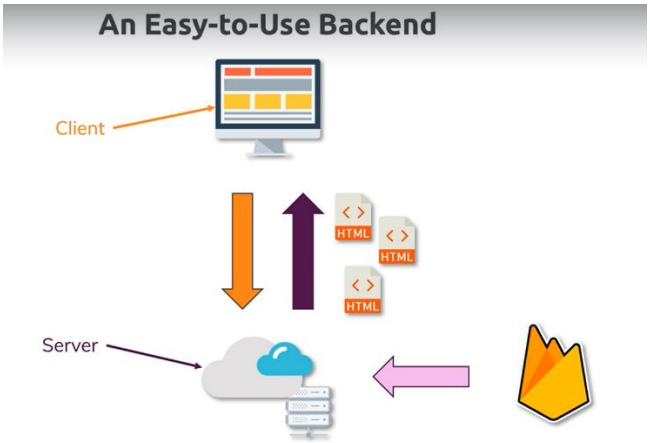
Angular, Firebase and Angularfire2

Firebase ve Angularfire2 kullanarak uygulamamızın backend kısmını halledeceğiz.

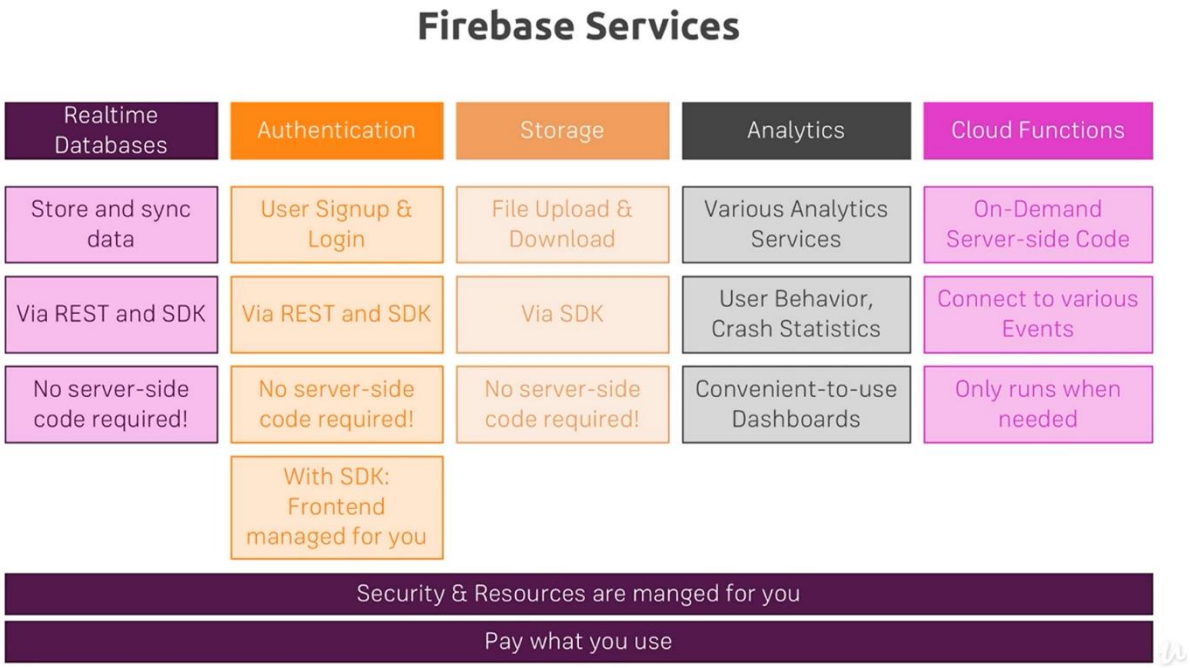
What is Firebase

Firebase temelde easy-to-use bir backend olarak düşünülebilir.



Firebase angular app'i host etmek için, hosting provider olarak kullanılabileceği gibi store data, fetch data run calculations ve user authentication gibi işlemler için de backendde kullanılabilir.

Farkı firebase servislerinden söz edilebilir. Aşağıda bu servisler yer alıyor:



Realtime Databases: Örneğin realtime database servisi ile angular uygulamamızda herhangi bir rest API request göndermeden database üzerindeki değişiklikleri anlık olarak uygulamamızda görebiliriz.

Authentication: Bu servis ile ıuser signup and login işlemleri kolaylıkla yapılabilir.

Storage: File upload donwload için kullanılan servistir.

Ayrıca **Analytics, Cloud Functions** servislerinden de söz edilebilir.

Bu servislerin güzel yanı şu server işlerini firebase bizim için hallediyor.

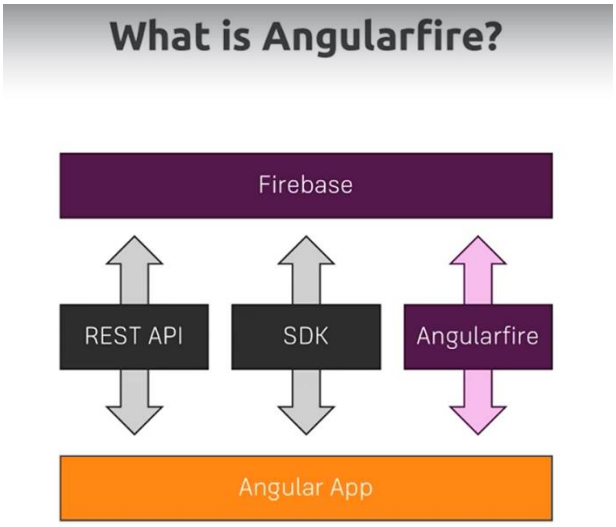
Biz bu kısımda sadece **Realtime Database** ve **Authentication** servislerine odaklanacağız.

Bu servisleri angular app'imiz içinde kullanmak için **Angularfire2** kullanacağız, ve ne kadar kolay şekilde işlerimizi halledeceğimizi göreceğiz.

What is Angularfire?

Eğer uygulamamızda firebase kullanıyorsak, angularfire: angular-firebase arasında bir köprü gibi davranır.

Angularfire bağımsız developer'lar tarafından geliştirilmiş bir şey yani alternatifi var, örneğin REST APIs ile de angular app içerisinden firebase'e ulaşabiliriz. Benzer şekilde official firebase sdk ile de herhangi bir javascript projesinde firebase'e ulaşabiliriz, sonuçta angular'dan da kullanabiliriz.



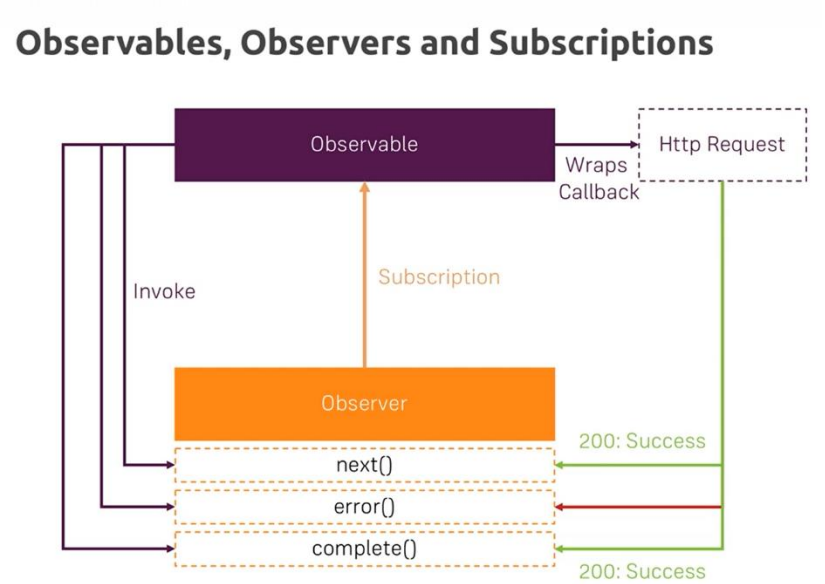
Angularfire'ı kullanarak sdk'in sağladığı tüm features'ı yine kullanabiliyoruz, bunun yanında observables açısından da angularfire'ın avantajı var. Bunun dezavantajı şu sadece angular app için kullanılabilir.

Buna karşın SDK'i herhangi bir javascript uygulamasından kullanabiliyoruz anladığım kadarıyla.

Burada Angular fire'ın observable-based data streams'e izin vermesi önemli ancak bunu kavramak için önce observables ve subscription kavramlarına bir bakacağız.

RxJS Observables Refresher

Bu modülde angularfire kullanılacak ve angularfire observables kullanacak. Bu yüzden bazı kavramları daha iyi anlamaya çalışalm.



Observable ve observer subscription ile birbirine bağlıdır. Kursumuzda da subject kullanmıştık ki bu da özel bir çeşit observable'dır. Subject'in next methodunu bazı value'ları emit etmek için kullanmıştık. Ayrıca subscribe methodu ile bu valuelara abone olmuştuk.

Subject'ten farklı olarak observable'ı next methodu olmayan, sadece subscribe edebileceğimiz bir yapı olarak düşünebiliriz. Ayrıca başka kaynaklara bağlı olarak event emit eden yapılardır.

Örneğin bir buton event'ini observable ile wrap edebiliriz, böylece butona ne zaman basılırsa event'i subscribe edilen yere ulaştırabiliriz.

Temelde her observer'ın trigger edilebilecek 3 methodu vardı. Bu methodlar observable'ın wrap ettiği source tarafından veya subject'te olduğu gibi next() methodu ile programatically trigger edilebilir.

Bu methodlar next(), error() ve completer() methodlarıdır. Çünkü observables yeni bir value emit edebileceği gibi, fail edebilir veya emit edilecek value'ları bitirebilir ve daha fazla emit olmayacağını belirtebilir.

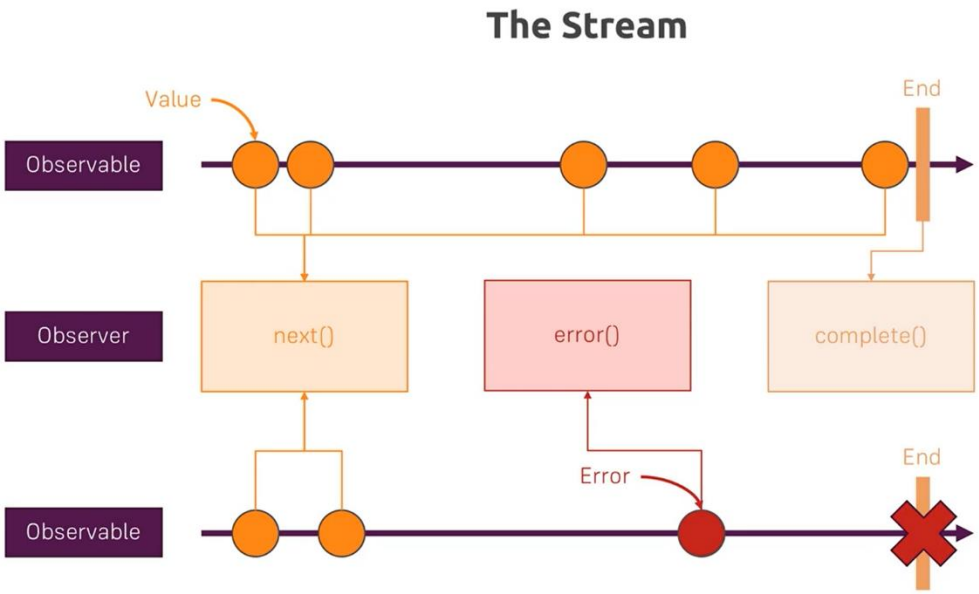
Sonuçta bu methodlar trigger edilerek çalışır yani invoke edilir.

Buton örneğine benzer şekilde örneğin bir observable bir http request'i wrap edebilir, eğer succeed ederse observable'ın next() ve complete() methodları çalışır, response emit edilir ve http request tamamlandığında başka bir değer beklenmediği için complete de çalışır. Ancak buton örneğinde elbette gelecekte click event'i alınabileceği için complete çalışmayacaktır.

Benzer şekilde http request başarılı olmazsa error ve complete methodları ile observable cancel edilebilir.

Bu observables'ı streams of data olarak düşünebiliriz. Sonuçta olay asenkron işleri gerçekleştirmek.

Örneğin http örneğine düşünelim, bir request gönderdik ve response alındı bu bizim value'muz olsun. Ya da butona tıklandı yeni bir value'muz var bu durumda event data'sını içeren bir object.

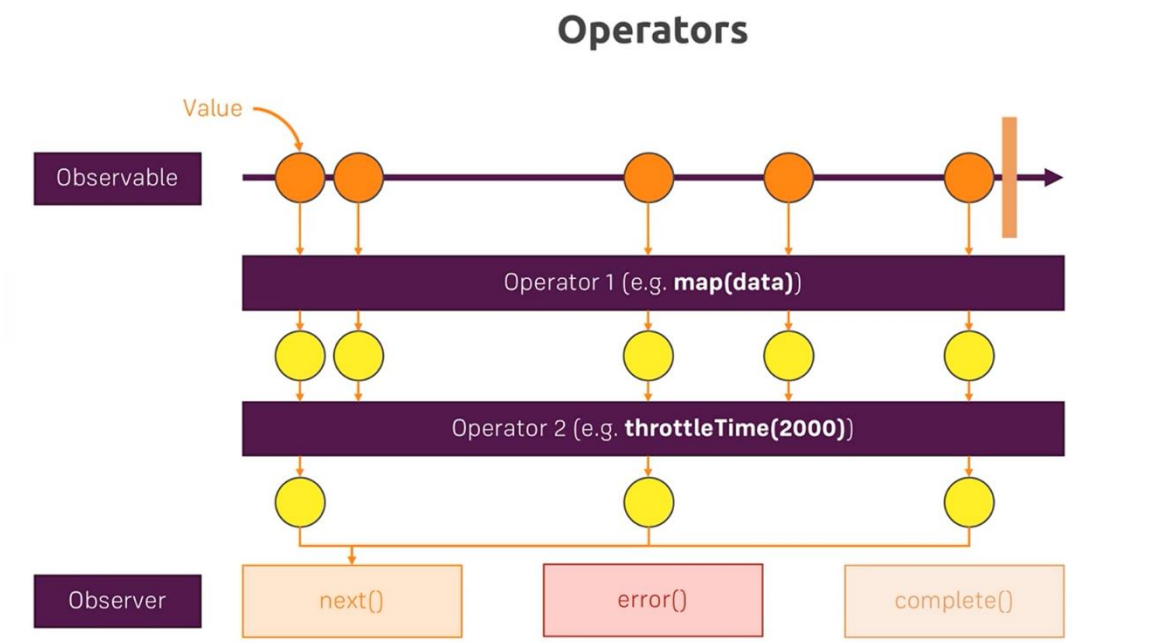


Observable value'yu başarılı şekilde elde ederse observer'ın next() methodu trigger ediliyor, örneğin buton click event'ini observable ile wrap ettiysek yukarıdaki gibi farklı anlarda farklı value'lar alınabilir ve hepsi next() methodunu trigger edecektir. Eğer observable tamamlanırsa observer'da complete methodu çalışacaktır.

Örneğin bir başka observable için 2 succesful data'dan sonra bir error alınabilir.

Observer temelde observable'a subscription ile bağlıdır ve values, errors ve completion durumlarında observable tarafından bilgilendirilir.

Observables'ın en iyi yanı observe edilen value'ların observer'da elde edilmeden önce üzerinde operasyonlar yapılabilmesi. Yani data observar'da elde edilmeden önce çeşitli transformasyonlara uğrayabiliyor hatta bu operasyonları sayesinde sonuçta observar'a en az 2 saniyede bir değer gelsin diyerek, işleri düzene sokabilirim.



Sonuçta value observable ile emit edildikten sonra fakat observer ile alınmadan önce bazı operasyonlar data üzerine uygulanabiliyor.

Bunları uygulamayla da göreceğiz ve her şey daha açık olacak.

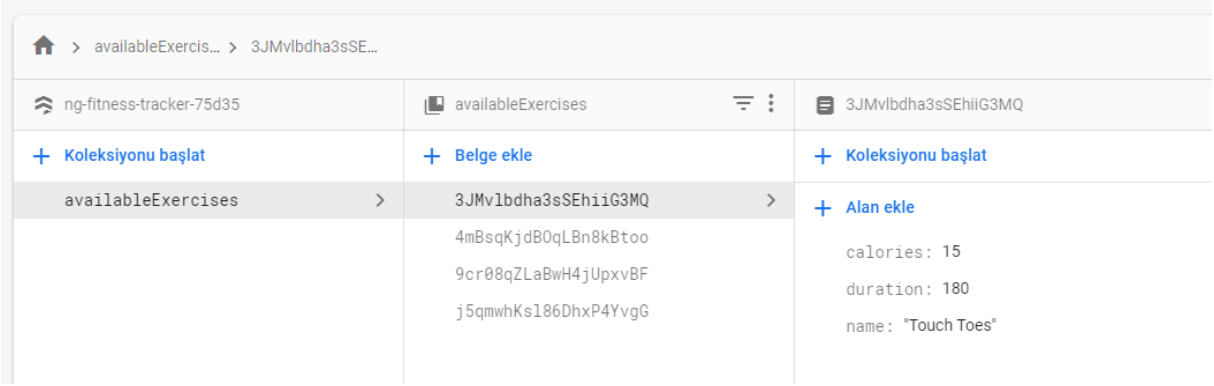
Diving Into Firebase

Bu kısımda temelde basit bir available exercises database'i oluşturduk. Bunun için cloud firestore kullandık.

Burada 3 temel yapı var: **Collections > Documents > Data**

Collections folder gibi düşünülebilir bunun içinde documents yer alır, her documents'ı bir Json objesi gibi düşünebiliriz, yani her documnets içerisinde data tutulur.

Örneğin available exercises için bir AvailableExercises collections'ı içinde her exercise için ayrı bir document kaydı olacak, her document kaydı içerisinde ise name, duration, calories gibi bilgiler tutulacak:



Ayrıca bir documnet içerisinde bir başka koleksiyon oluşturulabilir böyle dallanmalar mümkün.

Şimdi angular app'i bu database ile nasıl bağlayabileceğimize bakacağız. Bunu angularfire ile yapacağız hadi bu adıma geçelim.

Angularfire ile Firebase database'ine ulaşalım:

Öncelikle uygulamamıza angularfire2'yi kuracağız. Bunun için angularfire'ın github sayfasına bakabiliriz.

npm install angularfire2 firebase --save

Bu yüklemeyi yaptıktan sonra, angular app içerisinde src/environment/environment.ts dosyasına gidip içerisine, firebase'den elde edilen js object'i yerleştiriyoruz, bu object içerisinde apiKey, authDomain vesaire var, bu object sayesinde uygulamamız firebase ile iletişim kurabilecek.

```
export const environment = {
  production: false,
  firebase: {
    apiKey: "AIzaSyDaYAA3pdqlmbTZ8DiyzZ_xkF_H488kz1A",
    authDomain: "ng-fitness-tracker-75d35.firebaseapp.com",
    databaseURL: "https://ng-fitness-tracker-75d35.firebaseio.com",
    projectId: "ng-fitness-tracker-75d35",
    storageBucket: "ng-fitness-tracker-75d35.appspot.com",
    messagingSenderId: "332033698308",
    appId: "1:332033698308:web:0328da6258df4c56406351",
    measurementId: "G-23DTY8BR9K"
  }
};
```

Bu object'i firebase console'umuzdan projeye genel bakış dedikten sonra proje üzerine gelerek veya henüz yaratılmamışsa add firebase to your web app diyerek ulaşabiliriz.

Şimdi angularfire'ı kullanabilmek için app.module.ts içerisinde bazı import işlemleri yapılacak:

```
import { AngularFireModule } from '@angular/fire'
import { environment } from '../environments/environment';
```

```
  imports: [
    AngularFireModule.initializeApp(environment.firebase),
  ],
```

Artık angularfire'ı uygulamamızda kullanarak firebase işlemlerini yapabiliriz.

Şimdi angularfire'ı kullanmaya hazırız ancak, angularfire'ın kullanabileceğimizi farklı modülleri var, hangisini kullanmak istiyorsak ona göre ekstra bazı importlar yapıp, kullanıma geçebiliriz.

Örneğin biz firebase cloud-firestore ile datamızı sakladığımız için ve bu verilere ulaşmak istediğimiz için **AngularFireStoreModule**'ü impor edelim. Yine app.module.ts içerisine aşağıdaki gibi import'u gerçekleştiriyoruz.

```
import { AngularFirestoreModule } from '@angular/fire/firestore'
```

```
  imports: [
    AngularFirestoreModule
  ],
```

Yani biraz önceki import ile general firebase setup'ını yaptık, şimdi ise özellikle firestore ile ilgili fonksiyonları kullanabilmek için bu import'u yaptık.

Şimdi firestore üzerindeki database'imize new-training-component.ts üzerinden ulaşmak istiyoruz, daha önceden bu component içerisinde egzersizler aşağıdaki gibi componente inject edilen servis kullanılarak elde ediliyordu, ve daha sonra view içerisinde bu egzersizler kullanıcıya gösteriliyordu.

```
  ngOnInit() {
   this.exercises = this.trainingService.getAvailavleExercises();
  }
```

Şimdi ise biz burada servisi kullanmadan doğrudan component içerisinde firestore'a bağlantı sağlyalım:

Öncelikle component içerisinde AngularFirestore'u import edelim.

```
import { AngularFirestore } from 'angularfire2/firestore'
```

Anladığım kadarıyla AngularFirestore class'ı angularfire'ın bize sağladığı bir servis ve aynı başka bir servis gibi bu servisi component'e inject edip kullanacağız.

```
constructor(private trainingService: TrainingService, private db: AngularFires
tore ) { }
```

Artık db servisini kullanarak arkaplanda doğrudan firestore'a ulaşabileceğiz.

Injection'ı da halletiğimize göre artık bu servis üzerinden firestore üzerindeki database'imize ulaşabiliriz, bunun için componentin ngOnInıt methodu içerisinde aşağıdaki gibi bir işlem yapacağım:

```
  ngOnInit() {
    this.db.collection('availableExercises').valueChanges().subscribe( result
=> {
      console.log(result);
    } )
```

Tanımlanan db servisi üzerinden **collection()** methodu ile firestore üzerindeki istediğimiz collection'a ulaşabiliriz. Bizim ilgilendiğimiz collection availableExercises collection'ı olacak.

Daha sonra çağırılan **valueChanges()** methodu bir observable return edecektir, bu observable availableExercises koleksiyonu üzerinde bir değişiklil olduğunda tüm koleksiyonu emit eden bir observabledır.

O halde bu observable üzerinden **subscribe()** methodu da kullanılabilir ve böylece collection her değiştiğinde emit edilen collection value'su burada yakalanır. Daha sonra da bu yakalanan değer üzerinde subscribe methodu içerisindeki anonymous function çalışır yani bu değer konsolda yazdırılır.

Bu observable'ın program ilk çalıştığında da bir değişim algıladığını unutma yani ilk çalışmada da ilgili collection konsolda yazdırılacak.

```
▼ (4) [{…}, {…}, {…}, {…}]
  ▶ 0: {calories: 15, duration: 180, name: "Touch Toes"}
  ▶ 1: {name: "Burpees", duration: 60, calories: 8}
  ▶ 2: {calories: 8, duration: 30, name: "Crunches"}
  ▶ 3: {duration: 120, calories: 18, name: "Side Lunges"}
    length: 4
  ▶ __proto__: Array(0)
```

Görüldüğü gibi component yaratılınca yukarıdaki gibi collection'ı subscribe etmiş olduk. Ki component ne zaman index'e çağırılırsa o zaman yaratılmış oluyor, yani training sekmesine geçmeden önce component yaratılmıyor o yüzden de yukarıdaki çıktıyı alamıyoruz.

Logout deyip tekrar login dersek ve training'e gidersek component tekrar yaratılacak o yüzden ngOnInıt tekrar çalışacaktır.

Bunun dışında valueChanges() ile collection üzerinde değişen değer olunca next() methodu trigger edilen bir observable yaratmış olduk. Yani databse üzerinde bir değişiklik yaparsak collection'ın tümü tekrardan emit edilecek ve ngOnInit içinde de subscribe edilecek.

Emit ve subscribe edilen collection içerisindeki id'lerin tablodaki id'lerle aynı olmadığına dikkat et, bunun sebebi valueChanges() listener'ının çok basit bir listener olması ve id gibi bazı metadata'ları atarak documents içerideki value'ları verecek bir observable yaratması.

Şimdi ise bir başka yaklaşım yapalım ve subscription işlemini component içinde değil de component'in view'i içerisinde yapalım.

Bunun için valueChanges() ile return edilen observable'ı component içerisinde bir exercises değişkeni içine kaydedelim, bunun için öncelikle observable class'ını import ediyoruz:

```
import { Observable } from 'rxjs';
```

Daha sonra component içinde exercises adında bir observable tanımlıyorum, observable generic bir class, şuanda observable içerisine any tipini veriyoruz, aslında Exercise tipimiz var ama, database'den elde edilen değerler tam olarak exercise tipine uymuyor çünkü id'si yok.

```
export class NewTrainingComponent implements OnInit {

  exercises: Observable<any>;
```

Şimdi ise db servisi üzerinde az önceki gibi ilgili collection için valueChanges() methodu ile değişiklik olduğunda emit işlemi yapan bir observable'ı component'in exercises değişkeni içerisine atıyoruz.

```
  ngOnInit() {
    this.exercises = this.db.collection('availableExercises').valueChanges();
  }
```

Şimdi new-training'in view'i içerisine bakalım:

```
<mat-option *ngFor="let exercise of exercises" [value]="exercise.id">
  {{ exercise.name }}
</mat-option>
```

Burada daha önce exercises listesi içerisinden tek tek her exercise objesi için mat-option etiketinin yukarıdaki gibi yaratıldığını anlıyoruz. Ancak şuanda exercises bir liste değil bir observable bu yüzden ngFor'u bu şekilde kullanamayız.

O yüzden async pipe'ını kullanacağız ve otomatik olarak exercises observable'ı burada subscribe edilecek ve observable'ın return ettiği collection return edilecek. Sonuçta aşağıdaki gibi bir değişiklikle:

```
<mat-
option *ngFor="let exercise of exercises | async" [value]="exercise.name">
  {{ exercise.name }}
</mat-option>
```

Artık hem component yaratılınca hem de database her değiştiğinde egzersizler collection'ı subscribe/fetch edilecek:

```
▼ (4) [{…}, {…}, {…}, {…}]
  ▶ 0: {calories: 15, duration: 180, name: "Touch Toes"}
  ▶ 1: {name: "Burpees", duration: 60, calories: 8}
  ▶ 2: {calories: 8, duration: 30, name: "Crunches"}
  ▶ 3: {duration: 120, calories: 18, name: "Side Lunges"}
    length: 4
  ▶ __proto__: Array(0)
```

Ve fetch edilen collection içerisindeki her bir document elemanı bir exercise olarak alınacak ve ngFor bu şekilde kullanılacaktır.

Bu şekilde database her update edildiğinde uygulama refresh edilmeden, egzersiz listesinin de observable'ı subscribe etmesi sayesinde update edileceğini unutma!

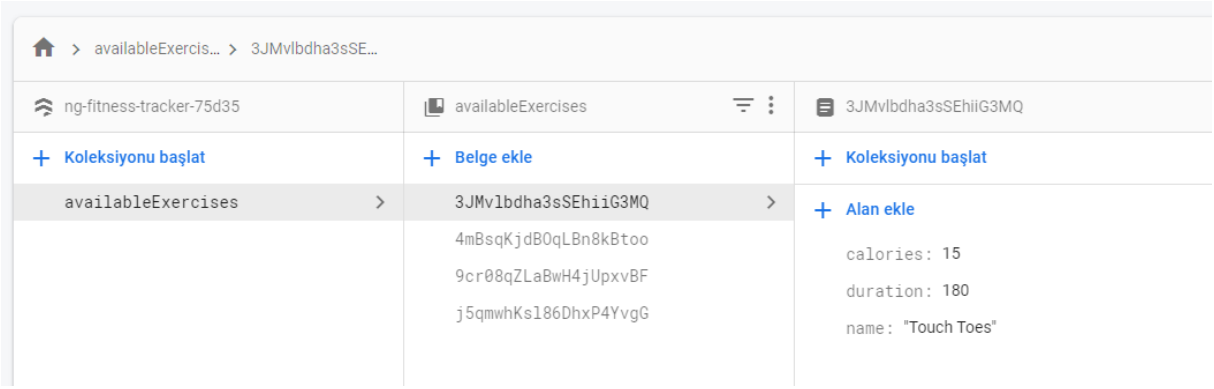
Listening to Snapshot changes

Önceki kısımda firestore üzerinde oluşturulan database'e observable yardımı ile ulaşmıştık, valueChanges() methodu ile database üzerindeki her değişiklikte collection'ı subscribe ederek uygulama içerisinde kullanıma sunmuştuk.

Bu amaç için component'in ngOnInit methodu içerisinde aşağıdaki gibi, valueChanges() methodu ile observable'ı elde etmiştik.

```
ngOnInit() {
  this.exercises = this.db.collection('availableExercises').valueChanges();
}
```

Ancak bu method ile subscribe ettiğimiz observable'ın emit ettiği value'lar sadece data kısmıydı, yani aşağıdaki database'in sadece en sağda kalan kısımları gönderiliyordu.




```
(4) [{…}, {…}, {…}, {…}]
  0: {calories: 15, duration: 180, name: "Touch Toes"}
  1: {name: "Burpees", duration: 60, calories: 8}
  2: {calories: 8, duration: 30, name: "Crunches"}
  3: {duration: 120, calories: 18, name: "Side Lunges"}
  length: 4
  __proto__: Array(0)
```

Halbuki biz documentID'leri de göndermek isteyebiliriz.

Bu amaçla aşağıdaki gibi snapshotChanges() event listener'ını kullanacağım. Bu valueChanges'dan farklı bir event listener'dır ve metadata'yı emit eder.

```
ngOnInit() {
  this.db.collection('availableExercises').snapshotChanges().subscribe(result
=> console.log(result));
}
```

Yukarıdaki gibi bir işlemle yine component oluştuğu zaman bir subscribe işlemi yapılır ancak burada subscribe edilen edilen değer önceki gibi değildir, ancak sonuçta bu subscription ile document ID değerine ulaşabilirim.

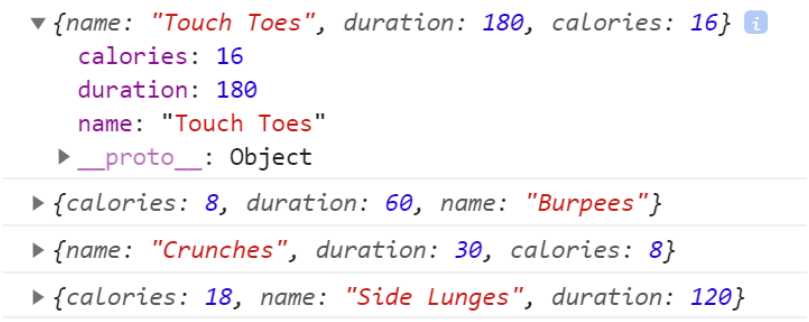
1. *(4) [{…}, {…}, {…}, {…}]*
    1. 0:
        1. payload:
            1. doc: n
                1. Cd: n {key: t, version: t, Xe: t, Ke: false, hasCommittedMutat
                   ions: false}
                2. Ef: null
                3. Fd: false
                4. Nd: false
                5. Tf: t {path: n}
                6. yd: t {od: n, hd: t, ad: FirebaseAppImpl, ud: t, INTERNAL: {…}
                   , …}
                7. exists: (...)
                8. id: "3JMvlbdha3sSEhiiG3MQ"
                9. metadata: (...)
                10. ref: (...)
                11. __proto__: t
            2. newIndex: 0
            3. oldIndex: -1
            4. type: "added"
            5. __proto__: Object
        2. type: "added"
        3. __proto__: Object
    2. 1: {type: "added", payload: {…}}
    3. 2: {type: "added", payload: {…}}
    4. 3: {type: "added", payload: {…}}
    5. length: 4
    6. __proto__: Array(0)

Hem document ID hem de data'ya ulaşmak istiyorsam ile farklı bir şey yapmam gerekecek:

Şimdi şunu yapalım, subscribe edilen değer yukarıda da görüldüğü gibi bir array of objects'tir. Bu yüzden bu result içerisinde loop dönebiliriz:

```
ngOnInit() {
  this.db.collection('availableExercises').snapshotChanges().subscribe(result
=> {
    for (const res of result){
      console.log(res.payload.doc.data());
    }
  });
}
```

Yani sonuçta biraz evvel elde edilen result listesi içerisindeki her bir object aslında database'deki bir document'a yani egzersize karşılık geliyordu, her bir document'in payload.doc.data()'sı bize ilgili egzersizin datasını sağladı:



Sonuçta bu yöntem ile hem documentID hem de data ulaşılabilir hale geldi!

Şimdi yapmak istediğimiz şey şu, database'den observable ile verileri çekerken araya bazı observable operators ekleyelim ve sonuçta hem documentID hem de data alınsın:

```
import { map } from 'rxjs/operators';
```

```
  ngOnInit() {
    this.db.collection('availableExercises')
    .snapshotChanges()
    .pipe(map(docArray => {
      return docArray.map(doc => {
        return {
          id: doc.payload.doc.id,
          ...doc.payload.doc.data()
        };
      });
    }))
    .subscribe(result => {
      console.log(result);
    });
  }
```

Yukarıdaki yöntem ile aşağıdaki gibi bir array subscribe edilmiş oldu her objesi bir document'e yani egzersize karşılık geliyor ve documentID'de buraya eklenmiş oldu!

```
1. (4) [{…}, {…}, {…}, {…}]
    1. 0: {id: "3JMvlbdha3sSEhiiG3MQ", calories: 16, duration: 180, name: "Touch Toes"}
    2. 1: {id: "4mBsqKjdBOqLBn8kBtoo", name: "Burpees", duration: 60, calories: 8}
    3. 2: {id: "9cr08qZLaBwH4jUpxvBF", calories: 8, duration: 30, name: "Crunches"}
    4. 3: {id: "j5qmwhKsl86DhxP4YvgG", duration: 120, name: "Side Lunges", calories: 18}
    5. length: 4
    6. __proto__: Array(0)
```

Şimdi ise daha önce yaptığımız gibi subscribe'u component içinde yapmayacağız bunun yerine snapshot ile elde edilen observable'ı doğrudan this.exercises observable'ına atayacağız, … yerine de ayrı ayrı tüm alanları yazdık:

```
  ngOnInit() {
    this.exercises = this.db.collection('availableExercises')
    .snapshotChanges()
    .pipe(map(docArray => {
      return docArray.map(doc => {
        return {
          id: doc.payload.doc.id,
          name: doc.payload.doc.data()['name'],
          duration: doc.payload.doc.data()['duration'],
          calories: doc.payload.doc.data()['calories']
        };
      });
    }));
  }
```

Ancak artık exercise observable'ına generic olarak any yerine Exercise[] verilebilir çünkü observable'dan subscribe edilen değer bir Exercise array oluyor!

```
  exercises: Observable<Exercise[]>;
```

Sonuçta view içerisinde de artık id kullanılabilir çünkü artık susbcribe edilen arrayimiz bir Exercise array'i ve her elemanı bir exercise olduğu için id'si de var:

```
<mat-option *ngFor="let exercise of exercises | async" [value]="exercise.id">
  {{ exercise.name }}
</mat-option>
```

Sonuç olarak yine databaseden egzersizler çekiliyor ancak bu kez document id de alındı!

Reconstructing the Code

Şuanda new-training component içerisinde aşağıdaki gibi bir method var, yeni bir training başlatınca çalışıyor, ve trainingService'in methodunu çağırıyor.

```
  onStartTraining(form: NgForm){
    this.trainingService.startExercise(form.value.exercise);
  }
```

Servis içerisinde de aşağıdaki methodla alınan egzersiz availableExercise içerisinde bulunup running exercise olarak atanıyordu.

```
    startExercise(selectedId: string){
        this.runningExercise = this.availableExercise.find(ex => ex.id === sel
ectedId);
        this.exerciseChanged.next({...this.runningExercise})
    }
```

Ancak şuanda form.value.exercise'ın bize verdiği değer firebase üzerindeki her bir available egzersizin documentID'si oluyor, yani servis içerisinde bu ID ile uyuşan bir availableExercise bulunamayacak çünkü artık source'lar farklı.

```
        <mat-
select placeholder="Select a workout" ngModel name="exercise" required>
          <mat-
option *ngFor="let exercise of exercises | async" [value]="exercise.id">
            {{ exercise.name }}
          </mat-option>
```

Dediğim gibi servis içerisindeki available exercises aşağıdaki gibi tanımlanmış ancak ben bu veriyi servis içerisinde tanımlamak istemiyorum, firebase'den çekmek istiyorum.

```
    private availableExercise: Exercise[] = [
        { id: 'crunches', name: 'Crunches', duration: 30, calories: 8 },
        { id: 'touch-toes', name: 'Touch Toes', duration: 180, calories: 15 },
        { id: 'side-
lunges', name: 'Side Lunges', duration: 120, calories: 18 },
        { id: 'burpees', name: 'Burpees', duration: 60, calories: 8 }
    ];
```

Yapmak istediğimiz şey availableExercise arrayini yine servis içerisinde tutmak ancak veriyi burada tanımlamak yerine firestore'dan çekeceğiz:

```
    private availableExercise: Exercise[] = [
    ];
```

Servis içerisindeki getAvailableExercises methodu yerine:

```
    fetchAvailableExercises(){
        return this.availableExercise.slice();
    }
```

Önceki başlıkta new-training-component'in ngOnInıt'inde yapılan snapshotChanges ile database'den veri çekme işi aşağıdaki gibiydi.

```
  ngOnInit() {
    this.exercises = this.db.collection('availableExercises')
    .snapshotChanges()
    .pipe(map(docArray => {
      return docArray.map(doc => {
        return {
          id: doc.payload.doc.id,
          name: doc.payload.doc.data()['name'],
          duration: doc.payload.doc.data()['duration'],
          calories: doc.payload.doc.data()['calories']
        };
      });
    }));
  }
```

Şimdi bu işlemi component yerine servis içine taşıyalım:

```
    fetchAvailableExercises(){
        this.db.collection('availableExercises')
            .snapshotChanges()
            .pipe(map(docArray => {
                return docArray.map(doc => {
                    return {
                    id: doc.payload.doc.id,
                    name: doc.payload.doc.data()['name'],
                    duration: doc.payload.doc.data()['duration'],
                    calories: doc.payload.doc.data()['calories']
                    };
                });
            }));
    }
```

Ancak training servis içine db ismindeki AngularFireStore servisi inject edilmediği için şuanda db burada undefined olacak.

Bu amaçla training servisimize AngularFireStore servisini inject etmeliyiz, bunun için servis class'ı üstünde injectbale decorator'ını import edip kullanmalıyız:

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

@Injectable()
export class TrainingService {
```

Daha sonra da TrainingService içine consturctor ekleyip istediğimiz servisi burada inject edebiliriz:

```
constructor(private db: AngularFirestore){}
```

Elbette bunun için gerekli importu da TrainingService içerisinde yapmalıyız, ayrıca map kullandığımız için onu da import edelim.

```
import { AngularFirestore } from 'angularfire2/firestore';
import { map } from 'rxjs/operators';
```

Artık fetchAvailableExercises methodumuz çalışacaktır, ancak önceden subscribe işlemini component'in view'inde yapıyorduk burada firebase'deki collection'ı çekip subscribe edelim bize gelen bir Exercise[] olacaktır, bu arrayi de servisin availableExercises arrayine atayalım.

```
    fetchAvailableExercises(){
        this.db.collection('availableExercises')
            .snapshotChanges()
            .pipe(map(docArray => {
                return docArray.map(doc => {
                    return {
                    id: doc.payload.doc.id,
                    name: doc.payload.doc.data()['name'],
                    duration: doc.payload.doc.data()['duration'],
                    calories: doc.payload.doc.data()['calories']
                    };
                });
            })).subscribe((exercises: Exercise[]) => {
                this.availableExercises = exercises;
            });
    }
```

Artık servisin availableExercises property'si firebase ile bağlanmış oldu!

Şimdi ise sırada, component içerisinde bu egzersiz listesini servisten çekmek var:

Bunun için ilk akla gelen yöntem component içerisinde servisi inject edip, servisin availableExercises listesini component içine çekmek, yalnız unutma bunu yaparsak, ilk başta tüm liste gelecek ancak listedeki bir değişiklik component'e yansımayacak bu yüzden, yapmamız gereken şey, bir Subject observable'ı ile servis içerisinden value emit etmek.

Firestore'dan her yeni subscribe işlemi yapıldığında, TrainingServis'ten component'e de exercises listesi emit edilsin istiyoruz, işte bu yüzden:

Öncelikle servis içerisinde yeni bir Subject yaratıyorum, emit etmek istediğimiz Exercise array olduğu için bunu belirtiyorum:

```
exercisesChanged = new Subject<Exercise[]>();
```

Şimdi firestore'dan her subscription yapıldığında ben de Subject'imi kullanarak servisin availableExercise listesini emit etmek istiyorum:

Bunun için fetchAvailableExercises methodunun içinde subscription'dan hemen sonra aşağıdaki gibi emit işlemini yapıyoruz, elbette availableExercises pointer olduğu için listenin kopyasını gönderiyoruz:

```
            })).subscribe((exercises: Exercise[]) => {
                this.availableExercises = exercises;
                this.exercisesChanged.next([...this.availableExercises]);
```

Şimdi de component içerisinde bu subject'e subscribe olup emit edilen verileri yakalayalım, component yaratıldıktan hemen sonra önce servis içinden egzersizler firestore'dan fetch edilsin dedik, daha sonra da biraz evveki subject'i subscribe ettik böylece firebase'in her değişiminde subscription sayesinde değişimden haberdar olacağız.

```
  ngOnInit() {
    this.trainingService.fetchAvailableExercises();
    this.trainingService.exerciseChanged.subscribe();
  }
```

Ancak bitmedi, biliyoruz ki component içinde bir subscription yapınca bunu unsubscribe etmeyi de unutmuyoruz sanırım bu da component silinince yine de subscribe işleminin devam etmesini ve hata almamızı engelliyor.

Bunun için önce component içerisinde Subscription'ı import edip yeni bir Subscription objesi oluşturuyoruz:

```
import { Component, OnInit, OnDestroy } from '@angular/core';
import { Observable, Subscription } from 'rxjs';
```

```
export class NewTrainingComponent implements OnInit, OnDestroy {

  exercises: Exercise[];
  exerciseSubscription: Subscription;
```

Yeni oluşturulan subscription'ın içini subscription'ımızla dolduruyoruz, bu sırada subscribe edilen egzersiz listesini de component'in property'sine aktardık.

```
  ngOnInit() {
    this.trainingService.fetchAvailableExercises();
    this.exerciseSubscription = this.trainingService.exercisesChanged.subscrib
e( exercises => (this.exercises = exercises));
  }
```

Daha sonra OnDestroy methodu içerisinde unsubscription işlemini gerçekleştiriyoruz:

```
  ngOnDestroy(){
    this.exercisesSubscription.unsubscribe();
  }
```

Son olarak view içerisinde async pipe'ına ihtiyacımız kalmadı çünkü artık subscription'ı view'de yapmıyoruz:

```
        <mat-
select placeholder="Select a workout" ngModel name="exercise" required>
          <mat-
option *ngFor="let exercise of exercises | async" [value]="exercise.id">
            {{ exercise.name }}
          </mat-option>
        </mat-select>
```

Sonuç olarak servis içerisinde firestore'a bir subscription söz konusu data buradan çekiliyor, daha sonra her yenilenmede servisten componente yine bir emitting işlemi gerçekleşiyor.

How Firebase Manages Subscriptions

Geçtiğimiz başlıkta kodumuzu reconstruct ettik ve firebase iletişimini component yerine servis içerisinde yaptık, ardından da servis içerisinden Subject ile değer emit edip, component içerisinde subscribe ettik.

New-training-component'in içerisinde aşağıdaki method vardı:

```
  ngOnInit() {
    this.trainingService.fetchAvailableExercises();
    this.exerciseSubscription = this.trainingService.exerciseChanged.subscribe
( exercises => this.exercises);
  }
```

Burada kırmızı satırla yapılan şey component her yaratıldığında servis içerisindeki fetchAvailableExercises() methodunu çağırmak. Bu component görünür olduğunda yaratılacak, örneğin logout yapıp tekrar login yaparsak, new-training-component silinip baştan yaratılacak.

O halde component her yaratıldığında servis firestore'dan subscription yapacak. Burada bir memory pollution söz konusu olmuyor.

Component kapanında servis içerisindeki subscription otomatik olarak unsubscribe ediliyor, servis bunu bizim için halletmiş oluyor, böyle olmasaydı, component kapatıp açılınça fetch methodu yine çalıştığı için servis içerisinde eski subscription kapanmadan yenisi oluşmuş olacaktı bu da problem yaratırdı.

Storing Completed Exercises on Firestore

Önceki kısımlarda firestore üzerindeki datayı nasıl uygulamaya çekebileceğimizi gördük.

Şimdi ise, uygulama içinden istediğimiz datayı firestore'a göndermeyi göreceğiz.

Normalde bir egzerisz tamamlanınca veya cancel edilince aşağıdaki methodlar çalışıyordu:

```
    completeExercise(){
        this.exercises.push({...this.runningExercise, date: new Date(), state:
 'completed'});
        this.runningExercise = null;
        this.exerciseChanged.next(null);
    }

    cancelExercise(progress:number){
        this.exercises.push({...this.runningExercise, duration:this.runningExe
rcise.duration*(progress/100) , calories:this.runningExercise.calories/(progre
ss/100), date: new Date(), state: 'cancelled'});
        this.runningExercise = null;
        this.exerciseChanged.next(null);
    }
```

Burada yapılan şey, date ve state bilgileri ile local exercises arrayine runningExercise'ı push etmek, ki runningExercise da startExercise methodu ile tanımlanıyordu.

Şimdi yapmak istediğimiz ise complete veya cancel edilen exercise'ları firebase'e yollamak ve yeni bir collection içerisinde saklamak!

Bunun için aşağıdaki methodu tanımlıyoruz, inject edilen db servisi üzerinden yeni bir finishedExercises servisi oluşturuyoruz, ve bu servise exercise'ı upload ediyoruz:

```
    private addDataToDatabase(exercise:Exercise){
        this.db.collection('finishedExercises').add(exercise);
    }
```

Elbette bu methodu completeExercise veya cancelExercise durumlarında kullanmamız lazım:

```
    completeExercise(){
        this.addDataToDatabase({...this.runningExercise, date: new Date(), sta
te: 'completed'});
        this.runningExercise = null;
        this.exerciseChanged.next(null);
    }

    cancelExercise(progress:number){
        this.addDataToDatabase({...this.runningExercise, duration:this.running
Exercise.duration*(progress/100) , calories:this.runningExercise.calories/(pro
gress/100), date: new Date(), state: 'cancelled'});
        this.runningExercise = null;
        this.exerciseChanged.next(null);
    }
```

Sonuçta bu methodun içine verilen şey de bir Exercise objesi olacak ve bu objeler finishedExercises koleksiyonu içinde her biri ayrı bir document olarak kaydedilecek!

Connecting the Data Table to Firestore

Servis içerisinde tanımlanan aşağıdaki method ile, artık finished ve cancelled exercises'ı database'e kaydediyoruz:

```
  private addDataToDatabase(exercise: Exercise) {
    this.db.collection('finishedExercises').add(exercise);
  }
```

Bu işlem de aşağıdaki iki methodun içerisinde addDataToDatabase methodunu çağırarak oluyor.

```
  completeExercise() {
    this.addDataToDatabase({
      ...this.runningExercise,
      date: new Date(),
      state: 'completed'
    });
    this.runningExercise = null;
    this.exerciseChanged.next(null);
  }
 cancelExercise(progress: number) {
    this.addDataToDatabase({
      ...this.runningExercise,
      duration: this.runningExercise.duration * (progress / 100),
      calories: this.runningExercise.duration * (progress / 100),
      date: new Date(),
      state: 'cancelled'
    });
    this.runningExercise = null;
    this.exerciseChanged.next(null);
  }
```

Şimdi past-trainings component'ine verinin nasıl geldiğine bakalım:

```
  ngOnInit() {
    this.dataSource.data = this.trainingService.getCompletedOrCancelledExercis
es();
  }
```

Component initialize edilince training service üzerinden getCompletedOrCancelledExercises methodu ile veri çekiliyordu.

Bu methodu fetchCompletedOrCancelledExercises olarak değiştirdik, artık burada da database'e ulaşıyoruz ve oradan valueChanges() listener'ı ile documents datasını çekiyoruz, ve subscribe ediyoruz. Artık finishedExercises koleysiyonu ne zaman değişse servis içerisine exercises düşecek ve bu exercises finishedExercisesChanges subject'i ile past-trainings component'ine emit edilecek.

```
  fetchCompletedOrCancelledExercises() {
    this.db.collection('finishedExercises').valueChanges().subscribe((exercise
s: Exercise[]) => {
      this.finishedExercisesChanged.next(exercises);
    });
  }
```

finishedEercisesChanged subject'i past-training içerisinde subscribe edilmeli ki ne zaman yeni data eklense past-trainings bundan haberdar olsun, elbette bu subject'in emit edilebilmesi için önce fetchComp… methodunun çağırılması lazım, bu yüzden past-trainings component'inin init methodu aşağıdaki gibi:

```
  ngOnInit() {
    this.exChangedSubscription = this.trainingService.finishedExercisesChanged
.subscribe((exercises: Exercise[]) => {
      this.dataSource.data = exercises;
    })
    this.trainingService.fetchCompletedOrCancelledExercises();
  }
```

Unsubscribe etmeyi de unutmayalım:

```
  ngOnDestroy(){
    this.exChangedSubscription.unsubscribe();
  }
```

Working with Documents

Official angularfire documentation'ı kullanarak detaylı bilgileri elde edebiliriz, yukarıdaki kısımlarda hep koleksiyonlarla çalıştık, bir dökümana ulaşırken de koleksiyon üzerinden ulaştık.

Direkt olarak doküman işlemleri yapmak da mümkün.

this.db.doc('availableExercises/docID') ile ilgili docID'li dökümana ulaşabiliriz. Bunun üzerinden **.set()** , **.delete()**, **.update()** methodlarını çağırabiliriz.

Örneğin **this.db.doc('availableExercises/docID').update({lastSelected: new Date()})** diyerek ilgili dökümana yeni bir data field eklemiş olduk.

Ayrıca tek bir dökümanın valueChanges() listener'ını kullanarak tüm koleksiyon yerine sadece tek bir dökümanı da dinleyebiliriz.

Sanırım bunu da **this.db.doc('path').valueChanges()** ile yapabiliyoruz.

Adding Firebase Authentication (Signup)

Şimdi authentication konusunu anlayalım, bu konuya geçen sefer hızlı bakmıştık bu sefer burada yazarak çalışacağım. En azından yöntemi iyi anlayalım.

Öncelikle bu kısıma kadar dummy authentication kullanıyorduk bunu anlayalım. Bunun için login ve signup componentlerinde alınan data authService üzerindeki login() registerUser() methodlarına gidiyordu:

Mesela login component'indeki form submit edilince aşağıdaki gibi, form bilgisi registerUser methoduna gönderiliyor.

```
onSubmit() {
  this.authService.registerUser({
    email: this.loginForm.value.email,
    password: this.loginForm.value.password
  })
}
```

AuthService içerisindeki bu methoda bakarsak yapılan şey authService içerisindeki user attribute'unu güncellemek ve sonra authChange subject'ini emit etmek, son olarak da route'ı değiştirmek.

```
registerUser(authData: AuthData){
  this.user = {
    email: authData.email,
    userId: Math.round(Math.random() * 1000).toString()
  };
  this.authSuccessfully();
}
```

```
 private authSuccessfully(){

  this.authChange.next(true);
  this.router.navigate(['/training'])
}
```

Yani login sonrası servis içindeki attribute değişiyor ve authChange subject'i ile header'a ve sidenav'e true değeri emit ediliyor bu değer ise o component'lerde isAuth attribute'una atanıyor böylece eğer auth gerçekleşmişse header'dan ve side-nav'den bazı yazılar siliniyor.

Ayrıca authService'in isAuth methodu current user'ın dolu olup olmadığına bakarak sonuç veriyodu

```
  isAuth(){
    return this.user != null;
  }
```

Ki bu sonuç da routh protection için auth-guard içinde kullanılıyordu, eğer isAuth true ise yani bir kullanıcı tanımlı ise guard'ın koruduğu route'lara erişim sağlanacaktı.

```
  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot){
     if (this.authService.isAuth()) { //Guarding koşulu
       return true;  //Eğer this.authService.isAuth() true ise, guard erişime
izin verecek.
     } else {
       this.router.navigate(['/login']); //Yok değilse, guard /login sayfasına
 yönlendirme yapacak.
     }

     //Hangi sayfaların guard edileceğine, router içerisinde karar vereceğiz.
  }
```

Imports and setup

Eski dummy mantığı anladığımıza göre, şimdi firebase kullanarak bu işlemleri kullanışlı biçimde nasıl yapacağımıza bakalım:

Öncelikle aynen Firestore database'i kullanırken yaptığımız gibi angular'ın auth servisini kullanmak için bazı importlar yapmamız gerek ilk olarak app.module.ts içerisinde, `AngularFireAuthModule'ü` import ediyoruz, aynen firestoremodule gibi:

```
import { AngularFirestoreModule} from '@angular/fire/firestore';
import { AngularFireAuthModule} from '@angular/fire/auth';
```

Elbette bunu imports içine de ekliyoruz.

Daha sonra auth.service içerisinde AngularFireAuth servisini import ediyoruz:

```
import {AngularFireAuth} from 'angularfire2/auth';
```

Bu servisi şimdi auth service içerisine inject edeceğiz:

```
  constructor(private router: Router, private afAuth: AngularFireAuth)
```

Artık firebase'in auth servisini kendi auth servisimiz içerisinde afAuth ismiyle kullanabiliriz.

Son olarak firebase console üzerinden uygulamamıza gidip > Authentication > Sign-in method üzerinden > E-post/Şifre kısmını etkinleştirmemiz gerek.

Şimdi auth service'imiz içerisinde eskiden tanımlı registerUser methodunu değiştirelim, burada formdan gelen veri ile bir user yaratıyorduk:

```
registerUser(authData: AuthData){
  this.user = {
    email: authData.email,
    userId: Math.round(Math.random() * 1000).toString()
  };
  this.authSuccessfully();
}
```

Şimdi ise bunu aşağıdaki forma çevirelim, burada ne yapıldı, afAuth üzerinden auth methodu üzerinden bir built-in method çağırıldı, bu method içerisine aldığı email ve password ile firebase üzerinde bir kullanıcı oluşturur.

```
registerUser(authData: AuthData){
  this.afAuth.auth.createUserWithEmailAndPassword(
    authData.email,
    authData.password
    ).then(result => {
      console.log(result);
    }).catch(error => {
      console.log(error);
    });
  this.authSuccessfully();
}
```

Bu oluşturma bittikten sonra da bu method bize bir promise return eder, bu promise'i then ve catch yapısı ile işleyebiliriz, eğer işlem başarılı ise then methodunun içi çalışır, result succes response'a karşılık gelir, eğer hata aldıysak catch çalışacaktır burada error error response'u temsil eder.

Artık kullanıcıyı firebase'e kaydettik. Şimdi sırada bu kullanıcı bilgisini kullanarak, kullanıcıyı login edebilmek var, bu amaçla yine auth.service üzerindeki login methodunu değiştiriyoruz:

```
  login(authData: AuthData){
    this.afAuth.auth.signInWithEmailAndPassword(
      authData.email,
      authData.password
      ).then(result => {
        console.log(result);
      }).catch(error => {
        console.log(error);
      });
    this.authSuccessfully();
    this.authSuccessfully();
  }
```

Burada da afAuth servisinin built-in bir diğer methodu olan signInWithEmailandPassword() methodunu kullanıyoruz ve eğer girilen email ve password firebase users'a kayıtlı ise then çalışacak ve success response alacağız yok hatalı giriş ise catch kısmı çalışacaktır.

Fakat unutma ki şuan sonucun başarılı olup olmamasına göre bize sadece bir response dönüyor, uygulama üzerinde bir routing yapılmadı veya user attribute'una bir atama yapılmadı.

Şimdi bu işlemler için auth.service içerisine bir attribute tanımlayalım:

```
private isAuthenticated = false;
```

Ayrıca login ve register methodları da şöyle değişti:

```
  registerUser(authData: AuthData){
    this.afAuth.auth.createUserWithEmailAndPassword(
      authData.email,
      authData.password
      ).then(result => {
        this.authSuccessfully();
      }).catch(error => {
        console.log(error);
      });
  }

  login(authData: AuthData){
    this.afAuth.auth.signInWithEmailAndPassword(
      authData.email,
      authData.password
      ).then(result => {
        this.authSuccessfully();
      }).catch(error => {
```

```
        console.log(error);
      });
  }
```

Artık başarılı register veya loginden sonra aşağıdaki method çağırılıyor:

```
  private authSuccessfully(){
    this.isAuthenticated = true;
    this.authChange.next(true);
    this.router.navigate(['/training'])
  }
```

Burada isAuthenticated attribute'u true yapılıyor, ve navigation gerçekleşiyor. Logout içinde ise isAuthenticated attribute'u tekrar false yapılıyor, ayrıca afAuth üzerinden signOut() çalıştırılıyor bu backend üzerindeki authentication'ı kırıyor, token'i siliyor.

```
  logout(){
    this.afAuth.auth.signOut();
    this.authChange.next(false);
    this.router.navigate(['/login'])
    this.isAuthenticated = false;
  }
```

Artık bu yöntemle, signup componenti üzerinden firebase' yeni kullanıcı ekleyebiliyoruz ve login üzerinden bu kullanıcılar login olurken firebase üzerindeki kullanıcılarla karşılaştırılıyor.

Fakat unutma şuanda login olmak sadece belirli route'lara erişim sağlıyor o kadar, örneğin egzersiz verileri ve past exercises hala tüm kullanıcılar için ortak.

Ayrıca şuan authentication sadece front-end ile ilgili yani formdan alınan veriler firebase verileri ile karşılaştırılıyor ancak bir kullanıcı authentike olduğunda tek değişen front-end üzerinde bu kullanıcıya belirli route'ların açılması, ve bu route açılması da yine front-end içerisinde saklanan bir isAuthenticated boolena'ı ile yapılıyor. Bu bir güvenlik açığı, bizim asıl authentication'ını backend ile yapmamız lazım, front-end üzerinde tutulan bir boolean ile authentication yapmak güvenli değil.

Şuanki configuration ile biri isAuthenticated=true yaptığı anda front-end üzerinden firebase'e ulaşabilir, çünkü firestore tüm requestlere cevap veriyor, security rules şuan öyle ayarlandı.

Configuring Firestore Security Rules

Firestore üzerinden rules kısmına gidersek güncel rules'u görebiliriz:

```
1. rules_version = '2';
2. service cloud.firestore {
3.   match /databases/{database}/documents {
4.     match /{document=**} {
5.       allow read, write;
6.     }
7.   }
8. }
9.
```

Şuanda allow read,write; olarak tanımlanmış yani environment içerisinde belirtilen configuration ile herkes firebase'e ulaşabiliyor.

```
1. rules_version = '3';
2. service cloud.firestore {
3.   match /databases/{database}/documents {
4.     match /{document=**} {
5.       allow read, write: if request.auth!=null;
6.     }
7.   }
8. }
9.
```

Yukarıdaki set-up ile artık firebase yalnızca authenticated kullanıcıların request'ine cevap verir.

Normalde bu set-up dan sonra login olunca direkt verilere ulaşamamayı bekleriz çünkü firebase'e authentike olduğumuzu hiçbiryerde söylemiyoruz, sadece login olurken aşağıdaki method ile login olduk o kadar

```
this.afAuth.auth.signInWithEmailAndPassword()
```

Ancak sadece bu methodu kullanmamız yeterli, firebase bizim için artık backend üzerinde authentication işlemlerini gerçekleştiriyor.

Yani olay şu: şuan firestore protected bir biçimde, sadece authenticated users ulaşabiliyor, bu users da sadece yukarıdaki method ile login olanlar, biri front-end'i kullanarak isAuthenticated boolean'ını değiştirerek hala training sayfasına girebilir ancak burada verileri göremez, verileri görmesi için mutlaka ve mutlaka yukarıdak method ile login yapması şart!

Managing Firestore Subscriptions

Şimdi firestore'a authentication security rule'unu ekledik artık sadece afAuth methodu ile login olan kullanıcıların request'leri devreye alınıyor, bunun dışında sadece front-end hilesi ile request göndermeye çalışırsak hata alıyoruz.

Ancak bu şöyle bir probleme yol açıyor, kullanıcı logout dediğinde authentication'ı sonlanıyor, ancak training service içerisindeki database subscriptions'ı sonlanmıyor, normalde servis içerisinde subscription yapılan bir method, component içinden çağırıldığında ve component sonra kapandığında service'in subscription'ını unsubscribe etmeye gerek duymazdık, çünkü servis üzerinden 2 kere üstüste subscribe olmak mümkün olmazdı.

Ancak, logout olunca bu subscription'lar sonlanmadığı için ve authentication olmadan firebase bu subscription'lara izin vermediği için hata alacağız, bu yüzden bu subscription'ları unsubscribe ederiz:

Training service içerisinde, tanımlarız:

```
  private fbSubs: Subscription[] = [];
```

Sonra iki farklı subscription'ını da bu array'e push ederiz, biri aşağıda verilmiş diğerini vermiyorum.

```
fetchAvailableExercises(){
    this.fbSubs.push(this.db.collection('availableExercises')
        .snapshotChanges()
        .pipe(map(docArray => {
            return docArray.map(doc => {
                return {
                id: doc.payload.doc.id,
                name: doc.payload.doc.data()['name'],
                duration: doc.payload.doc.data()['duration'],
                calories: doc.payload.doc.data()['calories']
```

```
                };
            });
        })).subscribe((exercises: Exercise[]) => {
            this.availableExercises = exercises;
            this.exercisesChanged.next([...this.availableExercises]);
        }));
}
```

Sonra cancelSubs methodu tanımlarız:

```
  cancelSubscriptions(){
    this.fbSubs.forEach(sub => sub.unsubscribe());
  }
```

Bu methodu da auth.service'in logout'u içinde çağırırız:

```
  logout(){
    this.trainingService.cancelSubscriptions();
    this.afAuth.auth.signOut();
    this.authChange.next(false);
    this.router.navigate(['/login'])
    this.isAuthenticated = false;
  }
```

Reorganizing the Code

Angularfire'ın bize sağladığı bir feature ile auth status'teki herhangi bir değişimden direkt haberdar olabiliriz, bu yüzden şuanda login, register, logout methodlarında kullanılan authSuccessfuly() methodu ve logout yapısı değişebilir.

Bunun için auth.Service içerisine yeni bir method tanımlıyorum, burada authState'i subscribe ediyorum, ne zaman statu değişti bana bir user gelecek, eğer user null ise authentication yok demektir o halde, logout işlemleri yapılacak, eğer auth var ise yani login yapılırsa hemen authsuccessful işlemleri yapılacak:

```
initAuthListener(){
  this.afAuth.authState.subscribe(user => {
    if(user){
      this.isAuthenticated = true;
      this.authChange.next(true);
      this.router.navigate(['/training'])
    } else {
      this.trainingService.cancelSubscriptions();
      this.authChange.next(false);
      this.router.navigate(['/login'])
      this.isAuthenticated = false;
    }
  });
}
```

Bu yüzden artık logout aşağıdaki gibi olacak:

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

Authsuccesfully'e gerek kalmadı, login ve register de aşağıdaki gibi olacak:

```
registerUser(authData: AuthData){
  this.afAuth.auth.createUserWithEmailAndPassword(
    authData.email,
    authData.password
    ).then(result => {
    }).catch(error => {
```

```
        console.log(error);
      });
  }

  login(authData: AuthData){
    this.afAuth.auth.signInWithEmailAndPassword(
      authData.email,
      authData.password
      ).then(result => {
      }).catch(error => {
        console.log(error);
      });
  }
```

Son olarak bu yeni yaratılan initAuthListener'ın çağırılması lazım, bunun app başladığı anda yapmak en mantıklısı:

İlk başlatılan component app.component.ts olduğu için auth service'i buraya inject edip bu methodu çağıralım:

```
export class AppComponent implements OnInit{

  constructor(private authService: AuthService){}

  ngOnInit(): void {
        this.authService.initAuthListener();
  }

}
```