

# Кеширование и бизнес-логика.

## Шпаргалка

### Горизонтальная и вертикальная оптимизация

Оптимизация веб-приложений проводится двумя основными способами:

#### 1. Горизонтальная оптимизация

- Масштабирование приложения за счет добавления новых серверов.
- Распределение нагрузки между несколькими серверами.

#### 2. Вертикальная оптимизация

- Улучшение производительности на уровне одного сервера.
- Использование эффективных алгоритмов и структур данных.
- Оптимизация взаимодействия с базой данных и минимизация задержек.

### Кеширование

**Кеширование** — это процесс сохранения копий данных или результатов вычислений в специальном месте (кеше) для ускорения последующего доступа к ним.

**Кеш** — это высокоскоростное место хранения данных, которое может находиться в оперативной памяти, на диске или на удаленном сервере.

### Примеры использования кеширования в веб-приложениях

- **Кеширование страниц** — полное сохранение HTML -кода страницы для быстрого ответа на повторные запросы.
- **Кеширование фрагментов страниц** — сохранение отдельных блоков страницы, таких как меню или виджеты, которые редко изменяются.
- **Кеширование запросов к базе данных** — сохранение результатов частых запросов к базе данных для уменьшения числа обращений к ней.

# Брокер для кеширования Redis

**Брокер кеширования** — это программное обеспечение, которое управляет сохранением и извлечением данных из кеша.

**Redis** (Remote Dictionary Server) — это высокопроизводительная нереляционная база данных, работающая в оперативной памяти и поддерживающая различные структуры данных, такие как строки, хеши, списки, множества и отсортированные множества.

## Подключение Redis к проекту Django

1. Установка библиотеки `redis`:

```
pip install redis
```

2. Настройка кеширования в файле `settings.py`:

```
# settings.py

CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.redis.RedisCache',
        'LOCATION': 'redis://127.0.0.1:6379/1',
    }
}
```

3. Использование кеша в проекте:

```
from django.core.cache import cache
from django.http import HttpResponse

def my_view(request):
    # Попытка получить данные из кеша
    data = cache.get('my_key')

    # Если данные не найдены в кеше, выполняем вычисления и сохраняем
    # результат в кеш
    if not data:
        data = 'some expensive computation'
        cache.set('my_key', data, 60 * 15) # Кешируем данные на 15 минут

    # Возвращаем ответ с данными
    return HttpResponse(data)
```

## Основные команды Redis и просмотр результатов кеширования

- **SELECT** — выбрать базу данных внутри Redis.

```
SELECT 1
```

- **SET** — установить значение для ключа.

```
SET my_key "some value"
```

- **GET** — получить значение по ключу.

```
GET my_key
```

- **DEL** — удалить ключ.

```
DEL my_key
```

- **KEYS** — получить все ключи, соответствующие шаблону.

```
KEYS *
```

- **EXPIRE** — установить время жизни ключа (в секундах).

```
EXPIRE my_key 900 # 900 секунд = 15 минут
```

- **TTL** — узнать оставшееся время жизни ключа.

```
TTL my_key
```

## Виды кеширования

**Серверное кеширование** — это процесс сохранения данных на стороне сервера; позволяет сократить время выполнения запросов и уменьшить нагрузку на базу данных и другие ресурсы.

**Кеширование страниц** — это метод кеширования, при котором сохраняется целая страница. Это позволяет существенно уменьшить время загрузки страниц для пользователей, поскольку готовые HTML-страницы могут быть быстро отданы сервером без повторной генерации.

```

from django.views.decorators.cache import cache_page
from django.utils.decorators import method_decorator
from django.views.generic import TemplateView

@method_decorator(cache_page(60 * 15), name='dispatch')
class MyView(TemplateView):
    template_name = 'my_template.html'

```

**Низкоуровневое кеширование** — это способ кеширования данных на уровне отдельных запросов или операций. Это позволяет существенно ускорить выполнение часто повторяющихся запросов, уменьшить нагрузку на базу данных и серверные ресурсы.

```

from django.core.cache import cache
from django.views.generic import ListView
from .models import MyModel

class CachedListView(ListView):
    model = MyModel
    template_name = 'my_template.html'

    def get_queryset(self):
        queryset = cache.get('my_queryset')
        if not queryset:
            queryset = super().get_queryset()
            cache.set('my_queryset', queryset, 60 * 15) # Кешируем данные на
15 минут
        return queryset

```

**Клиентское кеширование** — это процесс сохранения данных на стороне клиента (в браузере пользователя). Это позволяет ускорить загрузку страниц и снизить нагрузку на сервер за счет использования ранее загруженных данных.

```

<link rel="stylesheet" href="/static/css/styles.css" />
<script src="/static/js/scripts.js"></script>

```

## Сервисные прослойки и бизнес-логика

**Бизнес-логика** — это совокупность правил и операций, которые определяют, как обрабатываются и изменяются данные в вашем приложении в соответствии с требованиями бизнеса.

**Сервисный слой** — это компонент архитектуры приложения, который отвечает за выполнение бизнес-логики. Он отделяет логику обработки данных от пользовательского интерфейса и взаимодействия с базой данных, обеспечивая независимость и легкость управления кодом.

### Что можно выносить в сервисный слой

1. Вычисления и преобразования данных.
2. Правила валидации.
3. Бизнес-операции.
4. Работу с внешними API.
5. Сложные запросы к базе данных.
6. Кеширование и оптимизацию.

### Пример выноса бизнес-логики в сервисный слой

- Модели `Student` и `Grade`:

```
from django.db import models

class Student(models.Model):
    first_name = models.CharField(max_length=100)
    last_name = models.CharField(max_length=100)
    birth_date = models.DateField()

    def __str__(self):
        return f"{self.first_name} {self.last_name}"

class Grade(models.Model):
    student = models.ForeignKey(Student, on_delete=models.CASCADE)
    subject = models.CharField(max_length=100)
    score = models.FloatField()

    def __str__(self):
        return f"{self.subject}: {self.score}"
```

- Сервисный слой:

```
# services.py
from .models import Student, Grade

class StudentService:

    @staticmethod
    def get_full_name(student_id):
        # Получаем полное имя студента по его ID
        student = Student.objects.get(id=student_id)
        return f"{student.first_name} {student.last_name}"

    @staticmethod
    def calculate_average_grade(student_id):
        # Получаем все оценки студента
        grades = Grade.objects.filter(student_id=student_id)
        # Если оценок нет, возвращаем None
        if not grades.exists():
            return None
        # Вычисляем сумму всех оценок
        total_score = sum(grade.score for grade in grades)
        # Вычисляем средний балл
        average_score = total_score / grades.count()
        return average_score

    @staticmethod
    def has_passed(student_id, passing_score=60):
        # Вычисляем средний балл студента
        average_grade = StudentService.calculate_average_grade(student_id)
        # Если средний балл не вычислен (нет оценок), возвращаем False
        if average_grade is None:
            return False
        # Проверяем, сдал ли студент предмет (средний балл >= проходному
        баллу)
        return average_grade >= passing_score
```

- Использование сервисного слоя в представлениях:

```
# views.py
from django.views.generic import DetailView
from .models import Student
from .services import StudentService

class StudentDetailView(DetailView):
    model = Student
    template_name = 'students/student_detail.html'
    context_object_name = 'student'

    def get_context_data(self, **kwargs):
        # Получаем стандартный контекст данных из родительского класса
        context = super().get_context_data(**kwargs)
        # Получаем ID студента из объекта
        student_id = self.object.id
        # Добавляем в контекст полное имя, средний балл и статус сдачи
        # предмета
        context['full_name'] = StudentService.get_full_name(student_id)
        context['average_grade'] = StudentService.calculate_average_grade(
            student_id)
        context['has_passed'] = StudentService.has_passed(student_id)
        return context
```

- Шаблон для отображения деталей студента:

```
<!-- student_detail.html -->
{% extends 'base.html' %}

{% block title %}Детали студента{% endblock %}

{% block content %}
<h2>{{ full_name }}</h2>
<p>Средний балл: {{ average_grade }}</p>
<p>Сдал: {{ has_passed|yesno:"Да,Нет" }}</p>
<a href="{% url 'student_list' %}" class="btn btn-primary">Назад к списку
студентов</a>
{% endblock %}
```