THE ANGULAR FIREBASE SURVIVAL GUIDE

Book and Videos By

JEFF DELANEY

version
6.0

including
FIRESTORE

The Angular Firebase Survival Guide

Build Angular Apps on a Solid Foundation with Firebase

Jeff Delaney

This book is for sale at http://leanpub.com/angularfirebase

This version was published on 2018-05-23

To my loving wife, you inspire me daily.

Contents

Introduction

The Angular Firebase Survival Guide is about getting stuff done. No effort is made to explicitly cover high level programming theories or low level Angular architecture concepts – there are plenty of other books for that purpose. The focus of this book is **building useful app features**. Each section starts with a problem statement, then solves it with code.

Even for experienced JavaScript developers, the learning curve for Angular is quite steep. Mastering this framework is only possible by putting forth the effort to build your own features from scratch. Your journey will inevitably lead to moments of frustration - you may even dream about switching to VueJS or React - but this is just part of the learning process. Once you have Angular down, you will arrive among a rare class of developers who can build enterprise-grade realtime apps for web, mobile, and desktop.

The mission of this book is to provide a diverse collection of snippets (recipes) that demonstrate the combined power of Angular and Firebase. The format is non-linear, so when a client asks you to build a "Custom Username" feature, you can jump to section 6.1 and start coding. By the end of the book, you will know how to authenticate users, handle realtime data streams, upload files, trigger background tasks with cloud functions, process payments, and much more.

I am not sponsored by any of the brands or commercial services mentioned in this book. I recommend these tools because I am confident in their efficacy through my experience as a web development consultant.

Why Angular?

Angular can produce maintainable cross-platform JavaScript apps that deliver an awesome user experience. It's open source, backed by Google, has excellent developer tooling via TypeScript, a large community of developers, and is being adopted by large enterprises. I see more and more Angular2+ job openings every week.

Why Firebase?

Firebase eliminates the need for managed servers, scales automatically, dramatically reduces development time, is built on Google Cloud Platform, and is free for small apps.

Firebase is a Backend-as-a-Service (BaaS) that also offers Functions-as-a-Service (FaaS). The Firebase backend will handle your database, file storage, and authentication – features that would normally take weeks to develop from scratch. Cloud functions will run background tasks and microservices in a completely isolated NodeJS environment. On top of that, Firebase provides hosting with free SSL, analytics, and cloud messaging.

Furthermore, Firebase is evolving with the latest trends in web development. In March 2017, the platform introduced *Cloud Functions for Firebase*. Then in October 2017, the platform introduced the *Firestore Document Database*. I have been blown away at the sheer pace and quality of new feature roll-outs for the platform. Needless to say, I stay very busy keeping this book updated.

Why Angular and Firebase Together?

When you're a consultant or startup, it doesn't really matter what languages or frameworks you know. What does matter is **what** you can produce, **how fast** you can make it, and **how much** it will cost. Optimizing these variables forces you to choose a technology stack that won't disappoint. Angular does take time to learn (I almost quit), but when you master the core patterns, development time will improve rapidly. Adding Firebase to the mix virtually eliminates your backend maintenance worries and abstracts difficult aspects of app development - including user authentication, file storage, push notifications, and a realtime pub/sub database. The bottom line is that with Angular and Firebase you can roll out amazing apps quickly for your employer, your clients, or your own startup.

This Book is for Developers Who…

- Want to build real world apps
- Dislike programming books the size of War & Peace

- Have basic JavaScript (TypeScript), HTML, and SCSS skills
- Have some Angular experience – such as the demo on Angular.io
- Have a Firebase or GCP account
- Enjoy quick problem-solution style tutorials

Note for Native Mobile Developers

I am not going to cover the specifics of mobile or desktop frameworks, such as Ionic, Electron, NativeScript. However, most of the core principles and patterns covered in this book can be applied to native development.

Angular Firebase Starter App

To keep the recipes consistent, most of the code examples are centered around a book sharing app where users can post information about books and their authors.

The Firestarter App[1] provides an open-source live demo that shares much of its codebase with this book.

Package Versions

Change happens fast in the web development world. The package versions used in this book are as follows:

- Angular v6.0
- Angular CLI v6.0
- TypeScript v2.8
- Firebase JS SDK v5.0
- Firebase Functions v1.0
- Cloud Firestore vBeta

Everything else we build from the ground up.

Watch the Videos

The book is accompanied by an active YouTube channel that produces quick tutorials on relevant Angular solutions that you can start using right away. I will reference these videos throughout the book.

https://www.youtube.com/c/AngularFirebase

[1]https://github.com/codediodeio/angular-firestarter/blob/master/package.json

Join the Angular Firebase Slack Team

My goal is to help you ship your app as quickly as possible. To facilitate this goal, I would like to invite you to join our Slack room dedicated to Angular Firebase development. We discuss ideas, best practices, share code, and help each other get our apps production ready. Get the your Slack invite link here[2].

[2]https://angularfirebase.com

The Basics

The goal of the first chapter is discuss best practices and get your first app configured with Angular 4 and Firebase. By the end of the chapter you will have solid skeleton app from which we can start building more complex features.

1.1 Top Ten Best Practices

Problem

You want a few guidelines and best practices for building Angular apps with Firebase.

Solution

Painless development is grounded in a few core principles. Here are my personal top ten ten tips for Angular Firebase development.

1. Learn and use the Angular CLI.
2. Use AngularFire when working with Firebase.
3. Create generic services to handle data logic.
4. Create components/directives to handle data presentation.
5. Unwrap Observables in the template with the `async` pipe when practical.
6. Deploy your production app with Ahead-of-Time compilation to Firebase hosting.
7. Always define backend database and storage rules on Firebase.
8. Take advantage of TypeScript static typing features.
9. Setup separate Firebase projects for development and production.
10. Don't be afraid to use Lodash to simplify JavaScript.

1.2 Start a New App from Scratch

Problem

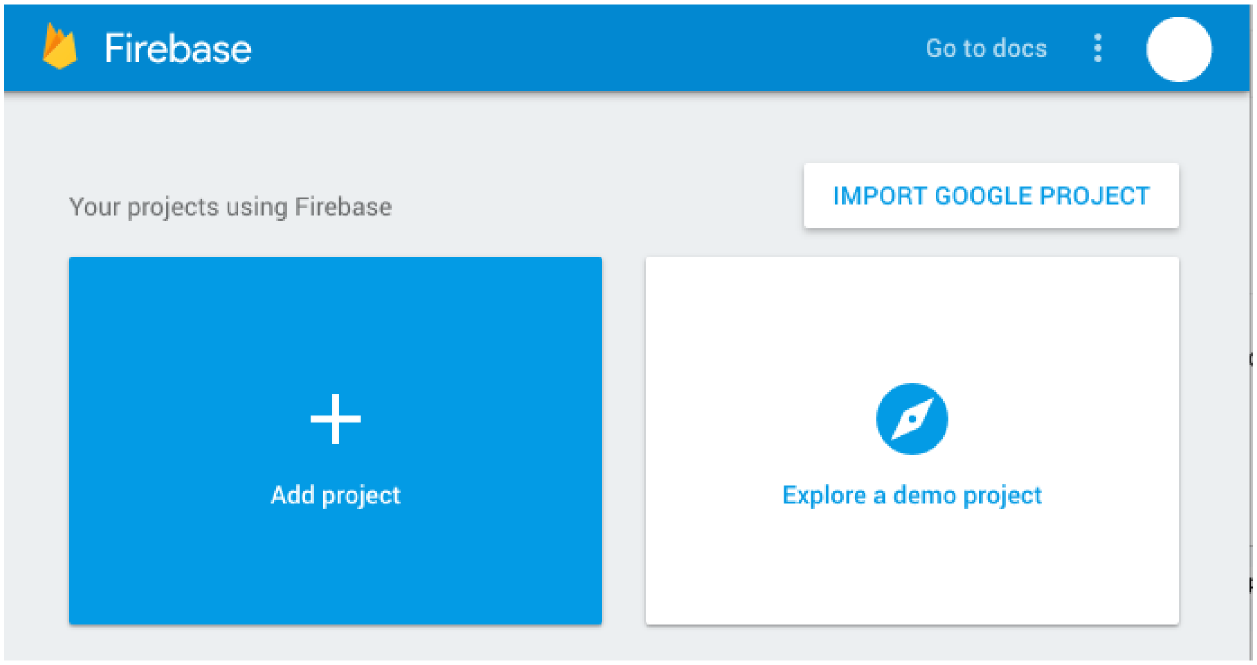
You want start a new Angular project, using Firebase for the backend.

Solution

Let's start with the bare essentials. (You may need to prefix commands with `sudo`).

```
npm install -g @angular/cli@latest
npm install -g typescript
npm install -g firebase-tools
```

Then head over to https://firebase.com and create a new project.



Setting up an Angular app with Firebase is easy. We are going to build the app with the Angular CLI, specifying the routing module and SCSS for styling. Let's name the app *fire*.

```
ng new fire --routing --style scss
cd fire
```

Next, we need to get AngularFire2, which includes Firebase as a dependency.

```
npm install angularfire2 firebase --save
```

In the `environments/environment.ts`, add your credentials. Make sure to keep this file private by adding it to `.gitignore`. You don't want it exposed in a public git repo.

```
export const environment = {
  production: false,
  firebaseConfig: {
    apiKey: '<your-key>',
    authDomain: '<your-project-authdomain>',
    databaseURL: '<your-database-URL>',
    projectId: '<your-project-id>',
    storageBucket: '<your-storage-bucket>',
    messagingSenderId: '<your-messaging-sender-id>'
  }
};
```

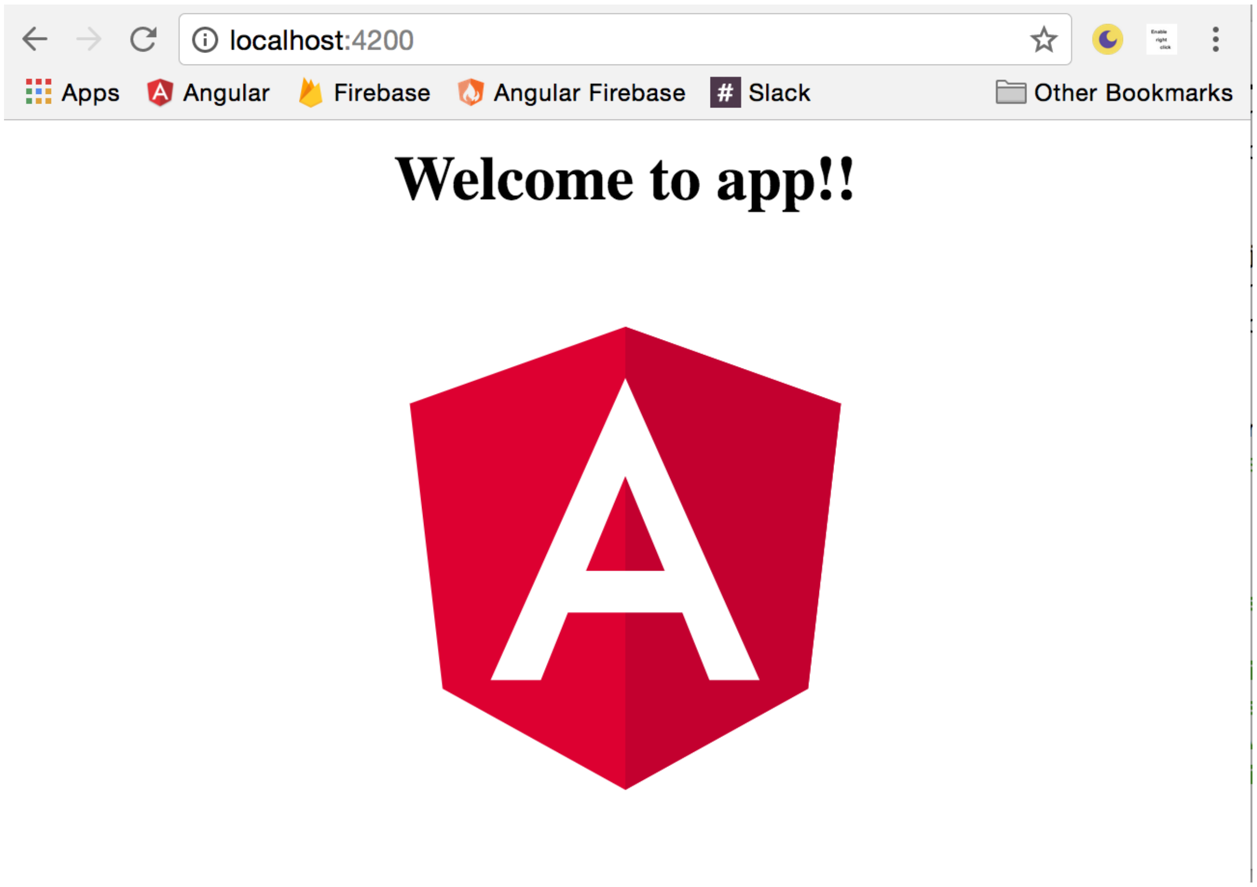
In the `app.module.ts`, add AngularFire2 to the imports. You only need to import the modules you plan on using.

```
import { AngularFireModule } from 'angularfire2';
import { AngularFireDatabaseModule } from 'angularfire2/database';
import { AngularFireAuthModule } from 'angularfire2/auth';
import { AngularFirestoreModule } from 'angularfire2/firestore';
import { AngularFireStorageModule } from 'angularfire2/storage';

import { environment } from '../environments/environment';
export const firebaseConfig = environment.firebaseConfig;
// ...omitted
@NgModule({
  imports: [
    BrowserModule,
    AppRoutingModule,
    AngularFireModule.initializeApp(environment.firebaseConfig),
    AngularFireDatabaseModule,
    AngularFireAuthModule,
    AngularFirestoreModule
  ],
  // ...omitted
})
```

That’s it. You now have a skeleton app ready for development.

```
ng serve
```



1.3 Separating Development and Production Environments

Problem

You want maintain separate backend environments for develop and production.

Solution

It's a good practice to perform development on an isolated backend. You don't want to accidentally pollute or delete your user data while experimenting with a new feature.

The first step is to create a second Firebase project. You should have two projects named something like **MyAppDevelopment** and **MyAppProduction**.

Next, grab the API credentials and update the `environment.prod.ts` file.

```
export const environment = {
  production: true,
  firebaseConfig: {
    apiKey: "PROD_API_KEY",
    authDomain: "PROD.firebaseapp.com",
    databaseURL: "https://PROD.firebaseio.com",
    storageBucket: "PROD.appspot.com"
  }
};
```

Now, in your `app.module.ts`, your app will use different backend variables based on the environment.

```
import { environment } from '../environments/environment';
export const firebaseConfig = environment.firebaseConfig;
// ... omitted
  imports: [
    AngularFireModule.initializeApp(firebaseConfig)
  ]
```

Test it by running `ng serve` for development and `ng serve --prod` for production.

1.4 Importing Firebase Modules

Problem

You want to import the AngularFire2 or the Firebase SDK into a service or component.

Solution

Take advantage of tree shaking with AngularFire2 to only import the modules you need. In many cases, you will only need the database or authentication, but not both. Here's how to import them into a service or component.

```
import { AngularFirestore } from 'angularfire2/firestore';
import { AngularFireDatabase } from 'angularfire2/database';
import { AngularFireAuth }     from 'angularfire2/auth';

///... component or service

constructor(
  private afs: AngularFirestore,
  private db: AngularFireDatabase,
  private afAuth: AngularFireAuth
  ) {}
```

You can also import the firebase SDK directly when you need functionality not offered by AngularFire2. Firebase is not a `NgModule`, so no need to include it in the constructor.

```
import * as firebase from 'firebase/app';
```

1.5 Deployment to Firebase Hosting

Problem

You want to deploy your production app to Firebase Hosting.

Solution

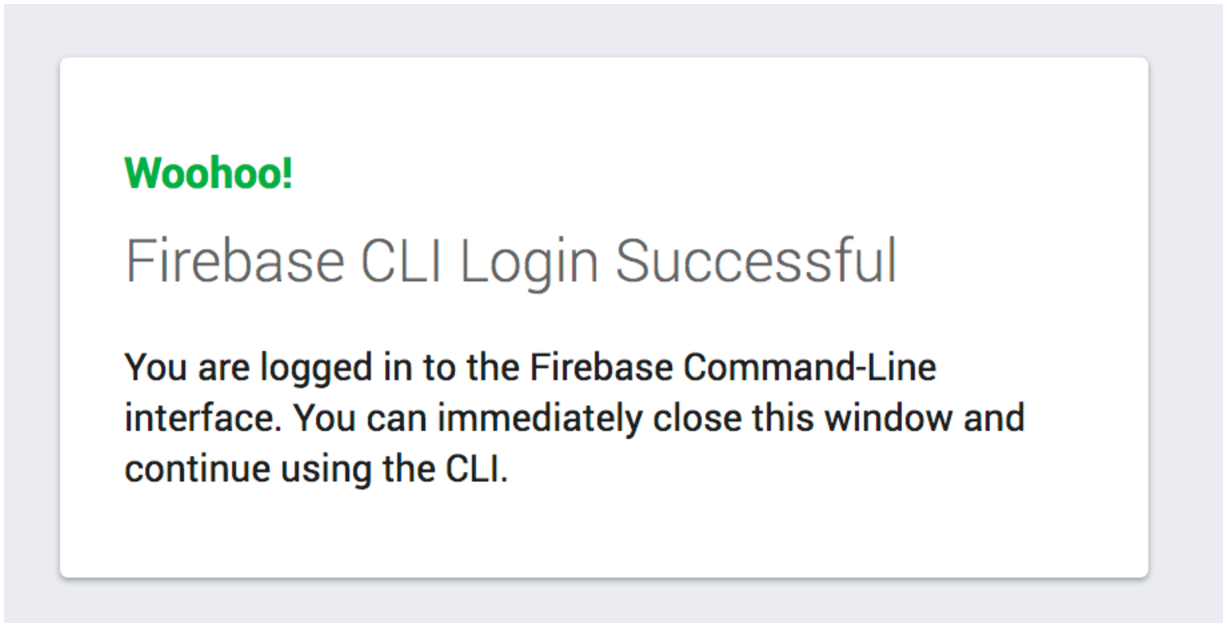
It is a good practice to build your production app frequently. It is common to find bugs and compilation errors when specifically when running an Ahead-of-Time (AOT) build in Angular.

During development, Angular is running with Just-In-Time (JIT) compilation, which is more forgiving with type safety errors.

```
ng build --prod
```

Make sure you are logged into firebase-tools.

```
npm install -g firebase-tools
firebase login
```



Woohoo!

Firebase CLI Login Successful

You are logged in to the Firebase Command-Line interface. You can immediately close this window and continue using the CLI.

Then initialize the project.

```
firebase init
```

```
You're about to initialize a Firebase project in this directory:
```

1. Choose hosting.
2. Change public folder to `dist/<your-app-name>` when asked (it defaults to public).
3. Configure as single page app? Yes.
4. Overwrite your index.html file? No.

```
firebase deploy
```

If all went well, your app should be live on the firebase project URL.

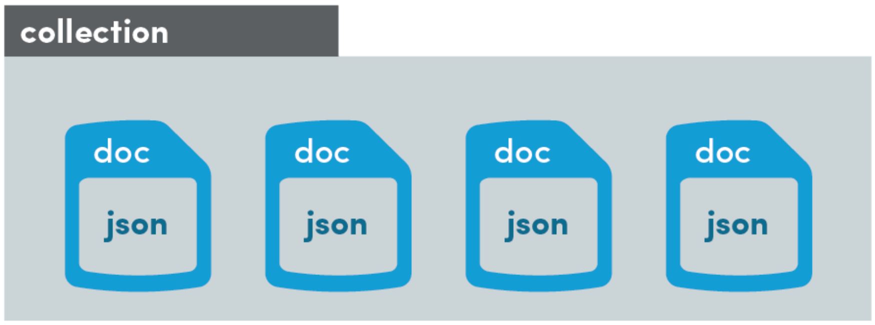
Cloud Firestore

Firestore was introduced into the Firebase platform on October 3rd, 2017. It is a superior alternative (in most situations) to the Realtime Database that is covered in Chapter 3.

What is Firestore?

Firestore is a NoSQL document-oriented database, similar to MongoDB, CouchDB, and AWS DynamoDB.

It works by storing JSON-like data into **documents**, then organizes them into **collections** that can be queried. All data is contained on the document, while a collection just serves as a container. Documents can contain their own nested subcollections of documents, leading to a hierarchical structure. The end result is a database that can model complex relationships and make multi-property compound queries.



Unlike a table in a SQL database, a Firestore document does not adhere to a data schema. In other words, document-ABC can look completely different from document-XYZ in the same collection. However, it is a good practice to keep data structures as consistent as possible across collections. Firestore automatically indexes documents by their properties, so your ability to query a collection is optimized by a consistent document structure.

The goal of this chapter is to introduce data modeling best practices and teach you how perform common tasks with Firestore in Angular.

2.0 Cloud Firestore versus Realtime Database

Problem

You're not sure if you should use Firestore or the Realtime Database.

Solution

I follow a simple rule - **use Firestore, unless you have a good reason not to**.

However, if you can answer TRUE to ALL statements below, the Realtime Database might worth exploring.

1. You make frequent queries to a small dataset.
2. You do not require complex querying, filtering, sorting.
3. You do not need to model data relationships.

If you responded FALSE to any of these statements, use Firestore.

Realtime Database billing is weighted heavily on data storage, while Cloud Firestore is weighted on bandwidth. Cost savings could make Realtime Database a compelling option when you have high-bandwidth demands on a lightweight dataset.

Why are there two databases in Firebase?

Firebase won't tell you this outright, but the Realtime Database has its share of frustrating caveats. Exhibit A: querying/filtering data is very limited. Exhibit B: nesting data is impossible on large datasets, requiring you to *denormalize* at the global level. Lucky for you, Firestore addresses these issues head on, which means you're in great shape if you're just starting a new app. Realtime Database is still around because it would be risky/impossible to migrate the gazillions of bytes of data from Realtime Database to Firestore. So Google decided to add a second database to the platform and not deal with the data migration problem.

2.1 Data Structuring

Firestore Quick Start Video Lesson

https://youtu.be/-GjF9pSeFTs

Problem

You want to know how to structure your data in Firestore.

Solution

You already know JavaScript, so think of a collection as an *Array* and a document as an *Object*.

What's Inside a Document?

A document contains JSON-like data that includes all of the expected primitive datatypes like strings, numbers, dates, booleans, and null - as well as objects and arrays.

Documents also have several custom datatypes. A `GeoPoint` will automatically validate latitude and longitude coordinates, while a `DocumentReference` can point to another document in your database. We will see these special datatypes in action later in the chapter.

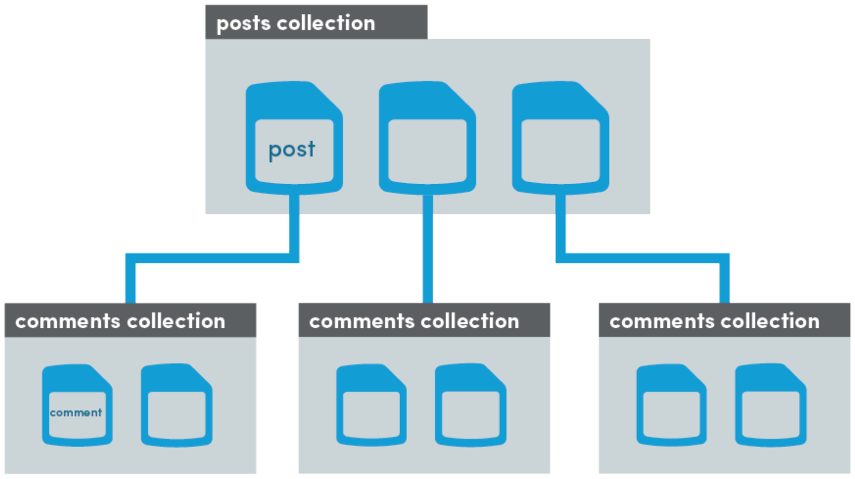
Best Practices

Firestore pushes you to form a hierarchy of data relationships. You start with (1) a collection in the root of the database, then (2) add a document inside of it, then (3) add another collection inside that document, then (4) repeat steps 2 and 3 as many times as you need.

1. Always think about HOW the data will be queried. Your goal is to make data retrieval fast and efficient.
2. Collections can be large, but documents should be small.
3. If a document becomes too large, consider nesting data in a deeper collection.

Let's take a look at some common examples.

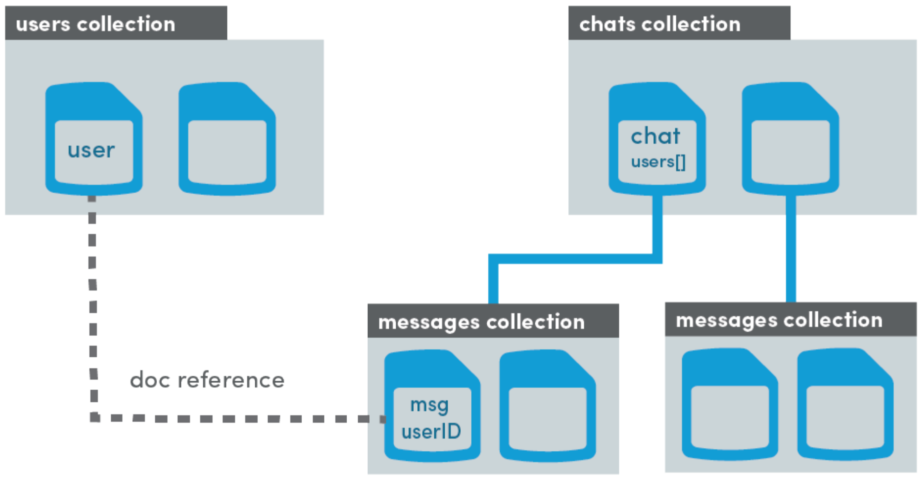
Example: Blog Posts and Comments



In this example, we have a collection of posts with some basic content data, but posts can also receive comments from users. We could save new comments directly on the document, but would that scale well if we had 10,000 comments? No, the memory in the app would blow up trying to retrieve this data. In fact, Firestore will throw an error for violating the 1 Mb document size limit well before reaching this point. A better approach is to nest a comments subcollection under each document and query it separately from the post data. Document retrieval is shallow - only the top level data is returned, while nested collections can be retrieved separately.

```
++postsCollection
    postDoc
      - author
      - title
      - content
      ++commentsCollection
          commentDocFoo
            - text
          commentDocBar
            - text
```

Example: Group Chat



For group chat, we can use two root level collections called *users* and *chats*. The user document is simple - just a place to keep basic user data like email, username, etc.

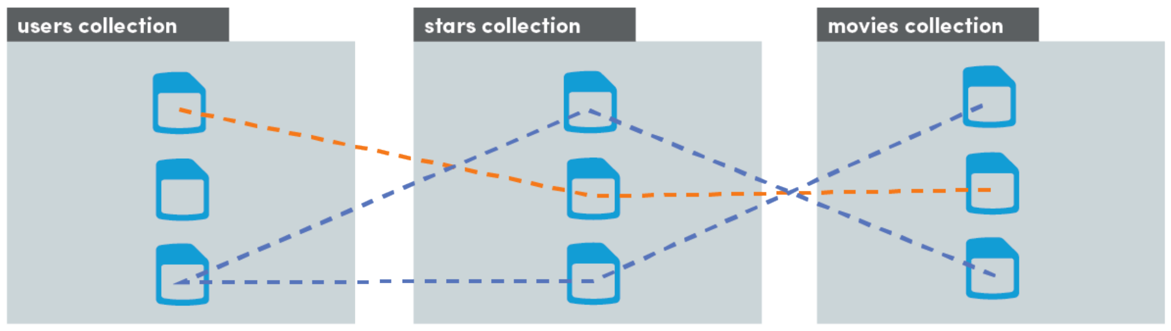
A chat document stores basic data about a chat room, such as the participating users. Each room has a nested collection of messages (just like the previous example). However, the message makes a reference to the associated user document, allowing us to query additional data about the user if we so choose.

A document reference is very similar to a foreign key in a SQL database. It is just a pointer to a document that exists at some other location in the database.

```
++usersCollection
    userDoc
      - username
      - email

++chatsCollection
    chatDoc
      - users[]
    ++messagesCollection
        messageDocFoo
          - text
          - userDocReference
        messageDocBar
          - userDocReference
```

Example: Stars, Hearts, Likes, Votes, Etc.



```
username            userID: string     title
...userData         movieID: string    ...movieData
                    value: number
```

In the graphic above, we can see how the movies collection and users collection have a two-way connection through the *middle-man* stars collection. All data about a relationship is kept in the star document - data never needs to change on the connected user/movie documents directly.

Having a root collection structure allows us to query both “Movie reviews” and “User reviews” independently. This would not be possible if stars were nested as a sub collection. This is similar to a *many-to-many-through* relationship in a SQL database.

```
++usersCollection
    userDoc
      - username
      - email

++starsCollection
    starDoc
      - userId
      - movieId
      - value

++moviesCollection
    movieDoc
      - title
      - plot
```

2.2 Collection Retrieval

Problem

You want to retrieve a collection of documents.

Solution

A *collection of documents* in Firestore is like a *table of rows* in a SQL database, or a *list of objects* in the Realtime Database. When we retrieve a collection in Angular, the endgame is to generate an Observable array of objects `[{...data}, {...data}, {...data}]` that we can show the end user.

The examples in this chapter will use the TypeScript `Book` interface below. AngularFire requires a type to be specified, but you can opt out with the `any` type, for example `AngularFirestoreCollection<any>`.

What is a TypeScript interface?

An interface is simply a blueprint for how a data structure should look - it does not contain or create any actual values. Using your own interfaces will help with debugging, provide better developer tooling, and make your code readable/maintainable.

```
export interface Book {
  author: string;
  title: string:
  content: string;
}
```

I am setting up the code in an Angular component, but you can also extract this logic into a service to make it available (injectable) to multiple components.

Reading data in AngularFire is accomplished by (1) making a reference to its location in Firestore, (2) requesting an Observable with `valueChanges()`, and (3) subscribing to the Observable.

Steps 1 and 2: book-info.component.ts

```
import { Component, OnInit } from '@angular/core';
import { Observable }        from 'rxjs';
import {
  AngularFirestore,
  AngularFirestoreCollection,
  AngularFirestoreDocument
} from 'angularfire2/firestore';

@Component({
  selector: 'book-info',
  templateUrl: './book-info.component.html',
  styleUrls: ['./book-info.component.scss']
})
export class BookInfoComponent implements OnInit {

  constructor(private afs: AngularFirestore) {}

  booksCollection: AngularFireCollection<Book>;
  booksObservable: Observable<Book[]>;

    ngOnInit() {
        // Step 1: Make a reference
        this.booksCollection = this.afs.collection('books');

        // Step 2: Get an observable of the data
        this.booksObservable = this.booksCollection.valueChanges();
    }

}
```

Step 3: book-info.component.html

The ideal way to handle an Observable subscription is with the `async` pipe in the HTML. Angular will subscribe (and unsubscribe) automatically, making your code concise and maintainable.

```html
<!-- Step 3: Subscribe to the data -->
<ul>
  <li *ngFor="let book of booksObservable | async">
    {{ book.title }} by {{ book.author }}
  </li>
</ul>
```

Step 3 (alternative): book-info.component.ts

It is also possible to subscribe directly in the Typescript. You just need to remember to unsubscribe to avoid memory leaks. Modify the component code with the following changes to handle the subscription manually.

```typescript
import { Subscription }      from 'rxjs';

/// ...omitted

    sub: Subscription;

    ngOnInit() {

        /// ...omitted

        // Step 3: Subscribe
        this.sub = this.booksObservable.subscribe(books => console.log(books))
    }

    ngOnDestroy() {
        this.sub.unsubscribe()
    }

}
```

End Preview

Real World Combined Examples

Now it's time to bring everything together. In this section, I solve several real-world problems by combining concepts from the Firestore, Realtime Database, user auth, storage, and functions chapters.

I've selected these examples because they are (A) commonly needed by developers and (B) implement many of the examples covered in this book. Each example also has a corresponding video lesson.

Important Update for Version 6.0

In version 6.0 of this book I decided to remove all code examples from this section. Is it because I'm lazy? Maybe. Is it because they were perpetually outdated by breaking changes? Getting warmer. Is it because I can provide something more useful? That's my goal!

Instead of dumping a bunch of complex examples in this section that will become quickly outdated, I will provide you with a curated list of lessons from AngularFirebase.com that I believe are critical to development success on this stack. These lessons are more thorough than what can be provided in a book format, are kept up-to-date, and accompanied by video content. If you hate this change to the book just send me a message on Slack - I'll make it up to you.

7.1 Auth with Firestore Custom User Data

Problem

You want to maintain custom user records that go beyond the basic information provided by Firebase authentication.

Solution

Episode 55: https://angularfirebase.com/lessons/google-user-auth-with-firestore-custom-data/

An app's auth system is the first thing I want to see when jumping into a new consulting project. When the user auth flow is screwed up it creates a cascading set of problems that can turn development into a nightmare.

After experimenting with various auth configurations in Firebase, I feel the approach presented in episode 55 is a great starting point. Most apps require their users to save some custom account data, so we wrap the Firebase user with a document in Cloud Firestore allowing us to add any custom data we want. From the developer's perspective, you get a centralized AuthService that can be injected anywhere in app to observe the current user.

7.2 Role-based Access Control

Problem

You want to assign users unique roles, then secure backend and frontend data based on their assigned roles.

Solution

Episode 75: https://angularfirebase.com/lessons/role-based-authorization-with-firestore-nosql-and-angular-5/

Role-based use authorization is a challenging feature to implement on any stack. There are many different ways to go about it, but I put together an approach that offers a high degree of flexibility. But most importantly, it shows you how to create security mechanisms with both the frontend and backend code. End-to-end security is critical for any access-control feature and this episode goes into great detail.

7.3 Drag and Drop File Uploads

Problem

You want users to drag and drop files into your app and upload them to a Firebase storage bucket.

Solution

Episode 82: https://angularfirebase.com/lessons/firebase-storage-with-angularfire-dropzone-file-uploader/

The book has an entire chapter dedicated to file uploads, but building your own dropzone uploader from scratch is a great exercise. In Episode 82 you will learn how to use angular to handle the drag events, then mix in AngularFireStorage to upload the file.

7.4 Firestore NoSQL Data Modeling

Problem

You're not sure how to model your Firestore data.

Solution

Episode 85: https://angularfirebase.com/lessons/firestore-nosql-data-modeling-by-example/

Modeling data in NoSQL is tricky. I have worked with MongoDB for many years, which is a document-oriented database that shares fundamental similarities to Firestore. What I've found over the years is that almost every data modeling problem has several viable solutions. *Solution A* might get you better performance, but require more data duplication. While *Solution B* might be less performant, but have the ability to scale infinitely. In Episodes 85 and 86 I provide a ton of different practical data modeling examples and discuss the tradeoffs for each.

7.5 Server Side Rendering

Problem

You need your Angular App to be search engine and social media linkbot friendly.

Solution

Episode 106 (Prerendering): https://angularfirebase.com/lessons/angular-6-universal-ssr-prerendering-firebase-hosting/

Episode 99 (SSR): https://angularfirebase.com/lessons/server-side-rendering-firebase-angular-universal/

Server-side rendering was once the Achilles' heal of AngularFirebase development. Universal SSR simply did not work with the Firebase SDK. Thankfully, that all changed in March 2018 and opened the door to building fully SEO optimized apps. There are two main strategies you can start using today:

Prerendering creates an entry-point page for every route in your Angular app and renders it at build-time. It's great when you have a small number of pages that need to be SEO-optimized, such as landing pages, product listings, etc.

SSR hosts your app on a NodeJS server and renders the app request-time. This means your entire app becomes SEO-optimized, but you have to deal with the added complexity and cost of deploying/scaling a server.