Firebase

Realtime Database

Agenda

About Firebase

End user demo

Code view

- Vue.js (time permitting)

What is Firebase?

Firebase helps you build and run successful **apps**

BaaS (backend as a service)

- build: Accelerate app development with fully managed backend infrastructure
  - Cloud Firestore, Realtime Database, Authentication, Cloud Functions, Cloud Messaging, Hosting, Cloud Storage
- release & monitor: Release with confidence and monitor performance and stability
  - Google Analytics, Performance Monitoring, Test Lab
- engage: Boost engagement with rich analytics, A/B testing, and messaging campaigns
  - Google Analytics, Crashlytics, Predictions, Cloud Messaging

Realtime Database vs Cloud Firestore

Firestore

- Firebase's newest database for mobile app development. It builds on the successes of the Realtime Database with a new, more intuitive data model. Cloud Firestore also features richer, faster queries and scales further than the Realtime Database.
- Store your data in documents, organized into collections. Documents can contain complex nested objects in addition to subcollections.
- Use queries to retrieve individual, specific documents or to retrieve all the documents in a collection that match your query parameters. Your queries can include multiple, chained filters and combine filtering and sorting.

Realtime Database

- Firebase's original database. It's an efficient, low-latency solution for mobile apps that require synced states across clients in realtime. (Translation: on Google deathwatch??)
- Store and sync data with our NoSQL cloud database. Data is synced across all clients in realtime, and remains available when your app goes offline.
- Store data in documents that contain fields mapping to values. These documents are stored in collections, which are containers for your documents that you can use to organize your data and build queries

Data modeling

Stores data as one large JSON tree.

- Simple data is very easy to store.
- Complex, hierarchical data is harder to organize at scale.
- Provide your own keys, such as user IDs or semantic names, or they can be provided for you using push()

Best practices

- Avoid nested data
  - when you fetch data at a location, you also retrieve all of its child nodes
  - granting access at a node grants access to all data under that node
- Flatten data structures
  - can be efficiently downloaded in separate calls, as it is needed
- Create an index
  - Two way relationships: inver the data by listing the IDs as keys and setting the value to true
  - Example: groups have members; what groups does a member belong to?

Security: Firebase Realtime Database Security Rules

JavaScript-like syntax and JSON structure

- .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 - a child to index to support ordering and querying.

Read and write rules cascade, validate and indexOn don’t.

Security: read and write rules

```
{
  "rules": {
    "foo": {
      ".read": true,
      ".write": false
    },
    "users": {
      "$uid": {
        ".write": "$uid === auth.uid"
      }
    }
  }
}
```

Security: validate

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

“data written to /foo/ must be a string less than 100 characters”

Security: index

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

Index the height and length fields for a list of dinosaurs.

Specify indexes for queries so that they continue to work as your app grows.

Offline

Offline support for iOS and Android clients (not web).

Data is persisted locally, and even while offline, realtime events continue to fire, giving the end user a responsive experience. When the device regains connection, the Realtime Database synchronizes the local data changes with the remote updates that occurred while the client was offline, **merging any conflicts automatically**.

- doesn’t really mean merge: latest update wins

Two choices:

- transactions
- prevent conflicts (append-only writes)

Offline: transactions

Transaction operation

- update function and an optional completion callback
- takes the current state of the data and returns the new desired state you want to write
- if current value is X, set to Y, otherwise retry
- example: https://firebase.google.com/docs/database/web/read-and-write#save_data_as_transactions
- doesn’t work offline

Offline: prevent conflicts

- appending new updates
- ref.push(), which generates unique locations in chronological order.
- each client reads all updates for that shape and recalculates

```
shapes

  shapeid1

    pushid1: {"shape": "rectangle", "stroke": 10, "color": "black"} /*
initial data */

    pushid2: { "stroke": 5 } /* first update */

    pushid3: { "stroke": 20 } /* second update */
```

Live demo

http://kielni-trading-post-brownbag.s3-website-us-east-1.amazonaws.com/

- real time sync (add/update)

Console view

- permissions
- data

Code view

Authentication

API key

Firebase web SDK

Initialize

Firebase ref

Query

Set

References

- Choose a database
- Realtime database web SDK
- Understanding conflict resolution
- Vue.js - getting started (Vue 2)
- Vuefire - Vue.js bindings for Firebase
- Shopping list sample app