ПЛОДУХИН Д. М.

РЕАЛИЗАЦИЯ МОДЕЛИ АВТОМАТИЗИРОВАННОГО ТЕСТИРОВАНИЯ

Аннотация. В статье приведено описание концепции модели автоматизированного тестирования и ее реализации. Описаны технологии, необходимые в разработке, и продемонстрированы примеры работы системы.

Ключевые слова: автоматизированное тестирование, веб-приложение, Kotlin, Selenium, Selenide, REST Assured, JDBC, Allure.

PLODUKHIN D. M.

IMPLEMENTATION OF AUTOMATED TESTING MODEL

Abstract. The article presents the concept model of automated testing and its implementation. The technologies necessary for development are described and examples of the system are demonstrated.

Keywords: automated testing, web application, Kotlin, Selenium, Selenide, REST Assured, JDBC, Allure.

Основная концепция модели автоматизированного тестирования заключается в том, что за многие годы практики разработки в тестировании был накоплен определенный опыт и набор требований к тестовым фреймворкам, которые используются при написании автотестов для того или иного веб-приложения. Рассматривать каждое приложение по очевидным причинам не представляется возможным, поэтому в данной модели собраны общие тенденции и практики, которые универсальны и переносимы на любой процесс автоматизированного тестирования [1; 2].

Тестовая модель представляет собой систему для тестирования «идеального», моделируемого веб-приложения, разработанного по современным требованиям и стандартам. Следует рассмотреть веб-приложение по трем основным его составляющим:

1. Клиентская часть веб-приложения (Front-end) – это графический пользовательский интерфейс. То, что видит пользователи на странице браузера. Пользователь взаимодействует с веб-приложением именно через эту часть, нажимая на различные элементы управления. К этой части относится вся логика, отвечающая за визуализацию контента.

2. Серверная часть веб-приложения (Back-end – иногда этим термином называют всю совокупность программно-аппаратной части сервиса) – это скрипт или программное обеспечение, которые располагаются на сервере и обрабатывают запросы пользователя (корректнее, запросы браузера).

3. База данных (БД) – программное обеспечение на сервере, которое хранит данные и выдает их в нужный момент. Большинство современных веб-приложений работают реляционными СУБД, но постепенно занимают свое место на рынке и нереляционные БД (NoSQL).

В соответствии с этими требованиями был сформирован технологический стек, необходимый для разработки.

1. **Kotlin** (Ко́тлин) – статически типизированный язык программирования, запускаемый поверх Java Virtual Machine и разрабатываемый компанией JetBrains. Это гибкий и лаконичный язык, заточенный под функциональное программирование и полностью совместимый с Java.

2. **Gradle** – система автоматической сборки, построенная на принципах Apache Ant и Apache Maven, но предоставляющая DSL на языках Groovy и Kotlin вместо традиционной XML-образной формы представления конфигурации проекта. Gradle идеально подходит для инкрементальных сборок и очень удобен.

3. **Selenium WebDriver** – программное обеспечение для автоматизации манипуляций с веб-браузером. Чаще всего используется для тестирования Web-приложений, но не ограничивается этим. Практически бескомпромиссное решение в работе с браузерами.

4. **Selenide** – библиотека для написания лаконичных и стабильных UI тестов с открытым исходным кодом. Selenide – это обёртка вокруг Selenium WebDriver, позволяющая быстро и просто его использовать при написании тестов. Для примера, не нужно акцентировать внимание на работе с ожиданиями элементов в процессе автоматизации тестирования динамических веб-приложений, а также на реализации высокоуровневых действий над элементами.

5. **TestNG/JUnit** – являются наиболее популярными средами модульного тестирования на текущий момент для Java/Kotlin. Выбор между ними не стоит так остро, поэтому имеет смысл поддержки разработки под обе среды.

6. **JDBC** (англ. Java DataBase Connectivity соединение с базами данных на Java) – платформенно независимый стандарт взаимодействия Java-приложений с различными СУБД, сделанный в виде пакета java.sql, входящего в состав Java SE. Данный стандарт – классическое решение для разработки на Java или Kotlin. Всё, что потребуется во время разработки, это написание удобной и понятной обертки для методов библиотек JDBC.

7. **REST Assured** – DSL для тестирования REST-сервисов, который встраивается в тесты на Java/Kotlin. Это решение существует уже более десяти лет и остается популярным из-за своей простоты и удобной функциональности.

8. **Allure** – инструмент, позволяющий создавать удобные и понятные отчеты, чтобы внести прозрачность в процесс создания и выполнения функциональных тестов. Allure помогает разработчикам и бизнесу более активно взаимодействовать, благодаря отображению результата тестирования в понятной форме для нетехнических специалистов.

Продукт, основанный на этой модели, уже применяется на некоторых коммерческих проектах, поэтому возможна лишь демонстрация существующей функциональности, с ограничением обзора исходного кода.

При тестировании веб-интерфейсов, в данном проекте поддерживается использование паттерна Page Object Model. Page Object моделирует страницы тестируемого приложения в виде объектов в коде. Основное преимущество Page Object заключается в том, что в случае изменения разработчиками приложения интерфейса, можно сделать исправление только в одном месте, а не править каждый тест, в котором этот интерфейс тестируется.

Реализован POM в данной работе в виде заведомо статических объектов, что допустимо благодаря “ленивой” инициализации элементов Selenide. Объявление Page Object и работа с ним выглядит следующим образом:

```
    val pageObject by PageObject()
class PageObject : Element by css("css") {
  val element1 by className("class")
  val element2 by tag("tag")
  val subPageObject by SubPageObject()
}
class SubPageObject : Element by linkText("text") {
  val element3 by xpath("//xpath")
}
```

Ключевым моментом здесь является сохранение контекста найденного элемента. Любой последующий вложенный элемент будет найден от родительского.

Структура условий перешла из Selenide практически без изменений. Но благодаря ей было реализовано одно из самых частых применений данного объекта – поиск по условию. Скорее всего самый распространенный из них это поиск по тексту (использование условий в примере ниже равнозначно):

```
    class SimpleTest {
  @Test
  fun simpleTest() {
    pageObject.element1[text("Some text 1")]
    pageObject.element1["Some text 1"]
```

```
    }
}
```

Определяя экземпляр объекта типа Element, будет находиться всегда первый элемент, удовлетворяющий заданному локатору и/или условию. Но иногда приходится работать и с коллекциями элементов.

В текущем случае был выбран вариант, объединяющий выражение all из чистого Kotlin и ElementsCollection из Selenide – Elements, поэтому с ним можно взаимодействовать как с обычным списком, так из коллекций из Selenide:

```
    class SimpleTest {
  @Test
  fun simpleTest() {
    // ElementsCollection - проверка размера коллекции с помощью CollectionCondition
    pageObject.element1.all.shouldBe(CollectionCondition.size(5))
    // Аналогично, средствами Kotlin
    assertEqual(pageObject.element1.all.size, 5)
  }
}
```

Для обзора всей функциональности работы с веб-элементами нет необходимости выводить весь листинг доступных методов для работы с веб-элементами (многие из них являются просто оберткой для Selenide). Однако стоит выделить несколько изменений различной степени значимости:

1. **Определение методов через свойства.** Использование геттеров и сеттеров для лаконичности вызова.
2. **Часто используемые действия вынесены в отдельные методы.** Например: нажатие кнопки через Javascript, очистка поля клавиатурой.
3. **Встроенные проверки.** Например, при вызове любого метода, элемент проверяется на видимость без лишнего указания на это.

Касательно веб-навигации, разработчику придется работать с данными объектами, только если потребуется произвести какие-то особые проверки и действия с браузерами и вкладками. В противном случае, есть объект window, который является по сути текущей активной вкладкой в браузер (и создает ее, если она еще не открыта). Общий вывод по тестированию пользовательских интерфейсов с данными спецификациями легче оценить на примере лаконичного теста:

```
    class SimpleTest {
  @Test
```

```
    fun simpleTest() {
        "Открытие окна" {
            window.url = "http://mysite.com"
        }
        "Авторизация" {
            authorizationForm {
                login.value = "login"
                password.value = "password"
                submit.click
            }
        }
        "Проверка отображения формы профиля и корректного текста" {
            profileForm {
                shouldBe(text("Your profile"))
                list["List #1"].shouldBe(exactText("List #1: item text"))
            }
        }
    }
}
```

Логика тестирования API состоит из обертки для хостов и запросов:

```
        val localApplicationClient by lazy {
    ApiClient(
        url = "localhost:8080",
        desc = "Локальное приложение"
    )
}
internal fun postRequest(param1: String, param2: Int) = Request(
    desc = "[POST] POST-запрос с параметром",
    spec = {
        addHeaders(
            mutableMapOf(
                "header1" to "hd",
                "header2" to "dh"
            )
        )
```

```
        setBody(
            """{
                "param1" : "$param1"
                "param2" : $param2
            }""".trimIndent()
        )
    },
    send = { post("/post") },
    verify = { HTTP_CREATED }
)
```

Сам процесс тестирования API выглядит следующим образом:

```
class SimpleTest {
    @Test
    fun simpleTest() {
        val result = localApplicationClient
            .send(postRequest("p1", 2))
            .jsonPath()
        assertEqual(result.getString("body.res1"), "result")
    }
}
```

Тестирование и работа с БД проводится по аналогичному принципу:

```
        val dbMSSQL by lazy {
    DbDriver.MsSql(
        "localhost:1234;databaseName=myBase;",
        "username",
        "password"
    )
}
fun dbPostgres(dataBaseName: String) = DbDriver.Postgres(
    "localhost:1234;databaseName=$dataBaseName;",
    "username",
    "password"
)
class SimpleTest {
    @Test
```

```
    fun simpleTest() {
        val result = dbMSSQL
            .execute(
                DbQuery(
                    "SELECT * FROM table",
                    "Select к таблице"
                )
            )
        assertEqual(result.first().get("name"), "myname")
    }
}
```

Естественно, вся указанная выше функциональность не является достаточной для построения полноценного жизненного цикла тестирования. Существует некоторый спектр улучшений, которые нельзя отнести к ранее описанным категориям во многом в рамках интеграции:

1. **Слушатель (listener) Allure и упрощение Allure Step.** Для автоматического формирования отчетов и удобного написания своих шагов без привязки к методу.
2. **Интеграция с системами отслеживания ошибок, системами управления проектами и тест-менеджерами.** Результаты тестирования автоматически будут отправляться в указанные системы.
3. **Запуск тестов.** Локальный и удаленный запуск будут конфигурироваться специальными параметрами запуска.
4. Работа с прокси-фильтрами и почтовыми серверами.

СПИСОК ЛИТЕРАТУРЫ

1. Русское руководство по языку Kotlin [Электронный ресурс]. – Режим доступа: https://kotlinlang.ru (дата обращения 18.05.2020).
2. Официальный сайт Selenide [Электронный ресурс]. – Режим доступа: https://ru.selenide.org (дата обращения 12.05.2020).