**Документация для тестирования серверной части приложения**

# **1. Тест план.**

# **Перечень работ.**

Перечень функциональных областей API, которые будут подвергаться тестированию:

1. Ввод данных: проверка корректности ввода данных, включая проверку на допустимые значения (например, при некорректном вводе даты или не заполненном обязательном поле должна вернуться соответствующая ошибка).
2. Авторизация в системе: возможность создать аккаунт, авторизоваться и получать данные об авторизованном пользователе.
3. Интерфейс API: можно ли делать запросы прямо на странице API.
4. Совместимость: проверка совместимости с другими приложениями, например, какие данные при подключении к API можно получить через Postman.
5. Безопасность: доступ к закрытым методам только для авторизованных пользователей, получение только информации авторизованного пользователя.
6. Документация: проверка наличия и качества документации, включая понятное описание API и всех методов.

# **Критерии качества.**

1. Надежность: способность программного продукта выполнять свои функции без сбоев в течение определенного времени или количества операций.
2. Безопасность: обеспечение защиты информации и ресурсов от несанкционированного доступа, изменений.
3. Производительность: способность программного продукта эффективно выполнять свои функции в разумные сроки и с минимальным потреблением ресурсов.
4. Масштабируемость: способность программного продукта адаптироваться к увеличению объема данных или количества пользователей без значительного снижения производительности.
5. Удобство использования: уровень удобства и простоты использования программного продукта для пользователя, включая понятный интерфейс, интуитивную навигацию и доступность функций.
6. Сопровождаемость: способность программного продукта быть легко модифицируемым, исправляемым и обновляемым без необходимости переписывания большого количества кода.
7. Поддерживаемость: возможность программного продукта быть поддержанным и обслуживаемым в течение всего его жизненного цикла, включая обновления, исправления ошибок и обучение пользователей.
8. Эффективность: способность программного продукта выполнять свои функции с минимальным использованием ресурсов, таких как память, процессорное время и сетевой трафик.

# **Оценка рисков.**

Таблица 2. Оценка рисков

|  |  |  |  |
| --- | --- | --- | --- |
| Риск | Оценка риска | Описание риска | Выход из ситуации |
| Некорректный ввод данных | Высокий | Неправильный ввод данных может привести к ошибочным результатам и недовольству пользователей. | Провести тщательную проверку ввода данных, проверяя возможность ввода пустой строки, null, отрицательных значений (для числовых полей), специальных символов, отсутствие поля в json запросе |
| Нестабильная работа приложения | Средняя | Нестабильная работа приложения может привести к потере данных и недовольству пользователей. | Провести тщательное тестирование на различных устройствах и в различных условиях использования - использовать механизмы отладки и мониторинга для выявления и исправления ошибок |
| Низкая производительность | Средняя | Низкая производительность приложения может привести к недовольству пользователей. | Оптимизировать код, использовав эффективные алгоритмы - произвести тестирование производительности и мониторинг нагрузки на приложение. |

# **Стратегия тестирования. (GUI & Ux-testing, Positive, Negative, White-box, Black-box).**

Стратегия тестирования – это план действий, который определяет, как будет проводиться тестирование приложения. Она включает в себя выбор методов и инструментов тестирования, а также определение приоритетов и целей тестирования.

GUI & UX-testing – это тестирование интерфейса пользователя и его удобства использования. Оно включает проверку навигации, доступности элементов управления и понятности сообщений об ошибках.

Positive testing – это тестирование, при котором приложение проверяется на корректную работу при вводе правильных данных и выполнении ожидаемых функций.

Negative testing – это тестирование, при котором приложение проверяется на работу при вводе неправильных или некорректных данных.

White-box testing – это тестирование, при котором проверяются внутренние компоненты и функциональность приложения. Оно включает проверку алгоритмов, обработку ошибок и безопасность.

Black-box testing – это тестирование, при котором проверяется работа приложения без знания его внутренних компонентов и функциональности. Оно включает проверку интерфейса пользователя, совместимости с различными операционными системами и браузерами, а также проверку производительности и надежности.

# **Ресурсы.**

Для разработки серверной части приложения StudyBuddy могут быть задействованы следующие ресурсы:

Человеческие ресурсы: человек, который будут заниматься разработкой API.

Механические ресурсы: это оборудование, которое будет использоваться для разработки приложения. Это рабочее место студента (АРМ) в колледже, его рабочее место дома.

Технические ресурсы: это программное обеспечение и инструменты, которые будут использоваться для разработки приложения. Это язык программирования C#, свободно распространяемая платформа для разработки динамических сайтов и веб-приложений ASP.NET, являющаяся частью NET.Framework, Nu Get пакеты для авторизации через JWT токен, для подключения к БД, для генерации кода (специальных таблиц для авторизации), и др., среда разработки Visual Studio, инструменты тестирования, наличие системы удаленных версий.

Информационные ресурсы: это данные, которые будут использоваться для разработки приложения. Видеоуроки и статьи в интернете, гайды по созданию API на ASP.NET с авторизацией через JWT токен, гайды по Entity Framework и как создавать БД через код, как документировать Swagger и др.

Организационные ресурсы: это структура и процессы, которые будут использоваться для разработки приложения. Это включает в себя управление проектом и планирование.

# **Метрики**

Точность: Программа должна правильно регистрировать пользователя в системе и выводить только данные авторизованного пользователя. Это может быть измерено путем ручного тестирования авторизации пользователя в системе и проверке получаемых данных через методы в API, требующие авторизации.

Скорость: Программа должна быть быстрой и эффективной в обработке данных.

Надежность: Программа должна быть надежной и стабильной, то есть она не должна вылетать или вызывать ошибки при обработке данных. Это может быть измерено путем тестирования программы на различных наборах данных и проверки ее стабильности.

Использование ресурсов: Программа должна потреблять минимальное количество ресурсов, таких как память и процессорное время.

# **Расписание и ключевые точки.**

Таблица 3. Расписание и ключевые точки.

|  |  |
| --- | --- |
| **Этап** | **Количество пар** |
| Разработка тестового набора и тест-кейсов | 1 |
| Реализация 10 unit-test к API | 1 |

# **Тестовый набор**

Тестовый набор – это набор тест-кейсов, в которых результат описывается предисловием, то есть очередность проводимых тестов.

* 1. **Выбор фреймворка для тестирования**

Для тестирования API будет использоваться инструмент для модульного тестирования xUnit. xUnit — это пакет с открытым исходным кодом для .NET Framework и .NET Core. Кроме фреймворков для создания и проведения юнит-тестов при тестировании часто бывают полезны такие фреймворки, которые позволяют имитировать или эмулировать какую-то функциональность или создавать мок-объекты. Подобных фреймворков существует множество, и одним из самых популярных является Moq, он и будет использоваться. Также будет использоваться модель тестов Arrange-Act-Assert, которая представляет целую парадигму тестирования, которая используется многими фреймворками юнит-тестов.

* 1. **Проверка аутентификации и авторизации при помощи модульного тестирования.**

В API присутствует система авторизации, поэтому необходимо проверить, корректно ли она работает. Проверка реализована в виде unit тестов, где для изоляции компонентов друг от друга и имитации их взаимодействия используются моки (тестовые объекты, которые имитируют зависимости), также создан тестовый контекст БД. Тестированию подвержен контроллер Account, который отвечает за обработку операций, связанных с учетными записями пользователей, таких как: регистрация, вход в систему, получение информации об авторизованном пользователе. В Таблице 1 ниже указаны тест-кейсы.

Таблица 1. Тест-кейсы для unit-тестов системы авторизации.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **№** | **Описание** | **Входные данные** | **Ожидаемый результат** | **Проверка** | **Прошел** |
| 1 | Проверка успешной регистрации нового пользователя с валидными данными | 1. Имя пользователя: Mark.  2. Почта: [markmarkovich@mail.ru](mailto:markmarkovich@mail.ru)  3. Пароль: 12345678 | – Возвращается код HTTP 200 (Ok).  – Создается новый пользователь в базе данных и возвращается. | result.Should().NotBeNull();  Assert.NotNull(newUserInContext); result.Should().BeAssignableTo<OkObjectResult>(); | + |
| 2 | Проверка, что при попытке регистрации с уже существующим именем пользователя возвращается ошибка. | 1. Имя пользователя: Mark.  2. Почта: [markmarkovich1@mail.ru](mailto:markmarkovich1@mail.ru)  3. Пароль: 12345678 | – Возвращается код HTTP 400 (BadRequest). | result.Should().NotBeNull();  result.Should().BeAssignableTo<BadRequestObjectResult>(); | + |
| 3 | Проверка успешной аутентификации пользователя с правильными логином и паролем. | 1. Почта: [markmarkovich1@mail.ru](mailto:markmarkovich1@mail.ru)  2. Пароль: 12345678 | - Возвращается код HTTP 200 (Ok). - Генерируется и возвращается JWT-токен. | result.Should().NotBeNull();  Assert.NotNull(resultDto.Token);  result.Should().BeAssignableTo<OkObjectResult>(); | + |
| 4 | Проверка, что при попытке авторизоваться с неправильными логином/паролем возвращается ошибка. | - Имя пользователя: invaliduser - Пароль: invalidpassword | - Возвращается код HTTP 401 (Unauthorized). | result.Should().NotBeNull();  result.Should().BeAssignableTo<UnauthorizedObjectResult>(); | + |
| 5 | Проверка получения информации об авторизованном пользователе | Эмуляция авторизации в системе (через моки) | - Возвращается код HTTP 200 (Ok).  - Возвращается модель данных об аккаунте | result.Should().NotBeNull();  Assert.NotNull(resultDto);  result.Should().BeAssignableTo<ActionResult<UserInfoDto>>(); | + |

* 1. **Проверка работы с данными у авторизованного пользователя.**

После авторизации в системе пользователю доступен ряд GET, POST, DELETE и PUT методов для управления информацией под своей учетной записью:

GET-запросы: используются для получения данных, например, списка экзаменов, информации о профиле пользователя, получение списка требований к предмету и др. Тесты должны проверять, что авторизованный пользователь получает только информацию, относящуюся к его аккаунту.

POST-запросы: используются для создания новых данных, например, добавления нового экзамена. Тесты должны проверять, что данные создаются корректно и привязываются к аккаунту пользователя.

DELETE-запросы: используются для удаления данных, например, удаления экзамена. Тесты должны проверять, что данные действительно удаляются.

PUT-запросы: используются для обновления данных, например, изменения информации об экзамене. Тесты должны проверять, что данные обновляются корректно.

Для проверки были написаны тест-кейсы, они указаны в Таблице 2.

Таблица 2. Тест-кейсы для unit-тестов работы с данными пользователя.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **№** | **Описание** | **Входные данные** | **Ожидаемый результат** | **Проверка** | **Прошел** |
| 1 | GET - Получение списка экзаменов пользователя | 1. Эмуляция авторизации в системе (через моки). | – Возвращается код HTTP 200 (Ok).  – Список экзаменов у пользователя | result.Should().NotBeNull();  Assert.NotNull(resultDto);  result.Should().BeAssignableTo<ActionResult<List<Exam>>>(); | + |
| 2 | POST – создание экзамена у пользователя | 1. Эмуляция авторизации в системе (через моки).  2. Объект создаваемого экзамена | – Возвращается код HTTP 200 (Ok).  – Добавленный экзамен у пользователя  – Количество экзаменов пользователя меняется в большую сторону | Assert.NotNull(resultDto);  Assert.True(countExamsBefore < countExamsAfter);  result.Should().NotBeNull();  result.Should().BeAssignableTo<ActionResult<Exam>>(); | + |
| 3 | POST – создание экзамена дубликата у пользователя | 1. Эмуляция авторизации в системе (через моки).  2. Объект создаваемого экзамена, который уже есть в БД | - Возвращается код HTTP 400 (BadRequest). | Assert.True(result.Result is BadRequestObjectResult); | + |
| 4 | DELETE –удаление экзамена у пользователя | 1. Эмуляция авторизации в системе (через моки).  2. Id экзамена, который будет удален | – Возвращается код HTTP 200 (Ok). | result.Should().NotBeNull();  result.Should().BeAssignableTo<ActionResult>(); | + |
| 5 | PUT – изменение экзамена у пользователя | 1. Эмуляция авторизации в системе (через моки).  2. Объект экзамена, который будет изменён | - Возвращается код HTTP 200 (Ok). | result.Should().NotBeNull();  result.Should().BeAssignableTo<ActionResult>(); | + |