[***Python*** 3](#_Toc131957288)

[**Интерпретатор *Python*** 3](#_Toc131957289)

[**Иммутабельность типов данных** 3](#_Toc131957290)

[***if \_\_name\_\_ == ‘\_\_main\_\_’*** 3](#_Toc131957291)

[***List/dict/set comprehension*** 3](#_Toc131957292)

[**Контекстный менеджер** 4](#_Toc131957293)

[**Отличия кортежа от списка** 4](#_Toc131957294)

[**Различия методов *type()* и *isinstance()*** 4](#_Toc131957295)

[**Магические атрибуты и методы (*\_\_slots\_\_, \_\_new\_\_* и т.д.)** 5](#_Toc131957296)

[**Различия *\_\_new\_\_* и *\_\_init\_\_*** 6](#_Toc131957297)

[**Протокол итерации (итерируемый объект)** 6](#_Toc131957298)

[**Итераторы** 6](#_Toc131957299)

[**Генераторы** 7](#_Toc131957300)

[**Отличия генераторов от итераторов** 9](#_Toc131957301)

[***\*args* и *\*\*kwargs*** 9](#_Toc131957302)

[**Виртуальное окружение** 10](#_Toc131957303)

[***GIL (Global Interpreter Lock)*** 10](#_Toc131957304)

[**Сборщик мусора *Python*** 11](#_Toc131957305)

[**Отличия операторов *is* и *= =*** 11](#_Toc131957306)

[***asyncio*** 12](#_Toc131957307)

[**Индексатор класса** 12](#_Toc131957308)

[**Поверхностное и глубокое виды копирования объектов** 13](#_Toc131957309)

[**Пакеты и модули** 13](#_Toc131957310)

[**Чистая функция** 14](#_Toc131957311)

[**Именованные кортежи (*namedtuple*)** 14](#_Toc131957312)

[**Миксины** 15](#_Toc131957313)

[***MRO* (порядок разрешения методов)** 15](#_Toc131957314)

[***classmethod, staticmethod*** 17](#_Toc131957315)

[**Абстрактный класс** 17](#_Toc131957316)

[**Метаклассы** 17](#_Toc131957317)

[**Хеш-таблицы (словари) в *Python*** 19](#_Toc131957318)

[**Множества в *Python*** 19](#_Toc131957319)

[***Django*** 20](#_Toc131957320)

[***Model-View-Template* (*MVT*, архитектурный подход *Django*)** 20](#_Toc131957321)

[**Оптимизация запросов (*select\_related*, *prefetch\_related*)** 20](#_Toc131957322)

[**Конкурентность** 21](#_Toc131957323)

[***Middleware*** 21](#_Toc131957324)

[**Наследование моделей** 22](#_Toc131957325)

[***WSGI и ASGI*** 22](#_Toc131957326)

[***ORM*** 23](#_Toc131957327)

[**Базы данных** 25](#_Toc131957328)

[**Схемы** 25](#_Toc131957329)

[**Виртуальное представление** 25](#_Toc131957330)

[**Материализованное представление** 25](#_Toc131957331)

[**Виды соединений таблиц** 26](#_Toc131957332)

[**Индексы** 26](#_Toc131957333)

[**Оптимизация запросов** 28](#_Toc131957334)

[**Масштабирование баз данных** 31](#_Toc131957335)

[***ACID* (свойства транзакционной СУБД)** 31](#_Toc131957336)

[**Уровни изолированности транзакций** 32](#_Toc131957337)

[***CAP*-теорема** 33](#_Toc131957338)

[**Репликация** 33](#_Toc131957339)

[**Денормализация** 35](#_Toc131957340)

[***COMMON*** 36](#_Toc131957341)

[***HTTP*** 36](#_Toc131957342)

[**Идемпотентность *HTTP* методов** 37](#_Toc131957343)

[***REST*/*RESTful*** 38](#_Toc131957344)

[**Как браузер обрабатывает запрос клиента** 39](#_Toc131957345)

[***SOLID*** 39](#_Toc131957346)

[***CI/CD*** 40](#_Toc131957347)

[***JSON Web Tokens (JWT)*** 41](#_Toc131957348)

[***Docker*** 42](#_Toc131957349)

[**Разница между *CMD* и *ENTRYPOINT*** 42](#_Toc131957350)

***Python***

## **Интерпретатор *Python***

Python является интерпретируемым языком. Интерпретатор переводит исходный код в платформо-независимый байт-код (а не в машинный), байт-код выполняется на виртуальной машине.

## **Иммутабельность типов данных**

Тип переменной *a* **иммутабелен**, если после перезаписи значения переменной изменился ее идентификатор (*id(a)*). Изменение идентификатора указывает на то, что после перезаписи значения создался новый объект, следовательно, старый объект нельзя было изменить.

## ***if \_\_name\_\_ == ‘\_\_main\_\_’***

Данное условие позволяет определить точку входа. По идее это делается для того, чтобы при импорте различных модулей их код не вызывался в текущем модуле (операция импорта инициирует выполнение кода импортируемого модуля). С проверкой внутри каждого из импортируемых модулей в текущем модуле вызовется только код текущего модуля.

Наглядный пример реализован в учебном проекте.

## ***List/dict/set comprehension***

***Comprehension*** – *pythonic* способ генерирования создания списков, множеств и словарей.

**Примеры**

l = [i \* i for i in range(10) if i % 2 == 0]

print(type(l))

d = {k: k + 5 for k in range(5)}

print(type(d))

s = {k for k in range(5)}

print(type(s))

>> <class 'list'>

>> <class 'dict'>

>> <class 'set'>

## **Контекстный менеджер**

Контекстный менеджер – конструкция вида

with SomeClass() as obj:

obj.do\_something()

позволяющая при необходимости высвободить ресурсы некоторого объекта или контекста.

Реализация контекстного менеджера предполагает один из способов:

* реализация методов \_\_*enter*\_\_ и \_\_*exit*\_\_.
* использование модуля *contextlib* (через декоратор *contextmanager*) и *yield*.

Пример есть в учебном проекте.

## **Отличия кортежа от списка**

Cписок (*list*) является ***mutable*** типом, а кортеж (*tuple*) ***immutable***.

## **Различия методов *type()* и *isinstance()***

*type* возвращает тип данных аргумента, а *isinstance* **булево значение**, говорящее о том, принадлежит ли объект определенному типу:

num = 4.44

print(isinstance(num, float))

print(type(num))

>> True

>> <class 'float'>

Также отличие от *type* состоит в том, что *isinstance* **"знает" о наследовании**. Функция воспринимает объект производного класса, как объект базового. Поэтому **нельзя использовать** *type* **для проверок наследующихся типов**.

**Пример**

class BaseExample:

pass

class DerivedExample(BaseExample):

pass

print(isinstance(DerivedExample(), BaseExample))

>> True

## **Магические атрибуты и методы (*\_\_slots\_\_, \_\_new\_\_* и т.д.)**

**Магические методы** – методы, вызывающиеся неявно при выполнении других определенных методов. Например, в случае приведения к строке объекта некоторого класса вызывается магический метод \_\_*str*\_\_, если он определен в классе. **Наименования магических методов обрамлены с обеих сторон двумя подчеркиваниями**.

\_\_***dict***\_\_ – словарь, хранящий пользовательские атрибуты объекта и его значения.

\_\_***slots***\_\_ – атрибут, позволяющий ограничить набор атрибутов класса (может пригодиться в целях уменьшения затрат по памяти). Если в классе определен \_\_*slots*\_\_, то объекты класса станут занимать существенно меньше памяти при большом количестве атрибутов, т.к. магический атрибут \_\_*dict*\_\_не будет создаваться (слоты можно хранить в виде кортежа, кортеж займет намного меньше памяти, чем хеш-таблица). Также ускоряет доступ к значениям атрибутов.

\_\_***new***\_\_ – **метод, создающий объект** **класса**, на вход принимает класс ***cls***. После выполнения отдает объект методу *\_\_init\_\_*, который в свою очередь инициализирует атрибуты объекта.

***\_\_call\_\_*** – определение этого метода делает объект *callable* (объект можно вызвать как метод):

class A:

def \_\_call\_\_(self, msg):

print(msg)

A()('test')

>> test

## **Различия *\_\_new\_\_* и *\_\_init\_\_***

*\_\_new\_\_* **создает и возвращает объект класса (является конструктором)**, он выполняется в первую очередь. В качестве первого параметра принимает ссылку на класс (*cls*).

*\_\_init\_\_* **отвечает лишь за инициализацию объекта класса**, он выполняется после *\_\_new\_\_* **и ничего не возвращает***.* В качестве первого параметра принимает ссылку на объект (*self*).

## **Протокол итерации (итерируемый объект)**

**Итерируемый объект (*iterable*)** – это объект, возвращающий элементы по одному, а именно объект, **реализующий** методы *\_\_iter\_\_* **и** *\_\_getitem\_\_* **или** *\_\_next\_\_.*

## **Итераторы**

**Итератор** – это объект, который может возвращать элементы последовательности по одному. Технически итератором является **любой объект**, **реализующий методы:**

* *\_\_next\_\_*, который должен вернуть **следующий элемент** или исключение *StopIteration,* если перечислены все элементы;
* *\_\_iter\_\_*, возвращающий объект, который будет использован как итератор (например, объект класса, как в примере ниже).

**Пример**

class It:

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

self.l = [x for x in range(1, 11)]

self.current\_index = 0

def \_\_iter\_\_(self):

return self

def \_\_next\_\_(self):

if self.current\_index > len(self.l) - 1:

raise StopIteration

value = self.l[self.current\_index]

self.current\_index += 1

return value

g = It()

for x in g:

print(x, end=' ')

Для того, чтобы итератор снова начал возвращать элементы, его надо **заново создать**.

**Итераторы полезны** тем, что они **отдают элементы по одному**. Например, при работе с файлом или любой последовательностью это полезно тем, что **в памяти будет находиться** не вся последовательность, а **только одно текущее значение** итератора.

## **Генераторы**

Генератор – подвид итератора, хранящий закономерность вычисления последовательности. Генераторы **хранят в памяти только текущий/последний элемент последовательности**.

Генератор генерирует значения, возвращающиеся по запросу, и после возврата одного значения выполнение функции-генератора приостанавливается до запроса следующего значения. Между вызовами **генератор сохраняет свое состояние, помнит о контексте**.

*Python* позволяет **создавать** генераторы **двумя способами**:

1. **генераторное выражение**;
2. **функция-генератор**.

**Генераторное выражение** использует такой же синтаксис, как *list comprehension* (с круглыми скобками), но **возвращает генератор**, а не список:

genexpr = (x\*\*2 for x in range(1,5))

next(genexpr)

>> 1

next(genexpr)

>> 4

next(genexpr)

>> 9

**Генераторные функции** – это функции, где есть хотя бы одно выражение *yield*. При запуске генератора, функция выполняется до первого выражения *yield*. Генератор при этом встанет «на паузу» до следующей итерации. При следующей итерации выполнение генератора продолжится до очередного *yield*:

def f\_gen(m):

s = 1

for n in range(1, m):

yield n\*\*2 + s

s += 1

a = f\_gen(5)

for i in a:

print(i)

>> 2

>> 6

>> 12

>> 20

Еще одна особенность, о которой нужно помнить: в генератор можно посылать значения с помощью метода *send*:

def gen\_func():

while True:

x = yield

x += 5

print(x)

g = gen\_func()

g.send(None)

g.send(3)

g.send(6)

>> 8

>> 11

## **Отличия генераторов от итераторов**

**Итератор** – более общая концепция, чем генератор. Можно сказать, что генератор является итератором, но не наоборот.

**Итератор** предоставляет интерфейс для перебора элементов коллекций и потоков данных.

**Генератор** хранит закономерность, по которой генерируется последовательность. **Генератор хранит в памяти только текущий элемент**, а не все элементы перебираемой коллекции.

## ***\*args* и *\*\*kwargs***

***\*args*** – **кортеж позиционных аргументов** (является типом *tuple*), позволяющий передавать в функцию неопределенное количество аргументов:

def getsum(\*numbers):

sum = 0

for num in numbers:

sum += num

return sum

print(add(2, 3))

print(add(2, 3, 5))

>> 5

>> 10

***\*\*kwargs*** – **словарь именованных аргументов**. Является типом *dictionary*. Позволяет передавать параметры в произвольном порядке:

def total\_fruits(\*\*fruits):

total = 0

for amount in fruits.values():

total += amount

return total

print(total\_fruits(banana=5, mango=7, apple=8))

print(total\_fruits(banana=5, mango=7, apple=8, oranges=10))

## **Виртуальное окружение**

Виртуальное окружение помогает изолировать зависимости проекта в отдельном пространстве (вместо глобального окружения ОС), привязанном к конкретному проекту. Это позволяет не конфликтовать зависимостям разных версий при наличии множества проектов на одной машине.

Также виртуальное окружение можно использовать для хранения приватных данных, необходимых для работы проекта (*API*-ключи, различные переменные, пароли и т.д.). Для этого можно создавать **переменные окружения** и получать их значения в конфигах приложения.

## ***GIL (Global Interpreter Lock)***

*GIL* – это **мьютекс (примитив синхронизации)**, защищающий внутренние структуры данных интерпретатора от некорректного взаимодействия с несколькими потоками и запрещающий одновременную работу более чем одного потока.

*GIL* нужен для того, чтобы сделать потокобезопасными механизмы внутри сборщика мусора (подсчет ссылок).

В истории имели место некоторые попытки избавления от *GIL* (попытки обеспечения потокобезопасности*)*, но без *GIL* однопоточные приложения работают медленно, а существующие *C*-расширения пришлось бы переписывать, т.к. они изначально не потокобезопасны.

## **Сборщик мусора *Python***

Каждый объект в *Python* имеет атрибут ***ref\_count*** – **счетчик ссылок**. Когда он становится равным 0, объект сразу деаллоцируется (удаляется из памяти).

При этом оператор *del* не удаляет объект, а декрементирует счетчик ссылок.

## **Отличия операторов *is* и *= =***

Ниже приведен наглядный пример

# определим три одинаковых списка

list\_1 = ['a', 'b', 'c']

list\_2 = list\_1

list\_3 = list(list\_1)

print(list\_1) # >> ['a', 'b', 'c']

print(list\_2) # >> ['a', 'b', 'c']

print(list\_3) # >> ['a', 'b', 'c']

# list\_1 равен list\_2 и по значению и как объект, т.к. оба указывают на один и тот же адрес в памяти

# (это можно проверить оператором is или сравнить id (эквивалентные способы))

print(list\_1 == list\_2, id(list\_1) == id(list\_2), list\_1 is list\_2)

>> True True True

# list\_2 равен list\_3 только ПО ЗНАЧЕНИЮ (==). А как объекты они разные

print(list\_2 == list\_3, id(list\_2) == id(list\_3), list\_2 is list\_3)

>> True False False

# добавим элемент в list\_1 и посмотрим, что станет с list\_2

list\_1.append('n')

print(list\_1)

>> ['a', 'b', 'c', 'n']

# list\_2 также изменится, потому что является ссылкой на list\_1

print(list\_2)

>> ['a', 'b', 'c', 'n']

# а list\_3 не изменится

print(list\_3)

>> ['a', 'b', 'c']

## ***asyncio***

Модуль *asyncio* решает проблемы медленного выполнения *IO-bound* задач. Эти проблемы возникают из-за блокирующих системных вызовов.

Сокет – **абстракция сетевого взаимодействия ОС**. Сокет поддерживает операции *read* и *write*.

Асинхронность устроена примерно так: сокет периодически опрашивается, чтобы получить его состояние (можем ли мы делать операции *read*/*write* в сокете). Если сокет недоступен, контекст переключается (подобно *yield* в генераторе) и выполняются следующие задачи.

Асинхронные функции возвращают **корутины** – **разновидность генератора.** Куча таких корутин лежит в очереди, и каждая из них вытаскивается из очереди на выполнение внутри цикла событий. За счет того, что корутина не теряет контекст выполнения подобно генератору, условно ОС может выделять на кусок кода асинхронной функции квант времени, чтобы *CPU* выполнил код. А на моменте блокирования основного потока можно недовыполненную асинхронную задачу положить обратно в очередь и взять на выполнение новую задачу по той же схеме. Таким образом основной поток не будет блокироваться и ресурсы *CPU* будут максимально утилизированы.

**Важное замечание**: асинхронный код **не будет работать**, если контекст выполнения не поддерживает асинхронности (контекстом может выступать ОС, сетевые драйверы и т.д.).

## **Индексатор класса**

Нужно реализовать метод *\_\_getitem\_\_* (см. пример в учебном проекте).

## **Поверхностное и глубокое виды копирования объектов**

*copy.copy(x)* – возвращает поверхностную копию *x*.

*copy.deepcopy(x)* – возвращает глубокую копию *x*.

Разница между **поверхностным** и **глубоким** копированием существенна только для **составных** объектов, содержащих изменяемые объекты (например, список списков, или словарь, в качестве значений которого – списки или словари).

**Пример**

import copy

l = [[1], [2]]

# поверхностная копия

l\_copy = copy.copy(l)

print(f'l: {l} l\_copy: {l\_copy}')

# 100 добавится в оба списка

# т.к. поверхностное копирование не проводит копирование вложенных объектов

l\_copy[0].append(100)

print(f'l: {l} l\_copy: {l\_copy}')

# глубокая копия

l[0].remove(100)

l\_copy = copy.deepcopy(l)

l\_copy[0].append(100)

print(f'l: {l} l\_copy: {l\_copy}')

>> l: [[1], [2]] l\_copy: [[1], [2]]

>> l: [[1, 100], [2]] l\_copy: [[1, 100], [2]]

>> l: [[1], [2]] l\_copy: [[1, 100], [2]]

## **Пакеты и модули**

**Модуль** представляет собой **один файл** *Python*, который можно импортировать.

**Пакет** состоит из нескольких файлов/модулей *Python*. По сути это директория, которая может содержать поддиректории и модули. Особенностью является также то, что **пакет** **должен** **содержать пустой файл** с названием *\_\_init\_\_.py.*

## **Чистая функция**

Чистая функция – функция, удовлетворяющая следующим критериям:

1. **всегда возвращает одно** и то же **значение** **для одних и тех же** значений **входных** **параметров**;
2. **не имеет побочных эффектов** (не влияет на код, содержащийся вне функции. Например, не меняет значения глобальных переменных, не использует потоки ввода-вывода).

В общих чертах это функция, возвращаемое значение которой зависит исключительно от входных данных.

**Преимущества**:

* предсказуемость;
* легкая тестируемость.

## **Именованные кортежи (*namedtuple*)**

Именованный кортеж – это то же самое, что и обычный кортеж, но с той разницей, что теперь к элементу кортежа можно будет обращаться по имени, что улучшает читабельность кода (фактически это будет аналогом перечисления *Enum* в *C#*).

**Пример**:

from collections import namedtuple

# 1 параметр – имя типа, далее – список имен элементов кортежа

States = namedtuple('States', ['In', 'Progress', 'Open'])

states = States(1, 4)

print(states)

>> States(In\_Progress=1, Open=4)

## **Миксины**

**Миксины/классы-примеси** – концепция, облегчающая создание сложных иерархий наследования классов. Применение миксин возможно только при наличии множественного наследования в языке.

Идея заключается в том, что **класс-примесь предоставляет переиспользуемый небольшой функционал** для других сущностей, при этом желательно, чтобы миксин не наследовался от других классов. Это облегчает создание и поддержку иерархии наследования классов.

В *Python* **нет специального синтаксиса для поддержки миксинов**, поэтому визуально миксины представляют из себя обычные классы. Но главное то, что *Python* позволяет реализовывать множественное наследование, что и позволяет наследоваться целевому классу от множества классов-миксинов.

## ***MRO* (порядок разрешения методов)**

*MRO* представляет из себя **алгоритм линеаризации иерархии класса для разрешения конфликтных ситуаций при множественном наследовании**. Разумеется, что это актуально тогда, когда класс-наследник не имеет собственной реализации метода.

В старых версиях *Python* реализация *MRO* напоминала обход бинарного дерева в глубину, но это порождало множество проблем, т.к. в глубину можно было дойти до *object* (вызывались бы одноименные методы *object’а*). Теперь *MRO* работает подобно обходу в ширину.

Чтобы увидеть линейный вид иерархии и понять приоритетность каждой сущности в иерархии наследования, можно распечатать результат вызова **у класса** метода *mro*() или магического атрибута \_\_*mro*\_\_ у объекта.

Как работает *MRO* можно рассмотреть на примере с комментариями ниже. Вкратце на типовые вопросы «в каком порядке выведутся принты в коде..» можно дать ответ, что они выведутся в обратном порядке *MRO*, то есть в обратном порядке по очереди наследования в коде (см. пример ниже).

**Пример**

class A:

    def \_\_init\_\_(self):

        print('A')

class B(A):

    def \_\_init\_\_(self):

        super().\_\_init\_\_()

        print('B')

class C(A):

    def \_\_init\_\_(self):

        super().\_\_init\_\_()

        print('C')

class D(B, C):

    def \_\_init\_\_(self):

        super().\_\_init\_\_()

        print('D')

"""

вывод будет идти в обратном порядке по MRO,

т.е. в обратном порядке по очереди наследования:

D(B, C)...A. Получаем вывод A -> C -> B -> D

"""

D()

print(D.mro())

>> class A

class C

class B

class D

[<class '\_\_main\_\_.D'>, <class '\_\_main\_\_.B'>, <class '\_\_main\_\_.C'>, <class '\_\_main\_\_.A'>, <class 'object'>]

Еще один пример с механикой работы *MRO* реализован в учебном проекте.

## ***classmethod, staticmethod***

Метод класса, объявленный с декоратором ***classmethod***, принимает на вход в качестве первого параметра *cls*, и может изменять только состояние атрибутов класса. Эти атрибуты связаны с классом, а не объектом, при этом объект класса видит такие атрибуты.

Метод класса, объявленный с декоратором ***staticmethod***, является обычным статическим методом, т.е. он не имеет доступа к классу, его объектам и его состоянию. Такой метод уместно внести в класс, если он имеет какое-то отношение к классу и к контексту сущности, описанной классом.

Пример реализован в учебном проекте.

## **Абстрактный класс**

Абстрактный класс – класс, у которого нельзя создать объект.

Для реализации в *Python* нужно импортировать из модуля *abs* класс *ABC* и метод *abstractmethod*. Целевой класс нужно унаследовать от *ABC*, а абстрактные методы объявить с декоратором @*abstractmethod*.

Если у абстрактного метода не объявить вышеупомянутый декоратор, то данный метод необязательно переопределять в наследниках (работает как *virtual*-методы в *C#*). А если декоратор прописан, то метод обязательно должен быть переопределен в наследниках. Пример реализован в учебном проекте.

## **Метаклассы**

Для начала нужно понимать, что абсолютно **все сущности** в языке *Python* **являются** **объектами**, включая классы.

Для определения типа класса можно вызвать метод *type* и увидеть, что метод возвращает <class 'type'>. То есть, **любой объект в *Python* имеет тип *type***.

Можно сказать, что **метакласс – это сущность, объекты которой являются классами**. Например, *type* является метаклассом и может работать в двух режимах, в первом из которых *type* принимает 1 аргумент – тип объекта, и возвращает его базовый тип, а во втором принимает 3 аргумента:

* ***name***: название класса;
* ***base***: кортеж родительских классов (может быть пустым);
* ***attrs***: словарь, содержащий имена и значения атрибутов.

Используя *type* во втором режиме, можно динамически создавать классы с атрибутами.

Также можно создать свой метакласс, наследуясь от *type*, и в определении целевого класса указать metaclass=<имя метакласса> (пример есть в учебном проекте).

На практике с метаклассом можно столкнуться, например, создавая класс собственной модели в *Django*.

В сообществе *Python* бытует высказывание от одного эксперта, смысл которого состоит в том, что обычный программист вряд ли когда-нибудь столкнется с необходимостью написания собственных метаклассов.

Примеры приведены в учебном проекте.

## **Хеш-таблицы (словари) в *Python***

Пустая хеш-таблица выглядит как массив определенной длины, каждый элемент которого называется бакетом.

**Добавление элемента**: **для ключа** вычисляется **хеш** с помощью встроенной функции *hash*. Вычисленный хеш используется для определения бакета, например bucket\_idx = hash\_value % len(arr), где *arr* – массив бакетов. Затем в бакет кладется пара ключ-значение.

Алгоритмическая сложность добавления элемента – *O(1)*.

**Коллизии**: ситуация, при которой алгоритм хеширования выдает для разных ключей одинаковые значения, называется коллизией. Коллизии можно резолвить несколькими способами. Например, методом цепочек, в котором вместо значения в бакет кладется связный список значений. При этом, алгоритмическая сложность получения элемента может вырасти до линейной, потому что придется пробежаться по всем элементам связного списка, чтобы найти искомый элемент.

**Получение элемента по ключу**: для ключа вычисляется хеш, определяется бакет и из него берется значение.

Алгоритмическая сложность получения любого значения по ключу составляет *O(1)* или постоянное время, равное времени работы алгоритма хеширования.

Хеш-таблицы занимают много памяти, т.к. необходимо выделять память под количество бакетов примерно равное количеству элементов.

**Ключами словаря** в *Python* **могут являться только хешируемые объекты** (**значения неизменяемых типов данных**).

## **Множества в *Python***

Множество в *Python* – структура неупорядоченных данных, основанная на хеш-таблице. Особенности множества:

* элементы множества не индексируются;
* множество не содержит дубликаты элементов;
* множество содержит только значения неизменяемых типов данных.

***Django***

## ***Model-View-Template* (*MVT*, архитектурный подход *Django*)**

*MVT* включает в себя три составляющие (не путать с аналогичными в модели *MVC*):

* ***M*** – модели, описывающие сущности приложения. Представляют собой классы, соответствующие таблица в БД;
* ***V*** – представления (или контроллеры из *MVC*). Данный слой связывает слои *Model* и *Template*. Конкретное представление должно быть связано с соответствующим маршрутом (*url*), при необходимости оно может обращаться к слою *Model*, также представление решает какой темплейт нужно отрендерить в браузере по конкретному запросу.
* ***T*** – темплейты, представляющие визуальный слой приложения *HTML*-страницы, фронтенд).

*Django* не предполагает наличия отдельного слоя бизнес-логики, поэтому этот слой лучше реализовать самостоятельно в виде отдельных пакетов и модулей.

## **Оптимизация запросов (*select\_related*, *prefetch\_related*)**

Рассмотрим пример: есть две связанные модели – **жанр** и **книга**, связь *1-N*. Есть страница**,** выводящая информацию об *N* книгах из БД по определенному жанру. На странице мы хотим видеть название жанра около каждой книги.

По умолчанию приложение отправило бы *N* запросов к таблице жанров для получения названия жанра, что очень неоптимально.

Эту проблему и решает *select\_related*. С ним мы делаем 1 запрос и используем информацию из него *N* раз, а без него делаем *N* запросов. **Его можно использовать с моделями, не связанными связью *N-N***.

*prefetch\_related* нужен в той же ситуации, что и *select\_related*, но в случае наличия связи *N-N.*

Влияние на количество запросов от данных методов наглядно видно в секции *SQL DebugToolbar.*

## **Конкурентность**

Средства *Django* для решения проблем с *concurrency*:

* ***F***-**выражение** – необходимо для корректного хранения значения из БД в актуальном виде;
* **декоратор**@transaction.atomic, предоставляющий возможность ручным управлением транзакциями. Когда запрос прилетает во *view*, обернутую этим декоратором, БД открывает транзакцию, и если запрос будет корректно обработан, в конце произойдет **коммит**. В случае возникновения исключений транзакция **откатится** (*db* *rollback*);
* **метод** select\_for\_update, блокирующий изменяемую строку**.**

**Пример** кода с использованием вышеописанных сущностей фреймворка:

**@transaction.atomic**

def buy\_something(request):

request.user.**update\_balance**(money)

return render(request, template, data)

class Profile(models.Model):

def **update\_balance**(self, balance):

Profile.objects.**select\_for\_update**().\

filter(pk=self.pk)\

.update(balance=**F**('balance') + balance)

## ***Middleware***

*Middleware* – **промежуточный программный слой**, позволяющий обрабатывать запросы из браузера, прежде чем они достигнут слоя представления *Django*, а также ответы от представлений до того как они попадут в браузер.

Путь запроса от браузера до *view*: ***Browser – M1 – M2 … M..N – View***.

Путь запроса от *view* до браузера: ***View – M..N … M2 – M1 – Browser***.

## **Наследование моделей**

*Django* позволяет реализовать 4 вида наследования: **абстрактная модель, мультитабличное наследование, модель-прокси** и **множественное наследование**.

**Абстрактная модель** позволяет разместить внутри себя некоторый общий код и использовать его в моделях-наследниках. **Таблицы в БД создаются ТОЛЬКО для моделей-наследников**. Данный вид наследования реализуется с помощью указания в подклассе *Meta* атрибута *abstract* = *True*.

**Прямое (мультитабличное) наследование** позволяет создать полноценную иерархию наследования, в рамках которой д**ля ВСЕХ моделей будет создана таблица** **в БД**. Между моделями-наследниками и родительской моделью создается неявная связь вида *OneToOneField*. **Данный вид наследования выглядит как обычное наследование классов в** *Python*.

**Модель-прокси** позволяет дополнить поведение родительской модели, но не ее состав полей. Для этого в подклассе *Meta* дочерней модели нужно указать атрибут *proxy = True*, чтобы эта модель стала прокси. **Для прокси-модели таблицы в БД НЕ СОЗДАЮТСЯ**.

**Множественное наследование** моделей реализуется как множественное наследование обычных классов *Python*.

## ***WSGI и ASGI***

***WSGI*** – стандарт взаимодействия между *Python*-программой, выполняющейся на стороне сервера и веб-сервером или веб-фреймворком.

***ASGI*** является развитием технологии ***WSGI*** и предоставляет стандарт для асинхронных приложений.

## ***ORM***

Выполнение команд:

1. *python manage.py shell*;
2. импорт моделей: *from newsapp.models import \**

|  |  |
| --- | --- |
| **Описание** | **Пример** |
| Объекты с *n* по *m-1* | News.objects.all()[2:6] |
| Разворот *queryset* | News.objects.all().**reverse**() |
| Сортировка по полю | News.objects.**order\_by**('-pk') |
| Первая и последняя записи *queryset* | News.objects.order\_by('title').**first**()  News.objects.order\_by('-title').**last**()  **Для полей типа даты используются методы** *earliest* и *latest* |
| Получение **одной** **записи** по значению поля | News.objects.get(pk=3) |
| Выборка по условию:  ***gt*** – строго больше  ***lt*** – строго меньше  ***gte*** – >=  ***lte*** – <=  ***contains*** – регистрозависимый поиск по шаблону. ***icontains*** – регистронезависимый вариант  ***in*** – вхождение в список значений | News.objects.filter(pk\_\_**gte**=3)  News.objects.filter(pk**\_\_in**=[3, 5, 7], title\_\_contains='Новость')  # перечисление через запятую работает как оператор *AND* |
| *<model>\_set:* **получение связанных данных** из первичной модели (первичной моделью является та, которая является для другой модели внешним ключом)  Алиас *<model>\_set* можно заменить переопределить с помощью *related\_name* у *ForeignKey* поля в нужной модели | c = Category.objects.get(pk=3)  c.**news\_set**.all()  # получили все новости у категории с первичным ключом = 3 |
| Обращение к полям связанных моделей:  *<model>\_\_<field>* | News.objects.filter(**category\_\_slug**\_\_icontains='vst')  Category.objects.filter(**news\_\_is\_published**=True) |
| Класс ***Q*** (выборка с логическими операторами):  **&** – **AND**  **|** – **OR**  **~** – **NOT** | from django.db.models import Q  News.objects.filter(**Q**(title\_\_icontains='vst') | **~Q**(content\_\_icontains='sample')) |
| ***exists***и***count*** | News.objects.filter(~Q(title\_\_icontains='vst')).**exists**()  News.objects.filter(~Q(title\_\_icontains='vst')).**count**() |
| Агрегирующие функции | from django.db.models import Min, Max, Sum, Avg, Count  News.objects.**aggregate**(**Max**('views\_count')) |
| Выборка по определенным полям | News.objects.**values**('title', 'category\_\_title').filter(pk\_\_gt=2) |
| Группировка | from django.db.models import Count  # выберем категории, в которых есть хотя бы одна опубликованная новость  Category.objects.filter(news\_\_is\_published=True).**annotate**(count=**Count**('news')).filter(count\_\_gt=0) |
| Класс ***F*** (обращение к значениям полей)  Класс ***F*** безопасен с точки зрения *concurrency* | from django.db.models import F  # увеличиваем счетчик просмотров новости, если ее читает пользователь  news\_obj.update(views\_count=F('views\_count') + 1) |
| Выполнение сырых *sql*-запросов | news = News.objects.raw("select \* from news") |

**Базы данных**

## **Схемы**

Есть несколько возможных объяснений, для чего стоит применять схемы:

* чтобы объединить объекты БД в логически связанные группы для облегчения управления ими.
* чтобы в одной базе сосуществовали разные приложения, и при этом не возникало конфликтов имён.

## **Виртуальное представление**

Виртуальное представление – объект БД, являющийся результатом выполнения запроса к БД. В отличие от материализованного представления виртуальное не хранит данные на диске, а забирает данные на основе запроса каждый раз в момент обращения к себе.

## **Материализованное представление**

Материализованное представление – это **копия результатов запроса**. Данные из запроса, связанного с материализованным представлением, будут сброшены на диск (закэшированы), что увеличит скорость доступа к данным такого представления.

## **Виды соединений таблиц**

1. *INNER JOIN* – **внутреннее** соединение. Объединяет из соединяемых таблиц только те строки, для которых выполняется условие *ON*.
2. *LEFT JOIN* – **левостороннее** соединение. Строки из **левой** таблицы в любом случае попадут в результирующую выборку.
3. *RIGHT JOIN* – **правостороннее** соединение. Строки из **правой** таблицы в любом случае попадут в результирующую выборку.
4. *CROSS JOIN* – **перекрестное** соединение. Каждая запись левой таблицы объединяется с каждой записью правой таблицы (декартово произведение).

## **Индексы**

Индекс БД – объект, создаваемый с целью повышения производительности поиска данных. Индекс хранит ссылки на страницы, в которых лежат записи.

Индексы могут уменьшить производительность, если применяются в таблицах, где часто происходят операции *INSERT*, *DELETE* и *UPDATE*, т.к. меняя содержимое таблицы нужно перестраивать и ее индексы.

Также нужно помнить о том, что **индексы занимают дополнительный объем памяти**, т.к. хранятся на диске.

|  |  |
| --- | --- |
| **Индекс** | **Кейсы для применения** |
| ***btree*** – **сбалансированное дерево**. Является индексом по умолчанию*.* Наиболее часто применяемый вид индекса | Любые данные, с которыми можно проводить операции сравнения |
| ***hash*-индексы** – хеширование значений индексированного поля таблицы | Работает только с простыми условиями равенства  Используется *PostgreSQL* для соединения таблиц, если у таблиц нет индексов. СУБД временно создает хеш-индексы для таблиц, соединяет их и затем удаляет хеш-индексы. |
| ***GiST (R-Tree)*** | Позволяет индексировать значения, перекрывающиеся между собой. Это могут быть значения многомерных структур хранения данных типа координат в многомерном пространстве  Индексация географических типов данных |
| ***GIN* (инвертированный индекс)** | Полнотекстовый поиск, поиск по *JSON*-полям |
| **Частичный индекс** | Применяется, когда нет цели индексировать данные целиком.  **Пример**: *CREATE INDEX idx\_table\_field ON table (field) WHERE field IS NOT NULL*  **Недостаток**: чтобы индекс применялся, необходимо в запросы «таскать» выражение, указанное при создании индекса (в примере это выражение *IS NOT NULL*). |
| **Функциональный индекс** | Применяется, когда есть необходимость проиндексировать не сами значения, а результаты функций от этих значений.  **Пример**: *CREATE INDEX idx\_table\_field ON table (LOWER(field))*  **Недостаток**: чтобы индекс применялся, необходимо в запросы «таскать» выражение, указанное при создании индекса (в примере это выражение *LOWER(field)*). |
| **Кластерный индекс** | Кластеризация сортирует данные по указанному индексу. То есть она применяется к таблице и ранее созданному индексу  Кластеризация является одноразовой операцией: последующие изменения в таблице нарушают порядок кластеризации |

## **Оптимизация запросов**

Сначала нужно понять к каким данным СУБД обращается чаще всего и какие запросы выполняются дольше всего.

**Долгое выполнение запросов может быть связано с внешними факторами**: например, в момент выполнения запроса выполнялось резервное копирование или в то же время сервер обрабатывал другие тяжелые запросы. **То есть дело может быть не в самом запросе.**

**1 шаг (статистика)**: собрать статистику выполнения запросов, например, с помощью расширения ***pg\_stat\_statements***. Сначала нужно активировать расширение в файле конфигурации ***postgresql.conf***, добавив в него строки:

shared\_preload\_libraries = ‘pg\_stat\_statements’

# **соберет статистику по 1000 наиболее используемых запросов**

pg\_stat\_statements.max = 1000

pg\_stat\_statements.track = all

**Обращение к статистике**:

SELECT query, calls, total\_time, rows,

100.0 \* shared\_blks\_hit / nullif(shared\_blks\_hit + shared\_blks\_read, 0) AS hit\_percent

FROM pg\_stat\_statements

ORDER BY total\_time DESC LIMIT 10;

**1 шаг (альтернативный способ: лог)**: подключить в ***postgresql.conf*** логирование запросов, время выполнения которых превышает заданную величину в миллисекундах:

log\_duration = on

log\_lock\_waits = on

# время выполнения (в мс.), превысив которое запрос попадает в лог

log\_min\_duration\_statement = 100

log\_filename = ‘postgresql-%Y-%m-%d\_%H%M%S’

log\_directory = ‘/var/log/postgresql’

log\_destination = ‘csvlog’

logging\_collector = on

Просматривать отчеты, сделанные на основе логов, можно с помощью утилиты ***pgbadger***, которую необходимо установить отдельно.

**2 шаг:** посмотреть план запроса с помощью операторов ***EXPLAIN ANALYZE* (**их можно использовать как вместе, так и раздельно, см. далее**)**.

***EXPLAIN*** построит и покажет только план выполнения, но не выполнит запрос.

***ANALYZE*** соберет статистику используемых таблиц, **выполнит** **запрос** **(!)** и покажет время его выполнения.

**Пример**

EXPLAIN SELECT \* FROM tenk1;

QUERY PLAN

-------------------------------------------------------------

Seq Scan on tenk1 (cost=0.00..458.00 rows=10000 width=244)

Числа, перечисленные в скобках (слева направо), имеют следующий смысл:

* приблизительная **стоимость запуска** (время, спустя которое начнётся этап вывода данных);
* приблизительная **общая стоимость (*cost*). 1 *cost* = 1 операция чтения страницы с диска**;
* ожидаемое **число строк**, которое должен вывести этот **узел плана**;
* ожидаемый **средний размер строк**, выводимых этим узлом плана (в байтах).

Если требуется проанализировать с помощью *EXPLAIN* *ANALYZE* запросы типа *DELETE*, *UPDATE* и *INSERT*, то необходимо обернуть их в транзакцию, т.к. *ANALYZE* выполняет запрос:

BEGIN;

EXPLAIN ANALYZE <query>;

ROLLBACK;

Важно помнить о том, что при выполнении запроса **планировщик выбирает план выполнения**, **опираясь на** **статистику**. Следовательно, при анализе запроса путем использования оператора *EXPLAIN* **нужно убедиться, что статистика не устарела**.

**Собрать статистику** можно, вызвав *ANALYZE* с именем таблицы, или просто *ANALYZE*, чтобы проанализировать все таблицы БД.

**Шаг 3: анализ**: при анализе нужно обращать внимание на следующие моменты:

* операции с большой ***cost***;
* узлы плана с последовательным чтением (***seq scan***), часто указывающие на отсутствие необходимых индексов;
* ***sort*** – операции сортировки являются дорогими.

**Дальнейшие возможные шаги**:

* на сервере **можно запускать** **автоочистку и автоанализ** (***autovacuum***) для удаления неиспользуемых записей (мертвых кортежей);
* **денормализация** – привнесение избыточности;
* **использование** **табличных пространств** – управление размещением объектов по физическим устройствам ввода/вывода. Например, активно используемые данные хранить на *SSD* дисках, а архивные – на более медленных *HDD*.
* **партиционирование** – разделение большой таблицы на множество более мелких;
* **замена подзапросов в запросах на соединения**;
* **деградация функциональности** – намеренное понижение количества передаваемых клиенту данных (например, вместо топ-100 друзей отдаем топ-10).

## **Масштабирование баз данных**

**Вертикальное масштабирование** предполагает наращивание мощности текущего сервера (добавление памяти, замена *CPU* и т.д.).

**Горизонтальное масштабирование** предполагает разделение данных по разным серверам или разным инстансам сервисов.

Реализовать масштабирование можно несколькими способами:

* **партиционирование** – разбиение данных на несколько частей по определенному признаку (например, разбиваем таблицу на две части по четности поля *ID*);
* **физическое или виртуальное шардирование** – хранение данных на разных серверах или инстансах БД;
* **репликация** (см. [Репликация](#_Репликация)).

## ***ACID* (свойства транзакционной СУБД)**

***Atomicity*** – **атомарность** операций. Транзакция должна быть выполнена полностью, либо вовсе не выполнена.

***Consisitency*** – **согласованность**. Обеспечение целостности данных, выполнения определенных правил и ограничений (триггеры, непустые поля, форейн-кеи и т.д.).

***Isolation*** – **изолированность**. Выполнение параллельных транзакций не должно влиять на итоговый результат другой транзакции. **Это очень дорогое требование, поэтому оно частично и по-разному выполняется в соответствии с уровнями изолированности транзакций**.

***Durability*** – **долговечность**. Изменения успешно завершенной транзакции не должны пропасть в случае сбоев.

## **Уровни изолированности транзакций**

Стандарт *SQL* определяет **4 уровня изоляции** транзакций.

Пусть *T1* – **первая транзакция**, а *T2* – **вторая** **транзакция**.

***L1. READ UNCOMMITTED*** («грязное» чтение) – *T1* **видит изменения**, выполненные **незавершенной** *T2*. Этот уровень не реализован в *PostgreSQL*, т.к. не представляет практической ценности.

***L2****\*****. READ COMMITTED*** (чтение зафиксированных данных) – *T1* увидит изменения, выполненные *T2*, после коммита (фиксации) *T2*.

Попытка изменения одних и тех же данных в *T2* до коммита изменений *T1* будет заблокирована. \***Является уровнем по умолчанию в *PostgreSQL***.

***L3. REPEATABLE READ*** (повторяемое чтение) – *T1* увидит изменения, выполненные *T2*, только после коммита *T2* и *T1*.

Попытка изменения одних и тех же данных в *T2* до коммита изменения в *T1* будет заблокирована.

***L4. SERIALIZABLE*** – уровень, на котором моделируется последовательное выполнение всех зафиксированных транзакций. Если транзакции обращаются к одной и той же таблице, зафиксируются изменения только той транзакции, которая зафиксируется быстрее всех.

Шаблон для наглядного представления и работы с уровнями изоляции (нужна заранее созданная БД с таблицей *test* и еще одним пользователем помимо *postgres*):

*T1* (сессия пользователя *postgres*):

-- READ COMMITTED

-- REPEATABLE READ

-- SERIALIZABLE

DELETE FROM test;

INSERT INTO test VALUES (1), (2);

SELECT \* FROM test

BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;

UPDATE test SET id = id + 10 WHERE id = 1;

COMMIT;

*T2* (сессия пользователя *test*):

SELECT \* FROM test

BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;

--- можно поменять id на 1, чтобы увидеть блокировки

UPDATE test SET id = id + 10 WHERE id = 2;

COMMIT;

## ***CAP*-теорема**

Это утверждение о том, что в любой реализации распределенных вычислений возможно обеспечить **не более двух** из трех следующих свойств (возможные комбинации свойств: ***CA, CP, AP***):

***Consistency*** – состояние согласованности данных во всех вычислительных узлах (одни и те же данные у мастеров и реплик и т.д.);

***Availability*** – доступность системы;

***Partition tolerance*** – система способна работать в случае отказа некоторых вычислительных узлов (**это свойство по умолчанию соблюдается микросервисной архитектурой.** Иными словами, для микросервисной архитектуры стоит выбор отказа от *Consistency* или *Availabiblity***).**

## **Репликация**

**Репликация** – механизм синхронизации нескольких наборов данных.

Существуют два вида репликации: **физическая** и **логическая**. Оба вида будут описаны ниже.

Также существуют **гарантии** **репликации**: **синхронная** и **асинхронная.**

**Синхронная** репликация подразумевает, что мастер-сервер фиксирует транзакцию только после момента подтверждения получения данных репликой.

**Асинхронная** же подразумевает обратное: **мастер не ждет подтверждения получения данных** от реплики.

**По виду взаимодействия** репликация может быть **мастер-слейв**и **мастер-мастер.**

**Мастер-слейв репликация** предполагает главного (мастер) сервера и подчиненного (слейв) сервера. **Слейв повторяет состояние мастера** и не может менять данные самостоятельно.

**Репликация** **мастер-мастер предполагает равенство серверов**. Оба сервера могут обрабатывать запросы на чтение и изменение данных.

**Применение** репликации:

* **балансировка нагрузки**;
* **обеспечение отказоустойчивости** (переход на реплику в случае сбоя основного сервера);
* **тестирование новых версий *PostgreSQL***.

**Физическая** (**потоковая**) репликация – вид репликации, в основе которого лежит **передача записей журнала транзакций** от мастера к реплике. **Реплика применяет изменения**, принятые от мастера, **чисто механически**, без «понимания смысла», работая **в режиме постоянного восстановления**. Поэтому данный вид репликации можно применить только в условиях полной программно-аппаратной совместимости мастер и слейв серверов (одна архитектура *CPU* и версия *PostgreSQL*).

**Недостатки** физической репликации:

* мастер может отдать слейву испорченные данные, например, в случае сбоев *RAM*;
* на реплике не может быть локальных изменений схемы данных;
* требовательность к платформе и версии *PostgreSQL*;
* невозможность репликации мастер-мастер.

В основе **логической** репликации лежит принцип ***publisher*-*subscriber***, при этом у серверов нет выделенных ролей: **один и тот же сервер может публиковать изменения и быть подписчиком**.

Сервер-публикатор читает журнальные записи и предварительно декодирует в «логический», платформо-независимый вид. Поэтому логическая репликация не требует жесткой совместимости серверов, реплика должна лишь понимать протокол репликации.

**Логическая репликация не реплицирует** команды *DDL* (все изменения схемы данных нужно переносить вручную), значения последовательностей.

**Плюсы** логической репликации:

* репликация остановится, если мастер имеет аппаратные неисправности;
* нетребовательность к программной совместимости;
* возможность использования различающихся схем данных на мастере и слейве;
* возможность настройки мастер-мастер репликации.

## **Денормализация**

Под денормализацией понимают **намеренное нарушение требований нормализации**.

Денормализация обычно проводится путем добавления данных, которые согласно требования нормальных форм должны выноситься в отдельную таблицу (избыточность данных).

**Применение**: в случае долгого выполнения соединений в запросе можно **внести дополнительное поле** из присоединяемой таблицы в нужную таблицу.

Также можно построить так называемые **вертикальные таблицы**, в которых строки будут использоваться для хранения наименований полей и их значений:



Денормализация приводит к избыточности и дублированию данных, поэтому выигрыш от нее должен оправдывать затраты.

***COMMON***

## ***HTTP***

*HTTP* (*HyperText Transfer Protocol*) – протокол, описывающий правила взаимодействия клиента и сервера.

**Взаимодействие** клиента и сервера осуществляется путем обмена *HTTP*-сообщениями. *HTTP*-запрос **клиента** состоит из следующих частей:

<**начальная строка**>: <**Метод**> <***URL***> *<***Протокол/версия>**

**Пример**: *PUT /create\_page* ***HTTP/1.1***

<**заголовки запроса**>

<**тело запроса**>(**требуется не для всех методов**)

**Заголовки** – пары вида ключ-значение, необходимые для передачи служебной информации (тип контента, браузер, устройство и т.д.).

**Метод** – последовательность символов, определяющую операцию для осуществления над ресурсом. Спецификация *HTTP* не ограничивает набор методов, но общепринято использовать конкретные методы:

* ***GET*** – получение ресурса (***HEAD*** –то же самое только без содержимого);
* ***POST*** – создание ресурса;
* ***PUT*** – изменение ресурса (***PATCH*** – частичное изменение);
* ***DELETE*** – удаление ресурса;
* ***OPTIONS*** – информация о соединении.

***URL*** – идентификатор/путь до ресурса.

*HTTP* не обязывает сервер реализовывать методы строго определенным образом и понимать все методы. Только метод *GET* является обязательным для реализации.

***HTTP***-**ответ** **сервера** выглядит следующим образом:

*HTTP*/<версия> <**Код состояния**> <Пояснение>

<**заголовки**>

<**тело ответа**>

**Код состояния** указывает на результат обработки запроса клиента. Коды принято классифицировать следующим образом:

* **1xx**: *Information* (**информационные**);
* **2xx**: *Success* (**успешные**);
* **3xx**: *Redirect* (**перенаправления**);
* **4xx**: *Client* *Error* (**клиентские ошибки**);
* **5xx**: *Server* *Error* (**серверные** **ошибки**).

## **Идемпотентность *HTTP* методов**

Метод *HTTP* является **идемпотентным**, если один и тот же неоднократно выполненный запрос оказывает один и тот же эффект, не изменяющий состояние сервера. Другими словами, идемпотентный метод не должен иметь никаких побочных эффектов (*side-effects*), кроме сбора статистики или подобных операций.

Методы [***GET***](https://developer.mozilla.org/ru/docs/Web/HTTP/Methods/GET)***,***[***HEAD***](https://developer.mozilla.org/ru/docs/Web/HTTP/Methods/HEAD)***,***[***PUT***](https://developer.mozilla.org/ru/docs/Web/HTTP/Methods/PUT), [***DELETE***](https://developer.mozilla.org/ru/docs/Web/HTTP/Methods/DELETE) и ***OPTIONS*** **идемпотентны** в отличие от методов [***POST***](https://developer.mozilla.org/ru/docs/Web/HTTP/Methods/POST)и ***PATCH***.

Для идемпотентности нужно рассматривать только изменение фактического внутреннего состояния сервера, а возвращаемые запросами коды статуса могут отличаться: первый вызов [*DELETE*](https://developer.mozilla.org/ru/docs/Web/HTTP/Methods/DELETE) вернёт код [**200**](https://developer.mozilla.org/ru/docs/Web/HTTP/Status/200), а любой следующий вызов вернет [404](https://developer.mozilla.org/ru/docs/Web/HTTP/Status/404) (поэтому *DELETE* идемпотентен).

## ***REST*/*RESTful***

*REST* – это архитектурный стиль построения клиент-серверных приложений.

*RESTful* приложение должно соблюдать следующие 5/6 принципов *REST*:

1. **клиент-серверная архитектура**;
2. ***stateless*** (**отсутствие хранения состояния**) – сервер не должен хранить информацию о предыдущих запросах клиента;
3. **кэшируемость** – должна быть возможность по требованию клиента сохранить данные в кэш и отдавать эти данные из кэша;
4. **единообразие** **интерфейса** – использование одного протокола для общения клиента и сервера;
5. **многоуровневая система –** система может быть сложной, но эта сложность скрыта от клиента;
6. **код по требованию** (необязательное требование *RESTful*) – при необходимости сервер может расширить функционал клиента (валидация номера телефона на форме и т.д.).

## **Как браузер обрабатывает запрос клиента**

1. браузер получает из адресной строки протокол (*http*/*https*), доменное имя хоста и ресурс;
2. браузер получает *IP* адрес хоста путем **одним из способов**:
   * из кэша ОС, если запрос на этот хост кидался ранее;
   * резолв доменного имени хоста в *IP* с помощью ***DNS***;
3. браузер устанавливает соединение с хостом **одним из способов**:

* незащищенное по протоколу *http* по 80 порту;
* защищенное по *https* по 443 порту с проверкой подлинности, хендшейками и т.д.

1. браузер получает ответ от хоста и рендерит *HTML*.

## ***SOLID***

*SOLID* это совокупность пяти базовых принципов проектирования объектно-ориентированных систем.

***S (Single Responsibility)*** – **принцип единственной ответственности**, согласно которому сущность должна выполнять только одну задачу (у сущности должна быть только одна причина для изменения).

***O (Open-Closed Principle)*** – **принцип открытости-закрытости**. Добавление и модификация функционала не должны приводить к изменениям существующего.

***L (Liskov Substitution Principle)*** – **принцип подстановки Лисков**. Классы-наследники не должны противоречить базовому классу.

**Пример**: **наследование класса *Square* от *Rectangle* является нарушением** данного принципа, т.к. эти сущности ведут себя по-разному:

# сущность типа Rectangle пройдет данный тест, но Square как наследник Rectangle – нет, т.к. у квадрата все стороны равны

def test\_shape\_area(figure: Rectangle):

figure.width = 10

figure.height = 20

assert figure.get\_area() == 200

Тест выше можно сделать корректным, **усложнив** его логику проверками типов:

if is\_instance(figure, Rectangle) …

if is\_instance(figure, Square) …

Подобные усложнения показывают, что мы делаем что-то не так. Полиморфизм и наследование придуманы для обобщения, а не для лишних проверок на типы, в зависимости от которых выполнялась бы разная логика.

***I (Interface Segregation Principle)*** – **принцип разделения интерфейсов**. Лучше создавать много мелких узкоспециализированных интерфейсов, чем большие, часть функционала которых будет не нужна некоторым классам.

***D (Dependency Inversion Principle)*** – **принцип инверсии зависимостей**. Лучше внедрять зависимости в виде абстракций (интерфейсы), а не в виде конкретных типов (классов).

**Пример**: <https://solidbook.vercel.app/dip/in-ideal-world>.

Также в контексте *SOLID* часто употребляются следующие два термина:

1. [**Зацепление**](https://ru.wikipedia.org/wiki/%D0%97%D0%B0%D1%86%D0%B5%D0%BF%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) — степень зависимости разных модулей друг от друга. Чем выше зацепление, тем хуже;
2. [**Связность**](https://ru.wikipedia.org/wiki/%D0%A1%D0%B2%D1%8F%D0%B7%D0%BD%D0%BE%D1%81%D1%82%D1%8C_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) — чем выше связность, тем строже модули следуют [*SRP*](https://solidbook.vercel.app/srp), тем выше сфокусирован модуль на конкретной задаче.

## ***CI/CD***

***CI*** (***Continuous*** ***Integration*, непрерывная интеграция**) – практика частого внесения изменений в ПО и автоматизации различных процессов (тестирование и т.д.).

***CD*** (***Continuous*** ***Delivery*, непрерывная доставка**) – процесс постоянной доставки ПО до пользователя (автосборка и авторазвертывание).

Принципы *CI/CD* позволяют построить единый пайплайн для всех разработчиков, получать ожидаемый результат от деплоя и ускорить доставку ПО в продакшн.

Типичная модель *CI/CD* цикла выглядит так:

1. написание кода;
2. сборка;
3. тестирование;
4. релиз и деплой.

Инструменты и способы реализации:

* *Docker* (*CD*);
* *GitHub* *Actions* (*CI*);
* репликация БД (*CD*).

## ***JSON Web Tokens (JWT)***

***JWT*** – это открытый стандарт для создания токенов, основанных на формате ***JSON***. Этот вид токенов **используется для аутентификации в клиент-сервисных приложениях**.

*JSON Web Token* делится на **3 составляющие** и может выглядеть следующим образом:

**eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV\_adQssw5c**

**Компоненты** *JWT*:

* **заголовок** (***header***) – содержит **закодированную** информацию об алгоритме шифрования **подписи** ({''alg'':''HS256'', ''typ'':''JWT''});
* **полезная информация** (***payload***) – любая **закодированная** информация, которую сервер может передать клиенту ({''email'':''example@mail.com''});
* **подпись (*signature*) –** строка видаsignature = MAC-SHA256(f'{header}.{payload}, SECRET\_KEY)', **зашифрованная** **секретным** **ключом** на стороне сервера аутентификации.

**Принцип работы *JWT***: любой пользователь сможет **раскодировать заголовок и полезную информацию**, т.к. они **закодированы**, а **не зашифрованы**. Изменение их содержимого потребует создания новой подписи, но злоумышленник не сможет сгенерировать корректную подпись, т.к. не знает секретного ключа.

Токены обычно генерируются парами:

* ***access\_token*** – **токен для аутентификации пользователя**, имеющий ограниченное время действия, заданное сервером аутентификации;
* ***refresh\_token*** – токен, необходимый для генерации нового *access\_token* после завершения срока действия последнего.

***Docker***

## **Разница между *CMD* и *ENTRYPOINT***

*ENTRYPOINT* определяет основную команду, а *CMD* позволяет полностью переопределить параметры.

**Примеры**:

**Команда**: *docker run sleeper sleep 10*

***Dockerfile***: *CMD sleep 5*

**Результат**: *sleep 10*

**Команда**: *docker run sleeper 10*

***Dockerfile***: *ENTRYPOINT ["sleep"]*

**Результат**: *sleep 10*

Совместное использование *CMD* и *ENTRYPOINT* позволит определить параметры по умолчанию:

**Команда**: *docker run sleeper*

***Dockerfile***:

*ENTRYPOINT ["sleep"]*

*CMD sleep 5*

**Результат**: *sleep 5*