adesso

Выбор инструмента автоматизации тестирова для проекта “Abcd-ef“ компании ABC GmbH

Anastasiia Kem, Test Engineer

Более 7 лет в тестировании в разных доменных областях(web, embedded, fintech, banking) в таких компаниях как GlobalLogic, Raiffeisen Bank, Hermes Europe

Почти 15 лет в IT вцелом.

adesso

Contents

1. **Определяем детали проекта:**
Определение домена, тех. характеристик, стади проекта.
Наявность требований (requirements) и др. документации, мануал- /авто- тестирования.
Определяем проблемные участки, чего хотим и в какие сроки

2. **Коротко основы:**
Пирамида тестирования. Варианты покрытия API уровня, варианты платных решений UI, варианты opensource решений UI.

3. **Выбор языка программирования**
Язык программирования Backend VS Язык программирования Frontend

4. **Обзор средств автотестов для UI:**
Selenium, Playwright, Cypress. Определение лидеров AT.

5. **Важность тестируемости и измеряемости**
Улучшение тестируемости приложения и интеграции автотестов в CI/CD пайплайн. Построение Quality Gate. Определение отчетности и метрик.

6. **Итоговые рекомендации.**

adesso

1. Определяем детали проекта

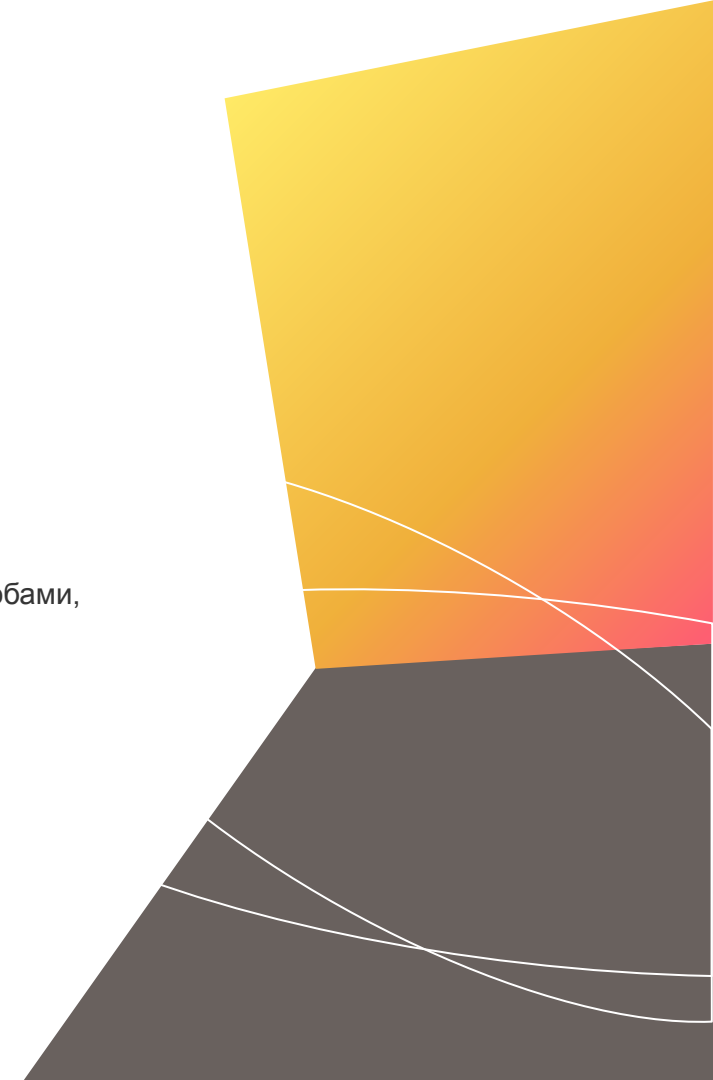
1.1 Определение домена, тех. характеристик, стади проекта

1.2 Наявность требований (requirements) и другой проектной документации

1.3 Наявность мануал- /авто- тестирования

1.4 Определяем проблемные участки, если есть (обращение пользователей с жалобами, улучшениями), внутренние ограничения.

1.5 Определяем чего хотим, в какие сроки, за какой бюджет



adesso

2. Коротко основы

2.1 Пирамида тестирования:

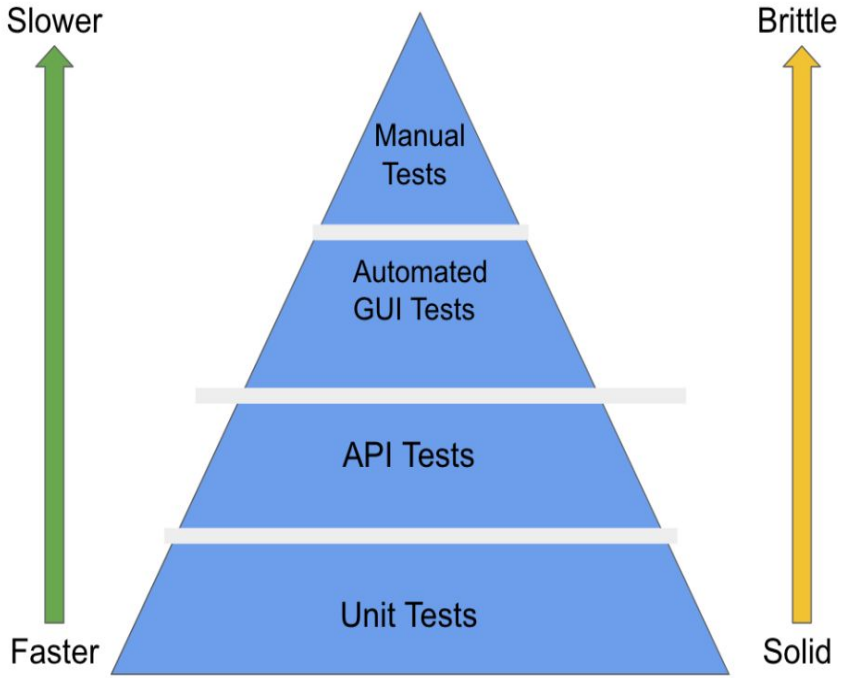
Unit Tests - покрытие кода тестами на низком уровне, пишут программисты приложения;
API Tests / Integration Tests - проверка обмена данными между двумя модулями программы, приложениями, сервисами.
GUI Tests - проверка внешних элементов приложения
Manual Tests - проверка “в ручную”, присутствует во всех кроме нижних уровнях. В идеале - мануальных тестов должно быть меньше всего. **Наша цель - добиться этого.**

2.2 Automation Tests:

Варианты покрытия API уровня:
Полуавтоматизированные: Postman, SoapUI, JMeter
Фреймфорки для автотестирования: REST-Assured /Java, RestAssured.Net/C#, Retrofit /Java

Варианты платных решений для E2E / UI тестов:
Katalon Studio, TestComplete, Ranorex

Варианты opensource решений E2E / UI тестов:
Selenium WebDriver /Java, C#, Python, Js Playwright /Java, C#, Python, Js, Cypress /Js



adesso

3. Выбор языка программирования

3.1 Язык программирования такой как на **Back-End** части проекта.

+ Можно удобно интегрировать API тесты
+ Можно переиспользовать часть кода основных классов проекта
+ Стабильный и популярный язык (Java, C#, Python)
+ Большое сообщество программистов и тестировщиков. Легко найти рабочие кадры.
- UI автотесты которые проводятся в браузере могут иметь некоторые задержки, так как для браузера нативный язык JavaScript / TypeScript
- Для Java, C# более высокий порог входа. (**Но для нас это не проблема:)**) =>

3.2 Язык программирования такой как на **Front-End** части проекта

+ JavaScript / TypeScript это нативный язык браузера, поэтому нужно меньше прослоек, чтоб взаимодействовать с элементами страницы
+ Playwright фреймворк разработанный Microsoft и был изначально написанный на TypeScript, что обеспечивает самые свежие фичи.
  (Но Playwright также портирован на все остальные популярные языки)
+ JavaScript / TypeScript также очень популярны у многих разработчиков.
- Нет типизации в языке JavaScript, что делает его более хаотичным
- Скриптовый язык JavaScript медленнее чем Java, C#.

adesso



4. Обзор средств автотестов для UI:

Selenium WebDriver – это Open-Source инструмент, для автоматизации UI тестирования в браузере. Обеспечивает кросс-браузерное тестирование (Chrome, Firefox, Safari) и мультиязыковую поддержку.

Принцип: Эмуляция реальных действий пользователя в браузере (клики, ввод текста, навигация, ожидание загрузки элементов) для проверки функциональности и стабильности веб-приложения.
WebDriver работает по модели Клиент-Сервер (Client-Server Model), используя стандартизированный протокол:

Как это происходит:
1. Наши классы и библиотеки, то есть то что мы будем разрабатывать, на выбранном языке (например, Java) использует встроенные классы и методы Selenium, где уже прописана основная логика взаимодействия с браузером. Например мы описываем команду, например, "**driver.element.click()**" - что равно клик мышки по элементу на странице, так как бы это делал пользователь.

2. JSON Wire Protocol / W3C WebDriver Standard
Наша команда **driver.element.click()** и ряд других вспомогательных команд на Java (мы в данном случае выступаем "Клиентом") сериализуется в виде HTTP-запроса с полезной информацией в формате JSON и отправляются далее на обработку драйверам-посредникам

3. Браузерные драйверы (Browser Drivers) - это уже готовые исполняемые файлы (например, chromedriver.exe, geckodriver.exe), предоставленные разработчиками браузеров Google, Mozilla, получают HTTP/JSON запросы от твоего кода, десериализуют их и переводят в команды, понятные для **API браузера.**

4. Браузер выполняет нативную переданную команду и сообщает Драйверу о результате. А Драйвер в свою очередь, формирует ответ в JSON-формате и отправляет его обратно в наш написанный фреймворк на Java.

Вывод:
Selenium не работает с браузерным кодом JavaScript напрямую, а управляет браузером через его собственный, предоставленный вендором, нативный интерфейс.
- Это немного влияет на скорость выполнения UI тестов. Каждая команда — это новый HTTP-запрос, что приводит к дополнительным накладным расходам и замедлению.
- Выглядит громоздким и многословным. Есть конкуренты более “легкие”.
+ Selenium WebDriver это часть W3C WebDriver стандарта, под который производители браузера обязаны разрабатывать и поддерживать совместимые драйверы.
+ Большое комьюнити, хорошая документация и много дополнительных плагинов и решений.
+ Пример такого решения **Selenide** - это обёртка вокруг Selenium WebDriver, позволяющая быстрее и просто его использовать при написании тестов.
+ Большое количество разработчиков как бюджетного уровня, так и дорогих.

Playwright — это более молодой и современный фреймворк для сквозного (End-to-End) тестирования, разработанный **Microsoft** и позиционируется как современная, быстрая и надежная альтернатива Selenium.
Ключевая особенностьЕдиный API для Chrome/Edge, Firefox и Webkit (Safari), включая работу с мобильными эмуляциями.
Основная цель и отличие от конкурентов: Обеспечить высокую **скорость** выполнения тестов, стабильность и поддержку современных веб-технологий (Single Page Applications, Shadow DOM, WebSockets).

Главное отличие Playwright от Selenium — это **способ взаимодействия с браузером.** Вот некоторые ключевые особенности:

1. Протокол Связи: WebSocket
Selenium: Основан на HTTP-запросах - JSON Wire Protocol, что медленнее.
Playwright: Использует WebSocket-соединение: то есть устанавливает **одно постоянное** двунаправленное **соединение** между нашим тестовым скриптом и браузером.

2. Подход "Out-of-Process"
Это обеспечивает минимальную задержку и более быструю передачу команд и ответов, что делает тесты значительно быстрее.
Playwright запускает сервер (Node.js), который, в свою очередь, управляет браузером напрямую.

+ Этот подход позволяет Playwright взаимодействовать с браузером на более низком уровне, что быстрее и менее ресурсоемко.
+ Поддерживает много современных фич анализ трафика, запись результатов теста.

adesso

- Иногда сложнее найти разработчиков, чем на Selenium.
- Скорее всего что большие и старые проекты уже имеют кодовую базу тестов на Selenium

Cypress - еще один сильный игрок фреймворк для End-to-End тестирования, имеет иной архитектурный подход по сравнению как с Selenium, так и с Playwright.
Ключевой принцип- тесты выполняются непосредственно в браузере, рядом с самим тестируемым приложением.
Cypress Test Runner (CLI-интерфейс и графический UI) запускает браузер. Тестовый код загружается и выполняется в том же цикле выполнения (Run Loop), что и твой код приложения.

+ Нативный. Полный доступ ко всем объектам браузера (DOM, window, document, localStorage), что невозможно для Selenium.

- Основан на JavaScript/TypeScript и работает только с экосистемой Node.js.

5. Важность тестируемости и измеряемости

Очень важно заниматься интеграцией автотестов в CI/CD пайплайн. Автотесты написанные “в стол” не приносят пользы.

Ключевая задача - построение Quality Gate, когда тесты интегрированы тесно в CI/CD и каждый релиз, обновление от разработчиков сопровождается запуском автотестов.

Так вот, **Quality Gate** (ворота качества) — это набор автоматических проверок, которые определяют, соответствует ли код (проект с обновлениями) необходимым критериям качества для перехода на следующий этап разработки.

Это система контрольных точек в конвейере CI/CD, которая останавливает продвижение кода, если он не проходит пороговые значения, тем самым предотвращая попадание дефектного кода в продакшн и решая проблемы на ранних стадиях

Так же важны **метрики** и **отчетность,** которые помогают оценивать эффективность тестирования.

Основные метрики включают процент автоматизации, тестовое покрытие кода, время исправления критических дефектов (MTTR) и плотность дефектов.

Отчеты, в свою очередь, так же предоставляют качественный обзор, который помогает команде понять статус проекта

6. Итоговые рекомендации по выбору инструмента для автотестов

Язык программирования: Java

Фреймворк для API и интеграционного тестирования: REST-Assured

Фреймворк для UI тестирования: Selenium Webdriver (с возможностью использования Selenide)


Спасибо