

DS_PROD-1. Подготовка модели к продакшену и деплой

1. Введение

В ЭТОМ РАЗДЕЛЕ ВЫ:

Узнаете:

- о различных способах применения ML-моделей;
- таких понятиях, как изолированность, оркестрация и контейнеризация;
- больше о жизненном цикле ML-моделей и о том, как происходит их разработка в DS-командах на стадии выведения модели в production;
- как учесть требование воспроизводимости среды в DS-проекте;
- особенности монолитной и микросервисной архитектур, организации взаимодействия между сервисами;

Научитесь:

- формулировать требования продакшн-среды к ML-моделям;
- сохранять ML-модели в зависимости от этих требований;
- разрабатывать простейшие веб-сервисы для деплоя моделей на Flask;
- создавать контейнеры для своих моделей и работать с Docker и Docker-compose;
- оценивать качество моделей после внедрения в production.

МОДЕЛИ В ВАКУУМЕ НИКОМУ НЕ НУЖНЫ

Вы уже умеете создавать модели машинного обучения: они выдают какие-то предсказания, и вы научились оценивать их с помощью метрик качества. Вы можете составлять графики и делать выводы на основе данных. Но что дальше?

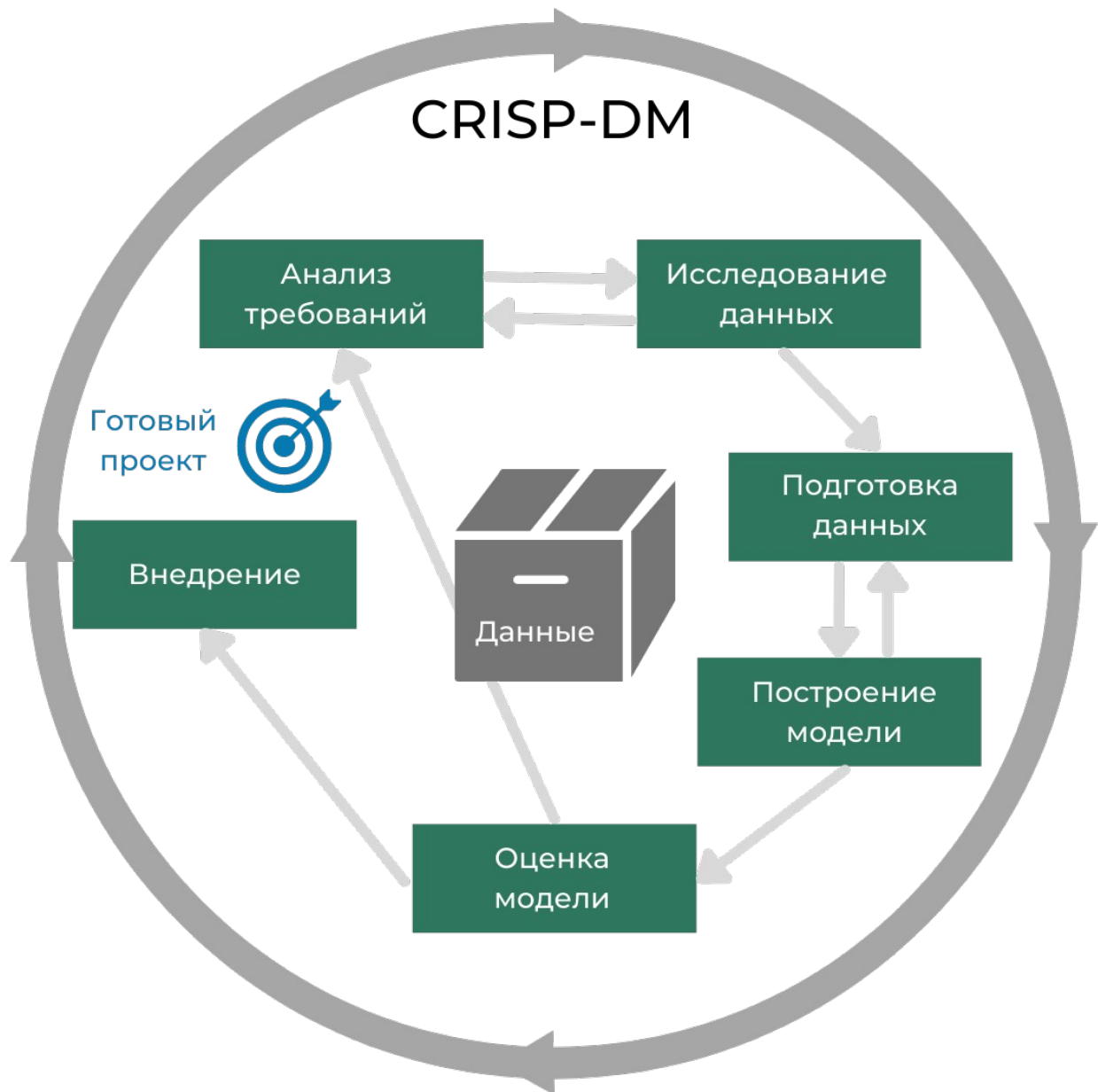
Важно задать себе следующие вопросы:

- Какими будут основные потребители результатов? Это будут другие сервисы? Или модель будет встроена в мобильное приложение?

- Как и в каком виде потребители ожидают получать результаты? По времени или по запросу? Если по запросу, то по каким правилам они будут его осуществлять?
- Какая для этого существует (или планируется) инфраструктура?

ВСПОМИНАЕМ CRISP-DM

Существует множество методологий для управления Data Science-проектами. Наиболее распространённой методологией разработки является знакомая нам модель Cross-Industry Standard Process for Data Mining, или CRISP-DM. Давайте ещё раз взглянем на её этапы:



Примечание. Если вы забыли, что из себя представляет методология разработки CRISP-DM, рекомендуем заглянуть в модуль [ML-1. «Теория машинного обучения»](#).

На **этапе внедрения** мы должны понять, будет ли происходить деградация модели во времени в связи с изменением распределений входных данных и возможно ли автоматизировать оценку качества, обновление моделей и их деплой.

Как учесть все эти сложности в процессе разработки моделей? Вот с этим мы и будем разбираться.

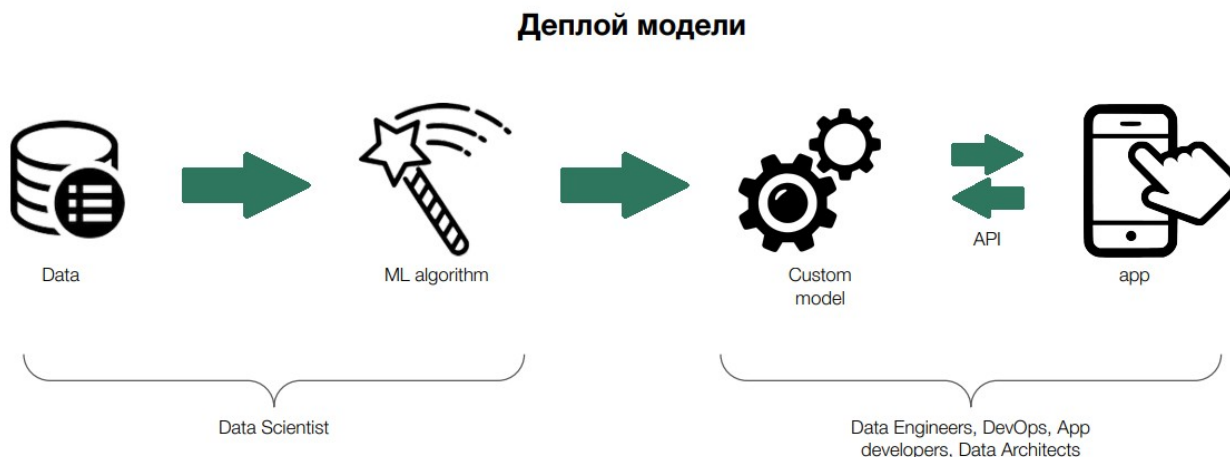
ДЕПЛОЙ МОДЕЛИ

Задача развёртывания приложения на сервере называется **деплой** (от англ. *deployment* — внедрение, развёртывание). По сути, это процесс трансформации исходного кода вашего приложения в рабочее состояние на конкретном сервере.

Для того чтобы внедрить модель в продакшн (в минимальном варианте), необходимо:

1. Сохранить обученную модель в файл.
2. Поднять сервер.
3. Доставить и запустить на нём свою модель.

Звучит довольно просто, не так ли?



К сожалению, при моделировании зачастую не учитываются вопросы, связанные с деплоем модели и её внедрением в уже существующие системы. Для чего мы обучили модель? Может быть, мы будем использовать её для чат-бота? Может, она должна быть встроена в приложение для iPhone? Как часто потребуется переобучать модель?

Опыт подсказывает, что если модель создаётся «в вакууме», то в итоге её просто нельзя будет вывести в продакшн.

Дополнительной проблемой является то, что большинство библиотек для машинного обучения сфокусированы именно на **обучении**, а не на **предсказании**.

Кроме того, в крупных компаниях дата-сайентисты, дата-инженеры, ML-инженеры, занимающиеся внедрением моделей в продакшн, могут быть не просто разными людьми, но и разными командами. На практике может оказаться так, что готовая обученная модель будет внедрена другой командой на другом языке программирования. При этом обычным

разработчикам модели машинного обучения могут представляться непрозрачными загадочными чёрными ящиками.

В этом модуле мы будем учиться подготавливать модели к внедрению в продакшн и разворачивать собственный веб-сервис. Как вы увидите дальше, код, который мы написали в Jupyter Notebook, практически никогда не попадает в продакшн без изменений.

ЦЕЛИ ЭТОГО МОДУЛЯ:

- Понять, что такое инференс модели и как организуется сохранение и загрузка модели.
- Узнать, что такое сериализация и десериализация и научиться их различать.
- Понять, что делать, если итоговый проект реализован на другом языке программирования, а ваша модель обучена на Python.
- Рассмотреть вопросы сетевого взаимодействия и узнать о настройке взаимодействия между серверами по сети.
- Закрепить умение писать запросы к серверу (мы затрагивали эту тему в модуле [PY-17. «Как получать данные из веб-источников и API»](#), но теперь веб-источник мы будем создавать сами).
- Разобрать основные отличия и области применения фреймворков для разработки веб-сервисов (Django, FastAPI и, конечно, Flask).
- Научиться использовать фреймворк Flask для реализации простейших сервисов моделей.
- Познакомиться с инструментами uWSGI и NGINX, которые помогают повысить пропускную способность и производительность сервера.

2. Сохранение и загрузка моделей: pickle и joblib

Представьте ситуацию: вы садитесь за руль автомобиля, и... вам нужно заново учиться водить! Сложная ситуация, не правда ли? В реальности мы просто заводим машину, включаем передачу и начинаем движение.

Примерно того же продакшн-среда требует от моделей. И это понятно, ведь намного дешевле с точки зрения расходования ресурсов хранить данные в виде готовой модели, чем каждый раз заново обучать модель на сервере из восьми видеокарт.

Именно поэтому код, который был написан для обучения модели и оценки её качества, крайне редко используется для **инференса** (от англ. inference — вывод). Так называется непрерывная работа алгоритма машинного обучения в конечном приложении. По этой причине при внедрении моделей в продакшн их принято сохранять в готовом виде, то есть уже обученными и готовыми решать реальные задачи.



Изучив материалы в этом юните, вы сможете:

1. Подробно разобраться с тем, как с помощью библиотеки `pickle` сохранять обученные модели машинного обучения, загружать их обратно и заново обращаться к ним для предсказаний.
2. Узнать о способах передачи моделей для внедрения на языках программирования, отличных от Python.
3. Разобраться с темой сериализации.

Совет. Обязательно попробуйте воспроизвести примеры, которые мы будем разбирать далее.

СЕРИАЛИЗАЦИЯ И ДЕСЕРИАЛИЗАЦИЯ

Как и почти всё в языке программирования Python, обученная модель является **объектом**. Этот объект не простой, поскольку модель содержит сложную иерархию классов — в каждом классе есть набор полей, ссылающихся на объекты других классов, и так далее.

Например, объект класса `RandomForestClassifier` из библиотеки `sklearn` содержит множество полей, часть из которых устанавливается во время инициализации модели (максимальная глубина, количество деревьев в ансамбле, критерий информативности и т. д.), а часть определяется во время обучения модели (последовательность предикатов внутри каждого дерева в ансамбле, значимость признаков). Наша задача — «законсервировать» этот объект (модель), сохранив значение всех полей, которые мы задали при инициализации объекта и получили по итогам обучения. То есть мы должны сохранить модель, включая её внешние и внутренние параметры.

Чтобы гарантировать сохранение всей структуры данных и получить её при загрузке обратно, используется сериализация.

Сериализация — это процесс трансформации любой структуры данных, поддерживаемой в языке, в последовательность битов (или байтов). Обратной операцией является десериализация.



ЗАЧЕМ ЭТО НУЖНО?

То, в каком виде данные записаны на диск, зачастую может сильно отличаться от того, в каком виде они существуют в памяти программы. И там, и там это будут байты информации, но в файле будет представлена просто их последовательность, а в памяти программы это может быть какой-то объект или даже несколько объектов со структурой.

Приведём простой пример с форматом CSV. Пусть у нас есть файл с одной строкой, где несколько слов записаны через запятую. На диске это будет записано просто как последовательность байтов.

Однако когда мы считываем эту последовательность внутри программы, мы хотим работать с ней как со **списком строк**, то есть нам нужно не только считать эти данные, но и применить к ним некоторое преобразование:

```
line = 'word1, word2, word3'
line.split(",")
## ['word1', 'word2', 'word3']
['word1', ' word2', ' word3']
```

Можно заметить, что после этого преобразования появилось много объектов (каждое слово списка) и некоторая структура (то, как слова расположены в этом списке). Процесс, который мы только что выполнили, называется **десериализацией**.

Также возможен обратный процесс, когда мы хотим сохранить в виде последовательности байтов какие-то данные из памяти программы. Например, нам нужно снова получить начальную строку:

```
", ".join(['word1', 'word2', 'word3'])
## word1,word2,word3
'word1,word2,word3'
```

Этот процесс как раз и называется **сериализацией**.

Мы разобрали простейший пример, но на практике всё гораздо сложнее. Сериализовывать можно не только в текст (как с CSV, JSON и подобными форматами), но и в **бинарный** формат, который человек не сможет прочитать.

Бывают форматы, которые могут описывать более сложные структуры (тот же JSON). Также можно добавить сжатие итогового набора битов.

Заметим, что программа должна потратить некоторый ресурс CPU, чтобы преобразовать объект в набор байтов и наоборот.

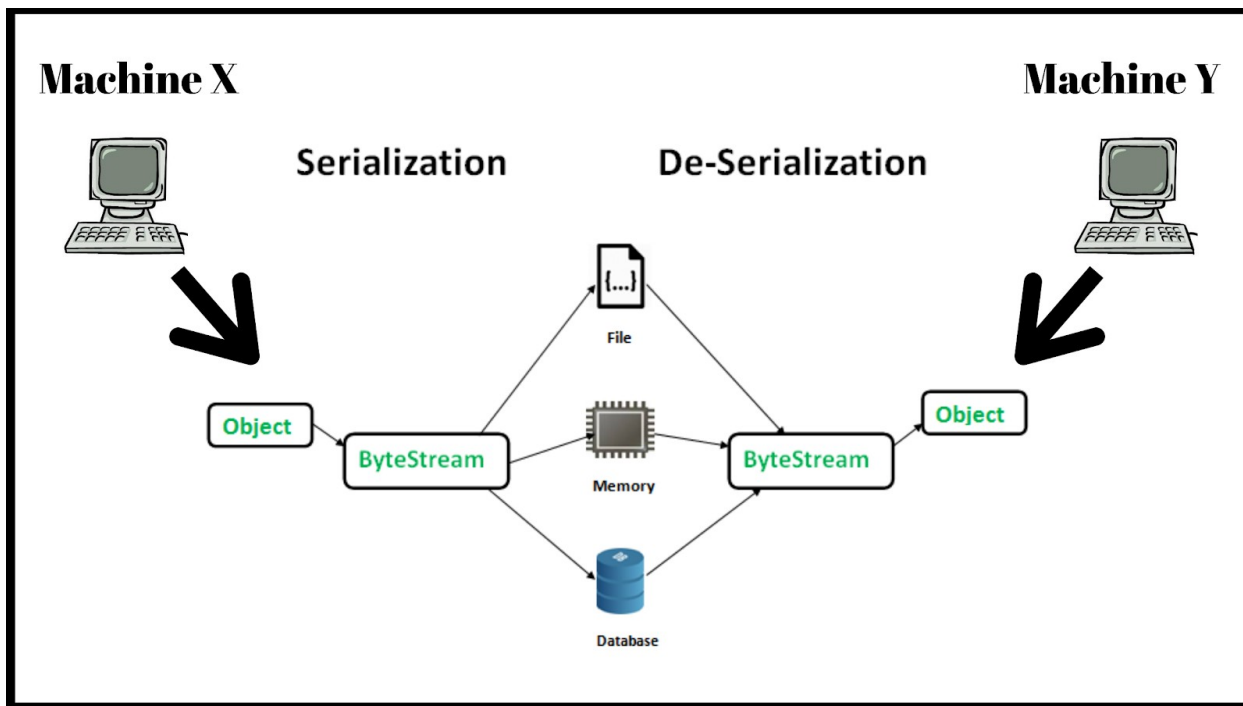
ИНСТРУМЕНТЫ СЕРИАЛИЗАЦИИ: PICKLE

Посмотрим, с помощью каких средств происходит сериализация объектов в Python.

В стандартную библиотеку Python входит модуль [pickle](#), который служит для сериализации почти всех объектов произвольного типа.

Мы помним, что объекты находятся в оперативной памяти и направляются в байтовые потоки ввода-вывода. В байтовые потоки может быть направлен любой файлоподобный объект.

В ходе десериализации исходный объект воссоздаётся в оперативной памяти с теми же самыми значениями, но с новой идентичностью — новым адресом в памяти.



Обратите внимание на предупреждение в официальной документации:

⚠ Warning. The pickle module is not secure. Only unpickle data you trust.

Так как законсервирован может быть абсолютно любой объект, в нём могут быть «спрятаны» различные вредоносные программы или данные. Поэтому будьте внимательны и не проводите десериализацию бинарных файлов, в происхождении которых вы не уверены.

Не переживайте: файлы, предлагаемые для десериализации в рамках нашего курса, безопасны.

Для иллюстрации работы модуля pickle последовательно пройдемся по всем шагам и разберемся с тем, как сериализовать уже обученные алгоритмы машинного обучения.

ШАГ №1

Обучим модель линейной регрессии на встроенном датасете о диабете — Diabetes dataset.

В данном датасете представлены десять исходных признаков: возраст, пол, индекс массы тела, среднее артериальное давление и шесть измерений сыворотки крови были получены для каждого из 442 пациентов с сахарным диабетом. Интерес представляет количественный показатель прогресса заболевания, замеренный через год после исходного измерения. Тип задачи — регрессия.

Примечание. Для простоты и наглядности мы опустим процесс предобработки данных, разведывательный анализ, разделение выборки на обучающую и тестовую, валидацию модели и подбор гиперпараметров, так как для наших целей это неважно. Мы уверены, что вы уже способны произвести эти этапы самостоятельно.

В качестве модели, прогнозирующей целевую переменную, возьмём простейшую линейную регрессию, и обучим её на исходных данных:

```
from sklearn.linear_model import LinearRegression

from sklearn.datasets import load_diabetes

# Загружаем датасет о диабете
X, y = load_diabetes(return_X_y=True)
# Инициализируем модель линейной регрессии
regressor = LinearRegression()
# Обучаем модель
regressor.fit(X,y)

## LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,
normalize=False)

LinearRegression()
```

В результате выполнения кода получился объект класса LinearRegression, на который ссылается переменная regressor. При этом атрибуты объекта (веса модели линейной регрессии) были сформированы во время обучения. То есть объект regressor теперь является обученной моделью.

ШАГ №2

Далее, когда мы получили обученную модель, нам необходимо сериализовать её, превратив объект Python в поток байтов. Для этого импортируем модуль pickle и воспользуемся функцией dumps(), в которую нужно передать объект Python.

```
import pickle

# Производим сериализацию обученной модели
```



```

model = pickle.dumps(regressor)

print(type(model))
print(type(regressor))
## bytes
## sklearn.linear_model._base.LinearRegression

<class 'bytes'>
<class 'sklearn.linear_model._base.LinearRegression'>

```

Как видим, мы создали объект model типа bytes.

ШАГ №3

Давайте попробуем восстановить (десериализовать) объект Python. Для этого в модуле pickle есть функция loads(), в которую нужно передать сериализованный объект (поток байтов).

```

# Производим десериализацию
regressor_from_bytes = pickle.loads(model)
regressor_from_bytes
## LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,
normalize=False)

LinearRegression()

```

В результате десериализации мы смогли восстановить исходный объект (модель).

ШАГ №4

Сохраним сериализованный объект прямо в файл. Для этого в pickle есть функция dump() (без s на конце). В неё необходимо передать имя файла или ссылку на открытый файл. Файл назовём myfile, его расширение — .pkl (формат данных pickle):

```

# Производим сериализацию и записываем результат в файл формата pkl
with open('myfile.pkl', 'wb') as output:
    pickle.dump(regressor, output)

```

Теперь у нас есть бинарный файл с готовой моделью, и мы можем передать его, например, ML-инженерам, которые будут заниматься деплоем модели на сервер.

ШАГ №5

Посмотрим на код, который восстанавливает (десериализует) обученную модель из файла myfile.pkl. Для этого в pickle есть функция load() (без s на конце). В неё необходимо передать имя файла или ссылку на открытый файл.

```

# Производим десериализацию и извлекаем модель из файла формата pkl
with open('myfile.pkl', 'rb') as pkl_file:
    regressor_from_file = pickle.load(pkl_file)

```

```
regressor_from_file
## LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,
normalize=False)
LinearRegression()
```

ШАГ №6

Убедимся, что методы и результаты предсказаний обученной модели и модели, загруженной из файла, совпадают:

```
# Проверяем, что все элементы массивов предсказаний совпадают между собой
all(regressor.predict(X) == regressor_from_bytes.predict(X))
## True

True

all(regressor.predict(X) == regressor_from_file.predict(X))
## True

True
```

Как видите, исходная и восстановленная из байтов и файла модели дают одинаковые предсказания. Это значит, что теперь мы можем импортировать наши обученные модели в любое Python-приложение и пользоваться ими, минуя этап обучения и все этапы, предшествующие ему.

ОГРАНИЧЕНИЯ

Как мы упоминали, у pickle есть ограничения. Например, мы не можем сериализовать лямбда-функции. Давайте посмотрим, что нам вернёт следующий код:

```
my_lambda = lambda x: x*2
with open('my_lambda.pkl', 'wb') as output:
    pickle.dump(my_lambda, output)

PicklingError: Can't pickle at 0x000002C7B7951940>: attribute lookup
on __main__ failed
```

Совет. В таких случаях лучше пользоваться пакетом [dill](#).

СОХРАНЕНИЕ ПАЙПЛАЙНА

Ранее мы рассмотрели простейший пример сериализации готовой модели.

У вас мог возникнуть вопрос: что делать, если перед подачей данных в модель их необходимо предобработать, например произвести стандартизацию, исключить неинформативные признаки? Неужели придётся прописывать все эти шаги в коде

инференса модели? А что если вопросами инференса занимаются совершенно другие специалисты, которые вообще ничего не знают о машинном обучении и не умеют производить предобработку данных?

Конечно, мы должны передать результаты в таком виде, чтобы ими можно было воспользоваться без лишних манипуляций.

Мы уже упоминали, что pickle работает с любыми объектами Python. Поэтому для сохранения может быть доступна не просто обученная модель, но и целый **пайплайн**, включающий предобработку данных.

Примечание. Если вы забыли, что такое пайплайны и как их формировать с помощью библиотеки sklearn, рекомендуем заглянуть в модуль [ML-8. «Продвинутые методы машинного обучения»](#).

Например, мы хотим сериализовать пайплайн, который включает в себя min-max-нормализацию и отбор пяти наиболее важных факторов на основе корреляции Пирсона. Полученные в результате данные отправляются на вход модели линейной регрессии.

```
import pickle

from sklearn.linear_model import LinearRegression
from sklearn.datasets import load_diabetes
from sklearn.feature_selection import SelectKBest, f_regression
from sklearn.preprocessing import MinMaxScaler
from sklearn.pipeline import Pipeline

# Загружаем датасет о диабете
X, y = load_diabetes(return_X_y=True)

# Создаём пайплайн, который включает нормализацию, отбор признаков и
обучение модели
pipe = Pipeline([
    ('Scaling', MinMaxScaler()),
    ('FeatureSelection', SelectKBest(f_regression, k=5)),
    ('Linear', LinearRegression())
])

# Обучаем пайплайн
pipe.fit(X, y)

Pipeline(steps=[('Scaling', MinMaxScaler()),
                 ('FeatureSelection',
                  SelectKBest(k=5,
                              score_func=<function f_regression at
0x000001513BB30F40>)),
                 ('Linear', LinearRegression())])
```

Пайплайн обучен. Давайте сохраним его в файл с помощью pickle:

```
# Сериализуем pipeline и записываем результат в файл
with open('my_pipeline.pkl', 'wb') as output:
    pickle.dump(pipe, output)
```

Если сериализация завершилась успешно, то при инференсе модели мы сможем восстановить её из файла:

```
# Десериализуем pipeline из файла
with open('my_pipeline.pkl', 'rb') as pkl_file:
    loaded_pipe = pickle.load(pkl_file)
```

Проверим, что результаты исходного и десериализованного пайплайнов идентичны:

```
# Сравниваем предсказания исходного и восстановленного пайплайнов
print(all(pipe.predict(X) == loaded_pipe.predict(X)))

## True

True
```

Примечание. Если мы хотим сохранять сериализованные пайплайны в виде потока байтов, нужно использовать функции `dumps()` и `loads()`, а не `dump()` и `load()`.

Однако в процессе предобработки могут возникнуть шаги, которые нельзя реализовать стандартными методами `sklearn`. Например, для решения многих задач в нашем курсе мы часто использовали `feature engineering`, чтобы повысить качество работы моделей. Как встроить этот шаг в исходный пайплайн?

Для этого в `sklearn` можно организовать так называемые **кастомные трансформеры**. Такой трансформер должен наследоваться от двух классов: `TransformerMixin` и `BaseEstimator`.

Посмотрим на шаблон кастомного трансформера:

```
from sklearn.base import TransformerMixin, BaseEstimator

class MyTransformer(TransformerMixin, BaseEstimator):
    '''Шаблон кастомного трансформера'''

    def __init__(self):
        '''
        Здесь прописывается инициализация параметров, не зависящих от
        данных.
        '''
        pass

    def fit(self, X, y=None):
```

```
    Здесь прописывается «обучение» трансформера.  
    Вычисляются необходимые для работы трансформера параметры  
    (если они нужны).  
    '''
```

```
    return self  
  
def transform(self, X):  
    '''  
    Здесь прописываются действия с данными.  
    '''  
    return X
```

- `__init__()` — метод, который вызывается при создании объекта данного класса. Он предназначен для инициализации исходных параметров. Например, у трансформера для создания полиномиальных признаков `PolynomialFeatures` из `sklearn` в методе `__init__()` параметр `degree` задаёт степень полинома.
- `fit()` — метод, который вызывается для «обучения» трансформера. Он должен возвращать ссылку на сам объект (`self`). Например, в трансформере `StandardScaler` в методе `fit()` прописано вычисление среднего значения и стандартного отклонения в каждом столбце таблицы, переданной в качестве параметра метода `fit()`.
- `transform()` — метод, который трансформирует приходящие на вход данные. Он должен возвращать преобразованный массив данных. Например, при вызове метода `transform()` у `StandardScaler` из `sklearn` внутри происходит преобразование — вычитание из каждого столбца среднего и деление результата на стандартное отклонение. Причём среднее и стандартное отклонение вычисляются заранее в методе `fit()`.

Примечание. Как мы знаем, у некоторых трансформеров из `sklearn`, например у того же `MinMaxScaler`, есть ещё и метод `fit_transform()`, который является комбинацией методов `fit()` и `transform()`.

Наш трансформер пока что ничего не делает. Предположим, мы хотим генерировать в данных новый признак, который является простым произведением первых трёх столбцов таблицы. Давайте пропишем в методе `transform()` эти действия.

Для работы такого трансформера нужны только исходные данные без дополнительных параметров, поэтому методы `__init__()` и `fit()` остаются без изменений.

```
import numpy as np  
  
class MyTransformer(TransformerMixin, BaseEstimator):  
    '''Шаблон кастомного трансформера'''  
  
    def __init__(self):  
        '''Здесь прописывается инициализация параметров, не зависящих  
        от данных.'''
```

```

pass

def fit(self, X, y=None):
    """
    Здесь прописывается «обучение» трансформера.
    Вычисляются необходимые для работы трансформера параметры
    (если они нужны).
    """
    return self

def transform(self, X):
    """Здесь прописываются действия с данными."""
    # Создаём новый столбец как произведение первых трёх
    new_column = X[:, 0] * X[:, 1] * X[:, 2]
    # Для добавления столбца в массив нужно изменить его размер на
    (n_rows, 1)
    new_column = new_column.reshape(X.shape[0], 1)
    # Добавляем столбец в матрицу измерений
    X = np.append(X, new_column, axis=1)
    return X

```

Посмотрим, как работает наш кастомный трансформер. Создадим объект трансформера, вызовем метод `transform` и посмотрим на результирующий размер таблицы.

```

# Инициализируем объект класса MyTransformer (вызывается метод
__init__)
custom_transformer = MyTransformer()
# Чисто формально вызываем метод fit, но у нас он ничего не делает
custom_transformer.fit(X)
# Трансформируем исходные данные (вызывается метод transform)
X_transformed = custom_transformer.transform(X)
print('Shape before transform: {}'.format(X.shape))
print('Shape after transform: {}'.format(X_transformed.shape))

## Shape before transform: (442, 10)
## Shape after transform: (442, 11)

Shape before transform: (442, 10)
Shape after transform: (442, 11)

```

Видно, что в результате трансформации в исходную матрицу наблюдений добавился новый столбец.

Теперь давайте встроим этот трансформер в сам пайплайн — для этого достаточно добавить новый шаг в пайплайн.

```

# Создаём пайплайн, который включает Feature Engineering,
# нормализацию, отбор признаков и обучение модели
pipe = Pipeline([
    ('FeatureEngineering', MyTransformer()),
    ('Scaling', MinMaxScaler()),
    ('FeatureSelection', SelectKBest(f_regression, k=5)),
    ('Linear', LinearRegression())
])

# Обучаем пайплайн
pipe.fit(X, y)

Pipeline(steps=[('FeatureEngineering', MyTransformer()),
                 ('Scaling', MinMaxScaler()),
                 ('FeatureSelection',
                  SelectKBest(k=5,
                              score_func=<function f_regression at
0x000001513BB30F40>)),
                 ('Linear', LinearRegression())])

```

Наконец можно сериализовать полученный pipeline:

```

# Сериализуем pipeline и записываем результат в файл
with open('my_new_pipeline.pkl', 'wb') as output:
    pickle.dump(pipe, output)

```

Теперь мы можем передать пайплайн и воспользоваться им для инференса, предварительно произведя десериализацию.

Задание 2.5

Десериализуйте полученный pipeline с добавленным в него кастомной трансформации из файла. Затем предскажите значение целевой переменной для наблюдения, которое описывается следующим вектором:

```

features = np.array([[ 0.00538306, -0.04464164,  0.05954058, -
0.05616605,  0.02457414, 0.05286081, -0.04340085,  0.05091436, -
0.00421986, -0.03007245]])

# Десериализуем pipeline
with open('my_new_pipeline.pkl', 'rb') as new_pkl_file:
    new_loaded_pipe = pickle.load(new_pkl_file)

new_loaded_pipe.predict(features)[0].round()

173.0

```

В дополнение предлагаем вам ознакомиться с примерами использования пайплайнов (на англ. языке):

- [Sklearn Pipeline Tutorial](#);
- [Pipeline Guide \(GitHub\)](#);
- [A Simple Guide to Scikit-learn Pipelines](#).

БИБЛИОТЕКА JOBLIB

Как мы видим, pickle прекрасно справляется со своей задачей: мы можем сериализовать и восстанавливать любые Python-объекты, включая модели и даже пайплайны. Однако иногда массивы данных, на которых обучаются модели, бывают настолько большими, что после загрузки из pickle невозможно восстановить объект полностью.

В таких случаях вместо pickle лучше использовать библиотеку `joblib`. Этот модуль более эффективен и надёжен для работы с объектами, которые содержат большие массивы данных. Пожалуй, единственный минус этого модуля в том, что он может «консервировать» только в файл, поэтому вы не сможете получить объект в виде бинарной строки и работать с ним. В модуле попросту отсутствуют методы для работы с бинарной строкой. Формат файлов для сохранения — `.joblib`.

В остальном работа с `joblib` полностью идентична работе с `pickle`: после обучения модели производим сериализацию с помощью функции `dump()`, а в коде самого приложения, где нужно использовать модель, выполняем десериализацию с помощью функции `load()`. В каждую из этих функций необходимо передать путь до файла для записи и чтения соответственно.

Для иллюстрации работы сохраним полученную линейную регрессию:

```
import joblib

# Загружаем датасет о диабете
X, y = load_diabetes(return_X_y=True)
# Обучаем модель линейной регрессии
regressor = LinearRegression()
regressor.fit(X, y)
# Производим сериализацию и сохраняем результат в файл формата .joblib
joblib.dump(regressor, 'regr.joblib')

## ['regr.joblib']

['regr.joblib']
```

Загрузим файл заново (загрузка может быть произведена в другом файле с кодом):

```
# Десериализуем модель из файла
clf_from_joblib = joblib.load('regr.joblib')
# Сравниваем предсказания
all(regressor.predict(X) == clf_from_joblib.predict(X))

## True

True
```


3. Практика: pickle



В первом практическом юните этого модуля мы будем учиться работать с модулем pickle.

Ваш коллега Василий обучил модель и теперь просит вас проверить её на ваших данных. Он присылает вам pickle-файл. Загрузите [модель](#), используя модуль pickle.

Задание 3.1

При загрузке вывелся секретный код. Введите его в поле ниже.

```
model = joblib.load('data\model.pkl')

secret word: skillfactory
how is this possible? answer is here: https://youtu.be/xm-A-h9QkXg

c:\Users\Home\AppData\Local\Programs\Python\Python311\Lib\site-
packages\sklearn\base.py:318: UserWarning: Trying to unpickle
estimator LinearRegression from version 1.0.2 when using version
1.2.2. This might lead to breaking code or invalid results. Use at
your own risk. For more info please refer to:
https://scikit-learn.org/stable/model\_persistence.html#security-
maintainability-limitations
  warnings.warn(
```

Задание 3.2

Проверьте, объект какого типа получился. Какую модель вам прислал коллега?

```
model

LinearRegression(positive=True)
```

Задание 3.3

Теперь необходимо применить модель. Сделайте предсказание для следующего набора фичей: [1, 1, 1, 0.661212487096872]. Введите результат, предварительно округлив его до трёх знаков после точки-разделителя.

```
model.predict([[1, 1, 1, 0.661212487096872]])  
array([0.666])
```

У присланной вам модели есть два поля (атрибута) с именами `a` и `b`. Создайте из них словарь с такими же именами ключей и значениями, а затем сохраните его в файл с помощью модуля `pickle`.

Чтобы вы могли проверить правильность решения задания, мы создали специальный проверочный скрипт. Скачайте его [здесь](#).

Сохраните его рядом с вашим `pickle`-файлом (в той же папке) и запустите, передав первым аргументом имя файла. Если вы всё сделали правильно, на экран выведется ответ для следующего задания.

КАК ЗАПУСТИТЬ СКРИПТ?

В ячейке Jupyter Notebook:

```
!python hw1_check_ol.py имя_pickle_файла.pkl
```

В терминале:

```
cd имя_папки_со_скриптом
```

```
python hw1_check_ol.py имя_pickle_файла.pkl
```

```
# Сериализуем pipeline и записываем результат в файл  
with open('model_dict.pkl', 'wb') as output:  
    pickle.dump({'a': model.a,  
                'b': model.b}, output)  
  
import os  
  
os.chdir('data')  
  
!python hw1_check_ol.py model_dict.pkl  
( 'secret code 2:', '3c508' )  
  
os.chdir('..')
```

4. Сохранение и загрузка моделей: PMML и ONNX-ML

Среда или требования к **инференсу** модели для вашего проекта могут быть устроены так, что потребуют реализации на языке программирования, отличном от Python. Например, если компания разрабатывает десктопное приложение, то для внедрения модели её потребуется «перевести» на Java или C++. Как это сделать?

PREDICTIVE MODEL MARKUP LANGUAGE

В таких случаях используется генерация файла формата PMML (Predictive Model Markup Language).

PMML — это XML-диалект, который применяется для описания статистических и DS-моделей. PMML-совместимые приложения позволяют легко обмениваться моделями данных между собой. Разработка и внедрение PMML осуществляется IT-консорциумом Data Mining Group.

Подробнее с PMML можно ознакомиться на [официальном сайте](#).

К сожалению, далеко не все библиотеки для машинного обучения (в том числе sklearn) поддерживают возможность сохранения обученной модели в указанном формате. Однако для этого можно использовать сторонние библиотеки, и одной из самых популярных является [Nyoka](#).

Давайте сохраним модель из предыдущего блока в формат PMML.

Для установки можно использовать систему управления пакетами pip:

```
# !pip3 install nyoka
```

или

```
# pip install nyoka
```

Рассмотрим пример работы с библиотекой:

```
from nyoka import skl_to_pmml
from sklearn.preprocessing import MinMaxScaler
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LinearRegression
from sklearn.datasets import load_diabetes

X, y = load_diabetes(return_X_y=True)
cols = load_diabetes()['feature_names']

scaler = MinMaxScaler()
pipe = Pipeline([
    ('Scaling', MinMaxScaler()),
    ('Linear', LinearRegression())
])
# Обучение пайплайна, включающего линейную модель и нормализацию признаков
pipe.fit(X, y)
# Сохраним пайплайн в формате pmml в файл pipeline.pmml
skl_to_pmml(pipeline=pipe, col_names=cols,
            pmml_f_name="pipeline.pmml")
```

Итак, мы построили пайплайн обработки данных и обучили модель линейной регрессии. После этого мы с помощью функции `skl_to_pmml` сохранили модель в файл `pipe.pmml`.

Откройте файл `pipe.pmml` с помощью любого текстового редактора.

Давайте рассмотрим этот файл подробнее:

- Секция содержит информацию о признаках, включая наименование и тип данных, используемых для построения модели.
- Секция содержит информацию о необходимых преобразованиях для каждого признака. Обратите внимание, что в этом блоке также содержится информация для трансформации. Так как мы использовали `MinMaxScaler()`, то в файле записаны минимальное и максимальное значения.

Таким образом, в файле содержится вся информация для того, чтобы пайплайн можно было использовать на любом языке программирования.

OPEN NEURAL NETWORK EXCHANGE

В разработке моделей на основе нейронных сетей сегодня наиболее распространён формат ONNX (Open Neural Network Exchange).

[ONNX \(Open Neural Network Exchange\)](#) — это открытый стандарт для обеспечения совместимости моделей машинного обучения. Он позволяет разработчикам искусственного интеллекта использовать модели с различными инфраструктурами, инструментами, средами исполнения и компиляторами.

Стандарт совместно поддерживается компаниями Microsoft, Amazon, Facebook и другими партнёрами как проект с открытым исходным кодом.

Часто стандарт ONNX и его библиотеки используют для конвертации из одного фреймворка в другой (например, из PyTorch в TensorFlow для использования в продакшене). Для конвертации различных фреймворков (не только DL) в формат ONNX и обратно существует [ряд библиотек](#):

- [ONNX-Tensorflow](#);
- [Tensorflow-ONNX](#);
- [Keras-ONNX](#);
- [Sklearn-ONNX](#).
- и другие.

Также в рамках стандарта ONNX есть инструмент ONNX-runtime. Он служит для ускорения инференса Python-моделей, а также инференса на других языках, например Java, C++.

```
import onnxruntime as rt
from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from skl2onnx import __1__
from skl2onnx.common.data_types import __2__
```

```

# загружаем данные
X, y = load_boston(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y,
random_state=7)
print(X_train.shape, X_test.shape)

# обучаем модель
model = LinearRegression()
model.fit(__3__, y_train)

# делаем инференс модели на тесте
test_pred = model.predict(__4__)
print('sklearn model predict:\n', test_pred)

# конвертируем модель в ONNX-формат
initial_type = [('float_input', __5__([None, __6__]))]
model_onnx = __7__(model, initial_types=initial_type)

# сохраняем модель в файл
with open("model.onnx", "wb") as f:
    f.write(model_onnx.SerializeToString())

# Делаем инференс на тесте через ONNX-runtime
sess = rt.__8__("model.onnx")
input_name = sess.get_inputs()[0].name
label_name = sess.get_outputs()[0].name
test_pred_onnx = sess.run([label_name],
                        {input_name: X_test.astype(np.float32)})[0].reshape(-
1)
print('onnx model predict:\n', test_pred_onnx)

```

5. Деплой модели. Протоколы сетевого взаимодействия

Помните вашего коллегу Василия — того, что прислал модель в pkl-формате и просил проверить её на ваших данных? Вот, кстати, и [ссылка на модель](#).

Завтра Василию предстоит защищать решение перед руководителями проекта, а для этого нужно показать работающий прототип (в нашем случае — **задеплоить модель на тестовый сервер**) и рассказать об эффективности предлагаемого решения. За деплоем Василий обратился к вам.

С чего начнём?

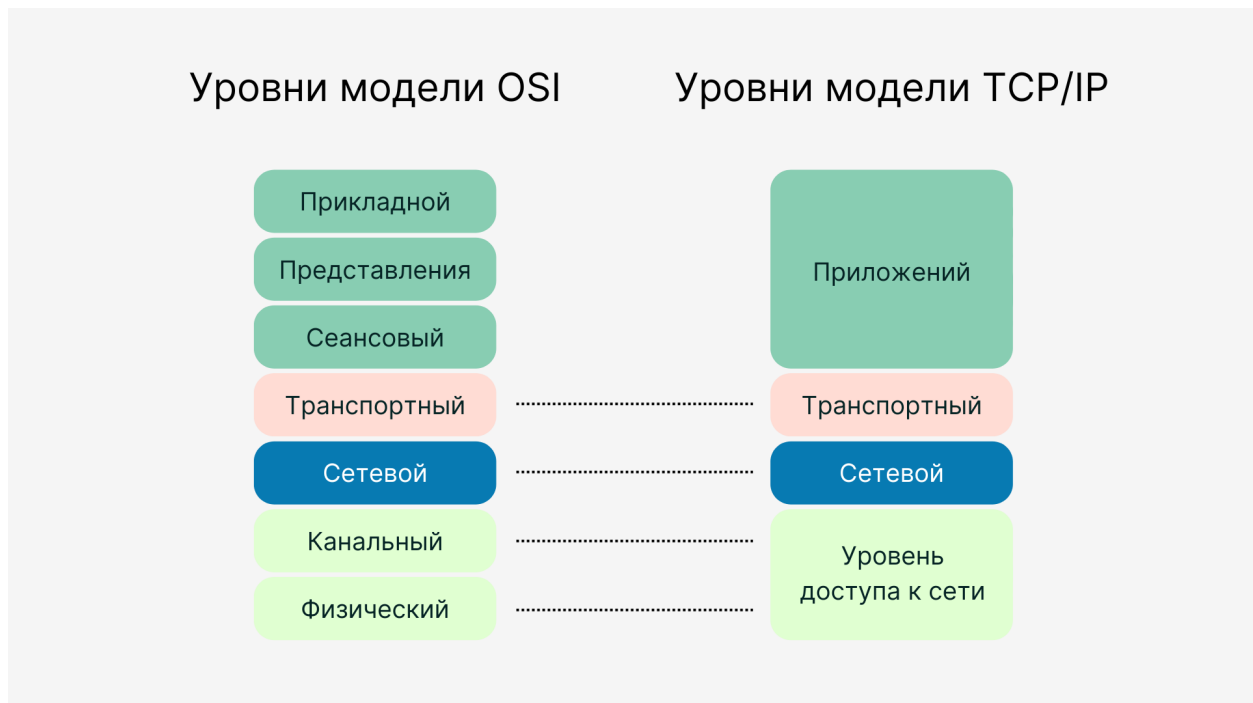
- Во-первых, нужно разобраться с тем, как происходит взаимодействие серверов по сети.

- Во-вторых, необходимо узнать, как написать сервер и обернуть в него модель. Какие есть фреймворки? Какой выбрать конкретно в нашем случае? Как его реализовать?

МОДЕЛИ СЕТЕВОГО ВЗАИМОДЕЙСТВИЯ

Начнём с первого вопроса и немного поговорим о том, как происходит взаимодействие между серверами по сети, то есть разберём процесс обмена информацией между компьютерами.

Наиболее известные модели сетевого взаимодействия — [OSI](#) и [TCP/IP](#).



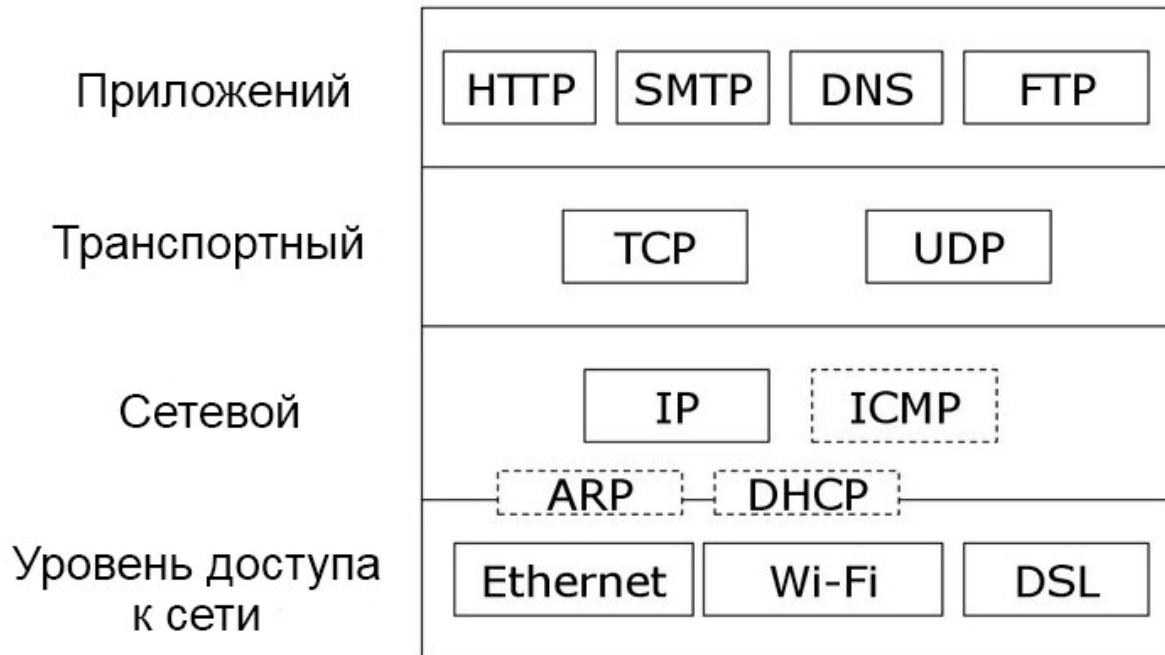
Эти модели распределяют сетевые протоколы по разным уровням взаимодействия. Что такое протокол?

Вообще **протокол** — это некоторый набор правил, определяющий принципы взаимодействия устройств в сети. В нашем случае это правила, по которым программа, получив по сети набор битов, понимает, как его прочесть и что он значит.

Для того чтобы обмен информацией между устройствами проходил успешно, все устройства (участники процесса) должны следовать условиям протокола. В сети поддержка протоколов встраивается или в аппаратную (в «железо»), или в программную часть (в код системы), или в обе этих части.

На схеме ниже представлены примеры протоколов, а также уровни их распределения в модели TCP/IP:

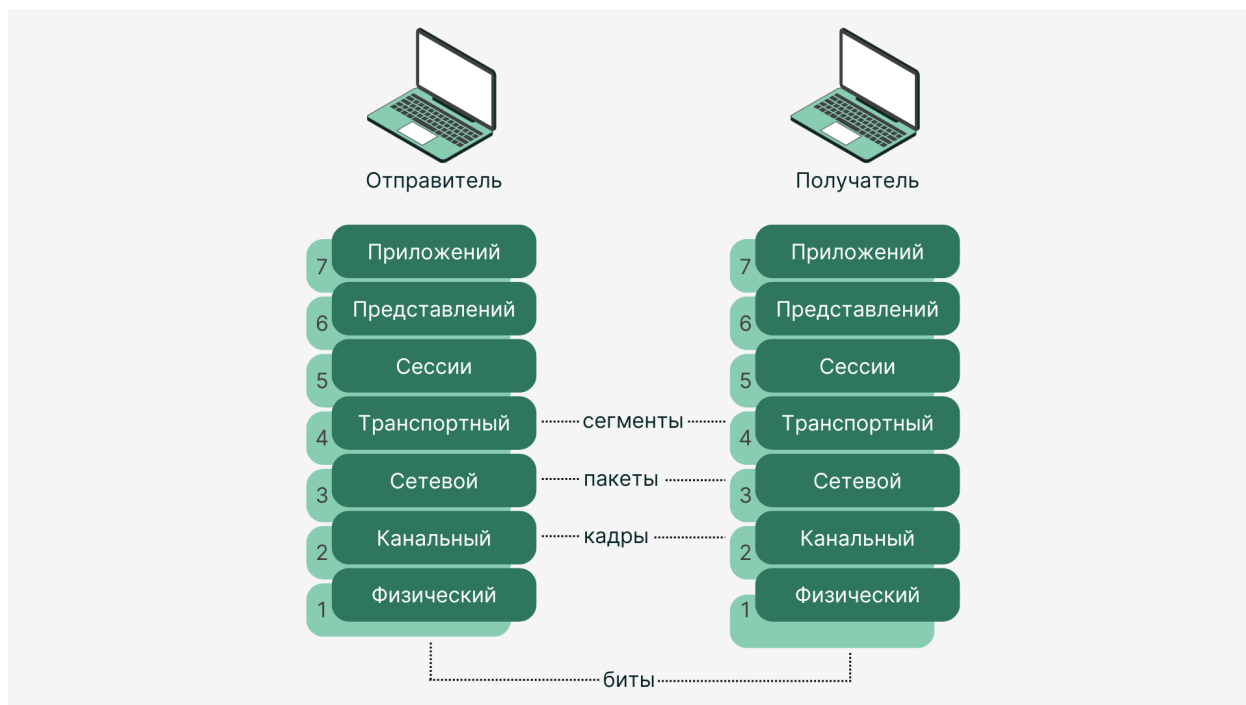
Стек протоколов TCP/IP



С другими протоколами и их назначением вы можете ознакомиться [здесь](#).

В процессе сетевого взаимодействия участвуют как минимум два устройства — устройство-отправитель и устройство-получатель. Говоря простым языком, каждая из моделей сетевых взаимодействий устанавливает правила и регламенты по отправке сообщений между компьютерами.

Отправленное сообщение проходит все уровни, начиная от прикладного уровня приложений и заканчивая физическим уровнем доступа к сети. Когда сообщение доходит до адресата, оно также проходит все уровни в обратном порядке.



Если привести аналогию из жизни, можно сказать, что обе модели описывают правила, по которым мем, который вы отправляете другу в мессенджере, будет преобразован до битов данных, передаваемых по электрическим проводам, а затем будет восстановлен по этим же битам и отображён на экране устройства вашего друга.

Нам как дата-сайентистам не нужно знать все подробности того, как работают и чем отличаются друг от друга разные уровни взаимодействия. Однако для общего развития и понимания того, как устройства обмениваются информацией, вы можете прочитать об уровнях моделей OSI и TCP/IP [здесь](#) и [здесь](#).

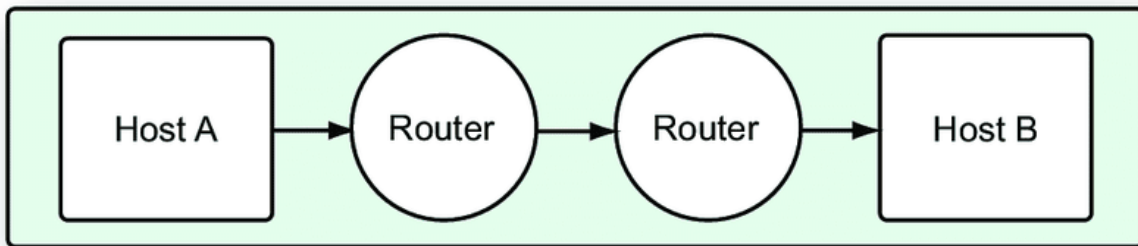
Для наших целей (деплой модели в прод) достаточно уметь работать всего с тремя протоколами:

IP — протокол сетевого уровня. Он определяет путь, по которому передаются данные.

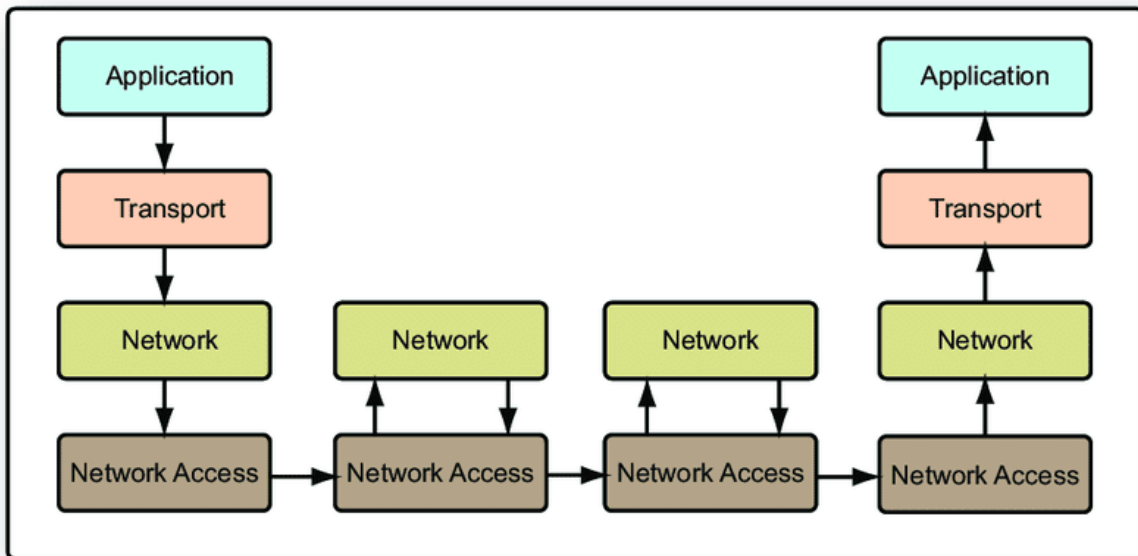
TCP — соответствует транспортному уровню, а значит, определяет, как передаются данные.

HTTP — относится к прикладному уровню, описывающему взаимодействие приложений с сетью.

Network Connections View



TCP/IP Model



IP

IP (Internet Protocol) — один из главных протоколов сетевого взаимодействия. Он отвечает за маршрутизацию трафика по сети, то есть определяет путь, по которому отправятся данные. Данные передаются пакетами (или датаграммами), которые формирует протокол IP.

Важным свойством IP является отсутствие гарантированной доставки пакетов и их целостности: пакеты могут прийти в другой очередности (не в той, в которой их отправляли), прийти повреждёнными (тогда они уничтожаются) или вообще не прийти.

Путь, по которому отправятся данные, строится на основе IP-адресов.

IP-адрес — это уникальный адрес, используемый для связи устройств внутри сети.

IP-адрес устроен довольно просто: чаще всего это четыре числа, разделённых точками (такой формат поддерживается в протоколе IPv4). Например, вот один из самых

популярных IP-адресов — 192.168.0.1. Вы могли вводить его, чтобы зайти в настройки своего роутера.

Каждое из чисел в адресе — это восьмизначное двоичное число, или, правильнее говорить, октет. Оно может принимать значения от 0000 0000 до 1111 1111 в двоичной системе или от 0 до 255 — в десятичной системе счисления, то есть 256 разных значений.

Получается, что диапазон IP-адресов стартует с 0.0.0.0 и заканчивается 255.255.255.255. Если посчитать количество всех адресов в этом диапазоне, получится чуть больше четырёх миллиардов.

Уникальность IP-адреса может быть глобальной (в рамках всего интернета) или локальной (в рамках локальной подсети). Некоторые IP-адреса не являются общедоступными и зарезервированы для специальных целей, например диапазоны IP-адресов:

Диапазон	Цели
127.0.0.1–127.255.255.255	используются для связи внутри локальной машины (localhost)
172.16.0.1–172.31.255.255	используются для частных подсетей (недоступных из интернета)
198.18.0.1–198.19.255.254	используются для тестирования производительности

localhost — зарезервированное доменное имя для IP-адресов из диапазона 127.0.0.1–127.255.255.255 (в сети из одного компьютера — для 127.0.0.1).

В компьютерной сети localhost относится к компьютеру, на котором запущена программа. Компьютер работает как виртуальный сервер. Тем самым создаётся так называемая «внутренняя петля»: обращаясь по IP-адресу localhost, вы, по сути, заставляете компьютер общаться с самим собой (хотя на самом деле внутри всё немного сложнее). Это нужно, например, для разработки и тестирования клиент-серверных приложений на одной машине (то есть и клиент, и сервер находятся на одном компьютере), что позволяет при разработке не использовать сетевое оборудование, дополнительные программные модули и тому подобное.

TCP

TCP (Transmission Control Protocol) — протокол транспортного уровня. Он отвечает за управление передачей данных и гарантирует:

- доставку пакетов (посылает пакеты повторно, если они не были доставлены);
- последовательность и целостность доставки пакетов (используя нумерацию и контрольные суммы для проверки);
- устраняет дубликаты в случае необходимости.

Важной особенностью TCP является то, что перед отправкой данных он «устанавливает соединение» с получателем — обменивается управляющей информацией. После отправки пакетов источник ждёт подтверждения от получателя, что пакеты были доставлены.

Обычно на одном **узле сети** (сервере, компьютере) работают несколько приложений/процессов одновременно. Для идентификации приложения на источнике и

получателе используется порт, который задаётся целым неотрицательным числом. Процесс или приложение могут зарезервировать у ОС определённый порт, например, для передачи данных по сети.

Порты разделяют на системные (0–1023), и пользовательские (1024–49151). Некоторые номера портов определены для конкретных приложений, например:

- 22 — протокол SSH для безопасной передачи данных;
- 25 — протокол SMTP для незащищённой передачи e-mail-сообщений;
- 80 — протокол HTTP.

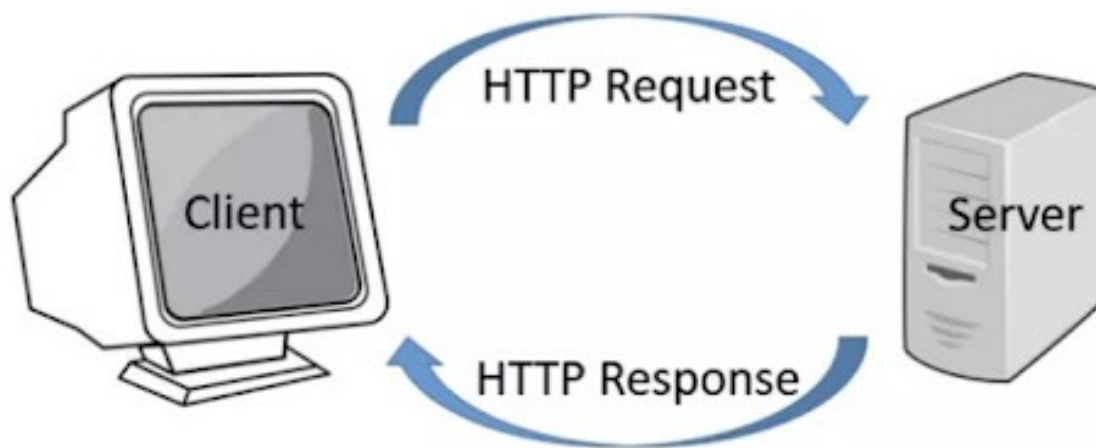
Например, если приложение доступно по адресу 172.16.0.11:8001, это значит следующее:

- 172.16.0.11 — IP-адрес;
- 8001 — TCP-порт, отведённый приложению.

HTTP

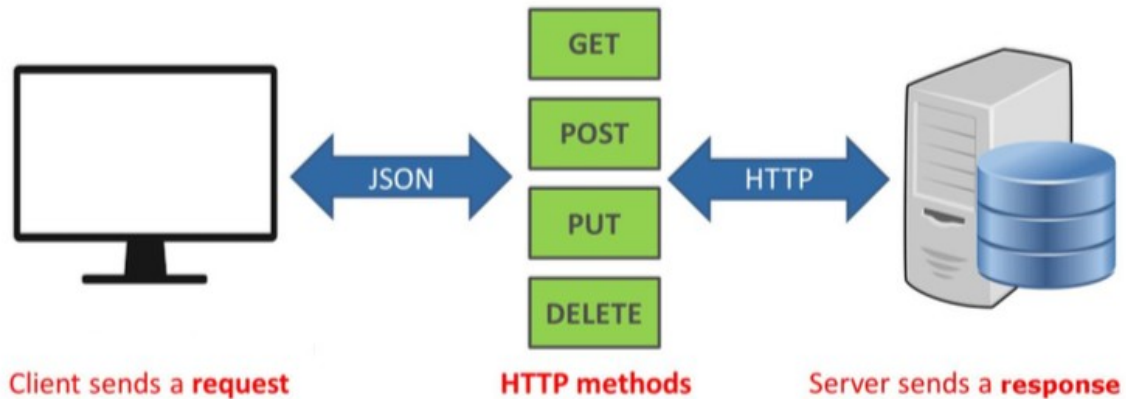
HTTP — это наиболее широко используемый протокол. Все сайты, на которые вы заходите, работают по этому протоколу. Он был разработан именно для передачи содержимого HTML-страниц в интернете, но впоследствии стал использоваться и для других целей. Например, HTTP применяется для налаживания взаимодействия между сервисами в сложных системах. Этим он нам и интересен.

Итак, HTTP — это протокол, который работает по принципу клиент-сервер.



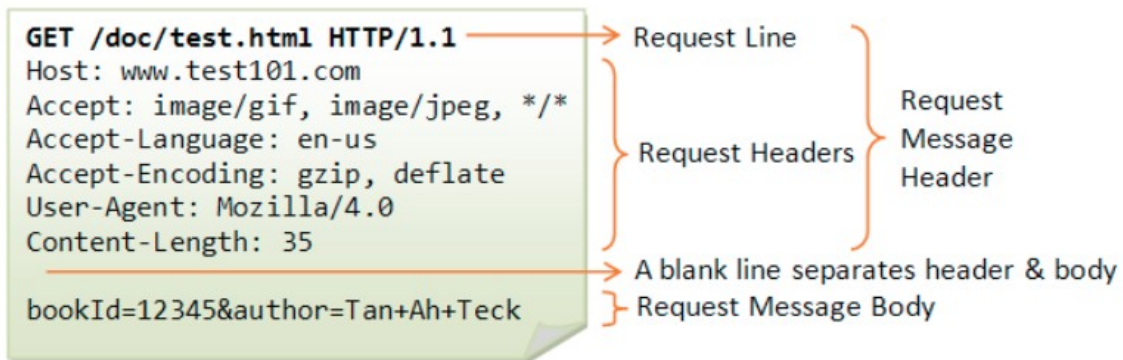
Это означает, что во взаимодействии участвуют две программы, причём в разных ролях. Одна из них — клиент, или «заказчик услуг», формирует запрос и отправляет его к серверу. Сервер, или «поставщик услуг», получив запрос, обрабатывает его, формирует ответ и возвращает его клиенту.

СТРУКТУРА HTTP-ЗАПРОСОВ

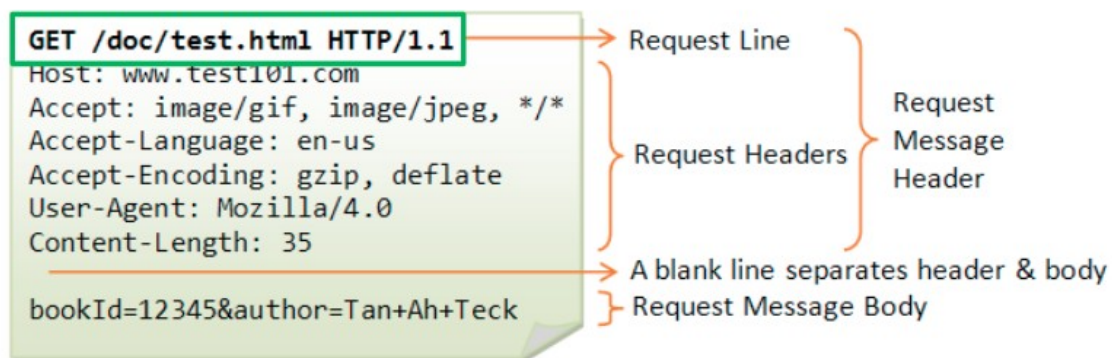


Запрос и ответ в HTTP являются строками, составленными в соответствии с протоколом.

Запрос состоит из трёх частей.



1. Стартовая строка, или Request Line — по ней определяется вид запроса.
2. Заголовки запроса, или Request Headers — дополнительные параметры запроса, в которых обычно передаётся служебная информация, например, в каком формате ожидается ответ или информация о клиенте.
3. Тело запроса, или Request Message Body — содержит данные для передачи. Эта часть присутствует не всегда. Давайте чуть подробнее разберём первую часть.



Первое, что указано в стартовой строке — это метод, или тип запроса. Есть набор стандартных методов, но теоретически вы можете создавать и свои.

Основные методы:

- GET — обычно означает получение содержимого ресурса и не содержит тела.

Например, когда вы заходите в каталог интернет-магазина, вы получаете страницу с товарами — ваш браузер отправляет GET-запрос на сервер интернет-магазина.

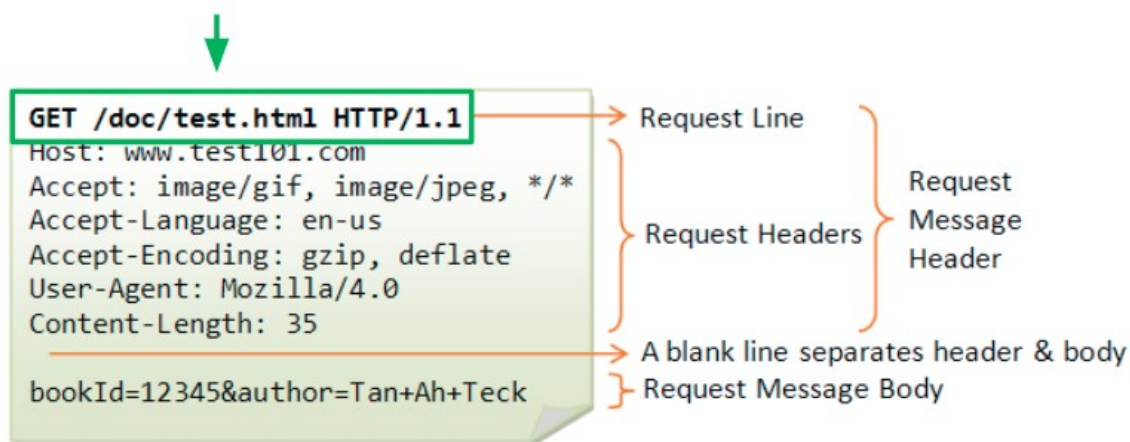
- POST — наоборот, передача данных ресурсу.

Например, когда вы заполняете форму авторизации на любом сайте и нажимаете кнопку для отправки своих данных, вы совершаете POST-запрос на сервер ресурса.

- PUT — обновление ресурса.
- DELETE — удаление ресурса.

В HTTP мы работаем с ресурсами, которые расположены по некоторому адресу на сервере. Изначально под ресурсами понимались HTML-файлы на сайте (вёрстка сайта), но сейчас это уже некоторое абстрактное понятие.

Адрес ресурса, или URI (Uniform Resource Identifier) — это то, что вы видите в адресной строке браузера. Он следует за методом в стартовой строке запроса.



Чем отличаются URI, URL и URN?

Для начала расшифруем аббревиатуры:

- URI — Uniform Resource Identifier (унифицированный идентификатор ресурса);
- URL — Uniform Resource Locator (унифицированный определитель местонахождения ресурса);
- URN — Uniform Resource Name (унифицированное имя ресурса).

Большинство считает, что <http://google.com> или <https://skillfactory.ru/> — это просто URL-адреса. Тем не менее, мы можем говорить о них как о URI. URI представляет собой комбинацию URL-адресов и URN. Таким образом, мы можем с уверенностью сказать, что все URL являются URI. Однако обратное неверно.

Давайте рассмотрим это на простом примере.

Имя «Джон Сноу» — это URN. Место, в котором живёт Джон, например «Улица Вестерос, 13» — это уже URL. Вас можно идентифицировать как уникальное лицо с вашим именем или вашим адресом. Эта уникальная личность — это уже URI. И хотя ваше имя может быть вашим уникальным идентификатором (URI), оно не может быть URL-адресом, поскольку имя не позволяет найти местоположение.

Теперь дадим расширенное определение терминам:

- URI — имя и адрес ресурса в сети. URI включает в себя URL и URN.

Пример: <https://wiki.merionet.ru/images/vse-chno-vam-nuzhno-znat-pro-devops/1.png>

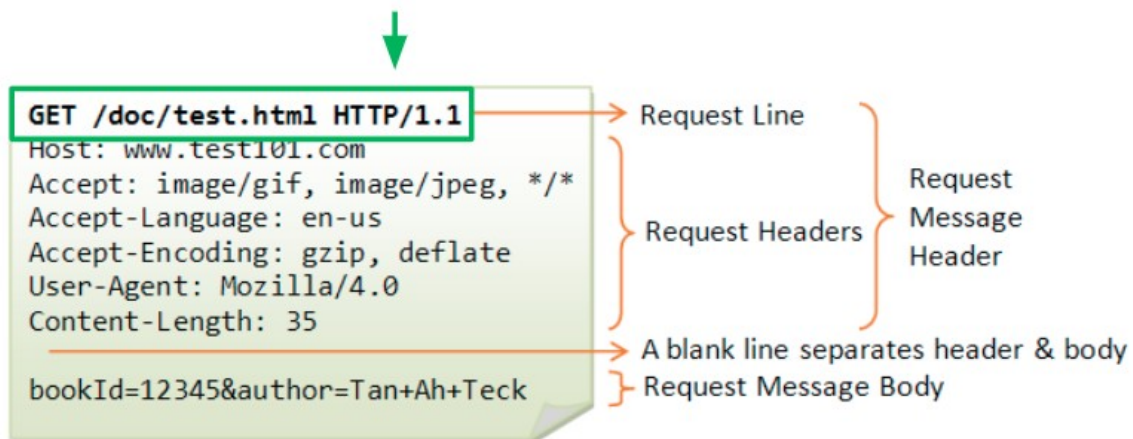
- URL — адрес ресурса в сети. URL определяет местонахождение и способ обращения к нему.

Пример: <https://wiki.merionet.ru>

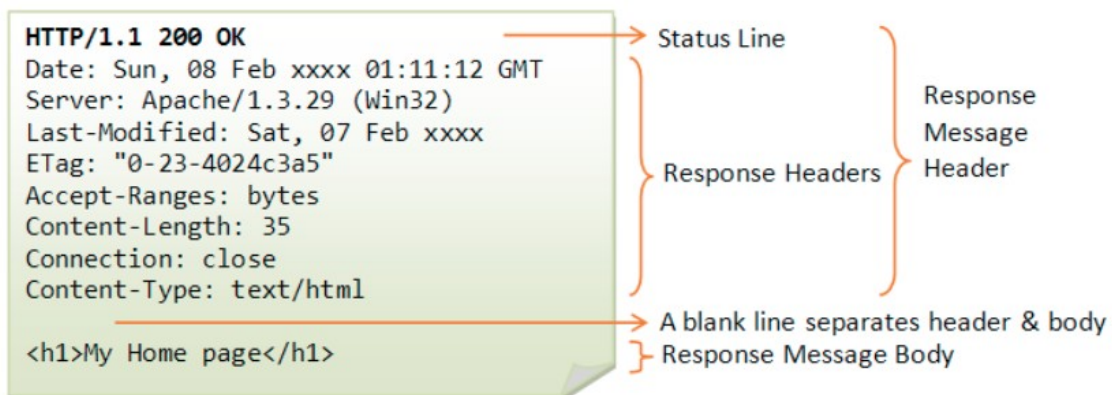
- URN — имя ресурса в сети. URN определяет только название ресурса, но не говорит, как к нему подключиться.

Пример: `images/vse-cto-vam-nuzhno-znat-pro-devops/1.png`

Последней идёт версия HTTP-протокола (кстати, последняя актуальная версия, 1.1, появилась ещё в 1999 году).



Ответ также состоит из стартовой строки, заголовков и тела.



Основное отличие — в стартовой строке: там вместо метода и URI указывается код состояния. Это численное значение, которое показывает результат обработки. Коды задаются протоколом, и вы наверняка встречали «ошибку 404» на сайтах.

404 — это как раз код состояния, означающий, что ресурса с заданным URI не существует на сервере.

Группы кодов состояния ответа HTTP-сервера делятся на следующие группы:

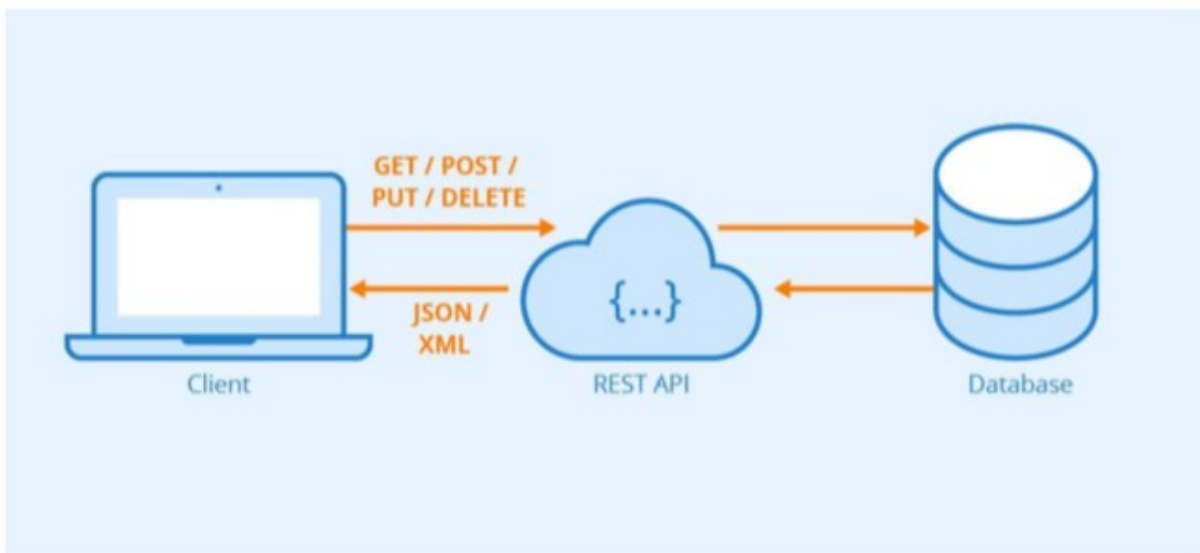
- информационные (100–199);
- успешно (200–299);
- перенаправление (300–399);
- ошибка клиента (400–499);
- ошибка сервера (500–599).

Примеры:

Код 200 означает, что запрос обработан успешно.

Код 500 означает, что сервер не смог обработать запрос.

REST (REPRESENTATIONAL STATE TRANSFER)



Спецификация HTTP не обязывает сервер понимать все методы, а также не указывает серверу, что он должен делать при получении запроса с тем или иным методом. Поэтому был изобретён архитектурный стиль REST.

Он даёт более верхнеуровневые указания, чем HTTP-протокол, а именно:

- как правильно организовывать адресацию к ресурсам;
- какие методы у этих ресурсов должны быть;
- какой ожидается результат.

Основная концепция философии REST заключается в том, что клиентом RESTful-сервера может быть что/кто угодно: браузер, другое приложение, разработчик. Веб-приложение, спроектированное по правилам REST, предоставляет информацию о себе в форме информации о своих ресурсах.

Ресурс может быть любым объектом, о котором сервер предоставляет информацию. Например, в API Instagram ресурсом может быть пользователь, фотография, хэштег. Каждая единица информации (ресурс) однозначно определяется URL.

- GET-запрос `/rest/users` — получение информации обо всех пользователях.
- GET-запрос `/rest/users/125` — получение информации о пользователе с `id=125`.
- POST-запрос `/rest/users` — добавление нового пользователя.
- PUT-запрос `/rest/users/125` — изменение информации о пользователе с `id=125`.
- DELETE-запрос `/rest/users/125` — удаление пользователя с `id=125`.

Поскольку спецификация REST является общепризнанной и широко распространённой, то следовать ей очень полезно.

Если вы говорите, что ваш сервис RESTful (то есть построен по этой спецификации), то любой разработчик сразу поймёт, как примерно устроен сервис, а значит, сможет быстро начать им пользоваться: отправлять правильные запросы и получать тот результат, который ожидает.

ПРОМЕЖУТОЧНЫЕ ИТОГИ

Для того чтобы наладить взаимодействие по HTTP, на стороне **клиента** вам необходимо сформировать запрос: указать адрес, куда отправится запрос, выбрать метод, а также задать заголовки и тело запроса, если это нужно для выбранного метода.

На стороне **сервера** при получении запроса необходимо знать, какой ресурс запрошен, чтобы понять, каким кодом его обрабатывать. Этот код, в зависимости от того, что он должен делать, может проверить значения заголовков и/или прочитать тело запроса.

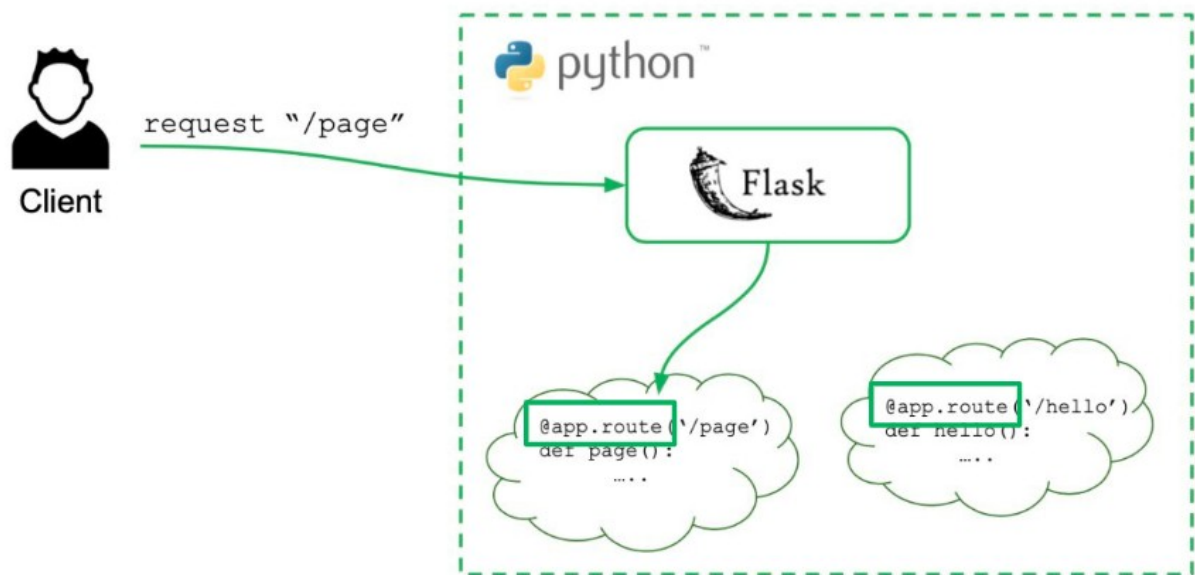
Отлично, со взаимодействием сервисов по сети мы разобрались. Эти теоретические знания обязательно пригодятся вам для построения собственного веб-сервиса. Теперь пора приступить к рассмотрению Python-фреймворков, которые как раз и помогут создать этот веб-сервис.

6. Деплой модели. Обзор фреймворков

Следующий шаг в подготовке к деплою — рассмотреть, какие существуют веб-фреймворки Python, которые позволяют реализовать серверную часть приложения и обеспечить к нему клиентский доступ.

FLASK

Flask позиционируется как микрофреймворк для написания **легковесных** сервисов, но при этом он достаточно гибкий и кастомизируемый, что позволяет использовать его в проектах любой сложности. Как и все остальные фреймворки, он работает на основе **маршрутизации запросов**, приходящих на сервер, который и должен обработать эти запросы.



В случае с Flask маршрутизация выглядит как обычные функции Python, которые вы помечаете специальным **декоратором** `@app.route`. Он связывает эту функцию с адресом, на который приходит запрос. В этой функции вы должны разобрать **параметры запроса**, возможно, выполнить какую-то логику и вернуть ответ клиенту.

Таким образом, для написания минимально функционального сервера достаточно добавить в скрипт всего несколько строк: необходимые импорты, создание Flask-приложения, декоратор, который свяжет адрес (**endpoint**, **эндпоинт**) с функцией, и запуск приложения. Последнее как раз и запустит работу Flask: он будет слушать запросы, маршрутизировать их и возвращать ответ.

Что такое эндпоинт?

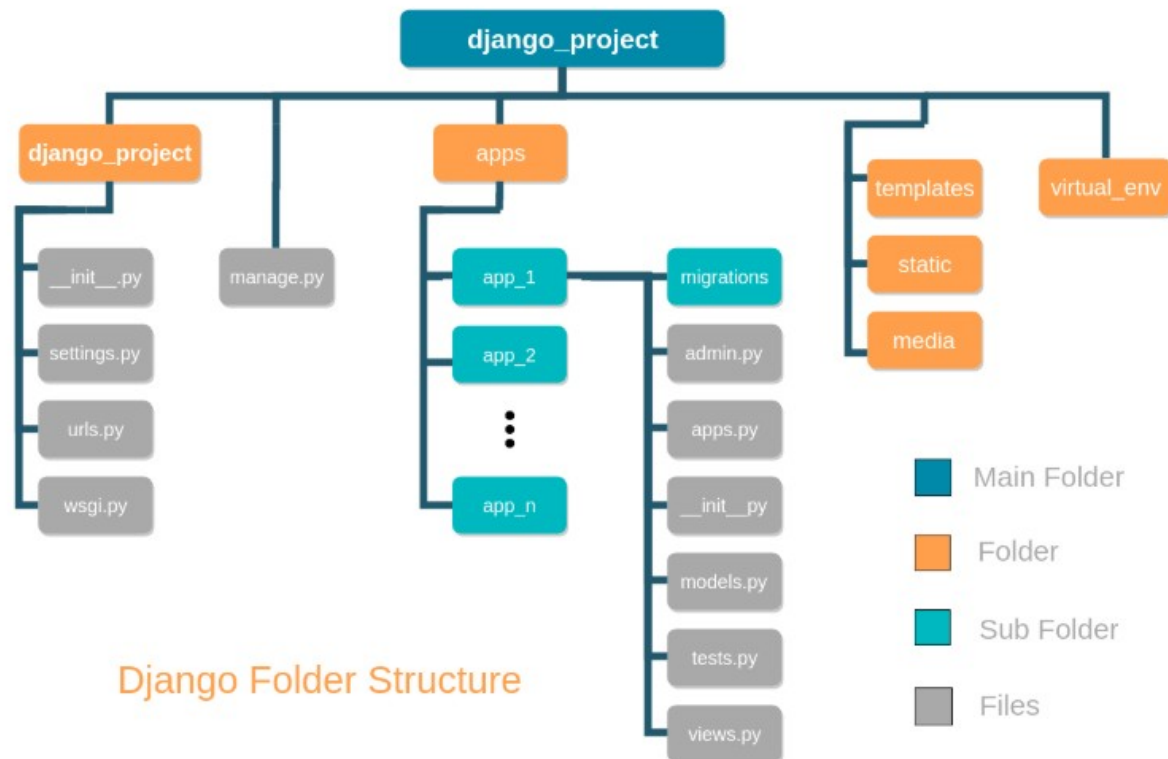
Рассмотрим на примере. Сайт почты Mail.ru имеет адрес `https://e.mail.ru`. Для того чтобы попасть в папку со входящими письмами, необходимо обратиться к соответствующему интерфейсу приложения. Для этого к адресу сайта добавляется `/inbox/`: `https://e.mail.ru/inbox`. Интерфейс, который предоставляет доступ к вашим отправленным письмам, находится по адресу `https://e.mail.ru/sent/`. Получается, что `/inbox/` и `/sent/` — это и есть эндпоинты.

У Flask есть множество продвинутых функций и плагинов, написанных сторонними разработчиками для решения самых разных задач. Однако нам достаточно научиться работать с Flask на минимальном уровне, чтобы вывести свою модель как отдельный маленький сервис (микросервис), решающий одну конкретную задачу — к сервису можно обращаться по сети и получать результаты (предсказания модели).

Реализация глобальной структуры сервиса (веб-приложения) и встраивание этой структуры в микросервис — это уже задача других специалистов, например веб-разработчиков.

DJANGO

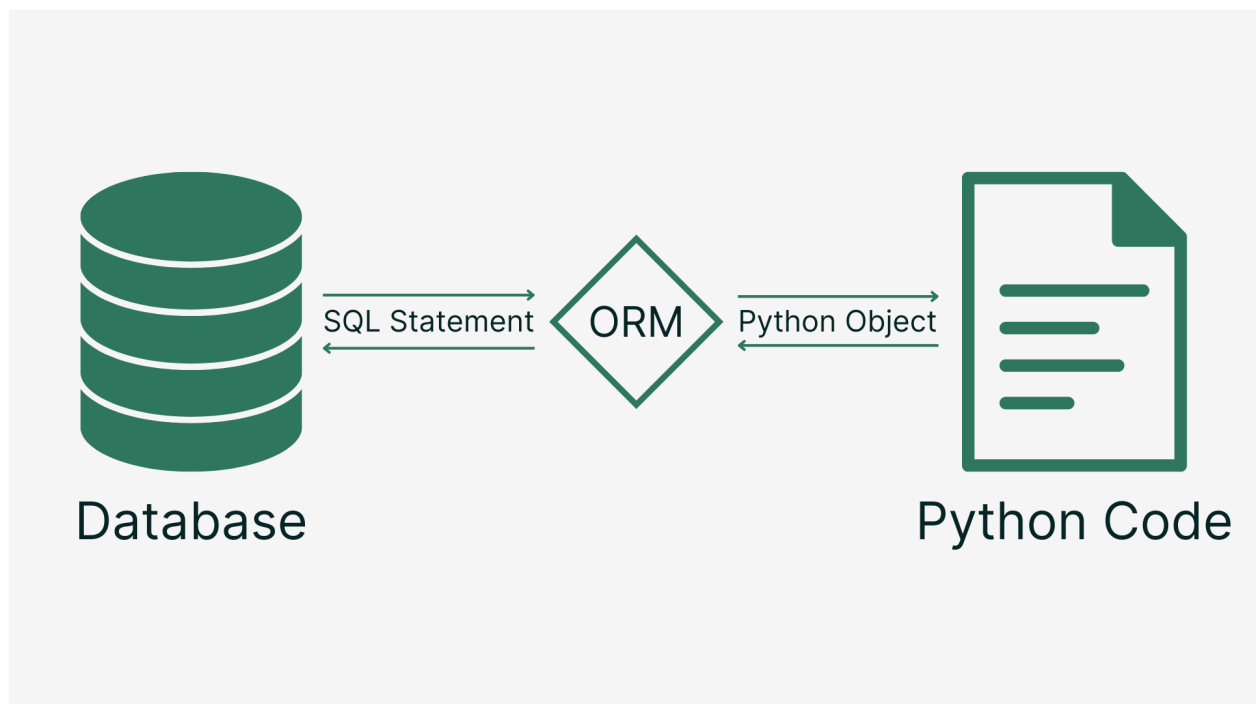
В отличие от Flask, Django нельзя назвать микрофреймворком. Напротив, даже минимальный проект для Django обычно генерируется **специальными скриптами** и включает с десяток файлов и папок.



Такая сложная структура нужна для громоздких приложений, чтобы не путаться в коде. Поэтому Django используют, когда нужно что-то **посерьёзней**, например целые сайты с базами данных.

ПРЕИМУЩЕСТВА DJANGO:

- Мощный движок для рендера HTML-страниц, основанный на шаблонах.
- Собственный ORM.



ORM (Object Relational Mapping) — это подход, который позволяет отображать нативные объекты языка (в нашем случае классы Python) в таблицы в базе данных. Благодаря этому разработчик может абстрагироваться от написания SQL-запросов и «на лету» менять используемую базу данных. В других фреймворках вам придётся использовать сторонние ORM (самая популярная — `sqlalchemy`).

- Огромное количество дополнительных плагинов. Один из них практически так же популярен, как и сам Django — речь о Django Rest Framework. Основываясь на ORM, он позволяет в несколько строк реализовывать endpoints протокола RESTful. Сам Django Rest Framework также имеет множество расширений. Благодаря этому написание сложного проекта можно свести к составлению правильной модели данных, установке и настройке подходящих расширений.

FLASK VS DJANGO

Давайте более подробно рассмотрим характеристики Flask и Django и узнаем, чем отличаются и чем похожи эти два фреймворка.

В Django есть модули для очень широкой функциональности. Чтобы получить к ним доступ, надо просто добавить нужную настройку в конфигурацию. Это отлично перекликается с девизом Django: “The web framework for perfectionists with deadlines” («Веб-фреймворк для перфекционистов, которые придерживаются сроков»).

Во Flask всё наоборот: сам фреймворк предоставляет только базовую функциональность. «Из коробки» будет доступно направление по адресам эндпойнтов, обработка запросов и ошибок, дебаггер и некоторые другие функции, явно связанные с сервером. Например, чтобы добавить авторизацию, придётся устанавливать **стороннюю библиотеку**. При этом вы можете выбрать ту, которая лучше подходит текущему проекту, лучше поддерживается или просто более удобна в использовании, а можете вообще не подключать библиотеки.

Flask нужен для оборачивания **инференса модели в API**, когда, например, по поступившему запросу запускается предикт модели. Знание Flask пригодится и на этапе создания прототипа, и на этапе тестирования.

Также некоторые компании строят весь продакшен на Flask, а значит, знать его основы будет полезно.

В связи с тем, что Django выпускается в релиз вместе со всеми сопутствующими модулями, у него достаточно длинные релиз-циклы. С Flask эта проблема менее актуальна: если какой-то функциональности не хватает, часто она есть в какой-то библиотеке. Этот подход жизнеспособен, потому что у Flask большое сообщество разработчиков, которые занимаются поддержкой библиотек для этого фреймворка.

Flask приобрёл популярность именно благодаря небольшим приложениям, для которых Django казался избыточным. Однако сегодня Flask используется не только для таких приложений.

Для справки приведём инструментарий, используемый в Django и Flask, чтобы продемонстрировать, как фреймворки отличаются друг от друга по функциональности. Запоминать таблицу не нужно.

Примечание. При разработке на Flask список чаще всего не ограничивается перечисленными в таблице библиотеками — здесь собраны только наиболее популярные и известные. Также есть библиотеки, у которых есть специальные расширения для Flask. Чаще всего они не обязательны для работы с библиотекой, но могут быть полезны.

ФУНКЦИОНАЛЬНОСТЬ	DJANGO	FLASK	FLASK EXTENSION
Шаблоны	Django templates	Jinja2	
ORM	Django ORM	sqlalchemy	Flask-sqlalchemy
Миграции	Django ORM	Alembic	Flask-alembic
Работа с MongoDB	Django ORM	PyMongo , MongoEngine	Flask-PyMongo , Flask-MongoEngine
Авторизация, аутентификация	Django Contrib Auth	Flask-Login , Flask-Principal	
Админка	Django Admin	Flask-Admin	
Кэширование	Django Cache Framework	Flask-Caching	
REST	Django REST	Flask-RESTful , Flask-RESTPlus	
Сериализация	Django REST	Marshmallow	Flask-marshmallow

КТО ПОБЕДИЛ?

Как мы видим, для основной функциональности, реализованной в Django, есть аналоги, которые работают с Flask. Нельзя сказать, что какой-то из фреймворков проигрывает другому по большому числу параметров.

Однако стоит понимать, что, выбирая Django, вы оставляете проект со всей экосистемой Django, и что-то изменить впоследствии будет трудно. Выбирая Flask, вы получаете систему, построенную из блоков, каждый из которых проще заменить.

FASTAPI

Это относительно молодой фреймворк, однако он уже успел завоевать некоторую популярность. Возможно, через некоторое время он превратится в стандарт индустрии, заменив Flask.

Примечание. Да-да, именно Flask! FastAPI тоже позиционируется как легковесный фреймворк. Более того, принцип его использования очень похож на Flask. Авторы утверждают, что, вдохновившись Flask, они написали **более быстрый** (за счёт использования более современных протоколов), но при этом не уступающий по функциональности фреймворк.

Основное нововведение в FastAPI — интеграция библиотеки `pydantic`, которая позволяет **декларативно*** описывать структуры запросов.

* Это значит, что вы можете написать класс с несколькими полями, указав их тип с помощью аннотаций, а функционал `pydantic` позволит вам сериализовывать такие объекты в JSON и обратно без написания дополнительного кода. Благодаря этому вы сможете избавиться от монотонного разбора параметров запроса и даже сделать его более надёжным.

Также в FastAPI есть **поддержка асинхронных функций** и [реализация парадигмы Dependency Injection](#).

```
from datetime import datetime
from typing import List, Optional
from pydantic import BaseModel

class User(BaseModel):
    id: int
    name = 'John Doe'
    signup_ts: Optional[datetime] = None
    friends: List[int] = []
```



Что такое асинхронные функции?

Асинхронная функция, или **асинхронный метод** — это функция, которая может быть приостановлена во время выполнения.

Это обычные функции и методы, с которым мы работали ранее. При вызове таких функций (методов) они либо выполняются до завершения, либо вызывают ошибку, либо никогда не завершаются (в функции есть бесконечный цикл).

Рассмотрим пример асинхронной функции.

Представьте, что вы когда-то поставили будильник на 8:30 и с тех пор он исправно выполняет свою работу. Чтобы понять, когда вставать, вам не нужно непрерывно смотреть на часы всю ночь. Нет нужды и посматривать на них периодически (скажем, с интервалом в пять минут). Асинхронная функция «подъём» находится в режиме ожидания. Как только произойдёт событие «на часах 8:30», она сама даст о себе знать.

Подробнее об асинхронных функциях можно прочитать [здесь](#).

ДРУГИЕ ФРЕЙМВОРКИ

Фреймворк aiohttp тоже близок к Flask, но основан на асинхронном взаимодействии. Aiohttp добавлен в Python версии 3.4.

Tornado — относительно старый и более общий асинхронный фреймворк для написания сетевых приложений, но также он умеет работать с HTTP. По заверениям разработчиков, он очень хорошо держит нагрузку. В 2019 году Tornado занимал третье место по популярности среди разработчиков после Django и Flask.

В этом юните мы рассмотрели основные фреймворки для реализации веб-сервисов на Python. Далее в курсе мы будем использовать фреймворк Flask, однако если вы уже знакомы с другими фреймворками, вы можете использовать их.

7. Пишем сервер на Flask

Если бы вы попробовали написать сервер в Jupyter Notebook, то почти сразу поняли бы, что он для этого не предназначен. Как минимум потому, что после запуска вы больше не смогли бы выполнить в ноутбуке ни одной команды — программа сервера переходит в режим прослушки сети и ожидания приходящих запросов. Поэтому разработку нашего веб-сервиса мы будем вести в обычных файлах .py.

Итак, давайте напишем простое приложение на Flask.

ЧАСТЬ I. ИНИЦИАЛИЗАЦИЯ ВЕБ-ПРИЛОЖЕНИЯ

Примечание. В видео эксперт использует IDE PyCharm. Вы можете установить её самостоятельно по [инструкции](#) либо использовать ту IDE, к которой вы привыкли, например VS Code.

Сначала необходимо установить Flask. Сделать это можно с помощью pip:

```
pip install flask
```

Теперь импортируем его и создадим объект Flask-приложения.

```
from flask import Flask
app = Flask(__name__)
```

Мы передаём `__name__` при инициализации класса Flask, чтобы определить имя, с которым будет использоваться этот модуль. Flask использует расположение файла как точку, к которой он привязывает ресурсы.

Совет. В абсолютном большинстве случаев можно передавать аргумент `__name__` для инициализации класса — приложение будет настроено верно.

Теперь мы можем написать функцию, которая будет обрабатывать запросы, и прикрепить её к какому-то пути (URI). Это делается с помощью специального декоратора `route`.

```
@app.route('/hello')
def hello_func():
    return 'hello!'
```

Наша функция пока не делает никакой обработки и просто отвечает строкой с приветствием. Нам осталось запустить приложение. Для этого выполним метод `run`, не забыв занести его в стандартный `main`.

```
if __name__ == '__main__':
    app.run('localhost', 5000)
```

ЧАСТЬ II. ПЕРЕДАЧА ПАРАМЕТРОВ ЗАПРОСА

Давайте немного усложним логику обработки — создадим функцию, которая принимает какой-то параметр и использует его для вычисления результата.

Параметры можно передавать тремя способами:

- через адресную строку;
- через заголовки;
- через тело.

В заголовках обычно передают сервисные параметры. Так как у метода GET не бывает тела, остаются только **параметры адресной строки**.

Вы наверняка видели параметры на сайтах, когда в конец адреса дописывается что-то вроде `?id=10&page=2`. Если поставить в конце адреса вопросительный знак, то после него можно добавлять параметры. Передавать можно любые параметры — они будут перечисляться через `&` и состоять из имени и значения (`id=10`). Например, если вы будете что-то искать в Google, ваш запрос будет видно в адресной строке в параметре с названием `q` (от англ. query).

Параметры запроса во Flask находятся в специальном объекте `request`, который нужно импортировать. Параметры адресной строки можно найти в поле `args` этого объекта, где `args` — это словарь.

Давайте немного модифицируем наш код — теперь сервер будет здороваться и обращаться по имени. Для этого занесём параметр `name`, полученный из `request`, в переменную и воспользуемся этим значением, поставив его в форматированную строку.


```

from flask import Flask, request

app = Flask(__name__)
@app.route('/hello')

def hello_func():
    name = request.args.get('name')
    return f'hello {name}!'

if __name__ == '__main__':
    app.run('localhost', 5000)

```

Перезапустите сервер и зайдите по адресу <http://localhost:5000/hello?name=world>.

В результате выполнения запроса вы должны увидеть в браузере текст "hello world".

Поменяйте значение параметра name в адресной строке на своё имя и выполните новый запрос к серверу, чтобы проверить, что при изменении параметра адресной строки меняется и возвращаемый результат.

ЧАСТЬ III. POST-ЗАПРОСЫ

Пока что мы написали обработчик GET-запроса.

Если мы хотим обрабатывать и другие методы, например POST-запросы, это необходимо указать в декораторе:

```
@app.route('/add', methods=['POST'])
```

Что должна возвращать функция-обработчик? Она будет возвращать объект Response, в котором мы должны были бы задать все компоненты ответа: код обработки, заголовки и тело. К счастью, Flask (а точнее, объект Response) умеет и превращать в ответы другие объекты, и самостоятельно формировать ответ.

Например, строка

```
return f'hello {name}!'
```

преобразуется в ответ с кодом 200 и телом, состоящим из этой строки:

```
return f'hello {name}!', 200
```

Также можно возвращать кортеж из двух элементов, строки и числа. Число используется как код обработки.

Примечание. Вспомогательная функция *jsonify* поможет преобразовать обычный питоновский словарь в ответ в формате JSON, который очень часто используется для передачи структурированных данных.

Вооружившись этими знаниями, напомним обработчик POST-запроса, который будет читать тело запроса в JSON-формате и составлять ответ на основе его содержимого.

```
from flask import Flask, request, jsonify

app = Flask(__name__)
```

Укажем, что функция обрабатывает метод POST:

```
@app.route('/add', methods=['POST'])
```

Напишем саму функцию:

```
def add():

    num = request.json.get('num')
```

Параметры тела доступны в поле data. Но если тело — это JSON-строка, то можно использовать поле json.

Напишем проверку и укажем код ошибки:

```
    if num > 10:
        return 'too much', 400

    return jsonify({
        'result': num + 1
    })
```

Тогда функция-обработчик будет иметь следующий вид:

```
@app.route('/add', methods=['POST'])
def add():
    num = request.json.get('num')
    if num > 10:
        return 'too much', 400
    return jsonify({
        'result': num + 1
    })
```

Запускаем:

```
if __name__ == '__main__':

    app.run('localhost', 5000)
```

К сожалению, браузеры не умеют писать POST-запросы самостоятельно. Это реализуется только через клиентские приложения, поэтому нам не хватит обычного браузера, чтобы проверить сервис.

Если вы любите визуальные редакторы запросов, рекомендуем обратить внимание на [Postman](#).

Напишем простой сервис в соседнем скрипте, используя библиотеку [requests](#), которая позволяет отправлять HTTP-запросы. Для этого создадим отдельный файл `client.py`, в котором опишем работу программы клиента.

Чтобы выполнить POST-запрос, нужно просто вызвать соответствующую функцию и передать ей адрес (URL) и содержимое тела запроса.

Напишем простой сервис в **соседнем скрипте**, используя библиотеку [requests](#), которая позволяет отправлять HTTP-запросы. Для этого создадим отдельный файл `client.py`, в котором опишем работу программы клиента.

Чтобы выполнить POST-запрос, нужно просто вызвать соответствующую функцию и передать ей адрес (URL) и содержимое тела запроса.

```
import requests

if __name__ == '__main__':
    r = requests.post('http://localhost:5000/add', json={'num': 5})
```

Далее выведем статус-код ответа:

```
print(r.status_code)
Теперь пропишем проверку:

if r.status_code == 200:
    print(r.json()['result'])
else:
    print(r.text)
```

Полный код нашего простейшего клиентского приложения будет выглядеть так:

```
import requests

if __name__ == '__main__':
    # выполняем POST-запрос на сервер по эндпоинту add с параметром json
    r = requests.post('http://localhost:5000/add', json={'num': 5})
    # выводим статус запроса
    print(r.status_code)
    # реализуем обработку результата
```

```
if r.status_code == 200:
    # если запрос выполнен успешно (код обработки=200),
    # выводим результат на экран
    print(r.json()['result'])
else:
    # если запрос завершён с кодом, отличным от 200,
    # выводим содержимое ответа
    print(r.text)
```

Примечание. Чтобы запустить код клиентского приложения параллельно работающему серверу в VS Code, создайте новый терминал и запустите файл client.py вручную, написав в терминале команду:

```
python client.py
```

```
PROBLEMY    ВЫХОДНЫЕ ДАННЫЕ    КОНСОЛЬ ОТЛАДКИ    ТЕРМИНАЛ    JUPYTER
```

```
> python +
```

```
* Serving Flask app "server" (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: off
* Running on http://localhost:5000/ (Press CTRL+C to quit)
```

Далее можно переключаться между терминалами и запускать в них необходимые команды.

A screenshot of a terminal window. The top bar has tabs labeled "ПРОБЛЕМЫ", "ВЫХОДНЫЕ ДАННЫЕ", "КОНСОЛЬ ОТЛАДКИ", "ТЕРМИНАЛ", and "JUPYTER". The "ТЕРМИНАЛ" tab is active. The terminal prompt shows the path "PS A:\Курс DS-3.0\PROD-1\web>" followed by the command "python client.py" which is underlined in red.

Выполнив данный код, мы произведём POST-запрос на локальный сервер по endpointу `add`. Так как переданное значение `num < 10`, то запрос будет выполнен успешно и в терминале, соответствующему программе-клиенту, будет выведено:

200

6

ДЕПЛОИМ МОДЕЛЬ НА ВЕБ-СЕРВИС

Итак, мы рассмотрели базовые принципы работы с фреймворком Flask и научились писать простейшие веб-сервисы. Этого достаточно, чтобы решить нашу первоначальную задачу — задеплоить модель.

Нам необходимо написать веб-сервис, который позволит клиентам делать запросы к нашей модели. В ответ клиенты должны получать ответы — предсказания модели. Формат предсказаний обсудим ниже.

Файл с сериализованной моделью вы можете найти [здесь](#).

У модели, как всегда, есть метод `predict`, который принимает на вход `numpy`-массив размерности (кол-во объектов; кол-во признаков). Всего у нас четыре анонимных признака, и их интерпретация сейчас не имеет значения.

Давайте обернём процедуру предсказания модели в обработчик POST-запросов.

Задание 7.6

Используя только что обретенные знания, напишите Flask-приложение, которое по эндпоинту `/predict` будет слушать POST-запросы на предсказания.

В теле POST-запроса — список из четырёх чисел в формате JSON (один объект, четыре признака). Эти числа можно извлечь из запроса следующим образом:

```
features = request.json
```

Так как на вход модели необходимо подать `numpy`-массив определённой размерности, а не список, не забудьте перевести результат в тип `np.array()` и скорректировать его под размер (1, 4) с помощью `reshape()`:

```
features = np.array(features).reshape(1, 4)
```

После этого подайте полученный вектор на вход модели и получите результат.

Обратите внимание, что результатом предсказания также станет не число, а `numpy`-массив, пусть и состоящий из одного числа. Нас же интересует именно предсказанное число, поэтому не забудьте извлечь результат из полученного массива (обратившись по индексу 0).

Ответом на запрос должен быть JSON-формат `{"prediction": *число - предсказание модели*}`.

Для проверки запустите свой сервер на порте 5000 локальной машины и выполните [проверочный клиентский скрипт](#).

Код сервера:

```
from flask import Flask, request, jsonify

# import joblib
import numpy as np
import pickle

with open('SF_DataScience/Блок 7. ML в бизнесе/5. DS_PROD-1.
Подготовка модели к продакшену и деплой/data/model.pkl', 'rb') as
pkl_file:
    model = pickle.load(pkl_file)

app = Flask(__name__)

@app.route('/predict', methods=['POST'])
```

```
def get_pred():
    features = np.array(request.json)
    features = features.reshape(1, 4)
    pred = model.predict(features)
    return {"prediction": pred[0]}

if __name__ == '__main__':
    app.run('localhost', 5000)
```

Переходим в директорию с проверочным скриптом:

```
cd 'E:\Study\Data science SF\SF_DS\SF_DataScience\Блок 7. ML в бизнесе\5. DS_PROD-1. Подготовка модели к продакшену и деплой\data'
```

Запускаем скрипт:

```
python hw2_check_ol.py
```

После запуска скрипта получаем секретный код.

Поздравляем — теперь у вас есть рабочий прототип модели, который работает на веб-сервисе и умеет выдавать предсказания в ответ на POST-запросы клиентов. Однако это ещё не всё: в следующем юните мы узнаем, как повысить производительность сервиса.

*8. GIL. uWSGI + NGINX

Этот юнит является дополнительным и предназначен для тех, кто хочет немного углубиться в процесс создания настоящих веб-сервисов. Задания в юните не оцениваются.

В предыдущем юните мы познакомились с основами веб-разработки и смогли запустить на локальной машине маленький веб-сервер, который может принимать запросы и обрабатывать их. В этой части модуля мы займёмся оптимизацией работы нашего веб-сервиса и поднимем настоящий, полноценный веб-сервер.

Сразу отметим: инструменты, рассматриваемые в этом юните, предназначены **исключительно для UNIX-операционных систем (Linux и MacOS)**.

В целом, разработка веб-приложений под Windows — не самая популярная идея. Большая часть серверов в продакшн работает на дистрибутивах ОС Linux. Для этого есть много причин, но сейчас мы не будем на них останавливаться. Важно понимать, что некоторые инструменты, такие как uWSGI, который мы будем рассматривать далее, просто не предназначены для ОС Windows, и попытки развернуть такие инструменты на этой ОС бессмысленны. Для Windows существуют свои решения, о которых мы также упомянем.

По упомянутой выше причине рекомендуем вам уже сейчас понемногу «перевозить» свою разработку на ОС Linux, если вы не сделали этого раньше. В качестве дистрибутива Linux советуем использовать Ubuntu (версии не ниже 16.04). Ниже приведены способы установки Ubuntu с подобными инструкциями.

СПОСОБЫ УСТАНОВКИ UBUNTU:

- Самый безопасный, простой и рекомендуемый нами способ — установить виртуальную машину Ubuntu с помощью программы виртуализации VirtualBox. Виртуальную машину можно представить как операционную систему внутри другой операционной системы. Подробную инструкцию по установке вы можете найти [здесь](#).
- Пользователям Windows также можно использовать Windows Subsystem for Linux (WSL) — подсистему ОС Windows 10, позволяющую разработчикам и тестировщикам запускать нативные приложения Linux, писать скрипты, выполнять команды непосредственно из Windows. О том, как установить WSL, вы можете прочитать [здесь](#).

Стоит отметить, что WSL не поддерживает рабочие столы или приложения с графическим пользовательским интерфейсом. Это обычная командная строка Linux. Однако Microsoft позаботились об этом и в IDE VS Code есть возможность вести разработку прямо в Windows, а выполнение программы (запуск сервера) организовывать непосредственно из WSL. То есть, по сути, вся Python-разработка будет выполняться из-под Windows, а работа приложения — на Linux. Подробнее вы можете почитать об этом [здесь](#).

- Самый продвинутый, но в то же время опасный способ — установить Linux в качестве второй системы параллельно Windows. Однако при настройке системы неопытным пользователем возможна потеря файлов на исходной ОС. Поэтому, если вы выберете именно этот способ, настоятельно рекомендуем вам быть предельно внимательными и следовать всем приведённым инструкциям. Подробную инструкцию по установке вы можете найти [здесь](#).

Если вы используете первый или третий способы, вам понадобится установить все необходимые инструменты: язык Python встроен в Ubuntu по умолчанию, однако он поставляется без менеджера пакетов `pip`. Уставив `pip`, вы сможете установить все необходимые библиотеки (`numpy`, `sklearn`, `flask` и `requests`). Также вам понадобится IDE (например, VS Code + Jupyter Notebook). О том, как установить эти инструменты на все самые популярные ОС, мы рассказывали в модуле [PY-8. «Инструменты для Data Science»](#).

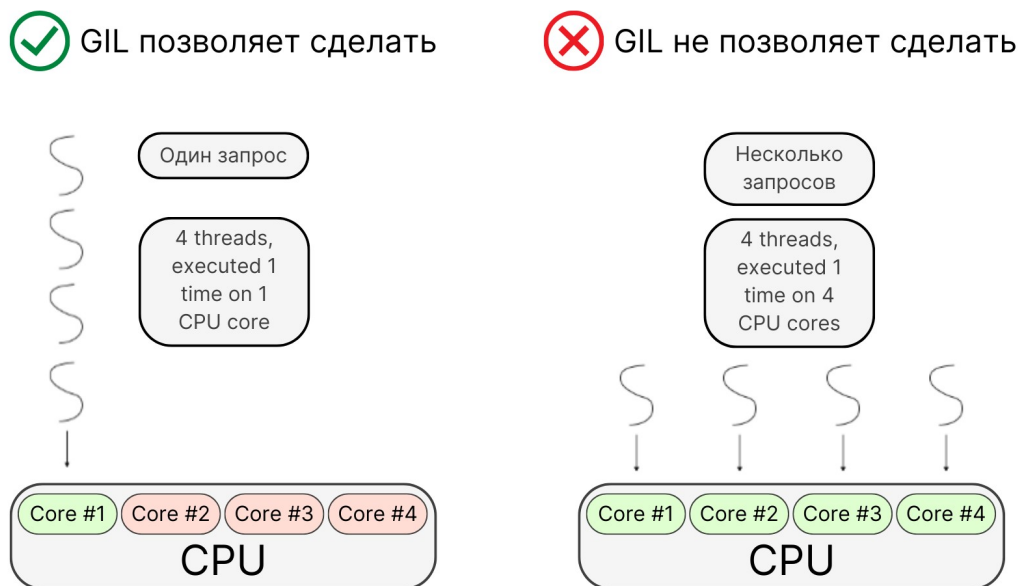
Итак, если вы установили ОС Linux и готовы повысить производительность своего веб-сервиса, давайте начнём.

GIL. ОГРАНИЧЕНИЕ НА МНОГОПОТОЧНОСТЬ

Представим ситуацию: мы хотим, чтобы сервер с нашей моделью одновременно обрабатывал **несколько процессов**. Если мы попробуем запустить такой сервер, то либо его производительность будет крайне низкой, либо процессы просто не запустятся. Почему?

Дело в механизме под названием GIL — Global Interpreter Lock. Он заключается в том, что в любом процессе Python одновременно может работать только один тред (один конкретный поток).

Примечание. Подробнее об отличиях процесса и потока можно узнать [здесь](#).



Таким образом, процесс Python не может утилизировать многоядерные системы. Для нашей задачи написания веб-сервера это означает, что сервер на Python одновременно может обрабатывать **только один запрос**, и это очень серьёзное ограничение даже для самых простых сервисов.

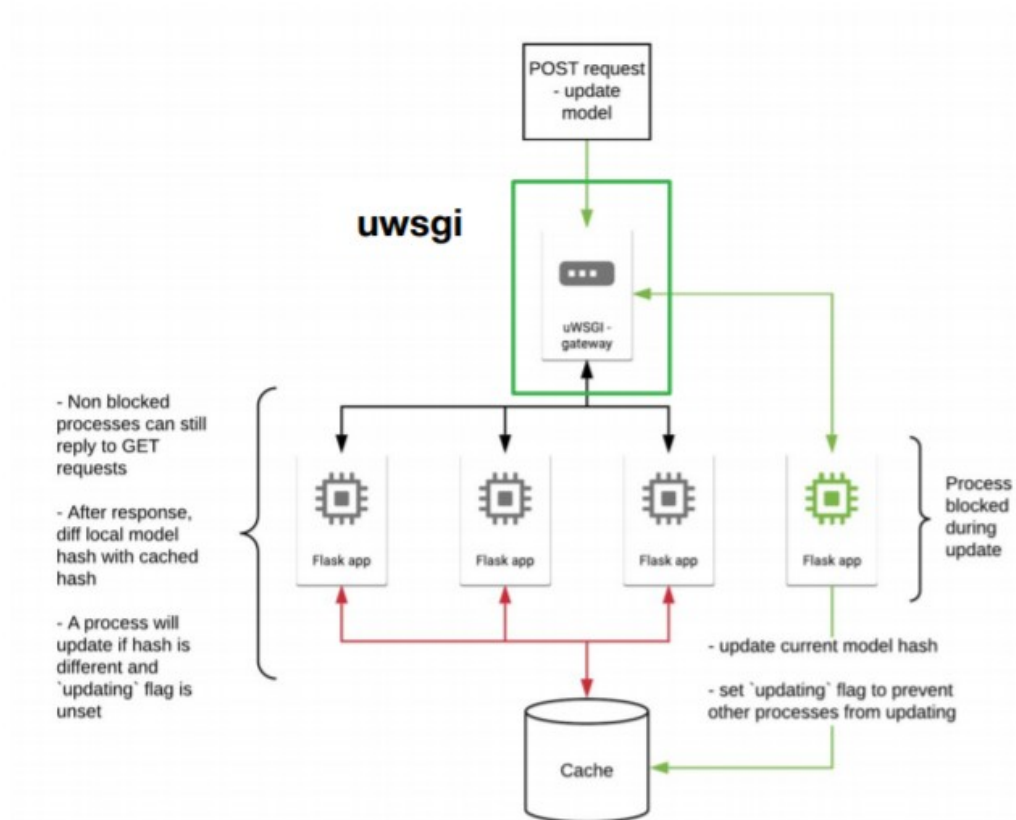
КАК РЕШИТЬ ПРОБЛЕМУ?

Хитрость заключается в том, что, раз один процесс может выполнять только один тред на одном ядре, мы просто запустим несколько процессов, чтобы они работали **параллельно**.

Существует несколько специализированных веб-серверов, решающих эту задачу, и все они имеют разную степень совместимости с веб-фреймворками Python. Однако принцип работы у них один: вы запускаете команду старта веб-сервера и передаёте в неё в качестве параметра указание на скрипт, в котором написано ваше приложение. Эта команда запустит несколько рабочих процессов (копий главного скрипта), а сама будет заниматься **прослушиванием** входящих запросов и передавать их на обработку в один из запущенных процессов.

Если вам интересно узнать, чем специализированные веб-серверы для Python отличаются друг от друга, прочитайте эту [статью](#).

Для Flask используются веб-серверы Gunicorn, связка NGINX + uWSGI и ещё несколько менее популярных.



В этом модуле мы разберём связку NGINX + uWSGI. Для вас не составит труда самостоятельно разобраться с любым другим способом, поскольку принцип их работы одинаков.

Важно! Мы рекомендуем вам всегда выполнять все изложенные шаги при настройке сервера. Игнорируйте это предупреждение, только если вы на 100 % уверены, что многопоточность вам не понадобится.

ПРЕДВАРИТЕЛЬНАЯ ПОДГОТОВКА

Перед началом давайте зафиксируем все вводные. Наш проект будет храниться в директории `/home//web`. В этой папке есть папка `models`, в которой хранится файл с моделью. Файл с Flask-приложением будет также располагаться в этой папке и будет называться `server.py`. Здесь же будет лежать файл с кодом клиентской части `client.py`.

```
└web
  └models
    └model.pkl
    └client.py
    └server.py
```

Веб-приложение будет работать локально на порте с номером 5000. У него будет два эндпоинта ('/' и '/predict'), каждому из которых соответствует свой интерфейс (функция, обернутая в декоратор `app.route`):

- Первый интерфейс будет соответствовать эндпоинту по умолчанию ('/') — <http://localhost:5000>. Он поможет нам в тестировании приложения. В результате выполнения запроса по данному адресу должно быть выведено сообщение о том, что сервер успешно работает.
- Второй интерфейс будет соответствовать эндпоинту <http://localhost:5000/predict>. Он предназначен для обработки поступающих POST-запросов. Это тот самый обработчик, который вы должны были написать в задании 7.6 — вставьте свой код на место пропущенной части.

Файл `server.py`:

```
from flask import Flask, request, jsonify
import pickle
import numpy as np

# загружаем модель из файла
with open('models/model.pkl', 'rb') as pkl_file:
    model = pickle.load(pkl_file)

# создаём приложение
app = Flask(__name__)

@app.route('/')
def index():
    msg = "Test message. The server is running"
    return msg

@app.route('/predict', methods=['POST'])
def predict():
    #ваш код здесь
    pass

if __name__ == '__main__':
    app.run('localhost', 5000)
```

Для тестирования выполнения POST-запросов на сервере, которые возвращают предсказания модели, будем использовать следующий клиентский скрипт:

Файл `client.py`:

```
import requests

if __name__ == '__main__':
    # выполняем POST-запрос на сервер по эндпоинту add с параметром json
    r = requests.post('http://localhost:5000/predict', json=[1, 0, 1,
```

```
24])
    # выводим статус запроса
    print('Status code: {}'.format(r.status_code))
    # реализуем обработку результата
    if r.status_code == 200:
        # если запрос выполнен успешно (код обработки=200),
        # выводим результат на экран
        print('Prediction: {}'.format(r.json()['prediction']))
    else:
        # если запрос завершён с кодом, отличным от 200,
        # выводим содержимое ответа
        print(r.text)
```

Предварительно протестируйте приложение, запустив файл `server.py` и перейдя в браузере по адресу <http://localhost:5000>, а также попробовав выполнить клиентский скрипт, формирующий POST-запрос.

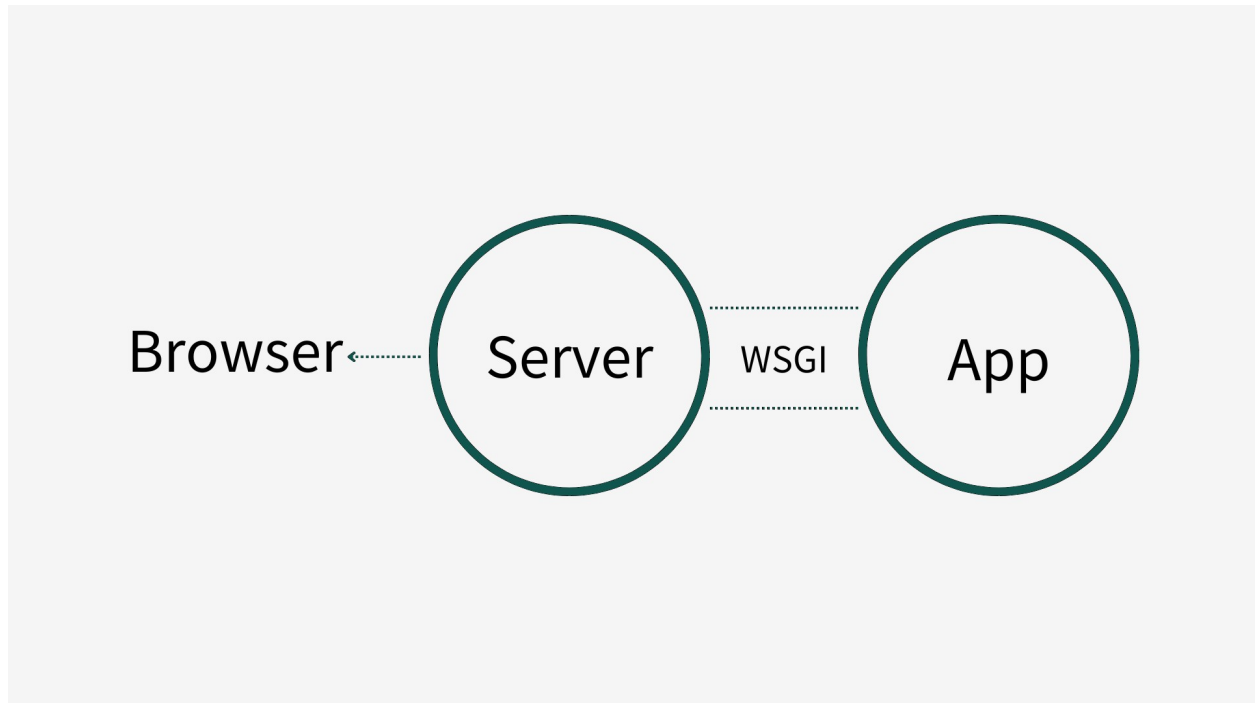
Если всё работает корректно, можно приступать к работе с uWSGI и NGINX.

WSGI И UWSGI

Давайте разберём основную терминологию.

Flask-приложению нужен интерфейс, который будет запускать приложение в нескольких процессах, обрабатывать поступающие запросы и возвращать пользователю ответ.

Для выполнения этих задач был разработан **WSGI (Web-Server Gateway Interface)** — протокол, стандарт взаимодействия между Python-программами, выполняющимися на веб-сервере и самим веб-сервером. Проще говоря, этот стандарт регламентирует, как Python должен обрабатывать запросы от приложений к веб-серверам и обратно.



Во взаимодействии, организованном по WSGI-протоколу, участвуют два компонента:

- Веб-сервер, который принимает поступающие запросы (uWSGI, Nginx, Apache и т. д.). На схеме выше обозначен как Server.
- Веб-приложение, написанное на языке Python. На схеме выше обозначено как App.

uWSGI — это веб-сервер, который запускает Python-приложения через протокол WSGI. У этого веб-сервера есть и специальная версия протокола WSGI, которая называется **uwsgi**.

uwsgi (не путать с uWSGI!) — это бинарная оптимизированная версия протокола WSGI, которая позволяет достичь ещё большей скорости взаимодействия между сервером и приложением.

УСТАНОВКА UWSGI

Чтобы запустить сервис в настоящем, боевом режиме, установим веб-сервер uWSGI. Это можно сделать с помощью стандартного менеджера пакетов Python `pip`.

В UNIX-системах (Ubuntu/Mac OS) установка производится довольно легко через выполнение перечисленных ниже команд.

Для Ubuntu:

```
$ sudo apt-get install build-essential python-dev
$ pip install uwsgi
```

Для MacOS:

```
$ brew install build-essential python-dev
$ pip install uwsgi
```

Чтобы проверить, что установка прошла успешно, выполните в терминале команду:

```
$ uwsgi --version
```

Если с установкой возникли проблемы, загляните в [инструкцию](#).

Запустим сервис (файл сервера `server.py`, который мы писали в прошлых уроках) с помощью веб-сервера uWSGI. Это довольно просто:

```
$ uwsgi --http :5000 --module server:app
```

Разберём аргументы.

- Аргумент `http :5000` указывает, что uWSGI должен обрабатывать HTTP-запросы на порте 5000. При необходимости перед двоеточием можно указать хост — `localhost`, `127.0.0.1` для локального запуска (он используется по умолчанию, если ничего не указано) или `0.0.0.0`, чтобы сервис был доступен по всем сетевым интерфейсам.
- Аргумент `server:app` указывает, какое приложение будет запускать uWSGI: модуль `server.py` (до двоеточия) и объект `app` внутри этого модуля (после двоеточия).

В результате выполнения команды появится большое сообщение с указанием всех параметров организованного соединения (начиная от версии uWSGI и операционной системы, на которой запущено приложение, и заканчивая номером процесса `pid`, закреплённым за запущенным приложением). Вот финальный фрагмент такого сообщения:

```
uwsgi http bound on 127.0.0.1:5000 fd 4
spawned uwsgi http 1 (pid: 201467)
uwsgi socket 0 bound to TCP address 127.0.0.1:45269 (port auto-assigned) fd 3
Python version: 3.10.7 (main, Sep 8 2022, 14:34:29) [GCC 12.2.0]
*** Python threads support is disabled. You can enable it with --enable-threads ***
Python main interpreter initialized at 0x55992390e8b0
your server socket listen backlog is limited to 100 connections
your mercy for graceful operations on workers is 60 seconds
mapped 72904 bytes (71 KB) for 1 cores
*** Operational MODE: single process ***
WSGI app 0 (mountpoint='') ready in 0 seconds on interpreter 0x55992390e8b0 pid: 201466 (default app)
*** uwsgi is running in multiple interpreter mode ***
spawned uwsgi worker 1 (and the only) (pid: 201466, cores: 1)
```

После этого uWSGI перейдёт в режим ожидания поступающих запросов — прослушивания сети.

Чтобы проверить, что ваш веб-сервер работает, обратитесь по адресу `localhost` в строке браузера, не забывая указать порт, который мы задали для прослушивания сети — <http://localhost:5000>. Вы должны увидеть сообщение об успешной работе сервера, которое мы прописали в функции `index()` при реализации приложения.

Давайте проверим количество процессов в системе, ведь нашей целью было с помощью uwsgi создать несколько процессов, каждый из которых занимается прослушиванием сети и обработкой поступающих запросов.

Для этого в Linux есть специальные инструменты, похожие на диспетчер задач в Windows, и самый продвинутый из них — утилита htop. Она позволяет отслеживать процессы в режиме реального времени, что очень полезно для администрирования серверов. Давайте установим её.

Для начала создайте ещё один терминал, не прерывая работу сервера. В новом терминале запустите следующую команду для установки:

```
$ sudo apt-get install htop
```

Теперь запустите утилиту htop, введя в терминал команду:

```
$ htop -t
```

Ключ -t означает, что процессы будут выводиться в виде дерева (от англ. tree), то есть в иерархии сверху будет находиться родительский процесс, а ниже будут располагаться те процессы, которые он породил (дочерние).

В результате выполнения данной команды будет выведена огромная таблица со всеми процессами, запущенными в системе. В этой таблице нас будут интересовать следующие поля:

- **PID** — идентификатор процесса;
- **USER** — пользователь, запустивший процесс;
- **TIME+** — время, измеренное в часах (указывает на то, сколько процесс находился в пользовательском и системном времени);
- **Command** — полная командная строка.

Если вы хотите больше узнать о команде htop и её результатах, загляните [сюда](#).

Для того чтобы отфильтровать только нужные нам процессы из полученной таблицы, мы должны нажать клавишу F4, а затем написать слово, которое мы ищем — uwsgi. В результате должно получиться примерно следующее:

```
CPU[|||||] 20.5% Tasks: 142, 494 thr, 67 kthr; 1 running
Mem[|||||] 2.05G/4.48G Load average: 0.97 0.44 0.47
Swp[|] 2.24M/2.12G Uptime: 00:43:35

[Main] [I/O]
PID USER PRI NI VIRT RES SHR S CPU% MEM% TIME+ Command
6921 andrey 20 0 102M 42832 18312 S 0.0 0.9 0:00.18 uwsgi --http :5000 --module server:app
6922 andrey 20 0 19248 9564 872 S 0.0 0.2 0:00.00 uwsgi --http :5000 --module server:app
```

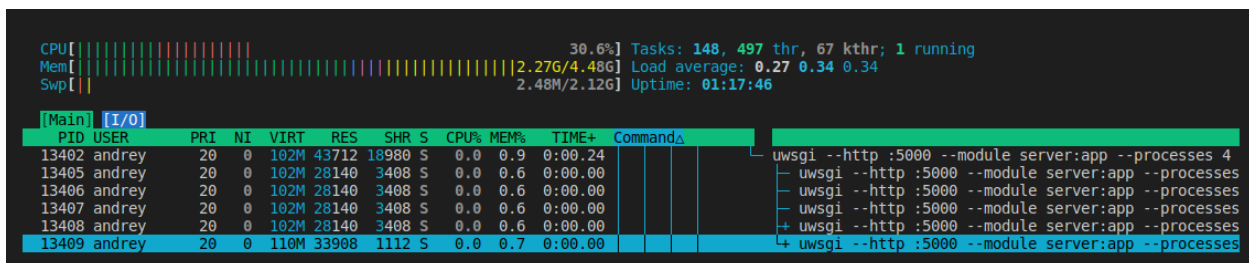
Что мы видим? У нас есть **один основной процесс** (он обозначен идентификатором 6921, но ваш идентификатор может отличаться). Этот процесс порождает один дочерний процесс 6922, который и выполняет прослушивание входящих запросов на нашем сервере.

Эта ситуация стандартная и пока что ничем не отличается от обычного запуска скрипта — работает только один процесс.

Давайте добавим больше процессов, чтобы увеличить пропускную способность сервера. Для этого завершим работу сервера (в терминале с запущенным сервером нажмите сочетание клавиш CTRL+C) и перезапустим его, увеличив количество работающих одновременно процессов до четырёх. Для этого в используемой ранее команде по запуску сервера необходимо дописать параметр `--processes`, указав количество процессов, и параметр `--master` (для чего он нужен — обсудим чуть позже):

```
$ uwsgi --http :5000 --processes 4 --master --module server:app
```

В соседнем терминале, где запущена утилита `htop`, мы увидим следующие изменения:



[Main] [I/O]										
PID	USER	PRI	NI	VIRT	RES	SHR	S	CPU%	MEM%	TIME+
13402	andrey	20	0	102M	43712	18980	S	0.0	0.9	0:00.24
13405	andrey	20	0	102M	28140	3408	S	0.0	0.6	0:00.00
13406	andrey	20	0	102M	28140	3408	S	0.0	0.6	0:00.00
13407	andrey	20	0	102M	28140	3408	S	0.0	0.6	0:00.00
13408	andrey	20	0	102M	28140	3408	S	0.0	0.6	0:00.00
13409	andrey	20	0	110M	33908	1112	S	0.0	0.7	0:00.00

Command	
uwsgi --http :5000 --module server:app --processes 4	
uwsgi --http :5000 --module server:app --processes	
uwsgi --http :5000 --module server:app --processes	
uwsgi --http :5000 --module server:app --processes	
uwsgi --http :5000 --module server:app --processes	
uwsgi --http :5000 --module server:app --processes	

Вновь есть главный (родительский) мастер-процесс с идентификатором 13402, который в свою очередь породил пять дочерних процессов. Четыре из пяти дочерних процессов рабочие — они занимаются прослушиванием сети, а о том, занимается пятый процесс, мы поговорим ниже.

Таким образом, мы увеличили число рабочих процессов до четырёх. Сделайте несколько запросов к своему веб-сервису и посмотрите на количество Python-процессов в системе.

Итак, мы **увеличили** пропускную способность нашего сервиса.

Обратите внимание! Это не значит, что сами запросы будут обрабатываться быстрее — в этом смысле код не изменился, и на его выполнение тратится столько же времени. Зато мы получили возможность параллельно обрабатывать до четырёх запросов.

Отметим ещё одну возможность, которая позволит облегчить работу с uWSGI. Ранее мы запускали веб-сервер, указывая параметры непосредственно в команде для его запуска. Однако когда параметров становится слишком много, то каждый раз набирать в терминале команду для запуска сервера становится проблематичным. В таком случае лучше определить конфигурацию в отдельном файле. Давайте назовём этот файл `"uwsgi.ini"` и разместим его в директории проекта.

Содержимое этого файла похоже на обычный код: слева от `"="` пишется имя параметра, а справа от него — значение.

В начало файла добавим заголовок `[uwsgi]`, чтобы указать uWSGI на необходимость применения настроек, а далее перечислим нужные нам параметры:

Файл `uwsgi_config.ini`

```
[uwsgi]
module = server:app

http = 127.0.0.1:5000

processes = 4
master = true
```

Теперь, чтобы запустить uWSGI с указанной в файле конфигурацией, необходимо просто передать путь до файла в параметре `--ini`:

```
$ uwsgi --ini uwsgi_config.ini
```

В результате выполнения данной команды в терминале вы увидите то же самое сообщение, что и ранее при запуске uWSGI.

Но и это ещё не всё. Как мы уже отметили, на скриншотах, полученных с помощью `htop` и после запуска `uwsgi`, видно, что процессов не четыре, а пять. Откуда взялся ещё один? Этот последний процесс является HTTP-роутером, который слушает HTTP-трафик и отправляет его на воркеры (рабочие процессы), то есть он выступает в роли пункта-распределителя.

Однако uWSGI — это не самая оптимальная реализация HTTP-роутера. Она существует скорее для отладки веб-разработчиками, что не является нашей зоной компетенции, поэтому сами авторы uWSGI советуют использовать полноценный веб-сервер. И тут появляется NGINX.

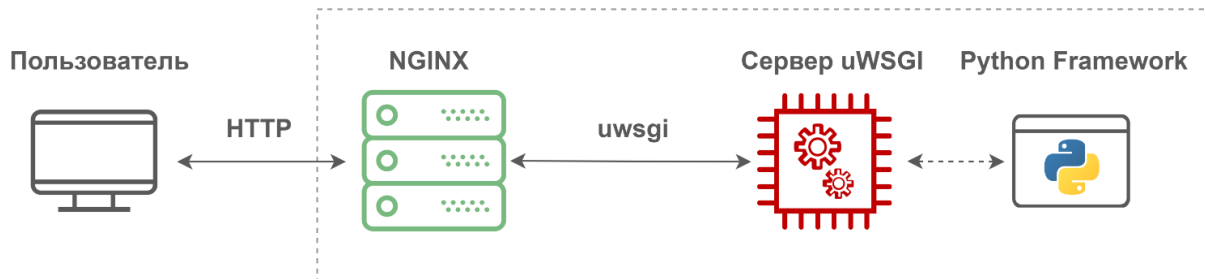
NGINX

NGINX — это веб-сервер, который оптимизирует нагрузку за счёт **асинхронной архитектуры**, управляемой событиями. Говоря простым языком, архитектура использует одно ядро на один процесс. При этом в одном процессе могут быть сотни тысяч входящих запросов от каждого пользователя. Совместная параллельная обработка позволяет не создавать новые потоки для каждого соединения — это быстро и удобно. Такая высокая производительность делает NGINX самым популярным веб-сервером в мире.

На сегодняшний день NGINX поддерживает протокол `uwsgi`, что позволяет приложениям WSGI, запущенным на uWSGI, лучше взаимодействовать с NGINX. Благодаря этому развёртывание приложений становится очень простым в настройке, чрезвычайно гибким, функциональным, а также может пользоваться преимуществами предоставляемых оптимизаций. Всё это делает сочетание uWSGI + NGINX великолепным вариантом для многих сценариев развёртывания.

ЧТО МЫ ХОТИМ УВИДЕТЬ В ИТОГЕ?

Мы хотим, чтобы входящие HTTP-запросы, которые поступают на наш сервис, принимал NGINX, так как у него очень высокая производительность. Далее необходимо, чтобы поступающие запросы по протоколу `uwsgi` передавались на uWSGI, в котором запущены процессы-воркеры, выполняющие обработку запросов на Flask.



Для того чтобы подключить сервер NGINX к нашему приложению, сначала нужно его установить.

УСТАНОВКА NGINX

В Linux-системах установка NGINX происходит с помощью одной команды в терминале:

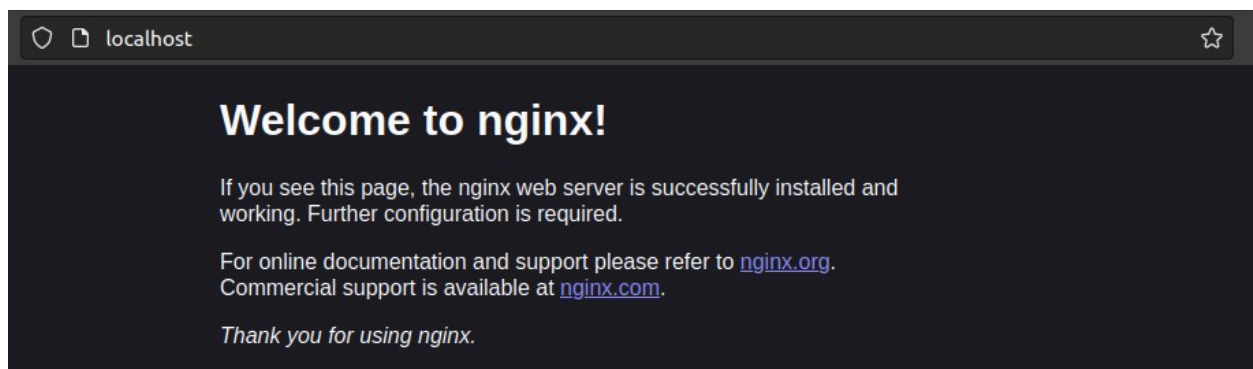
```
$ sudo apt install nginx
```

Для MacOS установка NGINX идентична установке в Linux с той лишь разницей, что вместо менеджера пакетов apt используется homebrew:

```
$ brew install nginx
```

Существует и версия для Windows, но некоторые важные функции не поддерживаются в Windows, что лишает веб-сервер значительной части производительности, и его использование под этой ОС как таковое вызывает сомнения.

После установки сервер NGINX запускается автоматически. Чтобы проверить, что установка была выполнена успешно, достаточно зайти в браузер и в поисковой строке написать <http://localhost> (без указания порта). В результате вы должны увидеть примерно следующий результат:



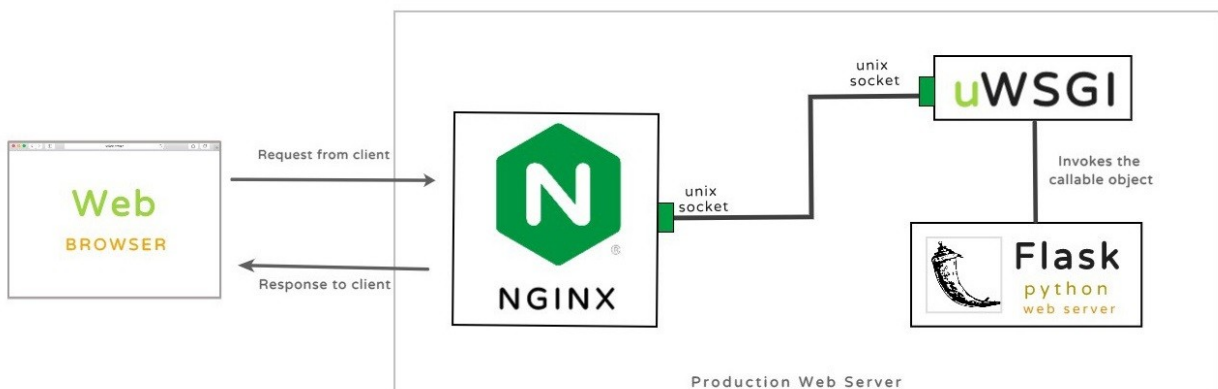
Также можно отследить, что процесс nginx запущен в системе. Для этого опять же воспользуемся утилитой htop и с помощью фильтрации таблицы с процессами (клавиша F4) найдём нужный нам процесс:

```
CPU[|||||] 14.7% Tasks: 157, 652 thr, 71 kthr; 1 running
Mem[|||||] 2.56G/4.48G Load average: 0.45 0.47 0.42
Swp[||] 23.5M/2.12G Uptime: 01:39:16

[Main] [I/O]
  PID USER   PRI  NI  VIRT   RES   SHR  S  CPU% MEM%   TIME+  Command
 19091 root     20    0 56328 11708 10096 S   0.0  0.2   0:00.00 nginx: master process /usr/sbin/nginx -g daemon on; m
 19094 www-data 20    0 56964 5476 3536 S   0.0  0.1   0:00.00 nginx: worker process
```

Теперь давайте свяжем uWSGI и NGINX. Чтобы это сделать, нам нужно запустить uWSGI без роутера — место роутера должен занять NGINX, который будет пересылать поступающий трафик, а uWSGI будет слушать трафик напрямую из **сокета**.

Сокет — это программный интерфейс для взаимодействия между двумя процессами (в нашем случае между NGINX и uWSGI). Можно представить себе сокет как специальный файл в памяти, в который один процесс пишет, а другой из него читает. Тогда наша схема организации взаимодействия между клиентами и веб-приложением будет выглядеть следующим образом:



Примечание. Если вы всё ещё на Windows, то, к сожалению, сокетов у вас нет и придётся использовать один из TCP-портов по [этой инструкции](#).

Итак, чтобы запустить взаимодействие через сокеты, нам достаточно лишь изменить параметр `http` на `socket` и указать в нём путь до файла-сокета, через который мы хотим организовать обмен информацией.

Файл с названием `socket` и расширением `.sock` будет находиться в специальной директории `/tmp`, которая в UNIX-системах предназначена для хранения временных объектов (`/tmp` — от англ. “temporary” — «временный»). Создавать его самим не нужно — за нас это автоматически сделает uWSGI при запуске.

Также мы выполним очистку сокета после остановки процесса uWSGI. Для этого будем использовать параметр `vacuum` — благодаря этой опции файл будет самостоятельно удаляться после отключения сервера. Отредактируем наш файл с конфигурацией uWSGI (`uwsgi_config.ini`):

Файл `uwsgi_config.ini`

```
[uwsgi]
module = server:app
```

```
socket = /tmp/socket.sock
vacuum = true

processes = 4
master = true
```

Теперь мы готовы перезапустить uWSGI:

```
$ uwsgi --ini uwsgi_config.ini
```

Убедитесь, что в директории /tmp был создан файл socket.sock. Для этого в соседнем терминале выполните команду:

```
$ ls /tmp/socket.sock
```

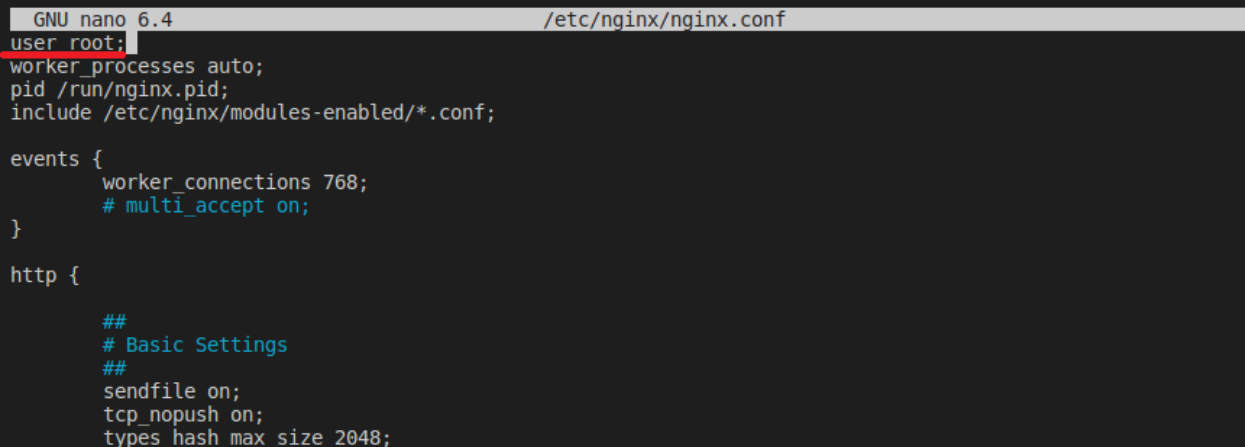
Если команда была выполнена без ошибок, то файл существует.

Последний большой шаг — настроить NGINX на передачу приходящих HTTP-запросов в созданный сокет-файл. Для этого нам нужно сначала немного изменить глобальную конфигурацию NGINX, а именно дать ему права на запуск от имени суперпользователя (root). Так NGINX сможет иметь доступ на чтение, изменение и исполнение всех файлов в системе, в том числе для файлов сокетов.

Открываем во встроенном редакторе nano (или любом другом редакторе) файл с конфигурацией nginx.conf — он находится в директории /etc/nginx:

```
$ sudo nano /etc/nginx/nginx.conf
```

В этом файле находим строку user и изменяем имя пользователя на root:



```
GNU nano 6.4 /etc/nginx/nginx.conf
user root;
worker_processes auto;
pid /run/nginx.pid;
include /etc/nginx/modules-enabled/*.conf;

events {
    worker_connections 768;
    # multi_accept on;
}

http {

    ##
    # Basic Settings
    ##
    sendfile on;
    tcp_nopush on;
    types_hash_max_size 2048;
```

Сохраняем изменения в файле (в nano — CTRL+S) и закрываем его (в nano — CTRL+X).

Формально у NGINX есть только один файл конфигурации — тот, который мы только отредактировали. Теоретически конфигурации для каждого приложения можно было бы прописывать именно в нём. Однако так делать не рекомендуется. Для этого уже существуют папки /etc/nginx/sites-available/ и /etc/nginx/sites-enabled.

- Первая просто содержит файлы конфигурации, в каждом из которых находится отдельный виртуальный хост.
- Вторая папка содержит ссылки на файлы из sites-available и подключена к основному конфигурационному файлу.

Чтобы составить конфигурацию nginx для нового приложения, мы сначала должны создать файл конфигурации в папке sites-available, а затем создать ссылку на него в папке sites-enabled. В результате таких неочевидных, но очень важных для безопасности манипуляций мы добавим в глобальную конфигурацию работы NGINX конфигурацию, нужную для нашего приложения.

Итак, создадим и откроем на редактирование файл server.conf в папке sites-available:

```
$ sudo nano /etc/nginx/sites-available/server.conf
```

Затем введём в него следующее содержимое:

Файл /etc/nginx/sites-available/server.conf

```
server {  
    listen 80;  
    server_name localhost;  
    location / {  
        uwsgi_pass unix:/tmp/socket.sock;  
        include uwsgi_params;  
    }  
}
```

Давайте разберёмся, что мы тут прописали. Структура файла очень напоминает JSON-формат. Корневой заголовок server указывает на то, что мы создаём конфигурацию нового сервера NGINX.

Внутри этого блока через ";" указываются следующие поля:

- listen — номер порта, который будет прослушивать сервер NGINX. По умолчанию протокол HTTP работает на порте 80. То есть по умолчанию обращение через браузер по любому IP-адресу (например, <http://198.102.103.8>) — это на самом деле обращение к нему по порту 80 (например, <http://198.102.103.8:80>). В параметре listen мы указали, что наш сервер NGINX будет слушать адрес 127.0.0.1 (адрес внутренней петли) на порте 80. То есть для обращения к нему будет достаточно ввести в поисковую строку браузера 127.0.0.1 или localhost без указания хоста. Если указать, другой порт, то в адресной строке браузера его также придётся прописывать.
- server_name — доменное имя. Так как у нас нет зарегистрированного доменного имени, то в качестве него мы указываем localhost.

- В блоке location мы указываем эндпоинт (по умолчанию всегда обращаемся к корневому эндпоинту "/"), а также параметры, соответствующие этому блоку:
 - uwsgi_pass — команда, указывающая на то, что поступающие запросы будут передаваться на uwsgi. В аргументах этой команды мы отметили, что взаимодействие организовано через UNIX-сокеты, и указали путь до сокета (/tmp/socket.sock), по которому происходит обмен данными между NGINX и uWSGI.
 - include — команда для подключения файлов. Мы подключаем файл uwsgi_params, в котором указаны общие параметры работы uWSGI.

После завершения редактирования файл server.conf сохраняем его содержимое (CTRL+S) и выходим из редактора (CTRL+X).

Когда файл с конфигурацией готов, создаём на него символическую ссылку (аналог ярлыков в Windows) в папке sites-enabled. В Linux за это отвечает команда ln. Её синтаксис выглядит следующим образом:

```
ln опции файл_источник файл_ссылки
```

Для создания символических ссылок используется ключ -s.

Подробнее о команде ln вы можете прочитать [здесь](#).

Тогда команда в терминале будет выглядеть следующим образом:

```
$ sudo ln -s /etc/nginx/sites-available/server.conf /etc/nginx/sites-enabled
```

Теперь наша конфигурация активирована.

Проверим, не было ли обнаружено ошибок при сборке конфигурации. Для этого используется команда nginx с ключом -t (test).

```
$ sudo nginx -t
```

Если ошибок не будет обнаружено, перезапустим процесс nginx для чтения новой конфигурации и для управления процессами (запуска/остановки и т. д.). Для этого в UNIX-системах есть утилита systemctl:

```
$ sudo systemctl restart nginx
```

Чтобы проверить результат, откройте браузер и сделайте запрос <http://localhost>. Если всё прошло успешно и в браузере вам было выведено отладочное сообщение, поздравляем — вы успешно запустили NGINX и связали его работу с uWSGI.

Обратите внимание, что для доступа к серверу нам теперь не нужно указывать номер порта, ведь мы запустили NGINX на порте 80 (указали это в файле server.conf), а он используется по умолчанию.

Что происходит?

При поступлении запроса на порт 80 NGINX транслирует его в обозначенный сокет по протоколу uwsgi. Процесс uWSGI, подключенный к этому сокету, обработает его на одном из запущенных процессов с нашим кодом, и ответ уйдёт тем же путём в обратной последовательности.

Таким образом, наш сервис стал **производительнее**, ведь он может параллельно обрабатывать несколько запросов.

А что дальше?

Мы проделали большую работу. Однако, как вы могли заметить, наш веб-сервис сейчас работает только локально, то есть доступ к нему можно получить только с вашего компьютера. В рамках нашего курса заводить полноценный веб-сервис, доступный в интернете, нам не нужно — нам было достаточно лишь прикоснуться к веб-программированию и потренироваться разворачивать собственный веб-сервис на локальной машине, поэтому мы смело можем закончить на этом этапе.

В реальности, как вы понимаете, доступ к сервису должен быть у всех компьютеров. Для того чтобы организовать возможность обращения к вашему серверу через интернет, вам нужно получить доменное имя (купить или получить [бесплатно](#)), привязанное к IP-адресу вашего сервера. После этого в конфигурации NGINX вашего веб-сервера (server.conf) нужно будет заменить параметр server_name с localhost на доменное имя или IP-адрес вашего сервера:

```
server_name your_domain www.your_domain;
```

Так ваш сервер сможет обрабатывать запросы, поступающие от внешних пользователей.

9. Итоги

☆ Поздравляем с прохождением первого модуля, посвящённого машинному обучению в продакшене!

В ЭТОМ МОДУЛЕ ВЫ:

Узнали:

- как сохранять обученные модели в файлы и обмениваться ими;
- как использовать обученные модели в приложениях, реализованных на языках программирования, отличных от Python;
- основные концепции межсетевого взаимодействия;
- несколько самых популярных веб-фреймворков Python и сравнили их между собой;
- как решать проблему GIL для Python с помощью протокола uwsgi;

Научились:

- отличать сериализацию от десериализации;

- реализовывать простейшие веб-сервисы на Flask;
- разворачивать свой сервер на NGINX.

Стоит отметить, что в процессе разработки нашего сервиса мы не учли одну очень важную особенность — воспроизводимость и изолированность. В чём сейчас состоит проблема? Когда мы передадим исходный код своему коллеге Василию, может оказаться так, что он у него попросту не запустится. Этому может быть множество причин: разница в версиях библиотек, в установленных путях до файлов, в конце концов — в операционных системах. Как же тогда быть?

Забегая вперёд, скажем, что для поддержки принципа воспроизводимости кода используются изоляция, виртуализация и контейнеризация. О них мы как раз и поговорим в следующем модуле.

Дополнительные материалы:

- [документация по работе с Pickle](#);
- [документация по работе с Joblib](#);
- [список фреймворков для разработки HTTP-серверов на странице](#);
- [обновляемый список ресурсов и плагинов для Flask](#);
- [описание различных уровней взаимодействия сетей по сетевой модели OSI](#);
- [исчерпывающий список кодов состояния протокола HTTP](#);
- [подробное описание REST-архитектуры](#);
- [Postman \(для любителей визуальных редакторов запросов\)](#).