CS807 - Final Project

André E. dos Santos
Jhonatan S. Oliveira

March, 2016

University of Regina

Department of Computer Science

Resource Constrained Computing

CS807 - Final Project

André E. dos Santos
Jhonatan S. Oliveira

1. Reviewer Dr. David Gerhard
Department of Computer Science
University of Regina

2. Reviewer Mr. Trevor Tomesh
Department of Computer Science
University of Regina

March, 2016

André E. dos Santos
Jhonatan S. Oliveira
CS807 - Final Project
Resource Constrained Computing, March, 2016
Reviewers: Dr. David Gerhard and Mr. Trevor Tomesh

University of Regina
Department of Computer Science
3737 Wascana Pkwy
Regina, S4S 0A2

Abstract

An interactive kiosk is a computer terminal that provides access to information and applications for communication. It contains specialized hardware and software designed within a public exhibit. Thus, an interactive kiosk must address resourceful solutions to minimize its possible constrains. In this project report we propose an interactive kiosk with speech recognition for academic environments. We utilized *Wit.ai*, a *cloud service* that turns speech or text into actionable data. Wit works on the project as a accessibility tool an interactive feature. The server, data modeling to the client and user interface was built with *Meteor.js*, a full-stack javascript framework. The hardware of chosen was Raspberry Pi 2, a low cost single-board computer with salient features to support efficiency and reliability for the project interactive kiosk.

In this project report we discuss the process and constrains to build an interactive kiosk with speech recognition. We discuss the challenges of localization and map apps that impose barriers efforts for simple implementations. Finally, we describe the implementation and functionality of high functioning search algorithms for information retrieval on the server.

Contents

1 Introduction

”

—
(TODO)

A *terminal information systems* is a computation system which [blank]. As a computation system, a terminal information systems is has specific constrains associated with its components. For instance, power constrains applies since the terminal usually is implemented in low power devices. In this report we present the proposal and implementation of a low cost terminal information systems with voice recognition feature.

Accessibility is a key assistive accommodation to enable people overcome physical, technological, or informational barriers [1]. In our project we sought for tools to enhance the experience and quality on a terminal information systems.The first tool we chose was Wit.ai, [blank]. Wit.ai is good [blank].

As foe an efficient system to manage the data and improve the information retrieval, the implementation was executed with meteor.js, [blank] Meteor is goof [blank].

The cost of the project is manly dictated by the hardware component that holds the system. Some platforms usually utilized for low cost system are [blank]. In our project the hardware chosen was Reaspberry.py, [blank]. Rep is goof [blank.]

With the implementation some issues were faced. For instance, one of our first goal of the project was to provide a map localization and direction to the user. This feature is commonly seen in big centers such malls, hospitals or airports. Althouhgt some systems are availabe, such as [blank]. we realized they are not at level for small systens, and when they are, not simply. This constarins are better descripted in Section [Blanck] We then turn our attentions for the soctware design and search engine of the termnal. We developted a semantic seach that permforms efficienty search on a tree structure. Well know algorith as breadth-fisrts and deapth serach was utilizled for this task.

The project presents many salient feature. Firts, it cn be voice activated and has state-of-the art speech recognition trhoeght the usage of the wit.ai. Second, it

performs an efficienty seach on the tree strcytue implemented on metero. Fetures as uncertanty decision,imcplemete infromation are well handled by the sustem. Thir, The implementations was condutc on a a reps.py, which is wel suited for the project since it is (i) [blank] and (ii) [blank]. This report is organized as follows. Section . Section . Section . Section .

Background

2

> *TODO.*
>
> — **TODO**
> (TODO)

2.1 Terminal Services

As defined by [15], “an interactive kiosk is a computer terminal featuring specialized hardware and software designed within a public exhibit that provides access to information and applications for communication, commerce, entertainment, and education.”

Integration of technology allows kiosks to perform a wide range of functions, evolving into self-service kiosks.

Types of kiosks Telekiosk Financial services kiosk Photo kiosk Internet kiosk Ticketing kiosk Information kiosk

As companies in a range of industries seek to reduce costs, increase revenue opportunities, and improve customer service, they are increasingly turning to interactive kiosks as an additional channel to enable their customers with self-service options. The number of interactive kiosks in operation will rise from approximately 1.6 million deployed in 2011 to nearly three million deployed globally by 2016. https://www.abiresearch.com/press/the-number-of-interactive-kiosks-in-operation-will/

While the interactive kiosk market is expected to grow strongly over the next five years, there remain challenges to address. “Interactive kiosks in various segments, such as healthcare, can face challenges regarding consumer acceptance, channel conflict with other means of interacting with the consumer, and with automated customer service not meeting a desired level of personalized support,” says Lucero.

Reliability is an important consideration, and as a result many specialised kiosk software applications have been developed for the industry. These applications interface with the bill acceptor and credit card swipe, meter time, prevent users from

changing the configuration of software or downloading computer viruses and allow the kiosk owner to see revenue. Threats to reliability come from vulnerabilities to hacking, allowing access to the OS, and the need for a session or hardware restart

The kiosk industry is divided into three segments: kiosk hardware, kiosk software, and kiosk application. Kiosk software locks down your operating system (be it Apple, Windows, Android, or Linux) to restrict access and/or functionality of a kiosk hardware device. This allows for users to interact with an application that serves a self-service purpose such as those mentioned above. Kiosk manufacturing industry

Historically electronic kiosks though are standalone enclosures which accept user input, integrate many devices, include a software GUI application and remote monitoring and are deployed widely across all industry verticals. This is considered "Kiosk Hardware" within the kiosk industry.

POS-related "kiosks" are "lane busting" check-outs such as seen at large retailers like Home Depot and Kroger.

Simple touchscreen terminals or panel-pcs are another segment and enjoy most of their footprint in POS retail applications and typically facing the employee. Terminals include NCR Advantage (740x terminal) and the IBM Anyplace computer terminal. These units are considered "kiosks" only in functionality delivered and typically only incorporate touchscreen, bar code scanner and/or magnetic stripe reader.

Market segments for kiosk and self-service terminal manufacturers include photo kiosks, government, airlines, internet, music, retail loyalty, HR and financial services, just to name some. [8]

2.2 Reactive Programming

In 2001, after the the bursting of the dot-com com bubble, the Web 2.0 phenomenon appeared and a survival way to the recent collapse [13]. New applications aimed to involve the user on the content creation process. One way of engaging the user is to offer a familiar environment, a similar to the one used in a desktop computer. Web platforms and frameworks made use of *reactive programming* [3] to achieve a desktop-like experience on the web. A consequence of turning the content creation responsibility to the user is the massive amount of data created by them, which originated another phenomenon called *big data* [16]. In order to manage all these data, new paradigms emerged. *Non-relational* or *No-SQL* [18] is a database paradigm that manage the data storage and retrieval without using a relational table. Novel

web platforms and frameworks had to incorporate these new aspects in order to model and maintain the complexity requirement of the new applications.

2.3 Graph Data Structures

Graphs are a fundamental data structure in the world of programming. Knowing the correct data structures to use with graph problems is critical [19]. Graphs can represent many different types of systems. A graph consists of two components: nodes and edges. A node (or vertex) is a discrete position in the graph. An edge (or connection) is a link between two vertices that can be either directed or undirected and may have a cost associated with it. An undirected edge means that there is no restriction on the direction you can travel along the edge. Formally, a *graph* $G = V, E$ is defined as a set of vertices, V, and a collection of edges (which is not necessarily a set), E [2]. An edge can then be defined as (u, v) where u and v are elements of V.

2.3.1 Trees

A tree is an undirected graph in which any two vertices are connected by exactly one path. Thus, a acyclic connected graph is also a tree.

2.3.2 Basic methods for searching graphs

There are two methods for searching graphs that are prevalent: *Depth First Search* and the *Breadth First Search* [7].

Breadth-first search

Breadth-first search (BFS) is a strategy for running through the vertices of a graph. It was presented by Moore [**?**] in 1959 within the context of traversing mazes. Lee [**?**] independently discovered the same algorithm in 1961 in his work on routing wires on circuit boards.

The basic BFS algorithm can be described as follows. Starting from a given vertex v of a graph G, we first explore the neighbourhood of v by visiting all vertices that are adjacent to v. We then apply the same strategy to each of the neighbours of v. The strategy of exploring the neighbourhood of a vertex is applied to all vertices of G. The result is a tree rooted at v and this tree is a subgraph of G. Algorithm **??**

presents a general template for the BFS strategy. The tree resulting from the BFS algorithm is called a *breadth-first search tree*.

The breadth-first search algorithm makes use of a special type of list called a *queue*. Formally, a queue Q is a list of elements. At any time, we only have access to the first element of Q, known as the *front* or *start* of the queue. We insert a new element into Q by appending the new element to the *rear* or *end* of the queue. The operation of removing the front of Q is referred to as *dequeue*, while the operation of appending to the rear of Q is called *enqueue*. That is, a queue implements a first-in first-out (FIFO) protocol for adding and moving elements. As with lists, the *length* of a queue is its total number of elements.

Depth-first search

A depth-first search (DFS) is a graph traversal strategy similar to breadth-first search. Both BFS and DFS differ in how they explore each vertex. Whereas BFS explores the neighbourhood of a vertex v before moving on to explore the neighbourhoods of the neighbours, DFS explores as deep as possible a path starting at v. One can think of BFS as exploring the immediate surrounding, while DFS prefers to see what is on the other side of the hill. In the 19th century, Lucas [**?**] and Tarry [**?**] investigated DFS as a strategy for traversing mazes. Fundamental properties of DFS were discovered in the early 1970s by Hopcroft and Tarjan [**?**, **?**].

Algorithm **??** formalizes the above description of depth-first search. The tree resulting from applying DFS on a graph is called a *depth-first search tree*. The general structure of this algorithm bears close resemblance to Algorithm **??**. A significant difference is that instead of using a queue to structure and organize vertices to be visited, DFS uses another special type of list called a *stack*. A list $L = [a_1, a_2, \ldots, a_k]$ of k elements is a stack when we impose the same rules for element insertion and removal. The top and bottom of the stack are $L[k]$ and $L[1]$, respectively. The operation of removing the top element of the stack is referred to as *popping* the element off the stack. Inserting an element into the stack is called *pushing* the element onto the stack. In other words, a stack implements a last-in first-out (LIFO) protocol for element insertion and removal, in contrast to the FIFO policy of a queue. We also use the term *length* to refer to the number of elements in the stack.

3 The Project

> *TODO.*
>
> — **TODO**
> (TODO)

3.1 Introduction

3.2 Related Works

3.3 The Platforms

3.3.1 Wit.ai

Wit.ai is *cloud service* that turns speech or text into actionable data. Wit is supported on every major platform and can be implemented in mobile apps, home automation, wearable devices, robots, and messenger agents. Thus, it is a interesting *natural language processing* tool for developers.

History and Background

Getting Started

In this section, we demonstrate the steps to build a voice application on Wit.ai that obtains colour information from voice recognition and changes one html object with it "on the fly." Note that, in order to sign up on Wit.ai and create the voice application, Wit only allow access through a [5] account.

All steps are drawn from [22] Quick Start Guide. For more details, please refer back to the original source.

The Console

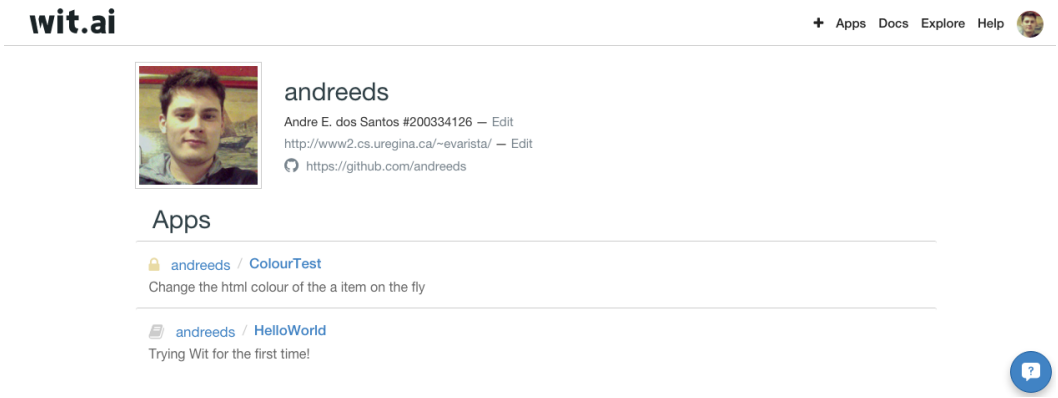

Fig. 3.1: The Wit Console.

Once with access to Wit.ai, one is already able to access to what is called the Wit Console. The Console is where we can manage the Wit.ai-powered apps, configure a voice app and improve it. Thus, on the Wit Console App is where the voice applications logic abides. For instance, the Wit Console with Apps $HelloWorld$ and $ColourTest$ is shown in Figure **??**. Note that if a user signs in for the first time, Wit.ai creates the first app and the user will land on its page. We can create new apps from the top-right menu.

Determining and Creating User Intent

Given that the user has created a new app, it is time to determine the its intents. An intent is something that the end-user wants to perform. For example, "ask about the weather", "set an alarm on their smartwatch", and "say hello to their robot". It is common to focus on a finite list of possible intents. Hence, each intent corresponds to one action in the app. For our application, the intent is to "change the colour of the object" and also"greetings."

Once with a raw speech or text input, Wit.ai will determine what is the user intent.

For instance, all the following expressions should be mapped to the same intent:

```
1 ''Change the colour to blue''
2 ''Colour: blue''
3 ''I want the colour to be blue''
```

There are many different ways to express the same intents. Thus, it is the job of Wit.ai to map these expressions to actual intents.

The user can browse existing intents from the community of Wit online. When typing examples for the intents, Wit will suggest these existing intents. If one fits the intent, clicking on "GET" gets a copy of the existing intent in the Wit app. If there not exist an intent that fits, it is necessary to create a new intent.

To create a new intent, the user must type a name for your new intent in the “Name your intent” field. Usually, intent names try to match the app functions or methods. At least three expressions, even with synonymous way to say the same command, must be added. Figure 3.2 shows the intents $colour$ and $greeting$ for the ColourTest App. Some of the expressions for $colour$ are listed in Example **??**.

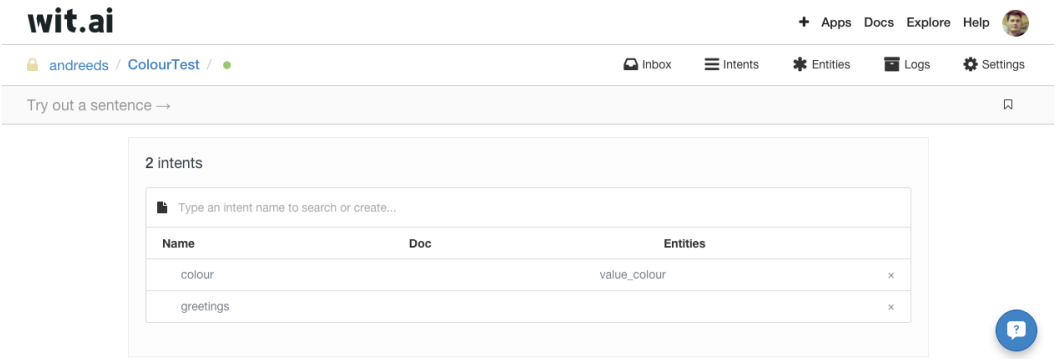

Fig. 3.2: The intents for the ColourTest App.

Training

It is time to query the voice app. At this point, we can already request the voice app via the Wit.ai API.

Notice that, before trainig the app, it is necessary to add the *Client Access Token* to the website. We will show more details about this step in Section 3.3.1.

In the Inbox of the App we can see the examples said when inputed troght website.

At Inbox we can validate the the correct sentences recorded by Wit. Once trained, Wit improves its speech recognitions an the *Confidence* ratio for the expression intents tends to raise. To train the voice is very simple. For each expression it reproduces the audio recorded. If it is correct, we can validate it. If not, we type the correct sentence and then validate it.

At Inbox we can validate matches with expression and intents that were correctly captured by Wit. For instance, Figure 3.3 shows three expressions captured by Wit, the first one was validated as intent $greetings$, the second was *Archived* since was any of the existing intents, and the third one was validated as intent $colour$. In our example, we also want to capture the targeted colour. This is called an *Entity*. We can create our own one or select a common one from the dropdown list. For instance, the entity $value\_colour$ was created for the intent $colour$, as depicted in Figure 3.3 bottom.

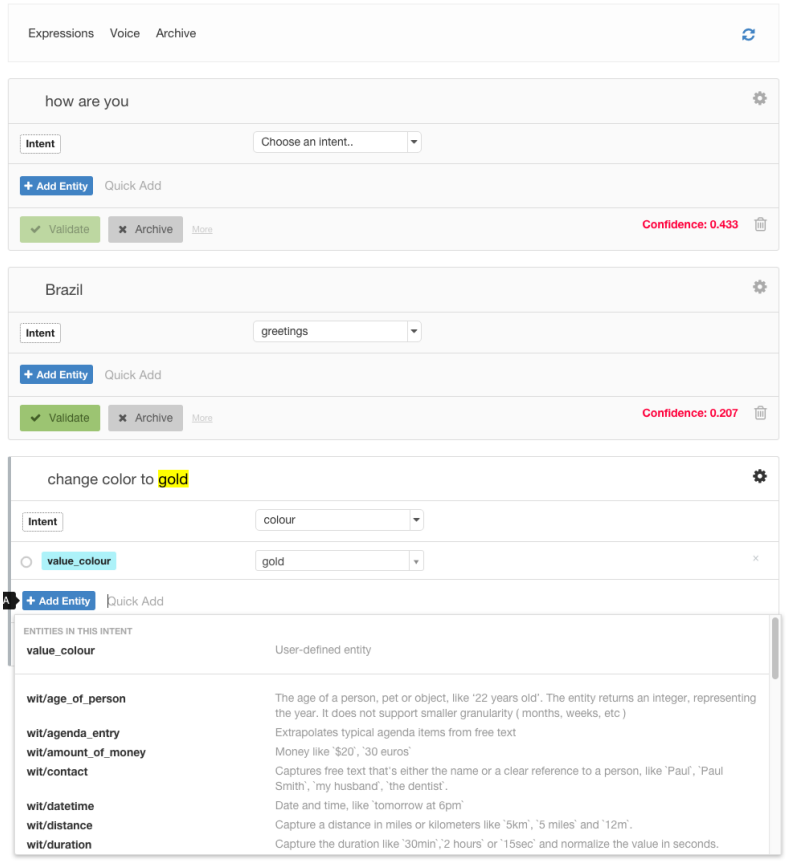

Fig. 3.3: Expressions captured by Wit.ai.

Web application

Now we show how to add Wit to a web app. As a prerequisites, the browser must support [21], which is the case of Chrome, Firefox and Opera.

To integrate the voice app with the web app we need to create a new folder for the app, download the *Web SDK* (the microphone app) and extract the archive:

```
1 mkdir myapp
2 cd myapp
3 curl -L <https://github.com/wit-ai/microphone/releases/download/0.7.0/
    microphone-0.7.0.tar.gz | tar xvzf -
4 mv microphone-* microphone
```

In the myapp folder, we must create a file $index.html$ containing the html provided by Wit on `https://wit.ai/docs/web/0.7.0`.

In the Setting page of the voice app we generate a *Client Access Token*, as illustrated in Figure 3.4 A client access token is unique and authorizes the domain to access the

voice app. To use the client access token we must replace $CLIENT\_TOKEN$ on the $index.html$ file. For example, replace

```
mic.connect("CLIENT_TOKEN");
```

for the client access token of our running example, given

```
mic.connect("AWRBY6WLIPAQP7PGCGMQQKKO45LELCWO");
```

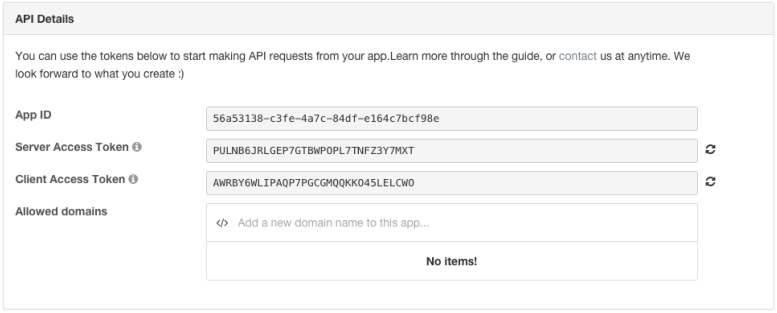

Fig. 3.4: A Client Access Token.

In Action

To see the web application in action we serve the app with a webserver. For instance, using Python:

```
python –m SimpleHTTPServer
```

Then, to load the page on the browser

```
http://localhost:8000
```

After allowing your microphone, we will be able to click on the microphone icon and say a command. The command will be streamed to the voice app.

Resource Constrains

3.3.2 Meteor.js

Meteor.js [10], or Meteor for short, is an open-source web platform to create applications using the Web 2.0 paradigmas. The platform provide a reactive approach by focusing on the data flow. This is done by creating and managing events in the application. Also, data management is done with a non-SQL database called *MongoDB* [11]. Meteor simplify the application development by providing an unique programming language, javascript, throughout the whole stack process. Moreover,

the platform makes available a set of common tools for business logic and data management. Finally, Meteor deploys the application in desktop and mobile without needing to change the source code.

History and Background

The Meteor platform was created by a company called Skybreak in 2011 [17]. Later, in 2012, the company changed its name to Meteor. The startup was incubated by YCombinator [6] and after receiving an investment of $11.2 M, the platform development increased considerably. The platform left beta in October 26th, 2015, with a version that currently provides multiplatform support.

Regarding its internal structure, Meteor is built on top of *Node.js* [12]. That means that Meteor is driven by events, in order to create an asynchronous model. This feature is implemented using callback functions: when an event happens a specific function is called to execute a portion of code. The server-side and client-side are both implemented using javascript. In the client-side, templates are used to design user interfaces. Here, a simple markup language defines the design and the events handled by the application. Meteor has support for more than one template language, being *Blaze* [10] the official one. In the server-side, Meteor handles data management using collections. The oficial non-SQL database supported is MongoDB, but new ones are being incorporated in future versions [9].

Getting Started

Now, we will follow a common flow for an app creation process. During this section, we will use a single running example which is an application for todo lists. This todo example can also be found on the official getting started tutorial in [10]. We will try to keep the same code conventions and names always that possible so the reader can use this paper as an extended guide for that tutorial. We assume that the Meteor program and required

The following command line is used to create a new application with Meteor:

```
1 meteor create todo
```

This creates a folder structure as shown below:

```
1 todo.js todo.html todo.css .meteor
```

The *todo.js* file is where the javascript code for the server and client stays. Here is where the code throws and handles events, besides business logic. Templates for

user interface is done in the *todo.html* file with pure HTML markup language and a template markup language. For this project Blaze is the template languages used. The *todo.css* file contains styles for the templates. Lastly, the *.meteor* stores settings and the meteor aplication itself in a hidden folder.

Templates

Templates are defined using a special tag called *template*. Once defined, a template can be included within any HTML code. In this way, a template is a interface module that can be reuses wherever it is required. All the HTML code and the templates have access to the data made available in that part of the HTML code. The basic flow is: the javascript code makes available some data to a specific area of the HTML code (or a specific template), so the template language can access that data and print the ones of interest for the user. Bellow, we show a simple todo list interface.

```
<head>
  <title>Todo List</title>
</head>

<body>
  <div class="container">
    <header>
      <h1>Todo List</h1>
      <form class="new-task">
        <input type="text" name="text" placeholder="Type to add new tasks
    " />
      </form>
    </header>

    <ul>
      {{#each tasks}}
        {{> task}}
      {{/each}}
    </ul>
  </div>
</body>

<template name="task">
  <li class="{{#if checked}}checked{{/if}}">
    <button class="delete">&times;</button>
    <input type="checkbox" checked="{{checked}}" class="toggle-checked" /
    >
    <span class="text">{{text}}</span>
  </li>
</template>
```

Here, the HTML code defines in its *body* a title and a list. Notice that Blaze commands are defined within {{ and }}. In the list, the Blaze command *#each* goes through a

list of data, as in a loop lace. The data variable *tasks* contains this list of data and was made available to this part of the code by the javascript code. The Blaze command > *name_of_template* prints a template previously defined with name *name_of_template*. In lines 22-28, the template named *task* is defined. This template expect to find available in its scope a data variable called *text*. In line 16, the template is included in that spot of the HTML code and the data variable *text* is available on the loop scope of the *#each* command. Note that *text* is a field within *tasks*.

On the javascript file we define code for the client and server. In order to distinguish between them, Meteors makes available a global variable called *isClient*, which goes to true if the running environment is the browser and goes to false if it is the Node.js one. If a code is called without being under the conditions of a client or server, the code is run in both environments. Follows the javascript code to create a simple database where the *tasks* are saved.

```
1  Tasks = new Mongo.Collection("tasks");
2
3  if (Meteor.isClient) {
4
5    // This code only runs on the client
6    Template.body.helpers({
7      tasks: function () {
8        return Tasks.find({});
9      }
10   });
11
12   Template.body.events({
13     "submit .new-task": function (event) {
14       // Prevent default browser form submit
15       event.preventDefault();
16
17       // Get value from form element
18       var text = event.target.text.value;
19
20       // Insert a task into the collection
21       Tasks.insert({
22         text: text,
23         createdAt: new Date() // current time
24       });
25
26       // Clear form
27       event.target.text.value = "";
28     }
29   });
30
31   Template.task.events({
32     "click .toggle-checked": function () {
33       // Set the checked property to the opposite of its current value
34       Tasks.update(this._id, {
```

```
35          $set: {checked: ! this.checked}
36        });
37      },
38      "click .delete": function () {
39        Tasks.remove(this._id);
40      }
41    });
42
43  }
```

Line 1 runs on the server and on the client, since it is not inside the conditional in line 3. In the server, that command creates a database called *tasks* if it does not exist already. In the client, it creates a cache of the same database where Meteor manages some saved data in order to reuse repeated queries from the user. From line 6 to 9, using the global variable *Template*, we make available to the *body* of the HTML whatever data the not named function in line 7 return. The function returns all the entries in the previously created *tasks* database. These returned data is available to the HTML code on a data variable called *tasks*, as determined in line 7.

Inserting Data

Regarding the data inclusion, this HTML code has a form which will receive the new input data from the user. Lines 9-11 defines a simple form where the user can input new tasks. When pressing enter, an event is thrown from the HTML code and can be handled in the javascript code.

The javascript code watch for an event for the form submission, as shown in line 13. From here, the default reaction of the browser, which is try to submit the form, is stopped. The, the value of the input text is saved in a variable. Next, the cached database variable *Tasks* is used to insert a new entry on the server database with the text saved from the user input. Finally, the text input is cleared.

Notice in the last insertion process that the cached database was used to update the server database. This is possible due to the way Meteor works in client and server. The client has only a cached version of the database. But whenever this cached version is updated, an event goes to the server (whenever there is connection available) making the server database to update. The other way around works in the same manner: if the server database is updated, an event goes to all client spreading the update.

Updating and Removing Data

If a task is done, we want to check it out from the list by updating its entry with a done flag. In the HTML code, line 23, the template *task* has a conditional statement

checking for a variable called *checked*. If the variable is true, the body of the condition, which is just a string *checked*, is executed. This variable, as expected, is set in the javascript code. Indeed, in line 32, the javascript watch for an event of a click on the checkbox. If it happens, the cached database *Tasks* is updated by setting a field *checked* to the opposite value that the user entered. Notice the use of a special variable *this* which provides context for HTML access from where the event occurred.

The deletion process is similar to updating but a different function is used on the cached database. The javascript code, line 24, watch for the HTML code, line 24. When the user clicks in the delete button, the javascript code is executed by removing the correspondent id of the entry. Again, note the use of *this* as a context variable for the entry where the event occurred.

Resource Constrains

Meteor can be used to build resource constrain products. In this type of application, the amount of processing and memory are often restricted. Therefore, the use of parallel computing or decentralized tasks are common and necessary. Meteor can be use to avoid heavy load computation in the limited hardware itself. A server can be used to process all the required heavy workload while the client simply show in real time the results.

One way of providing an interface to a Meteor server side is by using the official server-client protocol, called *Distributed Data Protocol* (DDP). The DDP protocol is a simple specification on how other languages can communicate with a Meteor server.

3.3.3 Raspberry Pi

The *Raspberry Pi* a low cost is single-board computer developed in the UK by the Raspberry Pi Foundation with the intention of stimulating the teaching of basic computer science in schools [14]. It is a capable small device that is capable of doing everything a desktop computer does, from browsing the internet and playing high-definition video, to making spreadsheets, word-processing, and playing games [4].

The Raspberry Pi has a Broadcom BCM2835 system on a chip (SoC), which includes an ARM1176JZF-S 700 MHz processor, VideoCore IV GPU, and 256 megabytes of RAM. It uses an SD card for booting and long-term storage. The Raspberry Pi 2 Model B is the second generation Raspberry Pi. It replaced the original Raspberry Pi

1 Model B+ in February 2015. Compared to the Raspberry Pi 1 it has: (i) A 900MHz quad-core ARM Cortex-A7 CPU and (ii) 1GB RAM [4].

History and Background

In 2006, early concepts of the Raspberry Pi were based on the Atmel ATmega644 microcontroller [23]. The Model A was not the first Raspberry Pi to hit general availability – that distinction went to the Model B. The A was, nevertheless, the truest to the bare-bones ethos of the Pi project, retailing for $25 and initially packing just 128MB of memory. (Later raised to 256MB.)

Getting Started

Required: SD Card. The newer Raspberry Pi Model A+, Raspberry Pi Model B+, Raspberry Pi 2 Model B, Raspberry Pi Zero, and Raspberry Pi 3 Model B require micro SD cards. Display and connectivity cables Keyboard and mouse Power supply

Not essential but helpful to have

Internet connection To update or download software, we recommend that you connect your Raspberry Pi to the internet, either via an Ethernet cable or a WiFi adapter.

Plugging in your Raspberry Pi

Before you plug anything into your Raspberry Pi, make sure that you have all the equipment listed above to hand. Then follow these instructions:

Begin by slotting your SD card into the SD card slot on the Raspberry Pi, which will only fit one way. Next, plug your USB keyboard and mouse into the USB slots on the Raspberry Pi. Make sure that your monitor or TV is turned on, and that you have selected the right input (e.g. HDMI 1, DVI, etc.). Then connect your HDMI cable from your Raspberry Pi to your monitor or TV. If you intend to connect your Raspberry Pi to the internet, plug an Ethernet cable into the Ethernet port next to the USB ports, otherwise skip this step. When you are happy that you have plugged in all the required cables and SD card, plug in the micro USB power supply. This action will turn on and boot your Raspberry Pi. If this is the first time your Raspberry Pi and NOOBS SD card have been used, then you will have to select an operating system and configure it. Follow the NOOBS guide to do this.

Read more in our documentation.

Resource Constrains

The main disadvantages of Raspberry Pi are [20]:

It does not have a real-time clock (RTC) with a backup battery but it can easily work around the missing clock using a network time server, and most operating systems do this automatically.

It does not have a Basic Input Output System (BIOS) so it always boots from an SD card.

It does not support Bluetooth or WiFi out of the box but these supports can be added by USB dongles.

Unfortunately, most Linux distributions are still a bit picky about their hardware, so it should be first checked whether flavor of Linux supports particular device.

It doesn't have builtin an Analog to Digital converter. External component must be used for AD conversion.

It has a relatively small number of digital I/O, but it can be expanded with external logic devices.

3.4 Implementation

How the project works.

- Administration scheme: - Services - Data - Knowledge tree - The tree is built in server not in client - External - Map with key-value being the key the name of the external service and the value a function which runs the service

- Searching the tree - scoring the tree with one pass - problem: when we have the query found in the parent and child. Eg: tainara -> tainara@gmail.com

3.4.1 Description of design process

Design process.

3.4.2 Discussion of group contributions

3.5 Results

Failures and successes.

4 The Ex-Project

> *TODO.*
>
> — **TODO**
> (TODO)

The initial plan for our final project included a service inside the app for indoor navigation. The idea was to allow the user to ask for a location within the University of Regina main campus and the app would trace a route from where the device hosting the app is to where the user requested. Indoor navigation itself is a complex and well known task problem in computer science and engineering. Our primary focus was to use some already available solution for indoor navigation, instead of trying to come up with our own solution. In this way, we wanted to show how resource constrain devices can still be used to provide such service by using cloud computing.

We did a broad research on publicly available solutions for indoor navigation, including paid, free or open source ones.

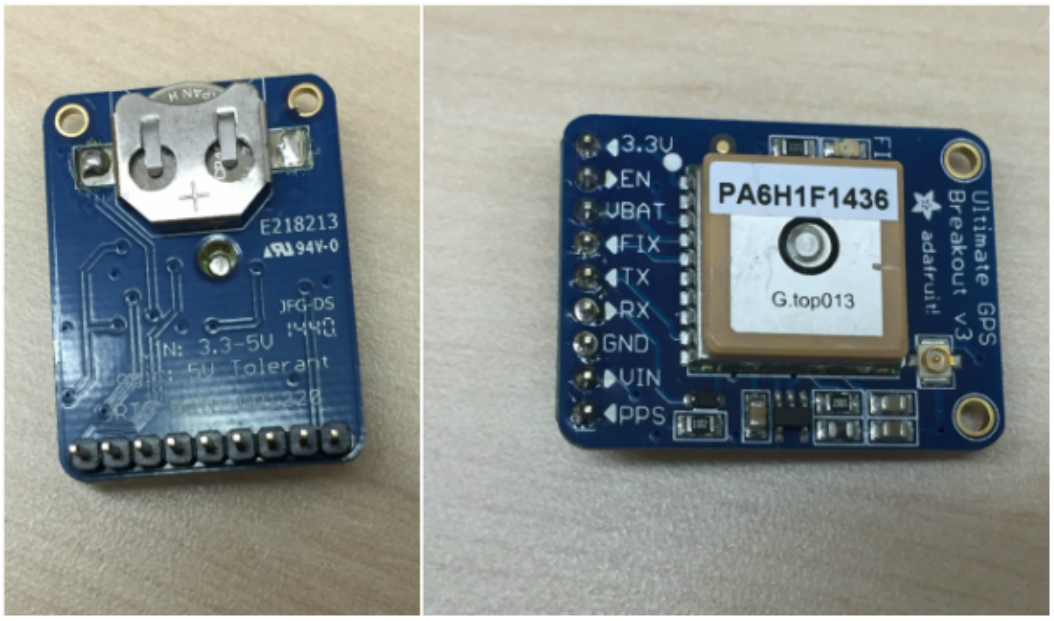
E218213
94V-0
JFG-DS
N: 3.3-5V
5V Tolerant
3.3V
EN
VBAT
FIX
TX
RX
GND
VIN
PPS
PA6H1F1436
G.top013
Ultimate GPS
Breakout v3
adafruit!

Fig. 4.1

5 Conclusion

> *TODO.*
>
> — **TODO**
> (TODO)

Bibliography

[1] Federal disability reference guide. Human Resources and Skills Development Canada, Gatineau, Québec (2012)

[2] Berge, C., Minieka, E.: Graphs and hypergraphs, vol. 7. North-Holland publishing company Amsterdam (1973)

[3] Bonér, J., Farley, D., Kuhn, R., Thompson, M.: The reactive manifesto (September 2014), http://www.reactivemanifesto.org

[4] Foundation, R.P.: Raspberry pi 2 model b (2016), https://www.raspberrypi.org/products/raspberry-pi-2-model-b/

[5] GitHub: Github · where software is built (2016), https://github.com/

[6] Griffith, E.: First github, now meteor: Andreessen horowitz backs another developer favorite (July 2012), https://pando.com/2012/07/25/first-github-now-meteor-andreessen-horowitz-backs-another-developer-favorites/

[7] Joyner, D., Van Nguyen, M., Cohen, N.: Algorithmic graph theory. Google Code (2010)

[8] Kelsen, K.: Unleashing the power of digital signage: content strategies for the 5th screen. CRC Press (2012)

[9] Lardinois, F.: Meteor acquires yc alum fathomdb for its development platform (October 2016), http://techcrunch.com/2014/10/07/meteor-acquires-yc-alum-fathomdb-for-its-web-development-platform/

[10] Meteor: (2016), https://www.meteor.com

[11] Mongo: (2016), https://www.mongodb.org

[12] Node.js: (2016), https://nodejs.org

[13] O'Reilly, T.: Design patterns and business models for the next generation of software (September 2005), http://www.oreilly.com/pub/a/web2/archive/what-is-web-20.html

[14] Pi, R.: Raspberry pi. Raspberry Pi 1 HDMI 13 Secure Digital 34 Universal Serial Bus 56 Python (programming language) 84 p. 1 (2012)

[15] Satish, K., Viswanadham, Y., Priya, I.L.: Money to atm–fake currency detection

[16] Sharma, S., Tim, U.S., Wong, J.S., Gadia, S.K., Sharma, S.: A brief review on leading big data models. Data Science Journal 13, 138–157 (2014)

[17] Skybreak: (2012), http://info.meteor.com/blog/skybreak-is-now-meteor

[18] Strauch, C.: NoSQL Databases. Stuttgart Media University (2012)

[19] Topcoder: Data science tutorials (2016), https://www.topcoder.com/

[20] Vujovic, V., Maksimovic, M.: Raspberry pi as a wireless sensor node: Performances and constraints. In: Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2014 37th International Convention on. pp. 1013–1018. IEEE (2014)

[21] WebRTC: Webrtc home | webrtc (2016), https://webrtc.org/

[22] Wit.ai: Wit — natural language for developers (2016), https://wit.ai/

[23] Wong, G.: Build your own prototype raspberry pi minicomputer. Ubergizmo, November 2 (2011)