Заголовок

Python:
типы данных
Функции высшего порядка (map, filter, reduce)
Какие функции из itertools и collections вы исполтьзуете
Какие проблемы есть в питоне
Garbage collector
slots
*args и **kwargs
Что такое дескрипторы
Интерпретируемый или компилируемый
Временная сложность алгоритма
SOLID
Паттерны проектирования
Методы ООП
Асинхронность
Метаклассы
Property
Асинхронность, потоки, процессы
GIL
Бинарное дерево
Хэш-функция
Корутина
Lock и Семафор
private, public, protected

SQL:
Индексы в базе данных
N+1
Констрейнт
Нереляционные базы данных и реляционные
Уровни изоляции транзакций
Транзакции (пример)
ACID
Команды SQL

Django:
MVT
generics
Где храните логику
Django
Middleware
REST
Токены
Сериализатор
Permissions
Тестирование
Формы и менеджеры в джанго
HTTP, Rest и Soap

Технологии:
Docker
Apache Kafka
Celery

MongoDB
Redis
Grafana
Сокет-соединения
System Design

Git:
git cherry pick
GIT FLOW
Rebase и Merge

MVT — это архитектурный паттерн, используемый в веб-разработке и широко ассоциируемый с Django. Похож на более известный паттерн "Model-View-Controller", но адаптирован под разработку веб-приложений с некоторыми особенностями Django.

Компоненты:

1. **Model (Модель)**
Представляет структуру данных приложения и бизнес-логику. В Django модели — это Python классы, которые определяют поля и поведение данных, которые вы хотите хранить. Django автоматически предоставляет API для доступа к данным (создание, изменение, удаление, запросы к базе данных) на основе определений моделей.

2. **View (Представление)**
Отвечает за обработку запросов пользователя и возвращение ответов. В Django представления — это Python функции или классы, которые принимают веб-запрос и возвращают веб-ответ. Представления взаимодействуют с моделями и передают данные в шаблоны. Они аналогичны контроллерам в паттерне MVC.

3. **Template (Шаблон)**
Отвечают за представление данных. Это текстовые файлы, которые позволяют миксовать HTML с Django Template Language (DTL) — специальным языком шаблонов Django для динамического генерирования HTML страниц. Шаблоны определяют, как данные, полученные от представлений, будут отображаться пользователю.

Работа MVT в Django:

1. **Запрос пользователя** поступает на веб-сервер и интерпретируется URL маршрутизатором Django.
2. **URL маршрутизатор** направляет запрос к соответствующему представлению на основе URL паттерна.
3. **Представление** обрабатывает запрос, взаимодействует с моделью для получения запрошенных данных или изменения данных в базе данных.
4. После обработки данных представление выбирает **шаблон** для генерации HTML страницы и передает в шаблон необходимые данные.
5. **Шаблон** рендерится в HTML, который возвращается пользователю в качестве ответа.

MVT делает веб-разработку в Django структурированной и гибкой. Разделение ответственности между моделями, представлениями и шаблонами облегчает разработку, тестирование и поддержку веб-приложений.

Обратно

---

generics

В Django REST Framework (DRF) generics представляют собой высокоуровневые классы представлений, которые обеспечивают готовую функциональность для выполнения основных операций CRUD (Create, Retrieve, Update, Delete) веб-приложений RESTful API. Generics в DRF помогают снизить повторяемость кода, упростить взаимодействие с моделями и улучшить производительность разработки. Они предоставляют много повторно используемого кода, который обычно нужен для создания RESTful API.

Вот некоторые основные классы generics в Django REST Framework:

1. GenericAPIView: Это базовый класс для всех generics представлений. Он предоставляет основные методы для обработки HTTP-запросов и управления API-взаимодействиями. Например, он содержит методы для обработки запросов GET, POST, PUT и DELETE.

2. CreateAPIView: Представление, обрабатывающее POST запросы для создания новых записей в модели.

3. ListAPIView: Представление, обрабатывающее GET запросы для получения списка записей из модели.

4. RetrieveAPIView: Представление, обрабатывающее GET запросы для извлечения отдельной записи из модели по ее первичному ключу.

5. UpdateAPIView: Представление, обрабатывающее PUT запросы для обновления отдельной записи в модели по ее первичному ключу.

6. DestroyAPIView: Представление, обрабатывающее DELETE запросы для удаления отдельной записи из модели по ее первичному ключу.

Generics также предоставляют стандартное поведение, такое как автоматическую обработку сериализации и десериализации данных, обработку аутентификации и авторизации, а также валидацию запросов.

Использование generics позволяет разработчикам экономить время и усилия при создании RESTful API, а также обеспечивает повторное использование кода и удобство в разработке приложений на базе Django REST Framework.

Обратно

Непрерывная интеграция (Continuous Integration, CI) и непрерывная поставка (Continuous Delivery, CD)

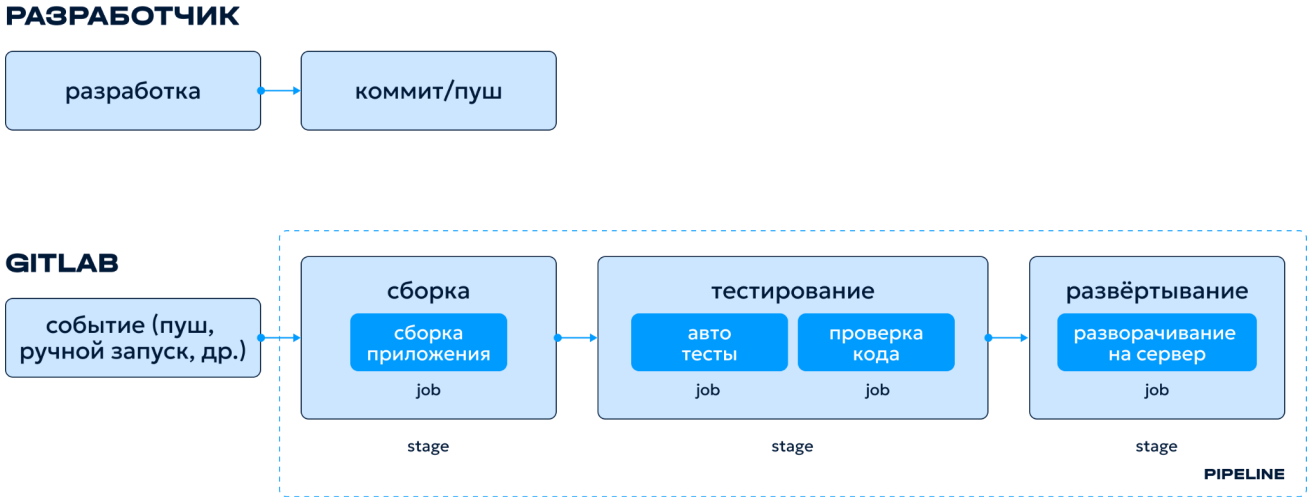
CI/CD (Continuous Integration / Continuous Deployment) — это непрерывная интеграция и развертывание, предназначенные для повышения удобства, частоты и надежности публикации изменений программного обеспечения или продукта, где:

CI — это практика разработки ПО, при которой изменения в коде автоматически собираются, тестируются и интегрируются в целевую ветку репозитория. Основная идея — минимизация разрыва между компонентами проекта и быстрая обратная связь о качестве кода, благодаря автоматической сборке и тестированию.

CD — это продолжение CI, которое позволяет автоматически разворачивать успешно собранный и протестированный код на сервере или другой среде реального применения.

Цель — автоматизация процесса разработки и развертывания приложения или программного продукта после всех этапов проверки и тестирования. Развертывание в продакшн должно выполняться после ручного подтверждения деплоя, чтобы предоставить дополнительный уровень контроля и безопасности.

Gitlab CI/CD — полностью интегрированная в GitLab система для автоматизации сборки, тестирования и развертывания программного кода. GitLab CI/CD использует файл конфигурации YAML в репозитории проекта для определения правил работы на каждом этапе в пайплайне. Поддерживает использование Docker-образов для определения окружения сборки — отсюда большая гибкость и повторное использование кода.



Алгоритм схемы следующий:

1. Разработчик пишет код и заливает его в GitLab-репозиторий проекта.
2. GitLab ищет в корне репозитория конфиг .gitlab-ci.yml и, когда находит, запускает пайплайн согласно описанной в конфиге логике.

Пайплайн (pipeline) представляет собой целиковый процесс из этапов или стадий (stage), которые состоят из задач (job). Каждая задача выполняется в изолированном процессе (используется GitLab Runner).

Что за термины мы описали выше?

- **Раннер** (gitlab runner) — приложение, в рамках которого выполняются задачи и которое можно развернуть на разных типах систем: Linux, macOS, Windows, Docker, Kubernetes и так далее.
- **Задачи** (jobs) — «кирпичики», из которых строится процесс CI/CD. Это может быть сборка проекта (компиляция, подтягивание зависимостей) или прогон автотестов, или публикация собранного кода в Docker-репозиторий. По умолчанию задачи выполняются изолированно, но их можно связать между собой при помощи **артефактов**.
- **Артефакты** (artifacts) — исполняемые файлы или пакеты для передачи результатов выполнения одной задачи на вход другой. Это позволяет управлять жизненным циклом программного продукта. Пример артефактов — скомпилированные бинарные файлы, архивы, образы контейнеров.
- **Этапы** (stages) — служат для группировки задач и определения порядка их выполнения. Задачи, принадлежащие одному этапу, выполняются параллельно, если доступно достаточное количество раннеров. Этапы будут выполняться в порядке, указанном в конфиге.
- **Пайплайн** (pipeline) — верхнеуровневый элемент процесса CI/CD, включающий в себя этапы и задачи.

Обратно

Типы данных

- Числа: `int`, `float`, и `complex`.
- Строки: `str`.
- Списки: `list`. (изм)
- Кортежи: `tuple`.
- Словари: `dict`. (изм.)
- Множества: `set`. (изм.)
- Булевы значения: `bool`

Эти типы данных можно объединить в такие группы:

- Числовые типы данных: `int`, `float`, и `complex`.
- Строковые типы данных: `str`.
- Коллекции: `list`, `tuple`, `dict`, и `set`.
- Булевы типы данных: `bool`.

Обратно

Что такое функции высшего порядка?

Функции высшего порядка - это функции, которые могут принимать другие функции в качестве аргументов или возвращать функции в качестве результата. Это является важным концептом в функциональном программировании и в целом упрощает написание кода, делая его более простым и модульным (если не перегибать).

В Python встроены несколько функций высшего порядка, таких как `map()`, `filter()` и `reduce()`.
Функция `map()` применяет заданную функцию к каждому элементу итерируемого объекта и возвращает итератор с результатами.
Функция `filter()` применяет заданную функцию к каждому элементу итерируемого объекта и возвращает итератор с элементами, для которых функция вернула True.
Функция `reduce()` объединяет элементы итерируемого объекта в одно значение, используя заданную функцию.

Пример использования `map()`:

```
def square(x):
  return x ** 2

numbers = [1, 2, 3, 4, 5]
squared_numbers = map(square, numbers)
print(list(squared_numbers))  # [1, 4, 9, 16, 25]
```
—

Пример использования `filter()`:

```
def is_even(x):
  return x % 2 == 0
```

```
numbers = [1, 2, 3, 4, 5]
even_numbers = filter(is_even, numbers)
print(list(even_numbers)) # [2, 4]
```
—

Пример использования `reduce()`:

```
from functools import reduce
def add(x, y):
  return x + y

numbers = [1, 2, 3, 4, 5]
sum_of_numbers = reduce(add, numbers)
print(sum_of_numbers) # 15
```

Обратно

Какие функции из collections и itertools вы используете?

В модулях `collections` и `itertools` в Python есть множество полезных функций, которые могут использоваться в различных задачах. Некоторые из наиболее часто используемых функций включают:

`defaultdict`: это удобный способ создания словаря с заданным значением по умолчанию для любого ключа, который еще не был добавлен в словарь.

`Counter`: это удобный способ подсчета количества встречаемых элементов в списке или другом итерируемом объекте. Он возвращает объект, который можно использовать как словарь, где ключами являются элементы, а значения - количество их вхождений.

`namedtuple`: можно создать именованный кортеж с заданными полями, что может быть удобно для работы с данными, которые имеют структуру, но не требуют создания класса.

`itertools.chain`: позволяет конкатенировать несколько итерируемых объектов в единый итератор.

`itertools.groupby`: позволяет группировать элементы итерируемого объекта по заданному ключу.

`itertools.combinations` и `itertools.permutations`: генерируют все различные комбинации или перестановки элементов из заданного множества.

—

```
from collections import defaultdict
d = defaultdict(int)
print(d['apple'])

d = defaultdict(list)
print(d['apple'])
```

```
d = defaultdict(set)
print(d['apple'])

# вывод:
# 0
# []
# set()
—
from collections import Counter
cnt = Counter(['red', 'blue', 'red', 'green', 'blue', 'blue'])
print(cnt)
print(dict(cnt))

# вывод:
# Counter({'blue': 3, 'red': 2, 'green': 1})
# {'red': 2, 'blue': 3, 'green': 1}
from collections import namedtuple

Point = namedtuple("Point", "x y")
print(issubclass(Point, tuple))

point = Point(2, 4)
print(point)

print(point.x)
print(point.y)

print(point[0])
print(point[1])

# вывод:
# True
# Point(x=2, y=4)
# 2
# 4
# 2
# 4
—
from itertools import chain

chained = chain('ab', [33])
print(next(chained))
print(next(chained))
print(next(chained))
# вывод:
# a
# b
# 33

for i in chain('1', [1, 2, 3], {3, 4}):
  print(i)
# вывод:
# 1
```

```
# 1
# 2
# 3
# 3
# 4

for i in chain('1', [1, {2, (5, '6')}, 3], {3, 4}):
  print(i)
# вывод:
# 1
# 1
# {2, (5, '6')}
# 3
# 3
# 4
—
from itertools import groupby

grouper = lambda item: item['country']

data = [
  {'city': 'Москва', 'country': 'Россия'},
  {'city': 'Новосибирск', 'country': 'Россия'},
  {'city': 'Пекин', 'country': 'Китай'},
]

for key, group in groupby(data, key=grouper):
  print(key, *group)

# Россия {'city': 'Москва', 'country': 'Россия'} {'city': 'Новосибирск', 'country': 'Россия'}
# Китай {'city': 'Пекин', 'country': 'Китай'}
from itertools import combinations

print(list(combinations('123', 2)))

# [('1', '2'), ('1', '3'), ('2', '3')]
—
from itertools import permutations

print(list(permutations('123', 2)))

# [('1', '2'), ('1', '3'), ('2', '1'), ('2', '3'), ('3', '1'), ('3', '2')]
```

Обратно

Что такое дескрипторы? Есть ли разница между дескриптором и декоратором?

Дескрипторы - это объекты Python, которые определяют, как другие объекты должны вести себя при доступе к атрибуту. Дескрипторы могут использоваться для

реализации протоколов, таких как протокол доступа к атрибутам, протокол дескрипторов и протокол методов.

Декораторы - это функции Python, которые принимают другую функцию в качестве аргумента и возвращают новую функцию. Декораторы обычно используются для изменения поведения функции без изменения ее исходного кода.

Разница между дескриптором и декоратором заключается в том, что дескрипторы используются для определения поведения атрибутов объекта, в то время как декораторы используются для изменения поведения функций. Однако, декораторы могут использоваться для реализации протоколов дескрипторов.

Например, декоратор `@property` можно использовать для создания дескриптора доступа к атрибутам. Он преобразует метод класса в дескриптор, который позволяет получать, устанавливать и удалять значение атрибута как обычный атрибут объекта.

Обратно

Какие проблемы есть в python?

Python, как и любой язык программирования, имеет свой набор потенциальных проблем и ограничений.

Вот некоторые из распространенных проблем, с которыми сталкиваются разработчики при работе с Python:

- **Глобальная блокировка интерпретатора (GIL)** — это механизм в реализации Python на CPython, который предотвращает одновременное выполнение кода Python несколькими потоками. В некоторых случаях это может ограничить производительность задач, связанных с процессором.
- **Управление пакетами и зависимостями**. Управление сторонними пакетами и зависимостями в Python иногда может быть сложным, особенно для крупных проектов или в сложных средах.
- **Производительность**. Хотя Python обычно считается быстрым языком, он не может быть оптимальным выбором для задач, требующих высокой производительности, таких как машинное обучение или научные вычисления.
- **Типизация и статический анализ**. Python — это язык с динамической типизацией, что может затруднить обнаружение определенных типов ошибок во время компиляции.
- **Управление памятью**: автоматическое управление памятью в Python может в некоторых случаях привести к утечке памяти или неэффективному использованию памяти.
- **Документация**: Хотя сообщество Python уделяет большое внимание документации, некоторые пакеты или библиотеки могут иметь неполную или устаревшую документацию, что может затруднить их эффективное использование.

Обратно

---

slots

Атрибут `__slots__` в классе Python используется для оптимизации памяти и ускорения работы с объектами класса. Он позволяет явно указать, какие атрибуты объекта будут использоваться, а какие нет.

Когда вы определяете класс, Python создает для каждого экземпляра этого класса словарь, который содержит все его атрибуты. Это может быть выгодным в том случае, если у вас много различных атрибутов, но может привести к большому расходу памяти, если вы создаете много экземпляров класса с небольшим количеством атрибутов.

Атрибут `__slots__` позволяет определить, какие атрибуты должны быть на самом деле созданы для каждого экземпляра класса, и в какой момент их можно будет получить.
Если вы используете атрибут `__slots__`, Python уже не будет создавать словарь для каждого экземпляра класса, а будет использовать непосредственно массив атрибутов, что может ускорить работу программы и уменьшить использование памяти.

Например, если у вас есть класс `Person` с атрибутами `name` и `age`, вы можете определить `__slots__` следующим образом:

```
class Person:
  __slots__ = ['name', 'age']
  def __init__(self, name, age):
     self.name = name
     self.age = age
```

Таким образом, каждый экземпляр класса `Person` будет содержать только атрибуты `name` и `age`, и никакие другие атрибуты не будут созданы.

Обратно

Garbage collector

Управление памятью осуществляется автоматически с помощью механизма сборки мусора (`Garbage collector`).

Когда объект в Python больше не нужен (например, после того как на него уже нет ссылок), он помечается как `garbage` (мусор), после чего он будет автоматически удален при следующем запуске сборщика мусора.

Используется метод подсчета ссылок для отслеживания того, когда объект уже не нужен, и этот объект должен быть освобожден. Кроме того, Python также использует циклический сборщик мусора (`Cycle detector`), который может определить и удалить объекты, на которые ссылается другой объект, на который уже нет ссылок.

Сборка мусора в Python использует алгоритм под названием `reference counting`, который подсчитывает количество ссылок на каждый объект в памяти. Когда количество ссылок на объект становится равным нулю, он помечается как мусор и память автоматически освобождается.

В Python также реализованы другие алгоритмы сборки мусора, такие как `generational garbage collection`, который разбивает объекты на несколько "поколений" и собирает мусор с различной частотой в зависимости от поколения, в котором они находятся, но `reference counting` является основой управления памятью в Python.

Модуль `gc` в Python также предлагает дополнительный функционал для управления памятью. Например, метод `gc.collect()` позволяет сделать принудительную сборку мусора.

ТАКЖЕ:

Python использует автоматическое управление памятью, что означает, что разработчику не нужно явно выделять или освобождать память в своем коде. Вместо этого в Python есть встроенный сборщик мусора, который автоматически управляет памятью для объектов, на которые больше нет ссылок.

Сборщик мусора запускается периодически и ищет объекты, на которые больше не ссылается ни одна переменная в коде. Затем эти объекты идентифицируются как мусор и удаляются из памяти. Сборщик мусора работает, отслеживая ссылки на объекты в памяти, используя механизм подсчета ссылок. Каждый раз, когда создается новая ссылка на объект, счетчик ссылок для этого объекта увеличивается. Точно так же, когда ссылка удаляется, счетчик ссылок уменьшается.

Однако одного подсчета ссылок недостаточно для обработки всех случаев управления памятью. В некоторых случаях могут быть циклические ссылки, когда два или более объекта ссылаются друг на друга и больше не нужны. Для обработки этих случаев сборщик мусора Python использует вторичный механизм, называемый «обнаружение циклов». Этот механизм периодически ищет циклические ссылки среди объектов, и если они найдены, он знает, что нужно удалить циклическую ссылку и освободить память.

В целом, сочетание подсчета ссылок и обнаружения циклов позволяет Python автоматически управлять памятью и обеспечивать очистку объектов, когда они больше не нужны. Это приводит к более эффективному использованию памяти и снижает риск нехватки памяти в приложениях, которые долго работают или интенсивно используют память.

Обратно

---

*args и **kwargs

`*args` и `**kwargs` - это специальные параметры в Python, которые позволяют передавать переменное количество аргументов в функцию.

Параметр `*args` используется для передачи переменного количества аргументов без ключевого слова. Он представляет собой кортеж из всех дополнительных аргументов, переданных функции.

Параметр `**kwargs` используется для передачи переменного количества именованных аргументов. Он представляет собой словарь из всех дополнительных именованных аргументов, переданных функции.

Символ `*` и `**` могут использоваться в определении функций для указания переменного числа аргументов, которые могут быть переданы в функцию. Символ `*` перед именем параметра означает, что все позиционные аргументы, которые не были использованы при определении других параметров, будут собраны в кортеж, который можно будет использовать внутри функции. Такой параметр называется `*args`.

Например:

```
def my_fun(a, b, *args):
        print(a, b, args)
```

```
my_fun(1, 2, 3, 4, 5)
# 1 2 (3, 4, 5)
```

Символ `**` перед именем параметра означает, что все именованные аргументы, которые не были использованы при определении других параметров, будут собраны в словарь, который можно будет использовать внутри функции. Такой параметр называется `**kwargs`.

Например:

```
def my_fun(a, b, **kwargs):
  print(a, b, kwargs)

my_fun(1, 2, x=3, y=4, z=5)
# 1 2 {'x': 3, 'y': 4, 'z': 5}
```

Краткий итог: использование `*args` и `**kwargs` позволяет создавать более гибкие функции, которые могут принимать любое количество аргументов. А именно: `*args` собирает все "лишние" аргументы в кортеж, а `**kwargs` собирает в словарь именованные аргументы.

Обратно

N+1

Проблема N+1 запросов

Проблема N + 1 возникает, когда фреймворк доступа к данным выполняет N дополнительных SQL-запросов для получения тех же данных, которые можно получить при выполнении одного SQL-запроса.

Какую проблему решают select_related/prefetch_related

В Django `select_related` и `prefetch_related` предназначены для остановки потока запросов к базе данных, вызванных доступом к связанным объектам.

По умолчанию Django не загружает связанные объекты вместе с основным запросом, а использует ленивую загрузку и откладывает запрос в базу до обращения к связанным объектам.
Такой подход упрощает работу со связанными объектами, но так же может привести к проблеме N + 1, когда для каждой связанной сущности генерируется дополнительный запрос в базу.

Мы используем **`select_related`**, когда объект, который вы собираетесь выбрать, является одним объектом, что означает пересылку `ForeignKey`, `OneToOne` и обратный OneToOne.

`select_related` работает путем создания соединения SQL и включения полей связанного объекта в оператор `SELECT`. По этой причине `select_related` получает связанные объекты в том же запросе к базе данных.

Мы используем **`prefetch_related`**, когда собираемся получить набор вещей.

Это означает обработку `ManyToMany` и обратных `ManyToMany`, `ForeignKey`. `prefetch_related` выполняет отдельный поиск для каждой связи и выполняет «объединение» в Python.

Он отличается от `select_related`. `prefetch_related` выполнял `JOIN` с использованием Python, а не в базе данных.

Минус select_related

Несмотря на то что select_related используют для оптимизаци, использование select_related также может привести замедлению выполнения запроса. Для загрузки данных select_related использует JOIN, в случае если в основной таблице записей много и они ссылаются на одни и те же данные в связанной таблице, в результирующей таблице данные будут повторяться.

Альтернативный подход - prefetch_related

В отличие от select_related, prefetch_related загружает связанные объекты отдельным запросом для каждого поля переданного в качестве параметра и производит связывание объектов внутри python.

Такой подход позволяет загружать загружать объекты для ManyToMany полей и записи которые ссылаются на нашу таблицу через ForeignKey поле используя related_name.

Однако prefetch_related можно также использовать там, где мы используем select_related, чтобы загрузить связанные записи используя дополнительный запрос, вместо JOIN.

Обратно

В Django вы можете использовать **транзакции** для группировки нескольких операций базы данных в одну логическую транзакцию, которая либо будет выполнена целиком, либо откатится полностью в случае возникновения ошибки. Вот пример того, как реализовать транзакцию в Django:

```
from django.db import transaction

@transaction.atomic
def my_view(request):
    # Начало транзакции
    with transaction.atomic():
        # Выполнение операций базы данных
        # Например, сохранение объектов в базе данных
```

```
            obj1 = MyModel(field1='value1')
            obj1.save()

            obj2 = MyModel(field1='value2')
            obj2.save()

        # Если все операции прошли успешно, транзакция будет зафиксирована
автоматически
        # Если возникла ошибка, транзакция будет отменена и все изменения будут отменены

        return HttpResponse('Транзакция завершена')
```

В этом примере мы используем декоратор @transaction.atomic для обозначения начала транзакции. Затем мы используем with transaction.atomic() для обозначения блока кода, который должен быть выполнен в рамках транзакции. В случае успешного выполнения всех операций, транзакция будет зафиксирована автоматически. В случае возникновения ошибки, транзакция будет отменена.

Помните, что использование транзакций важно для обеспечения целостности данных и избегания ситуаций, когда часть операций выполняется успешно, а другая нет.

Обратно

Паттерны проектирования

Существует три основные разновидности паттернов:

- архитектурные паттерны;
- паттерны проектирования;
- идиомы.

К первой категории относятся паттерны, которые представляют собой шаблоны высшего уровня. Они описывают структурную схему программной системы в целом. В этой схеме располагаются отдельные функциональные составляющие системы (подсистемы) и определяются отношения между ними.

В качестве примера архитектурного паттерна можно взять популярную программную парадигму «модель-представление-контроллер» (model-view-controller — MVC). При этом подсистемы содержат в себе архитектурные единицы более низкого уровня.

Вторая категория — паттерны проектирования. Они описывают схемы детализации программных подсистем и их отношений между собой. Такие паттерны никак не влияют на структуру программной системы в целом и не зависят от использования языка программирования.

Ярким примером являются паттерны GoF. Паттерны проектирования объектно-ориентированных систем — это описание взаимодействия объектов и

классов, которые адаптированы для решения основной задачи проектирования в определенном контексте.

Идиомы представляют собой паттерны низкого уровня. Их предметная область — реализация той или иной проблемы с учётом специфики соответствующего языка программирования. Вид идиомы может быть различен в зависимости от того, к какому языку она применяется. В некоторых случаях идиома и вовсе не будет иметь никакого смысла.

Основные виды паттернов проектирования

Можно выделить три типа шаблонов: порождающие, структурные и поведенческие. Перечислим основные паттерны проектирования для каждого из них.

Порождающие (Creational)

Данные паттерны выделяют процесс инстанцирования. С помощью них система перестает зависеть от метода формирования композиции и представления объектов.

- **Прототип (Prototype)**. Такой шаблон задаёт типы создаваемых объектов посредством экземпляра-прототипа. Более того, копируя прототип, паттерн формирует новые объекты. При использовании этого шаблона вы сможете отойти от реализации и придерживаться принципа «программирование через интерфейсы». В роли возвращающего типа выступает указанный интерфейс/абстрактный класс на вершине иерархии. При этом классы-наследники могут подставить в это место наследника, выполняющего реализацию данного типа. Иными словами, прототип позволяет создать объект посредством клонирования, а не с помощью конструктора.
- **Factory (Фабрика)**. Это не считается паттерном в строгом смысле слова. Правильнее будет обозначить Factory как подход, в рамках которого логика создания объектов выносится в отдельный класс.
- **Фабричный метод (Factory Method)**. Шаблон, который определят общий интерфейс для формирования объектов в суперклассе. Данный метод даёт подклассам возможность изменять вид создаваемых объектов.
- **Абстрактная фабрика (Abstract factory)**. Это порождающий паттерн проектирования, который предоставляет интерфейс для создания семейств взаимосвязанных или взаимозависимых объектов, без непосредственной спецификации их конкретных классов. Реализация такого шаблона обеспечивается за счёт создания абстрактного класса Factory. Этот класс выступает в качестве интерфейса для создания компонентов системы (скажем, применительно к оконному интерфейсу он может формировать окна и кнопки). После всего этого пишутся классы, которые реализуют данный интерфейс.

- **Одиночка (Singleton)**. Очередной порождающий паттерн. С помощью него обеспечивается наличие единственного экземпляра класса с глобальной точкой доступа в однопоточном приложении.
- **Строитель (Builder)**. Полезный порождающий паттерн, который, по сути, является методом создания составного объекта. Он дифференцирует сложный объект на конструирование и представление. Благодаря этому при выполнении одной и той же операции конструирования вы можете получить разные представления.

Структурные (Structural)

Такие шаблоны определяют всевозможные структуры высокого уровня сложности, которые вносят изменения в интерфейс уже созданных объектов или его реализацию. Это позволяет упростить процесс разработки и сделать программу более оптимизированной.

Шаблоны, входящие в эту разновидность, облегчают проектирование за счёт того, что они выявляют простейший метод реализации отношений между субъектами.

- **Мост (Bridge)**. Применяется в проектировании ПО. Он разделяет абстракцию и реализацию таким образом, чтобы они могли меняться независимо. Данный паттерн работает с помощью инкапсуляции, агрегирования и может применять наследование в целях распределения межклассовой ответственности.
- **Декоратор (Decorator)**. Данный шаблон был сформирован для динамического подключения дополнительного поведения к объекту. С его помощью практика создания подклассов получает гибкую альтернативу. Это позволяет сделать функционал более широким.
- **Адаптер (Adapter)**. Он необходим для организации применения функций объекта, который нельзя модифицировать, посредством специального интерфейса.
- **Компоновщик (Composite pattern)**. Такой шаблон способен объединять объекты в древовидную структуру. Это полезно для представления иерархии от частного к целому. Тем самым Composite pattern даёт клиентам возможность обращаться к отдельным объектам и к группам объектов одинаковым образом.
- **Заместитель (Proxy)**. Он предоставляет объект, контролирующий доступ к другому объекту, перехватывая при этом все вызовы. Иными словами, он выступает в качестве контейнера Фасад (Facade). Он помогает скрыть сложность системы. Принцип работы довольно прост: все возможные внешние вызовы сводятся к одному и тому же объекту, который передаёт эти вызовы соответствующим объектам системы.
- **Приспособленец (Flyweight, «легковесный (элемент)»)**. В процессе применения данного паттерна объект представляет себя как уникальный экземпляр в разных частях программы, однако, на самом деле это не так.

Поведенческие (behavioral)

Такие паттерны определяют алгоритмы и методы реализации взаимодействия каких-либо объектов и классов. Они выявляют общие закономерности связей между объектами, которые реализуют данные шаблоны. Использование поведенческих паттернов позволяет снизить уровень связности системы и упростить взаимодействие между объектами. Это позволяет сделать программный продукт более гибким.

- **Итератор (iterator)**. Это интерфейс, который даёт доступ к элементам коллекции (массива или контейнера), а также навигацию по ним. В зависимости от конкретной системы итераторы носят разные названия. Если говорить о системах управления, то это курсоры.
- **Интерпретатор (Interpreter)**. Он решает одну очень известную, но постоянно меняющуюся задачу. Альтернативное название — Little (Small) Language.
- **Цепочка обязанностей (Chain of responsibility)**. Такой шаблон нужен для организации уровней ответственности в системе.
- **Хранитель (Memento)**. С его помощью можно выполнить закрепление и сохранение внутреннего состояния объекта без нарушения инкапсуляции. Это нужно для того, чтобы в последующем можно было восстановить его в это самое состояние.
- **Команда (Command)**. Он представляет действие и применяется в рамках объектно-ориентированного программирования. Объект команды содержит в себе как само действие, так и его параметры.
- **Посредник (Mediator)**. Данный паттерн проектирования позволяет обеспечить взаимодействие нескольких объектов. В процессе этого создаётся слабая связанность и объекты избавляются от потребности в явных ссылках друг на друга.
- **Состояние (State)**. Применяется в ситуациях, когда в процессе выполнения программы объект должен изменять свое поведение, в зависимости от того, в каком состоянии он находится.
- **Наблюдатель (Observer)**. Его также называют «подчинённые» (Dependents). Формирует особый механизм у класса, который даёт возможность экземпляру объекта этого класса получать уведомления от остальных объектов. В оповещения содержится информация об изменении их состояния. Таким образом, происходит наблюдение за объектом.
- **Посетитель (visitor)**. Отвечает за описание операции, которая выполняется над объектами остальных классов. Если внести изменения в visitor, то вам не придётся корректировать обслуживаемые классы.
- **Стратегия (Strategy)**. Этот шаблон необходим для того, чтобы определять семейства алгоритмов и инкапсуляции каждого из них. Кроме того, данный паттерн позволяет обеспечить их взаимозаменяемость. Благодаря этому появляется возможность подбора алгоритма посредством определения соответствующего класса. С помощью Strategy

можно изменять применяемый алгоритм вне зависимости от использующих его объектов-клиентов.
- **Шаблонный метод (Template method)**. Определяет «фундамент» алгоритма и даёт наследникам возможность переопределять те или иные шаги алгоритма, оставляя его общую структуру нетронутой.

Обратно

Констрейнт (constraint) в базах данных - это правило или условие, которое задается для столбца или группы столбцов в таблице, чтобы обеспечить целостность данных и сохранить их консистентность. Констрейнты определяют различные ограничения на данные, которые могут быть добавлены или изменены в таблице.

Некоторые типы констрейнтов включают в себя:

1. **PRIMARY KEY** - уникальный идентификатор для каждой записи в таблице.

2. **FOREIGN KEY** - связь между столбцами в разных таблицах, обеспечивающая целостность ссылочной целостности.

3. **UNIQUE** - гарантирует уникальность значений в столбце или группе столбцов.

4. **NOT NULL** - требует, чтобы значения в столбце не были NULL (пустыми).

5. **CHECK** - позволяет определить условие, которое значения в столбце должны удовлетворять.

Использование констрейнтов помогает предотвратить ошибки ввода данных, обеспечивает целостность данных и облегчает поддержку базы данных.

Обратно

SOLID - это принципы разработки программного обеспечения, следуя которым Вы получите хороший код, который в дальнейшем будет хорошо масштабироваться и поддерживаться в рабочем состоянии.

S - Single Responsibility Principle - принцип единственной ответственности. Каждый класс должен иметь только одну зону ответственности.

O - Open closed Principle - принцип открытости-закрытости.
Классы должны быть открыты для расширения, но закрыты для изменения.

L - Liskov substitution Principle - принцип подстановки Барбары Лисков.
Должна быть возможность вместо базового (родительского) типа (класса) подставить любой его подтип (класс-наследник), при этом работа программы не должна измениться.

I - Interface Segregation Principle - принцип разделения интерфейсов.
Данный принцип обозначает, что не нужно заставлять клиента (класс) реализовывать интерфейс, который не имеет к нему отношения.

D - Dependency Inversion Principle - принцип инверсии зависимостей.
Модули верхнего уровня не должны зависеть от модулей нижнего уровня. И те, и другие

должны зависеть от абстракции. Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.

Обратно

Индексы

B-tree

B-tree (сбалансированное дерево) — это самый распространенный тип индекса в PostgreSQL. Он поддерживает все стандартные операции сравнения (>, <, >=, <=, =, <>) и может использоваться с большинством типов данных. B-tree индексы могут быть использованы для сортировки, ограничений уникальности и поиска по диапазону значений.

Пример создания **B-tree** индекса:

```
CREATE INDEX ix_example_btree ON example_table (column_name);
```

Hash

Hash-индексы предназначены для обеспечения быстрого доступа к данным по равенству. Они менее эффективны, чем B-tree индексы, и не поддерживают сортировку или поиск по диапазону значений. Из-за своих ограничений, Hash-индексы редко используются на практике.

Пример создания **Hash**-индекса:

```
CREATE INDEX ix_example_hash ON example_table USING hash (column_name);
```

GiST (Generalized Search Tree)

GiST-индексы являются обобщенными и многоцелевыми, предназначены для работы с сложными типами данных, такими как геометрические объекты, текст и массивы. Они позволяют быстро выполнять поиск по пространственным, текстовым и иерархическим данным.

Пример создания **GiST-индекса** для поиска в текстовых данных:

```
CREATE INDEX ix_example_gist ON example_table USING gist (to_tsvector('english', column_name));
```

SP-GiST (Space-Partitioned Generalized Search Tree)

SP-GiST индексы предназначены для работы с непересекающимися и неравномерно распределенными данными. Они эффективны для поиска в геометрических и IP-адресных данных.

Пример создания **SP-GiST** индекса:

```
CREATE INDEX ix_example_spgist ON example_table USING spgist (inet(column_name));
```

GIN (Generalized Inverted Index)

GIN-индексы применяются для полнотекстового поиска и поиска по массивам, JSON и триграммам. Они обеспечивают высокую производительность при поиске в больших объемах данных.

Пример создания **GIN-индекса** для полнотекстового поиска:

CREATE INDEX ix_example_gin ON example_table USING gin (to_tsvector('english', column_name));

BRIN (Block Range INdex)

BRIN-индексы используются для компактного представления больших объемов данных, особенно когда значения в таблице имеют определенный порядок. Они эффективны для хранения и обработки временных рядов и географических данных.

Пример создания **BRIN-индекса** для временного ряда:

CREATE INDEX ix_example_brin ON example_table USING brin (column_name);

Без индексов, базе данных приходится выполнять полный сканирование таблицы (sequential scan), чтобы найти нужные данные. Это может быть медленным и ресурсоемким процессом, особенно для больших таблиц. Индексы позволяют существенно ускорить поиск, так как они предоставляют структуру данных, которая указывает на местоположение нужной информации в таблице.

Заключение

Проанализировав особенности и применение разных типов индексов для разнообразных типов данных, мы можем сделать следующие основные выводы:

1. Выбор правильного индекса: Важно выбрать подходящий тип индекса в зависимости от типа данных и предполагаемых запросов к базе данных. Например, для текстовых данных B-tree индексы подходят для операций сравнения, в то время как GIN и GiST индексы предпочтительны для полнотекстового поиска.

2. Оптимизация запросов: Правильно подобранный и созданный индекс может существенно улучшить производительность запросов, особенно когда объем данных в таблицах растет.

3. Компромисс между производительностью чтения и записи: Хотя индексы улучшают производительность чтения данных, следует учитывать, что они могут замедлить операции записи в таблицу, такие как INSERT, UPDATE, и DELETE. Важно найти баланс между количеством и видами индексов для оптимальной общей производительности базы данных.

4. Эффективное использование индексов для различных типов данных: Разные типы данных имеют различные характеристики и требуют разных видов индексов. Важно знать, какие индексы подходят для каждого типа данных (например, GIN для массивов, JSONB и GiST для географических данных).

В заключение, индексирование и правильный выбор типов данных играют ключевую роль в обеспечении высокой производительности и эффективности работы с базой данных PostgreSQL. Разработчикам и администраторам баз данных следует тщательно проектировать индексы, оптимизировать запросы и регулярно поддерживать индексы в актуальном состоянии для обеспечения наилучшего функционирования системы.

Обратно

Методы ООП

- Наследование
- Инкапсуляция
- Полиморфизм
- Абстракция

  Наследование — свойство системы, позволяющее на основе существующего класса создать новый класс, и при этом частично или полностью заимствовать в наследнике функциональность или атрибуты родителя.

  Например, можно описать базовый класс `HockeyPlayer`, а затем создать дочерний класс `GoalKeeper`, который наследует атрибуты и методы класса `HockeyPlayer`, но в который будут добавлены специфические для вратаря свойства и методы — например, метод `not_my_fault()`.
- **Полиморфизм** — возможность единообразно обрабатывать объекты с различной реализацией при условии наличия общего интерфейса.

  Любому хоккеисту важно поддерживать хорошую физическую форму, независимо от того, какую роль он играет на льду: вратарь он, защитник или нападающий. Все они — объекты класса `Хоккеист`, и к ним можно применить один и тот же метод — `тренироваться()`. Для разных игроков метод `тренироваться()` будет выглядеть по-разному: вратарь будет больше времени уделять отражению бросков и тренировать реакцию, защитник будет накачивать физическую силу и отрабатывать оборону, а нападающий посвятит тренировку скорости и точности бросков.

  Тем не менее, для любого экземпляра класса `Хоккеист` и для экземпляров классов-наследников (для классов `Вратарь`, `Защитник` или `Нападающий`) тренер может вызвать метод `тренироваться()` — и этот метод сработает.

  Такой подход в ООП и называется **полиморфизмом**: для объектов базового класса и для объектов унаследованных классов можно вызывать одни и те же методы; методы сработают, но могут вести себя по-разному, в зависимости от объекта.
- **Инкапсуляция** — свойство системы, позволяющее объединить в объекте и данные, и методы для работы с ними. Данные объекта скрыты от остальной программы.

  Хоккейный болельщик не должен знать все детали тренировочного процесса: ему достаточно быть в курсе основных правил игры и наблюдать за матчем. Все детали подготовки, особенности диеты спортсменов, ход тренировок и разработка игровой тактики скрыты от наблюдателя и проявляются лишь по внешним результатам работы — по умению игроков контролировать шайбу, хорошо передвигаться по льду и забивать голы. Это и есть пример инкапсуляции в контексте хоккея.

- Абстракция используется, чтобы скрыть внутренние характеристики функции от пользователей.

Обратно

Middleware – обрабатывает запросы и ответы веб-приложения между тем, как они поступают от клиента и до того, как они достигают конечного обработчика (например, контроллера) или отправляются обратно клиенту. В различных фреймворках и технологиях middleware может работать по-разному, но обычно оно предоставляет следующие возможности:

1. **Промежуточная обработка запросов:** Может выполнить некоторую обработку запроса, например, проверить наличие заголовка аутентификации или преобразовать данные запроса перед тем, как они будут переданы обработчику запроса.

2. **Модификация ответов:** Может модифицировать ответы, возвращаемые веб-приложением, например, добавлять заголовки ответа или изменять содержимое ответа.

3. **Логирование и отладка:** Может использоваться для логирования запросов и ответов, что позволяет отслеживать процесс обработки запросов и выявлять проблемы в приложении.

4. **Кэширование:** Может кэшировать результаты запросов, чтобы ускорить доступ к данным и снизить нагрузку на сервер.

5. **Аутентификация и авторизация:** Может обеспечивать аутентификацию пользователей и проверку их прав доступа к ресурсам приложения.

6. **Обработка исключений и ошибок:** Может перехватывать и обрабатывать исключения и ошибки, возникающие во время обработки запросов, предоставляя пользователю более информативные сообщения об ошибках или выполняя дополнительные действия для восстановления работы приложения.

7. **Модификация потока управления:** Может изменять порядок обработки запросов и ответов, например, перенаправляя запросы на другие обработчики или останавливая обработку запроса на определенном этапе.

Middleware часто используется в веб-фреймворках для реализации различных аспектов функциональности приложения, таких как безопасность, производительность, отслеживание и аналитика. Оно обеспечивает гибкость и модульность веб-приложений, позволяя добавлять и изменять функциональность приложения без изменения его основного кода.

Обратно

REST определяет 6 архитектурных ограничений, соблюдение которых позволит создать настоящий RESTful API:

1. Единообразие интерфейса
2. Клиент-сервер
3. Отсутствие состояния
4. Кэширование
5. Слои
6. Код по требованию (необязательное ограничение)

Единообразие интерфейса Вы должны придумать API интерфейс для ресурсов системы, доступный для пользователей системы и следовать ему во что бы то ни стало. Ресурс в системе

должен иметь только один логичный URI, который должен обеспечивать способ получения связанных или дополнительных данных. Всегда лучше ассоциировать (синонимизировать) ресурс с веб-страницей.

Любой ресурс не должен быть слишком большим и содержать все и вся в своем представлении. Когда это уместно, ресурс должен содержать ссылки (HATEOAS: Hypermedia as the Engine of Application State), указывающие на относительные URI для получения связанной информации.

Кроме того, представления ресурсов в системе должны следовать определенным рекомендациям, таким как соглашения об именах, форматы ссылок или формат данных (xml или / и json).

Как только разработчик ознакомится с одним из ваших API, он сможет следовать аналогичному подходу для других API.

Клиент-сервер

По сути, это означает, что клиентское приложение и серверное приложение ДОЛЖНЫ иметь возможность развиваться по отдельности без какой-либо зависимости друг от друга. Клиент должен знать только URI ресурса и больше ничего. Сегодня это нормальная практика в веб-разработке, поэтому с вашей стороны ничего особенного не требуется. Будь проще.

Серверы и клиенты также могут заменяться и разрабатываться независимо, если интерфейс между ними не изменяется.

Отсутствие состояния

Рой Филдинг черпал вдохновение из HTTP, и это отражается в этом ограничении. Сделайте все клиент-серверное взаимодействие без состояний. Сервер не будет хранить информацию о последних HTTP-запросах клиента. Он будет рассматривать каждый запрос как новый. Нет сессии, нет истории.

Если клиентское приложение должно быть приложением с отслеживанием состояния для конечного пользователя, когда пользователь входит в систему один раз и после этого выполняет другие авторизованные операции, то каждый запрос от клиента должен содержать всю информацию, необходимую для обслуживания запроса, включая сведения об аутентификации и авторизации.

Клиентский контекст не должен храниться на сервере между запросами. Клиент отвечает за управление состоянием приложения.

Кеширование

В современном мире кеширование данных и ответов имеет первостепенное значение везде, где это применимо/возможно. Кэширование повышает производительность на стороне клиента и расширяет возможности масштабирования для сервера, поскольку нагрузка уменьшается.

В REST кэширование должно применяться к ресурсам, когда это применимо, и тогда эти ресурсы ДОЛЖНЫ быть объявлены кешируемыми. Кеширование может быть реализовано на стороне сервера или клиента.

Хорошо настроенное кеширование частично или полностью исключает некоторые взаимодействия клиент-сервер, что еще больше повышает масштабируемость и производительность.

Слои

REST позволяет вам использовать многоуровневую архитектуру системы, в которой вы

развертываете API-интерфейсы на сервере A, храните данные на сервере B, а запросы аутентифицируете, например, на сервере C. Клиент обычно не может сказать, подключен ли он напрямую к конечному серверу или к посреднику.

Код по требованию (необязательное ограничение)
Это опциональное ограничение. Большую часть времени вы будете отправлять статические представления ресурсов в форме XML или JSON. Но когда вам нужно, вы можете вернуть исполняемый код для поддержки части вашего приложения, например, клиенты могут вызывать ваш API для получения кода визуализации виджета интерфейса пользователя. Это разрешено

Все вышеперечисленные ограничения помогают вам создать действительно RESTful API, и вы должны следовать им. Тем не менее, иногда вы можете столкнуться с нарушением одного или двух ограничений. Не беспокойтесь, вы все еще создаете API RESTful, но не «труЪ RESTful».

Обратно

Требования **ACID** — набор требований, которые обеспечивают сохранность ваших данных.

1. Atomicity — Атомарность
2. Consistency — Согласованность
3. Isolation — Изолированность
4. Durability — Надёжность

Атомарность

Гарантирует, что каждая транзакция будет выполнена полностью или не будет выполнена совсем. Не допускаются промежуточные состояния.

Согласованность

Вытекает из предыдущего. Благодаря тому, что транзакция не допускает промежуточных результатов, база остается согласованной. То есть все ячейки таблицы обновлены нужным нам образом.

Изолированность

Во время выполнения транзакции параллельные транзакции не должны оказывать влияния на её результат. Это достигается блокированием данных с которыми работает транзакция или версионированием (это когда внутри базы при каждом обновлении создается новая версия данных и сохраняется старая).

Надёжность

Если пользователь получил подтверждение от системы, что транзакция выполнена, он может быть уверен, что сделанные им изменения не будут отменены из-за какого-либо сбоя. Обесточилась система, произошел сбой в оборудовании? На выполненную транзакцию это не повлияет.

Обратно

Асинхронность — это возможность программы выполнять задачи без ожидания их завершения.

Асинхронное программирование усложняет программы, но с его помощью можно их оптимизировать и повысить эффективность. Оно позволяет всем задачам в вашем коде выполняться одновременно (этого синхронные процессы обеспечить не могут).

Асинхронное программирование может быть полезным, если:

- программе требуется слишком много времени на выполнение всех задач;
- имеются операции ввода-вывода, требующие одновременного выполнения;
- есть задержка операций ввода и вывода.

Обратно

Метаклассы - это концепция, которая позволяет контролировать создание классов. Классы являются объектами, и они создаются с помощью других классов, которые называются метаклассами. Вот некоторые ключевые моменты о них:

1. **Классы как объекты:** Классы являются объектами первого класса, что означает, что они могут быть созданы, изменены и переданы как аргументы функций.

2. **Типы и метаклассы:** Каждый объект имеет тип, который определяется его классом. Этот класс, определяющий тип объекта, называется метаклассом. По умолчанию для всех классов метаклассом является *`type`*.

3. **Использование метаклассов:** Метаклассы можно использовать для изменения поведения создания классов. Это может быть полезно для автоматического добавления методов, проверки атрибутов или изменения наследования классов.

4. **Ключевые методы метакласса:** Для создания собственного метакласса обычно определяются методы *`__new__()`* и *`__init__()`*. Метод *`__new__()`* вызывается перед созданием экземпляра класса, а метод *`__init__()`* вызывается после создания экземпляра.

Когда Django конструирует ваш класс, она делает это с помощью своего метакласса. Чтобы при конструировании ей знать какие-то параметры вашего класса, ну, например модель или поля в вашем случае, она ищет в вашем классе класс с названием Meta.

Вообще вся эта магия с метаклассами очень важна в джанге и поэтому лучше саму логику становления класса не переопределять.

Метаклассы могут быть полезны в следующих случаях:

- При необходимости динамического изменения поведения класса, например, если вы хотите добавить или удалить атрибут или метод класса во время выполнения программы.
- При создании классов из данных, которые не заранее известны. Например, вы можете создавать классы на основе определенных условий во время выполнения программы.

- Для создания фреймворков и библиотек, которые нужно настраивать под конкретные требования и при этом сохранить простоту интерфейса.
- Они также могут использоваться для создания классов с определенными свойствами, например, классов, которые автоматически регистрируются в библиотеке или классов, которые автоматически сериализуются и десериализуются для совместимости с другими системами.

Обратно

Декоратор ***`property`*** используется для создания вычисляемых атрибутов класса. Он позволяет определить методы *`getter`*, *`setter`* и *`deleter`* для атрибута класса, что обеспечивает более гибкий доступ к данным объекта. Вот как работает каждый из этих методов:

1. **Getter (метод получения):**
 - Помеченный декоратором `@property`, представляет собой метод, который возвращает значение атрибута.
 - Вызывается при обращении к атрибуту без использования скобок или при обращении через свойство.

2. **Setter (метод установки):**
 - Помеченный декоратором *`@property.setter`*, представляет собой метод, который устанавливает значение атрибута.
 - Вызывается при попытке установить новое значение атрибута.

3. **Deleter (метод удаления):**
 - Помеченный декоратором *`@property.deleter`*, представляет собой метод, который удаляет атрибут.
 - Вызывается при использовании оператора *`del`* для удаления атрибута

Обратно

Асинхронность, threading и мультипроцессинг - это три различных подхода к параллельному выполнению задач каждый из которых имеет свои особенности и применения:

1. **Асинхронность *(Asynchronous)***

- Асинхронность предполагает выполнение задач без ожидания их завершения.
- Используется для работы с вводом-выводом (I/O), таким как чтение или запись файлов, сетевые запросы и т. д.
- В асинхронном коде задачи не блокируют основной поток выполнения, что позволяет эффективно использовать ресурсы процессора.

- Примеры асинхронных моделей включают в себя асинхронные функции и ключевые слова в Python (например, *`async`*, *`await`*).

2. **Threading (Потоки)**

- Потоки позволяют выполнять несколько частей кода (потоков) параллельно в пределах одного процесса.
- Используются для выполнения многозадачных операций, которые могут быть распределены между несколькими ядрами процессора.
- Потоки могут выполняться параллельно, но могут также конкурировать за общие ресурсы, что может привести к проблемам синхронизации и безопасности.
- В некоторых языках, таких как Python, использование потоков ограничено из-за GIL (Global Interpreter Lock), что может снижать эффективность при использовании множества потоков для CPU-интенсивных задач.

3. **Мультипроцессинг (Multiprocessing)**

- Мультипроцессинг также позволяет выполнять несколько частей кода параллельно, но каждая часть выполняется в отдельном процессе.
- Каждый процесс имеет свое собственное пространство памяти, что делает мультипроцессинг более подходящим для многозадачных вычислений на многоядерных системах.
- Процессы обычно имеют больший накладные расходы по сравнению с потоками, поскольку каждый из них требует своих собственных ресурсов памяти и управления.
- Мультипроцессинг избегает проблемы GIL, что делает его более эффективным для CPU-интенсивных задач в Python и других языках

Выбор между асинхронностью, потоками и мультипроцессингом зависит от конкретных требований вашего приложения, а также от характеристик вашей системы и языка программирования.

Обратно

GIL (global interpreter lock) **Глобальная блокировка интерпретатора,** который блокирует одновременное выполнение нескольких потоков. GIL - это мьютекс, который действует как ограничитель, позволяющий только одному потоку выполнять байткод Python в один момент времени. Это означает, что в многозадачной среде Python, в один и тот же момент времени только один поток может активно выполнять Python-код.GIL запрещает потокам выполняться одновременно, когда выполняется 1 поток, GIL блокирует все остальные.

Многопоточное и многопроцессорное приложения отличаются друг от друга в том, как они используют ресурсы компьютера.

В многопроцессорных приложениях каждый процесс имеет свой собственный набор ресурсов, включая память, открытые файлы, сетевые соединения и другие системные ресурсы. Многопроцессорность в Python может быть достигнута с помощью библиотек `multiprocessing` и `concurrent.futures`.

В многопоточных приложениях несколько потоков выполняются в рамках одного процесса, используя общие ресурсы. Это означает, что все потоки имеют доступ к общим данным. Реализация многопоточности в Python выполняется за счет стандартной библиотеки `threading`.

При правильном использовании оба подхода могут ускорить выполнение программы и улучшить управляемость ею, однако многопоточное приложение может иметь проблемы с блокировками и условиями гонки при доступе к общим ресурсам. В многопроцессорных приложениях каждый процесс защищен от других процессов и обеспечивает более высокую степень изоляции.

Причины использования GIL:

- Однопоточные сценарии выполняются значительно быстрее, чем при использовании других подходов обеспечения потокобезопасности;
- Простая интеграция библиотек на C, которые зачастую тоже не потокобезопасны;
- Простота реализации.

Минусы

Ограниченная многозадачность: Одна из наиболее известных проблем GIL - это ограничение на многозадачность. Не смотря на наличие множества потоков, лишь один может активно выполняться в определенный момент времени.

Производительность многозадачных приложений: Многозадачные приложения, которые должны эффективно использовать многие ядра процессоров, могут столкнуться с проблемами производительности, так как GIL ограничивает параллельное выполнение.

Сложности с разделением данных: Поделить данные между потоками может быть сложной задачей из-за GIL. Это может привести к гонкам данных и ошибкам.

Нестабильное время выполнения: Из-за конкуренции за GIL, время выполнения кода в потоках может быть непредсказуемым и меняться от запуска к запуску.

Ограничения на ресурсы: GIL также ограничивает доступ к ресурсам компьютера, таким как процессорное время, что может быть проблематично для многозадачных приложений.

Сложности с реализацией реальной многозадачности: Из-за GIL, реализация настоящей многозадачности в Python может быть более сложной и требовательной к ресурсам.

Неэффективное использование многоядерных процессоров: GIL делает Python менее эффективным на многоядерных процессорах, так как только одно ядро может быть активным в данный момент.

Способы обхода GIL

Один из наиболее эффективных способов обойти GIL - это использование многопроцессорной обработки вместо многозадачных потоков. Поскольку каждый процесс имеет свой собственный интерпретатор Python и собственный GIL, они могут параллельно выполняться на разных ядрах процессора.

Обратно

Docker - это платформа для разработки, доставки и запуска приложений в контейнерах. Контейнеры позволяют упаковывать приложения и все их зависимости в единую среду, что делает их переносимыми и масштабируемыми. Эта платформа предоставляет инструменты для создания, развертывания и управления контейнерами, облегчая процесс разработки и упрощая конфигурацию среды выполнения приложений.

Основные концепции Docker включают:

1. **Контейнеры:** Нужны для упаковки приложений и их зависимостей в единую среду. Контейнеры изолированы друг от друга и от хост-системы, что обеспечивает надежную и консистентную среду выполнения приложений.

2. **Образы:** Шаблоны для создания контейнеров. Они включают в себя все необходимые компоненты приложения и его зависимости. Образы могут быть созданы вручную или автоматически с использованием Dockerfile, который описывает конфигурацию контейнера.

3. **Dockerfile:** Текстовый файл, который содержит инструкции для создания образа. Он определяет все этапы установки и настройки приложения в контейнере, что позволяет автоматизировать процесс создания образов.

4. **Реестр Docker:** Сервис, который хранит образы. Он позволяет разработчикам делиться образами и использовать их для создания контейнеров на различных хост-системах.

5. **Docker Engine:** Основной компонент, который управляет созданием, запуском и управлением контейнерами. Он включает в себя клиентские и серверные компоненты, которые общаются между собой с помощью API.

6. **Docker Compose:** Инструмент для определения и запуска многоконтейнерных приложений. Он позволяет определять структуру приложения и его зависимости в файле docker-compose.yml, что упрощает развертывание и управление многочастными приложениями.

Docker облегчает процесс разработки, тестирования и развертывания приложений, обеспечивая консистентную и изолированную среду выполнения для приложений. Он также позволяет масштабировать приложения и обеспечивать их высокую доступность с помощью контейнерной оркестрации, такой как Kubernetes.

Обратно

Rebase и merge - это два основных способа интеграции изменений из одной ветки в другую в системе контроля версий Git. Оба способа имеют свои преимущества и недостатки, и

выбор между ними зависит от конкретной ситуации и предпочтений команды разработчиков. Вот основные отличия:

1. **Merge (слияние):**
 - Создает новый коммит, который объединяет изменения из двух веток (обычно текущей ветки и ветки, которую вы сливаете).
 - Этот новый коммит имеет двух родителей - по одному от каждой ветки, которые были слиты.
 - В результате история изменений остается четкой и информативной, так как она отображает точно, какие изменения были внесены в каждой ветке и когда они были объединены.
 - Однако это может привести к "клубку слияний", когда в истории изменений появляется много коммитов слияния, что усложняет анализ истории.

2. **Rebase (перебазирование):**
 - Берет коммиты из текущей ветки и "перебазирует" их на вершину целевой ветки (обычно на место, где текущая ветка была отделена).
 - Это означает, что история изменений выглядит как линейная последовательность коммитов без дополнительных коммитов слияния.
 - Позволяет поддерживать более чистую и простую историю изменений, что облегчает ее понимание и анализ.
 - Однако перебазирование может быть опасным при работе с общими ветками, так как это изменяет историю коммитов, что может привести к проблемам с синхронизацией изменений и потере данных.

Основное различие между rebase и merge заключается в том, каким образом изменения интегрируются в историю проекта. Merge создает дополнительные коммиты слияния, сохраняя историю каждой ветки, тогда как rebase перебазирует изменения на вершину другой ветки, создавая линейную историю изменений.

Обратно

Нереляционные базы данных (NoSQL)

Реляционные БД хранят структурированные данные в виде столбцов и строк. Скажем, это могут быть сведения о человеке, или о содержимом корзины для товаров в магазине, сгруппированные в таблицах, формат которых задан на этапе проектирования хранилища.

Нереляционные БД устроены иначе. Например, документо-ориентированные базы хранят информацию в виде иерархических структур данных. Речь может идти об объектах с произвольным набором атрибутов. То, что в реляционной БД будет разбито на несколько взаимосвязанных таблиц, в нереляционной может храниться в виде целостной сущности.

О выборе SQL-баз данных

Не существует баз данных, которые подойдут абсолютно всем. Именно поэтому многие компании используют и реляционные, и нереляционные БД для решения различных задач. Хотя NoSQL-базы стали популярными благодаря быстродействию и хорошей масштабируемости, в некоторых ситуациях предпочтительными могут оказаться структурированные SQL-хранилища. Вот две причины, которые могут послужить поводом для выбора SQL-базы:

1. Необходимость соответствия базы данных требованиям ACID (Atomicity, Consistency, Isolation, Durability — атомарность, непротиворечивость, изолированность,

долговечность). Это позволяет уменьшить вероятность неожиданного поведения системы и обеспечить целостность базы данных. Достигается подобное путём жёсткого определения того, как именно транзакции взаимодействуют с базой данных. Это отличается от подхода, используемого в NoSQL-базах, которые ставят во главу угла гибкость и скорость, а не 100% целостность данных.

2. Данные, с которыми вы работаете, структурированы, при этом структура не подвержена частым изменением. Если ваша организация не находится в стадии экспоненциального роста, вероятно, не найдётся убедительных причин использовать БД, которая позволяет достаточно вольно обращаться с типами данных и нацелена на обработку огромных объёмов информации.

О выборе NoSQL-баз данных

Если есть подозрения, что база данных может стать узким местом некоего проекта, основанного на работе с большими объёмами информации, стоит посмотреть в сторону NoSQL-баз, которые позволяют то, чего не умеют реляционные БД.

Вот возможности, которые стали причиной популярности таких NoSQL баз данных, как MongoDB, CouchDB, Cassandra, HBase:

1. Хранение больших объёмов неструктурированной информации. База данных NoSQL не накладывает ограничений на типы хранимых данных. Более того, при необходимости в процессе работы можно добавлять новые типы данных.

2. Использование облачных вычислений и хранилищ. Облачные хранилища — отличное решение, но они требуют, чтобы данные можно было легко распределить между несколькими серверами для обеспечения масштабирования. Использование, для тестирования и разработки, локального оборудования, а затем перенос системы в облако, где она и работает — это именно то, для чего созданы NoSQL базы данных.

3. Быстрая разработка. Если вы разрабатываете систему, используя agile-методы, применение реляционной БД способно замедлить работу. NoSQL базы данных не нуждаются в том же объёме подготовительных действий, которые обычно нужны для реляционных баз.

Масштабируемость

Одно из основных различий рассматриваемых технологий заключается в том, что NoSQL-базы лучше поддаются масштабированию. Например, в MongoDB имеется встроенная поддержка репликации и шардинга (горизонтального разделения данных) для обеспечения масштабируемости. Хотя масштабирование поддерживается и в SQL-базах, это требует гораздо больших затрат человеческих и аппаратных ресурсов.

Итоги

Занимаясь поиском системы управления базами данных, можно выбрать одну технологию, а позже, уточнив требования, переключиться на что-то другое. Однако, разумное планирование позволит сэкономить немало времени и средств.

Вот признаки проектов, для которых идеально подойдут SQL-базы:

- Имеются логические требования к данным, которые могут быть определены заранее.

- Очень важна целостность данных.
- Нужна основанная на устоявшихся стандартах, хорошо зарекомендовавшая себя технология, используя которую можно рассчитывать на большой опыт разработчиков и техническую поддержку.

А вот свойства проектов, для которых подойдёт что-то из сферы NoSQL:

- Требования к данным нечёткие, неопределённые, или развивающиеся с развитием проекта.
- Цель проекта может корректироваться со временем, при этом важна возможность немедленного начала разработки.
- Одни из основных требований к базе данных — скорость обработки данных и масштабируемость.

Обратно

Сериализатор - это инструмент, который преобразует объекты в определенном формате в формат, который можно легко сохранить или передать через сеть, и обратно. Обычно это используется для передачи данных между различными программами или процессами, работающими на разных устройствах или языках программирования.

Вот несколько причин, по которым сериализаторы могут быть полезными:

1. **Сохранение и загрузка данных:** Сериализация позволяет сохранять данные в файле или в базе данных в виде сериализованного объекта, что упрощает их сохранение и восстановление позже.

2. **Обмен данными между приложениями:** Приложения, написанные на разных языках программирования или работающие на разных платформах, могут использовать сериализацию для передачи данных друг другу в стандартизированном формате.

3. **Передача данных по сети:** Передача сложных структур данных между клиентом и сервером в сетевых приложениях может быть реализована с использованием сериализации. Например, веб-сервисы часто используют форматы сериализации, такие как JSON или XML, для обмена данными между клиентом и сервером.

4. **Хранение состояния приложения:** Сериализация может использоваться для сохранения состояния приложения, например, приложений игр, что позволяет пользователям сохранять свой прогресс и возобновлять игру позже.

5. **Кэширование данных:** Сериализация может использоваться для кэширования сложных данных, чтобы ускорить доступ к ним и сократить время обработки в будущем.

Обычно сериализаторы поддерживают различные форматы, такие как JSON, XML, YAML, Pickle и другие, каждый из которых имеет свои особенности и применение в зависимости от конкретной задачи.

Обратно

Корутина (Coroutine) - это специальный вид функции, который позволяет приостанавливать и возобновлять выполнение в произвольный момент времени. Корутины

обеспечивают кооперативную многозадачность, что означает, что они позволяют явно управлять потоком выполнения в своем коде.

Основные характеристики:

1. **Асинхронность:** Широко используются в асинхронном программировании, таком как асинхронные веб-серверы или сетевые клиенты. Они позволяют выполнять такие операции без блокирования основного потока выполнения.

2. **yield/yield from:** Для определения корутины используется ключевое слово **`yield`** или **`yield from`**, которое указывает места, где выполнение может быть приостановлено и возвращено обратно в вызывающий код.

3. **Возобновление выполнения:** Может быть возобновлена, чтобы продолжить выполнение с того же места, где она была приостановлена, и передать ей новые данные.

4. **Сопрограммы:** Их также называют сопрограммами (Subroutine), так как они представляют собой подпрограммы, которые могут быть вызваны и возобновлены.

Пример определения:

```
async def my_coroutine():
 print("Starting coroutine")
 await asyncio.sleep(1) # Асинхронное ожидание в течение 1 секунды
 print("Coroutine completed")

 # Вызов корутины
 coro = my_coroutine()
```

Здесь **`async def`** обозначает определение асинхронной функции (корутины). Ключевое слово **`await`** используется для ожидания выполнения асинхронной операции, такой как **`asyncio.sleep()`**. Когда **`await`** используется внутри корутины, выполнение корутины приостанавливается до завершения операции, переданной в **`await`**, а затем возобновляется с последующей строки.

Обратно

Redis (Remote Dictionary Server) — это высокопроизводительная система хранения данных в памяти, используется в качестве распределённого кэша, и для решения других задач, требующих быстрого доступа к данным. Вот основные сценарии использования:

1. **Кэширование данных**

Кэширование часто запрашиваемых данных, например, результатов запросов к базе данных или вычислений, требующих значительных ресурсов. Увеличивает скорость ответа приложения, снижая нагрузку на базы данных.

2. **Управление сессиями**

Используется для хранения данных сессий пользователей в веб-приложениях. Быстрое чтение и запись данных позволяет быстро сохранять и извлекать состояние сессии.

3. **Очереди сообщений**

Может использоваться как система очередей сообщений для асинхронной обработки задач и обмена сообщениями между различными компонентами приложения.

4. **Реализация блокировок**

Предлагает механизмы для реализации распределённых блокировок, что позволяет управлять доступом к ресурсам в распределённых системах, гарантируя, что одновременно к ресурсу имеет доступ только один процесс.

5. **Счетчики и статистика**

Могут использоваться для сбора статистики и метрик, например, для подсчёта числа посещений веб-страницы.

6. **Поддержка структур данных в реальном времени**

Поддерживает сложные структуры данных, такие как списки, множества, отсортированные множества и хэши.

7. **Публикация и подписка**

Механизм публикации и подписки (Pub/Sub) позволяет разрабатывать распределённые системы и приложения, реагирующие на события в реальном времени, такие как чаты, системы уведомлений и другие.
—

1. Что такое Redis и для каких задач он обычно используется?

- Ответ: Redis (Remote Dictionary Server) - это высокопроизводительная система управления базами данных, используемая как кэш, база данных и брокер сообщений. Он часто используется для кеширования, сессионного хранения, аналитики, ленты действий и многих других типов приложений.

2. Каковы основные особенности Redis?

- Ответ: Основные особенности Redis включают в себя поддержку различных структур данных (строки, списки, множества, хэши, сортированные множества), высокую производительность, встроенную поддержку репликации данных, и поддержку транзакций.

3. В чем разница между ключами EXPIRE и TTL в Redis?

- Ответ: Ключи EXPIRE и TTL используются в Redis для установки времени жизни ключа. Однако, основное отличие заключается в способе установки этого времени. EXPIRE устанавливает время жизни в секундах, в то время как команда TTL возвращает оставшееся время жизни ключа.

4. Как Redis обрабатывает конкурентные операции?

- Ответ: Redis является однопоточным приложением и обрабатывает команды последовательно, что делает его безопасным в многопоточной среде. Однако, Redis также поддерживает транзакции и CAS (Check and Set), что позволяет обрабатывать конкурентные операции.

5. Каковы методы обеспечения отказоустойчивости в Redis?

- Ответ: Redis обеспечивает отказоустойчивость с помощью механизмов репликации, ведения журнала и фильтрации данных. Он также поддерживает различные методы резервного копирования и сохранения данных на диск.

Обратно

Команда **git cherry-pick** используется для перенесения отдельных коммитов из одного места репозитория в другое, обычно между ветками разработки и обслуживания. Этот механизм отличается от привычных команд git merge и git rebase, которые переносят коммиты целыми цепочками.

Обратно

Git Flow — модель ветвления, предназначенная для работы над проектами и управления их версиями. Предлагает стандартизированный подход к ветвлению и слиянию.

Основные типы веток включают:

Основные:

- **master:** Содержит код продакшн-версии. Все релизы начинаются и завершаются в этой ветке.
- **develop:** Разработка, содержащая последние изменения кода для следующего релиза. Все фичи и исправления сначала сливаются сюда.

Вспомогательные:

- **feature:** Для разработки новых функций. Каждая новая функция создается в отдельной ветке feature и в конечном итоге сливается обратно в develop.
- **release:** Для подготовки новых продуктовых релизов. Они позволяют последние подгонки и исправления перед слиянием ветки release в master и develop.
- **hotfix:** Для немедленного исправления ошибок в продакшн-версии кода. Hotfix-ветки создаются от master и, после исправления, сливаются обратно в master и develop.

Преимущества:

- **Структурированность:** Структура ветвления помогает командам организовать процесс разработки и облегчает навигацию по репозиторию.
- **Гибкость:** Позволяет командам адаптироваться к различным сценариям разработки, будь то быстрое внедрение исправлений или длительная разработка новых функций.

- **Поддержка:** Распространенная модель с многочисленными руководствами и инструментами, поддерживающими этот подход.

Недостатки:

- **Сложность:** Для небольших проектов может показаться излишне сложной и громоздкой из-за большого количества веток и правил.
- **Непрерывная доставка:** Может быть не лучшим выбором для проектов, использующих методологии непрерывной интеграции (CI) и непрерывной доставки (CD), поскольку модель предполагает более длительные циклы разработки и релизов.

Git Flow популярная модель ветвления для крупных проектов и команд, которым необходим четкий и организованный процесс управления версиями.

Обратно

Проект **Celery** представляет собой распределенную систему для выполнения асинхронных задач. Он позволяет создавать, планировать и выполнять задачи параллельно на нескольких рабочих узлах или процессах, что обеспечивает масштабируемость и эффективное использование ресурсов.

Основные компоненты проекта Celery:

1. **Брокер сообщений (Message Broker):** Использует брокер сообщений для передачи задач между клиентскими приложениями и рабочими процессами. Популярными брокерами сообщений для Celery являются RabbitMQ, Redis и Apache Kafka.

2. **Очередь задач (Task Queue):** Очередь задач является центральной частью Celery и отвечает за прием, хранение и распределение задач между рабочими процессами. Очередь задач позволяет эффективно управлять и планировать выполнение задач в асинхронной среде.

3. **Рабочий процесс (Worker):** Рабочий процесс - это процесс или узел, который выполняет асинхронные задачи, полученные из очереди задач. Клиентское приложение отправляет задачи в очередь, а затем рабочие процессы забирают и выполняют эти задачи в фоновом режиме.

4. **Задача (Task):** Задача представляет собой функцию или метод, которая должна быть выполнена асинхронно. Она может быть определена с использованием декоратора `**@task**` и добавлена в очередь задач для выполнения.

5. **Результат (Result):** Позволяет получать результат выполнения задачи после ее завершения. Результат может быть получен немедленно или асинхронно, в зависимости от настроек и требований приложения.

Проект Celery обеспечивает гибкую и масштабируемую инфраструктуру для выполнения асинхронных задач. Он широко используется для обработки фоновых задач в веб-приложениях, обработки данных, отправки электронных писем, создания расписаний и многих других сценариев использования, где требуется асинхронная обработка и выполнение задач.

2. Какие компоненты составляют Celery?

- Ответ: Основными компонентами Celery являются:

    - Задачи (Tasks): Это функции, которые выполняются асинхронно.

    - Очереди (Queues): Хранят и управляют задачами, ожидающими выполнения.

    - Брокер сообщений (Message Broker): Передает сообщения между клиентами Celery и рабочими процессами.

    - Рабочие процессы (Workers): Выполняют задачи, полученные из очередей.

3. Какие брокеры сообщений поддерживает Celery?

- Ответ: Celery поддерживает различные брокеры сообщений, такие как RabbitMQ, Redis, Amazon SQS и другие. Они используются для обмена сообщениями между производителями и потребителями задач.

4. Как работает механизм периодических задач в Celery?

- Ответ: Celery поддерживает механизм периодических задач с помощью расширения "Celery Beat". Он определяет расписание задач с помощью CRON-подобной нотации времени и отправляет задачи на выполнение в определенное время.

5. Как обеспечивается масштабируемость и отказоустойчивость в Celery?

- Ответ: Масштабируемость Celery достигается за счет добавления дополнительных рабочих процессов и/или узлов, которые могут обрабатывать задачи. Отказоустойчивость обеспечивается с помощью механизмов повторной обработки (retry), обработки исключений и механизмов управления ошибками.

—

В веб-приложениях очереди часто используются для отложенной обработки событий или в качестве временного буфера между другими сервисами, тем самым защищая их от всплесков нагрузки.

pull-модель — консьюмеры сами отправляют запрос раз в n секунд на сервер для получения новой порции сообщений. При таком подходе клиенты могут эффективно контролировать собственную нагрузку. Кроме того, pull-модель позволяет группировать сообщения в батчи, таким образом достигая лучшей пропускной способности. К минусам модели можно отнести потенциальную разбалансированность нагрузки между разными консьюмерами, а также более высокую задержку обработки данных.

push-модель — сервер делает запрос к клиенту, посылая ему новую порцию данных. По такой модели, например, работает RabbitMQ. Она снижает задержку обработки сообщений и позволяет эффективно балансировать распределение сообщений по консьюмерам. Но для предотвращения перегрузки консьюмеров в случае с RabbitMQ клиентам приходится использовать функционал QS, выставляя лимиты.

Типичный жизненный цикл сообщений в системах очередей:

1 Продюсер отправляет сообщение на сервер.

2 Консьюмер фетчит (от англ. fetch — принести) сообщение и его уникальный идентификатор сервера.

3 Сервер помечает сообщение как in-flight. Сообщения в таком состоянии всё ещё хранятся на сервере, но временно не доставляются другим консьюмерам. Таймаут этого состояния контролируется специальной настройкой.

4 Консьюмер обрабатывает сообщение, следуя бизнес-логике. Затем отправляет ack или nack-запрос обратно на сервер, используя уникальный идентификатор, полученный ранее — тем самым либо подтверждая успешную обработку сообщения, либо сигнализируя об ошибке.

5 В случае успеха сообщение удаляется с сервера навсегда. В случае ошибки или таймаута состояния in-flight сообщение доставляется консьюмеру для повторной обработки.

Обратно

четыре уровня изоляции транзакций:

1. **READ UNCOMMITTED (Неподтвержденное чтение):**

- Это самый низкий уровень изоляции.
- Позволяет одной транзакции видеть изменения, внесенные другой, даже если эти изменения еще не зафиксированы (т.е., транзакция еще не завершена).
- Возможно возникновение проблем с неподтвержденным чтением и потерянными обновлениями.

2. **READ COMMITTED (Подтвержденное чтение):**

- Этот уровень гарантирует, что транзакция увидит только изменения, которые были подтверждены другими.
- Избегает проблем с неподтвержденным чтением, но может возникнуть проблема с повторяемым чтением.

3. **REPEATABLE READ (Повторяемое чтение):**

- Гарантирует, что каждый раз, когда транзакция читает данные, она видит те же самые данные, что и в начале, даже если другая вносит изменения в эти данные.
- Избегает проблемы с повторяемым чтением, но может привести к проблемам с фантомными записями (phantom reads).

4. **SERIALIZABLE (Сериализуемость):**

- Это самый высокий уровень изоляции.
- Гарантирует, что все операции чтения и записи будут видеть состояние данных, как если бы они выполнялись последовательно (одна за другой), даже если

фактически они выполняются параллельно.

- Избегает проблем с повторяемым чтением и фантомными записями, но может привести к увеличению блокировок и ухудшению производительности.

Выбор уровня изоляции зависит от конкретных требований вашего приложения к согласованности данных и производительности.

Обратно

Бинарное дерево - это структура данных, состоящая из узлов, каждый из которых имеет не более двух потомков: левого и правого. Каждый узел содержит некоторое значение (ключ) и ссылки на его левого и правого потомков (или на None, если потомок отсутствует).

Основные характеристики бинарного дерева:

1. **Корень (Root):** Это верхний узел дерева, от которого начинается структура.

2. **Уровень (Level):** Каждый узел дерева расположен на определенном уровне. Уровень корня равен 0, уровень его детей равен 1 и так далее.

3. **Листья (Leaves):** Это узлы дерева, у которых отсутствуют потомки. То есть они конечные в дереве.

4. **Высота (Height):** Это количество уровней в дереве. Максимальная высота бинарного дерева равна максимальному количеству уровней от корня до самого далекого листа.

5. **Балансировка (Balanced):** Бинарное дерево называется сбалансированным, если разница в высоте между левым и правым поддеревьями на каждом узле не превышает 1. Такие деревья обеспечивают эффективный поиск, вставку и удаление элементов.

6. **Порядок (Traversal):** Существуют различные способы обхода бинарного дерева для выполнения операций с его элементами. Эти методы включают в себя префиксный (pre-order), инфиксный (in-order) и постфиксный (post-order) обходы.

7. **Операции:** Бинарное дерево обычно поддерживает операции вставки, удаления и поиска элементов. Эти операции выполняются за время O(log n) в сбалансированных деревьях и до O(n) в худшем случае в несбалансированных.

Обратно

---

1. Что такое хэш-функция и для чего она используется?

- Ответ: Хэш-функция - это функция, которая преобразует входные данные произвольного размера в значение фиксированного размера. Она широко используется для хэширования паролей, поиска данных в базе данных, проверки целостности данных, создания уникальных идентификаторов и т.д.

2. Какие свойства должны иметь хорошая хэш-функция?

- Ответ: Хорошая хэш-функция должна обладать следующими свойствами:

- Равномерное распределение: Хэши должны равномерно распределяться по всему диапазону значений.

- Независимость от вида данных: Малые изменения во входных данных должны приводить к значительным изменениям в хэше.

- Устойчивость к коллизиям: Коллизии должны быть редкими, то есть различные входные данные не должны приводить к одинаковым хэшам.

3. Что такое коллизия в контексте хэш-функций?

- Ответ: Коллизия происходит, когда два разных входных значения приводят к одному и тому же хэшу. Хорошая хэш-функция должна сводить вероятность коллизий к минимуму.

4. Каковы методы решения коллизий в хэш-таблицах?

- Ответ: Существуют различные методы решения коллизий, включая метод цепочек (простое связывание), метод открытой адресации (открытое линейное и квадратичное зондирование), метод двойного хэширования и другие.

5. Что такое атака поиска предобразовании (Preimage Attack) на хэш-функции?

- Ответ: Атака на поиск предобразования — это атака, при которой злоумышленник пытается найти входные данные (сообщение), которые приведут к заданному хэш-значению. Хорошие хэш-функции должны быть устойчивы к таким атакам.

Обратно

---

1. **Что такое Django и для чего он используется?**

- **Ответ:** Django - это мощный веб-фреймворк на языке Python, который используется для разработки веб-приложений. Он предоставляет множество инструментов и функций для эффективной работы с базами данных, URL-адресацией, шаблонами, административной панелью и многими другими задачами.

2. **Каковы основные компоненты фреймворка Django?**

- **Ответ:** Основными компонентами Django являются ORM (Object-Relational Mapping), система шаблонов, система маршрутизации URL, аутентификация и административная панель.

3. **Каковы основные преимущества и недостатки Django?**

- **Ответ:** Преимущества Django включают в себя высокую производительность, хорошо организованную архитектуру, встроенную административную панель и обширное сообщество разработчиков. Среди недостатков можно отметить некоторую сложность для начинающих разработчиков и ограничения в выборе базы данных.

4**. **Что такое Django ORM?****

- **Ответ:** Django ORM (Object-Relational Mapping) - это инструмент, который позволяет взаимодействовать с базой данных на языке Python, используя объектно-ориентированный подход.

5. **Как организовать маршрутизацию URL в Django?**

- **Ответ:** Для организации маршрутизации URL в Django используется файл urls.py, в котором определяются пути и связанные с ними представления (views).

6. **Какова структура шаблонов в Django?**

- **Ответ:** Шаблоны в Django обычно организованы в отдельной директории templates, которая содержит HTML файлы и другие необходимые ресурсы.

7. **Что такое middleware в Django?**

- **Ответ:** Middleware в Django - это слои обработки запросов, которые пропускаются запросом при его обработке. Они используются для выполнения общих операций, таких как аутентификация, контроль доступа, логирование и другие.

8. **Как настраивается аутентификация в Django?**

- **Ответ:** Django предоставляет встроенную систему аутентификации, которая может быть настроена в settings.py. Она позволяет управлять пользователями, правами доступа и другими аспектами аутентификации.

9. **Как переопределить стандартное поведение административной панели в Django?**

- **Ответ:** Стандартное поведение административной панели Django может быть переопределено путем создания и регистрации собственных моделей в admin.py и определения соответствующих представлений в files.py.

10. **Какие инструменты в Django доступны для обработки форм?1

- **Ответ:** В Django доступны инструменты как для создания форм вручную, так и встроенные средства для автоматического создания форм на основе моделей.

11. **Как исполняется условная выборка в запросах к базе данных в Django ORM?**

- **Ответ:** Для условной выборки в Django ORM используется метод filter(), позволяющий фильтровать записи в соответствии с заданными условиями.

12. **Каковы способы внедрения сторонних пакетов (плагинов) в Django?**

- **Ответ:** Сторонние пакеты могут быть внедрены с помощью установки и добавления в файл requirements.txt или через pip, добавляя их в INSTALLED_APPS в settings.py.

13. **Какие методы кэширования доступны в Django?**

- **Ответ:** Django предоставляет несколько методов кэширования, включая кэширование в памяти, на диске, с использованием базы данных или сторонних кэширующих систем.

14. **Что такое миграции в Django и для чего они используются?**

- **Ответ:** Миграции в Django - это механизм, который автоматически изменяет схему базы данных при изменении моделей. Они используются для управления изменениями в базе данных.

15. **Как организуется масштабируемость Django-приложений?**

- **Ответ:** Масштабируемость Django-приложений может быть достигнута путем использования асинхронных задач в фоновом режиме, разделения микросервисов, горизонтального масштабирования и использования кэширования для улучшения производительности.

Обратно

Вопросы о Токен аутентификации (Authentication Tokens):

1. Что такое токен аутентификации и для чего он используется?

- Ответ: Токен аутентификации - это механизм, который используется для подтверждения личности пользователя и выдачи разрешений на доступ к ограниченным ресурсам во время выполнения запросов к API или веб-приложению.

2. Какие есть типы токенов аутентификации и как они работают?

- Ответ: Типы токенов аутентификации могут включать в себя токены безопасности, токены доступа, токены обновления и другие, которые предоставляют временные разрешения для доступа к конкретным ресурсам.

3. Как происходит обмен и обработка токенов аутентификации в веб-приложении?

- Ответ: Обычно обмен токенами аутентификации происходит после успешной проверки учетных данных пользователя и предоставления токена клиенту. Веб-приложение затем проверяет токен перед предоставлением доступа к ресурсам.

JWT (JSON Web Token) - это стандарт для создания токенов, которые могут использоваться для аутентификации и передачи информации между сторонами. Он использует формат JSON.

Вот как работает JWT:

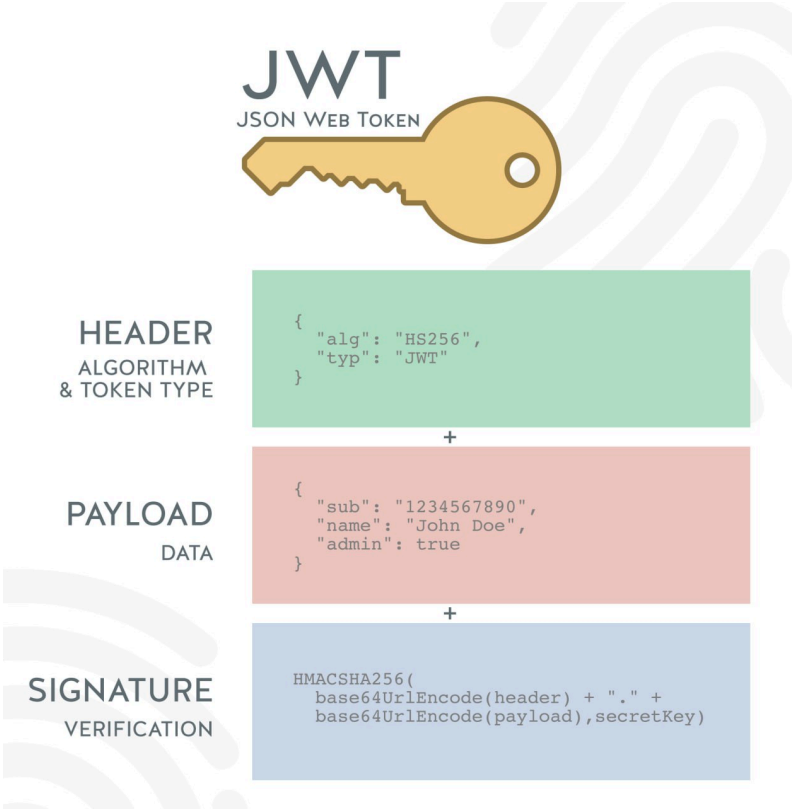
1. Создание токена: Пользователь или сервер создает токен, который содержит набор данных, таких как идентификатор пользователя или другая информация, которую они хотят передать.

2. Подпись токена: Токен подписывается с использованием секретного ключа. Это делает его неподдельным и обеспечивает его целостность.

3. Передача токена: Токен передается между клиентом и сервером через HTTP заголовок, обычно в форме Authorization: Bearer <токен>.

4. Проверка токена: Получатель токена может проверить его подпись, чтобы убедиться, что он не был изменен и что он был создан с использованием секретного ключа. Если проверка проходит успешно, получатель может использовать данные из токена для аутентификации пользователя или получения другой необходимой информации.

JWT обычно используется для создания токенов с информацией о пользователе после успешной аутентификации, чтобы идентифицировать пользователя при последующих запросах без необходимости повторной отправки учетных данных. Он также может использоваться для передачи других данных между сторонами, таких как права доступа или другие свойства пользователя.



Вопросы о JWT (JSON Web Tokens):

1. Что такое JWT и каковы его основные компоненты?

- Ответ: JWT (JSON Web Token) - это открытый стандарт (RFC7519) для безопасного обмена информацией между сторонами. Он состоит из заголовка, тела (payload) и подписи.

2. Каким образом JWT обеспечивает безопасность и подозрение в атаках?

- Ответ: JWT обеспечивает безопасность путем шифрования при помощи алгоритмов, таких как HMAC (with SHA-256) или RSA. Он также имеет механизм проверки подлинности и подписи.

3. В каких случаях рекомендуется использовать JWT для аутентификации в веб-приложениях?

- Ответ: JWT может быть рекомендован для использования в случаях, когда требуется безопасный обмен информацией между сторонами или аутентификация пользователя в различных областях приложения.

Вопросы о сессионных токенах (Session Tokens):

1. Какие возможности обеспечивают сессионные токены в веб-приложениях?

- Ответ: Сессионные токены позволяют отслеживать идентификацию сеанса пользователя и хранить состояние аутентифицированных пользователей на протяжении сеанса.

2. Как работают сессионные токены в рамках аутентификации и авторизации веб-приложения?

- Ответ: После успешной аутентификации, сервер создает уникальный сессионный токен, который сохраняется и используется для идентификации пользователя на протяжении его сеанса.

[Обратно]

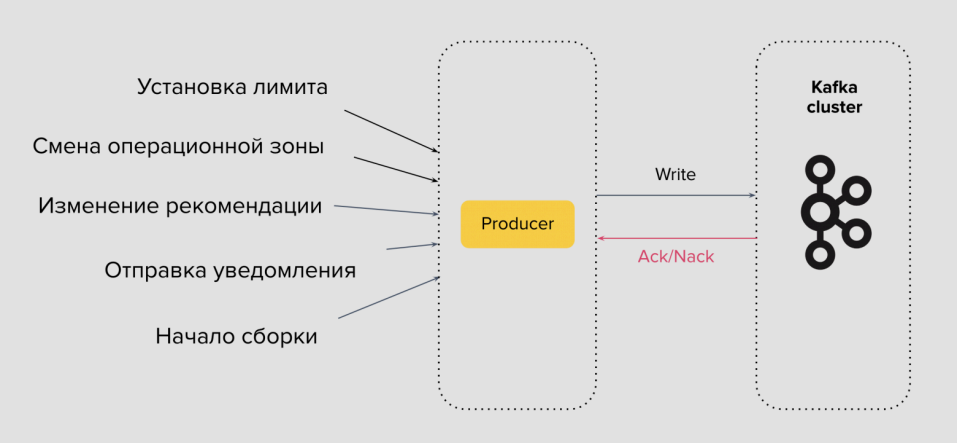
---

1. **Что такое Apache Kafka и для чего он используется?**

Ответ: Apache Kafka - это распределенная система потоковой обработки и обмена сообщениями. Он используется для упорядоченного, отказоустойчивого и масштабируемого обмена данными между приложениями. Kafka позволяет создавать потоки данных, обрабатывать события в реальном времени и строить системы, поддерживающие большие объемы данных.

Продюсеры

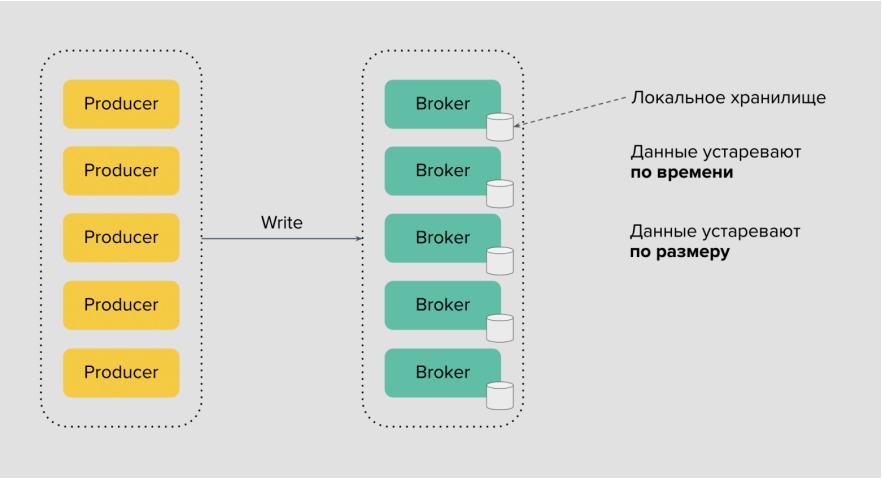
Для записи событий в кластер Kafka есть продюсеры — это приложения, которые вы разрабатываете.



Программа-продюсер записывает сообщение в Kafka, тот сохраняет события, возвращает подтверждение о записи или ***acknowledgement***. Продюсер получает его и начинает следующую запись.

Брокеры

Кластер Kafka состоит из брокеров. Можно представить систему как дата-центр и серверы в нём. При первом знакомстве думайте о Kafka-брокере как о компьютере: это процесс в операционной системе с доступом к своему локальному диску.

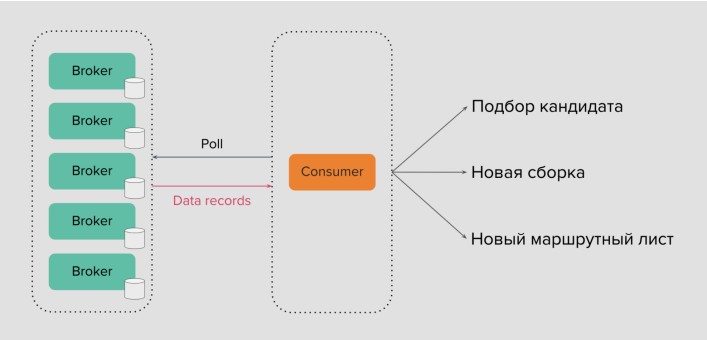


Все брокеры соединены друг с другом сетью и действуют сообща, образуя единый кластер. Когда мы говорим, что продюсеры пишут события в Kafka-кластер, то подразумеваем, что они работают с брокерами в нём.

Кстати, в облачной среде кластер не обязательно работает на выделенных серверах — это могут быть виртуальные машины или контейнеры в Kubernetes.

Консумеры

События, которые продюсеры записывают на локальные диски брокеров, могут читать консумеры — это тоже приложения, которые вы разрабатываете. В этом случае кластер Kafka — это по-прежнему нечто, что обслуживается инфраструктурой, где вы как пользователь пишете продюсеров и консумеров. Программа-консумер подписывается на события или ***поллит*** и получает данные в ответ. Так продолжается по кругу.



Основы кластера Kafka — это продюсер, брокер и консумер. Продюсер пишет сообщения в лог брокера, а консумер его читает.

Zookeeper — это выделенный кластер серверов для образования кворума-согласия и поддержки внутренних процессов Kafka. Благодаря этому инструменту мы можем управлять кластером Kafka: добавлять пользователей и топики, задавать им настройки.

Топик — это логическое разделение категорий сообщений на группы. Например, события по статусам заказов, координат партнёров, маршрутных листов и так далее.

Топики в Kafka разделены на ***партиции***. Увеличение партиций увеличивает параллелизм чтения и записи. Партиции находятся на одном или нескольких брокерах, что позволяет кластеру масштабироваться.

Партиции хранятся на локальных дисках брокеров и представлены набором лог-файлов — ***сегментов***. Запись в них идёт в конец, а уже сохранённые события неизменны.

Каждое сообщение в таком логе определяется порядковым номером — ***оффсетом***. Этот номер монотонно увеличивается при записи для каждой партиции.

Лог-файлы на диске устаревают по времени или размеру. Настроить это можно глобально или индивидуально в каждом топике.

Для отказоустойчивости, партиции могут реплицироваться. Число реплик или ***фактор репликации*** настраивается как глобально по умолчанию, так и отдельно в каждом топике.

Реплики партициий могут быть ***лидерами*** или ***фолловерами***. Традиционно консумеры и продюсеры работают с лидерами, а фолловеры только догоняют лидера.

Поток данных как лог

Сегмент тоже удобно представить как обычный лог-файл: каждая следующая запись добавляется в конец файла и не меняет предыдущих записей. Фактически это очередь FIFO (First-In-First-Out) и Kafka реализует именно эту модель.

Репликация данных

Если одна партиция будет существовать на одном брокере, в случае сбоя часть данных станет недоступной. Такая система будет хрупкой и ненадёжной. Для решения проблемы Kafka представляет механизм репликации партиций между брокерами.

У каждой партиции есть настраиваемое число реплик, на изображении их три. Одна из этих реплик называется ***лидером***, остальные — ***фолловерами***. Продюсер подключается к брокеру, на котором расположена лидер-партиция, чтобы записать в неё данные.

Семантики доставки

В любых очередях есть выбор между скоростью доставки и расходами на надёжность. Цветные квадратики на слайде — это сообщения, которые мы будем записывать в очередь, выбирая нужную из семантик.

- Семантика ***at-most once*** означает, что при доставке сообщений нас устраивают потери сообщений, но не их дубликаты. Это самая слабая гарантия, которую реализуют брокерами очередей
- Семантика ***at-least once*** означает, что мы не хотим терять сообщения, но нас устраивают возможные дубликаты
- Семантика ***exactly-once*** означает, что мы хотим доставить одно и только одно сообщение, ничего не теряя и ничего не дублируя.

Надёжность доставки

Со стороны продюсера разработчик определяет надёжность доставки сообщения до Kafka с помощью параметра ***acks***. Указывая ***0*** или ***none***, продюсер будет отправлять сообщения в Kafka, не дожидаясь никаких подтверждений записи на диск со стороны брокера.

Это самая слабая гарантия. В случае выхода брокера из строя или сетевых проблем, вы так и не узнаете, попало сообщение в лог или просто потерялось.

Указывая настройку в ***1*** или ***leader***, продюсер при записи будет дожидаться ответа от брокера с лидерской партицией — значит, сообщение сохранено на диск одного брокера. В этом случае вы получаете гарантию, что сообщение было получено по крайней мере один раз, но это всё ещё не страхует вас от проблем в самом кластере.

Представьте, что в момент подтверждения брокер с лидерской партиции выходит из строя, а фолловеры не успели отреплицировать с него данные. В таком случае вы теряете сообщение и снова не узнаёте об этом. Такие ситуации случаются редко, но они возможны.

Наконец, устанавливая acks в ***-1*** или ***all***, вы просите брокера с лидерской партицией отправить вам подтверждение только тогда, когда запись попадёт на локальный диск брокера и в реплики-фолловеры. Число этих реплик устанавливает настройка ***min.insync.replicas***.

Лидер группы отвечает за распределение партиций между всеми участниками группы. Процесс поиска лидера группы, назначения партиций, стабилизации и подключения консумеров в группе к брокерам называется **ребалансировкой консумер-группы**.

2. Какие ключевые компоненты составляют архитектуру Apache Kafka?

Ответ:

- Producer: Отправляет сообщения в темы (topics) Kafka.

- Broker: Сервер Kafka, хранящий данные и обслуживающий запросы клиентов.

- Topic: Категория, под которой сообщения хранятся.

- Consumer: Получает и обрабатывает сообщения из тем Kafka.

3. Чем отличаются Kafka Producer API и Kafka Consumer API

Ответ:

- Producer API: Используется для отправки сообщений в Kafka. Producer может выбирать между отслеживанием отправленных сообщений или не требовать подтверждения.

- Consumer API: Предназначен для вычитывания сообщений из Kafka. Consumer может работать в режиме с автоматическим или ручным управлением смещением.

4. Какие бывают стратегии сбора данных (data retention) в Kafka и как выбрать подходящую?

Ответ: В Kafka существуют две стратегии сбора данных: по времени и по размеру.

- По времени: Указывает, сколько времени Kafka будет хранить сообщения в теме.

- По размеру: Определяет максимальное количество данных для хранения в теме.

Выбор стратегии зависит от требований к данным и обработке потока сообщений.

5. Как обеспечить отказоустойчивость Kafka?

Ответ: Для обеспечения отказоустойчивости в Kafka можно использовать репликацию тем. Репликация позволяет создавать несколько копий данных на разных брокерах, обеспечивая высокую доступность и избежание потерь данных при сбоях.

6. Какие есть инструменты для мониторинга и управления Kafka?

Ответ: Для мониторинга Kafka можно использовать инструменты, такие как Kafka Manager, Confluent Control Center, Burrow, и другие. Эти инструменты позволяют отслеживать производительность, масштабировать кластеры, управлять темами и потребителями.

Завершение:

Знание Apache Kafka важно для Python Middle Developer, так как Kafka широко используется для обработки данных в режиме реального времени, поэтому понимание его ключевых компонентов, функциональности и возможностей поможет участнику собеседования проявить компетентность и готовность к работе с потоковой обработкой данных.

Обратно

Lock (замок):

В Python, Lock представляет собой примитив синхронизации, который используется для обеспечения доступа к общим ресурсам в многопоточной среде. Lock позволяет только одному потоку выполнить защищенный участок кода в определенный момент времени, тем самым предотвращая конфликты доступа к данным. Когда Lock заблокирован одним потоком, другие потоки ждут, пока Lock не будет освобожден, прежде чем продолжить выполнение.

Пример использования Lock в Python:

```
import threading


lock = threading.Lock()


def some_function():

lock.acquire()

try:

        # Secure section of code

        print("Critical section protected by Lock")

finally:

        lock.release()
```

 # Для использования:

```
t1 = threading.Thread(target=some_function)

t2 = threading.Thread(target=some_function)


t1.start()

t2.start()


t1.join()

t2.join()
```

Semaphore (семафор):

```
t2.join()
```

##- это синхронизационный механизм, который обычно используется для контроля доступа к пулу ограниченных ресурсов. В отличие от Lock, Semaphore позволяет ограничить количество потоков, которые могут получить доступ к общим ресурсам одновременно. Семафор содержит внутреннее счетчик, который уменьшается каждый раз при захвате семафора и увеличивается при его освобождении.

Пример использования Semaphore в Python:

```
import threading

# Создание Semaphore с начальным значением 2, что позволяет двум потокам одновременно получить доступ

semaphore = threading.Semaphore(2)


def some_function():

with semaphore:

        # Secure section of code

        print("Critical section protected by Semaphore")

# Для использования:

t1 = threading.Thread(target=some_function)

t2 = threading.Thread(target=some_function)

t3 = threading.Thread(target=some_function)


t1.start()

t2.start()

t3.start()


t1.join()

t2.join()
```

Заключение:

Lock используется для синхронизации доступа к общим ресурсам между потоками, позволяя только одному потоку за раз выполнять защищенный код. Semaphore, с другой стороны, позволяет ограничить количество потоков, которые имеют доступ к ресурсам одновременно.

Оба механизма синхронизации являются важными инструментами в многопоточном программировании Python и помогают избежать состязания за ресурсы между потоками, обеспечивая безопасность и согласованность данных

Обратно

---

1. Что такое Grafana и для чего его используют?

Ответ: Grafana - это популярный инструмент для визуализации данных и мониторинга. Он позволяет создавать графики, диаграммы и панели мониторинга в реальном времени на основе данных из различных источников, таких как базы данных, временные ряды, API и другие.

2. Какие источники данных поддерживает Grafana?

Ответ: Grafana поддерживает широкий спектр источников данных, включая базы данных (например, InfluxDB, Prometheus, MySQL), временные ряды (например, Graphite, OpenTSDB), облачные сервисы (например, AWS CloudWatch, Google Stackdriver) и многие другие.

3. Как создать новую панель (Dashboard) в Grafana?

Ответ: Для создания новой панели в Grafana, необходимо:

1. Зайти в интерфейс Grafana и выбрать "Create" -> "Dashboard".

2. Добавить новую панель и выбрать источник данных.

3. Настроить запрос данных и визуализацию, используя различные плагины и параметры.

4. Какие типы визуализации данных поддерживает Grafana?

Ответ: Grafana поддерживает различные типы визуализации данных, такие как графики временных рядов, столбчатые диаграммы, тепловые карты, географические карты, графики событий, таблицы и другие. При этом можно использовать различные плагины для расширения возможностей визуализации.

5. Как настроить оповещения (Alerting) в Grafana?

Ответ: Для настройки оповещений в Grafana:

1. Создайте оповещение в панели мониторинга, задав условия тревоги.

2. Выберите тип оповещения (например, по электронной почте, через API и т.д.).

3. Настройте расписание оповещений и другие параметры по необходимости.

6. Каким образом можно резервно сохранять данные и настройки Grafana?

Ответ: Для резервного копирования данных и настроек Grafana можно использовать различные методы, такие как:

- Регулярное резервное копирование файлов конфигурации.

- Использование инструментов резервного копирования базы данных, если данные хранятся в базе данных.

- Использование встроенной функциональности создания снимков (snapshots) в Grafana.

Заключение:

Понимание основных функций и возможностей Grafana помогает пользователям эффективно визуализировать и мониторить данные, управлять панелями и оповещениями, а также сохранять данные и настройки для обеспечения надежности и готовности к восстановлению.

Обратно

---

1. SELECT

- Описание: Извлекает данные из базы данных.

- Пример: SELECT * FROM table_name;

2. INSERT

- Описание: Добавляет новые строки данных в таблицу.

- Пример: INSERT INTO table_name (column1, column2) VALUES (value1, value2);

3. UPDATE

- Описание: Обновляет существующие строки данных в таблице.

- Пример: UPDATE table_name SET column1 = value1 WHERE condition;

4. DELETE

- Описание: Удаляет строки данных из таблицы.

- Пример: DELETE FROM table_name WHERE condition;

5. CREATE TABLE

- Описание: Создает новую таблицу в базе данных.

- Пример: CREATE TABLE table_name (column1 datatype, column2 datatype);

6. ALTER TABLE

- Описание: Меняет структуру существующей таблицы (добавляет, удаляет или изменяет столбцы).

- Пример: ALTER TABLE table_name ADD column_name datatype;

7. DROP TABLE

- Описание: Удаляет таблицу из базы данных.

- Пример: DROP TABLE table_name;

8. CREATE INDEX

- Описание: Создает индекс на одном или нескольких столбцах таблицы для ускорения поиска данных.

- Пример: CREATE INDEX index_name ON table_name (column_name);

9. JOIN

- Описание: Объединяет данные из двух таблиц.

- Пример: SELECT * FROM table1 JOIN table2 ON table1.column = table2.column;

10. GROUP BY

- Описание: Группирует строки на основе значения столбца.

- Пример: SELECT column1, COUNT(column2) FROM table_name GROUP BY column1;

11. ORDER BY

- Описание: Сортирует результаты запроса.

- Пример: SELECT * FROM table_name ORDER BY column_name ASC;

12. COUNT

- Описание: Возвращает количество строк в результате запроса.

- Пример: SELECT COUNT(*) FROM table_name;

Обратно

---

1. **Что такое сокет?**

— Сокет — это конечная точка для связи между двумя компьютерами по сети. Он состоит из IP-адреса и номера порта.

2. **В чем разница между сокетами TCP и UDP?**

- TCP обеспечивает надежную связь, ориентированную на установление соединения, тогда как UDP обеспечивает более быструю, но ненадежную связь без установления соединения.

3. **Как устанавливается сокетное соединение?**

- Соединение через сокет устанавливается посредством процесса, называемого «трехэтапным рукопожатием», при котором клиент и сервер обмениваются пакетами синхронизации и подтверждения для установления соединения.

4. **Каково значение IP-адреса и номера порта при сокетном соединении?**

- IP-адрес идентифицирует хост, а номер порта указывает приложение или службу на этом хосте. Вместе они обеспечивают связь между процессами.

5. **Как обрабатывать ошибки при программировании сокетов?**

— Ошибки в программировании сокетов можно обрабатывать, проверяя возвращаемые значения функций сокетов на наличие кодов ошибок и реализуя соответствующие механизмы обработки ошибок, такие как регистрация, повторная попытка или закрытие соединения.

6. **Объясните модель клиент-сервер в программировании сокетов.**

- В модели клиент-сервер сервер прослушивает входящие соединения от клиентов, в то время как клиенты инициируют соединения с серверами для запроса услуг или обмена данными.

7. **Что такое блокирующий и неблокирующий сокетный ввод-вывод?**

- При блокировке сокетного ввода-вывода программа ожидает отправки или получения данных, что может привести к зависанию программы. Неблокирующий ввод-вывод сокетов позволяет программе продолжать выполнение во время ожидания данных, улучшая параллелизм.

8. **Как обрабатывать несколько клиентских подключений в серверной программе?**

- Множественными клиентскими подключениями на сервере можно управлять с помощью таких методов, как многопоточность, многопроцессорность или программирование, управляемое событиями, для одновременной обработки и обработки входящих клиентских запросов.

9. **Объясните роль параметров сокета в программировании сокетов.**

- Параметры сокетов позволяют точно настраивать поведение сокетов, например устанавливать тайм-ауты, включать/отключать такие функции, как сохранение активности, а также настраивать размеры буфера для повышения производительности и стабильности.

10. **Каковы общие проблемы безопасности при программировании сокетов?

- Общие проблемы безопасности включают шифрование данных для обеспечения конфиденциальности, проверку ввода для предотвращения атак путем внедрения и механизмы авторизации для контроля доступа к ресурсам.

Обратно

---

1. **Что такое модульное тестирование в Python?

— Модульное тестирование в Python включает в себя изолированное тестирование отдельных модулей или компонентов программы, чтобы убедиться, что каждый модуль работает правильно.

2. **Как вы выполняете модульное тестирование в Python?

— Python предлагает встроенные библиотеки, такие как unittest и pytest, для написания и запуска модульных тестов. Эти платформы предоставляют функциональные возможности для создания тестовых примеров, запуска тестов и создания отчетов.

3. **Какова цель тестовых приспособлений при тестировании Python?

- Тестовые приспособления при тестировании Python используются для настройки предварительных условий перед запуском тестов и очистки ресурсов после выполнения тестов, обеспечивая согласованную и контролируемую среду для тестирования.

4. **Объясните концепцию макетирования при тестировании Python.

— Мокинг при тестировании Python предполагает создание макетов объектов для имитации поведения реальных объектов. Это полезно для изоляции тестируемого кода и сосредоточения внимания на конкретных компонентах без зависимостей.

5. **Как вы обрабатываете исключения при тестировании Python?

- При тестировании Python исключения могут обрабатываться с помощью таких конструкций, как блоки try-Exception в тестовых примерах, или с помощью методов утверждения, предоставляемых средами тестирования, для проверки ожидаемых исключений.

6. **Что такое регрессионное тестирование и почему оно важно?

- Регрессионное тестирование включает в себя повторный запуск предыдущих тестов, чтобы убедиться, что недавние изменения кода не оказали негативного влияния на существующие функциональные возможности. Это помогает поддерживать стабильность и надежность программного продукта с течением времени.

7. **Каковы рекомендации по написанию эффективных тестовых примеров на Python?

- Лучшие практики включают в себя написание независимых и изолированных тестовых примеров, использование описательных названий тестов, сосредоточение внимания на крайних случаях и граничных условиях, обеспечение простоты и читабельности тестов, а также обеспечение удобства обслуживания и простоты запуска тестов.

8. **Как вы измеряете покрытие кода при тестировании Python?

— Инструменты покрытия кода, такие как «coverage.py», можно использовать для измерения процента строк кода, выполненных во время тестирования. Это помогает выявить непроверенные части кода и обеспечивает более высокий уровень покрытия тестами.

9. **Что такое макетирование и как оно используется при тестировании Python?**

— Мокинг — это процесс создания поддельных объектов, имитирующих поведение реальных объектов. Он используется при тестировании Python для изоляции тестируемого кода путем замены внешних зависимостей фиктивными объектами, что обеспечивает более целенаправленное и контролируемое тестирование.

10. **Объясните разницу между unittest и pytest при тестировании Python.**

— «unittest» — это встроенная среда тестирования Python, а «pytest» — это сторонняя среда тестирования, предоставляющая больше возможностей и гибкости. pytest упрощает написание тестов, предлагает мощные инструменты и поддерживает параметризованное тестирование «из коробки».

Обратно

MongoDB — популярная система управления базами данных NoSQL, использующая документно-ориентированную модель данных. Вот некоторые распространенные вопросы на собеседованиях о MongoDB и ответы на них:

1. Что такое MongoDB?

Ответ: MongoDB — это программа базы данных NoSQL, использующая документно-ориентированную модель данных. Он разработан с учетом масштабируемости, гибкости и производительности.

2. Что такое документ в MongoDB?

Ответ: В MongoDB документ представляет собой структуру данных, состоящую из пар поле-значение. Он похож на объект JSON и является основной единицей хранения данных.

3. Что такое коллекция в MongoDB?

Ответ: Коллекция в MongoDB — это группа документов MongoDB. Это эквивалент таблицы в реляционной базе данных.

4. В чем разница между базами данных MongoDB и SQL?

Ответ: MongoDB — это база данных NoSQL, а базы данных SQL — реляционные базы данных. MongoDB использует гибкую схему, тогда как базы данных SQL имеют фиксированную схему.

5. Как работает индексирование в MongoDB?

Ответ: Индексы в MongoDB аналогичны индексам в реляционных базах данных и повышают производительность запросов, позволяя базе данных быстро находить документы.

6. Что такое шардинг в MongoDB?

Ответ: Шардинг в MongoDB — это процесс распределения данных по нескольким машинам для улучшения масштабируемости и производительности.

7. В чем преимущество использования MongoDB перед базами данных SQL?

Ответ: MongoDB предлагает большую масштабируемость и гибкость благодаря своей документно-ориентированной модели данных, что делает ее хорошо подходящей для обработки неструктурированных или быстро развивающихся данных.

8. Объясните концепцию репликации в MongoDB.

Ответ: Репликация в MongoDB подразумевает поддержание нескольких копий данных для обеспечения высокой доступности и избыточности данных в случае сбоев.

9. Как выполнить агрегацию в MongoDB?

Ответ: MongoDB предоставляет структуру агрегации, которая позволяет пользователям выполнять операции с несколькими документами и возвращать вычисленные результаты.

10. Какое значение имеет ObjectId в MongoDB?

Ответ: ObjectId — это уникальный идентификатор, создаваемый MongoDB для каждого документа. Он состоит из метки времени, идентификатора машины, идентификатора процесса и случайного значения.

Обратно

Формы в Django используются для создания HTML-форм для сбора и обработки вводимых пользователем данных. Они упрощают процесс обработки данных формы и проверки вводимых пользователем данных.

Менеджеры в Django — это служебные методы, позволяющие взаимодействовать с моделями баз данных. Они предоставляют методы для запросов к базе данных и выполнения различных операций над объектами модели.

Обратно

В Python понятия public, Private и protected являются соглашениями, а не строгими спецификаторами доступа, как в некоторых других языках программирования.

1. Общедоступный. По умолчанию все атрибуты и методы класса Python являются общедоступными. Доступ к ним можно получить из любого места, как внутри класса, так и за его пределами.

Пример:

```
класс МойКласс:
    защита __init__(сам):
    self.public_variable = 10
```

```
объект = МойКласс()

print(obj.public_variable) # Доступ к общедоступной переменной
```

2. Частный. В Python атрибуты и методы можно пометить как частные, добавив к ним двойное подчеркивание __. Эти атрибуты недоступны напрямую извне класса. Однако Python не обеспечивает строгий контроль видимости, и к ним все равно можно получить доступ с помощью изменения имени (`_ClassName__private_variable`).

Пример:

```
класс МойКласс:

    защита __init__(сам):

    self.__private_variable = 20


объект = МойКласс()

print(obj._MyClass__private_variable) # Доступ к частной переменной с использованием изменения имени
```

3. Защищено. В Python атрибуты и методы можно пометить как защищенные, добавив к ним одинарное подчеркивание _. Это указывает другим разработчикам, что к атрибуту или методу не следует обращаться напрямую, но это не обеспечивает строгий контроль доступа.

Пример:

```
класс МойКласс:

    защита __init__(сам):

    self._protected_variable = 30


объект = МойКласс()

print(obj._protected_variable) # Доступ к защищенной переменной
```

Помните, что хотя Python предоставляет эти соглашения, обеспечение инкапсуляции и контроля доступа в конечном итоге является обязанностью программиста, а не самого языка.

Обратно

1. Что такое HTTP и для чего он используется?

- Ответ: HTTP (Hypertext Transfer Protocol) - это протокол передачи данных в сети Интернет, который используется для передачи документов и других ресурсов в формате HTML и других типов данных. Он является основным протоколом для передачи информации в вебе.

2. Какие основные методы HTTP запросов вы знаете и в чем их различия?

- Ответ: Основные методы HTTP запросов включают GET, POST, PUT, DELETE, OPTIONS, HEAD и другие. Каждый метод имеет свою уникальную цель и характеристики, такие как получение данных, отправка данных, обновление или удаление ресурсов и т.д.

3. Чем отличаются HTTP и HTTPS?

- Ответ: HTTPS (HTTP Secure) - это защищенная версия HTTP, которая использует шифрование (SSL/TLS) для безопасной передачи данных. В отличие от HTTP, который передает информацию в открытом виде, HTTPS обеспечивает уровень безопасности и конфиденциальности данных.

4. Что такое статус код HTTP и какие основные категории кодов вы знаете?

- Ответ: Статус коды HTTP - это числовые коды, возвращаемые сервером для указания результата обработки запроса. Основные категории включают информационные (1xx), успешные (2xx), перенаправления (3xx), ошибки клиента (4xx) и ошибки сервера (5xx).

5. Что такое заголовки HTTP и какие функции они выполняют?

- Ответ: Заголовки HTTP - это метаданные, отправляемые вместе с запросом или ответом. Они могут содержать информацию о типе содержимого, длине сообщения, методе запроса, куки и другие данные, влияющие на обработку и взаимодействие между клиентом и сервером.

6. Как происходит установление и завершение соединения в протоколе HTTP?

- Ответ: Установление соединения в HTTP осуществляется через установку TCP-соединения между клиентом и сервером. После передачи запроса и ответа, соединение может быть закрыто, как правило, после завершения ответа от сервера.

Эти вопросы и ответы могут помочь углубить понимание принципов работы и основных концепций HTTP на собеседовании**.**

12 Какие методы HTTP-запросов поддерживаются REST?

Ответ: Метод HTTP-запроса указывает желаемое действие, которое сервер выполнит над ресурсом. В REST существует четыре основных метода HTTP-запросов клиента к серверу:

- GET: Запрашивает ресурс у сервера, этот метод только для чтения.

- POST: Создает новый ресурс на сервере.
- PUT: Обновляет существующий ресурса на сервере.
- DELETE: Удаляет ресурс с сервера.

13. В чем разница между методом POST и методом PUT?

Ответ: POST предназначен для создания ресурса на сервере, в то время как PUT — для замены ресурса на определенном URI другим ресурсом. Если использовать PUT на URI, который уже имеет связанный с ним ресурс, PUT заменит его. Если ресурс на указанном URI отсутствует, PUT создает его.

PUT является идемпотентным, то есть его многократный вызов приведет к созданию только одного ресурса. Это происходит потому, что каждый вызов заменяет существующий ресурс (или создает новый, если заменять нечего).

POST не является идемпотентным. Если, к примеру вызвать POST 10 раз, то на сервере будут созданы 10 различных ресурсов, каждый со своим URI.

Хотя это редко применяется, ответы POST можно кэшировать, а ответы PUT нельзя. Запросы POST обычно считаются некешируемыми, но их можно кешировать, когда они содержат ясную информацию о “свежести” данных. Подробнее можно ответить так, что ответ для запроса POST (или PATCH) может быть закеширован, если указан признак "свежести" данных и установлен заголовок Content-Location (en-US), но это редко реализуется. Поэтому кэширование POST стоит избегать, если это возможно.

14. Из каких основных частей состоит HTTP-запрос?

Ответ: В REST существуют следующие основные компоненты HTTP-запроса:

- **Метод запроса**, который будет выполняться к ресурсу (т.е. GET, POST, PUT, DELETE).
- **URI,** идентифицирующий запрашиваемый ресурс на сервере.
- **Версия HTTP**, – т.е. какая версия должна быть в ответе сервера.
- **Заголовок HTTP-запроса** содержит метаданные о запросе, такие как агент пользователя, форматы файлов, принимаемые клиентом, формат тела запроса, язык, предпочтения по кэшированию и т.д.
- **Тело HTTP-запроса**, оно содержит все данные, связанные с запросом. Это необходимо только в том случае, если запрос направлен на изменение данных на сервере с помощью методов POST или PUT.

15. Каковы основные части HTTP-ответа?

Ответ: Ответы HTTP отправляются от сервера клиенту. Они информируют клиента о том, что запрошенное действие было (или не было) выполнено, и о доставке любых запрошенных ресурсов. Существует четыре основных компонента HTTP-ответа:

- Используемая **версия HTTP**.
- **Строка состояния** со статусом запроса и кода состояния HTTP-ответа.
- **Заголовок HTTP-ответа** с метаданными об ответе, включая время, имя сервера, агент пользователя, форматы файлов возвращаемых ресурсов, информацию о кэшировании.
- **Тело HTTP-ответа** с данными о ресурсе, который был запрошен клиентом

16. Назовите как минимум 3 кода успешных HTTP-ответов сервера

Ответ: сервер возвращает следующие коды статуса операции при успешной обработке запроса:

- 200 OK: Запрос выполнен успешно.
- 201 Created: Запрос прошел успешно, и ресурс был создан.

- 202 Accepted: Этот статус означает, что сервер принял запрос клиента, но не завершил его обработку. Обработка может быть асинхронной.

17. Назовите как минимум 4 кода HTTP-ответа сервера при перенаправлении запроса

Ответ: сервер возвращает следующие коды статуса при перенаправлении запроса:

- 301 Moved Permanently: Этот статус говорит клиенту, что запрошенный ресурс был постоянно перемещен на другой URL, и клиент должен обращаться к новому URL для доступа к ресурсу.
- 302 Found: Этот статус указывает на то, что ресурс временно перемещен на другой URL, и клиент должен временно использовать новый URL.
- 307 Temporary Redirect: Аналогично коду 302, но клиент должен использовать тот же метод запроса при обращении к новому URL.
- 308 Permanent Redirect: Аналогично 301, но клиент должен использовать тот же метод запроса при обращении к новому URL

18. Назовите как минимум 4 кода неуспешных HTTP-ответов сервера.

Ответ: сервер возвращает следующие коды при неуспешной обработке запроса:

- 400 Bad Request: Запрос не был выполнен из-за ошибки в запросе, например, опечатки или отсутствия данных.
- 401 Unauthorized: Запрос не был выполнен, поскольку клиент не прошел аутентификацию или не имеет права доступа к запрашиваемому ресурсу.
- 403 Forbidden: Запрос не был выполнен, поскольку клиент аутентифицирован, но не авторизован для доступа к запрашиваемому ресурсу.
- 404 Not Found: Запрос не был выполнен, поскольку сервер не смог найти запрашиваемый ресурс.

19. Назовите как минимум 3 кода ошибки сервера.

Ответ: сервер возвращает следующие коды при ошибке на сервере:

- 500 Internal Server Error: Запрос не был выполнен из-за непредвиденной проблемы с сервером.
- 502 Bad Gateway: Запрос не был выполнен из-за некорректного ответа от вышестоящего сервера.
- 503 Service Unavailable: Сервер не смог обработать запрос из-за технического обслуживания, перегрузки или других временных помех.
- Обратно

В Django, permissions (права доступа) представляют собой механизм, который определяет, какие пользователи или группы пользователей имеют право выполнять определенные действия в приложениях. Они используются для контроля доступа к различным частям приложения и обеспечения безопасности данных.

Django предоставляет встроенную систему авторизации и управления правами доступа, которая включает встроенные права доступа: добавление, изменение, удаление и просмотр объектов в моделях, а также возможность создания пользовательских прав доступа.

Система permissions Django состоит из нескольких основных элементов:

1. Встроенные права доступа (Built-in permissions): Django предоставляет четыре встроенных права доступа: "add", "change", "delete" и "view". Эти права могут быть назначены для моделей Django и определяют, какие действия могут выполнять пользователи для каждого объекта модели.

2. Декораторы для управления правами доступа:

Django предоставляет декораторы, такие как @login_required и permission_required, которые могут быть использованы для ограничения доступа к представлениям на основе прав доступа.

3. Пользовательские права доступа (Custom permissions):

Пользователи могут создавать свои собственные права доступа, определяя их через классы или функции, используя систему прав доступа Django.

Пример использования:

```
from django.db import models

from django.contrib.auth.models import User

from django.db.models import Q

from django.contrib.auth.models import Permission


class YourModel(models.Model):

        # Your model fields


class YourView(models.Model):

        class Meta:

        permissions = [

        ("can_view_yourview", "Can view YourView"),

        ("can_change_yourview", "Can change YourView"),

        ]


# Assign permission to user

user = User.objects.get(username='yourusername')

permission = Permission.objects.get(codename='can_view_yourview')

user.user_permissions.add(permission)
```

```
# Check permission
if user.has_perm('yourapp.can_view_yourview'):
    # User has the permission
else:
    # User does not have the permission
```

Декораторы @loginrequired и @permissionrequired могут быть использованы для контроля доступа к представлениям. Например:

```
from django.contrib.auth.decorators import login_required, permission_required

@login_required
def your_view(request):
    # Only logged in users have access to this view
    pass

@permission_required('yourapp.can_view_yourview')
def your_view(request):
    # Only users with 'can_view_yourview' permission have access to this view
    pass
```

Использование прав доступа в Django позволяет создавать безопасные и управляемые веб-приложения, где различные пользователи могут иметь разные уровни доступа к различным частям приложения.

Обратно

Делаю как в других фреймворка. Логика в файлах service, в моделях только модель, во вьюхах дергаю только сервисы или привязку сервиса к шаблону.

1. **Представления.** Представления Django — обычное место для включения бизнес-логики. Они обрабатывают входящие запросы, обрабатывают данные и возвращают ответы. Простая логика может быть встроена непосредственно в представления.

2. **Модели.** Бизнес-логика, связанная с манипулированием данными и взаимодействием, может храниться в моделях Django. Методы внутри моделей могут инкапсулировать логику, связанную с проверкой данных, вычислениями и другими операциями.

3. **Модуль Utils.** Создайте в приложении отдельный файл utils.py для хранения повторно используемых функций и классов, которые не вписываются непосредственно в модели или представления. Это сохраняет ваш код организованным и способствует повторному использованию кода.

4. **Сервисы.** Объедините сложную логику в классы или модули сервисов. Сервисы помогают поддерживать разделение задач и обеспечивают возможность повторного использования и легкость тестирования бизнес-логики.

5. **Пользовательские модули.** В зависимости от сложности и размера вашего проекта вы можете создавать собственные модули для размещения определенных типов логики, таких как аутентификация, разрешения или взаимодействие с внешними API. Такой подход помогает поддерживать чистоту и удобство обслуживания вашей кодовой базы.

Обратно

Временная сложность алгоритма - это мера количества времени, необходимого для выполнения алгоритма, как функция от размера входных данных. В алгоритмической сложности обычно рассматривается наихудший сценарий исполнения алгоритма.

Выраженная обычно в форме "O-большое" (Big O notation) временная сложность алгоритма может быть использована для оценки его производительности. Например, алгоритм с временной сложностью O(n) означает, что время выполнения алгоритма линейно зависит от размера входных данных, алгоритм с временной сложностью O(n^2) означает, что время выполнения алгоритма зависит от квадрата размера входных данных, и так далее.

Временная сложность алгоритма не предсказывает точное время выполнения, а скорее оценивает, как алгоритм реагирует на увеличение объема ввода. Она помогает определить, как быстро будет расти время выполнения алгоритма при увеличении размера входных данных.

Примеры временной сложности:

- O(1) - константная сложность, время выполнения не зависит от размера входных данных.

- O(log n) - логарифмическая сложность, время выполнения увеличивается логарифмически от размера входных данных.

- O(n) - линейная сложность, время выполнения линейно зависит от размера входных данных.

- O(n^2), O(n^3) и др. - квадратичная, кубическая сложность и т.д., время выполнения увеличивается квадратично, кубически и т.д. от размера входных данных.

Анализ временной сложности алгоритма имеет важное значение для понимания его производительности и для выбора наиболее эффективных алгоритмов при работе с большими наборами данных.

Обратно

Если вас попросят спроектировать ленту новостей, вам следует прояснить требования, которые к ней предъявляются. Ваш разговор с интервьюером может выглядеть так:

Кандидат: «Это мобильное или веб-приложение? Или и то и другое?»

Интервьюер: «И то и другое».

Кандидат: «Какие самые важные возможности должны быть у этого продукта?»

Интервьюер: «Возможность публиковать статьи и видеть новостные ленты своих друзей».

Кандидат: «Новостная лента сортируется хронологически или в каком-то особом порядке? Особый порядок означает, что каждой статье назначается определенный вес. Например, статьи ваших близких друзей важнее статей, опубликованных в группе».

Интервьюер: «Чтобы не усложнять, предположим, что лента сортируется в обратном хронологическом порядке».

Кандидат: «Сколько друзей может быть у пользователя?»

Интервьюер: «5000».

Кандидат: «Какой объем трафика?»

Интервьюер: «10 миллионов активных пользователей в день (DAU)».

Кандидат: «Может ли лента содержать изображения и видеофайлы наряду с текстом?»

Интервьюер: «В ленте могут быть медиафайлы, включая изображения и видео».

Выше приведены некоторые из вопросов, которые вы можете задать своему интервьюеру. Важно определиться с требованиями и прояснить неоднозначные моменты.

Шаг 2: предложить общее решение и получить согласие

На этом этапе мы пытаемся выработать общее решение и согласовать его с интервьюером. Очень желательно в ходе этого процесса наладить совместную работу.

— Предложите начальный план архитектуры. Поинтересуйтесь мнением интервьюера. Относитесь к нему как к члену своей команды, с которым вы вместе работаете. Хорошие интервьюеры зачастую любят поговорить и поучаствовать в решении задачи.

— Нарисуйте на доске или бумаге блок-схемы с ключевыми компонентами, такими как клиенты (мобильные/браузерные), API-интерфейсы, веб-серверы, хранилища данных, кэши, CDN, очереди сообщений и т.д.

— Выполните приблизительные расчеты, чтобы понять, соответствует ли ваше решение масштабу задачи. Рассуждайте вслух. Прежде чем что-то считать, пообщайтесь с интервьюером.

По возможности пройдитесь по нескольким конкретным сценариям использования. Это поможет вам сформировать общую архитектуру и, скорее всего, обнаружить крайние случаи, о которых вы еще не думали.

Следует ли на этом этапе обозначать конечные точки API-интерфейса и схему базы данных? Это зависит от задачи. Если вас просят спроектировать что-то масштабное, такое как поиск Google, то лучше не углубляться настолько сильно. Если же речь идет о серверной части многопользовательской игры в покер, эти аспекты вполне можно указать. Общайтесь с интервьюером.

Пример

Давайте посмотрим, как происходит разработка общей архитектуры на примере новостной ленты. Вам не нужно понимать, как на самом деле работает эта система. Все детали будут описаны в главе 11.

На высоком уровне архитектура делится на два потока: публикацию статей и составление новостной ленты.

— Посты в ленте. Когда пользователь публикует пост, соответствующая информация записывается в кэш или базу данных, а сам пост появляется в ленте новостей его друзей.

— Составление ленты новостей. Новостная лента формируется путем группировки постов ваших друзей в обратном хронологическом порядке.

На рис. 3.1 и 3.2 показан общий принцип публикации постов и, соответственно, составления новостной ленты.

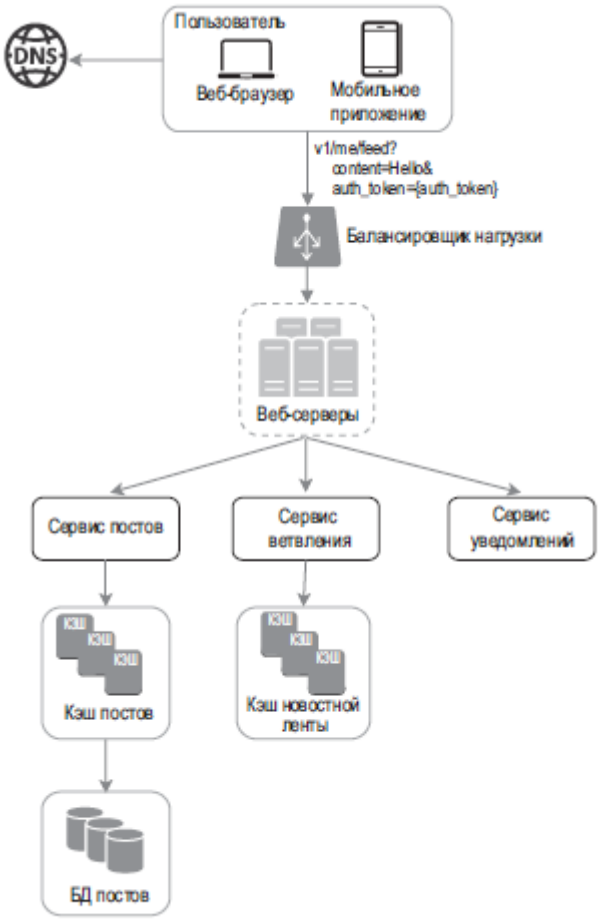

DNS
Пользователь
Веб-браузер
Мобильное приложение
v1/me/feed?
content=Hello&
auth_token={auth_token}
Балансировщик нагрузки
Веб-серверы
Сервис постов
Сервис ветвления
Сервис уведомлений
КЭШ
Кэш постов
Кэш новостной ленты
БД постов

Рис. 3.1

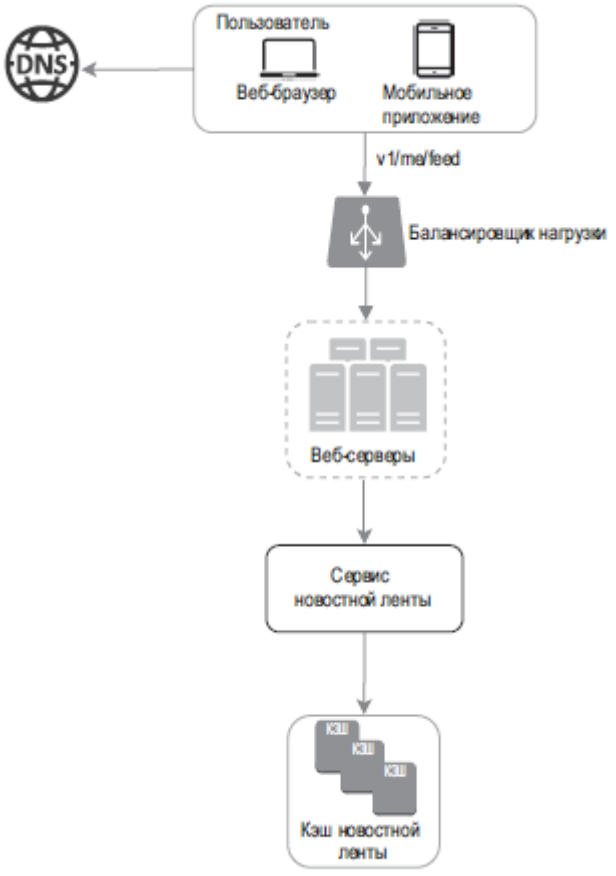

DNS
Пользователь
Веб-браузер
Мобильное приложение
v1/me/feed
Балансировщик нагрузки
Веб-серверы
Сервис новостной ленты
КЭШ
Кэш новостной ленты

Рис. 3.2

Шаг 3: глубокое погружение в проектирование

На этом этапе вы и интервьюер достигли следующих целей:

— согласовали общие требования и будущий масштаб;

— начертили примерную схему архитектуры;

— узнали мнение интервьюера о вашем общем решении;

— получили какое-то базовое представление о том, на каких областях нужно сосредоточиться при подробном проектировании (исходя из того, что ответил интервьюер).

Обратно
