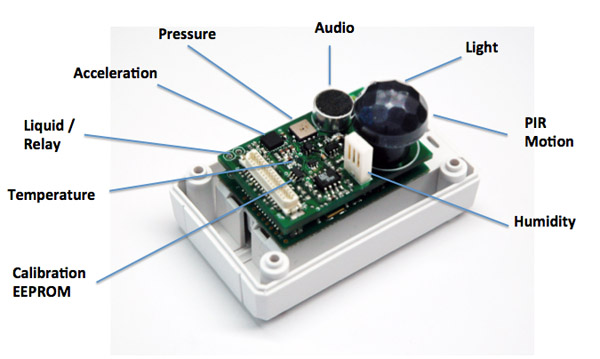
SenWeb

**Technical Report for Project 3**

**A Web Application for Managing the CMU Sensor Service Platform**



Jia Zhang

**& Design**

**Architecture**

**Fall 2013**

Instructor:

2013

Abhi Trivedi, Daniel Brown, Geoff Schaeffer, Rob Black

Team Mercury

12/4/2013

Contents

List of tables 5

Table of figures 5

Introduction 6

Project Overview 6

Objectives 6

Scope 6

Concerns 6

Constraints 6

Design Considerations 7

Design Patterns 7

UI Wizard 7

Facade 7

Data Access Object 7

Builder 8

Composite 8

Model-View-Controller 8

Web Framework 9

Architecture 11

Contextual View 11

Funtional View 12

Logical View 13

Information View 14

System Implementation 15

Query Layer 15

Query Manager 16

Object Layer 16

Sensor Type 16

Device Type 17

Sensor 17

Device 17

Sensor Reading 17

Presentation Layer 17

Documentation and Testing 18

Tutorial 19

Dashboard 20

Sensors 21

Devices 22

Device Agents 23

Sensor Types 24

Device Types 25

Log In 25

ATAM 26

ATAM Utility Tree 26

Risks, Sensitivities, and Tradeoffs 27

Future Work 28

Recommendations for Sensor Service Platform API 28

Conclusion 30

Bibliography 31

# List of tables

Table 1: Framework selection matrix 10

# Table of figures

Figure 1: UML Context Diagram 11

Figure 2: UML Functional (Component) Diagram 12

Figure 3: Logical View 13

Figure 4: Information View 14

Figure 5: Architectural Overview 15

# Introduction

Over the last seven weeks, Team Mercury has been working on developing a web application, which provides a convenient, intuitive, and easy-to-use user interface for the CMU Sensor Service Platform, which we codenamed: named SenWeb. In the following sections, we will describe the major processes that our team used in developing this application. We will also provide technical artifacts with the intent of providing information for all stakeholders. Since we were unable to implement all the features we intended to, we will also make recommendations so that future teams working on improving the application will have some guidance as to which areas to consider.

# Project Overview

In this section, we present an overview of the SenWeb development project from the perspective of our Team.

## Objectives

The SenWeb application is designed to provide an intuitive, easy to use interface for managing the Sensor Service Platform. It provides data from the devices that the Sensor Service Platform supports as well as providing a way for users to manage those devices. Additionally, it makes use of the Sensor Service Platform API, a REST API for interacting with the platform.

## Scope

The scope of the project is to re-write the application located at <https://cmu-sds.herokuapp.com/> in a Java web application. Additional functionality will be added to this new application as the API provides more ways for users to interact with the Sensor Service Platform. The current scope of the application is to match the functionality provided in the existing application, limited only by what data the API can provide.

## Concerns

While RESTful APIs make services more available to the general public, they also constrain the ways in which users can interact with data. Care must be taken to ensure that the API is sufficiently expressive to meet user demands and that it is efficient in providing data to users.

## Constraints

Because SenWeb depends heavily upon the Sensor Service Platform API for data, the availability and usability of SenWeb is tied directly to the availability of the API. Additionally, it should be noted that clients of the API such as SenWeb may exhibit slow performance while performing synchronous requests to the API due to the overhead of communicating over HTTP.

# Design Considerations

Having discussed the overview of the project, we will now discuss some of issues we considered during our deliberations for the design of the application. We begin with design patterns.

## Design Patterns

Given the nature of the SenWeb application – it is a web based application-some design patterns readily lend themselves for usage while other were not so obvious. In this section, we discuss the design patterns used and provide the necessary rationale for selecting each one.

### UI Wizard

Starting from the presentation layer, the major design pattern that we picked for SenWeb is the Wizard pattern. The tradeoffs in using the Wizard pattern are that, on the plus side, it minimizes user errors and misunderstanding by providing a clear flow and prompt feedback throughout the process. However, the drawback to this pattern is that it forces the user to follow steps in the process as we have defined them. We believe that the clarity and ease-of-use provided outweighs the fact that users don’t have full freedom in how they interact with the application.

### Facade

Facades are used at multiple levels within SenWeb. The project could be said to be a series of nested facades that insulate lower layers from the underlying API.  The implementation of our Query package provides a facade for the business logic to interact with the APIs. The web application as a whole provides a facade for a user to interact with the underlying APIs. Use of this pattern is highly effective. There are no real drawbacks to using facades, except for some additional time spent in up-front design. The benefits this pattern provides are a clarity of implementation, proper separation of concerns, and hiding of implementation details.

Facades also provide a convenient place to provide mocks for testing and development.

### Data Access Object

The Data Access Object (DAO) pattern is a way of separating the connection with an underlying data source from the rest of the business logic code (Data Access Object, 2013). In our case for SenWeb, the HANA API specific DAO is what is hiding behind our Query Facades. In the future, if the HANA API moves to a different protocol than HTTP, then only the HTTP-specific DAO objects would need to change for SenWeb to properly function. All levels of the code higher than the DAO could function unchanged.

The DAO pattern helps us to modularize our code by enforcing a proper separation of concerns. The downside is that this set up requires the creation and maintenance of more code. However, this additional code is what allows SenWeb to be flexible and modular, so it is worth paying the price.

### Builder

SenWeb uses the Builder pattern to manufacture API requests inside of the Query package. Using the Builder pattern allows us to reduce implementation errors in that the objects created will be correctly constructed. It also helps to reduce duplication as complex objects can be build out of simple pieces. This comes at the cost of being more verbose and having a slightly higher memory use, as the Builder needs to copy pieces into the whole.

### Composite

SenWeb uses the Composite Pattern to create a generic QueryResponse to QueryRequests. This pattern helps to insulate the business logic from the details of the API (Manglick, 2010). Our composite is a structure consisting of Arrays, Values, and a Collection of Key-Value mappings. It is presently being used to represent the JSON that is returned from the HANA API. However, if the API changes its return type in the future (to XML, Object-based, String-based, etc.) this Composite can still represent it. The only changes needed would be to the code that builds the Composite.

The price for this flexibility is a higher cognitive load on the development side to make sure that they don’t get lost in the structure when creating it.

### Model-View-Controller

The Model-View-Controller (MVC) Pattern is the standard pattern for building web applications.  It separates presentation from the underlying data and provides a high degree of maintainability for a codebase. The downside is that this pattern requires more effort to set up correctly, however, modern development frameworks provide most of this heavy lifting, so the benefits can be obtained at a lower cost.

## Web Framework

Given that the end product of SenWeb would be a web application, our first major task was to select a framework. The advantages of developing a web application within a framework are numerous. Our team was most interested in the built-in support for the MVC pattern, the ease of development, and provided web security.

Given the project constraints of a transition from Ruby on Rails to Java, the following frameworks were considered:

* Play - A newer Java/Scala Framework
* JRuby - Ruby interpreted in a Java Virtual Machine with the ability to use Java as well
* Spring - A well-established Java MVC framework
* Grails - A newer Java MVC framework that takes most of its convention cues from Ruby on Rails

We evaluated the frameworks on the criteria that we determined to be important.

Heroku Compatibility - For ease of deployment

* Language Extensibility - For potential changes in the future
* Ease of Use - For speed of development
* The ability to leverage existing work - For speed of development
* Existing HANA API Framework - For potential close integration in the future
* Satisfying the project requirements - For actually building what the customer wants

Our comparison of the different frameworks is as follows:

Table : Framework selection matrix

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Play | JRuby | Spring | Grails |
| Heroku | Supported | Supported | Supported | Supported |
| HANA APIs | Written in Play | N/A | N/A | N/A |
| Language extensibility | Could shift to Scala | N/A | N/A | N/A |
| Ease of use | High ease of use | No teammember familiarity | High ease of use | No teammember familiarity |
| Leverage existing work | N/A | Potential to leverage existing Ruby work | N/A | N/A |
| Satisfies Project Requirements | Yes | No - project requires shift away from Ruby | Yes | More research needed |

Our evaluation resulted in our selection of the Play Framework. Play had the edge in Ease of Use, Extensibility, and the fact that the HANA APIs are currently using Play. JRuby would have allowed leveraging of previous work, but didn’t fit the project guidelines of transitioning to pure Java. Spring was considered to be easy to use, but didn’t have as many advantages as Play. None of the team members was familiar with Grails. Since speed of development was a primary concern, this led to Grails not being ranked highly.

As illustrated in the above table, Play provides several advantages and will serve the SenWeb project well going forward.

# Architecture

In this section, we discuss the four views of the architecture for SenWeb, These views together, are intended to provide as much information to stakeholders as possible, in a way concise format.

## Contextual View

The contextual view of a software system describes the relationships, dependencies, and interactions between the system and its environment (the people, systems, and external entities with which it interacts) (Rozanski & Woods, 2013). We present the contextual view of SenWeb in Figure 1: UML Context Diagram.

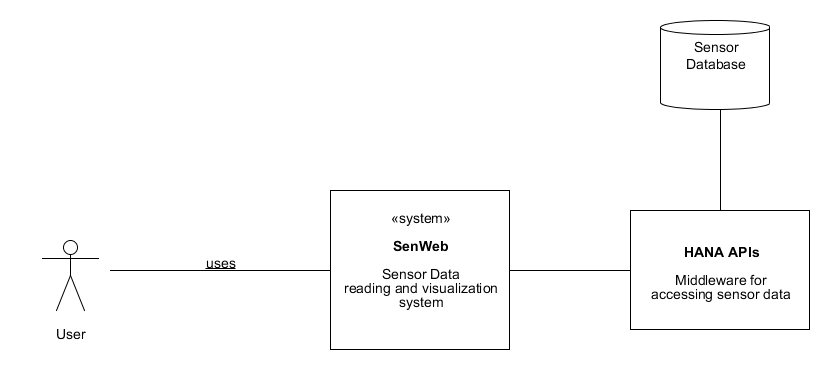


Figure : UML Context Diagram

## Funtional View

The functional view point describes the system’s runtime functional elements and their responsibilities, interfaces, and primary interactions (Rozanski & Woods, 2013). Thus it is used to document the functional capabilities of the system. We describe the functional view of the SenWeb in Figure 2.

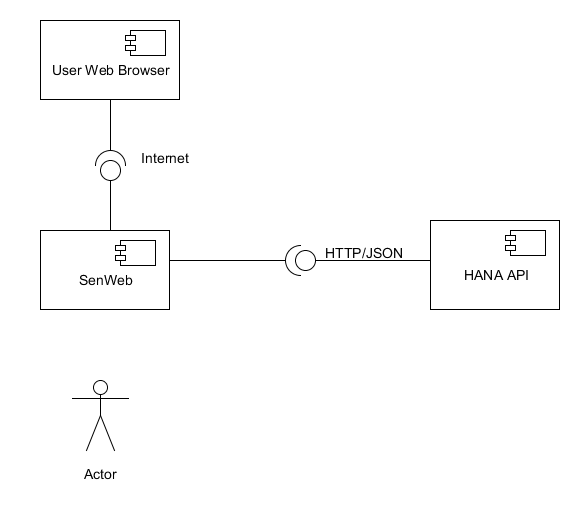


Figure : UML Functional (Component) Diagram

## Logical View

The logical view describes how various entities and services are linked together in the application and how they interact with each other in order to process information. Figure 3 describes the logical view of SecWeb application.

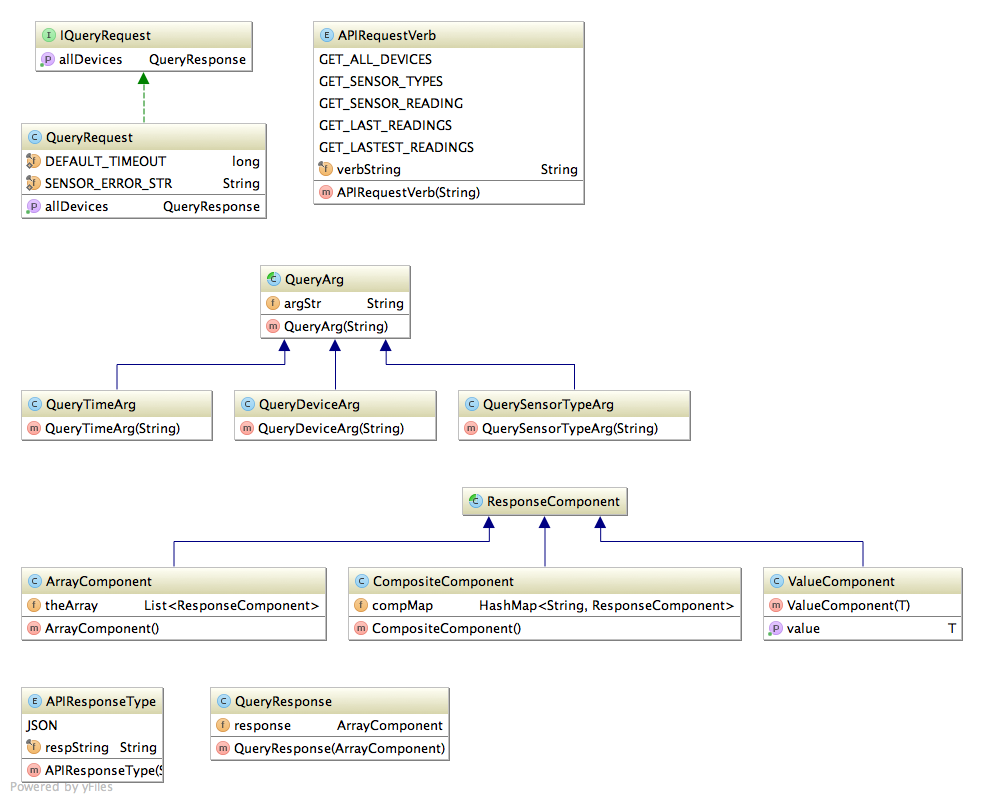


Figure : Logical View

## Information View

The information view primarily depicts the flow of information through the system; how this information is retrieved, stored and presented to the end user. Figure 4 describes the models and their relationship in the application.

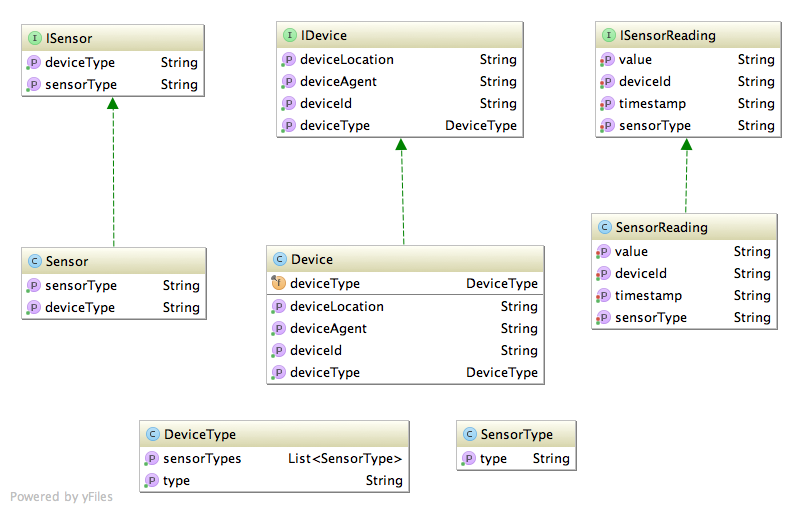


Figure : Information View

# System Implementation

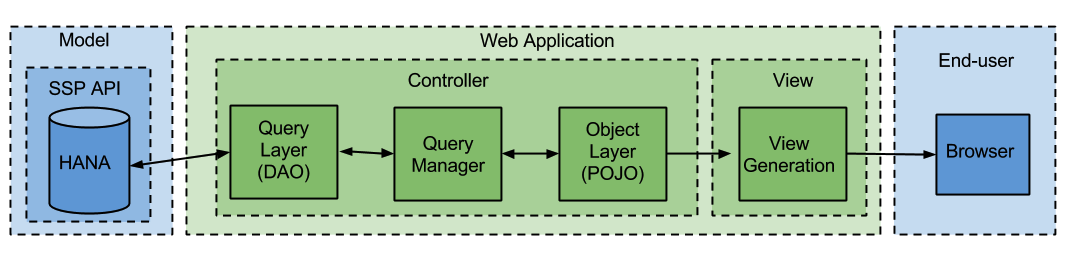
An overview of the final implementation of SenWeb looks like the following:

Figure : Architectural Overview

The design for SenWeb is highly modular, decouples the presentation from business logic and the business logic from data access, as well as promoting extensibility and reuse of components.

## Query Layer

The Query Layer insulates other modules from implementation details of the API by providing a Facade for a Data Access Object. The entry point into the Query Layer for the rest of the business logic is the IQueryRequest interface located in the /app/models/query directory of the project. IQueryRequest contains methods like:

* public QueryResponse getAllDevices();
* public QueryResponse getSensorTypes(QueryDeviceArg dType);

The IQueryRequest methods map directly to the HANA APIs. For example, getAllDevices maps to http://einstein.sv.cmu.edu/get\_devices/json and getSensorTypes maps to http://einstein.sv.cmu.edu/get\_sensor\_type/<”device\_type”>/json.

The return type of IQueryRequest methods is a QueryResponse. QueryResponse uses the Composite Pattern to provide a generic response to a query. Using a generic query response means that if the format of API responses changes in the future, the only SenWeb code that will have to change is the code that converts the specific API response into a QueryResponse.

Currently two classes implement IQueryRequest. QueryRequest, also located inside of the /app/models/query directory, and MockQueryRequest, located inside the /app/models/query/mock directory. QueryRequest provides the communication with the HANA APIs. MockQueryRequest returns properly formatted JSON strings that are hardcoded into the mock. The mock is used for testing purposes as well as development purposes when the HANA API is unavailable.

## Query Manager

The Query Manager layer provides another layer of insulation between the Object Layer and the Query Layer. It is responsible for providing the Object Layer with specific objects built from the generic QueryResponses. The Query Manager methods provide utility that the Object Layer wants in terms that the Object Layer understands. This allows the Object Layer to have no direct dependencies on the API implementation. The Query Manager layer is currently implemented in the DeviceManager class located in the /app/controllers directory. Examples of methods in DeviceManager are:

* public List<Device> getDevices()
* public List<SensorType> getSensorTypes(String deviceType)
* public List<DeviceType> getDeviceTypes()
* public List<DeviceType> getDeviceTypes(SensorType sensorType, List<DeviceType> deviceTypeList)
* public List<String> getDeviceIds(DeviceType deviceType)
* public List<SensorReading> getSensorReadings(Map<String, String> parameters)
* public List<String> getAllSensorTypes()

These methods map to IQueryRequest methods. For example, public List<Device> getDevices() maps to public QueryResponse getAllDevices(). Public List<SensorType> getSensorTypes(String deviceType) maps to public QueryResponse getSensorTypes(QueryDeviceArg dArg).

Internally, the DeviceManager methods follow the general pattern of instantiating an IQueryRequest using QueryRequestFactory, calling an IQueryRequest method, and converting the QueryResponse into Object Layer objects.

## Object Layer

The Object Layer provides the Presentation Layer with pure Java representations of the elements to display and manipulate. Objects contains in the Object Layer are located in the /app/models directory of the project. They include: SensorType, DeviceType, Sensor, Device, and SensorReading.

### Sensor Type

The SensorType object represents a type of sensor in the system. For example, “Temp”.

### Device Type

The DeviceType object represents a type of device in the system. For example, “Firefly\_v3”. It also contains a list of SensorTypes that the device type supports.

### Sensor

The Sensor object helps to map a DeviceType to a SensorType.

### Device

The Device object represents a specific device in the system. It contains the ID of the device, the DeviceType, the device agent, and the location.

### Sensor Reading

The SensorReading object represents a data value from a particular device. It contains the ID of the device making the reading, the type of sensor the reading is for, the value of the reading, and the timestamp the reading took place.

## Presentation Layer

The Presentation Layer provides a UI view to the end users of SenWeb. All files to implement views are located in the /app/views directory of the project. Since we are using the Play Framework, the files to render the views are written in Scala and receive information from the controllers in the form of the Java objects created for the Object Layer.

The different pages that make up the Presentation Layer will be examined in the Tutorial section of this document.

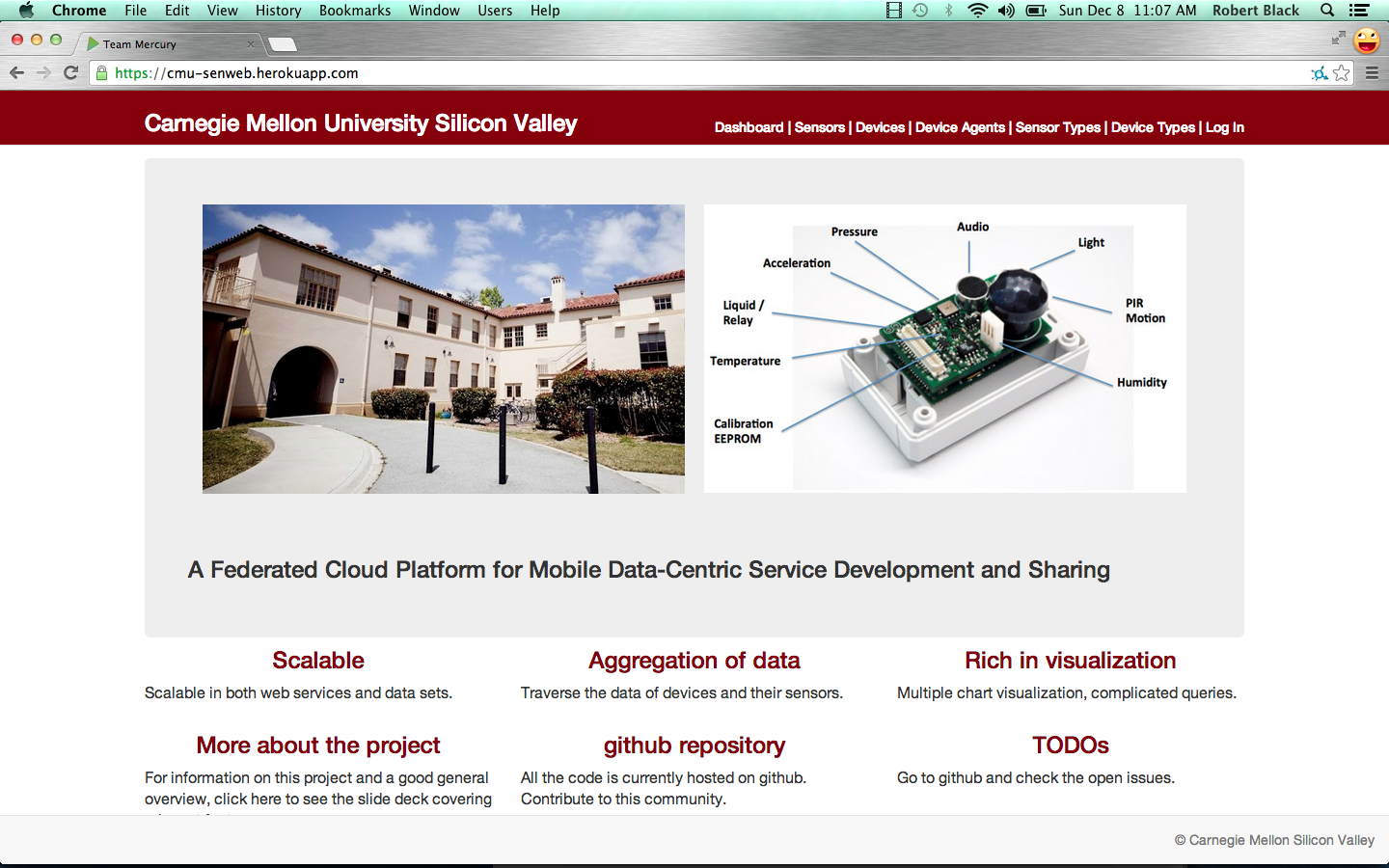
# Documentation and Testing

SenWeb follows the Documentation and Testing guidelines as defined by the Governance team. The Documentation Guidelines that SenWeb follows, defined by Governance, are to document all classes with class level and method level comments, as well as commenting the static variables appropriately in Interfaces and Classes. The SenWeb codebase is commented in order to generate documentation from the JDK “javadoc” tool.

The Testing Guidelines that SenWeb follows, defined by Governance, are to use JUnit 4.11 for testing by annotating methods with @Test and classes with @RunWith(JUnit4.class). SenWeb provides Unit and Integration test cases for all classes and logical modules. This will allow the SenWeb to be integrated into the weekly Jenkins job that Governance has proposed.

# Tutorial

The landing page of SenWeb provides information about the Sensor Service Platform. Additionally, users can navigate the application in the upper right portion of the screen, using the navigation bar.

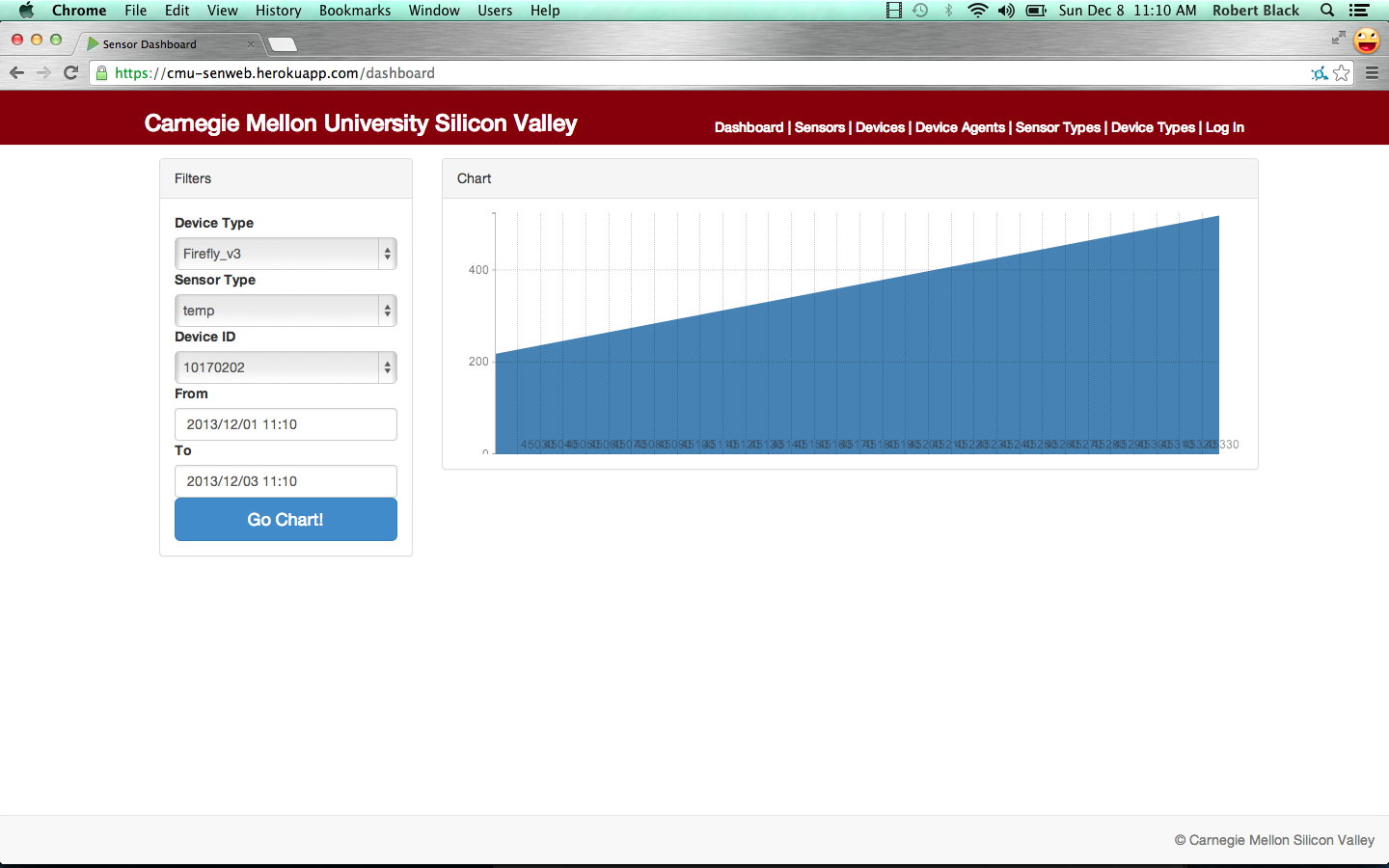


Each page of the application is described below.

## Dashboard

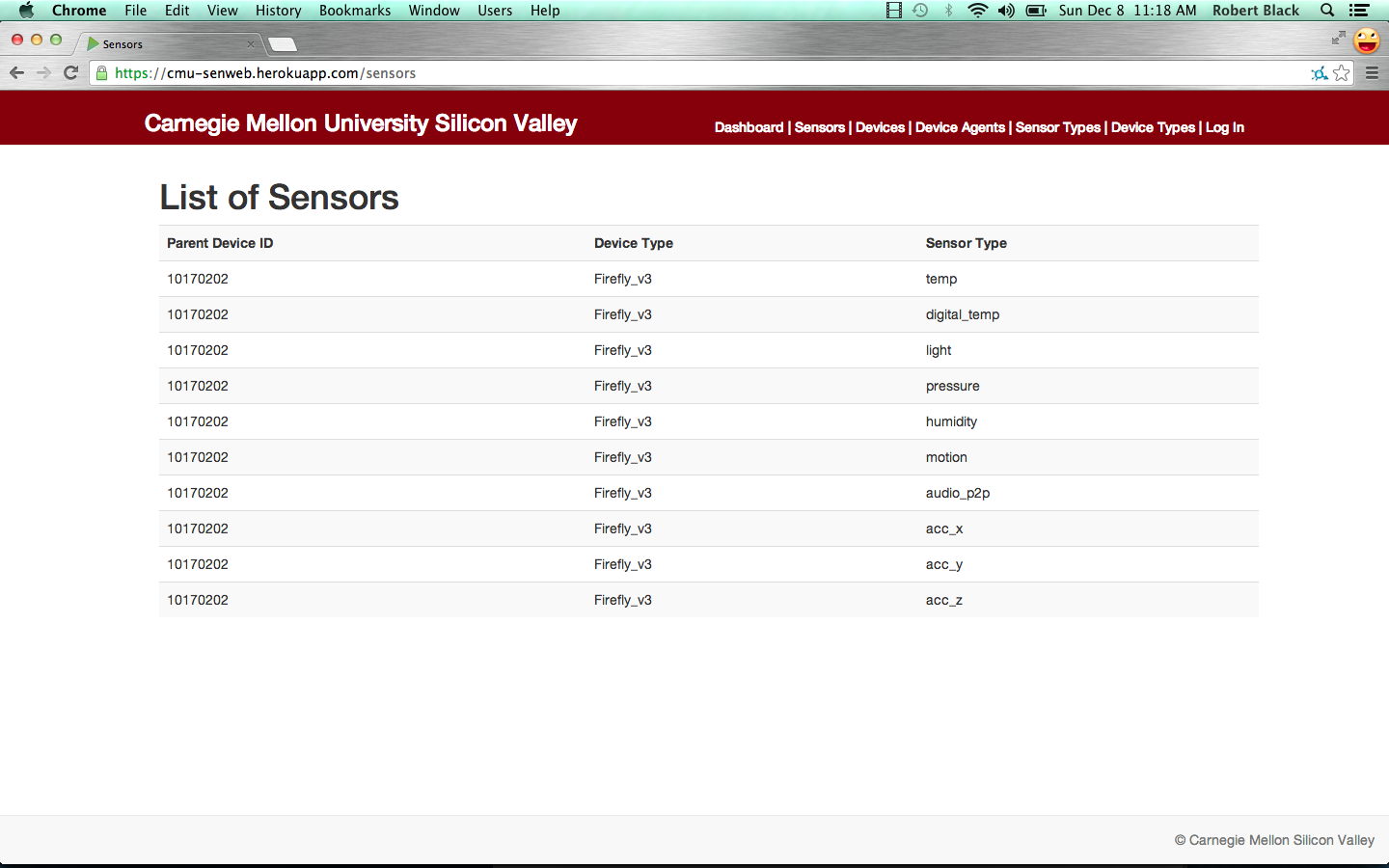
The dashboard provides users a way to visualize sensor reading data. Users select a Device Type, Sensor Type, and Device ID from the Filters section of the screen, enter a start and end datetime in the date fields, and click on “Go Chart!” to create a chart of their selected data.

\*\*Note – as of 12/4/2013, the Sensor Service Platform API does not provide valid Device IDs via Category 3 (metadata) APIs. Thus, the application will cannot query sensor readings because the Device IDs for stored sensor data are unavailable. See warning on <https://github.com/SensorServicePlatform/APIs> for more information.



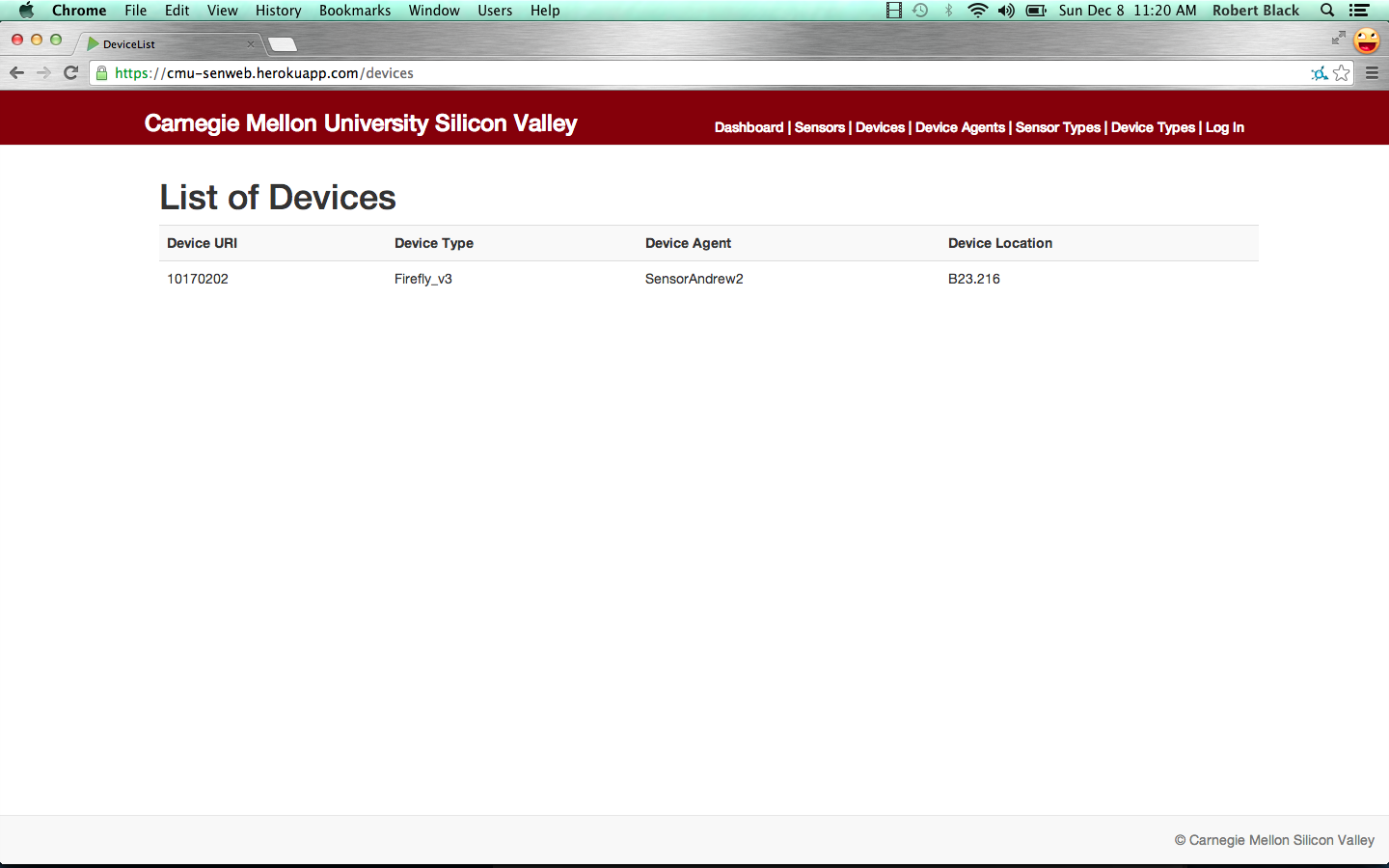
## Sensors

The sensors page provides a list of sensors available in the Sensor Service Platform. In the future, users will be able to modify sensor metadata from this page.



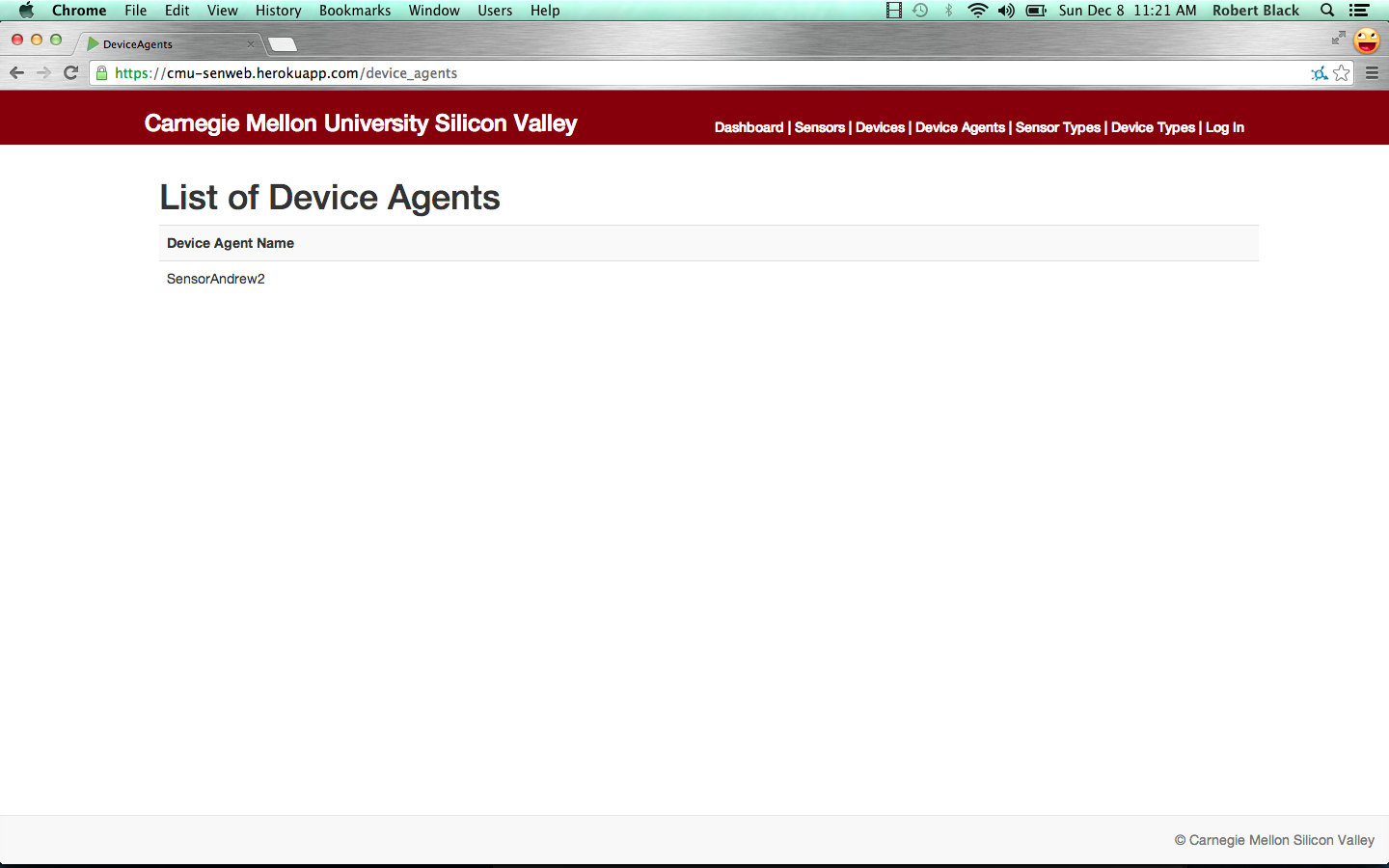
## Devices

The sensors page provides a list of sensors available in the Sensor Service Platform. In the future, users will be able to modify sensor metadata from this page.



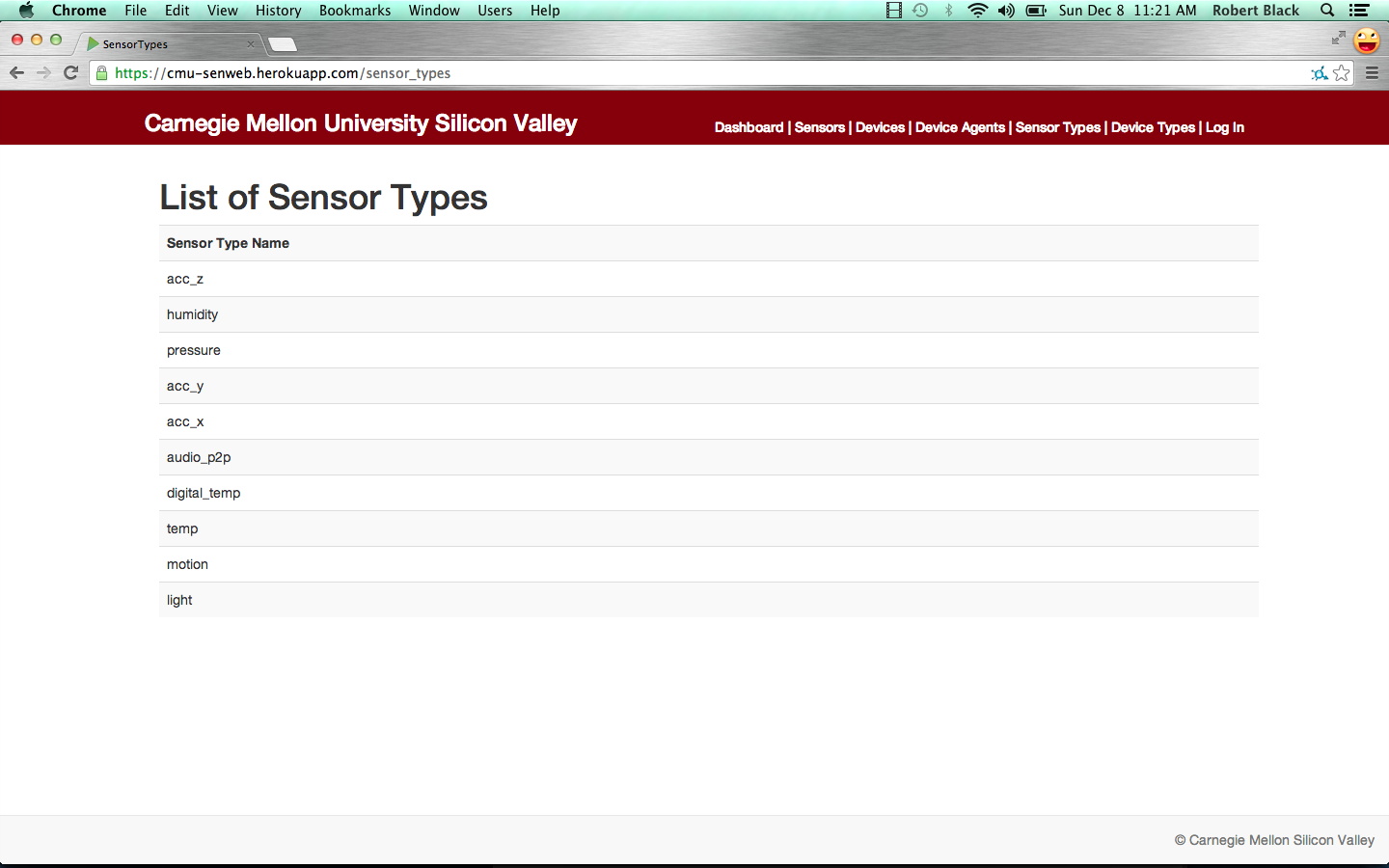
## Device Agents

The sensors page provides a list of sensors available in the Sensor Service Platform. In the future, users will be able to modify sensor metadata from this page.



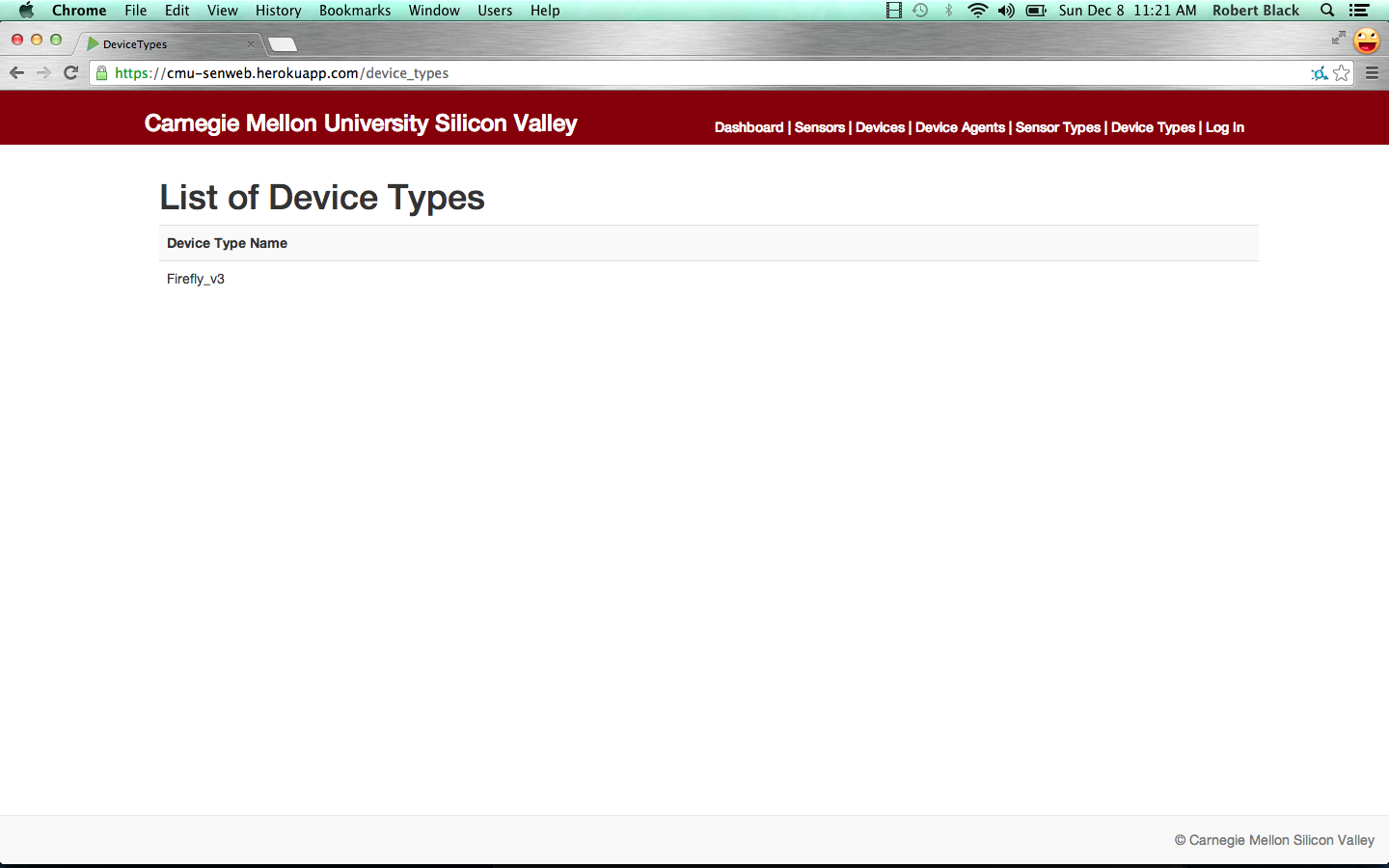
## Sensor Types

The sensors page provides a list of sensors available in the Sensor Service Platform. In the future, users will be able to modify sensor metadata from this page.



## Device Types

The sensors page provides a list of sensors available in the Sensor Service Platform. In the future, users will be able to modify sensor metadata from this page.



## Log In

Not yet implemented. The log in page will be where users log into the application.

# ATAM

The following are the main business drivers based on the goals for the SenWeb project:

* Provide an intuitive interface for users to access the Sensor Service Platform from the web.
* Provide a proof of concept application that demonstrates the capabilities of the Sensor Service Platform API.
* Make data accessible from multiple devices.
* Present data in an easily consumable format (e.g. graphs, charts).

The architectural approaches used in SenWeb are as follows:

* Client / Server - SenWeb accesses the Sensor Service Data Platform REST API in order to access sensor readings and device metadata. SenWeb then generates the appropriate views for this data to be shown on the client’s web browser.
* Data Access Isolation - Encapsulate the data models in one layer and provide the functionality for accessing that data in a separate layer.
* Business Logic Isolation - Isolate functionality that manipulates data or applies business rules from the mechanisms that obtain data from underlying sources. This allows data access implementation to vary independently of business logic implementation.
* Model-View-Controller framework (MVC) - We use an MVC framework to separate concerns about representations of objects from how they are presented to users. Controllers mediate the interaction of models and their associated views. The benefit here is that models can vary independently of views. Additionally multiple views can be created to represent models in different ways.

# ATAM Utility Tree

|  |  |
| --- | --- |
| Quality Attribute | Performance |
| Quality Factor #1 | Latency |
| Scenarios | 1. Pages should load with sub-second latency (High)  2. Pages should render asynchronously (High) |
| Quality Factor #2 | Transaction Throughput |
| Scenarios | 1. Transactions should update database in sub-second timeframes. (Low)  2. ACID guarantee (High) |
| Quality Attribute | **Availability** |
| Quality Factor #1 | Hardware Failures |
| Scenarios | 1. Hardware failures cause no more than 0.1% downtime in any given time frame. (Low)  2. Hardware failures do not adversely affect other quality attributes. (Medium) |
| Quality Factor #2 | Software Failures |
| Scenarios | 1. All errors should be logged and promptly addressed. (Low) |
| Quality Attribute | **Security** |
| Quality Factor #1 | Threat Vulnerability |
| Scenarios | 1. Web application should not be vulnerable to known web application attack vectors (e.g. XSS, XSRF) (High) |
| Quality Factor #2 | Authentication & Authorization |
| Scenarios | 1. Should ensure a user’s identity by requesting proper authentication credentials. (Low)  2. Should ensure that a user is authorized to view requested data and perform the requested actions. (Low) |
| Quality Attribute | **Modifiability** |
| Quality Factor #1 | API Changes |
| Scenarios | 1. The application should not require significant modification if the underlying API changes. (Medium)  2. The application should be able to easily accommodate new data from the API. (Medium) |
| Quality Factor #2 | Support for Current and New Devices |
| Scenarios | 1. The application should support a wide range of current display devices. (Low)  2. The application should be not require significant modification in order to present data on new devices as they become available. (Low) |
| Quality Attribute | **Usability** |
| Quality Factor #1 | Ease-of-use |
| Scenarios | 1. The application should be intuitive to a wide range of users (High) |
| Quality Factor #2 | Accessibility |
| Scenarios | 1. Screen readers should be able to read the markup in the web application sensibly. (Low)  2. Fonts, colors, and styles should be consistent with standard practices (Medium) |

## Risks, Sensitivities, and Tradeoffs

The largest risks for the SenWeb project are the dependencies on the Sensor Service Platform API. If this API is unavailable for any reason, SenWeb will also be unavailable.

SenWeb is highly sensitive to the choice of hosting platform, the availability of the Sensor Service Platform API, the latency of the Sensor Service Platform API, and any future changes made to the Sensor Service Platform API.

Because of the sensitivities to the Sensor Service Platform API, the tradeoff that has been made in SenWeb is one of Performance vs. Modifiability. In order to keep the system extensible and modifiable, there is a negative impact on Performance as processing needs to occur to create multiple objects to isolate Data Access from Business Logic.

# Future Work

Due to the fact that we had only a short time (six weeks) to implement SenWeb, we were unable to complete all of the features that we would like to have implemented. Future teams may want to consider implementing some of the following to improve SenWeb:

* Reduce HTTP overhead for device metadata - Once the Sensor Service Platform API is upgraded to allow for Device Type and Sensor Type data to be obtained for all existing devices in a single call, a number of HTTP requests can be eliminated between SenWeb and the Sensor Service Platform API server. This will result in much faster page loading and a better user experience.
* Implement CRUD operations for resources - Future teams may want to create functionality to add, edit, and update Devices, Device Agents, Device Types, Sensors, and Sensor Types. Once the Sensor Service Platform API is updated to provide CRUD operations on these resources, it will be possible to add this functionality.
* Implement user log in and RBAC - Once the target audience for this application and their uses have been defined, a role-based access control model can be created based upon different user role profiles and the permissions allowed to each role. Users should be able to log in and only have access to actions and data that they are allowed to perform and see.

## Recommendations for Sensor Service Platform API

In order to make the API easy to use and widely available, we recommend that the API be modified in the following ways:

* Discoverability - Services that implement discoverable APIs allow clients to generate code without having to be explicitly aware of the details of API implementation. For instance, the Sensor Service Platform API could be self-describing, allowing clients to create language specific API implementations that could grow along with the API. See (<http://www.w3.org/TR/discovery-api/>) for more detail.
* API expressiveness - The API should represent the underlying resources that it exposes to the extent needed by clients, but not more. This allows clients to receive just the data that they are interested in without the additional overhead of additional remote procedure calls or receiving unnecessary data. One way this could be implemented would be for the API to allow clients to specify which fields to return when making a call.

# Conclusion

The CMU Sensor Data platform is comprised of a number of components including the a HANA database and a set of APIs for accessing the database. SenWeb is a web app that uses the available platform APIs to access sensor reading data and also provides an interface for visualization those data. In the foregoing sections, we discussed the processes that our team used in developing this application. Additionally, we outlined some of the technical decisions we made along the way and the rationale behind those decisions.

Furthermore, we presented a number of diagrams which represent the various architectural views in order to provide stakeholders with a visual, concise overview of the system. Also outlined in this document is our approach to testing and the tools that we used for that purpose. Having perform an ATAM analysis of the entire system, our team thought it valuable to include that bit of information for the stakeholders.

Among other things, our successful development of SenWeb which utilizes the Sensor Data platform APIs provides a good template for anyone (developer) looking to use the API for a similar project or better yet, users who are looking for a quick and easy way to get sensor reading and/or visualize such reading.

Lastly, we made a number of recommendations, which we hope will be of use for future teams working on improving this application.

# Bibliography

*Data Access Object*. (2013, 10 04). Retrieved 12 04, 2013, from Best Practice Software Engineering: http://best-practice-software-engineering.ifs.tuwien.ac.at/patterns/dao.html

Manglick, A. (2010, 11 17). Retrieved 12 4, 2013, from http://arun-architect.blogspot.com/2010/11/composite-design-pattern-structural.html

Rozanski, N., & Woods, E. (2013). *Software Systems Architecture (second eidtion).* Upper Saddle River: Pearson Education, Inc.