

Firestore

DATA STORAGE IN REACT-NATIVE

Data storage in react-native

- ▶ Data is a key part of any mobile app. It gives meaning to a plain user interface. But storing, retrieving, and maintaining the data is the real hurdle. The use of different ways of storage mechanisms (encrypted store, offline store, service-oriented store, auto-sync store) is essential to store various kinds of data that can uplift the whole process of mobile app development.
- ▶ When we create React Native apps, there are several ways to store or persist data. Each storage method has its own strong advantage.

Async Storage

- ▶ AsyncStorage is an unencrypted, asynchronous, persistent, key-value storage system that can be accessed globally on the app.

On iOS, AsyncStorage is backed by native code that stores small values in a serialized dictionary and larger values in separate files. On Android, AsyncStorage will use either RocksDB or SQLite based on availability.

AsyncStorage **supports only 6 MB on Android** and a limitless amount of data on iOS. If you are aiming to build a cross-platform app, 6MB is the limit.

Supported platforms

- iOS
- Android
- Web
- MacOS
- Windows

The `AsyncStorage` JavaScript code is a facade that provides a clear JavaScript API, real `Error` objects, and non-multi functions. Each method in the API returns a `Promise` object.

```
npx expo install @react-native-async-storage/async-storage
```

- ▶ It is officially a way to persist data in React Native applications. This is given by the React native in their Documentation. Basically, AsyncStorage functions like a storage class and it gets key-value pairs to persist data as parameters.
- ▶ But AsyncStorage class is asynchronous; data storage on the device is not permanent and not encrypted, therefore when using this method, you will need to provide your backup and synchronization classes. So that if you'll be dealing with a large amount of data in your application do not use the AsyncStorage method.

Async Storage can only store `string` data, so in order to store object data you need to serialize it first.

For data that can be serialized to JSON you can use `JSON.stringify()` when saving the data and `JSON.parse()` when loading the data.

Importing

```
import AsyncStorage from '@react-native-async-storage/async-storage';
```

Storing string value

```
const storeData = async (value) => {  
  try {  
    await AsyncStorage.setItem('@storage_Key', value)  
  } catch (e) {  
    // saving error  
  }  
}
```

Storing object value

```
const storeData = async (value) => {  
  try {  
    const jsonValue = JSON.stringify(value)  
    await AsyncStorage.setItem('@storage_Key', jsonValue)  
  } catch (e) {  
    // saving error  
  }  
}
```

Reading data

`getItem` returns a promise that either resolves to stored value when data is found for given key, or returns `null` otherwise.

Reading string value

```
const getData = async () => {
  try {
    const value = await AsyncStorage.getItem('@storage_Key')
    if(value !== null) {
      // value previously stored
    }
  } catch(e) {
    // error reading value
  }
}
```

Reading object value

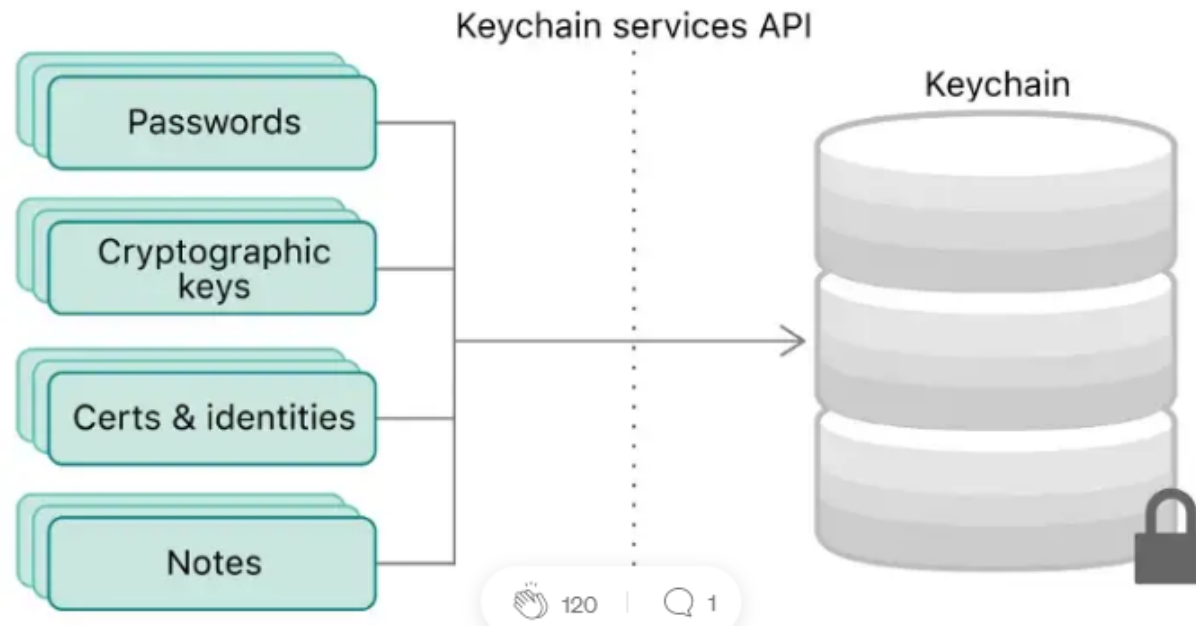
```
const getData = async () => {
  try {
    const jsonValue = await AsyncStorage.getItem('@storage_Key')
    return jsonValue != null ? JSON.parse(jsonValue) : null;
  } catch(e) {
    // error reading value
  }
}
```


code

```
▶ /**
▶  * Sample React Native App
▶  * https://github.com/facebook/react-native
▶  *
▶  * @format
▶  * @flow strict-local
▶  */
▶
▶ import React,{useState} from 'react';
▶
▶ import { View,Button,Text,TextInput} from 'react-native';
▶
▶ import AsyncStorage from '@react-native-async-storage/async-storage';
▶
▶ export default App=()=>{
▶
▶   const [myemail,setemail]=useState(null)
▶
▶   const [mypassword,setpassword]=useState(null)
▶
▶   person={email:myemail,password:mypassword}
▶
▶
▶   const storedata=async (value)=>{
▶
▶     try{
▶
▶       const data=JSON.stringify(value)
▶
▶       await AsyncStorage.setItem("mydata",data)}
▶
▶       catch(err){console.log(err)}
▶
▶     }
▶
▶   const retrievedata=async (key)=>{
▶
▶     try{
```

Secure Storage

Secure storage helps to store **encrypted data**. React Native does not come bundled with any way of storing sensitive data. However, there are pre-existing solutions for Android and iOS platforms.



On iOS, Keychain Services allows to securely store small chunks of sensitive info of the app. On Android, Shared Preference is the equivalent of a persistent key-value data store used for secure storage. Data in Shared Preferences is not encrypted by default, but Encrypted Shared Preferences wrap the Shared Preferences class for Android, and automatically encrypts keys and values.

But Android has another secure option than Shared Preferences called Android Keystore system that is used to store cryptographic keys in a container to make it more difficult to extract from the device. However, a branch of react-native-sensitive-info uses Android Keystore.

The ideal place to store certificates, tokens, passwords, and any other sensitive information that doesn't belong in Async Storage.



MMKV Storage

MMKV is an **efficient, small mobile key-value** storage framework that was developed by Tencent to use in WeChat.

MMKV uses mmap to keep memory synced with files, and *protobuf* to encode/decode values, making the most of Android to achieve the best efficiency performance. It supports concurrent read-read and read-write access between processes which allows multi-process concurrency. It is easy to keep up the data because of fully synchronous calls.

The ideal place to store common data of users, app-logic, and others. It is an **alternative for Async Storage**.

<https://github.com/mrousavy/react-native-mmkv>

<https://github.com/ammarahm-ed/react-native-mmkv-storage>

SQLite Storage

SQLite is a C-language library that implements a **small, fast, self-contained, high-reliability, full-featured, SQL database engine**. It is the most used database engine. It is **built into all mobile phones** and most computers and comes bundled inside countless other apps that people use every day. The file format is stable, cross-platform, and backward compatible and the developers pledge to keep it that way.

The ideal place to store more data than Async, Secure, and MMKV storage and it can support offline app development.

<https://github.com/andpor/react-native-sqlite-storage>



Database Services

There are different types of database services available to perform various functionalities of the data layer of the mobile apps by following different approaches. Those are listed here.

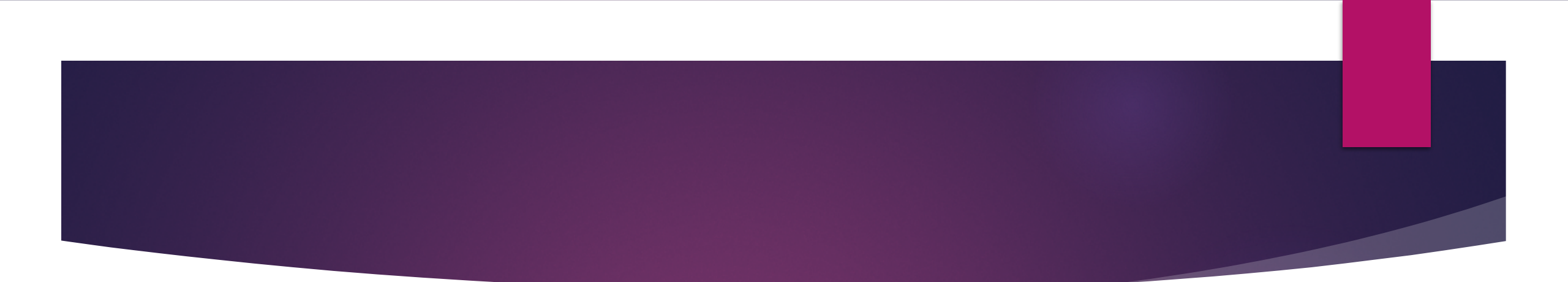
1. Firebase Firestore
2. Firebase Database
3. Firebase Storage
4. Realm by Mongo DB
5. Pouch DB

FireBase

- ▶ Firebase is cloud base service provided by google
- ▶ Firebase (a NoSQL JSON database) is a real-time database that allows storing a list of objects in the form of a tree. We can synchronize data between different devices.
- ▶ Firebase is a mobile backend-as-a-service that provides powerful features for building mobile apps. Firebase has three core services:
- ▶ Realtime database
- ▶ User authentication
- ▶ Hosting

Key Differences between SQL and NoSQL

- ▶ Key Differences between SQL and NoSQL
- ▶ SQL pronounced as “S-Q-L” or as “See-Quel” is primarily called RDBMS or Relational Databases, whereas NoSQL is a Non-relational or Distributed Database.
- ▶ Comparing SQL vs NoSQL databases, SQL databases are table-based databases, whereas NoSQL databases can be document-based, key-value pairs, and graph databases.
- ▶ SQL databases are vertically scalable, while NoSQL databases are horizontally scalable.
- ▶ SQL databases have a predefined schema, whereas NoSQL databases use a dynamic schema for unstructured data.

- 
- ▶ NoSQL is a non-relational DMS, that does not require a fixed schema, avoids joins, and is easy to scale. NoSQL database is used for distributed data stores with humongous data storage needs. NoSQL is used for Big data and real-time web apps. For example companies like Twitter, Facebook, Google that collect terabytes of user data every single day.
 - ▶ NoSQL database stands for “Not Only SQL” or “Not SQL.” Though a better term would NoREL NoSQL caught on. Carl Stroz introduced the NoSQL concept in 1998.
 - ▶ Traditional RDBMS uses SQL syntax to store and retrieve data for further insights. Instead, a NoSQL database system encompasses a wide range of database technologies that can store structured, semi-structured, unstructured and polymorphic data.
 - ▶ NoSQL is a non-relational database, meaning it allows different structures than a SQL database (not rows and columns) and more flexibility to use a format that best fits the data. The term “NoSQL” was not coined until the early 2000s. It doesn’t mean the systems don’t use SQL, as NoSQL databases do sometimes support some SQL commands. More accurately, “NoSQL” is sometimes defined as “not only SQL.”

Difference: Sql

Scalability

In general, SQL databases can scale vertically, meaning you can increase the load on a server by migrating to a larger server that adds more CPU, RAM or [SSD](#) capability. While vertical scalability is used most frequently, SQL databases can also scale horizontally through sharding or partitioning logic, although that's not well-supported.

Structure

SQL database schema organizes data in relational, tabular ways, using tables with columns or attributes and rows of records. Because SQL works with such a strictly predefined schema, it requires organizing and structuring data before starting with the SQL database.

Sql

Properties

RDBMS, which use SQL, must exhibit four properties, known by the acronym ACID. These ensure that transactions are processed successfully and that the SQL database has a high level of reliability:

- **Atomicity:** All transactions must succeed or fail completely and cannot be left partially complete, even in the case of system failure.
- **Consistency:** The database must follow rules that validate and prevent corruption at every step.
- **Isolation:** Concurrent transactions cannot affect each other.
- **Durability:** Transactions are final, and even system failure cannot “roll back” a complete transaction.

No SQL

Unlike SQL, NoSQL systems allow you to work with different data structures within a database. Because they allow a dynamic schema for unstructured data, there's less need to pre-plan and pre-organize data, and it's easier to make modifications. NoSQL databases allow you to add new attributes and fields, as well as use varied syntax across databases.

Scalability

NoSQL databases scale better horizontally, which means one can add additional servers or nodes as needed to increase load.

Structure

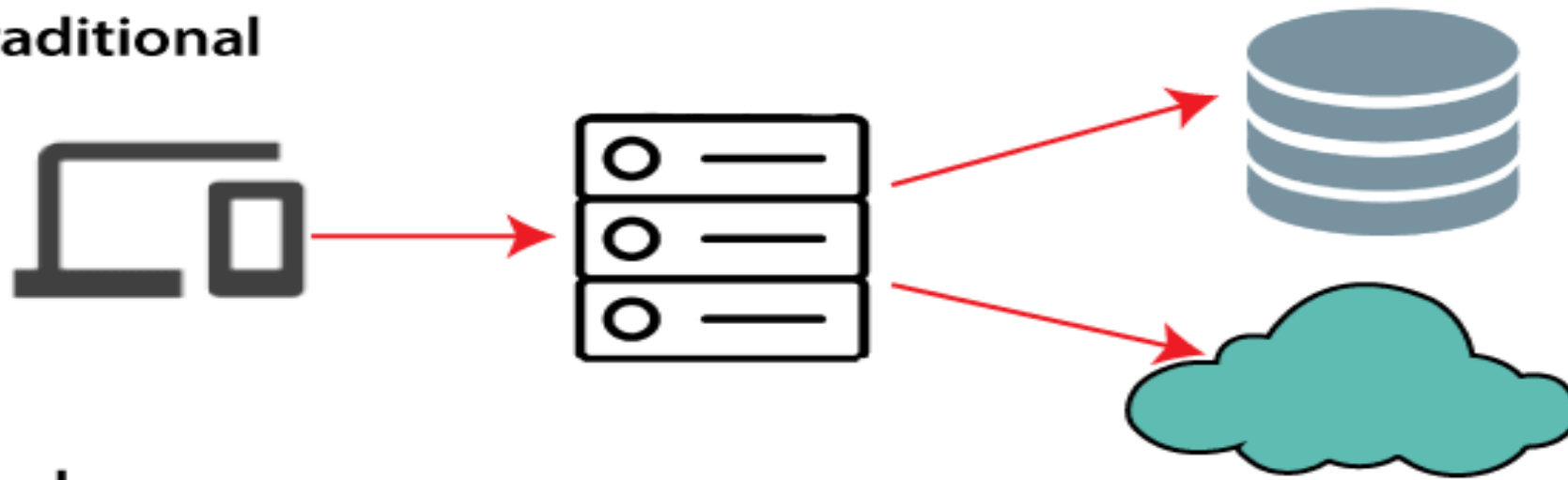
NoSQL databases are not relational, so they don't solely store data in rows and tables. Instead, they generally fall into one of four types of structures:

- **Column-oriented**, where data is stored in cells grouped in a virtually unlimited number of columns rather than rows.
- **Key-value stores**, which use an associative array (also known as a dictionary or map) as their data model. This model represents data as a collection of key-value pairs.
- **Document stores**, which use documents to hold and encode data in standard formats, including XML, YAML, JSON (JavaScript Object Notation) and BSON. A benefit is that documents within a single database can have different data types.
- **Graph databases**, which represent data on a graph that shows how different sets of data relate to each other. Neo4j, RedisGraph (a graph module built into Redis) and OrientDB are examples of graph databases.

	SQL Databases	NoSQL Databases
Data Storage Model	Tables with fixed rows and columns	Document: JSON documents, Key-value: key-value pairs, Wide-column: tables with rows and dynamic columns, Graph: nodes and edges
Development History	Developed in the 1970s with a focus on reducing data duplication	Developed in the late 2000s with a focus on scaling and allowing for rapid application change driven by agile and DevOps practices.
Examples	Oracle, MySQL, Microsoft SQL Server, and PostgreSQL	Document: MongoDB and CouchDB, Key-value: Redis and DynamoDB, Wide-column: Cassandra and HBase, Graph: Neo4j and Amazon Neptune

	SQL Databases	NoSQL Databases
Primary Purpose	General purpose	Document: general purpose, Key-value: large amounts of data with simple lookup queries, Wide-column: large amounts of data with predictable query patterns, Graph: analyzing and traversing relationships between connected data
Schemas	Rigid	Flexible
Scaling	Vertical (scale-up with a larger server)	Horizontal (scale-out across commodity servers)
Multi-Record ACID Transactions	Supported	Most do not support multi-record ACID transactions. However, some — like MongoDB — do.
Joins	Typically required	Typically not required
Data to Object Mapping	Requires ORM (object-relational mapping)	Many do not require ORMs. MongoDB documents map directly to data structures in most popular programming languages.

Traditional



Firebase



Google Firebase is Google-backed application development software which allows developers to develop **Android, IOS, and Web apps**. For reporting and fixing app crashes, tracking analytics, creating marketing and product experiments, firebase provides several tools.

DEVELOP



Realtime Database



Authentication



Cloud
Messaging



Storage



Hosting



Test Lab



Crash
Reporting

GROW

Notifications



Remote Config



App Indexing



Dynamic Links



Invites



AdWords



EARN



AdMod

Analytics



Firebase Authentication

- ▶ Most apps need to know the identity of a user. Knowing a user's identity allows an app to securely save user data in the cloud and provide the same personalized experience across all of the user's devices.
- ▶ Firebase Authentication provides backend services, easy-to-use SDKs, and ready-made UI libraries to authenticate users to your app. It supports authentication using passwords, phone numbers, popular federated identity providers like Google, Facebook and Twitter, and more.
- ▶ Firebase Authentication integrates tightly with other Firebase services, and it leverages industry standards like OAuth 2.0 and OpenID Connect, so it can be easily integrated with your custom backend.

Key capabilities

You can sign in users to your Firebase app either by using FirebaseUI as a complete drop-in auth solution or by using the Firebase Authentication SDK to manually integrate one or several sign-in methods into your app.

FirebaseUI Auth

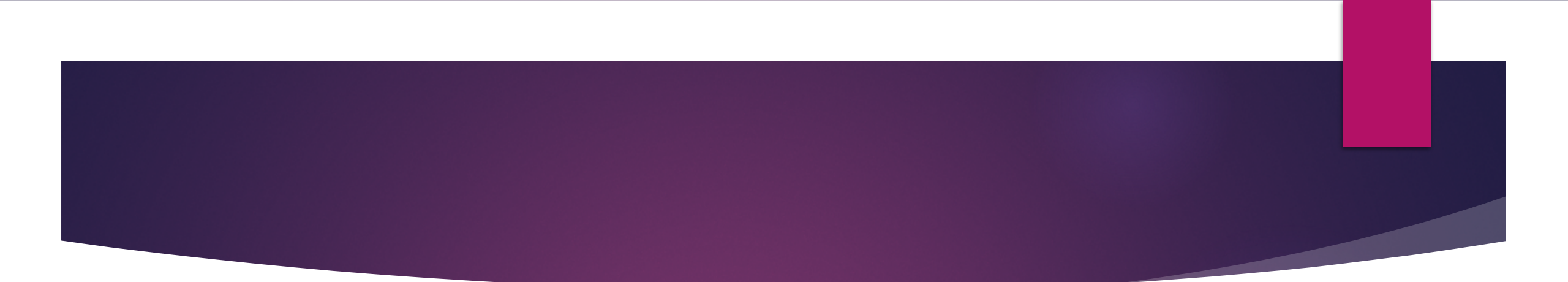
Drop-in authentication solution

The recommended way to add a complete sign-in system to your app.

FirebaseUI provides a drop-in auth solution that handles the UI flows for signing in users with email addresses and passwords, phone numbers, and with popular federated identity providers, including Google Sign-In and Facebook Login.

The FirebaseUI Auth component implements best practices for authentication on mobile devices and websites, which can maximize sign-in and sign-up conversion for your app. It also handles edge cases like account recovery and account linking that can be security sensitive and error-prone to handle correctly.

FirebaseUI can be easily customized to fit in with the rest of your app's visual style, and it is open source, so you aren't constrained in realizing the user experience you want.



Firestore SDK Authentication

Email and password based authentication

Authenticate users with their email addresses and passwords. The Firebase Authentication SDK provides methods to create and manage users that use their email addresses and passwords to sign in. Firebase Authentication also handles sending password reset emails.

[iOS](#) [Android](#) [Web](#) [C++](#) [Unity](#)

Firestore Database

- ▶ Store and sync data with our NoSQL cloud database. Data is synced across all clients in realtime, and remains available when your app goes offline.

Firestore offers two cloud-based, client-accessible database solutions that support realtime data syncing:

- **Cloud Firestore** is Firestore's newest database for mobile app development. It builds on the successes of the Realtime Database with a new, more intuitive data model. Cloud Firestore also features richer, faster queries and scales further than the Realtime Database.
- **Realtime Database** is Firestore's original database. It's an efficient, low-latency solution for mobile apps that require synced states across clients in realtime.



Which database does Firebase recommend?

Your choice of database solution will depend on many factors, but when it comes to certain features, we can make recommendations about which database is right for you.

Both solutions offer:

- Client-first SDKs, with no servers to deploy and maintain
- Realtime updates
- Free tier, then pay for what you use

Data model

Both Realtime Database and Cloud Firestore are NoSQL Databases.

Realtime Database	Cloud Firestore
<p>Stores data as one large JSON tree.</p> <ul style="list-style-type: none">• Simple data is very easy to store.• Complex, hierarchical data is harder to organize at scale. <p>Learn more about the Realtime Database data model.</p>	<p>Stores data as collections of documents.</p> <ul style="list-style-type: none">• Simple data is easy to store in documents, which are very similar to JSON.• Complex, hierarchical data is easier to organize at scale, using subcollections within documents.• Requires less denormalization and data flattening. <p>Learn more about the Cloud Firestore data model.</p>

Realtime and offline support

Both have mobile-first, realtime SDKs and both support local data storage for offline-ready apps.

Realtime Database	Cloud Firestore
Offline support for Apple and Android clients.	Offline support for Apple, Android, and web clients.



Presence

It can be useful to know when a client is online or offline. Firebase Realtime Database can record client connection status and provide updates every time the client's connection state changes.

Realtime Database	Cloud Firestore
Presence supported.	Not supported natively. You can leverage Realtime Database's support for presence by syncing Cloud Firestore and Realtime Database using Cloud Functions. See Build presence in Cloud Firestore .

Querying

Retrieve, sort, and filter data from either database through queries.

Realtime Database	Cloud Firestore
<p>Deep queries with limited sorting and filtering functionality.</p> <ul style="list-style-type: none">• Queries can sort or filter on a property, but not both.• Queries are deep by default: they always return the entire subtree.• Queries can access data at any granularity, down to individual leaf-node values in the JSON tree.• Queries do not require an index; however the performance of certain queries degrades as your data set grows.	<p>Indexed queries with compound sorting and filtering.</p> <ul style="list-style-type: none">• You can chain filters and combine filtering and sorting on a property in a single query.• Queries are shallow: they only return documents in a particular collection or collection group and do not return subcollection data.• Queries must always return whole documents.• Queries are indexed by default: Query performance is proportional to the size of your result set, not your data set.

Writes and transactions

Realtime Database

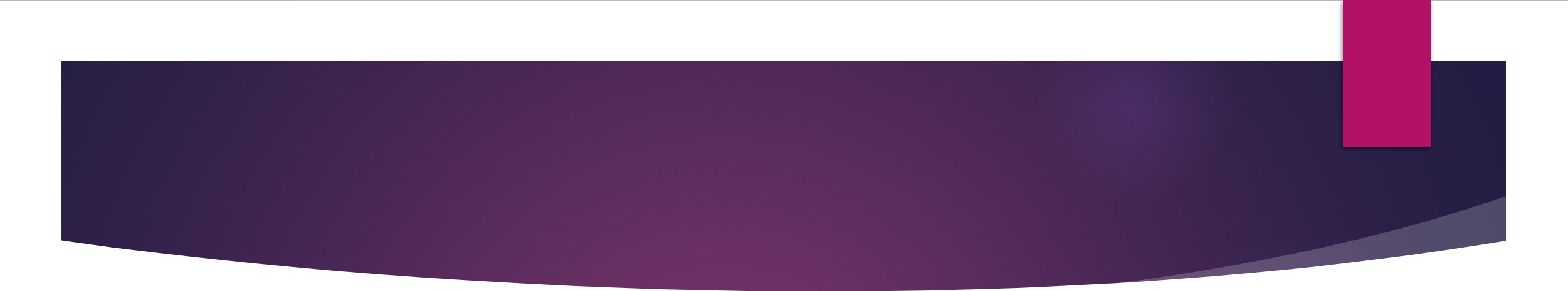
Basic write and transaction operations.

- [Write data](#) through set and update operations.
- [Transactions](#) are atomic on a specific data subtree.

Cloud Firestore

Advanced write and transaction operations.

- [Write data operations](#) through set and update operations as well as advanced transformations such as array and numeric operators.
- [Transactions](#) can atomically read and write data from any part of the database.



Reliability and performance

Realtime Database	Cloud Firestore
<p>Realtime Database is a regional solution.</p> <ul style="list-style-type: none">• Available in regional configurations. Databases are limited to zonal availability within a region.• Extremely low latency, ideal option for frequent state-syncing. <p>Read more about Realtime Database performance and reliability characteristics in the Service Level Agreement.</p>	<p>Cloud Firestore is a regional and multi-region solution that scales automatically.</p> <ul style="list-style-type: none">• Houses your data across multiple data centers in distinct regions, ensuring global scalability and strong reliability.• Available in regional or multi-regional configurations around the world. <p>Read more about Cloud Firestore performance and reliability characteristics in the Service Level Agreement.</p>

Security

Realtime Database

Cascading rules language that separates authorization and validation.

- Reads and writes from mobile SDKs secured by [Realtime Database Rules](#).
- Read and write rules cascade.
- You [validate data](#) separately using the `validate` rule.

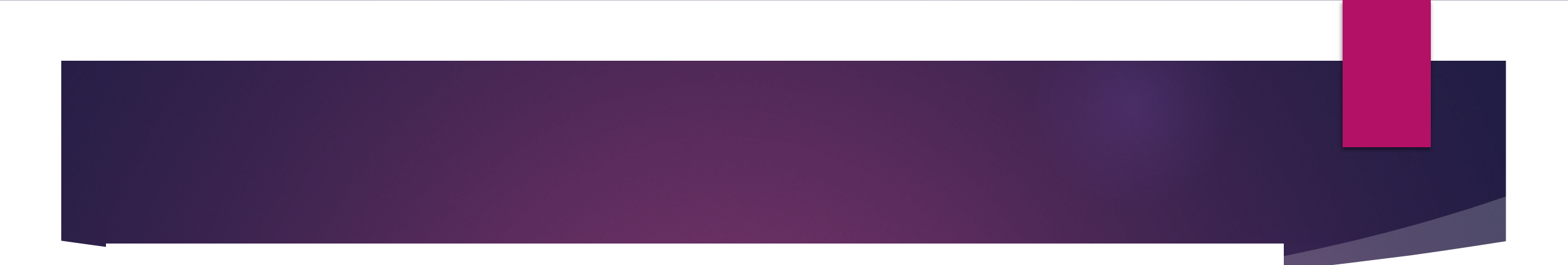
Cloud Firestore

Non-cascading rules that combine authorization and validation.

- Reads and writes from mobile SDKs secured by [Cloud Firestore Security Rules](#).
- Reads and writes from server SDKs secured by [Identity and Access Management \(IAM\)](#).
- Rules don't cascade unless you use a wildcard.
- Rules can constrain queries: If a query's results might contain data the user doesn't have access to, the entire query fails.

Cloud Firestore

- ▶ Cloud Firestore is a flexible, scalable database for mobile, web, and server development from Firebase and Google Cloud. Like Firebase Realtime Database, it keeps your data in sync across client apps through realtime listeners and offers offline support for mobile and web so you can build responsive apps that work regardless of network latency or Internet connectivity. Cloud Firestore also offers seamless integration with other Firebase and Google Cloud products, including Cloud Functions.



Key capabilities

Flexibility

The Cloud Firestore data model supports flexible, hierarchical data structures. Store your data in documents, organized into collections. Documents can contain complex nested objects in addition to subcollections.

Expressive querying

In Cloud Firestore, you can use queries to retrieve individual, specific documents or to retrieve all the documents in a collection that match your query parameters. Your queries can include multiple, chained filters and combine filtering and sorting. They're also indexed by default, so query performance is proportional to the size of your result set, not your data set.

Realtime updates

Like Realtime Database, Cloud Firestore uses data synchronization to update data on any connected device. However, it's also designed to make simple, one-time fetch queries efficiently.

Offline support

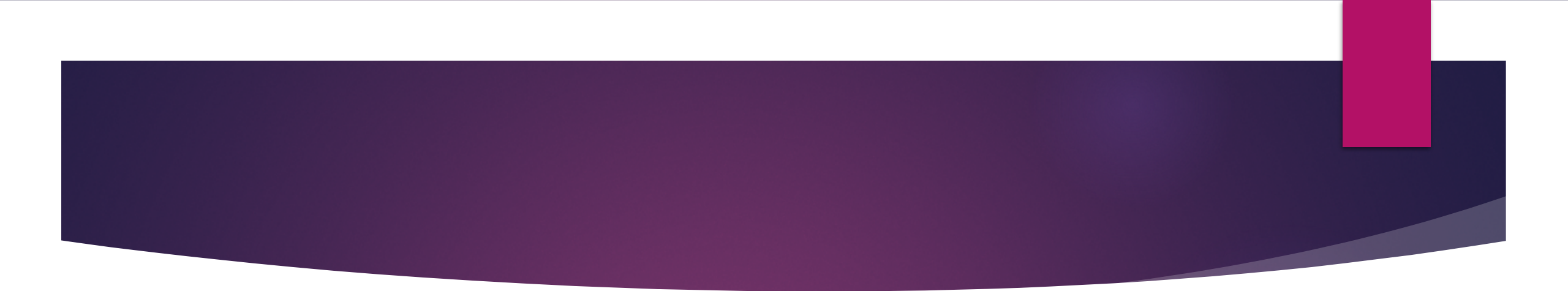
Cloud Firestore caches data that your app is actively using, so the app can write, read, listen to, and query data even if the device is offline. When the device comes back online, Cloud Firestore synchronizes any local changes back to Cloud Firestore.

Designed to scale

Cloud Firestore brings you the best of Google Cloud's powerful infrastructure: automatic multi-region data replication, strong consistency guarantees, atomic batch operations, and real transaction support. We've designed Cloud Firestore to handle the toughest database workloads from the world's biggest apps.

How does it work?

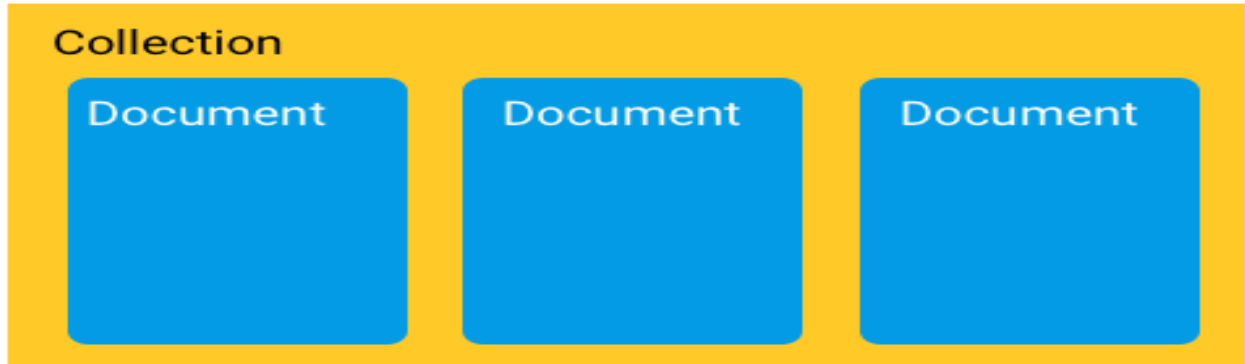
- ▶ Cloud Firestore is a cloud-hosted, NoSQL database that your Apple, Android, and web apps can access directly via native SDKs. Cloud Firestore is also available in native Node.js, Java, Python, Unity, C++ and Go SDKs, in addition to REST and RPC APIs.
- ▶ Following Cloud Firestore's NoSQL data model, you store data in documents that contain fields mapping to values. These documents are stored in collections, which are containers for your documents that you can use to organize your data and build queries. Documents support many different data types, from simple strings and numbers, to complex, nested objects. You can also create subcollections within documents and build hierarchical data structures that scale as your database grows. The Cloud Firestore data model supports whatever data structure works best for your app.

- 
- ▶ Additionally, querying in Cloud Firestore is expressive, efficient, and flexible. Create shallow queries to retrieve data at the document level without needing to retrieve the entire collection, or any nested subcollections. Add sorting, filtering, and limits to your queries or cursors to paginate your results. To keep data in your apps current, without retrieving your entire database each time an update happens, add realtime listeners. Adding realtime listeners to your app notifies you with a data snapshot whenever the data your client apps are listening to changes, retrieving only the new changes.
 - ▶ Protect access to your data in Cloud Firestore with Firebase Authentication and Cloud Firestore Security Rules for Android, Apple platforms, and JavaScript, or Identity and Access Management (IAM) for server-side languages.

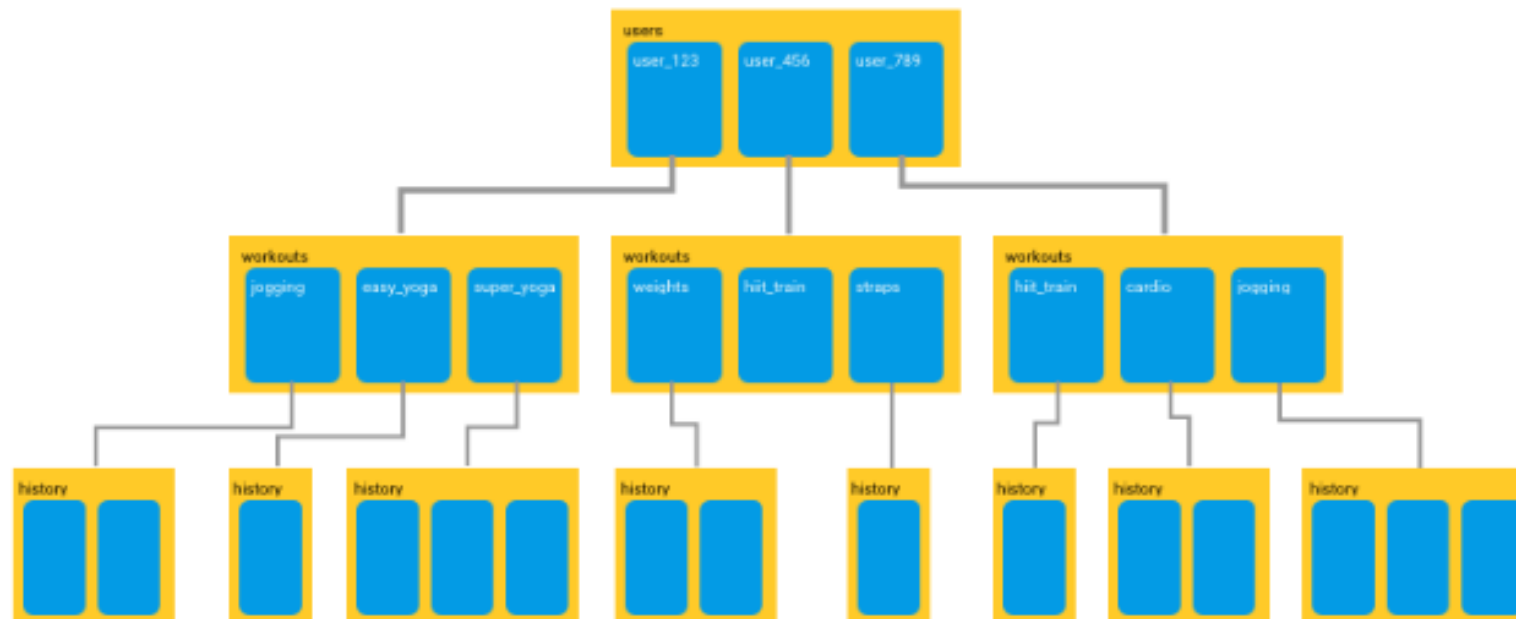
Cloud Firestore

Better querying and more structured data

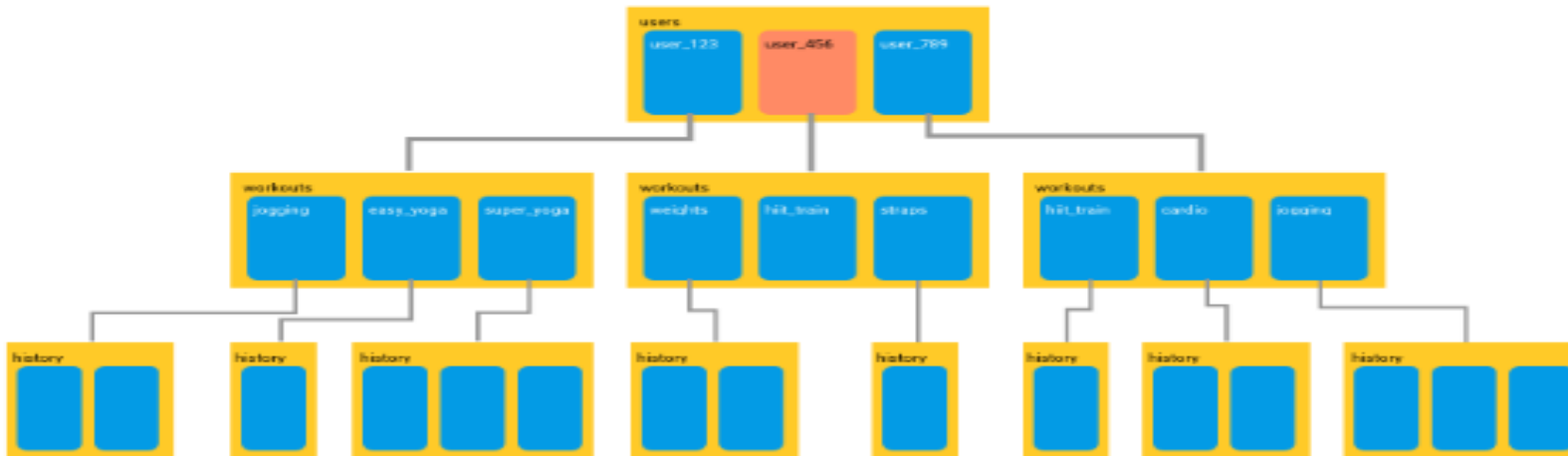
While the Firebase Realtime Database is basically a giant JSON tree where anything goes and lawlessness rules the land^[^1], Cloud Firestore is more structured. Cloud Firestore is a document-model database, which means that all of your data is stored in objects called *documents* that consist of key-value pairs — and these values can contain any number of things, from strings to floats to binary data to JSON-y looking objects the team likes to call *maps*. These documents, in turn, are grouped into *collections*.



Your Cloud Firestore database will probably consist of a few collections that contain documents that point to subcollections. These subcollections will contain documents that point to other subcollections, and so on.



For starters, all queries are *shallow*, meaning that you can simply fetch a document without having to fetch all of the data contained in any of the linked subcollections. This means you can store your data hierarchically in a way that makes sense logically without worrying about downloading tons of unnecessary data.



In this example, the document at the top can be fetched without grabbing any of the documents in the subcollections below

Second, Cloud Firestore has more powerful querying capabilities than the Realtime Database. In the Realtime Database, trying to create a query across multiple fields was a lot of work and usually involved denormalizing your data.

For example, imagine you had a list of cities, and you wanted to find a list of all cities in California with a population greater than 500k.



Cities, stored in the Realtime Database

Cities, stored in the Realtime Database

In the Realtime Database, you'd need to conduct this search by creating an explicit "states plus population" field and then running a query sorted on that field.



```
citiesRef.orderByChild('state_and_pop')
    .startAt('California_00500000')
    .endAt('California_99999999');
```

Creating a combined state_and_population field, just for queries

With Cloud Firestore, this work is no longer necessary. In some cases, Cloud Firestore can automatically search across multiple fields. In other cases, like our cities example, Cloud Firestore will guide you towards automatically building an index required to make these kinds of queries possible...

Add composite index

Composite indexes are required for queries that include specific values and a range or order. [Learn more](#)

Index	Collection(s)	Fields
	cities	↑ state ↑ population

CANCELCREATE INDEX

Creation time depends on the size of your data

...and then you can simply search across multiple fields.

Cities

city_123
name: "San Francisco"
population: 878887
state: "California"

city_456
name: "Boston"
population: 673184
state: "Massachusetts"

...

```
citiesRef
.where("state", "==", "California")
.where("population", ">", 500000)
```

Cloud Firestore will automatically maintain this index for you throughout the lifetime of your app. No combo fields required!

Designed to Scale

While the Realtime Database does scale to meet the needs of many apps, things can start to get difficult when your app becomes really popular, or your dataset gets truly massive.

Cloud Firestore, on the other hand, is built on top of the same Google Cloud infrastructure that powers some [pretty popular apps](#). So it will be able to scale much more easily and to a much greater capacity than the Realtime Database can. While the Realtime Database tops out at about 100,000 concurrent connections, for instance, Cloud Firestore will accept up to 1,000,000 concurrent client connections per database. For a complete list of Cloud Firestore's limits, be sure to visit the [documentation](#).

Cloud Firestore also has a more robust Service Level Agreement than the Realtime Database. Cloud Firestore guarantees 99.999% uptime in multi-region instances (more on that below) and 99.99% uptime in regional instances. The Realtime Database, by contrast, guarantees 99.95% uptime.

And with the new querying structure, all Cloud Firestore queries scale to the size of your result set — not the size of your data. This means that a search for the top 10 restaurants in Chicago for a restaurant review app will take the same amount of time whether your database has 300 restaurants, 300 thousand or 30 million. As one engineer here likes to put it, "It's basically impossible to create a slow query in Cloud Firstore."

Installing Dependencies

Installation

<https://rnfirebase.io/firestore/usage>

This module requires that the `@react-native-firebase/app` module is already setup and installed. To install the "app" module, view the [Getting Started](#) documentation.

```
# Install & setup the app module  
yarn add @react-native-firebase/app  
  
# Install the firestore module  
yarn add @react-native-firebase/firestore  
  
# If you're developing your app using iOS, run this command  
cd ios/ && pod install
```

If you're using an older version of React Native without autolinking support, or wish to integrate into an existing project, you can follow the manual installation steps for [iOS](#) and [Android](#).

If you have started to receive a `app:mergeDexDebug` error after adding Cloud Firestore, please read the [Enabling Multidex](#) documentation for [more information on](#) how to resolve this error.

🔗 What does it do

Firestore is a flexible, scalable NoSQL cloud database to store and sync data. It keeps your data in sync across client apps through realtime listeners and offers offline support so you can build responsive apps that work regardless of network latency or Internet connectivity.

🔗 Usage

🔗 Collections & Documents

Cloud Firestore stores data within "documents", which are contained within "collections", and documents can also contain collections. For example, we could store a list of our users documents within a "Users" collection. The `collection` method allows us to reference a collection within our code:

```
import firestore from '@react-native-firebase/firestore';  
  
const usersCollection = firestore().collection('Users');
```

The `collection` method returns a `CollectionReference` class, which provides properties and methods to query and fetch the data from Cloud Firestore. We can also directly reference a single document on the collection by calling the `doc` method:

```
import firestore from '@react-native-firebase/firestore';  
  
// Get user document with an ID of ABC  
const userDocument = firestore().collection('Users').doc('ABC');
```

The `doc` method returns a `DocumentReference`.

A document can contain different types of data, including scalar values (strings, booleans, numbers), arrays (lists) and objects (maps) along with specific Cloud Firestore data such as `Timestamps`, `GeoPoints`, `Blobs` and more.

Read Data

Cloud Firestore provides the ability to read the value of a collection or document. This can be read one-time, or provide realtime updates when the data within a query changes.

One-time read

To read a collection or document once, call the `get` method on a `CollectionReference` or `DocumentReference`:

```
import firestore from '@react-native-firebase/firestore';

const users = await firestore().collection('Users').get();
const user = await firestore().collection('Users').doc('ABC').get();
```


Snapshots

Once a query has returned a result, Firestore returns either a `QuerySnapshot` (for collection queries) or a `DocumentSnapshot` (for document queries). These snapshots provide the ability to view the data, view query metadata (such as whether the data was from local cache), whether the document exists or not and more.

QuerySnapshot

A `QuerySnapshot` returned from a collection query allows you to inspect the collection, such as how many documents exist within it, access to the documents within the collection, any changes since the last query and more.

To access the documents within a `QuerySnapshot`, call the `forEach` method:

QuerySnapshot

A `QuerySnapshot` returned from a collection query allows you to inspect the collection, such as how many documents exist within it, access to the documents within the collection, any changes since the last query and more.

To access the documents within a `QuerySnapshot`, call the `forEach` method:

```
import firestore from '@react-native-firebase/firestore';

firestore()
  .collection('Users')
  .get()
  .then(querySnapshot => {
    console.log('Total users: ', querySnapshot.size);

    querySnapshot.forEach(documentSnapshot => {
      console.log('User ID: ', documentSnapshot.id, documentSnapshot.data());
    });
  });
```

Each child document of a `QuerySnapshot` is a `QueryDocumentSnapshot`, which allows you to access specific information about a document (see below).

DocumentSnapshot

A `DocumentSnapshot` is returned from a query to a specific document, or as part of the documents returned via a `QuerySnapshot`. The snapshot provides the ability to view a documents data, metadata and whether a document actually exists.

To view a documents data, call the `data` method on the snapshot:

```
import firestore from '@react-native-firebase/firestore';

firestore()
  .collection('Users')
  .doc('ABC')
  .get()
  .then(documentSnapshot => {
    console.log('User exists: ', documentSnapshot.exists);

    if (documentSnapshot.exists) {
      console.log('User data: ', documentSnapshot.data());
    }
  });
```

A snapshot also provides a helper function to easily access deeply nested data within a document. Call the `get` method with a dot-notated path:

```
function getUserZipCode(documentSnapshot) {  
  return documentSnapshot.get('info.address.zipcode');  
}  
  
firestore()  
  .collection('Users')  
  .doc('ABC')  
  .get()  
  .then(documentSnapshot => getUserZipCode(documentSnapshot))  
  .then(zipCode => {  
    console.log('Users zip code is: ', zipCode);  
  }));
```

🔗 Writing Data

The [Firebase documentation](#) provides great examples on best practices on how to structure your data. We highly recommend reading the guide before building out your database.

For a more in-depth look at what is possible when writing data to Firestore please refer to this [documentation](#)

🔗 Adding documents

To add a new document to a collection, use the `add` method on a [CollectionReference](#):

```
import firestore from '@react-native-firebase/firestore';

firestore()
  .collection('Users')
  .add({
    name: 'Ada Lovelace',
    age: 30,
  })
  .then(() => {
    console.log('User added!');
  });
```

The `add` method adds the new document to your collection with a random unique ID. If you'd like to specify your own ID, call the `set` method on a `DocumentReference` instead:

```
import firestore from '@react-native-firebase/firestore';

firestore()
  .collection('Users')
  .doc('ABC')
  .set({
    name: 'Ada Lovelace',
    age: 30,
  })
  .then(() => {
    console.log('User added!');
  });
```


Code

```
▶ /**
▶  * Sample React Native App
▶  * https://github.com/facebook/react-native
▶  *
▶  * @format
▶  * @flow strict-local
▶  */
▶
▶ import React,{useState} from 'react';
▶
▶ import {
▶   SafeAreaView,
▶   ScrollView,
▶   StatusBar,
▶   StyleSheet,
▶   Text,Button,
▶   useColorScheme,
▶   View,
▶ } from 'react-native';
▶
▶ import firestore from '@react-native-firebase/firestore';
▶
▶ import { firebase} from '@react-native-firebase/firestore';
▶
▶ const firebaseConfig = {
▶   apiKey: "AIzaSyBbNCbVMjcPclIgilMu6g3nixVhLpr-17s",
▶   authDomain: "reactnativeproject-d1681.firebaseio.com",
```