Persistence

SWE 432, Fall 2019

Web Application Development

Today

- More on design considerations in identifying resources and REST
- Persistence

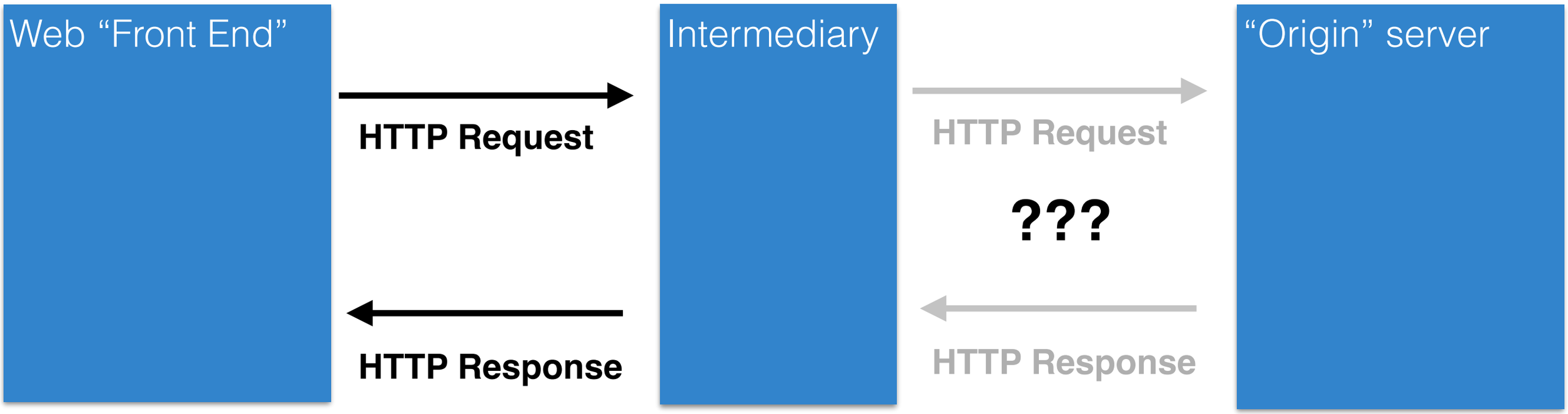
Review: Building a microservice w/ Express

Review: Application Programming Interface

- Microservice offers public **interface** for interacting with backend
  - Offers abstraction that hides implementation details
  - Set of endpoints exposed on micro service

- Users of API might include
  - Frontend of your app
  - Frontend of other apps using your backend
  - Other servers using your service

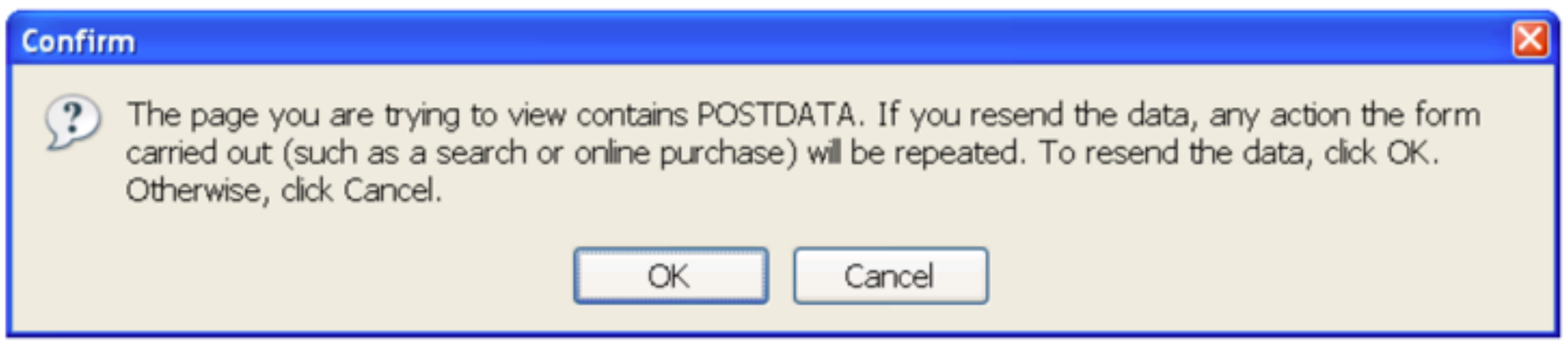
Review: Intermediaries



- Client interacts with a resource identified by a URI
- But it never knows (or cares) whether it interacts with origin server or an unknown intermediary server
  - Might be randomly load balanced to one of many servers
  - Might be cache, so that large file can be stored locally
    - (e.g., GMU caching an OSX update)
  - Might be server checking security and rejecting requests

Review: HTTP Actions

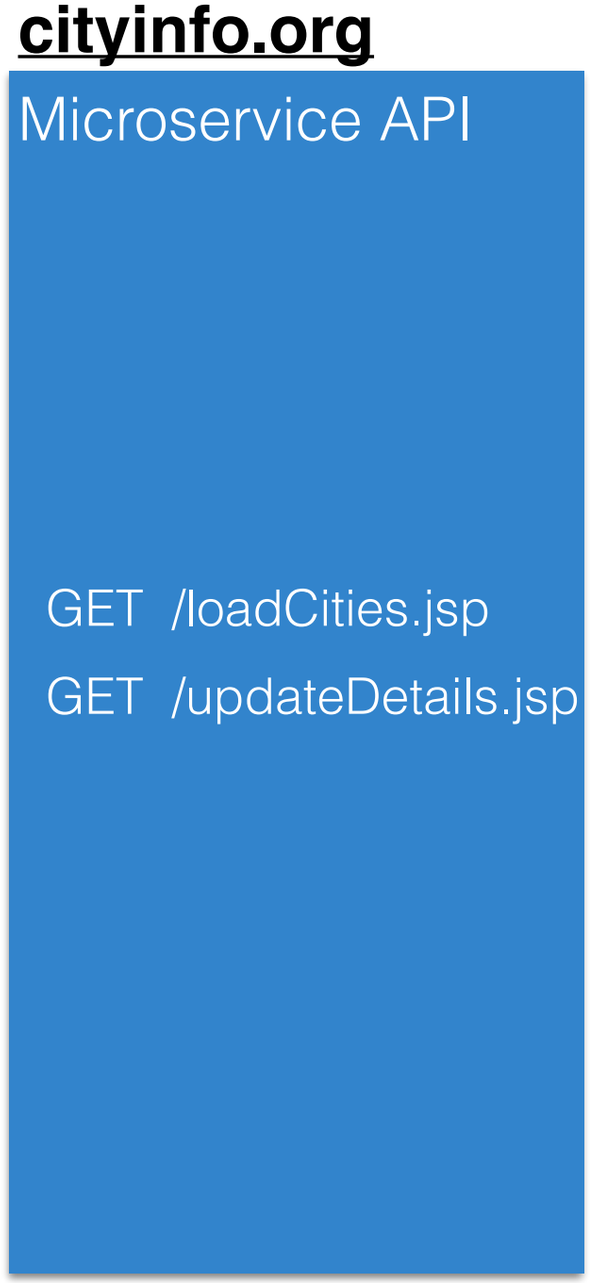
- GET: safe method with no side effects
  - Requests can be intercepted and replaced with cache response
- PUT, DELETE: idempotent method that can be repeated with same result
  - Requests that fail can be retried indefinitely till they succeed
- POST: creates new element
  - Retrying a failed request might create duplicate copies of new resource



Support scaling

- Yesterday, cityinfo.org had 10 daily active users. Today, it was featured on several news sites and has 10,000 daily active users.
- Yesterday, you were running on a single server. Today, you need more than a single server.

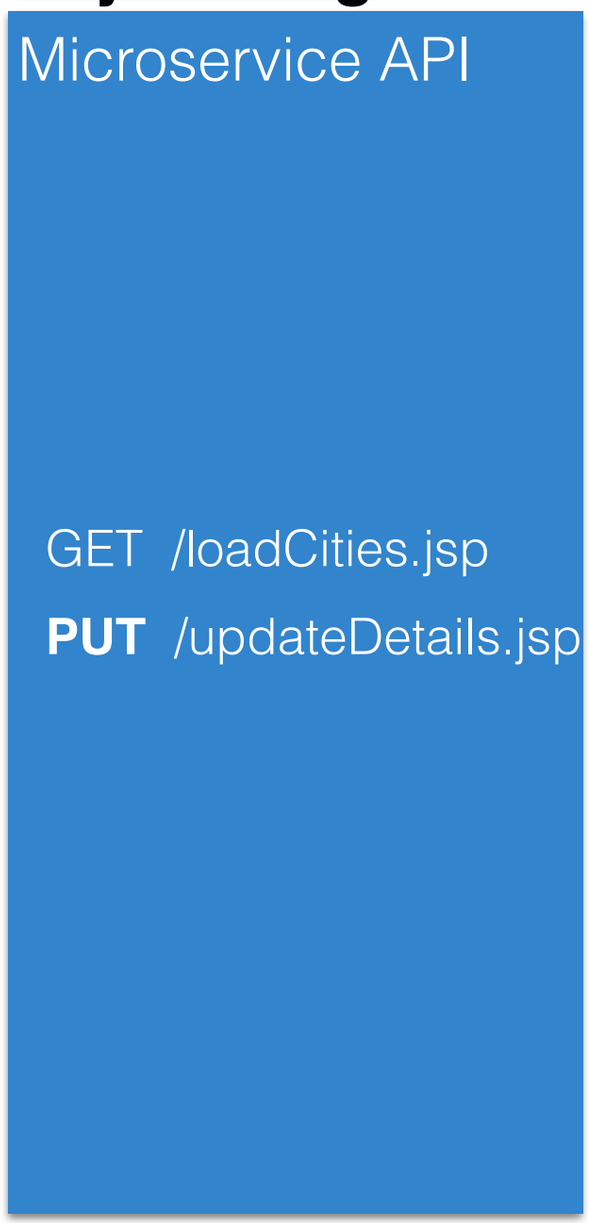
- Can you just add more servers?
  - What should you have done yesterday to make sure you can scale quickly today?



Support scaling

- Yesterday, cityinfo.org had 10 daily active users. Today, it was featured on several news sites and has 10,000 daily active users.
- Yesterday, you were running on a single server. Today, you need more than a single server.

- Can you just add more servers?
  - What should you have done yesterday to make sure you can scale quickly today?



Versioning

- Your web service just added a great new feature!
  - You'd like to expose it in your API.
  - But… there might be old clients (e.g., websites) built using the old API.
    - These websites might be owned by someone else and might not know about the change.
  - Don't want these clients to throw an error whenever they access an updated API.

Cool URIs don’t change

- In theory, URI could last forever, being reused as server is rearchitected, new features are added, or even whole technology stack is replaced.

- “What makes a cool URI?
  A cool URI is one which does not change.
  What sorts of URIs change?
  URIs don't change: people change them.”
  - https://www.w3.org/Provider/Style/URI.html
  - Bad:
    - https://www.w3.org/Content/id/50/URI.html (What does this path mean? What if we wanted to change it to mean something else?)
- Why might URIs change?
  - We reorganized our website to make it better.
  - We used to use a cgi script and now we use node.JS.

URI Design

- URIs represent a contract about what resources your server exposes and what can be done with them
- Leave out **anything that might change**
  - Content author names, status of content, other keys that might change
  - File name extensions: response describes content type through MIME header not extension (e.g., .jpg, .mp3, .pdf)
  - Server technology: should not reference technology (e.g., .cfm, .jsp)
- Endeavor to make all changes backwards compatible
  - Add new resources and actions rather than remove old
- If you must change URI structure, support old URI structure **and** new URI structure

Support change

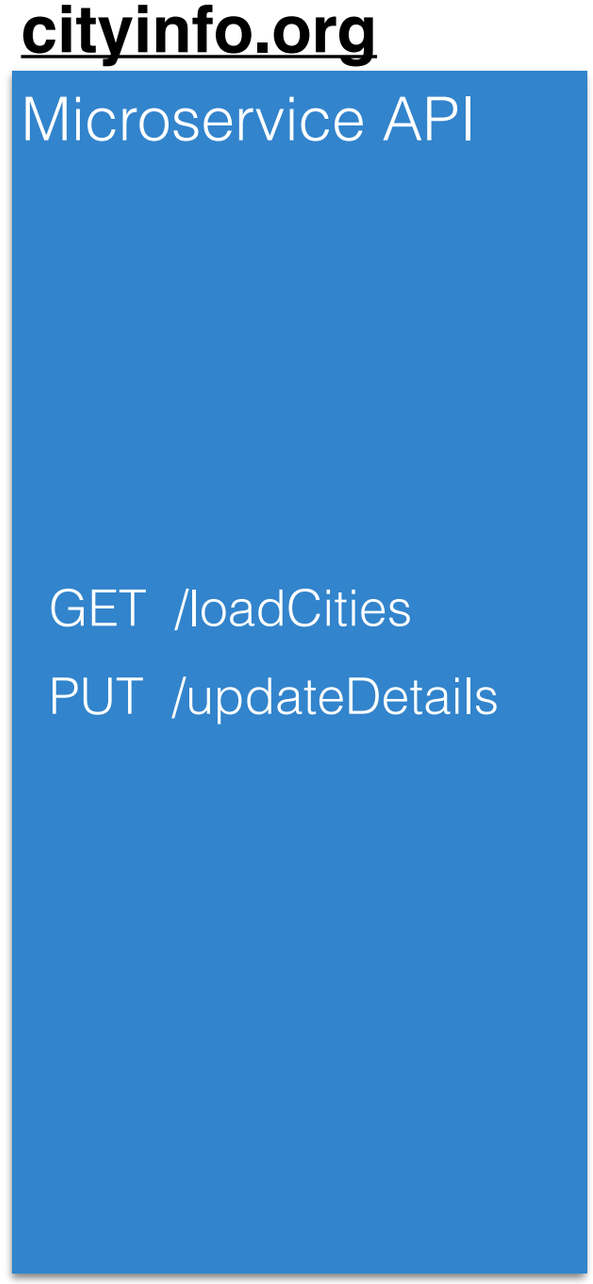
- Due to your popularity, your backend data provider just backed out of their contract and are now your competitor.
- The data you have is now in a different format.
- Also, you've decided to migrate your backend from PHP to node.js to enable better scaling.

- How do you update your backend without breaking all of your clients?

Support change

- Due to your popularity, your backend data provider just backed out of their contract and are now your competitor.
- The data you have is now in a different format.
- Also, you've decided to migrate your backend from PHP to node.js to enable better scaling.

- How do you update your backend without breaking all of your clients?



Nouns vs. Verbs

- URIs should hierarchically identify **nouns** describing **resources** that exist
- Verbs describing actions that can be taken with resources should be described with an HTTP **action**

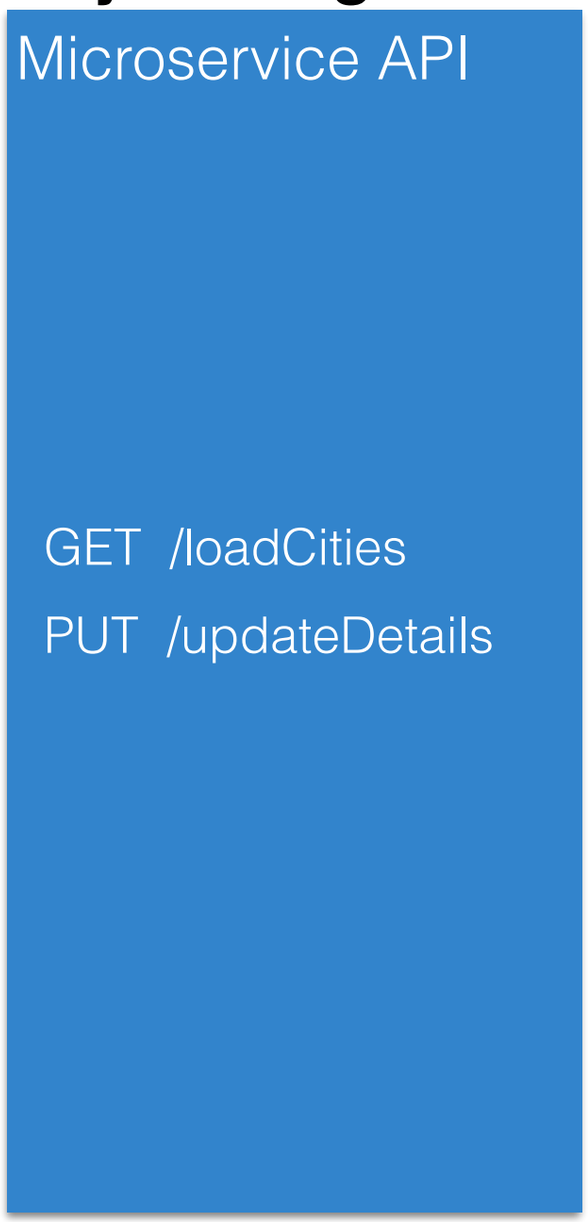
- PUT /cities/:cityID (nouns: cities, :cityID)(verb: PUT)
- GET /cities/:cityID (nouns: cities, :cityID)(verb: GET)

- Want to offer **expressive** abstraction that can be reused for many scenarios

Support reuse

- You have your own frontend for cityinfo.org. But everyone now wants to build their own sites on top of your city analytics.

- Can they do that?



Support reuse

cityinfo.org

Microservice API

/topCities GET
/topCities/:cityID/descrip PUT, GET

/city/:cityID GET, PUT, POST, DELETE
/city/:cityID/averages GET
/city/:cityID/weather GET
/city/:cityID/transitProvders GET, POST
/city/:cityID/transitProvders/:providerID GET, PUT, DELETE

What happens when a request has many parameters?

- /topCities/:cityID/descrip PUT

- Shouldn't this really be something more like
  - /topCities/:cityID/ descrip/:descriptionText/:submitter/:time/

Solution 1: Query strings

- PUT /topCities/Memphis?submitter=Dan&time=1025313

```
var express = require('express');
var app = express();

app.put('/topCities/:cityID', function(req, res){
    res.send(`descrip: ${req.query.descrip} submitter: ${req.query.submitter}`);
});

app.listen(3000);
```

- Use req.query to retrieve
- Shows up in URL string, making it possible to store full URL
  - e.g., user adds a bookmark to URL
- Sometimes works well for short params

Solution 2: JSON request body

- PUT /topCities/Memphis
  { "descrip": "Memphis is a city of ...",
    "submitter": "Dan", "time": 1025313 }
- Best solution for all but the simplest parameters (and often times everything)
- Use body-parser package and req.body to retrieve

```
$npm install body-parser
```

```
var express    = require('express');
var bodyParser = require('body-parser');

var app = express();

// parse application/json
app.use(bodyParser.json());

app.put('/topCities/:cityID', function(req, res){
    res.send(`descrip: ${req.body.descrip} submitter: ${req.body.submitter}`);
});

app.listen(3000);
```

https://www.npmjs.com/package/body-parser

Persistence

- The user sent you some data.
- You retrieved some data from a 3rd party servcie.
- You generated some data, which you want to keep reusing.

- Where and how could you store this?

What forms of data might you have

- Key / value pairs
- JSON objects
- Tabular arrays of data
- Files

Options for backend persistence

- Where it is stored
  - On your server or another server you own
    - SQL databases, NoSQL databases
    - File system
  - Storage provider (not on a server you own)
    - NoSQL databases
    - BLOB store

Storing state in a global variable

- **Global variables**

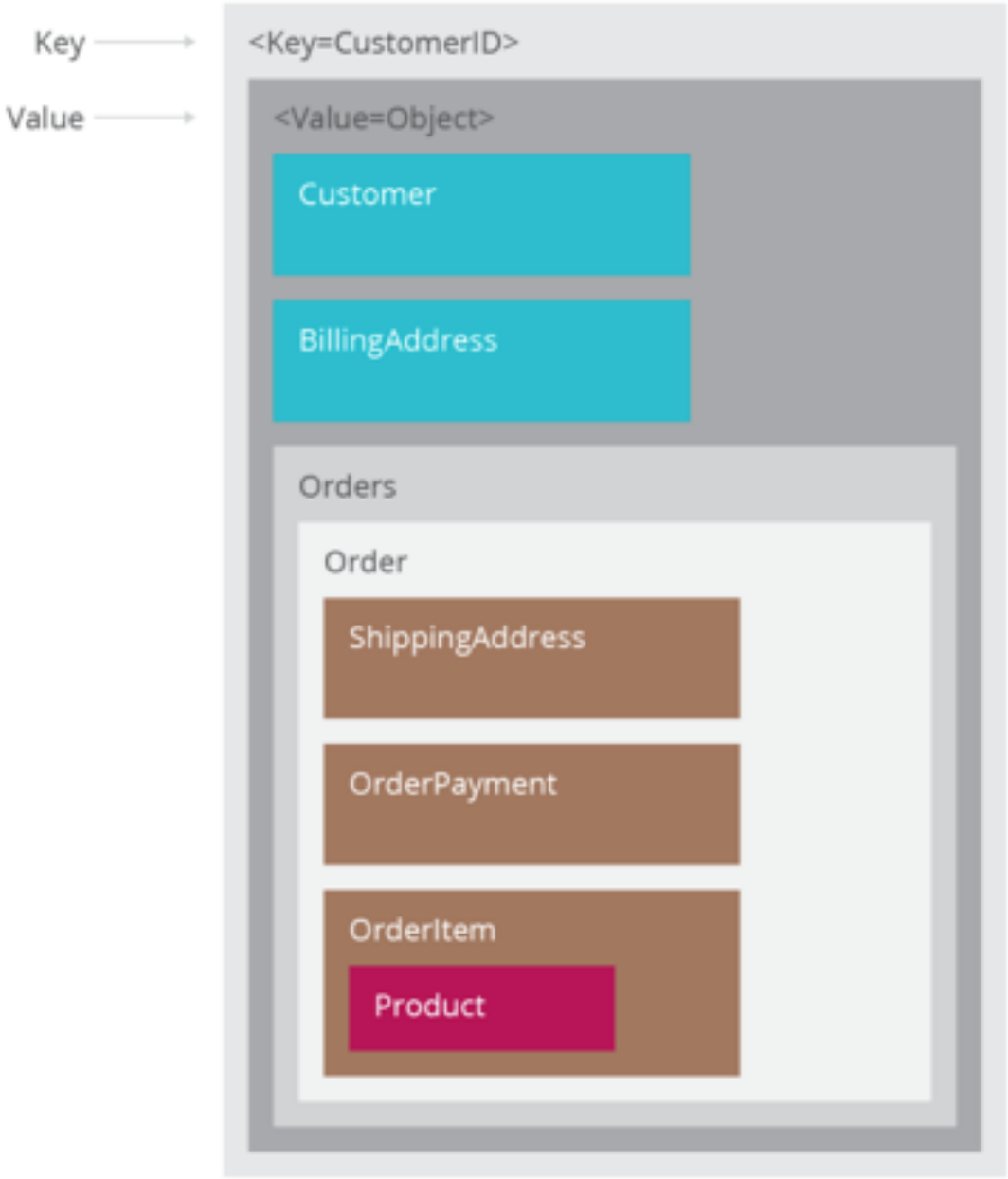
```
var express = require('express');
var app = express();
var port = process.env.port || 3000;

var counter = 0;
app.get('/', function (req, res) {
    res.send('Hello World has been said ' + counter + ' times!');
    counter++;
});

app.listen(port, function () {
    console.log('Example app listening on port' + port);
});
```

- Pros/cons?
  - Keep data between requests
  - **Goes away** when your server stops
    - Should use for transient state or as cache

NoSQL

- **non SQL**, non-relational, "not only" SQL databases
- Emphasizes **simplicity** & scalability over support for relational queries
- Important characteristics
    - Schema-less: each row in dataset can have different fields (just like JSON!)
    - Non-relational: no structure linking tables together or queries to "join" tables
    - (Often) weaker consistency: after a field is updated, all clients *eventually* see the update but may see older data in the meantime
- Advantages: greater scalability, faster, simplicity, easier integration with code
- Several types. We'll look only at key-value.

Key-Value NoSQL



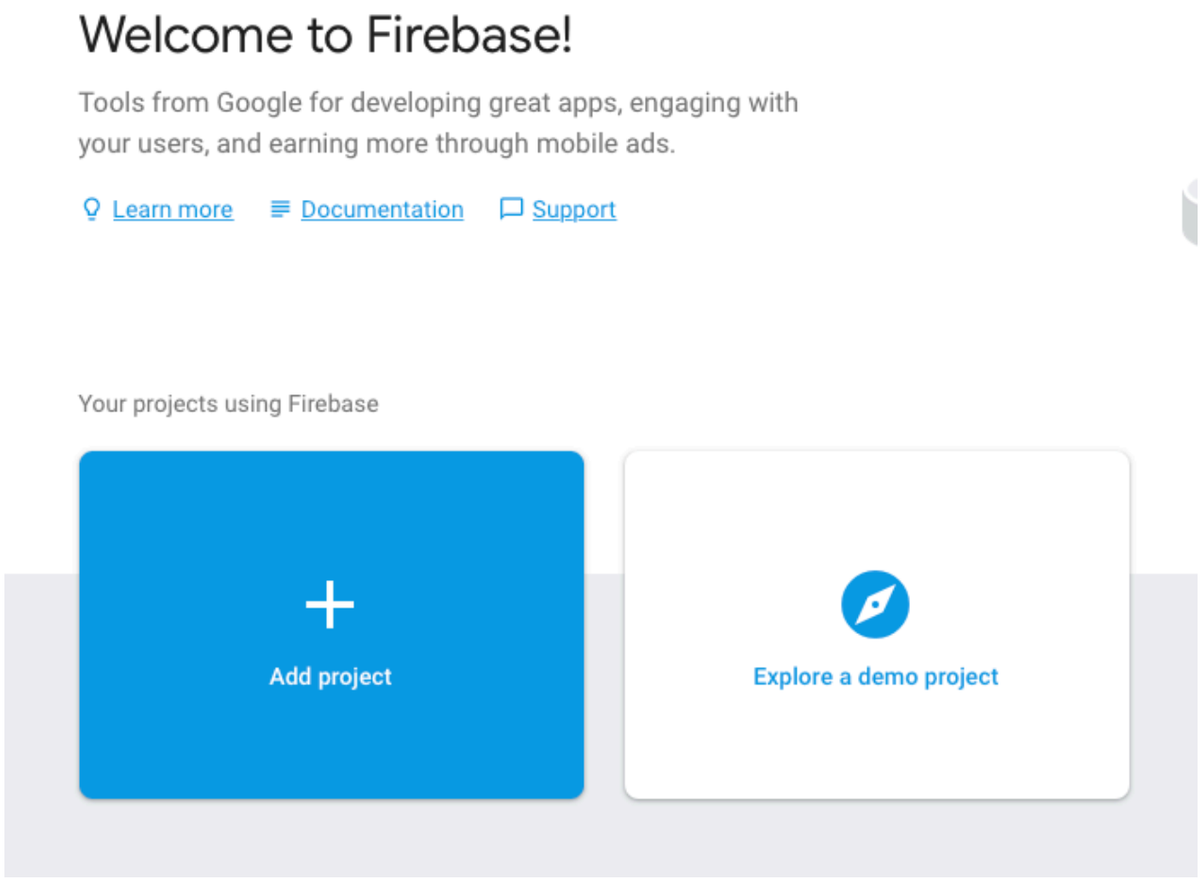
https://www.thoughtworks.com/insights/blog/nosql-databases-overview

Firebase Cloud Firestore

- Example of a NoSQL datastore
- Google web service
  - https://firebase.google.com/docs/firestore/
- “Realtime” database
  - Data stored to remote web service
  - Data synchronized to clients in real time
- Simple API
  - Offers library wrapping HTTP requests & responses
  - Handles synchronization of data
- Can also be used on frontend to build web apps with persistence without backend

Setting up Firebase Cloud Firestore

- Detailed instructions to create project, get API key
  - https://firebase.google.com/docs/firestore/quickstart



Setting up Firebase Realtime Database

- Go to https://console.firebase.google.com/, create a new project

- Install firebase module `npm install firebase-admin --save`

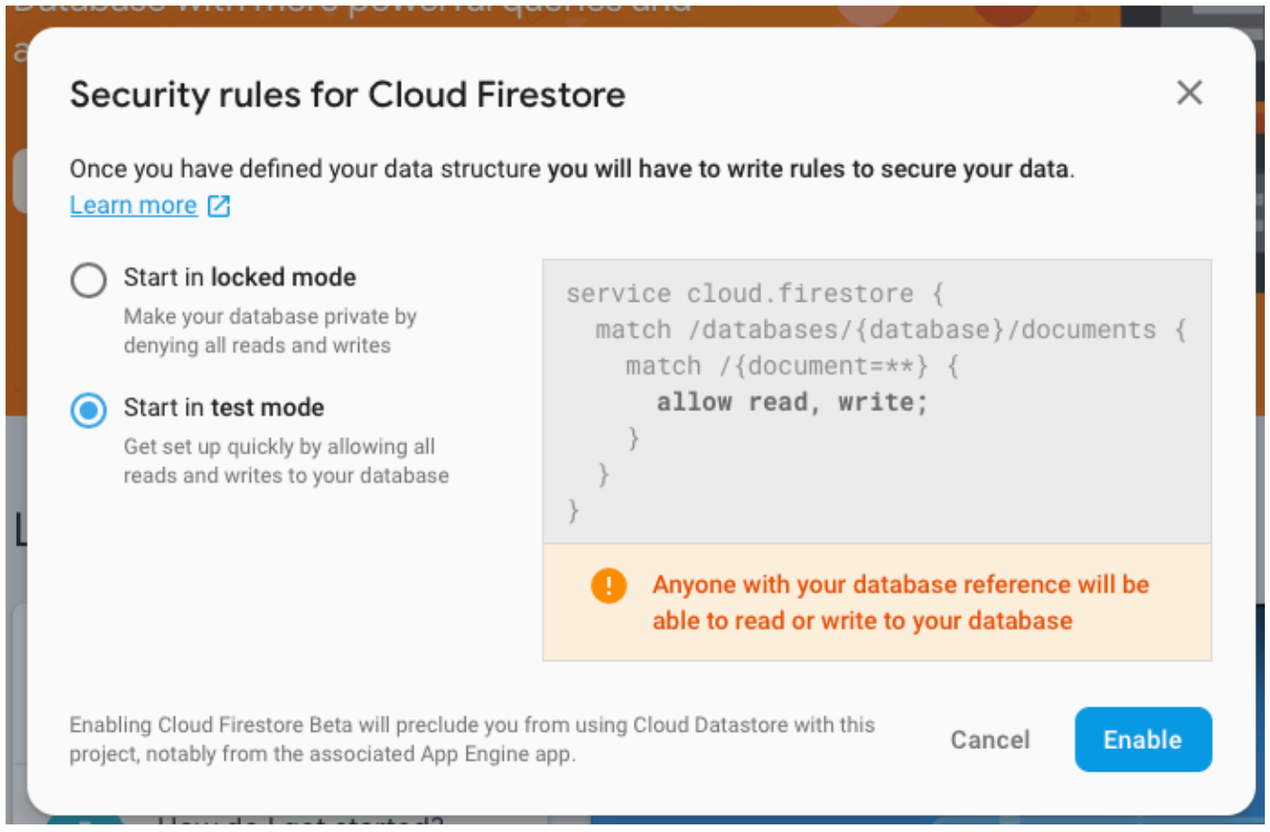
- Go to IAM & admin > Service accounts, create a new private key, save the file.

- Include Firebase in your web app

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

let serviceAccount = require('path/to/serviceAccountKey.json');

admin.initializeApp({
  credential: admin.credential.cert(serviceAccount)
});

let db = admin.firestore();
```

Permissions

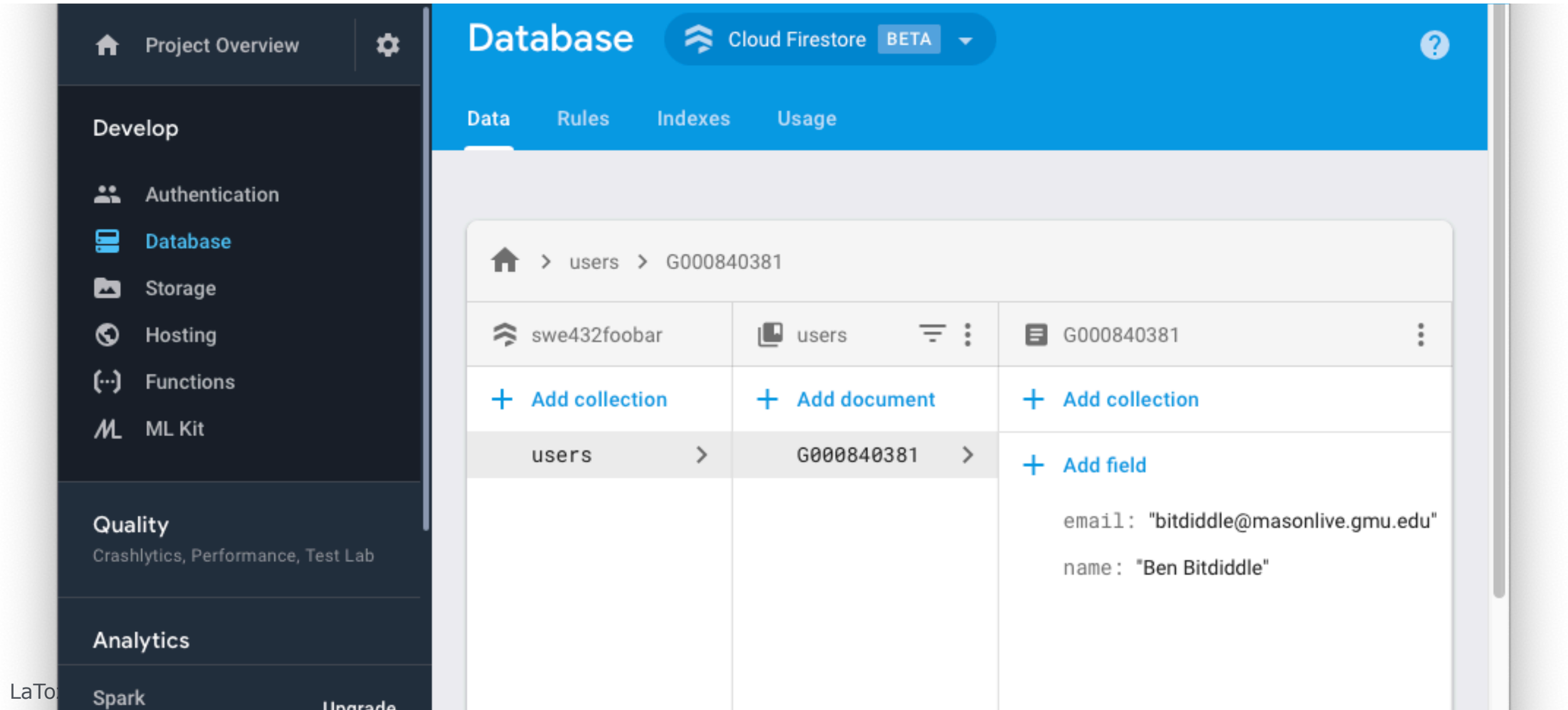
- “Test mode” - anyone who has your app can read/write all data in your database
  - Good for development, bad for real world
- “Locked mode” - do not allow everyone to read/write data
  - Best solution, but requires learning how to configure security



Firebase Console

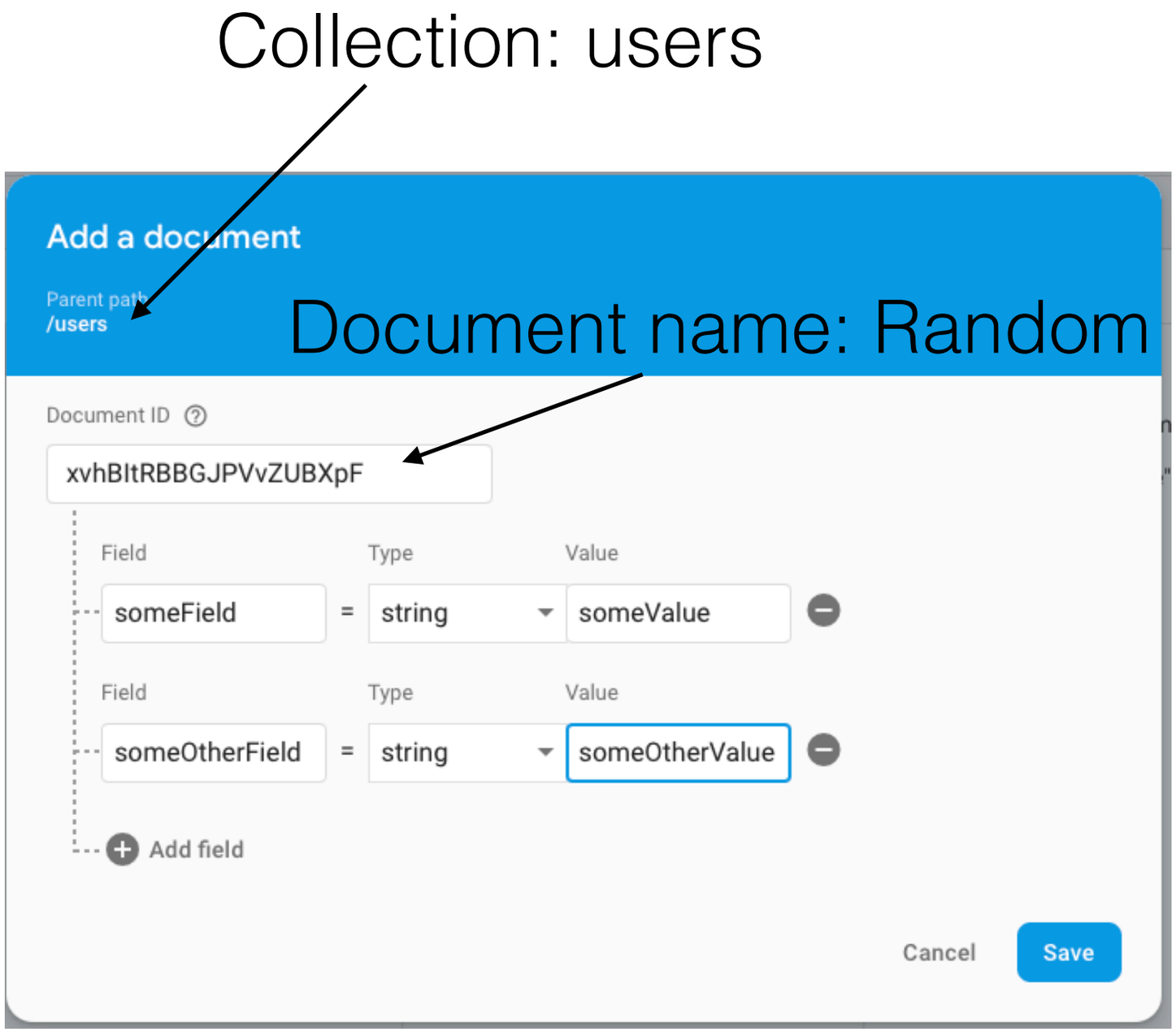
- See data values, updated in realtime
- Can edit data values

https://console.firebase.google.com

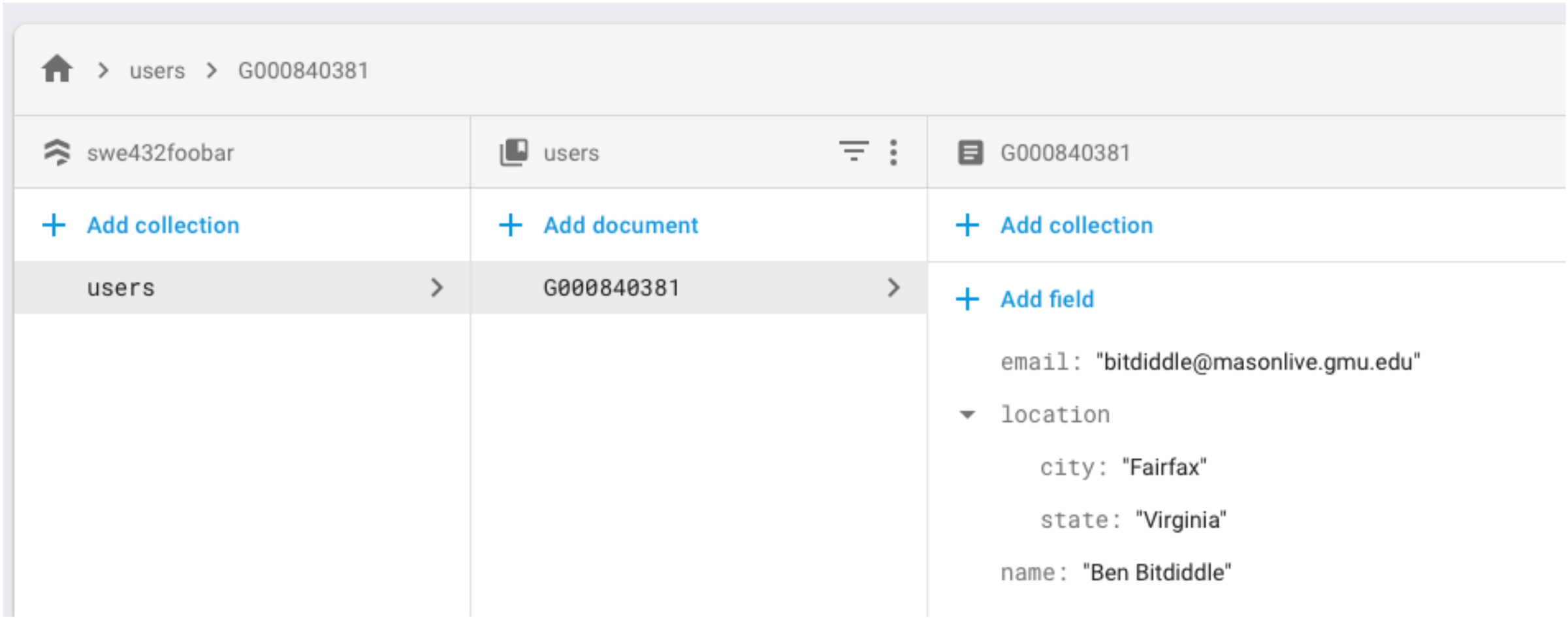


Firebase data model: JSON

- **Collections** of JSON documents
  - Hierarchic tree of key/value pairs
  - Can view as one big object
  - Or describe path to descendent and view descendent as object



JSON is JSON...


users > G000840381
swe432foobar
users
G000840381
+ Add collection
+ Add document
+ Add collection
users
G000840381
+ Add field
email: "bitdiddle@masonlive.gmu.edu"
location
city: "Fairfax"
state: "Virginia"
name: "Ben Bitdiddle"

Demo: Simple test program

- After successfully completing previous steps, should be able to replace config and run this script. Can test by viewing data on console.

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

let serviceAccount = require('[YOUR JSON FILE PATH HERE');

admin.initializeApp({
    credential: admin.credential.cert(serviceAccount)
});

let db = admin.firestore();

let docRef = db.collection('users').doc('alovelace');

let setAda = docRef.set({
    first: 'Ada',
    last: 'Lovelace',
    born: 1815
});
```

Structuring Data

- I want to build a chat app with a database
- App has chat rooms: each room has some users in it, and messages
- How should I store this data in Firebase? What are the collections and documents?

Structuring data

- Should be considering what types of records clients will be requesting.
    - Do not want to force client to download data that do not need.
- Better to think of structure as **lists** of data that clients will retrieve

Storing Data: Set

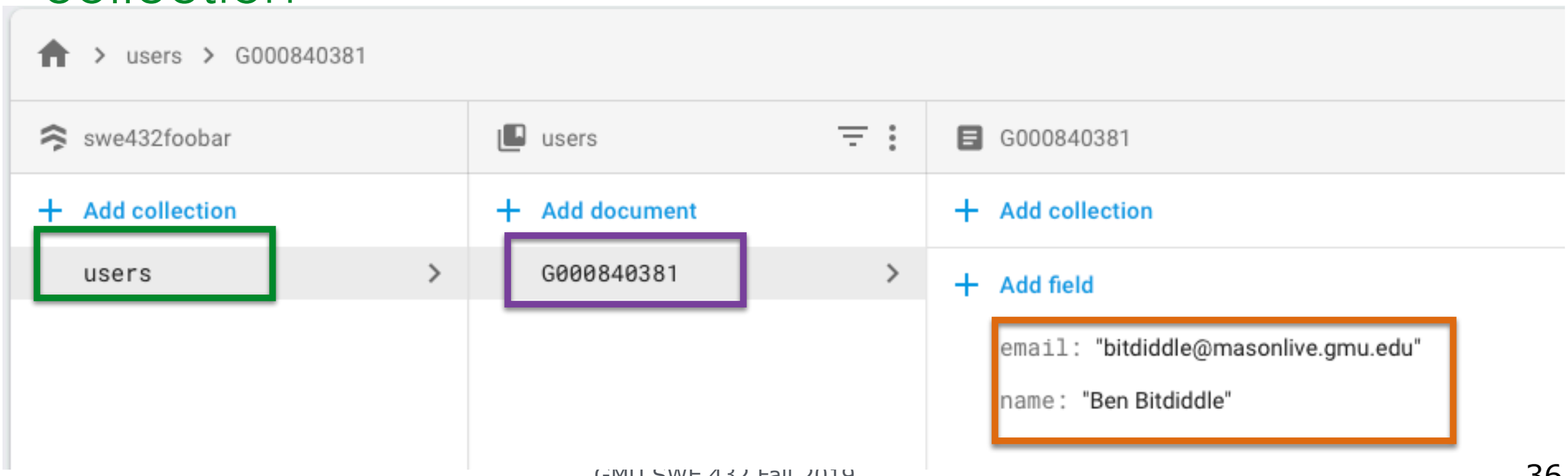
(because firebase is asynchronous)

```
async function writeUserData(userID, newName, newEmail) {
    return database.collection("users").doc(userID).set({
        name: newName,
        email: newEmail
    });
}
```

Create this one user by ID

Set the val

Get the users collection



Storing Data: Add

- Where does this ID come from?
  - It MUST be unique to the document
- Sometimes easier to let Firebase manage the IDs for you - it will create a new one uniquely automatically

```
async function addNewUser(newName, newEmail) {
    return database.collection("users").add({
        name: newName,
        email: newEmail
    });
}
async function demo(){
   let ref = await addNewUser("Foo Bar","fbar@gmu.edu")
    console.log("Added user ID " + ref.id)
}
```

Storing Data: Update

- Can either use “set” (with {merge:true}) or “update” to update an existing document (set will possibly create the document if it doesn’t exist)

```
database.collection("users").doc(userID).update({
    name: newName
});
```

Storing Data: Delete

```
database.collection("users").doc("ojtp4HrEeGB4Y9jErz0T").delete();
```

Removes a document

```
database.collection("users").doc(userID).update({
    name: firebase.firestore.FieldValue.delete()
});
```

Removes a field

- Can delete a key by setting value to null
  - If you want to store null, first need to convert value to something else (e.g., 0, '')

Fetching Data (One Time)

```
async function getUser(userId){
    return database.collection("users").doc(userId).get();
}
async function demo(){
   let user = await getUser("G000840381");
    console.log(user.data());
}
```

Can also call get directly on the collection

Listening to data changes

```
let doc = db.collection('cities').doc('SF');

let observer = doc.onSnapshot(docSnapshot => {
  console.log(`Received doc snapshot: ${docSnapshot}`);
  // ...
}, err => {
  console.log(`Encountered error: ${err}`);
});
```

“When values changes, invoke function”

Specify a subtree by creating a reference to a path. This listener will be called until you cancel it

- Read data by *listening* to changes to specific subtrees
- Events will be generated for initial values and then for each subsequent update

Ordering data

- Data is by, default, ordered by document ID in ascending order
  - e.g., numeric index IDs are ordered from 0…n
  - e.g., alphanumeric IDs are ordered in alphanumeric order
- Can get only first (or last) n elements

```
let firstThree = citiesRef.orderBy('name').limit(3);
```

- Can use where statements to query

```
citiesRef.where('population', '>', 2500000).orderBy('population');
```