) {

message.value = err.response.data.non\_field\_errors

? err.response.data.non\_field\_errors[0]

: "Грешка при регистрацията!";

} else {

message.value = "Грешка при регистрацията!";

}

})

.finally(() => { loading.value = false; });

}

return { form, courses, message, loading, submitForm };

},

template: `

<h2>Запиши се за курс</h2>

<form @submit.prevent="submitForm">

<label>Име на дете:

<input v-model="form.child\_name" required>

</label>

<label>Възраст:

<input v-model="form.child\_age" type="number" min="3" max="18" required>

</label>

<label>Телефон:

<input v-model="form.phone" required>

</label>

<label>Имейл:

<input v-model="form.email" type="email" required>

</label>

<label>Избери курс:

<select v-model="form.course" required>

<option value="" disabled selected>Избери...</option>

<option v-for="course in courses" :key="course.id" :value="course.id">

{{ course.name }} (Места: {{ course.max\_students }})

</option>

</select>

</label>

<button type="submit" :disabled="loading">Изпрати заявка</button>

</form>

<div class="message" v-if="message">{{ message }}</div>

`

}).mount('#app');

</script>

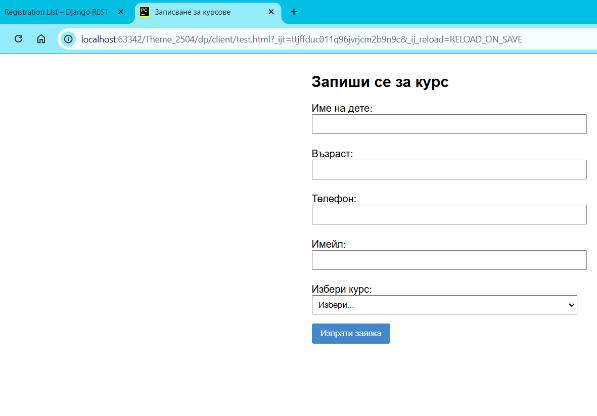
</body>

</html>

## Причини за добавянето на интерфейса

* **Демонстрация, че API-то работи реално** – през потребителска форма, а не само чрез административния панел или инструменти като Postman.
* **Завършеност на разработката** – показва целия жизнен цикъл: от въвеждане на потребителски данни, до обновяване на базата през API.
* **Тест на CORS** – тъй като HTML файлът се изпълнява независимо (примерно от локалната файлова система или различен уеб сървър), това изисква CORS настройките на бекенда да са коректни.
* **Почти реална среда** – тази архитектура се използва масово в практиката: отделен фронтенд (например SPA) и бекенд.

## Примерен вид на интерфейса (screenshot)

Добавянето на отделен потребителски интерфейс с Vue.js и Axios, макар и извън преките изисквания, придава завършеност на проекта, отлично демонстрира работещ REST API, улеснява реалното тестване и показва как на практика се интегрират frontend и backend части на съвременна уеб система.

## Примери за работа с приложението

Разработеното приложение предоставя удобен REST API, който може да се използва както от отделни потребителски интерфейси (фронтенд), така и директно през вградения интерфейс на Django Rest Framework (DRF). DRF автоматично създава документaция, лесна за разглеждане и тестови заявки през браузър, което изключително улеснява работата на разработчици и тестери.

### Примерни заявки и отговори

* **Извличане на всички курсове**

GET /courses/

**Примерен отговор:**

[

{

"id": 1,

"name": "Летен курс по Python",

"age\_group": "12-14",

"start\_date": "2024-07-01",

"end\_date": "2024-07-15",

"teacher": "Иван Георгиев",

"max\_students": 12

},

{

"id": 2,

"name": "Въведение в роботиката",

"age\_group": "10-12",

"start\_date": "2024-08-01",

"end\_date": "2024-08-12",

"teacher": "Мария Колева",

"max\_students": 10

}

]

* **Добавяне на записване (регистрация) към курс**

POST /enrollments/

Content-Type: application/json

{

"child\_name": "Петър Петров",

"child\_grade": "6 клас",

"parent\_name": "Анна Петрова",

"parent\_phone": "+359881234567",

"parent\_email": "anna@example.com",

"course": 1

}

**Примерен отговор при успех:**

{

"id": 5,

"child\_name": "Петър Петров",

"child\_grade": "6 клас",

"parent\_name": "Анна Петрова",

"parent\_phone": "+359881234567",

"parent\_email": "anna@example.com",

"course": 1

}

* **Преглед на всички записвания**

GET /enrollments/

**Отговор:**

[

{

"id": 5,

"child\_name": "Петър Петров",

"child\_grade": "6 клас",

"parent\_name": "Анна Петрова",

"parent\_phone": "+359881234567",

"parent\_email": "anna@example.com",

"course": 1

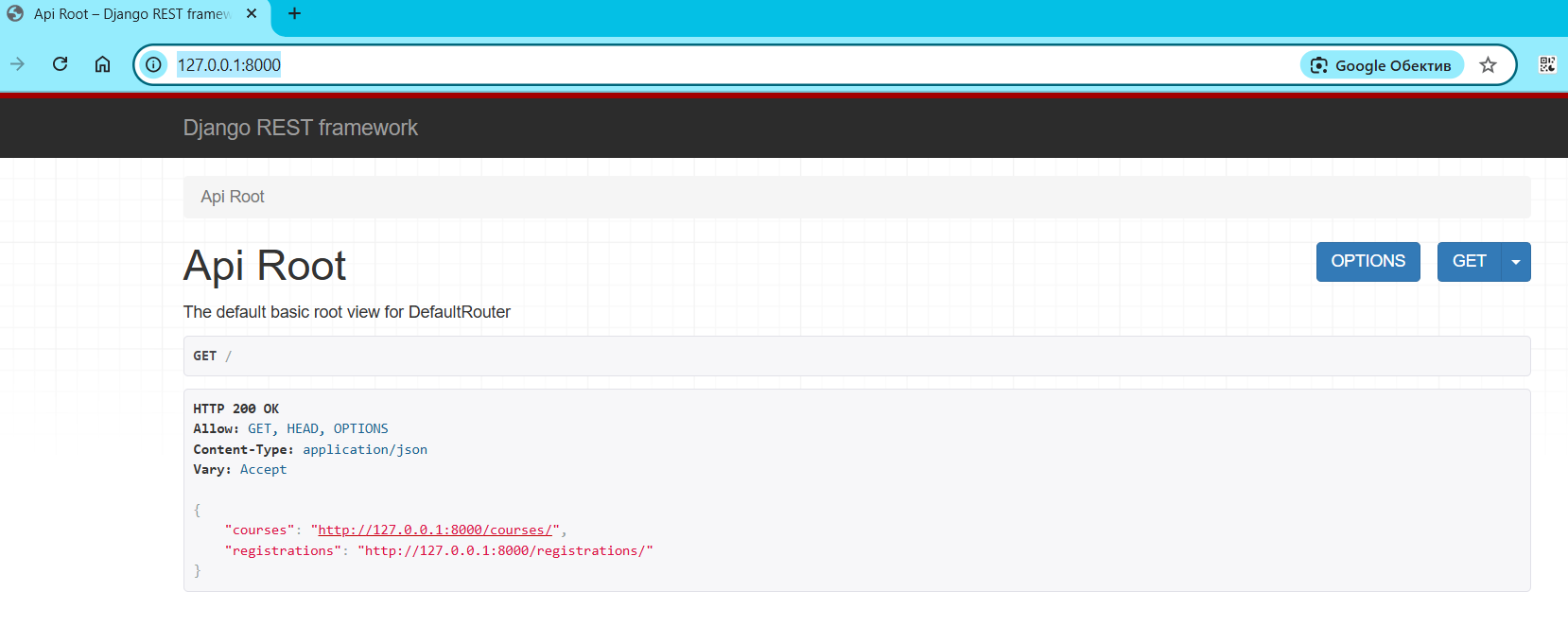
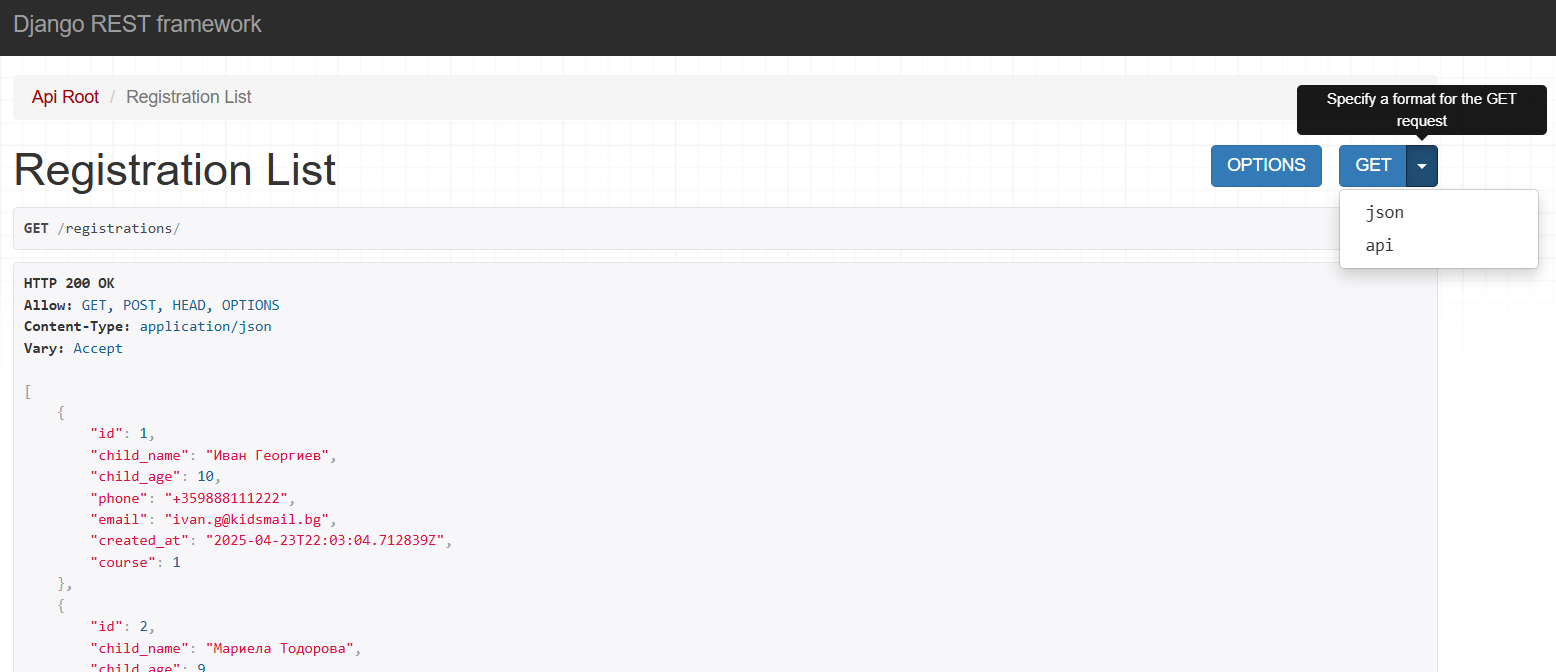
}

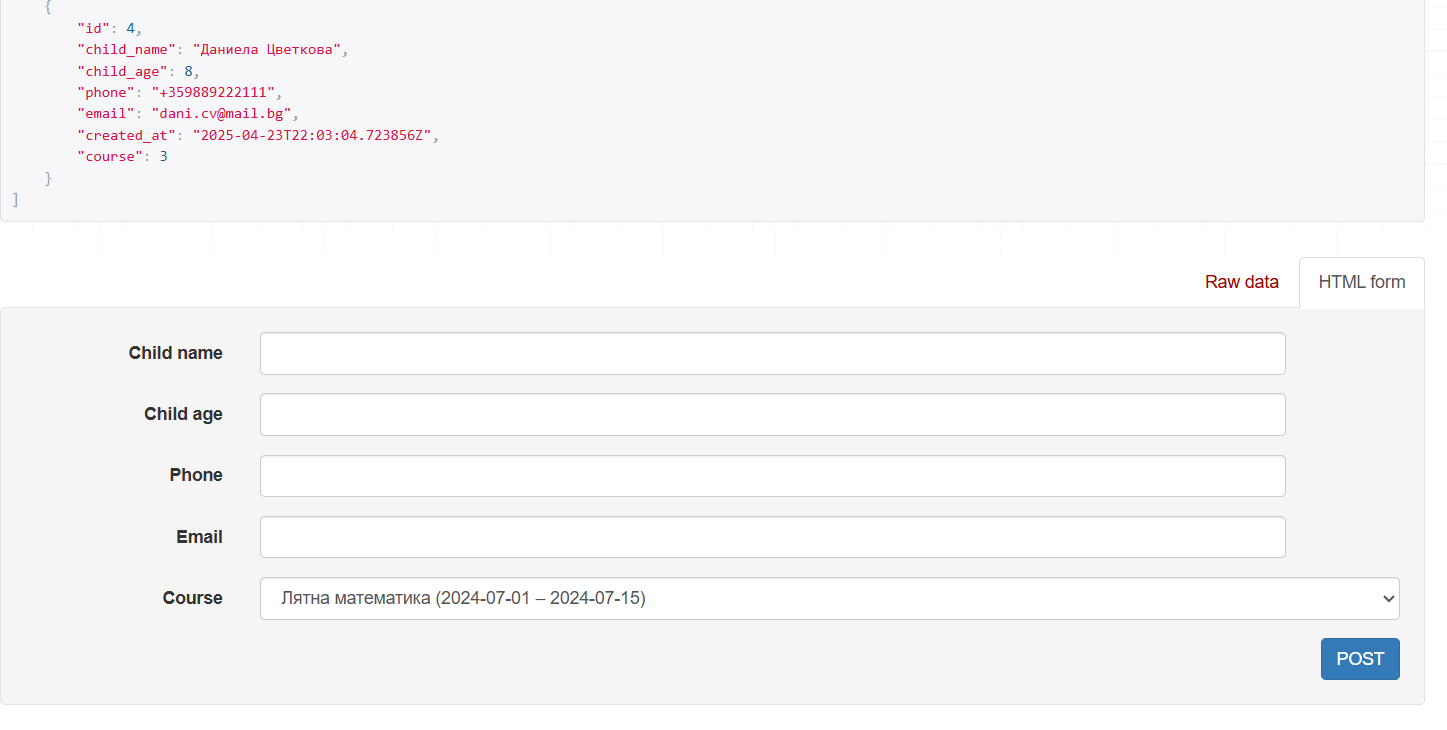
]

### Работа с вградения уеб интерфейс на DRF

Django REST Framework автоматично предоставя уеб-базиран интерфейс на всички регистрирани API адреси (/courses/, /enrollments/ и т.н.). Този интерфейс дава възможност:

* Да се разглеждат списъци и детайли за ресурсите (GET заявки).
* Да се създават, редактират и изтриват обекти през уеб форма (POST, PUT, PATCH, DELETE).
* Да се проследява входа и изхода на данните, както и формата на грешките, благодарение на динамично генерираната документация.

****



Този подход прави тестването и презентирането на работещото API изключително лесно за всеки, дори и без допълнителни инструменти като Postman или curl.

# Тестове и резултати

## Проведени тестове

В хода на разработката и интеграцията на приложението бяха проведени разнообразни тестове, както през вградения DRF интерфейс, така и чрез външния фронтенд с Vue.js:

1. **Тест на API функционалност:**

* Успешно добавяне на курс.
* Опит за добавяне на записване към курс с непопълнени задължителни полета (очаквана грешка).
* Добавяне на записване с коректни данни (очакван успех).

1. **Тест на свързаност между frontend и backend:**

* Извличане на списъка с курсове във формата за регистрация (чрез Axios).
* Изпращане на регистрация – проверка, че формата работи и когато REST API e отделно (т.е. CORS настроен коректно).

1. **Обработка на неуспехи:**

* Предизвикване на невалидни заявки през уеб формата и през API – например изпращане на лош имейл или липсваще поле.

1. **Тест на автоматичния уеб-интерфейс на DRF:**

* Достъп до /courses/ и /registration/ през браузър, проверка на всички CRUD операции през auto-generated form.

## Констатации

* **Всички основни CRUD операции работят коректно** както през frontend формата, така и през DRF уеб интерфейса.
* **Валидиращи грешки** (напр. липса на данни, погрешен тип) се показват ясно и на двата интерфейса.
* **CORS ограниченията са преодолени** – отделният HTML интерфейс зарежда и записва данни към API сървъра, което демонстрира гъвкавост при интеграция.
* **Производителност:** Отговорите от API-то са бързи и стабилни при множество последователни заявки.
* **Документацията и тестването през DRF интерфейса изисква минимални усилия**, което е голямо предимство в учебна и развойна среда.

**Извод:**

Разработеното приложение демонстрира напълно работещ REST API за курсове и записвания, с възможност за пълноценно тестване и разширяване. Вградената автоматична документация на DRF значително ускорява и опростява процеса на разработка и проверка, а външният фронтенд потвърждава реалната пригодност на решението.

# ЗАКЛЮЧЕНИЕ

Реализацията на дипломния проект демонстрира на практика как чрез съвременни технологии и добри архитектурни решения може да се изгради работещо, надеждно и лесно разширяемо REST API. Използването на Python, Django и Django Rest Framework позволява бързо и ефективно създаване на уеб услуги, които са в съответствие с най-новите стандарти в индустрията. Проектът показва, че с правилно структуриране на моделите, сериализаторите и изгледите, както и с автоматизирана маршрутизация, може да се постигне яснота, модулност и лесна поддръжка на кода.

Добавянето на отделен потребителски интерфейс с помощта на Vue.js и Axios не само доказва, че API-то работи коректно, но и демонстрира как може да се интегрира с външни системи и приложения. Това е особено важно в съвременната практика, където разделението между фронтенд и бекенд е стандарт, а комуникацията между тях се осъществява именно чрез REST API.

Проведените тестове потвърждават, че всички основни CRUD операции функционират правилно, че валидацията на данните работи както трябва, а CORS ограниченията са преодолени. Вградената автоматична документация и уеб интерфейс на DRF значително улесняват процеса на разработка, тестване и обучение, като позволяват бързо откриване и отстраняване на грешки.

Възможностите за доразвиване на проекта са многобройни. На първо място, може да се добави система за автентикация и авторизация, така че различните потребители да имат различни права за достъп до ресурсите. Може да се разшири моделът на данните с нови функционалности – например добавяне на разписания, преподаватели, автоматично известяване на родители при записване и др. Възможно е и интегриране на по-сложни валидации, както и добавяне на статистики и отчети за записванията.

Проектът може да бъде използван като основа за реални приложения в образователни институции, школи, клубове и други организации, които предлагат курсове и обучения. Освен това, той е отличен пример за обучение на начинаещи програмисти и ученици, които искат да се запознаят с принципите на изграждане на уеб услуги и интеграционни решения.

В заключение, реализираното приложение изпълнява поставените цели – демонстрира на практика изграждането на REST API, показва възможностите за интеграция с външни интерфейси и служи като стабилна основа за бъдещо развитие и обучение в областта на съвременните уеб технологии.

# Литература:

1. Django Software Foundation. Django Documentation. https://docs.djangoproject.com/en/stable/
2. Django REST Framework. Official Documentation. https://www.django-rest-framework.org/
3. Richardson, L., & Ruby, S. (2007). RESTful Web Services. O’Reilly Media.
4. Python Software Foundation. Python 3 Documentation. https://docs.python.org/3/
5. Vue.js – The Progressive JavaScript Framework. Official Guide. https://vuejs.org/guide/introduction.html
6. Axios – Promise based HTTP client for the browser and node.js. https://axios-http.com/docs/intro
7. django-cors-headers Documentation. https://github.com/adamchainz/django-cors-headers
8. OpenAPI Specification. https://swagger.io/specification/
9. Postman Learning Center – API Testing and Documentation. https://learning.postman.com/docs/getting-started/introduction/
10. W3Schools. HTML, CSS, JavaScript Tutorials. https://www.w3schools.com/
11. SoftUni Blog – Django и Vue.js: Съвременен подход към уеб разработката. https://softuni.bg/blog/vue-js-january-2021
12. Mozilla Developer Network (MDN) – HTTP Methods and CORS. https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS
13. Swagger UI – API Documentation Tools. https://swagger.io/tools/swagger-ui/
14. GitHub – Примерни проекти с Django REST Framework. https://github.com/encode/django-rest-framework