Firebase

Martin Galajda, Lenka Janečková

Introduction

- Cloud-hosted database
- Backend-as-a-Service (BaaS)
- Started as a YC11 startup, acquired by Google in 2014
- Data stored in JSON and synchronized to every connected client
- Supports iOS, Android, C++, Web apps, REST API, Unity, ...
- Used by Shazam, Skyscanner, Booking.com, Viber, ...

Introduction

- Other features Firebase supports:
  - Storage
  - Hosting
  - Authentication
  - Notifications
  - Cloud functions
  - Cloud messaging
  - Analytics
  - Remote config
  - Crash reporting

How does it work?

- The clients connect directly to the database in the cloud and don't have to go through the application's server
- No need to worry about the backend server, database, real-time component (socket.io) or writing REST API
- App is connected to Firebase through WebSockets
- The app just sends data to Firebase and it handles saving and syncing across all connected devices / sites
- All data is synced through the single WebSocket connection

Writing data offline

- Every client connected to a Firebase database maintains its own internal version of any active data
- Data is written to this local version first
- The Firebase client synchronizes that data on a "best-effort" basis.
- All writes to the database trigger local events immediately, before any data is written to the server
- Once connectivity is reestablished, the app receives the set of events so that the client syncs with the current server state

Authentication

- Built in email/password authentication system
- Supports OAuth2 for Google, Facebook, Twitter and GitHub
- Integrates directly into Firebase database – can be used to control access to data

Firebase Database Rules

- Determine who has:
  - read and write access to the database
  - how data is structured
  - what indexes exist
- These rules live on the Firebase servers and are enforced automatically at all times
- Every read and write request will only be completed if the rules allow it.
- By default only authenticated users can read/write data

Firebase Database Rules

- .read
  - if and when data is allowed to be read by users
- .write
  - if and when data is allowed to be written
- .validate
  - what a correctly formatted value will look like, whether it has child attributes, and the data type
- .indexOn
  - specifies a child to index to support ordering and querying

Firebase Database Rules Example

- Built-in variables and functions that allow you to refer to other paths, server-side timestamps, authentication information, …

```
{
  "rules": {
    "users": {
      "$uid": {
        ".write": "$uid === auth.uid"
      }
    }
  }
}
```

```
{
  "rules": {
    "foo": {
      ".validate": "newData.isString() && newData.val().length < 100"
    }
  }
}
```

Database indexes

- Indexes are specified using the *.indexOn* rule
- Example index declaration that would index the height and length fields for a list of dinosaurs:

```
{
  "lambeosaurus": {
    "height" : 2.1,
    "length" : 12.5,
    "weight": 5000
  },
  "stegosaurus": {
    "height" : 4,
    "length" : 9,
    "weight" : 2500
  }
}
```

```
{
  "rules": {
    "dinosaurs": {
      ".indexOn": ["height", "length"]
    }
  }
}
```

Structuring the database

- You need to plan for how data is going to be saved and later retrieved to make that process as easy as possible
- Data is stored as JSON objects
- When you add data to the JSON tree, it becomes a node in the existing JSON structure with an associated key
- You can provide your own keys, such as user IDs or semantic names, or they can be provided for you using *push()*

Scaling database - sharding and data replication

- does not provide data replication by default
- sharding can be achieved by creating multiple firebase instances (projects)
- e.g. firebase instance for each aggregate entity

Structuring the database - validation

```
{
  "rules": {
    // write is allowed for all paths
    ".write": true,
    "widget": {
      // a valid widget must have attributes "color" and "size"
      // allows deleting widgets (since .validate is not applied to delete rules)
      ".validate": "newData.hasChildren(['color', 'size'])",
      "size": {
        // the value of "size" must be a number between 0 and 99
        ".validate": "newData.isNumber() &&
                      newData.val() >= 0 &&
                      newData.val() <= 99"
      },
      "color": {
        // the value of "color" must exist as a key in our mythical
        // /valid_colors/ index
        ".validate": "root.child('valid_colors/' + newData.val()).exists()"
      }
    }
  }
}
```

- Rules are made up of Javascript-like expressions contained in a JSON document

Basic usage

Initializing the Realtime Database

```
// Set the configuration for your app
// TODO: Replace with your project's config object
var config = {
  apiKey: "apiKey",
  authDomain: "projectId.firebaseapp.com",
  databaseURL: "https://databaseName.firebaseio.com",
  storageBucket: "bucket.appspot.com"
};
firebase.initializeApp(config);

// Get a reference to the database service
var database = firebase.database();
```

Write operation

- Method *set()* saves data to a specified reference, replacing any existing data at that path, including any child nodes

```
function writeUserData(userId, name, email, imageUrl) {
  firebase.database().ref('users/' + userId).set({
    username: name,
    email: email,
    profile_picture : imageUrl
  });
}
```

Read operation

- The *value* event is fired every time data is changed at the specified reference, including changes to child nodes
- The event callback is passed a snapshot containing all data at that location which existed at the time of the event

```
var starCountRef = firebase.database().ref('posts/' + postId + '/starCount');
starCountRef.on('value', function(snapshot) {
  updateStarCount(postElement, snapshot.val());
});
```

Update operation

- *update()* method called on a reference to the location of data
- Enables simultaneous updates to multiple locations in the JSON tree with a single call
- Simultaneous updates made this way are atomic: either all updates succeed or all updates fail

```
// Get a key for a new Post.
var newPostKey = firebase.database().ref().child('posts').push().key;

// Write the new post's data simultaneously in the posts list and the user's post list.
var updates = {};
updates['/posts/' + newPostKey] = postData;
updates['/user-posts/' + uid + '/' + newPostKey] = postData;

return firebase.database().ref().update(updates);
```

Read data once

- snapshot of your data without listening for changes
- *once()* method - it triggers once and then does not trigger again.

Delete operation

- *remove()* method called on a reference to the location of the data

Promise

- When the data is committed to the database, *set()* and *update()* operations can return *Promise*
- A Promise represents an eventual (asynchronous) value
- When it gets resolved, *.then()* callback function will be called
- if it gets rejected, *.catch()* callback will be called

Detach listeners

- Method *off()* on a database reference

Transaction operation

- When working with data that could be corrupted by concurrent modifications
- *transaction()* method takes an update function and an optional completion callback.
- The update function takes the current state of the data as an argument and returns the new desired state
- If the transaction is rejected, the server returns the current value to the client, which runs the transaction again with the updated value.
- This repeats until the transaction is accepted or aborted.

Firebase CLI

- provides a variety of tools for managing, viewing, and deploying to Firebase projects
- **npm install –g firebase-tools**
  - Provides a globally available firebase command available from any terminal window
- **firebase login**
  - connects your local machine to your Firebase account and grants access to your projects
- **firebase list**
  - lists of all of your Firebase projects
- **firebase init**
  - steps you through setting up your project directory, including asking which Firebase features you want to use

Firebase CLI

- **firebase serve**
  - Starts a local web server with Firebase Hosting configuration
- **firebase deploy**
  - creates new releases for all deployable resources in your project directory
  - A project directory **must** have a firebase.json
- **firebase database:get | database:set | database:update | database:push | database:remove**
  - Database commands

Profiling the database

- Supports a database profiler tool, built into the Firebase CLI
- Logs all the activity in the database over a given period of time, then generates a detailed report
- **firebase database:profile**
  - Starts profiling the database
- The profiler tool aggregates the data about the database's operations and displays the results in three primary categories:
  - **Speed** - response time for each operation
  - **Bandwidth** - how much data is consumed across incoming and outgoing operations
  - **Unindexed queries**

DEMO

https://shrouded-ridge-84643.herokuapp.com/

More resources about Firebase

- Official Firebase page:
  - https://firebase.google.com/
- Official Firebase Realtime Database page:
  - https://firebase.google.com/docs/database/

Thanks for your attention.

Any questions?