СОФИЙСКА ПРОФЕСИОНАЛНА ГИМНАЗИЯ ПО ЕЛЕКТРОНИКА „ДЖОН АТАНАСОВ“

ДЪРЖАВЕН ИЗПИТ ЗА ПРИДОБИВАНЕ НА ТРЕТА СТЕПЕН НА ПРОФЕСИОНАЛНА КВАЛИФИКАЦИЯ

Д И П Л О М Е Н П Р О Е К Т

Тема: Уеб базирана платформа за съвместно учене

Практика: Разработка на уеб базирана платформа за съвместно учене

професия: 481020 „Системен програмист“

специалност: 4810201 „Системно програмиране“

Дипломант: Александър Рангелов, 12а клас

Ръководител: инж. Веселина Маринова

Подпис: ……

(дипломант)

Подпис:……

(ръководител)

София, 2025 г.

Уеб базирана платформа за съвместно учене

Александър Рангелов

Съдържание

Уеб базирана платформа за съвместно учене

Александър Рангелов

1. Въведение

1.1. Анализ на заданието

Целта на този проект е да се разработи удобно за ползване уеб приложение за съвместно споделяне и решаване на учебни тестове. Учебната платформа ще разполага с удобен, минималистичен и мобилно предназначен потребителски интерфейс, който позволява създаването на два вида профили - потребителски и администраторски. Администраторът ще има пълен контрол върху платформата, тестовете и всеки потребител:

- Възможност за създаване на нови потребителски и администраторски профили.
- Възможност за промяна информацията на всеки съществуващ профил.
- Възможност за принудителен изход на потребител от платформата (log out).
- Възможност за създаване на учебен тест.
- Възможност за редакция на всеки тест, въпрос и отговор.
- Възможност за изтриване на всеки тест, въпрос и отговор.
- Възможност за промяна на автора на съответен тест.
- Възможност за промяна на публичността на съответен тест.
- Възможност за харесване на тест.

Потребителят от своя страна ще има лимитирани права и достъп единствено до собствени тестове и лична информация. Тази лимитация има за цел да осигури поверителност и да запази данните на потребителите.

Всеки потребител ще има възможността да разглежда и решава съвкупността от всички публични тестове, както и свои частни такива. Приложението ще има търсачка на тестове, способна на комплексно търсене и филтриране, въз основа на множество параметри като потребителско име на автора, предмет или категория и частично съответствие на заглавие въведено от потребителя. Приложението ще разполага и със

сортиране на тестове и предмети по азбучен ред или по най-скоро публикувани. Персонализираният каталог, комбиниран с търсачката, ще улесни намирането на подходящите тестове според академичните нужди на потребителя, спомагайки за интуитивното и удобно ползване на платформата.

1.2. Основна идея на заданието

Основната идея на заданието е да революционализира традиционния модел на обучение и да изпълни ежедневието ни със знания чрез колаборативен подход към споделянето на знания. Всеки потребител, не зависимо дали ученик, студент или преподавател да може да създава, и разпространява знания, като по този начин децентрализира достъпа до качествени учебни материали. Незабавната обратна връзка улеснява проследяването на напредък. Всички гореспоменати идеи, обединени в концепцията за дигитален "учебен приятел" - “StddyBddy”. Приятел, предоставящ една фокусирана учебна среда, достъпна за всички, навсякъде, независимо от образователното ниво или техническата грамотност на човек.

1.3. Поставени цели

Целта на дипломната работа е да се изготви адаптивно уеб приложение, предлагащо отлично потребителско изживяване, независимо от устройството. Основната задача е разработката на уеб платформа с оптимизирана, модулна и стандартизирана архитектура, допринасяща за бъдещото развитие на приложението.

Следва изготвянето на удобен и интуитивен за ползване графичен интерфейс, предоставящ лесна навигация и приятно потребителско изживяване при използване на платформата. Приложението има за цел да предостави лесна навигация и да преведе потребителите през предназначената гладка и целенасочена интеракция с приложението.

Освен това приложението трябва да предлага възможност за проследяване на резултатите от решенията на съответен тест. Това има за цел да окуражи напредъка на всеки потребител с всяко изминало решаване на тест. Като допълнение, потребителят ще има възможността да хареса даден тест.

Накрая, платформата има за цел оптимизира всекидневната ѝ полза. Затвореният тип отговори премахват нуждата от ръчно попълване на отговори от отворен тип, позволявайки на потребителя да решава повече тестове, извличайки максимално ползи от платформата. Приложението е разработено с максимална мобилна съвместимост, позволявайки на потребител да се образова на всяко устройство, дори и в движение.

Със съчетанието от упоменатите цели, платформата ще предлага удобство, ефективност, надеждност, задоволявайки нуждите и очакванията на потребителската аудитория.

2. Основна част

2.1. Избор на технологии

Този раздел представя подробна информация за всяка от избраните технологии, използвани при разработването на уеб приложението. Всяка технология е описана според нейната функция, характеристики и обосновка за използването ѝ в разработката на проекта. Всеки избор на технология набляга върху на съвременни практики за разработка, сигурност, ефективност и удобство на поддръжка.

2.1.1. Програмен език

JavaScript

JavaScript е основополагащият език върху който е изграден съвременният интернет. Това го прави естественият избор за основа на разработката на проекта. JavaScript е динамично типизиран скриптов език, поддържащ пълна съвместимост с всеки уеб браузър, позволяващ дълбоко ниво на интерактивност и динамично манипулиране на графичното съдържание в браузъра. Чрез среди за изпълнение като Node и Bun или Deno, той също така може да бъде изпълняван извън браузъра, съответно използван и за сървър или приложение в терминала. Макар че популярността му и обширната му екосистема са значителни предимства, динамичното му типизиране (проверката на типа се извършва по време на изпълнение) усложнява обмена на данни в по-големите и сложни приложения, като потенциално води до грешки по време на изпълнение. Подобни грешки могат да спрат изцяло процеса на работа на

приложението, както и да затруднят откриването и поправката им по време на разработка. Това влияние негативно върху бързината на разработка и чистотата на код базата, както и на дългосрочната поддръжка. Въпреки това, ролята му на основен и безконкурентен програмен език за взаимодействие с уеб и браузъра както и средите за изпълнение като Node го правят незаменим.

TypeScript

TypeScript е статично типизирано супер множество на JavaScript, което означава, че включва всички отличаващи функционалности на JavaScript, като добавя статични дефиниции на типове. Това статично типизиране позволява проверка на типовете по време на компилация (транспониране), а не по време на изпълнение. Кодът на TypeScript се транспонира в стандартен JavaScript, готов за изпълнение. Основното предимство е подобреното качество и устойчивост на кода, тъй като много потенциални грешки, свързани с несъответствия на типовете, се хващат и поправят още по време на разработка. Освен това статичното типизиране, Typescript значително подобрява производителността на разработчиците чрез усъвършенствани инструменти в редактора на код, като например автоматично попълване на параметри, рефакториране и форматиране на код и допринася за по-добра яснота и поддръжка на кода в малки и големи проекти. Богатата му интеграция със съвременни библиотеки и инструменти го направи предпочитан избор за осигуряване на надеждна и поддържана архитектура в целия стек. Поради гореспоменатите тези причини, TypeScript е основният програмен език за разработка на цялото приложение.

2.1.2. База данни

SQL

Structured Query Language или SQL е език, използван за управление и взаимодействие с данните, съхранявани в системи за управление на релационни бази данни (СУРБД). Той предоставя синтаксис за дефиниране на структури от данни (таблици, схеми), вмъкване, актуализиране, изтриване и извличане на данни. Стандартизираният му характер позволява използването на еднакъв синтаксис за взаимодействие с различните реализации на СУРБД.

SQLite

SQLite е минималистична C библиотека, която реализира самостоятелна, безсървърна и с нулева конфигурация SQL база данни. За разлика от традиционните решения за сървърни СУРБД, като PostgreSQL или MySQL, SQLite съхранява цялата база данни в един добре структуриран, оптимизиран бинарен файл във файловата система на хост машината, без да изисква отделен сървърен процес. Тази архитектурна значително намалява конфигурационните и административни разходи и елиминира проблемите със свързаността, което прави SQLite идеална за бърза разработка, тестване и миграция на приложения с ограничени нужди като този проект. Също така, придържането му към SQL стандарта улеснява миграцията към по-мащабни СУРБД данни като горепосочените PostgreSQL и MySQL.

PostgreSQL

PostgreSQL е мощна система за управление на релационни бази данни (СУРБД), предпочитана заради своята популярност, надеждност, цялостност и стриктното спазване на SQL стандартите. За разлика от чисто релационните системи тя предлага разширени възможности, като например съвместимост с JSON/JSONB и репрезентация на сложни типове от данни като гео координати, комплексни методи за индексиране, каскадни операции при манипулация на данни с множество релации. PostgreSQL изисква специално управление на сървър, което контрастира с безсървърния характер на SQLite. Докато SQLite осигурява достатъчна лекота на използване за първоначалната разработка и малкото натоварване на платформата, PostgreSQL представлява логичната и препоръчителна алтернатива. Ако приложението получи значителен ръст в потребителския трафик или обема на данните, които имат шанс да надхвърлят възможностите на SQLite, миграцията към PostgreSQL би била подходяща. Мащабируемостта, производителността при натоварване и усъвършенстваните функции правят PostgreSQL подходяща СУРБД за бъдещо разширяване и разработка на StddyBddy.

Drizzle ORM

Взаимодействието между сървъра и базата данни се осъществява с помощта на Drizzle ORM. Object-Relational Mapper или ORM служи като абстрактен слой, който превежда обектно-ориентираните структури в кода в релационните структури на SQL базите данни. Drizzle ORM е модерен, олекотен и специално разработен за TypeScript, наблягащ на безопасността на типовете и производителността. Разработчиците дефинират схеми на релационни таблици и изграждат заявки с помощта на TypeScript, които биват преведени като валидни SQL заявки. Използва типизацията на Typescript, за да улавя грешки по време на компилация. Drizzle превъзхожда алтернативи за ORM като Prisma и TypeORM поради интуитивния синтаксис, подобен на SQL. Също така, Drizzle поддържа както SQLite, така и PostgreSQL, елиминирайки усложненията при миграция между различните СУРБД, тъй като използват еднакви схеми и стандартизирани заявки.

2.1.3. Сървърни технологии

Node.js

Node.js е среда за изпълнение, като позволява изпълнението на JavaScript и транспониран TypeScript код извън рамките на уеб браузъра. Node.js използва асинхронен модел на изпълнение на инструкции, управляван от събития. Архитектура е подходяща за изграждане на мащабируем софтуер, свързан с входно-изходни операции, като уеб сървър или API, тъй като ефективно обработва множество едновременни връзки, без нуждата от допълнителни нишки. Node.js позволява използването на JavaScript за създаване сървърната бизнес логика, елиминирайки нуждата от друг програмен език. Node.js разполага с огромна екосистема от библиотеки, налични чрез пакетният мениджър Node Package Manager (NPM). Въпреки наличието на други платформи като Django (Python) или Spring (Java), Node.js е доказан избор при изграждането на софтуер написан на JavaScript.

Bun

Bun е високопроизводителна алтернатива на Node.js. Представлява универсален набор от инструменти за JavaScript, проектиран специално за бързина и удобство на работа, включващ среда за изпълнение, инструменти за тестване и бърз мениджър на пакети, съвместим със съществуващата екосистема от пакети на Node.js (npm). Bun се стреми към пълна съвместимост с приложните програмни интерфейси на Node, като същевременно предлага значителни подобрения във времето за стартиране, скоростта на изпълнение на програмния код и цялостното преживяване на разработчиците. Експоненциалното ускорение на работните процеси на разработката (по-бързо инсталиране на пакети, по-бързо изпълнение, по-бързо тестване), богатият асортимент от инструменти и автоматичното транспониране на Typescript са само някои от причините, поради които Bun е основата среда за изпълнение на сървърната логика на приложението.

Express.js

Express.js е минималистичен уеб фреймуърк за Node.js. Осигурява минимална структура за обработка на уеб заявки, дефиниране на API маршрути и интегриране на междинен софтуер или middleware (функции, които обработват последователно заявките). Гъвкавостта на Express дава пълен контрол на разработчика върху архитектурата на приложението. Express.js е утвърден лидер в уеб индустрията, получаващ широка подкрепа от общността, обширна екосистема от междинен софтуер и изчерпателна документация. Макар и стара технология, спрямо алтернативи като Elysia и Hono, доказаната ѝ ефективност при изграждането на REST API сървъри, в комбинация с бързината на Bun, създава прагматична платформа за бързо и ефикасно разработване на цялостната сървърна логика. Отново, TypeScript спомага за дефинирането на безопасни и типизирани заявки, резултати и маршрути.

2.1.4. Клиентски технологии

HTML

Hypertext Markup Language (HTML) е стандарт за маркиране на документи, предназначени за показване в уеб браузър. Стандартът представлява речник от елементи (например <p>, <h1>, , <input>), които определят семантичната структура и организация на информацията в страницата. StddyBddy се придържа към най-актуалната версия на стандарта - HTML5.

CSS

Cascading Style Sheets (CSS) е език, използван за описание на представянето и визуалното оформление на HTML документ. Той контролира аспекти като оформление, цветове, шрифтове, разстояния и адаптивност при устройства с различна резолюция. CSS може да бъде внедрен директно в HTML документа чрез <style> елемента или да бъде импортиран като отделен файл. Предлага и групиране на стилистични свойства под формата на клас, за всички или точно определена инстанция на елемент, присъстващ в документа.

Tailwind CSS

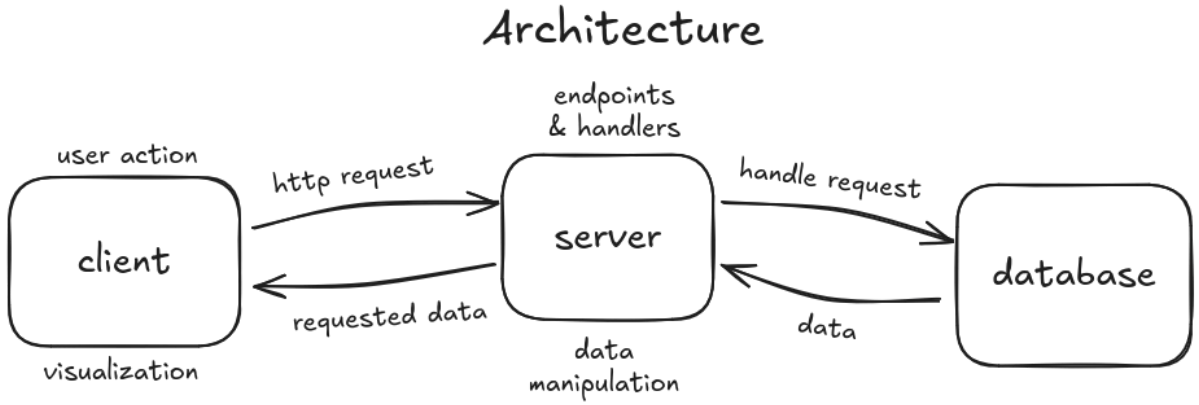
Tailwind CSS представлява изчерпателен набор от преконфигурирани независими CSS класове (.pt-4, .flex, .text-red-500), които се прилагат директно в HTML маркировката. Този подход елиминира нуждата от множество отделни CSS файлове, позволявайки изграждането на адаптивни потребителски интерфейси с минимално количество код. Ползите включват повишена скорост на разработване, чистота на кода и лесна поддръжка на потребителският интерфейс и файловата структура на клиентската част. Tailwind CSS е главният стилистичен подход при разработката на StddyBddy, поради отличната синергия с компонентни библиотеки като shadcn/ui, DaisyUI и frontend библиотеки като React, Vue и Svelte.

React

React е JavaScript библиотека, улесняваща създаването на интерактивни потребителски интерфейси. React изгражда капсулирани, независими мултиплициращи се компоненти, които управляват собственото си състояние. Библиотеката ефективно актуализира и визуализира съответните компоненти при промяна на състоянието на данните, променяйки виртуалния DOM - ключова функционалност при уеб разработката. Това опростява разработването на сложни потребителски интерфейси, разделяйки сложни логически елементи на по-прости такива. Цялата логика, API заявки, HTML маркировка, Tailwind CSS стилизация нужна за успешното визуализиране на компонента се съдържа в един .tsx файл, изцяло модулирайки целият подход на изграждане на графичният интерфейс. Макар че съществуват алтернативи като Angular или Vue.js, React библиотеката беше избрана поради обширната си екосистема, множеството компоненти библиотеки, богатата си документация, възможността за многократна употреба на дефинирани логически компоненти и отличната интеграция с TypeScript и Tailwind CSS, придържаща се към целите и ценностите на проекта.

2.2. Системен дизайн

2.2.1. Софтуерна архитектура



Фиг. 2.2.1.1. Софтуерна архитектура на приложението

Приложението използва трислойна архитектура. Разделена е на три части - клиентска, сървърна и база данни. Същинската реализация представлява приложение с една страница (Single Page Application), който взаимодейства с отделен, независим бекенд REST API сървър. Архитектурата насърчава ясното разделение на задачите: клиентската част или фронтенда е отговорен за визуализирането на потребителския интерфейс и логиката на взаимодействие с него, докато сървърът на бекенда управлява бизнес логиката и достъпа до данни и основните функционалности. Това разделение позволява независима разработка, мащабиране и внедряване на компонентите от клиентската и сървърна част.

Клиентска част

Клиентската част на проекта е реализирана като Single Page Application с помощта на библиотеката React. SPA или приложение с една страница е уеб приложение, което зарежда само един уеб документ/страница, актуализирайки само съдържанието на този документ, чрез функционалност, предоставена от React. Това позволява на потребителите да използват приложението, без да зареждат цели нови страници, което води до повишаване на производителността и потребителското изживяване, с някои недостатъци, като липса на SEO (оптимизация на търсачката), нужда от поддържане на състоянието на информацията измежду презарежданията на интерфейса и навигация.

Клиентската част е разработена с помощта на инструмента за изграждане Vite и средата за изпълнение Bun, използвани за бързо и надеждно автоматично актуализиране промени в кода и съвместимост с браузъра по време на разработката. Основната отговорност на фронтенда е да визуализира потребителския интерфейс и да управлява всички взаимодействия на потребителите в браузъра. Компонентите на React, визуализират и улавят интеракциите на потребителя. Въз основа на действията на потребителя (напр. натискане на бутон, подаване на формуляр), предварително дефинираната логика в тези компоненти формулира и изпраща асинхронни HTTP заявки към съответните крайни точки бекенд API сървъра. При получаване на информацията от отговора от бекенда фронтендът обработва данните - обикновено актуализира състоянието на компонентите, използвайки функциите за управление на

състоянието на React (useState, useEffect, ...) - което задейства актуализации на потребителския интерфейс, осигурявайки динамично и интерактивно преживяване, без да се налага пълно презареждане на страницата. Навигацията в приложението използва React библиотеката React Router.

Сървърна част

Сървърната част представлява API сървър, следващ REST методологията, изграден с уеб рамката Express.js. Той действа като ядро на приложението, обработващо бизнес логиката, валидирането на данните, удостоверяването и оторизирането на потребителите както и взаимодействието със базата данни. REST методологията за изграждане на сървърна логика насърчава използването на отделни, структурирани, лесноразпознаваеми и целенасочени крайни точки или endpoints, съответстващи на различни ресурси или функционалности на приложението (например /api/users, /api/tests, /api/auth). Всяка крайна точка е маршрут към който може да бъде изпратена определена HTTP заявка (GET, POST, PUT, PATCH, DELETE, ...), всеки от който има съпоставена определена обработваща функция. Тези функции отговарят за получаването на входящи HTTP заявки, анализиране на данните от заявките (параметрите, низовете, телата на заявките, заглавия и бисквитките, съдържащи токени за удостоверяване и оторизация) и последователността от необходимите действия за изпълнение на заявката.

База данни

Крайните точки от сървъра служат като посредник между входящите клиентски заявки и базата данни. На този етап, приложението използва SQLite релационна база данни. SQLite е подходяща за разработка и за първоначалния малък мащаб на приложението. Взаимодействието между действията на потребителя, категоризирани в маршрути от Express.js, и базата данни се управлява от Drizzle ORM. Drizzle предоставя интерфейс за дефиниране на таблици на бази данни като JavaScript обекти и изпълнение на CRUD (Create, Read, Update, Delete) операции. Когато дадена функция трябва да получи достъп до или да модифицира данни, той използва програмният интерфейс на Drizzle, за да изгради и изпълни съответните SQL заявки към базата

данни. След това извлечените данни от базата данни, се обработват в съответствие със специфичните изисквания на бизнес логиката за предназначението и целта на функцията.

```
// SQL
SELECT FROM "users" WHERE "id" = 1;

// TypeScript със Drizzle ORM
db.select().from("users").whre(eq(users.id, 1));
```

Фигура 2.2.1.2. Съпоставка между SQL и Drizzle ORM

Цикъл на изпълнение

Типичният работен цикъл на всяко действие и заявка в архитектурата на приложението следва тази последователност:

1. Потребителят извършва действие в потребителския интерфейс (напр. кликва върху бутон за създаване на учебен тест).
2. Логиката на съответния React компонент изпраща HTTP заявка (напр. POST заявка към /api/test), съдържаща необходимите данни (заглавие на теста, въпроси, ... в тялото на заявката) и данни за удостоверяване и оторизация (бисквитки и сесиен ключ).
3. Сървърът получава входящата заявка.
4. Express пренасочва заявката към определения маршрут и съответстващата обработваща функция, като евентуално първо преминава през междинен софтуер (удостоверяване и оторизация).
5. Функцията извлича данни от заявката, извършва валидиране и използва Drizzle ORM, за да изпълни необходимите операции с базата данни (например вмъкване на нов тест в базата данни).
6. Въз основа на успеха или неуспеха на операцията се изгражда подходящ HTTP отговор (напр. код на състоянието 201 Created с данните за новия тест или 400

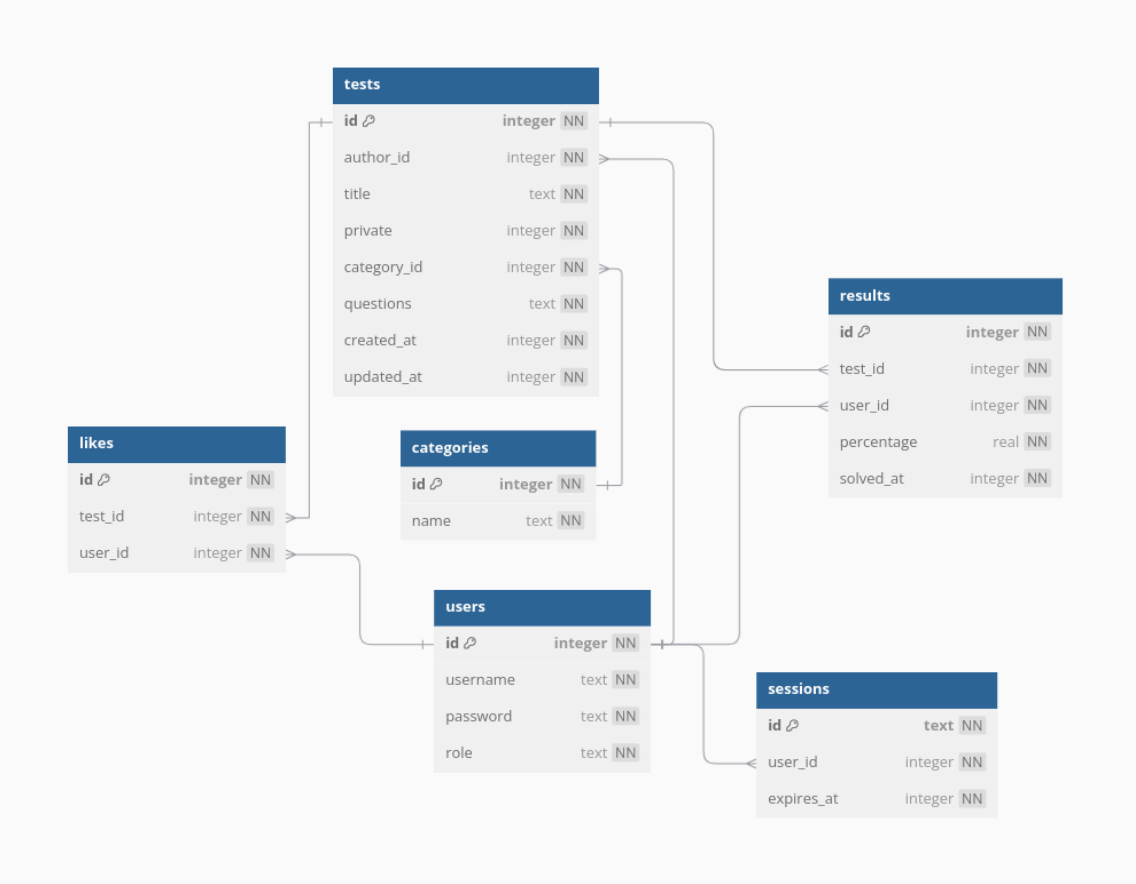
Bad Request, ако валидирането е неуспешно). Данните, включени в отговора, се обработват, като съдържат само необходимата информация.

7. Сървърът изпраща HTTP отговора обратно на клиента.

8. Клиента получава отговора. Клиентската логика обработва данните от отговора (напр. превръща време запазено в милисекунди в дата) и актуализира съответните компоненти на потребителския интерфейс спрямо новото състояние, за да отрази промяната.

Този цикъл на изпълнение гарантира модулност и лесна поддръжка чрез ясно разграничаване на отговорностите на клиента, сървъра и базата данни в рамките на цялостната архитектура на приложението.

2.2.2. Дизайн на базата данни



Фигура 2.2.2.1. Диаграма на релационните таблици на базата данни

Users

Съхранява информация за потребителите на системата.

- **id**: Първичен ключ, уникален идентификатор за всеки потребител
- **username**: Потребителско име
- **password**: bcrypt хеш на парола
- **role**: Роля на потребителя (администратор или потребител)

Sessions

Съхранява сесиите за удостоверяване и оторизация на потребителите.

- **id**: Първичен ключ, уникален идентификатор на сесията, 16 байтов, хекс енкодиран низ
- **user_id**: Външен ключ, свързващ таблицата users
- **expires_at**: Показва кога изтича сесията в милисекунди

Tests

Това е централната таблица, в която се съхранява информация за всички тестове в приложението.

- **id**: Първичен ключ, уникален идентификатор за всеки тест
- **author_id**: Външен ключ, свързващ таблицата users, сочи към създателя на теста
- **title**: Заглавието на теста
- **private**: Булев флаг (true или false), показващ дали тестът е частен или публичен
- **category_id**: Външен ключ, свързващ таблицата categories, сочи към категорията/предмета на теста

- **questions**: Съхранява въпросите на теста, в структуриран JSON формат, превърнат в низ за по-лесно съхранение
- **created_at**: Кога е създаден тестът в милисекунди
- **updated_at**: Кога тестът е бил последно актуализиран в милисекунди

Categories

Организира тестовете в различни предмети и области.

- **id**: Първичен ключ, уникален идентификатор за всяка категория
- **name**: Име на категорията

Results

Резултат от решения на тестовете.

- **id**: Първичен ключ, уникален идентификатор за всяка категория
- **test_id**: Външен ключ, свързващ таблицата tests
- **user_id**: Външен ключ, свързващ таблицата users
- **percentage**: Резултат от решението в проценти
- **solved_at**: Кога тестът е бил решен в милисекунди

Likes

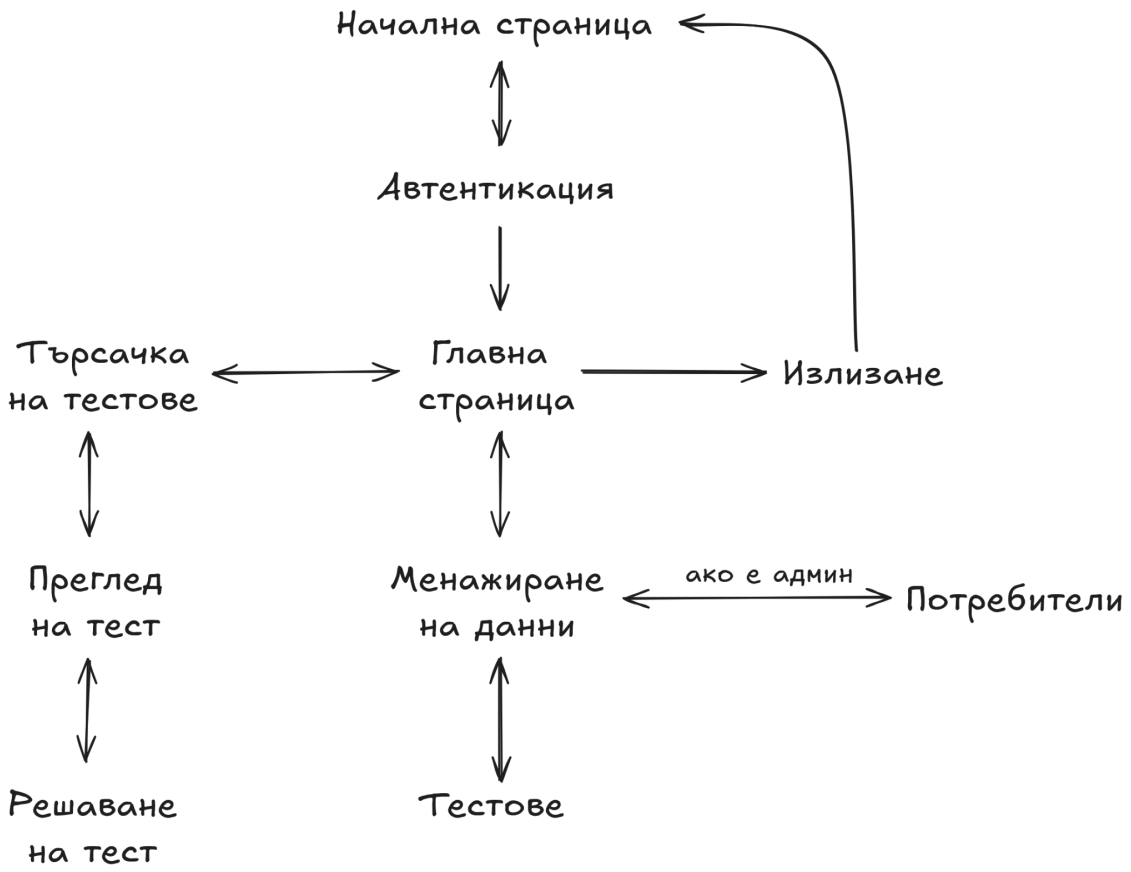
Съхранява харесваните тестове.

- **id**: Първичен ключ, уникален идентификатор за всяко харесване
- **test_id**: Външен ключ, свързващ таблицата tests
- **user_id**: Външен ключ, свързващ таблицата users

Релации

- **Потребители към тестове**: едно към много - един потребител може да създава множество тестове.
- **Категории към тестове**: едно към много - една категория може да съдържа множество тестове.
- **Тестове към резултати**: едно към много - един тест може да съдържа множество резултати от различни потребители.
- **Потребители към резултати**: един към много - един потребител може да има резултати от няколко теста.
- **Тестове към харесвания**: едно към много - даден тест може да получи харесвания от множество потребители.
- **Потребители към харесвания**: едно към много - един потребител може да харесва няколко теста.
- **Потребители към сесии**: едно към много - даден потребител може да има няколко активни сесии.

2.2.3. Взаимодействие с приложението



Фигура 2.2.3.1. Data flow диаграма на приложението

Интеракцията на всички видове потребители с онлайн платформата може да се види схематично на Фиг. 2.2.3.1.

Начална страница

Потребителите започват пътуването си от началната страница на, която служи като основна входна точка към приложението. Оттук те се насочват към процеса на автентикация.

Автентикация

Страницата за автентикация предоставя възможността на потребителите да влязат в системата със съществуващ профил или да се регистрират като нов потребител. При успешно удостоверяване потребителите се насочват към главната страница.

Главна страница

Главното табло функционира като централен навигационен център, предлагащ три основни пътя: достъп до функцията за търсене и решаване на тестове, достъп до панела за управление на тестове и потребители (ако потребителя има администраторска роля).

Излизане

Потребителят има възможността да излезе от профила си по всяко време.

Търсене на тестове

Потребителите могат да преглеждат, филтрират и сортират наличните тестове и да изберат конкретен тест, за да бъдат препратени към страницата за преглед на тест.

Преглед на тест

От страницата за преглед те могат да видят информация за конкретният тест като автор, предмет, дата на създаване и препратка към решаване на теста.

Решаване на тест

След като завършат и изпратят отговорите си, потребителите се пренасочват обратно към страницата за преглед на тестовете, където се показват резултатите им потребителите могат да се върнат към екрана за търсене на тестове, за да намерят допълнителни тестове.

2.3. Обяснение на кода

Кодът на приложението е организиран в 2 основни, независими директории, които отразяват трислойната архитектура - frontend и backend. Файловата структура има за цел да създаде среда за лесно мащабиране и имплементиране на нови функционалности.

2.3.1. Оторизация

Какво е сесия?

HTTP по своята същност е без състояние, което означава, че всяка заявка се обработва независимо, независимо от предишни взаимодействия. За да се запази идентичността на потребителя при множество заявки след влизане в системата, Приложението използва управление на сесии. Сесията предоставя на сървъра механизъм за „запомняне“ на влезлия в приложението потребител за определен период от време.

Управление на сесии

В реализацията на приложението, при успешно удостоверяване на потребителя сървърът създава сесия. Това включва генериране на уникален, защитен сесиен ключ или токен (в този случай, 16 символен, произволен низ).

```
// функцията генерира 16 байтов, хекс енкодиран низ,
// използван за идентификация на потребител
export function generateToken() {
  return randomBytes(16).toString("hex");
}
```

Фигура 2.3.1.1. Функция за генериране идентификационен сесиен ключ

Този ключ служи като временно удостоверение. От решаващо значение е, че този ключ се асоциира от страна на сървъра с основни данни за потребителя - неговия id и възложена роля. След това генерираният ключа се изпраща обратно към браузъра на клиента и се съхранява в сесийна бисквитка. Браузърът автоматично включва тази бисквитка във всяка следваща заявка, направена към сървъра.

```
export function setSessionCookie(res: Response, token: string, expiresAt: Date) {
  res.cookie("session", token, {
    httpOnly: true,  // Предотвратява JS достъп, защитава от XSS атаки
    sameSite: "lax", // Изпраща бисквитки само при навигация към вашия сайт
    expires: expiresAt, // Задава кога браузърът ще изтрие бисквитката
    path: "/",  // Прави бисквитката достъпна във всички маршрути
    secure: true,  // Бисквитката се изпраща само през HTTPS връзки
  });
}
```

Фигура 2.3.1.1. Функция за за изпращане на сесийна бисквитка

За всяка входяща заявка към защитени крайни точки сървърът трябва да провери самоличността на потребителя. Обикновено това се обработва от специален междинен софтуер, който се изпълнява преди основната обработваща на маршрута. Ключът бива извлечен от сесийната бисквитка, включена в заявката. След това ключът се валидира - проверява се неговата автентичност, дали е изтекъл и дали съответства на активна сесия в базата данни.

```
export async function validateSession(token: string) {
  // Заявка към базата данни за сесията и свързания потребител
  const result = await db
    .select({ user: usersTable, session: sessionsTable })
    .from(sessionsTable)
    .innerJoin(usersTable, eq(sessionsTable.user_id, usersTable.id))
    .where(eq(sessionsTable.id, token));

  // Ако не е намерена съвпадаща сесия, връща null стойности
  if (result.length < 1) return { session: null, user: null };

  const { user, session } = result[0];

  // Проверка дали сесията е изтекла
  if (Date.now() >= session.expires_at.getTime()) {
    // Изтриване на изтеклата сесия от базата данни
    await db.delete(sessionsTable).where(eq(sessionsTable.id, session.id));
    return { session: null, user: null };
  }

  // Ако сесията изтича в рамките на 3 дни, удължава я с 1 ден
  const THREE_DAYS = DAY_IN_MILLIS * 3;
  if (Date.now() >= session.expires_at.getTime() - THREE_DAYS) {
    // Изчисляване на ново време за изтичане (1 ден от сега)
    session.expires_at = new Date(Date.now() + DAY_IN_MILLIS);

    // Обновяване на срока на сесията в базата данни
    await db
      .update(sessionsTable)
      .set({ expires_at: session.expires_at })
      .where(eq(sessionsTable.id, session.id));
  }

  // Връщане на валидната сесия и потребител
  return { session, user };
}
```

Фигура 2.3.1.2. Функция за валидиране на сесия чрез сесиен ключ

2.3.2. CRUD операции

CRUD е съкращение от Create (създаване), Read (четене), Update (актуализиране) и Delete (изтриване). Тези четири операции представляват основните действия, необходими за управление на данни. Те са в основата на повечето приложения, които взаимодействат със съхранена информация. Създаване се отнася до добавянето на нови записи на данни, четенето е свързано с извличане на съществуващи данни, конкретни записи или списъци от записи, актуализация означава промяна на съществуващи записи с данни и накрая, изтриване се отнася до окончателното премахване на записи на данни. Ефективното прилагане на тези CRUD операции е от съществено значение за управлението на жизнения цикъл на данните на приложението. Ще разгледаме CRUD операциите за тестове, като отличителната част в същността на приложението

Създаване (Create)

```
export async function createTest(req: Request, res: Response) {
  // Извличане на токена на сесията от бисквитките
  const token = getSessionFromCookie(req);
  if (!token) return res.status(307).send("Session expired");

  // Валидиране на сесията и потребителя
  const { user, session } = await validateSession(token);
  if (!session) return res.status(401).send("Unauthorized");

  // Обновяване на бисквитката на сесията
  setSessionCookie(res, token, session.expires_at);

  // Асинхронна валидация на данните за теста
  const validation = await TestValidation.safeParseAsync(req.body);
  if (!validation.success) return res.status(400).send("Invalid test");
```

```
  // Създаване на обект с данни за теста
  const testData = {
    ...validation.data,
    author_id: user.id,
    created_at: new Date(),
    updated_at: new Date(),
  };

  // Записване в базата данни
  const newTest = await db
    .insert(testsTable)
    .values(testData)
    .returning() // Връща обекта след въвеждане
    .catch(error => res.status(500).send("Internal server error"))
      .limit(1);

  // Връщане на създаден тест към потребителя
  res.status(201).send(newTest);
}
```

Фигура 2.3.2.1. Функция за създаване на тест

Функцията createTest обработва HTTP POST заявки за създаване на нови тестове. Първоначално тя проверява сесията на потребителя с помощта на сесийният ключ от бисквитката, като при неуспешно удостоверяване веднага отговаря със съответните кодове на състоянието (307 за изтекли сесии, 401 за неоторизиран достъп). След като заявката бъде оторизирана, валидацията на данните на теста гарантира, че тялото на заявката отговаря на очаквания формат. Ако валидирането премине успешно, функцията обогатява тестовите данни с информация за автора и времеви маркери, преди да ги съхрани в базата данни. Функцията осигурява полезна обратна връзка чрез ясни HTTP кодове на състоянието и съобщения - 400 за невалидни тестови данни, 500 за грешки на сървъра и 201 с новосъздадените тестови данни при успех. Функцията ефективно управлява целия процес на създаване на тестове от удостоверяването до съхранението на данните, като през цялото време обработва правилно грешките.

Четене (Read)

```
export async function getTests(req: Request, res: Response) {
  // Извличане на токена от бисквитката на сесията
  const token = getSessionFromCookie(req);
  if (!token) res.status(307).send("No session found");

  // Валидация на сесията и извличане на потребител
  const { user, session } = await validateSession(token);
  if (!session) res.status(401).send("Unauthorized");

  // Обновяване на бисквитката на сесията
  setSessionCookie(res, token, session.expires_at);

  // Извличане на тестовете от базата данни
  // Връща публични тестове и тестове, създадени от текущия потребител
  const tests = await db
    .select()
    .from(testsTable)
    .where(
      or(eq(testsTable.isPrivate, false), eq(testTable.author_id === user.id)),
    );

  // Изпращане на тестовете като отговор със статус код 200 (OK)
  res.status(200).send(tests);
}
```

Фигура 2.3.2.2. Функция за четене на всички тестове

Функцията getTests обработва заявки за извличане на тестове от базата данни. Първо проверява дали потребителят има валидна сесия. Ако сесията е валидна, функцията обновява бисквитката и след това извлича тестове от базата данни. Важен аспект е, че функцията прилага филтър, който гарантира, че потребителят вижда само публични тестове и тестове, които самият той е създал (ако са частни). Това осигурява правилно ниво на достъп до данните. Накрая, функцията връща намерените тестове като JSON с HTTP статус 200 (OK).

Използването на асинхронни функция (async/await) тук е ключово, защото позволява неблокиращо изпълнение на операциите за валидиране на сесия и достъп до базата данни, което значително подобрява производителността на сървъра при множество паралелни заявки

Актуализиране (Update)

```
export async function updateTest(req: Request, res: Response) {
  // Проверка дали съществува сесия
  const token = getSessionFromCookie(req);
  if (!token) res.status(307).send("Session expired");

  // Валидиране на сесията
  const { user, session } = await validateSession(token);
  if (!session) res.status(401).send("Unauthorized");

  // Обновяване на бисквитката със сесията
  setSessionCookie(res, token, session.expires_at);

  // Проверка дали URL параметъра id е валидно число
  const testId = Number(req.params.id);
  if (isNaN(testId)) res.status(400).send("Invalid test id");

  // Търсене на теста в базата данни
  const existingTest = await db
    .select()
    .from(testsTable)
    .where(eq(testsTable.id, testId));
  if (existingTest.length === 0) res.status(404).send("Test not found");

  // Проверка дали потребителят има право да редактира теста
  if (existingTest[0].author_id !== user.id && user.role !== "admin") {
    res.status(401).send({ error: "Unauthorized" });
  }

  // Валидиране на данните за теста
  const validation = await TestValidation.safeParseAsync({
    ...existingTest[0], // старият тест
    ...req.body, // новата информация
  });
  if (!validation.success) res.status(400).send("Invalid test data");

  // Обновяване на теста в базата данни
  const updatedTest = await db
    .update(testsTable)
    .set(updatedTest)
    .where(eq(testsTable.id, testId))
    .returning()
    .catch((error) => res.status(500).send("Internal server error"));

  // Връщане на успешен отговор с обновените данни
  res.status(200).send(updatedTest);
}
```

Фигура 2.3.2.3. Функция за актуализиране на тест

Функцията updateTest служи за обновяване на съществуващ тест. След успешна валидация на сесията, функцията извлича съществуващия тест от базата данни с помощта на валидиран URL параметър id и проверява дали потребителят има право да го редактира. Данните от заявката се валидират, комбинирайки съществуващите данни с промените. Ако валидацията е успешна, тестът се обновява в базата данни с новите данни и актуализирана дата на промяна. Накрая, актуализираният тест се изпраща към клиента за обновяване на потребителският интерфейс.

Изтриване (Delete)

```
export async function deleteTest(req: Request, res: Response) {
  // Проверка дали URL параметъра id е валидно число
  const testId = Number(req.params.id);
  if (isNaN(testId)) res.status(400).send({ error: "Invalid test data" });

  // Проверка дали съществува бисквитка
  const token = getSessionFromCookie(req);
  if (!token)  res.status(307).send({ error: "Session expired" });

  // Валидиране на сесията
  const { user, session } = await validateSession(token);
  if (!session)  res.status(401).send({ error: "Unauthorized" });
  // Обновяване на бисквитката със сесията
  setSessionCookie(res, token, session.expires_at);

  // Проверка дали тестът съществува
  const existingTest = await db
    .select()
    .from(testsTable)
    .where(eq(testsTable.id, testId))
    .limit(1);
  if (existingTest.length === 0) res.status(404).send("Test not found" );

  // Проверка дали потребителят има право да изтрие теста
  if (existingTest[0].author_id !== user.id && user.role !== "admin")
    res.status(403).send("Forbidden" );

  try {
    // Първо изтриване на всички харесвания, свързани с теста
    await db.delete(likesTable).where(eq(likesTable.test_id, testId));
    // След това изтриване на самия тест
    await db.delete(testsTable).where(eq(testsTable.id, testId));
    // Връщане на успешен отговор
    res.status(200).send("Test deleted");
  } catch (dbError) {
    // Обработка на грешки при работа с базата данни
    res.status(500).send("Internal server error");
  }
}
```

Фигура 2.3.2.4. Функция за изтриване на тест

Функцията deleteTest отговаря за изтриването на тест от приложението. Процесът започва с валидация на идентификатора на теста и оторизация на потребителя. Функцията първо проверява дали тестът съществува в базата данни. Следва проверката за права - само авторът на теста или администратор могат да го изтрият. След това функцията изпълнява операцията, при която първо се изтриват всички свързани харесвания, а след това и самият тест. Тъй като SQLite не поддържа каскадни операции, се изготвят две заявки за изтриване, елиминирайки грешки с външните ключове на вече несъществуващи тестове. Структурата try/catch осигурява правилна обработка на грешки при взаимодействието с базата данни, гарантирайки надеждност на операцията по изтриване.

3. Заключение

3.1. Постигнати цели

Проектът на тема - уеб базирана платформа за съвместно учене (StddyBddy), беше разработен успешно. Платформата отговаря на поставените цели за изграждане на модерно уеб приложение. Трислойната архитектура ефективно разделя логиката, като включва и интуитивен, динамичен потребителски интерфейс, изграден с библиотеката React. Клиентската част комуникира надеждно с REST API сървър, разработен върху уеб рамката Express.js, работещ в средата за изпълнение Bun.

Ключовите функционалности на приложението също бяха успешно реализирани. Те включват надеждна регистрация на потребителите, сигурни механизми за удостоверяване и оторизация, базирани на сесии, управление на достъпа на потребителите. Освен това отличителната единица данни, за това приложение - „тестовете“ - може да бъде напълно управлявана чрез реализирани CRUD операции (създаване, четене, актуализиране, изтриване). Проектът успешно демонстрира интеграцията и практическото приложение на избрания съвременен технологичен стек (TypeScript, React, Bun, Express, Drizzle, SQLite, TailwindCSS) и създава солидна и поддържана основа на кодовата база.

3.2. Потенциално развитие

Въпреки че настоящата версия на StddyBddy осигурява функционално завършен продукт, съществуват многобройни възможности за бъдещо развитие, за да се подобрят значително възможностите, потребителското изживяване и техническата надеждност. Следните области представляват потенциални възможности за подобрение.

Разпечатване на тестове

Практично подобрение би било въвеждането на функционалност, която позволява на потребителите да генерират удобни за печат версии на своите тестове директно от приложението. Това би било подходящо за потребители, които предпочитат офлайн методи на обучение или учители решили да използват нечии тест за класна дейност и се нуждаят от физически копия за класната стая или група.

Интеграция с училищни заведения

Приложението би могло да бъде разширено с интеграция с училищни заведения. Това би могло да включва управление на съдържанието и потребителите от различни училища, класове и паралелки.

Интернационализация

За да се разшири обхватът и достъпността на приложението, прилагането на интернационализация и поддръжка на множество езици би било ценно допълнение. Това включва адаптиране на текста на потребителския интерфейс и евентуално на форматите за дата/числа за различни локации.

Миграция на база данни

Тъй като приложението предвижда потенциално нарастване на броя на потребителите и обема на данните, силно се препоръчва мигриране на базата данни от SQLite към по-мащабна, сървърна СУРБД като PostgreSQL. Това преминаване ще осигури по-добра мащабируемост, обработка на паралелни операции и

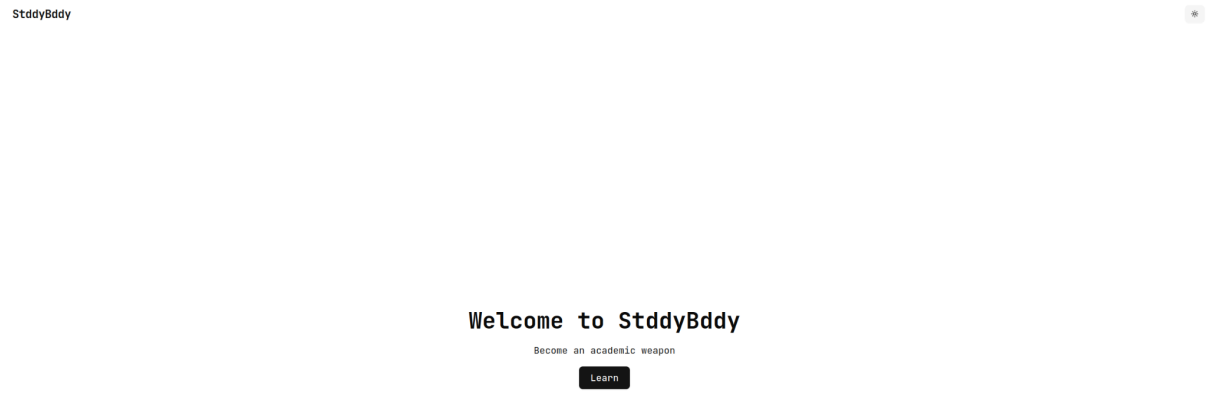
производителност при натоварване, като гарантира, че приложението ще остане отзивчиво и надеждно, докато се разширява.

4. Източници

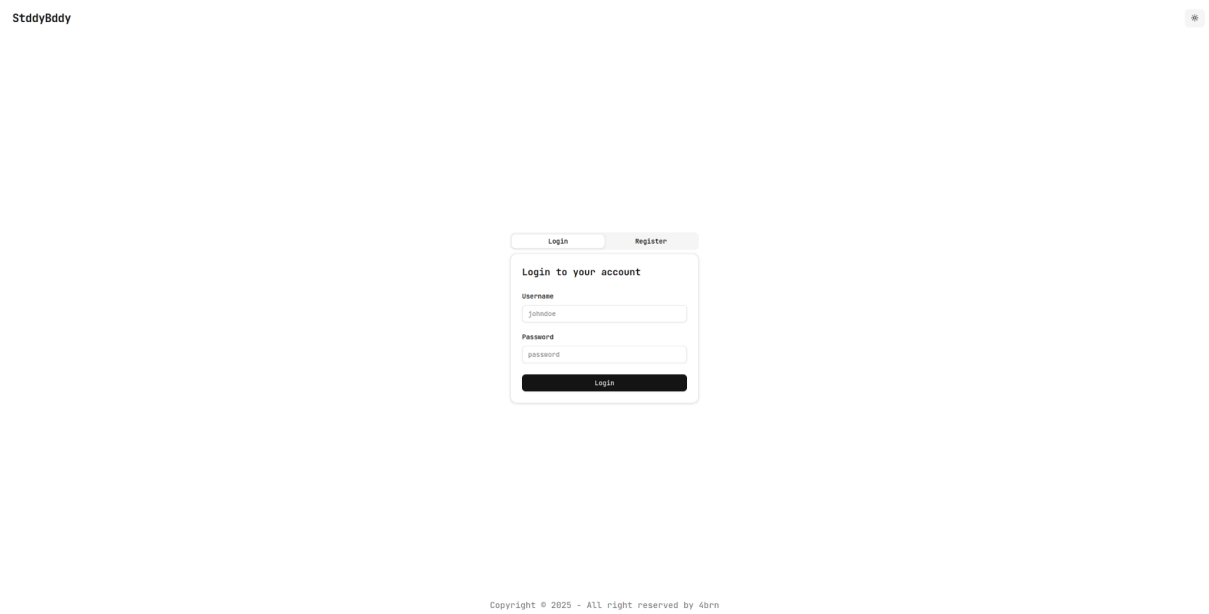
Повечето информация е взета от документациите на технологиите използвани за разработката на приложението:

1. Обяснение и анализ на архитектурата клиент-сървър и нейните ползи: https://www.geeksforgeeks.org/client-server-architecture-system-design/
2. Официалният HTML стандарт: https://html.spec.whatwg.org/
3. Какво е CSS: https://developer.mozilla.org/en-US/docs/Glossary/CSS
4. Документацията на Typescript: https://www.typescriptlang.org/docs/
5. REST методологията: https://developer.mozilla.org/en-US/docs/Glossary/REST
6. CRUD операции: https://en.wikipedia.org/wiki/Create,_read,_update_and_delete
7. База данни SQLite: https://sqlite.org
8. Документацията на Drizzle ORM: https://orm.drizzle.team/docs/
9. Среда за изпълнение Bun: https://bun.sh/docs
10. Минималистичната уеб рамка Express.js: https://expressjs.com/
11. Документацията на React: https://react.dev/
12. Навигация в React приложение с една страница: https://reactrouter.com
13. Динамичен дизайн с Tailwind CSS: https://tailwindcss.com/docs/
14. Компонентната библиотека: https://ui.shadcn.com/docs/
15. Документация за оторизацията и управлението на сесии: https://lucia-auth.com/
16. Валидация на данни със Zod библиотеката: https://zod.dev/

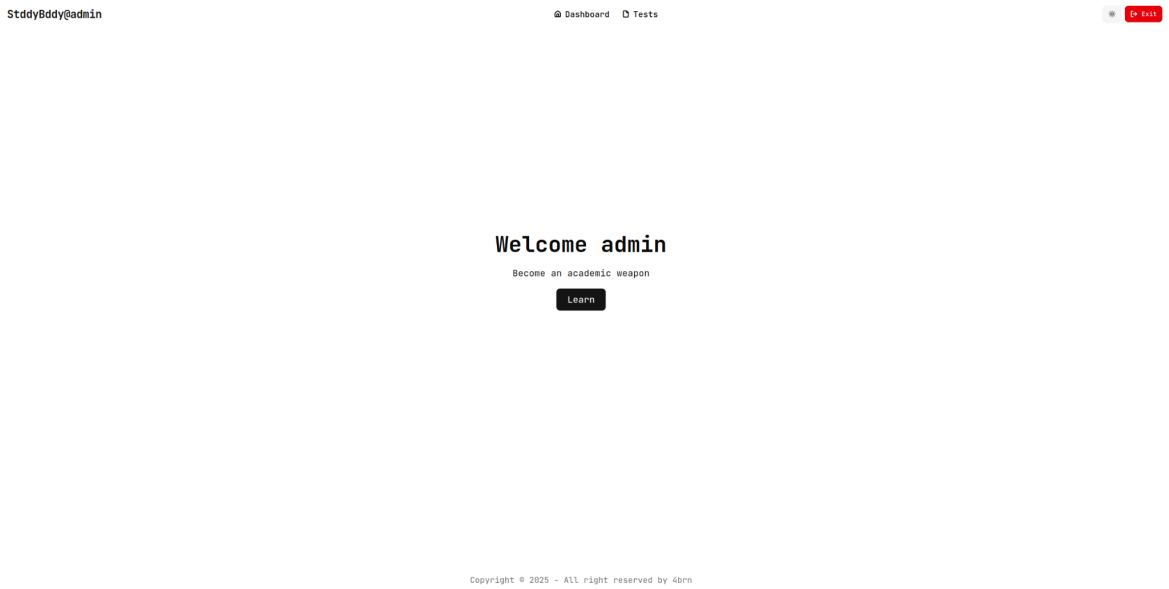
5. Приложения



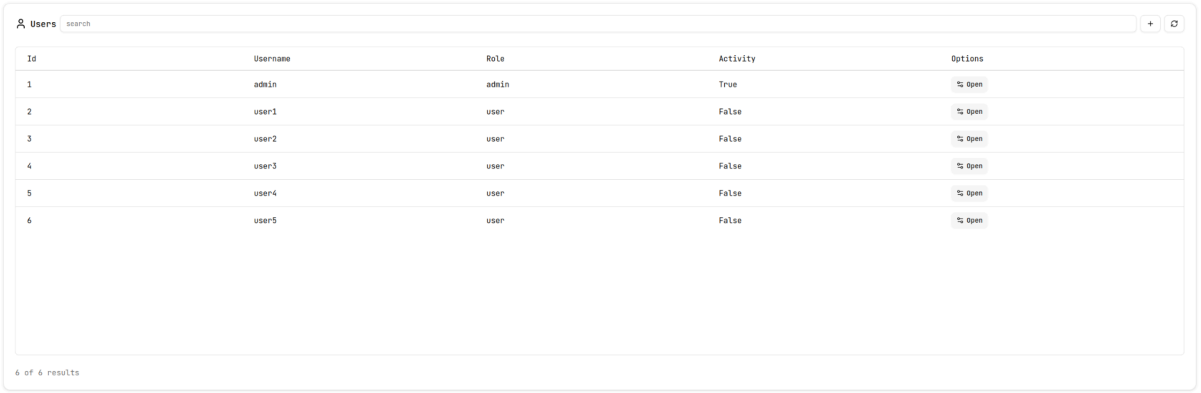
Фиг. 5.1. Начална страница



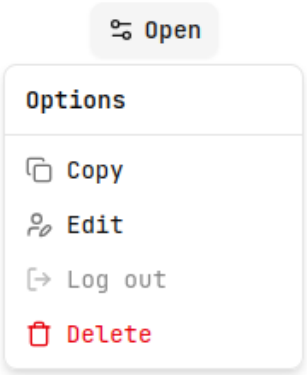
Фигура 5.2. Страница за автентикация и регистрация



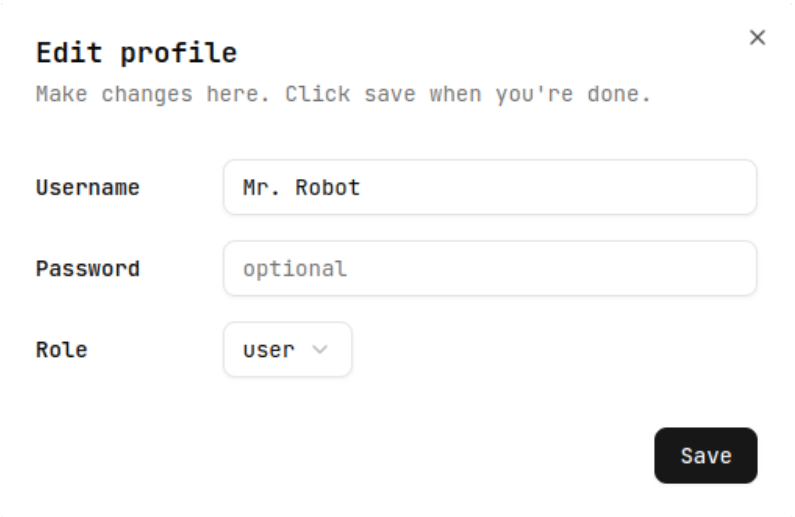
Фигура 5.3. Начална страница след автентикация на потребителя



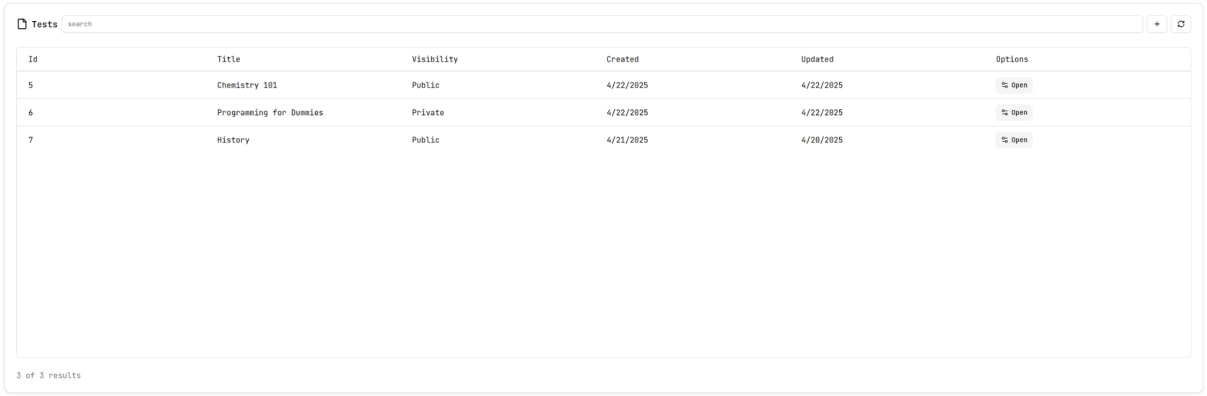
Фигура 5.4. Компонент за управление на потребители



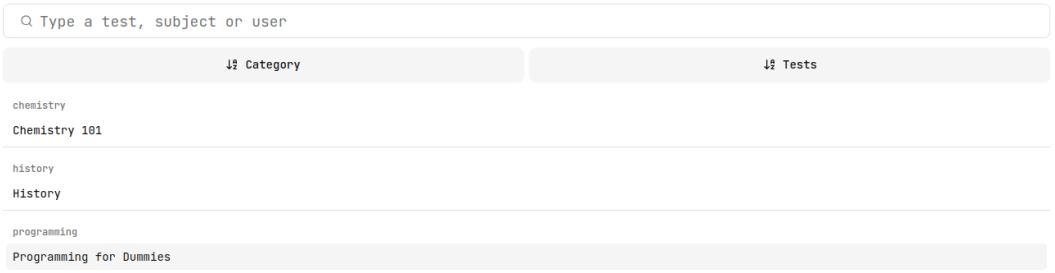
Фигура 5.5. Меню с възможности за манипулация на потребител



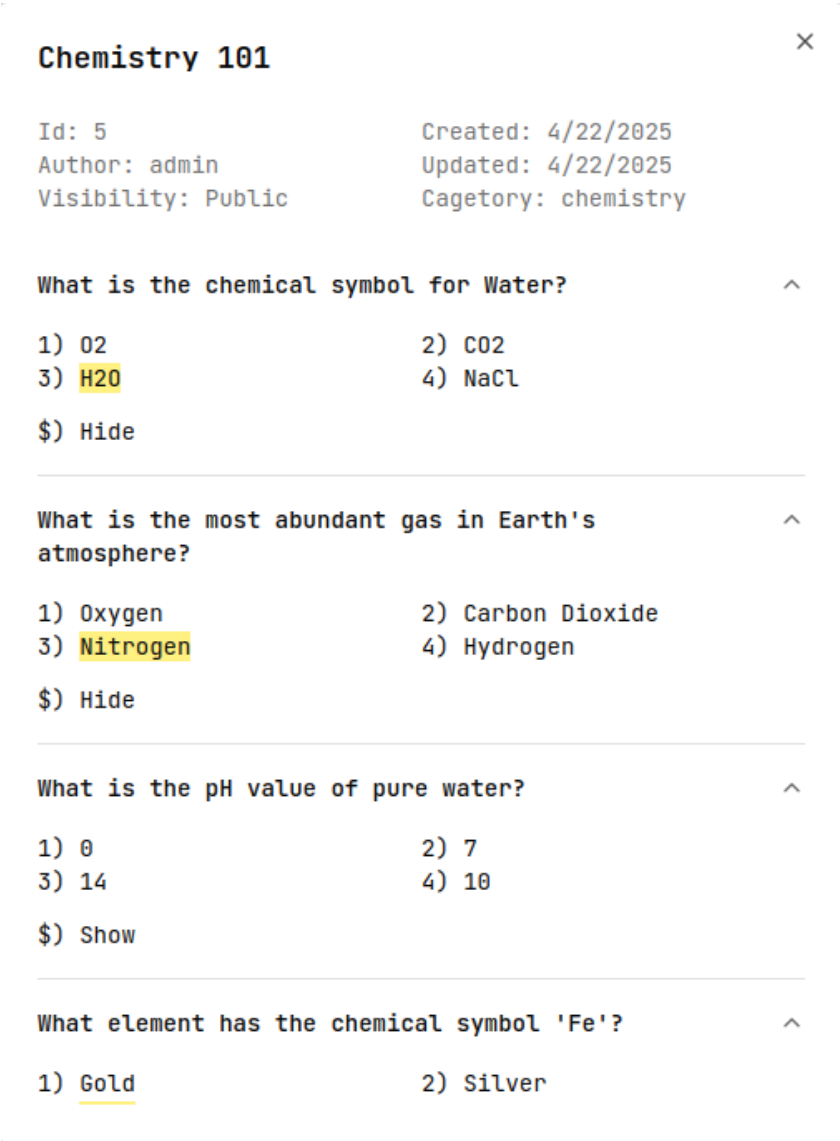
Фигура 5.6. Актуализиране информацията на потребител



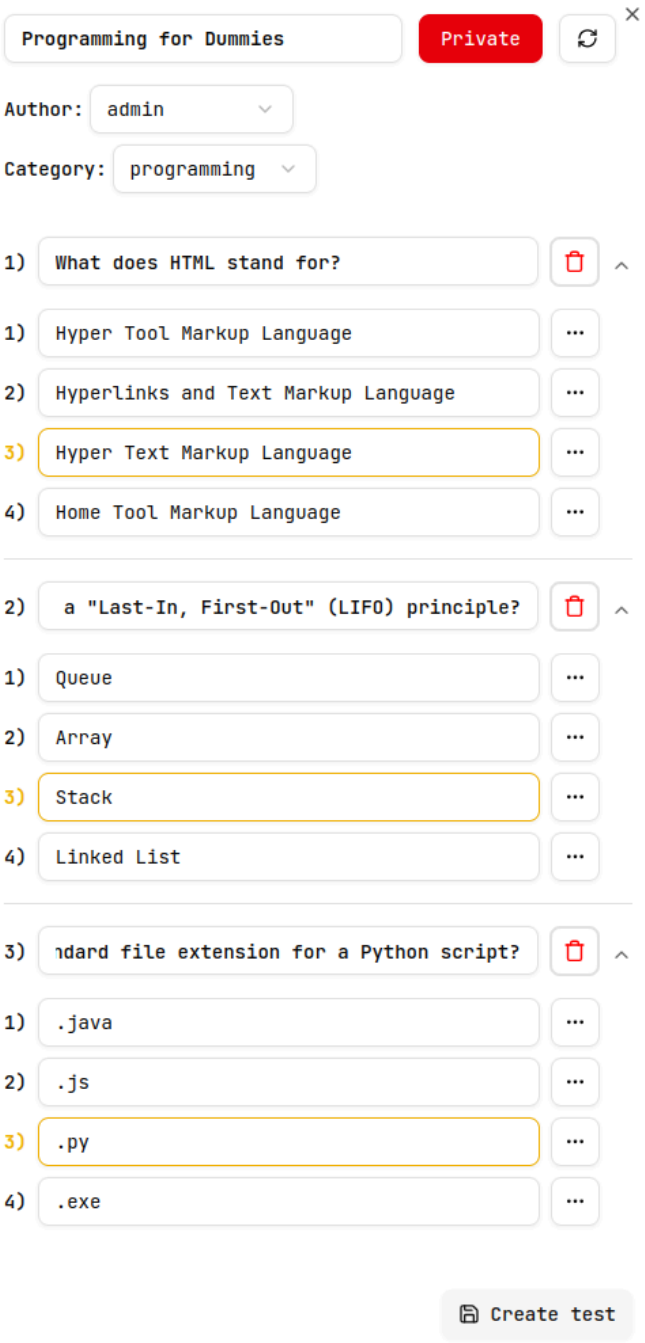
Фигура 5.7. Компонент за управление на тестове



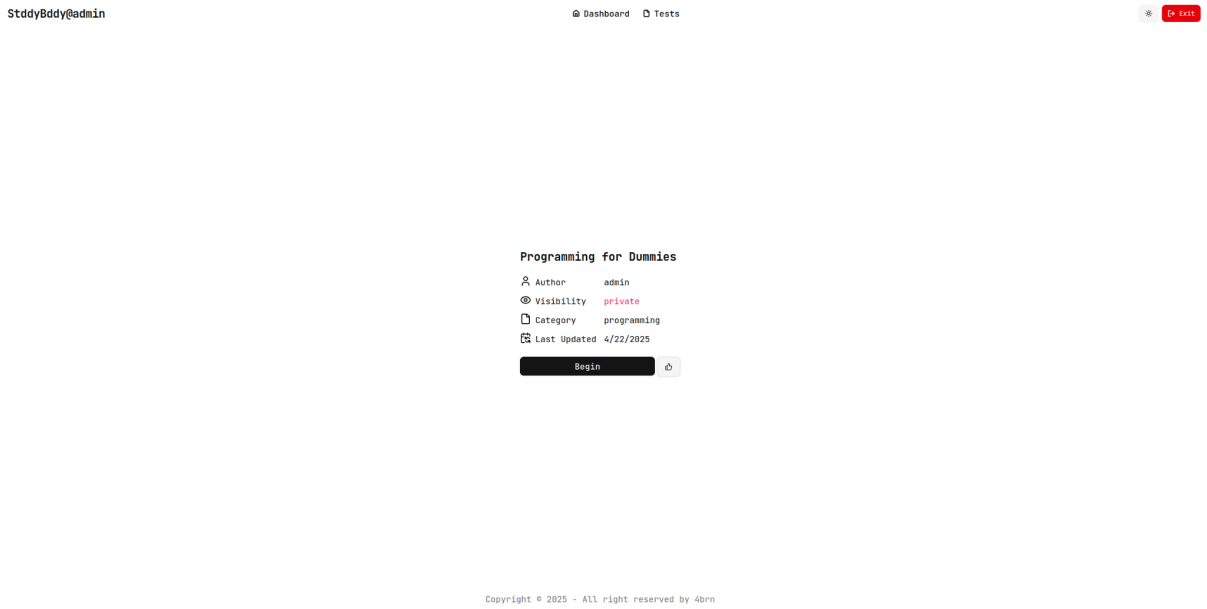
Фиг. 5.8. Търсачка на тестове



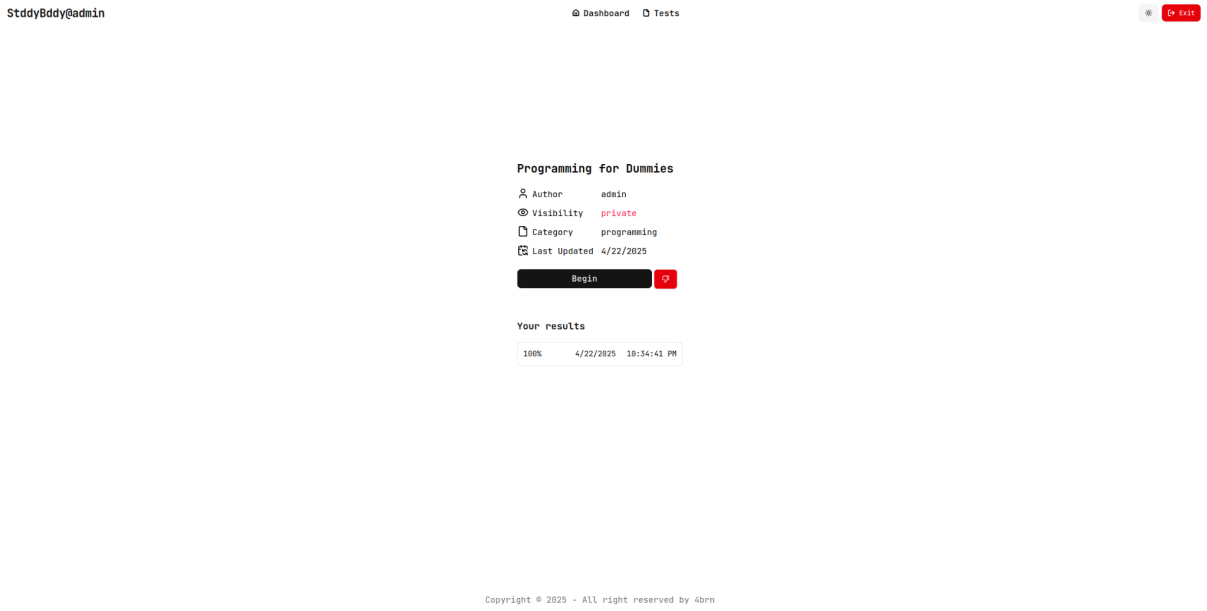
Фигура 5.9. Преглед на верните отговори на тест



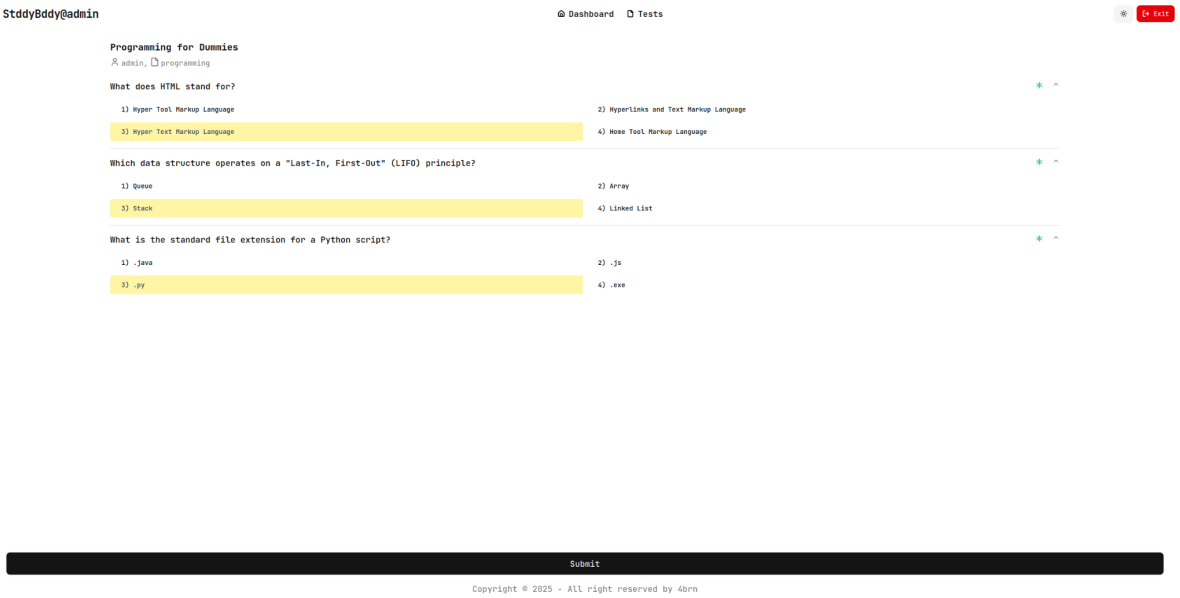
Фигура 5.10. Създаване или актуализиране на тест



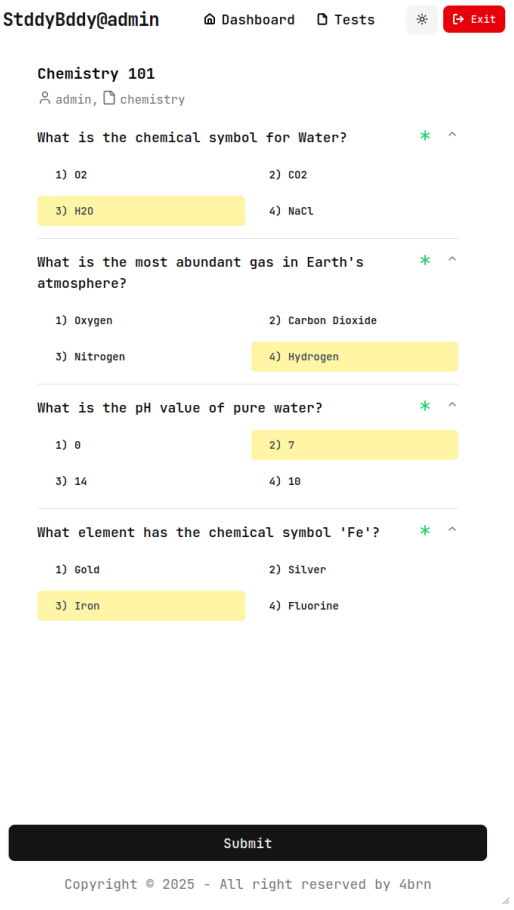
Фигура 5.11. Преглед и обща информация на тест



Фигура 5.12. Преглед и резултати на решения на даден тест



Фигура 5.12. Решаване на тест от компютър



Фигура 5.13. Решаване на тест от мобилно устройство