An Extensible Software Infrastructure for Computer Aided Custom Monitoring of Patients in Smart Homes

Ritwik Dutta

Archbishop Mitty High School, Junior Year Student, San Jose, CA 95129 USA

This paper describes the design from scratch of a self-contained health dashboard software system that provides customizable data tracking for patients in smart homes. The system comprises a front-end and a back-end component. Built with HTML, CSS, and JavaScript, the front-end allows adding users, logging into the system, editing profiles, selecting metrics, and specifying health goals. The back-end consists of a NoSQL Mongo database, a Python script, and a Python SimpleHTTPServer. The database stores user profiles and health data in JSON format. The Python script makes use of the PyMongo driver library to query the database, and displays formatted data as a daily snapshot of user health metrics against target goals. Any number of standard and custom metrics can be added to the system, and corresponding health data can be fed automatically, via sensor APIs or manually, as text or picture data files. A real-time METAR request API permits correlating weather data with patient health, and an advanced query system is implemented to allow trend analysis of selected health metrics over custom time intervals.

***Index Terms*— Flask, Java, JavaScript, health monitoring, long term care, Mongo, Python, SimpleHTTPServer, smart home.**

I. Introduction

HOME automation is the next frontier in healthcare. Take a simple case: if a smart home system could keep track of when a person gets up and goes to bed, and follow him or her around during the day, it could furnish valuable information about how the person is doing. In the case of long-term-care patients, home automation is particularly useful because it can provide valuable means for health monitoring without moving the patients to a health-care facility.

In an intelligent environment like a smart home, monitoring can range from simple tracking of motion and sleep data to complex analysis of behavioral patterns via machine learning algorithms (1). Conference and journal papers abound with details on concepts and technologies to monitor behavioral patterns and health status (2) (3) (4) (5) (6). Taking the cue from this growing consciousness, government, academia, and business professionals have responded to the situation by designing educational programs, personal monitoring devices, and technology that transmits patient health data to the care provider (7). For example, Apple recently unveiled new software platforms during its Worldwide Developers Conference that will organize data from the growing number of mobile medical applications and smart home gear on its iOS operating system (8). And ADT Security Services is now offering medical alert monitoring as well (9).

However, even without sophisticated devices, plugins and apps, a lot can be achieved by connecting simple sensors, monitoring personnel, and caregivers. What is needed is a simple-to-use and extensible software infrastructure that allows registering patients, setting goals, collecting data (manually or from sensors), displaying patient data in an easy-to-understand format on a dashboard, and allowing the caregiver to act based on the data collected. The current project tries to achieve exactly that.

The goal of our project has been to create a fully functional, free, and open source software system for monitoring patients. We wanted to design and develop a system that anyone can use. Easy scalability, simple deployment, and clear and accessible code have been high priorities for the implementation as well. Finally, we wanted to evaluate whether such a framework can help research into how data from many simple sensors can be combined to yield valuable information. The paper describes our journey to realize the goals.

The paper is organized as follows. Section II describes the overall front-end interface design of the project. Section III provides an overview of the software architecture and organization. In Section IV, we go into the software implementation details. Section V draws the conclusions and talks about future work.

II. Project Design

Simply stated, the design provides functionality for patient tracking through a variety of different data types. Each type of data that is tracked is called a "metric". There are different types of metrics --- ones that provide information directly related to a patient (*e.g.*, motion, sleep, photos, *etc.*), and others that yield information about the environment (*e.g.*, weather) that affects the patient. We have specifically chosen motion, sleep, photos, and weather as the metrics for our first implementation.

There are two different classes of metrics that we have designed – *standard* and *custom*. Standard metrics are obtained from files in a standardized data format. Custom metrics receive data from any programmatically accessible source. All of the information and data about the various metrics are viewed through three different web pages: *login*, *dashboard*, and *query*.

The "login" page, shown in Figure 1, provides the functionality for adding and deleting users, as well as the functionality to log into the project to view patient data. The *Add User* button gives a series of options to create a new user with specific health targets. The *Delete User* and *Login* dialogs show a list of users where one can delete or log in as a user.

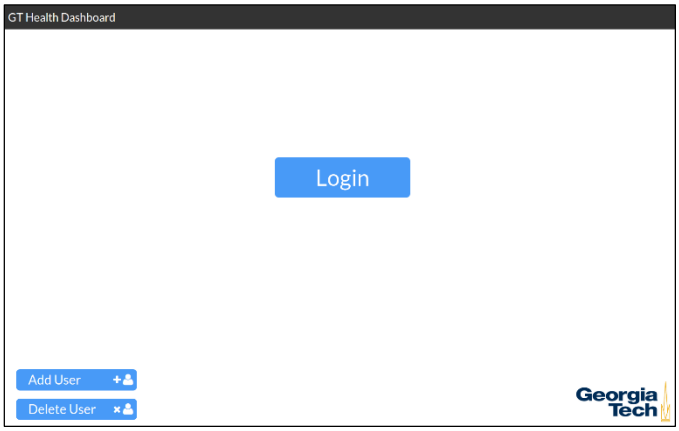

Figure 1. Login page

The second page is the dashboard page as illustrated in Figure 2. This page provides the functionality to view all of the metric data for a single day. The metrics can be displayed in four different ways: percentage, graph, raw text/numbers, and photo. Each metric has a display format field which determines how it is to be displayed on the dashboard page.

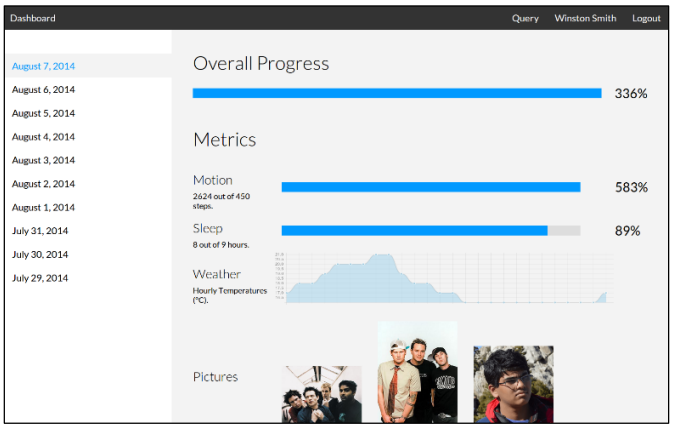

Figure 2. Dashboard page

The dashboard page display only a single day's (as selected) worth of data at any one time; the user can view data for the last 10 days. In order to view data trends over multiple days, however, the user must go to the query page. The dashboard page is most useful for obtaining daily details --- either specific (*e.g.*, motion or sleep) or general (*e.g.*, pictures or weather).

The query page, shown in Figure 3, is where the user can select several metrics and a date range. Each metric's daily data is processed into a single number, and then each metric is graphed over the selected date range. This page can be used to view trends over time and make predictions about the future.

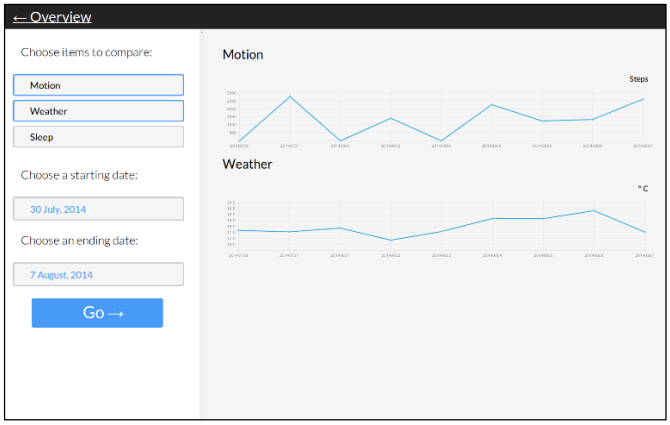

Figure 3. Query page

III. SOFTWARE OVERVIEW

The project essentially consists of a front-end and a back-end. The front-end includes the three different webpages described before to provide all the functionality on the user side. The web pages are written using a combination of HTML, CSS, and JavaScript. The back-end includes a Mongo database (10), a Flask server (11), and a Python SimpleHTTPServer (12).

The database stores all of the user metadata and the data for the user metrics. MongoDB was chosen over an SQL type database to make it scalable and fast. Documents are stored in the database in JSON (JavaScript Object Notation) format, and can be easily searched with complex queries.

The Flask server processes the requests made by the front-end, and returns the corresponding data from the database back to the webpage. Flask was chosen over the more common Django web server due to the fact that it is lighter and simpler to use.

The Python SimpleHTTPServer is a bare-bones HTTP server and was chosen over more typical web servers such as nginx or Apache because it has no external dependencies; only thing needed is that Python should be installed.

The project homepage is currently hosted on GitHub Pages (13) and the code is hosted on the GitHub public VCS system. The gzipped version of the project is 15.7 megabytes, and the extracted directory is 40.8 megabytes. To run the application, the project needs to be downloaded and extracted first. One can then go into the project directory and type "./run start" to start the application; running "./run stop" will stop it. Following the descriptions given in the paper one can not only download and execute the code, but add sensors and custom metrics as well.

IV. IMPLEMENTATION

A. *Front-end*

The entire front-end of the project has been written from scratch, without using any frameworks such as Bootstrap or Foundation. An entirely new implementation was chosen over using a framework due to the greater flexibility afforded by such a fully custom design. The front-end is largely empty static content that is populated by data from the database. To obtain

data from the database, the front-end JavaScript sends AJAX requests to the Flask server, which processes the requests, pulls the requested data from the database, formats the data, and returns it to the calling script. Once the data is obtained, the JavaScript formats the data further into HTML and inserts it into appropriate pages on the front-end.

There are several different formats to display the data. For the *percent* format, a progress bar is displayed by setting a width of a child <*div*>[1] element relative to a parent <*div*> element, and a percentage is displayed to the right. For the *raw* format, the data is simply dumped into a text box, and if the text is formatted with HTML markup, the formatting will be visible. For the *picture* format, images are displayed in a lightbox (that expands when clicked); the lightbox plugin selected is FancyBox that is easy to use and compatible with many different browsers. For the *graph* format, the data is displayed as a line graph; the graph library chosen for this is Chart.js due to its ease of use and responsive[2] nature.

B. Back-end

Flask is a framework for a full-fledged Python web server, but in the current project, it is only used for processing the back-end requests from the front-end JavaScript. The script for the Flask server itself is only partially custom-implemented for the project; the cross-domain request code was obtained from a public Flask snippet (14). The Flask server script contains only request-processing functionality. The script includes several other custom-written modules for additional functionality.

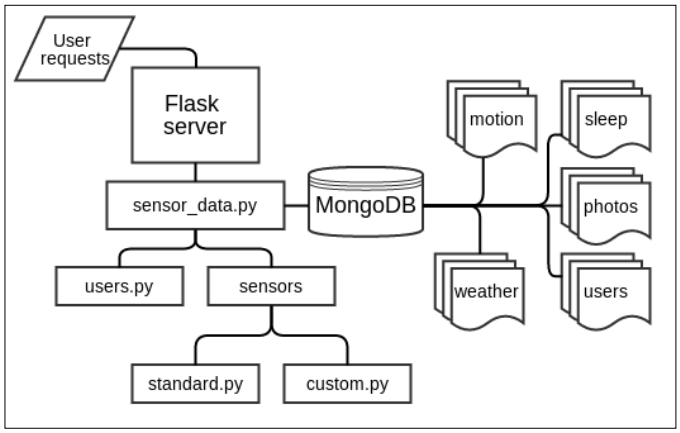

Figure 4. Back-end component connectivity

The back-end component connectivity is illustrated in Figure 4. The **sensor_data.py**[3] module contains the *methods* for logging all of the data, as well as the class to run the process that automatically logs the data. The **standard.py** and **custom.py** modules are included in the **sensors** module and they contain the *sensor* classes to log data for both the *standard* and the *custom* metrics. The **users.py** module contains the two classes that are used to add and delete users.

As mentioned earlier, there are two types of metrics: standard and custom. Every standard metric is automatically read and processed into the database from a file, but the custom metrics require a specific class per metric and a new clause in the control flow statement in **sensor_data.py**. There is no actual database handling in the sensor classes because it makes more sense to offload all database operations to an external class to ensure a uniform document format for both standard and custom metrics. All of the sensor classes return sensor data using a common function, **getMetricData**, and a method called from **sensor_data.py** then inserts that data into the database, keyed by date, metric, and username. The Python scripts use the PyMongo driver to interface with the database. The automatic logging runs on a thread independently of the Flask server thread. The Python back-end is multithreaded in order to ensure that any problems with the automatic data logging will not halt the process to deliver data from the database.

The database software used for the project is MongoDB. It is a NoSQL-type database that stores documents in JSON format. The structure of the database is relatively flat. At the highest level, there is the "data" database, which has several collections for the various different pieces of data (*e.g.*, users, motion, pictures, sleep, *etc.*). Each collection contains a series of *documents*[4] in JSON format, keyed by username, date, and metric. MongoDB supports key-value pairs for queries, which makes it easy to search for and find data keyed by date, metric, and username (or, any combination of the three). Each user in the database has his/her metadata (name, username, metrics) stored in a user object which is keyed by the username. Every username is dynamically generated by appending the epoch time to a whitespace-stripped version of the user's full name. This method ensures that every username is unique and prevents a name collision. All metrics are stored as JSON keyed by metric name in a larger JSON object that contains all the metrics. The JSON format is very easy to parse in both Python and JavaScript. A sample user object is shown in Figure 5.

The standard metrics in the user object contain a path that defines the directory for files that correspond to the data for that metric. The custom metrics have no such path, and data can be obtained from any programmatically accessible source. For the two custom metrics currently implemented, motion and sleep, the data is retrieved from a third party source.

For the **motion** metric, the Jawbone UP band (15) is used to obtain motion data that Jawbone stores in the cloud. Jawbone's proprietary API provides the hourly number of steps a user has walked with the device on. The protocol used to authenticate with the Jawbone API is known as OAuth. The OAuth process requires an HTTPS server callback. In practice, an HTTPS certificate is very expensive, so a temporary setup is created --- a one-time process --- to generate an authentication code for the API used to pull the data from the cloud. The data is returned in JSON format. The steps are retrieved from the "steps" field of the JSON object and inserted into the database keyed by username, date, and metric (motion). The code corresponding to a single day's worth of data is shown in Figure 6.

[1] A <*div*> is an HTML element used to display floating blocks that may or may not contain text.

[2] The library generates graphs that fit many different screen resolutions.

[3] The .py file extension denotes a Python script.

[4] A *document* in MongoDB is a specific piece of data.

```
{
 "name":"Ritwik Dutta",
 "username":"ritwikdutta1406421674",
 "metrics":{
   "sleep":{
     "source":"file",     … where the data will be pulled from
     "format":[
       "percentage",      … format that  data will be displayed in
       8,                 … target goal for percentage-type data
       "hours"            … unit of measurement for  data
     ],
     "path":[
… specific folder where  data is to be retrieved from
       "mongo/data/db/",
       "sleep"
     ]
   },
   "weather":{
     "source":"custom",
     "format":[
       "graph",
       "° C"
     ],
     "type":"weather",
     "location":"ksfo"
   },
   "pictures":{
     "source":"file",
     "format":[
       "picture"
     ],
     "path":[
       "mongo/data/db/",
       "pictures"
     ]
   },
   "motion":{
     "source":"custom",
     "format":[
       "percentage",
       400,
       "steps"
     ],
     "type":"jawbone",
     "auth":"b6_3pfGGwEjReOXSnWIyQO0-
AI13wvvmyZNaiuNHtPrR6_kAcLtg_W1PaWiav9FR8EvaJSu
mcI0GoYT-
V9UbpVECdgRIo_GULMgGZS0EumxrKbZFiOmnmAPChBP
DZ5JP"
   }
  }
}
```

Figure 5. User object code example

```
   {
    "time_completed": 1406874960,
    "xid": "PUDl9aE5chyt0sTPkkpIcA",
    "title": "2,781 steps",   … total steps walked in a day
    "type": "move",
    "time_created": 1406851800,
    "time_updated": 1406924420,
    "details": {
      "active_time": 1487,
      "tzs": [
        [
          1406851800,
          "America/Los_Angeles"
        ]
      ],
      "wo_count": 0,
      "wo_longest": 0,
      "bmr": 1703.7847,
      "steps": 2781,
      "bg_calories": 100.284996,
      "hourly_totals": {
        "2014073117": {
          "distance": 1176,
          "active_time": 800,
          "calories": 58.641,
          "inactive_time": 1200,
          "longest_idle_time": 660,
          "steps": 1548,
          "longest_active_time": 756
        },
        "2014073118": {
          "distance": 716,
          "active_time": 576,
          "calories": 33.747,
          "inactive_time": 1920,
          "longest_idle_time": 660,
          "steps": 1016,
          "longest_active_time": 287
        },
      },
      "bmr_day": 1703.7847,
      "wo_active_time": 0,
      "sunrise": 0,
      "distance": 2055,
      "tz": "America/Los_Angeles",
      "longest_active": 756,
      "longest_idle": 9240,
      "calories": 130.12175,
      "km": 2.055,
      "inactive_time": 25920,
      "wo_calories": 0,
      "wo_time": 0,
      "sunset": 0
    },
    "date": 20140731,
    "snapshot_image":
  "/nudge/image/e/1406874969/PUDl9aE5chyt0sTPkkpIcA/U
DqByBVIzAY.png"
  }
```

Figure 6. Jawbone object code example

For the **weather** metric, the weather data is retrieved using a government-provided API that returns the past 24 hours of weather data for a given airport code. The data in this case is returned in METAR format, which is commonly used to store weather information. A sample METAR string is shown in Figure 7. The relevant data is the temperature (27) in degrees

```
KFSO 011955Z AUTO 19003KT 10SM CLR 27/17 A3012 RMK AO2 T02740167
```

Figure 7. Sample METAR string

Celsius (°C). The METAR string is processed, whereby the temperature is extracted and appended to an array. After parsing every METAR string, the array contains the past 24 hours worth of temperature data. This data is inserted into the database keyed by username, date, and metric (weather).

C. Adding new metrics

This section talks about how a new metric can be added beyond what exists in the project. A metric needs to be added as a key and a value in the metrics object in the user object. As mentioned earlier, the metric can have two possible sources on the back-end, which define how the data is collected. The metric can either have a *custom* data source, where everything has to be custom-implemented, or a *file* data source, where everything is automatically pulled from a directory. The path to the directory for a metric is currently defined as **/mongo/data/db/user/<username\>/<metric\>**.

The metric can have four possible formats on the front-end, which define how the data is displayed. They are:

- Percentage: where the metric is displayed as a progress bar towards a target
- Picture: where the metric is displayed as a series of photos
- Graph: where the metric is displayed as a graph
- Raw: where the data is displayed as pure raw numbers or text

1) File metric

To add a new *file* metric, a new metric entry needs to be created in the user object with the selected options for display format and path. The directories for the user and associated metrics will be automatically created, after which the folders should be populated with data.

2) Custom metric

To add a new *custom* metric, new metric entry needs to be created in the user object with the **source** set to custom and the selected options for display format as well as any other fields deemed necessary (*e.g.*, a location for weather, or an API key for the Jawbone UP). A custom Python class needs to be implemented to pull and process the data, and inserted into the file **gt-dashboard/server/sensors/custom.py**. The custom class for the Jawbone UP is shown in Figure 8.

```
class jawboneLib:
    def __init__(self, userAuthorizationToken):
        self.userAuthorizationToken = userAuthorizationToken
    def getMetricData(self):
        apiAuthorizationHeaders = {'Authorization' : 'Bearer ' + self.userAuthorizationToken.encode('ascii', 'ignore') }
        userJawboneData = requests.get('https://jawbone.com/nudge/api/v.1.1/users/@me/moves', headers = apiAuthorizationHeaders)
        userJawboneDay = json.loads(userJawboneData.text)['data']['items'][0]
        userJawboneDaySteps = 0
        currentDate = getFormattedTime("%Y%m%d")
        if (str(userJawboneDay['date']) == currentDate):
            userJawboneDay = userJawboneDay['details']['hourly_totals']
            for userJawboneHour in userJawboneDay.keys():
                userJawboneDaySteps += userJawboneDay[userJawboneHour]['steps']
    return userJawboneDaySteps
```

Figure 8. Jawbone UP custom class

Next, **gt-dashboard/server/sensor_data.py** is to be edited to insert the custom call to the class into the if/elif statements shown in Figure 9.

```
elif userMetricsList[metric]['source'] == 'custom':
        customData = ""
        if userMetricsList[metric]['type'] == 'jawbone':
            customDataObject = custom.jawboneLib(userMetricsList[metric]['auth'])
            customData = customDataObject.getMetricData()
        elif userMetricsList[metric]['type'] == 'weather':
            customDataObject = custom.weatherLib(userMetricsList[metric]['location'])
            customData = customDataObject.getMetricData()
```

Figure 9. Custom metric control flow

The only thing that the custom metric class needs to return is raw data. It will be automatically tagged by date, username, and metric so that it can later be found in the database.

D. Creating custom displays

It is also possible to create custom display formats on the front-end. However, this process requires knowledge of **HTML**, **CSS**, and **JavaScript** for the front-end in addition to **Python** for the back-end.

A function needs to be written in this case to generate the markup for the desired format, append it to the table of metrics, and set it to the correct value. This should be consistent with the current theme. The functions for displaying metrics are in **gt-dashboard/web/js/metrics.js**. The code snippet for displaying a percentage is implemented as shown in Figure 10.

```
function addPercentMetric(metricName, metricDescription, metricPercent) {
    function genPercentMarkup(metricName, metricDescription) {
        var metricMarkupTemplate = ['<tr class="metrics element container" data-metric="',
            '"><td class="metrics element info"> <div class="metrics element title">',
            '</div> <div class="metrics element description">',
            '</div></td><td class="metrics element progress"><div class="metrics element base"><div class="metrics element fill ',
            '"></div></div></td><td class="metrics element number ',
            '"><div class="metrics element percent ',
            '"></div></td></tr>'
        ];
        var metString = metricMarkupTemplate[0] +
            metricName + metricMarkupTemplate[1] +
            metricName + metricMarkupTemplate[2] +
            metricDescription + metricMarkupTemplate[3] +
            metricName + metricMarkupTemplate[4] +
            metricName + metricMarkupTemplate[5] +
            metricName + metricMarkupTemplate[6];
        return metString;
    };
    function setPercent(metricName, metricPercent) {
        $(".metrics.element.percent."+ metricName).text(metricPercent.toString() + "%");
        metricPercent = (metricPercent > 100) ? 100 : metricPercent;
        $(".metrics.element.fill." + metricName).animate({
            "width": metricPercent.toString() + "%"
        }, 250);
    }
    var generatedMetric = genPercentMarkup(metricName, metricDescription);
    $(".metrics.table").append(generatedMetric);
    setPercent(metricName, metricPercent);
}
```

Figure 10. Display code example

The switch statement in **gt-dashboard/web/js/dash.js** is to be also edited to add in the custom display format. The code snippet where this is done is illustrated in Figure 11.

V. Conclusion

We set out to create a free and open source system for patient monitoring. Combining cutting-edge front-end and back-end technologies, we created the piece of software from scratch that met our goals and performed really well. The distinguishing features of our software are:

- MIT licensed and freely redistributable
- Absence of any closed or proprietary code
- Easy expandability for additional metrics and sensors
- Simple download-and-run deployment

```
switch (requestedMetricFormat[0]) {
    //Show percentage
    case "percentage":
        tempDescription= loggedInUserData[requestedMetric].toString() +
            ' out of ' + requestedMetricFormat[1] + ' ' +
            requestedMetricFormat[2] + '.';
        requestedMetricPercent=Math.round(100* loggedInUserData[requestedMetric] /
            requestedMetricFormat[1]);
        //Add to array to determine final percentage
        percents.push(requestedMetricPercent);
        addPercentMetric(requestedMetric,
            tempDescription,
            requestedMetricPercent);
        break;
    //Show pictures
    case "picture":
        addPhotoMetric(requestedMetric,"", loggedInUserData[requestedMetric]);
        break;
    //Show raw data
    case "raw":
        addRawMetric(requestedMetric,"", loggedInUserData[requestedMetric]);
        break;
    case "graph":
        addGraphMetric(requestedMetric,
            (requestedMetric == "weather") ? "Hourly Temperatures (&deg;C)." : "",
            loggedInUserData[requestedMetric]);
        break;
}
```

Figure 11. Metric display control flow

We are planning to make the following improvements to increase the quality of the project:

- More sophisticated template system for custom metrics
- Addition of new sensors, particularly network cameras
- Editable user configuration for goals
- Time tagged metric data
- Prediction using trends
- Moving from application download-and-run deployment to preconfigured image deployment

VI. Acknowledgment

I would like to thank Prof. Marilyn Wolf for her guidance and for providing me with the opportunity of a summer project at the Department of Electrical and Computer Engineering at Georgia Tech.

VII. References

1. **Prafulla N. Dawadi, Diane J. Cook, and Maureen Schmitter-Edgecombe.** Automated assessment of cognitive health using smart home technologies. [Online] http://www.eecs.wsu.edu/~cook/pubs/thms12.pdf.
2. *Health-status monitoring through analysis of behavioral patterns.* **Barger TS, Brown DE, and Alwan M.** 35(1), s.l. : IEEE Trans Syst Man Cybern, 2005, Vol. Part A .
3. *Unobtrusive sensing of activities of daily living: a preliminary report.* **M, Carter J and Rosen.** s.l. : Proc First Joint BMES/EMBS Conf, October 13-16, 1999.
4. **Cook DJ and Das SK.** *Smart environments: technologies, protocols and applications.* . 2004 : Hoboken: John Wiley and Sons.
5. **Helal A, Mokhtari M, Abdulrazak, and B. Hoboken.** *The engineering handbook on smart technology for aging, disability and independence.* s.l. : John Wiley and Sons, 2007.
6. *Long-term remote behavioral monitoring of elderly by using sensors installed in ordering houses.* **Ogawa M, Suzuki R, Otake S, Izutsu T, Iwaya T, and Togawa T.** s.l. : Proc Int IEEE-EMBS Special Topic Conf Microtechnologies in Medicine and Biology, May 2–4, 2002.
7. **Abdelsalam Helal, Mark Schmalz, Diane J. Cook.** Smart Home-Based Health Platform for Behavioral Monitoring and Alteration of Diabetes Patients. *Journal of Diabetes Science and Technology.* [Online] http://www.ncbi.nlm.nih.gov/pmc/articles/PMC2769843/#b10.
8. **Apple Unveils Health, Smart Home Software.** ***US News.*** **[Online] http://www.usnews.com/news/articles/2014/06/02/apple-unveils-health-smart-home-software.**
9. Medical Alert Monitoring. ***ADT.*** **[Online] http://www.adt.com/medical-alarm.**
10. Agile and Scalable. ***mongoDB.*** **[Online] http://www.mongodb.org/.**
11. Flask web development, one drop at a time. ***Flask.*** **[Online] http://flask.pocoo.org/.**
12. 20.19. SimpleHTTPServer — Simple HTTP request handler¶. ***Python - Documentation » The Python Standard Library » 20. Internet Protocols and Support »*****. [Online] https://docs.python.org/2/library/simplehttpserver.html.**
13. Dutta, Ritwik. Welcome to the project page for the Georgia Tech Health Dashboard! ***Georgia Tech Health Dashboard.*** **[Online] http://gtd.ritwikd.com/.**
14. Ronacher, Armin. Decorator for the HTTP Access Control. ***Flask Snippets.*** **[Online] http://flask.pocoo.org/snippets/56/.**
15. UP - A SMARTER UP A FITTER YOU. ***JAWBONE.*** **[Online] https://jawbone.com/up.**