# Лекция 11
# Авторизация. Сессии. Куки

Разработка интернет приложений

Канев Антон Игоревич

# Аутентификация

**Аутентифика́ция** (*authentication*) — процедура проверки подлинности, например:

- проверка подлинности пользователя путём сравнения введённого им пароля (для указанного логина) с паролем, сохранённым в базе данных пользовательских логинов;
- подтверждение подлинности электронного письма путём проверки цифровой подписи письма по открытому ключу отправителя;
- проверка контрольной суммы файла на соответствие сумме, заявленной автором этого файла.

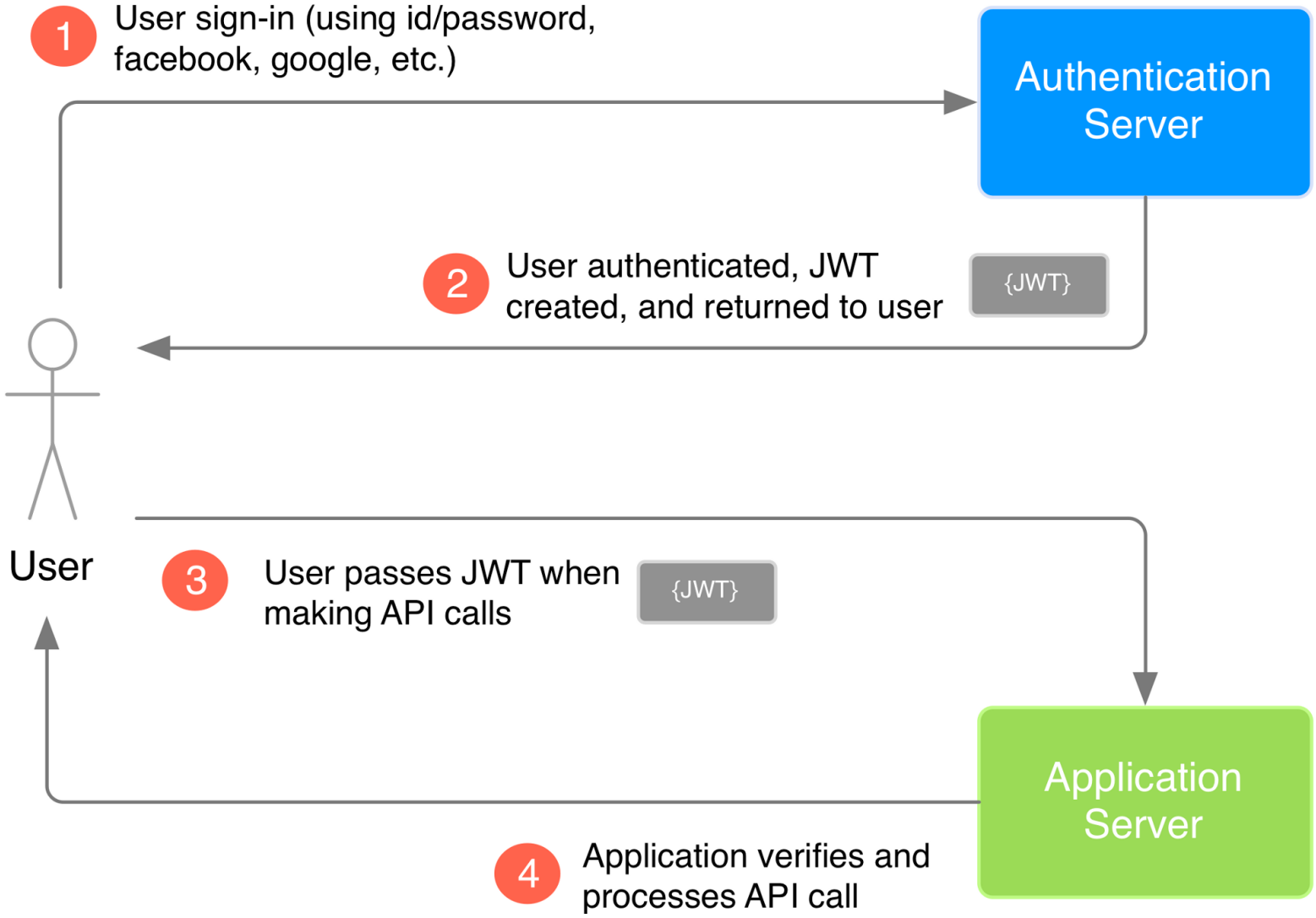
# Авторизация

- **Авториза́ция** (*authorization* «разрешение; уполномочивание») — предоставление определённому лицу или группе лиц прав на выполнение определённых действий; а также процесс проверки (подтверждения) данных прав при попытке выполнения этих действий.

- Авторизация производит контроль доступа к различным ресурсам системы в процессе работы легальных пользователей после успешного прохождения ими аутентификации.

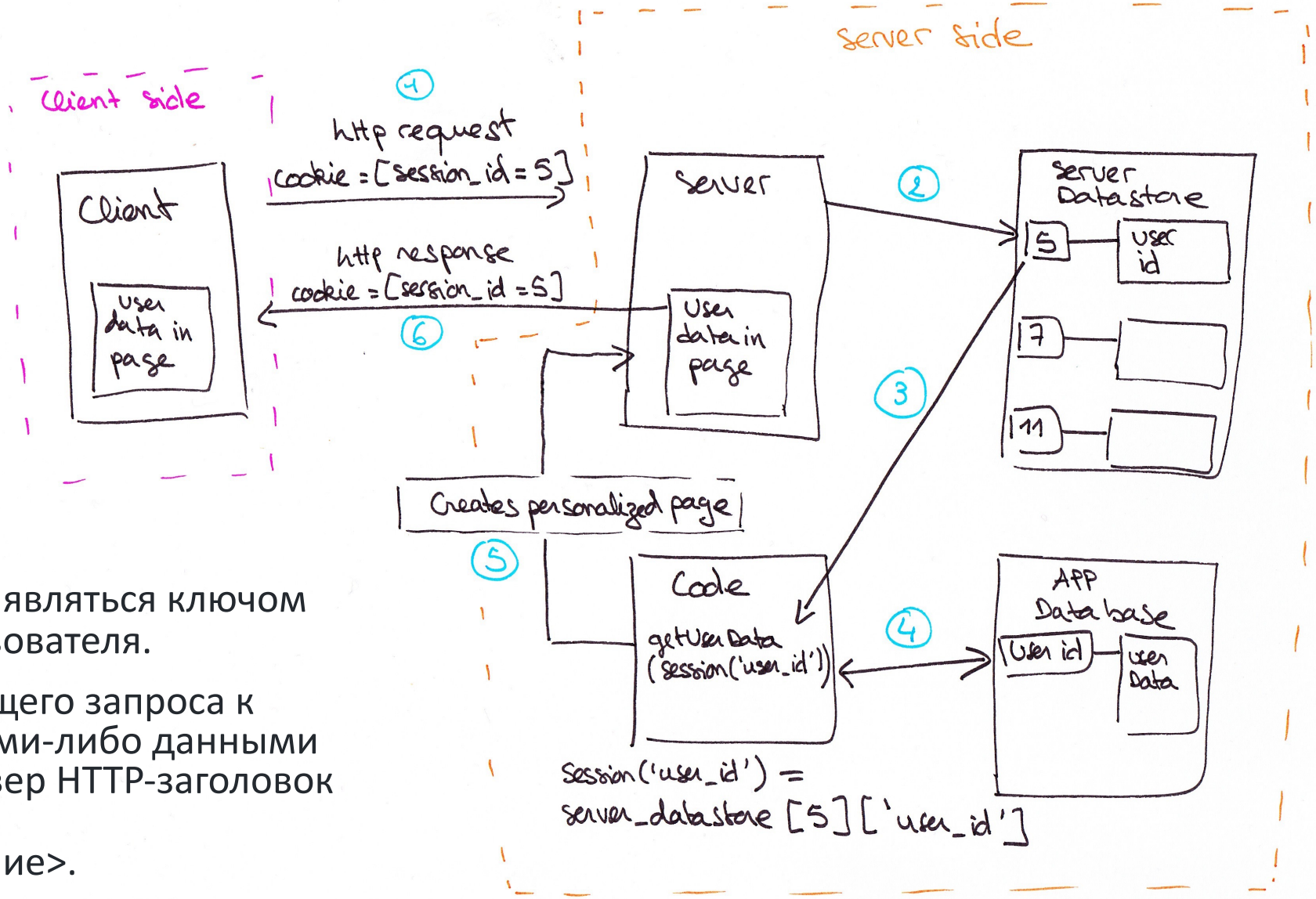
# JWT

- **JSON Web Token**
- Как правило, используется для передачи данных для аутентификации в клиент-серверных приложениях.
- Токены создаются сервером, подписываются секретным ключом и передаются клиенту, который в дальнейшем использует данный токен для подтверждения подлинности аккаунта.



# Сессии

- При авторизации на сайте сервер отсылает в ответ HTTP-заголовок Set-Cookie, чтобы сохранить куки в браузере с уникальным идентификатором сессии («session identifier»).
- Это идентификатор будет являться ключом уникальным сессии пользователя.
- Во время любого следующего запроса к этому же серверу за какими-либо данными браузер посылает на сервер HTTP-заголовок Cookie, в которым в формате <ключ>=<значение>.
- Таким образом, сервер понимает, кто сделал запрос.

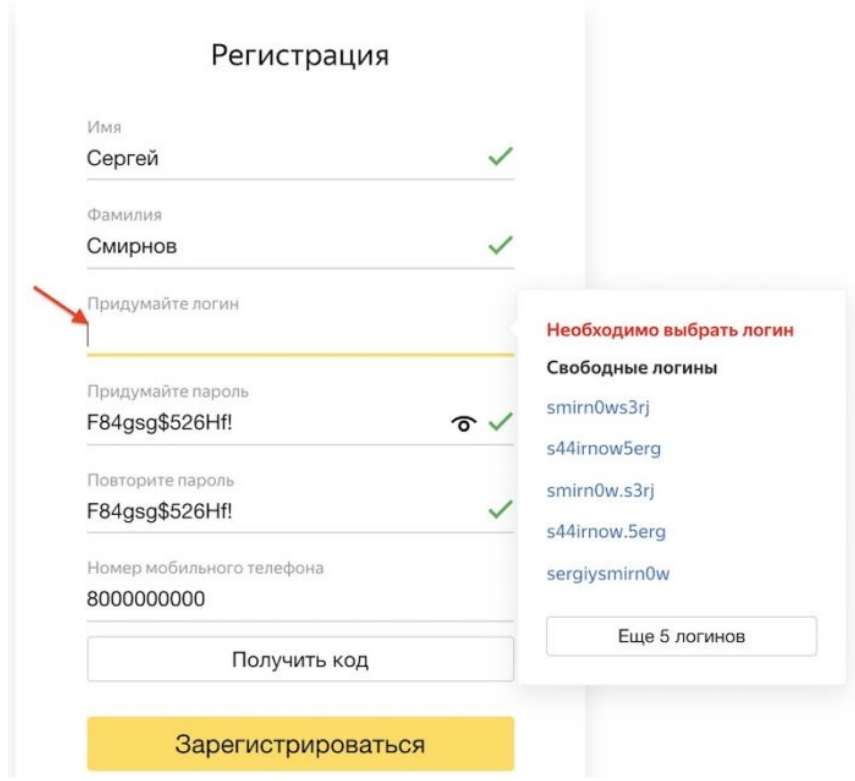


# Куки

- **Ку́ки** (*cookie*, букв. — «печенье») — небольшой фрагмент данных, отправленный веб-сервером и хранимый на компьютере пользователя.
- Веб-клиент (обычно веб-браузер) всякий раз при попытке открыть страницу соответствующего сайта пересылает этот фрагмент данных веб-серверу в составе HTTP-запроса.
- Применяется для сохранения данных на стороне пользователя

# Пользователи

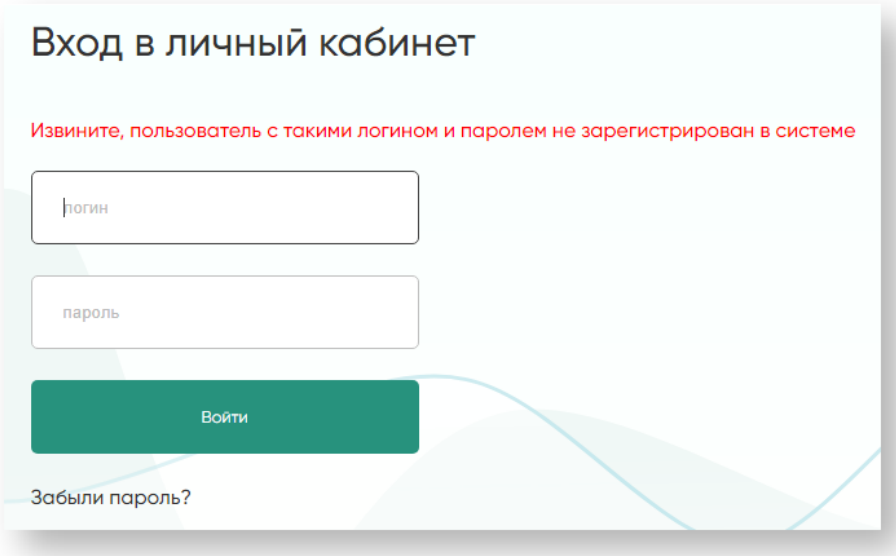
- Концептуально сущность пользователя должна содержать его личные данные, такие данные:
- номер телефона
- почта
- имя
- никнейм
- и тд...




```python
from django.contrib.auth import models as user_models
from django.contrib.auth.models import PermissionsMixin

class User(user_models.AbstractBaseUser, PermissionsMixin):
    username = models.CharField(max_length=150, unique=True)
    ...
```

# DRF аутентификация

- Создадим view для авторизации пользователей
- Чтобы зарегистрировать пользователя в системе используйте login(). Он принимает объект HttpRequest и объект User.
- login() сохраняет идентификатор пользователя в сессии, используя фреймворк сессий Django.

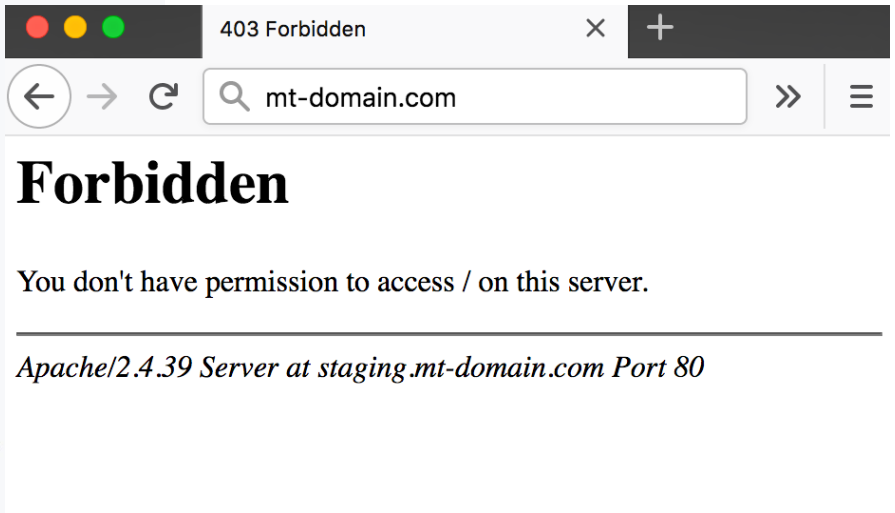



```python
from django.contrib.auth import authenticate, login
from django.http import HttpResponse

def auth_view(request):
    username = request.POST["username"] # допустим передали username и password
    password = request.POST["password"]
    user = authenticate(request, username=username, password=password)
    if user is not None:
        login(request, user)
        return HttpResponse("{'status': 'ok'}")
    else:
        return HttpResponse("{'status': 'error', 'error': 'login failed'}")
```

# Ограничения на бекенде

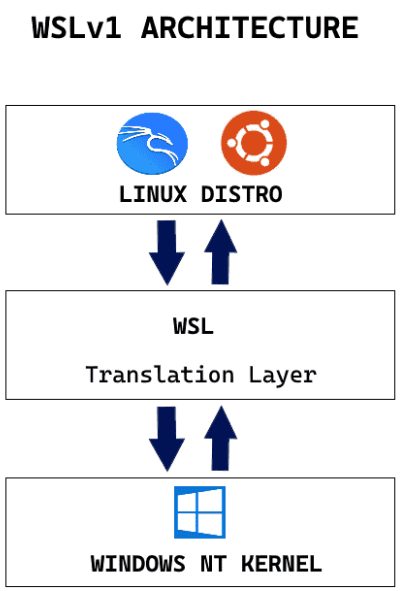
- Чтобы ограничить неавторизованным пользователем доступ к контенту, создадим view и добавим туда authentication_classes и permission_classes

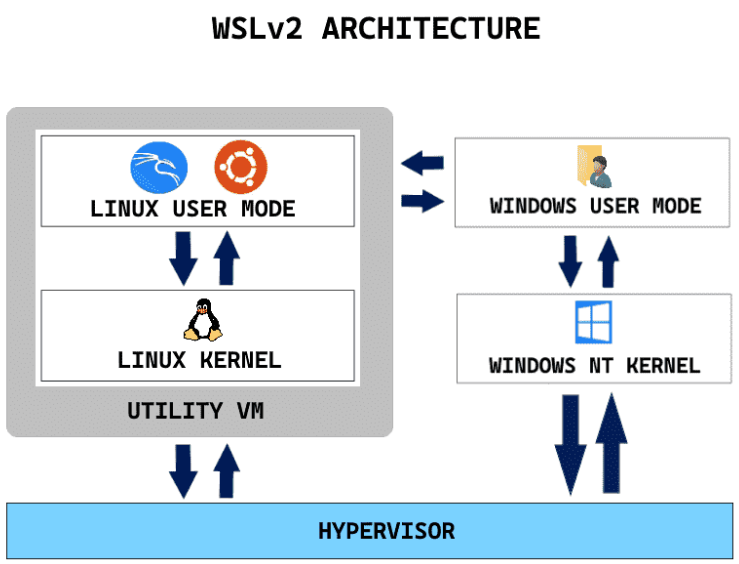
```python
from rest_framework.authentication import SessionAuthentication, BasicAuthentication
from rest_framework.permissions import IsAuthenticated
from rest_framework.response import Response
from rest_framework.views import APIView

class ExampleView(APIView):
    authentication_classes = [SessionAuthentication, BasicAuthentication]
    permission_classes = [IsAuthenticated]

    def get(self, request, format=None):
        content = {
            'user': str(request.user),  # `django.contrib.auth.User` instance.
            'auth': str(request.auth),  # None
        }
        return Response(content)
```



# WSL. Linux

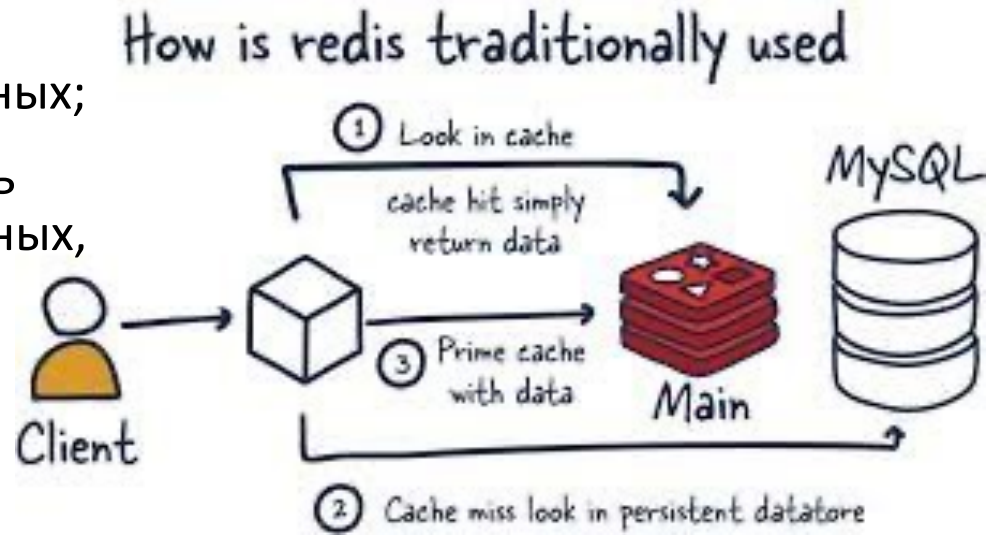
- **Windows Subsystem for Linux** (WSL) — слой совместимости для запуска Linux-приложений (двоичных исполняемых файлов в формате ELF) в ОС Windows



# Redis. Применение

- для хранения пользовательских сессий (HTML-фрагменты веб-страниц или товары корзины интернет-магазина);
- для хранения промежуточных данных (поток сообщений на стене, голосовалки, таблицы результатов);
- как брокер сообщений (стратегия «издатель-подписчик» позволяет создавать новостные ленты, групповые чаты);
- как СУБД для небольших приложений, блогов;
- для кэширования данных из основного хранилища, что значительно снижает нагрузку на реляционную базу данных;
- для хранения «быстрых» данных — когда важны скорость и критичны задержки передачи (аналитика и анализ данных, финансовые и торговые сервисы).



# Redis. Пример

- HSET — сохраняет значение по ключу
- создали объект person1 с двумя полями (name и age) и соответствующими значениями.

```
127.0.0.1:6379> HSET person1 name "Aleksey"
(integer) 1
127.0.0.1:6379> HSET person1 age 25
(integer) 1
```

# Redis. Пример

- HGET — получение значения по ключу (для определённого поля)
- Получили значение поля name у ключа person1

```
127.0.0.1:6379> HGET person1 name
"Aleksey"
```

# Redis

- Установить

```
curl -fsSL https://packages.redis.io/gpg | sudo gpg --dearmor -o /usr/share/keyrings/redis-archive-keyring.gpg

echo "deb [signed-by=/usr/share/keyrings/redis-archive-keyring.gpg] https://packages.redis.io/deb $(lsb_release

sudo apt-get update
sudo apt-get install redis
```

- Запустить

```
sudo service redis-server start
```

- Использовать

```
redis-cli
127.0.0.1:6379> ping
PONG
```

# Redis с Django

- Зайдем в файл settings.py и пропишем туда сокет запущенной БД:

```
REDIS_HOST = 'localhost'
REDIS_PORT = 6379
```

- Далее создадим библиотечный инстанс нашего хранилища сессий в файле views.py:

```
from django.conf import settings
import redis

# Connect to our Redis instance
session_storage = redis.StrictRedis(host=settings.REDIS_HOST, port=settings.REDIS_PORT)
```

# Аутентификация с Redis

```python
from django.contrib.auth import authenticate, login
from django.http import HttpResponse
import uuid

def auth_view(request):
    username = request.POST["username"] # допустим передали username и password
    password = request.POST["password"]
    user = authenticate(request, username=username, password=password)
    if user is not None:
        random_key = uuid.uuid4()
        session_storage.set(random_key, username)

        response = HttpResponse("{'status': 'ok'}")
        response.set_cookie("session_id", random_key) # пусть ключем для куки будет session_id
        return response
    else:
        return HttpResponse("{'status': 'error', 'error': 'login failed'}")
```

# Авторизация Redis

Соответственно в методах, в которых нужно проверить имеет ли пользователя доступ к запрашиваемой информации, мы должны:

- взять из запроса куки (через ssid = request.COOKIES["session_id"])
- посмотреть есть ли в хранилище сессий такая запись, и достать идентификатор пользователя (session_storage.get(ssid))
- проверить, можно ли данному пользователю смотреть запрошенную информацию (зависит от бизнес-логики вашего проекта)

# Доработки фронтенда

- Добавить окна регистрации и авторизации
- Добавить логику проверки авторизации пользователя – после успешной авторизации меняется состояние приложения
- Авторизованный пользователь может разлогиниться
- Авторизованному пользователю доступен больший объем функционала в зависимости от его роли