Мастер-класс: Построение фреймворка автоматизации с нуля на примере Java для UI/API и запуске тестов в CI

Автор: [Геннадий Чурсов](https://linktr.ee/chursovg) в рамках сообщества [Serbian QA Hub](https://linktr.ee/serbianQAHub)

Данный МК рассматривает практическое применение подходов по созданию фреймворка на Java с нуля до работающих тестов в CI/CD.

Частично данный воркшоп реализует шаги по построению фреймворка от Bas Dijkstra: <https://github.com/basdijkstra/a-test-automation-project> c небольшими улучшениями и слегка измененным порядком шагов.

Нас ждут следующие шаги:

* Выбор технологического стека
* Подготовка окружения
* Создание нового проекта
* Создание проекта в GitHub
* Работа с GitHub и GitFlow
* Подключение зависимостей в проект
* Написание и запуск первого API теста
* Добавление статического анализа кода
* Использование паттернов, подходов
* Добавление класса констант, тестовых данных
* Параметризация тестов
* Написание и запуск первого UI теста
* Добавление ожиданий
* Добавление POM
* Добавление property файлов
* Добавление Dependency Injection
* Добавление конфигурации для запуска тестов через командную строку
* Добавление параметров для параллельного запуска
* Добавление отчетов
* Добавление логирования
* Настройка запуска тестов в CI/CD

Легенда: покрыто в мастер-классе, не покрыто, но важно для фреймворка

# 1 - Выбор технологического стека и подготовка окружения

Сначала нужно выбрать язык программирования.

Обычно это либо язык разработки, либо язык уже нанятых Automation инженеров, либо наиболее удобный язык для конкретной команды.

Список SDK для разных языков программирования:

* [Java](https://www.oracle.com/java/technologies/downloads/)
* [C#](https://dotnet.microsoft.com/en-us/download)
* [Python](https://www.python.org/downloads/)
* [JavaScript (Node.JS)](https://nodejs.org/en/download)

В зависимости от выбранного языка нужно установить среду разработки:

* Java - **IntelliJ**, Eclipse
* C# - Visual Studio
* Python - PyCharm
* JavaScript - Visual Studio Code, WebStorm

Нам в качестве примера подойдет IntelliJ Community Edition. Саму Java мы установим уже внутри IDE.

Теперь нам нужно определить какие зависимости мы будем использовать для Java языка.

Для UI: **Selenium**, Selenide, Playwright

Для API: **RestAssured**, Playwright, любой другой http клиент

Json: **Jackson**, Gson

Для запуска тестов: **JUnit**, TestNG

Assert’ы: **AssertJ**, Hamcrest

Сборщик проекта: Maven, **Gradle**

Отчеты: JUnit report, **Allure**, Report portal, Cucumber report

BDD: Cucumber, JBehave

Dependency injection: Spring, **Guice**

Генераторы кода: **Lombok**

Работа с БД: hibernate, jakarta

Property файлы: **Owner**

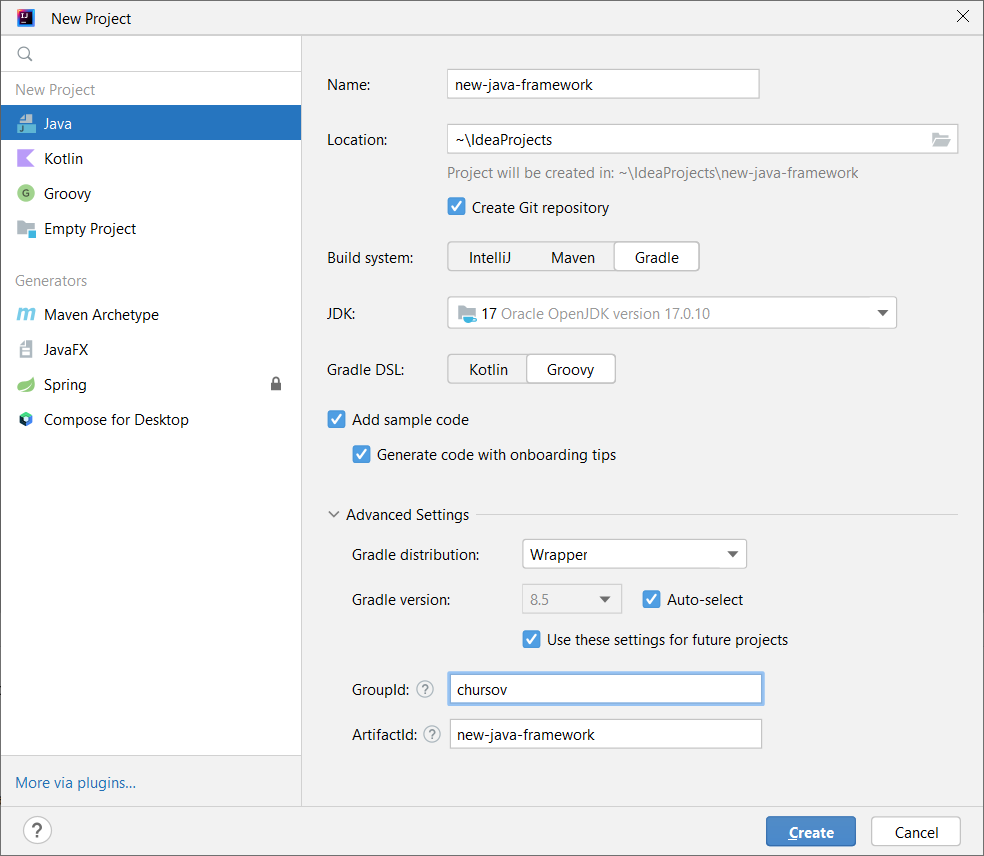
В рамках воркшопа будут использоваться **выделенные зависимости**.

# 2 - Создание нового проекта

<https://github.com/basdijkstra/a-test-automation-project/blob/main/01-creating-a-new-empty-project.md>

Используя IntelliJ создаем новый проект:

1. Указываем имя проекта, например, new-java-framework
2. Можем выбрать создание git repository (но для этого нужно авторизоваться под своим аккаунтов в IntelliJ)
3. Указываем Build System как Gradle. Если что у меня есть примеры и Maven проектов, либо можете изучить это самостоятельно.
4. Выбираем JDK 17. Если нужной JDK нет, можно выбрать Download и в Oracle JDK скачать любую, например Amazon Corretto.
5. Выбрать Gradle DSL: Groovy. На DSL Kotlin также можете найти примеры самостоятельно.
6. Нажмите Create.



1. Подождите пока не обновятся все базовые зависимости. Если необходимо, то нужно еще раз выбрать Setup SDK (File -> Project Structure -> Project -> SDK -> Java 17).
2. Откройте класс src/main/java/groupId/Main.java и запустите main метод.
3. Если в логе у вас есть текст:

“Hello and welcome!i = 1

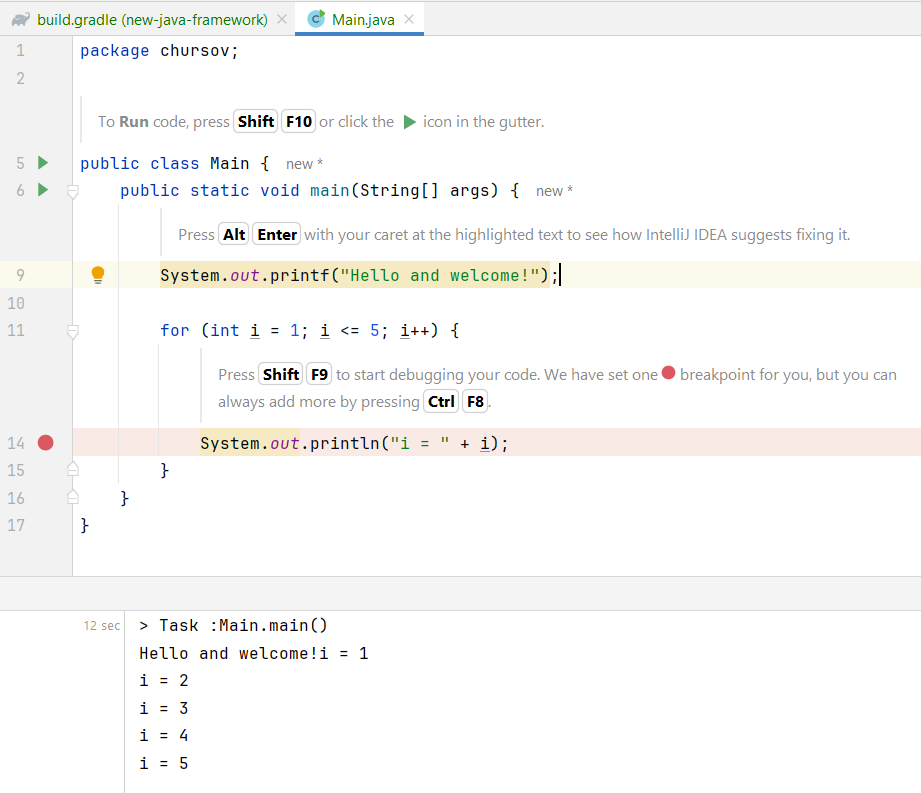
i = 2

i = 3

i = 4

i = 5”

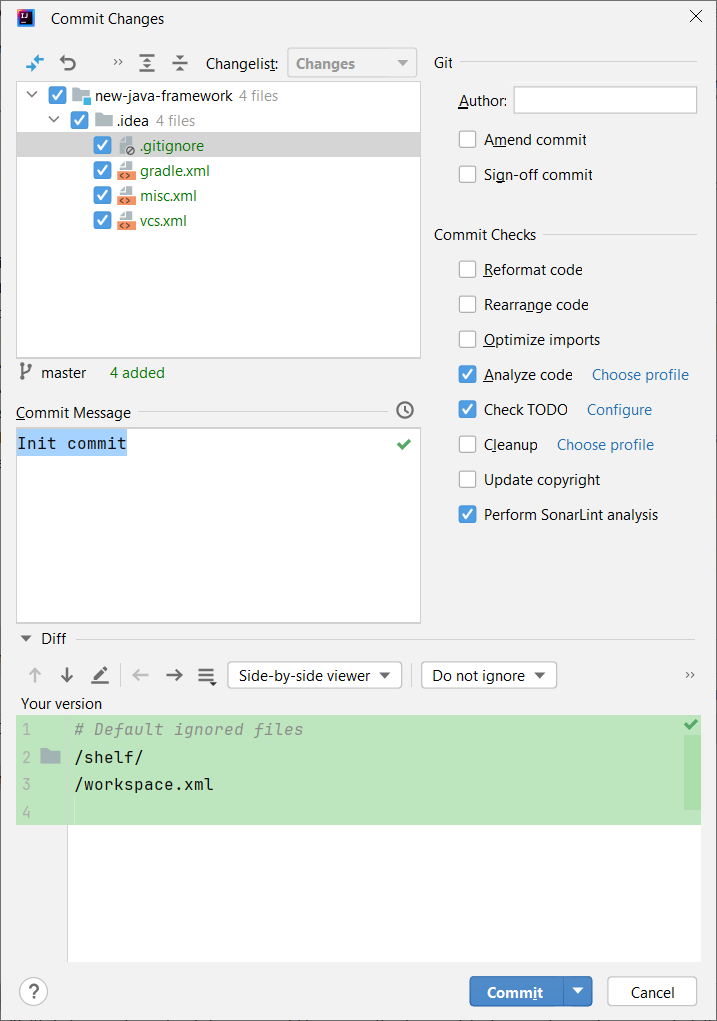
Значит вы все настроили правильно!



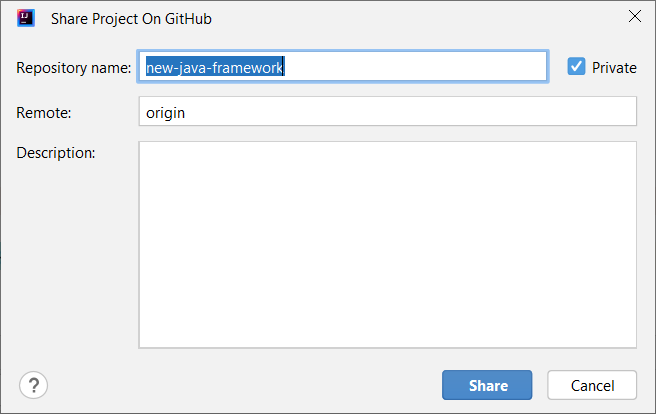
# 3 - Создание проекта в GitHub

Теперь пришло время залить наш код в удаленный репозиторий. Если вы изначально нажали галочку на создание Git репозитория, то теперь можно приступать к созданию коммита: Git -> Commit.

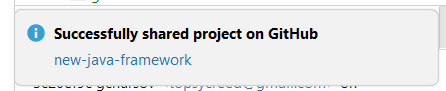
В диалоговом окне нужно убрать галочки с лишних файлов. В нашем случае это будут файлы из папки “.idea”. Отжимайте все эти файлы. Пишите комментарий в поле Commit message, например, Init commit и нажимайте на кнопку “Commit”.

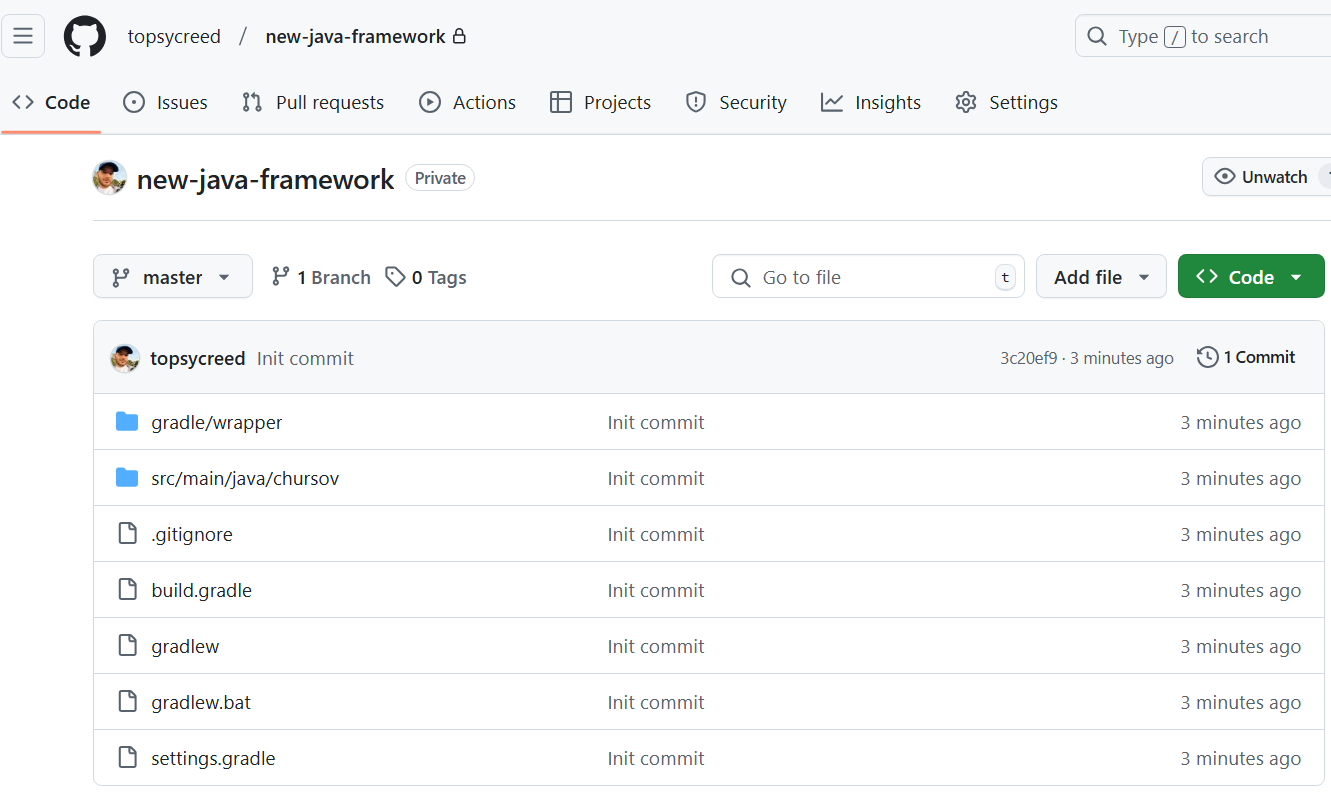


Затем нужно сделать создать репозиторий: Git -> GitHub -> Share project on GitHub. Выберете оставить ли проект приватным и нажмите Share.



Можете проверить, что в вашем GitHub появился новый репозиторий с единственный Init commit.





# 4 - Работа с GitHub и GitFlow

Теперь после первого коммита нужно подумать о двух вещах.

Первое - это о так называемом GitFlow. По хорошему мы больше не делаем коммиты напрямую в master/main ветку, а создаем новые ветки под каждую задачу. Также у нас могут быть разные варианты веток для релизов. Например, есть варианты с ветками master/dev. Ветками под конкретные версии релизов или так называемый trunk based. Почитать про это можно в этой статье <https://www.atlassian.com/continuous-delivery/continuous-integration/trunk-based-development>

Для наших целей давайте договоримся просто создавать новые ветки, делать в них локальные коммиты, после полной реализации делать push в удаленный репозиторий, делать pull request и после ревью кода мержить нашу ветку в master/main ветку нашего репозитория.

Сделайте новую ветку с помощью Git -> New Branch с названием, например: feat/no-ref/gitignore по шаблону: feat/JIRA-123/e2e-tests. **Я буду именовать ветки с нумерацией, чтобы вы видели постепенный прогресс проекта.**

*Примечание: В нашем случае у нас просто нет задачи в Jira под нужное нам изменение. На вашем проекте может быть свое наименование веток, а может быть даже настроена проверка, если что-то указано неправильно.*

Второе: это те самые файлы в папке .idea которые вечно будут пытаться попасть в наш репозиторий. Для того, чтобы раз и навсегда запретить эти папки в commit диалоге нужно обновить наш файл .gitignore

Изначально наш .gitignore имеет такие строки:

*### IntelliJ IDEA ###*

.idea/modules.xml

.idea/jarRepositories.xml

.idea/compiler.xml

.idea/libraries/

Нам нужно обновить до:

*### IntelliJ IDEA ###*

.idea/modules.xml

.idea/jarRepositories.xml

.idea/compiler.xml

.idea/libraries/

.idea/gradle.xml

.idea/misc.xml

.idea/vcs.xml

.idea/workspace.xml

Либо смотря какие файлы у вас хотят залиться. Перечислять весь список файлов на самом деле неудобно и легче игнорировать всю директорию с помощью:

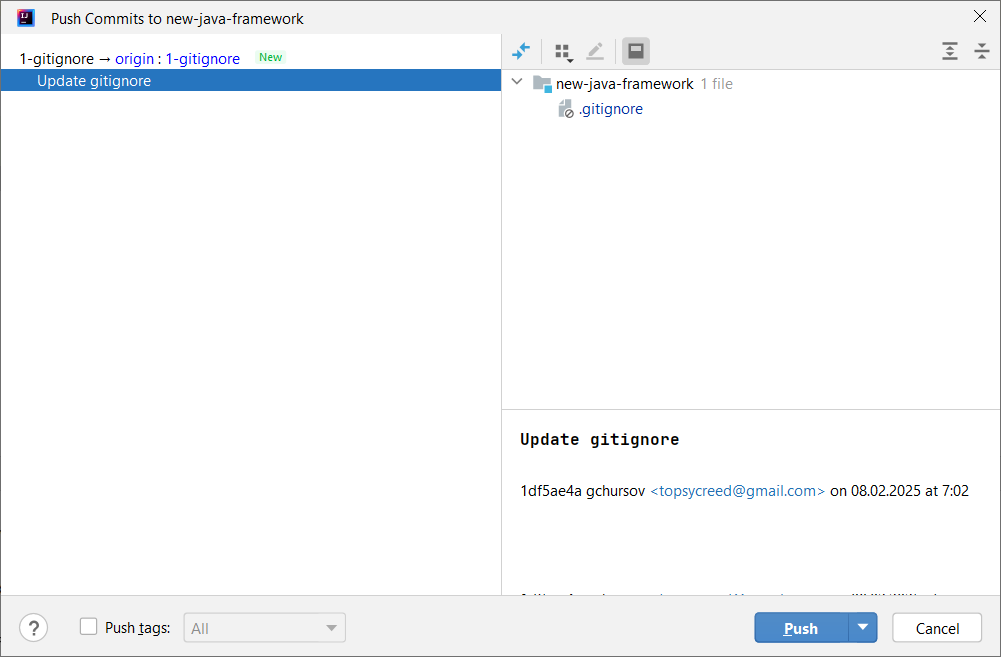
*### IntelliJ IDEA ###*

.idea/

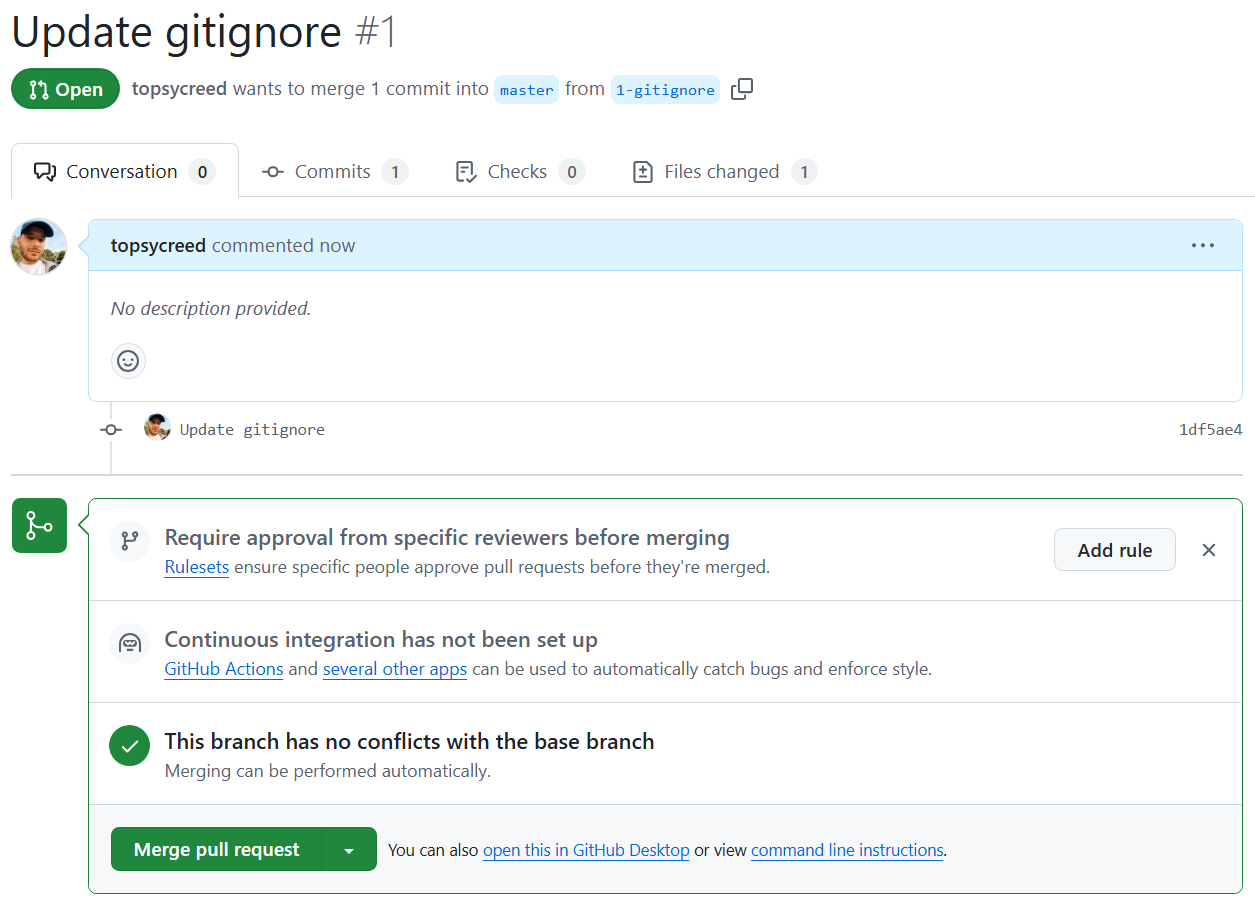
*Примечание: Если вы уже написали код в master/main ветке, то его можно предварительно сохранить в Stash через Git -> VCS operations popup -> Stash Changes. Дайте какое-то осмысленное название временному стешу. После этого создайте, перейдите в новую ветку и выполните: Git -> VCS operations popup -> Unstash Changes. Выберете свое осмысленное название и нажмите Apply (Pop тоже применит изменение, но одновременно удалит его из стеша). Если у вас не получается выполнить Unstash из-за изменившихся файлов в .idea то можно перенести изменения в .gitignore вручную.*

*Примечание 2: Если файлы из игнорируемой папки уже были добавлены через git add (отображаются в окне коммитов даже после добавления в gitignore), то сбросить можно с помощью git reset.*

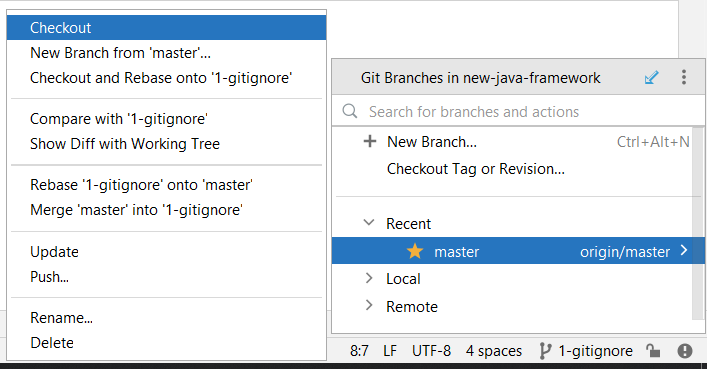
Сделаем коммит, push. Откройте GitHub и сделайте pull request, а потом откройте его через Compare & pull request и сделаете Pull Request и потом merge.

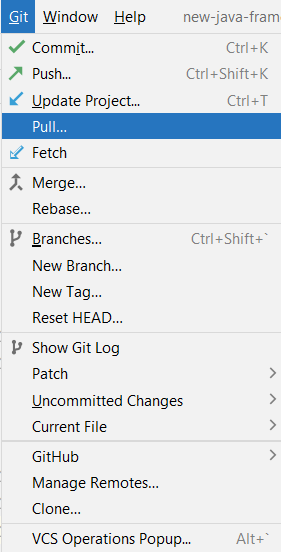


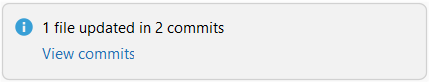




Теперь можно вернуться в наш проект, выбрать ветку master/main, сделать pull, чтобы подтянуть новые коммиты и сделать новую ветку под новую задачу.







Код ветки после обновления gitIgnore: <https://github.com/topsycreed/new-java-framework/tree/1-gitignore> и его pull request: <https://github.com/topsycreed/new-java-framework/pull/1>

# 5 - Подключение зависимостей в проект

<https://github.com/basdijkstra/a-test-automation-project/blob/main/02-working-with-package-managers.md>

Теперь когда у нас базовый проект, нужно добавить необходимые зависимости для старта автоматизации. Добавлять лучше только те зависимости, которые используются в текущей ветке git - не стоит добавлять зависимости на будущее, которые не применяются прямо сейчас.

В зависимости от сборщика: maven, gradle и DSL для gradle - меняется синтаксис для подключения зависимостей. Ниже будут примеры для Gradle + Groovy.

Добавим тест раннер для юнит тестов. JUnit 5 уже добавлен по умолчанию.

dependencies **{**

testImplementation platform('org.junit:junit-bom:5.10.0')

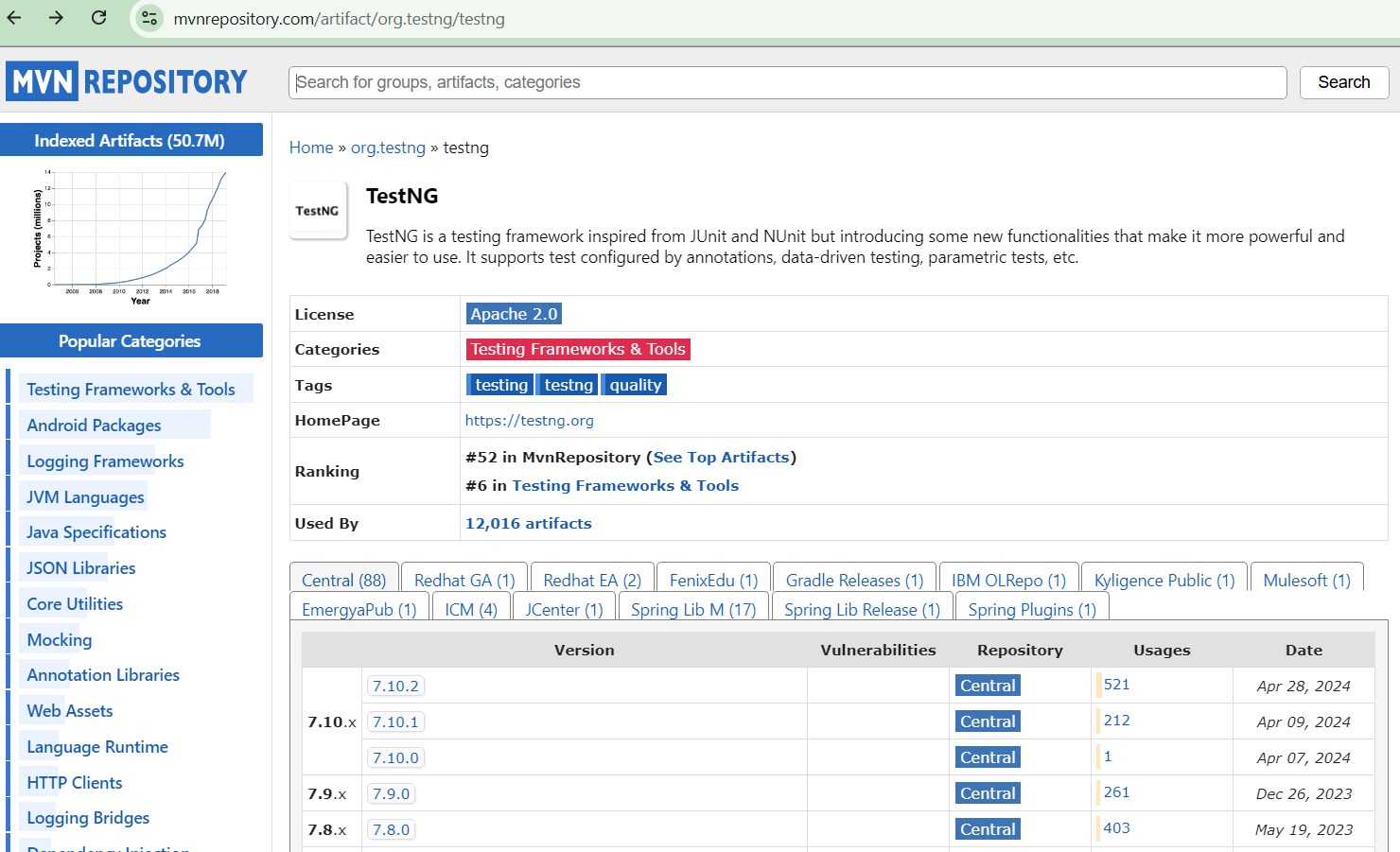
testImplementation 'org.junit.jupiter:junit-jupiter'

**}**

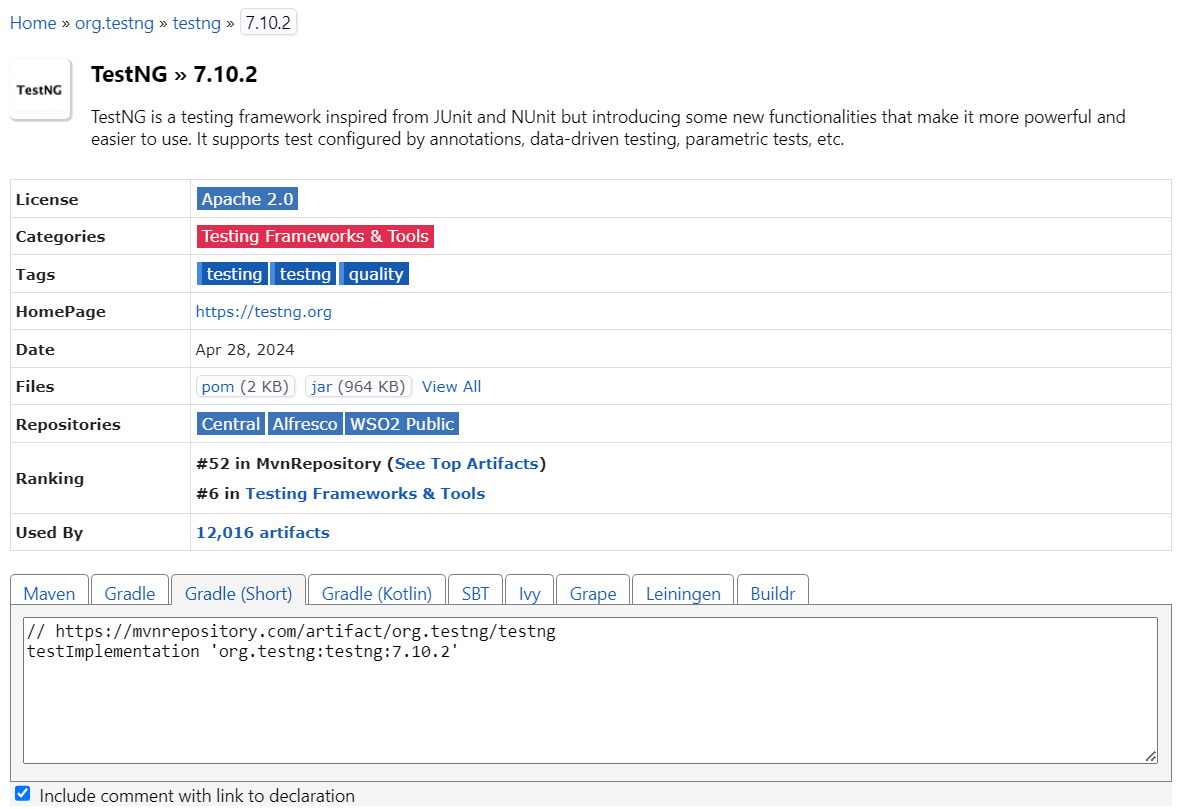
Чтобы добавить, например, TestNG, нужно добавить новую зависимость. Найти ее можно на сайте <https://mvnrepository.com/>. Вот ссылка на именно TestNG: <https://mvnrepository.com/artifact/org.testng/testng>

Страница зависимости/библиотеки содержит краткое описание, лицензию, категорию, теги, ссылку на сайт разработчиков, а также репозитории в которых можно скачать, мы будем использовать Central.

В репозитории артефакты библиотеки разделяются на версии, количество использований и дату релиза. Рекомендую использовать максимально возможную рабочую для вашего проекта версию.

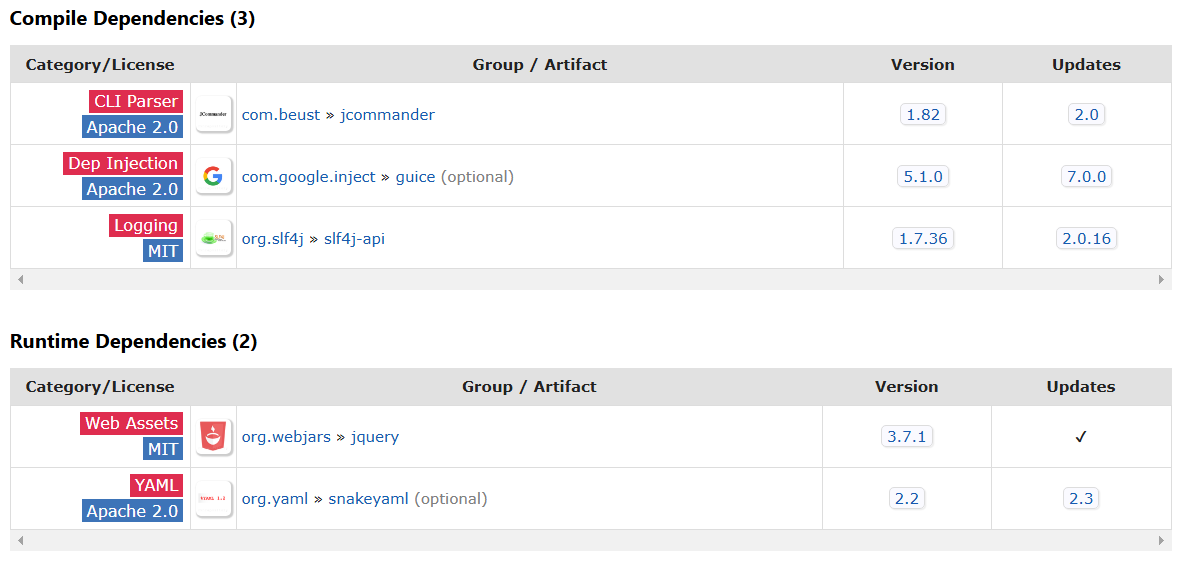


Конкретная версия артефакта уже содержит код для добавление в проект. По умолчанию это раздел Gradle (short). Но также есть варианты для Maven, Gradle (Kotlin).

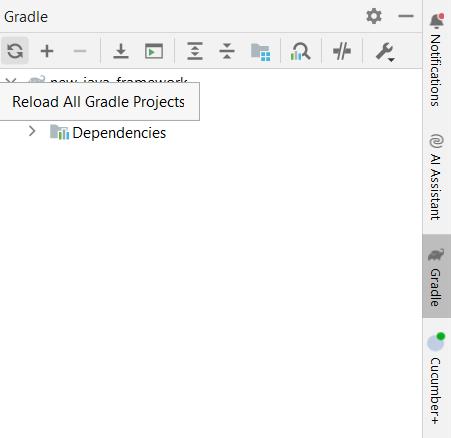


Также ниже на странице указаны дочерние зависимости артефакты в Compile Dependencies, с указанием их версий.

*Для расширенных настроек зависимостей иногда нужно специально отключать дочерние зависимости, чтобы использовать свой отдельный артефакт нужной версии.*

**

После добавления новой зависимости нужно обновить Gradle/Maven проект. Сделать это можно, например в окно Gradle через “Reload all Gradle projects”.



Проверить свои зависимости можно здесь же в Dependencies или продвинутый вариант через кнопку Analyze Dependencies (может пригодится как раз таки чтобы найти конфликтующие дочерние зависимости).

# 6 - Написание и запуск первого API теста

<https://github.com/basdijkstra/a-test-automation-project/blob/main/03-writing-and-running-a-first-test.md> + <https://github.com/basdijkstra/a-test-automation-project/blob/main/06-working-with-an-api-testing-library.md>

Давайте теперь напишем свой первый API тест на примере общедоступного Swagger: <https://petstore.swagger.io/>

1. Нужно добавить зависимость для работы с API вызовами в Java, желательно чтобы это был фреймворк для тестирования с нужными нам методами. Хороший вариант - библиотека REST Assured. <https://mvnrepository.com/artifact/io.rest-assured>

Итого у нас будут зависимости:

dependencies **{**

testImplementation platform('org.junit:junit-bom:5.10.0')

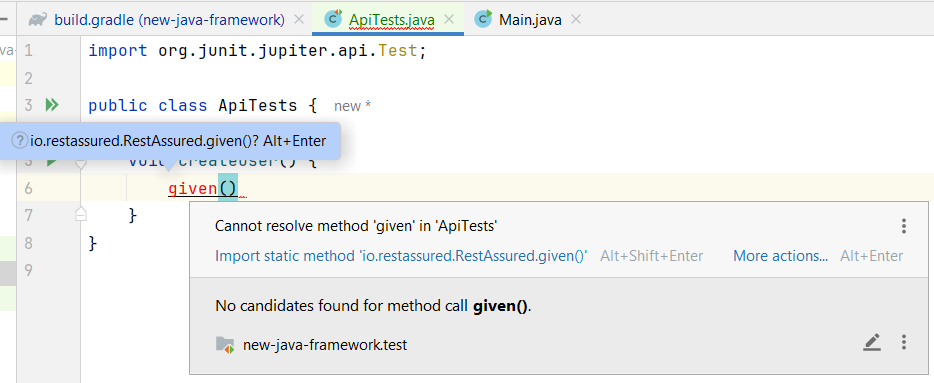
testImplementation 'org.junit.jupiter:junit-jupiter'

testImplementation 'io.rest-assured:rest-assured:5.5.0'

**}**

1. Пишем первый тест (он должен быть просто рабочим, пока что без всяких улучшалок).

По мере написания теста в класс с тестами нужно добавить импорт зависимостей. IDE подскажет какие зависимости можно автоматически добавить, если навести курсор мыши на код.



Получившийся тест:

@Test

void createUser() {

String baseUrl = "https://petstore.swagger.io/v2/";

String body = """

{

"id": 0,

"username": "string",

"firstName": "string",

"lastName": "string",

"email": "string",

"password": "string",

"phone": "string",

"userStatus": 0

}""";

Response response = *given*()

.baseUri(baseUrl)

.header("accept", "application/json")

.header("Content-Type", "application/json")

.body(body).

when().post("user").andReturn();

response.body().prettyPrint();

Assertions.*assertEquals*(200, response.statusCode());

}

1. Тест работает, даже несмотря на только лишь базовые проверки, можно пользоваться только RestAssured. Он позволяет проверять и конкретные значения полей в том числе, например, если использовать метод then():

@Test

void checkUserResponseBody() {

String baseUrl = "https://petstore.swagger.io/v2/";

String body = """

{

"id": 0,

"username": "string",

"firstName": "string",

"lastName": "string",

"email": "string",

"password": "string",

"phone": "string",

"userStatus": 0

}""";

*given*()

.baseUri(baseUrl)

.header("accept", "application/json")

.header("Content-Type", "application/json")

.body(body).

when()

.post("user")

.then()

.statusCode(200)

.body("code", *equalTo*(200))

.body("type", *equalTo*("unknown"))

.body("message", *notNullValue*(String.class));

}

1. Создаем коммит (для новой ветки), делаем push, создаем pull request, мержим его, снова переходим в master ветку, где делаем pull.

Код ветки после добавления простых API тестов: <https://github.com/topsycreed/new-java-framework/tree/2-api> и его Pull request: <https://github.com/topsycreed/new-java-framework/pull/2>

*Примечание: в тестах я уже использовал формат AAA. Arrange -> Act -> Assert для визуального разделения теста на этапы. Поподробнее можно почитать:* [*https://java-design-patterns.com/patterns/arrange-act-assert/#programmatic-example-of-arrange-act-assert-pattern-in-java*](https://java-design-patterns.com/patterns/arrange-act-assert/#programmatic-example-of-arrange-act-assert-pattern-in-java)

# 7 - Использование паттернов, подходов

<https://github.com/basdijkstra/a-test-automation-project/blob/main/07-parameterizing-your-tests.md> + <https://github.com/basdijkstra/a-test-automation-project/blob/main/08-serializing-objects-to-json.md>

Даже уже после добавления второго теста можно увидеть ряд проблем:

1. Дублируем baseUrl.
2. Дублируем body, а еще и храним json файл прямо в тестах.
3. Дублируем добавление одних и тех же headers.
4. В целом все тоже самое, только разные проверки в конце.
5. Автотест выглядит слишком перегруженным и детали реализации можно скрыть оставив только наши намерения в виде последовательности шагов: Отправить post запрос на endpoint /user с валидным body и проверить: статус код 200, 3 поля из ответа.
6. Кроме того, в других тестах мы часто будем переиспользовать методы POST, GET, DELETE - чтобы создать нового юзера, проверить его данные и удалить в конце теста, чтобы не засорять базу данных.

Давайте начнем исправлять данные проблемы. Одним из вариантов исправления может быть разделение кода на контроллеры, которые соответствуют контроллерам разработчиков и Swagger и предоставлению методов для работы с CRUD операциями: GET, POST, PUT, PATCH, DELETE.

Давайте создадим новый пакет controllers и класс UserController.

1. Вынесем RequestSpecification requestSpecification;как отдельное поле, которое будем содержать базовые настройки и хедеры.
2. В конструкторе класса заполним requestSpecification нужными значениями.
3. Вынесем базовый url в константу класса: private static final String *BASE\_URI* = "p "; и также private static final String *USER\_ENDPOINT* = "user";
4. Реализуем шаги для POST/user (createUser), GET/user/{username} (Get user by username), PUT/user/{username} (Updated user) и DELETE/user/{username} (Delete user).

В итоге получится:

public Response createUser(User user) {

return *given*(this.requestSpecification)U

.body(user)

.when()

.post(*USER\_ENDPOINT*)

.andReturn();

}

public Response updateUser(User user) {

return *given*(this.requestSpecification)

.body(user)

.when()

.put(*USER\_ENDPOINT* + "/" + user.getUsername())

.andReturn();

}

public Response getUserByUsername(String username) {

return *given*(this.requestSpecification)

.when()

.get(*USER\_ENDPOINT* + "/" + username)

.andReturn();

}

public Response deleteUserByUsername(String username) {

return *given*(this.requestSpecification)

.when()

.delete(*USER\_ENDPOINT* + "/" + username)

.andReturn();

}

1. На данном этапе у вас не будет класса User который нужно добавить в отдельный package - models или dto. Сгенерировать данный класс можно, например, с помощью <https://www.jsonschema2pojo.org/>. *Также на этом этапе можно подключить дополнительную библиотеку Lombok, чтобы с помощью аннотаций генерировать: геттеры, сеттеры, конструкторы, билдеры, методы toString, equals/hashCode и т.д.*

*ext* ***{***

*lombokVersion = '1.18.36'*

***}***

*compileOnly "org.projectlombok:lombok:$****{****lombokVersion****}****"*

*annotationProcessor "org.projectlombok:lombok:$****{****lombokVersion****}****"*

*testCompileOnly "org.projectlombok:lombok:$****{****lombokVersion****}****"*

*testAnnotationProcessor "org.projectlombok:lombok:$****{****lombokVersion****}****"*

В самом простом варианте с Lombok класс будет выглядеть так:

@Data

public class User {

private int id;

private String username;

private String firstName;

private String lastName;

private String email;

private String password;

private String phone;

private int userStatus;

}

*Задание со звездочкой: исправьте поле userStatus - на самом деле оно должно быть не типом int, а Enum UserStatus со значениями: ACTIVE(0), INACTIVE(1), SUSPENDED(2);*

1. Теперь давайте напишем аналогичные нашим первым тесты на User, но используя уже контроллер:

Пример получившегося теста:

@Test

void createUserControllerTest() {

User user = new User(0,

"username",

"firstName",

"lastName",

"email",

"password",

"phone",

0);

Response response = userController.createUser(user);

AddUserResponse createdUserResponse = response.as(AddUserResponse.class);

Assertions.*assertEquals*(200, response.statusCode());

Assertions.*assertEquals*(200, createdUserResponse.getCode());

Assertions.*assertEquals*("unknown", createdUserResponse.getType());

Assertions.*assertFalse*(createdUserResponse.getMessage().isEmpty());

}

Уже лучше? В качестве домашнего задания реализуйте тесты на PUT, GET, DELETE методы.

1. Запустите данный тест, оказалось что он упал с ошибкой: Cannot serialize object because no JSON serializer found in classpath. Please put Jackson (Databind), Gson, Johnzon, or Yasson in the classpath. Чтобы исправить ее можно добавить любую из перечисленных зависимостей. Давайте добавим библиотеку Jackson, также я работал и с Gson. <https://mvnrepository.com/artifact/com.fasterxml.jackson.core/jackson-databind>
2. Ура, тест работает! Можно создавать коммит, мержить, все дела.

*Также здесь есть вариант вместо конструктора для создания объекта User использовать паттерн Builder. Его легко добавить с помощью аннотации Lombok @Builder.*

*Так будет выглядеть наш объект:*

*User userBuilder = User.builder()*

*.username("username")*

*.firstName("firstName")*

*.lastName("lastName")*

*.email("email")*

*.phone("password")*

*.userStatus(0)*

*.build();*

Код ветки после добавления контроллера и обновления API тестов: <https://github.com/topsycreed/new-java-framework/tree/3-controller> и его Pull request: <https://github.com/topsycreed/new-java-framework/pull/3>

# 8 - Добавление класса констант, тестовых данных

Еще один вариант улучшить код автотестов - это вынести повторяющиеся переменные в класс констант, а также сделать отдельный класс для тестовых данных.

Претенденты на вынесение у нас уже есть: *BASE\_URI*

и создание объекта User.

1. Создадим новый класс в package constants.

Перенесем туда нашу константу: public static final String *BASE\_URI* = "https://petstore.swagger.io/v2/";

Теперь уже как public поле, так как доступ у нее должен быть для любого класса. После этого нужно удалить константу в контроллере и сделать новый импорт.

1. Создадим новый класс TestData в package testdata.

Перенесем туда нашего созданного с помощью Builder паттерна пользователя, а также тут же можно добавлять новые тест данные, например, пустого пользователя для негативного теста.

public static final User *DEFAULT\_USER* = User.*builder*()

.username("username")

.firstName("firstName")

.lastName("lastName")

.email("email")

.phone("password")

.userStatus(0)

.build();

public static final User *INVALID\_USER* = User.*builder*().build();

Обновите тесты чтобы использовать тестовые данные.

Тест будет выглядеть теперь так:

@Test

void createUserControllerTest() {

Response response = userController.createUser(*DEFAULT\_USER*);

AddUserResponse createdUserResponse = response.as(AddUserResponse.class);

Assertions.*assertEquals*(200, response.statusCode());

Assertions.*assertEquals*(200, createdUserResponse.getCode());

Assertions.*assertEquals*("unknown", createdUserResponse.getType());

Assertions.*assertFalse*(createdUserResponse.getMessage().isEmpty());

}

Намного минималистичнее, при этом смысл сохраняется благодаря названию константы.

*Еще есть вариант без создания отдельного класса - можно насоздавать пользователей прямо в классе User.*

*Кроме классов с тест данными для проекта можно использовать библиотеку* [*Java Faker*](https://mvnrepository.com/artifact/com.github.javafaker/javafaker/1.0.2)*, которая генерирует рандомные тестовые данные. Поподробнее про библиотеку:* [*https://www.baeldung.com/java-faker*](https://www.baeldung.com/java-faker)

1. Коммит, мерж, все дела 🙂

Код ветки после добавления констант и обновления API тестов: <https://github.com/topsycreed/new-java-framework/tree/4-constants> и его Pull request: <https://github.com/topsycreed/new-java-framework/pull/4>

# 9 - Параметризация тестов

<https://github.com/basdijkstra/a-test-automation-project/blob/main/07-parameterizing-your-tests.md>

В прошлый раз у нас получилось 2 странных теста, они практически одинаковые, так как даже на пустого пользователя сервис все равно возвращает 200 код. Единственное, что отличается, это что message возвращается как “0”.

Давайте попробуем объединить оба теста в один параметризованный и вынести только разницу между тестами в параметры.

Для параметризации с помощью JUnit 5 можно использовать аннотацию @ParameterizedTest. Для TestNG - это будет @DataProvider. Подробнее: <https://toolsqa.com/testng/testng-dataproviders/>

А мы реализуем примеры на JUnit 5.

static Stream<User> users() {

return Stream.*of*(*DEFAULT\_USER*, *INVALID\_USER*);

}

@ParameterizedTest

@MethodSource("users")

void createUserParametrizedTest(User user) {

Response response = userController.createUser(user);

AddUserResponse createdUserResponse = response.as(AddUserResponse.class);

Assertions.*assertEquals*(200, response.statusCode());

Assertions.*assertEquals*(200, createdUserResponse.getCode());

Assertions.*assertEquals*("unknown", createdUserResponse.getType());

Assertions.*assertFalse*(createdUserResponse.getMessage().isEmpty());

}

Тест выполняется два раза, так как в static Stream<User> users() мы задали оба варианта: *DEFAULT\_USER*, *INVALID\_USER*

Не забудьте исправить отдельный тест, где нужно проверять:

Assertions.*assertEquals*("0", createdUserResponse.getMessage());

Подробнее про параметризованные тесты на JUnit 5: <https://www.baeldung.com/parameterized-tests-junit-5>

Как обычно: коммит, мерж, и т.д.

Код ветки после добавления параметризированных тестов: <https://github.com/topsycreed/new-java-framework/tree/5-parametrized> и его Pull request: <https://github.com/topsycreed/new-java-framework/pull/5>

# 10 - Написание и запуск первого UI теста

<https://github.com/basdijkstra/a-test-automation-project/blob/main/09-adding-a-first-ui-test.md>

Давайте теперь напишем первый UI тест.

Тестировать будем специальный сайт <https://bonigarcia.dev/selenium-webdriver-java/> на котором много элементов для тренировок.

1. Нужно добавить зависимость на UI библиотеку, это может быть что угодно из списка: Selenium, Selenide, Playwright. Я выберу базу - чистый Selenium. <https://mvnrepository.com/artifact/org.seleniumhq.selenium/selenium-java>
2. Теперь собственно сам тест:

WebDriver driver;

@BeforeEach

void setup() {

driver = new ChromeDriver();

}

@AfterEach

void tearDown() {

driver.quit();

}

@Test

void submitWebFormTest() {

driver.get(*UI\_BASE\_URL*);

driver.findElement(By.*linkText*("Web form")).click();

driver.findElement(By.*id*("my-text-id")).sendKeys("Text");

driver.findElement(By.*xpath*("//button[@type = 'submit']")).click();

WebElement title = driver.findElement(By.*className*("display-6"));

Assertions.*assertEquals*("Form submitted", title.getText());

}

Тут нам нужно вынести создание браузера и закрытие страницы в @BeforeEach и @AfterEach чтобы браузер закрывался даже в случае падения теста. А также нужно добавить публичную константу *UI\_BASE\_URL*

1. Как обычно: коммит, мерж, и т.д.

Код ветки после добавления первого UI теста: <https://github.com/topsycreed/new-java-framework/tree/6-ui> и его Pull request: <https://github.com/topsycreed/new-java-framework/pull/6>

# 11 - Добавление ожиданий

<https://github.com/basdijkstra/a-test-automation-project/blob/main/11-waiting-and-synchronization-techniques.md>

Если мы захотим создать несколько наборов тестов в разных классах, то нам нужно вынести создание драйвера, поднятие и закрытие браузера в отдельный класс. Тут наконец-то применим наследование!

Давайте создадим класс BaseTest c содержимым:

public class BaseTest {

WebDriver driver;

@BeforeEach

void setup() {

driver = new ChromeDriver();

}

@AfterEach

void tearDown() {

driver.quit();

}

}

Соответственно в классе с самими тестами можно удалить этот код и добавить наследование: class UiTests extends BaseTest

Теперь если мы хотим задать ожидание по умолчанию его можно добавить сразу же в BaseTest класс.

@BeforeEach

void setup() {

driver = new ChromeDriver();

driver.manage().timeouts().implicitlyWait(Duration.*ofSeconds*(2));

}

Также можно сразу же создать самые часто используемые explicit ожидания, например:

WebDriver driver;

WebDriverWait longWait;

@BeforeEach

void setup() {

driver = new ChromeDriver();

driver.manage().timeouts().implicitlyWait(Duration.*ofSeconds*(2));

longWait = new WebDriverWait(driver, Duration.*ofSeconds*(10));

}

Теперь давайте напишем несколько тестов на implicit и explicit ожидания:

@Test

void loadingImagesImplicitWaitTest() {

driver.get("https://bonigarcia.dev/selenium-webdriver-java/loading-images.html");

WebElement compass = driver.findElement(By.*id*("compass"));

WebElement calendar = driver.findElement(By.*id*("calendar"));

WebElement award = driver.findElement(By.*id*("award"));

WebElement landscape = driver.findElement(By.*id*("landscape"));

SoftAssertions softly = new SoftAssertions();

softly.assertThat(compass.getAttribute("src")).containsIgnoringCase("compass");

softly.assertThat(calendar.getAttribute("src")).containsIgnoringCase("calendar");

softly.assertThat(award.getAttribute("src")).containsIgnoringCase("award");

softly.assertThat(landscape.getAttribute("src")).containsIgnoringCase("landscape");

softly.assertAll();

}

@Test

void loadingImagesExplicitWaitTest() {

driver.get("https://bonigarcia.dev/selenium-webdriver-java/loading-images.html");

WebElement landscape = longWait.until(ExpectedConditions.*presenceOfElementLocated*(By.*id*("landscape")));

*assertThat*(landscape.getAttribute("src")).containsIgnoringCase("landscape");

}

*Здесь я для удобства воспользовался еще одной библиотекой* [*https://mvnrepository.com/artifact/org.assertj/assertj-core*](https://mvnrepository.com/artifact/org.assertj/assertj-core) *для того, чтобы воспользоваться дополнительными методами и SoftAssertion.*

*Поподробнее про ожидания в Selemium:* [*https://www.baeldung.com/selenium-implicit-explicit-wait*](https://www.baeldung.com/selenium-implicit-explicit-wait)

Как обычно мержим наши изменения в мастер после создания pull request.

Код ветки после добавления ожиданий в UI тесты: <https://github.com/topsycreed/new-java-framework/tree/7-waits> и его Pull request: <https://github.com/topsycreed/new-java-framework/pull/7>

# 12 - Добавление Page Object Model (POM)

<https://github.com/basdijkstra/a-test-automation-project/blob/main/12-separating-the-what-from-the-how.md>

Также как и с контроллерами для UI тестов тоже есть свои паттерны, одним из них считается Page Object Model или сокращенно POM. Почитать про него: <https://www.selenium.dev/documentation/test_practices/encouraged/page_object_models/>

Создаем сначала базовый класс в пакет pages:

public class BasePage {

protected WebDriver driver;

protected WebDriverWait wait;

public BasePage(WebDriver driver, WebDriverWait wait) {

this.driver = driver;

this.wait = wait;

PageFactory.*initElements*(driver, this);

}

}

Потом от него создаем нужные нам страницы в том же пакете pages:

public class LoginPage extends BasePage {

@FindBy(id = "username")

private WebElement usernameInput;

@FindBy(id = "password")

private WebElement passwordInput;

@FindBy(xpath = "//button[@type='submit']")

private WebElement loginButton;

private static final String *VALID\_USER* = "user";

private static final String *VALID\_PASSWORD* = "user";

public LoginPage(WebDriver driver, WebDriverWait wait) {

super(driver, wait);

driver.get("https://bonigarcia.dev/selenium-webdriver-java/login-form.html");

}

public void login() {

usernameInput.sendKeys(*VALID\_USER*);

passwordInput.sendKeys(*VALID\_PASSWORD*);

loginButton.click();

}

}

Тест будет выглядеть так:

class UiPomTests extends BaseTest {

@Test

void loginPomTest() {

LoginPage loginPage = new LoginPage(driver, longWait);

loginPage.login();

*assertThat*(driver.getCurrentUrl()).contains("login-sucess");

}

}

Тесты нужно перенести в package tests, чтобы сделать наследование.

*Задание со звездочкой: Добавьте переход со страницы на страницу. Также можно создать отдельные классы для повторяющихся элементов страниц, обычно это header, footer, какие-то меню или формы.*

Коммит мерж, все дела.

Код ветки после добавления page object в UI тесты: <https://github.com/topsycreed/new-java-framework/tree/8-pom> и его Pull request: <https://github.com/topsycreed/new-java-framework/pull/8>

# 13 - Добавление property файлов и конфигурации для запуска через командную строку

Что если наши тесты содержат конфиденциальные данные, которые мы не хотим добавлять в код и шарить на весь интернет? Или у нас есть несколько тестовых окружений: test, dev, intergration, prod и мы не хотим в классе констант плодить столько версий BASE\_URL?

Решение есть - проперти файлы и библиотека Owner для простой работы с ними.

Добавим библиотеку: <https://mvnrepository.com/artifact/org.aeonbits.owner/owner>

Создадим конфигурационный класс:

@Config.Sources({

"classpath:${env}.properties",

"classpath:default.properties"

})

public interface TestPropertiesConfig extends org.aeonbits.owner.Config {

@Key("apiBaseUrl")

String getApiBaseUrl();

@Key("uiBaseUrl")

String getUiBaseUrl();

@Key("login")

String getLogin();

@Key("password")

String getPassword();

}

Также в resources создадим сами файлы properties:

apiBaseUrl=https://petstore.swagger.io/v2/

uiBaseUrl=https://bonigarcia.dev/selenium-webdriver-java/

login=

password=

Чтобы использовать переменные в классах с API, UI тестами нужно инициализировать конфиг с помощью: TestPropertiesConfig configProperties = ConfigFactory.*create*(TestPropertiesConfig.class, System.*getProperties*());

Задавать конфиг и конкретные значения можно, если добавить запуск через консоль в build.gradle файл:

test **{**

useJUnitPlatform()

systemProperties(System.*getProperties*())

**}**

Это позволит читать системные переменные и запустить тесты через консольную команду: gradle test -DuiBaseUrl="test"

**Не коммите в проперти файлы свои логины и пароли, добавьте только пустой проперти файл, добавьте его в gitignoreи потом храните логины только локально!**

Коммит, мерж, и так далее.

Код ветки после добавления проперти файлов: <https://github.com/topsycreed/new-java-framework/tree/9-properties> и его Pull request: <https://github.com/topsycreed/new-java-framework/pull/9>

# 14 - Добавление отчетов

<https://github.com/basdijkstra/a-test-automation-project/blob/main/13-further-improvements-for-our-code.md>

Если запустить наши тесты с конфигом для test где лежат неправильные BASE\_URL часть наших тестов упадет. Но теперь когда мы запускаем тесты пачкой сложно отследить что с ними не так. Удобно на этом этапе добавить логирование и отчеты, например, Allure Report.

Чтобы добавить его нужно добавить зависимость: <https://mvnrepository.com/artifact/io.qameta.allure/allure-junit5> и <https://mvnrepository.com/artifact/io.qameta.allure/allure-rest-assured>

И добавить плагин <https://plugins.gradle.org/plugin/io.qameta.allure>

plugins **{**

id 'java'

id 'io.qameta.allure' version '2.12.0'

**}**

Чтобы добавить автоматическое логирование запросов, ответов нужно добавить в конфиг контроллера:

public UserController() {

this.requestSpecification = *given*()

.accept(*JSON*)

.contentType(*JSON*)

.baseUri(configProperties.getApiBaseUrl())

.filter(new AllureRestAssured());

}

Для логирования UI тестов нужно добавить аннотации Allure и генерацию скриншотов при падениях.

Также для выполнения команды по генерации Allure отчетов его нужно сначала установить: <https://allurereport.org/docs/install/>

Для gradle нужно перейти в папку build/allure-results и выполнить команду allure serve.

Код ветки после добавления проперти файлов: <https://github.com/topsycreed/new-java-framework/tree/10-allure> и его Pull request: <https://github.com/topsycreed/new-java-framework/pull/10>

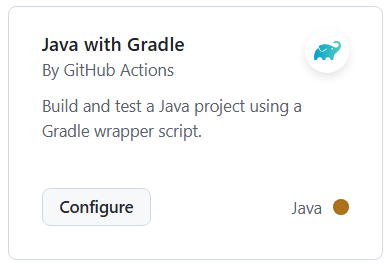
# 15 - Настройка запуска тестов в CI/CD: <https://github.com/basdijkstra/a-test-automation-project/blob/main/05-running-our-tests-in-a-ci-pipeline.md>

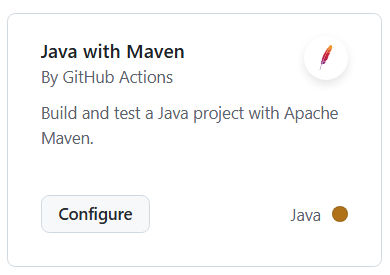
Примеры проектов с настроенным GitHub Actions:

1. <https://github.com/topsycreed/new-java-framework/blob/master/.github/workflows/gradle.yml>
2. <https://github.com/topsycreed/unitTestsGradle/actions/workflows/smoke.yml>
3. <https://github.com/allure-framework/allure-demo/blob/master/.github/workflows/build.yaml>

Шаги:

1. Откройте вкладку Actions
2. Выберите шаблон для Java (Gradle или Maven) через Configure





1. У вас сгенерируется базовый шаблон для вашего workflow в виде .yml файла. Этот GitHub Actions workflow предназначен для автоматической сборки Java-проекта с Gradle и управления зависимостями. Разберём его по блокам:
   1. name: Java CI with Gradle - имя workflow, можно поменять, например, на API Smoke tests.
   2. ‘on’ - определяет, когда запускать этот workflow, При push в ветку master. При pull\_request в ветку master.

on:

push:

branches: [ "master" ]

pull\_request:

branches: [ "master" ]

Если хочется настроить запуск джобы не по триггеру, а по ручному запуску, то будет выглядеть как:

on:

workflow\_dispatch:

* 1. Определяется Job с именем build, ‘runs-on’ - которая выполняется на последней версии Ubuntu. ‘permissions: contents: read’ - эта job имеет доступ только на чтение репозитория.

jobs:

build:

runs-on: ubuntu-latest

permissions:

contents: read

В нашем случае, для того, чтобы выкладывать allure отчеты нужно добавить права на запись, в итоге будет выглядеть как:

jobs:

build:

runs-on: ubuntu-latest

permissions:

contents: write

pages: write

* 1. Теперь переходим к шагам ‘steps:’ в джобе build. uses: actions/checkout@v4 - заготовленный шаг, который клонирует код репозитория, actions/setup-java@v4 - устанавливает JDK 17 (Temurin), где через ‘with:’ можно добавить параметры в шаг.

steps:

- uses: actions/checkout@v4

- name: Set up JDK 17

uses: actions/setup-java@v4

with:

java-version: '17'

distribution: 'temurin'

* 1. gradle/actions/setup-gradle - устанавливает Gradle с кэшем зависимостей, а ./gradlew build - это уже команда самого gradle, чтобы собрать проект. Это тоже самое, как локально мы делали запуск тестов через командную строку через gradle clean test.

- name: Setup Gradle

uses: gradle/actions/setup-gradle@af1da67850ed9a4cedd57bfd976089dd991e2582 # v4.0.0

- name: Build with Gradle Wrapper

run: ./gradlew build

Тут нам нужно заменить ./gradlew build на наш запуск автотестов, например: ./gradlew test -Denv="test"

* 1. dependency-submission: - необходимо для Dependabot Alerts — автоматического отслеживания уязвимостей в зависимостях. В нашем случае можно либо оставить, либо удалить, напрямую на тесты это не влияет.

dependency-submission:

runs-on: ubuntu-latest

permissions:

contents: write

steps:

- uses: actions/checkout@v4

- name: Set up JDK 17

uses: actions/setup-java@v4

with:

java-version: '17'

distribution: 'temurin'

* 1. Если оставить теперь все как есть, то мы сможем запускать автотесты, но у нас не будет генерироваться Allure отчет, для этого нужно добавить еще несколько шагов в yml файл. Сначала нам нужно настроить копирование отчетов из отдельной ветки ‘gh-pages’ (позднее нужно будет добавить ее в настройках проекта). ‘if: always()’ гарантирует выполнение этого шага всегда, даже если предыдущие шаги упадут. Шаг нужен чтобы сохранить историю предыдущих Allure отчетов, а не перезаписывать их полностью при каждом запуске.

- name: Copy History

uses: actions/checkout@v2

if: always()

with:

ref: gh-pages

path: gh-pages

* 1. Шаг самой сборки Allure отчета. Тут используется сторонний action simple-elf/allure-report-action@v1.10, почитать поподробнее про него можно [тут](https://github.com/simple-elf/allure-report-action). ‘if: always()’ - шаг нужно выполнять даже если тесты не прошли и упали. Через with нужно указать путь к allure-results. Для gradle - build/allure-results, для maven скорее всего target/allure-results. allure\_history: allure-history - папка, в которую копируется история отчетов из gh-pages. keep\_reports: 20 — хранит 20 последних отчетов, более старые удаляются.

- name: Build Report

uses: simple-elf/allure-report-action@v1.10

if: always()

with:

allure\_results: build/allure-results

allure\_history: allure-history

keep\_reports: 20

* 1. Ну и последний шаг c action: peaceiris/actions-gh-pages@v2 - публикация отчета в GitHub Pages. PERSONAL\_TOKEN: ${{ secrets.GITHUB\_TOKEN }} — авторизация через токен GitHub. PUBLISH\_BRANCH: gh-pages — обновляет ветку gh-pages, где хранится отчёт. PUBLISH\_DIR: allure-history — загружает новый отчет в ветку gh-pages.

- name: Publish Report

if: always()

uses: peaceiris/actions-gh-pages@v2

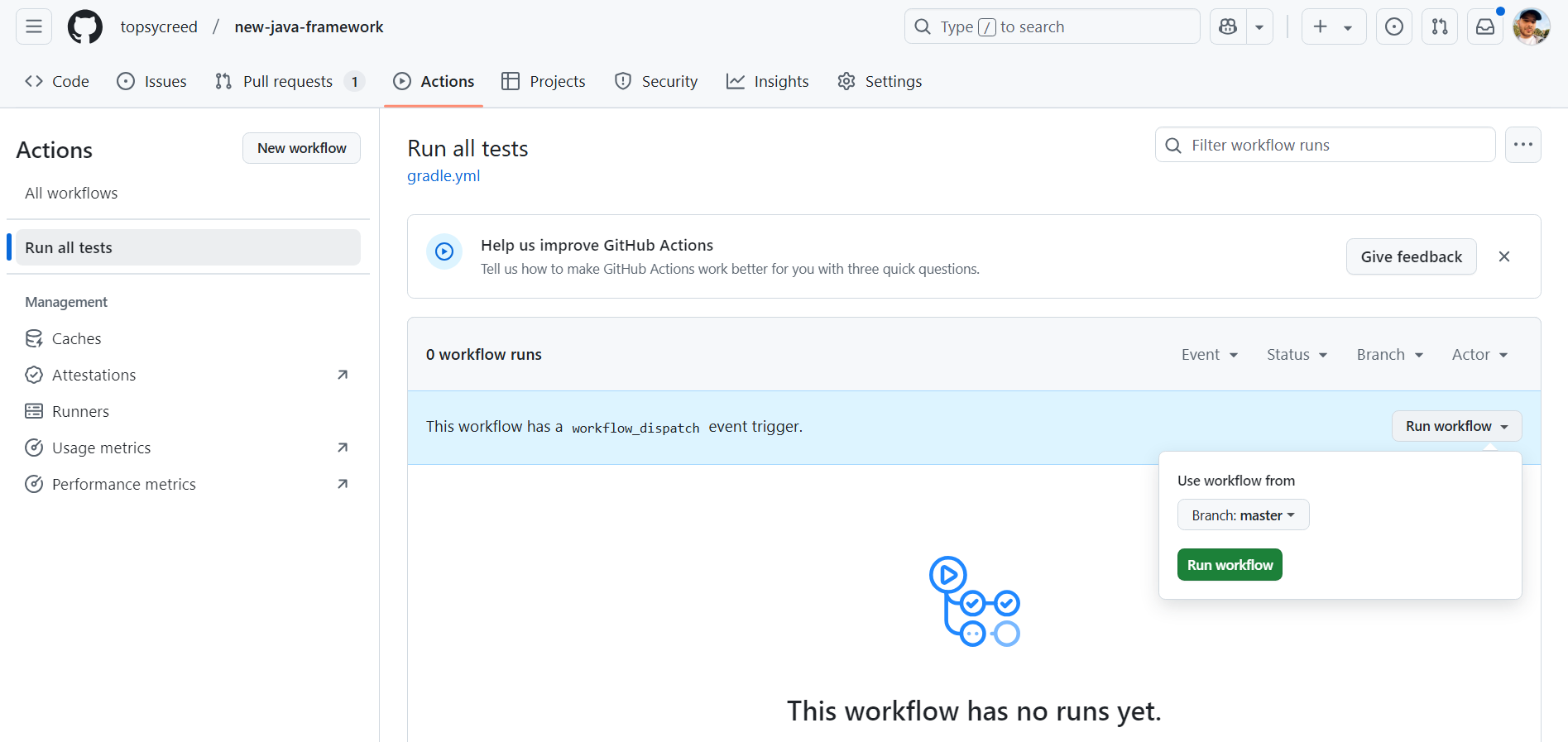
env:

PERSONAL\_TOKEN: ${{ secrets.GITHUB\_TOKEN }}

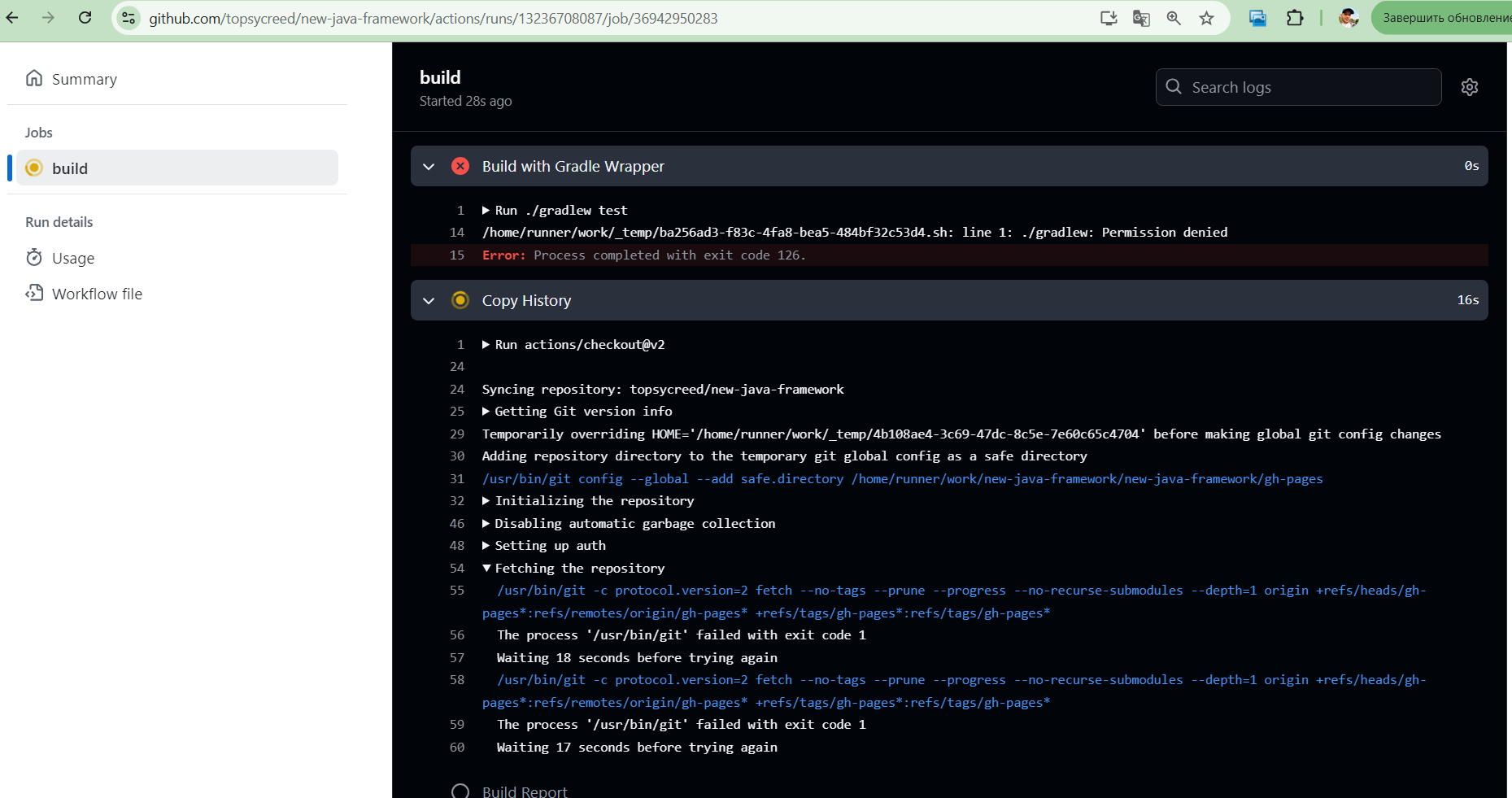
PUBLISH\_BRANCH: gh-pages

PUBLISH\_DIR: allure-history

1. Теперь можно запустить первый прогон нашего Action:



1. По мере прогона в случае чего можно посмотреть логи:



Если у вас, например, есть проблемы с permission denied, то разрешения можно также задать в настройках проекта.

Например, я добавил шаг:

- name: Grant execute permission for Gradle

run: chmod +x gradlew

1. Что еще может пойти не так? Все зависит от того что вы запускаете. Если у вас есть UI тесты с Selenium, то у вас скорее всего будет ошибка: org.openqa.selenium.SessionNotCreatedException at ProtocolHandshake.java:114 - что ожидаемо, так мы не добавили в агента ни Selenium, ни Chrome браузер. Давайте исправим это через добавление Selenium Remote через докер:

services:

selenium:

image: selenium/standalone-chrome:latest

options: --shm-size 2g

ports:

- 4444:4444

Кроме этого нужно добавить также переменную для remote url:

- name: Build with Gradle Wrapper

env:

SELENIUM\_REMOTE\_URL: http://localhost:4444/wd/hub

run: ./gradlew clean test

Со стороны yml файла все готово, но нужно еще настроить запуск WebDriver во фреймворке в headless режиме, так как нам не нужен графический интерфейс браузера в CI/CD. Ниже пример кода для того, чтобы это сделать:

@BeforeEach

void setup() {

*driver* = initDriver();

*driver*.manage().timeouts().implicitlyWait(Duration.*ofSeconds*(2));

longWait = new WebDriverWait(*driver*, Duration.*ofSeconds*(10));

}

private WebDriver initDriver() {

String remoteUrl = System.*getenv*("SELENIUM\_REMOTE\_URL");

if (remoteUrl != null) {

ChromeOptions options = new ChromeOptions();

options.addArguments("--headless"); *// Add headless mode*

options.addArguments("--disable-gpu"); *// Switch off GPU, because we don't need it in headless mode*

options.addArguments("--no-sandbox"); *// Switch off sandbox to prevent access rights issues*

options.addArguments("--disable-dev-shm-usage"); *// Use /tmp instead of /dev/shm*

options.setCapability("goog:loggingPrefs", Map.*of*("browser", "ALL"));

try {

*driver* = new RemoteWebDriver(new URL(remoteUrl), options);

} catch (MalformedURLException e) {

throw new RuntimeException("Malformed URL for Selenium Remote WebDriver", e);

}

} else {

*driver* = new ChromeDriver();

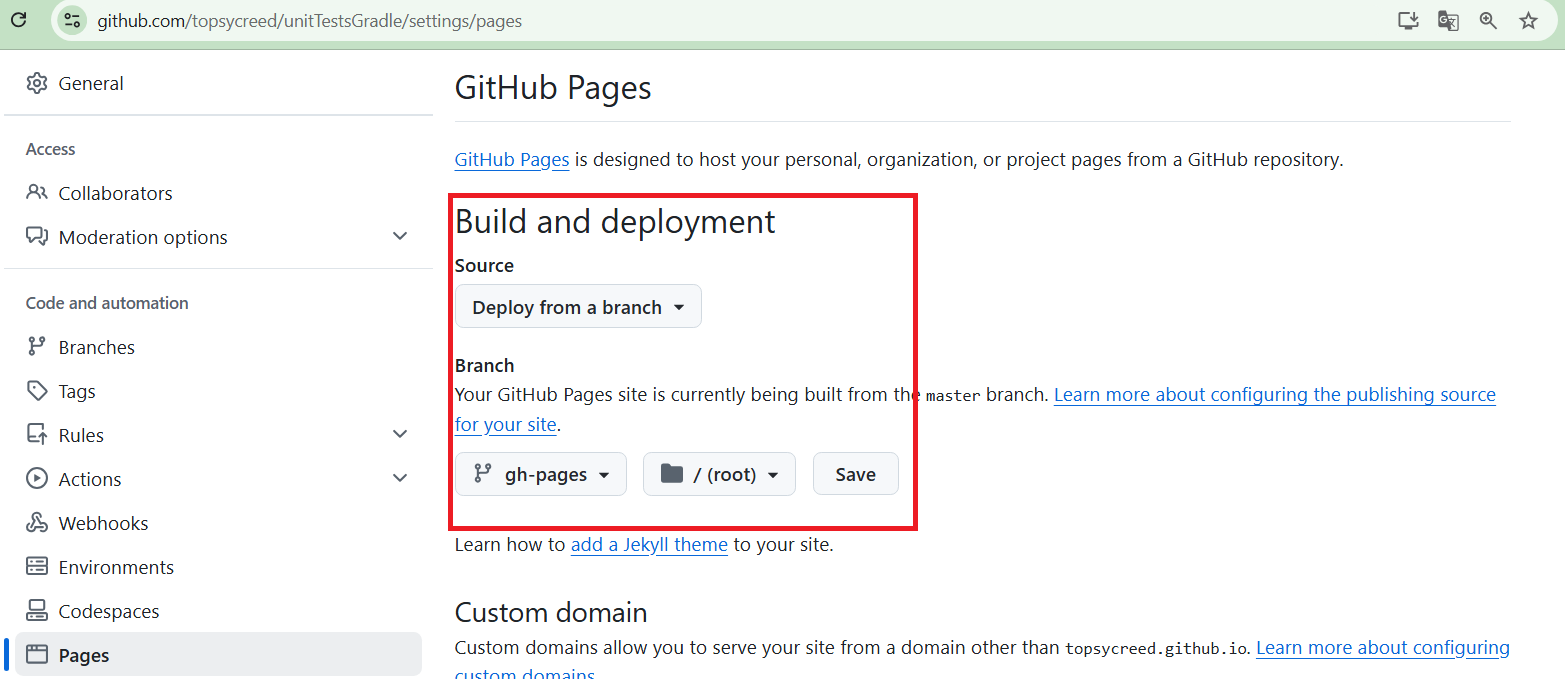
}

*driver*.manage().window().maximize();

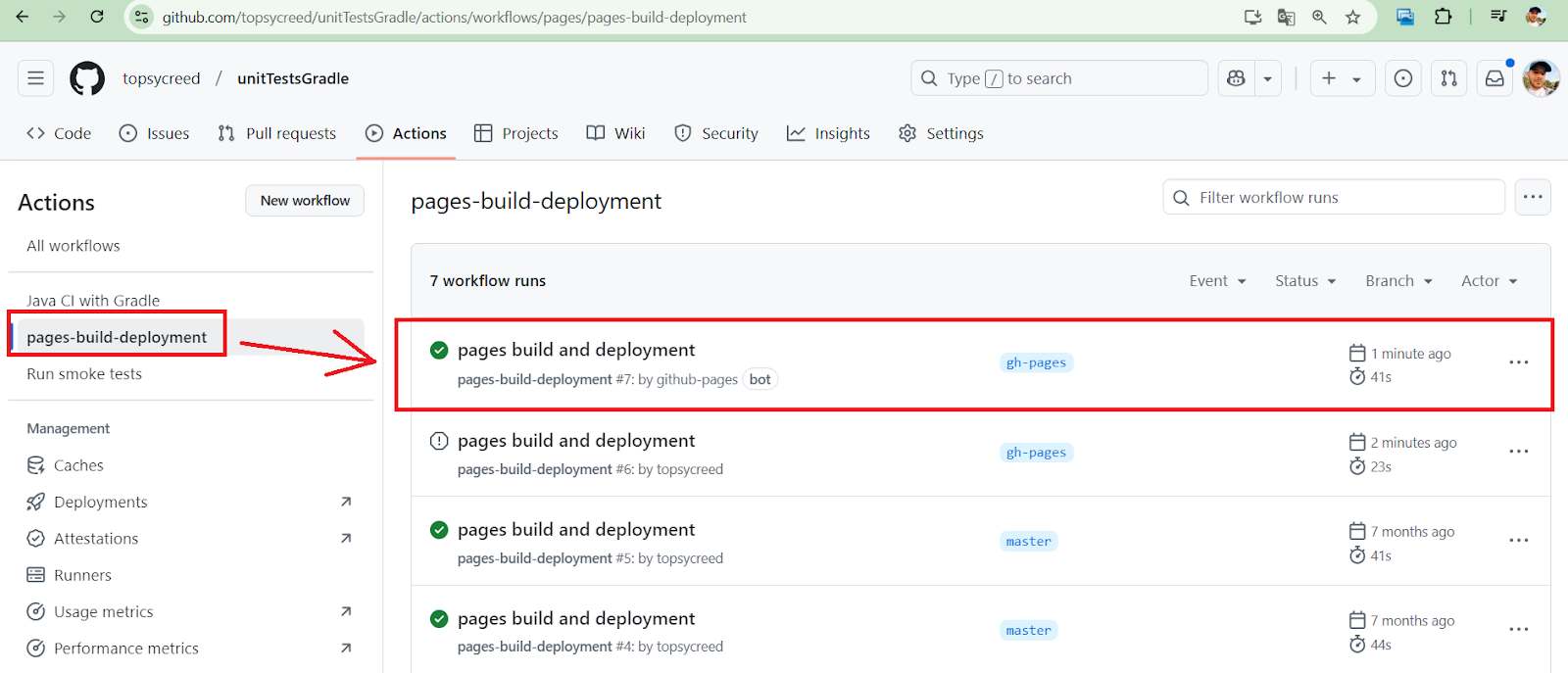
return *driver*;

}

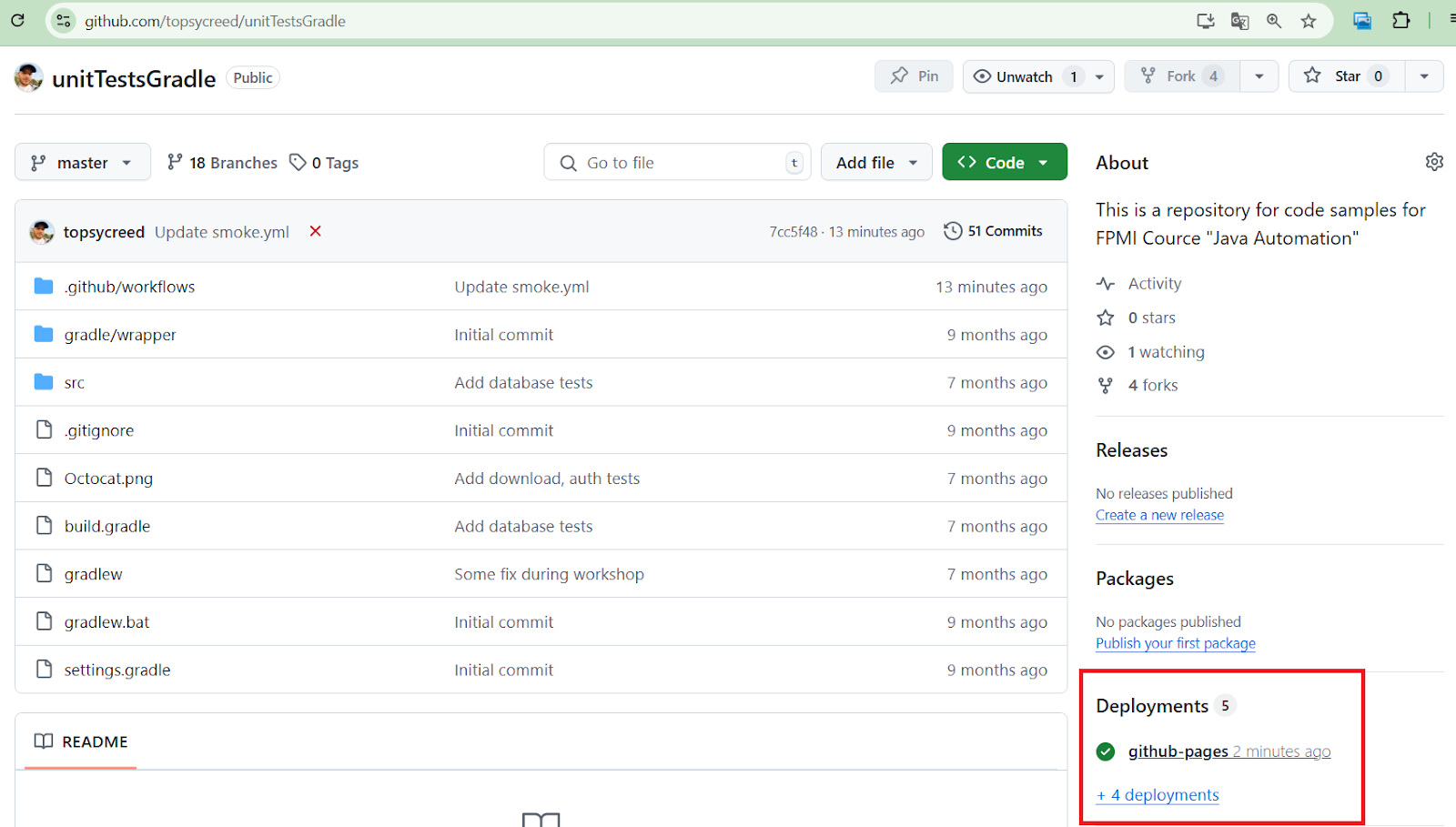
1. После первого прогона тестов (не обязательно даже успешного) нужно настроить публикацию в GitHub Pages в настройках самого проекта, Settings - Pages.



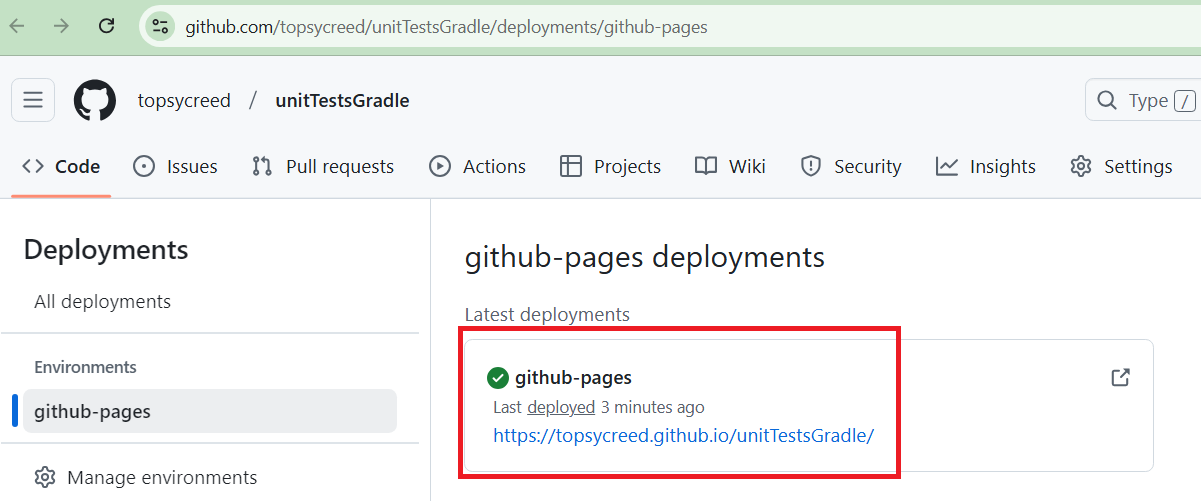
Теперь у вас должна появится джоба, которая деплоит саму страницу в GitHub Pages



На главной странице проекта в Deployments должна отобразиться последняя версия, внутри которой будет ссылка на Allure отчет.



Нажимаем на ссылку и смотрим наш Allure report.



1. Что дальше? Например, в зависимости от вашего проекта настройте правила, чтобы Smoke автотесты обязательно запускались на каждый запуск, либо запускайте расширенные автотесты вручную, добавьте нотификации (почта, slack, telegram) и т.д.

Итоговый файл <https://github.com/topsycreed/new-java-framework/blob/master/.github/workflows/gradle.yml>

Код ветки после добавления проперти файлов: <https://github.com/topsycreed/new-java-framework/tree/11-github-actions> и его Pull request: <https://github.com/topsycreed/new-java-framework/pull/11>, <https://github.com/topsycreed/new-java-framework/pull/12> и <https://github.com/topsycreed/new-java-framework/pull/13>

# Полезные ссылки:

1. <https://github.com/topsycreed/superhero> пример проекта на Maven, TestNG
2. <https://www.youtube.com/watch?v=7Z0CjtSbrXE> урок по автоматизации первого API, UI автотеста

# Вопросы/ответы

| **Вопрос** | **Ответ** |
| --- | --- |
| Не работает gradle с новой версией Java | Версия Gradle должна соответствовать версии JDK, смотри таблицу <https://docs.gradle.org/current/userguide/compatibility.html> |
| Нет вкладки Git | Либо она у вас называется VCS, либо вам нужно в настройках добавить свой GitHub аккаунт |
|  |  |