

# **REST API & Retrofit**

# Материалы



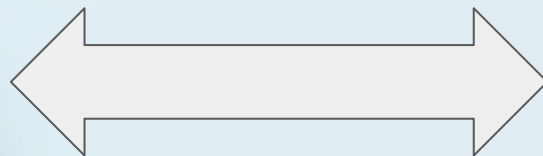
[github.com/adamxrvn/hse-lyceum-android-course](https://github.com/adamxrvn/hse-lyceum-android-course)

# REST - Пример

Приложение может общаться с сервером посредством REST API



**REST API**



# Что такое REST?

**REST** - **Re**presentational **S**tate **T**ransfer («передача репрезентативного состояния») - это архитектурный подход взаимодействия сайтов и приложений с сервером

## Преимущества REST:

- Простота/стандартизация
- Масштабируемость/отсут. состояний
- Производительность/кэширование

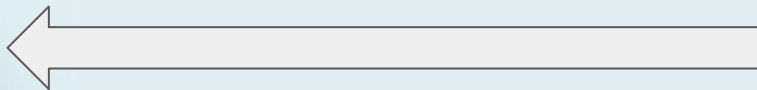
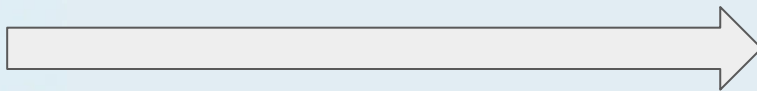
# REST - Пример

Приложение

Endpoint:  
<https://сендвичи.рф/api/menu>



Запрос



Ответ



# Запросы

Функции работы с БД

Методы HTTP-запросов

Create → Post

Read → Get

Update → Put

Delete → Delete

# Запросы

Из чего состоят запросы?

Content-Type, Authorization, и т.д.

Заголовки

GET/POST/PUT/DELETE/и т.д.

Метод

ADDRESS

Endpoint

Параметры/тело

# Пример GET запроса

Запрос:

Заголовки

**GET**

Метод

<https://сэндвичи.рф/api/menu>

Endpoint

Параметры/тело

Ответ:

```
[
  {
    "id": 0,
    "name": "Classic Italian",
    "price": 290,
    "icon": "https://img.com/ClassicItalian.png"
  },
  {
    "id": 1,
    "name": "Turkey Ranch & Swiss",
    "price": 400,
    "icon": "https://img.com/TurkeyRanch.png"
  },
  {
    "id": 2,
    "name": "Veggie Delight",
    "price": 52,
    "icon": "https://img.com/Veggie.png"
  }
]
```



# Пример PUT запроса

Запрос:

Заголовки

**PUT**

Метод

<https://сэндвичи.рф/api/menu/1>

Endpoint

`{"name": "Chicken Sandwich", "price": 350}`

Параметры/тело

Ответ:

```
{  
  "id": 1,  
  "name": "Chicken Sandwich",  
  "price": 350,  
  "icon": "https://img.com/TurkeyRanch.png"  
}
```

# Пример POST запроса

Запрос:

Заголовки

**POST**

Метод

<https://сэндвичи.рф/api/menu/>

Endpoint

```
{"name": "RESTful Sandwich", "price": 999,  
"icon": "https://img.com/rest.png"}
```

Параметры/тело

Ответ:

```
{  
  "id": 3,  
  "name": "RESTful Sandwich",  
  "price": 999,  
  "icon": "https://img.com/rest.png"  
}
```

# Пример документации

(ссылка на [гит](#))

**FastAPI** 0.1.0 OAS 3.1

[/openapi.json](#)

default



GET	/sandwiches/	Get Sandwiches	▼
POST	/sandwiches/	Add Sandwich	▼
PUT	/sandwiches/{sandwich_id}	Update Sandwich	▼
DELETE	/sandwiches/{sandwich_id}	Delete Sandwich	▼



Метод



Ресурс



Описание

# Пример документации

Параметры запроса

Тело запроса

Варианты ответа сервера

PUT

/sandwiches/{sandwich\_id} Update Sandwich

⌵

Parameters

Try it out

Name

Description

sandwich\_id \* required

integer

(path)

sandwich\_id

Request body required

application/json

Example Value | Schema

```
{  "name": "string",  "price": 0,  "icon": "https://example.com/"}
```

Responses

Code

Description

Links

200

Successful Response

No links

Media type

application/json

Controls Accept header.

Example Value | Schema

```
{  "id": 0,  "name": "string",  "price": 0,  "icon": "https://example.com/"}
```

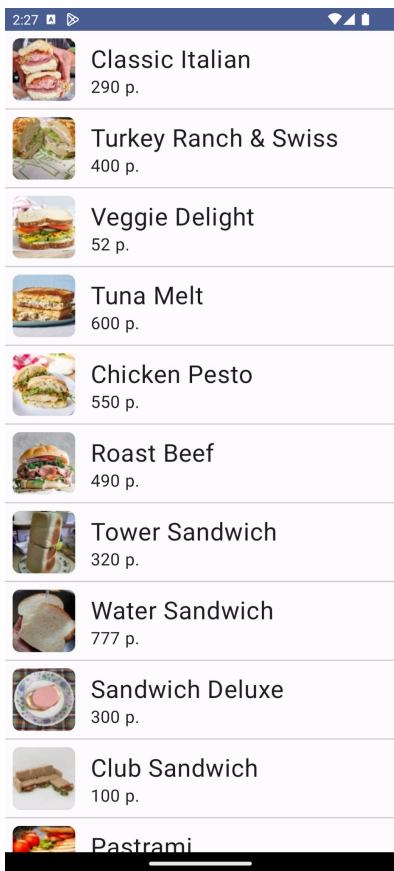
422

Validation Error

No links

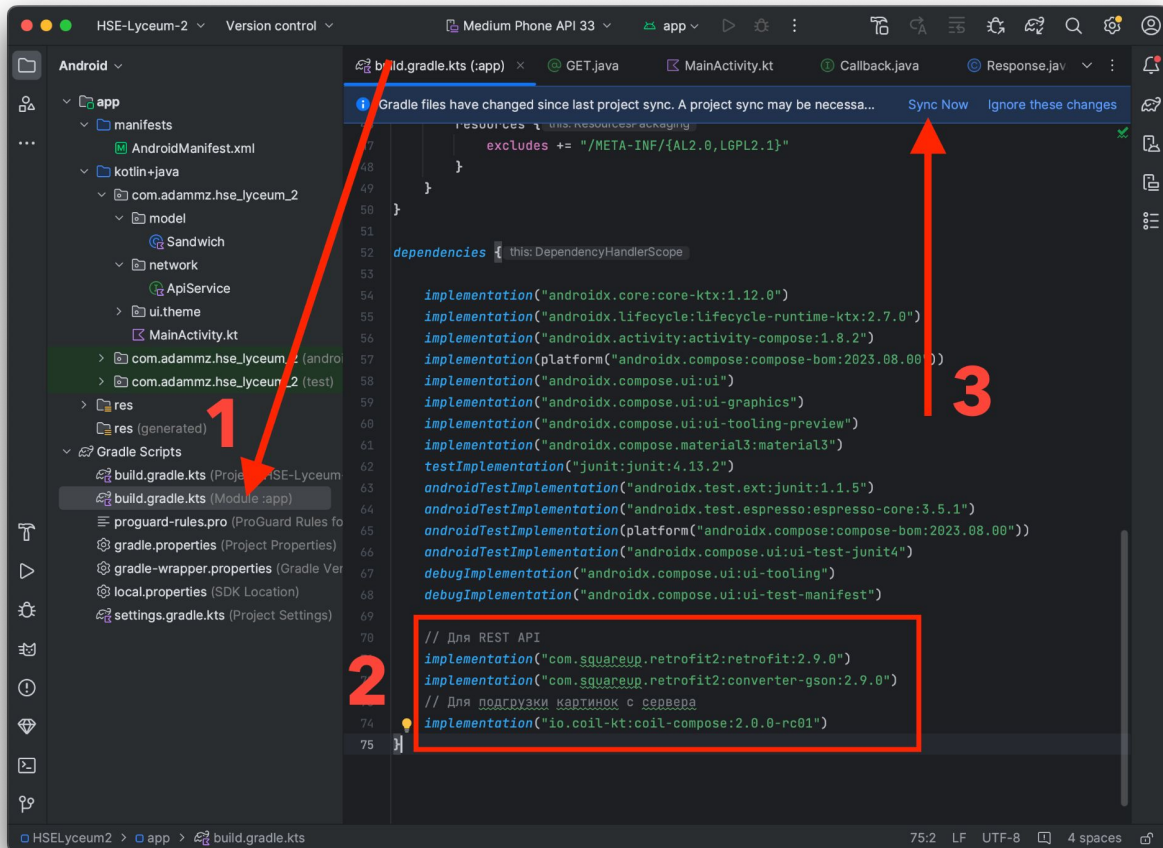
Media type

# Задача на сегодня



- Просмотреть [документацию API](#)
- Сделать GET запрос на сервер и получить данные
- Отобразить данные списком

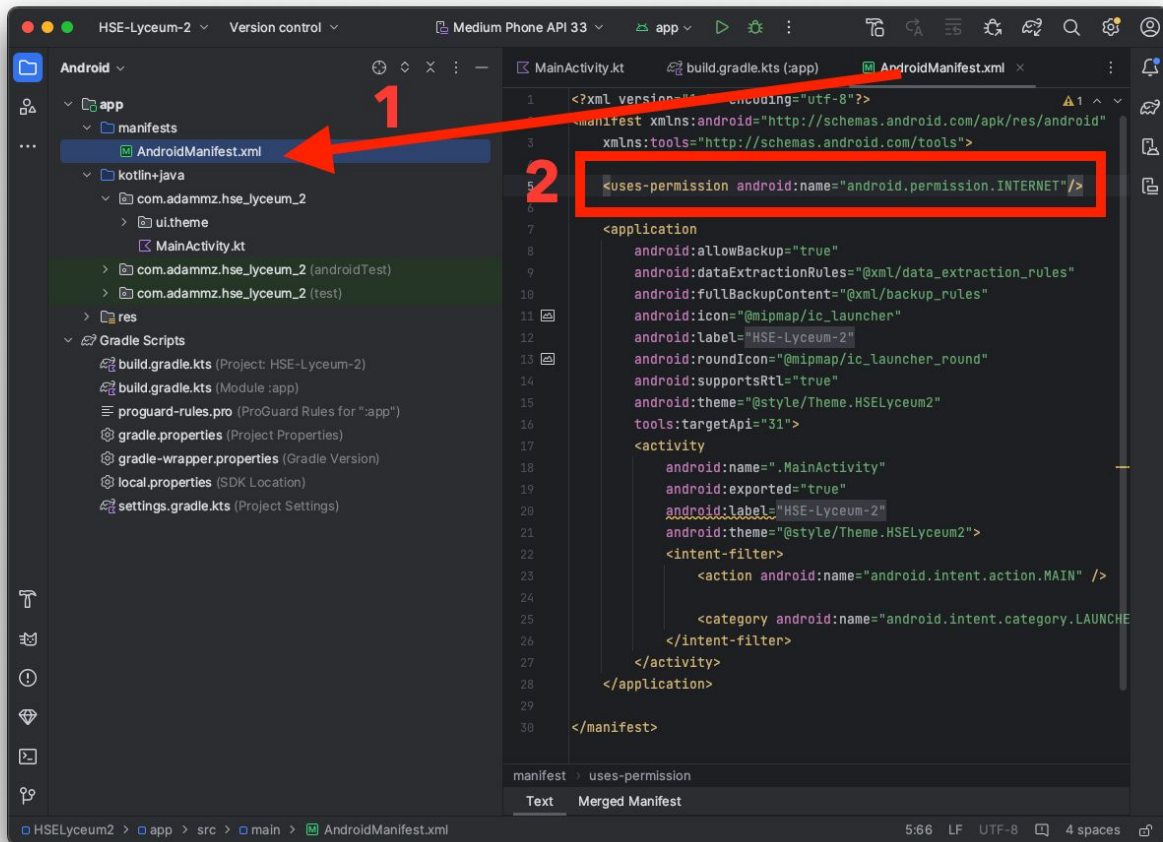
# Перейдем к Android!



Для работы с REST API  
нам нужно добавить  
библиотеку Retrofit

Также добавим  
библиотеку для  
асинхронной загрузки  
изображений

# Internet Permission



Добавляем разрешение  
для интернета

# Создание дата класса

В разработке программ часто нужны объекты, в основном для хранения информации. Kotlin предлагает **data class** — специальный тип класса, оптимизированный для создания простых структур данных, с фокусом на содержимом данных.

Data class в Kotlin упрощают создание классов для хранения данных, автоматически генерируя важные методы, как **equals()**, **hashCode()**, и **toString()**.

Примеры:

```
data class User(val name: String, val age: Int)
```

```
data class Product(val id: Int, val name: String, val price: Double)
```



# Создание дата класса

Мы знаем в каком формате сервер возвращает данные, на этой основе создадим data class

Массив объектов

```
[
  {
    "id": 0,
    "name": "Classic Italian",
    "price": 290,
    "icon": "https://img.com/ClassicItalian.png"
  },
  {
    "id": 1,
    "name": "Turkey Ranch & Swiss",
    "price": 400,
    "icon": "https://img.com/TurkeyRanch.png"
  },
  {
    "id": 2,
    "name": "Veggie Delight",
    "price": 52,
    "icon": "https://img.com/Veggie.png"
  }
]
```

Объект, в нашем случае сэндвич,  
со следующими полями:

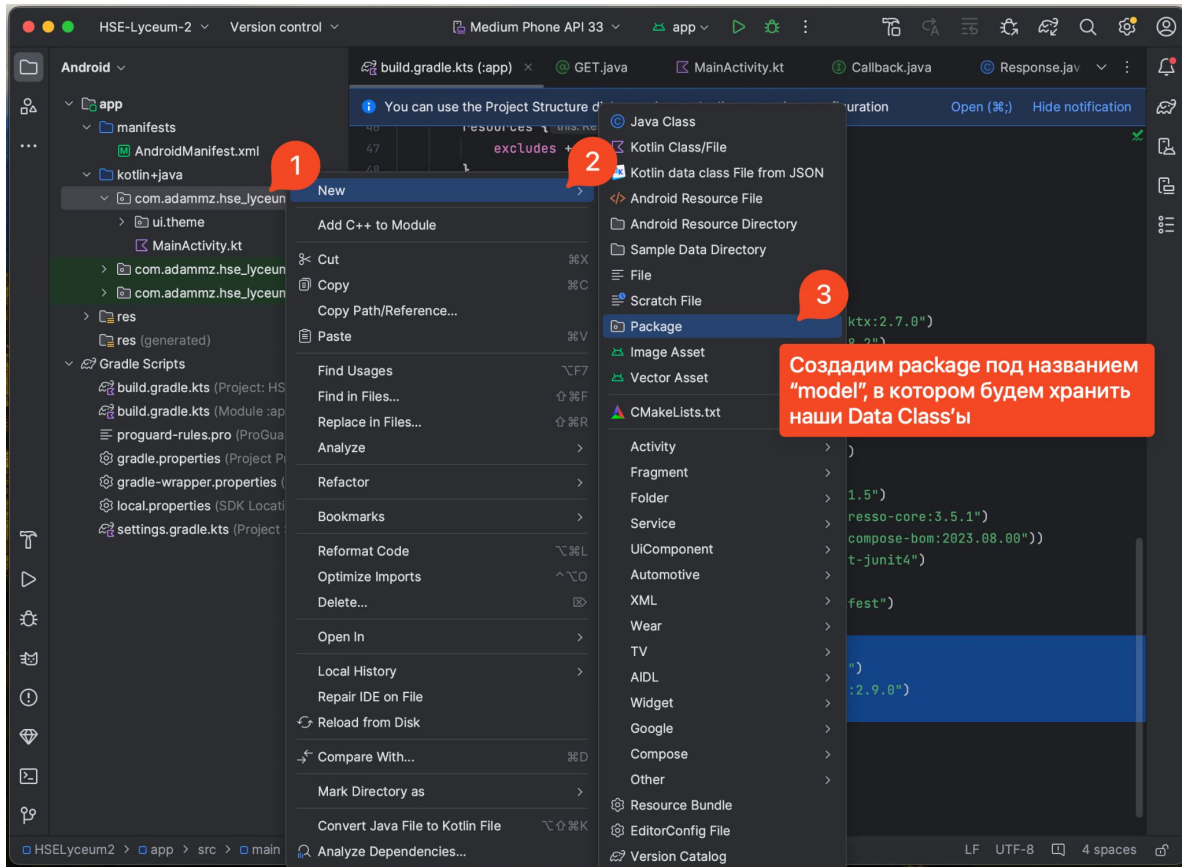
Id - Int

Name - String

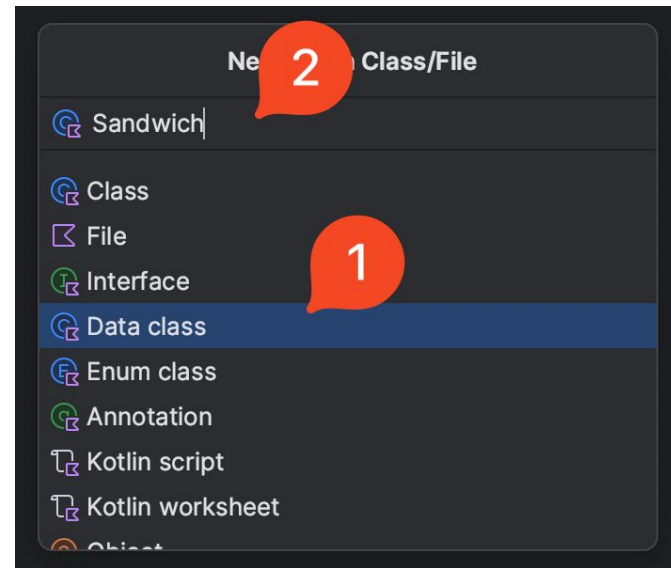
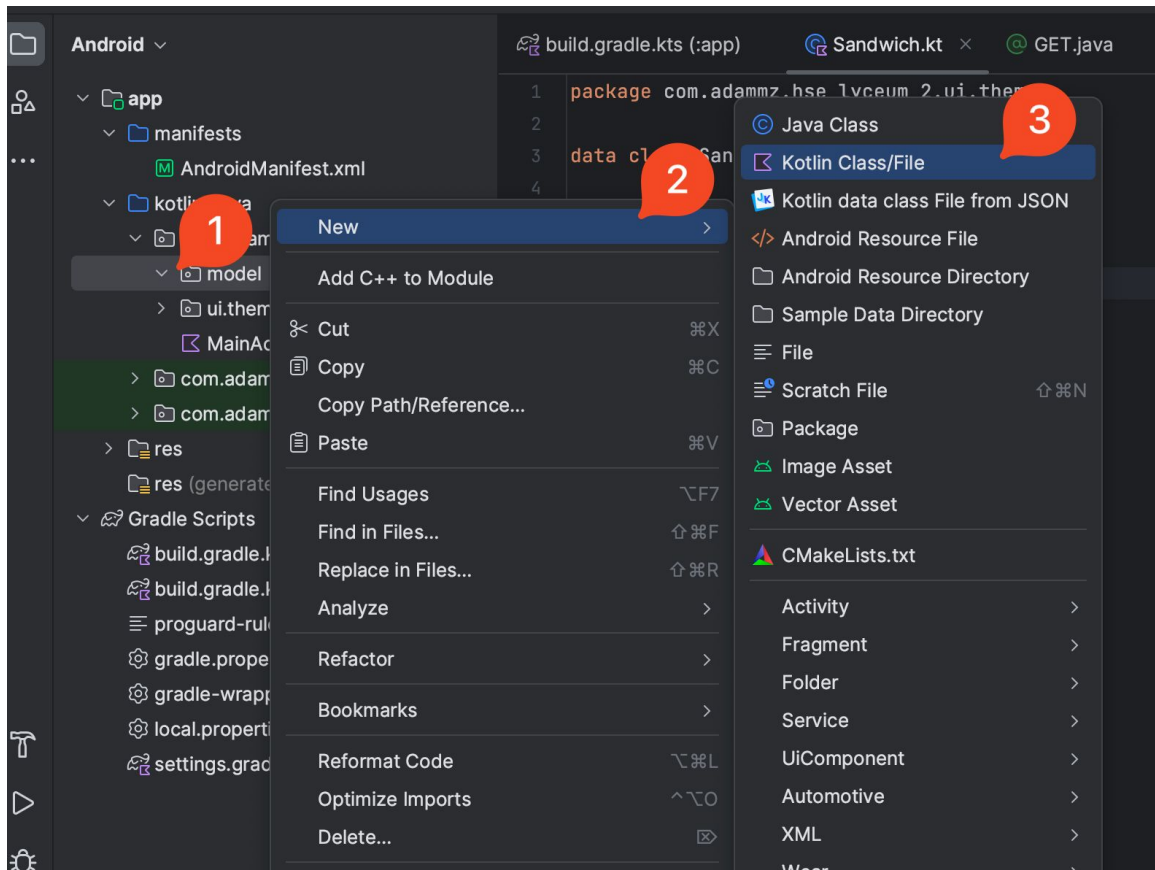
Price - Int

Icon - String

# Создание дата класса



# Создание дата класса

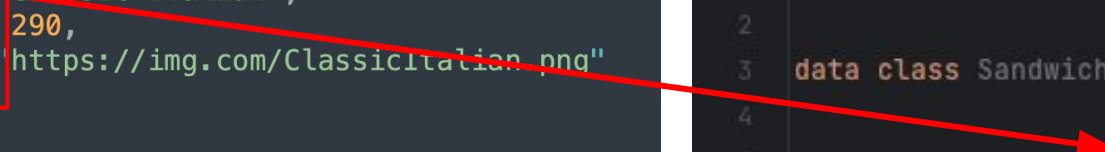


# Создание дата класса

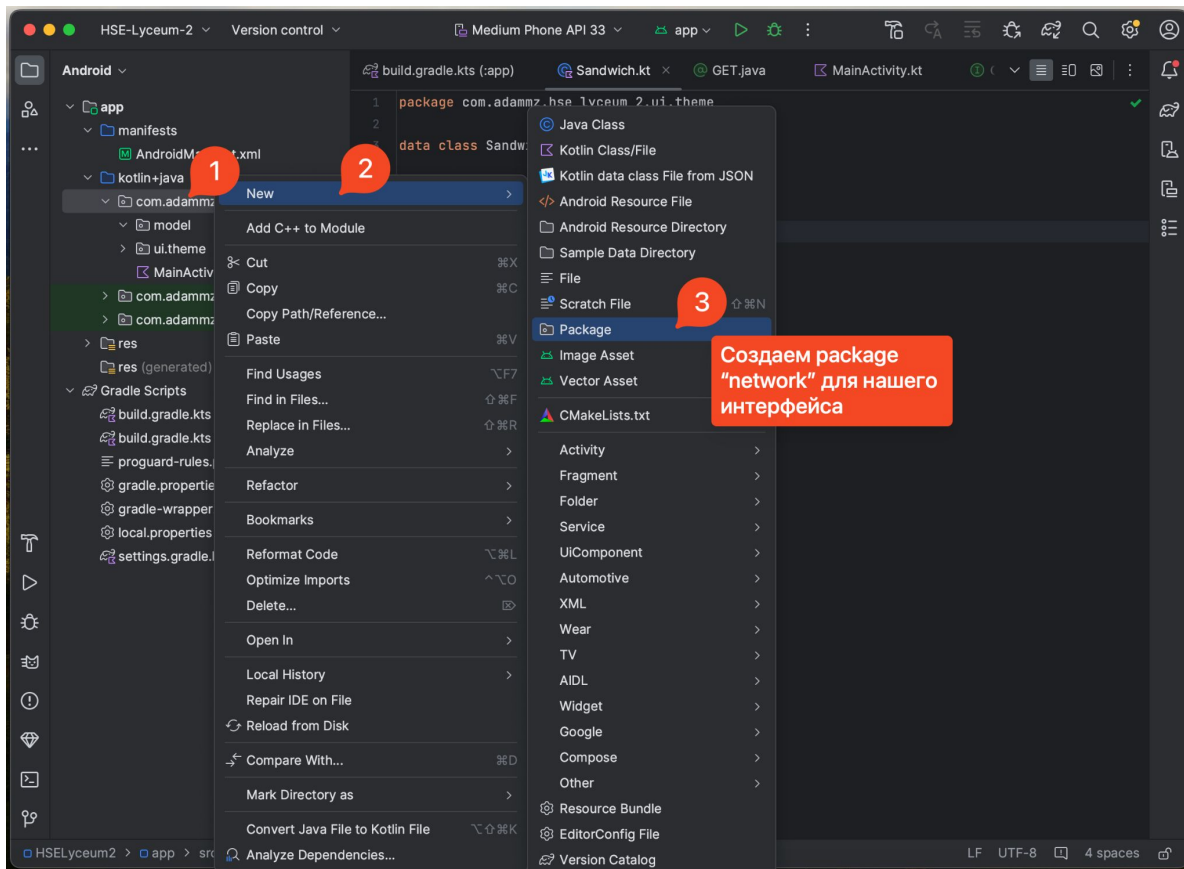
```
[  
  {  
    "id": 0,  
    "name": "Classic Italian",  
    "price": 290,  
    "icon": "https://img.com/ClassicItalian.png"  
  },  
  {  
    "id": 1,  
    "name": "Turkey Ranch & Swiss",  
    "price": 400,  
    "icon": "https://img.com/TurkeyRanch.png"  
  },  
  {  
    "id": 2,  
    "name": "Veggie Delight",  
    "price": 52,  
    "icon": "https://img.com/Veggie.png"  
  }  
]
```

Sandwich.kt ×

```
1 package com.adammz.hse_lyceum_2.model  
2  
3 data class Sandwich(val id: Int,  
4                     val name: String,  
5                     val price: Int,  
6                     val icon: String)  
7
```

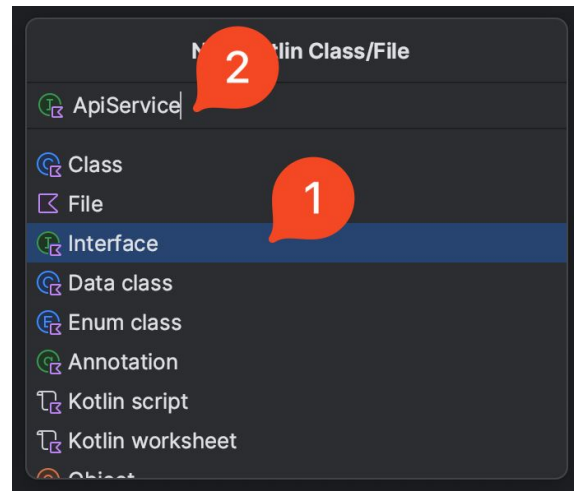
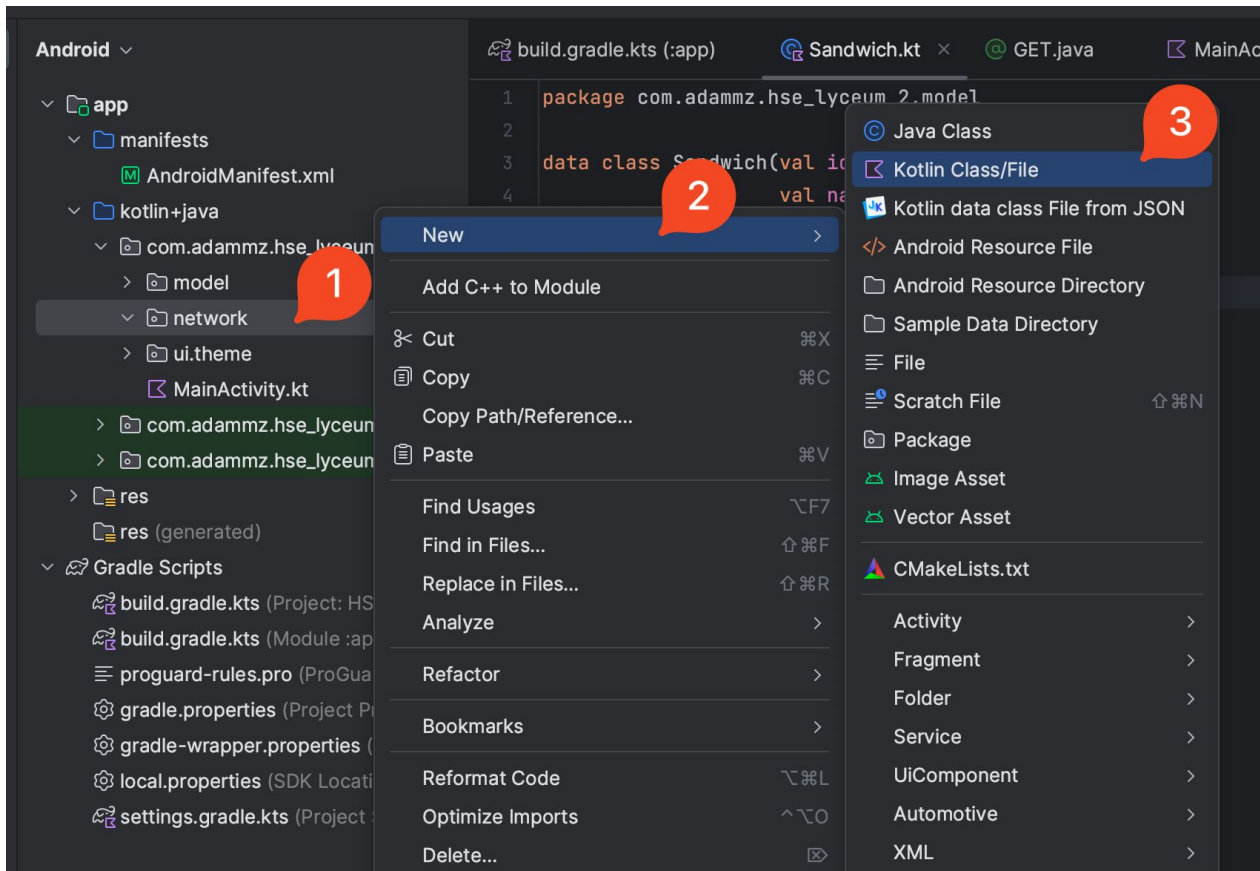


# Создаем интерфейс



В интерфейсе определяются запросы, которые будут отправлены серверу. Эти запросы объединяются с основным адресом сайта, полученным через метод `baseUrl()`, формируя таким образом полный URL-адрес к конкретному ресурсу.

# Создаем интерфейс



# Создаем интерфейс

```
interface ApiService {  
    Метод Путь к endpoint
```

В каком формате сервер возвращает  
ответ (см. документацию API)

```
    @GET("/sandwiches")
```

```
    fun getSandwiches(): Call<List<Sandwich>>
```

```
    @POST("/sandwiches")
```

```
    fun addSandwich(@Body sandwich: Sandwich): Call<Sandwich>
```

```
    @PUT("/sandwiches/{id}") Body параметер
```

```
    fun updateSandwich(@Path("id") id: Int): Call<Sandwich>
```

```
    @DELETE("/sandwiches/{id}") Path параметер
```

```
    fun deleteSandwich(@Path("id") id: Int)
```

```
}
```

# Создаем интерфейс

Аннотация	Описание
@GET()	Отправляет GET-запрос. Можно добавить дополнительные параметры запроса.
@POST()	Отправляет POST-запрос. Параметры запроса могут быть указаны в скобках.
@Path	Заменяет часть URL-адреса на значение переменной, например, {id} заменяется на конкретный идентификатор пользователя.
@Query	Определяет параметр запроса с его значением, добавляемый к URL.
@Body	Преобразует данные объекта Java в JSON для тела POST-запроса.
@Header	Добавляет заголовок HTTP-запроса с указанным значением.
@Headers	Определяет несколько HTTP-заголовков одновременно.
@Multipart	Применяется для отправки файлов или изображений в теле запроса.
@FormUrlEncoded	Используется для отправки данных формы в формате ключ-значение через POST-запрос.
@FieldMap	Позволяет отправлять данные формы в формате ключ-значение через POST-запрос, используя карту значений.
@Url	Используется для указания полного или частичного URL-адреса запроса динамически.



# Создаем интерфейс

GET

/sandwiches/ Get Sandwiches



```
@GET("/sandwiches")  
fun getSandwiches(): Call<List<Sandwich>>
```

POST

/sandwiches/ Add Sandwich



```
@POST("/sandwiches")  
fun addSandwich(@Body sandwich: Sandwich): Call<Sandwich>
```

PUT

/sandwiches/{sandwich\_id} Update Sandwich



```
@PUT("/sandwiches/{id}")  
fun updateSandwich(@Path("id") id: Int): Call<Sandwich>
```

DELETE

/sandwiches/{sandwich\_id} Delete Sandwich



```
@DELETE("/sandwiches/{id}")  
fun deleteSandwich(@Path("id") id: Int)
```