FIREBASE JUMPSTART

ANGULAR, FIREBASE AND ANGULARFIRE CRASH COURSE

ANGULAR UNIVERSITY

Table Of Contents

Firebase Jumpstart

Section 1 - Introduction, Why Firebase?

Welcome to the Firebase Jumpstart Book, thank you for joining! The goal of this book is to give you a practical overview of the Firebase real-time database.

I hope you will enjoy this book, please send me your feedback at admin@angular-university.io.

In this book we will go over some reasons on why the latest Firebase might be just as impactful in web development as Angular itself, and why the two combined could be the best thing that happened to web development in a long time.

After all the advances in technology, building web applications is still way harder than what it should be, but maybe not so much anymore if we can use something similar to Firebase.

Which backend should I choose?

Let's say that you are starting a new application from scratch, and you have decided to use Angular for the frontend. That's a great start!

And then you start building your backend in your usual preferred technology, under the form of a REST API that gets data in and out data out of Postgresql, MySql or your favorite SQL Database, or even Mongo.

You probably will use one of the following:

- If you are a Ruby Developer there is a very high chance that you go with Rails
- If you are a Python Developer you might go with Django
- If you are a Node Developer, maybe you go for the MEAN stack, maybe you will use Hapi instead of Express, or go Websockets and try Socket.io or even Meteor. Using a SQL Database? Sequelize works great!
- If you are an enterprise Java Developer you will probably roll up a Spring/Hibernate backend with Spring Boot and Spring Data and might even throw in Spring REST if you don't simply use Spring MVC
- If you are in the C# / .Net world you might go with the .NET framework and the Entity Framework, etc.

You would probably go for your Go-To solution that you have been using for years and build a custom REST API as you are used to, and that would work.

But I'm going to ask you before doing that to consider an alternative that could boost a lot your productivity as a Web Developer.

And if you really still want to use REST, we still have some very good news for you!

Most Applications need a solution for the same problems

When building an application, we sometimes think that the application has unique features that require a very specific custom design. In reality, this tends to not be the case: a large portion of applications are really pulling in and out data from the database using a couple of very common data modification patterns and search criteria.

This usually does not apply to the whole application: it's common that an app has a series of screens that are simply CRUD screens, but there is maybe another part of the app that is super custom and looks like an inbox system for exchanging messages like Gmail.

A missed opportunity to build apps faster

In this mix of some very common data modification functionality mixed with some very custom and specific functionality unique to a given application lies both a huge opportunity for optimization and a source of lack of productivity:

We are many times building a fully custom backend when there are large parts of it that could be up and running out of the box. But because a part of it needs to be custom, we end up building the whole thing custom.

Backendless development does not exist in practice

The notion of backendless development and the whole BaaS wave might accidentally have done more harm than good to help spread Backend as a Service as a more commonly used option for application development.

Because it creates the notion that your whole backend can work out of the box, which we know that most of the times is not true, our chat application cannot be fully built without any backend and we know it.

We will likely always need some sort of backend

CRUD operations can be made to work out of the box and many common queries as well, but we will always need some sort of batch job that sanitizes the chat messages for forbidden words for example.

And based on this we might end up not choosing a BaaS solution and going for a fully custom solution where we handcraft a lot of REST controllers one by one to make some very common modification operations, which can be very time consuming, error-prone and repetitive.

BaaS is not an all or nothing value proposition

Using a BaaS solution is not necessarily an all or nothing choice: if a part of our system is fully custom we don't have to build our whole system in a custom way.

We might build 90% of our system using an out of the box solution, and focus our development time, energy and brain cycles on the custom 10% which is probably the heart of our app.

> *And that's what Firebase allows us to do, plus it makes building the custom part of the Backend much simpler, as we will see*

The REST to Relational Impedance Mismatch

Many of us are already building our application as single page apps that consume RESTful JSON endpoints, we are used to seeing data as JSON. But many backends are built using technologies other than Javascript and use a relational database.

This is the source of a huge lack of productivity because we are continuously loading data into a format that is very different than JSON, and we need to map it.

Also, modifications are expensive, because we need ORM frameworks and mapping frameworks to do even the simplest operation. For example, take a look at this JSON object on the frontend:

```
{
  "url": "angular2-hello-world-write-first-application",
  "description": "Angular 2 Tutorial For Beginners - Build Your First App -
  "duration": "2:49",
  "tags": "BEGINNER",
  videoUrl: "https://www.youtube.com/embed/du6sKwEFrhQ",
  "longDescription": "This is step by step guide to create your first Angula
}
```

If we want to modify the lesson title, why can't we modify the object and simply call `db.save(modifiedObject)`, or something very close to it?

Instead what we usually end up doing is:

- build a REST Ajax PUT or PATCH call
- get it in our backend and maybe map into a C#, Java etc object using a mapping framework
- if we are using Node mapping is not an issue
- use an ORM to save this data into a relational table, or an ODM to map into a document store

Why JSON Data Stores?

This is a lot of work to do something very simple: saving a JSON object. The ideal would be to have a JSON data store and that is what the

Firebase Real-Time database essentially is.

The database does have real-time capabilities and a great performance if we need it, but those are not the biggest reasons why we should consider using it.

The biggest reasons to use the Firebase Database

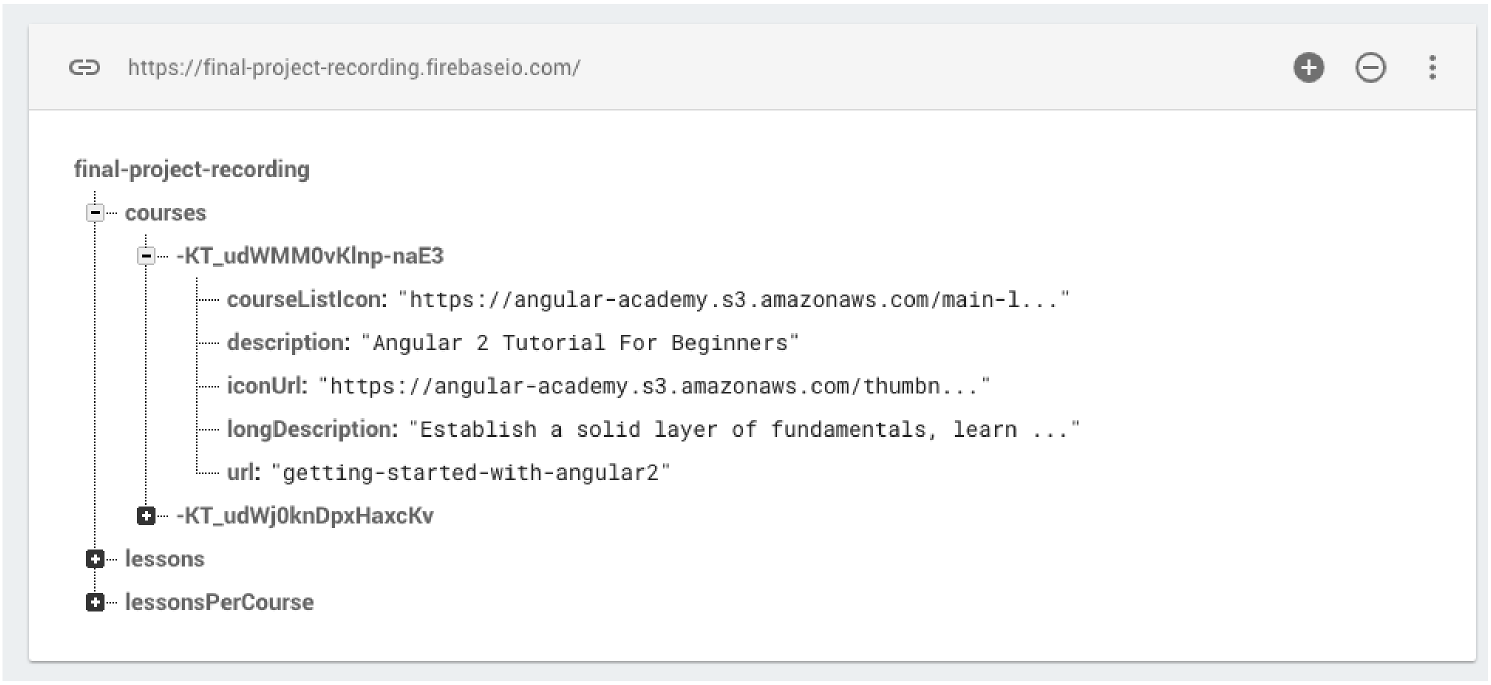
The biggest reason to use the Firebase database its because its a very simple to use JSON data store, which is exactly what we need to build web apps faster.

There is maybe the perception that the Firebase Database it's something to be used to build chat-rooms or apps with real-time needs like games. It's actually a very general purpose database (it's a fork of MongoDB).

Unlike what you might think, most of your knowledge of SQL databases and how to model data will still mostly apply while using the Firebase database, we are going to cover some data modeling in a moment.

Why use JSON Data Stores for Web Development

Using a JSON Data Store removes a lot of the impedance problems that we had in the solutions mentioned above. This is what the data navigation console of a typical database might look like:



```
https://final-project-recording.firebaseio.com/

final-project-recording
  courses
    -KT_udWMM0vKlnp-naE3
      courseListIcon: "https://angular-academy.s3.amazonaws.com/main-l..."
      description: "Angular 2 Tutorial For Beginners"
      iconUrl: "https://angular-academy.s3.amazonaws.com/thumbn..."
      longDescription: "Establish a solid layer of fundamentals, learn ..."
      url: "getting-started-with-angular2"
    -KT_udWj0knDpxHaxcKv
  lessons
  lessonsPerCourse
```

Note: we will go over that strange looking key "KT-etc."

We can think of the Firebase database as a giant JSON object in the sky, the thing is just one big object!

But take a look at the first level of the object: You find nodes named Courses, Lessons. Those suspiciously sound like the names of SQL Tables, don't they?

Section 2 - Firebase Data Modelling

We are going to show how you can easily do data modeling in Firebase by going over an example of a One to Many association.

The goal here is to pass the idea that Data Modeling in Firebase is a lot more similar to the SQL world than we might think!

In our example we are going to have as the Model:

- some courses
- some lessons
- One course can have many lessons
- each lesson only belongs to one course

So it's a pretty familiar one to many association scenarios. So how do we model this in Firebase? This will depend on your application, but for most cases, there is a good way and a wrong way.

Let's start with what is likely the wrong way

Let's say that we add to each course a property named lessons, which is an array of lesson objects:

```
{
  "url": "angular2-hello-world-write-first-application",
  "description": "Angular 2 Tutorial For Beginners - Build Your First App -
  "duration": "2:49",
  "tags": "BEGINNER",
  videoUrl: "https://www.youtube.com/embed/du6sKwEFrhQ",
  "longDescription": "This is step by step guide to create your first Angul
  lessons: [
          {
          "url": "angular2-hello-world-write-first-application",
          "description": "Angular 2 Tutorial For Beginners - Build Your Fir
          "duration": "2:49",
          "tags": "BEGINNER",
          videoUrl: "https://www.youtube.com/embed/du6sKwEFrhQ",
          "longDescription": "This is step by step guide to create your fir
        },
        {
          "url": "angular2-build-your-first-component",
          "description": "Building Your First Angular 2 Component - Compone
          "duration": "2:07",
            "tags": "BEGINNER",
            videoUrl: "https://www.youtube.com/embed/VES1eTNxi1s",
            "longDescription": "In this lesson we are going to see how to inc
          },
          ...
    ]
}
```

> *This might seem like a sensible thing to do, but we probably just created a performance problem.*

What could have gone wrong here?

In some rarer cases, this might be exactly what you need. Let's say that the notion of courses does not make sense in our application without its lessons. That there isn't really any scenario where we will need the course without its lessons.

Then we might as well nest the lessons inside the course, because we are going to need them all the time. But in most cases that is not the case. In most cases, we want to have the data for the courses without having the data for the lessons.

In Firebase a Read Always Reads Everything

And in Firebase, if we do a query to read a course, you will get **all** of the course, including its lessons. There is no way to retrieve only the course data and exclude the lessons property. If it's nested inside the node we query, we get back everything.

In some cases, this might be what we need. But imagine a course with 500 lessons, and you only want to show its description and duration in a list of courses. Real Time or not, modeling the data like we did above would likely bring the database to a halt with only a small number of users.

What do we usually do in the SQL world?

In SQL databases what we would do is:

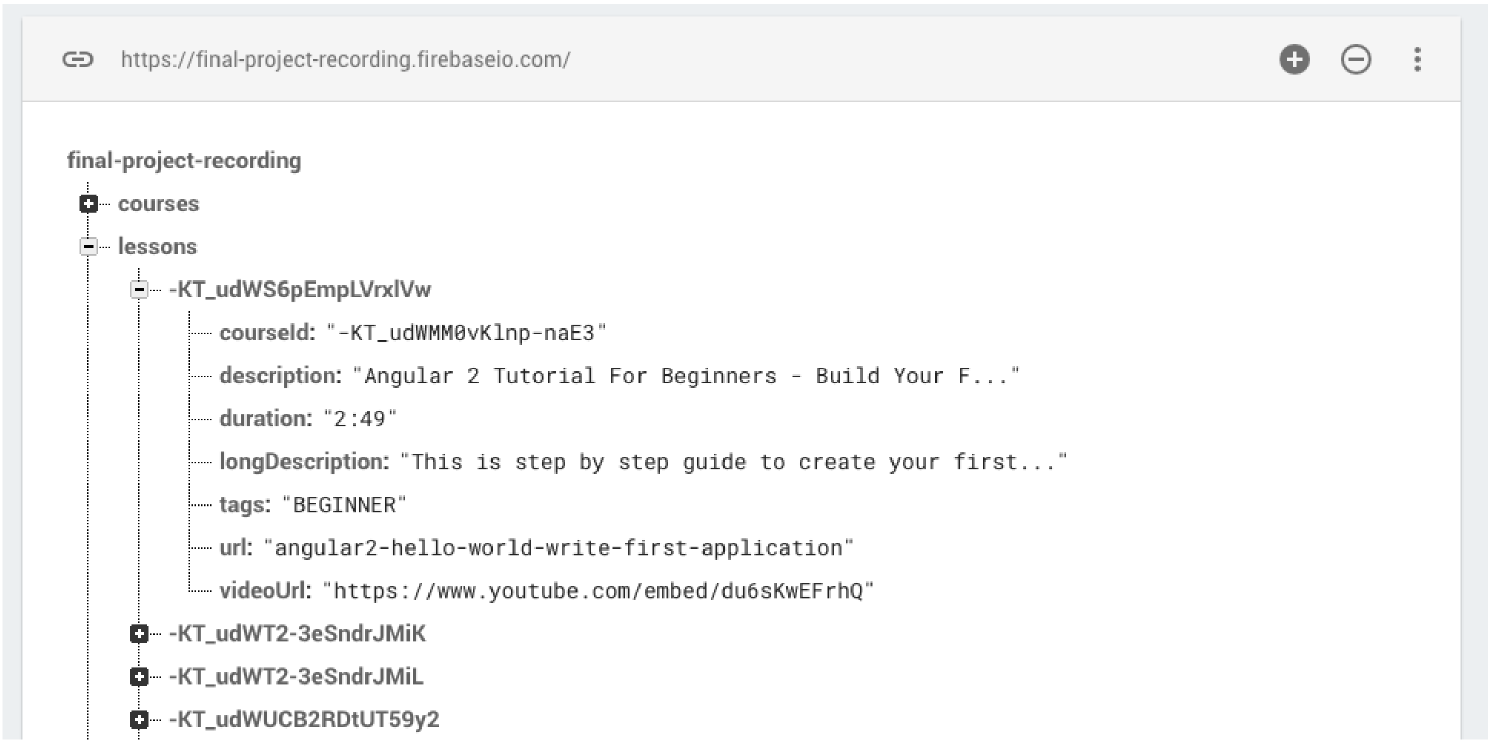
- separate the two relational entities (courses and lessons) into two separate tables
- then query only Courses if we need only the description
- join the two tables if we want the lessons of a given course
- link the lessons with the course via foreign key

And that is exactly what we should do as well in the case of Firebase. The above concepts still apply, some terminology changes but the overall idea is the same.

Modeling a One To Many Association in Firebase

What we will start by doing do is, take the lessons data from inside the course altogether. You might consider leaving in just a list of lesson keys, but imagine having 500 keys there for large courses.

Let's remove the lessons completely and store them in a separate top-level node, which is the closest we can get in Firebase to a relational table:


https://final-project-recording.firebaseio.com/

```
final-project-recording
  courses
  lessons
    -KT_udWS6pEmpLVrxlVw
      courseId: "-KT_udWMM0vKlnp-naE3"
      description: "Angular 2 Tutorial For Beginners - Build Your F..."
      duration: "2:49"
      longDescription: "This is step by step guide to create your first..."
      tags: "BEGINNER"
      url: "angular2-hello-world-write-first-application"
      videoUrl: "https://www.youtube.com/embed/du6sKwEFrhQ"
    -KT_udWT2-3eSndrJMiK
    -KT_udWT2-3eSndrJMiL
    -KT_udWUCB2RDtUT59y2
```

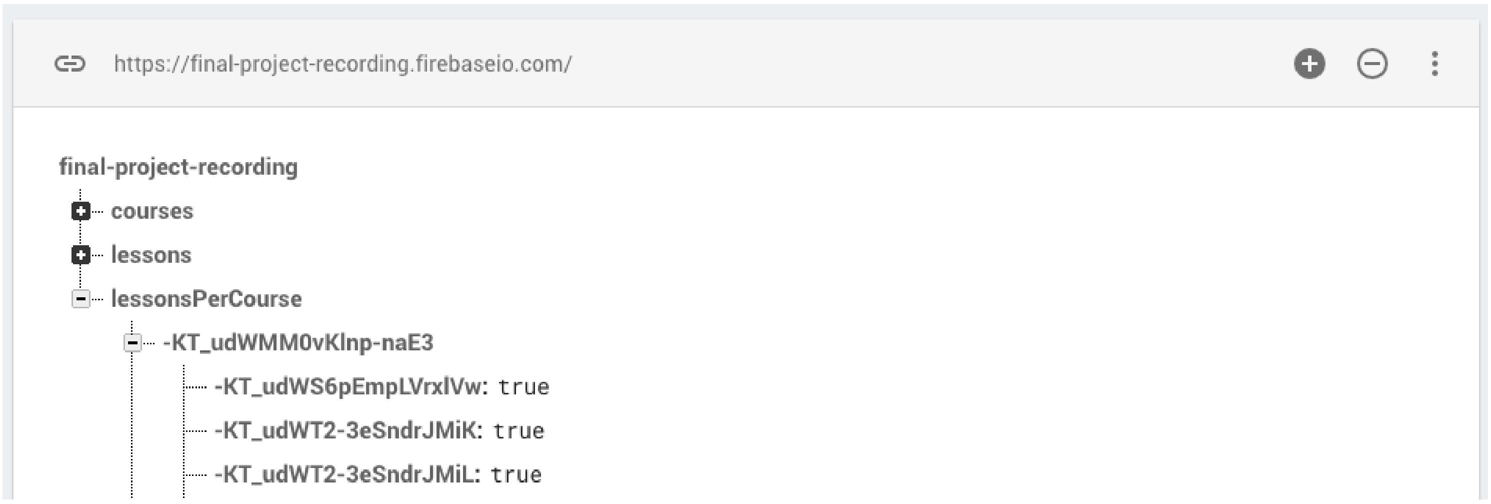
We will get to those keys in a second, and they do have some awesome properties! We can now see that the `lessons` node is a list and contains a list of IDs, which are the lesson unique identifiers. By expanding the ID property we get the lesson data plus a `courseId`.

This property suspiciously looks like a foreign key in a SQL database. A foreign key is more than that, it's a declarative data integrity constraint, and Firebase does support declarative constraints via the Security Rules, which despite the name are not only for security in the traditional sense.

How do we then link now the course to its lessons?

Creating an association node in Firebase

In order to link the data together, what we can do is create a `lessonsPerCourse` node in our database:


https://final-project-recording.firebaseio.com/

```
final-project-recording
  courses
  lessons
  lessonsPerCourse
    -KT_udWMM0vKlnp-naE3
      -KT_udWS6pEmpLVrxlVw: true
      -KT_udWT2-3eSndrJMiK: true
      -KT_udWT2-3eSndrJMiL: true
```

And under it we create one entry per course, using the course key. Then we create a list under that property with the list of lesson keys. In JSON we cannot add a property to an object without adding a value as well.

But in this case we really only want to create a link between course and lesson, so we don't need the value. So by convention, we are going to set the value to "true".

If by any change the association between lesson and course had association properties, like an association start and end date, we would like to store those association properties as the value of this node (instead of "true").

Model cautiously, denormalize if needed

The solution that we presented above to do an association would likely be the best choice in most cases. In general, you want to keep your data as flat as possible, but it's possible that you might need a different structure.

This is just like the case of SQL databases: sometimes the fully normalized data model does not apply and we need to denormalize.

When in doubt doing data modeling in Firebase, we suggest to error on the side of flatness. At the same time, in Firebase denormalization is normal: do not hesitate to create multiple views of the same data that are specific to multiple screens, and keep them in sync with multi-path updates.

We suggest to always include in your database a fully normalized version of the data, and create extra denormalized views of the same data when and if needed.

Other data modeling concepts could also be explained in similar terms, like many to many associations etc. it would all be very familiar. But one thing that might be bugging you by now is, what are those strange looking keys?

Do we really need to use them? Can't we just use integer sequential numbers instead?

Firebase Keys

I really think you should embrace the Firebase keys from the beginning, it's best to not try to go around them as they will save you a world of pain:

- These keys are designed to work in a distributed environment
- They can be synchronously generated on the client side even in offline mode

Is all just random characters?

No, the first part of the key contains a timestamp up to the millisecond, and the second part of the key contains a random number. But it's better not to try to extract the timestamp from the key even though it would be possible if we need the timestamp it's better to store it separately in an object property instead.

This means that if we use these keys as the keys of your lists, the lists will be naturally ordered by timestamp up to the millisecond precision, and after that inside each millisecond the keys will be in random order, not necessarily in the order in which they were generated.

We are going to see these keys in action in the upcoming sections, right now you are probably wondering the following:

> *How do I query the data, how do I modify it What about Joins, how do I join the data back?*

We are going to answer these questions, but before that one thing that its better to know is that right now you cannot do SQL `like` queries directly in Firebase.

If you want powerful search its better for the moment to set up a dedicated search solution like its explained here, the Firebase API is constantly expanding and might include more search options in the future. But nothing replaces this type of Google-like full-text search capability.

Still, the Firebase SDK does give us some very powerful querying capabilities, without the need to learn a separate query language.

Section 3 - The Firebase SDK

The Firebase SDK, how to use it with Typescript

In order to read and write from the Firebase Real Time Database we have a couple of options:

- use only the Firebase SDK directly as a Javascript library
- use the Firebase SDK with Typescript
- use AngularFire, which allows us to access the Firebase SDK as well

Let's start by installing all of this as we might be using everything at a given point:

```
npm install firebase angularfire2 @types/firebase
```

The Firebase DB is really just one big object

To demonstrate that the Firebase database is really one big JSON object in the sky, we are going to read the whole database in one go and print it to the console:

```
import {initializeApp,database} from 'firebase';

export const firebaseConfig = {
    apiKey: "AIzaSyA0BcUcu4V8aHT_gM-32BhRcmqji4z-xts",
    authDomain: "some-app.firebaseapp.com",
    databaseURL: "https://some-app.firebaseio.com",
    storageBucket: "some-app.appspot.com",
    messagingSenderId: "290354329699"
};

initializeApp(firebaseConfig);

```

```
database().ref().on('value', snapshot => console.log(snapshot.val()));


```

This prints the whole database to the console, you do not want to do this! You usually want to query an inner node of the database, like a course or a lesson.

But there is something more going on here:

> *If someone changes anything in the database, a whole new value of the database will be streamed back to the browser and printed in the console.*

The callback that we pass in that receives a snapshot member variable and logs it will be called each time that the database is changed. This will happen via server push and we are going to learn more about it later.

Something else happened here:

> *you have just cached the whole database in the browser!*

The data read from the Firebase SDK is cached, there is a cache layer that will ensure that if you repeat the same query the data is retrieved from the client side cache.

Firebase References and Snapshots

The Firebase SDK uses in its API a couple of notions some of which we just saw here: References and Snapshots.

What we are going to do now is understand what happened with the new value being received from the database, how does this work. Did we use a long polling Ajax request?

WebSockets and Their Advantages

The communication between the Firebase SDK client running in the browser and the Firebase Real Time Database is done via a Websocket if the client browser supports it, otherwise, the SDK will transparently fallback to Ajax long polling.

Note that this is all transparent and internal to the Firebase SDK, we will never have to code Websockets directly in order to use Firebase.

Why are Websockets faster?

A Websocket uses a long-running TCP/IP connection so transferring data back and forth does not include the setup cost of a TCP/IP connection each time, unlike the case of an Ajax Request.

Also with a websocket if you want to send an object over the wire, you only pay the payload cost of the object, and not all the standard HTTP headers that the browser automatically includes in a normal Ajax request. Those headers can actually be many times larger than the data itself (for small payloads).

If you just want to send a counter over the wire, you might be transferring a payload hundreds of times larger than what you would expect due to these extra headers.

Data Joins in Firebase

Because the connection between the SDK client and the database is so fast as its via a Websocket, it's doable to do data joins on the client side up for the amount of data that is normal to display in a user interface, and even quite some mote data than that.

Firebase by itself does not have support for joining queries on the server, you need to query each path you need by doing several queries:

- get the course from `courses`, get its ID
- get the ids of the lessons for the course from `lessonsPerCourse`
- get the lessons for the course going into `lessons` based on each lesson ID

If this proves an issue at scale, you can always create a "view" for the data, where you create a node where you have the lessons data for a given course. Chances are, this will never happen if you are paging the data.

This is not so different to what we do in SQL databases: If a query is too heavy we do it before hand and store the results in another Reporting output table. That table is easier to query and provides a persistent aggregated view of the more fine-grained data.

A Summary of What the Firebase SDK includes

To summarize, the Firebase SDK includes everything that we need to interact with the Firebase real-time database (we are going to cover authentication below):

- a callback based API for "subscribing" to parts of the database
- a client-side cache

- a server-side push enabled websockets based transport layer
- fallback to Ajax if needed
- the ability to do database transactions, meaning atomic multi-path updates

The Real Time database and Observables

The notion of subscribing to parts of the database is really convenient, as is very well modeled by using RxJS Observables. Promises are not a natural API for a real-time database, because we need an asynchronous primitive that allows you to handle multiple values over time, and a promise only returns one value.

And that is where the AngularFire Library comes in, notice that if you include AngularFire in your app you also get the Firebase SDK as well and you can use it directly, more on that in a moment.

Section 4 - The AngularFire Library

Let's start by adding AngularFire to our application:

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

We then configure AngularFire in our application module:

```
@NgModule({
  declarations: [
    AppComponent,
    ...
  ],
  imports: [
      BrowserModule,
      AngularFireModule.initializeApp(firebaseConfig),
      AngularFireDatabaseModule,
      AngularFireAuthModule,
      RouterModule.forRoot(routerConfig)
],
  providers: [...],
  bootstrap: [AppComponent]
})
export class AppModule { }

```

With this simple configuration, we have all the AngularFire injectables available anywhere in our application. We can for example inject the main Firebase SDK app, and use it directly:

```
import {FirebaseApp} from "angularfire2";

@Injectable()
export class LessonsService {

    constructor( @Inject(FirebaseApp) fb) {

        const rootDbRef = fb.database().ref();

        rootDebRef.on('value', snapshot => console.log(snapshot.val()));

        ...
    }

}

```

This is the equivalent of the example we have seen before, it reads the whole database, don't do this! But it's a good Hello World do to if you are just getting started.

Let's say that instead of reading the whole database you want to read the list of courses, or a given lesson. The code would look like this:

```
constructor( @Inject(FirebaseApp) fb) {

        const rootDbRef = fb.database().ref();

        rootDebRef.child('courses').on('value',
            snap => console.log('Received the whole courses node',snap.val())

        rootDebRef.child('lessons/-KT_udWS6pEmpLVrxlVw:').on('value',
            snap => console.log('Received lesson with a given Id',snap.val())

}

```

But notice that here we are still using a callback interface, and not an Observables-based API. This is because we are here in fact using the API of the Firebase SDK directly.

AngularFire complements this with two observable binding APIs that allow you to subscribe to any part of the Firebase database:

- You can subscribe to a whole list
- Or you can subscribe to a single object instead

Let's now see the AngularFire Observable API in action, and see how it really is a perfect client-side complement for the Realtime Database.

Accessing the Real Time Database via AngularFire

Let's say that we want to subscribe to the list of courses, to know when a new course is available. We have seen before how to do this using the

Firebase SDK directly, let's now use AngularFire instead:

```
import {Injectable} from '@angular/core';
import {Observable, Subject} from "rxjs/Rx";
import {Lesson} from "./lesson";
import {AngularFireDatabase} from "angularfire2";


@Injectable()
export class LessonsService {

    constructor(private db:AngularFireDatabase) {
    }

    findAllLessons():Observable<Lesson[]> {
        return this.db.list('lessons')
            .do(console.log)
            .map(Lesson.fromJsonList);
    }

}

```

In this example, we have injected the AngularFireDatabase injectable, which is the service that we use to interact with the real-time database.

Querying a List in AngularFire

So how do we use the AngularFire database to read a list of lessons? The database object has two main API methods for querying data: `list` and `object`.

In the service method `findAllLessons()` above we are using `list` to query the whole `lessons` child node, which is immediately under the Firebase database root node.

The return value of the `list` call is an RxJs Observable, whose emitted values are the content of the lessons list:

- the first value emitted will be the current value of the list
- if no one changes any lesson on the lessons list, no new value will get emitted
- if someone changes any lesson on the list, a second value will be emitted with a whole new list

The key thing to bear in mind with the Observable returned by AngularFire is that it does not complete after the first value (although we could call `first()` on it to get that behavior). This is unlike for example the Observables returned by the Angular HTTP Library.

The Observable remains "running" (it does not complete) so we get new values over time, which is actually the most natural behavior of an Observable.

The `list` call can be configured with a query configuration object do things like pagination, search based on a property, etc.

Querying Objects in AngularFire

The same way that we can query lists, we can also query a specific object using AngularFire via the `object` API method. Here is an example:

```
findLessonById(lessonId:string):Observable<Lesson> {
    return this.db.object(`lessons/${lessonId}`)
    .map(Lesson.fromJson);
}

```

This query would return us an Observable whose values over time are the different versions of a lesson with a given Id.

The two main AngularFire APIs are super convenient for handling parts of the Real-Time Database as Observables. But we also can modify data with AngularFire.

Modifying Lists and Objects in AngularFire

Let's say that we have courses observable whose values are the list of all courses. The courses observable is called `courses$`, so the variable ends in the dollar sign to indicate that this is observable.

We could write a new course to the end of this list in the following way:

```
courses$.push({description: 'New Course'})
  .then(
        () => console.log('item added'),
        console.error
   );

```

Notice that the call to push does not return an Observable, we get back a promise-like object on which we can call `then()`.

The capabilities of writing to the database using Angular Fire are available, but they are by design only a subset of what we can do with the SDK. We should not hesitate to inject the SDK app and use it directly while using AngularFire, and use the full power of the SDK.

> *But don't think that you have to use the Firebase SDK to be able to use the Realtime database at all!*

This is true, if you want you don't even need the SDK, let's go over why.

Firebase and REST

If you don't like the idea of using the Firebase client to cache data, or if you don't need the real-time capabilities of the database, you can simply use Ajax calls with Firebase as well. You don't need the SDK at all, just your favorite REST client.

It turns out that everything in Firebase has an URL, starting at the root URL. And that URL can be used to perform data retrieval and modification operations:

- an HTTP GET will read all the data under a given node
- a PUT will replace all existing data with new data
- a PATCH can be used to change only some properties of the data
- a DELETE deletes the data as expected
- a POST adds an entry to a list

It's really that simple and you didn't have to write a single custom REST endpoint! There is one catch though, you need to add `.json` at the end of the URL.

For example, let's say that the root URL of your database is:

```
https://final-project-recording.firebaseio.com
```

And you want to read all courses under the first-level node `courses`. If you do a GET against the following Url, you would get back JSON containing all the courses:

```
https://final-project-
recording.firebaseio.com/courses.json
```

And here is the resulting data:

```
{
  "-KT_udWMM0vKlnp-naE3": {
    "courseListIcon": "https://angular-academy.s3.amazonaws.com/main-logo/m
    "description": "Angular 2 Tutorial For Beginners",
    "iconUrl": "https://angular-academy.s3.amazonaws.com/thumbnails/angular
    "longDescription": "Establish a solid layer of fundamentals, learn what
    "url": "getting-started-with-angular2"
  },
  "-KT_udWj0knDpxHaxcKv": {
    "courseListIcon": "https://angular-academy.s3.amazonaws.com/course-logo
    "description": "Angular 2 HTTP and Services",
    "iconUrl": "https://angular-academy.s3.amazonaws.com/thumbnails/service
    "longDescription": "<p class='course-description'>Build Services using
    "url": "angular2-http"
  }
}
```

So you get all this out of the box REST functionality without writing a single line of code!

But you will probably prefer to use the SDK: it has caching built-in and you can do atomic multi-path updates, unlike using the REST API.

But if you want to simply do CRUD, it works perfectly and its very convenient.

Section 5 - Firebase Authentication

Firebase is much more than just the Real Time Database. One of its numerous features is the ability to authenticate users out of the box, via either email and password or via external providers like Github.

Authentication is the kind of logic that we don't want to build ourselves: it's both very easy to get wrong and business critical at the same time, and it's exactly the same functionality in every single application.

How many `Users` and `Roles` SQL Database tables have you seen in the projects that you worked on? It's the kind of thing that can be instantly working out of the box from day one if we use a BaaS solution, but instead we often rewrite it from scratch in each project.

Example of using Firebase Authentication

If we are using both the Firebase SDK or AngularFire, we can have authentication working with a couple of API calls. Let's say that we want to authenticate a user via email and password, here is what it looks like using the Firebase SDK:

```
import {firebaseConfig} from "./src/environments/firebase.config";
import {initializeApp, auth,database} from 'firebase';

initializeApp(firebaseConfig);

auth()
    .signInWithEmailAndPassword('admin@angular-university.io', 'test123')
    .then(onLoginSuccess)
    .catch(onLoginError);

function onLoginSuccess() {
    ...
}

function onLoginError() {
    ...
}

```

Authentication is only one of the many problems that Firebase can solve for us out of the box. Other functionality that we need to implement an application includes Cloud Messaging, Hosting, Storage, a Test Lab, Crash Reporting and much more.

Firebase Storage, Hosting and HTTP/2

If you want to build your application as a single page app, chances are if you are not using bundle splitting that your frontend is really only 3 static files:

- the index.html, which is literally the single html page of your app, and is mostly empty
- the CSS bundle
- The Javascript bundle

And that's it, 3 static files! One of the advantages of single page apps that does not get mentioned a lot is how easy it is to deploy them in production: it's just a few static files, upload them to Amazon S3 or your nginx / apache server and it's done! At least the frontend part.

What better place to upload those files than to the Firebase servers themselves, so that you don't have to do a separate DNS lookup to get those files from somewhere else? There is an SSL certificate provisioned so it's secure out of the box.

Not to mention that you get full HTTP 2 support if using Firebase Hosting for that. You also want to upload all your images to Firebase Hosting as well. You want to benefit from that blazing performance of having only one TCP/IP connection to get your app `index.html`, CSS and Javascript bundles as well as your images, all in one single DNS request.

And if you don't have HTTP 2 support, Firebase Hosting is still a pretty convenient solution just like the rest of Firebase as its really simple to upload files via the command line, have a look at this Firecast on Hosting to see it in action.

Section 6 - Conclusions & Bonus Content

Summary

To sum it up these are the main reasons why Firebase might have a considerable impact on the current state of web development:

- it gives us out of the box a JSON database which is exactly what we need to reduce the mismatch against SQL databases
- if using Angular, we have a very convenient way of interact with Firebase by using AngularFire
- its a lot easier to create custom backends using Firebase Queue and the Firebase SDK itself which works the same both on the client and on the server
- it solves a bunch of problems that we don't have to hand code from scratch each time: authentication is just an example

- it gives us a full stack CRUD solution for the CRUD parts of our app
- a lot of the knowledge gained while building other systems still applies

I'm convinced that Firebase is just a better way of building Web Applications: it's easier to reason about, it's fast and provides a great developer experience. It allows us to focus on our app and getting the app out the door to our users and overall saves us a ton of work and is just a joy to work with.

I hope this helps get the most out of Firebase and AngularFire! If you have any questions about the book, any issues or comments I would love to hear from you at admin@angular-university.io

I invite you to have a look at the bonus material below, I hope that you enjoyed this book and I will talk to you soon.

Kind Regards,
Vasco
Angular University

Typescript - A Video List

In this section, we are going to present a series of videos that cover some very commonly used Typescript features.

Click Here to View The Typescript Video List

These are the videos available on this list:

- Video 1 – Top 4 Advantages of Typescript – Why Typescript?
- Video 2 – ES6 / Typescript let vs const vs var When To Use Each? Const and Immutability
- Video 3 – Learn ES6 Object Destructuring (in Typescript), Shorthand Object Creation and How They Are Related
- Video 4 – Debugging Typescript in the Browser and a Node Server – Step By Step Instructions
- Video 5 – Build Type Safe Programs Without Classes Using Typescript
- Video 6 – The Typescript Any Type – How Does It Really Work?
- Video 7 – Typescript @types – Installing Type Definitions For 3rd Party Libraries
- Video 8 – Typescript Non-Nullable Types – Avoiding null and undefined Bugs

- Video 9 - Typescript Union and Intersection Types– Interface vs Type Aliases
- Video 10 - Typescript Tuple Types and Arrays Strong Typing