Web Controlled Smart Drainage and Sub Irrigation Control System

DEC 1602 Client: Agri Drain CORPORATION Advisor: Nicola Bowler

Rodney Barto, Griffen Clark, Anne Ore, Michael Parker, Adam Wolter

Introduction

Problem:

Farmers want to be able to remotely observe water irrigation data from their fields, visually see the location of their irrigation devices, and be notified of unexpected changes in sensor data.

Solution:

An ABE capstone group will create the hardware for devices. Our team will create a web interface. Agri Drain will create a mechanism for communicating between the two.

Design Requirements

The intended users are admins (Agri Drain employees) and regular users (farmers).

Functional requirements (users):

- ➔ Read & update device settings
- ➔ Current & historical device sensor data
- ➔ Interactive map
- ➔ Alert users when an alarm is triggered

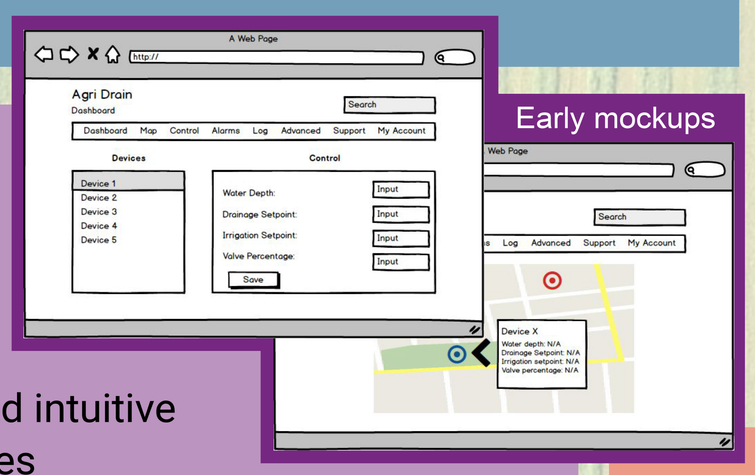
Functional requirements (admins):

- ➔ Add new users; edit user information
- ➔ Add & display users' devices
- ➔ Modify settings of any device

Non-functional requirements:

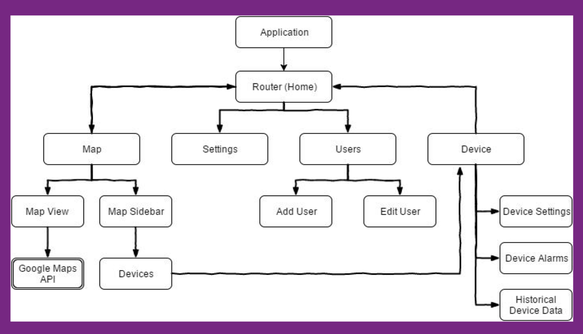
- ➔ Visually appealing & intuitive on desktop and mobile
- ➔ Documentation for future developers

Design Approach

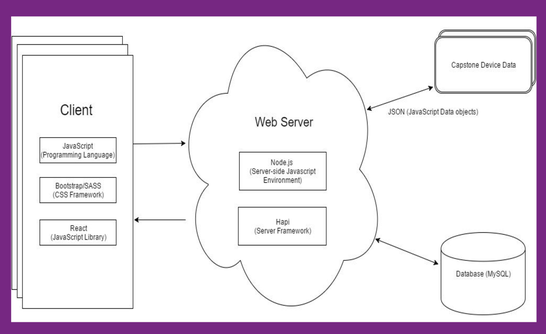


Front-End User Interface

- ➔ Focus on being user friendly and intuitive
- ➔ Navbar visible to user at all times
- ➔ Each link in the Navbar is a different module in code
- ➔ Functionality is highly modularized (see block diagram below)



Front-End Block Diagram



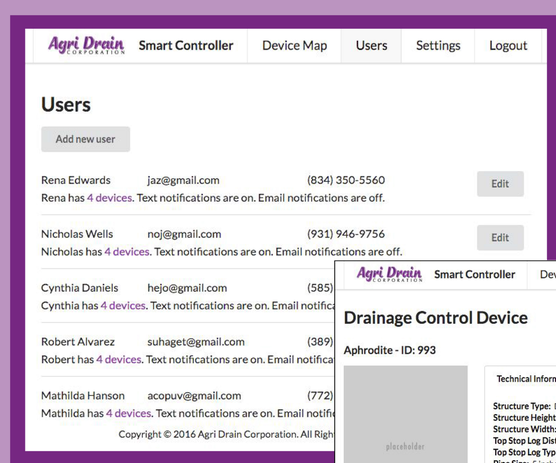
Entire Project Architecture

Back-End

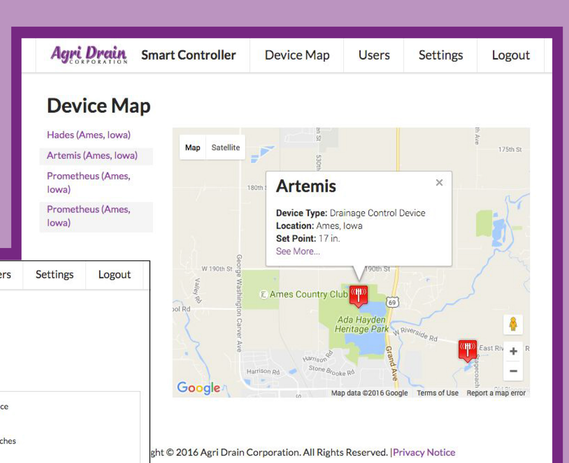
- ➔ Using Express.js (server side framework) to read & write to a MySQL database
- ➔ Serves as intermediary between the front-end client and the database

Modular Design

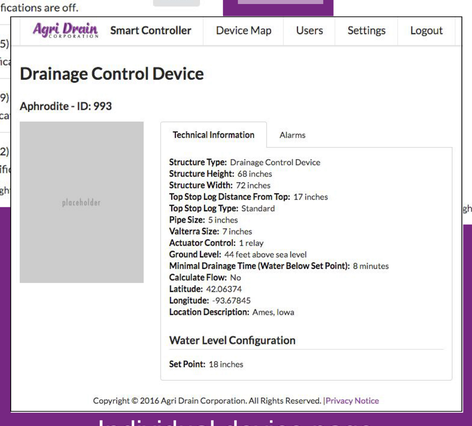
- ➔ Map page shows the visual location of each device (implemented with Google Maps API), and allows users to select devices
- ➔ Individual device pages show information and settable characteristics (type, dimensions, pipe size, water level, flow rate, alarms, irrigation setpoint)
- ➔ Users page allows admins to view, create & edit user accounts



Users page



Map page



Individual device page

Technical Details

Technologies Used:

Node.js - Server-side JavaScript runtime environment
Express.js - Web application framework
React - Design library for creating modular user interfaces
Mocha/Chai - Frameworks are for testing our components
Istanbul.js - JavaScript code coverage tool
Semantic UI - UI framework for responsive front end web pages
SASS - CSS extension language
Sinon - Standalone test stubs and mocks
Webpack - Module bundler to consolidate client and server files
MySQL - Database hosted through AWS RDS
WebStorm - JavaScript integrated development environment
GitHub - Online project repository hosting and version control
CircleCI - Continuous integration and delivery platform

Our project uses a very modular based design with individual JavaScript files for each component (e.g. header, footer, map, etc..)

Testing

All files
97.86% Statements 137/140 83.33% Branches 20/24 98.72% Functions 77/78 97.58% Lines 121/124

File	Statements		Branches		Functions		Lines	
client	100%	26/26	100%	2/2	100%	20/20	100%	20/20
client/components	100%	25/25	50%	2/4	100%	17/17	100%	21/21
client/pages	100%	25/25	100%	4/4	100%	14/14	100%	23/23
client/pages/device	100%	19/19	100%	6/6	100%	10/10	100%	17/17
client/pages/map	93.02%	40/43	75%	6/8	94.12%	16/17	92.68%	38/41
server/controllers	100%	2/2	100%	0/0	100%	0/0	100%	2/2

- ➔ Unit testing is provided through Mocha/Chai frameworks
- ➔ Allows each individual module to be thoroughly tested in a easy to understand structure
- ➔ Each component must be thoroughly tested so that if a single line changes unintentionally, it causes tests to fail
- ➔ Our code test coverage is proven by Istanbul.js (shown above)
  - ◆ Istanbul.js provides statement/branch/function coverage
  - ◆ Istanbul.js shows exactly what is and what isn't tested
- ➔ CircleCI enforces that all code is tested and verified prior to being merged into the main production branch