AgniaChallenge Руководство

В данном руководстве описано, как настроить **Telegram**-бота для работы с ИИ-агентом, чтобы выполнять запросы в корпоративных системах (**TeamFlame, Todoist, GitFlame**) и добавлять новые интеграции для расширения возможностей системы.

Полные технические детали, примеры и исходный код доступны в YandexCloud.

❗ Примечание ❗

- Убедитесь, что вы используете версию **Python 3.11+**

Обзор системы

ИИ-агент позволяет взаимодействовать с корпоративными системами через простые текстовые запросы, отправленные в **Telegram-бота**. На основе этих запросов агент формирует план последовательных действий, используя доступные API действия.

Цель хакатона — расширить функциональные возможности агента, добавляя новые интеграции с различными системами. Участники могут создавать как новые интеграции, так и расширять функционал уже поддерживаемых платформ (например, **Todoist**, **TeamFlame**, **GitFlame**).

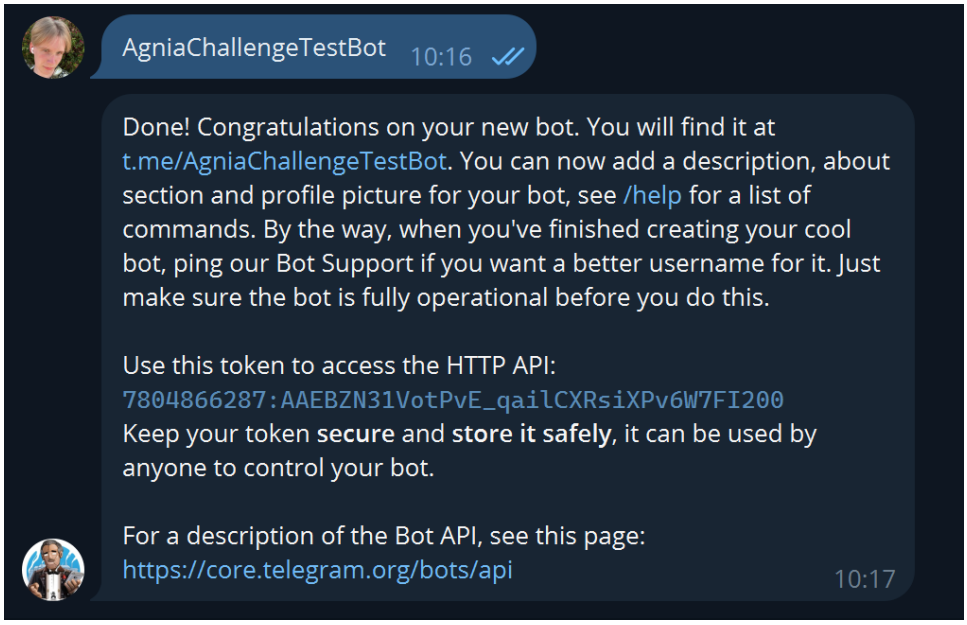
Подключение и настройка Telegram-бота

1. Получение Team ID

В день начала хакатона **менторы** отправят капитанам команд уникальные **Team ID (UUID)**. Этот ID нужен для настройки бота и подключения к системе.

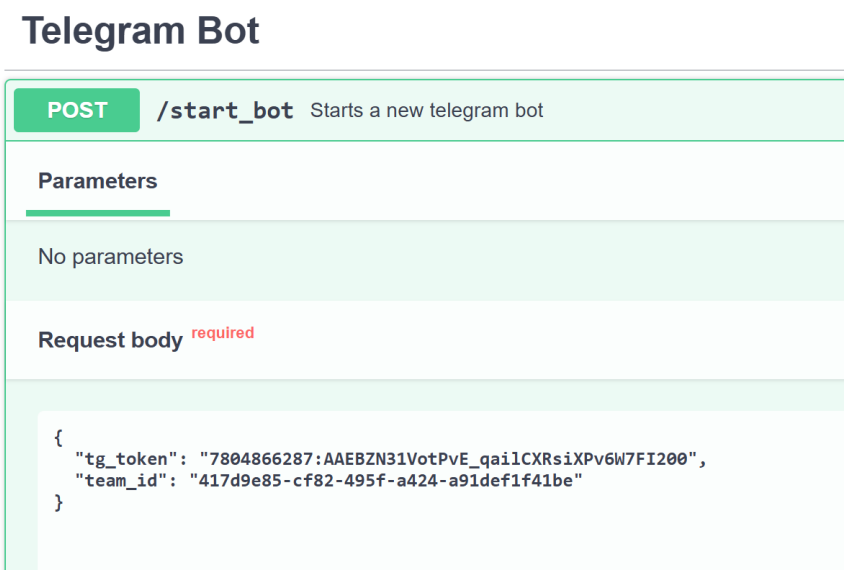
2. Создание Telegram-бота

Команде необходимо создать бота в Telegram через BotFather. После создания сохраните **токен бота**, так как он потребуется для дальнейшей настройки.



3. Активация Telegram-бота

Активируйте бота с помощью эндпоинта **/start_bot**. Отправьте запрос на эндпоинт с **Telegram-токеном** и вашим **Team ID**.



Установка репозитория AgniaChallenge

Репозиторий **AgniaChallenge** с примерами и инструкциями можно скачать по ссылке: https://disk.yandex.ru/d/8LfMniC-sqZhEA.

Добавление интеграций с внешними системами

ИИ-агент поддерживает интеграции с корпоративными системами, такими как
Todoist, **TeamFlame** и **GitFlame**, что позволяет автоматизировать задачи и запросы в этих системах.

Примечание: Хотя **Todoist** уже содержит набор реализованных действий, мы не рассматриваем его как полностью интегрированную систему.
Авторизация для **Todoist** будет использоваться в качестве примера для

интеграции с новой системой, чтобы участники могли написать подобные авторизации для других систем. При этом **Todoist** остается доступен для использования в задачах и дополнений действий.

Интеграция новой системы

Если для задачи требуется новая система, участник может создать интеграцию с ней.

Для этого необходимо выполнить следующие шаги:

Шаг 1: Запустить авторизационный сервис локально

Поднимите локальный сервис, который позволит вам авторизоваться в тех системах, которые вы будете интегрировать. Без этого сервиса не будет получится выполнять действия в добавленных системах черег **Telegram-бота**.

Для запуска авторизационного сервиса:

1. **Установите зависимости**

Находясь в корне проекта, выполните:

```
pip install -r authorizations/requirements.txt
```

Для удобства работы, можно поставить зависимости в отдельное вирт. окружение, например на Linux/MacOS:

```
python -m venv myvenv
source myvenv/bin/activate
pip install -r authorizations/requirements.txt
```

Примеры для Windows можно найти тут.

2. **Добавьте токен Telegram-бота в настройки**

Перейдите в файл `authorizations/src/settings.py`. Здесь, в параметре `user_token_from_tg_bot` нужно указать токен из **Telegram-бота**. Для получения токена в телеграм боте можно воспользоваться командой `/info`.

3. **Запустите авторизационный сервис**

Запустите сервис, выполнив следующую команду из корня проекта:

```
uvicorn authorizations.src.main:app --reload --port 9000
```

В случае, если сервис запустился успешно, вы должны увидеть **Swagger**-документацию по адресу http://localhost:9000/docs .

При внесении изменений в директории `authorizations` запущенный сервис обновится автоматически.

Шаг 2: Добавить авторизацию для новой системы

На данном шаге представлен общий алгоритм для добавления авторизации. Пример реализации авторизации для **Todoist** представлен в изачальной версии сервиса, предоставленной вам.

Содержимое авторизационного модуля

Изначально, структура авторизационного модуля `authorizations` выглядит так:

```
.
├── requirements.txt
└── src
        ├── authorization_services
```

```
│   ├── init.py
│   └── todoist.py
├── hackathon_utils.py
├── init.py
├── main.py
└── settings.py
```

В файле
`main.py` находятся **API-эндпоинты** для авторизации в системе.
В файле
`settings.py` находятся общие настройки. Туда же можно добавлять настройки для авторизации в конкретной системе.
В директории
`authorizations_services` находятся Python-модули с логикой авторизации. Например, в модуле ***todoist.py*** находится логика авторизации для **Todoist.**
В
`hackathon_utils` находится служебная функция, сохраняющая авторизационные данные для вашей системы в ИИ-агенте для их последующего использования в **Telegram-боте.**

Алгоритм написания авторизации для новой системы
Чтобы добавить авторизацию в новой системе, нужно:

1. Добавить API-эндпоинт, через который вы будете авторизовываться в своей системе, в `main.py`. В изначальном `main.py` представлены API-эндпоинты для авторизации в Todoist через OAuth.
2. Написать логику авторизации внутри API-эндпоинта, чтобы получить токены авторизации. Логику авторизации можно при желании вынести в директорию `authorization_services` (в этой директории изначально уже содержится пример для Todoist).
3. Завершить функцию для API-эндпоинта вызовом:

```
return save_authorization_data_and_return_response(
    your_auth_data, system_name="your_system_name"
)
```

где `your_auth_data` - это токены/другие типы авторизационных данных, которые вы получили при авторизации. Это гарантирует, что ваши авторизационные данные будут сохранены в нашей системе и могут быть использованы при выполнении действий через телеграм-бот. При выполнении действий, сохраненные авторизационные данные подставятся в словарь `authorization_data` в следующем формате: `{"your_system_name": your_auth_data}`. О написании действий читайте ниже.

4. Добавить необходимые зависимости в requirements.txt.

❗ **Важно** ❗ **:** Авторизационные данные будут сохранены на наших серверах, поэтому не входите в учетные записи, содержащие конфиденциальную информацию (лучше создать тестовые учетные записи).

Шаг 3: Авторизоваться в добавленной системе

Выполнять авторизацию нужно через добавленные **API-эндпоинты**. Пример авторизации для **Todoist** представлен ниже.

Пример добавления авторизации для Todoist

Процесс авторизации в Todoist

Для лучшего понимания, процесс авторизации в **Todoist** выглядит следующим образом:

1. На стороне авторизационного сервиса генерируется ссылка для авторизации.
2. Пользователь переходит по ссылке и попадает на страницу **Todoist**, где предоставляет доступ к своему аккаунту.
3. Пользователь перенаправляется обратно в авторизационный сервис, с кодом авторизации.
4. Авторизационный сервис обращается в **Todoist** для обмена кода авторизации на токены.
5. Полученные токены отправляются в нашу систему и там сохраняются.

Перед добавлением авторизации в **Todoist** необходимо создать **OAuth приложение**, т.к. авторизация в Todoist реализована через OAuth протокол**.** Создать приложение можно, предварительно авторизовавшись в **Todoist** и перейдя по ссылке https://developer.todoist.com/appconsole.html .
В качестве
OAuth redirect URL нужно указать http://localhost:9000/todoist/get-token.
Если вы запустили авторизационный сервис на другом порту - порт необходимо поменять.

Процесс добавления новой авторизации в аторизационный сервис

Пройдем по шагам добавления новой авторизации:

1. *Добавить API-эндпоинт, через который вы будете авторизовываться в своей системе, в* `main.py`

   В `main.py` можно увидеть 2 API-эндпоинта для авторизации:

   - `/todoist/authorize` - основной API-эндпоинт, через который пользователь запускает процесс авторизации в **Todoist.**

- `/todoist/get-token` - служебный API-эндпоинт, в который Todoist отправляет авторизационный код.

2. *Написать логику авторизации внутри API-эндпоинта, чтобы получить токены авторизации*

   Логика авторизации находится в `authorizations/todoist.py`. Также, для авторизации в **Todoist**, нужно будет добавить значения из **OAuth-приложения** в настройки. В файле `settings.py` , в `todoist_client_id` и `todoist_client_secret` укажите Client ID и Client secret вашего **Todoist** OAuth-приложения. Если вы запускаете авторизационный сервис на порту, отличном от *9000*, измените значение в `todoist_redirect_url`.

3. *Завершить функцию для API-эндпоинта вызовом* `save_authorization_data_and_return_response`

   В API-эндпоинте `/todoist/get-token` в `main.py`, после получения токенов, они сохраняются в систему Agnia через вызов `save_authorization_data_and_return_response`.

4. *Добавить необходимые зависимости в* `requirements.txt`

   В нашем случае новые библиотеки не были использованы, поэтому оставляем без изменений.

Авторизация в добавленной системе

Для авторизации в добавленной системе, нужно запустить авторизационный сервис и обратиться по
API. Например, через **Swagger** в соответствующий **API-эндпоинт**.
Предполагается, что авторизационный сервис запускается у каждого участника отдельно, локально.

В примере с
Todoist, мы отправляем запрос на http://localhost:9000/todoist/authorize, переходим по полученной ссылке, даем доступ к нашему аккаунту. Далее, получаем сообщение, что авторизационные данные были успешно сохранены.

Pretty-print ☐

{"message":"Authorization successful and data saved."}

Использование существующих интеграций (TeamFlame, GitFlame)

Чтобы использовать действия в подключенных системах, пользователь должен пройти авторизацию через Telegram-бота, используя команду `/login`. После этого агент сможет пользоваться действиями существующих систем, используя доступ к указанным сервисам.

Создание действий

ИИ-агент выполняет запросы пользователей, разбивая их на последовательность действий. Действия представляют собой отдельные функции, которые агент использует для выполнения задач в системах (например, создание задачи в **TeamFlame**, получение информации об *issue* в **GitFlame** и пр.).

Перед добавлением новых действий, обязательно укажите свой `team_id` в настройках `team_actions/src/settings.py`.

Создание действия `create_task` для Todoist

В качестве примера рассмотрим создание действия

`create_task` для создания задачи в **Todoist**. Работаем в директории `/team_actions/src/actions/Todoist/`.

Основные шаги

1. **Определение типов данных**:
    - Используем **Type Hints** для определения входных/выходных типов параметров. Эти типы данных были собраны вручную на основе официальной документации API Todoist: Todoist API.
2. **Определение структуры данных (модели Pydantic)**:
    - Используем **Pydantic** для создания моделей, которые описывают структуру данных, возвращаемую **API**.

    Примечание: такие **структуры данных** и **type hints** необходимы для корректной работы ИИ-агента, чтобы он мог распознавать формат возвращаемых значений. В самой функции аннотировать входные и выходные данные не обязательно. По факту, функция будет возвращать `dict` (JSON), но мы описываем его через модель **Pydantic** для согласования структуры данных с агентом.
3. **Определение функции добавляемого действия**:
    - Создаём функцию `create_task` (в нашем примере она находится в `actions.py`), которая будет выполнять **API** вызов для создания новой задачи.
    - Получаем в теле функции авторизационный токен / ключ из словаря `authorization_data` по названию системы. Токен / ключ будет подставлен туда автоматически при выполнении действий.
4. **Добавление регистрационного декоратора**:
    - Применяем декоратор для регистрации действия, который связывает функцию с агентом.
5. **Добавление информации о системе**:
    - Добавляем описание к доступным системам, чтобы ИИ-агент мог составлять корректные планы.

- Добавляем Python-модуль с действиями для системы в роутер, если мы подключили новую систему.

6. **Написание документации к действиям системы**:
   - Оформляем документацию, чтобы указать, какие параметры принимают функции и что они возвращают.
7. **Добавление новых зависимостей в `requirements.txt`**:
   - Если в ваших действиях вы использовали новые библиотеки, добавляем их в requirements.txt.

Пример кода

1-2. Определение типов и структуры данных

```
from typing import Annotated, Optional, List, Literal
from pydantic import BaseModel, Field, HttpUrl

# Определяем Type Hints для входных параметров
Id = Annotated[str, Field(pattern="^[0-9]+$")]
ProjectId = Annotated[Id]
SectionId = Annotated[Id]
TaskName = Annotated[
    str,
    Field(description="Task name")
]
Datetime = Annotated[
    str,
    Field(
        pattern=(
            r"^\d{4}-\d{2}-\d{2}T\d{2}:"
            r"\d{2}:\d{2}(\.\d+)?Z$"
        )
    )
]
ShortDatetime = Annotated[
```

```
    str,
    Field(pattern=r"^\d{4}-\d{2}-\d{2}$")
]
DueString = Annotated[
    str,
    Field(
        description="Due date in English",
        example="tomorrow"
    )
]
DueLang = Annotated[
    str,
    Field(pattern="^[a-z]{2}$", default="en")
]
DurationUnit = Literal['minute', 'day']
Priority = Annotated[int, Field(ge=1, le=4)]

# Определяем модели данных для выходных параметров
class Due(BaseModel):
    string: Optional[DueString]
    date: Optional[ShortDatetime]
    is_recurring: bool
    datetime: Optional[Datetime]
    timezone: Optional[str]

class Duration(BaseModel):
    amount: Optional[int]
    unit: Optional[DurationUnit]

class Task(BaseModel):
    id: Id
    assigner_id: Optional[Id]
    assignee_id: Optional[Id]
    project_id: ProjectId
    section_id: Optional[SectionId]
    parent_id: Optional[Id]
```

```
    order: int
    content: TaskName
    description: Optional[str]
    is_completed: bool
    labels: Optional[List[str]]
    priority: Priority
    comment_count: int
    creator_id: Id
    created_at: Datetime
    due: Optional[Due]
    url: HttpUrl
    duration: Optional[Duration]
```

3. Определение функции действия

Создадим функцию в `team_actions/src/actions/Todoist/actions.py`:

```
authorization_data = {}
# Держите это поле пустым изначально.
# После регистрации действий в системе, сюда будут автоматиче
ски
# добавлены авторизационные данные участников.

def create_task(
        content: TaskName,
        description: Optional[str] = None,
        project_id: Optional[ProjectId] = None,
        section_id: Optional[SectionId] = None,
        labels: Optional[List[str]] = None,
        priority: Optional[Priority] = 1,
        due_string: Optional[DueString] = None,
        due_lang: Optional[DueLang] = "en",
        due_date: Optional[ShortDatetime] = None,
        due_datetime: Optional[Datetime] = None,
        duration: Optional[Duration] = None,
        duration_unit: Optional[DurationUnit] = None
```

```
) -> Task:
    # Логика вызова API Todoist для создания задачи
    response = requests.post(
        "https://api.todoist.com/rest/v2/tasks",
        headers={
            "Authorization": (
                f"Bearer {authorization_data['Todoist']}"
              )
          },
        json={
            "content": content,
            "description": description,
            "project_id": project_id,
            "section_id": section_id,
            "labels": labels,
            "priority": priority,
            "due_string": due_string,
            "due_lang": due_lang,
            "due_date": due_date,
            "due_datetime": due_datetime,
            "duration": duration,
            "duration_unit": duration_unit,
        }
    )
    response.raise_for_status()
    data = response.json()
    return data
```

4. Добавление регистрационного декоратора

Чтобы ИИ-агент получил всю необходимую информацию о действиях и системах участников, действия необходимо обернуть в декоратор `register_action`. Работаем также в `team_actions/src/actions/Todoist/actions.py`.

```
@register_action(
    system_type="task_tracker",
```

```
     # Может ли действие быть использовано в плане
    include_in_plan=True,
    signature=(
        "(content: TaskName, "
        "description: Optional[str] = None, "
        "project_id: Optional[ProjectId] = None, "
        "section_id: Optional[SectionId] = None, "
        "labels: Optional[List[str]] = None, "
        "priority: Optional[Priority] = None, "
        "due_string: Optional[DueString] = None, "
        "due_date: Optional[ShortDatetime] = None, "
        "due_datetime: Optional[Datetime] = None, "
        "due_lang: Optional[DueLang] = None, "
        "duration: Optional[Duration] = None, "
        "duration_unit: Optional[DurationUnit] = None) "
        "-> Task"
    ),
    arguments=[
        "content", "description", "project_id", "section_id",
        "labels", "priority", "due_string", "due_date",
        "due_datetime", "due_lang", "duration", "duration_unit"
    ],
    description="Creates a new task",
)
def create_task(
        content: TaskName,
        description: Optional[str] = None,
        project_id: Optional[ProjectId] = None,
        section_id: Optional[SectionId] = None,
        labels: Optional[List[str]] = None,
        priority: Optional[Priority] = 1,
        due_string: Optional[DueString] = None,
        due_lang: Optional[DueLang] = "en",
        due_date: Optional[ShortDatetime] = None,
        due_datetime: Optional[Datetime] = None,
        duration: Optional[Duration] = None,
```

```
        duration_unit: Optional[DurationUnit] = None
) -> Task: ...
```

5. Добавление информации о системе

Чтобы ИИ-агент понимал, для каких целей используются различные системы, поддерживается словарь `systems_info` в файле `team_actions/systems_config.py`. Этот словарь включает описание категорий систем (например, системы управления задачами) и описания конкретных систем. Пример содержимого словаря:

```
systems_info: Dict[str, Any] = {
    "task_tracker": {
        "description": (
          "Manages tasks, issues, and projects (e.g., "
          "Todoist, TeamFlame)."
        ),
        "systems": {
          "TeamFlame": (
            "TeamFlame is a task tracking system where spaces "
            "hold projects, which contain Kanban boards. "
            "Boards manage tasks across columns like 'To Do', "
            "'In Progress', and 'Done'. Tasks have statuses "
            "such as epic, bug, etc., ensuring efficient "
            "organization and workflow control."
            ),
          "Todoist": (
            "Todoist is a task management app that organizes "
            "tasks into projects, allows setting due dates, "
            "priorities, and creating sub-tasks. It supports "
            "collaboration, custom filters, reminders, and "
            "tracks productivity across devices."
            ),
        },
    },
```

```
    "version_control_system": {
        "description": (
          "Tracks code changes, versions, and issues (e.g., "
          "GitHub, GitFlame)."
        ),
        "systems": {
          "GitFlame": (
            "GitFlame is a version control and collaboration "
            "tool that integrates Git repositories with task "
            "management."
            ),
        },
    },
}
```

Если вы планируете добавить новый тип системы или новую систему, вам необходимо дополнить этот словарь своими описаниями систем и типов систем.

Также, **при добавлении новой системы** необходимо дополнить файл `team_actions/src/router.py` :

```
from team_actions.src.utils.action_router import ActionRouter

# Import actions module for new system here
from team_actions.src.actions.Todoist import actions as todoist_

# Add your module to the router
ActionRouter.add_actions_for_module(todoist_actions)

# Keep it as is
action_router = ActionRouter()
```

6. Добавление документации к действиям системы

Подробная информация для написания документации к действию описана в документе “Документация к действиям системы”.

7. Добавление новых зависимостей в requirements.txt

Добавляем новые библиотеки, которые мы использовали, в `requirements.txt`. Для действия из примера мы не использовали новые библиотеки, поэтому оставляем `requirements.txt` без изменений.

Регистрация и запуск обработчика действий

После того как вы написали свои действия, обернули их в декоратор и подготовили документацию, выполните следующие шаги:

1. Перейдите в корневую директорию проекта AgniaChallenge.
2. Запустите следующий скрипт:

```
python -m team_actions.src.initial_setup
```

Этот скрипт автоматически подгрузит всю необходимую информацию о действиях в ИИ-агента.
Также, ваши действия будут запакованы в zip-архив и отправлены к нам на сервер. Результаты запуска обработчика ваших действий можно будет увидеть в консоли.

Также, при необходимости можно загрузить ваши действия в ИИ-агента вручную. Для этого отправьте zip-архив папки `team_actions` и `requirements.txt` в эндпоинт.

Проверка действий

Ваши действия были зарегистрированы и запущены.

Теперь вы можете испытать ваши действия в Telegram-боте!