



Урок 8

Backend — Firebase

Используем внешний сервер для хранения данных приложения. Осваиваем Firebase Framework. Подключаем механизмы «Регистрация» и «Авторизация пользователей».

[Frontend и backend](#)

[Firebase](#)

[База данных для realtime-приложений](#)

[Firebase Hosting](#)

[Simple Login](#)

[Cloud Messaging](#)

[Firebase Analytics](#)

[Firebase Storage](#)

[Firebase Crash Reporting](#)

[Firebase Notifications](#)

[Firebase Remote Config](#)

[Firebase App Indexing](#)

[Firebase ML Kit](#)

[Firebase Crashlytics](#)

[Firebase AdMob](#)

[Практическая часть](#)

[Подключаем Firebase](#)

[Аутентификация](#)

[Использование Firebase](#)

[Практическое задание](#)

[Дополнительные материалы](#)

[Используемая литература](#)

Frontend и backend

Frontend и backend — это термины в программной инженерии, которые отражают принцип разделения ответственности между внешним представлением и внутренней реализацией. Frontend — интерфейс взаимодействия между пользователем и основной программно-аппаратной частью (backend).

Например, в проектировании ПО архитектурный паттерн Model-View-Controller обеспечивает frontend и backend между базой данных, компонентами обработки данных и пользователями.

Разделение между программными системами на frontend и backend упрощает разработку. Front-сторона (или клиент) — любой компонент, управляемый пользователем. Back-сторона выполняется на сервере. Затруднения возникают, когда необходимо применить frontend-изменения к файлам на стороне сервера.

Firestore

Firestore — поставщик облачных сервисов и приложений. В октябре 2014 года компания была приобретена Google.

База данных для realtime-приложений

Это основное направление Firestore. Предоставляет облачную NoSQL БД для realtime-приложений как сервис. Благодаря его API разработчики обеспечивают синхронизацию данных приложения между клиентами и хранение информации в облаке Firestore. Сервис интегрирован с Android, iOS, JavaScript, Java, Objective-C, Swift и Node.js-приложениями.

Firestore Hosting

Firestore Hosting — хостинг-сервис, поддерживающий хранение статических файлов: CSS, HTML, JavaScript. Файлы передаются по Content Delivery Network (CDN) с использованием SSL.

Simple Login

Firestore Simple Login — сервис, позволяющий аутентифицировать пользователей, используя код только на стороне клиента (client-side code). Поддерживается вход в Facebook, GitHub, Twitter и Google. Дополнительно разработчики могут аутентифицировать пользователей по данным из Firestore-облака.

Cloud Messaging

Firestore Cloud Messaging (FCM) — это кроссплатформенное решение для Android, iOS и web, которое доставляет сообщения и уведомления надежно и без затрат. Ранее было известно как Google Cloud Messaging (GCM).

Firestore Analytics

Firestore Analytics — бесплатное решение для анализа статистики. Оно дает представление об использовании приложений и взаимодействии с пользователем.

Analytics интегрируется через Firestore и анализирует до 500 событий, предоставляя неограниченную отчетность. События можно определить с помощью Firestore SDK. Благодаря Firestore Analytics можно

делать выводы о поведении пользователей, принимать обоснованные маркетинговые решения и оптимизировать производительность.

Firestore

Firestore предназначено для разработчиков приложений, которым необходимо хранить и обслуживать пользовательский контент: например, фотографии, аудио или видео.

Хранилище увеличивает безопасность Google при загрузке файлов и Firestore-приложений независимо от качества сети. Оно поддерживается Google Cloud Storage и является мощным, простым и экономически эффективным сервисом.

Firebase Crash Reporting

Firebase Crash Reporting — всесторонний инструмент для диагностики и устранения проблем в приложении. Создает подробные отчеты по ошибкам, объединяя их в группы и сортируя по критичности воздействия на приложение. В дополнение к автоматическим отчетам можно отслеживать действия пользователя, которые привели к «падению» приложения.

Firebase Notifications

Firebase Notifications — бесплатный сервис для создания пользовательских уведомлений для мобильных приложений.

Он необходим разработчикам и организациям, нуждающимся в гибкой платформе уведомлений. Имеет графическую консоль для отправки сообщений, а для начала работы требует минимального программирования. С помощью консоли можно создать и сохранить собственные правила для рассылки сообщений пользователям системы.

Firebase Remote Config

Firebase Remote Config — облачный сервис, позволяющий изменять поведение и внешний вид приложения, не требуя от пользователя загрузки обновлений. Значения для контроля приложения создаются разработчиком по умолчанию, а затем переопределяются через Firestore-консоли. Изменения можно направить на всех пользователей приложения или отдельные сегменты. Приложение проверяет необходимость обновления и применяет новые настройки с незначительным влиянием на производительность.

Firebase App Indexing

Firebase App Indexing добавляет приложение в Google-поиск. Если оно уже установлено и пользователи ищут что-то в связи с его содержанием, приложение запустится непосредственно из результатов поиска. Если приложение еще не установлено, поиск выдаст ссылку на установку.

Firebase ML Kit

Недавно добавлен в Firestore ML Kit. Предоставляет готовые решения для реализации типичных задач машинного обучения: распознавания текста, определения лиц на фотографии, сканирования штрих-кодов. Позволяет загружать и использовать свои ML-модели в приложении.

Firebase Crashlytics

Crashlytics, приобретенный Google у компании Fabric, позволяет легко и быстро собирать информацию (дампы) о крашах приложения на устройствах пользователей. Отслеживает

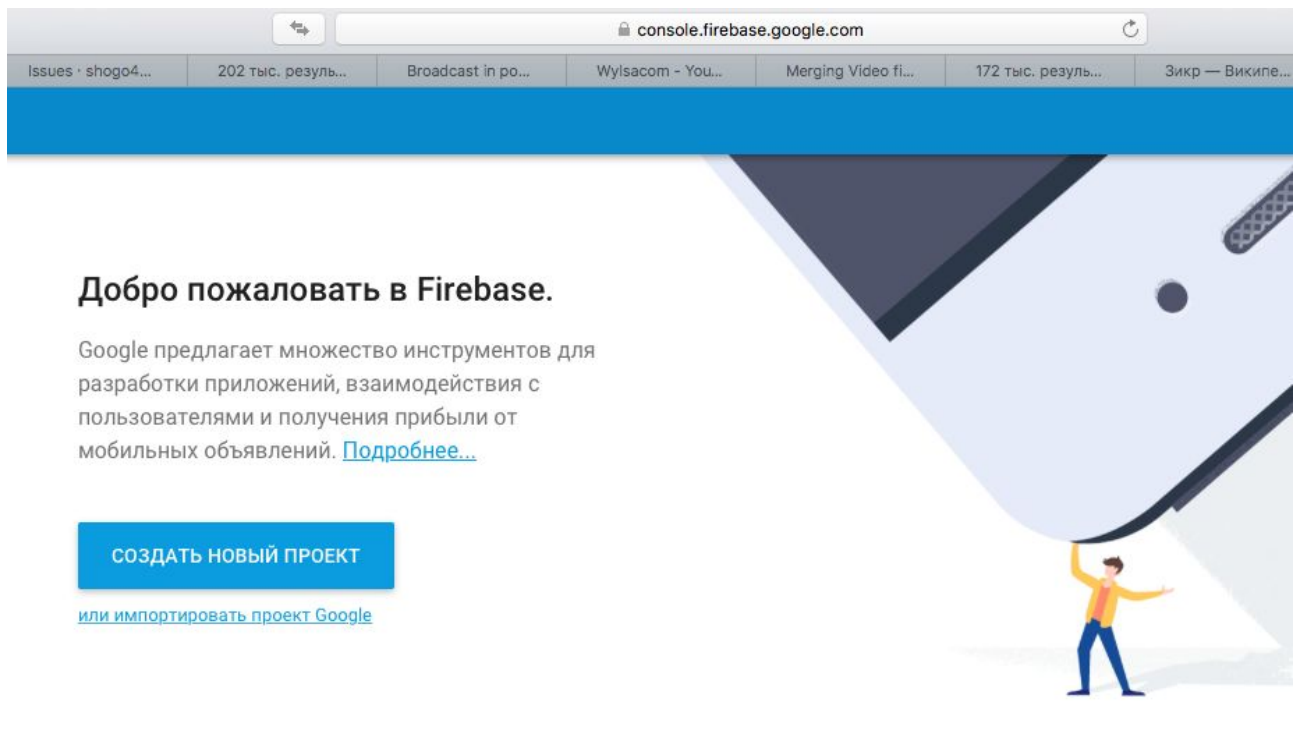
стабильность работы в процентном соотношении в зависимости от модели устройства, версии операционной системы и других факторов.

Firebase AdMob

Firebase AdMob позволяет интегрировать в приложение таргетированную рекламу от Google. Это простой и популярный способ монетизировать небольшие стартапы.

Практическая часть

Подключаем Firebase



По адресу <https://console.firebase.google.com/> попадаем в консоль управления сервисами Firebase. Сейчас она почти пустая, так как у нас еще нет проектов. Создадим первый — нажимаем кнопку «Создать новый проект» и заполняем форму:

Создать проект

Название проекта

Weather

Страна/регион ?

Россия

Данные Firebase Analytics нужны, чтобы сделать другие сервисы Firebase и Google лучше. Проверить, к каким именно сведениям Firebase Analytics есть доступ у других сервисов, можно в любое время. [Подробнее...](#)

Нажимая кнопку ниже, вы соглашаетесь использовать сервисы Firebase в своем приложении и принимаете действующие [Условия использования](#).

ОТМЕНА

СОЗДАТЬ ПРОЕКТ

Затем попадаем в консоль администрирования сервисов для проекта. Добавим в него приложение. Для этого нажимаем кнопку:



И заполняем форму:

Добавление Firebase в приложение для iOS

1

2

3

4

Введите данные приложения

Скопируйте файл конфигурации


Установите пакет

Добавьте код инициализации

Идентификатор пакета iOS ?

home.WeatherV2

Псевдоним приложения (необязательно) ?

Weather 

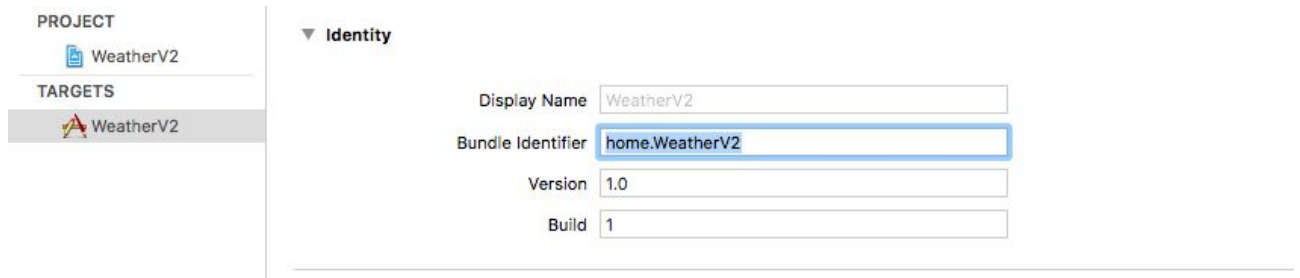
Идентификатор App Store (необязательно) ?

123456789

ОТМЕНА

ДОБАВИТЬ ПРИЛОЖЕНИЕ

На первом шаге заполняем два обязательных поля: «Идентификатор проекта iOS» и «Псевдоним приложения». Псевдоним можно придумать любой, а идентификатор должен строго совпадать с тем, что указан у вас в проекте:



После этого можно переходить ко второму шагу.

Добавление Firebase в приложение для iOS

1

2

3

4


Введите данные приложения

Скопируйте файл конфигурации

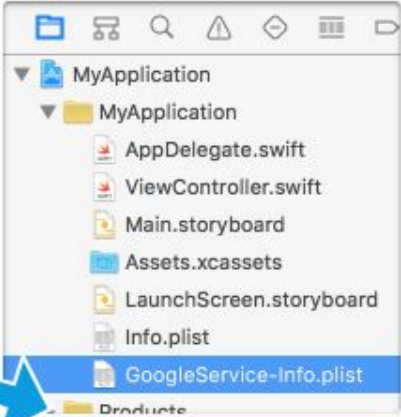
Установите пакет

Добавьте код инициализации

Переместите скаченный файл **GoogleService-Info.plist** в корневой каталог проекта Xcode и добавьте его во все целевые объекты.



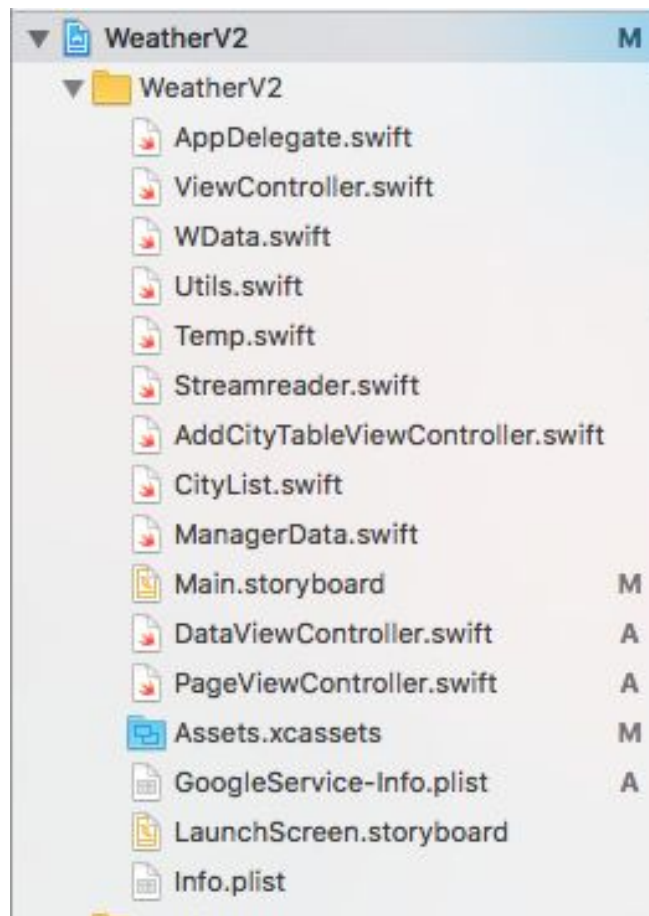
GoogleService-Info.plist



Уже создали пакет и добавили код инициализации?
[Заккрыть](#)

ПРОДОЛЖИТЬ

Загружаем конфигурационный файл и добавляем его в проект:



Переходим к третьему шагу.

Добавление Firebase в приложение для iOS

1

2

3

4

Введите данные приложения

Скопируйте файл конфигурации

Установите пакет

Добавьте код инициализации

Для установки зависимостей и управления ими в сервисах Google используется [CocoaPods](#). Откройте окно терминала и перейдите в каталог проекта Xcode для вашего приложения.

1. Создайте файл Podfile, если у вас его ещё нет:

\$ pod init

2. Добавьте в него следующий текст:

```
pod 'Firebase/Core'
```

сервис *Firebase Analytics* включен по умолчанию

?

3. Сохраните файл и выполните следующую команду:

\$ pod install

Для вашего приложения будет создан файл `.xcworkspace`, который нужно использовать при дальнейшей разработке.

Уже создали пакет и добавили код инициализации?
[Заккрыть](#)

продолжить

Устанавливаем фреймворк для работы с Firebase из репозитория CocoaPods. Этот процесс мы разбирали в первом курсе. Краткие инструкции есть на самом экране третьего шага.

Переходим к четвертому:

Добавление Firebase в приложение для iOS

1

2

3

4

Введите данные приложения

Скопируйте файл конфигурации

Установите пакет

Добавьте код инициализации

Чтобы приложение при запуске подключалось к Firebase, добавьте указанный ниже код инициализации в основной класс AppDelegate.

☒ Swift ☐ Objective C

```
import UIKit
import Firebase

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {

    var window: UIWindow?

    func application(application: UIApplication,
        didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?)
        -> Bool {
        FIRApp.configure()
        return true
    }
}
```

ГОТОВО

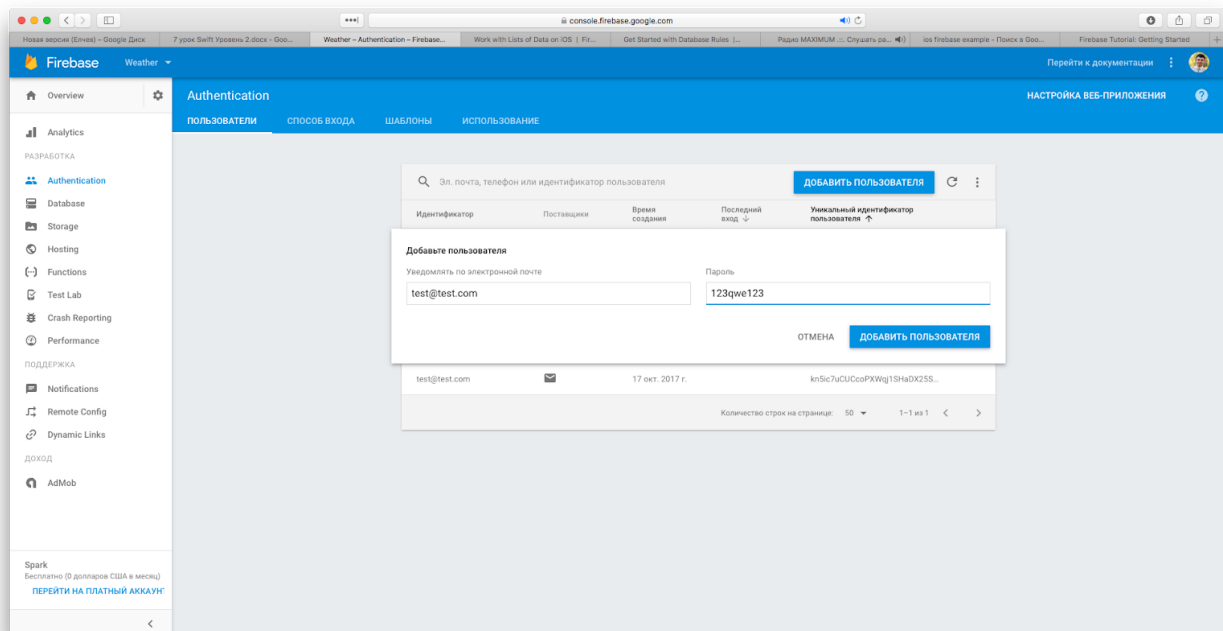
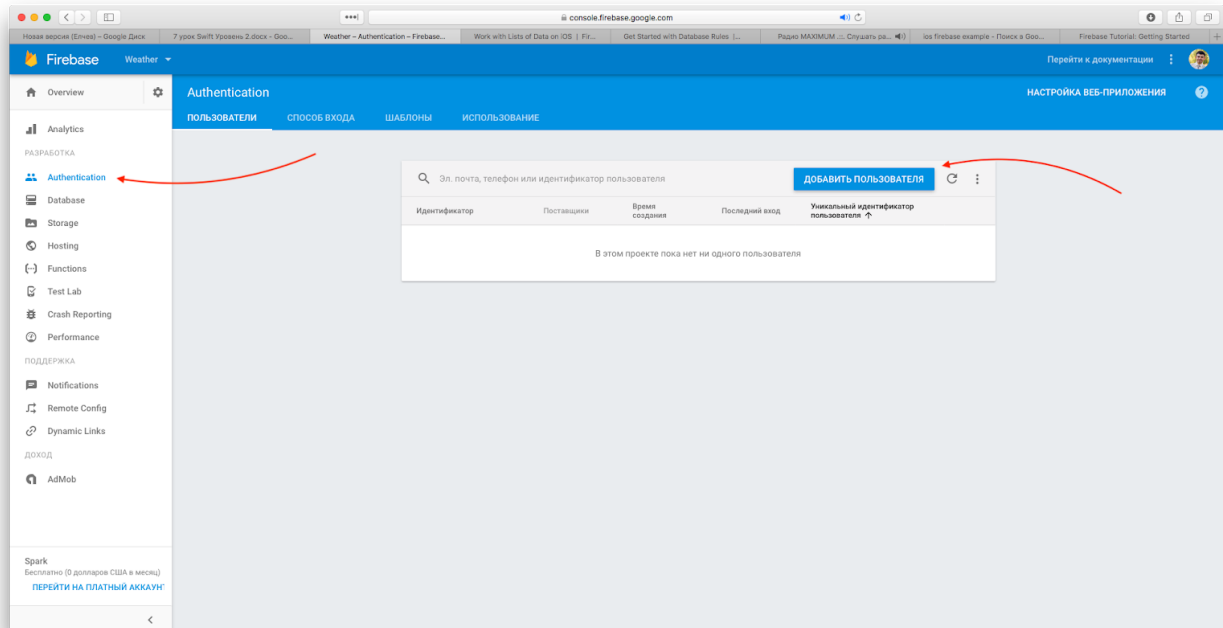
Видим код, который необходимо добавить в файл **AppDelegate.swift** для начальной конфигурации.

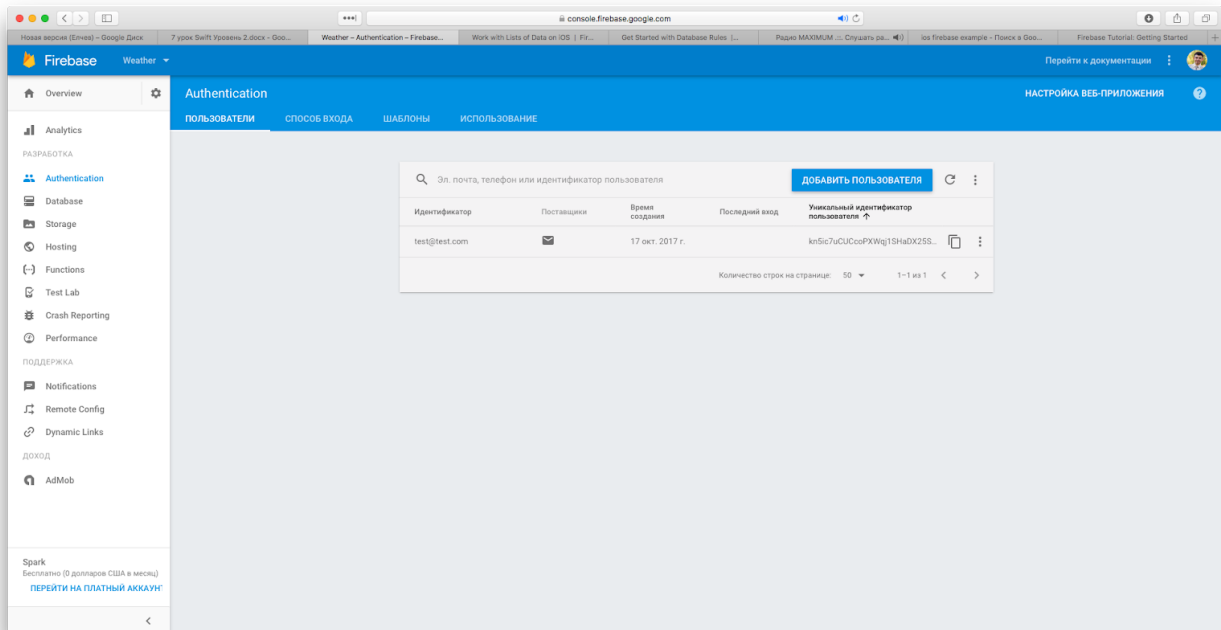
```
9 import UIKit
10 import Firebase
11
12 @UIApplicationMain
13 class AppDelegate: UIResponder, UIApplicationDelegate {
14
15     var window: UIWindow?
16
17
18     func application(_ application: UIApplication,
19         didFinishLaunchingWithOptions launchOptions:
20         [UIApplicationLaunchOptionsKey: Any]?) -> Bool {
21         // Override point for customization after application launch.
22         FIRApp.configure()
23         return true
24     }
25 }
```

Аутентификация

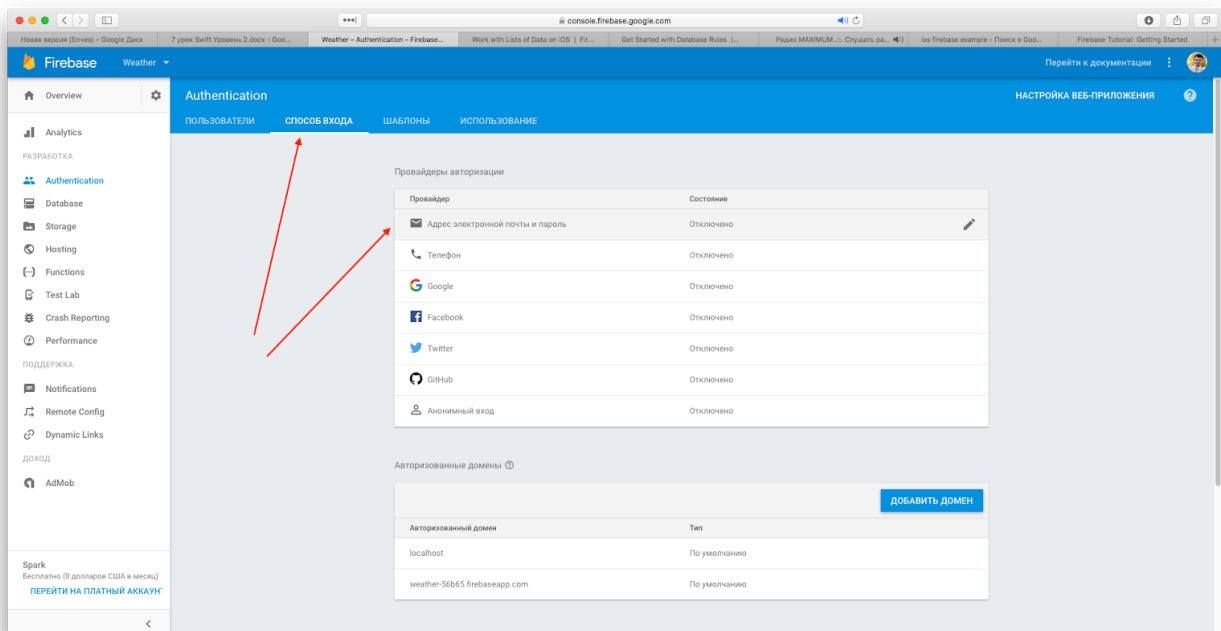
Прежде чем использовать базу данных, настраиваем аутентификацию на вкладке **Authentication**.

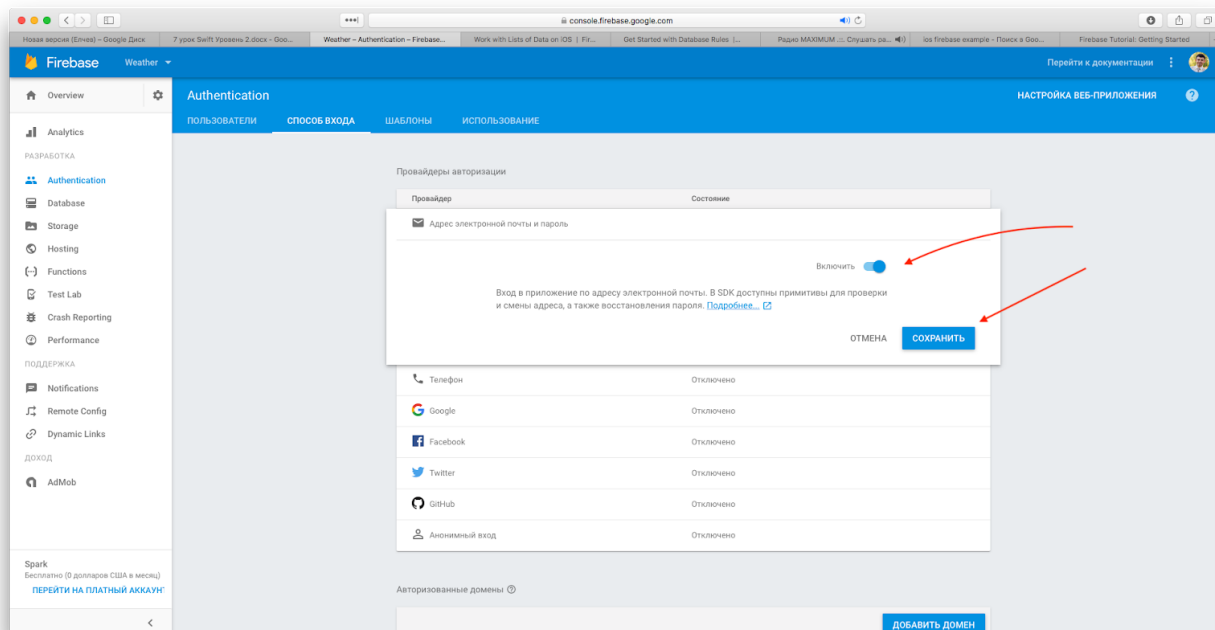
Добавляем пользователя:



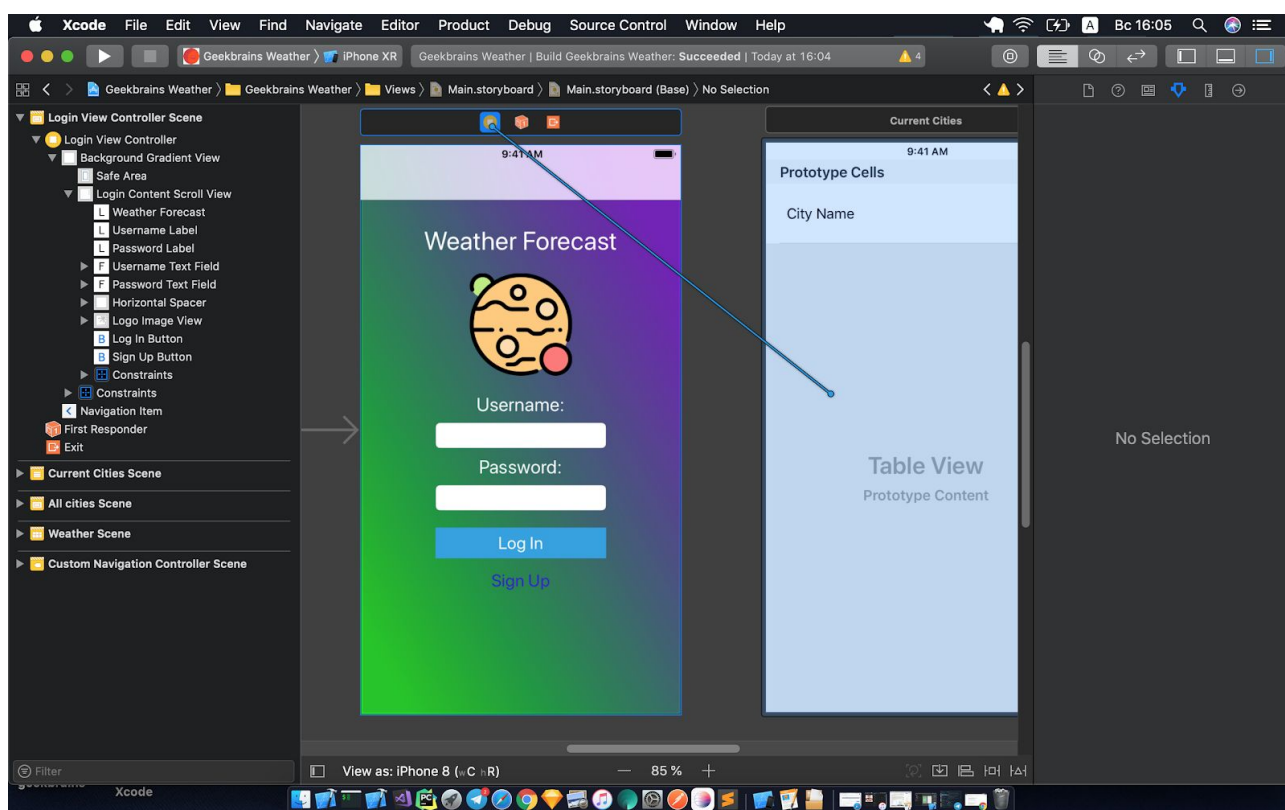


Активируем авторизацию с помощью почты и пароля:





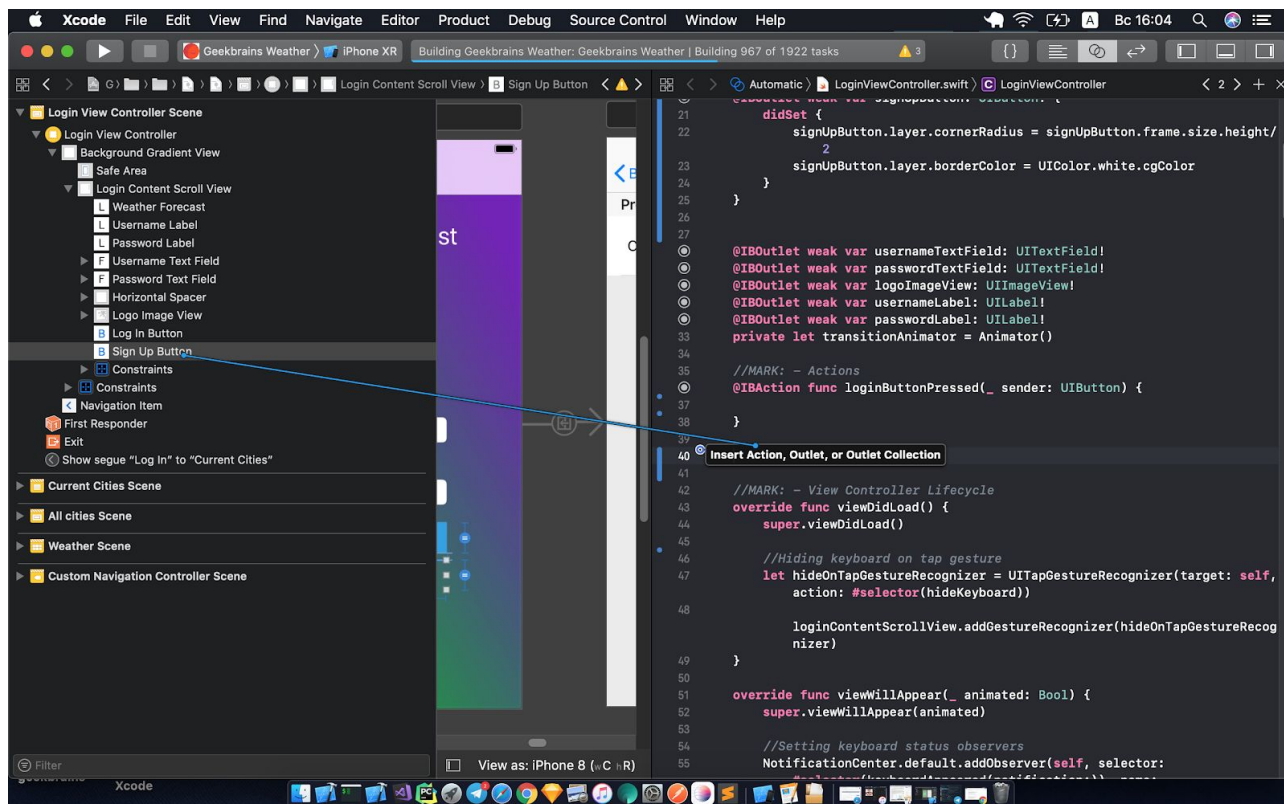
С помощью **CocoaPods** добавляем в проект библиотеку для поддержки авторизации. Имя библиотеки в pod-файле — **pod 'Firebase/Auth'**.



Открываем **Storyboard** и изменяем переход от окна авторизации: не от кнопки входа, а от самого контроллера. Не забываем установить идентификатор перехода после его пересоздания.

Добавляем дополнительный **UIButton** под кнопкой «Войти», который будем использовать для создания аккаунта. Зададим этой кнопке другой стиль, чтобы пользователю было удобно различить их функциональность, и переопределим текст на «**Sign Up**». Так как в качестве логина будем

использовать e-mail, для удобства пользователя выбираем `usernameTextField`, переходим в **Attributes Inspector** и задаем тип Keyboard Type — **E-mail Address**.



Связываем нажатие кнопок **Log In** и **Sign Up** с соответствующими **@IBAction** в коде **LoginViewController**:

Так как теперь у нас за состоянием логина пользователя будет следить **Firestore**, можем удалить функцию `checkUserData()`.

В первую очередь реализуем метод создания пользователя по нажатию кнопки **Sign Up**. Надо не забыть импортировать модуль **FirebaseAuth** в начале кода проекта. Затем создадим пользователя в **@IBAction**:

```
@IBAction func signUpButtonPressed(_ sender: UIButton) {  
    // 1  
    let alert = UIAlertController(title: "Register",  
                                message: "Register",  
                                preferredStyle: .alert)  
  
    // 2  
    alert.addTextField { textField in  
        textField.placeholder = "Enter your email"  
    }  
    alert.addTextField { textField in  
        textField.isSecureTextEntry = true  
        textField.placeholder = "Enter your password"  
    }  
    // 3  
    let cancelAction = UIAlertAction(title: "Cancel",  
                                    style: .cancel)
```



```

// 4
let saveAction = UIAlertAction(title: "Save", style: .default) { _ in
    // 4.1
    guard let emailField = alert.textFields?[0],
          let passwordField = alert.textFields?[1],
          let password = passwordField.text,
          let email = emailField.text else { return }

    // 4.2
    Auth.auth().createUser(withEmail: email, password: password) { [weak
self] user, error in
        if let error = error {
            self?.showAlert(title: "Error", message:
error.localizedDescription)
        } else {
            // 4.3
            Auth.auth().signIn(withEmail: email, password: password)
        }
    }
}
// 5
alert.addAction(saveAction)
alert.addAction(cancelAction)
present(alert, animated: true, completion: nil)
}

```

1. Для представления пользователю форм ввода данных создаем **UIAlertViewController**.
2. Прикрепляем два текстовых поля к контроллеру: первый — для ввода e-mail, второй — для пароля.
3. Создаем **Action** для отмены формы, без **completion handler**.
4. В этом пункте происходит самое интересное. Когда создали **Action**, в хендлере получаем доступ к текстовым полям контроллера и их значениям (4.1). Если получили их удачно, то при помощи метода **createUser(withEmail:, password:)**, определенного в синглтоне **Auth.auth()**, отправляем запрос о создании пользователя на сервер **Firebase** (4.2). Самое важное — это вызов метода авторизации **Auth.auth().signIn(withEmail:, password:)**. Передаем ему логин и пароль, а в замыкании ловим результат авторизации. Доступны аргументы **user** и **error**. Если авторизация будет успешной, в **user** будет пользователь. В противном случае в **error** появится пояснение, что пошло не так. Проверяем, произошла ли ошибка. Если нет — сразу авторизуем пользователя в приложении при помощи метода того же синглтона (4.3). В случае ошибки вызываем **AlertController** с описанием ошибки.

Когда все подготовительные действия выполнены, можем добавить **action**-ы к контроллеру и презентовать его пользователю.

Реализуем метод **loginButtonPressed(_ sender:)**. Логика его выполнения предельно похожа на авторизацию, которую мы реализовывали в **completion handler** метода **createUser(withEmail: password:)**. Еще нужно получить данные логина и пароля — их забираем из тестовых полей на основном экране. Если какие-либо данные не указаны, выводим предупреждение (1). После этого вызываем метод **signIn(withEmail:, password:)** и обрабатываем ошибку, если она есть (2).


```

@IBAction func loginButtonPressed(_ sender: UIButton) {
    // 1
    guard
        let email = textFieldLoginEmail.text,
        let password = textFieldLoginPassword.text,
        email.count > 0,
        password.count > 0
    else {
        self?.showAlert(title: "Error", message: "Login/password is not
entered")
        return
    }
    // 2
    Auth.auth().signIn(withEmail: email, password: password) { user, error in
        if let error = error, user == nil {
            self?.showAlert(title: "Error", message: error.localizedDescription)
        }
    }
}

```

Третьей и завершающей модификацией кода **LoginViewController** является добавление **Listener**-а состояния авторизации в **Firestore**. **Listener** реализует паттерн **Observer** и работает аналогично **NotificationToken** из библиотеки **Realm**.

```
private var handle: AuthStateDidChangeListenerHandle!
```

Если изменится статус аутентификации пользователя, он вызывает замыкание, которое определяется при помощи метода **Auth.auth().addStateDidChangeListener**. Добавим следующий код в метод **viewWillAppear()** контроллера:

```

//Adding authorization status listener
self.handle = Auth.auth().addStateDidChangeListener { auth, user in
    if user != nil {
        self.performSegue(withIdentifier: "Log In", sender: nil)
        self.usernameTextField.text = nil
        self.passwordTextField.text = nil
    }
}

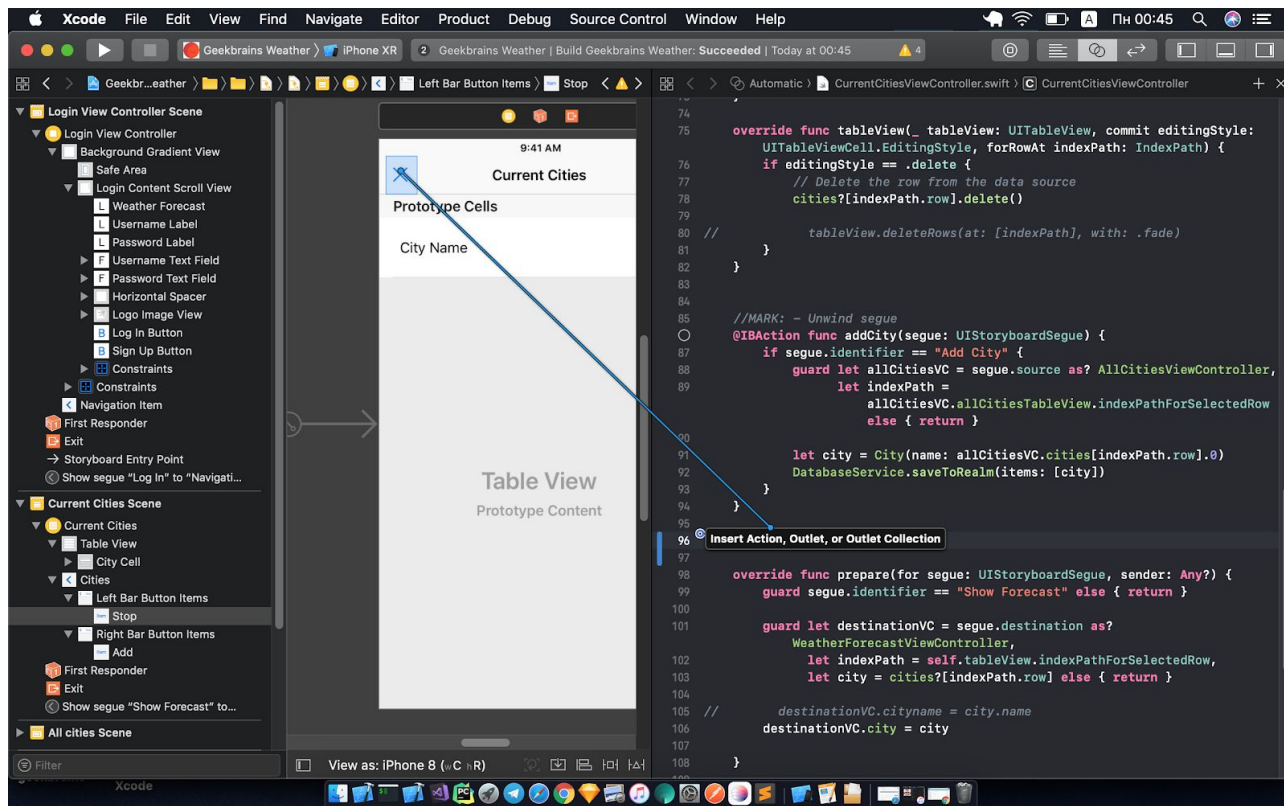
```

Мы отслеживаем появление авторизованного пользователя (параметр **user**), вызываем segue-переход на следующий экран, а значение полей ввода зануляем. После успешной авторизации можно отправлять запросы к базе данных в **Firestore**. Также не забываем отписаться от наблюдения за изменением статуса. Добавим следующий код в метод **viewDidDisappear()** контроллера:

```
Auth.auth().removeStateDidChangeListener(handle)
```

Теперь контроллер автоматически определит, залогинен ли пользователь, сохранит его статус и переадресует на следующий экран — даже при повторном запуске приложения.

Остается реализовать возможность сделать **log out** — на случай, если пользователь захочет использовать два аккаунта в сервисе или у нескольких пользователей будет одно устройство. Для этого перенесем **NavigationController** на экран после **LoginViewController** и добавим новый **BarButtonItem** для выхода.

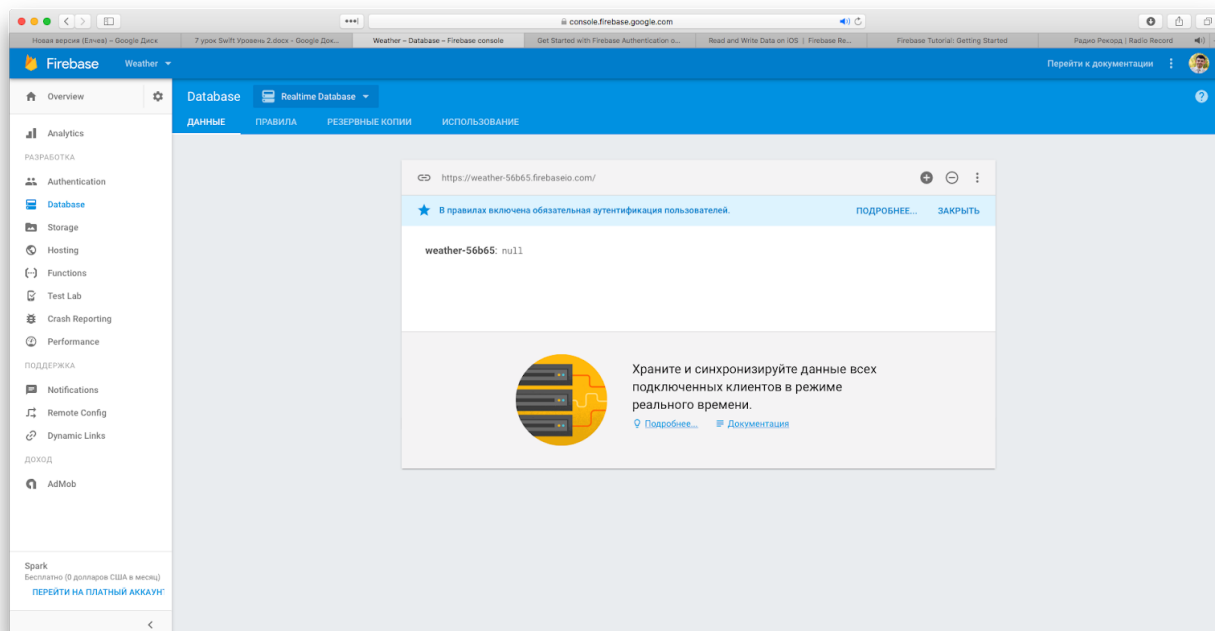
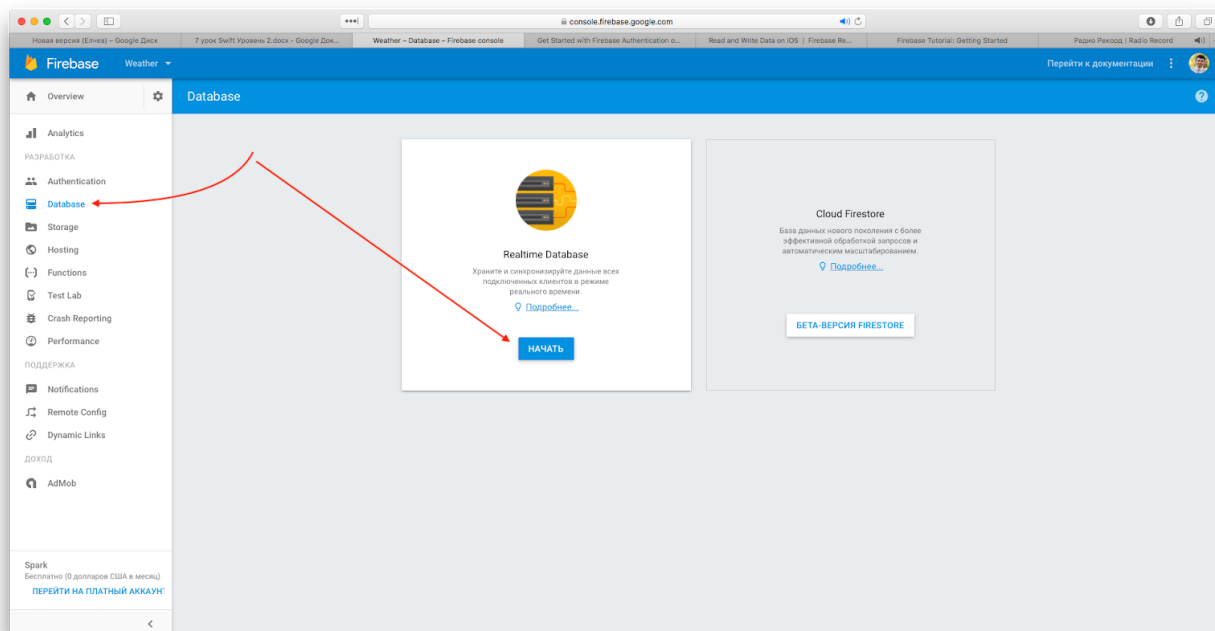


Перетащим **@IBAction** теперь уже в следующий контроллер, который отображает список сохраненных городов (**CurrentCitiesViewController**), и добавим в него приведенный на листинге код. Выход из аккаунта выполняется при помощи директивы **Auth.auth().signOut()**. Обратите внимание, что этот метод может выбросить ошибку: для этого мы оборачиваем его вызов в конструкцию **do/catch** (1). Когда выход из аккаунта выполнен, возвращаемся на прошлый экран при помощи метода **dismiss(animated:)**. В случае ошибки выводим в консоль сообщение (2).

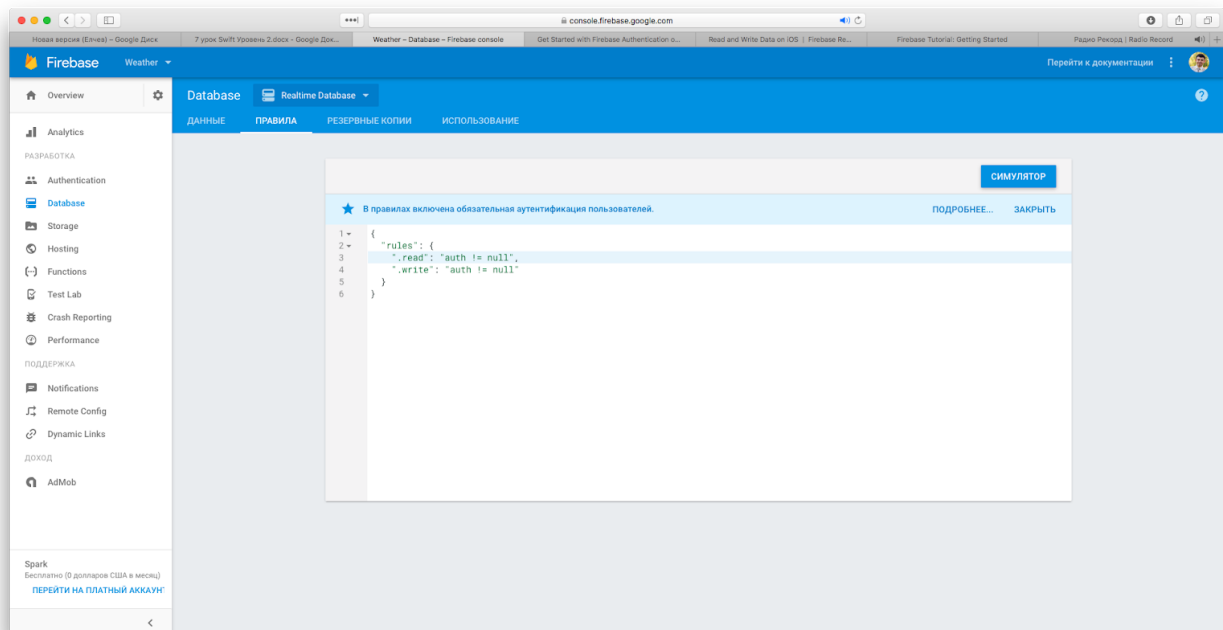
```
@IBAction func logOutButtonPressed(_ sender: UIBarButtonItem) {
    do {
        // 1
        try Auth.auth().signOut()
        self.dismiss(animated: true, completion: nil)
    } catch (let error) {
        // 2
        print("Auth sign out failed: \(error)")
    }
}
```

Использование Firebase

Переходим на вкладку **Database**. Перед нами стоит выбор: использовать «проверенный» вариант **Realtime Database** или более прогрессивный, но и более сырой **Cloud Firestore**, который пока находится в бета-тестировании. Сначала познакомимся с **Realtime Database** — нажимаем кнопку «Начать».



Видим пустую базу и предупреждение, что в правилах установлены ограничения на запись и чтение. По умолчанию доступ разрешен только авторизованным пользователям. Так как авторизация уже настроена, у нас будет доступ к базе. При желании его можно открыть всем.



Изменим принцип сохранения городов, прогнозы погоды в которых нас интересуют. Теперь вместо **Realm** будем использовать **Firebase**. Для начала создадим новый класс **FirebaseCity**, который будет реализовывать модель города для сохранения. База данных в Firebase представляет собой древовидную структуру, каждая ветка которой имеет два параметра: строковый тип (название ветки) и ссылку на дочернюю ветку или value — значение, хранящееся в данной ветке (в этом случае она называется листом). Наша структура данных будет представлять собой корневой объект (назовем его **cities**), содержащий:

- все города;
- дочерние ветки, хранящие данные каждого отдельного города;
- листья с данными.

Сохраним строковый параметр **name** (имя города) и числовой параметр **index** (индекс города).

В нашем классе определим три параметра (1):

- **name: String** будет хранить название города;
- **index: Int** будет отвечать за индекс и представлять числовой тип;
- **ref: DatabaseReference** — класс библиотеки **Firebase**, который определяет ссылку на интересующий нас объект в базе данных. У нас всегда будет быстрый доступ к объекту в базе.

В классе определим два конструктора и функцию, превращающую объект в словарь типа **[String: Any]**.

Первый конструктор (2) будем использовать при создании объекта вручную: зададим значения, а референс будет нулевым. Это логично, так как этот объект еще не загружен в базу **Firebase**.

Второй конструктор будем использовать, получая ответ от **Realtime Database**. Для этого будем пытаться инициализировать объект специальным классом **DataSnapshot**. Он представляет собой **JSON**-объект типа «ключ-значение», но и хранит вспомогательные параметры — например, ссылку типа **DatabaseReference**. Обратите внимание, что инициализатор объявлен как **failable** (3), то есть в случае неудачи при парсинге он вернет **nil**. Последняя функция **toAnyObject()** простейшим образом

пробегаются по всем сущностям и составляет словарь (4). В нем ключ совпадает с именем параметров-членов класса, а их значение использует как значение в словаре (представляет объект в виде **JSON**).

```
import Foundation
import Firebase

class FirebaseCity {
    // 1
    let name: String
    let zipcode: Int
    let ref: DatabaseReference?

    init(name: String, zipcode: Int) {
        // 2
        self.ref = nil
        self.name = name
        self.zipcode = zipcode
    }

    init?(snapshot: DataSnapshot) {
        // 3
        guard
            let value = snapshot.value as? [String: Any],
            let zipcode = value["zipcode"] as? Int,
            let name = value["name"] as? String else {
                return nil
            }

        self.ref = snapshot.ref
        self.name = name
        self.zipcode = zipcode
    }

    func toAnyObject() -> [String: Any] {
        // 4
        return [
            "name": name,
            "zipcode": zipcode
        ]
    }
}
```

Перейдем к глобальному рефакторингу класса **CurrentCitiesViewController**. Чтобы модули Firebase были доступны, импортируем их в начале файла.

```
import FirebaseDatabase
import FirebaseAuth
```

Так как теперь не будем пользоваться ни объектом типа **Results**, ни объектом типа **NotificationToken**, их со всеми вызовами можно закомментировать. Вместо них объявим два других параметра:

- **cities** — способ хранения объектов в массиве;
- **ref** — тоже референс, только не на объект, а на ветку нашей таблицы, в данном случае **cities**. Если она не существует, **Firebase** создает ее автоматически).

```
private var cities = [FirebaseCity]()
private let ref = Database.database().reference(withPath: "cities")
```

Чтобы успокоить компилятор, возмущенный внезапным изменением источника данных, переопределим методы **delegate** и **dataSource**. Количество возвращаемых ячеек у теперь будет следующим:

```
override func tableView(_ tableView: UITableView, numberOfRowsInSection section:
Int) -> Int {
    return cities.count
}
```

А значение ячейки будет определяться следующим образом:

```
override func tableView(_ tableView: UITableView, cellForRowAt indexPath:
IndexPath) -> UITableViewCell {
    guard let cell = tableView.dequeueReusableCell(withIdentifier: "City Cell",
for: indexPath) as? CityCell else { return UITableViewCell() }

    let city = cities[indexPath.row]
    cell.cityNameLabel.text = city.name

    return cell
}
```

Незначительно изменим поведение функции **addCity(_ sender:)**: вместо класса **City(name:)** создадим (1) объект **FirebaseCity(name:zipcode:)**. Самое интересное происходит двумя строками ниже: используя объект **ref** (ссылка на корневую ветку, объявленную в начале контроллера), при помощи метода **child()** мы добавляем к ней подветку, которую называем именем города (2). Значение, которое будет храниться на этой подветке, задаем методом **setValue()**.

```

@IBAction func addCity(_ sender: Any) {
    let alertVC = UIAlertController(title: "Enter a city name please", message: nil,
    preferredStyle: .alert)

    let saveAction = UIAlertAction(title: "Save", style: .default) { _ in
        guard let textField = alertVC.textFields?.first,
            let cityname = textField.text else { return }

        // 1
        let city = FirebaseCity(name: cityname,
                                zipcode: Int.random(in: 100000...999999))
        // 2
        let cityRef = self.ref.child(cityname.lowercased())

        cityRef.setValue(city.toObject())
    }

    let cancelAction = UIAlertAction(title: "Cancel",
        style: .cancel)

    alertVC.addTextField()

    alertVC.addAction(saveAction)
    alertVC.addAction(cancelAction)

    present(alertVC, animated: true, completion: nil)
}

```

В качестве параметра **setValue** нужно передать словарь **JSON**-формата. Чтобы его сформировать, воспользуемся методом **toAnyObject()**, объявленным в классе **FirebaseCity**.

Обратите внимание: мы не обновляем изначальный массив с данными **cities**. Как и в работе с **Realm**, будем делать это автоматически при помощи **Listener**.

В проекте реализуем только метод общего наблюдения над всем деревом объектов, начиная с ветки **ref**. В качестве параметров функции **observe** передаем объект типа **DataEventType**, описывающий, какие изменения отслеживать. В нашем случае это **.value**, то есть все возможные изменения этой ветки. Также передаем замыкание, которое будет вызываться в случае этих изменений. В замыкании присутствует параметр **snapshot** — снимок состояния наблюдаемой ветки. Ее подветки — и есть сохраненные города, поэтому пробегаемся по ним циклом **for in** и инициализируем при помощи конструктора **FirebaseCity(snapshot:)**. Когда все ветки получены, пересоздаем массив данных и обновляем таблицу (3).

```

override func viewDidLoad() {
    super.viewDidLoad()
    // 1
    ref.observe(.value, with: { snapshot in
        var cities: [FirebaseCity] = []
        // 2
        for child in snapshot.children {
            if let snapshot = child as? DataSnapshot,
                let city = FirebaseCity(snapshot: snapshot) {
                cities.append(city)
            }
        }
        // 3
        self.cities = cities
        self.tableView.reloadData()
    })
}

```

Осталось реализовать возможность удаления данных из таблицы. При помощи параметра **FirebaseCity.ref** сделать это совсем просто:

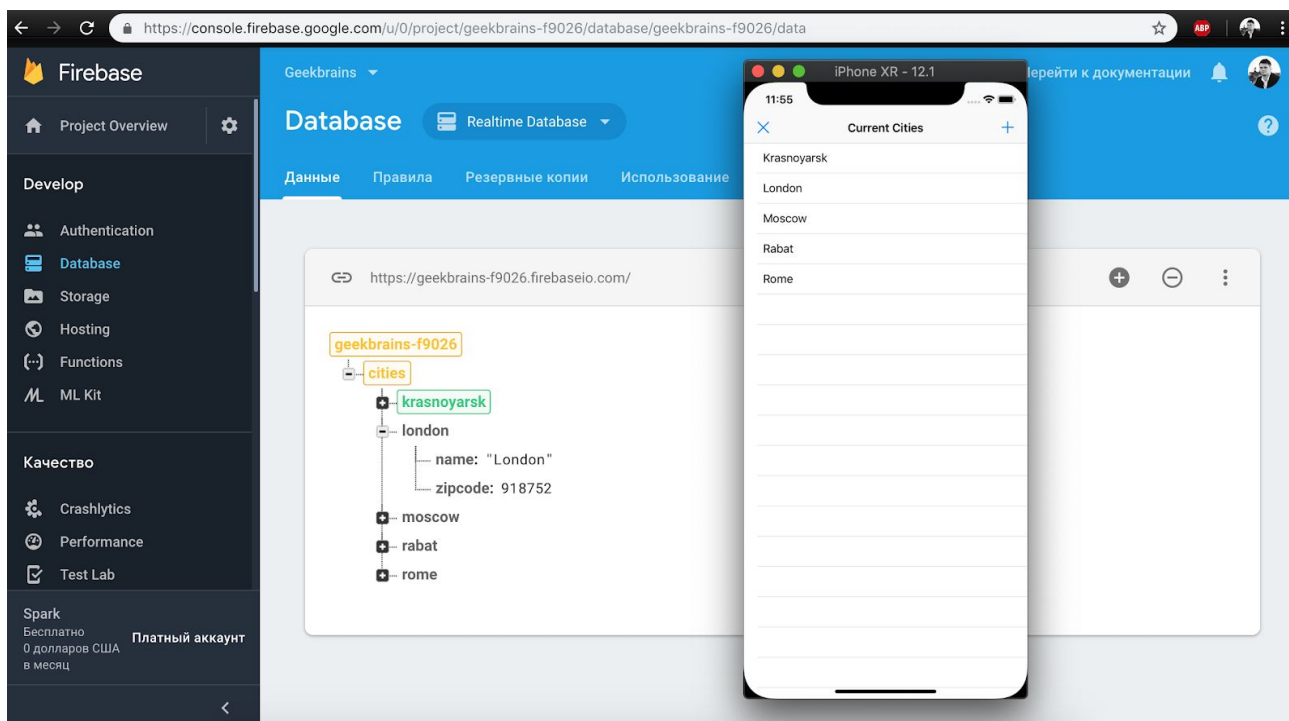
```

override func tableView(_ tableView: UITableView, commit editingStyle:
UITableViewCellEditingStyle, forRowAt indexPath: IndexPath) {
    if editingStyle == .delete {
        // Delete the row from the data source
        let city = cities[indexPath.row]
        city.ref?.removeValue()
    }
}

```

Добавим несколько городов и удостоверимся, что они создаются в **Realtime Database**. Если открыть консоль **Firebase** и перейти в ней на страницу базы данных, можно убедиться, что при добавлении сущности почти мгновенно отображаются в консоли и подсвечиваются зеленым, а при удалении — красным. Нажав на плюс, можно раскрыть ветку (подветку). Листы, не имеющие подветок, отображены в формате «ключ-значение». Обратите внимание, что аналогично **JSON** строковые значения выделены кавычками, а числовые (как **Int** в нашем случае) — нет.

С **Realtime Database** познакомимся. Теперь запустим **Firestore** и сохраним в него результаты прогнозов погоды. Для этого в **Firebase Console** перейдем на вкладку **Database** и активируем **Firestore**. Видим другую структуру представления данных: БД **Firestore** представляет собой хранилище коллекций, выполняющих функции папок. В коллекции может находиться сколько угодно документов, отражающих сущности и содержащих данные. Переработаем **ForecastWeatherController**, чтобы он сохранял данные не только в **Realm**, но и в коллекции **Forecasts**, создавал документы, соответствующие городам, и сохранял в них результаты прогнозов погоды.



В начале файла не забудем импортировать библиотеку.

```
import FirebaseFirestore
```

А для реализации сохранения определим **private**-функцию. Важно помнить, что для сохранения в **Firestore** необходимо представить объект как словарь типа **[String: Any]**. Добавим следующую функцию в класс **Weather**. В отличие от класса **FirestoreCity**, в данном случае у нас ключ не фиксированный: он формируется из значения даты. В качестве значения передадим температуру.

```
func toFirestore() -> [String: Any] {
    return [
        String(format: "%0.f", date) : temperature
    ]
}
```

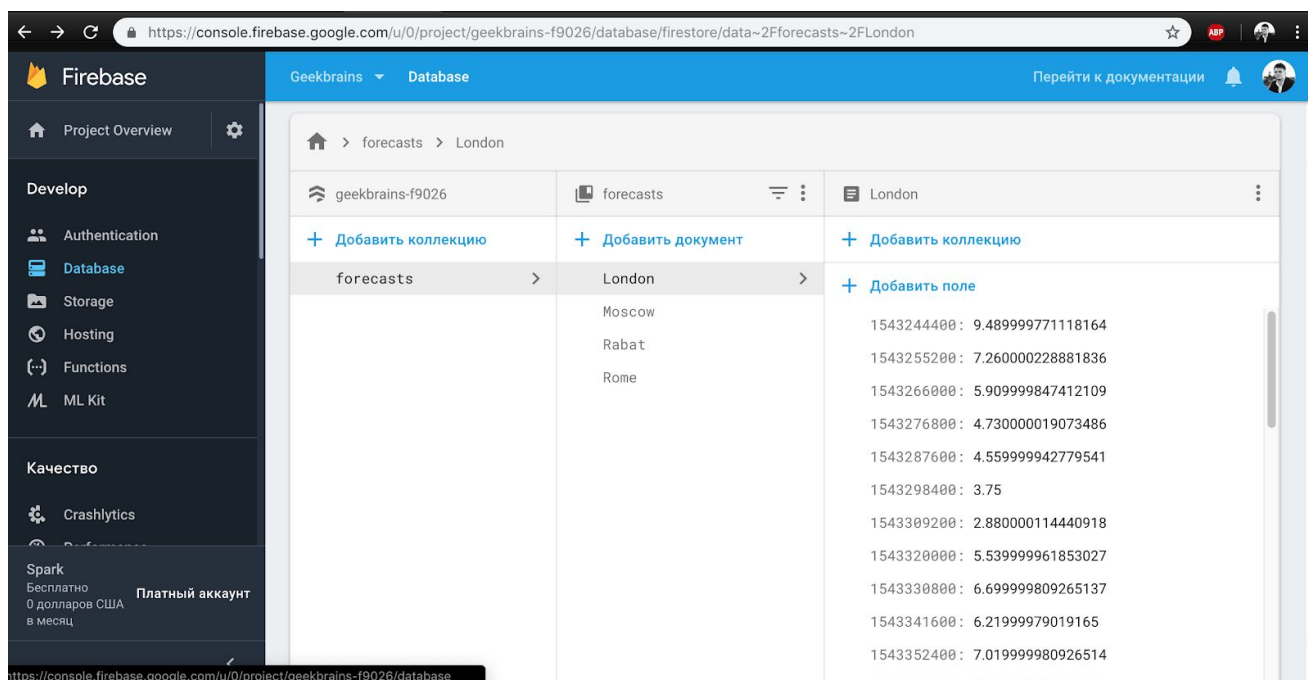
Можно создавать функцию сохранения. В первую очередь получим доступ к базе данных (1). Метод **Firestore.firestore()**. **Settings** — это настраиваемые параметры базы данных. Мы выставим временные отметки по умолчанию.

Второй важной частью функции будет подготовка общего словаря для всех прогнозов погоды, чтобы не создавать 40 обращений к серверу для каждого прогноза (2). Для этого сформируем из массива прогнозов погоды массив данных, которые хотим отправить. Потом при помощи функции **reduce** объединим массив словарей в один словарь.

Как и в **Realtime Database**, обратимся к коллекции **forecasts** — если ее не существует, то она будет создана. Назовем документ именем города (его надо подготовить заранее, например в **viewDidLoad()**). Если не знаем или не хотим именовать, то **Firebase** предоставляет механизм автоматической генерации и присвоения уникальных имен документам, схожий с **UUID().uuidString** в Swift. В качестве данных отправим словарь (3). Результат операции обрабатывается в **Handler**, который выведет ошибку или порадует сообщением «**data saved**».

```
private func saveToFirestore(_ weathers: [Weather]) {
    // Setting up the Cloud Firestore
    // 1
    let database = Firestore.firestore()
    let settings = database.settings
    settings.areTimestampsInSnapshotsEnabled = true
    database.settings = settings
    // 2
    let weathersToSend = weathers
        .map { $0.toFirestore() }
        .reduce([:]) { $0.merging($1) { (current, _) in current } }
    // 3
    database.collection(«forecasts»).document(self.cityname).setData(weathersToSend,
merge: true) { error in
        if let error = error {
            print(error.localizedDescription)
        } else { print("data saved") }
    }
}
```

Перейдем в браузер и удостоверимся, что прогнозы погоды сохраняются в базе данных.



Обратите внимание, что значение документа не ограничивается типом «ключ-значение». Можно сохранить вложенную коллекцию в качестве данных документа, в ней еще одну — и так до бесконечности.

Практическое задание

1. Создать учетную запись в Firebase.
2. Интегрировать в приложение.
3. Настроить изменение данных в Firebase без авторизации.
4. Записывать в базу пользователей, которые авторизовались в приложении (id).
5. Записывать каждому пользователю группы, которые он добавлял в приложении.

Дополнительные материалы

1. [Firebase. Installation & Setup on iOS.](#)
2. [Get Started with Firebase Authentication on iOS.](#)

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. [Firebase. Installation & Setup on iOS.](#)