

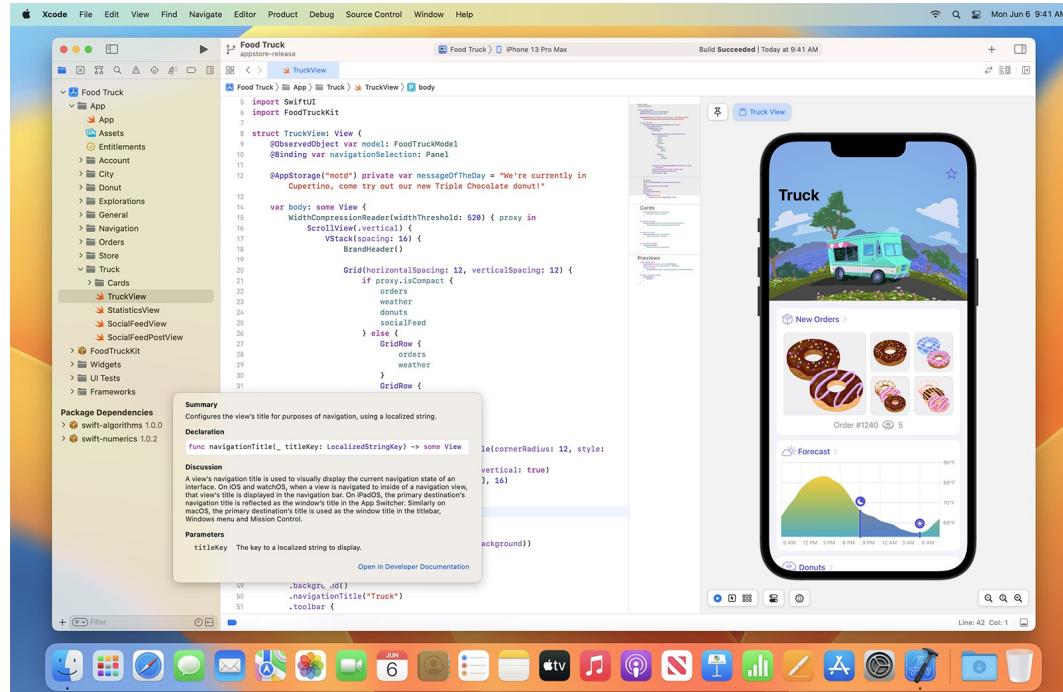
# Лекция 14

# Мобильные приложения

Проектирование систем и продуктовая веб-разработка

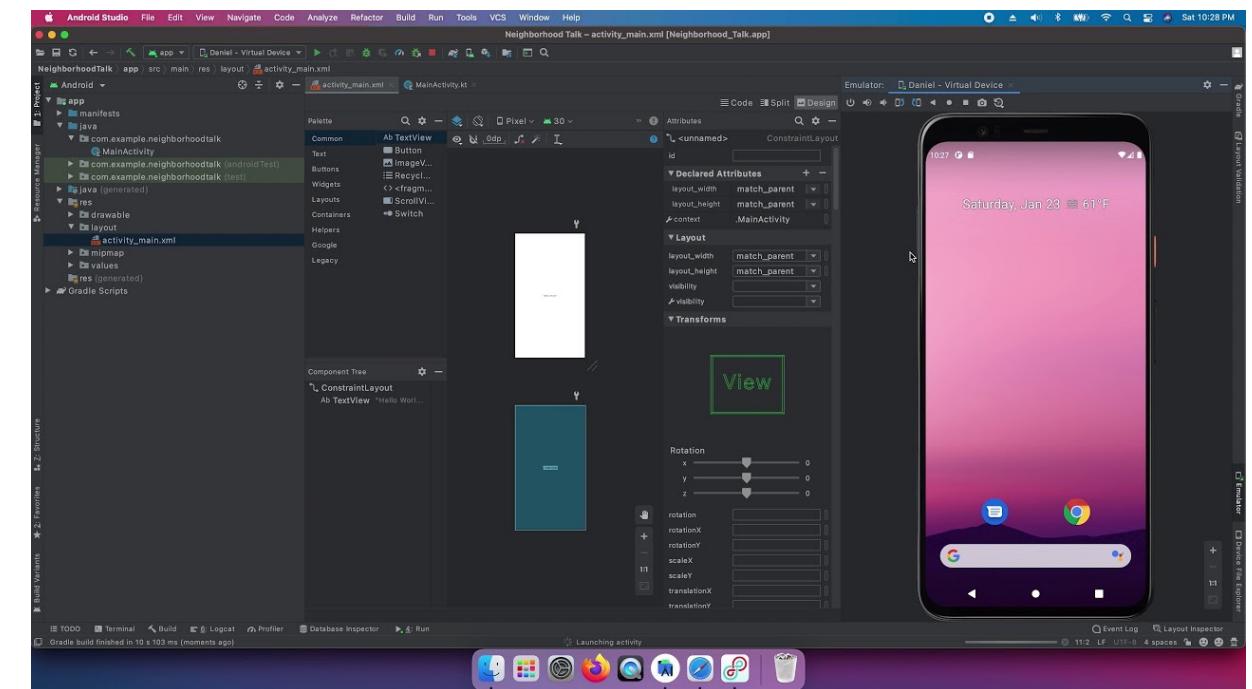
Канев Антон Игоревич

# Мобильные приложения



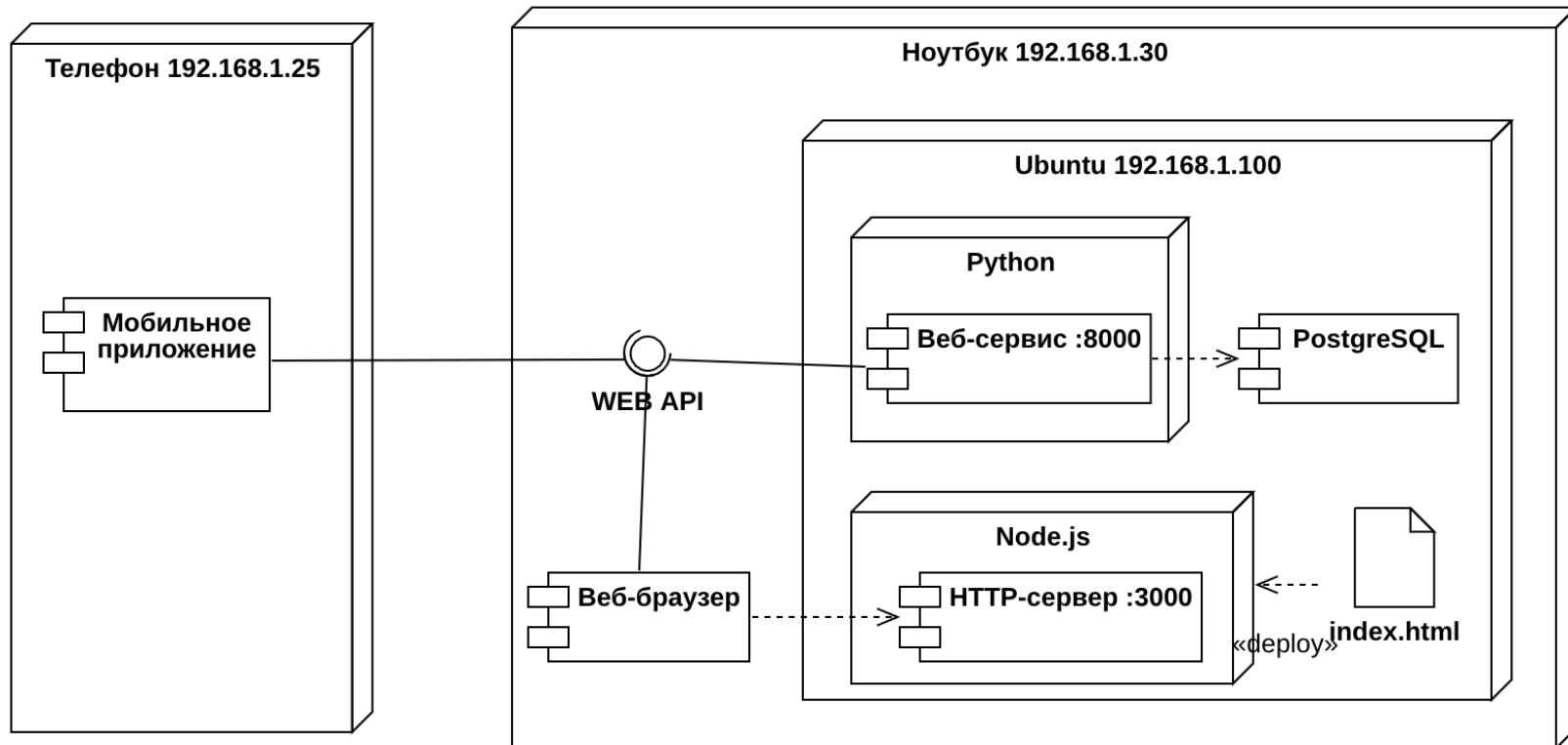
- iOS: среда разработки Xcode, язык программирования Swift

- Мобильное приложение — разновидность прикладного ПО для работы на смартфонах, планшетах и других мобильных устройствах
- Android: среда разработки Android Studio, язык программирования Kotlin



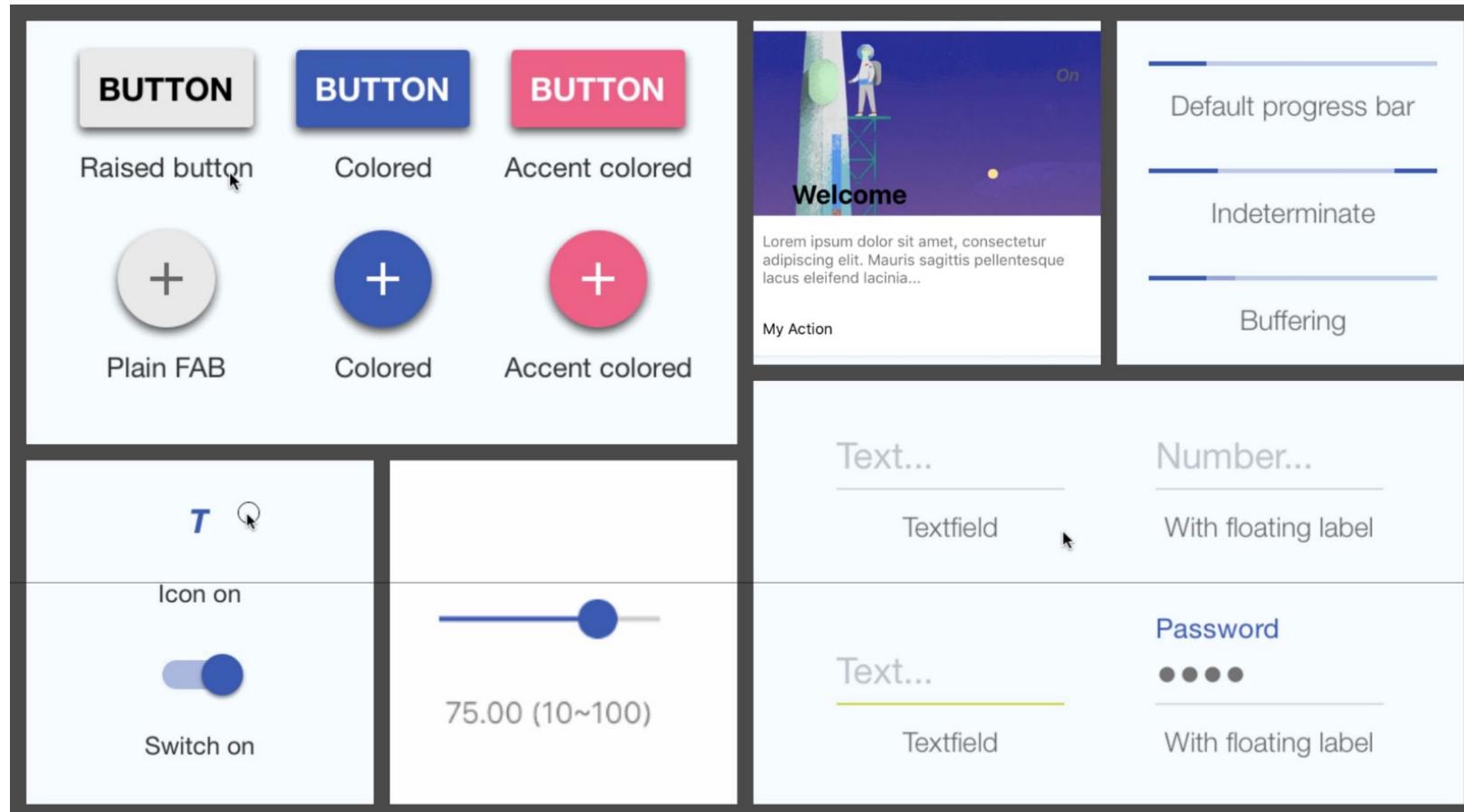
# Трехзвенная архитектура. API

- Наше десктопное, кроссплатформенное или мобильное приложение должно обращаться к разработанному нами API
- Использовать два запроса: GET списка услуг и GET одной услуги



# React Native

- React Native – фреймворк для кроссплатформенной разработки на JavaScript
- Позволяет создавать нативные приложения с помощью известных нам технологий: axios, redux-toolkit и тд



# React Native

- Мы можем вести разработку в VS Code и смотреть изменения в телефоне через QR
- Или в Android Studio и эмуляторе



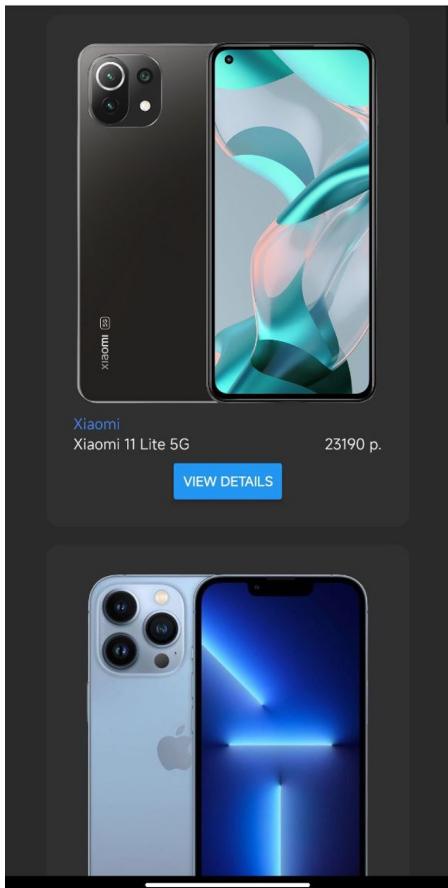
A screenshot of the Visual Studio Code interface. On the left, the Explorer sidebar shows a project structure for "SHOP MOBILE" containing files like .expo, .idea, assets, node\_modules, App.js, app.json, babel.config.js, package.json, and yarn.lock. The main editor area displays the following code in App.js:

```
You: 1 секунда назад | 1 автор (You)
1 import { StatusBar } from 'expo-status-bar';
2 import { StyleSheet, Text, View } from 'react-native';
3
4 export default function App() {
5   return (
6     <View style={styles.container}>
7       <Text>React Native</Text>
8       <StatusBar style='auto' />
9     </View>
10 }
11
12 const styles = StyleSheet.create({
13   container: {
14     flex: 1,
15     backgroundColor: '#fff',
16     alignItems: 'center',
17     justifyContent: 'center',
18   },
19 });
20
21 |
```

The bottom status bar shows "Сроки 21, создан 1 Протокол 4 UTF-8 LF {} JavaScript Go Live Colorize Variables Colorize Prettier". On the right, there is a mobile application emulator showing a white screen with a "Refreshing..." button at the bottom.

# React Native

- Создадим карточки и заполним их данными из API



```
export default function ShopScreen({ navigation }) {
  const dispatch = useDispatch();
  const { devices } = useSelector((store) => store.device);

  useEffect(() => {
    async function getAllDevices() {
      await axiosInstance.get('/device').then((response) => dispatch(setDevices(response?.data)));
    }
    getAllDevices();
  }, [dispatch]);

  return (
    <ScrollView>
      <View style={styles.page}>
        {!!devices &&
          devices.map((device) => <DeviceCard key={device.id} {...device} navigation={navigation} />)
        }
      </View>
    </ScrollView>
  );
}

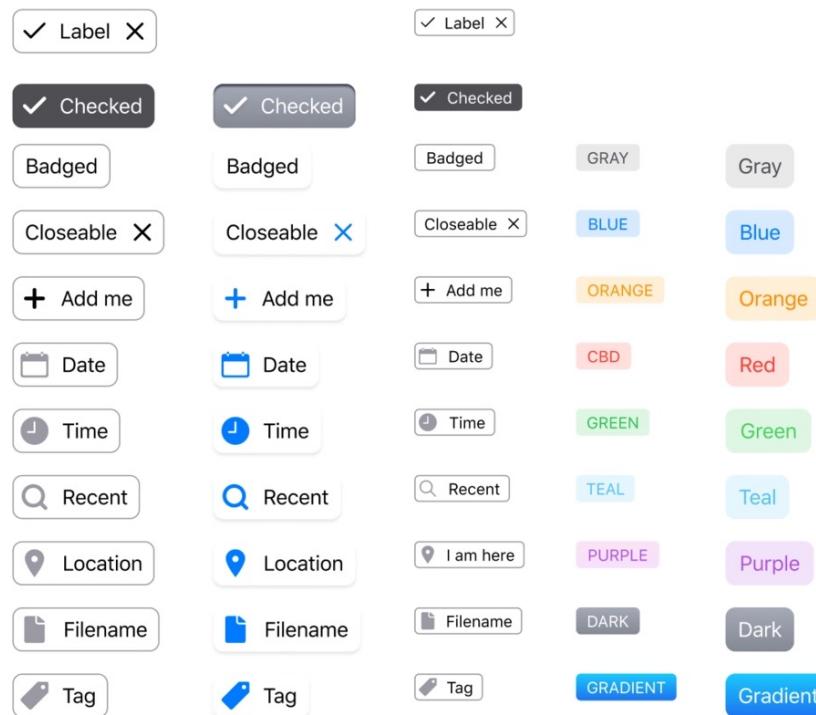
const styles = StyleSheet.create({
  page: {
    display: 'flex',
    width: '100%',
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#2a2a2a',
  },
});
```

# UI компоненты

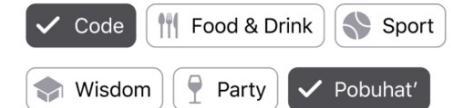
- Уже знакомый нам термин UI kit
- Использование кнопок, изображений, списков и других компонентов

iOS components

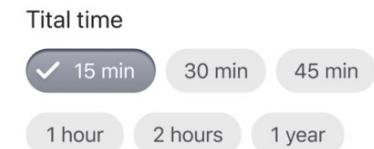
## Chips



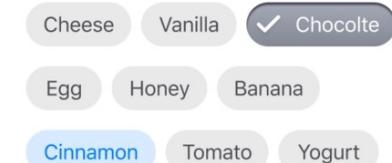
## What to do?



## Pick filters (2)



## Ingredient



## Summary keywords



# Верстка страницы с деталями

```
final class WeatherInfoViewController: UIViewController {
    //добавим на экран элементы, которые хотим отобразить на экране
    private let imageView = UIImageView()
    private let degreeLabel = UILabel()
    private let windLabel = UILabel()
    private let pressureLabel = UILabel()
    private let feelslikeLabel = UILabel()

    //создадим переменную для хранения детальной информации об объекте
    private var weatherData: WeatherData           //зададим базовые настройки для текстовых полей и добавим их на экран
                                                    //private func configureDataElements() {
        override func viewDidLoad() {
            super.viewDidLoad()
            configure()
            configureDataElements()
        }
    }

    //зададим базовые настройки для текстовых полей и добавим их на экран
    private func configureDataElements() {
        [degreeLabel, windLabel, pressureLabel, feelslikeLabel].forEach {
            $0.translatesAutoresizingMaskIntoConstraints = false
            $0.font = UIFont.systemFont(ofSize: 20, weight: .bold)
            $0.textColor = .white
            view.addSubview($0)
        }
    }

    //зададим констрайнты и базовые настройки для картинки
    imageView.translatesAutoresizingMaskIntoConstraints = false
    view.addSubview(imageView)
    imageView.heightAnchor.constraint(equalToConstant: 250).isActive = true
    imageView.widthAnchor.constraint(equalToConstant: 200).isActive = true
    imageView.leftAnchor.constraint(equalTo: view.leftAnchor, constant: 5).isActive = true
    imageView.topAnchor.constraint(equalTo: view.safeAreaLayoutGuide.topAnchor).isActive = true

    imageView.image = UIImage(named: "sunny")
}
```

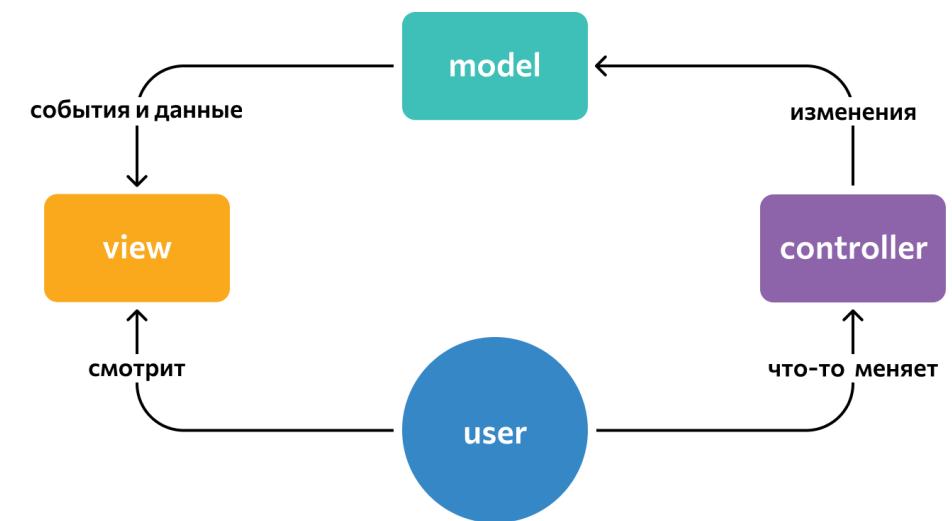
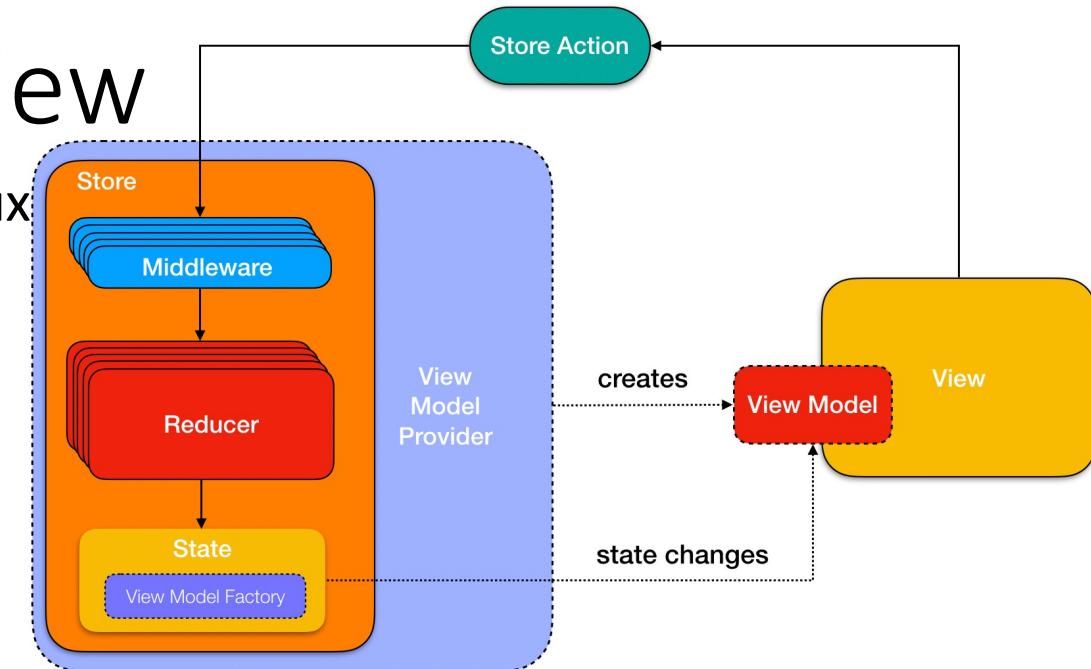
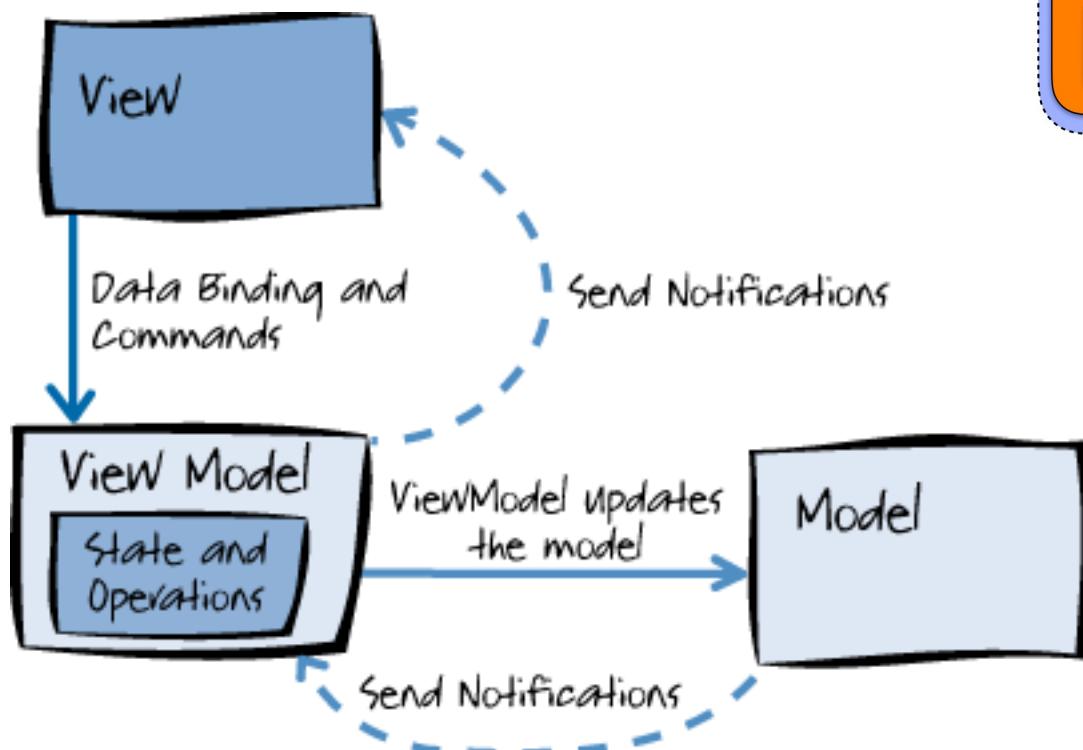


# Model View Model View

Паттерн MVVM уже знаком на примере Redux

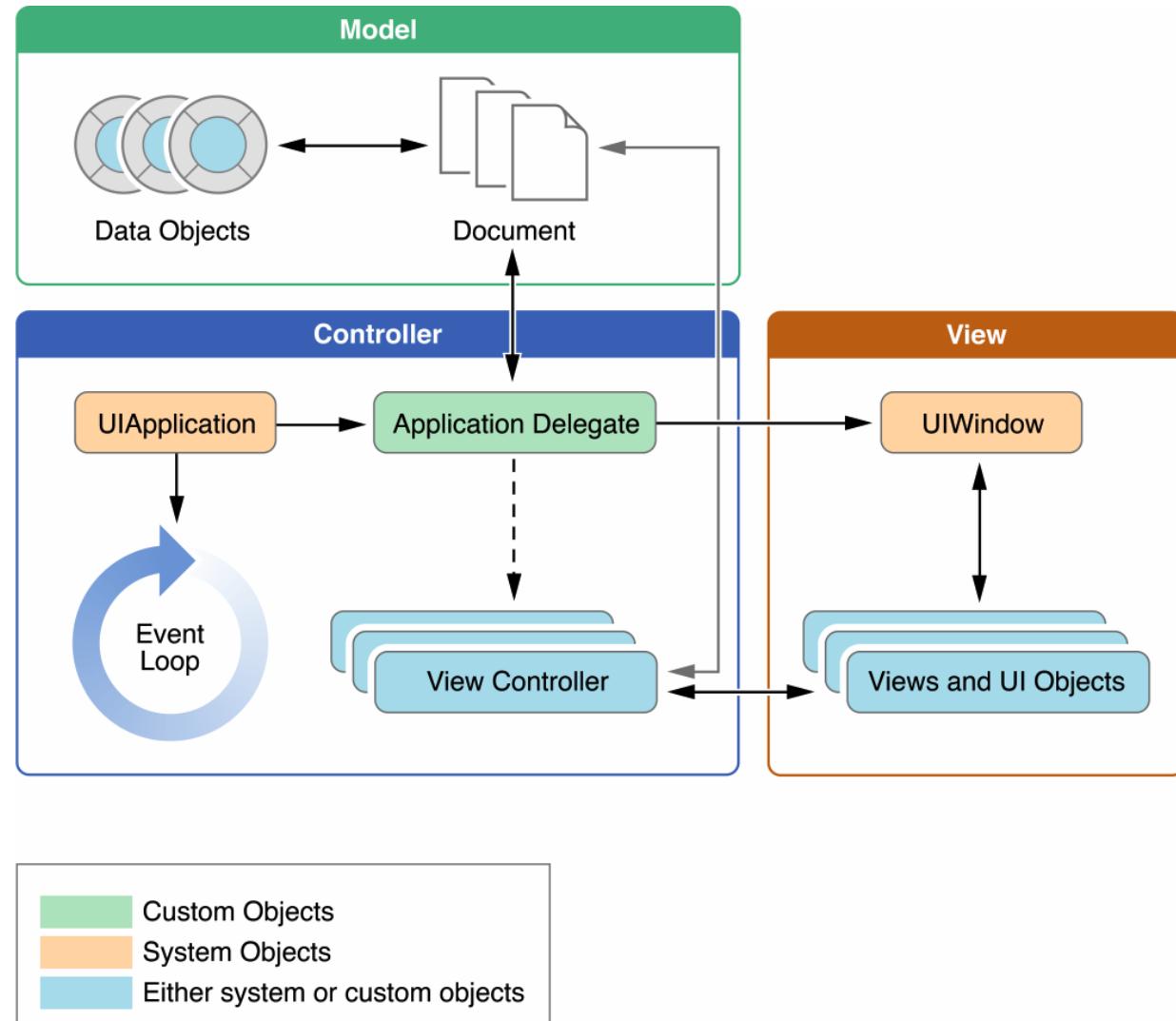
MVVM отличается от MVC, **ViewModel**

связывает в обе стороны Представление и полностью отделяет его от модели (API, LocalStorage)



# Архитектура Swift приложения

- **Модель** – отвечает за использование предметных (бизнес, domain) данных. Активные модели уведомляют окружающих об изменениях, а пассивные — нет.
- **Представление (View)** – отвечает за слой представления (GUI), но не обязательно связан с UI отрисовкой. Помимо представления данных пользователю, у него есть ещё задача - принять событие пользователя.
- **Контроллер/Презентер/ViewModel** – так или иначе отвечают за связь модели с представлением. В основном занимаются тем, что прорасыывают события модели в представление, а события представления – в модель, преобразуя и обрабатывая.



# Модель данных

- В данном пункте мы создадим модель данных, которая соответствует тому, что вы уже создали на бэкенде.
- В эту модель данных будет парситься json. Также мы создадим запрос к вашему сервису и сам парсинг ответа.
- Прежде чем приступать к созданию подключения сервиса необходимо задать модель с данными, которые придут в ответе от сервиса.

```
import Foundation

struct WeatherData: Codable {
    var location: Location
    var current: Current
}

struct Location: Codable {
    var name: String
    var country: String
    var region: String
}

struct Current: Codable {
    var observation_time: String
    var temperature: Int
    var wind_speed: Int
    var pressure: Int
    var feelslike: Int
}
```

# Генерация запроса

- Добавляем обращение в внешнему API
- В вашей лабораторной вы заменяете URL на ваш API
- Для эмулятора можно указать localhost
- Для показа IP в локальной сети, например 192.168.100.108

```
func configureURLRequest(city: String) -> URLRequest {  
    var request: URLRequest  
    let acsessToken: String = "b849bbbe085e655065bb8546ec2a8dd5" // нужен для weather-api  
  
    let queryItems = [  
        URLQueryItem(name: "access_key", value: acsessToken),  
        URLQueryItem(name: "query", value: "'\\"(city)'")  
    ]  
    guard var urlComponents = URLComponents(string: "http://api.weatherstack.com/current") else {  
        // если не получится создать компоненты из своих query параметров, то переходим на google  
        return URLRequest(url: URL(string: "https://google.com")!)  
    }  
  
    urlComponents.queryItems = queryItems  
  
    guard let url = urlComponents.url else {  
        // если не получится создать url из своего адреса, то переходим на google  
        return URLRequest(url: URL(string: "https://google.com")!)  
    }  
  
    request = URLRequest(url: url)  
    request.httpMethod = ApiMethods.post.rawValue // устанавливаем метод запроса через enum  
    return request  
}
```

# Запросы к API

- Создание обработчиков запросов к собственному API сервису в отдельном файле
- Указываем в какие переменные мы должны положить полученные файлы

```
import Foundation

final class ApiService {

    func getWeatherData(city: String, completion: @escaping (WeatherData?, Error?) -> ()) {
        let request = configureURLRequest(city: city) // конфигурация кастомного запроса

        URLSession.shared.dataTask(with: request, completionHandler: { data, response, error in // completionHandler

            if let error = error {
                print("error")
                completion(nil, error)
            }
            if let response = response {
                print(response)
            }
            guard let data = data else {
                completion(nil, error)
                return
            }

            do {
                let weatherData = try JSONDecoder().decode(WeatherData.self, from: data) // декодируем json в созданную структуру
                completion(weatherData, nil)
            } catch let error {
                completion(nil, error)
            }
        }).resume() // запускаем задачу
    }
}
```

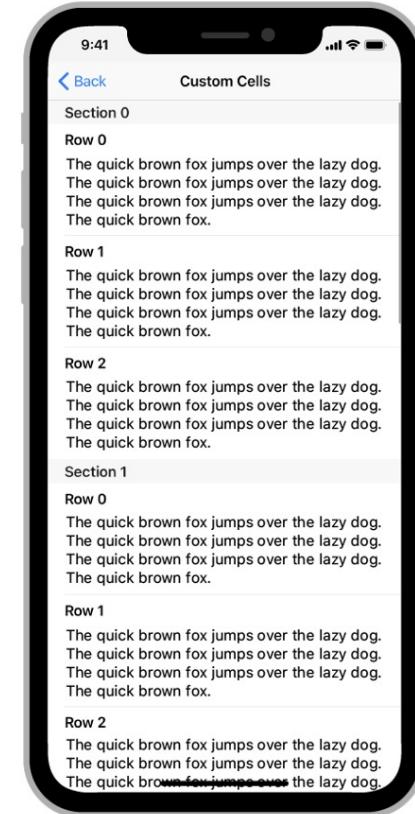
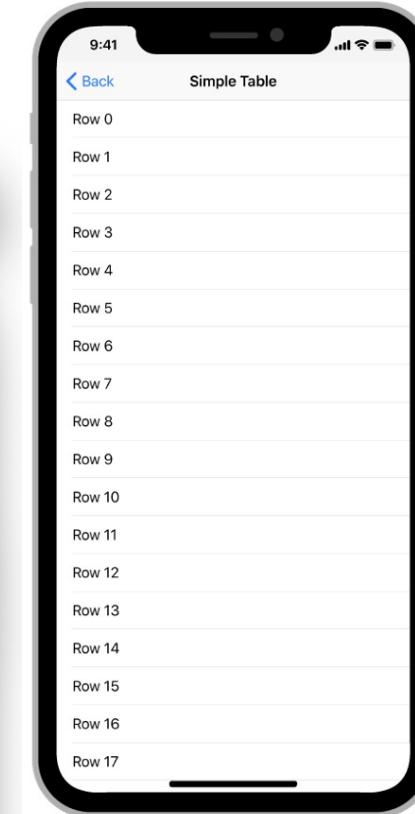
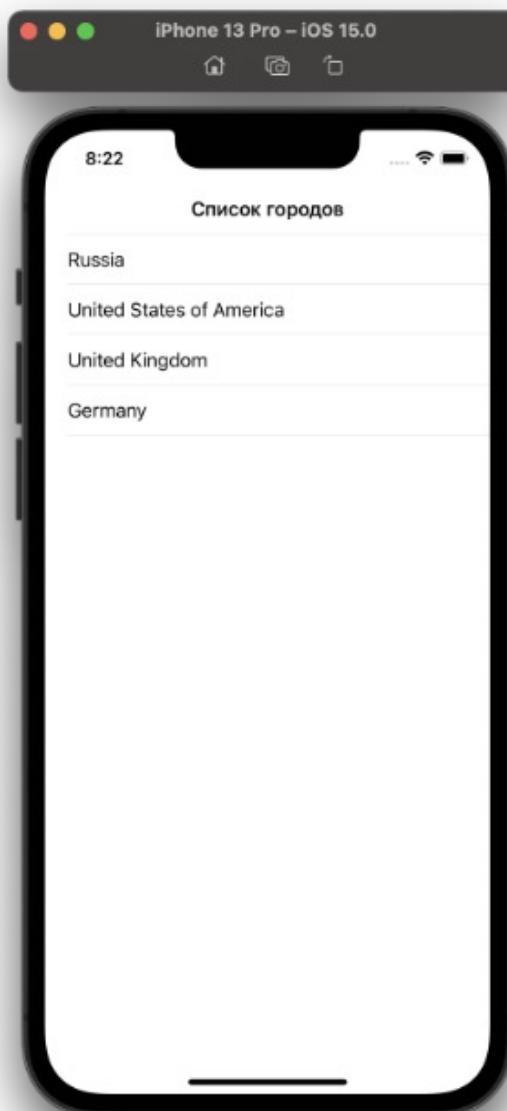
# Заполнение страницы данными

```
private func loadWeatherData(cities: [String]) {
    guard let apiService = apiService else { // раскрытие опциональной переменной apiService
        return
    }

    cities.forEach {
        apiService.getWeatherData(city: $0, completion: { [weak self] (weatherData, error) in // weak self для
            DispatchQueue.main.async { // запуск асинхронной задачи на main потоке из-за обработки на ui !!!
                guard let self = self else { return }
                if let error = error {
                    // показ ошибки
                    self.present(UIAlertController(title: "ERROR", message: error.localizedDescription, preferredStyle: .alert))
                    return
                }
                if let weatherData = weatherData {
                    self.weatherListData.append(weatherData) // массив с данными о погоде
                }
                self.weatherListTableView.reloadData() // перезагрузка таблицы для отображения новых данных
            }
        })
    }
}
```

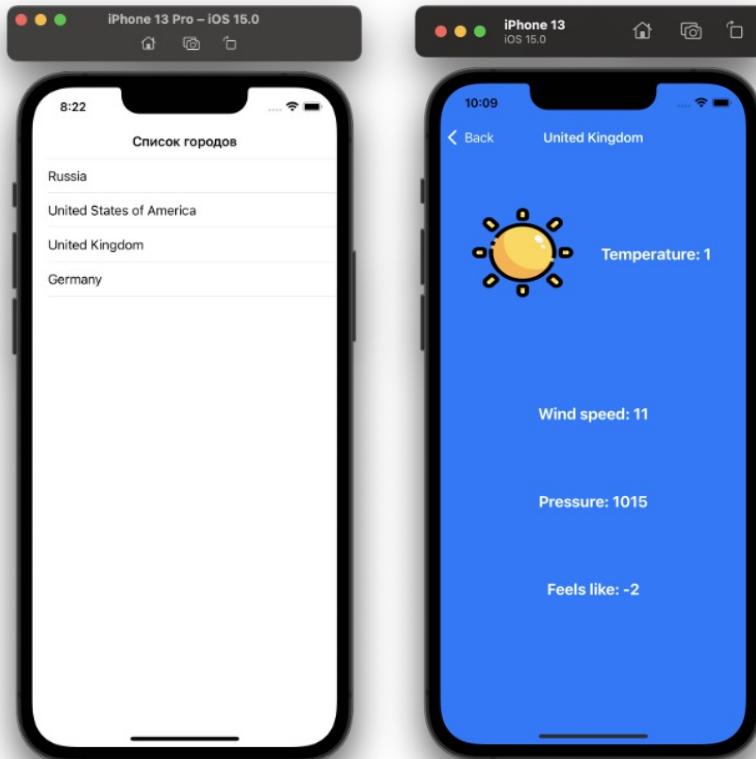
# UITableView

- Для того, чтобы на экране отобразилась таблица, необходимо создать переменную класса `WeatherViewController` типа `UITableView` и задать там первичные настройки



# Переходы между страницами

- Далее необходимо добавить переход на данный экран с основного



```
import Foundation
import UIKit

final class WeatherInfoViewController: UIViewController {
    private var weatherData: WeatherData

    override func viewDidLoad() {
        super.viewDidLoad()
    }

    init(weatherData: WeatherData) {
        self.weatherData = weatherData
        super.init(nibName: nil, bundle: nil)
    }
}
```

```
func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath) {
    let weatherInfoViewController = WeatherInfoViewController(weatherData: self.weatherListData[indexPath.row])
    navigationController?.pushViewController(weatherInfoViewController, animated: true)
}
```

# Заполнение детальной информации

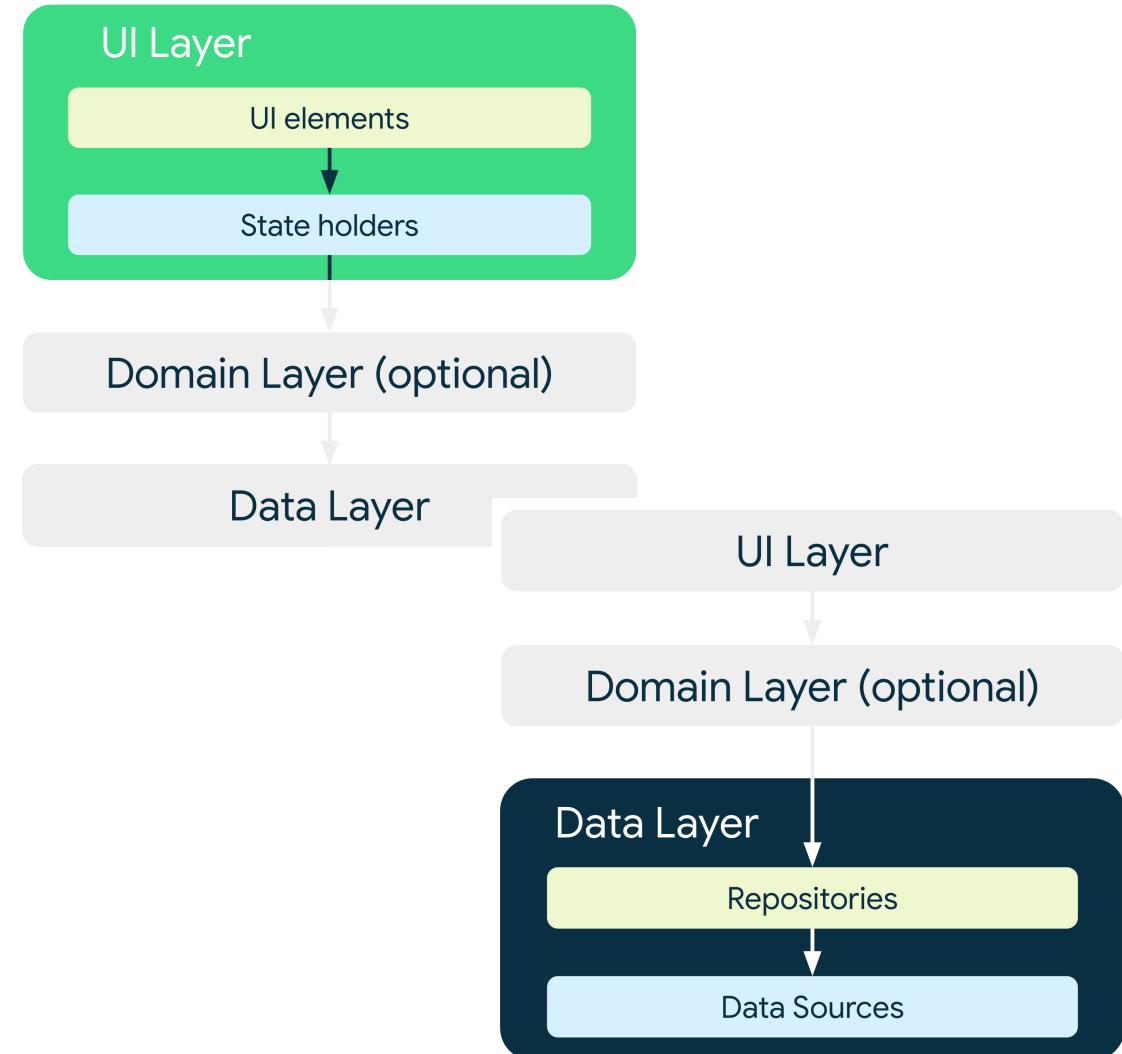
- Создадим функцию, которая будет сохранять в текстовые лейблы значения строк с детальной информацией об объекте, которые мы передали с первого экрана.
- которая вызывается из инициализатора контроллера

```
func fillData(withModel: WeatherData) {  
    degreeLabel.text = "Temperature: " + String(withModel.current.temperature)  
    windLabel.text = "Wind speed: " + String(withModel.current.wind_speed)  
    pressureLabel.text = "Pressure: " + String(withModel.current.pressure)  
    feelslikeLabel.text = "Feels like: " + String(withModel.current.feelslike)  
}
```

```
private var weatherData: WeatherData  
  
init(weatherData: WeatherData) {  
    self.weatherData = weatherData  
    super.init(nibName: nil, bundle: nil)  
    fillData(withModel: weatherData)  
}
```

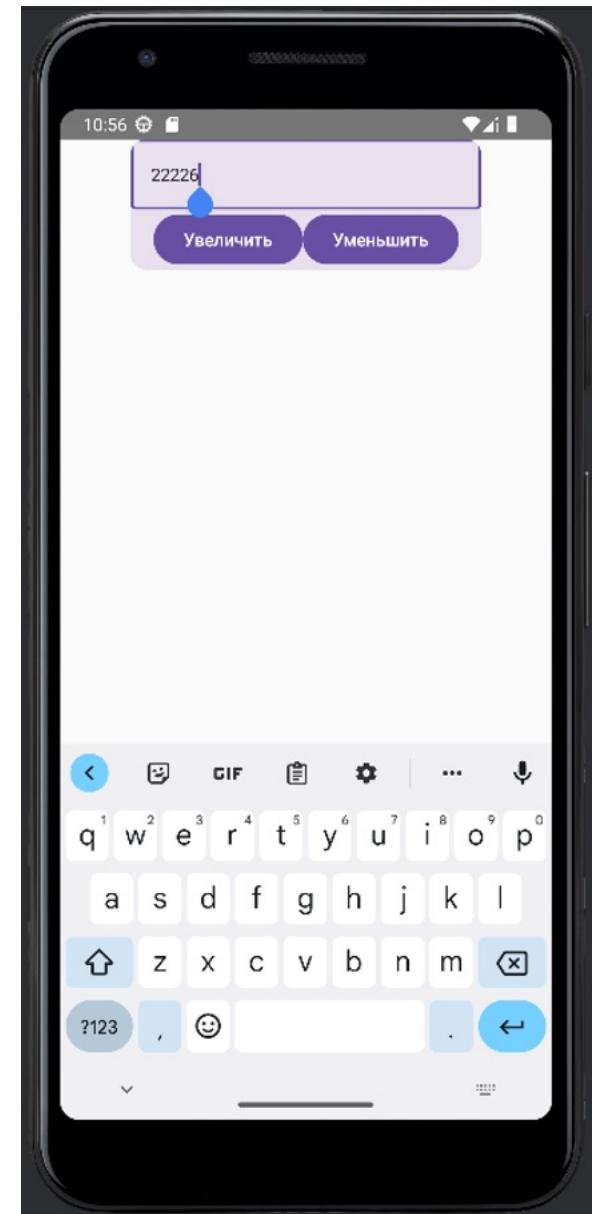
# Архитектура Android приложения

- Взглянем с другой стороны и как в чистой архитектуре выделим в модели дополнительный **слой бизнес-логики**
- **Роль слоя UI (или слоя представления)** — отображать на экране данные приложения.
- **Слой данных** в приложении содержит *бизнес-логику* — правила, по которым приложение создаёт, хранит и изменяет данные.
- **Доменный слой** располагается между слоями UI и данных. Доменний слой отвечает за инкапсуляцию сложной бизнес-логики или простой бизнес-логики, которую переиспользуют несколько ViewModel.



# Счетчик в Android Studio

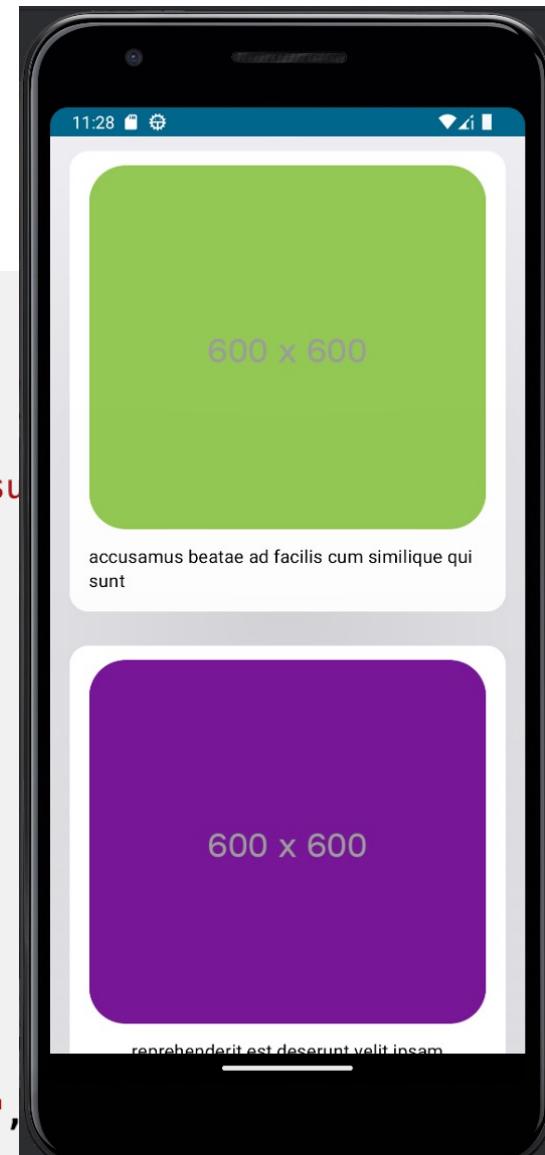
```
@Composable
fun Counter(){
    var counterValue by remember {
        mutableIntStateOf(0)
    }
    Card {
        Column(horizontalAlignment = Alignment.CenterHorizontally) {
            OutlinedTextField(value = counterValue.toString(), onValueChange =
{counterValue = it.toInt()})
            Row {
                Button(onClick = {counterValue+=1}) {
                    Text(text = "Увеличить")
                }
                Button(onClick = {counterValue-=1}) {
                    Text(text = "Уменьшить")
                }
            }
        }
    }
}
```



# Окно списка

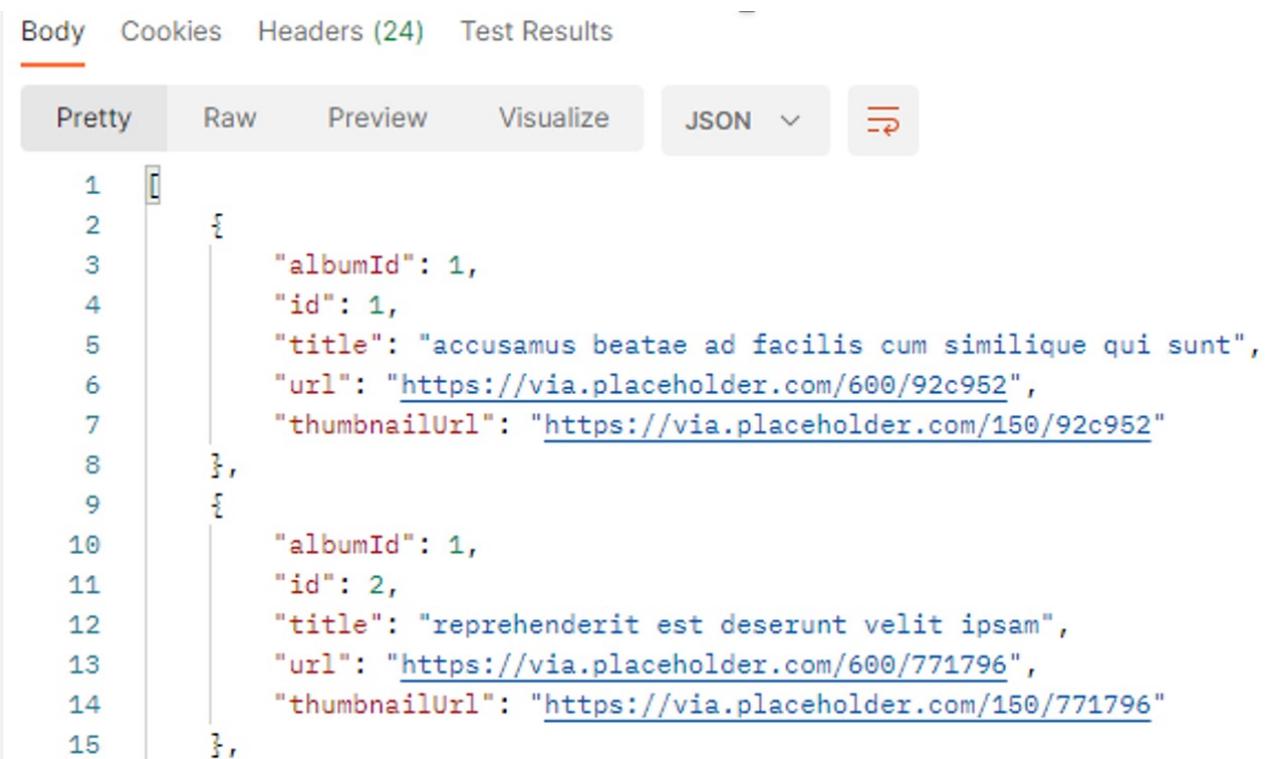
- Создаем карточки на основе мок
- Помещаем данные из модели в компоненты нашего экрана

```
val photos = listOf<Photo>(  
    Photo(  
        albumId = 1,  
        id = 1,  
        title = "accusamus beatae ad facilis cum similique qui su",  
        url = "https://via.placeholder.com/600/92c952",  
        thumbnailUrl ="https://via.placeholder.com/150/92c952"  
    ),  
    Photo(  
        albumId = 1,  
        id = 2,  
        title = " reprehenderit est deserunt velit ipsam",  
        url = "https://via.placeholder.com/600/771796",  
        thumbnailUrl ="https://via.placeholder.com/150/771796"  
    ),  
    Photo(  
        albumId = 1,  
        id = 3,  
        title = " officia porro iure quia iusto qui ipsa ut modi",  
        url = "https://via.placeholder.com/600/24f355",  
        thumbnailUrl ="https://via.placeholder.com/150/24f355"  
    )  
)
```



# Модель для полученных данных

- Сделаем структуру – модель для полученных от API данных



The screenshot shows the 'Body' tab of the Postman interface with a JSON response. The response is a list of two objects, each representing a photo. The schema includes fields: albumId, id, title, url, and thumbnailUrl. The 'url' and 'thumbnailUrl' fields contain placeholder URLs.

```
1  [
2  {
3      "albumId": 1,
4      "id": 1,
5      "title": "accusamus beatae ad facilis cum similique qui sunt",
6      "url": "https://via.placeholder.com/600/92c952",
7      "thumbnailUrl": "https://via.placeholder.com/150/92c952"
8  },
9  {
10     "albumId": 1,
11     "id": 2,
12     "title": " reprehenderit est deserunt velit ipsam",
13     "url": "https://via.placeholder.com/600/771796",
14     "thumbnailUrl": "https://via.placeholder.com/150/771796"
15 }
```

```
// Добавляем необходимые импорты, используя `Alt +  
import com.google.gson.annotations.SerializedName  
  
// Класс `Photo` представляет собой модель данных  
// соответствующие полям в JSON-структуре.  
  
data class Photo(  
    @SerializedName("albumId")  
    val albumId: Int,  
    @SerializedName("id")  
    val id: Int,  
    @SerializedName("title")  
    val title: String,  
    @SerializedName("url")  
    val url: String,  
    @SerializedName("thumbnailUrl")  
    val thumbnailUrl: String  
)
```

# Обращение к API

- Используется REST клиент для Android Retrofit для выполнения HTTP запросов к API
- Для эмулятора можно указать localhost
- Для показа IP в локальной сети, например 192.168.100.108

```
interface CodelabApi {  
  
    @GET("photos")  
    suspend fun getAllPhotos(): List<Photo>  
  
    companion object RetrofitBuilder{  
        private const val BASE_URL = "https://jsonplaceholder.typicode.com/"  
  
        private fun getRetrofit(): Retrofit {  
            return Retrofit.Builder()  
                .baseUrl(BASE_URL)  
                .addConverterFactory(GsonConverterFactory.create())  
                .build()  
        }  
        val api: CodelabApi = getRetrofit().create()  
    }  
}
```

# Окно детализации и поиск

- Поиск нужно переделать на бэкенд: мобильное приложение должно работать вместе с веб-сервисом
- Также необходимо добавить второй экран с детальной информацией

```
//Добавляем поле для ввода
OutlinedTextField(
    value = filterText,
    textStyle = TextStyle(
        fontSize = 16.sp,
        lineHeight = 20.sp,
        fontWeight = FontWeight(400),
        color = Color(0xFF818C99),
        letterSpacing = 0.1.sp,
    ),
    //При изменении состояния поля (ввод символов), ищем карточки
onValueChange = {
    filterText = it
    scope.launch {
        filteredPhotos = photos.filter { photo ->
            photo.title.contains(filterText)
                || photo.id.toString()
                    .contains(filterText) || photo.albumId.toString()
                    .contains(filterText)
    }
}
```

