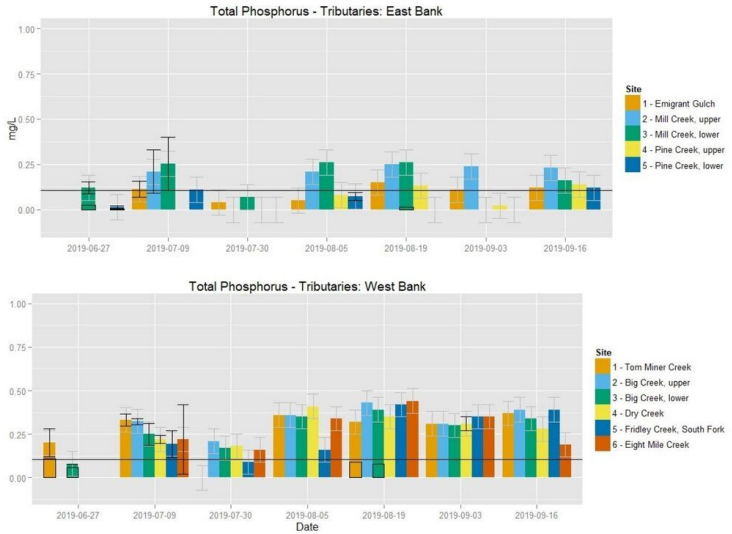
Rivernet 3
Software Development

William Roberts, Anna Jinneman, Jonah Hlastala

Our Goal for RiverNET

YELLOWSTONE ECOLOGICAL RESEARCH CENTER

We set out to create a web application for Yellowstone Ecological Research Center that would optimize and simplify the process of storing data collected from water samples taken from the Yellowstone River.



System Architecture

High Level Components

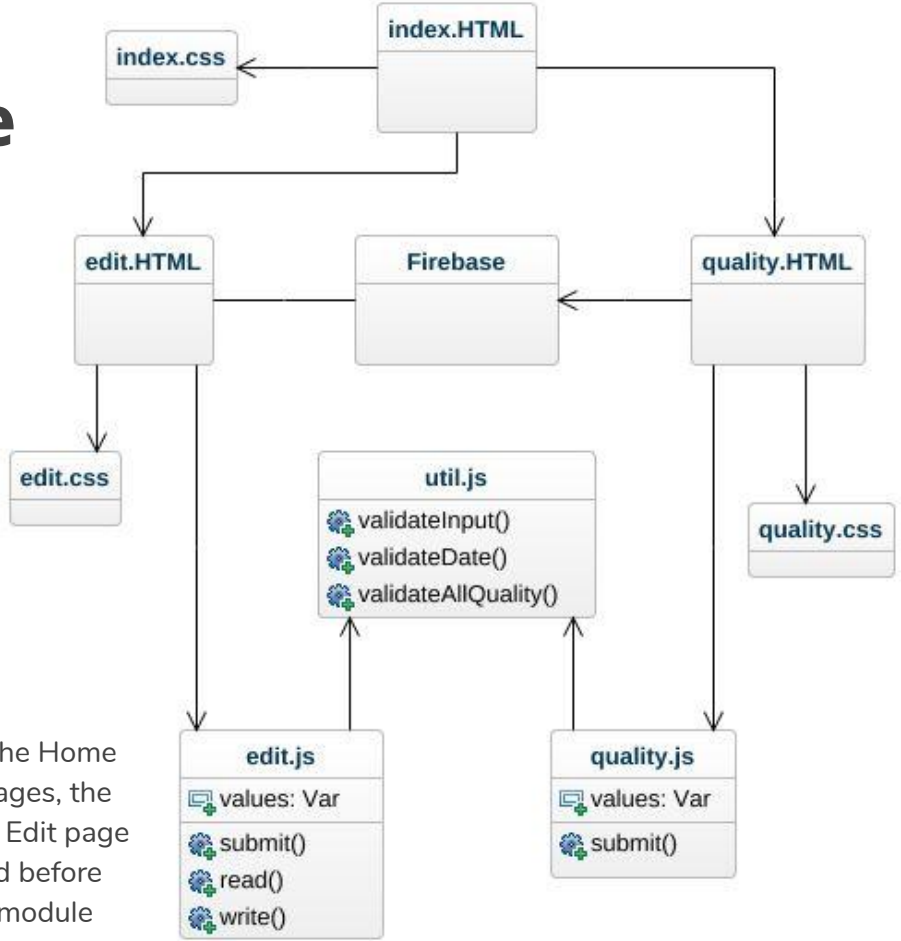
3 Main Pages

- Home Page (Index)
- Quality
- Edit

Each page has their own Javascript Module

- Util.js
- Quality.js
- Edit.js

There are 3 major pages used in our web application: the Home Page where you can navigate to the submit and edit pages, the Quality page where data can be entered by jar and the Edit page where recently submitted data can be found and edited before being sent away. Each of these pages has a javascript module associated with them that allows for their functionality.



System Architecture

High Level Components

Util.js:

- Validate Input
- Validate Date
- Validate All Quality Data

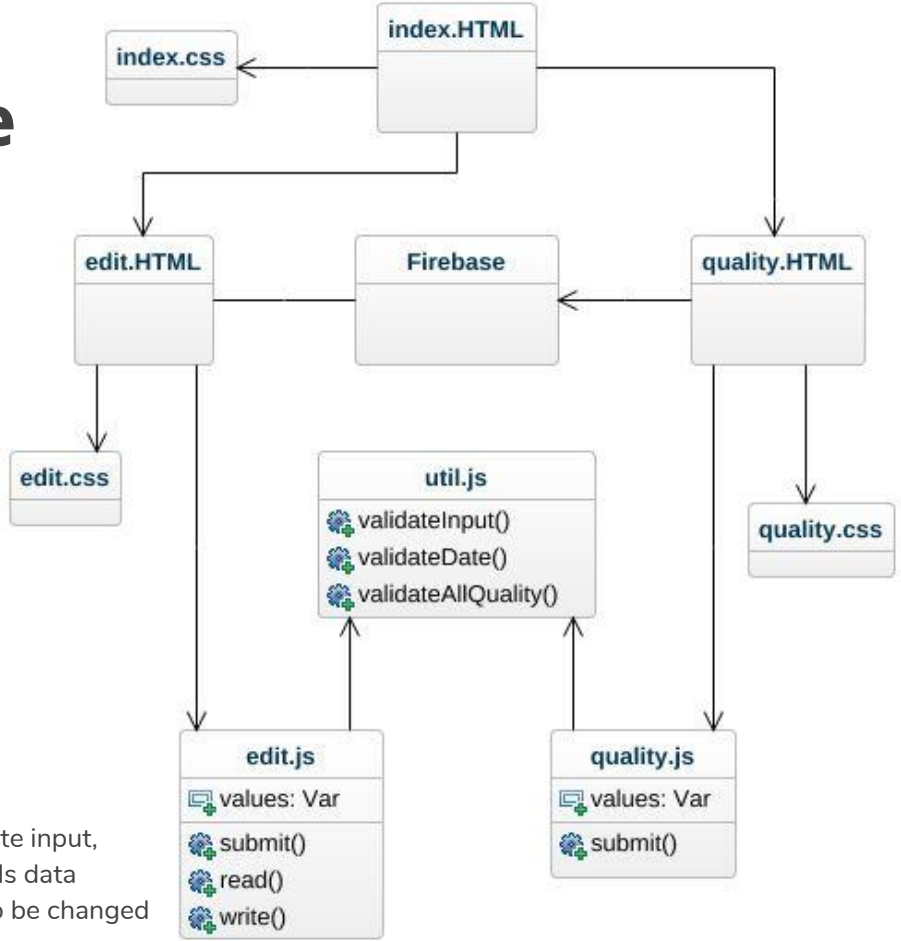
Quality.js:

- Submit Data

Edit.js:

- Submit Data
- Read Data
- Edit/Write

Util.js provides utility functions including functions to Validate input, date and all quality data, Quality.js is the module that controls data submission to our database and Edit.js allows for the data to be changed after being submitted to the database.



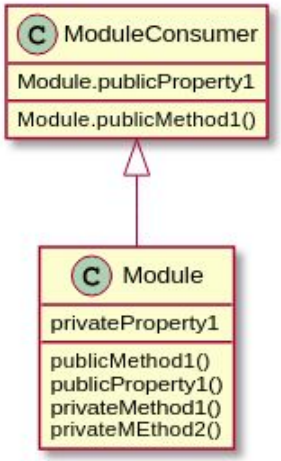
System Design

Design Patterns

The Module Design Pattern

This design pattern is the most commonly used of all JavaScript design patterns. It is focused on the public and private access to methods and variables and is commonly used to mimic classes in conventional software engineering. Further, its target is to reduce globally scoped variables in an effort to decrease the chances of collision with other code within an application. Some projects which make use of the module design pattern are jQuery, Dojo, ExtJS, and YUI.

We implement the Module design pattern in several places throughout our application. For example, edit.js and quality.js are the consumer modules for util.js.

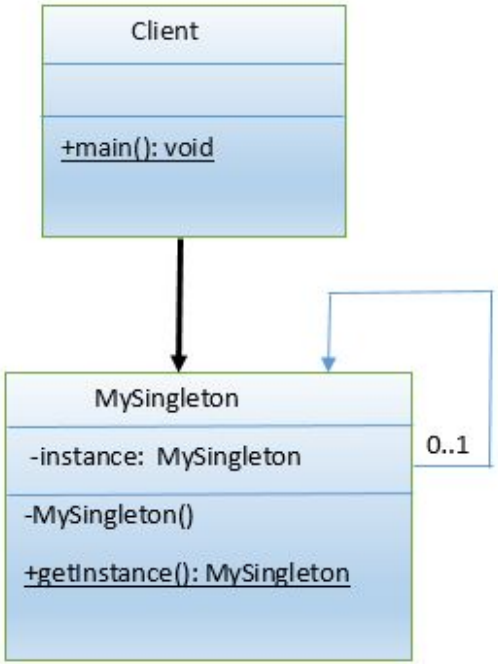


System Design

Design Patterns

The Singleton Design Pattern

The singleton design pattern is implemented with the Firebase App so we can initialize our database once and only once. Our Firebase load.js initializes the app, gets the database reference and then sends out references to the edit and quality modules.



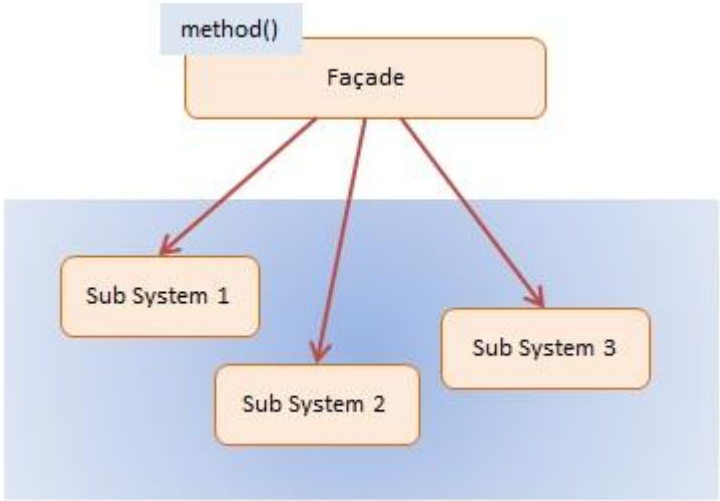
System Design

Design Patterns

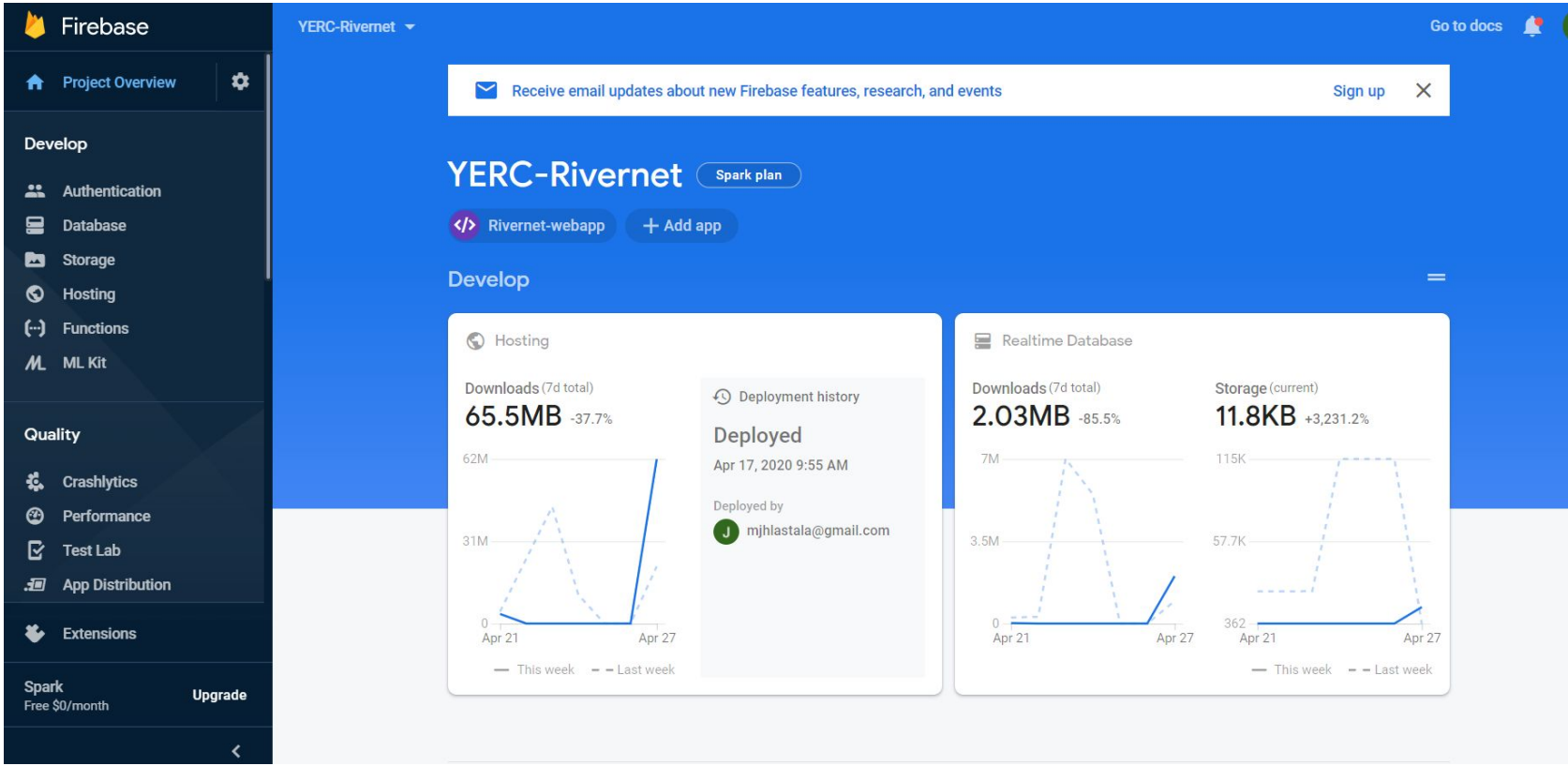
The Facade Design Pattern

The Façade pattern provides an interface which shields clients from complex functionality in one or more subsystems. It is a simple pattern that may seem trivial but it is powerful and extremely useful. It is often present in systems that are built around a multi-layer architecture.

The Facade design pattern is implemented with Firebase hosting the application so that users only see the front-end web application.

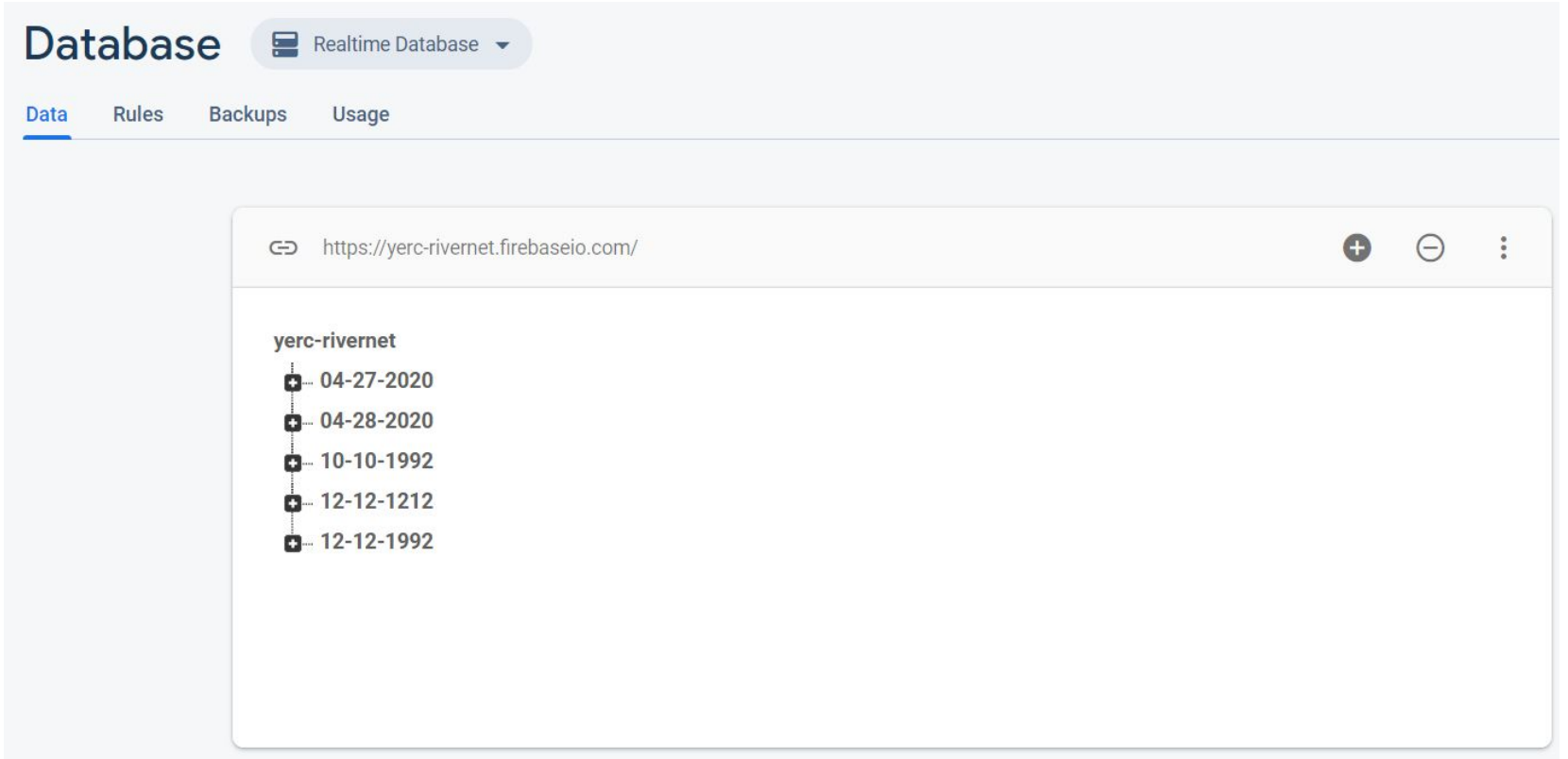


How to navigate Firebase



Database

The firebase database is a very simple and useful tool

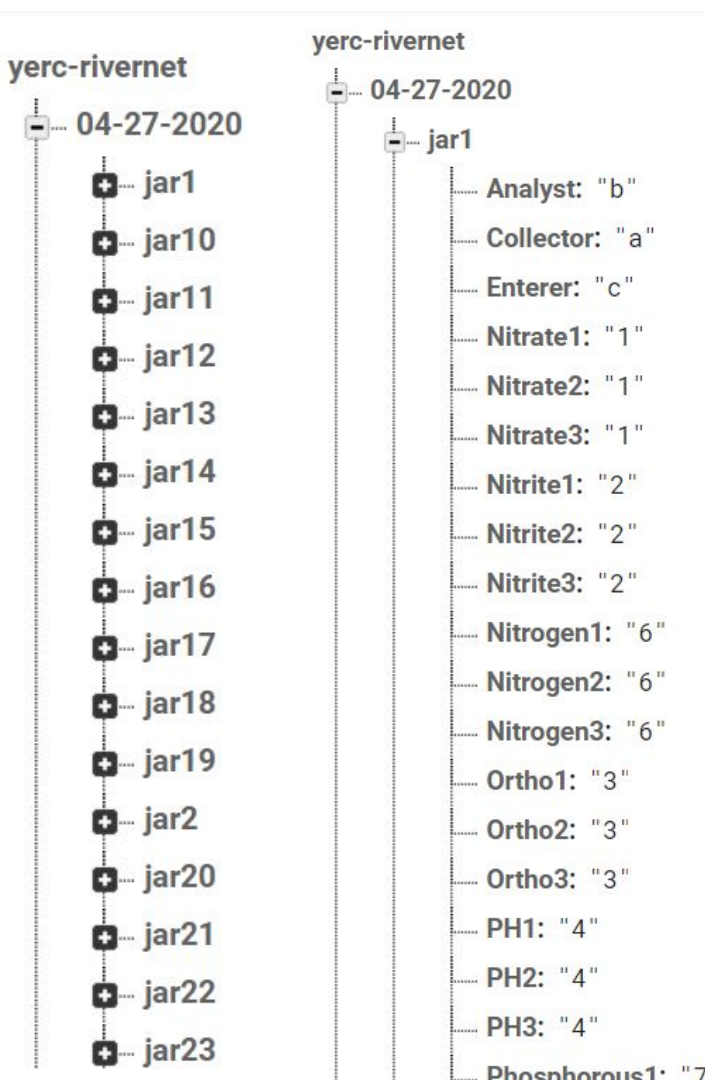


Navigating the database

Within each of the date nodes is a child

Each child represents one jar

Each jar contains all data and possible data



Reading and writing data

Reading and writing data to the database is pretty simple

To write:

```
var dataSubmit = getDatabaseReference().ref(vDate + "/" + "jar" + jar +"/");

//These three lines submit the information regarding the people who interacted with the data
dataSubmit.child("Collector").set(collector.value);
dataSubmit.child("Analyst").set(analyst.value);
dataSubmit.child("Enterer").set(enterer.value);
```

First, create a database reference to the location you want to write data too

With this reference you can you can submit new “children” to the database, and set the value of the children

Documentation: https://firebase.google.com/docs/database/web/read-and-write

Reading and writing data

Reading and writing data to the database is pretty simple

To read from a known location:

```
var currentJar = snapshot.child(vDate + "/jar" + jar).val();//Used for reading data

//The blocks of text below this are broken into categories for readability
//Each line assigns the value of the text box to the value associated in the databa
document.getElementById("collector").value = currentJar.Collector;
document.getElementById("analyst").value = currentJar.Analyst;
document.getElementById("enterer").value = currentJar.Enterer;
```

If you know exactly where you want to pull data from, then you can call upon a database reference to get the value within that reference

Documentation: https://firebase.google.com/docs/database/web/read-and-write

Reading and writing data

Reading and writing data to the database is pretty simple

Parsing a database:

```
//Used to access the data in the database at a specefic moment
return getDatabaseReference().ref().once('value').then(function(snapshot) {
```

To parse a database for specific data you can grab a ‘snapshot’ of the data

With this snapshot you can do a variety of operations

Snapshot was used in our program to check if a specific date existed in the database

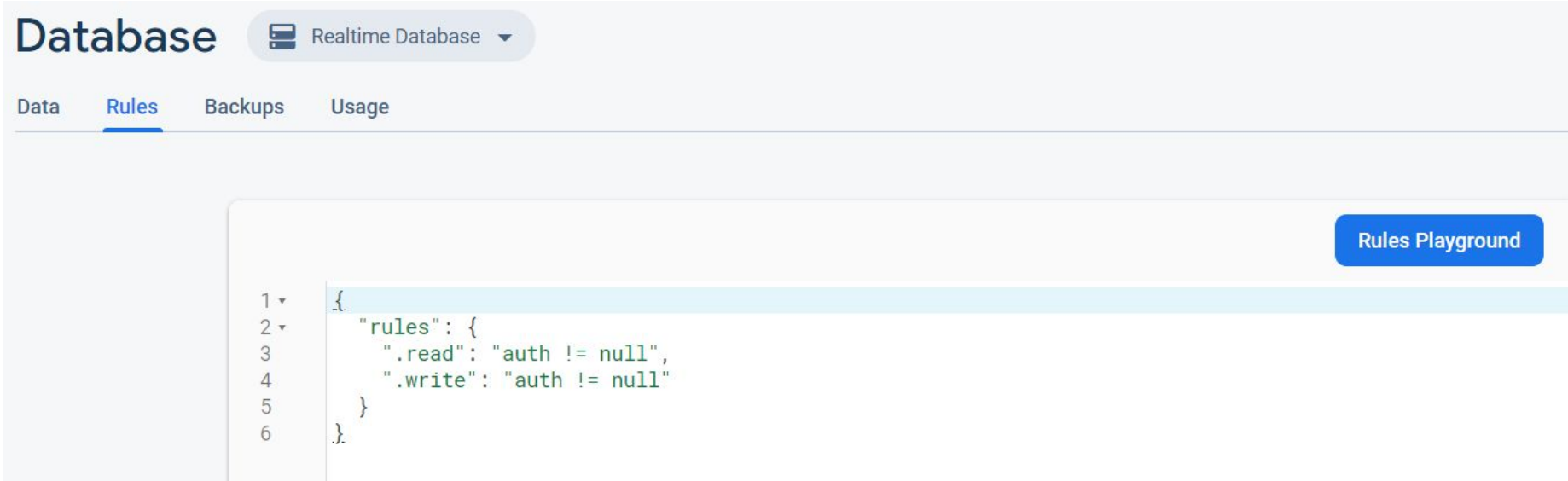
Documentation: https://firebase.google.com/docs/database/web/read-and-write

Database rules

In order to read and write from the database the user needs permission

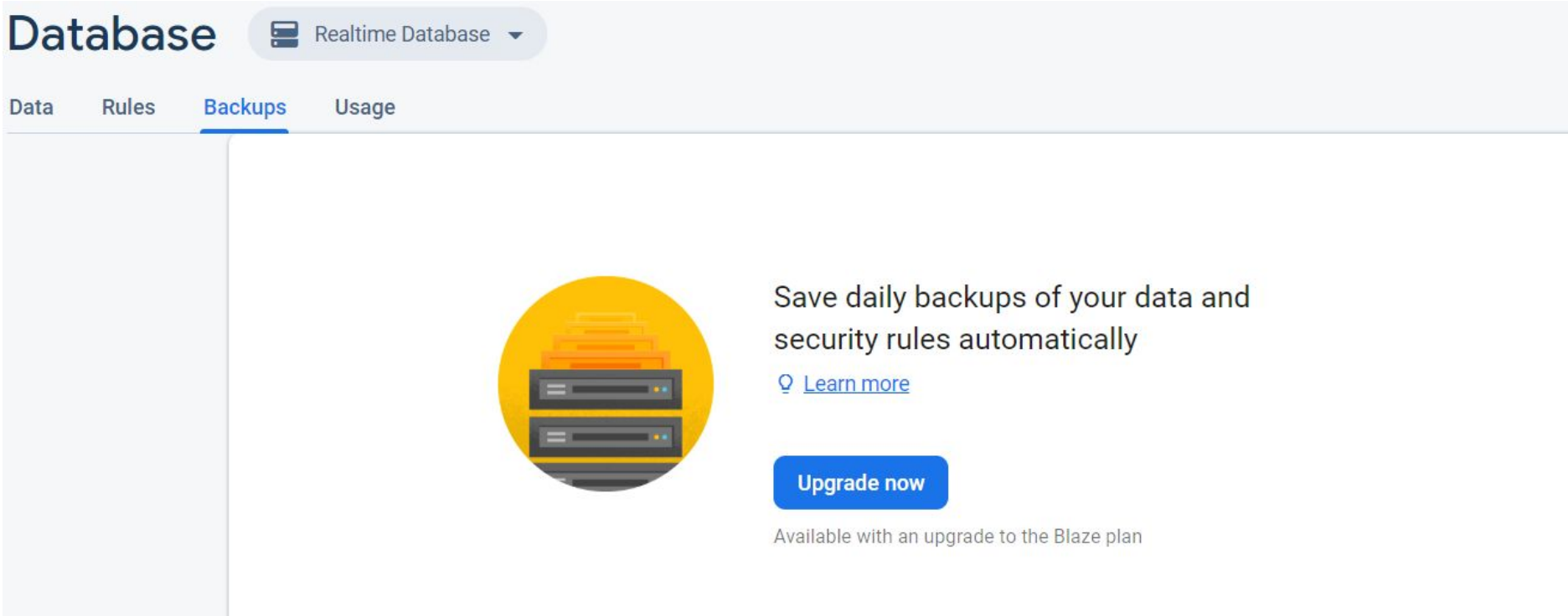
While developing, an open database is very useful

Rules can be changed and tested within firebase or externally and uploaded



Database backups

For extra safety, firebase allows you to backup your database if upgraded



Deploying to Firebase

Deploying to firebase allows the user to deploy from their local machine using PowerShell

The first steps involve initializing the project and identifying what features you want to use

```
PS C:\Users\jhlas\GitRepos\Rivernet> firebase init

     ######## #### ########  ######## ########     ###     ######  ########
     ##        ##  ##     ## ##       ##     ##  ##   ##  ##       ##
     ######    ##  ########  ######   ########  #########  ######  ######
     ##        ##  ##    ##  ##       ##     ## ##     ##       ## ##
     ##       #### ##     ## ######## ########  ##     ##  ######  ########

You're about to initialize a Firebase project in this directory:

  C:\Users\jhlas\GitRepos\Rivernet

? Are you ready to proceed? Yes
? Which Firebase CLI features do you want to set up for this folder? Press Space to select features, then Enter to confirm your choices
 (*) Database: Deploy Firebase Realtime Database Rules
 ( ) Firestore: Deploy rules and create indexes for Firestore
 ( ) Functions: Configure and deploy Cloud Functions
>(*) Hosting: Configure and deploy Firebase Hosting sites
 ( ) Storage: Deploy Cloud Storage security rules
 ( ) Emulators: Set up local emulators for Firebase features
```

Deploying to Firebase

Next you choose what project from your account you will be deploying to

Since we opted into use database, we will be prompted to name db rules, or use default

```
=== Project Setup

First, let's associate this project directory with a Firebase project.
You can create multiple project aliases by running firebase use --add,
but for now we'll just set up a default project.

? Please select an option: Use an existing project
? Select a default Firebase project for this directory: yerc-rivernet (YERC-Rivernet)
i  Using project yerc-rivernet (YERC-Rivernet)

=== Database Setup

Firebase Realtime Database Rules allow you to define how your data should be
structured and when your data can be read from and written to.

? What file should be used for Database Rules? (database.rules.json)
```

Deploying to Firebase

Since we opted hosting, we will be asked what folder will be uploaded when we deploy

We use dist because that folder contains the compiled project

Do no overwrite any of the previously created files

Firebase deploy will deploy the site after all of these steps are finished

```
=== Hosting Setup

Your public directory is the folder (relative to your project directory) that
will contain Hosting assets to be uploaded with firebase deploy. If you
have a build process for your assets, use your build's output directory.

? What do you want to use as your public directory? dist
? Configure as a single-page app (rewrite all urls to /index.html)? No
+  Wrote dist/404.html
? File dist/index.html already exists. Overwrite? No
i  Skipping write of dist/index.html

i  Writing configuration info to firebase.json...
i  Writing project information to .firebaserc...

+  Firebase initialization complete!
PS C:\Users\jhlas\GitRepos\Rivernet>
```

How it All Sticks Together

The Server

Our project was written with Node.js, a Javascript runtime environment for executing Javascript outside of web browsers.

Node.js offered a powerful way to work on our webapp before pushing it to the hosted server, but that meant we had to learn server hosting.

This next section goes over how we “compiled” our webapp into a full, cohesive product using Node.js, Express.js, and Webpack.

What is Node.js?

Node.js is a platform built on Chrome's JavaScript runtime for easily building fast and scalable network applications. It contains the framework to allow for unlimited extensions, which we can talk about next.

It uses an event-driven model that makes it lightweight and efficient, "perfect for data-intensive real-time applications that run across distributed devices."

```
module.exports = {
  entry: {
        main: './src/index.js',
        edit: './src/js/edit.js',
        quality: './src/js/quality.js'
  },
  output: {
    path: path.join(__dirname, 'dist'),
    publicPath: '/',
    filename: '[name].js'
    },
  mode: 'development',
  target: 'web',
```

Why we used Node.js

We used Node.js for our front-end, but it powerfully allowed us to glue the back-end and front-end together in an efficient and expandable way. Node.js allows us to work locally on our machine (with localhost) without pushing every iteration to our hosting service!

```
"scripts": {
  "build": "rm -rf dist && webpack --mode development --config webpack.server.config.js && webpack --mode development",
  "start": "node ./dist/server.js",
  "test": "jshint src/js/edit.js; jshint src/js/quality.js; jshint src/server/server.js; jshint src/js/util.js; jest --forceExit"
},
```

However, this model was only a single step in the right direction of designing our webapp, we still needed to solve the problem of developing the app!

```
const PORT = process.env.PORT || 8080;
app.listen(PORT, () => {
    console.log(`App listening to ${PORT}....`);
    console.log('Press Ctrl+C to quit.');
});
```

Developing a Webapp: Express

Express is a minimal and flexible Node.js web app server framework that provides developers a robust set of features for web development.

Express dictates the entirety of our web app, mostly from a sense of connecting the separate pages and modules into a functioning system.

```
const app = express(),
    DIST_DIR = __dirname,
    HTML_FILE = path.join(DIST_DIR, 'index.html'),
    compiler = webpack(config);

const router = express.Router();
```

The app variable is referenced constantly through our system, and follows another simple case of the singleton design pattern, we only want one app!

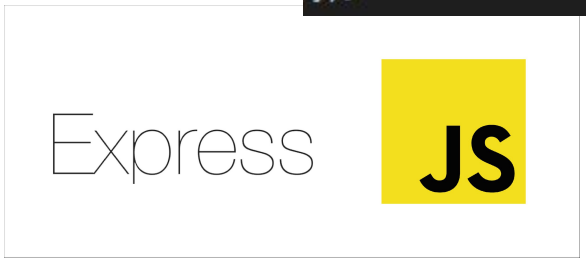
express() automatically ensures that there is only one app at any time.

Developing a Webapp: Express

Express is the de facto standard server framework.

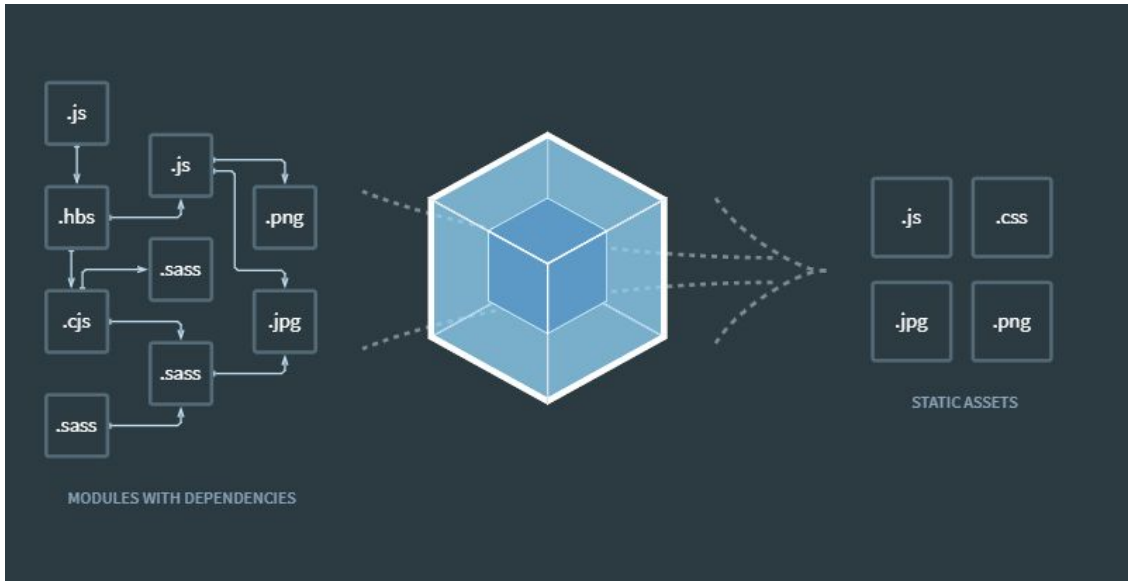
Express fulfills most of our back-end, alongside Firebase.

```
//Here, we see what page the user is on, and depending on the url extension, send the correct html.
app.get('*', (req, res, next) => {
    compiler.outputFileSystem.readFile(HTML_FILE, (err, result) => {
        if (err) {
            return next(err);
        }
        res.set('content-type', 'text/html');
        res.send(result);
        res.end();
    });
    //res.sendFile(HTML_FILE);
});
```



Express and Node together manages our server, but that server needs to be able to actually digest all our modules. How do we pack all of our source code into a simple package that the browser can read, especially when we're using ECMAScript 6 (ES6)?

How we Pack our Code



Creating HTML/css filled with images, scripts, and other large files is a very costly process (and can be quite slow in-browser). We need to bundle our code so the browser can digest it far quicker!

Webpack middleware handles this issue very efficiently, quickly bundling images, scripts, styles, etc in a way that is expandable and powerful.

webpack

How we Pack our Code

```
//app.use(express.static(DIST_DIR));
app.use(webpackDevMiddleware(compiler, {
    publicPath: config.output.publicPath
}));
```

Rather than simply using the files as-is, webpack bundles the code into reusable modules from our js, images, and css.

```
compiler.outputFileSystem.readFile(path.join(DIST_DIR, '/quality.html'), (err, result) => {
    if (err) {
        return next(err);
    }
    res.set('content-type', 'text/html');
    res.send(result);
    res.end();
});
```

Webpack “compiles” our code! It runs Babel to convert ES6 to ES5, meaning that without webpack, we couldn’t use the new standard version of Javascript.

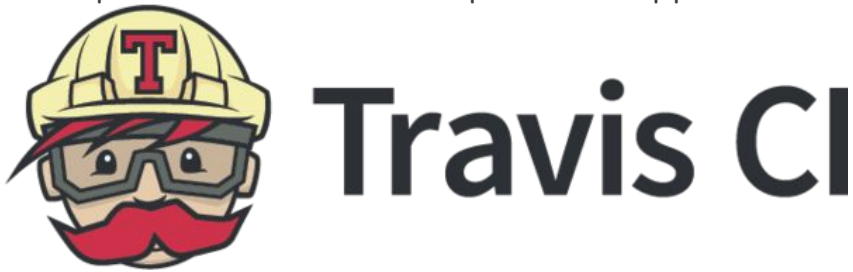
webpack

Quality Assurance

So we have a complete web app, capable of being displayed in the browser and extended upon! But how do we know it works? What were the other steps we took in getting here?

As a rule, our team focused on Continuous Integration, and what better tools to enable us to do so than Github and Travis? Through CI we worked tirelessly to test each and every stage of development, and Jest Testing Framework helped us do that.

The next few slides will go over how we used these powerful tools to develop our web app.



Our Software Development Platform

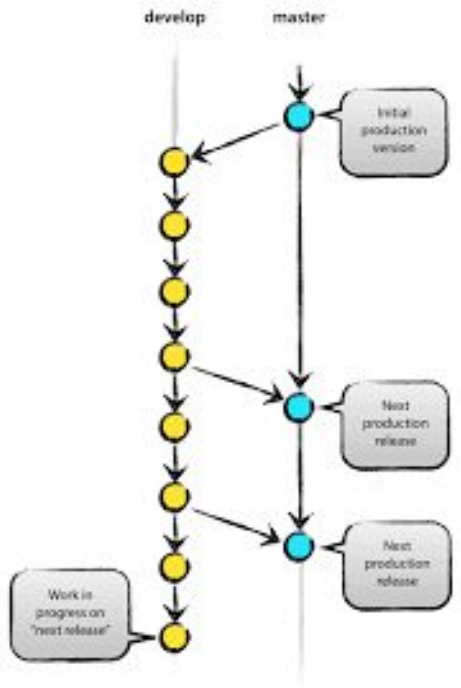
GitHub

Our first choice in repository hosting service (and version control), Github never let our team down as we continuously pushed our nearly 300 commits up into the web!

Github allowed us to collaborate quickly and efficiently across the team, and provided a very efficient means of pooling our efforts together.

We used the branch-merge style of handling our remote repository, meaning we always had at least a “master” and a “dev” branch, and everyone pushed their changes to the “dev” branch.

At the end of each sprint, dev was merged with master!

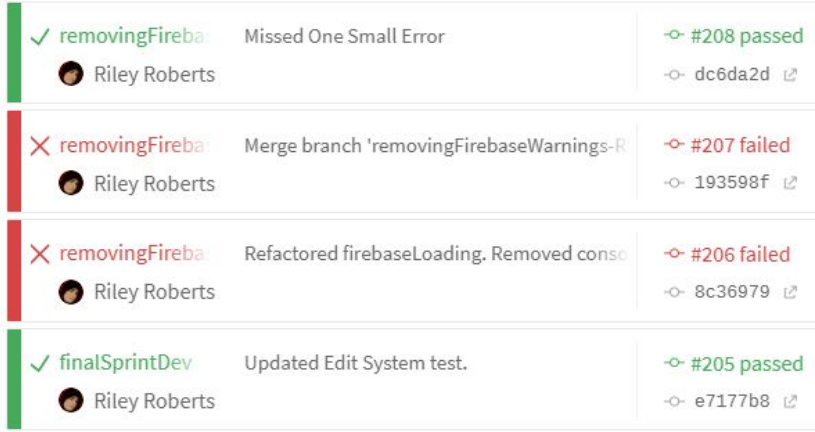


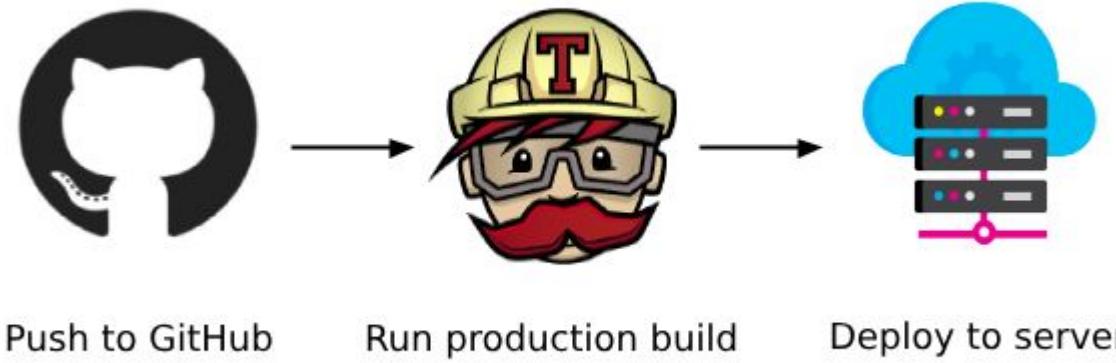
Continuous Integration

Travis CI was an obvious choice because how effectively it integrates with Github.

Travis automatically detects when there is a new commit pushed to our repository and immediately begins a build on it's virtual machines.

```
language: node_js
node_js:
  - "node"
dist: trusty
addons:
  chrome: stable
before_install:
  - google-chrome-stable --headl
before_script:
  #- npm run build; npm start
notifications:
```





Our Testing

```
----------|---------|----------|---------|---------|-------------------
File      | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
----------|---------|----------|---------|---------|-------------------
All files |     100 |      100 |     100 |     100 |
 util.js  |     100 |      100 |     100 |     100 |
----------|---------|----------|---------|---------|-------------------
Test Suites: 1 passed, 1 total
Tests:       18 passed, 18 total
Snapshots:   0 total
Time:        38.559s
Ran all test suites.
```

We chose Jest because it is another chosen standard within Javascript programming. It is one of the easiest packages to set up, and offers power and efficiency in it's tests!
Testing for coverage is as easy as adding the --coverage tag.

Tests were created using Jest, a powerful and light-weight testing framework.

We created unit tests using a simple black-box method, testing strictly input/output of the function in question. Several functions required integration tests to ensure that function integration was handled correctly.

Other functions (within other files) were too complicated for unit/integration tests, so they were tested with several System tests.

Time to Demo

Enough talking! Time to show you what we created.

Git graph of the last sprint (right to left)