THE

ANGULAR FIREBASE

SURVIVAL GUIDE

Book and Videos By

JEFF DELANEY

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 2017-10-23

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

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 2+ 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.

- Angular v4.4
- Angular CLI v1.4.3
- TypeScript v2.3
- AngularFire2 v5.0

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

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[1].

[1] 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. Learn and use AngularFire2.
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 possible.
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. Use Lodash to simplify JavaScript.

1.2 Start a New App from Scratch

Problem

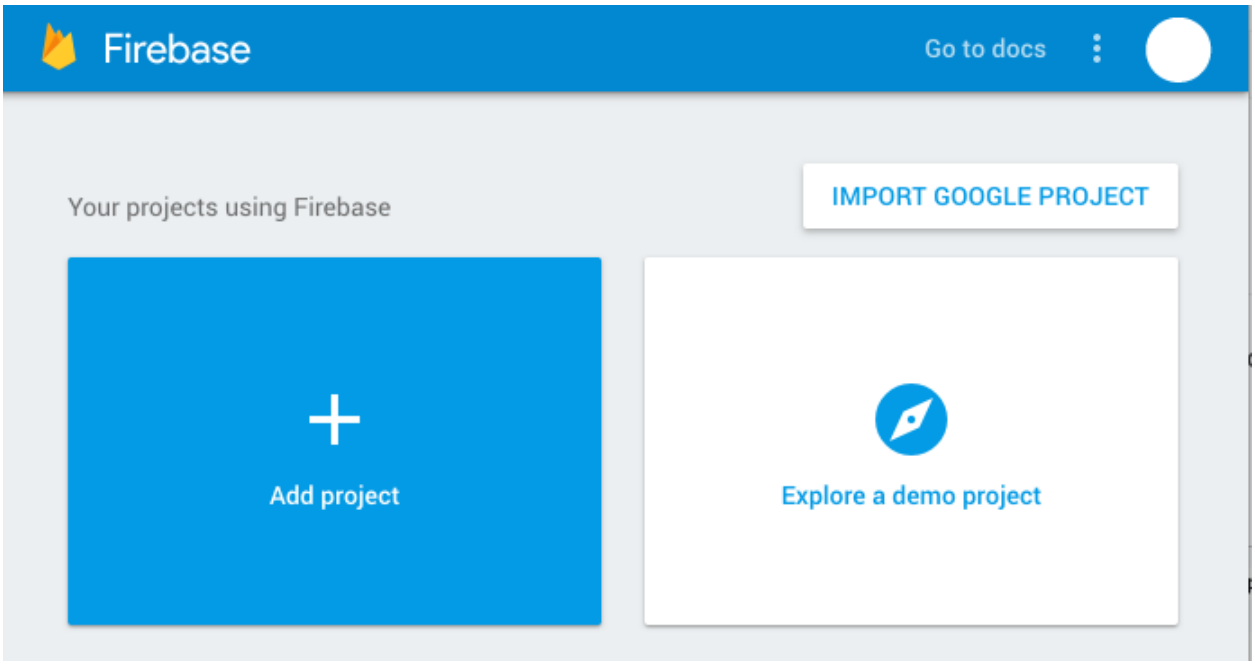
You want start a new Angular project, using Firebase for the backend.

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 typings
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 true --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 the AngularFire2 to the app.

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

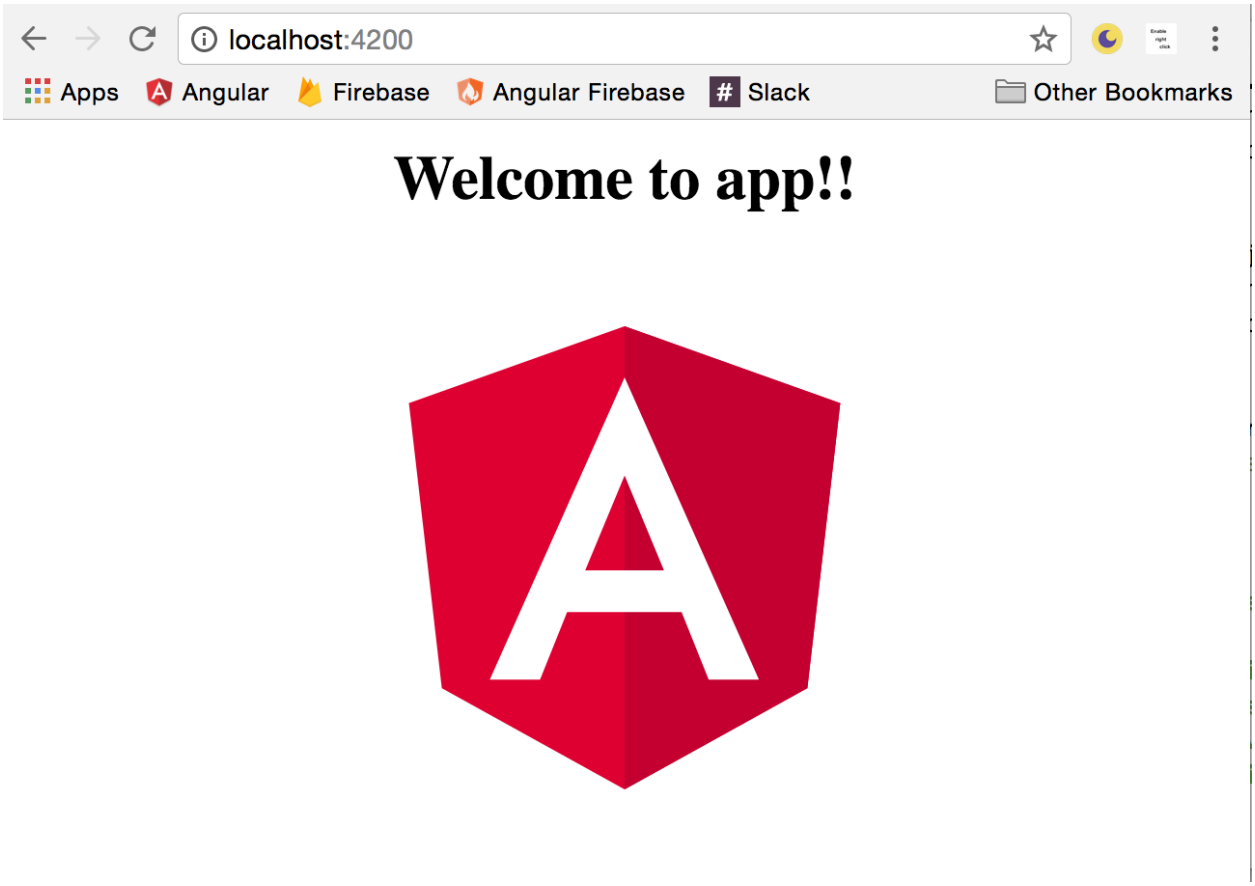
import { environment } from '../environments/environment';
export const firebaseConfig = environment.firebaseConfig;
// ...omitted
@NgModule({
  imports: [
    BrowserModule,
    AppRoutingModule,
    AngularFireModule.initializeApp(environment.firebaseConfig),
    AngularFireDatabaseModule,
    AngularFireAuthModule,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

Lastly, let's not forget our good friend Lodash.

```
npm install --save lodash
npm install --save @types/lodash
```

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

```
1 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

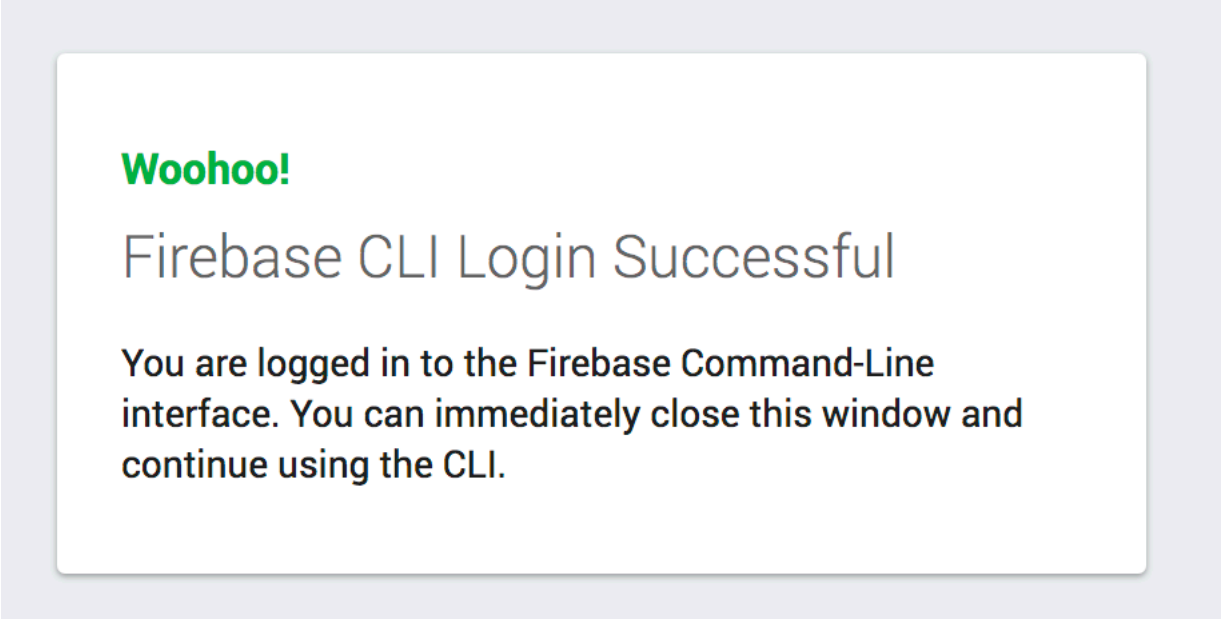
It is a good practice to build your production app frequently. It is common to find bugs and compilation errors when specifically when running an Ahead-of-Time (AOT) build in Angular.

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
```



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` 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.

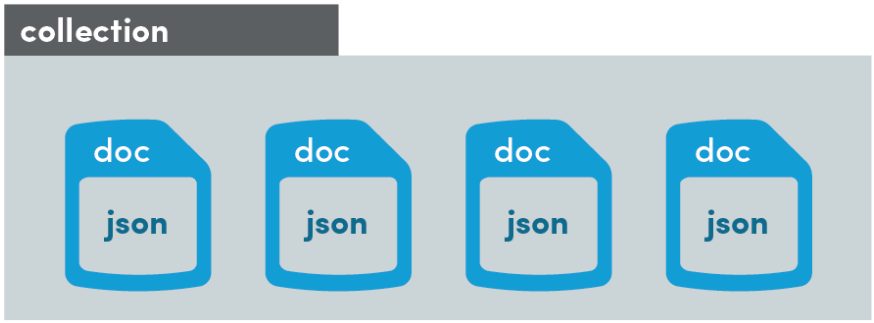
Cloud Firestore

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.

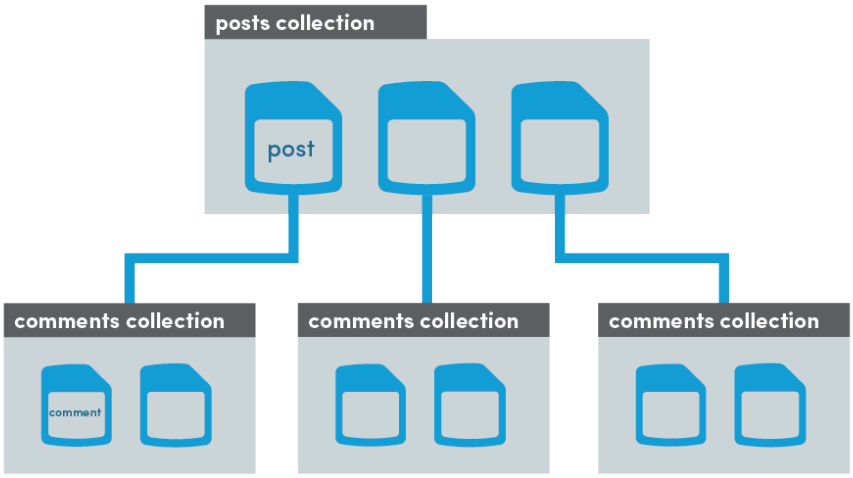
Best Practices

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

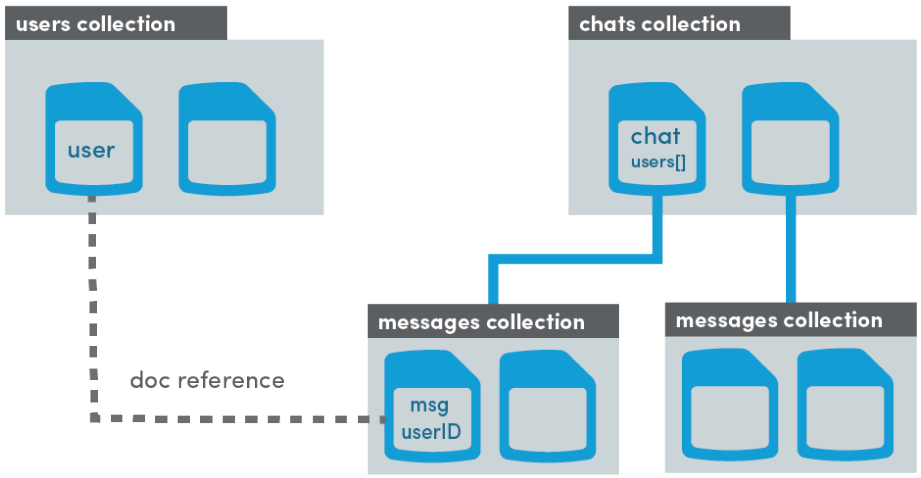


In this example, we have a collection of posts with some basic content data, but posts can also receive comments from users. We could save new comments directly on the document, but would that scale well if we had 10,000 comments? No, the memory in the app would blow up trying to retrieve this data. 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.

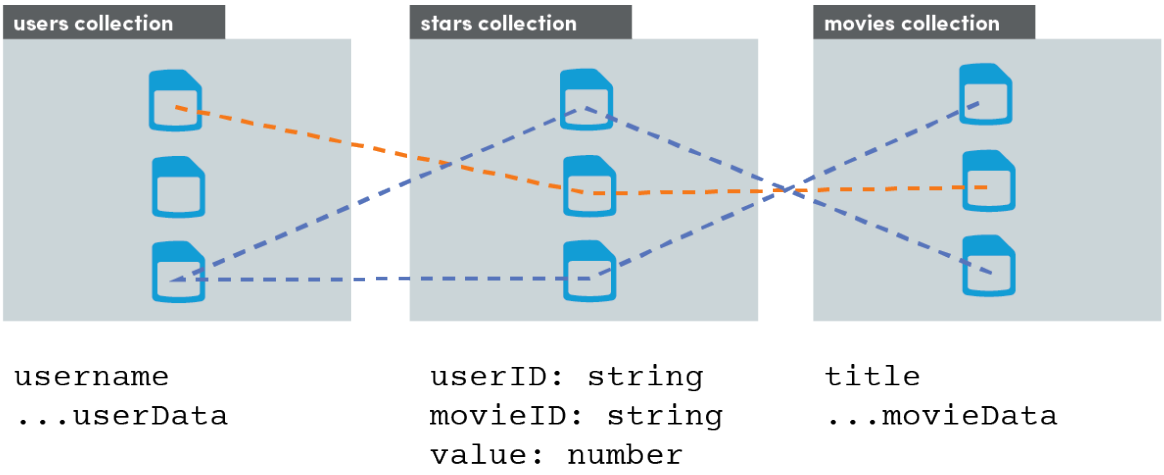
A chat document stores basic data about a chat room, such as the participating users. Each room has a nested collection of messages (just like the previous example). However, the message makes a reference to the associated user document, allowing us to query additional data about the user if we so choose.

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.



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/Observable';
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.

```
<!-- 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.

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

/// ...omitted

    sub: Subscription;

    ngOnInit() {

        /// ...omitted

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

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

}
```

2.3 Document Retrieval

Inferring Documents vs. Collections

The path segment to a collection is ODD, while the path to a document is EVEN. For example, `root(0)/collection(1)/document(2)/collection(3)/document(4)`. This rule always holds true in Firestore.

Problem

You want to retrieve a single document.

Solution

Every document is created with a auto-generated unique ID. If you know the unique ID, you can retrieve the document with the same basic process as a collection, but using the `afs.doc('collection/docId')` method.

```
export class BookInfoComponent implements OnInit {

  constructor(private afs: AngularFirestore) {}

  bookDocument: AngularFireDocument<Book>;
  bookObservable: Observable<Book>;

    ngOnInit() {
        // Step 1: Make a reference
        this.bookDocument = this.afs.doc('books/bookID');

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

}
```

book-info.component.html

When working with an individual document, it is useful to set the unwrapped Observable as a template variable in Angular. This little trick allows you to use the async pipe once, then call any property on the object - much cleaner than an async pipe on every property.

```
<!-- Step 3: Subscribe to the data -->
<div *ngIf="bookObservable | async as book">
    {{ book.title }} by {{ book.author }}
</div>
```

2.4 Include Document Ids with a Collection

Problem

You want the document IDs included with a collection.

Solution

By default, `valueChanges()` does not map the document ID to the document objects in the array. In many cases, you will need the document ID to make queries for individual documents. We can satisfy this requirement by pulling the entire snapshot from Firestore and mapping it's metadata to a new object.

```
this.booksObservable = booksCollection.snapshotChanges().map(arr => {
    return arr.map(snap => {
        const data = snap.payload.doc.data();
        const id = snap.payload.doc.id;
        return { id, ...data };
    });
});

// Unwrapped data: [{ id: 'xyz', author: 'Jeff Delaney', ...}]
```

This is not the most beautiful code in the world, but it's the best we can do at this point. If you perform this operation frequently, I recommend building a generic Angular service that can apply the code to any collection.

2.5 Add a Document to Collections

Problem

You want to add a new document to a collection.

Solution

Collections have an `add()` method that takes a plain JavaScript object and creates a new document in the collection. The method will return a Promise that resolves when the operation is successful, giving you the option to execute additional code after the operation succeeds or fails.

```
const collection = this.afs.collection('books');

new data = {
    author: 'Jeff Delaney'
    title: 'The Angular Firebase Survival Guide',
    year: 2017
}

collection.add(data)
          /// optional Promise methods
          .then(() => console.log('success') )
          .catch(err => console.log(err) )
```

2.6 Set, Update, and Delete a Document

Problem

You want to set, update, and delete individual documents.

Solution

Write operations are easy to perform in Firestore. You have the following three methods at your disposal.

- `set()` will destroy all existing data and replace it with new data.
- `update()` will only modify existing properties.
- `delete()` will destroy the document.

```
const doc = this.afs.doc('books/bookID');

const data = {
    author: 'Jeff Delaney'
    title: 'The Angular Firebase Survival Guide',
    year: 2017
};

doc.set(data); // reset all properties with new data
doc.update({ publisher: 'LeanPub' }); // update individual properties
doc.delete(); // update individual properties
```

All operations return a Promise that resolves when the operation is successful, giving you the option to execute additional code after the operation succeeds or fails.

```
doc.update(data)
   .then(() => console.log('success') )
   .catch(err => console.log(err) )
```

2.7 Create References between Documents

Problem

You want to create a reference between two related documents.

Solution

Document references provide a convenient way to model data relationships, similar to the way foreign keys work in a SQL database. We can set them by sending a `DocumentReference` object to firestore. In AngularFire, this is as simple as calling the `ref` property on the document reference. Here's how we can host a reference to a user document on a book document.

```
const bookDoc = this.afs.doc('books/bookID');
const userDoc = this.afs.doc('users/userID');

bookDoc.update({ author: userDoc.ref });
```

2.8 Set a Consistent Timestamp

Problem

You want to maintain a consistent server timestamp on database records.

Solution

Setting timestamps with the JavaScript `Date` class does not provide consistent results on the server. Fortunately, we can tell Firestore to set a server timestamp when running write operations.

I recommend setting up a TypeScript getter to make the timestamp call less verbose. Simply pass the object returned from `FieldValue.serverTimestamp()` as the value to any property that requires a timestamp.

```
const bookDoc = this.afs.doc('books/bookID');
bookDoc.update({ updatedAt: this.timestamp });

get timestamp() {
    return firebase.firestore.FieldValue.serverTimestamp();
}
```

2.9 Use the GeoPoint Datatype

Problem

You want to save geolocation data in Firestore.

Solution

We need to send geolocation data to Firestore as an instance of the `GeoPoint` class. I recommend setting up a helper method to return the instance from the Firebase SDK. From there, you can use the GeoPoint as the value to any property that requires latitude/longitude coordinates.

```
const bookDoc = this.afs.doc('books/bookID');
const geopoint = this.geopoint(38.23, -119.77);

bookDoc.update({ location: geopoint });


geopoint(lat: number, lng: number) {
  return new firebase.firestore.GeoPoint(lat, lng);
}
```

2.10 Atomic Writes

Problem

You want to perform multiple database writes in a batch that will succeed or fail together.

Solution

Using the firebase SDK directly, we can create batch writes that will update multiple documents simultaneously. If any single operation fails, none of the changes will be applied. It works setting all operations on the `batch` instance, then runs them with `batch.commit()`. If any operation in the batch fails, the database rolls back to the previous state.

```
const batch = firebase.firestore().batch();
/// add your operations here

const itemDoc = firebase.firestore().doc('items/itemID');
const userDoc = firebase.firestore().doc('users/userID');

const currentTime = this.timestamp

batch.update(itemDoc, { timestamp: currentTime });
batch.update(userDoc, { timestamp: currentTime });

/// commit operations
batch.commit();
```

2.11 Order Collections

Problem

You want a collection ordered by a specific document property.

Solution

Let's assume we have the following documents in the books collection.

Keep in mind, Firestore does not order by ID, so it is important to set documents with an property that makes sense for ordering, such as a `timestamp`.

```
afs.doc('books/atlas-shrugged').set({ author: 'Ayn Rand', year: 1957 })
afs.doc('books/war-and-peace').set({ author: 'Leo Tolstoy', year: 1865 })
```

To order by year in ascending order (oldest to newest).

```
const books = afs.collection('books', ref => ref.orderBy('year') )

// { author: 'Leo Tolstoy', year: 1865 }
// { author: 'Ayn Rand', year: 1957 }
```

To order by year in descending order (newest to oldest).

```
const books = afs.collection('books', ref => ref.orderBy('year', 'desc') )

// { author: 'Ayn Rand', year: 1957 }
// { author: 'Leo Tolstoy', year: 1865 }
```

Ordering is not just limited to numeric values - we can also order documents alphabetically.

```
const books = afs.collection('books', ref => ref.orderBy('name' )
```

2.12 Limit and Offset Collections

Problem

You want a specific number of documents returned in a collection.

Solution

As your collections grow larger, you will need to limit collections to a manageable size.

For the sake of this example, let's assume we have millions of books in our collection.

The `limit()` method will return the first N documents from the collection. In general, it will always be used in conjunction with `orderBy()` because documents have no order by default.

```
afs.collection('books', ref => ref.orderBy('year').limit(100) )
```

When it comes to offsetting data, you have four methods at your disposal. I find it easier write them out in a sentence.

- **startAt** - Give me everything after this document, including this document
- **startAfter** - Give me everything after this document, excluding this document.
- **endAt** - Give me everything before this document, including this document.
- **endBefore** - Give me everything before this document, excluding this document.

If we want all books written after a certain year, we run the query like so:

```
afs.collection('books', ref => ref.orderBy('year').startAt(1969) )

/// Like saying books where year >= 1969
```

If we change it to `startAfter()`, books from 1969 will be excluded from the query.

```
afs.collection('books', ref => ref.orderBy('year').startAfter(1969) )

/// Like saying books where year > 1969
```

These methods are very useful when it comes to building pagination and infinite scroll features in apps.

2.13 Querying Collections with Where

Problem

You want query documents with equality and/or range operators.

Solution

The `where()` method provides an expressive way to filter data in a collection. The beauty of the method is that it works just like it reads. It requires three arguments `ref.where(field, operator, value)`.

- `field` is any property on your document, i.e. author or year
- `operator` is any of the following logical operators: `==`, `<`, `<=`, `>`, or `>=`. (notice `!=` is not included)
- `value` is the value you're comparing. i.e. 'George Orwell' or 1984

Let's look at some examples and read them like sentences. First, we can filter by equality.

```
afs.collection('books', ref => ref.where('author', '==', 'James Joyce') )

// Give me all books where the author is James Joyce
```

Our we can use logical range operators

```
afs.collection('books', ref => ref.where('year', '>=', 2001) )

// Books where the year published is greater-than or equal-to 2001.
```

```
afs.collection('books', ref => ref.where('year', '<', 2001) )

// Books where the year published is less-than 2001.
```

We can also chain the `where` method to make multi-property queries.

```
afs.collection('books', ref => ref.where('author', '==', 'James Joyce').where('y\
ear', '>=', 1920) )

// Books where author is James Joyce AND year is greater-than 1920.
```

But there is one major exception! You cannot combine range operators on multiple properties.

```
afs.collection('books', ref => ref.where('year', '>=', 2003).where('author', '>'\
, 'B') )

// ERROR
```

2.14 Creating Indices

Problem

You want to order a collection by multiple properties, which requires a custom index.

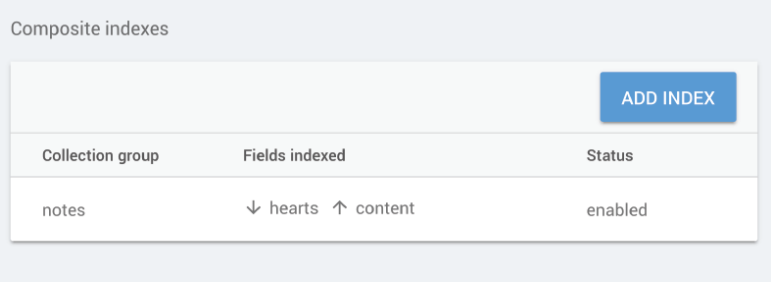
Solution

Firestore will automatically create an index for every individual property in your collection. However, it would result in an enormous amount of indices if Firestore indexed every combination of properties. A document with just 7 properties has 120 possible index combinations if you follow the rule of Eularian numbers.

The best way to create an index is to simply wait for Firestore to tell you when one is necessary. If we try to order by two different properties, we should get an error in the browser console.

```
afs.collection('books', ref => ref.orderBy('year').orderBy('author') )

// Error, you need to create an index for the query
```



The error message will provide a URL link to the Firestore console to create the index. Create the index, and the error will have disappeared the next time you run the query.

2.15 Backend Firestore Security Rules

Problem

You want to secure your backend data to authorized users with firestore security rules.

Solution

All Firestore rules must evaluate to a boolean value (true or false). Writing a rule is like saying "If some condition is true, I grant you permission to read/write this data".

There are thousands of possible security rule scenarios and I can't cover them all, but I can show you what I consider the most useful configurations.

Firestore Rules do NOT Cascade

If you've used rules in the Realtime Database you might be used to cascading rules, where higher level rules apply to nested data. It does not work like this in Firestore unless you explicitly use the =** wildcard operator.

Applying Rules to Documents

Before we write any rules, let's look at how we target rules to specific documents. There are three different options, as outlined below.

1. Apply to exact document:

```
    match /itemsCollection/itemXYZ
```

1. Apply to all documents at this level:

```
    match /itemsCollection/{itemID}
```

1. Apply to all documents at this level AND its nested subcollections:

```
    match /itemsCollection/{itemID=**}
```

No Security: Everybody can read and write

To make your database completely accessible to anyone.

```
service cloud.firestore {
  match /databases/{database}/documents {
    match /{document=**} {
      allow read;
      allow write;
    }
  }
}
```

Note: From here on out, I am going to omit the code surrounding the database to avoid repeating myself.

Full Security: Nobody can read or write

If you need to lock down your database completely, add this rule.

```
    match /{document=**} {
      allow read: if false;
      allow write: if false;
    }
```

Authenticated Security: Logged in users can read or write

This allows logged-in users full access to the database. Keep in mind, it does not secure data at the user level - for example, userA can still read/write data that belongs to userB. You can also combine actions on a single line to avoid duplicating identical rules.

```
    match /{document=**} {
      allow read, write: if request.auth != null;
    }
```

User Security: Users can only write data they own

This is perhaps the most common and useful security pattern for apps. It locks down anything nested under a userID to that specific user.

```
    match /users/{userId} {
      allow read, write: if request.auth.uid == userID;
    }
```

Role Based Security: Only Moderators can Write Sata

Many apps give certain users special moderator/admin privileges. These types of rules can get quite verbose, but Firestore allows you to define your own custom reusable functions.

This rule will only allow users who have the `isModerator == true` attribute on their user account to delete posts in the forum.

```
    function isModerator(userId) {
        get(/databases/$(database)/documents/users/$(userId)).data.isModerator =\
= true;
    }

    match /forum/{postID} {
      allow delete: if isModerator(request.auth.uid);
    }
```

Regex Security

You can perform a a regular expression match to ensure data adheres to a certain format. For example, this rule will only allow writes of the email address ends in `@angularfirebase.com`

```
    match /{document} {
    allow write: if document.matches('.*@angularfirebase\.com')
    }
```

Time Security

You can also get the exact timestamp in UTC format from the request to compare to an existing timestamp in the database.

```
    match /{document} {
        allow write: if request.time < resource.data.timestamp + duration.value(\
1, 'm');
    }
```

Realtime Database

Firebase provides a **realtime NoSQL database**. This means all clients subscribe to one database instance and listen for changes. As a developer, it allows you to handle database as an asynchronous data stream. Firebase has abstracted away the pub/sub process you would normally need to build from scratch using something like Redis.

Here are the main things you should know when designing an app with Firebase.

- It is a NoSQL JSON-style database
- When changes occur they are published to all subscribers.
- Operations must be executed quickly (SQL-style joins on thousands of records are not allowed)
- Data is retrieved in the form of an RxJS Observable
- Data is unwrapped asynchronously by subscribing to Observables

Injecting the AngularFire Database

ALL code examples in this chapter assume you have injected the `AngularFireDatabase` into your component or service. Example 3.2 is the only snippet that shows this process completely.

Would you rather use the Firestore database?

In most cases, the Firestore (section 2) document database is superior to the realtime database. It provides better querying methods and data structuring flexibility. You should have good reason to use Realtime Database over Firestore.

3.0 Migrating from AngularFire Version 4 to Version 5

Problem

You want to migrate an existing app from AngularFire <= v4 to v5. (If you're brand new to AngularFire, skip this snippet).

Solution

AngularFire v5.0.0 was released in October 2017 and was a complete rewrite of the realtime database API. It introduced significant breaking changes to previous versions, so I want to provide a quick migration guide for developers in the middle of existing projects.

Quick Fix

After you upgrade to v5, your database code will break catastrophically. Fortunately, the AngularFire core team realized this issue and kept the old API available under a different namespace of `database-deprecated`. You can make your code work by simply updating your imports.

Do a project search for "angularfire2/database" and replace all instances with "angularfire2/database-deprecated".

You code should now look like this:

```
import {
  AngularFireDatabase,
  FirebaseObjectObservable,
  FirebaseListObservable
} from 'angularfire2/database-deprecated';
```

Full Fix

Fully migrating to the new API is going to be a little more tedious. The main difference in v5 is the decoupling of the `Observable` from its reference to firebase.

Let's compare the APIs.

```
/// *** Version 4 ***

const item: FirebaseObjectObservable<Item[]> = db.object('items/someKey')
item.update(data)
item.remove()

item.subscribe(data => console.log(data) )

/// *** Version 5 ***

const item: AngularFireObject<Item> = db.object('items/someKey')
item.update(data)
item.remove()

// Notice how the Observable is separate from write options
const itemObservable: Observable<Item> = object.valueChanges()
itemObservable.subscribe(data => console.log(data) )
```

Here is the basic process you will need to follow to update from v4 to v5:

1. For database write operations (push, update, set, remove), you will need to convert every `Firebase(List | Object)Observable` into the new `AngularFire(List | Object)` reference.
2. To read data as an `Observable` you will need to call `valueChanges()` or `snapshotChanges()` on the reference created in the previous step.

3.1 Data Modeling

Firebase NoSQL Data Modeling

https://youtu.be/2ciHixbc4HE

Problem

You want to know how to model data for Firebase NoSQL.

Solution

In NoSQL, you should always ask "**How am I going to be querying this data?"**, because operations must be executed quickly. Usually, that means designing a database that is shallow or that avoids large nested documents. You might even need to duplicate data and that's OK - I realize that might freak you out if you come from a SQL background. Consider this fat and wide design:

```
-|users
  -|userID
    -|books
      -|bookID
        -|comments
          -|commentID
            -|likes
```

Now imagine you wanted to loop over the users just to display their usernames. You would also need load their books, the book comments, and the likes – all that data just for some usernames. We can do better with a tall and skinny design - a *denormalized* design.

```
-|users
  -|userID

-|books
  -|userId
    -|bookID

-|comments
  -|bookID

-|likes
  -|commentID
```

3.2 Database Retrieval as an Object

Build a Firebase CRUD App

https://youtu.be/6N_1vUPlhvk

Problem

You want to retrieve and subscribe to data from Firebase as a single object.

Solution

You should retrieve data as an object when you do not plan iterating over it. For example, let's imagine we have a single book in our database.

The `AngularFireObject<T>` requires a TypeScript type to be specified. If you want to opt out, you can use `AngularFireObject<any>`, but it's a good idea to statically type your own interfaces:

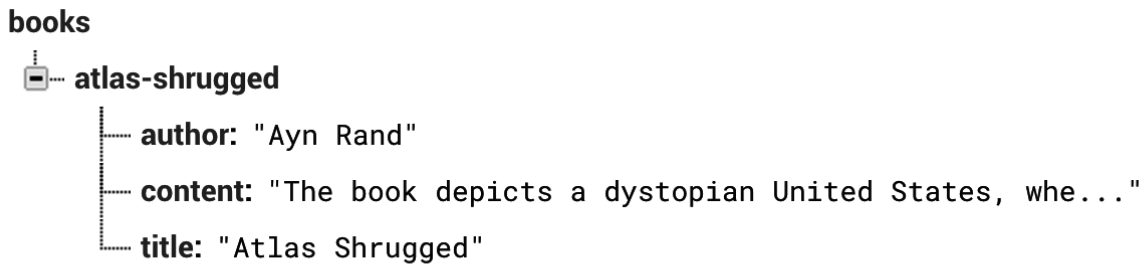
What is a TypeScript interface?

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 mke your code readable/maintainable.

Let's create a custom type for your `Book` data.

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

AngularFireList<>



We can observe this data in an Angular Component.

```
import { Component, OnInit } from '@angular/core';
import { Observable }        from 'rxjs/Observable';
import {
  AngularFireDatabase,
  AngularFireObject,
  AngularFireList
} from 'angularfire2/database';

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

  constructor(private db: AngularFireDatabase) {}

  bookRef: AngularFireList<Book>;
  bookObservable: Observable<Book>;


  ngOnInit() {
```

```
23      // Step 1: Make a reference
24      this.bookRef = this.db.object('books/atlas-shrugged');
25
26      // Step 2: Get an observable of the data
27      this.bookObservable = this.bookRef.valueChanges()
28    }
29
30  }
```

3.3 Show Object Data in HTML

Problem

You want to show the Observable data in the component HTML template.

Solution

We have a `Observable<Book>`. How do we actually get data from it? The answer is we subscribe to it. Angular has a built async pipe[2] that will subscribe (and unsubscribe) to the Observable from the template.

```
1  <article>
2  {{ bookObservable | async | json }}
3
4  {{ (bookObservable | async)?.content }}
5  </article>
```

We unwrap the Observable in parenthesis before trying to call its attributes. Calling `bookObservable.author` would not work because that attribute does not exist on the Observable itself, but rather its emitted value. The result should look like this:

[2] https://angular.io/api/common/AsyncPipe

Atlas Shrugged

{ "author": "Any Rand", "content": "The book depicts a dystopian United States, wherein many of society's most prominent and successful industrialists abandon their fortunes and even the nation, in response to aggressive new regulations, whereupon most vital industries collapse", "title": "Atlas Shrugged" }

If you have an object with many properties, consider setting the unwrapped Observable as a template variable in Angular. This little trick allows you to use the async pipe once, then call any property on the object - much cleaner than an async pipe on every property.

```
<article *ngIf="bookObservable | async as book">
  {{ book.author }}
  {{ book.title }}
  {{ book.content }}
</article>
```

3.4 Subscribe without the Async Pipe

Problem

You want to extract Observable data in the component TypeScript before it reaches the template.

Solution

Sometimes you need to play with the data before it reaches to the template. We can replicate the `async` pipe in the component's TypeScript, but it takes some extra code because we must create the subscription, then unsubscribe when the component is destroyed to avoid memory leaks.

```
//// book-info.component.ts
import { Subscription } from 'rxjs/Subscription';

subscription: Subscription;
bookRef: AngularFireList<Book>;
bookData: Book;

ngOnInit() {
  this.bookRef = this.db.object('books/atlas-shrugged');

  this.subscription = this.bookRef.valueChanges()
                          .subscribe(book => {
                            this.bookData = book
                          })
}

ngOnDestroy() {
  this.subscription.unsubscribe()
}
```

In the HTML, the `async` pipe is no longer needed because we unwrapped the raw data in the TypeScript with `subscribe`.

```
{{ bookData | json }}

{{ bookData?.content }}
```

3.5 Map Object Observables to New Values

RxJS Quick Start Video Lesson

https://youtu.be/2LCo926NFLI

Problem

Problem you want to alter Observable values before they are emitted in a subscription.

Solution

RxJS ships with all sorts to helpful operators to change the behavior of Observables. For now, I will demonstrate `map` because it is the most frequently used in Angular.

Let's get the object Observable, then map its author property to an uppercase string.

```
this.bookObserbable = this.bookRef
                          .map(book => book.author.toUpperCase() )
```

The HTML remains the same.

```
{{ bookObservable | async }}
```

But the result will be a string of **AYN RAND**, instead of the JS object displayed in section 3.3.

3.6 Create, Update, Delete a FirebaseObjectObservable data

Problem

You know how to retrieve data, but now you want to perform operations on it.

Solution

You have three available operators to manipulate objects.

1. Set - Destructive update. Deletes all data, replacing it with new data.
2. Update - Only updates specified properties, leaving others unchanged.
3. Remove - Deletes all data.

Here are three methods showing you how to perform these operations on an `AngularfireObject`.

```
createBook() {
  const book = { title: 'War and Peace' }
  return this.db.object('/books/war-and-peace')
                .set(book)
}

updateBook(newTitle) {
  const book = { title: newTitle }
  return this.db.object('/books/war-and-peace')
                .update(book)
}

deleteBook() {
  return this.db.object('/books/twilight-new-moon')
                .remove()
}
```

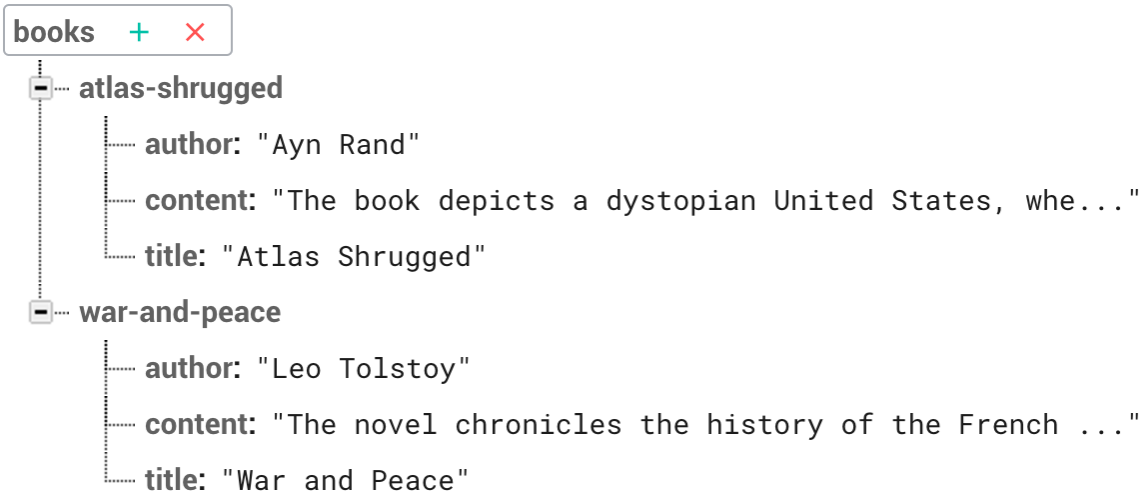
3.7 Database Retrieval as a Collection

Problem

You want to retrieve data from Firebase as a list or array.

Solution

The `AngularFireList` is ideal when you plan on iterating over objects, such as a collection of books. The process is exactly the same as an object, but we expect an `Array` of objects.



RxJS Observable Naming Preferences

It is common for Observable streams to be named with an ending $, such as book$. Some love it, some hate it. I will not be doing it here, but you may see this come up occasionally in Angular tutorials.

```
//// books-list.component.ts

booksRef: AngularFireList<Book>;
booksObservable: Observable<Book[]>; // <-- notice the [] here

ngOnInit() {
  // Step 1: Make a reference
  this.booksRef = this.db.list('books');

  // Step 2: Get an observable of the data
  this.bookObservable = this.booksRef.valueChanges();
}
```

3.8 Viewing List Data in the Component HTML

Problem

You want to iterate over an Observable list in the HTML template.

Solution

Again, you should take advantage of Angular's async pipe to unwrap the `Observable` in the template. This will handle the subscribe and unsubscribe process automagically.

```
<ul>
  <li *ngFor="let book of booksObservable | async">
    {{ book.title }} by {{ book.author }}
  </li>
</ul>
```

The result should look like this:

- Atlas Shrugged by Any Rand
- War and Peace by Leo Tolstoy

3.9 Limiting Lists

Problem

You want to limit the number of results in a collection.

Solution

You can pass a second callback argument to `db.list(path, queryFn)` to access Firebase realtime database query methods. In this example, we limit the results to the first 10 books in the database.

```
queryBooks() {
    return this.db.list('/books' ref => ref.limitToFirst(10) )
}
```

3.10 Filter Lists by Value

Never use orderByPrority

Firebase has an option to `orderByPrority`, but it only exists for legacy support. Use other ordering options instead.

Problem

You want to return list items that have a specific property value.

Solution

This time, let's filter the collection to all books with an `author` property of *Jack London*.

```
queryBooks() {
    return this.db.list('/books', ref => {
      return ref.orderByChild('author').equalTo('Jack London')
    })
}
```

3.11 Create, Update, Delete Lists

Problem

You want create, update, or remove values in a list Observable.

Solution

When creating new books, we push them to the list. This will create a **push key** automatically, which is an encoded timestamp that looks like "-Xozdf2i23sfdf73". You can think of this the unique ID for an item in a list.

Update and delete operations are similar to objects, but require the `key` of the item as an argument. The key is not returned with `valueChanges()`, so I included a helper method `booksWithKeys` that will return an Observable array with the pushKeys included.

```
/// Helper method to retrieve the keys as an Observable
booksWithKeys(booksRef) {
  return this.booksRef.snapshotChanges().map(changes => {
    return changes.map(c => ({ key: c.payload.key, ...c.payload.val() }));
  });
}

pushBook() {
  const book = { title: 'Call of the Wild' }
  return this.db.list('/books').push(book)
}

updateBook(pushKey) {
  const data = { title: 'White Fang' }
  return this.db.list('/books').update(pushKey, data)
}

deleteBook(pushKey) {
  return this.db.list('/books').remove(pushKey)
}
```

Obtain the Push Key on New Items

When pushing to a list, you might want the `$key` from new item. You can obtain it with `this.db.list('/books').push(book).key`

3.12 Catch Errors with Firebase Operations

Problem

You want to handle errors gracefully when a Firebase operation fails.

Solution

Data manipulation (`set`, `update`, `push`, `remove`) functions return a `Promise`, so we can determine success or error by calling `then` and/or `catch`. In this example, a separate error handler is defined that can be reused as needed. You might want to add some logging, analytics, or messaging logic to the `handleError` function.

```
this.createBook()
    .then( () => console.log('book added successfully'))
    .catch(err => handleError(err) );

this.updateBook()
      .then( () => console.log('book updated!'))
      .catch(err => handleError(err) );

private handleError(err) {
  console.log("Something went horribly wrong...", err)
}
```

3.13 Atomic Database Writes

Problem

You want to update multiple database locations atomically, to prevent data anomalies.

Solution

You will often find situations where you need to keep multiple collections or documents in sync during a single operation. In database theory, this is known as an `atomic` operation. For example, when a user comments on a book, you want to update the user's comment collection as well as the book's comment collection simultaneously. If one operation succeeded, but the other failed, it would lead to a data mismatch or anomaly.

In this basic example, we will update the tag attribute on two different books in a single operation. But be careful - this example will perform at destructive `set`, even though it calls update.

```
atomicSet() {
  let updates = {};
  updates['books/atlas-shrugged/tags/epic'] = true;
  updates['tags/epic/atlas-shrugged'] = true

  this.db.object('/').update(updates)
}
```

3.14 Backend Database Rules

Database Rules Video Lesson

https://youtu.be/qLrDWBKTUZo

Problem

You want to secure read/write access to your data on the backend.

Solution

Firebase allows you to define database security logic in JSON format that mirrors to the structure of your database. You just write logical statements that evaluate to `true` or `false`, giving users access to read or write data at a given location.

First, let's go over a few special built-in variables you should know about.

auth – The current user's auth state. root – The root of the database and can be traversed with .child('name'). data – Data state before an operation (the old data) newData – Data after an operation (the new data) now – Unix epoch timestamp ${wildcard} – Wildcard, used to compare keys.

Common Pitfall - Cascading Rules

You cannot grant access to data, then revoke it later. However, you can do the opposite – revoke access, then grant it back later. That being said, it is usually best to deny access by default, then grant access when the ideal conditions have been satisfied deeper in the tree.

Let's start by locking down the database at the root. Nothing goes in, nothing comes out.

```
"rules": {
  ".read": false,
  ".write": false
}
```

Now, let's give logged in users read access

```
"rules": {
  ".read": "auth != null",
  ".write": false
}
```

Now let's allow users to write to the books collection, but only if the data is under their own UID.

```
"rules": {
  ".read": "auth != null",
  "books": {
     "$uid": {
       ".write": "$uid === auth.uid"
     }
   }
 }
```

Now, let's assume we have moderator users, who have access to write to any user's book. Notice the use of the OR `||` operator in the rule to chain an extra condition. You can also use AND `&&` when multiple conditions must be met.

```
"rules": {
  ".read": "auth != null",
  "books": {
     "$uid": {
       ".write": "$uid === auth.uid
                  || root.child('moderators').child(auth.uid).val() === true"
     }
   }
 }
```

3.15 Backend Data Validation

Problem

You want to validate data before it's written to the database.

Solution

Firebase has a third rule, `.validate`, which allows you to put constraints on the type of data that can be saved on the backend. The incoming data will be in the `newData` Firebase variable.

Difference between Write and Validate

(1) Validation rules only apply to non-null values. (2) They do not cascade (they only apply to the level at which they are defined.)

```
"rules": {
  "books": {
    "$bookId": {
       "title": {
         ".validate": "newData.isString()"
       }
     }
   }
 }

 You will likely want to chain multiple validations together.

```json
{
  ".validate": "newData.isString()
                && newData.val().matches('regex-expression')"
}
```

You might have a list of allowed values in your database, let's image categories. You can validate against them by traversing the database.

```
{
  ".validate": "root.child('categories/' + newData.val()).exists()"
}
```

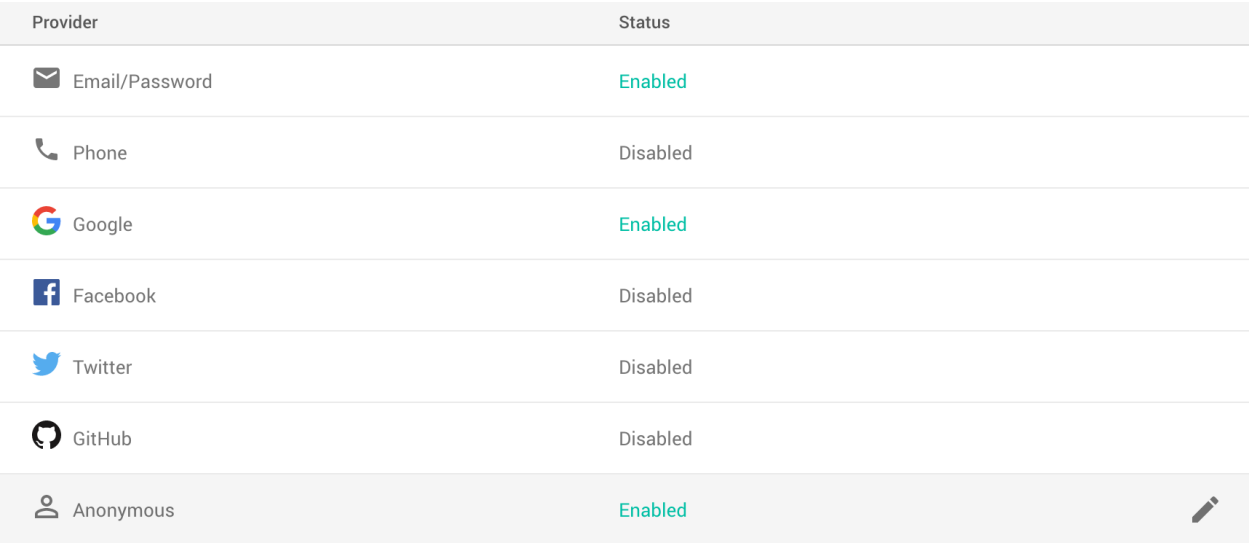
When creating an object, you might want to validate it has all the required attributes.

```
{
  "$bookId": {
    ".validate": "newData.hasChildren(['title', 'body', 'author'])",
     "title": {
       ".validate": "newData.isString()"
     },
     "body": {},
     "author": {}
   }
}
```
```

User Authentication

Firebase provides a flexible authentication system that integrates nicely with Angular and RxJS. In this chapter, I will show you how use three different paradigms, including:

- OAuth with Google, Facebook, Twitter, and Github
- Email/Password
- Anonymous



Injecting AngularFire Auth and Database

Most code examples in this chapter assume you have injected the `AngularFireDatabase` and `AngularFireAuth` into your component or service. If you do not know how to inject these dependencies, revisit section 1.4.

4.1 Getting Current User Data

Problem

You want to obtain the current user data from Firebase.

Solution

AngularFire2 returns an `authState` Observable that contains the important user information, such as the UID, display name, email address, etc. You can obtain the current user as an Observable like so.

```
import { Component, OnInit } from '@angular/core';
import { AngularFireAuth } from 'angularfire2/auth';
import * as firebase from 'firebase/app';
import { Observable } from 'rxjs/Observable';

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

  currentUser: Observable<firebase.User>;

  constructor(private afAuth: AngularFireAuth) { }

  ngOnInit() {
    this.currentUser = this.afAuth.authState;
  }

}
```

Alternatively, you can unwrap the auth observable by by subscribing to it. This may be necessary if you need the UID to load other data from the database

```
currentUser = null;

// or ngOnInit for components
constructor(afAuth: AngularFireAuth) {
  afAuth.authState.subscribe(userData => {
      this.currentUser = userData
    });
}
```

At this point, the `authState` will be null. In the following sections, it will be populated with different login methods.

4.2 OAuth Authentication

OAuth Video

https://youtu.be/-3rkY8X2EWc

Problem

You want to authenticate users via Google, Facebook, Github, or Twitter.

Solution

Firebase makes OAuth a breeze. In the past, this was the most difficult form of authentication for developers to implement. From the Firebase console, you need to manually activate the providers you want to use. Google is ready to go without any configuration, but other providers like Facebook or Github, require you to get your own developer API keys.

Here's how to handle the login process in a service.

```
googleLogin() {
  const provider = new firebase.auth.GoogleAuthProvider()
  return this.socialSignIn(provider);
}

facebookLogin() {
  const provider = new firebase.auth.FacebookAuthProvider()
  return this.socialSignIn(provider);
}

private socialSignIn(provider) {
  return this.afAuth.auth.signInWithPopup(provider)
}
```

Now you can create login buttons in your component HTML that trigger the login functions on the click event and Firebase will handle the rest.

```
<button (click)="googleLogin()"></button>
<button (click)="facebookLogin()"></button>
```

4.3 Anonymous Authentication

Anonymous Auth Video

https://youtu.be/dyQDAaDq2ag

Problem

You want lazily register users with anonymous authentication.

Solution

Anonymous auth simply means creating a user session without collecting credentials to re-authenticate, such as an email address and password. This approach is beneficial when you want a guest user to try out the app, then register later.

```
anonymousLogin() {
  return this.afAuth.auth.signInAnonymously()
}
```

That was easy, but the trick is upgrading their account. Firebase supports account upgrading, but it's not supported by AngularFire2, so let's tap into the Firebase SDK. You can link or upgrade any account by calling `linkWithPopup`.

```
import { AngularFireAuth } from 'angularfire2/auth';
import * as firebase from 'firebase/app';

linkGoogle() {
  const provider = new firebase.auth.GoogleAuthProvider()
  firebase.auth().currentUser.linkWithPopup(provider)
}

linkFacebook() {
  const provider = new firebase.auth.FacebookAuthProvider()
  firebase.auth().currentUser.linkWithPopup(provider)
}
```

4.4 Email Password Authentication

Problem

You want a user to sign up with their email and password.

Solution

Email/password auth is the most difficult to setup because we need to run some form validation and generate different views for *new user sign up* and *returning user sign in*. Here's how you might handle the process in a component.

Full Code Example

The code below is a minimal implementation for the book. Checkout the full example in the demo app at https://github.com/codediodeio/angular-firestarter

```
userForm: FormGroup;

constructor(private fb: FormBuilder, private afAuth: AngularFireAuth) {}

ngOnInit() {
  this.userForm = this.fb.group({
    'email': ['', [
        Validators.required,
        Validators.email
      ]
    ],
    'password': ['', [
      Validators.pattern('^(?=.*[0-9])(?=.*[a-zA-Z])([a-zA-Z0-9]+)$'),
      Validators.minLength(6),
      Validators.maxLength(25)
    ]
  ]
  });
}

emailSignUp() {
  let email = this.userForm.value['email']
  let password = this.userForm.value['password']
```

```
  return this.afAuth.auth.createUserWithEmailAndPassword(email, password)
}

emailLogin() {
   let email = this.userForm.value['email']
   let password = this.userForm.value['password']
   return this.afAuth.auth.signInWithEmailAndPassword(email, password)
}
```

Then create the form in the HTML

```
<form [formGroup]="userForm" (ngSubmit)="emailSignUp()">

  <label for="email">Email</label>
  <input type="email" formControlName="email" required>

  <label for="password">Password</label>
  <input type="password" formControlName="password" required>

  <button type="submit">Submit</button>

</form>
```

4.5 Handle Password Reset

Problem

You need a way for users to reset their password.

Solution

Firebase has a built-in flow for resetting passwords. It works by sending the user an email with a tokenized link to update the password - you just need a way to trigger the process directly via the Firebase SDK.

```
userEmail: string;

resetPassword() {
  const fbAuth = firebase.auth();
  fbAuth.sendPasswordResetEmail(userEmail)
}
```

Use `ngModel` in the HTML template to collect the user's email address. Then send the reset password email on the button click.

```
  <input type="email" [(ngModel)]="userEmail" required>

  <button (click)="handlePasswordReset()">Reset Password</button>
```

4.6 Catch Errors during Login

Problem

You want to catch errors when login fails.

Solution

The login process can fail[3] for a variety of reasons, so let's refactor the social sign in function from section 4.2. It is a good idea to create an error handler, especially if you use multiple login methods.

```
private socialSignIn(provider) {
  return this.afAuth.auth.signInWithPopup(provider)
             .then(() => console.log('success') )
             .catch(error => handleError(error) );
}

private handleError(error) {
  console.log(error)
  // alert user via toast message
}
```

4.7 Log Users Out

Problem

You want to end a user session.

[3] https://firebase.google.com/docs/reference/js/firebase.auth.Error

Solution

As you can imagine, logging out is a piece of cake. Calling `signOut()` will destroy the session and reset the current `authState` to null.

```
logout() {
  this.afAuth.auth.signOut();
}
```

4.8 Save Auth Data to the Realtime Database

Problem

You want to save a user's auth information to the realtime database.

Solution

The Firebase login function returns a `Promise`. We can catch a successful response by calling `then` and running some extra code to update the database. Let's refactor the our sign in function from section 4.2 to save the user's email address to the realtime database after sign in.

A good database structure for this problem has data nested under each user's UID.

```
-| users
    -| $uid
        email: string
        moderator: boolean
        birthday: number
```

In the component, we call the desired signin function, which returns a Promise. When resolved, the Promise provides a credential object with the user data that can be saved to the database.

```
private socialSignIn(provider) {
  return this.afAuth.auth.signInWithPopup(provider)
              .then(credential =>  {
                    const user = credential.user
                    this.saveEmail(user)
               })
}

private saveEmail(user) {
  if (!user) { return; }

  const path = `users/${user.uid}`;
  const data = { email: user.email }

  this.db.object(path).update(data)
}
```

4.9 Creating a User Profile

Problem

You want to display user data in profile page.

Solution

The Firebase auth object has some useful information we can use to build a basic user profile, especially when used with OAuth. This snippet is designed to show you the default properties available.

Let's assume we have subscribed to the `currentUser` from section 4.1. You can simply call its properties in the template.

```
<aside>
  <p>{{ currentUser?.displayName }}</p>
  <img [src]="currentUser?.photoUrl || defaultAvatar">
</aside>
```

Here are the Firebase default properties you can use to build user profile data.

- uid
- displayName

- photoUrl
- email
- emailVerified
- phoneNumber
- isAnonymous

You can add additional custom user details to the realtime database using the technique described in section 4.9.

4.10 Auth Guards to Protect Routes

Problem

You want to prevent unauthenticated users from navigating to certain pages.

Solution

Guards provide a way to lock down routes until its logic resolves to true. This may look complex (most of it is boilerplate), but it's actually very simple. We take the first emission from the `AuthState` Observable, map it to a boolean, and if `false`, the user is redirected to a login page. You can generate the guard with the CLI via `ng generate guard auth`;

```
import { Injectable } from '@angular/core';
import { CanActivate, ActivatedRouteSnapshot, RouterStateSnapshot, Router } from\
 '@angular/router';
import { Observable } from 'rxjs/Observable';
import { AngularFireAuth } from 'angularfire2/auth';
import 'rxjs/add/operator/do';
import 'rxjs/add/operator/map';
import 'rxjs/add/operator/take';

@Injectable()
export class AuthGuard implements CanActivate {
  constructor(private afAuth: AngularFireAuth, private router: Router) {}

  canActivate(
    next: ActivatedRouteSnapshot,
    state: RouterStateSnapshot): Observable<boolean> | boolean {

      return this.afAuth.authState
```

```
            .take(1)
            .map(user => !!user)
            .do(loggedIn => {
              if (!loggedIn) {
                console.log("access denied")
                this.router.navigate(['/login']);
              }
          })

  }
}
```

In the routing module, you can activate the guard by adding it to the `canActivate` property.

```
{ path: 'private-page', component: SomeComponent, canActivate: [AuthGuard] }
```

Firebase Cloud Storage

File storage used to be a major development hassle. It could take weeks of development fine tuning and optimizing a web app's file uploading process. With Firebase, you have a GCP Storage Bucket integrated into every project, along with security, admin console management, and a robust API.

File uploads must be handled directly from the the Firebase SDK, rather than AngularFire2. However, we can still take advantage of AF2 to save a file's metadata and download URL to the realtime database.

Future Support in AngularFire2

As of October 2017, AngularFire2 has not integrated Firebase storage, but this module is under active development. I will update this chapter when the time comes. https://firebase.google.com/docs/storage/web/start

First, let's start with this shell of a component to handle the file uploading process.

```
import { Component, OnInit } from '@angular/core';
import { AngularFireDatabase } from 'angularfire2/database';
import * as firebase from 'firebase';

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

  selectedFiles: FileList;
  uploadTask;

  constructor(private db: AngularFireDatabase) { }

  ngOnInit() {
  }


}
```

5.1 Creating an Upload Task

File Uploads Video

https://youtu.be/5qoU1EirSmo

Problem

You want to initiate an Upload task.

Solution

Important Caveat

The path to a file in a storage bucket must be unique. If two users upload a file to `/images/my_pet_pug.jpg`, only the first file will be persisted. If this could be a problem with your file structure, you may want to add a unique token or timestamp to every file name.

An upload task is a promise to store a file in Firebase Storage. You create the task like so:

1. Get a JavaScript `File` object via a form input (See Section 5.4)
2. Make a reference to the location you want to save it in Firebase
3. Call the `put` method to return an upload task Promise.

```
upload(file: File) {
  const storageRef = firebase.storage().ref().child('/images');
  return storageRef.put(file);
}
```

5.2 Handling the Upload Task Promise

Problem

You want to handle the progress, success, and failure of the upload task.

Solution

Let's modify the example in 5.1. Firebase has a `TaskState` object, with only one event - `STATE_CHANGED`.

```
upload(file: File) {
  const storageRef = firebase.storage().ref().child('images');
  const uploadTask =  storageRef.put(file);

  return uploadTask.on(firebase.storage.TaskEvent.STATE_CHANGED,
      (snapshot) =>  {
        // upload in progress
        console.log(snapshot)
      },
      (error) => {
        // upload failed
        console.log(error)
      },
      () => {
        // upload success
        console.log("success")
      }
    );
}
```

5.3 Saving Data about a file to the Realtime Database

Problem

You want to save properties from an uploaded file to the realtime database.

Solution

Saving upload information to the database is almost always required, as you will want to probably reference the download URL. Here's what we can get from a file snapshot.

https://firebase.google.com/docs/reference/js/firebase.storage.UploadTaskSnapshot

- downloadURL
- totalBytes
- metadata (contentType, contentLanguage, etc)

When the upload task completes, we can use the snapshot to save information to the database. Again, we are building on the `upload` function in examples 5.1 and 5.2.

```
upload(file: File) {

  // omitted see 5.2...
  () => {
    // upload success
    this.saveDownloadUrl(uploadTask.snapshot.downloadURL)
  }
}

saveDownloadUrl(url) {
  this.db.list('images')
         .push({ downloadUrl: url })
}
```

In Angular, you can use the url to display images or links to access downloadable files.

```
<img [src]="someImage.downloadUrl">
```

5.4 Uploading a Single File

Problem

You want to enable users to upload a single file from Angular.

Solution

Now that you know how to upload files on the backend, how do you actually receive the necessary `File` object from a user?

Here we have an input element for a file, that triggers a `detectFiles` function when it changes (when they select a file on their device). Then the user can start the upload process by clicking the button attached to `uploadSingle`.

```
<input type="file" (change)="detectFiles($event)">

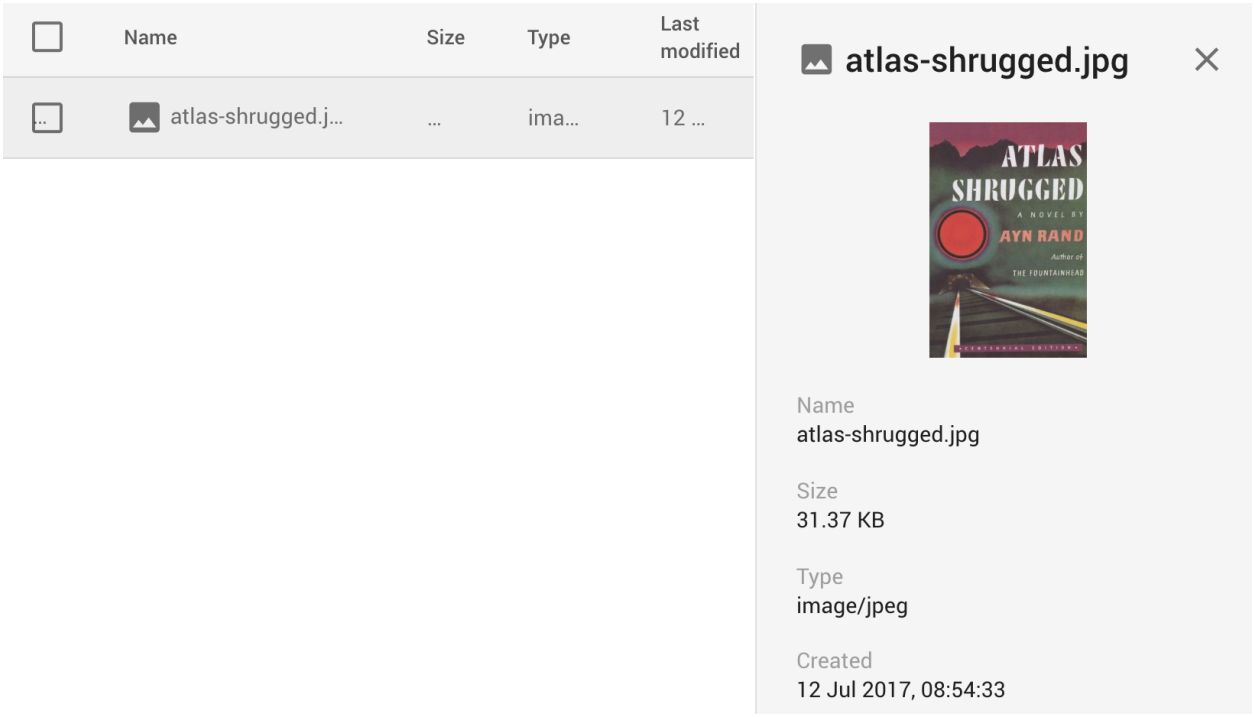
<button (click)="uploadSingle()">
```

Now let's define these event handlers in the TypeScript. The `change` event on the form input will contain a `FileList`, which can be obtained with `$event.target.files`. When the upload button is clicked, the file is sent to Firebase with upload function from section 5.1.

```
selectedFiles: FileList;

detectFiles($event) {
    this.selectedFiles = $event.target.files;
}

uploadSingle() {
  let file: File = this.selectedFiles.item(0)
  this.uploadTask = this.upload(file)
}
```



5.5 Uploading a Multiple Files

Problem

You want users to upload multiple files at a time.

Solution

Uploading multiple files can be done with a few slight changes. In the template, the form input needs to include the `multiple` attribute.

```
<input type="file" (change)="detectFiles($event)" multiple>

<button (click)="uploadMultiple()">
```

To loop over the FileList, we bring in Lodash (`_.each` and `_.range`). A `FileList object` is not an array that you can just iterate over, but you can iterate over a range of integers based on its length.

```
selectedFiles: FileList;

detectFiles($event) {
    this.selectedFiles = $event.target.files;
}

uploadMultiple() {
  let filesIndex = _.range(files.length)

  _.each(filesIndex, (idx) => {
    this.upload(selectedFiles[idx])
  })
}
```

5.6 Delete Files

Problem

You want users to be able to delete their files.

Solution

Deleting files follows the same process as uploading, but you need to know the location of the file. In most cases, this means you should have the image name or path saved in the database. Let's imagine looping through some images in the database.

```
<div *ngFor="let image of databaseImages | async">
  <button (click)="deleteFile(image.name)">
    Delete {{ image.name }}
  </button>
</div>
```

Now, we can pass that image name to a storage reference and delete it.

```
deleteFile(name) {
  firebase.storage()
          .ref(`/images`)
          .child(name).delete()
}
```

5.7 Validate Files on the Frontend

Problem

You want to alert a user when their file is not valid.

Solution

You should always validate files on the frontend because it creates a better user experience (but validate the backend also, see the next section). To do this, we use the built-in `File` object in javascript to collect some useful information about the file blob. The `size` and `type` attributes are probably the most common for validation.

```
validateFile(file: File) {
  const sizeMb = file.size / 1024 / 1024
  const mimeType = file.type.split('/')[0]

  validationErrors = []
  const sizeError = "Must be less than 10 Megabytes"
  const mimeError = "Must be an image"

  if (sizeMb > 10) validationErrors.push(sizeError)
  if (mimeType != 'image') validationErrors.push(mimeError)

  return validationErrors
}
```

5.8 Showing a Progress Bar

Problem

You want to show users the progress of their upload.

Solution

If you recall example 5.2, the upload task will return a snapshot of the upload after each state change event. The snapshot has two properties we can use to build a progress bar - `totalBytes` and `bytesTransferred`. Just apply some simple math to convert them to a percentage.

```
currentProgress: number;

upload {
  return uploadTask.on(firebase.storage.TaskEvent.STATE_CHANGED,
      (snapshot) =>  {
        this.currentProgress = (
          snapshot.bytesTransferred / snapshot.totalBytes
        ) * 100
      },
      /// omitted... see 5.2
    )
}
```

So now we have the percentage that will be updated on the component after each upload task event. Simply pass the `currentProgress` percentage as the value to the HTML `progress` element.

```
<label>Upload is {{currentProgress}}% Complete</label>
<progress min=1 max=100 [value]="currentProgress"></progress>
```

5.9 Upload Images in Base64 Format

Problem

You want to put a base64 encoded file into storage

Solution

You might have images encoded as a Base64 to avoid depending on an external file. There is no need to convert it - you can still upload it via `putString`, which also returns an upload task Promise.

```
uploadBase64() {
  const imageString = '5c6p7Y+2349X44G7232323...'
  return firebase.storage().ref('/images')
                          .putString(imageString)
}
```

5.10 Validating Files on the Backend

Problem

You want to prevent users from uploading extremely large files or certain file types to your storage bucket.

Solution

Backend validation is extremely important when dealing with file uploads from users. File storage rules are similar to database rules in principle, but use a slightly different syntax.

Here's the default security settings in Firebase. Users can only read/write if they are logged in.

```
service firebase.storage {
  match /b/{bucket}/o {
    match /{allPaths=**} {
      allow read, write: if request.auth != null;
    }
  }
}
```

Let's authorize writes for the image owner only, but allow any user to read the images.

```
match /images/{userId}/{allImages=**} {
     allow read;
     allow write: if (request.auth.uid == userId);
   }
```

Now let's validate file size and type. It must be less than 10 Mb and have an image MIME type.

```
match /{imageId} {
       allow read;
       allow write: if request.resource.size < 10 * 1024 * 1024
                    && request.resource.contentType.matches('image/.*')
```

You can also give buckets their own unique rules

```
match /b/bucket-PUBLIC.appspot.com/o {
  match /{allPaths=**} {
    allow read, write;
  }
}

match /b/bucket-PRIVATE.appspot.com/o {
  match /{allPaths=**} {
    allow read, write: if request.auth != null;
  }
}
```

Firebase Cloud Functions

Cloud functions are Functions-as-a-Service (FaaS) that allow you to run code on demand without ever worrying about server deployment.

- No server management
- Isolated codebase
- Billed on demand

When you deploy to a Platform-as-a-Service (PaaS), such as Heroku, you are billed a monthly rate even if the volume is miniscule. I find it annoying to pay $X per month for a background task that only runs 500 times per month.

The great thing about Cloud Functions is that you're billed by the millisecond. Your function runs for 400ms, then you're billed $0.00001 or whatever the actual cost.

It's also really helpful to isolate code outside of Angular, because you really need your Angular app to stay lean and agile. If you think of an app like a retail store, Angular is the customer service team and the cloud functions are the warehouse workers. The reps need to be available quickly and offer a responsive and engaging experience. Meanwhile, the warehouse workers need to handle all the heavy lifting and maintenance behind the scenes.

6.1 Initialize Cloud Functions in an Angular Project

Problem

You want to initialize cloud functions in your project

Solution

Cloud functions are managed with the firebase-tools CLI.

Run `firebase init`, choose functions and install dependencies, then `cd functions` and `npm install`.

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

From there, you have an isolated NodeJS environment to build microservices, setup HTTP endpoints, and run background tasks. You may also need to save environment variables, such as API keys.

```
firebase functions:config:set someApiKey="XYZ"
```

You can access your environment variables by calling `functions.config().someApiKey` inside the environment.

The `index.js` file is where you will define the function. Most commonly, you will import the admin database to override any read/write/validate rules (see 2.14). You define functions calling `exports.functionName`, which we will see in the upcoming examples in this chapter.

```
var functions = require('firebase-functions');
const admin = require('firebase-admin');
admin.initializeApp(functions.config().firebase);

exports.emptyFunction = functions.https.onRequest((req, res) => {
  console.log("hello world!")
  return;
})
```

6.2 Deploy Cloud Cloud Functions

Problem

You want to deploy your cloud functions.

Solution

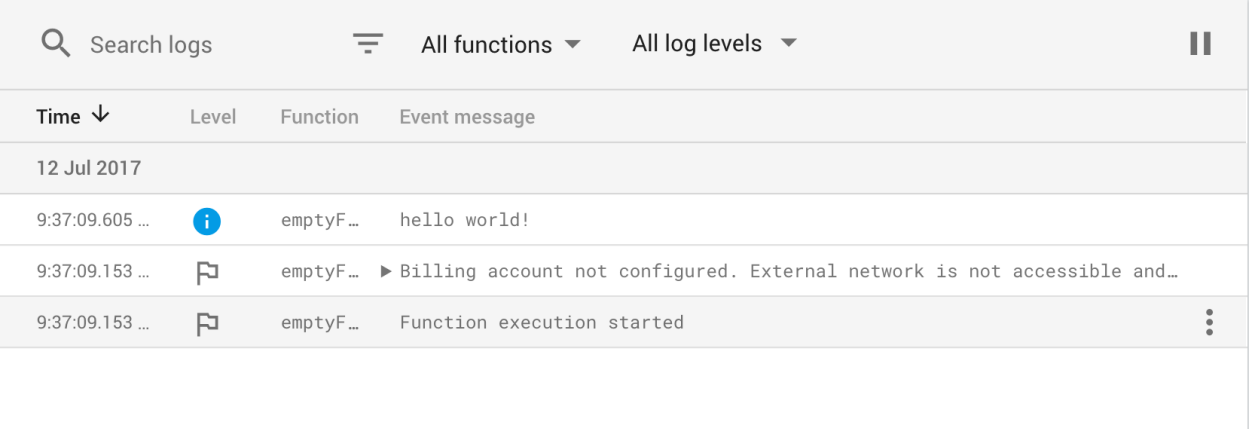
Let's deploy the function from 6.1. It's as simple as:

```
firebase deploy --only functions
```

Firebase should have returned the endpoint URL to trigger this function. Let's hit with cURL, then check the logs to see if it's working.

Function URL (emptyFunction): `your-project.cloudfunctions.net/emptyFunction`

In the firebase console, you should see something like this:



Tip: If you have a custom domain in your project, requests can be proxied to that domain when calling HTTP functions.

6.3 Setup an HTTP Cloud Function

Problem

You want to create a cloud function that is triggered over HTTP.

Solution

An HTTP cloud function will allow you arbitrarily execute code from any event, such as a button click, form submission, etc. It gives you an API endpoint without the need to manage a backend server.

HTTP cloud functions have a request `req` and a response `res`. In most cases, you will parse the request parameters, then finish by calling `response.send()` to send JSON back to the requester.

In this example, the HTTP function returns a word count in JSON format for a specific book in the database. It makes a reference to the database and calls `once('value')` - this will return a single snapshot of the data at this location. From there, we can parse the data into a word count object, convert it to JSON, then send the response.

```
exports.bookWordCount = functions.https
       .onRequest((req, res) => {

  const bookId = req.body.bookId

  if (!bookId) return;

  return admin.database()
              .ref(`/books/${bookId}`)
              .once('value')
              .then(data =>  {
                return data.val()
              })
              .then( book => {
                const wordCount = book.content.split(' ').length;
                const json = JSON.stringify({ words: wordCount })
                res.status(200).send(json)
              })

})
```

Deploy it, then test it using cURL.

```
curl -H "Content-Type: application/json" -d '{"bookId":"atlas-shrugged"}' https:\
//your-endpoint
```

Or you can call it from Angular using the HTTP module.

```
import { Http, Response } from '@angular/http';

constructor(private http: Http)

getWordCount(bookId) {
  const path = "https://your-endpoint/bookWordCount"
  let params = new URLSearchParams();
  params.set('bookId', bookId)

  return this.http.get(path, { search: params})
}
```

6.4 Setup an Auth Cloud Function

Problem

You want to trigger a function when a user signs up or deletes their account.

Solution

Firebase offers two triggers for authentication events of `onCreate` and `onDelete`. Common use cases for the `onCreate` event could be sending a transactional email or updating a notification feed. The `onDelete` function can be used to delete a user's data when they close their account.

```
exports.deleteUserData = functions.auth
       .user()
       .onDelete(event => {

     const userId = event.data.uid;
     const email  = event.data.email;

     return admin.database()
                 .ref(`users/${userId}`).remove();

});
```

6.5 Setup a Database Cloud Function

Problem

You want to update a user's notification feed when their content is liked.

Solution

Database triggers are the most useful type of Firebase Cloud Functions because they solve many common background situations that are impractical/difficult to perform in Angular.

You invoke functions by referencing a specific point in the database, then specify the type of operation trigger. There are four possible triggers.

`onWrite()` - All operations `onCreate()` - New data created `onUpdate()` - Existing data updated `onDelete()` - Data removed

In this example, we will update the user's toast notification feed when they gain a new follower.

```
exports.sendToast = functions.database
       .ref('/followers/{userId}/{username}')
       .onCreate(event => {

     const data = event.data.val()
     const userId = event.params.userId
     const follower = event.params.username

     const message = { message: `You've been followed by ${username}` }

     return admin.database()
                 .ref(`toasts/${userId}`)
                 .push(message);

});
```

Choose the Reference Point Carefully

The database reference point you choose for a cloud function will also fire on any child nodes nested within it. A function that references `/users` will be invoked on a write to `users/userId/deep/data`. You should always point functions to deepest level possible to avoid unnecessary invocations.

6.6 Setup a Firestore Cloud Function

Problem

You want to trigger a cloud function when data changes in a Firestore document.

Solution

The cloud function triggers are identical for Firestore and the Realtime DB, just to recap:

`onWrite()` - All operations `onCreate()` - New document created `onUpdate()` - Existing document updated `onDelete()` - Document removed

Because Firestore is so similar to the Realtime DB, I just going to highlight the important differences. It really just boils down to slightly different terminology:

1. Get data with `event.data.data()`.
2. Get the previous data state with `event.data.previous.data()`.
3. Write data with `event.data.ref.update` or `set` or `delete`.

```
exports.myFunctionName = functions.firestore
  .document('books/bookID').onUpdate((event) => {

    // Current data state
    const data = event.data.data();

    // Data state before the update
    const previousData = event.data.previous.data();

    // Update data on the document
    return event.data.ref.update({
      hello: 'world'
    });
  });
```

6.7 Setup a Storage Cloud Function

Problem

You want to resize images uploaded to firebase storage into thumbnails of various sizes.

Solution

Storage functions are similar to the database functions, but you only have one trigger `onChange()`, which fires on both create and delete. File objects have `resourceState` metadata, or 'exists', or 'not_exists' to distinguish between create and remove.

This final cloud function is by far the most complex. I wanted to demonstrate what a fully fledged, relatively complex, cloud function can look like. Here we are using the sharp NPM package to resize the image, save it to the function's local storage on the underlying virtual instance, then upload it to Firebase.

```
const functions = require('firebase-functions');
const gcs = require('@google-cloud/storage')();
const sharp = require('sharp')
const _ = require('lodash');
const path = require('path');
const os = require('os');

exports.thumbnail = functions.storage
       .object('uploads/{imageName}')
```

```
10        .onChange(event => {
11
12    const object = event.data; // storage object
13
14    const fileBucket = object.bucket;
15    const filePath = object.name;
16    const contentType = object.contentType;
17    const resourceState = object.resourceState;
18
19    const SIZES = [64, 256, 512]; // Resize pixel targets
20
21    if (!contentType.startsWith('image/') || resourceState == 'not_exists') {
22      console.log('This is not an image.');
23      return;
24    }
25
26    if (_.includes(filePath, '_thumb')) {
27      console.log('already processed image');
28      return;
29    }
30
31
32    const fileName = filePath.split('/').pop();
33    const bucket = gcs.bucket(fileBucket);
34    const tempFilePath = path.join(os.tmpdir(), fileName);
35
36    return bucket.file(filePath).download({
37      destination: tempFilePath
38    }).then(() => {
39
40      _.each(SIZES, (size) => {
41
42        let newFileName = `${fileName}_${size}_thumb.png`
43        let newFileTemp = path.join(os.tmpdir(), newFileName);
44        let newFilePath = `/thumbs/${newFileName}`
45
46        sharp(tempFilePath)
47          .resize(size, null)
48          .toFile(newFileTemp, (err, info) => {
49
50            bucket.upload(newFileTemp, {
51              destination: newFilePath
```

```
52              });
53            });
54      })
55    })
56  })
```

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.

7.1 Firestore Custom User Data

Firestore OAuth

https://youtu.be/e8GA1UOj8mE

Problem

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

Solution

There's a good chance you want to keep track of more than just an email address, display name, and userID.

NgModule Setup

Although technically optional, it is a good design pattern to keep your authentication setup in a core module. The purpose of a core module is to provide services that your app will use globally, such as authentication, logging, toast messages, etc.

```
1  ng g module core
```

Add the AngularFire Firestore and Auth modules to your core module.

```
// core.module.ts
import { NgModule } from '@angular/core';
import { AuthService } from './auth.service';
import { AngularFireAuthModule } from 'angularfire2/auth';
import { AngularFirestoreModule } from 'angularfire2/firestore';
@NgModule({
  imports: [
    AngularFireAuthModule,
    AngularFirestoreModule
  ],
  providers: [AuthService]
})
export class CoreModule { }
```

Then import the core module in the app module.

```
// app.module.ts

import { CoreModule } from './core/core.module';
// ...
@NgModule({
  // ... omitted
  imports: [
    BrowserModule,
    AppRoutingModule,
    AngularFireModule.initializeApp(yourConfig),
    CoreModule // <-- add core module
  ]
})
export class AppModule { }
```

Auth Service

The auth service is where most of the magic happens. It facilitates the sign-in process, watches the user session, and allows us to save custom user data to the Firestore database. Here's a breakdown of the important steps in the service code.

- `interface User`: The interface declares the properties of the custom user object. Feel free to add any custom data you want here to extend the basic Firebase auth data.
- `constructor()`: The constructor will set the Observable. First it receives the current Firebase auth state. If present, it will hit up Firestore for the user's saved custom data. If null, it will return an `Observable.of(null)`.

- `googleLogin()`: This method triggers the popup window that authenticates the user with their Google account. It returns a Promise that resolves with the auth credential. The `oAuthLogin()` method is useful if you have multiple OAuth options because it can be reused with different providers.
- `updateUserData()`: This private method runs after the user authenticates and sets their information to the Firestore database.

```
import { Injectable } from '@angular/core';
import { Router } from '@angular/router';

import * as firebase from 'firebase/app';
import { AngularFireAuth } from 'angularfire2/auth';
import { AngularFirestore, AngularFirestoreDocument } from 'angularfire2/firesto\
re';

import { Observable } from 'rxjs/Observable';
import 'rxjs/add/operator/switchMap'

interface User {
  uid: string;
  email: string;
  photoURL?: string;
  displayName?: string;
  favoriteColor?: string;
}


@Injectable()
export class AuthService {

  user: Observable<User>;

  constructor(private afAuth: AngularFireAuth,
              private afs: AngularFirestore,
              private router: Router) {

      //// Get auth data, then get firestore user document || null
      this.user = this.afAuth.authState
        .switchMap(user => {
          if (user) {
            return this.afs.doc<User>(`users/${user.uid}`).valueChanges()
          } else {
```

```
36              return Observable.of(null)
37            }
38          })
39    }
40
41    googleLogin() {
42      const provider = new firebase.auth.GoogleAuthProvider()
43      return this.oAuthLogin(provider);
44    }
45
46    private oAuthLogin(provider) {
47      return this.afAuth.auth.signInWithPopup(provider)
48        .then((credential) => {
49          this.updateUserData(credential.user)
50        })
51    }
52
53
54    private updateUserData(user) {
55      // Sets user data to firestore on login
56
57      const userRef: AngularFirestoreDocument<any> = this.afs.doc(`users/${user.ui\
58  d}`);
59
60      const data: User = {
61        uid: user.uid,
62        email: user.email,
63        displayName: user.displayName,
64        photoURL: user.photoURL
65      }
66
67      return userRef.set(data, { merge: true })
68
69    }
70
71
72    signOut() {
73      this.afAuth.auth.signOut().then(() => {
74          this.router.navigate(['/']);
75      });
76    }
77  }
```

Auth Guard to Protect Routes

```
ng g guard core/auth
```

The next thing we want to do is protect our routes from unauthenticated users. Now that we have an Observable in the service, we can handle this task easily with a `canActivate` guard. We just pass it the Observable of the user from the auth service. If it emits true, the route can be accessed. If false, the user is redirected to the login page.

```
import { Injectable } from '@angular/core';
import { CanActivate, ActivatedRouteSnapshot, RouterStateSnapshot, Router } from\
 '@angular/router';

import { AuthService} from './auth.service'
import { Observable } from 'rxjs/Observable';
import 'rxjs/add/operator/do';
import 'rxjs/add/operator/map';
import 'rxjs/add/operator/take';

@Injectable()
export class AuthGuard implements CanActivate {
  constructor(private auth: AuthService, private router: Router) {}


  canActivate(
    next: ActivatedRouteSnapshot,
    state: RouterStateSnapshot): Observable<boolean> | boolean {

      return this.auth.user
           .take(1)
           .map(user => !!user)
           .do(loggedIn => {
             if (!loggedIn) {
               console.log('access denied')
               this.router.navigate(['/login']);
             }
         })

  }
}
```

You can use the guard in your router by simply adding it to the `canActivate` array for a given route, for example:

```
const routes: Routes = [
  ///...
  { path: 'url-path', component: SuperSecretComponent, canActivate: [AuthGuard] \
},
];
```

User Profile Component

As a final step, let's create a user profile so you can see how to use the `user` Observable in the HTML.

```
ng g component user-profile -m app
```

The component TypeScript just needs to have the auth service injected as a public property in the constructor.

```
import { Component } from '@angular/core';
import { AuthService } from '../../core/auth.service';

@Component({
  selector: 'user-profile',
  templateUrl: './user-profile.component.html',
  styleUrls: ['./user-profile.component.scss']
})
export class UserProfileComponent {

  constructor(public auth: AuthService) { }
}
```

In the component HTML, we have two separate templates that are shown conditionally based on the `user` Observable data. If it's null, we show the guest template, but if it's present we can show the `authenticated` template and corresponding user data.

I have also added a Google Login button to the profile, but you might consider making it a standalone component that you can use outside of the user profile.

```
<div *ngIf="auth.user | async; then authenticated else guest">
        <!-- template will replace this div -->
</div>


<!-- User NOT logged in -->
<ng-template #guest>
    <h3>Howdy, GUEST</h3>
    <p>Login to get started...</p>

    <button (click)="auth.googleLogin()">
        <i class="fa fa-google"></i> Connect Google
    </button>

</ng-template>


<!-- User logged in -->
<ng-template #authenticated>
    <div *ngIf="auth.user | async as user">
        <h3>Howdy, {{ user.displayName }}</h3>
        <img  [src]="user.photoURL">
        <p>UID: {{ user.uid }}</p>
        <p>Favorite Color: {{ user?.favoriteColor }} </p>
        <button (click)="auth.signOut()">Logout</button>
    </div>
</ng-template>
```

7.2 Star Rating System with Firestore

Firestore Quick Start Video Lesson

https://youtu.be/I2i3gXoTmcw

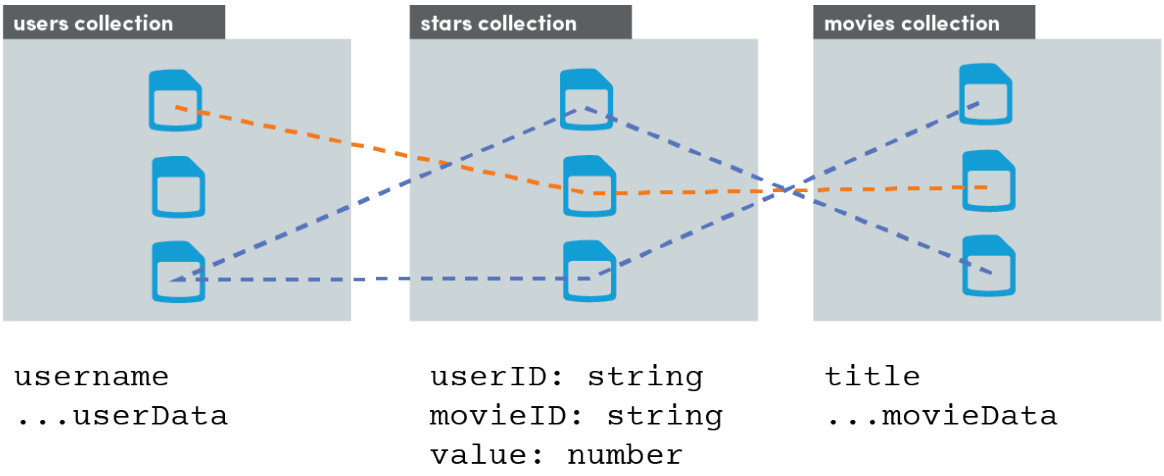
Problem

You want to create your own five-star-review system (like Yelp!) with Firestore from scratch.

Solution

Knowing how to implement star reviews is an important skill for a developer because the same concepts are used for likes, hearts, votes, and many other common UX features.

Firestore Data Structure



How do we model star-ratings in a NoSQL database like firestore? In the SQL world, this is known as a `many-to-many-through` relationship where *Users have many Movies through Reviews* AND *Movies have many Users through Reviews*

Big Collections, Small Documents

Firestore is optimized to handle large collections of small documents - always keep this in mind when designing your backend architecture.

In the diagram 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.

App Component or Parent Component

Passing Data to Child Components

For simplicity, I am using the app component, but you can apply this logic to any parent-child component relationship.

For this demonstration, I am going to manually create a few `user` documents and a `movie` document. As you will see, the `StarService` is completely independent, so you can easily drop it into your app. All you need is a reference to the two `AngularFirestoreDocument` objects that you want to connect.

Here's what the database collections and documents look like in Firestore.

In the app component, we will make a reference to the current user and movie. In the real world, you would get your user from an auth service. Movies might come from a URL param or a collection of movies. These issues are not directly relevant to our star feature, so I am simply hard coding them in the app component.

I also created a couple of getters to retrieve the document ID from the `AngularFirestoreDocument`.

```
import { Component, OnInit } from '@angular/core';
import { AngularFirestore, AngularFirestoreDocument } from 'angularfire2/firesto\
re';
import { Observable } from 'rxjs/Observable';

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

  userDoc: AngularFirestoreDocument<any>;
  movieDoc: AngularFirestoreDocument<any>;

  user: Observable<any>;
  movie: Observable<any>;

  constructor(private afs: AngularFirestore) { }

  ngOnInit() {
    this.userDoc = this.afs.doc('users/test-user-3')
    this.movieDoc = this.afs.doc('movies/battlefield-earth')

    this.movie = this.movieDoc.valueChanges()
    this.user = this.userDoc.valueChanges()
  }

  get movieId() {
    return this.movieDoc.ref.id
  }
```

```
32
33    get userId() {
34      return this.userDoc.ref.id
35    }
36
37  }
```

The typescript getters will allow us to conveniently pass the document ids to a child component, which we are going to build later in the lesson.

```
1  <star-review [movieId]="movieId" [userId]="userId"></star-review>
```

```
1  ng g service star --module app
```

The star service will handle interaction with the stars collection in the Firestore back-end database.

- Get reviews by User
- Get reviews by Movie
- Set/update a review relationship

```
1  import { Injectable } from '@angular/core';
2  import { AngularFirestore, AngularFirestoreDocument } from 'angularfire2/firesto\
3  re';
4  import { Observable } from 'rxjs/Observable';
5
6  export interface Star {
7    userId: string;
8    movieId: string;
9    value: number;
10 }
11
12
13 @Injectable()
14 export class StarService {
15
16   constructor(private afs: AngularFirestore) { }
17
18   // Star reviews that belong to a user
19   getUserStars(userId) {
```

```
    const starsRef = this.afs.collection('stars', ref => ref.where('userId', '==\
', userId) );
    return starsRef.valueChanges();
  }

  // Get all stars that belog to a Movie
  getMovieStars(movieId) {
    const starsRef = this.afs.collection('stars', ref => ref.where('movieId', '=\
=', movieId) );
    return starsRef.valueChanges();
  }

  // Create or update star
  setStar(userId, movieId, value) {
    // Star document data
    const star: Star = { userId, movieId, value };

    // Custom doc ID for relationship
    const starPath = `stars/${star.userId}_${star.movieId}`;

    // Set the data, return the promise
    return this.afs.doc(starPath).set(star)
  }

}
```

The StarReview Component

```
ng g component star-review
```

You can drop the star review component into any other component that has access to a movie reference. It acts as a child component that will display/update reviews for its parent component - in this case movies.

The star review component will perform the following tasks.

- Show the average star rating
- List other users star ratings
- Allow user so set their own rating

The star service only requires a `userId` and `movieId`, therefore we can pass these values from the parent component using the `@Input` decorator.

```
import { Component, OnInit, Input } from '@angular/core';
import { StarService } from '../star.service';
import { Observable } from 'rxjs/Observable';

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


  @Input() userId;
  @Input() movieId;

  stars: Observable<any>;
  avgRating: Observable<any>;

  constructor(private starService: StarService) { }

  ngOnInit() {
    this.stars = this.starService.getMovieStars(this.movieId)

    this.avgRating = this.stars.map(arr => {
      const ratings = arr.map(v => v.value)
      return ratings.length ? ratings.reduce((total, val) => total + val) / arr.\
length : 'not reviewed'
    })
  }

  starHandler(value) {
    this.starService.setStar(this.userId, this.movieId, value)
  }


}
```

To implement the clickable star buttons, we need to style radio buttons as star icons. In this case, we have 10 clickable radio buttons ranging from 0.5 to 5 stars. When clicked buy the user, it will trigger the `starHandler(val)` method and update the corresponding data in Firestore.

Rather than code 10 different inputs, I loop over 5 integers and wrap the full-star and half-star in an `ng-container` - that reduces the HTML code by ~80%.

Note: It's important that the the `id` on the input matches the `for` property on the label.

```
<h3>Average Rating</h3>

{{ avgRating | async }}


<h3>Reviews</h3>
<div *ngFor="let star of stars | async">

  {{ star.userId }} gave {{ star.movieId }} {{ star.value }} stars

</div>


<h3>Post your Review</h3>

<fieldset class="rating">
  <ng-container  *ngFor="let num of [5, 4, 3, 2, 1]">
      <!-- full star -->
      <input (click)="starHandler(num)"
            [id]="'star'+num"
            [value]="num-0.5"
            name="rating"
            type="radio" />

      <label class="full" [for]="'star'+num"></label>

      <!-- half star -->
      <input (click)="starHandler(num-0.5)"
             [value]="num-0.5"
             [id]="'halfstar'+num"
             name="rating"
             type="radio"  />

      <label class="half" [for]="'halfstar'+num"></label>

  </ng-container>
</fieldset>
```

Lastly, I wanted to include the CSS that makes the star UI possible, originally used in this CodePen[4]. Note: you will also need FontAwesome icons installed in your project for the CSS to work properly.

[4] https://codepen.io/jamesbarnett/pen/vlpkh

```
fieldset, label { margin: 0; padding: 0; }
body{ margin: 20px; }
h1 { font-size: 1.5em; margin: 10px; }

.rating {
  border: none;
  float: left;
}

.rating > input { display: none; }
.rating > label:before {
  margin: 5px;
  font-size: 1.25em;
  font-family: FontAwesome;
  display: inline-block;
  content: "\f005";
}

.rating > .half:before {
  content: "\f089";
  position: absolute;
}

.rating > label {
  color: #ddd;
 float: right;
}

.rating > input:checked ~ label,
.rating:not(:checked) > label:hover,
.rating:not(:checked) > label:hover ~ label { color: #FFD700;  }

.rating > input:checked + label:hover,
.rating > input:checked ~ label:hover,
.rating > label:hover ~ input:checked ~ label,  selection
.rating > input:checked ~ label:hover ~ label { color: #FFED85;  }
```

7.3 Give Firebase Users Custom Usernames (Realtime Database)

Custom Usernames Video

https://youtu.be/NLWHEiH1FZY

Problem

You want to give firebase users custom usernames, then validate them asynchronously after each keypress.

Solution

Get started by implementing the user authentication paradigm of your choice from chapter 3.

You will notice that Firebase authentication is convenient, but you cannot assign custom usernames out of the box. In this example, we are going to give users custom usernames and asynchronously validate their availability during the signup process. On every `keyup` event, the username will be checked against the database for duplicates.

When a user signs up, the app will look for a username. If it is missing, Angular will keep them on the login component and require a username to be entered, then validate its availability by querying the database for a match.

Modeling the Data

The database has a collection of users for basic record keeping. However, the quickest way to asynchronously check username availability is to save all usernames in their own collection, with the keys being the usernames that are not available to new users. The database structure looks like this:

```
-|users
    -|$authUID
        username: "jeffd23"

-|usernames
    jeffd23: $authUID
```

First, let's create a `User` class to simplify the auth object. We only care about the uid and the username. As a constructor, it will take the Firebase AuthState from AngularFire2.

We need to subscribe to both the Firebase auth object and the extra user information (i.e. username) in the database at the same time. So, how do we handle nested subscriptions with RxJS? In this case, we are going to use `switchMap`, which will emit the Firebase auth object, then get the user values from the database, keeping everything packaged up as an Observable. Create a new service to handle this logic with `ng generate service auth`.

auth.service.ts

```
import { Injectable } from '@angular/core';
import { AngularFireDatabase } from 'angularfire2/database';
import { AngularFireAuth } from 'angularfire2/auth';
import * as firebase from 'firebase';
import 'rxjs/add/operator/switchMap';

export class User {

  uid: string;
  username: string = '';

  constructor(auth) {
    this.uid = auth.uid
  }

}

@Injectable()
export class AuthService {

  currentUser: User;

  constructor(private afAuth: AngularFireAuth,
              private db: AngularFireDatabase) {

              this.afAuth.authState.switchMap(auth => {
                  if (auth) {
                    this.currentUser = new User(auth);
                    const ref = this.db.object(`/users/${auth.uid}`;
                    return ref.valueChanges()
                  } else return [];
                })
```

```
                .subscribe(user => {
                    this.currentUser['username'] = user.username;
                })

            }

  googleLogin() {
    const provider = new firebase.auth.GoogleAuthProvider()
    return this.afAuth.auth.signInWithPopup(provider)
  }
}
```

Now it's time to verify the availability of the selected username. First, the username collection is queried with the user's text input. Querying with this method only targets a single key value pair, rather than an entire list, making it much more efficient. We also use a TypeScript getter to determine if the current user already has a username. Here's how the remainder of the service should look:

```
  get hasUsername() {
    return this.currentUser.username ? true : false
  }

  checkUsername(username: string) {
    username = username.toLowerCase()
    const ref = this.db.object(`usernames/${username}`)
    return ref.valueChanges()
  }

  updateUsername(username: string) {
    let data = {}
    data[username] = this.currentUser.uid

    this.db.object(`/users/${this.currentUser.uid}`)
           .update({"username": username})

    this.db.object(`/usernames`)
           .update(data)
  }
```

The service is ready to go - let's wire up the component. It runs the query to Firebase after each `keydown` event to determine if a matching username exists. If not, the user can go ahead and select it. Generate a component with `ng generate component user-login`, then inject the auth service.

user-login.component.ts

```
import { Component } from '@angular/core';
import { AuthService } from "../auth.service";

@Component({
  selector: 'user-login',
  templateUrl: './user-login.component.html',
  styleUrls: ['./user-login.component.scss']
})
export class UserLoginComponent {

  usernameText: string;
  usernameAvailable: boolean;

  constructor(public auth: AuthService) { }

  checkUsername() {
    this.auth.checkUsername(this.usernameText)
             .subscribe(username => {
                this.usernameAvailable = !username.$value
    })
  }

  updateUsername() {
    console.log
    this.auth.updateUsername(this.usernameText)
  }

  signInWithGoogle() {
    this.auth.googleLogin()
  }
}
```

user-login.component.html

In the template, these event handlers are bound to the corresponding elements.

```
<h1>Login</h1>
<button (click)="signInWithGoogle()"
        *ngIf="!auth.currentUser">
         Connect Google
</button>

<button  *ngIf="auth.currentUser"
         (click)="logout()">
          Logout
</button>

<div *ngIf="auth.currentUser && !auth.hasUsername">

  <h3>Choose a Username</h3>

  <input type="text"
         [(ngModel)]="usernameText"
         (keyup)="checkUsername()">

  <p *ngIf="usernameAvailable && usernameText">
    Success!! @{{usernameText}} is available
  </p>

  <p *ngIf="!usernameAvailable && usernameText">
    Danger!! @{{usernameText}} has already been taken
  </p>

  <button [disabled]="!usernameAvailable || !usernameText"
          (click)="updateUsername()">
            Select Username1
  </button>
</div>
```

That's it. You now have a simple custom username system.

database.rules.json

Just one last thing to consider. The current username validation is great as a frontend UX feature, but it is still vulnerable to accidental duplication. Let's add an extra layer of security by creating a Firebase database rule that ensures a username cannot be accidentally duplicated on the backend.

```
"users": {
  ".write": "auth != null",
  "username": {
      ".validate": "!root.child('usernames').child(newData.val()).exists()"
  }
},
"usernames": {
   ".write": "auth != null"
}
```

7.4 Accept Payments with Stripe (Realtime Database)

Stripe Payments Video

https://youtu.be/_lZc2O2oUJk

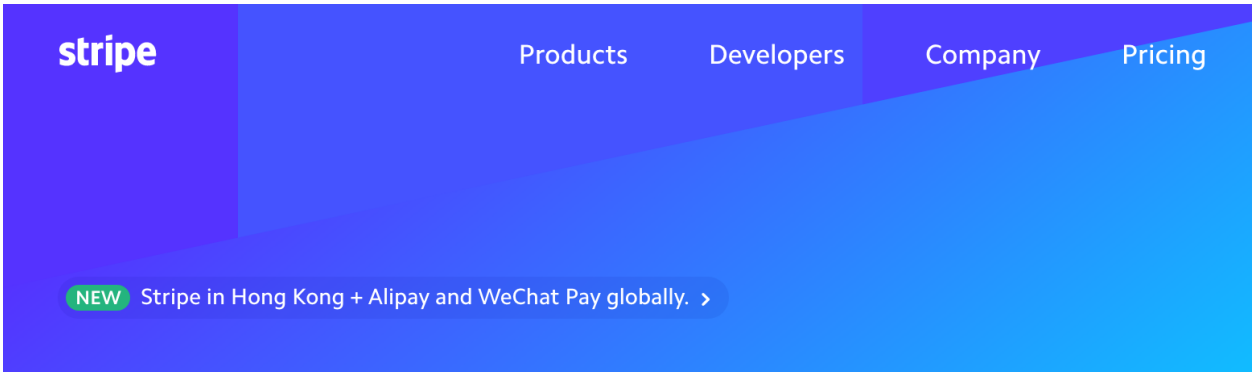
Problem

You want to collect payments from customers with Stripe.

Solution

The final example will use the Stripe Payments API to collect credit card information from customers (without sensitive information ever touching your app), then charge the card on the backend with a Firebase Cloud Function.

First things first, head over to stripe.com and obtain an API key.



Let's review how the payment process works with Stripe, Angular, and Firebase.

1. User triggers the checkout window, enters card details
2. Stripe returns a charge token
3. Token is saved to the Firebase database
4. Cloud function processes the actual charge and updates the database.

Initial Setup

We need the stripe checkout library in our project. Stripe recommends making this script global to enable better fraud detection, so let's add it the head of the `index.html`. In fact, it must be loaded directly from Stipe – using a local NPM package is not supported.

```
<head>
  <script src="https://checkout.stripe.com/checkout.js"></script>
</head>
```

The StripeCheckout class is not known to TypeScript, so we need declare it in `typings.d.ts`.

```
declare var StripeCheckout:any;
```

Lastly, you should add your Stripe API keys to your environment files. Add the test key to the `environment.ts` file.

```
export const environment = {
  production: false,
  stripeKey: 'YOUR_STRIPE_TEST_KEY',
  // Your Firebase Config {}
};
```

Database Design

Stripe returns the data for the token and charge in JSON. Each payment will be nested under its associated `userId` in the database.

```
payments
    $userId
        $paymentId
            amount: number
            token: object (from Stripe)
            charge: object (from Stripe)
```

make-payment.component.html

The payment component's html is very simple. Just a button to trigger the stripe checkout window.

```
<button (click)="handlePayment()">
  Add Credits to your Account
</button>
```

make-payment.component.ts

Now we need to modify the Stripe Custom Integration[5] to work with Angular. When stripe returns the token, we save it to Firebase using the service defined in the next step.

Note: Stripe amounts are equal to 1/100th of the base currency, so an amount of 500 equals $5.00.

```
import { Component, OnInit, HostListener } from '@angular/core';
import { PaymentService } from '../payment.service';
import { environment } from '../../../environments/environment';
@Component({
  selector: 'make-payment',
  templateUrl: './make-payment.component.html',
  styleUrls: ['./make-payment.component.scss']
})
export class MakePaymentComponent implements OnInit {

  handler: any;
  amount: number = 500;

  constructor(private paymentSvc: PaymentService ) { }

  ngOnInit() {
    this.handler = StripeCheckout.configure({
      key: environment.stripeKey,
      image: '/your/awesome/logo.jpg',
      locale: 'auto',
      token: token => {
        this.paymentSvc.processPayment(token, this.amount)
      }
    });
  }

  handlePayment() {
    this.handler.open({
      name: 'FireStarter',
      description: 'Deposit Funds to Account',
      amount: this.amount
```

[5] https://stripe.com/docs/checkout#integration-custom

```
32      });
33    }
34
35    @HostListener('window:popstate')
36      onPopstate() {
37        this.handler.close()
38      }
39
40  }
```

payment.service.ts

The service just needs to get the current user's ID, then save the Stripe token to the database.

```
1   import { Injectable } from '@angular/core';
2   import { AngularFireDatabase } from 'angularfire2/database';
3   import { AngularFireAuth } from 'angularfire2/auth';
4
5   @Injectable()
6   export class PaymentService {
7
8     userId: string;
9
10    constructor(private db: AngularFireDatabase,
11                private afAuth: AngularFireAuth) {
12      this.afAuth.authState.subscribe((auth) => {
13        if (auth) this.userId = auth.uid
14      });
15    }
16
17     processPayment(token: any, amount: number) {
18       const payment = { token, amount }
19       return this.db.list(`/payments/${this.userId}`).push(payment)
20     }
21
22  }
```

At this point, the card has NOT actually been charged. We just have a token that needs to be processed on a backend server.

Follow these steps to initialize Firebase Cloud Functions in your project and set an environment variable for your Stripe API key.

Note: You can skip `firebase init functions` if your project is already configured with Firebase Cloud Functions.

```
firebase init functions
cd functions
npm install stripe --save
firebase functions:config:set stripe.testkey="YOUR_STRIPE_TEST_KEY"
```

index.js

The function will be triggered during the `onWrite()` database event. When data is written the `/payments/{userId}/{paymentId}` reference point, it will trigger the function below. It chains together three separate promises that do the following:

1. Queries the database for the user's record.
2. Sends the token to the Stripe API to charge the card.
3. Saves Stripe's charge response to the database.

```
const functions = require('firebase-functions')
const admin = require('firebase-admin')

admin.initializeApp(functions.config().firebase);

const stripe = require('stripe')(functions.config().stripe.testkey)

exports.stripeCharge = functions.database
                                .ref('/payments/{userId}/{paymentId}')
                                .onWrite(event => {                              \


  const payment = event.data.val();
  const userId = event.params.userId;
  const paymentId = event.params.paymentId;

  // checks if payment exists or if it has already been charged
  if (!payment || payment.charge) return;

  return admin.database()
              .ref(`/users/${userId}`)
              .once('value')
              .then(snapshot => {
                  return snapshot.val();
               })
               .then(customer => {
```

```
27
28                  const amount = payment.amount;
29                  const idempotency_key = paymentId;  // prevent duplicate charges
30                  const source = payment.token.id;
31                  const currency = 'usd';
32                  const charge = {amount, currency, source};
33
34
35                  return stripe.charges.create(charge, { idempotency_key });
36
37                })
38
39                .then(charge => {
40                    admin.database()
41                         .ref(`/payments/${userId}/${paymentId}/charge`)
42                         .set(charge)
43                   })
44
45});
```

Now deploy the function with `firebase deploy --only functions` and your app is ready to start accepting payments.

Angular has a realtime connection with Firebase, so the user will automatically see the charge details updated clientside, assuming they are subscribed to this data.

The End

That's it! My hope is that this book serves as a useful reference in your Angular Firebase app development journey. Things change quickly in the web development world, so make sure stay up to date with the latest trends.

- Watch the Latest YouTube Lessons[6]
- Chat on the Angular Firebase Developers Slack Team[7]
- Access advanced content as a PRO Member[8]

Talk to you soon!

– Jeff

[6] https://www.youtube.com/c/angularfirebase
[7] https://angularfirebase.slack.com
[8] https://angularfirebase.com/pro/