GeekBrains

Автоматизация тестирования Web UI на Java

Работа с отчётами и логами

[Java 11]

© geekbrains.ru

На этом уроке

1. Узнаем, что такое Allure.
2. Разберём протоколирование и наблюдение за действиями драйвера.
3. Рассмотрим получение доступа к логам браузера.

Оглавление

Зачем и для чего

На курсе по основам тестирования программного обеспечения мы узнали, что результатом работы (сессии тестирования) инженера считается отчёт.

Отчёт — это полезный элемент, так как содержит в себе конкретику:

- какой инструментарий протестировался;
- в каком объёме;
- сколько тест-кейсов пройдено успешно, а сколько — завалено.

Отчёт составлен в такой форме, что любое заинтересованное лицо, далёкое от разработки ПО (например, заказчик), могло его понять и проанализировать, сделать нужные выводы, например, об изменении качества в динамике.

До сегодняшнего дня мы запускали автотесты в IDE и консоли и получали результат в виде логов. Это прекрасно, т. к логи — замечательный инструмент. Читая логи, часто можно понять, что пошло не так. Но отчёты о тестировании читаем не только мы. Заинтересованные лица хотят получать отчёты вне зависимости от того, как именно протестирован продукт — вручную или автоматически. Важен результат, а не инструмент.

Allure

Allure — фреймворк с открытым исходным кодом. Этот фреймворк разработан компанией «Яндекс» специально, чтобы избавить инженеров по автоматизации тестирования от «боли» составления красивых отчётов о прохождении автотестов. Ссылка на статью с подробным объяснением механики работы фреймворка находится в секции «Дополнительные материалы».

Подключение к проекту

Чтобы не заниматься перекопированием материала, ниже приводится ссылка на код для подключения к проекту Allure (на примере JUnit 5) на странице официальной документации.

Использование аннотаций

Достоинство фреймворка заключается в том, что для его использования не надо изменять действующий код. Достаточно аннотировать нужные классы и методы специальными маркерами: наиболее употребимые перечислены в таблице ниже:

`DisplayName`	Добавление человеко-читаемого названия тестовому методу
`Description`	Добавление описания к тестовому методу
`Attachment`	Добавление вложения в отчёт
`Link / Issue / TmsLink`	Добавление ссылки на систему менеджмента тест-кейсами / баг-трекер
`Severity`	Добавление поля Severity для тестового метода в отчёт
`Epic / Feature / Story`	Разделение тестовых методов или классов на функционал-специфичные группы
`Step`	Название шага тест-кейса. Обычно применяется к методам Page Object классов

Получение отчётов

Чтобы получить отчёт, надо:

- открыть терминал;
- открыть папку проекта;
- ввести команду:

```
allure serve allure-results
```

где allure-results — папка с результатами тестирования.

Затем на локалхосте запустится сервер Allure, и откроется отчёт в веб-форме.

Протоколирование действий

Мы познакомились с пакетом org.openqa.selenium.support, но познакомились не со всеми его классами. В этом уроке обратим внимание на следующие два представителя, позволяющие отслеживать и протоколировать действия драйвера во время тестов.

Первый — интерфейс WebDriverEventListener, предоставляющий событийные методы. Методы имеют понятную семантику:

```
void beforeAlertAccept(WebDriver driver);
void afterAlertAccept(WebDriver driver);
void afterAlertDismiss(WebDriver driver);
void beforeAlertDismiss(WebDriver driver);
void beforeNavigateTo(String url, WebDriver driver);
void afterNavigateTo(String url, WebDriver driver);
void beforeNavigateBack(WebDriver driver);
void afterNavigateBack(WebDriver driver);
void beforeNavigateForward(WebDriver driver);
void afterNavigateForward(WebDriver driver);
void beforeNavigateRefresh(WebDriver driver);
void afterNavigateRefresh(WebDriver driver);
void beforeFindBy(By by, WebElement element, WebDriver driver);
void afterFindBy(By by, WebElement element, WebDriver driver);
void beforeClickOn(WebElement element, WebDriver driver);
void afterClickOn(WebElement element, WebDriver driver);
void beforeChangeValueOf(WebElement element, WebDriver driver, CharSequence[]
keysToSend);
void afterChangeValueOf(WebElement element, WebDriver driver, CharSequence[]
keysToSend);
void beforeScript(String script, WebDriver driver);
void afterScript(String script, WebDriver driver);
void beforeSwitchToWindow(String windowName, WebDriver driver);
void afterSwitchToWindow(String windowName, WebDriver driver);
void onException(Throwable throwable, WebDriver driver);
<X> void beforeGetScreenshotAs(OutputType<X> target);
<X> void afterGetScreenshotAs(OutputType<X> target, X screenshot);
void beforeGetText(WebElement element, WebDriver driver);
void afterGetText(WebElement element, WebDriver driver, String text);
```

Второй — EventFiringWebDriver — javadoc класса говорит, что это обёртка над экземпляром WebDriver, позволяющая регистрировать WebDriverEventListener для логирования.

Выглядит «оборачивание» следующим образом:

```
@BeforeEach
public void setUp() {
   EventFiringWebDriver driver = new EventFiringWebDriver(new ChromeDriver());
   driver.register(new CustomEventListener());
}
```

В качестве экземпляра драйвера воспользуемся EventFiringWebDriver, конструктор которого принимает экземпляр «настоящего» драйвера.

Далее для протоколирования действий драйвера регистрируем собственный «слушатель событий», реализующий интерфейс WebDriverEventListener. В чём именно будет заключаться протоколирование, решает тестировщик в зависимости от своих пожеланий:

- в выводе сообщения в консоль;
- в сохранении сообщений в специальный log-файл.

Сохранение скриншота при получении исключения

Стоит обратить внимание на возможность получения и сохранения скриншотов веб-страниц, которые создаются в процессе выполнения теста. Сохранять скриншот для последующего анализа можно постоянно, но наиболее логично будет их сохранение при получении исключений драйвера. Например, когда драйвер не находит элемент по указанному ID.

Интерфейс WebDriverEventListener имеет подходящий метод — onException, в него мы и поместим подходящую нам логику.

Но для начала в пакете утильных (вспомогательных) классов создадим класс ScreenshotMaker со статическим методом makeScreenshot:

```
public class ScreenshotMaker {

   public static void makeScreenshot(WebDriver driver, String filename) {
       File temp = ((TakesScreenshot) driver).getScreenshotAs(OutputType.FILE);
       File destination = new File("./target/" + filename);
       try {
           FileUtils.copyFile(temp, destination);
       } catch (IOException exception) {
           exception.printStackTrace();
       }
   }
}
```

Суть метода проста. Используя интерфейс TakesScreenshot, просим драйвер сделать скриншот и записать результат во временный файл. Подобное приведение типов уже встречалось, когда мы исполняли JavaScript-код.

Далее создаём файл с определённым именем в конкретной папке и копируем содержимое временного файла (со скриншотом) в файл на диске.

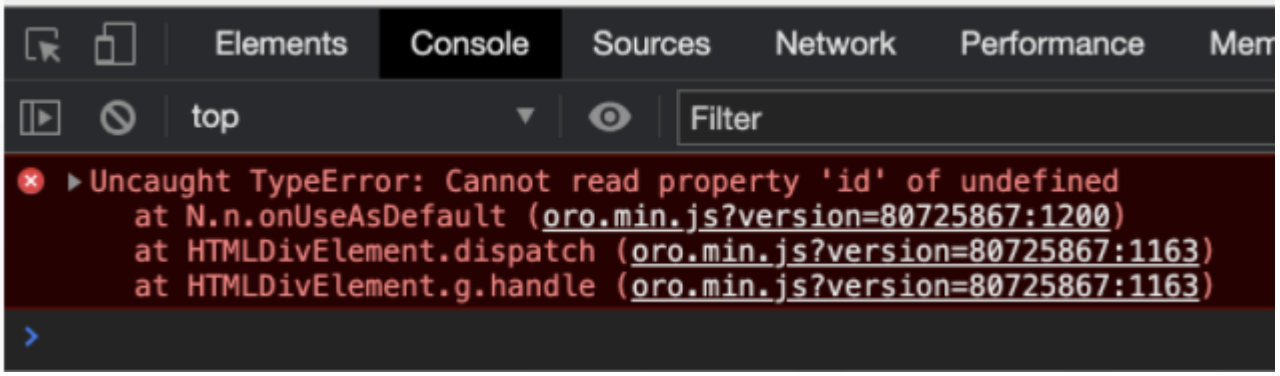
Используем в методе «слушателя»:

```
@Override
public void onException(Throwable throwable, WebDriver driver) {
   String fileName = "failure-" + System.currentTimeMillis() + ".png";
   ScreenshotMaker.makeScreenshot(driver, fileName);
```

```
    logger.error("onException: Screenshot saved in target/" + fileName);
}
```

Логи браузера

В процессе ручного тестирования внимательные инженеры обращают внимание на ошибки в консоли браузера, так как появление этих ошибок свидетельствует о некорректно работающей логике.



В рамках автоматизированного тестирования, разумеется, нет возможности открывать консоль и смотреть на её состояние. Тесты не запускаются на локальных машинах, но возможность получения логов предоставляет WebDriver.

```
@AfterEach
public void tearDown() {
    // Вывод всех ошибок браузера после каждого теста
    LogEntries browserLogs = driver.manage().logs().get(LogType.BROWSER);
    List<LogEntry> allLogRows = browserLogs.getAll();

    if (allLogRows.size() > 0 ) {
    // Обработка ситуации
    }

    // --------------------------------
    driver.quit();
}
```

Рассмотрим код построчно:

1. Драйвер возвращает браузерные логи в виде объекта LogEntries — имплементирующий интерфейс Iterable .

2. Получение объекта списка логов.

3. Если размер списка отличается от нуля, предпринимаем какие-нибудь действия, например, создаём файл с браузерными ошибками.

Практическое задание

1. Добавьте аллюр-репортинг к нашим тестам: CRM и своему проекту.
2. Добавьте браузерные логи, если это потребуется.

Дополнительные материалы

1. Статья «Allure 2: тест-репорты нового поколения».

Используемая литература

1. Unmesh Gundecha. Selenium Testing Tools Cookbook.
2. Статья Allure Framework.