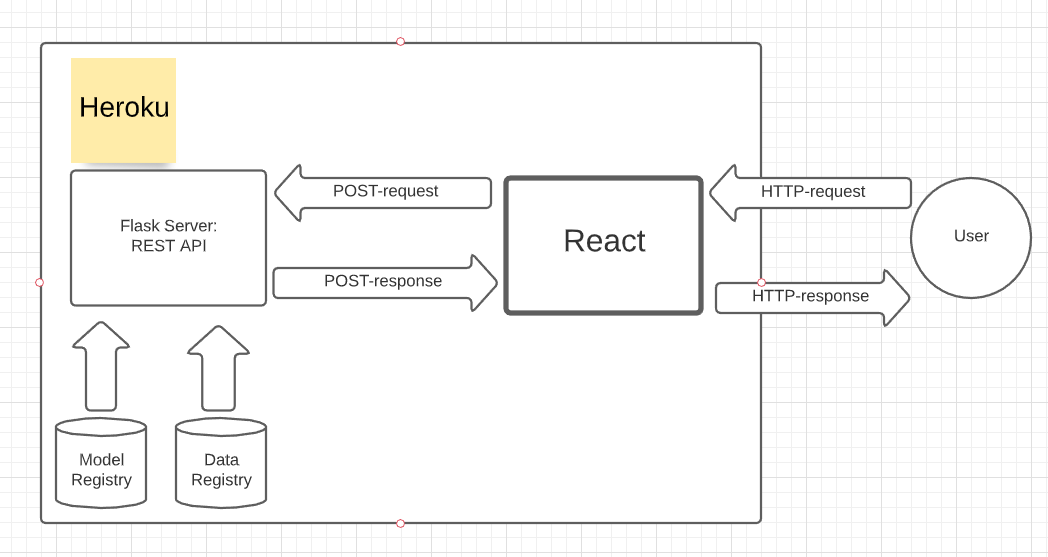
Проект «Переводчик»

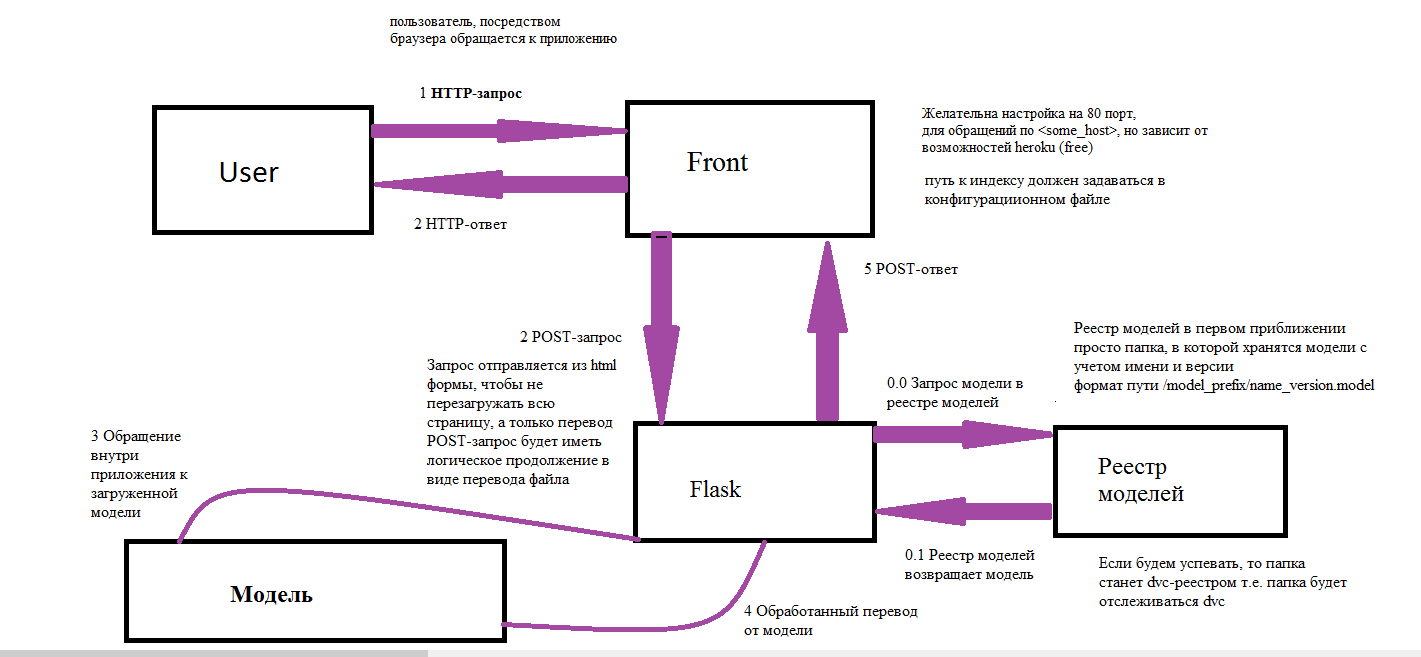
Структура проекта:

* Модуль REST API
* Модуль для тренировки новых моделей
* Модуль приложения translator
* Фронт

Верхнеуровнево архитектура приложения выглядит следующим образом:



Взаимодействие между компонентами ожидается следующее



### Модуль REST API

Содержит в себе REST API для внешних вызовов, в частности от модуля Фронт

На данный момент REST API имеет конечные точки:

* /v1/translate

Для /v1/translate тело запроса представлено в виде  
  
{

"source\_lang": <язык переводимого текста>,

"target\_lang": <язык, на который нужно перевести>

"text": <some text>

}

\*\_lang – **2** прописные буквы от принятого сокращения языка (RU, EN), файл с примерами в **docs/request\_format.txt**

# Возможные кейсы:

* Text – пустой
* Text – имеет очень большой объем (на данный момент перевод ограничивает 250 символами, чтобы обойтись без подключения gpu)
* 1 <= Текст <= 250 – штатный режим

# Формат ответа:

{

"code": <http-code>,

"status": <status перевода>,

"translated\_text": <переведенный текст>,

"comment": ""

}

* сode – http-код ответа
* status перевода – error/success
* translated\_text – переведенный текст
* comment – в случае статуса error должен присутствовать комментарий

Файл с примерами **docs/response-format.txt**

Возможные коды ответа:

* **200 OK** – пришел валидный запрос на сервер, получен валидный ответ,

работа в штатном режиме

* **400 Bad request –** пришел не поддерживаемый язык в source/target,

В данном случае должен прийти и заполненный **comment**

* **503 Service Unavailable –** ответа от модели в течение заданного таймаута не получили, считаем, что сервис не доступен
* **400 Bad request –** пришла слишком длинная строка (больше 250 символов)

Проверки осуществляются с обеих сторон (клиент/сервер) – в случае если добавиться новый клиент, то не будет новых проблем

Api оперирует некоторым классом **Predictor (интерфейс).**

Ожидаемый дизайн:

Class Predictor:

def \_\_init\_\_(self, path\_to\_model):

def predict(self, text: str) -> str:

pass

def evaluate(self, metric: str, …)->str:

pass

В процессе работы интерфейс может меняться.

**Predictor – внутри** работает с некоторым абстрактным классом BaseModel, конкретная реализация которого задается в настройках.

Predictor взят из модуля приложения Translator

### Модуль тренировки новых моделей

Ожидаемый результат – запуск тренировки произвольных моделей.

Запуск должен происходить примерно так

*python training/pipeline.py \*

*--stage=<указать этап, с которого должна начаться обработка> \*

*--model=<модель из списка диспетчера моделей> \*

*--path\_to\_train\_data=<путь до тренировочных данных> \*

*--output=<путь до сохраненной модели, включая имя модели> \*

*--path\_to\_yaml\_params=<путь до конфига модели с параметрами/batch/epochs> \*

*--device=<gpu/cpu/tpu>*

*На данном этапе пайплайн запускается скриптом, впоследствии эта ответственность ляжет на dvc, как трекера pipeline*

Требование к запуску экспериментов – результаты должны логироваться в mlflow, чтобы все участники команды могли видеть отслеживаемые результаты

Логируем:

1. Гиперпараметры
2. Параметры моделей
3. Размерность набора данных
4. Наименование признаков
5. Метрики

Артефакты пока не будем логировать в mlflow

Обработка может идти в несколько этапов

1. Скачать данных
2. Препроцессинг
3. Создание DataSet’a
4. Обучение
5. Сохранение результатов

Необязательно все этапы выполнять заново каждый раз, поэтому в *stage* передаем название этапа, с которого начинаем выполнение процесса

Модуль тренировки новых моделей использует абстрактный класс BaseModel

class BaseModel:

"""Base class, to be subclassed by predictors for specific type of data."""

def \_\_init\_\_(self):

self.name = f'{self.\_\_class\_\_.\_\_name\_\_}\_{dataset\_cls.\_\_name\_\_}\_{network\_fn.\_\_name\_\_}'

@property

def weights\_filename(self) -> str:

DIRNAME.mkdir(parents=True, exist\_ok=True)

return str(DIRNAME / f'{self.name}\_weights.h5')

def fit(self, dataset, batch\_size: int = 32, epochs: int = 10, augment\_val: bool = True, callbacks: list = None):

pass

def evaluate(self, x, y, batch\_size=16, verbose=False): # pylint: disable=unused-argument

pass

def loss(self): # pylint: disable=no-self-use

pass

def optimizer(self): # pylint: disable=no-self-use

pass

def metrics(self): # pylint: disable=no-self-use

pass

def load\_weights(self):

self.network.load\_weights(self.weights\_filename)

def save\_weights(self):

self.network.save\_weights(self.weights\_filename)

Остальные модели будут доопределять свои методы или переопределять текущие, т.к. образом,

Код будет универсальным для моделей, нужно будет только писать новый враппер модель для тренировки.

Так же в текущем модуле присутствует класс Predictor, который внутри себя также оперирует

BaseModel. На вход, реализации Predictor’a подаем путь к весам (задаем в файле конфигурации yaml)

## Модуль приложения translator

По-умолчанию, берем перевод eng2rus c базовой моделью

<https://huggingface.co/Helsinki-NLP/opus-mt-en-ru/tree/main#>,

также добавляем модель на основе трансформеров eng2de от huggingface

<https://huggingface.co/transformers/model_doc/t5.html#>

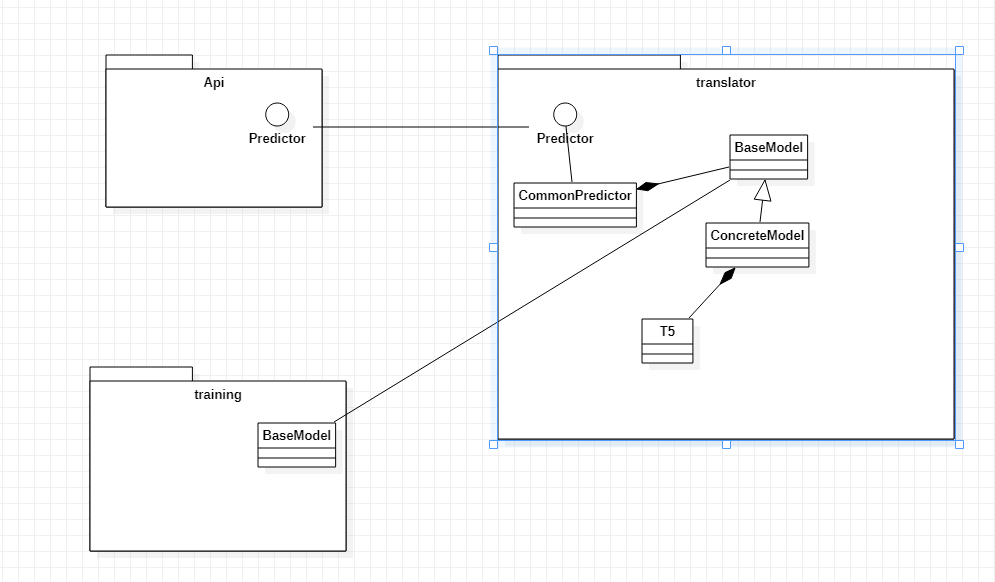
Необходимо определить базовый класс, основываясь на классе BaseModel,

Внутри используем сети соответствующие сети.

Во-второй итерации, нужно исследовать вопрос по сокращению ресурсов сети:

1. Исследовать дистилляцию в применении к используемым моделям

Видение основных общих частей, которые будут переиспользоваться



В рамках этого модуля, необходимо произвести исследование по вопросу качества модели.

1. Выбрать набор данных (5\_000+ экземпляров)
2. Исследовать имеющиеся метрики
3. Определиться с возможностью тестирования, основываясь на

подходах из https://github.com/RobustNLP/TestTranslation

1. Сравнить на данном наборе качество с гугл/яндекс/deepl по имеющейся метрике