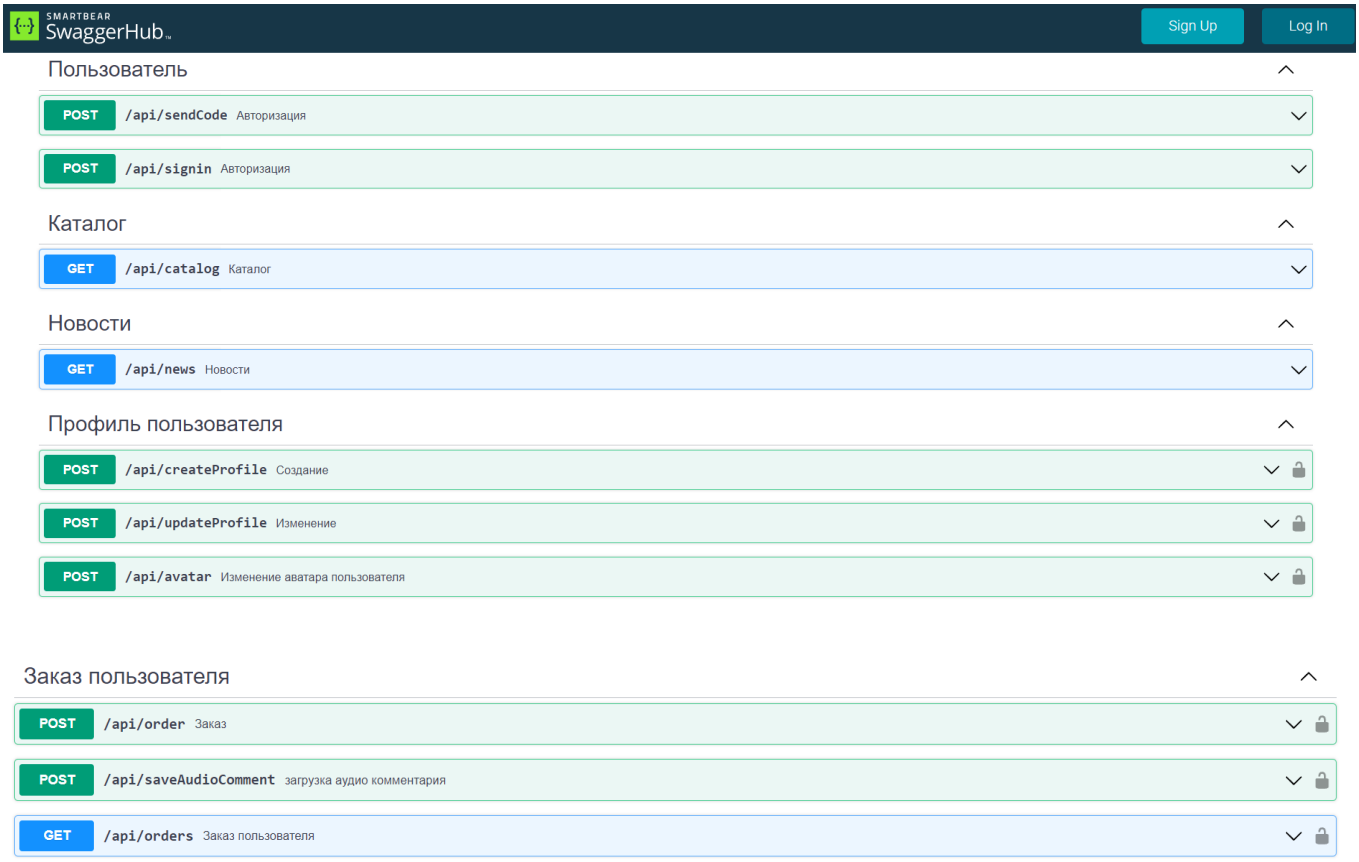
Работа с RestApi - Retrofit

Дергаем запросы с сервера

В первую очередь нужно понимать, что Retrofit - это просто высокоуровневая оберточка над Okhttp3. Если вам нужно перехватывать соединение, работать с сокетами и тщательно отслеживать сессии пользователя, то он подходит для это лучше.

Остальной пласт задач Retrofit покрывает сполна. Дергать простые и не очень запросы с его помощью достаточно просто. Кодогенерация и GsonConverter в этом очень сильно помогают.

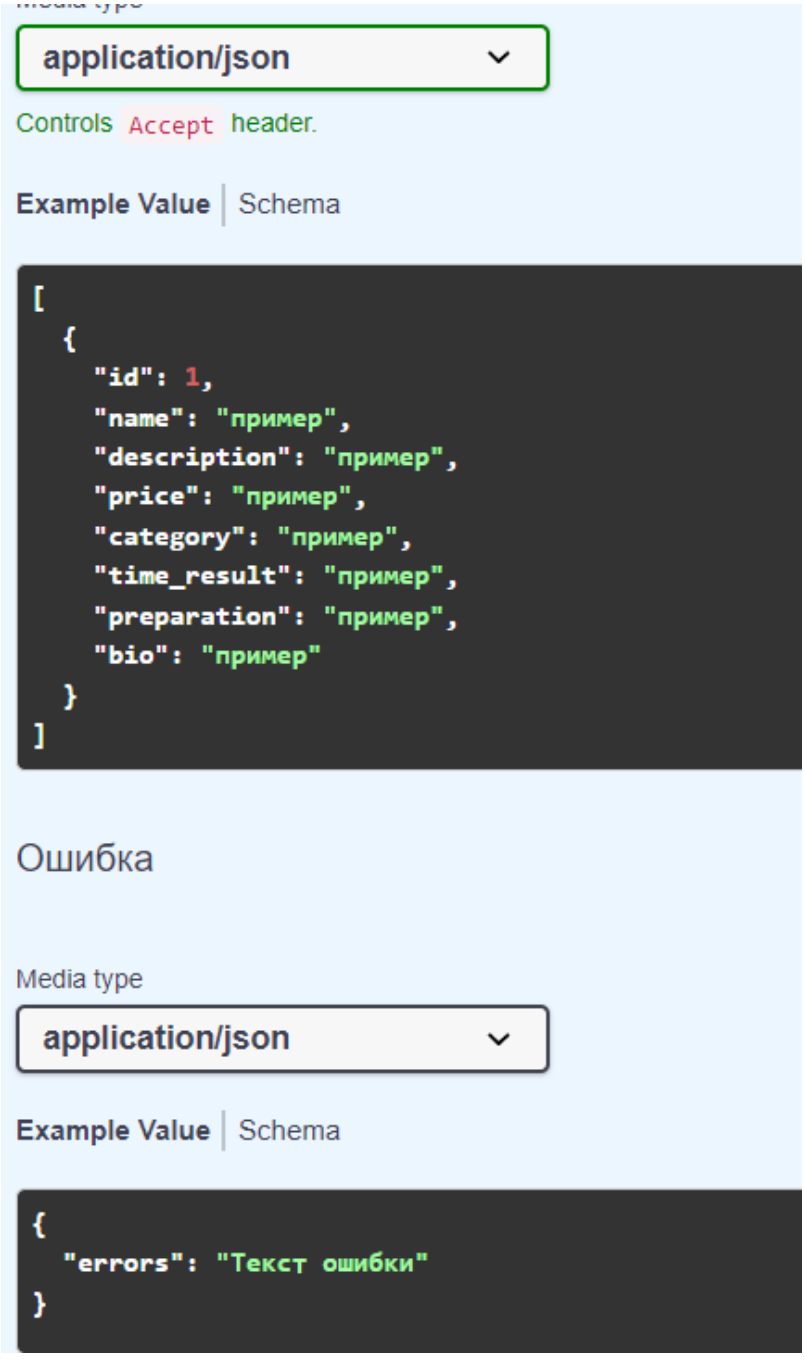
Для начал, посмотрим на API с которым нам предстоит работать
API : https://medic.madskill.ru/
Документация: https://app.swaggerhub.com/apis-docs/serk87/APIfood/FRBHWRIOJAFIDSNKJF
Документацию нужно изучать в обязательном порядке и очень тщательно, от неё мы будем проектировать структуры классов в нашем приложении.



Видим, что в данном случае нам доступны PUT и GET запросы, так же там недавно появились и PUT запросы.

GET

С POST запросами разберемся попозже, а пока посмотрим, что у нас происходит в GET. Пойдем сверху вниз.



Это ответ с метода catalog. Видим структуру json'a.
Она достаточно простая, т.е. нам приходит массив с полями, id, name, description и т.д.
Если произойдет ошибка, то вернется exception.

Т.к. у нас уже есть формат ответа, то мы можем сразу же написать класс в который будут мапиться полученные данные

```
data class CatalogCloudItem(
    val bio: String,
    val category: String,
    val description: String,
    val id: Int,
    val name: String,
    val preparation: String,
    val price: String,
    val time_result: String
)
```

Он нужен для того что бы GsonConverter смог распарсить ответ и по названию параметров присвоить им значения.

```
class CatalogCloud : ArrayList<CatalogCloudItem>()
```

Теперь можно приступить к работе.
Сперва, для использования ретрофита нужно добавить зависимости в build.gradle модуля проекта.

```
implementation 'com.squareup.retrofit2:retrofit:2.9.0'
implementation 'com.squareup.retrofit2:converter-gson:2.9.0'
```

Так же я добавлю зависимости для viewmodel и livedata. К данной теме они не относятся, но они пригодятся для запуска корутин.

```
implementation 'androidx.lifecycle:lifecycle-viewmodel-ktx:2.6.1'
implementation 'androidx.lifecycle:lifecycle-runtime-ktx:2.6.1'
implementation 'androidx.lifecycle:lifecycle-livedata-ktx:2.6.1'
```

И не отходя от кассы добавим нужные разрешения в manifest. А именно разрешение на интернет и на ACCESS_NETWORK_STATE. Так же добавим usesCleartextTraffic для того, что бы можно было работать и с http

```
//Android Manifest
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />

<application
    android:name=".MedicApp"
    android:usesCleartextTraffic="true"
```

ApiService

RetrofitBuilder должен получить класс, в котором будут описаны все запросы. В котлине для этого отлично подходят интерфейсы.

```
interface ApiService {
    @GET("catalog")
   suspend fun getCatalog() : CatalogCloud
}
```

Вот тут мы остановимся поподробнее.
Аннотация GET принимает в себя string, в котором будет указано название метода, который мы запрашиваем с сервера.
Очень важен здесь возвращаемый тип, если он будет указан неверно, то GsonConverter не сможет нормально смапить результат, будет ошибка. В данном случае, мы будем возвращать список объектов CatalogCloudItem.

RetrofitBuilder

Мы описали интерфейс, но мы даже не знаем по какому адресу стучаться.
Для этого нам нужно написать RetrofitBuilder. Т.к. он может использоваться в многих местах приложения, то нужно сделать его синглтоном. В данном случае, он написать просто в обджекте, но правильной реализацией будет инициализировать его в application.

```
object RetrofitBuilder {
    private const val BASE_URL = "https://medic.madskill.ru/api/"
    private val retrofit = Retrofit.Builder()
        .baseUrl(BASE_URL)
        .addConverterFactory(GsonConverterFactory.create())
        .build()
    val apiService = retrofit.create(ApiService::class.java)
}
```

Здесь у нас будет приватная переменная с базовым url.
Смысл в том, что при генерации запроса этот базовый url будет объединён с путем, который написан внутри аннотации GET и на сервер будет послан url следующего вида:
https://medic.madskill.ru/api/catalog
Далее нам нужно написать сам RetrofitBuilder, он представляет собой паттерн Builder(кто бы сомневался) и в него нужно передать, минимум 2 параметра:

- BASE_URL

- Конвертер Json, в данном случае это Gson от Google, который мы добавляли второй зависимостью.
  Далее нужно вызвать метод build.

В этом файле будет только одна паблик переменная, с помощью которой мы будем получать доступ к ApiService. У билдера нужно просто вызывать метод create и передать в него интерфейс с описанными запросами.

Sealed class или как красиво обработать результат

Сразу отмечу, что Sealed классы являются плюшевыми, по той причине, что они являются плюшевыми и по сути ничего не делают, однако в контексте обработки стейта они очень удобны.

```
sealed class NetworkResult<T>(val data: T? = null, val message: String? = null) {
    class Success<T>(data: T) : NetworkResult<T>(data)
        class Error<T>(errorMessage: String?, data: T? = null
        ) : NetworkResult<T>(data, errorMessage)
    class Loading<T> : NetworkResult<T>()
}
```

Обрабатываем результат

```
interface CloudDataSource {
    suspend fun getCatalog(): NetworkResult<CatalogCloud>

    class Base(private val apiService: ApiService) : CloudDataSource {
        override suspend fun getCatalog(): NetworkResult<CatalogCloud>  {
            return try {
                NetworkResult.Success(apiService.getCatalog())
            } catch (e: Exception) {
                NetworkResult.Error(e.message ?: "unknown exception")
            }
        }
    }
}
```

На этом этапе у нас работа с ретрофитом заканчивается и осталось просто обработать работу с сервером. Ведь помимо успешного запроса, может произойти и ошибка. По любой причине.
Для того, что бы не завязываться на какой-то конкретный объект мы будем работать просто с NetworkResult. Здесь в примере так же показана суть interfaceSegregation, мы просто разделили интерфейсы на два разных.

Так же напишем класс репозитория, на текущий момент у нас нет CacheDataSource, но в случае если он появится мы сможем легко

```
class RepositoryImpl(
    private val cloudDataSource: CloudDataSource
    ) : Repository {
    override suspend fun getCatalog(): NetworkResult<CatalogCloud> {
       return cloudDataSource.getCatalog()
    }
}
```

Пришло время для viewModel в нашем случае viewModel будет максимально простая. У нас будет просто suspend метод getCatalog в котором мы будем получать данные

```
class MainViewModel(
    private val repository: RepositoryImpl,
    private val networkCommunication: NetworkCommunication,
) : ViewModel(), BaseObserve<NetworkResult<CatalogCloud>> {
    override fun observe(
        owner: LifecycleOwner,
        observer: Observer<NetworkResult<CatalogCloud>>,
    ) {
        networkCommunication.observe(owner, observer)
    }
    suspend fun getCatalog() {
        networkCommunication.map(NetworkResult.Loading())
        val result = repository.getCatalog()
        networkCommunication.map(result)
    }
}
```

networkCommunication и интерфейс BaseObserve это просто обертки над LiveData. В целом можно написать и в стиле google, но в этом случае мы будем отдавать поля класса наружу, а это дурной тон.

```
//ViewModel
private val _catalogList = MutableLiveData<NetworkResult>()
val catalogList : LiveData<NetworkResult>
    get() = _catalogList
suspend fun getCatalog() {
    _catalogList.value = repository.getCatalog()
```

Стоит уделить внимание тому как инжектится viewModel. Это происходит в классе Application. В котором мы можем пробросить все необходимые зависимости. Так же нужно будет добавить этот класс в тег android:name=".MedicApp" в manifest.

```
class MedicApp : Application() {

    lateinit var mainViewModel: MainViewModel
    private val apiService = RetrofitBuilder.apiService
    override fun onCreate() {
        super.onCreate()
        mainViewModel = MainViewModel(
            RepositoryImpl(CloudDataSource.Base(apiService)),
            NetworkCommunication.Base()
        )
    }
}
```

Перейдем к activity

```
class MainActivity : AppCompatActivity() {
    lateinit var binding : ActivityMainBinding
    lateinit var viewModel: MainViewModel
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        viewModel = (application as MedicApp).mainViewModel
        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)
        getData()
        observe()
    }
    private fun getData(){
        lifecycleScope.launch {
            viewModel.getCatalog()
        }
    }
    private fun observe(){
        viewModel.observe(this){
            when(it){
                is NetworkResult.Success
                -> {Log.d("market_twits",  it.data.toString() )}
                is NetworkResult.Error
                -> {Log.d("market_twits",  it.message.toString() )}
                is NetworkResult.Loading
                -> {Log.d("market_twits",  "loading") }
            }
```

```
        }
    }
}
```

Здесь раскрывается суть sealed классов, а именно то, что мы можем через проверку на instance обрабатывать состояние вот таким образом.
Т.е. в зависимости от того, какой state пришел нам во viewModel мы будем выполнять какие-то действия. Посмотрим, что получилось.

```
PROCESS STARTED (23154) for package com.example.medicapp --------------------------
--
D  loading
D  [CatalogCloudItem(bio=Слизистая,category=Популярные, description=Клинический
анализ крови – это самое важное комплексное....
```

Как видим, как и положено, у нас отображается сначала progress, а потом мы получаем данные в лог. Проверим как отработает проверка ошибок. Отключим интернет

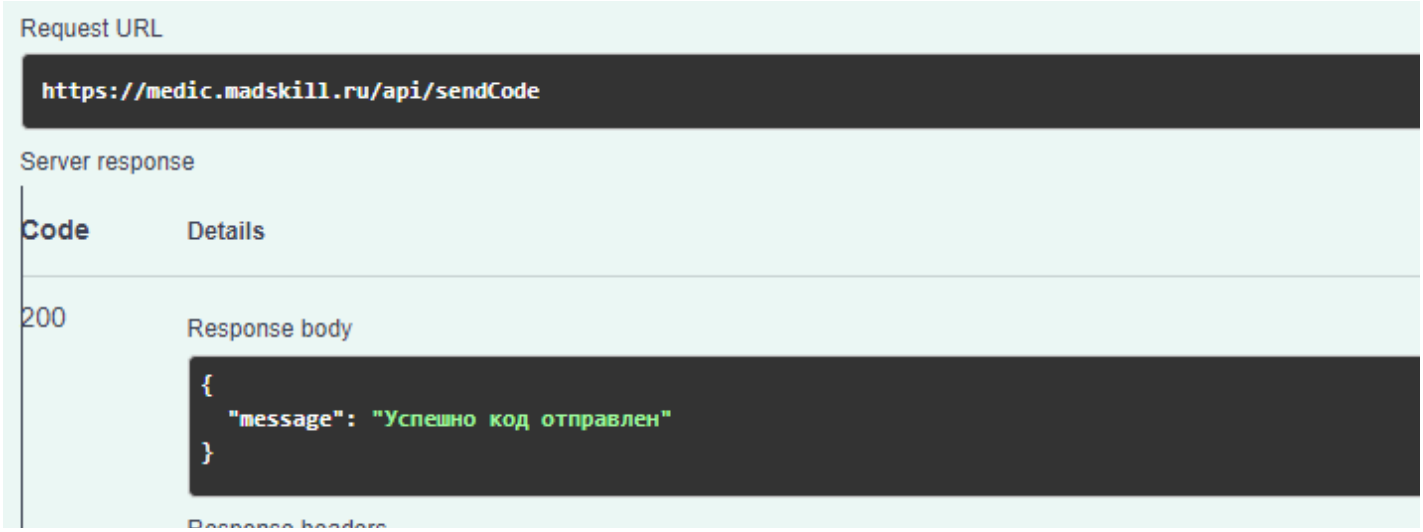
```
D  loading
D  Unable to resolve host "medic.madskill.ru": No address associated with hostname
```

Все хорошо, сначала отображается progress, потом выдается ошибка.

То как данные сервера обрабатывать и отображать пользователю уже второстепенно, данные мы успешно получили.

POST

Так же идем в документацию и смотрим, что нам у нас



Response headers

```
access-control-allow-origin: *
cache-control: no-cache
content-encoding: gzip
content-type: application/json; charset=utf-8
date: Wed,12 Apr 2023 04:20:52 GMT
etag: W/"82-yAAYpIwgOlHzT65AgSgWtZpeVcM"
expires: -1
status: 200 OK
x-content-type-options: nosniff
x-frame-options: SAMEORIGIN
x-ratelimit-limit: 60
x-ratelimit-remaining: 59
x-xss-protection: 1; mode=block
```

Request duration

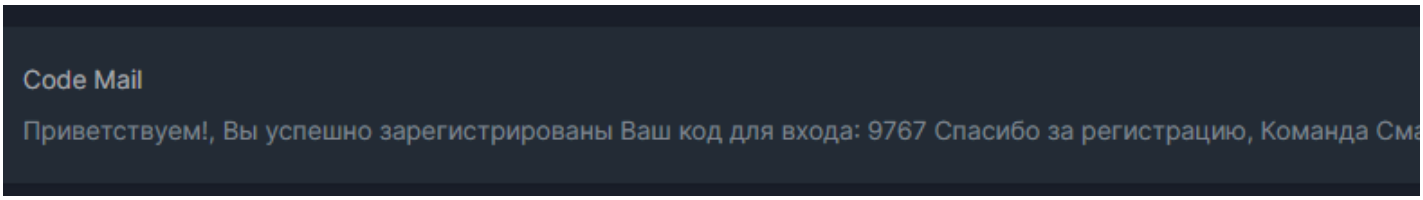
```
1988 ms
```

Responses

Code	Description
200	Успешно код отправлен Media type application/json Controls `Accept` header. Example Value | Schema `{ "message": "Успешно код отправлен" }`
422	Ошибка Media type application/json Example Value | Schema `{ "errors": "Текст ошибки" }`

В swagger очень удобно можно проверить работу api. Можно просто нажать на try it

out, ввести header и запрос отправиться.



Здесь все ещё проще. В заголовке мы отправляем email на который нужно отправить код, а в ответе получим одну строку. Есть только нюанс с ответом. Сошлепаем модельку

```
data class SendCodeResponseCloud(
    val message : String?,
    val errors : List<String>
)
```

В ApiService нам нужно описать метод для отправки кода авторизации

```
@POST("sendCode")
suspend fun sendAuthCode(
    @Header("email") email : String
) : SendCodeResponseCloud
```

В post запрос мы так же передаем название метода, а внутри параметров функции указываем аннотацию Header, именно её, а не Headers, она нужна для передачи другой информации. Т.е. в этот метод и будет приходить email который ввел пользователь

Далее напишем метод для отправки данных в CloudDataSource

```
//CloudDataSource
override suspend fun sendAuthCode(email: String):
NetworkResult<SendCodeResponseCloud> {
    return try {
        val request = apiService.sendAuthCode(email)
        NetworkResult.Success(request)
    } catch (e: Exception) {
        when (e) {
            is HttpException -> {
                val error = gson.fromJson(
                    e.response()?.errorBody()?.string(),
                    SendCodeResponseCloud::class.java
                )
                val errorMessage = error.errors[0]
                NetworkResult.Error(errorMessage)
```

```
            }
            else -> NetworkResult.Error(e.message)
        }
    }
}
```

Т.к. получить ответ в catch блоке мы не можем, то будем добывать его от туда через тело ответа из HttpException. Он приходит в Json формате, поэтому так же нужно будет его спарсить в нужный вид. Во всех остальных случаях будет бросаться стандартный exception. Так же есть варианты работы без try/catch. Но это не про сегодняшний случай.

Реализацию репозитория пока так же оставим

```
//ReposiotryImpl
override suspend fun sendEmail(email: String): NetworkResult<SendCodeResponseCloud>
{
    return cloudDataSource.sendAuthCode(email)
}
```

ViewModel аналогична

```
class AuthViewModel(
    private val repository: RepositoryImpl,
    private val authCommunication: AuthCommunication,
) : ViewModel(), BaseObserve<NetworkResult<SendCodeResponseCloud>> {
    override fun observe(
        owner: LifecycleOwner,
        observer: Observer<NetworkResult<SendCodeResponseCloud>>,
    ) {
        authCommunication.observe(owner, observer)
    }
    suspend fun sendAuthCode(email : String) {
        authCommunication.map(NetworkResult.Loading())
        val result = repository.sendEmail(email)
        authCommunication.map(result)
    }
}
interface AuthCommunication : Communication<NetworkResult<SendCodeResponseCloud>> {
    class Base() : Communication.Abstract<NetworkResult<SendCodeResponseCloud>>(),
AuthCommunication
}
```

AuthActivity

```
override fun getData(){
    lifecycleScope.launch {
        viewModel.sendAuthCode("ivanov@rogaikopyta.com")
    }
}
override fun observe(){
    viewModel.observe(this){
        when(it){
            is NetworkResult.Success -> {
                Log.d("market_twits",  it.toString() )}
            is NetworkResult.Error -> {
                Log.e("market_twits",   it.message.toString() )}
            is NetworkResult.Loading -> {
                Log.d("market_twits",  "loading") }
        }
    }
}
```

Замете, в activity нет больше никакой логики, она подписывается на livedata и просто прослушивает.

Запустим, проверим работу.

```
Success SendCodeResponseCloud(message=Успешно код отправлен, errors=null)
```

Отлично, теперь проверим остальные два сценария:

1. Когда у нас нет интернета
2. Когда данные на сервер пришли не валидные

Отключим интернет на устройстве и посмотрим в лог.

```
E/market_twits: Unable to resolve host "medic.madskill.ru": No address associated
with hostname
```

Отлично, ошибка прошена правильная
Отправим теперь не отвалидированные данные

```
override fun getData(){
    lifecycleScope.launch {
        viewModel.sendAuthCode("down this shit it hit me fast !")
    }
}
```

```
E/market_twits: The email must be a valid email address.
```

Отлично, основные кейсы прошли.

Вот мы получили данные с сервера. Что же с ними теперь делать ?
Да что угодно, в нашем случае мы выводили их в лог, в действительности их нужно отображать клиенту и реализовывать уже ui функционал. О нем как нибудь потом