Memo

To:	Professor Pisano
From:	SmartLoo (Arturo Asmal, Ben Corn, Bonnie To, Brandon Ng)
Team:	SmartLoo: Team 20
Date:	4/29/18
Subject:	SmartLoo - Resource CD Software Report

1.0 SmartLoo Cloud

1.1 ASP.NET Core 2.0 Framework

1.1.1 Dependencies from NuGet

The following dependencies can be installed from Visual Studio Nuget Package Manager. To ensure successful installation and compiling of the project, please ensure the correct versions listed below are selected during setup. When compiling the project from the repository, the following dependencies will be retrieved and installed automatically for you. This information is simply provided for documentation.

- Microsoft.AspNetCore.All 2.0.0
- Microsoft.Azure.DocumentDB.Core 1.7.0
- Microsoft.EntityFrameworkCore.Design 2.0.2
- MongoDB.Driver 2.5.0

1.1.2 Loo/API/Sensors.cs

Sensors.cs is the primary API controller, inheriting from the Core 2.0 controller class. This class defines all of the API endpoints the cloud app and hardware devices use.

[GET api/clients]
Retrieves all client names in the MongoDB database, without duplicates, and returns them as a JSON array. All client names are represented as the login email provided upon initial sign up.

[GET api/sensor]
Retrieves a sensor document stored in MongoDB by querying the 8 digit sensor ID provided in the request. Returns a single document (if found) as JSON object.

[GET api/buildings]
Retrieves all building names in the MongoDB database, without duplicates, and returns them as a JSON array. The email of the user currently logged in making the request is used as a query filter to obtain only the building names for that client (e.g. query filter would be johndoe@gmail.com)

[GET api/restrooms]
Retrieves all restroom names in the MongoDB database, without duplicates, and returns them as a JSON array. The email of the user currently logged in making the request is used as a query filter in combination with a request query parameter specifying the building name to retrieve restroom names from (e.g. query filter would be johndoe@gmail.com *and* Studio C as the building name)

[GET api/sensors]
Retrieves all sensor documents in the MongoDB database matching a provided building name in the request. Utilizes the email of the user logged in making the request. (e.g. query filter would be johndoe@gmail.com *and* Studio C as the building name). Return result would be a JSON array of sensor objects.

[GET api/bridge]
Retrieves a sensor document matching a provided 32-bit SmartLoo Bridge ID. This endpoint is used to retrieve location data when setting up a new sensor. When a sensor is turned on and begins transmitting, the Bridge ID is attached to all requests made to the SmartLoo API. During the setup process for a sensor, we are able to query the database and look for any other sensors that have already been setup with the same Bridge ID. This allows the user to bypass the process of entering location data (such as building name and restroom name). This information is extracted from the sensor document and used for the new sensor. If no matching documents are found, [] is returned and handled by the AngularJS controller.

[**GET api/sensors]**
Retrieves sensor documents by querying the building name, restroom name, and client name. Sensor documents are returned as a JSON array of sensor objects. (e.g. Studio C, 1st Floor Women's, johndoe@gmail.com)

[POST api/sensor]
Updates a sensor document specified by the unique 8 digit identifier. The payload is a **SensorUpdate** object, containing the Sensor ID and the updated sensor value. The MongoDB database is queried first to find the sensor document. If one doesn't exist, a new one is created with the Sensor ID in the payload object. The raw sensor value is used in combination with the sensor calibration parameters (**CInitialDist, CMinDist**) to calculate the corrected sensor value to be used on the front end. All measurements are provided in centimeters by the Ultrasonic sensor. The first letter of the 8-digit sensor ID is used to identify it as a **S**oap, Waste **R**eceptacle, or **P**aper sensor. The value update process is the same for paper towel and toilet paper and are not separated. When the document has been updated, the document is replaced in MongoDB and a history item is placed in the History collection in the MongoDB. A History item contains the Sensor ID, Sensor Value, and a TimeStamp.

[POST api/add_accessory]
Adds sensor data input during the sensor setup process, including the Sensor Name, restroom location and building name (if applicable) and the client name. The payload should be a **Sensor** object, in JSON format.

[GET api/validate]
Checks the MongoDB for a sensor matching the SensorID (accessory code) passed as a request parameter. This endpoint is used during the Sensor Setup Process from the cloud application to determine if the Sensor ID input by the user during setup is valid and in the MongoDB. When a sensor is turned on, it should immediately begin transmitting data to the API regardless of if it has been setup. So during setup, the API will validate whether the sensor is in fact transmitting successfully / also check to see if the accessory code is in the correct format.

1.1.3 Loo/Controllers/AccountController.cs
AccountController handles all login and registration logic.

[POST /login]
Utilizes ASP.NET Identity to authenticate a user with the email address and password provided in the login form.

[POST /register]
Creates an AuthenticatedUser object with the email, first and last name, and phone number passed during registration. Additionally, it marks the

NotificationOptIn parameter as true/false if the user request SMS notification to be enabled for critical alerts.

1.1.4 Loo/Controllers/HomeController.cs
HomeController handles routing logic for pages. Returns the HTML view associated with each route in the controller, including Index (“/”) and Restrooms (“/restrooms”).

1.1.5 Loo/Models/AuthenticatedUser.cs
Inherits from IdentifyUser, contains properties associated with a SmartLoo registered user.

1.1.6 Loo/Models/Client.cs
Contains classes that represent the Sensor, Sensor History, and Building Address (property of the Sensor).

1.1.7 Loo/Models/RegisterViewModel.cs
Contains classes that represent a new user registrant and a user logging in.

1.1.8 Loo/Views/Account/Logincshtml
HTML view served when routed to Login page.

1.1.9 Loo/Views/Account/Registration.cshtml
HTML view served when routed to registration page.

1.1.10 Loo/Views/Home/Restrooms.cshtml
HTML view served when routed to the main page of the SmartLoo Cloud application. Displays sensor information.

1.1.11 Loo/Views/Home/Schedule.cshtml
HTML view served when routed to the Scheduling page of the SmartLoo Cloud. Displays a list of restrooms that will require servicing for the day based on previous results.

1.1.12 Loo/Views/Shared/Layout.cshtml
HTML partial view combining header HTML and injecting the primary page content.

1.1.13 Loo/Views/Account/Login.cshtml
HTML view served when routed to Login page.

1.1.14 Loo/wwwroot/js/angular/registrationController.js

AngularJS controller handling front-end logic during registration. Contains error check and validation to ensure email addresses match, password meetings requirements, and no fields are empty. Utilizes the $http service injected into the Angular app to make a POST request to the SmartLoo API with the JSON registration payload.

1.1.15 Loo/wwwroot/js/angular/restroomController.js

AngularJS controller handling front-end logic for the core pages of the SmartLoo Cloud application, including making requests for fetching building, restroom, and sensor data, updating sensor data every 5s, and handling the logic for adding a new sensor. This controller also contains logic for converting sensor values from the database to meaningful results for the progress bar.

1.1.16 Loo/wwwroot/js/angular/scheduleController.js

AngularJS controller handling front-end logic for the scheduling portion of the SmartLoo application. Makes requests to fetch sensor data and correlates the sensor values from the database to be implemented via the calendar view for easy access and convenient viewing.

1.1.17 Loo/wwwroot/css/loo.css

Styling for the Index, Restroom, and Scheduling pages of the SmartLoo Cloud Application.

1.1.18 Loo/wwwroot/css/schedule.css

Styling for the Scheduling page of the SmartLoo Cloud Application.

1.1.19 Loo/wwwroot/css/registration.css

Styling for the Login/Registration pages of the SmartLoo Cloud Application.

1.1.20 Loo/wwwroot/css/site.css

Styling for overall layout of SmartLoo Cloud application, including styles for the *html* and *body* tags.

1.2 jQuery 3.2.1

Dependency for AngularJS, Bootstrap, and many of the core functionalities of the front-end code for the SmartLoo Cloud application.

1.3 Bootstrap 4.0.0

Used throughout the SmartLoo Cloud application front-end components for styling. Our custom styling css files are listed above.

1.4 AngularJS 1.6.6

Provides data-binding for the front-end, allowing for a dynamic interface to be created without manually updating the DOM. Javascript based.

1.5 FontAwesome 5.0.6

Icon library used throughout SmartLoo Cloud application.

2.0 SmartLoo Database

2.1 MongoDB Atlas

Our MongoDB database was setup on Mongo's DaaS platform, Atlas with an M2 size instance. We have two collections, **Sensors** and **SensorHistory**. The Sensor collection contains documents for each sensor that has been connected to a SmartLoo Bridge. The SensorHistory collection contains documents with historical sensor data for each time a sensor sent an updated sensor value. This data is used for the scheduling page to provide a "heat-map" style analysis of the data to determine when restrooms are busy.

Example documents from both MongoDB Collections listed below.

```
{
    "_id" : ObjectId("5a8514d054cdbb6af06512ad"),
    "ClientName" : "bcorn@bu.edu",
    "ClientId" : "bcorn@bu.edu",
    "BuildingName" : "Photonics Center",
    "BuildingCode" : "PHO",
    "BuildingAddress" : {
        "BuildingStreet" : "15 St. Mary's St.",
        "BuildingCity" : "Boston",
        "BuildingState" : "MA",
        "BuildingZip" : "02215"
    },
    "SensorName" : "Trash Can Near Door",
    "SensorId" : "WI4209NJ",
    "BridgeId" : "c19c38b9-22b2-43f2-a63a-8875a3a43315",
    "LocationName" : "1st Floor Womens",
    "LocationCode" : "1FLW",
    "SensorValue" : 35.0,
    "CMinDist" : 5.0,
    "CInitialDist" : 35.0,
    "TimeStamp" : ISODate("2018-04-24T21:59:12.268+0000")
}
```

Figure 1: Sensor Document from Sensors MongoDB Atlas Collection

```
{
    "_id" : ObjectId("5adfa8b0d0d6e630ccaba9d6"),
    "SensorId" : "W52858b18-0bb7-45b0-836c-d4d5f335558e",
    "SensorValue" : 0.0,
    "Timestamp" : ISODate("2018-04-24T21:59:12.268+0000")
}
```

Figure 2: History Document from SensorHistory MongoDB Atlas Collection

2.2 SQL Server, Azure

ASP.NET Identity utilizes SQL Server for storing user identify data by default. Our Identity database is hosted on an Azure [SIZE HERE] SQL instance. User data is contained in the *AspNetUsers* table and the following schema is used:

- FirstName
- LastName
- Email
- Phone
- NotificationOptIn
- UserName

2.3 ESP32 Wi-Fi Module

The ESP32 Wi-Fi module is programmed with Arduino C. It requires the expressif/ESP library for Arduino which includes support for WEP, WPA, WPA2-Personal, and WPA2-Enterprise. The device is programmed to search for a Wi-Fi network, connect to it, and then when an X-bee message is received, it is parsed into the following format:

```
"{\"sensorId\": \"" + sensorId + "\",\"sensorValue\": " + sensorValue + ", + "}"
```

There is no optimization on this device since it is intended to be connected to wall power.

2.4 Water Sensor

The Grove Water Sensor utilizes interchanged Signal and Ground traces on the surface of the sensor in order to detect the presence of water. The Signal traces are nominally pulled high, but when water comes into contact with the traces on the surface of the sensor, the Signal traces are short circuited and pulled low. The groveSender.ino Arduino sketch used to program the Grove Water Sensor-compatible SmartLoo Sensor Nodes are written to check the status of the Grove Water Sensor Signal line and report a “1” if there is water on the surface of the sensor and the Signal is low, or a “0” if there is no water on the surface of the sensor and the Signal is high. The sketch also includes logic to put the Sensor Node in a low power consumption sleep mode, when the Sensor Node does not need to be used.

2.5 Ultrasonic Sensor

The HC-SR04 ultrasonic sensor sends a sound wave and waits for it to return. At the moment it is returned, the sensor calculates the distance in the following manner:

$$\text{Distance} = (\text{high level time} \times 340\text{m/s}) / 2$$

The sketch also includes logic to put the Sensor Node in a low power consumption sleep mode, when the Sensor Node does not need to be used.