Автор: Шарифуллин Марат Марсович

Дипломная работа по направлению QA-тестировщик

Тема: Сравнительный анализ методов тестирования фронтенда и бэкенда: Исследование особенностей и различий между тестированием фронтенда (frontend) и бэкенда (backend) веб-приложений, а также разработка современных подходов к их интеграционному тестированию.

ведение;

Выбор темы «Сравнительный анализ методов тестирования фронтенда и бэкенда» обоснован тем, что **тестирование веб-приложений — важный процесс, который требует от тестировщиков знаний и навыков как на фронтенде, так и на бэкенде**.

Фронтенд — это всё, что пользователь видит на сайте и с чем он взаимодействует. Бэкенд — это серверная часть веб-приложения, которая работает с базами данных, обрабатывает запросы от фронтенда и возвращает ответы. Эти две части работают вместе и зависят друг от друга, поэтому необходимо внимательно тестировать каждый компонент приложения, уделяя особое внимание проверке взаимодействия между фронтендом и бэкендом.

Таким образом, сравнительный анализ методов тестирования фронтенда и бэкенда позволяет обеспечить общую функциональность приложения и выявить возможные ошибки на разных уровнях разработки.

В современном мире веб-разработки тестирование является неотъемлемой частью процесса создания качественных программных продуктов. Веб-приложения состоят из двух основных компонентов: фронтенда (клиентская часть) и бэкенда (серверная часть). Каждый из этих компонентов требует специфических методов тестирования, которые учитывают их особенности и функциональность. Цель данной работы — провести сравнительный анализ методов тестирования

фронтенда и бэкенда, а также разработать современные подходы к их интеграционному тестированию.

Глава 1: Основы тестирования веб-приложений

1.1. Определение тестирования

Тестирование — это **процесс оценки качества продукта или его компонентов с целью установить, соответствует ли он требованиям и ожиданиям пользователей**. В контексте веб-разработки, тестирование включает проверку кода, интерфейсов, функциональности и других аспектов веб-сайта или приложения на наличие ошибок, несоответствий и нежелательного поведения.

1.2. Важность тестирования в разработке ПО

Важность тестирования в разработке программного обеспечения (ПО) заключается в следующем:

Обеспечение качества. Тестирование помогает обнаружить и исправить ошибки, дефекты и неправильное поведение продукта. Это повышает удовлетворённость пользователей и уменьшает вероятность возникновения проблем в реальной эксплуатации.

Выявление проблем. Тестирование обнаруживает ошибки и несоответствия между ожидаемым поведением программы и её фактическим поведением. Это помогает разработчикам и команде проекта улучшить продукт, исправив ошибки до выпуска на рынок.

Экономия времени и ресурсов. Раннее тестирование и выявление ошибок в начале разработки помогает избежать дорогостоящих исправлений и модификаций в более поздние этапы. Также тестирование позволяет уменьшить затраты на техническую поддержку и обслуживание после выпуска продукта.

Улучшение пользовательского опыта. Тестирование помогает убедиться, что программный продукт соответствует потребностям и ожиданиям пользователей. Это позволяет создать положительный и удовлетворяющий опыт использования продукта, что немаловажно для его успешной адаптации на рынке.

Соблюдение требований и стандартов. Тестирование помогает удостовериться, что программный продукт соответствует заданным требованиям и стандартам безопасности, производительности и функциональности.

Снижение рисков. Тестирование помогает уменьшить риски неблагоприятного влияния программного продукта на бухгалтерские, финансовые или операционные процессы организации. Это способствует предотвращению потенциальных проблем и ущерба для бизнеса.

1.3. Основные виды тестирования

Основные виды тестирования программного обеспечения (ПО) включают:

Функциональное тестирование. Направлено на проверку того, что именно способна выполнить программа, то есть какими возможностями она обладает. Включает в себя модульное, интеграционное и системное тестирование.
Нефункциональное тестирование. Направлено на определение эффективности программного продукта. Рассматриваются нефункциональные параметры, к числу которых относятся: безопасность, производительность, масштабируемость, совместимость и т. д.. Включает нагрузочное, на проникновение, на совместимость, стресс-тестирование, на отказоустойчивость, пользовательское и на восстановление.
Системное тестирование. Проверяет всю систему в целом. Включает проверку пользовательского интерфейса, баз данных и сетевых соединений.
Приёмочное тестирование. Проводится для того, чтобы убедиться, что ПО соответствует требованиям заказчика. Обычно его выполняют конечные пользователи или представители заказчика.
Юзабилити тестирование. Проверяет, насколько удобно и интуитивно понятно ПО для конечных пользователей. Например, насколько легко пользователю найти нужные функции в интерфейсе.
Тестирование совместимости. Проверяет, как ПО работает на различных платформах, устройствах и браузерах. Например, проверка работы веб-приложения в разных браузерах.

Выбор основных видов тестирования зависит от конкретных требований проекта.

Глава 2: Тестирование фронтенда

2.1. Особенности фронтенда Особенности фронтенда включают:

Создание видимой части сайта, с которой взаимодействуют пользователи. Это меню, слайдеры, текстовые блоки, футер и другие элементы.
Использование трёх языков для создания элементов страницы. Это HTML, CSS и JavaScript.
Адаптивность сайта. Контент подстраивается под разные разрешения и хорошо выглядит независимо от размера дисплея.
Постоянное развитие. Необходимые для работы инструменты, языки, функции часто обновляются или меняются.
Взаимодействие с нетехническими специалистами. Фронтендер работает с дизайнерами, UX-аналитиками, маркетологами, менеджерами продукта.

Фронтенд связан с бизнес-логикой продукта, но её разработкой занимаются бэкенд-программисты.

2.2. Методы тестирования фронтенда **Некоторые методы тестирования фронтенда:**

Юнит-тестирование. Проверка отдельных компонентов приложения, таких как функции и модули, чтобы убедиться, что они функционируют правильно.
Интеграционное тестирование. Проверка взаимодействия между различными компонентами приложения, чтобы убедиться, что они работают вместе правильно.
Визуальное тестирование. Проверка пользовательского интерфейса приложения, чтобы убедиться, что он выглядит правильно в разных средах и устройствах.
Тестирование производительности. Проверка скорости и отзывчивости приложения, чтобы убедиться, что оно хорошо работает в различных условиях.
Тестирование доступности. Проверка доступности приложения для пользователей с ограниченными физическими возможностями.
Тестирование безопасности. Проверка функций безопасности приложения, чтобы убедиться, что пользовательская информация защищена.
Сквозное (end-to-end) тестирование. Проверка пользовательских путей по приложению, воспроизведение действий реальных пользователей.

- Юнит-тестирование **Юнит-тестирование** (модульное тестирование) — **это процесс в программировании, позволяющий проверить на корректность отдельные модули исходного кода программы**.
Под словом «юнит» чаще понимают функцию, метод или класс в исходном коде.

Цель юнит-тестирования — изолировать отдельные части программы и показать, что по отдельности эти части работоспособны.

Юнит-тестирование позволяет:

избежать ошибок или быстро исправить их при обновлении или дополнении ПО новыми компонентами, не тратя время на проверку программного обеспечения целиком;
проверить, не привело ли очередное изменение кода к регрессии, то есть к появлению ошибок в уже оттестированных местах программы, а также облегчает обнаружение и устранение таких ошибок.

Два основных вида юнит-тестирования — ручное и автоматическое:

Ручное тестирование более гибкое, его результаты можно анализировать подробнее, но оно занимает много времени.
Автоматизированное тестирование использует специальные инструменты для написания и выполнения тестов. Автоматизированные тесты выполняются быстрее, чем ручные,

- Интеграционное тестирование **Интеграционное тестирование** — это этап в проверке качества программного обеспечения, который следует за модульным тестированием. На этом этапе отдельные

модули или компоненты приложения объединяют и проверяют на совместимость друг с другом.

Главная цель интеграционного тестирования — подтвердить, что различные программные компоненты, модули и подсистемы работают вместе как единая система, обеспечивая требуемую функциональность и производительность.

Некоторые задачи интеграционного тестирования:

проверка взаимодействия модулей;
обеспечение совместимости;
раннее обнаружение проблем;
повышение общей надёжности системы;
повышение качества ПО за счёт выявления и устранения ошибок до того, как это станет более сложным и дорогостоящим процессом.

Существует несколько подходов к интеграционному тестированию:

Большой взрыв (Big Bang тестирование). Все модули объединяются одновременно и затем проверяются.
Нисходящее. Тестирование начинается с верхнего уровня архитектуры системы и постепенно спускается вниз.
Восходящее. Процесс начинается с нижних уровней и движется вверх.
Смешанное (песочные часы или сэндвич). Комбинация двух предыдущих методов.

Выбор инструментов для интеграционного тестирования зависит от приложения, целей и инфраструктуры.

-**Функциональное тестирование** — вид тестирования, при котором проверяется, **что делает программный продукт**. Например, проверка API, базы данных, пользовательского интерфейса, функциональности тестируемого продукта на соответствие спецификациям и бизнес-требованиям.

Цели функционального тестирования:

Обнаружить дефекты. Выявить ошибки и несоответствия в функциональности на ранних стадиях разработки.
Проверить соответствие требованиям. Убедиться, что каждая опция системы выполняется согласно требованиям, которые были заданы заказчиком или разработчиком.
Улучшить пользовательский опыт. Гарантировать интуитивно понятный интерфейс и корректное поведение программы при различных сценариях использования.
Устранить регрессии. Проверить, что изменения в коде не нарушают работоспособность уже существующих функций.

Гарантировать безопасность информации. Убедиться, что система защищает пользовательские сведения от утечек или несанкционированного доступа.

Этапы функционального тестирования:

Определение и анализ, какую функциональность необходимо протестировать. Перед началом нужно изучить тестируемый функционал: какие у него требования, как он должен работать, как пользователь будет его использовать. 1

Написание тест-кейсов. В них тестировщик пошагово описывает сценарий проверки определённой функциональности.

Подготовка тестовых данных. Для тестирования используются данные, которые максимально приближены к тем, что могут использовать пользователи. Сбор тестовых данных основывается на требованиях.

Проведение тестирования. Происходит сравнение фактического результата и ожидаемого. 1

Составление отчёта по результатам тестирования. По завершении тестирования необходимо собрать отчёт с результатами прогонки тестов, списком багов и рекомендациями по улучшению продукта.

Некоторые виды функционального тестирования:

Модульное тестирование. Выполняется разработчиками на этапе разработки приложения. Цель модульного тестирования заключается в проверке работы отдельной функциональности.

Интеграционное тестирование. Проверка того, что модули работают корректно как группа.

Системное тестирование. Проводится на полноценной и полностью интегрированной системе для подтверждения, что система работает согласно исходным требованиям.

Регрессионное тестирование. Проводится после того, как в приложение были внесены какие-либо изменения.

Sanity тестирование (санитарное тестирование). Проводится, когда тестировщик получает новую сборку с минорными изменениями.

- Тестирование пользовательского интерфейса (UI) **Тестирование пользовательского интерфейса (UI)** — это проверка корректности работы и удобства использования интерфейса приложения. Его цель — найти проблемы ещё до того, как с ними столкнутся пользователи.

Некоторые аспекты, которые проверяют при UI-тестировании:

Внешний вид интерфейса. Соответствие дизайна макетам, правильность отображения на разных устройствах и разрешениях, соблюдение стилей и цветовой гаммы.

Элементы управления. Корректная работа кнопок, полей ввода, выпадающих списков и других элементов управления, а также их доступность для пользователя.

Навигация. Простота и понятность перемещения между разделами приложения, наличие хлебных крошек, работоспособность ссылок и меню.

Отзывчивость. Скорость загрузки страниц, отсутствие зависаний и ошибок при взаимодействии с интерфейсом.

Сообщения об ошибках и подсказки. Наличие и корректность отображения сообщений об ошибках, а также подсказок и инструкций для пользователя.

Есть два основных вида UI-тестирования: ручное и автоматизированное. При ручном тестировании QA-инженер самостоятельно проверяет, насколько корректно выглядит интерфейс и правильно ли работает тот или иной компонент. Для этого он воспроизводит действия пользователя. При автоматизированном тестировании QA-инженер пишет и запускает скрипты на основе тестового сценария. Эти скрипты имитируют взаимодействие пользователя с интерфейсом и проверяют код на ошибки.

Процесс UI-тестирования включает в себя шесть этапов: сбор требований, определение объектов тестирования и основных шагов, разработка тестовых сценариев, выбор методов и техник, тестирование и составление баг-репортов.

- Тестирование производительности **Тестирование производительности** — это вид тестирования программного обеспечения, направленный на определение его способности работать с определёнными рабочими нагрузками и в определённых условиях. **Основная цель тестирования производительности** — выявить проблемы, связанные с быстродействием, надёжностью и стабильностью системы, а также определить возможные узкие места и точки отказа.

Некоторые виды тестирования производительности:

Нагрузочное тестирование. Проверка системы на работоспособность под определённой нагрузкой, например, при одновременной работе определённого количества пользователей или при выполнении определённого количества запросов. 3
Стресс-тестирование. Определение предельных рабочих условий системы, при которых она продолжает функционировать, но с ограничениями по производительности, а также выявление точек отказа системы.
Тестирование стабильности. Проверка системы на способность работать без сбоев и с заданной производительностью на протяжении длительного времени.
Тестирование масштабируемости. Определение способности системы адекватно функционировать при увеличении рабочих нагрузок или количества пользователей.

Для проведения тестирования производительности существует множество инструментов, таких как JMeter, LoadRunner, Gatling, WebLOAD. Выбор инструмента зависит от требований к проекту, опыта тестировщика и предпочтений команды.

- Тестирование кроссбраузерности **Кроссбраузерное тестирование** — это процесс проверки веб-приложений на совместимость с различными веб-браузерами. Цель такого тестирования — убедиться, что веб-сайт или веб-приложение корректно отображается и функционирует во всех популярных браузерах, таких как Google Chrome, Mozilla Firefox, Safari, Microsoft Edge и Яндекс.Браузер.

Основные этапы кроссбраузерного тестирования:

Определение целевых браузеров и устройств. Это зависит от целевой аудитории.
Разработка тестовых сценариев. Они должны быть максимально подробными и охватывать все возможные варианты использования сайта.
Выполнение тестов. Запустить тесты на всех выбранных браузерах и устройствах, обратить внимание на любые различия в отображении и функциональности.
Анализ результатов и исправление ошибок. Проанализировать результаты тестов и исправить обнаруженные ошибки. Повторить тестирование после внесения изменений, чтобы убедиться, что проблемы устранены.

Для кроссбраузерного тестирования можно использовать различные инструменты, например:

BrowserStack. Облачная платформа для тестирования, которая позволяет тестировать веб-приложения на реальных устройствах и браузерах.
CrossBrowserTesting. Один из популярных онлайн-сервисов для кроссбраузерного тестирования, позволяющий проверять веб-приложения на различных платформах.
Sauce Labs. Облачный сервис для тестирования веб-приложений на различных браузерах и устройствах.

2.3. Инструменты для тестирования фронтенда Несколько инструментов для тестирования фронтенда:

Cypress. Фреймворк для автоматизированного тестирования веб-приложений. Поддерживает перезагрузку тестов в реальном времени и сохраняет снапшоты приложений на разных стадиях тестирования.
Jest. Фреймворк для тестирования приложений на JavaScript, который позволяет проводить тесты без предварительной настройки. Тестовая среда изолирована, результаты одного теста не влияют на результаты другого.
Playwright. Инструмент для кросс-браузерного тестирования, который имитирует особенности работы различных девайсов и моделирует разные браузерные условия.
PhantomCSS. Запоминает состояние веб-страницы, позволяя отслеживать регрессии (например, пропавшую кнопку или другой элемент).

BrowserStack. Сервис для кроссбраузерного тестирования на реальных устройствах.
Selenium IDE. Работает как надстройка к браузеру, умеет заполнять поля, нажимать кнопочки и так далее.

Выбор инструмента зависит от особенностей фронтенда в проекте и требований к тестированию.

- Jest
- Mocha
- Cypress
- Selenium

Глава 3: Тестирование бэкенда **Тестирование бэкенда** — это процесс проверки функциональности и производительности серверной части программного обеспечения. Он может включать в себя тестирование базы данных, проверку работоспособности API и других сервисов, таких как кэширование или очереди сообщений.
Некоторые типы тестирования бэкенда:

Структурное. Валидация всех элементов в репозитории данных (то есть в хранилище данных, в БД). Например, тестирование схем, таблиц и колонок, аутентификации и авторизации.
Функциональное. Проверка на соответствие требованиям транзакций и операций, выполняемых конечными пользователями. Например, тестирование чёрного ящика (проверка функциональности и целостности базы данных) и белого ящика (проверка внутренней структуры БД).
Нефункциональное. Чаще всего это нагрузочное тестирование, стресс-тестирование и проверка соблюдения уровня устойчивости, стабильности и масштабируемости бэкенд-части приложения.

Этапы тестирования бэкенда:

Создание и настройка тестового окружения. Когда написание кода приложения завершено или близко к завершению, наступает первый этап тестирования бэкенда: создание тестового окружения и подбор инструментов.
Генерация тест-кейсов. Когда собрана команда, подобраны инструменты и готов план, пишутся тест-кейсы, направленные на проверку выполнения бизнес-требований.
Выполнение тест-кейсов. То есть непосредственно тестирование. Процесс требует высокой квалификации и опыта.
Анализ результатов. По результатам выполнения инструменты могут подсказывать, как решить ту или иную проблему, обнаруженную в процессе.

Для тестирования бэкенда используются различные протоколы, такие как HTTP, HTTPS, SOAP и другие. Например, для проверки API применяются Postman, SoapUi, Swagger.

3.1. Особенности бэкенда собенности бэкенда включают:

Работа на стороне сервера. Бэкенд обеспечивает функциональность и обработку данных.
Управление базами данных. Backend организует хранение данных, их чтение, быстрый доступ и резервное копирование.
Подключение внешних сервисов и ресурсов. Например, API платёжных систем, рекомендательных сервисов, систем аналитики.
Продумывание системы безопасности. Бэкенд защищает ресурс от хакерских атак.
Оптимизация и масштабирование сайтов. Современные сайты должны грамотно расходовать ресурсы и поддаваться изменениям.

От того, насколько эффективно и надёжно функционирует бэкенд, зависит пользовательский опыт и успех веб-приложения.

3.2. Методы тестирования бэкенда **Некоторые методы тестирования бэкенда:**

Модульное тестирование. Фокус на отдельных модулях или компонентах кода бэкенда в изоляции. Цель — убедиться, что каждая уникальная единица работает так, как задумано.
Интеграционное тестирование. Проверка, что разные компоненты бэкенда работают вместе так, как ожидается. Также проверяется корректность работы точек интеграции между бэкенд-системами и базами данных.
Тестирование производительности. Проверка, что бэкенд может обрабатывать ожидаемые пользовательские нагрузки и запросы. Фокус на таких показателях, как время ответа, пропускная способность, использование ресурсов и стабильность при больших нагрузках.
Тестирование безопасности. Направлено на поиск уязвимостей кода или системы бэкенда, которые могут привести к эксплойтам. Оценивается безопасность API, базы данных, сети и проверка ввода.
Структурное тестирование. Валидация всех элементов в репозитории данных (то есть в хранилище данных, в БД). Например, тестирование схем, таблиц и колонок.

Также к методам тестирования бэкенда можно отнести **нагрузочное и стресс-тестирование** — проверку соблюдения уровня устойчивости, стабильности и масштабируемости бэкенд-части приложения.

- Юнит-тестирование **Юнит-тестирование** (модульное тестирование) — **это процесс в программировании, позволяющий проверить на корректность отдельные модули исходного кода программы**.

Под словом «юнит» чаще понимают функцию, метод или класс в исходном коде.

Цель юнит-тестирования — изолировать отдельные части программы и показать, что по отдельности эти части работоспособны.

Юнит-тестирование позволяет:

избежать ошибок или быстро исправить их при обновлении или дополнении ПО новыми компонентами, не тратя время на проверку программного обеспечения целиком;
проверить, не привело ли очередное изменение кода к регрессии, то есть к появлению ошибок в уже оттестированных местах программы.

Два основных вида юнит-тестирования — ручное и автоматическое:

Ручное тестирование более гибкое, его результаты можно анализировать подробнее, но оно занимает много времени.
Автоматизированное тестирование использует специальные инструменты для написания и выполнения тестов. Автоматизированные тесты выполняются быстрее, чем ручные, а риск пропустить ошибки в них меньше.

- Интеграционное тестирование **Интеграционное тестирование** — это этап в проверке качества программного обеспечения, который следует за модульным тестированием. На этом этапе отдельные модули или компоненты приложения объединяют и проверяют на совместимость друг с другом.
Главная цель интеграционного тестирования — подтвердить, что различные программные компоненты, модули и подсистемы работают вместе как единая система, обеспечивая требуемую функциональность и производительность.

Некоторые задачи интеграционного тестирования:

проверка взаимодействия модулей;
обеспечение совместимости;
раннее обнаружение проблем;
повышение общей надёжности системы;
повышение качества ПО за счёт выявления и устранения ошибок до того, как это станет более сложным и дорогостоящим процессом.

Существует несколько подходов к интеграционному тестированию:

Интеграционное тестирование «большого взрыва». Этот вид тестов предполагает одновременную интеграцию всех программных модулей и их тестирование как целостной системы.

Нисходящее. Тестирование начинается с верхнего уровня архитектуры системы и постепенно спускается вниз.
Восходящее. Процесс начинается с нижних уровней и движется вверх. 2
Смешанное (песочные часы или сэндвич). Комбинация двух предыдущих методов.

Выбор подхода зависит от особенностей приложения и текущей задачи.

- Функциональное тестирование **Функциональное тестирование** — вид тестирования, при котором проверяется, **что делает программный продукт**. Например, проверка API, базы данных, пользовательского интерфейса, функциональности тестируемого продукта на соответствие спецификациям и бизнес-требованиям.

Цели функционального тестирования:

Обнаружить дефекты. Выявить ошибки и несоответствия в функциональности на ранних стадиях разработки.
Проверить соответствие требованиям. Убедиться, что каждая опция системы выполняется согласно требованиям, которые были заданы заказчиком или разработчиком.
Улучшить пользовательский опыт. Гарантировать интуитивно понятный интерфейс и корректное поведение программы при различных сценариях использования.
Устранить регрессии. Проверить, что изменения в коде не нарушают работоспособность уже существующих функций.
Гарантировать безопасность информации. Убедиться, что система защищает пользовательские сведения от утечек или несанкционированного доступа.

Этапы функционального тестирования:

Определение и анализ, какую функциональность необходимо протестировать. Перед началом нужно изучить тестируемый функционал: какие у него требования, как он должен работать, как пользователь будет его использовать.
Написание тест-кейсов. В них тестировщик пошагово описывает сценарий проверки определённой функциональности.
Подготовка тестовых данных. Для тестирования используются данные, которые максимально приближены к тем, что могут использовать пользователи. Сбор тестовых данных основывается на требованиях.
Проведение тестирования. Происходит сравнение фактического результата и ожидаемого.
Составление отчёта по результатам тестирования. По завершении тестирования необходимо собрать отчёт с результатами прогонки тестов, списком багов и рекомендациями по улучшению продукта.

Некоторые виды функционального тестирования:

Модульное тестирование. Выполняется разработчиками на этапе разработки приложения. Цель модульного тестирования заключается в проверке работы отдельной функциональности.
Интеграционное тестирование. Проверка того, что модули работают корректно как группа.
Системное тестирование. Проводится на полноценной и полностью интегрированной системе для подтверждения, что система работает согласно исходным требованиям.
Регрессионное тестирование. Проводится после того, как в приложение были внесены какие-либо изменения.
Sanity тестирование (санитарное тестирование). Проводится, когда тестировщик получает новую сборку с минорными изменениями.

- Тестирование API **Тестирование API (программного интерфейса приложений)** — это ручная или автоматическая проверка обмена данными между двумя модулями программы, разными приложениями, веб-сервисами и серверами. Оно помогает выявить ошибки и оценить общую работоспособность системы.

Некоторые виды тестирования API:

Тестирование методов. Проверяется каждый метод API по отдельности: входные данные, выполнение операций и вывод результатов.
Тестирование взаимодействий. Проверяется взаимодействие API с другими API, компонентами программы и сервисами. **Тестирование авторизации и аутентификации**. Проверяется доступ к API: как работают механизмы авторизации, кто и к каким функциям и данным имеет доступ.
Тестирование обработки ошибок. Проверяется поведение API в случае непредвиденных ситуаций и ошибок — например, передачи некорректных данных.
Тестирование производительности. Проверяется работа API при повышенных нагрузках — его пропускная способность и производительность.
Тестирование безопасности. Проверяются меры безопасности API и проводятся тесты на проникновение для установления возможных уязвимостей.

Для тестирования API можно использовать различные инструменты, например:

Postman. Позволяет создавать, отправлять и тестировать HTTP-запросы и получать ответы от API.
SoapUI. Позволяет тестировать и отлаживать SOAP и REST API: создавать и отправлять запросы, автоматизировать тестирование, генерировать тестовые отчёты, мониторить производительность.
JMeter. Позволяет отправлять HTTP-запросы и проводить нагрузочное тестирование, чтобы проверить, как API справляется с высокими нагрузками, насколько оно производительно и масштабируемо.

- Тестирование производительности **Тестирование производительности** — это вид тестирования программного

обеспечения, направленный на определение его способности работать с определёнными рабочими нагрузками и в определённых условиях.
Основная цель тестирования производительности — выявить проблемы, связанные с быстродействием, надёжностью и стабильностью системы, а также определить возможные узкие места и точки отказа.

Некоторые виды тестирования производительности:

> **Нагрузочное тестирование**. Проверка системы на работоспособность под определённой нагрузкой, например, при одновременной работе определённого количества пользователей или при выполнении определённого количества запросов.
> **Стресс-тестирование**. Определение предельных рабочих условий системы, при которых она продолжает функционировать, но с ограничениями по производительности, а также выявление точек отказа системы.
> **Тестирование стабильности**. Проверка системы на способность работать без сбоев и с заданной производительностью на протяжении длительного времени.
> **Тестирование масштабируемости**. Определение способности системы адекватно функционировать при увеличении рабочих нагрузок или количества пользователей.

Для проведения тестирования производительности существует множество инструментов, таких как JMeter, LoadRunner, Gatling, WebLOAD. Выбор инструмента зависит от требований к проекту, опыта тестировщика и предпочтений команды.

- Тестирование безопасности **Тестирование безопасности** — это процесс оценки информационных систем, сетей и приложений с целью выявления уязвимостей, которые могут быть использованы злоумышленниками.
Основная цель такого тестирования — не просто найти «дыры» в безопасности, но и понять, насколько система устойчива к различным типам атак.

Некоторые виды тестирования безопасности:

> **Тестирование на проникновение (пентестинг)**. Пентестеры пытаются найти все возможные способы проникнуть в систему — от банальных SQL-инъекций до сложных многоступенчатых атак с использованием социальной инженерии.
> **Тестирование на уязвимости**. Специальные инструменты сканируют систему на предмет известных уязвимостей — от устаревших версий библиотек до небезопасных конфигураций.
> **Тестирование безопасности приложений**. Этот вид тестирования начинается ещё на этапе разработки. Проверяется код на наличие потенциальных уязвимостей, анализируется архитектура приложения и тестируются все возможные входные точки.

Тестирование безопасности сетей. Проверяется вся сетевая инфраструктура: открытые порты, незащищённые сервисы, слабые пароли на сетевом оборудовании.

После завершения тестирования специалисты составляют отчёт, который включает в себя описание найденных уязвимостей и рекомендации по их устранению.

Некоторые популярные инструменты для проведения нагрузочного и стресс-тестирования веб-приложений и сайтов:

Apache JMeter. Инструмент с открытым исходным кодом поддерживает множество протоколов, включая HTTP, HTTPS, FTP, JDBC и SOAP. JMeter предоставляет удобный интерфейс для создания и выполнения тестов, а также генерирует подробные отчёты.
LoadRunner. Мощный коммерческий инструмент от Micro Focus поддерживает широкий спектр протоколов и предоставляет обширные возможности для создания сценариев тестирования. LoadRunner также интегрируется с различными системами CI/CD.
Gatling. Инструмент с открытым исходным кодом, разработанный на языке Scala. Он предназначен для тестирования производительности веб-приложений и API. Gatling предоставляет удобный DSL (Domain-Specific Language) для создания сценариев тестирования и генерирует подробные отчёты.
k6. Современный инструмент для тестирования производительности, разработанный на языке Go. Он ориентирован на разработчиков и DevOps-инженеров и предоставляет простой в использовании интерфейс для создания сценариев тестирования на языке JavaScript.
Artillery. Инструмент с открытым исходным кодом для тестирования производительности, написанный на языке JavaScript. Он поддерживает тестирование HTTP, WebSocket и Socket.io приложений и предоставляет удобный интерфейс для создания сценариев тестирования.

Выбор инструмента зависит от конкретных требований проекта.

3.3. Инструменты для тестирования бэкенда

- Postman
- JUnit
- pytest
- SoapUI

Глава 4: Сравнительный анализ методов тестирования фронтенда и бэкенда

4.1. Сравнение подходов к тестированию **Ручное тестирование**.

Тестировщики выполняют тестовые сценарии вручную, взаимодействуя с приложением так, как это будет делать конечный пользователь. **Преимущества**:

Интуиция и гибкость. Тестировщик способен выявлять ошибки и аномалии, которые могут быть упущены автоматизированными скриптами. **Тестирование пользовательского интерфейса (UI)**. Для проверки внешнего вида и удобства использования интерфейса ручное тестирование зачастую эффективнее, так как тестировщик оценивает визуальные элементы, удобство навигации и общий пользовательский опыт. **Адаптивность**. Ручное тестирование подходит для тестирования сложных и нестандартных сценариев, где автоматизация может быть сложной и дорогостоящей. **Недостатки**: большие временные затраты, субъективность результатов, ограниченная масштабируемость.

Автоматическое тестирование. Для выполнения тестов используются специализированные инструменты и скрипты. **Преимущества**:

Скорость и эффективность. Автоматизация позволяет выполнять тесты намного быстрее, особенно в случае большого количества тестов. **Повторяемость**. Автоматические тесты всегда выполняются одинаково, что обеспечивает стабильность и исключает человеческий фактор. **Недостатки**: высокая стоимость начальной разработки, ограниченность сценариев (автоматические тесты хорошо справляются с повторяющимися задачами, но они менее эффективны при тестировании творческих, непредсказуемых или сложных пользовательских сценариев).

Комбинированный подход. Сочетает достоинства обоих методов. Например, ручное тестирование может использоваться для исследования новых функций, а автоматизация — для регрессионных тестов и рутинных проверок. Такой подход обеспечивает максимальную гибкость и оптимизирует ресурсы команды.

Выбор между подходами зависит от целей проекта, бюджета и временных рамок.

- Различия в методах и инструментах

Основные различия между методами и инструментами:

Метод — это систематизированная совокупность шагов, действий, которые необходимо предпринять, чтобы решить определённую задачу или достичь определённой цели.

Инструмент — это средство воздействия на объект.

Таким образом, **метод может быть одним из инструментов, а инструмент служит для реализации метода**.

- Общие принципы тестирования

Некоторые общие принципы тестирования:

Тестирование показывает наличие ошибок, а не их отсутствие. Даже многократное тестирование не может гарантировать, что программное обеспечение на 100% не содержит ошибок.
Исчерпывающее тестирование невозможно. Невозможно протестировать все функциональные возможности со всеми допустимыми и недопустимыми комбинациями данных.
Раннее тестирование. Ошибка, выявленная на ранних этапах жизненного цикла разработки ПО, обойдётся гораздо дешевле.
Кластеризация дефектов. Небольшое количество модулей содержит в себе большинство обнаруженных ошибок.
Тестирование зависит от контекста. Подход к тестированию зависит от типа системы, её целей и условий эксплуатации.
Парадокс пестицида. Многократное повторение одних и тех же тестовых кейсов с одними и теми же тестовыми данными не приведёт к обнаружению новых ошибок. Поэтому необходимо проанализировать тестовые кейсы и обновить их или добавить другие, чтобы найти новые ошибки.

- Влияние архитектуры приложения на тестирование

Архитектура приложения влияет на тестирование следующим образом:

Усложняет параллельное или селективное выполнение тестов. Это негативно влияет на время выполнения сценариев тестирования и увеличивает стоимость их поддержки.
Усложняет методы верификации поведения. Результат тестового сценария может отличаться время от времени, что требует специальных подходов.
Затрудняет разделение на модули. Это мешает быстрее локализовывать ошибки и делать небольшие точечные исправления без переписывания большого количества кода.

Однако **продуманная архитектура может сделать тесты более наглядными и понятными для чтения**. Например, если тесты представляют собой последовательность вызовов понятных методов, то за последовательностью шагов будет просто проследить.

Таким образом, архитектура приложения должна быть такой, чтобы тестирование было эффективным и позволяло оптимизировать производительность и надёжность программного обеспечения.

4.2. Примеры успешного тестирования фронтенда и бэкенда

- Кейсы из практики

Примеры успешного тестирования фронтенда:

Контроль отображения интерфейса на разных устройствах и браузерах, удобства использования, отклика на действия пользователей, правильности выполнения клиентских скриптов и запросов к API. Для этого используются инструменты автоматизации фронтенда (например, Selenium, Cypress).
Тест-кейс «Добавить нового питомца в магазин». С его помощью проверяют, как API реагирует на вызов метода POST, и оценивают корректность ответа от сервера. Для этого используют сайт petstore.swagger.io.

Примеры успешного тестирования бэкенда:

Проверка правильности работы серверных скриптов, обработки данных, безопасности, производительности сервера, корректной работы API. Для этого используются инструменты для автоматизации бэкенд-тестирования (например, Postman, JUnit, Mockito).
Тест-кейс «Регистрация пользователя». С его помощью проверяют, как API обрабатывает регистрацию пользователя на сайте try.vikunja.io. Для этого отправляют POST-запрос try.vikunja.io с определёнными параметрами. Ожидаемый результат — успешная регистрация пользователя, при которой от сервера возвращается тело ответа в формате JSON.
Тестирование страницы логина. Если введённые юзернейм и пароль правильные, система должна впустить пользователя и перенаправить его на следующую страницу.

Глава 5: Интеграционное тестирование

Интеграционное тестирование — это этап тестирования программного обеспечения, на котором проверяется взаимодействие между различными модулями или компонентами системы. Цель интеграционного тестирования заключается в выявлении проблем, возникающих при взаимодействии между модулями, которые могут не проявляться при тестировании каждого модуля по отдельности.

Основные аспекты интеграционного тестирования:

1. Цели интеграционного тестирования:
   - Проверка корректности взаимодействия между модулями.
   - Выявление ошибок, связанных с интерфейсами и взаимодействием.
   - Проверка совместимости различных компонентов системы.

2. Подходы к интеграционному тестированию:
   - Тестирование "снизу вверх" (Bottom-Up Testing): Сначала тестируются низкоуровневые модули, а затем интегрируются более высокоуровневые модули.

- Тестирование "сверху вниз" (Top-Down Testing): Сначала тестируются высокоуровневые модули, а затем добавляются низкоуровневые модули.
- Тестирование "по большому взрыву" (Big Bang Testing): Все модули интегрируются одновременно, и тестирование проводится на всей системе.
- Инкрементное тестирование (Incremental Testing): Модули интегрируются и тестируются поэтапно.

3. Типы интеграционного тестирования:
- Тестирование интерфейсов: Проверка взаимодействия между модулями через их интерфейсы.
- Тестирование API: Проверка корректности работы программных интерфейсов.
- Тестирование баз данных: Проверка взаимодействия приложения с базой данных.
- Тестирование производительности: Оценка производительности системы при взаимодействии различных компонентов.

4. Инструменты для интеграционного тестирования:
- JUnit, TestNG: Для тестирования Java-приложений.
- Postman, SoapUI: Для тестирования API.
- Selenium: Для тестирования веб-приложений.
- Jenkins, Travis CI: Для автоматизации тестирования и CI/CD.

5. Примеры интеграционного тестирования:
- Проверка, что данные, отправленные из одного модуля, корректно обрабатываются и отображаются в другом модуле.
- Тестирование взаимодействия между клиентским приложением и сервером через API.
- Проверка, что изменения в базе данных корректно отражаются в пользовательском интерфейсе.

Итог;

Интеграционное тестирование является важным этапом в процессе разработки программного обеспечения, так как оно помогает выявить проблемы, которые могут возникнуть при взаимодействии различных

компонентов системы. Это позволяет обеспечить более высокое качество и надежность конечного продукта.

5.1. Понятие интеграционного тестирования

Интеграционное тестирование — это этап в проверке качества программного обеспечения, который следует за модульным тестированием. На этом этапе отдельные модули или компоненты приложения объединяют и проверяют на совместимость друг с другом. **Основная цель интеграционного тестирования** — убедиться, что различные модули программы правильно взаимодействуют друг с другом. Это помогает выявить ошибки на стыках между компонентами.

Некоторые задачи интеграционного тестирования:

проверка взаимодействия модулей;
обеспечение совместимости;
раннее обнаружение проблем;
повышение общей надёжности системы;
повышение качества ПО за счёт выявления и устранения ошибок до того, как это станет более сложным и дорогостоящим процессом.

Выделяют несколько типов интеграционного тестирования:

Интеграционное тестирование «большого взрыва». Этот вид тестов предполагает одновременную интеграцию всех программных модулей и их тестирование как целостной системы.
Интеграционное тестирование сверху вниз. Компоненты интегрируются и тестируются от самого высокого уровня к самому низкому.

Выбор подхода к интеграционному тестированию зависит от особенностей приложения и текущей задачи.

5.2. Подходы к интеграционному тестированию фронтенда и бэкенда

Существует несколько подходов к интеграционному тестированию:

«Большой взрыв» (Big Bang тестирование). Все модули объединяются одновременно и затем проверяются.
Нисходящее. Тестирование начинается с верхнего уровня архитектуры системы и постепенно спускается вниз.
Восходящее. Процесс начинается с нижних уровней и движется вверх.
Смешанное (песочные часы или сэндвич). Комбинация двух предыдущих методов.

На практике подход к интеграционному тестированию выбирают в зависимости от особенностей приложения и текущей задачи.

Для проверки интеграции фронтенда с бэкендом можно использовать тестовые фреймворки Playwright или Cypress, которые позволяют автоматизировать пользовательские сценарии. Эти инструменты помогают отследить сетевые запросы, оценить корректность данных, передаваемых между frontend и backend, а также проверить, как API отвечает на действия пользователя.

- Тестирование взаимодействия между компонентами

Интеграционное тестирование — это процесс проверки взаимодействия между различными компонентами системы. Это важный этап в процессе разработки программного обеспечения, так как он позволяет выявить проблемы и ошибки во взаимодействии между различными модулями, которые могут быть незаметны во время отдельного тестирования каждого компонента.

Основные цели интеграционного тестирования:

проверка корректности взаимодействия между компонентами;
выявление проблем в архитектуре системы;
повышение надёжности системы.

Некоторые инструменты и подходы к интеграционному тестированию:

Использование автоматических инструментов тестирования. Например, Selenium, JUnit или TestNG.
Тестирование на уровне API. Проверяется корректность работы между различными слоями системы.
Тестирование на уровне пользовательского интерфейса. Для проверки корректности работы системы на уровне пользовательского интерфейса можно использовать инструменты автоматического тестирования UI, такие как Selenium или Appium.

- Использование моков и стабов

Моки и стабы — это два основополагающих вида тестовых объектов в mock-тестировании.

Моки — это объекты, которые имитируют поведение реального компонента системы во время тестирования, однако действия этого объекта полностью контролируются тестом. Моки позволяют задать результаты вызова того или иного метода, а также убедиться в том, что тестируемый код взаимодействует с зависимостями так, как это ожидается.

Стабы — это тоже тестовые объекты, но они возвращают заранее определённые значения на вызовы методов. То есть стабы просто выполняют роль имитируемого объекта: формируют запросы к сервису и возвращают нужные (уже определённые) значения, при этом не отслеживая взаимодействие с зависимостями, как это делают моки. Стабы могут использоваться, когда поведение зависимости не оказывает сильного влияния на тестируемый компонен

5.3

. Современные подходы к интеграционному тестированию

Некоторые современные подходы к интеграционному тестированию:

Метод «большого взрыва» (Big Bang). Все модули собираются воедино и тестируются как целостная система. Метод имеет ограничения: сложно локализовать источник проблем, а при возникновении ошибок приходится анализировать всю систему целиком.
Метод «снизу вверх» (Bottom-Up). Тестирование начинается с компонентов нижнего уровня, постепенно продвигаясь к верхним слоям системы. Модули тестируются группами, используя специальные драйверы для симуляции работы верхних уровней. Такой метод особенно эффективен, когда нижние уровни системы достаточно стабильны и хорошо определены.
Метод «сверху вниз» (Top-Down). Тестирование начинается с верхних уровней системы, постепенно спускаясь к нижним компонентам. Для имитации работы нижних модулей используются заглушки. Этот подход позволяет рано получить прототип системы и проверить основную функциональность.
Смешанный метод (Sandwich). Объединяет преимущества подходов «сверху вниз» и «снизу вверх», позволяя проводить тестирование одновременно с обоих концов системы. Это обеспечивает более полное покрытие и эффективное использование ресурсов.

Выбор подхода зависит от особенностей приложения и текущей задачи.

- Контейнеризация (Docker)

Docker — это **платформа контейнеризации с открытым исходным кодом**, с помощью которой можно автоматизировать создание приложений, их доставку и управление.
Платформа позволяет быстрее тестировать и выкладывать приложения, запускать на одной машине требуемое количество контейнеров.

Основные компоненты контейнеризации в Docker:

Dockerfile. Это файл для предварительной работы, набор инструкций, который нужен для записи образа. В нём описывается, что должно находиться в образе, какие команды, зависимости и процессы он будет содержать.
Docker Image. Это образ — неизменяемый файл, из которого разворачивается контейнер. Для этого нужно запустить образ в клиенте с помощью специальной команды: docker run <образ>.
Docker Registry. Это реестр, или репозиторий — открытая или закрытая база образов. К ней можно подключиться через клиент Docker и загрузить нужный с помощью команды: docker pull <образ>.
Docker Container. Это уже готовый и развёрнутый контейнер, который находится на каком-либо устройстве.

Docker контейнер — это стандартизированный, изолированный и портативный пакет программного обеспечения, который включает в себя всё необходимое для запуска приложения, включая код, среду выполнения, системные инструменты, библиотеки и настройки.

- CI/CD (непрерывная интеграция и доставка)

CI/CD (Continuous Integration, Continuous Delivery) — это технология автоматизации тестирования и доставки новых модулей разрабатываемого проекта заинтересованным сторонам (разработчикам, аналитикам, инженерам качества, конечным пользователям и др.).

CI состоит из двух частей:

CI (Continuous Integration) — непрерывная интеграция. Разработчики могут предлагать изменения в основной код проекта фактически неограниченное количество раз в день. При этом любые изменения проходят через автоматические тесты.
CD (Continuous Deployment) — непрерывная доставка. После сборки проекта с внесёнными изменениями, артефакты сборки автоматически развёртываются на серверы. Главная задача CD — оперативно и без перебоев доставлять обновления пользователям без необходимости включения разработчика или прерывания работоспособности сервиса.

Цель CI/CD — уделение достаточного внимания бизнес-требованиям, безопасности и качеству кода конечного продукта. Она позволяет сократить сроки разработки, отобрать перспективные варианты кода и повысить качество тестирования.

Заключение

В заключении работы подводятся итоги сравнительного анализа методов тестирования фронтенда и бэкенда, а также предлагаются рекомендации по выбору методов и инструментов для тестирования в зависимости от специфики проекта. Также рассматриваются перспективы развития тестирования в контексте новых технологий и подходов.

Список литературы

1. Мартин, Р. (2011). "Чистый код: создание, анализ и рефакторинг".
2. Фаулер, М. (2009). "Паттерны корпоративных приложений".
3. Беннет, К. (2015). "Тестирование программного обеспечения: теория и практика".
4. Документация по инструментам тестирования (Jest, Cypress, Postman и др.).
