Recipe Ready Documentation

Know what's in your kitchen. Anytime, anywhere

TEAM 7: Clayton Warren, Rocky (Michael) Petkov, Niki Lin, Chuck Fu, Jack Qiao

Table of Contents

Project Outline

Our Project “Recipe Ready” is an IoT device for the kitchen. It seamlessly combines hardware and software that provides a consumer the ability to have immediate knowledge of what they can, or could make with the ingredients that have present in their kitchen. The user sets up the hardware in their kitchen by placing sensors around their cupboard and fridge that are all connected to a central hub. This hub is connected to their Wi-Fi and knows (based on the sensors) what they have in their kitchen. The user accesses their “kitchen” via any web browser at criwcomputing.com. This website gives them an interface to set what sensors are to indicate what ingredients, and will provide them with the recipe information they desire! That’s all there is to it. Set up your sensors and start finding our what you can, could, or should make in the kitchen!

Problems Encountered

As a team we encountered very few problems. What we found most challenging was picking from the many services offered for interacting, and running things in the cloud. As some of our team members went into the project with experience with certain seveives, it made our selection of the correct easy.

Our original choice of hardware was to base the project off of the Raspberry Pi Zero. This brought some difficulties with it as it meant that connecting the Arduino to the Pi relied upon using the GPIO pins. Unfortunately, those pins have no level of protection should the pin be supplied with too much voltage (in the case of accidentally routing data to a ground pin) and it meant that our original PI was fried in the process of making this project. We were able to compensate by getting a Raspberry Pi 3. While this provided significantly more power, the existence of more USB ports meant that we could forgo using the GPIO headders.

As one can expect, the majority of our issues came when harmonising the hardware end of the project with the cloud service. There were some misunderstandings between team members over the exact content of the HTTP requests. For the most part, these were easy fixes and involved simply renaming attributes and slightly reformatting how the Python programme sent data out to the cloud.

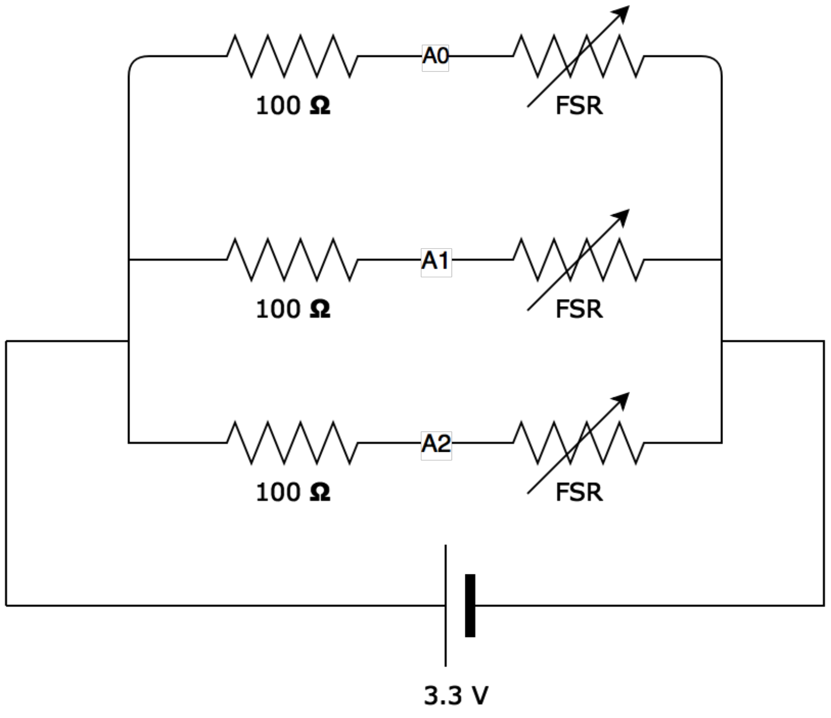
For our Cloud server, and front-end, we benefitted greatly from one group member having their own MEAN boiler plate. This gave us the template for the database, web formatting and an architecture for dealing with HTTP requests. As a result, we had great agility when spinning up our app in the cloud a quick process. As a result, we were able to spend more of our time focusing on the implementation of our idea, not the infrastructure for it.

I/O devices

Force Sensitive Resistor

Force sensitive resistors (FSR) are an essential part to our IOT devices, as they are the source of input for our system. Each FSR has a sensitive padding of 3.8 cm by 3.8 cm, these sensitive paddings will sense any weight on the top, and alter their resistance based on the amount of pressure is applied. With no pressure, the resistor will have functionally infinite resistance (In excess of 10 million ohms). As pressure is being applied, the amount of resistance will descend logarithmically, with the lower bound capped at 200 ohms, allowing much more current to go through. This difference in current flow leads to a change in the voltage over the data out wire.

Since the force resistive sensors output an analogue voltage, we needed to have some means of performing an Analog – Digital Conversion (as the Raspberry PI lacks the ability to do ADC on any more than one device. As a result, we incorporated an Arduino Uno to serve in the same capacity a dedicated ADC Pi addon board might. We designed our circuit so that the three FSRs are connected in parallel; they share the 3.3 volt power source from the Arduino, while connected to the same ground pin. Each FSR also output their reading to a particular analog read pin on the Arduino.



Arduino Uno

The purpose of the Arduino Uno is to collect data from the force sensitive resistors, translate the analogue voltages into digital equivalents, analyze the information to determine the availability of the ingredients and transmit that information to the Raspberry Pi.

***Processor*: ATmega328 microcontroller**

Memory: 32 KB flash memory

I/O Ports:

- 14 Digital I/O pins including pins for:
  - Serial data (RX and RX)
  - External interrupts
  - 6 PWN output
  - SPI communication
  - LED
  - TWI communication
- 6 Analog Input Pins
- 1 x Micro USB port
- 1 x USB port

Power Requirements

The Arduino Uno can be powered via a USB connection or with an external power supply. The recommended range is 7-12 volts. If supplied with less than 7 voles, the 5V pin may supply less than 5V and the board may be unstable. If supplied with more than 12V, then the voltage regulator may overheat and damage the board.

There are four power pins:

- VIN – Input voltage when using an external power source
- 5V – Regulated power supply
- 3V3 – A 3.3V supply generated by the regulator, with a maximum current draw of 50mA
- GND – Ground pins

Raspberry Pi

The Raspberry PI serves as the bridge between the Arduino and the Recipe Ready cloud service. To serve in this role, it forwards data between collected by the Arduino and the cloud service.

RPI Technical Specifications

Note: Due to a failure of the Raspberry Pi Zero, we had to carry on using a Raspberry PI 3B. These two devices have differing specifications and for the sake of completeness, both are presented here

Raspberry Pi Zero W
CPU: Broadcom BCM 2835. 1x1Ghz. Armv6Z 32 bit architecture
Memory: 512 MB
I/O Ports:

- 1 x Micro USB port
- Raspberry Pi 40 pin GPIO headder
- 1 x Mini HDMI out

Power Supply:

- 1x Micro USB @ 5.1v
- GPIO pin power input @ 5.0v
- This input has no safety. Oversupplying power will cause system failure
- Operates on 5.0 volt logic

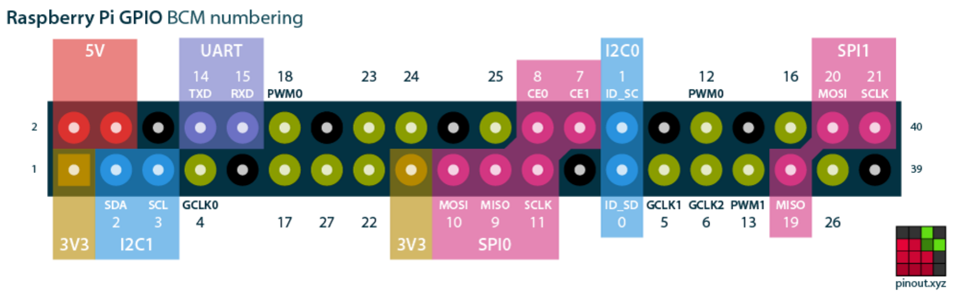
Raspberry Pi 3B
CPU: Broadcom BCM 2837. 4x 1.2 GHZ. ARM v8A 64 bit architecture)
Memory: 1 GB (shared with GPU)
I/O Ports

- 4x USB 2.0 Ports
- Raspberry Pi 40 pin GPIO Headder
- 1x HDMI out
- Ethernet Port
- 3.5 MM audio out jack.

Power Supply

- 1x Micro USB 5.1v input
- GPIO Power Input @ 5.0v
- This input has no safety. Oversupplying power will cause system failure
- Operates on 5.0 volt logic

Pinout



Above is an image of the Raspberry PI pinout. For our project it was not necessary to make use of the pinouts as we connected the Arduino Uno and Raspberry PI 3b directly via USB.

Programming Environment

The data forwarding programme was developed using Python 2.7. This environment was chosen mainly due to Python's strength as a rapid prototyping language. The short development cycle of this project meant that our primary concern would be simply making the project work, rather than trying to make it work in the most efficient possible manner. To this end, two libraries proved to be invaluable in our development of this project: PySerial and Requests.

PySerial & Serial Communications

PySerial is a library which encapsulates the operations of a serial port. As a result, the low level realities of reading bits as they come off the wire and later interpreting that raw data into a string is unseen and unknown to our programme. Utilising this library enabled us to divert most of our attention to the actual contents of the messages and how to structure the message in a manner that would make for easy interpretation by the PI but also minimise the size of the message. While any inefficiencies would not be apparent with just three sensors, if the project were to scale up to include 20, 30 or even 50 sensors (not unrealistic if we were to take such a product to market), a lightweight communications protocol would be vital in ensuring a responsive product.

The Protocol we decided upon was simple, elegant and reasonably scalable. The message is simply a string where the n-th character reflects if the n-th sensor is detecting pressure (1 if it is detecting pressure, 0 if it is not). Selection of this protocol made sense for two reasons.

1. Both the Arduino serial library and PySerial naturally work with strings. As we will detail later, to read a string from a serial port involves simply making a port.readline() call, while if we were to implement a byte array, additional processing on the Python end would be needed to reconstitute the data in a form that the rest of the programme would work with.

2. For home use, the largest sensor array we could see being installed would be no more than 50 sensors. This means the largest serial communication between the Arduino and the PI would 50 bytes plus the overhead for the serial communications. We deemed this to be an acceptable message payload size especially considering that USB 2.0 is capable

of 35 MB/s.

- Note: Were the project to scale up much more than to 50 or so sensors, we would consider using a bit array which, for the purposes of serial communication is recast as a series of ASCII characters over the wire and disassembled on the PI side.

Using PySerial

Below is a brief introduction to PySerial as we used it with code snippets. This represents only a fraction of what the library is capable of. For a full description of what the library is capable of, the full documentation is available at: http://pythonhosted.org/pyserial/

Opening Serial Ports

To open a serial port, one simply has to call the function

Port = serial.serial(port_location, baud_rate, timeout)

- port_location: On a unix based system this would be the path to the memory mapped file of the port.
- baud_rate: Rate of communications. For our project we chose a baudrate of 9600
- timeout: This is the maximum amount of time the programme will wait on incomming serial communications, unit is in seconds. For our project we set this to be half a second. While this means there are some polling loops where no new information is ascertained by the PI. This is acceptable as the PI programme simply ignores these attempts at reading data off of the wire.
- “Port” this is the name of the object we are encapsulating our serial port within.

Reading From Serial Port

To read from the serial port, there are two choices.

Port.read(n)

- Reads at most n bytes from the serial port.

Port.readline()

- Reads from the serial buffer until a newline character is encountered.
- We opted for this methodology for our project as we are reading strings off of the serial port rather than bytes.

Closing Serial Ports

To safely close the serial port interface one can simply call

Port.close()

Requests Library & Cloud Communications

Our programme communicates with the Recipe Ready Server by sending HTTP POST requests to an endpoint that is located at http://criwcomputing.com/kitchen/sensors/update. In order to make the job of our cloud platform easier, updated sensor information is not sent as one long POST request but instead as a series of multiple smaller POST requests which only have the information pertaining to one sensor.

The body of the POST requests is a JSON string consisting of three simple fields:

- kitchenID: This is an unique id belonging to the device and informs the server whose profile the update is supposed to be reflected on.
- Sensor: The number of the sensor the message pertains to
- update: Whether or not the sensor is sensing weight. Unlike the Arduino -> PI communications, this sent as a string as either “true” or “false”,

To facilitate communications with the cloud server, we used the Requests library. This third party library improves upon python’s built in urllib library by providing a much easier to use interface for making HTTP requests than urllib.

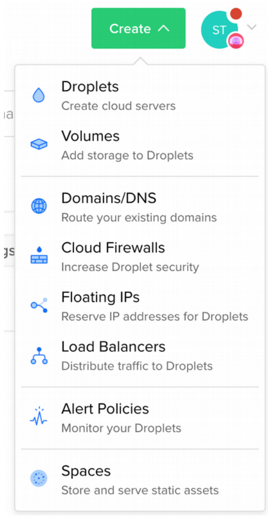
Cloud Service:

Technology Used:

Digital Ocean: A cloud computing provider that will run Ubuntu with 512 MB of RAM. 20 GB of SSD disk space, and a single core processor. This service provider lets you scale these servers horizontally or vertically. Most likely an initial vertical scale would be required to bring the spec of each instance (Digital Ocean calls this a droplet) up for a higher processing power. This will be followed by a horizontal scale when each instance can handle a sufficient TPS (transaction per second) and volume. In growth, a horizontal scale using Digital Ocean paired with the other services they offer, such as load balancers will keep our "hardware in the cloud" cost elastically low. We can create instances in different geographical locations to better serve regions as well.

Services offered

Digital Ocean's main offering is cloud computing. They give remote access to any number of servers, or droplets as DO calls them. You can set up a "one click app" that will come with pre-installed software for quickly spinning up instances of what you need. We have chosen the smallest size droplet as described above. There are many different development "stacks" that are available, and we have chosen the MEAN stack. This is Mongo DB, express, Angular, and Node. This is all the software we need to run a REST API server, that can perform IO over HTTP, store data in a database, as well run a front end webpage.



Sensor data processing

Once the data is received from the IoT device it is parsed through an asynchronous process. As data is received in JSON format it is parsed and stored as local variables in the process. A call is made to our database to find the specific kitchen to be updated. Once this kitchen document is found, another process calls an external API called "recipe puppy" through an HTTP request with the ingredients in the header. This is to find out what recipes can be made with the updated ingredients. This data is received by the server in JSON format back from the API and parsed as well using the built in JSON methods of Node.js. Once the data is parsed, a process generates HTML to be display on the front end, and saves all of this updated information in the "kitchen document" to our database. This server side rendering of most every component of information allows our IoT device and our front end to do very little work, and therefore run on very minimal hardware. The front end can be any web browser on a desktop or mobile device!

Development (programming) environment.

There are many options to develop the code running on this cloud server. You can access the server directly through SSH and edit code files through a bash terminal using something like VIM or Nano. Developing JavaScript (node.js) can be done in any text editor however. First we used GitHub to create a remote project that can be accessed by anyone with the git URL. Using any bash terminal, you can “clone” the repository of code for local editing.

Installing the most recent node.js packages from their website, as well as installing MongoDB on the local development computer allows the code to be run and tested locally. Node.js comes with a built in package manager called NPM (node package manager). This allows external packages to be installed in the code repository for use. Packages such as express (for HTTP acceptance) and mongoose (a wrapper to access mongo DB in an easier fashion) were installed for our project. The full list of NPM packages used can been seen in the final code, under the folder node_modules. To test his code locally, first one must open a bash terminal and run the command “mongo”. This will create a process that allows the node.js server to connect to for storing and retrieving data. Once this process starts, you can run the command “node app” in another bash window. This starts the server application that handles incoming and outgoing connections to and from the IoT device, the front-end application, the database, and any external recourses such as API’s for recipes.

This is the same process that is required to run the application on the cloud server. First one must SSH in to the server with the root user. After inputting the password, they would start a “screen” process. This allows processing to run even after the SSH session closes. Once the screen has been enabled you clone the repository using the github URL. cd in to the application directory, and use the command “npm install” to install all package dependencies. After this you can simply run the command “node app” to start the server. Mongo is already a running process on the instance we have used so there is no need to start it. After the server starts running, any kitchen (a recipe ready hardware) can start POSTing their data, and accessing their kitchen dashboard from the website criwcomputing.com.

Communication with IOT device and End Data Application.

Communication protocells and methods have been discussed in detail in many other sections. In short, all communication is processed through HTTP requests. The IoT device makes POST requests to a single endpoint. The front-end web browser views data, and makes POST requests the the server using HTML form data.

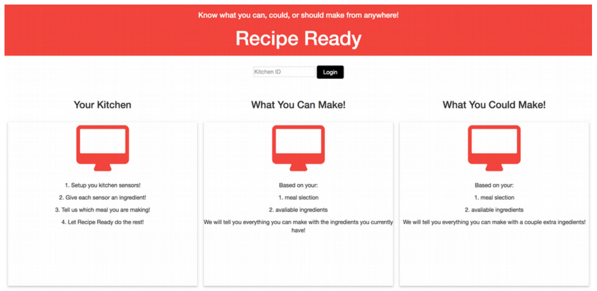
Returning data to the IoT device is simple, and only involves a status code, and the successful information that was updated. This would allow the hardware to perform other simple processing if necessary.

Returning data to the front-end is slightly more extensive. A JSON object is returned to the front-end when it makes requests. This JSON object is the whole kitchen document that was edited when a call to the server was made. The object contains information to redirect the web browser, and HTML renderings to display on the page.

End Data Application:

Description (it's just a browser)

The front end can can be any web browser, preferably one capable of handling HTML 5 (basically anything but internet explorer). Here are some screen shots of the display on our website, on desktop first, and then mobile!



Know what you can, could, or should make from anywhere!

Recipe Ready

Kitchen ID Login

Your Kitchen

1. Setup you kitchen sensors!
2. Give each sensor an ingredient!
3. Tell us which meal you are making!
4. Let Recipe Ready do the rest!

What You Can Make!

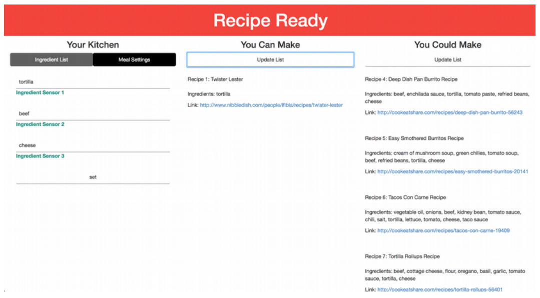
Based on your:
1. meal slection
2. avaliable ingredients

We will tell you everything you can make with the ingredients you currently have!

What You Could Make!

Based on your:
1. meal slection
2. avaliable ingredients

We will tell you everything you can make with a couple extra ingedients!



Recipe Ready

Your Kitchen

Ingredient List | Meal Settings

tortilla
Ingredient Sensor 1

beef
Ingredient Sensor 2

cheese
Ingredient Sensor 3

set

You Can Make

Update List

Recipe 1: Twister Lester

Ingredients: tortilla

Link: http://www.nibbledish.com/people/lfibla/recipes/twister-lester

You Could Make

Update List

Recipe 4: Deep Dish Pan Burrito Recipe

Ingredients: beef, enchilada sauce, tortilla, tomato paste, refried beans, cheese

Link: http://cookeatshare.com/recipes/deep-dish-pan-burrito-56243

Recipe 5: Easy Smothered Burritos Recipe

Ingredients: cream of mushroom soup, green chilies, tomato soup, beef, refried beans, tortilla, cheese

Link: http://cookeatshare.com/recipes/easy-smothered-burritos-20141

Recipe 6: Tacos Con Carne Recipe

Ingredients: vegetable oil, onions, beef, kidney bean, tomato sauce, chili, salt, tortilla, lettuce, tomato, cheese, taco sauce

Link: http://cookeatshare.com/recipes/tacos-con-carne-19409

Recipe 7: Tortilla Rollups Recipe

Ingredients: beef, cottage cheese, flour, oregano, basil, garlic, tomato sauce, tortilla, cheese

Link: http://cookeatshare.com/recipes/tortilla-rollups-56401

Description of method of communication with the Cloud Service.

Again, these methods have been described in other section, but here is more detail. The data is inputted in HTML forms, such as which kitchen to log in to, or what ingredients to set in that kitchen. Once the buttons are clicked, a JavaScript process starts on the front end to make an HTTP PUT request the the server address at the designated endpoint. This process is returned a JSON object that contains status codes, and information on what HTML strings to display. Having this process only make the request and display the HTML allows us to continuously change on front-end display without interrupting service to clients! This can be seen in the HTML file "loggedIn.HTML" which has very minimal processing and mostly skeltons the HTML.

Description of communication protocol and capabilities.

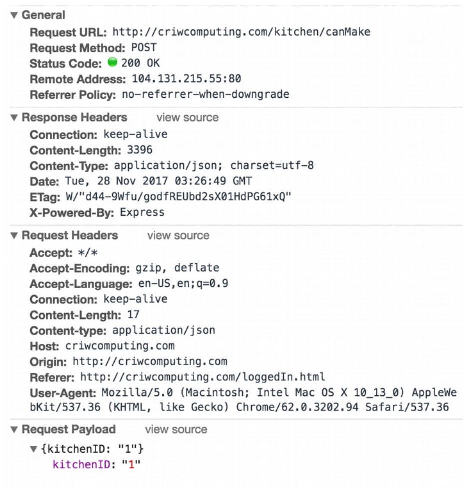
There are many abilities of a front end browser to communicate over HTTP, or HTTPS. Many modern browsers can run a myriad of scripts. As we wanted to keep our front-end as light as possible, all communication is done over HTTP, with simple request headers.

Description of any data processing done.

The data processing that is done is as minimal as possible. An HTML string is contained in one of the elements of the response body. This string is stored on the front end, and rendered in the specific div (section on the screen)

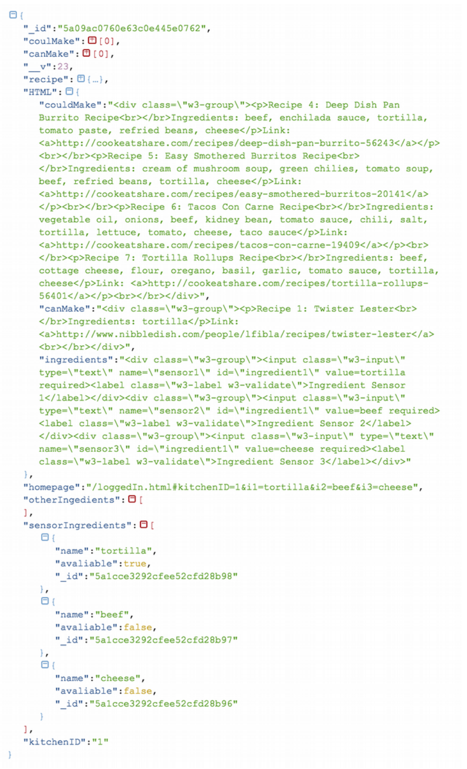
Data example.

Request (screen shot from chrome inspector)



```
▼ General
    Request URL: http://criwcomputing.com/kitchen/canMake
    Request Method: POST
    Status Code: ● 200 OK
    Remote Address: 104.131.215.55:80
    Referrer Policy: no-referrer-when-downgrade
▼ Response Headers      view source
    Connection: keep-alive
    Content-Length: 3396
    Content-Type: application/json; charset=utf-8
    Date: Tue, 28 Nov 2017 03:26:49 GMT
    ETag: W/"d44-9Wfu/godfREUbd2sX01HdPG61xQ"
    X-Powered-By: Express
▼ Request Headers      view source
    Accept: */*
    Accept-Encoding: gzip, deflate
    Accept-Language: en-US,en;q=0.9
    Connection: keep-alive
    Content-Length: 17
    Content-type: application/json
    Host: criwcomputing.com
    Origin: http://criwcomputing.com
    Referer: http://criwcomputing.com/loggedIn.html
    User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_0) AppleWe
    bKit/537.36 (KHTML, like Gecko) Chrome/62.0.3202.94 Safari/537.36
▼ Request Payload      view source
    ▼{kitchenID: "1"}
        kitchenID: "1"
```

Response (JSON object logged in chrome inspector, actual render is shown in screen shots above



```
{
  "_id":"5a09ac0760e63c0e445e0762",
  "coulMake":[0],
  "canMake":[0],
  "__v":23,
  "recipe":{…},
  "HTML":{
    "couldMake":"<div class=\"w3-group\"><p>Recipe 4: Deep Dish Pan
    Burrito Recipe<br></br>Ingredients: beef, enchilada sauce, tortilla,
    tomato paste, refried beans, cheese</p>Link:
    <a>http://cookeatshare.com/recipes/deep-dish-pan-burrito-56243</a></p>
    <br></br><p>Recipe 5: Easy Smothered Burritos Recipe<br>
    </br>Ingredients: cream of mushroom soup, green chilies, tomato soup,
    beef, refried beans, tortilla, cheese</p>Link:
    <a>http://cookeatshare.com/recipes/easy-smothered-burritos-20141</a>
    </p><br></br><p>Recipe 6: Tacos Con Carne Recipe<br></br>Ingredients:
    vegetable oil, onions, beef, kidney bean, tomato sauce, chili, salt,
    tortilla, lettuce, tomato, cheese, taco sauce</p>Link:
    <a>http://cookeatshare.com/recipes/tacos-con-carne-19409</a></p><br>
    </br><p>Recipe 7: Tortilla Rollups Recipe<br></br>Ingredients: beef,
    cottage cheese, flour, oregano, basil, garlic, tomato sauce, tortilla,
    cheese</p>Link: <a>http://cookeatshare.com/recipes/tortilla-rollups-
    56401</a></p><br></br></div>",
    "canMake":"<div class=\"w3-group\"><p>Recipe 1: Twister Lester<br>
    </br>Ingredients: tortilla</p>Link:
    <a>http://www.nibbledish.com/people/lfibla/recipes/twister-lester</a>
    <br></br></div>",
    "ingredients":"<div class=\"w3-group\"><input class=\"w3-input\"
    type=\"text\" name=\"sensor1\" id=\"ingredient1\" value=tortilla
    required><label class=\"w3-label w3-validate\">Ingredient Sensor
    1</label></div><div class=\"w3-group\"><input class=\"w3-input\"
    type=\"text\" name=\"sensor2\" id=\"ingredient1\" value=beef required>
    <label class=\"w3-label w3-validate\">Ingredient Sensor 2</label>
    </div><div class=\"w3-group\"><input class=\"w3-input\" type=\"text\"
    name=\"sensor3\" id=\"ingredient1\" value=cheese required><label
    class=\"w3-label w3-validate\">Ingredient Sensor 3</label></div>"
  },
  "homepage":"/loggedIn.html#kitchenID=1&i1=tortilla&i2=beef&i3=cheese",
  "otherIngedients":[
  ],
  "sensorIngredients":[
    {
      "name":"tortilla",
      "avaliable":true,
      "_id":"5a1cce3292cfee52cfd28b98"
    },
    {
      "name":"beef",
      "avaliable":false,
      "_id":"5a1cce3292cfee52cfd28b97"
    },
    {
      "name":"cheese",
      "avaliable":false,
      "_id":"5a1cce3292cfee52cfd28b96"
    }
  ],
  "kitchenID":"1"
}
```

References

Foundation, N. (n.d.). Docs. Retrieved November 28, 2017, from https://nodejs.org/en/docs/

Mongoose. (n.d.). Retrieved November 28, 2017, from http://mongoosejs.com/docs/guide.html

The MongoDB 3.4 Manual. (n.d.). Retrieved November 28, 2017, from https://docs.mongodb.com/manual/

Statement of Review and Approval

Each team member must sign this form acknowledging that they have reviewed and approve of their team project documentation as submitted. This is important as team members receive individual marks associated with their contribution to the project. Submit this completed form to the course instructor with the final documentation when submitted.

Statement of Review and Approval

Each team member must sign this form acknowledging that they have reviewed and approve of their team project documentation as submitted. This is important as team members receive individual marks associated with their contribution to the project. Submit this completed form to the course instructor with the final documentation when submitted.

Team #: 7

Team Leader: Rocky Petkov

Date Team Project was uploaded to the course web site for marking: Nov 28th 2017

Team Member Signatures

Team Member Name (print)	Date	Signature	Approval of Submission (please check for yes or leave blank if no)
Clayton Warren	Nov 27 2017	[illegible]	Yes
Rocky Petkov	Nov 27 2017	[illegible]	Yes
Jack Qiao	Nov 27 2017	Jack Qiao	Yes
Niki Lin	Nov 27 2017	[illegible]	Yes
Chuck Fu	Nov 27 2017	[illegible]	Yes