Agenda

- Understanding Cloud Firestore
- Cloud Firestore Data Model
- Setting up development environment
- CRUD using Cloud Firestore

Cloud Firestore

- Database provided by Firebase and Google Cloud Platform (GCP)

- Cloud-hosted NoSQL database
- Used for mobile, web and server development
- Can be accessed via native SDKs

Cloud Firestore Capabilities

- **Flexible** and Scalable
- **Expressive querying**
- Uses **realtime** listeners to keep the data in sync across client apps
- Addresses network latency and internet connectivity issues by offering **offline support** for mobile and web apps
- Supports seamless **integration** with other Firebase and GCP products

Cloud Firestore Data Model

- Cloud Firestore is a **NoSQL**, document-oriented database.
- There are **no tables or rows** in Cloud Firestore.
- The data is stored in **documents**, which are organized into **collections**.
- Cloud Firestore is optimized for storing large collections of small documents.
- All documents must be stored in collections.

Documents

- The document is the unit of storage in Cloud Firestore.
- It is a **lightweight record** that contains **fields, which map to values**.
- Each document is identified by a name.
- It supports **common data types** such as strings, numbers as well as nested objects.
- The documents can be considered as **lightweight JSON** records as they **limited in size to 1 MB**.

Example: Document in Cloud Firestore

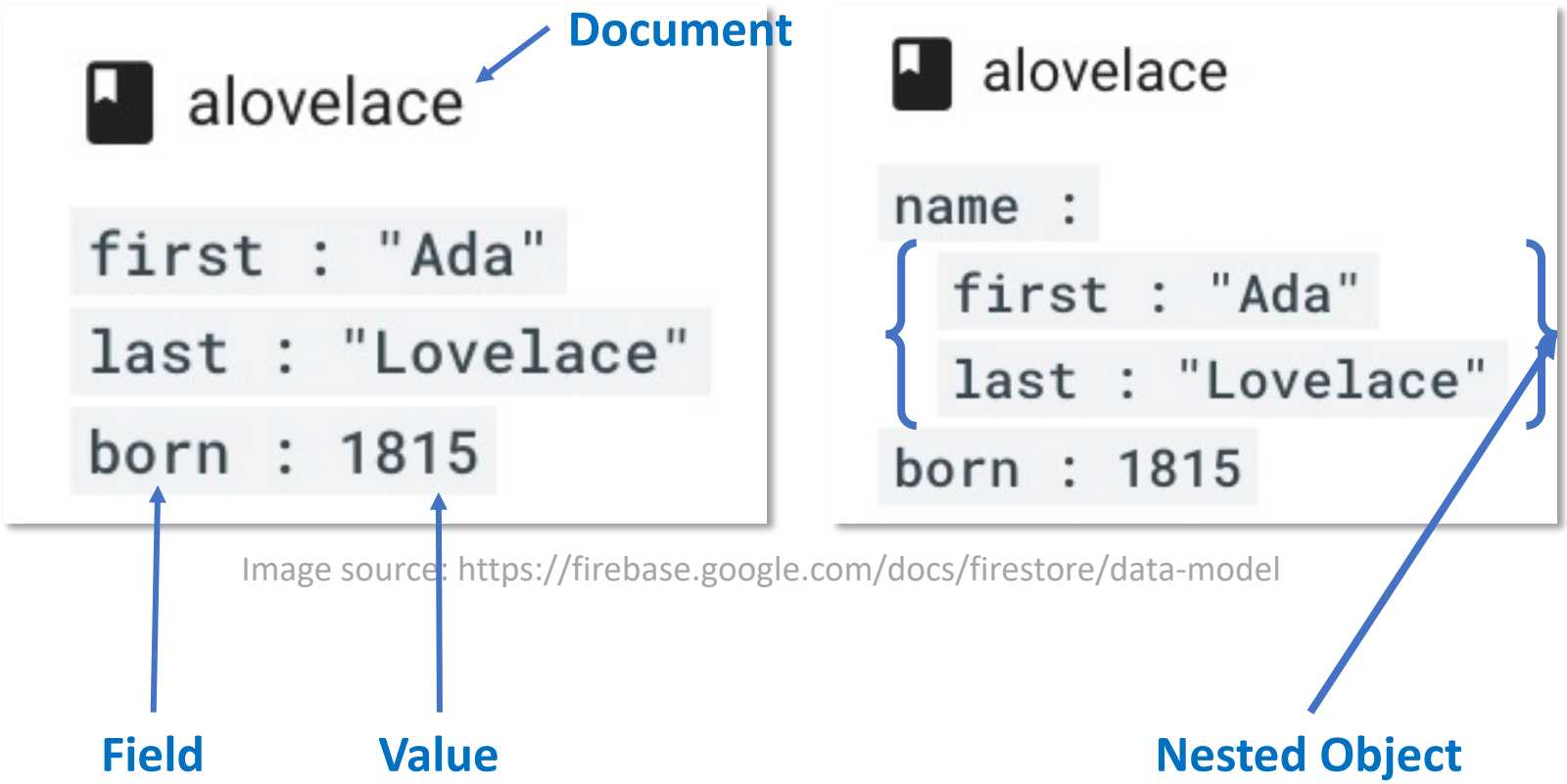


Image source: https://firebase.google.com/docs/firestore/data-model

Collections

- The collections are **containers** for documents.
- Cloud Firestore is a **schemaless** database which allows users to store different fields in each document with different types of values in it.
- Collections **can't contain any fields** with values **or collections** within it.
- Collection is **automatically created** when the first document is created.
- If all the documents within collection is deleted, the collection no longer exists.

Example: Collection in Cloud Firestore

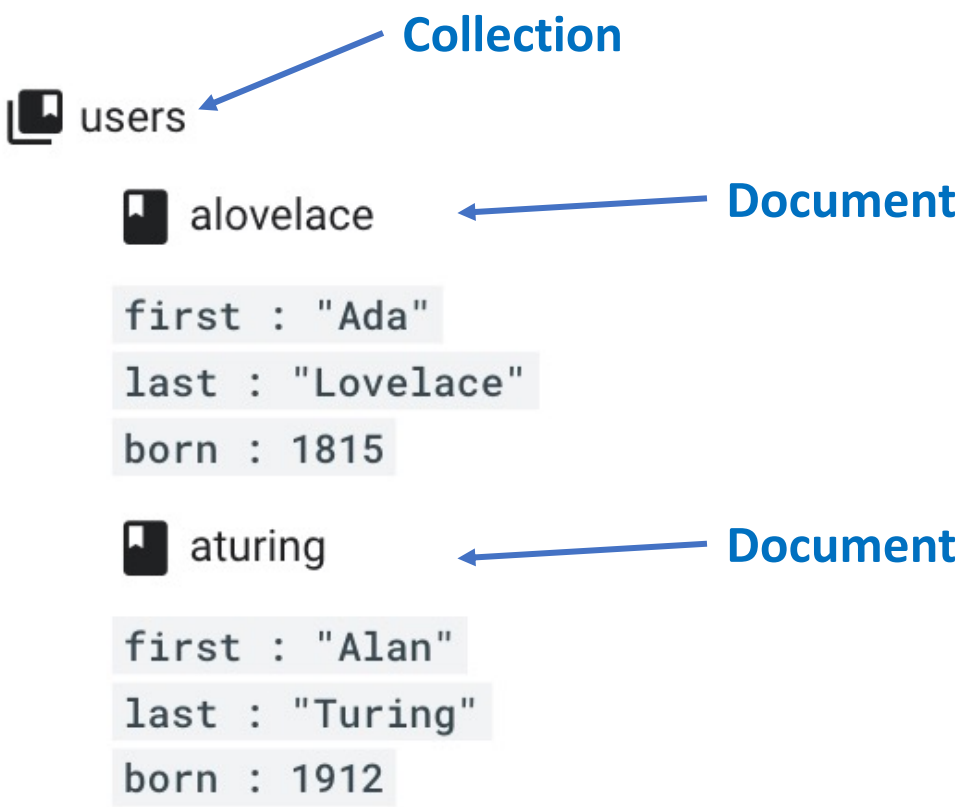


Image source: https://firebase.google.com/docs/firestore/data-model

References

- A reference is a **lightweight object** that points to a **location** in your database which uniquely **identifies each document** in Cloud Firestore.

- The reference does not perform any network operations.
- You may create references for documents to read or write individual document or a reference to collection to query the documents in collection.

Example: Creating Reference

```
let alovelaceDocumentRef = db.collection("users").document("alovelace")
```

Image source: https://firebase.google.com/docs/firestore/data-model

```
let usersCollectionRef = db.collection("users")
```

Image source: https://firebase.google.com/docs/firestore/data-model

Subcollections

- Subcollections are useful to create **hierarchical data structures** in Cloud Firestore.
- It is a collection associated with a specific document.
- Subcollections are helpful in creating **lightweight documents**.
- Documents in subcollections can **contain subcollections** as well, allowing you to further nest data. You can nest data up to **100 levels deep**.

Example: Subcollections

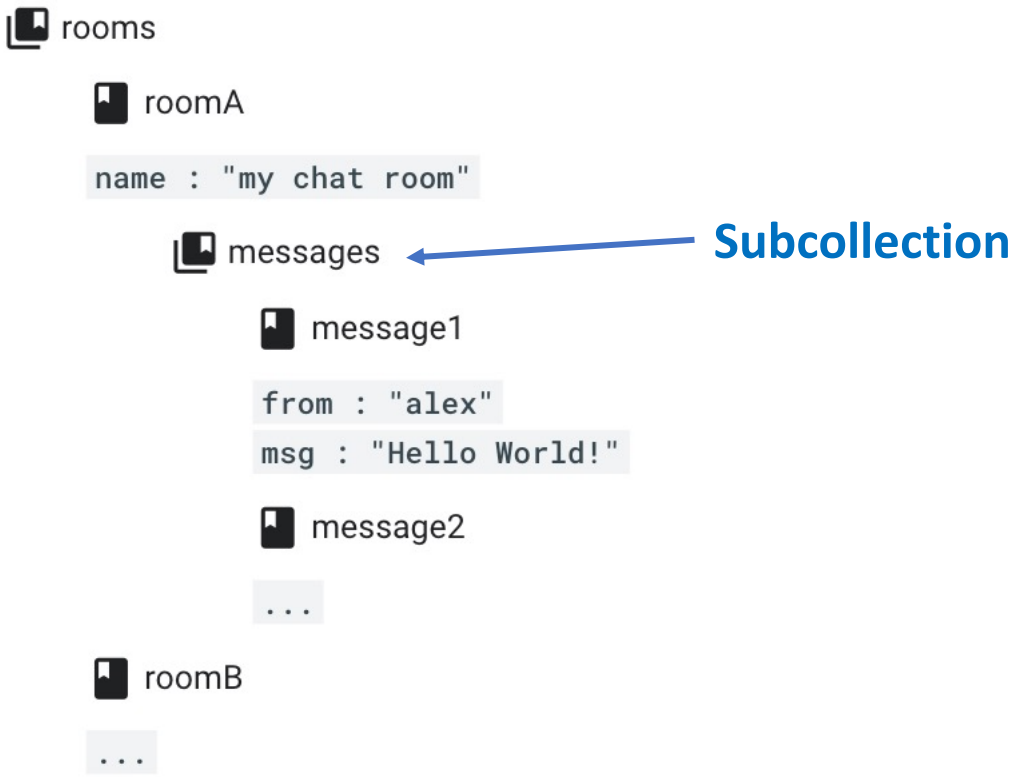


Image source: https://firebase.google.com/docs/firestore/data-model

Setting Up Development Environment

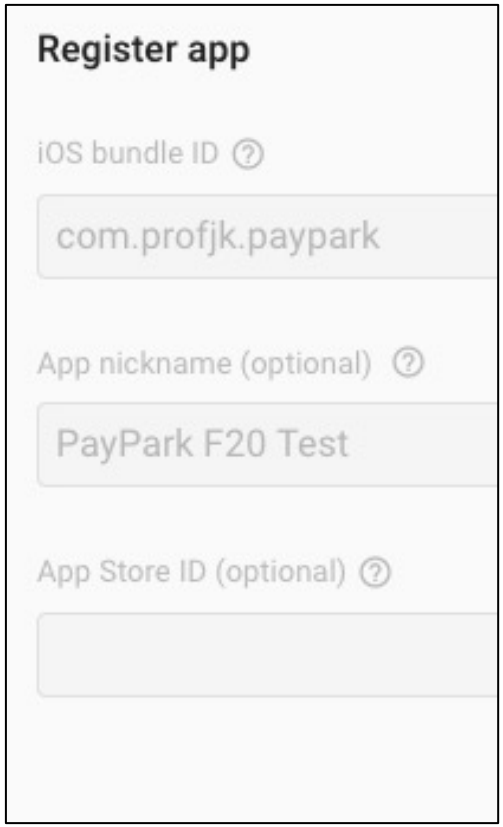
Integrating Firebase in iOS project

1. Create Firebase Project
2. Register your app with Firebase
3. Add Firebase Configuration File
4. Add Firebase SDKs to your app
5. Initialize Firebase in your app

- Find the detailed instruction here.

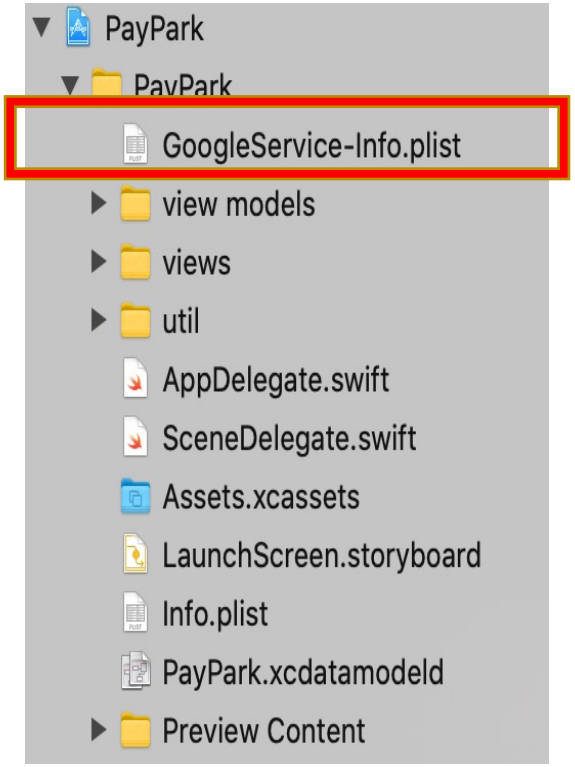
Step 1 : Create Firebase Project

1. In the Firebase console, click **Add project**
2. Enter a **Project name**
3. *(Optional)* If you are creating a new project, you can edit the **Project ID**.
4. Click **Continue**.
5. *(Optional)* Set up Google Analytics for your project
6. Click **Create project**



Step 2 : Register your app with Firebase

1. In the Firebase console, click on the project name you create to visit the **Project Overview** page.
2. On the project overview page, click the **`iOS`** icon to launch the setup workflow.
3. Enter your app's bundle ID in the **iOS bundle ID** field.
4. Click **Register App**.



Step 3 : Add Firebase Configuration File

1. Download the **`GoogleService-Info.plist`** file

2. Add the downloaded file into the root of your Xcode project and add it to all targets

Step 4 : Add Firebase SDKs to your app

```
target 'PayPark' do
  use_frameworks!

  # Pods for PayPark

  #Firebase
  pod 'Firebase'

  #Cloud Firestore
  pod 'Firebase/Firestore'

  #Swift extensions
  pod 'FirebaseFirestoreSwift'
end
```

1. Open a **terminal** window and navigate to the location of the Xcode project for your app.
2. Create a pod file for your project

   `pod init`

3. Open the **Podfile** and add the pod dependencies

   `pod 'Firebase'`

   `pod 'Firebase/Firestore'`

4. Install the pods

   `pod install`

Note: This will create **.xcworkspace** file for your app. Use this file for all further development of your app.

Step 5 : Initialize Firebase in your app

1. Add the Firebase initialization code in the AppDelegate class [Note: you should open project using **.xcworkspace**]

```
import Firebase

@main
class AppDelegate: UIResponder, UIApplicationDelegate {

    func application(_ application: UIApplication,
                     didFinishLaunchingWithOptions launchOptions:
                        [UIApplication.LaunchOptionsKey: Any]?) -> Bool {

        FirebaseApp.configure()

        return true
    }
```

Integrating Cloud Firestore in iOS project

1. In the Firebase console, from the Project Overview page, select **Cloud Firestore**
2. Select **Create database**
3. Select a starting mode for your Cloud Firestore Security Rules:
   - Product mode – private data
   - **Test mode** – open data **[Select this option]**

- Find the detailed instruction here.

Integrating Cloud Firestore in iOS project cont…

4. Select a **location** for your database and select **Enable**.
5. Initialize Cloud Firestore in the **SceneDelegate** class.

```
import FirebaseFirestore

class SceneDelegate: UIResponder, UIWindowSceneDelegate {

    var window: UIWindow?
    let db = Firestore.firestore()
```

CRUD using Cloud Firestore

Add data to Cloud Firestore

- There are several ways to write data to Cloud Firestore:
  - Set the data of a document within a collection, explicitly specifying a document identifier.
  - Add a new document to a collection. In this case, Cloud Firestore automatically generates the document identifier.
  - Create an empty document with an automatically generated identifier, and assign data to it later

Data Types

```
let docData: [String: Any] = [
    "stringExample": "Hello world!",
    "booleanExample": true,
    "numberExample": 3.14159265,
    "dateExample": Timestamp(date: Date(
    "arrayExample": [5, true, "hello"],
    "nullExample": NSNull(),
    "objectExample": [
        "a": 5,
        "b": [
            "nested": "foo"
        ]
    ]
]
```

- Cloud Firestore lets you write a variety of data types inside a document, including strings, booleans, numbers, dates, null, and nested arrays and objects.
- Cloud Firestore always stores numbers as doubles, regardless of what type of number you use in your code.

Custom Objects

- Cloud Firestore supports document creation from the custom classes.
- The custom class must contain data types which are compatible with Cloud Firestore data types.

- It must conform to **Codable** protocol which make your data types encodable and decodable for compatibility with external representations such as JSON.

Example: Custom Class

```
import FirebaseFirestoreSwift

struct TaskMO: Codable{
    @DocumentID var id: String? = UUID().uuidString
    var title: String = ""
    var completion: Bool = false

    init(){}

    init(title: String, completion: Bool) {
        self.title = title
        self.completion = completion
    }
}
```

Add a document

- You can use `addDocument()` method to create a new document in the Cloud Firestore collection.
- If the specified collection doesn't exist on the Firestore, it will be created.
- The addDocument() method encodes an instance of Encodable and adds a new document to this collection with the encoded data, assigning it a document ID automatically.
- It returns a **DocumentReference** pointing to the newly created document.

Example: Add a document

```
@Published var taskList = [TaskMO]()

let db = Firestore.firestore()

func insertTask(newTask: TaskMO){
    do{
        _ = try db.collection("Tasks").addDocument(from: newTask)
    }catch let error as NSError{
        print(#function,
              "Error inserting document: \(error.localizedDescription)")
    }
}
```

Retrieving document

- You can use getDocuments() method to retrieve all the documents from collection once.

- Alternatively, you can use **SnapshotListener** to get realtime updates from Cloud Firestore.

Example: Retrieving document

```
func getTasks(){

    db
        .collection("Tasks")
        .order(by: "title", descending: true)
        .addSnapshotListener({ [self](querySnapshot,error) in
            guard let snapshot = querySnapshot else {
                print(#function,
                      "Error fetching snapshot results: \(error!)")
                return
            }

            snapshot.documentChanges.forEach{ (doc) in
                do{
                    let task = try doc.document.data(as: TaskMO.self)!
                    print(#function, task)

                    if doc.type == .added{
                        //TODO for new tasks
                    }

                    if doc.type == .modified{
                        //TODO for modified tasks
                    }

                    if doc.type == .removed{
                        //TODO for deleted tasks
                    }
                }catch let error as NSError{
                    print(error)
                    print(error.localizedDescription)
                }
            }
        })
}
```

Update a document

- You can update the entire document by using **updateData()** method for the document.

- It accepts an array containing the fields to update with new value to set.

Example: Update a document

```
func updateTask(documentID: Int, task: TaskMO){

    db
        .collection("Tasks")
        .document(self.taskList[documentID].id!)
        .updateData(["completion" : task.completion])
        { (error) in
            if let error = error {
                print(#function,
                      "Error updating document: \(error.localizedDescription)")
            } else {
                print(#function,
                      "Document successfully updated!")
            }
        }
}
```

Delete a document

- You can use **delete()** method to delete the document once you have identified it.

- The delete() method will not delete any subcollections of a document.

Example: Delete a document

```
func deleteTask(documentID: Int) {
    db
        .collection("Tasks")
        .document(self.taskList[documentID].id!)
        .delete{ (error) in
            if let error = error {
                print(#function,
                      "Error removing document: \(error.localizedDescription)")
            } else {
                print(#function,
                      "Document successfully removed!")
            }
        }
}
```

References

- https://firebase.google.com/docs/firestore/quickstart#ios
- https://firebase.google.com/docs/firestore
- https://firebase.google.com/docs/firestore/data-model
- https://firebase.google.com/docs/firestore/manage-data/add-data
- https://firebase.google.com/docs/firestore/manage-data/delete-data
- https://firebase.google.com/docs/firestore/query-data/listen