ЛАНИТ
ЭКСПЕРТИЗА

NC AI Platform – Low-code платформа для автоматизации тестирования приложений с применением нейросетей

Руководство администратора

Москва

2023

Содержание

Список сокращений

Сокращение	Определение
NC AI Platform, ПО	NC AI Platform – Low-code платформа для автоматизации тестирования приложений с применением нейросетей
QA	специалист по обеспечению качества разработки программного обеспечения
TE	Task Executor
АРМ	автоматизированное рабочее место
АТ	автоматизированное тестирование; автотест
СТП	служба технической поддержки
ТК	тест кейс, тестовый сценарий
ФТ	функциональное тестирование; тестовый сценарий для выполнения функционального тестирования

1 ОБЩИЕ СВЕДЕНИЯ

Программный комплекс «NC AI Platform – Low-code платформа для автоматизации тестирования приложений с применением нейросетей» позволяет без программирования создавать и проводить автоматизированные тесты с использованием встроенного редактора. ПО позволяет получать сведения об используемых для тестирования элементах UI-интерфейса с помощью методов Computer Vision и нейросетей (реализована интеграция с решением NeuroControl, осуществляющим данную функциональность)

NC AI Platform позволяет пользователю разрабатывать сценарии, запускать их на своем оборудовании или на удаленных машинах, производить отладку тестовых сценариев в режиме реального времени, а также группировать сценарии в проекты для разграничения работы.

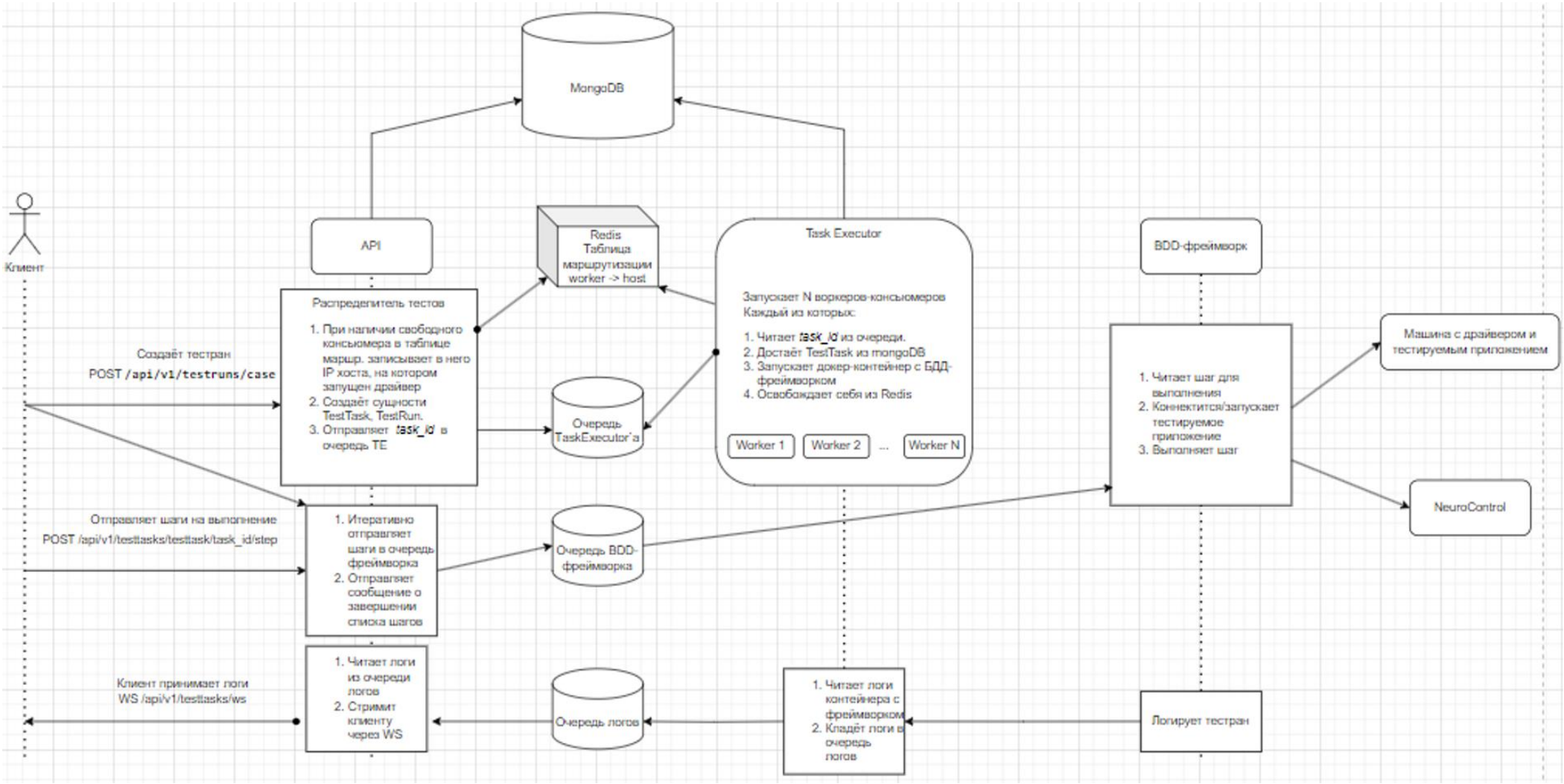
Запуски множества сценариев также могут быть сгруппированы в тест-планы.

После прохождения сценариев возможен обзор отчетов Allure и просмотр запуска на платформе NeuroMoon.

.

2 ОПИСАНИЕ ФУНКЦИОНАЛЬНОСТИ ПО

2.1 Инфраструктура ПО



2.2 Порядок работы фреймворка

Task Executor (TE) по соответствующему запросу **API** запускает Docker-контейнер с **BDD-фреймворком**, декларируя необходимый набор переменных окружения, в частности, наименование выполняемого тестового сценария, версии используемых моделей нейронных сетей, идентификатор очереди сообщений, и иные конфигурационные параметры.

API итеративно отправляет шаг(-и) в очередь сообщений **фреймворка**, и последним сообщением сигнализирует о завершении списка шагов. **Фреймворк** логирует ход своей работы.

Фреймворк завершает свою работу, формирует Allure-отчёт, сохраняет в Allure-сервисе, соответствующий контейнер останавливается на стороне **TE**, удаляется.

3 НАЧАЛО РАБОТЫ С ПО

Порядок развертывания ПО приведён в отдельном документе Инструкция по развертыванию (Приложение 1).

После запуска **фреймворка**, он выступает в качестве консьюмера, прослушивая очередь сообщение для получения тестовых шагов.

4 ОСНОВНЫЕ ПРИЁМЫ РАБОТЫ С ПО

4.1 Запуск тестируемого приложения

Запуск тестируемого приложения производится через **Flanium-драйвер**. Необходимые для использования **драйвера** параметры передаются во **фреймворк** через переменные окружения, а именно `DRIVER_URL`, `APP_PATH`, `LAUNCH_DELAY`, `CONNECT_TO_RUNNING_APP`.

4.2 Получение шага(-ов) на выполнение

Список тестовых шагов **фреймворк** получает от **API** через очередь **RabbitMQ**, идентификатор которой задан в переменной окружения **QUEUE_NAME**. Формат сообщения с шагом имеет следующий вид.

```
[
        {
                text: str
                row_id: int
                meta: Optional[dict]
                params: Optional[list[str]]
                configuration: Optional[dict]
        }
]
```

text - текст тестовый шага. ***row_id*** - номер строки шага. ***meta*** - []. НЕОБЯЗАТЕЛЬНОЕ поле. ***params*** - список параметров шага. НЕОБЯЗАТЕЛЬНОЕ поле. ***configuration*** - []. НЕОБЯЗАТЕЛЬНОЕ поле.

После завершающего шага отправляется такая же структура с одним из специальных флагов в поле **text**.

- `"TEST_END"` - флаг, сигнализирующий окончание тестового сценария.
- `"Cancel {QUEUE_NAME}"` - флаг, сигнализирующий отмену выполнения тестового сценария.

После получения любого из флагов выше **фреймворк** штатно завершает своё выполнение, прекращает прослушивать очередь сообщений **RabbitMQ**, удаляет её. Не гарантируется, что это последнее сообщение, однако обработка последующих не производится.

4.3 Выполнение шага

Фреймворк реализует выполнение тестовых шагов. В консоль выводится сообщение о начале выполнения шага в формате `Start step {step_context}`, где `step_context` - контекст шага, структура, полностью соответствующая полученной через очередь сообщений. Подробнее про вывод в консоль см. консольный вывод.

Пример консольного вывода начала выполнения шага:

```
Start step {'text': '[FL] проверить наличие элемента с текстом "Добро пожаловать в торговый терминал"', 'row_id': 0, 'meta': {}, 'params': [], 'configuration': {}}
```

4.4 После выполнения шага

После выполнения шага, независимо от результата выполнения, **фреймворк** подтверждает сообщение.

4.5 Логирование

Для связи **фреймворка** с **ТЕ** используются консольный вывод на стороне **фреймворка**.

4.6 Консольный вывод

Любой вывод в консоль переадресовывается на **UI** пользователю как информационное сообщение. Предусмотрена возможность маркировки логируемых в консоль строк посредством добавления специального префикса.

Список доступных префиксов для выводимых в консоль строк (без кавычек).

- `"::ERROR::"` - ошибка, например, выполнения шага, в т. ч. *NoSuchElementException*.
- `"::INTERNAL_ERROR::"` - внутренняя ошибка **фреймворка**.
- `"::TRACEBACK::"` - трассировка стека.

Примеры маркировки сообщений:

```
::TRACEBACK::Traceback (most recent call last):
::TRACEBACK::  File
"C:\\Users\\Srud\\PycharmProjects\\python_bdd_project\\[main.py](<http://main.py/>
)", line 16, in <module>
::TRACEBACK::    run_service()
::TRACEBACK::  File
"C:\\Users\\Srud\\PycharmProjects\\python_bdd_project\\[runner.py](<http://runner.
py/>)", line 31, in run_service
::TRACEBACK::    init_executor()
...
::TRACEBACK::  File
"C:\\Users\\Srud\\PycharmProjects\\python_bdd_project\\venv\\lib\\site-
packages\\urllib3\\util\\[retry.py](<http://retry.py/>)", line 592, in increment
::TRACEBACK::    raise MaxRetryError(_pool, url, error or ResponseError(cause))
::TRACEBACK::urllib3.exceptions.MaxRetryError:
HTTPConnectionPool(host='localhost', port=9999): Max retries exceeded with url:
/session (Caused by NewConnectionError('<urllib3.connection.HTTPConnection object
at 0x000001D564931430>: Failed to establish a new connection: [WinError 10061]
Подключение не установлено, т.к. конечный компьютер отверг запрос на
подключение'))
::TRACEBACK::
::INTERNAL_ERROR::HTTPConnectionPool(host='localhost', port=9999): Max retries
exceeded with url: /session (Caused by
NewConnectionError('<urllib3.connection.HTTPConnection object at
0x000001D564931430>: Failed to establish a new connection: [WinError 10061]
Подключение не установлено, т.к. конечный компьютер отверг запрос на
подключение'))
::ERROR::NoSuchElementException Message: [FlaNium ERROR]: Element cannot be found;
For documentation on this error, please visit:
<https://www.selenium.dev/documentation/webdriver/troubleshooting/errors#no-such-
element-exception>
```

4.7 Neuromoon

По окончанию выполнения сценария **фреймворк** формирует ссылку на **Neuromoon-прогон**. Ссылка формируется по следующему шаблону: `"{link}/runs/{run_id}/step/0/request/0"`, где `link` - адрес **Neuromoon** ((https://neuromoon.rnd.lanit.ru) извлечь его можно из переменной окружения `NEUROMOON_LINK` с помощью `NEUROMOON_LINK.rstrip('/')`); `run_id` - значение `TOKEN`.

4.8 Allure

По окончанию выполнения сценария **фреймворк** формирует **Allure-отчёт**. Адрес развёрнутого Allure-сервиса содержится в переменной окружения `ALLURE_SERVICE`, report_dir хранится в `ALLURE_RESULT_DIR`.

4.9 После окончания сценария

После окончания выполнения сценария фреймворк выводит в консоль ссылку на Neuromoon и Allure-отчёт отдельными строками. Последним сообщением фреймворк выводит результат выполнения сценария. После этого штатно завершает своё исполнение.

Пример корректной ссылки на **Allure-отчёт**. http://192.168.88.102:5050/allure-docker-service/projects/win10crm/reports/226/index.html

Пример корректно ссылки на **Neuromoon**. https://neuromoon.rnd.lanit.ru/runs/593/step/0/request/0

Ссылка на **Neuromoon** может отсутствовать, если её невозможно сформировать из-за инициации ошибки до обращения к нейросетевым сервисам. Ссылка на **Allure** может отсутствовать, если выполнение фреймворка прервано до окончания выполнения первого тестового шага. Если ссылки невозможно сформировать, они не выводятся в консоль.

4.10 Результат выполнения сценария

Последним сообщением **фреймворк** выводит результат выполнения сценария.

Если сценарий завершился успешно, **фреймворк** выводит строку с количеством последних полученных и выполненных шагов. Данная строка должна начинаться с `"Steps executed"`. Пример строки с результатом при успешном завершении тестового сценария: `"Steps executed: {len(steps)}."`.

Если сценарий завершился ошибкой (например, NoSuchElementException), **фреймворк** выводит соответствующее сообщение с префиксом `::ERROR::`, и штатно завершает свою работу. Пример такого сообщения: ****`"::ERROR::NoSuchElementException"`.

5 ОЧЕРЕДЬ СООБЩЕНИЙ

Для связи **ТЕ** с **фреймворком** использует **очередь сообщений RabbitMQ**. **API** создаёт очередь, которая затем используется **ТЕ** и **фреймворком** для передачи/получения шагов.

Фреймворк получает из переменной окружения `QUEUE_NAME` идентификатор очереди сообщений, которую слушает для получения тестовых шагов на выполнение. Формат именования очереди имеет следующий вид `task_{task_id}` (`task_id` - идентификатор задачи в коллекции **tasks**).

Фреймворк использует блочное соединение (blocking connection) со следующими параметрами: `username` - значение переменной окружения `PIKA_LOGIN`; `password` - значение переменной окружения `PIKA_PASS`; `host` - значение переменной окружения `PIKA_HOST`; `port` - значение переменной окружения `PIKA_PORT`.

Канал во фреймворке задаёт `prefetch_count` равным единице.

Очередь, используемая **фреймворком**, определяется следующими параметрами: `queue` - значение переменной окружения `QUEUE_NAME`; `arguments` - словарь, содержащий ключ `"x-max-priority"` со значением `10`; `passive` - `False`; `durable` - `False`; `exclusive` - `False`; `auto_delete` - `False`.

6 ЗАПУСК ТЕСТОВ

6.1 Сущности запуска тестов

Test Tasks - запуск и результат запуска одного сценария. Содержит ссылку на сценарий, содержит в себе результат запуска, в том числе и логи.
Test Run - набор сущностей Test Tasks. Даже при запуске одного сценария будет сформирован Test Run. Необходим для группировки нескольких запусков, просмотра результатов, повторного запуска. Создается, когда пользователь запускает тесты.
Test Plan - набор тестов, которые созданы пользователем, сохранены и могут быть запущены неоднократно. Определен в разрезе одного проекта и включает сценарии из этого проекта. По сути, являются аналогом проекта в сфере запуска, облегчают запускать сразу некий пользовательский набор сценариев.

6.2 Варианты запуска тестов

1. Запуск **единичного сценария**
    1. Запуск сценария из редактора в нормальном режиме
    2. Запуск сценария с извлечением шагов из БД в нормальном режиме
    3. Запуск сценария в debug-режиме (пошаговом) из редактора
2. Запуск **проекта**
3. Запуск **тест-плана**

6.3 Место выполнения тестов и источники конфигурации

Для запуска сценария/проекта в любых режимах и из любого места UI, информация о том, где запустить сценарии будет взята из информации о пользователе. Предполагается, что это будет локальная или доступная пользователю машина.

Для запуска тест-плана информация о том. где будут запущены сценарии будет браться из тест-плана. При создании тест-плана необходимо указать пул машин, на которых будут выполняться сценарии.

❗ Далее, маршрутизацией сценарии будут распределены по этим машинам

Существует возможность некоторым сценариям в тест-плане задать конкретные для выполнения машины, но они должны быть в пуле выполнения и зарегистрированы для проекта.

❗ И для тест-планов и для запуска сценариев, большая часть конфигурации по прежнему будет браться из сценария, перезаписывая только адрес драйвера и путь к приложению

7 ОБЩИЙ ПОТОК СОБЫТИЙ

При запуске пользователем сценария/проекта/тест-плана происходит следующий общий поток событий.

- UI отправляет на API запрос на создание сущности TestRun с вариантами создания (сценарий/проект/тестплан; нормальный режим/дебаг; редактор/база)

- API для каждого сценария в TestRun создает TestTask и записывает в нее конфигурацию, используя соответствующие источники информации.
- API рассылает идентификаторы задач в очереди, согласно маршрутизации
- TE принимает сообщение, конфигурирует поднятие контейнера с фреймворком.
- Фреймворк подключается к отдельной очереди задачи и готов принимать команды на выполнения шагов

8 РЕЖИМЫ ЗАПУСКА СЦЕНАРИЯ НА ВЫПОЛНЕНИЕ

Пользователь, находясь в редакторе сценария, может отправить его на выполнение в двух режимах: нормальном режиме и режиме отладки.

💡 Не находясь в редакторе, пользователь может отправить сценарий на выполнение только в нормальном режиме с источником шагов из базы данных.

4. Нормальный режим запуска в редакторе

При нажатии пользователя на кнопку запуска поведение практически полностью соответствует общему потоку событий. Для передачи шагов сценария для выполнения происходит следующее:

API после создания TestRun и отправки TestTask подключается к очереди фрейморка (зная `routing_key`) и шлет туда весь набор шагов.

💡 При запуске из редактора, список шагов передается в запросе.

В конце сценария, посылается специальная команда-флаг завершения сценария

5. Отладочный режим

В отладочном режиме, часть шагов (например, до точки останова) может быть отправлена аналогично как в нормальном режиме. Последующие шаги (или набор шагов, если используем точки останова) отправляются по нажатию пользователя, на эндпоинт выполнения шага, который отправляет шаги в очередь на выполнение.

9 МАРШРУТИЗАЦИЯ

В зависимости от выполнения сценариев на локальных машинах пользователей и запуске TestPlan на пуле хостов, реализуются разные механизмы маршрутизации, во избежание коллизий выполнения на одинаковых хостах.

9.1 Маршрутизация при запуске на локальных машинах

Основные положение, варианты реализации и задачи находятся здесь:

Для избежание коллизий запуска разных сценариев на одной машине, с учетом того, что машины заранее не особо известны, добавлена таблица маршрутизации.

Таблица маршрутизации содержит в себе адреса машин, на которых сейчас проходят сценарии и routing_key consumers обрабатывающих эти запросы. Совпадающие сценарии с уже имеющимися адресами машин идут к тем же консумерам. Другим - назначаются свободные консумеры.

9.2 Маршрутизация при запуске TestPlan

Для запуска TestPlan адреса машин должны быть заранее известны и быть в базе. При поднятии системы ТЕ поднимает столько воркеров-консумеров (потоки) сколько записей хостов в базе. Каждый воркер жестко привязан к хосту (по routing_key). При отправке задач, консумеры назначаются задачам циклически.

9.3 Описание алгоритма

9.3.1 Привязка хостов к проектам

В настройках проекта/сценария сохранен `DRIVER_URL` и `APP_PATH` , которые связаны друг с другом (приложение должно находиться по пути на машине). Эти параметры изменяемые, их может заменить любой пользователь, чтобы запустить сценарий на свой машине.

Соответственно, при запуске тест-плана на пуле выбранных хостов, эти параметры необходимо перезаписывать на известные.

Так как проект по своей сути равен тестированию одного приложения, и тест-планы определены в рамках одного проекта (не подпроекта), было принято решение о привязки (регистрации) хостов и информации о нем к проекту. Одни и те же хосты могут быть привязаны к разным проектам.

Для этого, в схему хранения проектов добавлено поле `available_host_maps` со следующей схемой:

```
available_host_maps: Optional[List[HostAppMapping]] = []
class HostAppMapping(BaseModel):
    host_id: int = Field(..., description="id of register host")
    app_path: Optional[Text] = Field(None, description="Path to app on this host.
If None use global conf")
```

На страницу конфигурации проекта добавится поле привязки хостов, где пользователь может добавить хосты к текущему проекту и указать путь к тестируемому приложению для данного проекта.

Список доступных хостов можно получить через `GET /api/v1/hosts`

9.3.2 Создание TestPlans

💡 Так как TestPlan определен в разрезе проекта (не подпроекта) в дальнейшем для краткости будет использоваться слово **проект**

Со стороны пользователя, создание UI представляет собой выбор проектов или сценариев из дерева. Если выбран проект - то все сценарии (в том числе в его подпроектах) будут выбраны.

В том числе, TestPlan можно указать имя, описание, и выбрать хосты из пула хостов проекта (которые были зарегистрированы в предыдущем пункте)

Для каждого выбранного проекта/сценария можно настроить конфигурацию запуска. На данный момент конфигурация запуска представляет собой выбор определенного хоста для запуска сценария/сценариев из проекта на нем.

Так как настройка отдельных сценариев/проектов представляется редким случаем, в схеме хранения помимо списка идентификаторов проектов и сценариев определены два поля:

```
class InputTestPlan:
        configuration_cases_map: Dict[Text, EntityConfiguration] = Field({},
description="Custom configuration for case")
        configuration_projects_map: Dict[Text, EntityConfiguration] = Field({},
description="Custom configuration for project")
        configuration: TestPlanConfiguration = Field(None,
description="Configuration")
....
class EntityConfiguration(BaseModel):
    # В дальнейшем можно наследоваться от BaseConfiguration
    main_host_id: Optional[int] = Field(None, description="Selected host for
running this entity")

class TestPlanConfiguration(BaseModel):
    available_host_ids: List[int] = Field(..., description="Id of available hosts
for running test plan")
```

9.3.3 Выполнение TestPlans и проброс конфигурации

Для выполнения TestPlan необходимо создать TestRun

При создании TestRun будут определены TestTasks для каждого сценария, из пула хостов будут назначены необходимые машины, сообщения пойдут на обработку TE для поднятия фреймворка и прохождения сценариев.

Но, если хост (`DRIVER_URL`) можно получить благодаря `routing_key` информацию о приложении, и других переопределенных параметрах запуска тест-плана можно получить только из сущности TestTask. Поэтому, при создании TestTask, API перезаписывает необходимые параметры и сохраняет их в новом поле `configuration` в теле TestTask. Например, `APP_PATH` . Из сущности задачи берутся настройки и прокидываются во фреймворка или регулируют запуск TE.

10 КОНФИГУРАЦИЯ СЦЕНАРИЯ

Для гибкой настройки запусков сценариев возможна конфигурация. Оно позволяет настроить параметры таймаутов, адреса API, выбор фреймворка для запуска и т.д

11 УПРАВЛЕНИЕ ЗАПУЩЕННЫМ СЦЕНАРИЕМ

11.1 Отмена уже запущенного сценария

Уже запущенный сценарий можно отменить. Так как шаги на выполнения также идут в очередь, отмена сценария реализована через посыл сообщения об завершении сценария с более высокими приоритетом. При этом невыполненные шаги отбрасываются, меняется статус задачи.

Сообщение о завершении посылается через API

11.2 Отмена сценария в очереди

Сценарий, который запущен, но все еще ждет в очереди для выполнения также можно отменить. Это происходит путем запроса на API, который изменит статус сценария и отправит очередь фреймворка сообщение о завершении

❗ Возможно также делать проверку при старте задачи в TE - но это не реализовано, также вопрос, что будет с очередью фреймворка.

12 СТАТУСЫ И РЕЗУЛЬТАТ TESTTASKS И TESTRUN

TestTasks имеет два поля для отображения процесса и результата: `status` и `result`

Status

Status	В каких случаях
pending	Задача в очереди
processing	Задача обрабатывается
finished	Выполнение завершено

Result

Result	В каких случаях
passed	Тест прошел
failed	Тест упал
undefined	Неопределен
aborted	Тест упал по таймауту
cancelled	Тест отменен пользователем

13 ПЕРЕМЕННЫЕ ОКРУЖЕНИЯ

Ниже представлен набор переменных окружения и их значений по умолчанию.

```
# Фреймворк.
STEP_REPEAT_TIME: float = field(default=os.environ.get("STEP_REPEAT_TIME", 1.0))
STEP_REPEAT_DELAY: float = field(default=os.environ.get("STEP_REPEAT_DELAY", 0.5))
STEP_DELAY: float = field(default=os.environ.get("STEP_DELAY", 0.0))
DEBUG_MODE: Text = field(default=os.environ.get("DEBUG_MODE", "False"))
DELAYED_APP_LAUNCH: bool = field(default=os.environ.get("DELAYED_APP_LAUNCH",
"True"))

# Драйвер.
DRIVER_URL: Text = field(default=os.environ.get("DRIVER_URL",
"<http://localhost:9999>"))
APP_PATH: Text = field(default=os.environ.get("APP_PATH",

r'<LOCALAPPDATA>\\GoInvestPro\\GoInvestPro.exe'))
LAUNCH_DELAY: int = field(default=os.environ.get("LAUNCH_DELAY", 1.5 * 1000))
CONNECT_TO_RUNNING_APP: bool =
field(default=os.environ.get("CONNECT_TO_RUNNING_APP", True))
# PROCESS_NAME: Text = field(default=os.environ.get("PROCESS_NAME", "HD-Player"))
API_URL: Text = field(default=os.environ.get("API_URL",
"<https://nc.rnd.lanit.ru>"))
PROJECT_NAME: Text = field(default=os.environ.get("PROJECT_NAME", "СТУДИО_тестовое
название-0"))

# Настройки API.
NEUROMOON_LINK: Text = field(default=os.environ.get("NEUROMOON_LINK",
"<http://localhost/runs>"))
TOKEN: Text = field(default=os.environ.get("TOKEN"))
MODEL_VERSION: Text = field(default=os.environ.get("MODEL_VERSION", "invest-
640s"))
MODEL_LOCALIZE_VERSION: Text =
field(default=os.environ.get("MODEL_LOCALIZE_VERSION", "desktop-640s-td"))
TEMPLATES_PROJECT: Text = field(default=os.environ.get("TEMPLATES_PROJECT",
"goinvest"))

# Сценарии.
SCENARIO_NAME: Text = field(default=os.environ.get("SCENARIO_NAME", "Win10CRM for
GoInvest test 2.0"))
DIR_SCREENSHOTS: Text = field(default="__tmp__")

# Кейсы.
ALIASES: Dict = field(default=os.environ.get("ALIASES", None))

# Rabbit MQ.
PIKA_LOGIN: Text = field(default=os.environ.get("PIKA_LOGIN", ...))
PIKA_PASS: Text = field(default=os.environ.get("PIKA_PASS", ...))
PIKA_HOST: Text = field(default=os.environ.get("PIKA_HOST", ...))
PIKA_PORT: Text = field(default=os.environ.get("PIKA_PORT", "5672"))
QUEUE_NAME: Text = field(default=os.environ.get("QUEUE_NAME", "task_564"))

# Allure.
ALLURE_RESULT_DIR: Text = field(default=os.environ.get("ALLURE_RESULT_DIR",
r"report"))
ALLURE_SERVICE: Text = field(default=os.environ.get("ALLURE_SERVICE",
"<http://192.168.88.102:5050>"))
```

14 РЕЖИМ ОТЛАДКИ

Всё вышеописанное определяет обычный режим прогона тестового сценария. **Фреймворк** может быть запущен в отладочном режиме. Если переменная окружения `DEBUG_MODE` установлена в положительное значение, **фреймворк** запускается в режиме отладки.

Если **фреймворк** работает в отладочном режиме, он не завершает своё исполнение на ошибке, логируя её как следует, продолжает слушать очередь сообщений до тех пор, пока не придёт сообщение с текстом окончания, или отмены сценария. После завершающего сообщения **фреймворк** производит стандартный вывод при завершении теста.

Если **фреймворк** получил несколько шагов, однако выполнение всех шагов не было произведено (например, в результате ошибки NoSuchElementException), **фреймворк** выводит в консоль сообщение с количеством полученных и отклонённых шагов из последнего пака и очищает очередь. Пример соответствующего кода. Пример выводимого сообщения `"Steps executed: >=0, rejected {cleaned} steps."`.

После получения одного из сообщений, сигнализирующих завершение отладки, результат выполнения сценария выводится как успешно пройденный, **фреймворк** завершает своё исполнение в штатном режиме.

15 НЕШТАТНОЕ ЗАВЕРШЕНИЕ РАБОТЫ ФРЕЙМВОРКА

TE поднимает Docker-контейнер, ожидая консольного вывода. Если в течении заданного интервала времени ожидания логов от **фреймворка** не последует консольного вывода, **TE** завершит исполнение контейнера.

Кроме того **TE** завершает выполнение контейнера автоматически при истечении общего таймаута на выполнение сценария.

Данные таймауты задаются индивидуально для каждого запуска тестового сценария и, соответственно, контейнера.

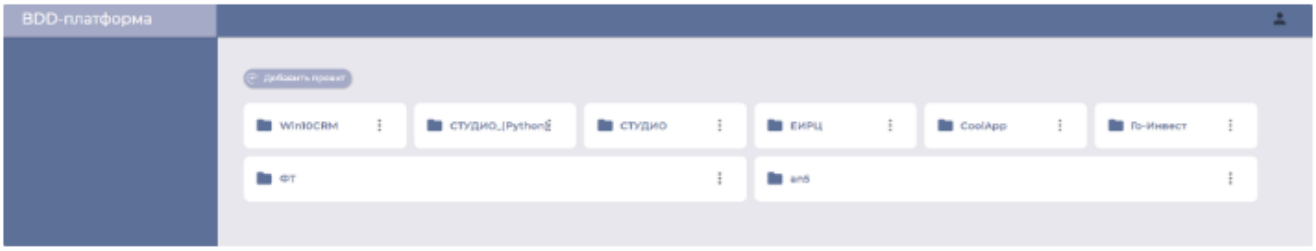
16 РУКОВОДСТВО ПО ЗАПУСКУ СЦЕНАРИЯ

16.1 Проекты

Проекты являются своеобразными папками в платформе. Они содержат в себя набор сценариев. Также они определяют основные настройки конфигурации, которые наследуют вложенные подпроекты и сценарии.

Внутри проекта можно создать как подпроект, так и сценарий.

⚠ На данный момент нет ограничения в количестве создания подпроектов или сценариев на одном уровне. Определено лишь максимальное число вложенных уровней - не получится создать более 10 вложенных подпроектов.



Страница проектов

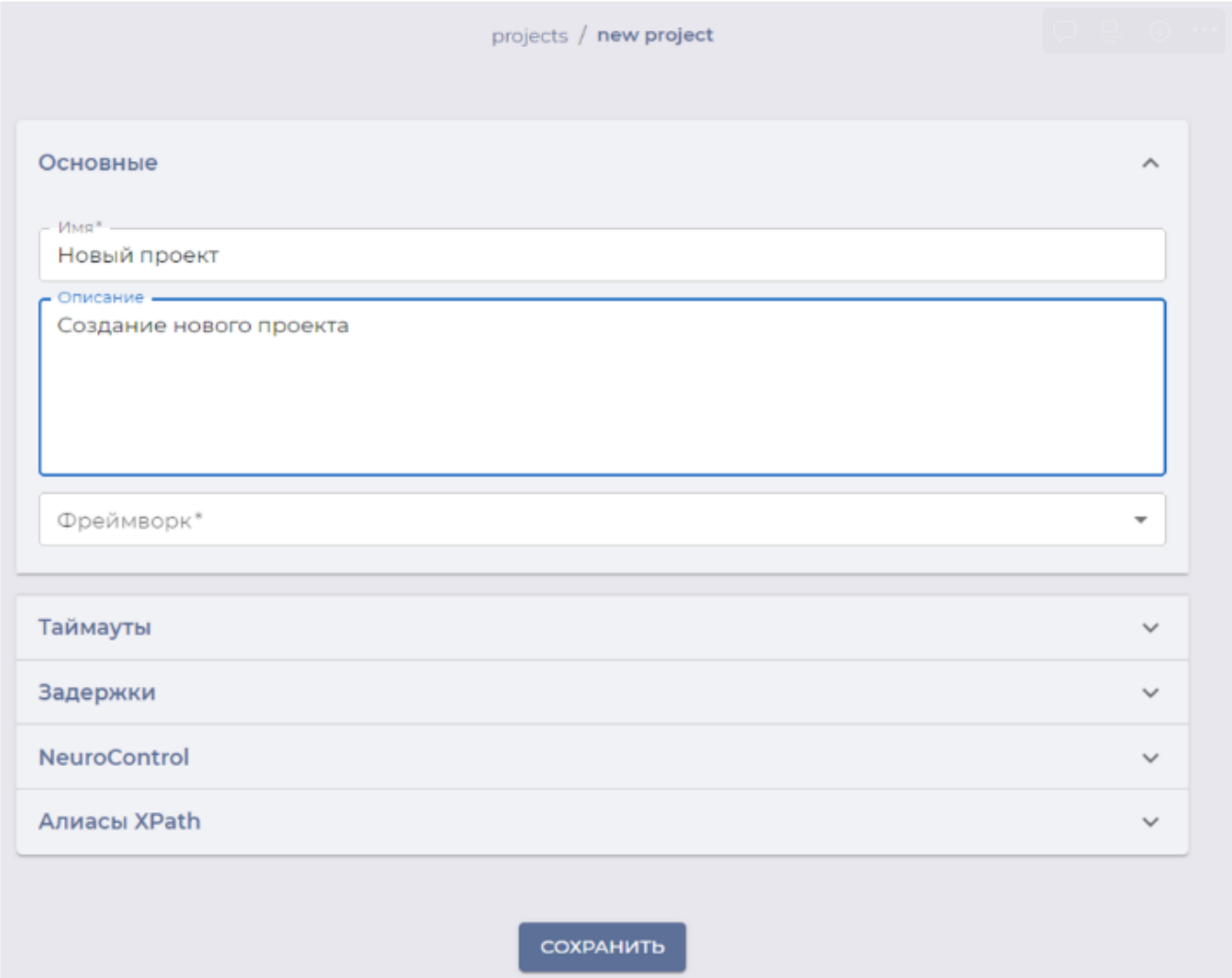
16.2 Создание проекта

Для создания проекта необходимо нажать на кнопку “Добавить проект” на главной странице.

После нажатия потребуется заполнить поля для создания проекта.

Обязательными полями являются `Имя проекта`. Остальные поля можно заполнить позже

⚠ Все сценарии этого проекта будут наследовать заполненные поля конфигурации, если вы не перепишите их отдельно.



Создание нового проекта

После нажатия кнопки сохранить будет произведено создание нового проекта

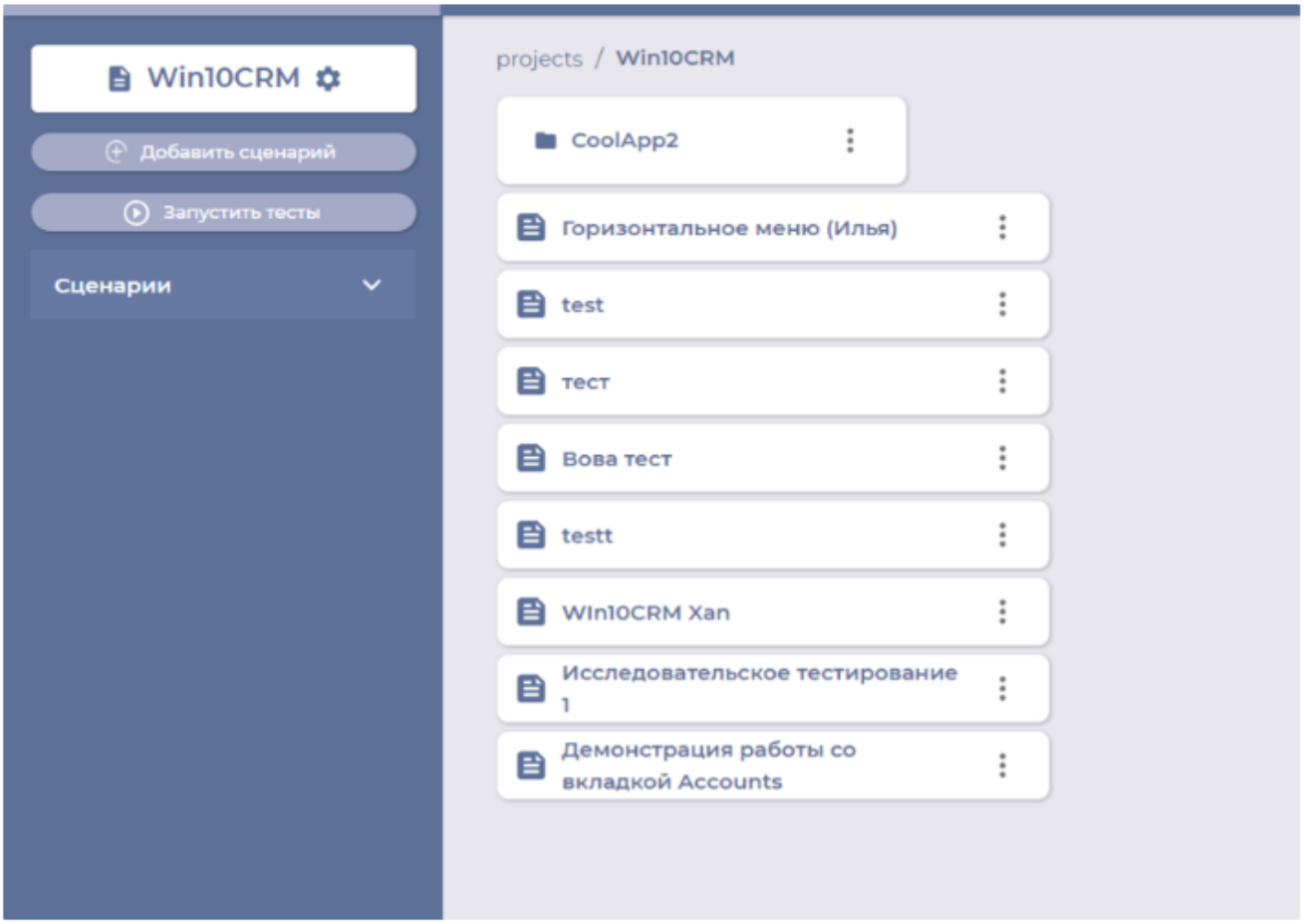
⚠ Имя проекта должно быть уникальным

16.3 Просмотр проекта

Внутри проекта будет отображен список доступных подпроектов и сценариев, которые лежат непосредственно в этом проекте.

Проект наследует папочную архитектуру, так что можете рассматривать проекты и подпроекты - как папки, а сценарии - как файлы.

⚠ Единственное отличие подпроекта и проекта - у проекта нет родителя, он является корневой “директорий”



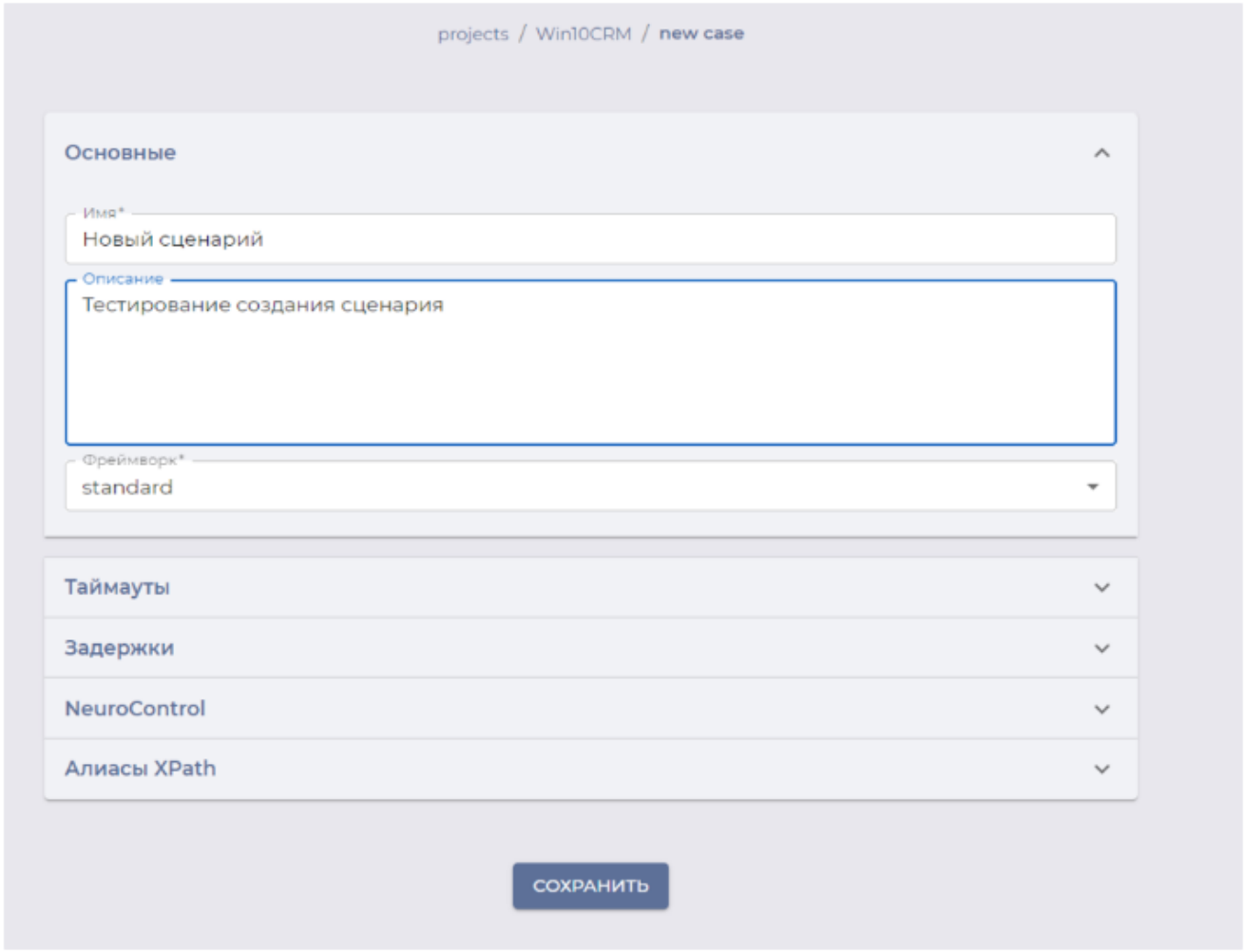
Внутри проекта

16.4 Сценарии

Сценарий является сущностью тестового сценария.

16.5 Создание сценария

При нажатии кнопки “Добавить сценарий” из проекта будет открыто окно создания сценария.


projects / Win10CRM / new case
Основные
Имя*
Новый сценарий
Описание
Тестирование создания сценария
Фреймворк*
standard
Таймауты
Задержки
NeuroControl
Алиасы XPath
СОХРАНИТЬ

Приложение 1

Инструкция по развертыванию

1 ТЕКУЩИЕ ОСОБЕННОСТИ

При старте сервисов, возможно придется перестартовать API и TE после того, как добавили реплики к базе.

RabbitMQ долго поднимается, API и TE должны ожидать его поднятия. Если этого не произошло, стоит перезапустить API и TE после того, как RabbitMQ будет готов.

2 ПОДГОТОВКА

2.1 Импортирование фреймворков

Для импортирования базы фреймворков, необходимо сгенерировать файлы экспорта или же импортировать данные сразу в поднятую базу данных

Для получения файлов экспорта достаточно выполнить следующие команды:

```
cd {директория-фреймворка}
python scripts/step.py
```

Команда создаст файлы для экспорта и сохранит их в текущей директории. В дальнейшем эти файлы необходимо передать в сервис API для подгрузки их при старте сервиса.

Для импорта в поднятую базу данных необходимо добавить данные доступа к БД:

```
cd {директория-фреймворка}
python scripts/step.py --db_host {database_host} --db_port {database_port}
```

Скрипт создаст файлы экспорта и сам же их импортирует в указанную базу данных. Имя БД и коллекция будут созданы автоматически.

3 СБОР ОБРАЗОВ

3.1 Сбор и именование образов фреймворков

Образ фреймворка собирается отдельно, так как не включен в `docker-compose.yml` и запускается каждый раз при старте теста из контейнера **`Task Executor`**

Мы придерживаемся определенные правил по именования образов фреймворка для получения к ним автоматического доступа из **Task Executor**

Формат образа фреймворка следующий: `{ImagePrefix}{FrameworkName}:FrameworkVersion`

⚠ Параметры `FrameworkName` и `FrameworkVersion` должны быть указаны в `version.ini` файле в директории фреймворка и использоваться при импорте в БД

Соответственно, для изменения имени фреймворка необходимо изменить данные в `version.ini` файле и обновить базу данных. После этого можно изменить имя образа фреймворка

Имя образа будет доступно Task Executor по идентификатору фреймворку из базы данных.

⚠ Поле `{ImagePrefix}` является опциональным и позволяет использовать Docker-регистр без переименования фреймворка

Для сбора образа фреймворка:

```
docker build -t {FrameworkName}:FrameworkVersion .
```

Для автоматического получения `{FrameworkName}:FrameworkVersion` из `version.ini`:

```
FRAMEWORK_NAME=$(awk -F "=" '/name/ {print $2}' version.ini)
FRAMEWORK_VERSION=$(awk -F "=" '/version/ {print $2}' version.ini)
docker build  -t $FRAMEWORK_NAME:$FRAMEWORK_VERSION .
```

3.2 Сбор образа API для импорта фреймворков при старте API

⚠ Если вы не импортируете фреймворки через старт API или собираетесь использовать тома - пропустите этот раздел. Использование томов - предпочтительнее.

Полученные файлы фреймворков можно загрузить в базовый образ API, тогда они будут импортированы в БД при старте сервиса.

Перейдите в директорию с исходниками сервиса API и создайте директорию `app/docs`

```
cd bdd-api/app
mkdir docs
```

В созданную директорию необходимо перенести сгенерированные файлы фреймворка.

После этого соберите образ контейнера

```
docker build -t bdd-api .
```

4 ПОДГОТОВКА ПЕРЕМЕННЫХ ОКРУЖЕНИЯ

Для развертывания необходимо два .env файла
Эти файлы должны лежать в директории для платформы.
Основной .env файл.

RABBITMQ_DEFAULT_USER={*выдаётся по доп. запросу}*

RABBITMQ_DEFAULT_PASS=*{выдаётся по доп. запросу}*

ALLURE_PUBLIC_URL=http://192.168.88.102:5050

В нем необходимо заменить значение переменной ALLURE_PUBLIC_URL на свое доменное имя или IP-адрес с указанием порта сервиса Allure. Эта информация будет использована для сохранения отчетов Allure фреймворком.

Также здесь вы можете поменять доступы до RabbitMQ

Переменные окружение для Executor: .env.executor

IMAGE_PREFIX=registry.rnd.lanit.ru/dev-desktopai-python-bdd-project

PIKA_HOST=127.0.0.1

PIKA_PORT=5672

- Несмотря на то, что контейнеры Docker-Compose по умолчанию находятся в одной сети и имеют доступ друг к другу, контейнер фреймворка по умолчанию не поднимается в этой сети и требует явный доступ до связанных сервисов. Поэтому в переменных для Executor требуется явно указать адрес **RabbitMQ** для доступа к очереди задач.
- Укажите свой **IMAGE_PREFIX** для образа фреймворка. Если используются рекомендации по сборке фреймворка, то **IMAGE_PREFIX** переменная будет представлять собой пусть до образа, исключая его имя и тэг

❗ Часть имени может входить в префикс, зависит от того, как именовали образ. **Executor** рассчитывает получить имя образа путем сложения префикса, имени фреймворка и добавления тэга. Имя фреймворка и тэг будут взяты из БД (с учетом импорта фреймворков) при запуске первого сценария.

❗ В любом случае, вы всегда можете спуллить/собрать образ фреймворка с необходимым именем.

5 РАЗВЕРТЫВАНИЕ

1. Убедитесь, что установлен Docker/Docker-compose
2. Создайте директорию для платформы
3. `mkdir bdd`
4. `cd bdd`
5. Создайте `**.env**` и `**.env.executor**` файлы как указано в Подготовка переменных окружения
6. Если производите импорт шагов через тома без встройки в образ API, как указано здесь Сбор образа API для импорта фреймворков при старте API
7. Создайте директорию

   ```
   mkdir frame-migrations
   ```

8. Положите экспортированные файлы томов в эту директорию.

В приложенном файле `docker-compose.yml` добавьте том на директорию `/usr/src/app/docs` в сервис **`bdd-api`**

```
volumes:
      - ${PWD}frame-migrations:/usr/src/app/docs
```

9. Скопируйте приложенный docker-compose.yml
10. Выполните следующие команды для пуллинга и поднятия контейнеров.

```
docker-compose pull

docker-compose up -d
```

11. Выполните следующие команды для конвертации базы в ReplicaSet

```
docker exec -it bdd_mongo_1  mongo

rs.initiate(
 {
   _id : 'rs0',
   members: [
    { _id : 0, host : "mongo:27017" }
   ]
 }
)
```

При разворачивании могут быть встречены проблемы. Как правило, они решаются перезапуском отдельных сервисов.

```
docker-compose stop api
...
docker-compose up -d api
```

После проделанных шагов откройте `{domain_name/ip}:9037` в браузере.