

Лекция 5

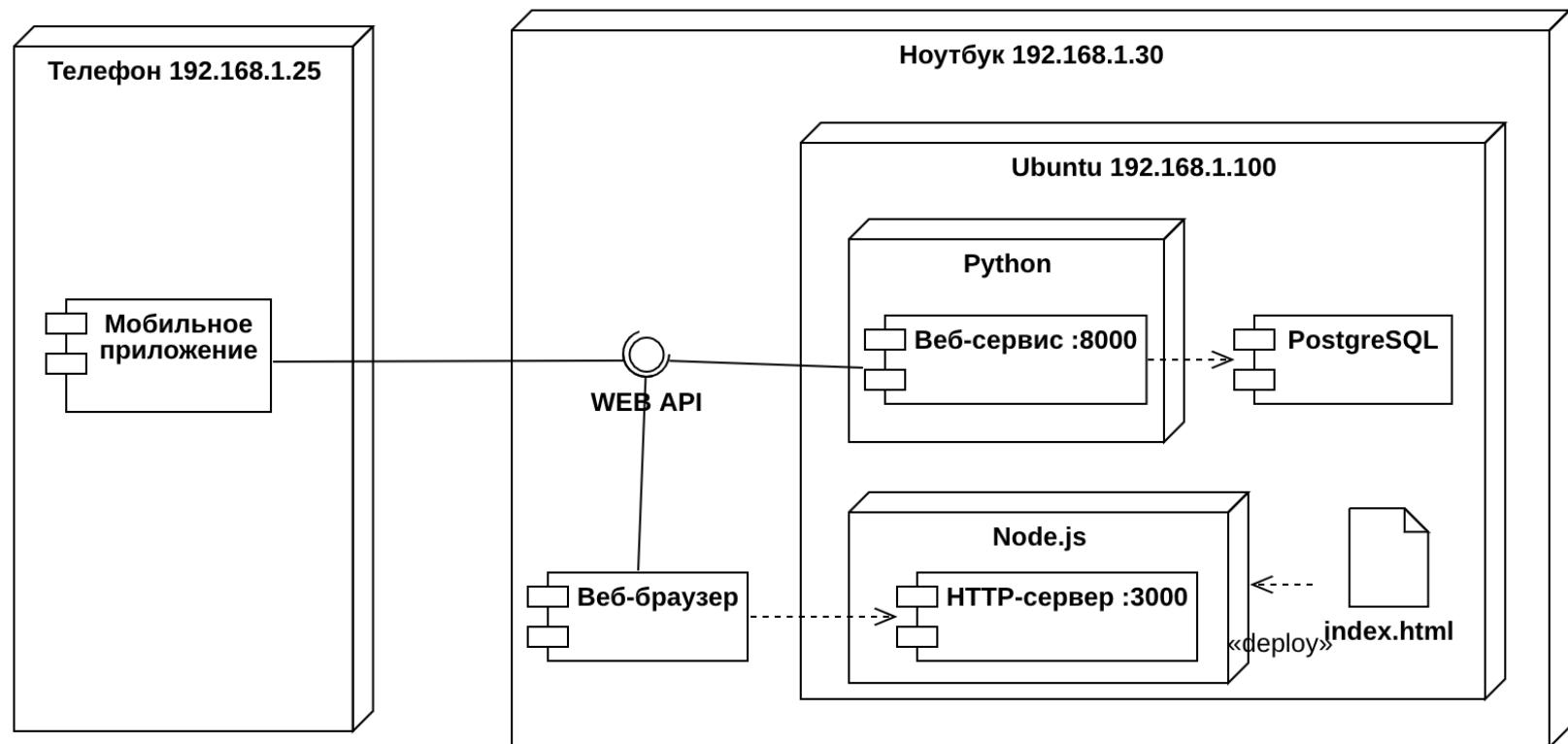
Веб-сервисы

Разработка интернет приложений

Канев Антон Игоревич

Трехзвенная архитектура. API

- Чтобы использовать веб-сервер как универсальный источник данных для других приложений, нам нужно сделать веб-сервис, который предоставит нам Web API
- Теперь данные мы будем получать не как HTML, а структурированно в формате JSON



Веб-служба

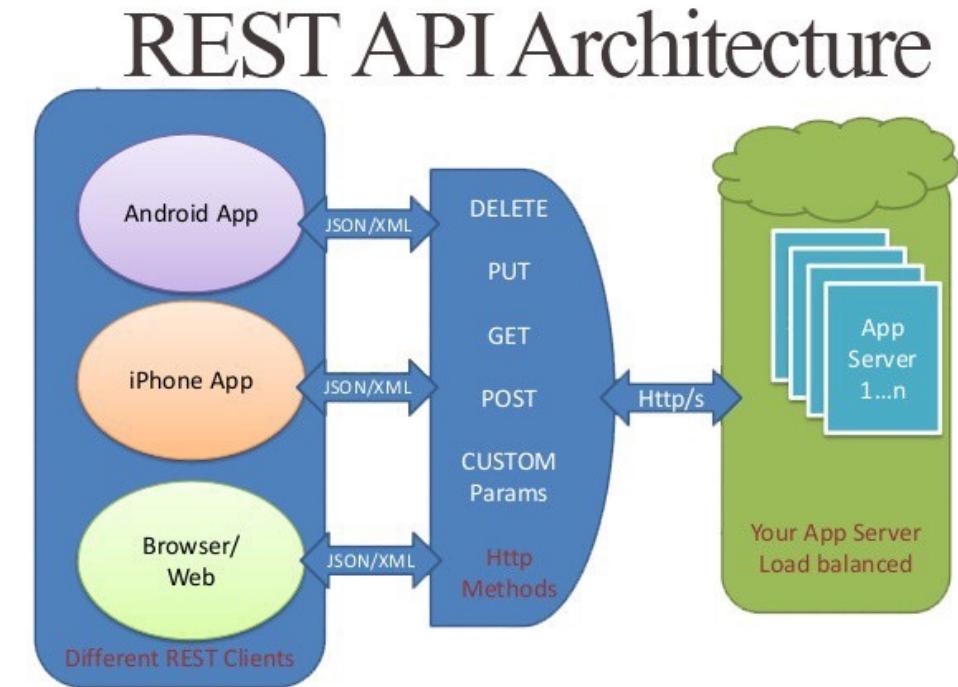
- Веб-служба, веб-сервис (web service) — идентифицируемая уникальным веб-адресом (URL-адресом) программная система со стандартизованными интерфейсами.
- Веб-службы могут взаимодействовать друг с другом и со сторонними приложениями посредством сообщений, основанных на определённых протоколах (SOAP, XML-RPC и т. д.) и соглашениях (REST). Веб-служба является единицей модульности при использовании сервис-ориентированной архитектуры приложения.

API

- API (Application Programming Interface) — описание способов взаимодействия одной компьютерной программы с другими.
- Обычно входит в описание какого-либо интернет-протокола, программного фреймворка или стандарта вызовов функций операционной системы.
- Часто реализуется отдельной программной библиотекой или сервисом операционной системы.
- Проще говоря, это набор компонентов, с помощью которых компьютерная программа (бот или же сайт) может использовать другую программу.

REST

- REST - Representational State Transfer. Набор правил того, как программисту организовать написание кода серверного приложения, чтобы все системы легко обменивались данными и приложение можно было масштабировать
- RESTful – веб-службы не нарушающие ограничений
- **Ограничения (знать!)**: клиент-сервер, отсутствие состояния, кэширование, единообразие интерфейса, слои, код по требованию



RPC

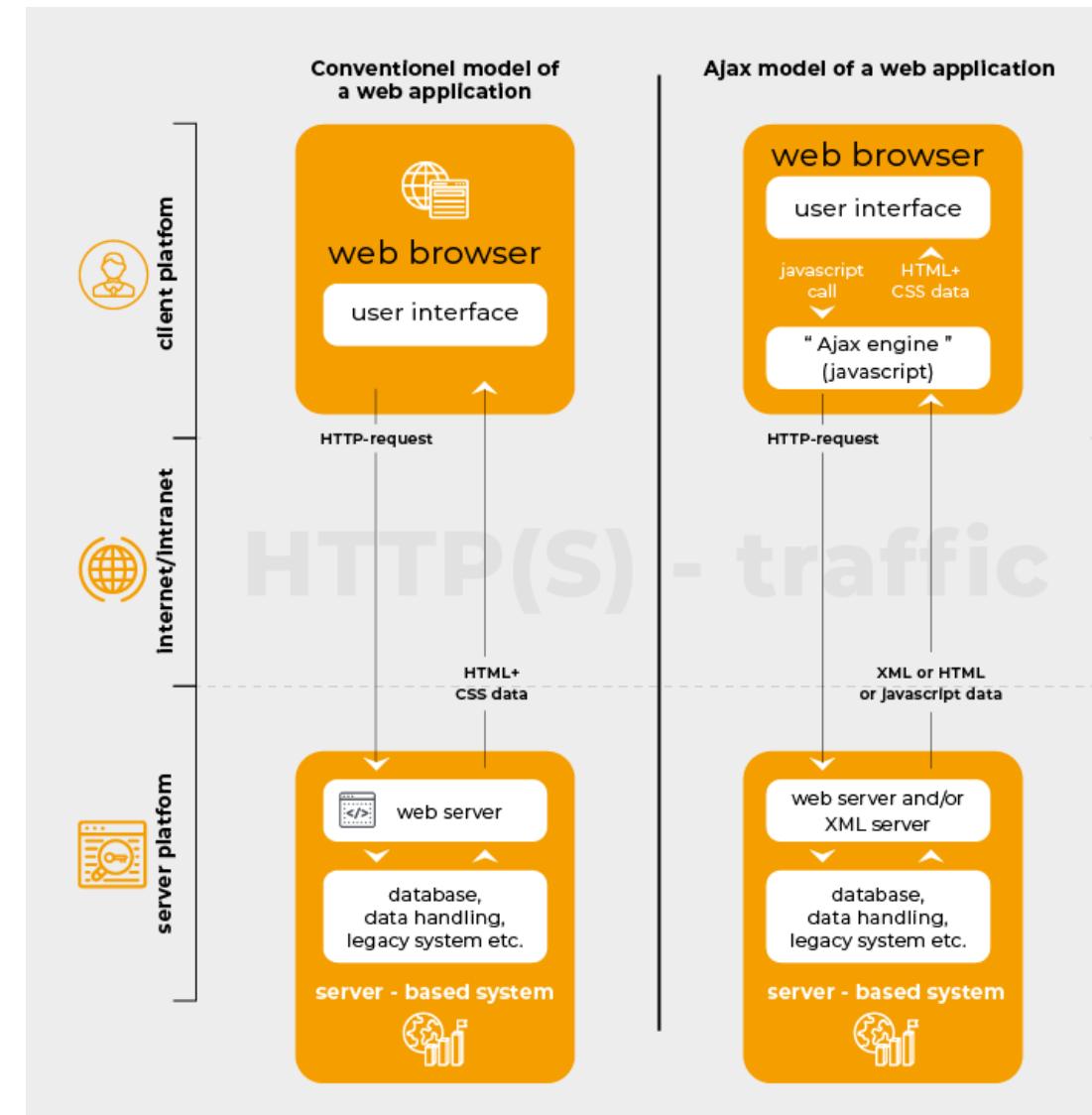
- RPC - remote procedure call. Удалённый вызов процедур
- Реализация RPC-технологии включает два компонента: сетевой протокол для обмена в режиме клиент-сервер и язык сериализации объектов (или структур для необъектных RPC)
- JSON-RPC
- XML RPC
- gRPC
- SOAP

gRPC

- **gRPC** (Remote Procedure Calls) — это система удалённого вызова процедур (RPC) с открытым исходным кодом, первоначально разработанная в Google
- В качестве транспорта используется HTTP/2, в качестве языка описания интерфейса — Protocol Buffers.
- gRPC предоставляет такие функции как аутентификация, двунаправленная потоковая передача и управление потоком, блокирующие или неблокирующие привязки, а также отмена и тайм-ауты.

AJAX

- **AJAX**, Ajax (Asynchronous Javascript and XML — «асинхронный JavaScript и XML») — подход к построению интерактивных пользовательских интерфейсов веб-приложений, заключающийся в «фоновом» обмене данными браузера с веб-сервером.
- В результате при обновлении данных веб-страница не перезагружается полностью, и веб-приложения становятся быстрее и удобнее.
- **JSON-RPC** (JavaScript Object Notation Remote Procedure Call — JSON-вызов удалённых процедур) — протокол удалённого вызова процедур, использующий JSON для кодирования сообщений.



XMLHttpRequest

- **XMLHttpRequest** (XMLHTTP, XHR) — API, доступный в скриптовых языках браузеров, таких как JavaScript.
- Использует запросы HTTP или HTTPS напрямую к веб-серверу и загружает данные ответа сервера напрямую в вызывающий скрипт.

- Информация может передаваться в любом текстовом формате, например, в XML, HTML или JSON. Позволяет осуществлять HTTP-запросы к серверу без перезагрузки страницы.

```
var http_request = new XMLHttpRequest();
http_request.onreadystatechange = function () {
    if (http_request.readyState !== 4)
        return;

    if (http_request.status !== 200)
        throw new Error('request was defeated');

    do_something_with_object(JSON.parse(http_request.responseText));
    http_request = null;
};
http_request.open("GET", url, true);
http_request.send(null);
```

JSON

- JSON (JavaScript Object Notation) — текстовый формат обмена данными, основанный на JavaScript.
- Заменил XML как формат данных для HTTP методов

```
<person>
  <firstName>Иван</firstName>
  <lastName>Иванов</lastName>
  <address>
    <streetAddress>Московское ш., 101, кв.101</streetAddress>
    <city>Ленинград</city>
    <postalCode>101101</postalCode>
  </address>
  <phoneNumbers>
    <phoneNumber>812 123-1234</phoneNumber>
    <phoneNumber>916 123-4567</phoneNumber>
  </phoneNumbers>
</person>
```

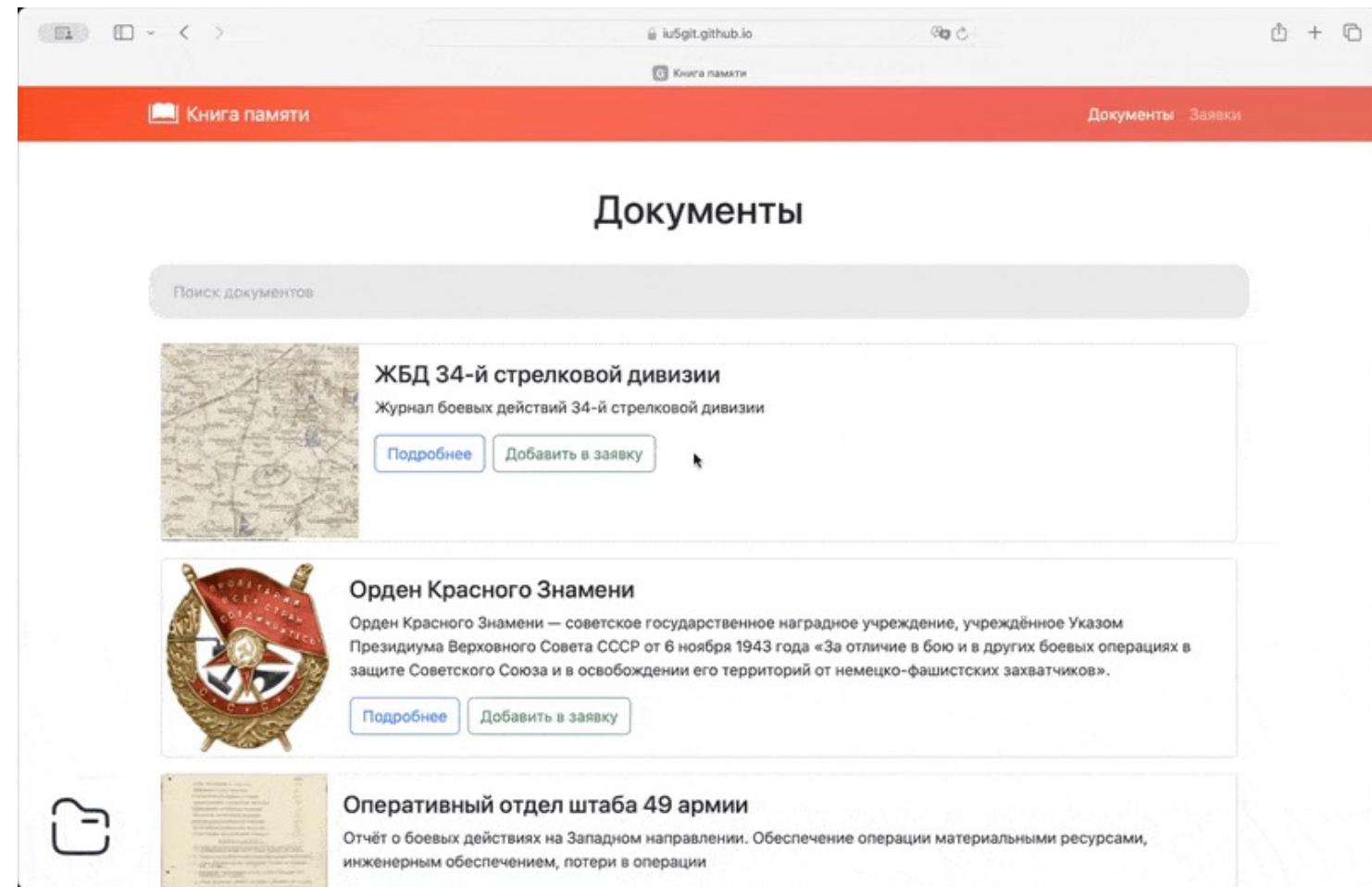
```
<person firstName="Иван" lastName="Иванов">
  <address streetAddress="Московское ш., 101, кв.101" city="Ленинград" postalCode="101101" />
  <phoneNumbers>
    <phoneNumber>812 123-1234</phoneNumber>
    <phoneNumber>916 123-4567</phoneNumber>
  </phoneNumbers>
</person>
```

```
{
  "firstName": "Иван",
  "lastName": "Иванов",
  "address": {
    "streetAddress": "Московское ш., 101, кв.101",
    "city": "Ленинград",
    "postalCode": 101101
  },
  "phoneNumbers": [
    "812 123-1234",
    "916 123-4567"
  ]
}
```

Итоговое приложение курса

Теперь нам нужно вспомнить итоговое приложение (в реальности Figma). В нем 12 страниц, но только 9 используют наши методы:

- Список услуг и Одна услуга
- Список заявок и Одна заявка
- Список услуг таблицей и Редактирование услуги
- Регистрация, авторизация, Личный кабинет
- Главная, 403 и 404



Методы по вашей теме

Услуги

- Список услуг
- Добавление услуги
- Получение услуги
- Редактирование услуги
- Удаление услуги
- Добавление услуги в последнюю заявку
- Добавление/замена изображения

Заявки

- Список заявок
- Получение заявки
- Редактирование заявки
- Изменение статуса создателем
- Изменение статуса модератором
- Удаление заявки

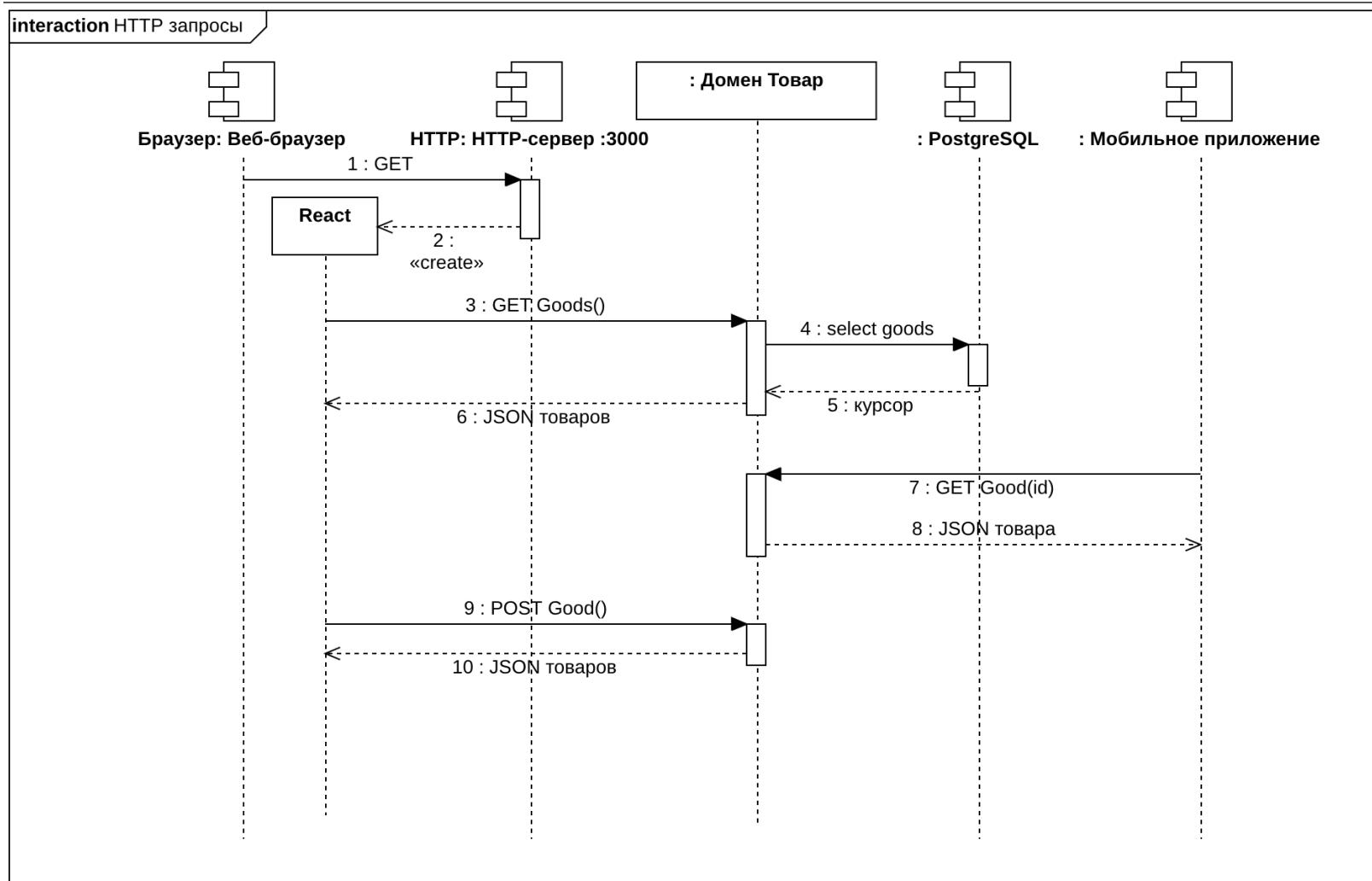
м-м через 2 id

- Удаление из м-м
- Изменение количества/порядка/значения в м-м

Пользователь

- Регистрация
- Обновление пользователя
- Аутентификация (4 лаб)
- Деавторизация (4 лаб)

AJAX запросы REST API + SPA

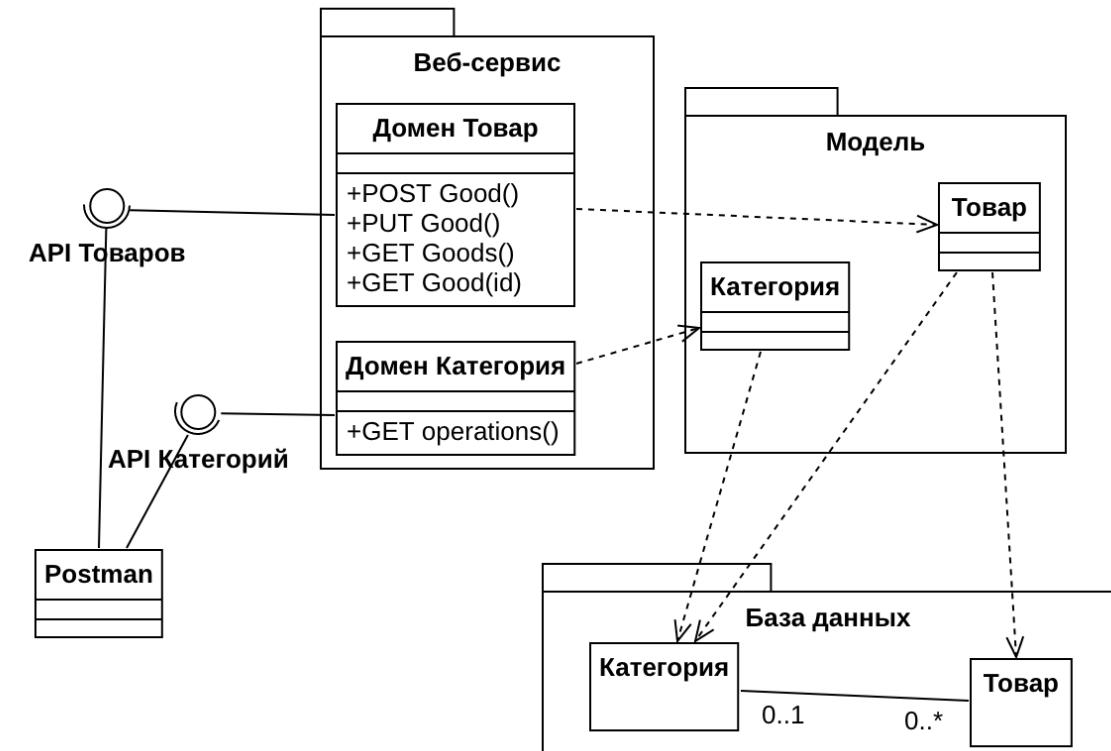


Django Rest Framework

- Django REST <https://www.django-rest-framework.org>
- <https://django.fun/ru/docs/django-rest-framework/3.12/>

Требуется

- Python (3.6, 3.7, 3.8, 3.9, 3.10)
- Django (2.2, 3.0, 3.1, 3.2, 4.0)
- pip install djangorestframework



Модель

```
from django.db import models

class Stock(models.Model):
    company_name = models.CharField(max_length=50, verbose_name="Название компании")
    price = models.DecimalField(max_digits=8, decimal_places=2, verbose_name="Цена акции")
    is_growing = models.BooleanField(verbose_name="Растет ли акция в цене?")
    date_modified = models.DateTimeField(auto_now=True, verbose_name="Когда последний раз обновлялось значени
```

- python manage.py makemigrations (создаем файлы миграций)
- python manage.py migrate (применяем файлы миграций к базе)

Сериализация пример

- **Сериализация** — процесс перевода структуры модели в JSON формат и обратно.
- Каждый раз мы должны получить данные из модели и передать их на клиент в виде JSON. Обратно данные мы получаем также в виде JSON и должны использовать методы модели для изменения БД. Все это делает сериализатор

```
from stocks.models import Stock
from rest_framework import serializers

class StockSerializer(serializers.ModelSerializer):
    class Meta:
        # Модель, которую мы сериализуем
        model = Stock
        # Поля, которые мы сериализуем
        fields = ["pk", "company_name", "price", "is_growing", "date_modified"]
```

Сериализация поля

- Нам необходимо определить поля, которые будет использовать сериализатор
- Все поля, часть или все кроме
- Автоматические поля – из модели
- Декларируемые – мы пропишем их сами в сериализаторе до class Meta

```
class WriterModelSerializer(serializers.ModelSerializer):  
  
    class Meta:  
        model = Writer  
        fields = '__all__'
```

```
class WriterModelSerializer(serializers.ModelSerializer):  
  
    class Meta:  
        model = Writer  
        fields = ['firstname', 'lastname']
```

```
class WriterModelSerializer(serializers.ModelSerializer):  
  
    class Meta:  
        model = Writer  
        exclude = ['firstname', 'lastname']
```

Сериализация вложенность

- Вложенные модели нужно передавать в том же запросе
- Поэтому сериализатор должен поддерживать вложенность
- А лучше передать сериализатор в качестве декларируемого поля

```
class TownSerializer(serializers.ModelSerializer):
    class Meta:
        model = Town
        fields = ['id', 'name', 'writers']

s = TownSerializer(instance=Town.objects.first())
print(s.data)
-----
{'id': 1, 'name': 'Вологда', 'writers': [6, 7]}
```

```
class TownSerializer(serializers.ModelSerializer):
    class Meta:
        model = Town
        depth = 1
        fields = ['id', 'name', 'writers']

    writers = WriterSerializer(many=True) # Добавили сериализер в качестве поля

class TownSerializer(serializers.ModelSerializer):
    class Meta:
        model = Town
        fields = ['id', 'name', 'writers']
```

API View

- В Django в нашем курсе используем только API View (классовый или функциональный)
- Этот подход похож на подход в Go
- Позволяет нам сделать конкретные бизнес методы, а не просто набор CRUD операций

```
@api_view(['Post'])
def post_list(request, format=None):
    """
Добавляет новую акцию
    """
    print('post')
    serializer = StockSerializer(data=request.data)
    if serializer.is_valid():
        serializer.save()
        return Response(serializer.data, status=status.HTTP_201_CREATED)
    return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)

@api_view(['Get'])
def get_detail(request, pk, format=None):
    stock = get_object_or_404(Stock, pk=pk)
    if request.method == 'GET':
        """
Возвращает информацию об акции
        """
        serializer = StockSerializer(stock)
        return Response(serializer.data)
```

URL API View

- В url указываем домен
(группа методов),
название метода и
идентификаторы

```
urlpatterns = [
    path('', include(router.urls)),
    path(r'stocks/', views.get_list, name='stocks-list'),
    path(r'stocks/post/', views.post_list, name='stocks-post'),
    path(r'stocks/<int:pk>/', views.get_detail, name='stocks-detail'),
    path(r'stocks/<int:pk>/put/', views.put_detail, name='stocks-put'),
    path(r'stocks/<int:pk>/delete/', views.delete_detail, name='stocks-delete'),
    path('api-auth/', include('rest_framework.urls', namespace='rest_framework')),

    path('admin/', admin.site.urls),
]
```

ViewSet – не используем

- В нашем курсе мы не используем `ViewSet`, но он похож на классовый API View

```
from rest_framework import viewsets
from stocks.serializers import StockSerializer
from stocks.models import Stock

class StockViewSet(viewsets.ModelViewSet):
    """
        API endpoint, который позволяет просматривать и редактировать акции компаний
    """
    # queryset всех пользователей для фильтрации по дате последнего изменения
    queryset = Stock.objects.all().order_by('date_modified')
    serializer_class = StockSerializer # Сериализатор для модели
```

ViewSet – не используем

```
class UserViewSet(viewsets.ViewSet):
    """
    Example empty viewset demonstrating the standard
    actions that will be handled by a router class.

    If you're using format suffixes, make sure to also include
    the `format=None` keyword argument for each action.
    """

    def list(self, request):
        pass

    def create(self, request):
        pass

    def retrieve(self, request, pk=None):
        pass

    def update(self, request, pk=None):
        pass

    def partial_update(self, request, pk=None):
        pass

    def destroy(self, request, pk=None):
        pass
```

```
from django.contrib.auth.models import User
from rest_framework import status, viewsets
from rest_framework.decorators import action
from rest_framework.response import Response
from myapp.serializers import UserSerializer, PasswordSerializer

class UserViewSet(viewsets.ModelViewSet):
    """
    A viewset that provides the standard actions
    """

    queryset = User.objects.all()
    serializer_class = UserSerializer

    @action(detail=True, methods=['post'])
    def set_password(self, request, pk=None):
        user = self.get_object()
        serializer = PasswordSerializer(data=request.data)
        if serializer.is_valid():
            user.set_password(serializer.validated_data['password'])
            user.save()
            return Response({'status': 'password set'})
        else:
            return Response(serializer.errors,
                           status=status.HTTP_400_BAD_REQUEST)

    @action(detail=True, methods=['post', 'delete'])
    def unset_password(self, request, pk=None):
        ...
```

URLViewSet – не используем

```
from django.contrib import admin
from stocks import views as stock_views
from django.urls import include, path
from rest_framework import routers

router = routers.DefaultRouter()
router.register(r'stocks', stock_views.StockViewSet)

# Wire up our API using automatic URL routing.
# Additionally, we include login URLs for the browsable API.
urlpatterns = [
    path('', include(router.urls)),
    path('api-auth/', include('rest_framework.urls', namespace='rest_framework')),

    path('admin/', admin.site.urls),
]
```

Go

Для Go также необходимо добавить:

- сериализацию в JSON
- обращение к БД через ORM

После методички
рекомендуется
посмотреть наш ролик
VK по авторизации
Golang

```
func main() {
    router := gin.Default()

    // This handler will match /user/john but will not match /user/ or /user/
    router.GET("/user/:name", func(c *gin.Context) {
        name := c.Param("name")
        c.String(http.StatusOK, "Hello %s", name)
    })

    // However, this one will match /user/john/ and also /user/john/send
    // If no other routers match /user/john, it will redirect to /user/john/
    router.GET("/user/:name/*action", func(c *gin.Context) {
        name := c.Param("name")
        action := c.Param("action")
        message := name + " is " + action
        c.String(http.StatusOK, message)
    })
}
```

Инструменты тестирования

Инструменты для нашего курса

- Postman
- Insomnia

Очень удобно, но необходимо реализовать

- Swagger

The screenshot shows a web browser window with the URL `https://localhost:5001/index.html`. The page title is "JWT Auth API v1 OAS3". Below the title, there is a link to `/swagger/v1/swagger.json`. A descriptive text states "A simple example JWT Auth API" and provides links to "GitHub Repository" and "Website". On the right side of the page, there is a green "Authorize" button with a lock icon. The main content area is titled "Account" and contains a single POST method: "POST /Account/login JWT login". Below this, there is a section titled "Values" containing three GET methods:

- "GET /api/Values" - API allows anonymous access.
- "GET /api/Values/jwt" - API requires JWT auth.
- "GET /api/Values/basic" - API requires Basic auth.

Each method entry includes a lock icon on the right.

Swagger

Developing APIs

- When creating APIs, Swagger tooling may be used to automatically generate an Open API document based on the code itself. This embeds the API description in the source code of a project and is informally called code-first or bottom-up API development.
- Alternatively, using Swagger Codegen, developers can decouple the source code from the Open API document, and generate client and server code directly from the design. This makes it possible to defer the coding aspect.

Interacting with APIs

- Using the Swagger Codegen project, end users generate client SDKs directly from the OpenAPI document, reducing the need for human-generated client code. As of August 2017, the Swagger Codegen project supported over 50 different languages and formats for client SDK generation.

Documenting APIs

- When described by an OpenAPI document, Swagger open-source tooling may be used to interact directly with the API through the Swagger UI. This project allows connections directly to live APIs through an interactive, HTML-based user interface. Requests can be made directly from the UI and the options explored by the user of the interface.

Insomnia

The screenshot shows the Insomnia API client interface. At the top, the title bar reads "BMSTU Lab 6 – create new stock". Below it, the navigation bar includes "Dashboard / BMSTU Lab 6", "No Environment", "Cookies", and "Send" (highlighted with a purple arrow). The main area shows a GET request to "http://127.0.0.1:8000/stocks". The status bar indicates "200 OK", "18.8 ms", and "2 B". A large white arrow points from the word "URL" to the URL input field. Another large white arrow points from the word "Статус" (Status) to the status code "200 OK". A third large white arrow points from the word "Ответ" (Response) to the "Preview" section, which displays the JSON response: "1 []".

BMSTU Lab 6 – create new stock

Dashboard / BMSTU Lab 6

No Environment Cookies

Send

200 OK

18.8 ms

2 B

Just Now

URL

Статус

Ответ

Select a body type from above

\$.store.books[*].author

Insomnia

The screenshot shows the Insomnia API client interface. At the top, it displays "BMSTU Lab 6 – create new stock". Below the header, there's a toolbar with "Dashboard / BMSTU Lab 6", environment dropdown ("No Environment"), cookies, and a "POST" button pointing to "http://127.0.0.1:8000/stocks/". The main area shows a "Send" button, a status bar indicating "201 Created", "8.34 ms", "110 B", and "Just Now", and a "Preview" tab showing a JSON response. The "JSON" tab is selected, and the request body contains the following JSON:

```
{"company_name": "Mail.Ru", "price": 100500, "is_growing": true}
```

Three large white arrows point from the explanatory text below to the "JSON" tab, the request body, and the response body respectively.

Выбор типа тела запроса JSON

JSON структура запроса

JSON структура ответа

Beautify JSON \$.store.books[*].author

Insomnia

The screenshot shows the Insomnia API client interface. At the top, the title bar reads "BMSTU Lab 6 – create new stock". The left sidebar has a "Dashboard / BMSTU Lab 6" icon and a "No Environment" dropdown. The main request panel shows a "GET" request to "http://127.0.0.1:8000/stocks/". The response status is "200 OK" with a duration of "23.6 ms" and a size of "355 B", timestamped "Just Now". The response body is displayed in JSON format:

```
1  {"company_name": "Mail.Ru", "price": 100500, "is_growing": true}
2
3  [
4    {
5      "pk": 1,
6      "company_name": "Mail.Ru",
7      "price": "100500.00",
8      "is_growing": true,
9      "date_modified": "2021-10-25T18:19:31.705915Z"
10 },
11 {
12   "pk": 2,
13   "company_name": "Mail.Ru",
14   "price": "100500.00",
15   "is_growing": true,
16   "date_modified": "2021-10-25T18:28:42.563412Z"
17 },
18 {
19   "pk": 3,
20   "company_name": "Mail.Ru",
21   "price": "100500.00",
22   "is_growing": true,
23   "date_modified": "2021-10-25T18:28:56.302564Z"
24 }
```

At the bottom, there are "Beautify JSON" and ".store.books[*].author" buttons.

Postman

- Другое распространенное приложение для тестирование HTTP-методов
- Также можно тестировать grpc-методы

The screenshot shows the Postman interface for a **PUT** request to `http://0.0.0.0:8000/stocks/8/put/`. The **Body** tab is selected, showing the JSON payload:

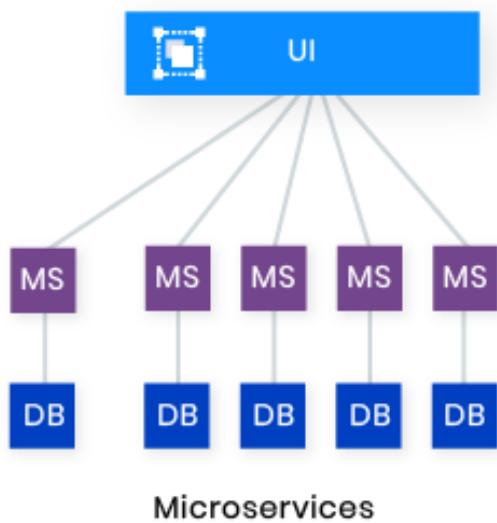
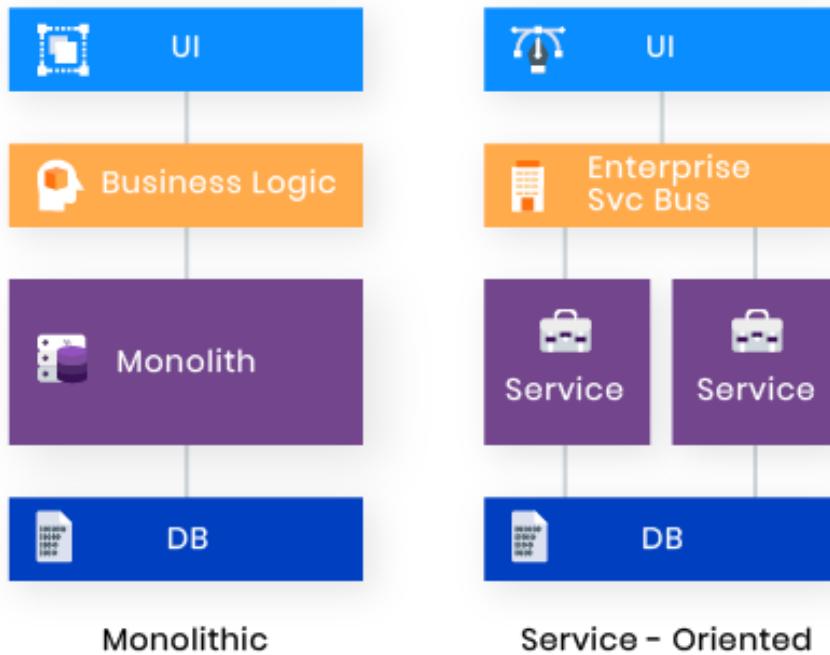
```
1 { "company_name": "Zarathustra", "price": 200000, "is_growing": true }
```

The response body is displayed in the **Pretty** tab:

```
1 {
2   "pk": 8,
3   "company_name": "Zarathustra",
4   "price": "200000.00",
5   "is_growing": true,
6   "date_modified": "2023-09-30T13:02:04.857242Z"
7 }
```

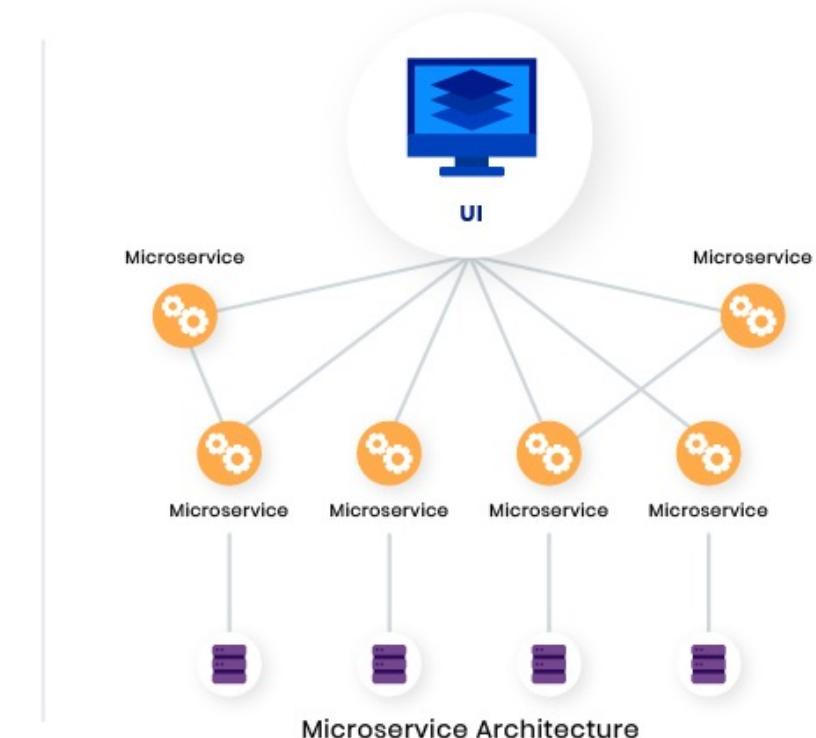
Сервис-ориентированная архитектура

- **Сéрвис-ориентированная архитектúра** (СОА, англ. service-oriented architecture- SOA) — модульный подход к разработке ПО на основе использования по стандартизованным протоколам распределённых, слабо связанных и легко заменяемых сервисов со стандартизованными интерфейсами.
- Большие системы могут взаимодействовать друг с другом через SOA (доп. задание к ДЗ). Ранее могли через DB-лиинки



Микросервисная архитектура

- **Микросервисная архитектура** — вариант сервис-ориентированной архитектуры программного обеспечения на основе взаимодействия небольших, слабо связанных и легко изменяемых модулей - микросервисов
- Теперь каждая система состоит из множества отдельных сервисов, выполняющих самостоятельную бизнес-функцию
- Например можно разделить услуги, заявки и авторизацию
- Получила распространение в середине 2010-х годов в связи с развитием практик гибкой разработки и DevOps



Файловое хранилище

Отправка, получение файлов в файловом хранилище

- Необходимо настроить файловое хранилище S3 и разработать методы доступа к нему.
- Хранилище необходимо для синхронизации двух узлов (условные клиент и сервер облака).
- Должна быть предусмотрена синхронизация, если одна из версий устарела (на клиенте не было интернета)

S3

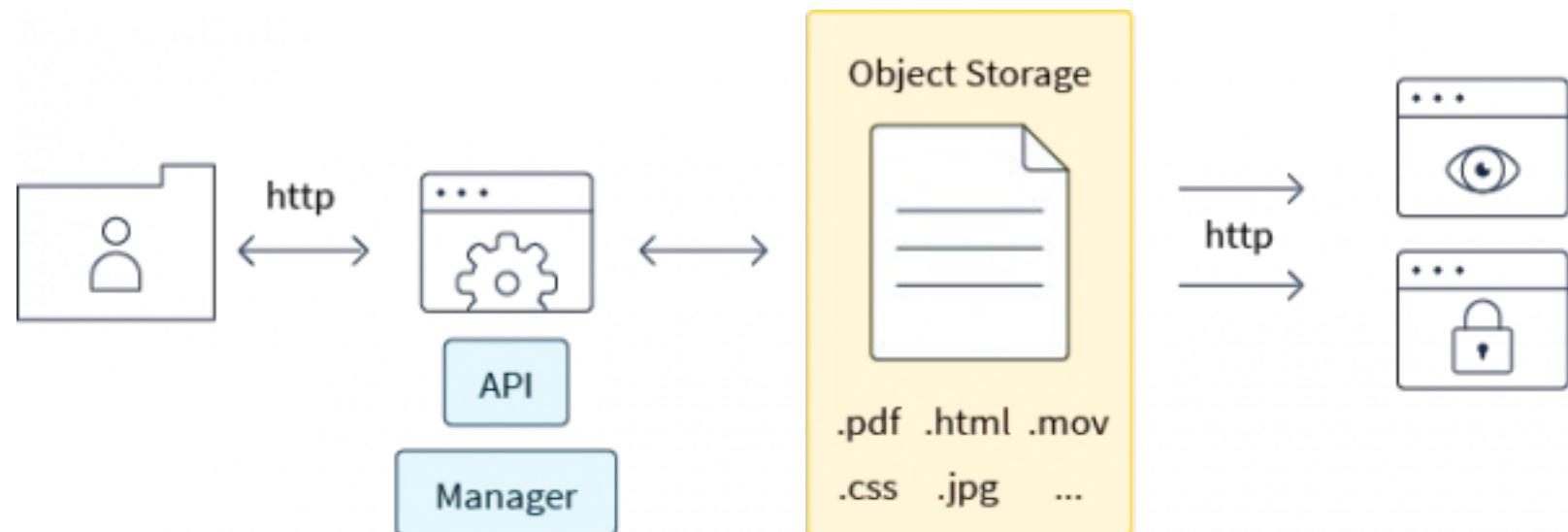
- S3 - Simple Storage Service
- Ceph, Minio и тд

Создание озера данных

- Приложения для аналитики больших данных, искусственного интеллекта (ИИ), машинного обучения (ML) и высокопроизводительных вычислений (HPC).
- Объектное хранилище S3 — масштабируемый и гибкий сервис, позволяющий хранить и обрабатывать данные в исходном формате
- Данные размещаются в виде объектов в плоском адресном пространстве, что дает возможность для доступа к хранилищу по API из любого места
- Данные помещаются во множество контейнеров (папок)
- Содержимое любого контейнера можно просматривать, перемещать или удалять
- У каждого контейнера и объекта есть адрес в виде уникального идентификатора

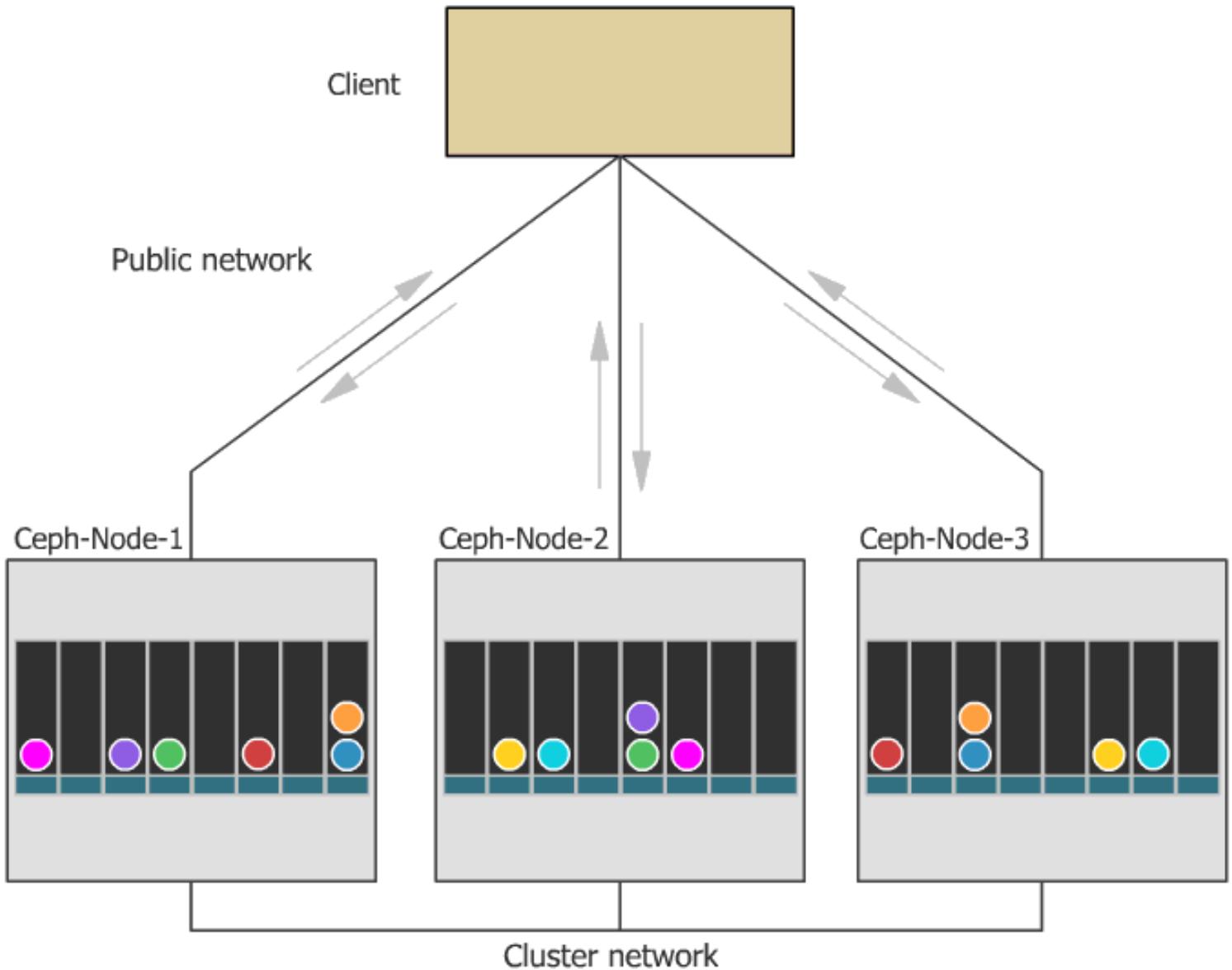
Серф

- Свободная программная объектная сеть хранения
- Обеспечивает файловый и блочный интерфейсы доступа
- Объектное хранилище



Ceph

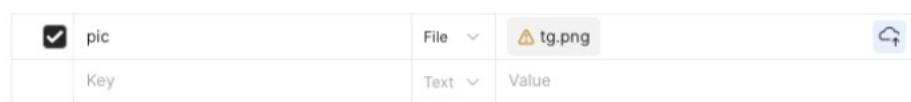
- При выходе любого диска, узла или группы узлов из строя Ceph обеспечит сохранность данных
- Ceph восстановит утраченные копии на других узлах до тех пор, пока вышедшие из строя узлы или диски не заменят на рабочие.



Minio загрузка изображения

- Мы не загружаем изображения напрямую в Minio из браузера, потому что далее мы ограничим на это права пользователей
- Нам требуется загрузить/заменить изображение в Minio, а в Postgres - url

```
AWS_STORAGE_BUCKET_NAME = 'logo'  
AWS_ACCESS_KEY_ID = 'minioadmin'  
AWS_SECRET_ACCESS_KEY = 'minioadmin'  
AWS_S3_ENDPOINT_URL = 'localhost:9000'  
MINIO_USE_SSL = False
```



Body Cookies Headers (10) Test Results

```
client = Minio(  
    endpoint=settings.AWS_S3_ENDPOINT_URL,  
    access_key=settings.AWS_ACCESS_KEY_ID,  
    secret_key=settings.AWS_SECRET_ACCESS_KEY,  
    secure=settings.MINIO_USE_SSL)
```

```
client.put_object('logo', image_name, file_object, file_object.size)  
return f"http://localhost:9000/logo/{image_name}"
```

Pretty Raw Preview Visualize JSON

```
1 {  
2     "pk": 11,  
3     "company_name": "telegram",  
4     "price": "2.00",  
5     "is_growing": true,  
6     "date_modified": "2024-01-08T01:54:03.838713Z",  
7     "user": "Иван Иванов",  
8     "url": "http://localhost:9000/logo/11.png"  
9 }
```

Облачные сервисы и Firebase

- Хостинги для вашего бекенда: Heroku, AWS и тд
- Платно + администрирование
- Любая архитектура

Firebase

- Удобно использовать для мобильных приложений
- Low/No code – готовая конфигурация

The screenshot shows the Firebase console interface. On the left, there's a sidebar with various services: Project Overview, Firestore Database, Functions, Analytics Dashboard, Crashlytics, Extensions (NEW), **Authentication** (which is highlighted with a green border), and App Check. The main area is titled "Authentication" and contains tabs for Users, Sign-in method, Templates, Usage, and Settings. The "Settings" tab is selected and shows a section for "Authorized domains". It includes a table with one row for "localhost" under the "Type" column. A blue arrow points from the "Authentications" section in the sidebar to the "Settings" tab in the main content area.

Authorized domains

Authorized domain	Type
localhost	Default

Firebase
The Realtime Backend
for your App!

Synchronize Across Web and Mobile Devices

Realtime Database
Authentication
Cloud Messaging
Storage
Hosting
Remote Config
Test Lab
Crash Reporting