

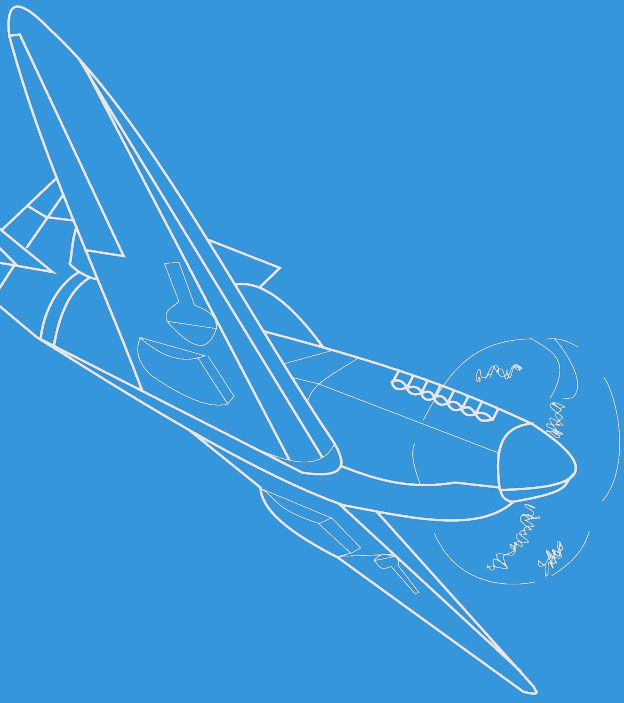
# Реализация API

на котором расскажут про API, какие бывают форматы передачи данных, и погрузимся в формы в Django, как их валидировать, и подключим Django Rest Framework

---

## Содержание занятия

1. API;
2. Текстовые протоколы;
3. Двоичные протоколы;
4. REST и RPC;
5. gRPC;
6. Формы;
7. Валидация форм;
8. Django Rest Framework;



API

# Application programming interface (API)



Описание способов (набор классов, процедур, функций, структур или констант), которыми одна компьютерная программа может взаимодействовать с другой программой.

# Виды совместимости приложений



- **Обратная совместимость** — более новый код способен читать данные, записанные более старым;
- **Прямая совместимость** — более старый код способен читать данные, записанные более новым.

# REST (REpresentational State Transfer)



**REST API** подразумевает под собой простые правила:

- Каждый URL является ресурсом;
- При обращении к ресурсу методом GET возвращается описание этого ресурса;
- Метод POST добавляет новый ресурс;
- Метод PUT изменяет ресурс;
- Метод DELETE удаляет ресурс.



- Конечные точки в URL — имя существительное, не глагол;
  - + `/posts/`
  - `/getPosts/`
- Используйте множественное число для названия своих REST сервисов;
- Документирование программного обеспечения является общей практикой для всех разработчиков;
- Версионность
  - URI версии.
  - Мультимедиа версии.



**JSON-RPC** (JavaScript Object Notation Remote Procedure Call — JSON-вызов удалённых процедур) — протокол удалённого вызова процедур, использующий JSON для кодирования сообщений.





Формат входного запроса:

- `method` — строка с именем вызываемого метода;
- `params` — массив объектов, которые должны быть переданы методу, как параметры;
- `id` — значение любого типа, которое используется для установки соответствия между запросом и ответом.



Формат ответа:

- `result` — данные, которые вернул метод. Если произошла ошибка во время выполнения метода, это свойство должно быть установлено в `null`;
- `error` — код ошибки, если произошла ошибка во время выполнения метода, иначе `null`;
- `id` — то же значение, что и в запросе, к которому относится данный ответ.

# JSON-RPC



Пример запроса:

```
{ "method": "echo", "params": ["Hello JSON-RPC"], "id": 1 }
```

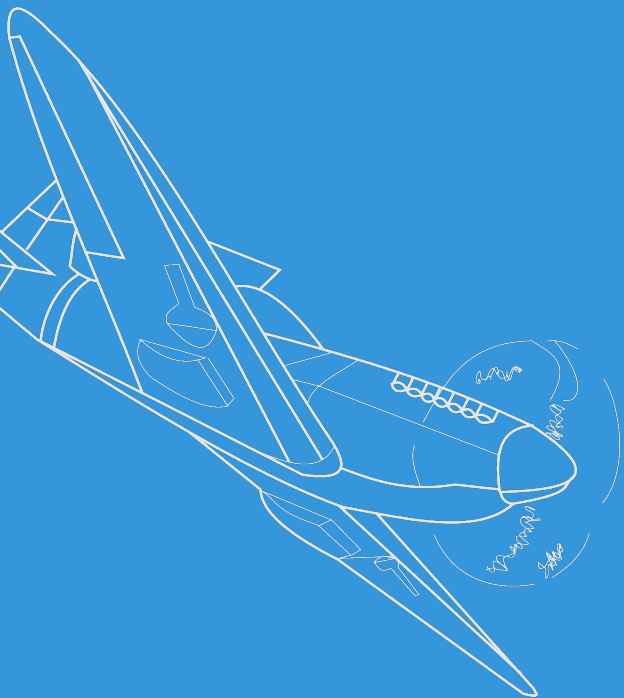
Пример ответа:

```
{ "result": "Hello JSON-RPC", "error": null, "id": 1 }
```

# Форматы передачи данных



- Текстовые форматы (JSON, XML, CSV);
- Бинарный формат (Apache Thrift, Protocol Buffers);



# Текстовые форматы



- Каждая строка файла — это одна строка таблицы.
- Разделителем значений колонок является символ запятой (,)
- Однако на практике часто используются другие разделители.

```
year,vendor,model ,desc,price
```

```
1997,Ford,E350,"ac, abs, moon",3000.00
```

```
1999,Chevy,"Venture «Extended Edition»","",4900.00
```

```
1996,Jeep,Grand Cherokee,"MUST SELL! air, moon roof,  
loaded",4799.00
```



**XML** (eXtensible Markup Language) – язык разметки, позволяющий стандартизировать вид файлов-данных, используемых компьютерными программами, в виде текста, понятного человеку.

```
<note>  
  <to>Tove</to>  
  <from>Jani</from>  
  <heading>Reminder</heading>  
  <body>Don't forget me this weekend!</body>  
</note>
```

# Формат XML



- Синтаксис XML избыточен;
- XML не содержит встроенной в язык поддержки типов данных;
- + Есть схема;
- + Человекочитаемый.



# Формат JSON

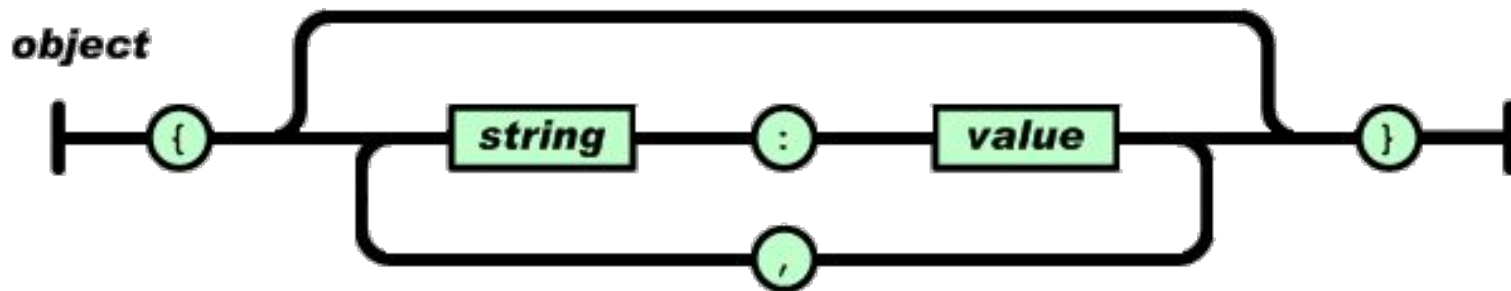


**JSON** (JavaScript Object Notation) — текстовый формат обмена данными, основанный на JavaScript.

```
{  
    "first_name": "Иван",  
    "last_name": "Иванов",  
    "phone_numbers": [  
        "812 123-1234",  
        "916 123-4567"  
    ]  
}
```

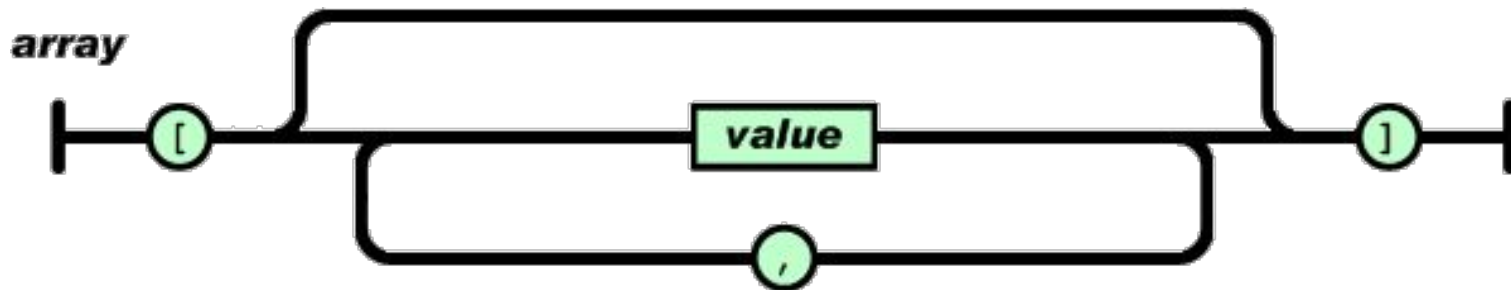
JSON основан на двух структурах данных:

1. Коллекция пар ключ/значение. В разных языках, эта концепция реализована как объект, запись, структура, словарь, хэш, именованный список или ассоциативный массив;



JSON основан на двух структурах данных:

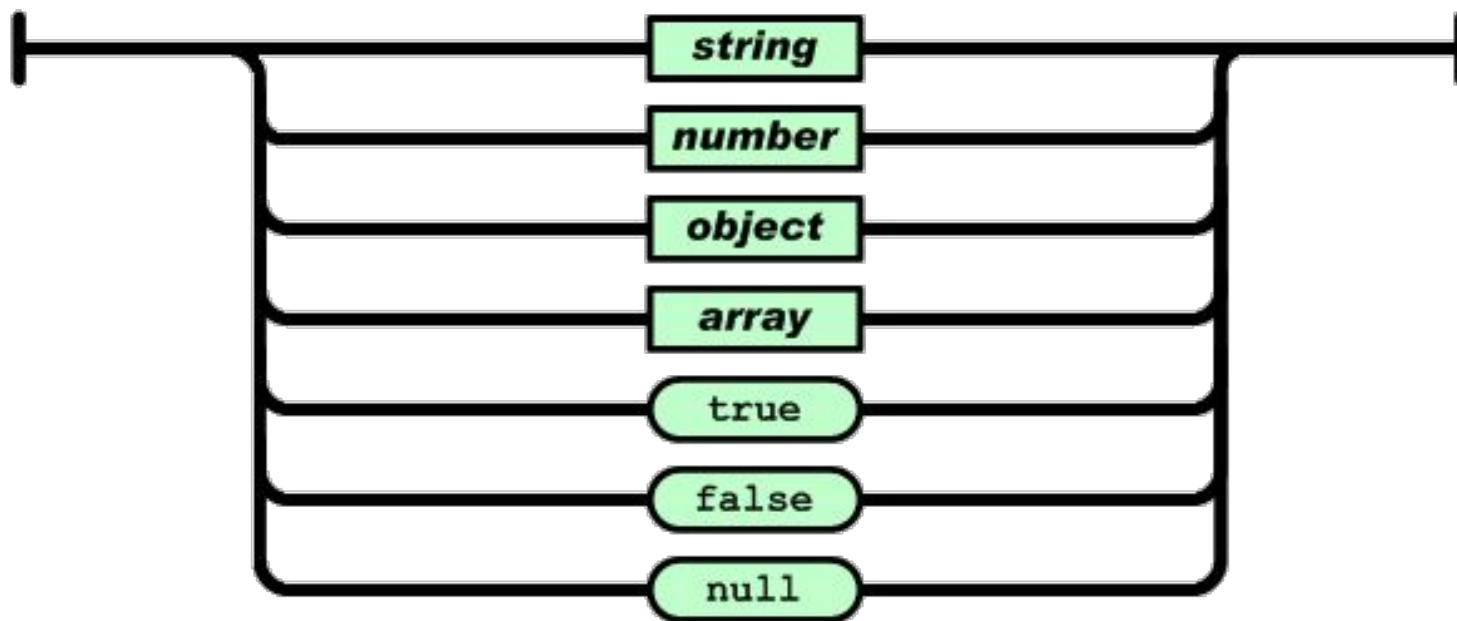
2. Упорядоченный список значений. В большинстве языков это реализовано как массив, вектор, список или последовательность.



# Формат JSON



***value***



# Преимущества JSON



- Легко читается человеком;
- Компактный;
- Для работы с JSON есть множество библиотек;
- Больше структурной информации в документе.

# Преимущества JSON



- JSON - это формат данных - он содержит только свойства, а не методы;
- JSON требует двойных кавычек, которые будут использоваться вокруг строк и имен свойств;
- Вы можете проверить JSON с помощью приложения, такого как jsonlint;
- JSON может фактически принимать форму любого типа данных, который действителен для включения внутри JSON, а не только массивов или объектов.



# Двоичные форматы

# Преимущества двоичного кодирования



- Они могут быть намного компактнее различных вариантов “двоичного JSON”, поскольку позволяют не включать названия полей в закодированные данные;
- Схема — важный вид документа, вы всегда можете быть уверены в её актуальности;
- Пользователем языков программирования со статической типизацией окажется полезная возможность генерировать код на основе схемы, позволяющая проверять типы во время компиляции.



# Protocol buffers



**Protocol Buffers** — протокол сериализации (передачи) структурированных данных, предложенный Google как эффективная бинарная альтернатива текстовому формату XML. Проще, компактнее и быстрее, чем XML.

# Protocol buffers



```
syntax = "proto3";
```

```
package tutorial;
```

```
option go_package = "./";
```

```
message Person {
```

```
    string user_name = 1;
```

```
    int64 favorite_number = 2;
```

```
    repeated string interests = 3;
```

```
}
```

# Protocol buffers

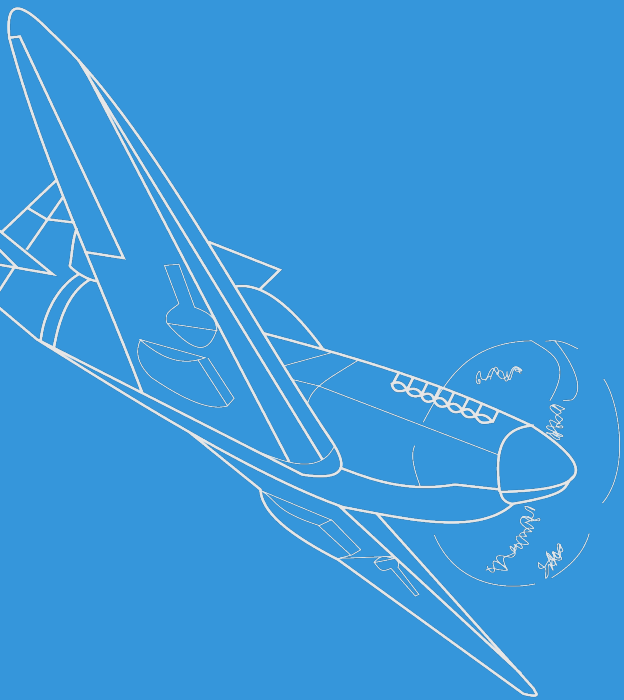


```
$ protoc -I=./src/ --go_opt=paths=source_relative  
--python_out=python/ --cpp_out=cpp/ --go_out=golang/  
./src/person.proto
```

```
$ ls python/  
person_pb2.py
```

# Работа в питоне

```
from person_pb2 import Person  
person = Person()
```



# gRPC

g for goat!\*

\* - [https://grpc.github.io/grpc/core/md\\_doc\\_g\\_stands\\_for.html](https://grpc.github.io/grpc/core/md_doc_g_stands_for.html)

# Преимущества gRPC



- + Высокая эффективность;
- + Простые, чёткие интерфейсы и форматы сообщений;
- + Сильная типизация;
- + Многоязычие;
- + Двухнаправленная потоковая передача;
- + Встроенные практические возможности.

# Недостатки gRPC



- gRPC может не подойти для сервисов, доступных снаружи;
- Кардинальные изменения в определении сервисов требуют больших усилий;
- Относительно небольшая экосистема.

# Процесс передачи сообщений в RPC



- Клиентский процесс вызывает функцию `<function>` из сгенерированной заглушки;
- Клиентская заглушка создаёт HTTP-запрос типа POST с закодированным сообщением;
- HTTP-запрос с сообщением отправляется на серверный компьютер по сети;
- Получив сообщение, сервер анализирует его заголовки, чтобы узнать, какую функцию нужно вызвать, и передает его заглушке сервиса;
- Заглушка сервиса преобразует байты сообщения в структуры данных конкретного языка;
- Затем сервис, используя преобразованное сообщение, вызывает локальную функцию `<function>`;
- Ответ функции сервиса кодируется и возвращается клиенту.



- Унарный (Unary RPC);
- Серверный стрим (Server streaming RPC);
- Клиентский стрим (Client streaming RPC);
- Двухнаправленный стрим (Bidirectional streaming);





```
syntax = "proto3";

package account;

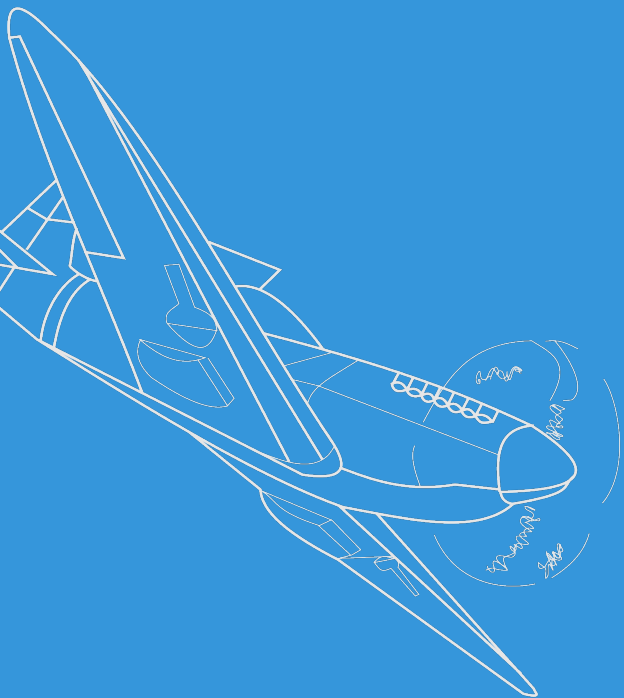
import "google/protobuf/empty.proto";

service UserController {
    rpc List(UserListRequest) returns (stream User) {}
    rpc Create(User) returns (User) {}
    rpc Retrieve(UserRetrieveRequest) returns (User) {}
    rpc Update(User) returns (User) {}
    rpc Destroy(User) returns (google.protobuf.Empty) {}
}
```

```
message User {
    int32 id = 1;
    string username = 2;
    string email = 3;
    repeated int32 groups = 4;
}

message UserListRequest {
}

message UserRetrieveRequest {
    int32 id = 1;
}
```



# Декораторы в Python

# Декораторы в Python



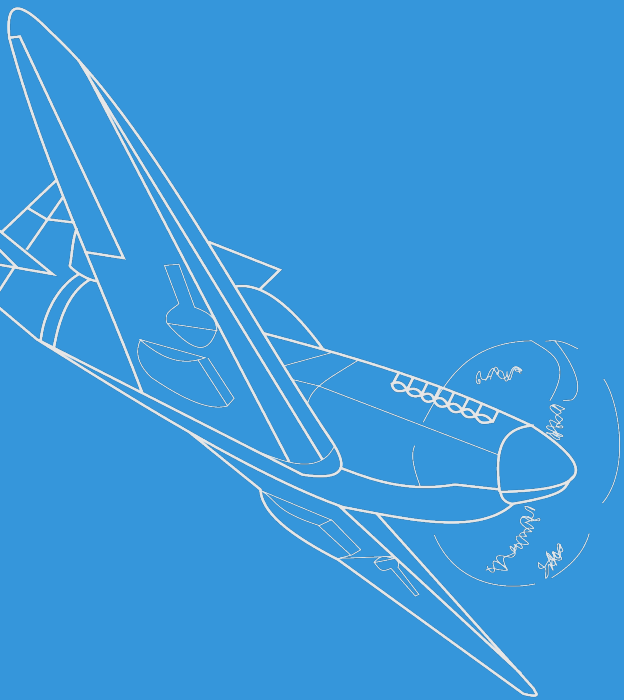
Это функция, которая принимает функцию в качестве единственного аргумента и возвращает новую функцию, с дополнительными функциональными возможностями.

```
def my_decorator(function):  
    def wrapper(*args, **kwargs):  
        print('It is decorator logic')  
        return function(*args, **kwargs)  
    return wrapper  
  
@my_decorator  
def foo():  
    print('It is main function')
```

# Декораторы с параметрами



Вызываем функцию с требуемыми параметрами, и она вернёт декоратор, который будет использован для декорирования следующей за ним функцией.



# Формы в Django



- **Всегда** проверять пользовательские данные;
- Для форм, изменяющие данные, использовать метод **POST**;
- Не заставлять вводить данные повторно;
- Сообщать об ошибках детально - по полям;
- Сообщать об успешном сохранении формы;
- При успешном сохранении делать перенаправление.

# Django Form



```
# forms.py

from django import forms

class FeedbackForm(forms.Form):
    email = forms.EmailField(max_length=100)
    message = forms.CharField()
    def clean(self):
        if is_spam(self.cleaned_data):
            self.add_error('message', 'Это спам')
```

# Django Form



```
# forms.py

from django import forms

class PostForm(forms.Form):
    title = forms.CharField(max_length=100)
    text = forms.CharField()
    days_active = forms.IntegerField(required=False)
    def clean_text(self):
        if is_correct(self.cleaned_data['message']):
            return self.cleaned_data['message']
        return 'Текст содержал нецензурную лексику и был удален'
    def save(self):
        return Post.objects.create(**self.cleaned_data)
```



# Django Form



`BooleanField` — флаг

`IntegerField` — целый тип

`CharField` — текстовое поле

`EmailField` — почтовый адрес

`PasswordField` — пароль

`DateTimeField` — дата

`DateTimeField` — время и дата

`FileField` — загрузка файла

# Model Forms



```
# forms.py

from django import forms

class PostForm(forms.ModelForm):
    class Meta:
        model = Post
        fields = ['title', 'text']
```

- метод `save` уже определен
- сохраняем в модель, указанную в `Meta`
- валидация полей проходит через типы, объявленные в модели

# Валидация формы в views



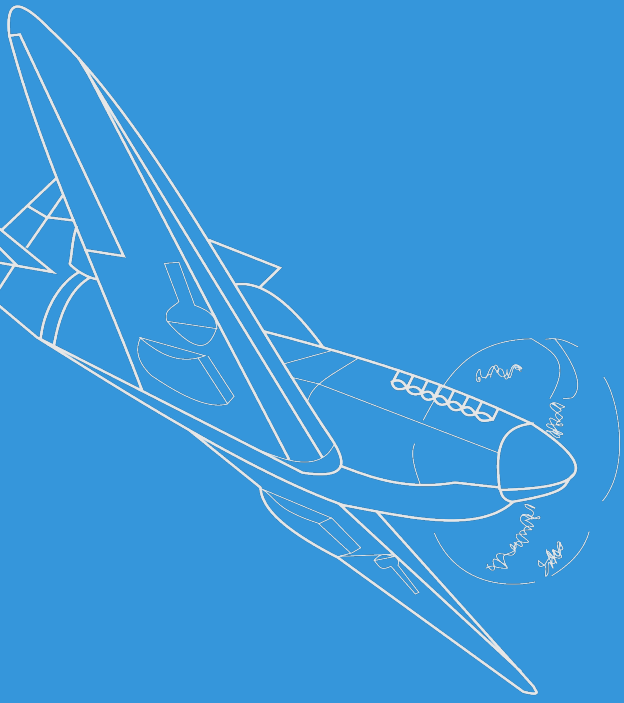
```
def add_post(request):  
    form = PostForm(request.POST)  
    if form.is_valid():  
        post = form.save()  
        return JsonResponse({  
            'msg': 'Пост сохранен',  
            'id': post.id  
        })  
    return JsonResponse({'errors': form.errors}, status=400)
```

# Сериализация данных из БД



```
def chat_detail(request, chat_id):
    chat = get_object_or_404(Chat, id=chat_id)
    return JsonResponse({
        'data': {'id': chat_id, 'title': chat.title}
    })

def chat_list(request):
    chats = Chat.objects.filter(is_active=True).values('id', 'title',
        'description')
    return JsonResponse({
        'data': list(chats)
    })
```



# Django Rest Framework

# Django Rest Framework



# Устанавливаем DRF

```
pip install djangorestframework
```

# Добавляем приложение в settings.py

```
INSTALLED_APPS = [  
    ...  
    'rest_framework',  
]
```

# Django Rest Framework



```
def add_post(request):  
    form = PostForm(request.POST)  
    if form.is_valid():  
        post = form.save()  
        return JsonResponse({  
            'msg': 'Пост сохранен',  
            'id': post.id  
        })  
    return JsonResponse({'errors': form.errors}, status=400)
```



1. Добавить в проект `djangoRESTframework`;
2. Валидировать входные параметры API с помощью форм

Переписать заглушки всех предыдущих методов.



---

## Рекомендуемая литература

1. [Высоконагруженные приложения. Программирование масштабирования поддержка | Клеппман Мартин](#)
2. [gRPC: запуск и эксплуатация облачных приложений. Go и Java для Docker и Kubernetes](#)
3. [Google Protocol Buffers](#)

Для саморазвития (опционально)  
[Чтобы не набирать двумя пальчиками](#)



Спасибо за  
внимание!

**Антон Кухтичев**



[a.kukhtichev@mail.ru](mailto:a.kukhtichev@mail.ru)



[@toshunster](https://www.instagram.com/toshunster)