ˇИнструктаж

***\*В тексте для примера указаны ветки, папки, ссылки и прочее для элементов фреймворка Vuetify (выделены жирно с подчеркиванием). Уточни у тимлида с чем вы работаете сейчас!***

Table of Contents

[Работа над задачей 1](#_Toc1099095177)

[1. Начало работы 2](#_Toc1144523181)

[2. Выбираем задачу 3](#_Toc1014319405)

[3. Выполняем задачу 3](#_Toc916708417)

[4. Проверка 3](#_Toc287405253)

[5. Добавление Javadoc 3](#_Toc1839781763)

[6. Создание ПР (пулл реквеста) 3](#_Toc1049928856)

[7. Добавляем документацию 3](#_Toc1399097593)

[8. Завершаем 3](#_Toc1489298099)

[Best Practices 3](#_Toc1356380467)

[Оформление 3](#_Toc456504128)

[Элемент 3](#_Toc1243362572)

[Asserts 3](#_Toc492616950)

[@JDIAction 3](#_Toc1562587657)

[Тесты 3](#_Toc1950271922)

[Страница 3](#_Toc1533776408)

[DataProvider 3](#_Toc1619336696)

[Создание задачи (issue) 3](#_Toc776319494)

[FAQ 3](#_Toc1240077916)

[Полезные ссылки: 3](#_Toc1249175890)

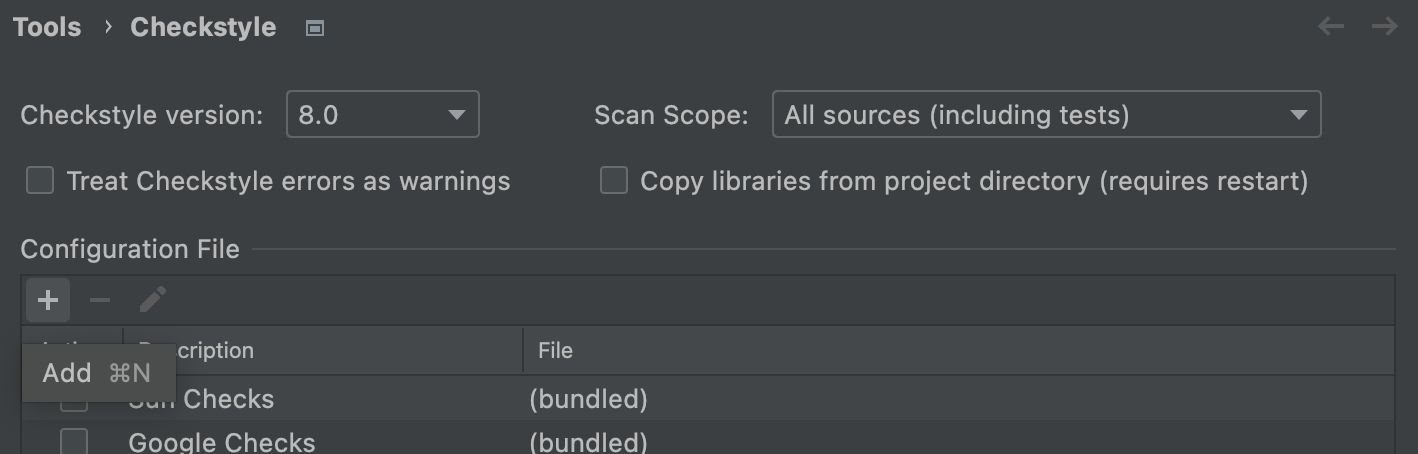
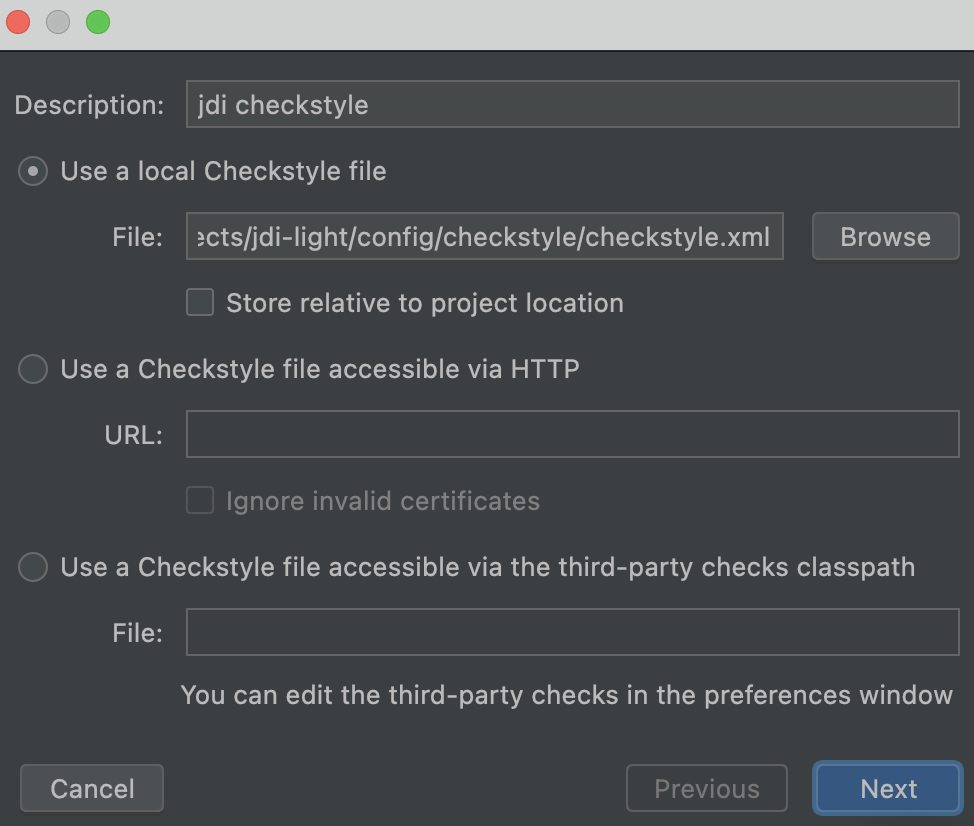
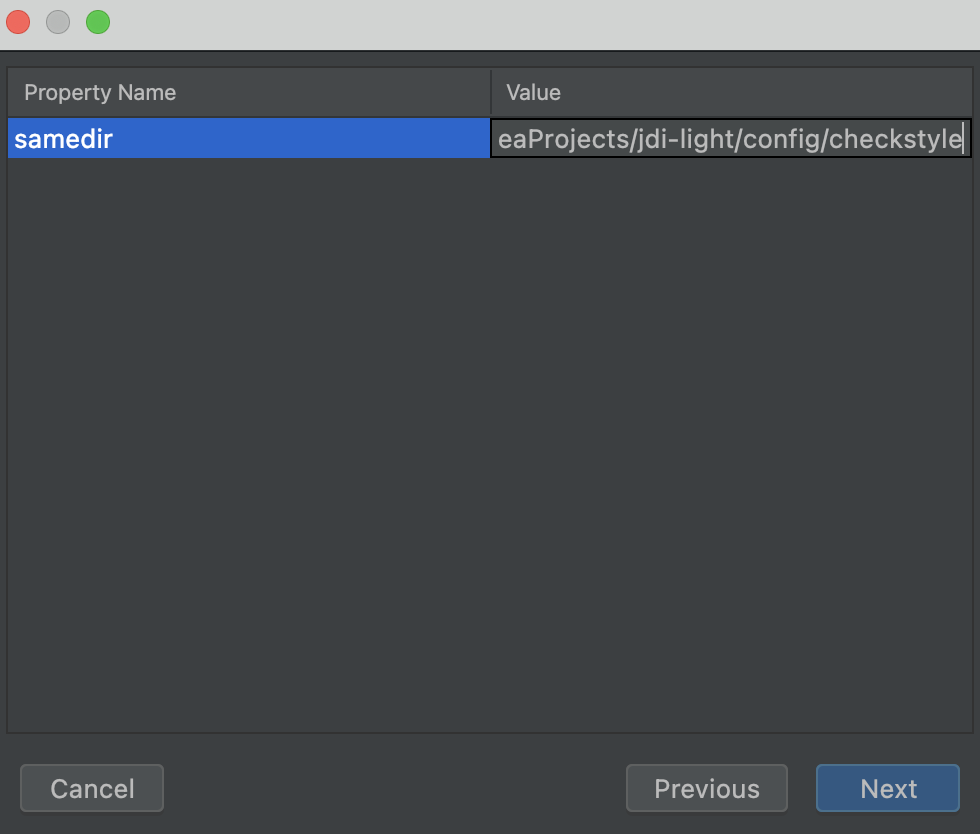
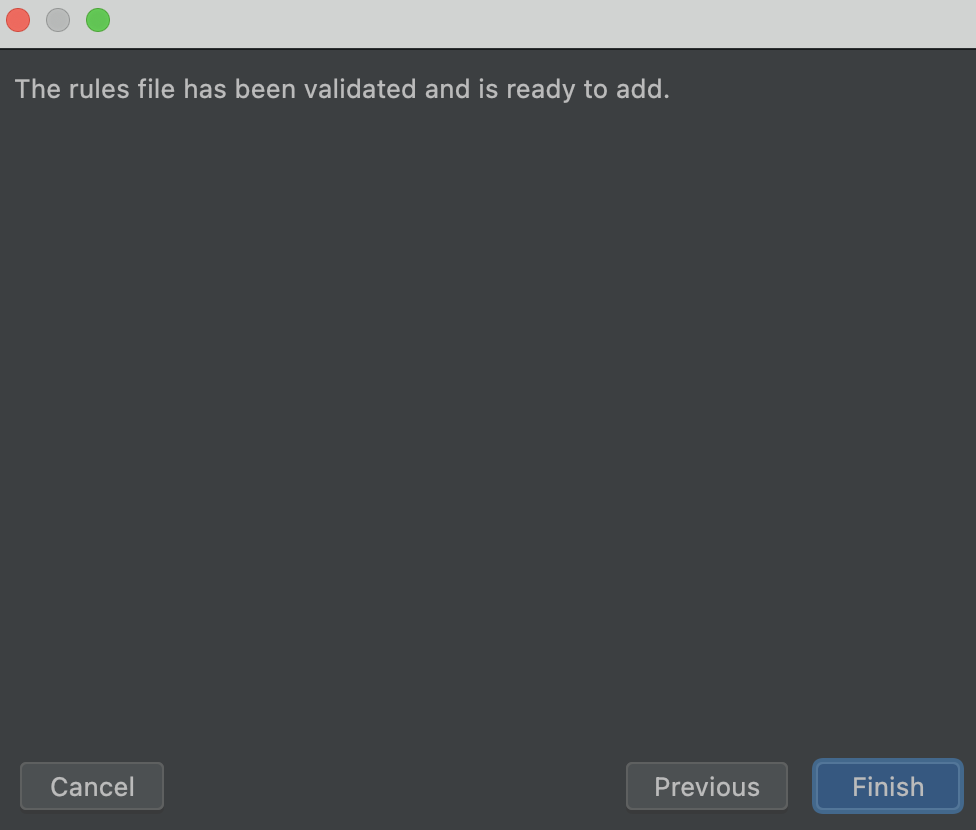
# Работа над задачей

## Начало работы

* 1. Клонируем репозиторий с гитхаба (https://github.com/jdi-testing/jdi-light)
  2. Открываем в IDEA как Maven проект
  3. Открываем вкладку Maven и нажимаем кнопочку “Toggle ‘Skip Tests’ mode”, (черная молния в голубом кружочке), чтобы при билде пропустить прогон тестов
  4. В настройках идеи запрещаем создание импортов со «звездочкой»:

A screenshot of a computer

Description automatically generated

* 1. Настраиваем чекстайл.   
     В корне проекта есть config/checkstyle – checkstyle конфигурация.   
     Открываем в настройках IntelliJ IDEA Tools/Checkstyle и устанавливаем версию чекстайла 8.0 и добавляем конфигурационный файл:  
       
       
     Дальше выбираем конфигурацию чекстайла из config/checkstyle:  
       
       
     Указываем путь до директории с конфигурациями:  
       
       
     Готово.   
     

## Выбираем задачу

* 1. Из раздела “ToDo” на доске проекта (<https://github.com/orgs/jdi-testing/projects/1>) выбираем задачу с лейблом **Vuetify**
  2. Ассайним задачу на себя (в карточке задачи указываем себя в качестве assignee)
  3. Перетаскиваем задачу в раздел «In progress»
  4. Находим и изучаем свой элемент на странице тестируемого сайта <https://jdi-testing.github.io/jdi-light> (также можно найти документацию фреймворка)

## Выполняем задачу

* 1. Создаем новую ветку из текущей рабочей ветки **vuetify-develop**:
     1. Обновить локально текущую рабочую ветку, подтянуть изменения
     2. Назвать её по такому шаблону:   
        Task<номер\_таски>\_<короткое\_описание\_таски>  
        Например: Task1935\_implement\_checkbox
     3. Перейти на созданную ветку
  2. Запускаем mvn clean install –DskipTests для модулей: JDI Light, JDI Light Html Elements, **JDI Light Vuetify**
  3. Изучаем проект и проверяем (тщательно!!!) существует ли в директориях jdi-light-html и jdi-light элемент, похожий на наш (с необходимыми нам основными методами для нашего элемента) и:
     1. *если можем его использовать в тестах:*
        1. Создаем с этими элементами Page Object своей тестируемой страницы (ищи свою страницу тут <https://jdi-testing.github.io/jdi-light>) в папке **src/main/java/io/github/com/pages**
        2. В папке **src/test/java/io/github/epam/vuetify/tests** создаем класс с тестами и тестируем элементы страницы, тем самым тестируя и методы найденного элемента (работают ли его методы и методы родительских классов для тестируемого элемента и если нет, то нужно писать свой элемент (см. п. 3.3.2)
     2. *если не нашли или его функциональности не хватает:*
        1. Создаем класс своего элемента в папке **src/main/java/com/epam/jdi/light/vuetify/elements** по следующим правилам:
* Если в директориях jdi-light-html и jdi-light есть элемент, похожий на наш (с необходимыми нам методами для нашего элемента):
  + наследуемся от него
  + переопределяем неработающие для нашего элемента методы
  + при необходимости, добавляем новые
* Если уже имеющиеся в jdi-light-html и jdi-light элементы нам никак не подходят, пишем свой кастомный, наследуясь от:
  + UIListBase - когда элемент не может существовать в одиночку, а может только с другими такими же элементами (пример – радио баттонс), с которыми он связан како либо логикой
  + UIBaseElement - одиночный элемент
* Не имплементируем элементы вида секции (неопределенного количества и состава элементов)  Следует уточнить у тимлида, как поступить с элементом, некоторые элементы вида секции таки имплементируются.
* Имплементируем значимые для пользователя элементы
* Имплементируем элементы по возможности просто
  + - 1. В класс элемента добавляем ссылку на страницу с описанием этого элемента на сайте фреймворка **Vuetify** (пример: *«To see examples of Badge web element please visit* [*https://vuetifyjs.com/en/components/badges/»*](https://vuetifyjs.com/en/components/badges/)
      2. Все методы ассерта, а также публичные методы элемента, относящиеся к “пользовательскому интерфейсу” элемента, должны быть помечены аннотацией [@JDIAction](#_@JDIAction_1) с осмысленным описанием.   
         При вызове метода пользователь увидит описание из аннотации, куда будет подставлено имя элемента вместо {name} и значение первого параметра метода вместо {0}.
      3. Проверьте интерфейсы, имеющиеся в jdi-light, jdi-light-html и в модуле фреймворка. Возможно, какие-то подходят вашему элементу и тогда хорошо бы их использовать
      4. Не ориентируемся на код своих коллег как на истину в последней инстанции
      5. Если это complex элемент и путь к элементам в нём может меняться – создавай кастомную аннотацию, в которой можно указывать локаторы элементов
      6. Посмотри [Best Practices](#_Элемент)
  1. Создаем со своими элементами Page Object своей тестируемой страницы (ищи свою страницу тут <https://jdi-testing.github.io/jdi-light>) в папке **src/main/java/io/github/com/pages**  
     [Рекомендации](#_Страница)
  2. В папке **src/test/java/io/github/epam/vuetify/tests** создаем класс с тестами и тестируем элементы страницы, тем самым тестируя и методы своего элемента (работают ли его методы и методы родительских классов для тестируемого элемента).   
     Требования к тестам:
     + - Базовая валидация
       - Поведение элемента, один тест на бизнес экшен
       - Структура тестов - наиболее важные тесты должны быть первыми
       - В тестах не используем jdiAssert
       - Не используем mocks.  
         [Рекомендации по тестам](#_Тесты)
  3. Если стандартных/родительских ассертов не хватает, то в папке **src/main/java/com/epam/jdi/light/vuetify/asserts** создаем кастомные. В этом случае в классе элемента нужно будет переопределить метод is(), пример:

@Override  
 public IconAssert is*() {*  
return new IconAssert*()*.set*(*this*)*;  
 *}*

## Проверка

* 1. Проверка соответствия кода требованиям Java Code Conventions: ctrl+A, ctrl+alt+L для выравнивания
  2. Проверка прохождения тестов
     1. Запускаем свои тесты с помощью файла src/test/resources/general.xml (правой кнопкой мыши по файлу и Run), чтобы проверить пройдут ли они на всех версиях JDK (начиная с восьмой)

## Добавление Javadoc

[Javadoc Guide](https://epam-my.sharepoint.com/:w:/p/alexey_girin/EdsHdkXf7a5OgWoBV3LM9wYBqCFpqHB3yAHnvqB22y6zYA?e=P5uaJd)

Наличие Javadoc является обязательным для всех сущностей, кроме тестов и сопутствующих им вспомогательных ресурсов.

* Сущности - всё, для чего может быть написан Javadoc: класссы, интерфейсы, методы, поля и константы и т.д.).
* Вспомогательные ресурсы - DataProvider’ы, Page-классы и т.д. (исключение - кастомные элементы, перечисления и прочие “нересурсные” сущности, для них также нужны Javadoc)

Это позволяет всем участникам проекта быстро и в удобной форме получать справочную информацию для разработки прямо в процессе кодирования, экономя время и помогая избежать ошибок. В этой связи тем более важно верно указывать информацию в Javadoc и грамотно их оформлять.

Основные правила следующие:

* Все публичные и защищенные сущности должны иметь полноценно оформленный Javadoc (приватные сущности - опционально).
* Полнота, корректность и ясность информации - ключевые моменты, поэтому лучше описывать сущность максимально подробно, если нужно, то приводить примеры использования в коде и т.п. Не следует бояться длинных Javadoc, особенно если есть неочевидные моменты, касающиеся использования описываемой сущности, таковых много и в самой Java (см. интерфейс List), но следует избегать ненужной детализации, если что-то и так очевидно.
* **Javadoc для класса/интерфейса** должен как минимум описывать, чем этот класс/интерфейс является/что эмулирует.
* **Javadoc для метода** должен как минимум описывать, что метод делает, все параметры, возвращаемое значение и возможные исключения (при наличии). Для возвращаемого значения указывается его тип. Проверяемые исключения обязательны к документированию, непроверяемые - опционально.
* **Javadoc для переопределенных методов** опционален и имеет смысл, если есть неочевидные моменты в его использовании, по умолчанию происходит ссылка на родительский метод.
* **Javadoc для методов аннотаций** должен описывать его как поле класса/константу (см. далее), приводить значение по умолчанию, если таковое имеется, и возвращаемое значение.
* **Javadoc для поля/константы** (кроме констант перечислений, для которых он не нужен) должен как минимум описывать, для чего нужно поле/что оно хранит.
* Javadoc должен быть грамотно и по правилам отформатирован с использованием тэгов Javadoc и HTML, если последние требуются.
* Все ссылки на сущности должны быть оформлены через тэг @link.
* Вся дополнительная справочная внешняя информация или рекомендации, куда посмотреть, должны быть оформлены через тэг @see.
* Deprecated-методы должны содержать тэг @deprecated.
* Примитивные типы и вставки кода должны оформляться через тэг @code.
* Тэги @author, @version, @since пока не надо использовать.  
    
  Список тэгов Javadoc:

|  |  |  |
| --- | --- | --- |
| **Тег** | **Описание** | **Синтаксис** |
| @author | Добавляет автора класса. | @author John Doe |
| {@code} | Отображает текст шрифтом кода без интерпретации текста как разметки HTML или вложенных тегов javadoc. | {@code myProp.setValue(true)} |
| {@docRoot} | Представляет относительный путь к корневому каталогу сгенерированного документа от любой сгенерированной страницы. | {@docRoot} |
| @deprecated | Добавляет комментарий, указывающий, что этот прикладной программный интерфейс больше не следует использовать. | @deprecated This method is deprecade, please use otherMethod() instead |
| @exception | Добавляет подзаголовок **Throws** в сгенерированную документацию с именем класса и текстом описания | @exception RuntimeException |
| {@inheritDoc} | Наследует комментарий от **ближайшего** наследуемого класса или реализуемого интерфейса | Inherits a comment from the immediate surperclass. |
| {@link} | Вставляет встроенную ссылку с видимой текстовой меткой, которая указывает на документацию для указанного пакета, класса или имени члена указанного класса. | {@link Java.util.List#get} |
| {@linkplain} | Идентично {@link}, за исключением того, что подпись ссылки отображается в виде обычного текста, а не шрифта кода. | {@linkplain Java.util.List#get} |
| @param | Добавляет параметр с указанным именем параметра, за которым следует указанное описание, в раздел «Параметры». | @param index – element index in this list |
| @return | Добавляет раздел **Return** с текстом описания. | @return the sum of given numbers |
| @see | Добавляет заголовок **See also** со ссылкой или текстовой записью, указывающей на ссылку. | @see java.lang.String |
| @serial | Используется в комментарии к документу для сериализуемого поля по умолчанию. | @serial field-description | include | exclude |
| @serialData | Документирует данные, записанные методами writeObject() или writeExternal(). | @serialData data-description |
| @serialField | Документирует компонент ObjectStreamField. | @serialField field-name field-type field-description |
| @since | Добавляет заголовок **Since** с указанным текстом в созданную документацию. | @since 1.2 |
| @throws | Теги @throws и @exception являются синонимами. | @throws RuntimeException |
| {@value} | Когда {@value} используется в комментарии к документу статического поля, он подставляет значение этой константы. | {@value root.MyClass#field1} |
| @version | Добавляет подзаголовок **Version** с указанным текстом версии в сгенерированные документы при использовании параметра -version. | @version 1.0 |

При написании следует придерживаться следующего порядка тегов Javadoc:

* @author (классы и интерфейсы)
* @version (классы и интерфейсы)
* @param (конструкторы и методы)
* @return (методы)
* @exception (или @throws)
* @see
* @since
* @serial (или @serialField, или @serialData)
* @deprecated

Для просмотра окончательного вида Javadoc можно просто навести курсор на имя сущности или нажать на левом поле в IDEA кнопку “Toggle rendered view” (появится при наведении курсора на строку с Javadoc-комментарием). Хорошие примеры Javadoc в большом количестве есть в исходных классах самой Java. Также можно обратиться к [официальному руководству от Oracle](https://www.oracle.com/technical-resources/articles/java/javadoc-tool.html).

Пример Javadoc из JDI:

*/\*\**  
 *\* Represents list MUI component on GUI.*  
 *\**  
 *\* @see <a href="https://mui.com/components/lists/">List MUI documentation</a>*  
 *\* @see <a href="https://jdi-testing.github.io/jdi-light/material">MUI test page</a>*  
 *\*/*  
*public class* MUIList extends UIBaseElement<MUIListAssert> {  
  
 */\*\**  
 *\* Locator for complex list item which can contain some other interactable elements like buttons, checkboxes etc.*  
 *\*/*  
protected static final String *LIST\_ITEM\_CONTAINER\_LOCATOR* = ".MuiListItem-container";  
  
 */\*\**  
 *\* Constructs new MUIList object using given element.*  
 *\**  
 *\* @param element element on web page to be treated as list*  
 *\*/*  
public MUIList(UIElement element) {  
 core().setCore(element);  
 }  
  
 */\*\**  
 *\* Gets specific item of this list using its text (full equality is used by*   *searching).*  
 *\**  
 *\* @param itemText full text content of item to be found*  
 *\* @return item of this list as {@link MUIListItem}*  
 *\* @throws RuntimeException if no items found*  
 *\*/*  
@JDIAction("Get the first item in '{name}' with text '{0}'")  
 public MUIListItem item(String itemText) {  
 for (MUIListItem item : items()) {  
 if (item.getText().equals(itemText)) {  
 return item;  
 }  
 }  
 throw *runtimeException*(*format*("The list has no item with text '%s'", itemText));  
 }  
  
 */\*\**  
 *\* Gets nested lists of this list if any. Targets only first layer of depth. If there are no nested lists, an empty list is returned.*  
 *\**  
 *\* @return nested lists of this list as {@link List}*  
 *\*/*  
@JDIAction("Get list of Material UI lists nested directly within '{name}'")  
 public List<MUIList> nestedLists() {  
 return finds(".//ul[not(parent::ul)]").stream()   
.map(nestedList -> new MUIList().setCore(MUIList.class, nestedList))  
 .collect(Collectors.*toList*());  
 }

}

## Создание ПР (пулл реквеста)

* 1. Добавляем, коммитим и пушим изменения в СОЗДАННУЮ ТОБОЙ ВЕТКУ.
  2. Создаём ПР на созданные изменения:
     1. В качестве целевой ветки указываем ТЕКУЩУЮ РАБОЧУЮ ВЕТКУ **vuetify-develop**
     2. Название пул реквеста:  
        Task <task number>: <prefix> <short description>  
        Prefix (описание действия): Fix, Implement, Refactor, Rework, Verify.  
        Пример: Task 3895: Fix Drawer tests
     3. В описании указывай, что ты сделал, какие проблемы решил, или просто информацию, которую важно знать коллегам/ревьюверу
     4. В качестве связанного проекта указываем ‘JDI Light Board’
     5. Указываем себя в качестве ассайни
     6. Ставим лейблы (если есть) – такие же, как у твоей задачи
     7. В linked issues указываем задачу, которую ты выполнял
     8. Проверить прогон тестов в Actions проекта или Checks в пулл реквесте
     9. Когда работа над ПР закончена, указываем в качестве ревьюера LightReviewTeam или тимлида **(уточни кого указывать)**

## Добавляем документацию

* 1. Клонируем проект с документацией <https://github.com/jdi-docs/jdi-light>
  2. Создаем свою ветку из ветки мастер
  3. В ней вносим изменения (смотри примеры коллег) в файл **source/includes/documentation/\_vue\_js\_elements.md**
  4. В разделе обязательно должны быть:
* Название элемента
* Ссылка на страницу с ним с сайта фреймворка **Vuetify**
* Пример теста
* Описание элемента (можно с сайта фреймворка **Vuetify**)
* Изображение элемента
* Методы элемента
* Ссылка на файл с твоими тестами элемента в репозитории
* Картинка с html-code должна начинаться на том же уровне, что и java-code
* Вложенность нумерации пунктов описания элементов контролируется символом #, на каждом уровне углубленности добавляется доп. знак)
* Не нужно приводить в примере тестов все тесты, только самые значимые
* Соблюдайте очередность пунктов в соответствии с тем, который представлен на сайте фреймворка **Vuetify**
  1. Изображения, которые хотим использовать, подгружаем сюда - **source/images/vuetify** и ссылкой добавляем в редактируемую часть документации (примеры, опять же, у коллег)
  2. Пушим изменения в свою ветку
  3. Создаем ПР на изменения, мерджим их и проверяем правильность отображения на сайте <https://jdi-docs.github.io/jdi-light>. Если что-то пошло не так, редактируем заново.

## Завершаем

* 1. После того, как работа над элементом в пулл реквесте закончена, перемести свою таску в раздел ‘Waiting for Demo’
  2. После того, как аппрувер смерджил твой пулл реквест, ветка должна быть удалена

# Best Practices

## Оформление

* Пустые строки ([White Space](http://www.cs.toronto.edu/km/istar/OME3/codeconv/CodeConventions.doc7.html))
  + Одна пустая строка может использоваться в следующих случаях:
    - Разделения логических блоков внутри метода
    - Между методами
    - Между локальными переменными в методе и первой командой внутри метода
  + Две пустые строки могут использоваться для:
    - Между импортами и описанием класса
    - Between sections of a source file
    - Between class and interface definitions
  + Не должно быть
    - Двух пустых строк подряд в коде (кроме описанного выше)
    - Пустой строки перед закрывающейся скобкой

## Элемент

* Что нельзя делать
  + В элементах не может быть айдишников или **специфичных** для тестового случая данных (т.е. локаторов, которые могут меняться), за исключением классов, гарантируемых фреймворком (см. документацию фреймворка **Vuetify**
  + В библиотеке не может быть **частных** случаев с нашего тестового сайта (в классе элемента мы описываем только его общие свойства и поведения, в независимости от конкретной реализации этого элемента)
* Не надо создавать
  + Методы-обёртки – если они не используются часто  
    public String messageText() {return message().getText(); }
  + Методы получения css свойств, так как их можно получить по классу
* Используйте интерфейсы из jdi-light и jdi-light-html модулей
* Методы
  + Именование
    - В названии метода не должно быть префикса ‘get’
    - Если это Boolean метод – **is**Enabl**ed**, **has**…
  + Если метод должен что-то вернуть, но не может, то он должен вернуть null, не exception
  + Использовать core() перед методами:
    - hasClass
    - hasAttribute
    - attr
    - getTagName
    - find/finds
  + Заменять методы jdi более короткими:
    - getHref -> href
    - getColor -> color
    - attr(“class”).contains -> hasClass(…)

## Asserts

* Не используй в ассертах Matchers.is(true/false) – в таком случае используется тернарный оператор
* Перегрузка методов с параметром Matcher для методов:
  + Text
  + Value

## @JDIAction

* Использование параметров:
  + ‘{name}’ - имя элемента, у которого вызывается метод
  + {0}, {1}, … - параметры метода
    - Если тип параметра - Matcher, то не используем кавычки и не используем is/has:   
      @JDIAction(“Check that '{name}' **text {0}**")
* Элемент
  + Get-method -> @JDIAction(“**Get** …")
  + Set-method -> @JDIAction("**Set** text '{name}' text area **as ‘**{0}’")
  + Boolean method -> @JDIAction("**Check that** '{name}' text area **is** empty")
* Ассерты
  + @JDIAction("**Assert that** '{name}' text **is** {0}")

## Тесты

* Что проверяем
  + Проверяем только необходимый функционал
  + Если функциональность определяется специальным классом (например, во фреймворке) – достаточно проверить только класс
  + Не проверяем то, где нет функциональности (конкретные случаи можно обсуждать): outlined/filled/dense
  + Для базовой функциональности создаём отдельный метод (например, basicFunctionalityTest). Там можно проверить стили
* Используй show() - в начале каждого теста показывай элемент, с которым будет происходить работа, чтобы его было видно на скриншоте при ошибке
* Если работаешь с элементами листа, используй локальную переменную для листа или элемента, с которым работаешь

## Страница

* Описываем все элементы на странице для примера пользователям
* Лучше создавать локаторы, которые находят именно то, что нужно:
  + Если нужен один элемент, не используйте локаторы, по которым находятся несколько элементов
* [Про локаторы из документации](https://jdi-docs.github.io/jdi-light/#jdi-locators-as-simple-as-css-as-powerful-as-xpath)
* Элементы в классе страницы должны быть в том же порядке, в каком они расположены на странице

## DataProvider

* Создаём как отдельный класс, не среди тестов
* Не создаём дата провайдер, если только один набор данных, - можно данные использовать сразу в тесте

# Создание задачи (issue)

1. Название: <фреймворк>: <краткое описание>  
   Пример: “Material UI: Refactor Stepper”
   1. Фреймворк: Materail UI, Vuetify
2. Описание
   1. Для подзадач можно использовать такое оформление: “- [ ] subtask”
3. Лейблы
   1. Material-UI, Vuetify - фреймворк
   2. Refactoring - надо переделать элемент
   3. 🔍 investigate - проблема мало изучена, надо разобраться (падающие тесты, другие способы решения задач)
   4. ♻ changes requested - проблема не полностью починена или заново появилась
4. Проект
5. JDI Light Board

После создания задачи, отправь её Александру Цветаеву на проверку.

# FAQ

**Q:** В моей директории есть поддиректории с разбиением элементов на [common|complex|composite], как понять в какой папке должен быть мой класс?

**A:** Прочитай описание ниже. Также в качестве примера можешь использовать другие директории проекта и посмотреть к какой категории находится такой же/подобный элемент.

*Common* (“простые”) элементы – в большинстве случаев в чистом Селениуме для их описания требуется один локатор или один WebElement. Характерным примером будет Checkbox: по сути, это один HTML-элемент input с атрибутом type=checkbox. Чтобы взаимодействовать с ним, в чистом Селениуме достаточно будет вызвать click() у соответствующего WebElement.

*Complex* (“непростые”) элементы – их можно условно разделить на две группы. Первая группа – это, по сути, наборы из нескольких common элементов: например, Checklist это просто набор из нескольких Checkbox. Для их описания в чистом Селениуме мы, очевидно, использовали бы List<WebElement>. Вторая группа – это элементы UI, совмещающие в себе функциональность нескольких common элементов. Например, в MultiDropdown есть кнопка, раскрывающая выпадающий список; чекбоксы в списке; и текстовое поле, показывающее результаты выбора. В чистом Селениуме такое не описать ни одним WebElement, ни одним List<WebElement>.

*Composite* (“составные”) элементы – а вот это уже как раз контейнеры, которые могут содержать как common, так и complex элементы, а также другие контейнеры, в качестве своих полей. Самый простой пример это Section, раздел (секция) страницы. Мы можем, например, описать шапку, подвал, или навигационное меню сайта как Section, полями которого будут являться экземпляры common, complex элементов, или других Section. Form вдобавок к этому позволяет использовать свои поля для заполнения формы данными объекта DataClass.

**Q:** Я не могу получить доступ ко всем страницам тестируемого сайта?

**A:** Залогинься (credentials: Roman, пароль Jdi1234)

**Q:** Я не могу подобрать стабильный локатор к своему элементу на тестовой странице, так как у него нет айдишника, что делать?

**A**: Если нужно, можно скорректировать саму тестовую страницу (как правило это делается для добавления id элементу). Это выполняется следующим образом:

* Клонируем проект [**тестируемого сайта**](https://github.com/jdi-testing/test-site/tree/vuetify_surfaces) (выбрать нужную ветку) и создаем свою ветку от ветки master.
* Меняем **vuetify-site/src/app/<element group>/<element>/<element>.html** (или .ts или .css файлы если нужно). При добавлении id проверяем, чтобы он не повторял id другого элемента, а также чтобы это был не просто рандомный набор символов.
* Для проверки изменений локально воспользуйся [https://github.com/jdi-testing/test-site/wiki/Vuetify-setup-and-build-process](https://github.com/jdi-testing/test-site/wiki/Angular-setup-and-build-process) (это необязательно для простых изменений как добавление id)
* Создай пул реквест в мастер ветку. Если ты делаешь это впервые, добавь LightReviewTeam в качестве ревьюера. Если нет – просто вмерджи изменения в мастер ветку. После этого автоматически создастся пулл реквест в gh-pages и ты сможешь увидеть изменения на сайте. Если у тебя возникнут какие-то проблемы, свяжись со своим тим лидом.

(Возможно, при выполнении указанных манипуляций вам также чем-то поможет инструкция для такого случая в Material UI:   
[jdi-testing/test-site at material\_surfaces (github.com)](https://github.com/jdi-testing/test-site/tree/material_surfaces) )

# Полезные ссылки:

Если доступа нет, попроси тим лида предоставить его тебе.

1. [Доска с задачами](https://github.com/orgs/jdi-testing/projects/1)
2. [Проект](https://github.com/jdi-testing/jdi-light)
3. [Документация JDI light](https://jdi-docs.github.io/jdi-light/)
4. [Гайд от Никиты Нечепуренко](https://github.com/nechepurenkoN/jdi-light-half-blood-prince-diary) с более глубоким описанием проекта, его основных классов, используемого во фреймворке аспектно-ориентированного программирования и т.п.
5. [Code review guide](https://epam-my.sharepoint.com/:t:/r/personal/alexey_girin_epam_com/Documents/JDI/JDI%20Light/review_guide.md?csf=1&web=1&e=FlyiKa)
6. [Видео с принципами разработки модуля materials от Романа Иовлева](https://epam-my.sharepoint.com/:v:/r/personal/alexey_girin_epam_com/Documents/JDI/JDI%20Light/%5bJDI-%D0%9A%D0%9C%D0%91%5d%20%D0%9F%D1%80%D0%B8%D0%BD%D1%86%D0%B8%D0%BF%D1%8B%20%D1%80%D0%B0%D0%B7%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B8%20%D1%8D%D0%BB%D0%B5%D0%BC%D0%B5%D0%BD%D1%82%D0%BE%D0%B2%20Material%20UI%20%D0%BD%D0%B0%20%D0%BE%D1%81%D0%BD%D0%BE%D0%B2%D0%B5%20%D0%BE%D0%BF%D1%8B%D1%82%D0%B0%20Angular.mp4?csf=1&web=1&e=uuh1Nk)
7. [Видеокурс молодого бойца](https://epam-my.sharepoint.com/:v:/r/personal/alexey_girin_epam_com/Documents/JDI/JDI%20Light/%5bJDI-%D0%9A%D0%9C%D0%91%5d%20%D0%9F%D1%80%D0%B8%D0%BD%D1%86%D0%B8%D0%BF%D1%8B%20%D1%80%D0%B0%D0%B7%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B8%20%D1%8D%D0%BB%D0%B5%D0%BC%D0%B5%D0%BD%D1%82%D0%BE%D0%B2%20Material%20UI%20%D0%BD%D0%B0%20%D0%BE%D1%81%D0%BD%D0%BE%D0%B2%D0%B5%20%D0%BE%D0%BF%D1%8B%D1%82%D0%B0%20Angular.mp4?csf=1&web=1&e=uuh1Nk)
8. [Шаропапки](https://epam-my.sharepoint.com/personal/alexey_girin_epam_com/_layouts/15/onedrive.aspx?id=%2Fpersonal%2Falexey%5Fgirin%5Fepam%5Fcom%2FDocuments%2FJDI) (можно что-то полезное попробовать поискать)
9. [Javadoc Guide](https://epam-my.sharepoint.com/:w:/p/alexey_girin/EdsHdkXf7a5OgWoBV3LM9wYBqCFpqHB3yAHnvqB22y6zYA?e=P5uaJd)