**SenWeb Assumptions and Considerations**

Geoff Schaeffer, Abhi Trivedi, Danny Brown, Rob Black

## 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 1: 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.