Web-based On-board Real-time Rendering Data System (WORRDS)

Final Report

WSU CASAS

Mentor:
Aaron Crandall

Team Wildlings
Bryan Barrows
Benjamin Cheung
Daniel Pepka
Joanne Magtibay
Pat McKee
Trey Ottaway

CptS 423 Software Design Project II
Spring 2016
Instructor: Sakire Arslan Ay
# TABLE OF CONTENTS

I. Introduction

II. System Requirements Specification
   II.1. Project Stakeholders
   II.2. Use Cases
   II.3. Functional Requirements
   II.4. Non-Functional Requirements

III. Software Design
   III.1. Architecture Design
      III.1.1. Overview
      III.1.2. Subsystem Decomposition
   III.2. Data Design
   III.3. User Interface Design

IV. Test Case Specifications and Results
   IV.1. Test Plans Overview
   IV.2. Test Results

V. Description of Final Prototype

VI. Conclusions and Future Work
   VI.1. Limitations and Recommendations
   VI.2. Future Work

VII. Acknowledgements

VIII. References

IX. Appendix A – Team Information

X. Appendix B - Project Management
I. Introduction

The Center for Advanced Studies in Adaptive Systems (CASAS) at Washington State University produces a “Smart Home in a Box”, or SHiB. SHiBs contain a variety of sensors which collect and report data in realtime to the CASAS lab. Typical SHiB sites consist of existing homes and elderly care facilities. Before the start of this project, the raw sensor data was simply parsed and stored in the CASAS database.

In order to interact with the large amount of sensor data and the real-time messaging capability of the SHiBs, CASAS commissioned the Web-based On-Board Real-time Rendering Data System (WORRDS) and tasked its creation to Team Wildlings. WORRDS would help system maintainers decide to change batteries at a site by providing a means of viewing the battery data on a browser. It would further aid maintainers by automating the process of creating and editing sensor maps of SHiB sites. WORRDS would also allow researchers to visualize behavioral data for easier analysis, as well as enable caregivers to monitor the activities of the SHiB occupants in real-time. Additionally, users would have accounts on WORRDS with different authorizations to the data. The overall motivation behind WORRDS was to save stakeholders time and money and better inform their decisions.

All of the aforementioned goals for WORRDS were addressed in the final prototype, which boasts the following features:

- Real-time and historical data visualization via charts, tables, and other widgets
- 2D map creation using the Google Tango WORRDS app
- Custom user and security management
- View and edit SHiB layouts with a built-in map editor
- Reliable database daemon to sync WORRDSdb with CASASdb
II. System Requirements Specification

II.1. Project Stakeholders

The stakeholders for the project include the members of the development team, the project mentor, the professor, CASAS researchers, and community members who have a SHiB.

By visualizing the SHiB data, WORRDS enables researchers, like the team’s mentor, to better analyze data on the movements and sleep patterns of people over a long period of time to identify individuals likely to develop illnesses, such as Alzheimer's or Parkinson's, in the future. WORRDS also helps CASAS researchers decide when to visit a site to change batteries and determine which batteries to change. In addition, WORRDS automates the process of creating and editing sensor maps and will save researchers time during testbed setup.

For community members, owning a smart home will provide a number of benefits, such as the ability to receive emergency medical assistance or monitor an elderly loved one from afar. WORRDS in particular will enable medical personnel and other users to see the movements of an individual in real-time on a map and determine if they are in need of aid.

The development team also has a stake in WORRDS; the success of WORRDS will determine their eligibility for graduation.

II.2. Use Cases

![Use case diagram for Logging In to WORRDS.](image)

Figure 1. Use case diagram for Logging In to WORRDS.
Figure 1 describes the end user’s interaction with Login. The user enters their username and password (Submit Credentials) on the login webpage. This sends a request to the Nginix Server, which redirects the request (Redirect Request) to the Flask Server. In order for the Flask server to determine whether the user has access (Validate Credentials), it queries the database to see if the credentials are valid (Query Database). The database then receives the query and attempts to respond (Respond to Query). If it is discovered that the credentials are in the database, Flask allows the user to log in. Otherwise, Flask denies the user access.

Figure 2: Use case diagram for a Dynamic Web Page, a web page that updates in real-time. This use case assumes the user has already been logged in.

Figure 2 describes an end user’s interaction with a dynamic web page. When a user switches pages (Switch View), the User can either view floor plans of existing smarthomes with sensors changing in real-time (Map View), build 2D layouts of new smarthomes (Map Editor View), or view real-time sensor data charts and statistics (Reporting View). Depending on the view, the user is able to choose which sites and sensors they can see on their current page (Customize Display). Whenever a Customize Display or Switch View occurs, the page must be re-rendered (Update View). This sends a request to the Nginix Server, which redirects the request (Redirect Request) to the Flask Server. The Flask server responds with the updated webpage (Respond to Request). Similarly, while the user views the Map View, Map Editor View, or Reporting View, requests for real-time data to populate the page are sent to the Nginix Server (Redirect Request), which are then redirected to the Flask Server (Flask Server). Whenever the Flask Server needs to respond to a real-time data request, it will consume real-time messages from RabbitMQ (Consume Messages from RabbitMQ) and periodically query the WORRDS database (Query Database) for data to perform
calculations on. In the latter case, the WORRDS database will reply with the data requested (Respond to Query).

Figure 3. Use case diagram for a Static Webpage, a webpage that does not update in real-time.

Figure 3 describes an end user’s interaction with the static webpage. The main purpose of the static page is to allow the user to observe archived data from smarthomes (Historical Reporting View). The user is also able to configure the way in which they want the data to be presented (Customize Display). Whenever a refresh of the Historical Reporting View or Customize Display occurs, the Nginx server re-renders the view (Update View).

Figure 4. Use case diagram for the Map Editor View, which allows the user to edit 2D layouts of a site.
Figure 4 describes an end user’s interaction with the Map Editor View. Inside the map editor the user has the option of making a new map (Create New Map) or working on an existing map (Load Map). While working on the map, the user is able to add objects such as walls or sensors (Add Object), remove objects (Remove Object), or adjust the position of existing objects (Move Object). The user also has the option to record their progress at any point (Save Map) as well as the option to dispose of a map that has outlived its usefulness (Delete Map). When Delete Map or Load Map occurs, the page must be re-rendered (Update View). This sends a request to the Nginix Server, which redirects the request (Redirect Request) to the Flask Server. The Flask server then sends a message to the database (Query Database) to either load an existing map or delete an existing map, at which point the database executes whatever query Flask sent (Respond to Database Query). Save Map also goes through the same process as Update View, except Flask requests to insert a map into the database.

II.3. Functional Requirements

II.2.1 WORRDS Server

II.2.1.1 Extract sensor data from RabbitMQ and WORRDS database
Requirement: The Flask server pulls real-time sensor data from RabbitMQ and historical data from the WORRDS database.
Source: CASAS Lab.
Priority: 1

II.2.1.2 Perform calculations and other processing on sensor data
Requirement: The Flask server runs relevant processing on sensor data in an efficient manner, such that it provides data for the user interface in approximately real-time. Some example processes that Flask will perform on the data include converting sensor messages to a format readable by the user interface (e.g. JSON), determining the sensor with the lowest battery level at a site, and calculating what percentage of batteries need changing at a site.
Source: CASAS Lab.
Priority: 1

II.2.1.3 Serve webpages with real-time data
The Nginx server receives the processed sensor data and displays it to the user.
Source: CASAS Lab.
Priority: 1

II.2.2 WORRDS Database

II.2.2.1 Cache recent sensor data taken from the main database
The database is populated with recent sensor data from the main database, creating a cache that the WORRDS Server queries.
Source: CASAS Lab.
Priority: 1
II.2.2.2 Save map data
The database also stores data for maps created by Tango and/or edited with the map tool.
Source: CASAS Lab.
Priority: 1

II.2.3 Map Editor

II.2.3.1 Drag-and-drop sensors onto a map
The user can drag-and-drop sensors onto a 2D layout of the smart home.
Source: CASAS Lab.
Priority: 1

II.2.3.2 Convert Project Tango 3D-maps to 2D maps
An application will convert 3D maps of smart home sites generated by the Google Tango device to 2D layouts that can be used in the map editor. Noise captured by the Tango is cleaned out in the conversion process in order to provide accurate, easy-to-understand layouts.
Source: CASAS Lab.
Priority: 1

II.2.4 User Interface

II.2.4.1 Display homepage with tabs for various reports
The user's homepage contains tabs for navigating different real-time and historical reports on their smart home's sensor data.
Source: CASAS Lab.
Priority: 1

II.2.4.2 Display relevant sensor data charts for maintenance
At least one of the sensor data charts displays information useful for maintenance, e.g. battery levels and when the sensor was last online.
Source: CASAS Lab.
Priority: 1

II.2.4.3 Display real-time maps optimized for a web browser
Real-time maps of a user's smart home can be viewed online, outside of the CASAS Lab.
Source: CASAS Lab.
Priority: 1

II.2.5 Security

II.2.5.1 Secure account access for users
Users can log in to their own customized homepage, but cannot view other users' smart home data.
Source: CASAS Lab.
Priority: 2
II.4. Non-Functional Requirements

II.3.1: Data

II.3.1.1. Displaying Data
Correct and coherent data needs to be displayed to users so that relevant information about the smarthomes can quickly be gleaned by the user.

II.3.1.2. Security
Users should only be able to view data that they have access to. Data from smart homes in extremely private and sensitive and therefore only smart home owners and administrators should be able to view the data for a particular smart home.

II.3.2: Documentation

II.3.2.1 Maintainability
All code written should be robust, comprehensible, modular, and thoroughly commented and documented so that it is easy for current and future developers to reuse modules and add new ones to the system.

II.3.2.2. Stand-ups
Weekly stand-up meetings were held with project owners and mentors to confirm all stakeholders are in agreement on the project’s progress.

II.3.3: Software

II.3.3.1. Google Tango
WORRDS must be able to accept 3D data collected with a Google Tango and translate it into an accurate 2D map that can later be displayed to a user.

II.3.3.2. D3
WORRDS will use D3 to visualize and communicate data to users.

II.3.3.3. MySQL
WORRDS will use MySQL to store sensor events cached from the main CASAS database.

II.3.3.4. Nginx
WORRDS will use Nginx to display real-time data to our dynamic web pages.

II.3.3.5. RabbitMQ
WORRDS will use RabbitMQ to store and receive real-time sensor data.

II.3.3.6. Flask
WORRDS will use Flask to process sensor data from MySQL and RabbitMQ.

WORRDS – Final Report 9
II.3.3.7. Working Prototype
The team will provide a working prototype of WORRDS by the end of the semester.

II.3.3.8. HTML5 Canvas
HTML5 Canvas is used to create the map editor.

II.3.3.9. Usability
Usability is key for the success of the WORRDS web and Tango application interfaces. Navigation of both should be intuitive with none or hardly noticeable lag so the user can accomplish tasks efficiently. The webpages should be uncluttered and contain charts that are meaningful to the user with access to them. When relevant, a link to an FAQ or tips are provided for the module being viewed (e.g. map editor, Tango application, site page).
III. Software Design

III.1. Architecture Design

![Diagram of WORRDS three-tier architecture. The diagram depicts what machines/processes logically belong in which layer; for instance, dev-data-gateway and bunny are not located on the WORRDS server. The Google Tango app connects to the main WORRDS UI.](image)

**Figure 5.** Diagram of WORRDS three-tier architecture. The diagram depicts what machines/processes logically belong in which layer; for instance, dev-data-gateway and bunny are not located on the WORRDS server. The Google Tango app connects to the main WORRDS UI.

### III.1.1. Overview

WORRDS utilizes a three-tier architecture with a user interface, a server, and a database as illustrated in Figure 5. Each layer provides services to the layer above and requests services from the layer below. Non-adjacent layers do not interact directly with each other.

This architecture adds security, as an end user on the UI cannot directly interact with the sensitive information stored in the database; the server must moderate all communication between the two layers. This design pattern further suits WORRDS because it enforces clean separation between the types of services involved and is easy for future developers to maintain. This architecture also allows for modularity and is well-suited to loosely coupled components; any of the original components may be substituted for new ones with different implementation, given the services they provide stay the same.

The user interface includes the WORRDS website and Google Tango UI. The website leverages Javascript, HTML5, and D3.js to render everything from the charts to the map editor. Web pages are populated with data from the server layer. The Google Tango application, which leverages Unity and C#, has an interface to aid users in mapping out a room and adding sensor locations on the map. The Tango then uploads its maps through the WORRDS web interface for use on the map editor.
The server layer consists of the Flask and Nginx servers and bunny, a RabbitMQ server, and performs data processing and webpage serving. The client connects to an Nginx webserver, which performs load balancing and forwards requests to the WORRDS Flask app. The Flask app performs analytics on both historical data it retrieves from the local WORRDS database and real-time sensor data it consumes from RabbitMQ. If the client needs to save/load a file (e.g. save/load map image to the map editor), Flask oversees its storage in/retrieval from the database. Flask also oversees storage of RabbitMQ data in the database and user authentication with the database.

The WORRDS MySQL database (WORRDSdb) mainly serves as a cache of events from the main CASAS database (CASASdb) in order to improve the performance of the Flask app. Periodically, WORRDSdb automatically syncs with CASASdb via a daemon written in Java. The syncing ensures the WORRDS database does not miss a dropped RabbitMQ message. In addition to caching CASASdb events and storing RabbitMQ events, WORRDSdb stores map editor objects and images and user information.

III.1.2. Subsystem Decomposition

IV.1.1. User Interface (UI)

a). Description

This subsystem handles graphic elements through which the user accomplishes tasks and visualizes sensor data. Potential user actions include navigating site pages, editing map images, storing and loading map images, and logging in and logging out of the site. The user also accesses admin utilities that allow them to create new users and admins, modify user information, edit testbed information, and manage associations between users and testbeds. Depending on their permissions, a user has limited access to tools and pages they can see. The UI displays dynamic webpages in real-time with low latency.

b). Concepts and Algorithms Generated

Drupal, a content management system, was originally suggested by the mentor to host the webpages. Due to Drupal’s inability to render dynamic webpages and difficulty in writing customizable plug-ins, the team ultimately decided to use Flask to manage content and security manually and support dynamic webpages easily.

IV.1.2. Server

a). Description

This subsystem is responsible for retrieving real-time data from the RabbitMQ message queue, querying data from the WORRDS database, and processing the
data for display on webpages. It is also responsible for web hosting and interfacing between the UI and database tiers.

b). Concepts and Algorithms Generated

The primary function of the server is to service the user interface in a timely, efficient manner.

Sensor event data from the SHiBs initially arrive at bunny. Then the WORRDS server consumes the events from the queues on bunny and renders them in real-time. Various simple calculations (e.g. the percentage of batteries at the site that require changing) is performed on sensor data retrieved from the local WORRDS database to relieve the strain on the UI. Data in the WORRDS database comes from the database daemon (see IV.2.3) and the real-time data the server receives from RabbitMQ.

The server tier also returns data to the UI in a common format that is easy to parse, such as JSON. Flask, a microframework, is ideal for both implementing and running fast, simple processes. Nginx provides inherent load-balancing across multiple application instances, which enables more UI functionality - since more applications can be supported - with improved performance and low latency.

IV.1.3. Database

a). Description

The database tier is responsible for archiving real-time data as it is received by the WORRDS server, storing Tango and map editor files, storing sensor and user information, and caching data from the CASAS database.

b). Concepts and Algorithms Generated

Due to the large size of the CASAS database and its significant role in the CASAS system, the WORRDS database daemon executes carefully timed, optimized queries to the CASAS database so as not to decrease performance. Two major considerations that contribute to the effectiveness of the daemon algorithm include running the process during times of low traffic to the WORRDS server and limiting the amount of data requested by the queries.

III.2. Data design

Two databases provide the data to support WORRDS: the CASAS PostgreSQL database, which contains all the historical data ever received from the SHiBs, and the WORRDS MySQL database, which serves as a cache of the CASAS database. Except for the WORRDS database daemon that periodically queries the CASAS database for historical data from the last 90 days, all WORRDS applications built by the development team only depend on the WORRDS database.
The following is a list of important tables in the WORRDS database:

- **User.** User information for logging in and registration.
- **Permission.** Defines type of page access (e.g. map, battery).
- **Sensor.** Information on each sensor.
- **Package.** Information on each package (a device).
- **Battery.** Information on each target’s battery.
- **Testbed.** Information on each testbed.
- **Testbed Map.** Information on each map created on the map editor.
- **Testbed Map Image.** Contains uploaded map images to be used as the background of the map editor.
- **Site Access.** Defines which users have access and edit/view permissions to certain testbed pages.
- **All Events.** Contains cache of the All Events table in CASASdb; all events ever sent by the SHiBs to CASAS lab.
- **Experiment All Data.** Contains cache of the ExpAllData table in CASASdb; all events tagged with a behavior label.
- **Map Dragable, Map Package, Map Line.** Tables to store objects created in the map editor.

The sensor messages received in real-time from the sites are received as a string with nine tab delimited fields. These fields, in order, are the date and time the sensor was fired, the sensor ID, the sensor name, its status (ON/OFF/OPEN/CLOSE), the type of sensor (entity/state/control/system), the sensor agent, the sensor type, the package type, and the event type. The real-time data is stored in the All Events table for record and converted to JSON for user interface display as it arrives at the WORRDS server.

### III.3. User Interface Design

The final prototype interface resembles interface design patterns the end-users are familiar with. It is implemented using Bootstrap CSS, HTML5, JavaScript, and D3 (the latter was used to design pleasing and readable dynamic charts and graphs). Whenever possible, Flask templates and existing CSS classes were used to establish a consistently clean look and feel.

Each webpage layout on the WORRDS website is designed to allow end users to quickly grasp the state of the system or a specific testbed, and thus accomplish their tasks faster. When relevant, page summaries are generated and positioned at the top. Graphs and other visuals are present not only to look pleasing, but to provide the fastest means of communicating the information. Unnecessary elements are left out to limit vertical scrolling on the UI, for the convenience of the user. In many cases, there are multiple methods of accessing the same webpage to save the user time navigating, especially if they are new to WORRDS. Finally, WORRDS is designed with a mobile-first approach using Bootstrap and renders beautifully on mobile devices. Hence CASAS users can access WORRDS tools on-the-go, particularly while visiting a testbed.
Figure 6: The WORRDS homepage (above) and Battery Overview page (bottom). On the homepage, note the summary of testbed and page access information in the first row, followed by the real-time graph of incoming events/second, followed by the event counters. In a matter of seconds, a user can quickly conclude how many pages they have access to, how real-time events are functioning, and how active the system or particular testbeds are today. Most webpages, like the Battery Overview, follow a similar arrangement, rendering the most important information at the top, followed by less important information. A user could arrive at the Testbed Overview webpage by either clicking “View Details” in the green System Info box on the homepage or by clicking “Testbeds” in the left sidebar; this is one example of the redundant linking present throughout WORRDS.
IV. Test Case Specifications and Results

IV.1. Test Plans Overview

Testing activities were conducted on the following components:

- WORRDS Nginx and Flask servers
- WORRDS Local Database
- WORRDS UI

Additionally, functionality for authorizing users, providing personalized webpage views, the map editor, and miscellaneous pages were tested.

Unit testing was mainly provided in the form of black box testing and code coverage, with a goal of 100% code coverage. Integration testing of subsystems and acceptance testing was also performed.

The objectives for testing activities were:

- To create an automated test framework for WORRDS.
- To design software such that automated tests see a 100% pass rate by the end of the semester.
- To ensure software satisfies all conditions for categories of user inputs.
- To provide source code and live demonstrations of the framework at multiple times throughout the semester.
- To ensure all components function together to provide low latency webpage display.

The following were milestones for the test plan:

- Complete unit tests for Nginx and Flask server applications.
- Complete unit tests for local database queries.
- Complete acceptance tests for user accounts.
- Complete acceptance tests for dynamic webpage tools.
- Complete acceptance tests for static webpage tools.

IV.2. Test Results

IV.2.1 Unit Tests

IV.2.1.1 Cache recent sensor data taken from the main database.

1. **Test Title/Name**: Functionality test and data integrity test for the DBLoadSave application
   
2. **Test Summary/Description**: Verify that DbLoadSave is correctly retrieving data from the CASASDb PostgreSQL database and is correctly inserting each entry into the WORRDSDb MySQL database.
   
3. **Assumptions**:
   
   a. The CASASDb PostgreSQL database is online and reachable.
b. The WORRDSDB MySQL database is online and reachable.
c. The DDL for each table in CASASDB is correct.
d. The DDL for each table in WORRDSDB is correct.
e. The entries in CASASDB are stored correctly.
f. The entries in WORRDSDB are stored correctly.

4. Input Specification: WORRDS is a database currently running on MySQL on the WORRDS server. CASASDB is a database currently running on PostgreSQL on the adb server.

5. Output Specification:
   1. Expected output: The data stored in the WORRDSDB MySQL database by DbLoadSave exactly matches the data stored in the CASASDB PostgreSQL database.
   2. Actual Output: The data stored in the WORRDSDB MySQL database by DbLoadSave exactly matched the data stored in the CASASDB PostgreSQL database.

9. Test Result (Pass/Fail): PASS

IV.2.1.2 Seed database with map data.
   1. Test Title/Name: Seed map editor database unit tests
   2. Test Summary/Description: Unit test adding items to the database for use with the map editor.
   3. Assumptions: The WORRDS database exists and has been successfully migrated. The contents of the database do not matter.
   4. Input Specification: Preset unit test for creating maps, map dragables, and map lines found in MapEditorRepository.py
   5. Output Specification:
      a. Expected output: Each test should report successfully that it wrote itself to the database and that reading the content back out successfully matched what was put in.
      b. Actual Output:
         Map: Tokyo
         SUCCESS: unitTestGetTestBedMap success!
         Map: default name _ 1 _ 2
         SUCCESS: unitTestGetTestBedMap success!
         MapDragable: 1
         SUCCESS: unitTestGetMapDragable success!
         MapDragable: 2
         SUCCESS: unitTestGetMapDragable success!
         MapLine: 1
         SUCCESS: unitTestGetMapLine success!
         Test run is complete.

6. Test Result (Pass/Fail): PASS
   7. Pass/Fail Criteria: The test fails if any of the seed methods fail to read/write the data that was put in. The pass criteria is that no failures are detected.
IV.2.1.3 Create Dragable for map.
1. **Test Title/Name**: Create Dragable for TestBed
2. **Test Summary/Description**: Go to the page for a testbed map and click the add new dragable button. Then click on the canvas.
3. **Assumptions**: A testbed map already exists with 0 or more dragables.
4. **Input Specification**: An existing testbed map.
5. **Output Specification**:
   a. **Expected output**: A new testbed map dragable should appear where you clicked on the canvas.
   b. **Actual Output**:

   Before:

   ![Before Image]

   After:

   ![After Image]

6. **Test Result (Pass/Fail)**: PASS
7. **Pass/Fail Criteria**: The test passes if the new testbed dragable is created. The test fails if the new testbed dragable is not created, or if the testbed dragable is created anywhere other than where you clicked.

IV.2.1.4 Create Line for map.
1. **Test Title/Name**: Create Line for TestBed
2. **Test Summary/Description**: Go to the page for a testbed map and click the add new line button. Then click and drag between two dragables.
3. **Assumptions**: A testbed map already exists with 2 or more dragables.
4. **Input Specification**: An existing testbed map and at least two existing dragables.
5. **Output Specification**:
   a. **Expected output**: A new testbed map line should appear connecting the two dragables.
b. **Actual Output:**

**Before:**

![Before image]

**After:**

![After image]

6. **Test Result (Pass/Fail):** PASS

7. **Pass/Fail Criteria:** The test passes if the new testbed line is created. The test fails if the new testbed line is not created, or if the testbed line is created anywhere other than between the two dragables that were clicked.

IV.2.1.5 Drag Line for map.

1. **Test Title/Name:** Drag Dragable with line

2. **Test Summary/Description:** Go to the page for a testbed map and click the drag button. Then click and drag on a dragable that contains at least one line.

3. **Assumptions:** A testbed map already exists with two or more dragables with lines connecting them.

4. **Input Specification:** An existing testbed map and at least one dragable line pair.

5. **Output Specification:**
   a. **Expected output:** The dragable and line should update its location and be visible on the screen as you drag it them.
   
   b. **Actual Output:**

   **Before:**

   ![Before image]
6. **Test Result (Pass/Fail):** PASS
7. **Pass/Fail Criteria:** The test passes if the testbed draggable is successfully moved on the canvas and the line is also updated as the draggable is moved. The test fails if the draggable does not move, or moves anywhere other than where it was dragged. The test will also be a failure if the line does not update properly with the draggable.

IV.2.1.6 Remove Dragable for map.
1. **Test Title/Name:** Remove Dragable without line
2. **Test Summary/Description:** Go to the page for a testbed map and click the remove draggable button. Then click on a draggable containing no lines.
3. **Assumptions:** A testbed map already exists with at least one draggable without any lines.
4. **Input Specification:** An existing testbed map and at least one draggable without any lines.
5. **Output Specification:**
   a. **Expected output:** The draggable should be removed from the canvas.
   b. **Actual Output:**

Before:

After:
6. **Test Result (Pass/Fail):** PASS

7. **Pass/Fail Criteria:** The test passes if the testbed draggable is successfully removed from the canvas. The test fails if the draggable does not become removed, or if any other lines or dragables also disappear.

IV.2.1.7 Remove Draggable with Line for map.
1. **Test Title/Name:** Remove Draggable with line
2. **Test Summary/Description:** Go to the page for a testbed map and click the remove draggable button. Then click on a draggable containing at least one line.
3. **Assumptions:** A testbed map already exists with at least two dragables a line between them.
4. **Input Specification:** An existing testbed map and at least two dragable with a line between them.
5. **Output Specification:**
   a. **Expected output:** The draggable and connecting lines should be removed from the canvas.
   b. **Actual Output:**

   Before:

   ![Before Image]

   After:

   ![After Image]

6. **Test Result (Pass/Fail):** PASS

7. **Pass/Fail Criteria:** The test passes if the testbed draggable is successfully removed from the canvas and all lines connected to the draggable is removed. The test fails if the draggable does not become removed or if any of the lines connected to the draggable fail to be removed. The test will also be a failure if any other any other lines or dragables also disappear not connected to the clicked on draggable disappear.
IV.2.1.8 Remove Line from map

1. **Test Title/Name**: Remove line
2. **Test Summary/Description**: Go to the page for a testbed map and click the remove line button. Then click on a line.
3. **Assumptions**: A testbed map already exists with at least two dragables a line between them.
4. **Input Specification**: An existing testbed map and at least two dragable with a line between them.
5. **Output Specification**:
   a. **Expected output**: Line should be removed from the canvas.
   b. **Actual Output**:
      
      Before:
      ![Before image]
      
      After:
      ![After image]

6. **Test Result (Pass/Fail)**: PASS
7. **Pass/Fail Criteria**: The test passes if the testbed line is successfully removed from the canvas. The test fails if the dragable connected to the line is also removed, or if any other lines / dragables are removed.

IV.2.1.9 Convert Project Tango 3D maps to 2D maps.

1. **Test Title/Name**: Save 2D map from 3D model
2. **Test Summary/Description**: Save 2D image file from the 3D model that was built.
3. **Assumptions**: There is already a room that was scanned by the Tango App.
4. **Input Specification**: The save image button in the Tango App is pressed by the user.
5. **Output Specification**:
   a. **Expected output**: The map should be saved as an image to the Tango device.
b. **Actual Output:**

![Image of an arrow pointing to a location on a map]

6. **Test Result (Pass/Fail):** PASS
7. **Pass/Fail Criteria:** The test is passed if the image can be located in the Tango App’s data folder.

IV.2.1.10 Secure account access for users.
1. **Test Title/Name:** Brute Force Login
2. **Test Summary/Description:** Prevent scripts from logging in by brute forcing a username/password.
3. **Assumptions:** The attacker is using the same ip address the entire time, all attempts are incorrect to begin with.
4. **Input Specification:** Web browser open to login page, javascript that cycles through “all” possible username/password combos.
5. **Output Specification:**
   a. **Expected output:** Fail2ban blocks the attacker from attempting to login after 5 failed login attempts
   b. **Actual Output:** Fail2ban blocked the attacker from attempting to login after 5 failed login attempts
6. **Test Result (Pass/Fail):** PASS

IV.2.1.11 Secure account access for users.
1. **Test Title/Name:** SQL Injection Login
2. **Test Summary/Description:** Script which attempts to use SQL to force login without proper credentials.
3. **Assumptions:** Tests do not include any existing username/password combos.
4. **Input Specification:** Web browser open to login page, javascript that uses SQL lines as input to username and/or password.
5. **Output Specification:**
   a. **Expected output:** Login attempts fail.
   b. **Actual Output:** Login attempts failed.
6. **Test Result (Pass/Fail):** PASS
IV.2.1.12 Secure account access for users.

1. **Test Title/Name:** Edge Case Logins
2. **Test Summary/Description:** Script which uses a variety of username/password combos, including symbols, numbers, empty fields, and overflowing fields.
3. **Assumptions:** Tests do not include any existing username/password combos.
4. **Input Specification:** Web browser open to login page, javascript that uses set username/password tests in username and password fields.
5. **Output Specification:**
   a. **Expected output:** Login attempts fail.
   b. **Actual Output:** Login attempts failed.
6. **Test Result (Pass/Fail):** PASS

IV.2.1.13 Verify that users only see data they are supposed to see.

1. **Test Title/Name:** Correct Data Display
2. **Test Summary/Description:** This test makes sure that shown data is correct for the logged in user.
3. **Assumptions:** Someone is logged in
4. **Input Specification:** Logged in user.
5. **Output Specification:**
   a. **Expected output:** Data being displayed as expected.
   b. **Actual Output:** Correct data was displayed
6. **Test Result (Pass/Fail):** PASS
7. **Pass/Fail Criteria:** To pass, each page must display the data as expected. The test fails if any page has added, or missing data.

IV.2.1.14 Verify only admins can see admin pages.

1. **Test Title/Name:** Admin pages, admins
2. **Test Summary/Description:** This test makes sure that all admins can see pages that only admins are allowed to see.
3. **Assumptions:** Current user is an admin.
4. **Input Specification:** Logged in admin.
5. **Output Specification:**
   a. **Expected output:** Correct pages are available.
   b. **Actual Output:** Correct pages were available.
6. **Test Result (Pass/Fail):** PASS

IV.2.1.15 Verify non-admins can’t see admin pages.

1. **Test Title/Name:** Admin pages, non-admins.
2. **Test Summary/Description:** This test makes sure that admin pages are unavailable to non-admins.
3. **Assumptions:** Current user is not an admin.
4. **Input Specification:** Logged in user.
5. **Output Specification:**
   a. **Expected output:** Admin pages are blocked.
   b. **Actual Output:** Admin pages were blocked.
6. **Test Result (Pass/Fail):** PASS
IV.2.1.16 Add user.

1. **Test Title/Name**: Add user using the Add User form.
2. **Test Summary/Description**: This test makes sure that a new user is correctly added
3. **Assumptions**: All fields requesting info are filled out, current user is an admin.
4. **Input Specification**: Add User form has info in given fields.
5. **Output Specification**:
   a. **Expected output**: New user information is added to the database.
   b. **Actual Output**: New user information was added to the database.
6. **Test Result (Pass/Fail)**: PASS
7. **Pass/Fail Criteria**: To pass, the information input needs to be correctly added to the database. The password also must salted and hashed instead of being stored as plain text.

IV.2.2 Integration Tests

IV.2.2.1 Flask server and RabbitMQ stream and HTML Pages.

1. **Test Title/Name**: Integration testing of RealtimeController (Flask server), RabbitMQ stream, and HTML Pages.
2. **Test Summary/Description**: Verify that the WORRDS Flask server can consume from RabbitMQ and display the events in real-time on a webpage. This test validates the software requirement II.2.1.3 Serve web pages with real-time data from the Project Requirements Specification [2].
3. **Assumptions**:
   a. RealtimeController uses the correct connection credentials and queue information, e.g. exchange name, queue name, username, password, RabbitMQ server address.
   b. The RabbitMQ server “bunny” exists on the CASAS network.
   c. Sensor messages published to bunny are in JSON format with recognized fields.
   d. The “normal” publishing rate to bunny is 2-3 messages/second.
4. **Input Specification**: Real events from SHiBs are published to the all.testbed.casas exchange on bunny. RealtimeController will connect to bunny, consume the events, and render them on a webpage in real-time.
5. **Output Specification**:
   a. **Expected Output**: The real-time table and graph on the webpage will update in real-time with real sensor events.
   b. **Actual Output**: The real-time table and graph on the webpage update in real-time with real sensor events.
6. **Test Result (Pass/Fail)**: Pass

IV.2.2.2 Flask server and WORRDS Database and HTML Pages.

1. **Test Title/Name**: Integration testing of RealtimeController (Flask server) and WORRDSDb.
2. **Test Summary/Description**: Verify that RealtimeController stores sensor events in WORRDSdb in real-time. Also, verify that Flask can successfully query WORRDSdb.

3. **Assumptions**:
   a. RealtimeController uses the correct connection credentials and queue information, e.g. exchange name, queue name, username, password, RabbitMQ server address.
   b. The RabbitMQ server “bunny” exists on the CASAS network.
   c. Sensor messages published to bunny are in JSON format with recognized fields.
   d. The “normal” publishing rate to bunny is 2-3 messages/second.

4. **Input Specification**: Server bunny is simulated by a RabbitMQ server on localhost, while a Python program, rmqSim, will simulate SHiB publishers. rmqSim will publish 50 events. The All_Events table in WORRDSdb is initially empty.

5. **Output Specification**:
   a. **Expected Output**: To verify that Flask can store into the database, we expect the All_Events table to be populated with all 50 sensor events published by rmqSim. To verify that Flask can query the database, we expect the the left navbar in the UI to render all available testbeds, which are only stored in WORRDSdb.
   b. **Actual Output**: The All_Events table was populated with all 50 sensor events published by rmqsim. The navbar listed the expected testbeds.

6. **Test Result (Pass/Fail)**: Pass

IV.2.2.3 WORRDS Database and CASAS Database.
1. **Test Title/Name**: Data integrity test for the SQLAlchemy Python code in WORRDSRepository.py and __init__.py
2. **Test Summary/Description**: Verify that the SQLAlchemy object relationship mapping matches the WORRDSdb MySQL database.
3. **Assumptions**:
   a. The WORRDSdb MySQL database is online and reachable.
   b. The DDL for each table in WORRDSdb is correct.
   c. The entries in each table in WORRDSdb are stored correctly.
4. **Input Specification**: WORRDS is a database currently running on MySQL on the WORRDS sever. An object relationship mapping migration has been created using __init__.py.
5. **Output Specification**:
   a. **Expected output**: Each function that retrieves data using the SQLAlchemy object relationship mapping successfully executes without error. When comparing the data retrieved by the function with data
retrieved by executing a direct SQL query against the database, the data matches exactly.

b. **Actual Output**: Each function that retrieves data using the SQLAlchemy object relationship mapping successfully executed without error. When comparing the data retrieved by the function with data retrieved by executing a direct SQL query against the database, the data matched exactly.

6. **Test Result (Pass/Fail)**: PASS
7. **Pass/Fail Criteria**: Each function that retrieves data using the SQLAlchemy object relationship mapping successfully executed without error. When comparing the data retrieved by the function with data retrieved by executing a direct SQL query against the database, the data matched exactly. Therefore, the test passed.

IV.2.2.4 Flask server and Map Editor Save.
1. **Test Title/Name**: Save Map data.
2. **Test Summary/Description**: Go to the page for a testbed map and change the name and description then press the save button.
3. **Assumptions**: A testbed map already exists.
4. **Input Specification**: Modify the name and the description to contain new values.

5. **Output Specification**:
   a. **Expected output**: Once you hit save map, you should be able to refresh the page and have the changes retained.
   b. **Actual Output**:
      
      **Before:**
      
      **NAME:**
      
      Tokyo

      **DESCRIPTION**
      
      Map For Tokyo

      **After:**

      **NAME:**
      
      Tokyo - Changed

      **DESCRIPTION**
      
      Map For Tokyo - Changed

6. **Test Result (Pass/Fail)**: PASS
7. **Pass/Fail Criteria**: The test is considered passed if the map data shows the new values after the page is refreshed. The test is considered a failure if the text reverts back to the original value or anything other than the inputted text.
IV.2.2.5 Flask server and Map Editor Reset

1. **Test Title/Name:** Reset Map data
2. **Test Summary/Description:** Go to the page for a testbed map and change the name and description then press the reset button.
3. **Assumptions:** A testbed map already exists.
4. **Input Specification:** Modify the name and the description to contain new values.
5. **Output Specification:**
   a. **Expected output:** Once you hit reset map, the page should immediately reset to its original value.
   b. **Actual Output:**
      Before:
      ![Before Image]
      After:
      ![After Image]
6. **Test Result (Pass/Fail):** PASS
7. **Pass/Fail Criteria:** The test is considered passed if the map data reverts to its original state. The test is considered a failure if the text does not reset or changes to anything other than its original values.

IV.2.3 System Tests

IV.2.3.1 Verify that all integration and unit tests hold true when deployed in an environment similar to the production environment.

1. **Test Case Identifier:** Test behavior of WORRDS in an environment similar to the production environment.
2. **Test Title/Name:** System testing of WORRDS.
3. **Test Summary/Description:** Verify that core WORRDS features, including real-time display, map editor, and security, work as expected on a system similar to the worrds server.
4. **Assumptions:**
   a. Test machine should be installed with Debian 8 (Jessie), Nginx 1.6.2, Flask 0.10.1, and MySQL 5.7.1.
5. **Input Specification**: The current version of the WORRDS program, where core WORRDS features are known to work on the team’s personal Debian machines, running on the test machine worrds-dev.

6. **Output Specification**:
   a. **Expected Output**: Core features (real-time display, map editor, and security) continue to function.
   b. **Actual Output**: Core features (real-time display, map editor, and security) continue to function.

7. **Test Result (Pass/Fail)**: PASS

**IV.2.4 Performance Tests**

IV.2.4.1 Verify behavior when WORRDSdb is unreachable.
1. **Test Title/Name**: Robustness test for the Flask Python code
2. **Test Summary/Description**: Verify that the Flask Python code in WORRDSRepository.py and SecurityRepository.py exits gracefully when WORRDSdb is unreachable.
3. **Assumptions**:
   a. WORRDSdb is either offline or is not reachable over the CASAS network.
4. **Input Specification**: WORRDS, the database running on the WORRDS server, is a database that is currently not running on the WORRDS server.
5. **Output Specification**:
   a. **Expected output**: The Python code in WORRDSRepository.py and SecurityRepository.py will fail gracefully and display an error message on the webpage for the user.
   b. **Actual Output**: The Python code in WORRDSRepository.py and SecurityRepository.py failed gracefully and display an error message on the webpage for the user.

6. **Test Result (Pass/Fail)**: Pass
7. **Pass/Fail Criteria**: The Python code in WORRDSRepository.py and SecurityRepository.py failed gracefully and displayed an error message on the webpage for the user. Therefore, the test passed.

IV.2.4.2 Verify behavior when CASASDb is unreachable.
1. **Test Title/Name**: Robustness test for DbLoadSave.
2. **Test Summary/Description**: Verify that DbLoadSave exits gracefully when CASASDb is unreachable.
3. **Assumptions**:
   a. WORRDS is turned on. WORRDSdb is currently running on an instance of MySQL, which is bound to port 3306.
4. **Input Specification**: DbLoadSave is running and CASASDb is either offline or is not reachable over the CASAS network.
5. **Output Specification**: 

WORRDS – Final Report 29
a. **Expected output:** DbLoadSave will fail gracefully and print an error message to the console.

b. **Actual Output:** DbLoadSave failed gracefully and printed an error message to the console.

6. **Test Result (Pass/Fail):** PASS

7. **Pass/Fail Criteria:** DbLoadSave failed gracefully and printed an error message to the console. Therefore, the test passed.

IV.2.4.3 Verify the WORRDS application responds appropriately in situations where real-time data is temporarily inaccessible.

1. **Test Title/Name:** Robustness testing of RealtimeController (Flask server code that manages all real-time functionality).

2. **Test Summary/Description:** Verify the behavior of WORRDS when the Flask server cannot access real-time data. The following scenarios were tested:
   a. RealtimeController initially fails to connect to bunny server.
   b. Server “bunny” stops and restarts.
   c. No data is being published to bunny by SHiBs.

3. **Assumptions:**
   a. RealtimeController uses the correct connection credentials and queue information, e.g. exchange name, queue name, username, password, RabbitMQ server address.
   b. The RabbitMQ server “bunny” exists on the CASAS network.
   c. Sensor messages published to bunny are in JSON format with recognized fields.

4. **Input Specification:** Server bunny is simulated by a RabbitMQ server on localhost, while a Python program, rmqSim, will simulate SHiB publishers. SHiBs are publishing JSON sensor messages to “bunny” at ~2.0 messages/second.

5. **Output Specification:**
   a. **Expected Output:** While it is unable to connect to bunny (cases 3a and 3b) RealtimeController repeatedly tries to reconnect and resumes consuming once bunny is reachable. If no data is being published for awhile (case 3c), RealtimeController continues to listen and resumes consuming once a message is published. Rather than break, WORRDS should always throw exceptions or print error messages.
   b. **Actual Output:**
      i. **Case a:** RealtimeController initially fails to connect to bunny server.
         RealtimeController repeatedly tries to reconnect, printing “Creating RabbitMQ connection. Failed to connect to server. No connection. Attempting to reconnect…” Once bunny is live again, RealtimeController establishes a connection and waits to consume events, printing “Creating RabbitMQ connection. Declaring queue and exchange. [x] Waiting for events. To exit, press Ctrl+C.”
ii. **Case b: Server “bunny” stops and restarts.**

When bunny is stopped in the middle of consuming, RealtimeController times out and repeatedly tries to reconnect, printing “No connection. Attempting to reconnect…” When bunny is restarted, RealtimeController successfully connects and resumes consuming, printing “Creating RabbitMQ connection. Declaring queue and exchange. [x] Waiting for events. To exit, press Ctrl+C. [x] Received: ...”

iii. **Case c: No data is being published to bunny by SHiBs.**

RealtimeController continues to listen and resumes consuming when events are published. The received message “Attempting to reconnect...” indicates a publisher reconnected to bunny and began to publish events.

6. **Test Result (Pass/Fail): PASS**

IV.2.4.4. Verify that all pages load in reasonable time.

1. **Test Title/Name:** Page Load Time
2. **Test Summary/Description:** This test makes sure that all of the data visualization pages load in a reasonable amount of time.
3. **Assumptions:** User is logged in.
4. **Input Specification:** Load a page and start a timer
5. **Output Specification:**
   a. **Expected output:** Each page loads in a reasonable amount of time.
   b. **Actual Output:** N/A or each page was able to load in a reasonable amount of time. However we got a bunch of bad gateways with the Real Time Data and System Information tables, but this was very random and is most likely a problem with the CASAS network.

6. **Test Result (Pass/Fail): PASS**
7. **Pass/Fail Criteria:** To pass, the load time on each of the pages must load in a reasonable amount of time. If any of the pages fail to meet this standard the test fails.

IV.2.4.5 Verify the user interface responsiveness of the Tango App.

1. **Test Title/Name:** Test Tango App Responsiveness.
2. **Test Summary/Description:** This test makes sure that all of the functionality of the Tango App is resolved in a timely enough manner such as to avoid inducing frustration in the user.
3. **Assumptions:** Android device is powered on with the Tango app installed.
4. **Input Specification:** Start using the Tango app installed.
5. **Output Specification:**
   a. **Expected output:** A time below a certain threshold for each function.
   b. **Actual Output:** All Tango App functions done within 0.5 seconds

6. **Test Result (Pass/Fail): PASS**
7. **Pass/Fail Criteria:** To pass, each Tango App function resolves themselves within 0.5 seconds of starting. If any of the functions fail to meet this standard the test fails.
IV.2.4.6 File upload reliability of a file generated by the Tango Device
1. **Test Title/Name**: Test Tango App File Upload Reliability
2. **Test Summary/Description**: This test makes sure that the maps generated by the Tango App are successfully stored into the WORRDS database.
3. **Assumptions**: There exists a map image file to upload to the WORRDS database.
4. **Input Specification**: The file is dragged and dropped into the drag and drop zone on the upload webpage.
5. **Output Specification**:
   a. **Expected output**: The file is successfully uploaded to the WORRDS database
   b. **Actual Output**: Succeeds from computer to WORRDS database but fails from Tango Device to WORRDS database.
6. **Test Result (Pass/Fail)**: FAIL
7. **Pass/Fail Criteria**: To pass, the map image file generated by the Tango App must have all been successfully uploaded to the WORRDS database.

IV.2.4.7 Real-time UI Performance
1. **Test Title/Name**: Test Real-time UI Performance Under Stress.
2. **Test Summary/Description**: Verify the behavior of the real-time UI (e.g. real-time event tables and graphs) when a lot of sensor messages are published to bunny in a short time period.
3. **Assumptions**:
   a. RealtimeController uses the correct connection credentials and queue information, e.g. exchange name, queue name, username, password, RabbitMQ server address.
   b. The RabbitMQ server “bunny” exists on the CASAS network.
   c. Sensor messages published to bunny are in JSON format with recognized fields.
   d. The “normal” publishing rate to bunny is 2-3 messages/second.
4. **Input Specification**: Server bunny is simulated by a RabbitMQ server on localhost, while a Python program, rmqSim, will simulate SHiB publishers. rmqSim will read from a text file of thousands of real sensor messages and publish to the RabbitMQ server at 5 messages/second and 10 messages/second.
5. **Output Specification**:
   a. **Expected Output**: The real-time UI will keep up with the increase in input with no detriment.
   b. **Actual Output**: In both cases, all events received were eventually rendered in the All Events real-time table and graph. However, the graph’s accuracy proved limited, as it could not render the difference between a message rate of 5 messages/second and 10 messages/second.
i. Publishing at 5 messages/second (1 message every 0.2 seconds)

![Realtime Events For All Sites](image1)

ii. Publishing at 10 messages/second (1 message every 0.1 seconds)

![Realtime Events For All Sites](image2)

i.

6. **Test Result (Pass/Fail):** PASS

7. **Pass/Fail Criteria:** All actual output matched expected output, thus the tests passed. Although the graph did not render the difference between high speed message rates, it still demonstrated that events were received and consumed at a constant rate, which was its original purpose.

**IV.2.5 User Acceptance Tests**

IV.2.5.1 User can view dynamic webpages.
1. **Test Title/Name:** User can view Dynamic Web Pages
2. **Test Summary/Description:** Make sure the user is able to navigate to a dynamic web page.
3. **Assumptions:** User is logged in.
4. **Input Specification:** A user will log into WORRDS and navigate to a dynamic web page.
5. **Output Specification:**
   a. **Expected output:** The user should be able to navigate to a dynamic web page.
   b. **Actual Output:** The user was able to successfully navigate to a dynamic web page which was http://worrds-dev/realtimeoverview in this case. Below is an image:
6. **Test Result (Pass/Fail):** PASS

7. **Pass/Fail Criteria:** The test passes if the user is able to correctly navigate to a dynamic web page and fails otherwise.

---

**IV.2.5.2 User can view static web files.**

1. **Test Title/Name:** User can view static web files.

2. **Test Summary/Description:** Make sure the user is able to navigate to a static web page.

3. **Assumptions:** User is logged in.

4. **Input Specification:** A user will log into WORRDS and navigate to a static web page.

5. **Output Specification:**
   a. **Expected output:** The user should be able to navigate to a given static web page.
   b. **Actual Output:** Successfully navigated to the static web file, in this case it was worrds-dev/static/scripts/bootstrap.js.

6. **Test Result (Pass/Fail):** PASS

7. **Pass/Fail Criteria:** The test passes if we are able to correctly navigate to a static web file.

---

**IV.2.5.3 User can view and create new testbed maps.**

1. **Test Title/Name:** Create Map for TestBed.

2. **Test Summary/Description:** Go to the page for a testbed and click the add new map button. A new map should show up on the page to be edited.

3. **Assumptions:** A testbed already exists with 0 or more maps.

4. **Input Specification:** An existing testbed.

5. **Output Specification:**
   a. **Expected output:** A new testbed map should appear in the list next to the desired testbed.
b. Actual Output:

Before:

<table>
<thead>
<tr>
<th>id</th>
<th>Name</th>
<th>Description</th>
<th>Maps:</th>
</tr>
</thead>
</table>
| 1  | Tokyo  | site for populd  | Tokyo
    |        |                  | Desc: Map For Tokyo                     |

After:

<table>
<thead>
<tr>
<th>id</th>
<th>Name</th>
<th>Description</th>
<th>Maps:</th>
</tr>
</thead>
</table>
| 1  | Tokyo  | site for populd  | Tokyo
    |        |                  | Desc: Map For Tokyo                     |

6. Test Result (Pass/Fail): PASS

7. Pass/Fail Criteria: The test passes if the new testbed map is created. The test fails if the new testbed map is not created, or if the testbed map is created for a testbed other than the one we are trying to add to.

IV.2.5.4 Multiple user logins.

1. Test Title/Name: Multiple user login.
2. Test Summary/Description: Multiple people are able to login concurrently
3. Assumptions: At least two users exist in the database.
4. Input Specification: Logged in users
5. Output Specification:
   a. Expected output: Each person can access only their own personal information.
   b. Actual Output: Each person can access only their own personal information.

6. Test Result (Pass/Fail): PASS

7. Pass/Fail Criteria: To pass, each user must have the permissions specific to them, and not those of other people also currently logged in.
V. Description of Final Prototype

Figure 7. Overview of WORRDS in relation to the CASAS network. WORRDS components are in red and yellow.

Figure 7 depicts the data flow to WORRDS, which resides on the WORRDS server. Copies of the SHiB events are published directly to the RabbitMQ server, bunny. The Flask app on the WORRDS server consumes the events from bunny and queries WORRDSdb in order to populate the dynamic webpages, which contain real-time charts, maps, and tables. A user on the web browser can navigate to these webpages, served by Nginx.

VI.1 User Interface

The user interface consists of several web pages, each of which uses Bootstrap CSS, and KnockoutJS to provide a scaling, mobile-friendly interface. Before a user can reach any of the WORRDS pages, they must login with the login page. They must also be authorized to view the page in the WORRDS database. If they are not logged in when they navigate to the site, the user is prompted with a login page.
VI.1.1 Login

Figure 8. WORRDS login page. The login page allows the user to enter their username and password so they may access the home page. Each user in the WORRDS system can have different permissions which allows them access to different parts of WORRDS.

VI.1.2 User Management

Figure 9. Register User page (top) allows an admin to add a new user. The Admin Management page (bottom) allows an admin to grant an existing user admin privileges, or take those privileges away from an admin. Users with administrator privileges have the ability to create users with the Register User page. Administrators also have the ability to promote regular users to become administrators with the admin management page.
VI.1.3 Home Page

Figure 10. WORRDS Home Page. The home page lists all of the SHiB site pages that the authenticated user has access to according to their permissions and three real-time widgets displaying messages/second, total events received today, and events received today from each testbed.

VI.1.4 Real-time data flow

Figure 11. Real-time reports page for all sites. Every time a new event is received, the scrolling table is updated and the frequency graph spikes.
VI.1.5 Battery Pages

Figure 12. The battery overview page displays a table with all batteries a user has access to. The columns of the table are sortable, so users can view the batteries in order of id, target name, battery percentage, last seen, and testbed.

Figure 13. The battery diagnostics page for a single testbed. The summary row at the top allows users to draw conclusions about the state of the testbed’s batteries at a glance. Dropdown lists for critical, warning, and good battery levels give users a list of batteries to watch. Finally, the D3 graph contains pie charts representing each battery.
As a battery drops in percentage, it increases in size and appears closer to the border of the graph. Hovering over a pie graph brings up its name, percentage, and last seen date in a tooltip.

**VI.1.6 Map Editor**

![Map Editor Page](image)

**Figure 14.** Map editor page in Edit mode. Map editor tools (above) allow a user to add/remove/move dragables, lines, and packages on the map (below). Users can also save the current version of the map, undo changes, and change the background of the map from blue to an image of their choice that is uploaded to a webserver.
**Figure 15.** Map editor page in Live mode. When a sensor sends an event, its associated package (gray box) blinks a different color in real-time.

**VI.1.7 Behavior Overview Page**

**Figure 16.** Behavior overview page. Each ring represents a week, with the outer ring being the most recent week and the inner ring being four weeks ago. Each segment represents an hour's time. The color of the segment symbolizes the most dominant activity during that hour. Hovering over a segment reveals a tooltip with the hour
description and name of the dominant behavior. This chart allows researchers to observe patterns in behavior over a month.

**IV.1.8 Map Upload Page**

![Upload Testbed Map](image)

**Figure 17.** Map upload page. Users can upload map images, such as ones generated using the Google Tango app, to the database for use on the map editor.
IV.I.9 Control Panel Page

**Figure 18.** Control panel page (admins only). The control panel contains scrolling tables that display user, permission, and testbed information currently in the database. This page allows admins to check on which users have which permissions to which testbeds. Each row also links to its respective page, e.g. clicking on a user brings the viewer to the user's profile.
**Figure 19.** User profile page. Each user can access their own profile and view their permissions.
VI.2 WORRDS Server

The WORRDS server component contains two main servers: a Flask application server and an Nginx web server. Both were created and configured as part of this project.

VI.2.1 Flask Server Functionality

Flask is a lightweight Python microframework that is well-suited for dynamic web programming and system customization via extensions. In this project, the Flask server performs several functions: serving webpages, routing, providing a Web API, and storing real-time events in the database.
VI.2.1.1 Serving Dynamic Webpages

First and foremost, Flask populates webpages, or templates, with data from WORRDSdb and real-time sensor data from RabbitMQ. For example, to render an instance of a template for a particular testbed, Flask functions query for data relevant to the testbed and pass the results as variables to the page. The extensibility provided by Flask templating facilitated the creation of many types of webpages for many testbeds.

Almost as fast as the RabbitMQ simulation can produce sensor messages in the queue, the Flask application consumes them, converts them to JSON, and sends them via socket to the template in real-time. Gunicorn, a Python WSGI HTTP server that interfaces between the web server and web applications, facilitates the communication. As the template receives relevant events, it updates the UI. Thus the end user can view real-time sensor data in textual or chart format with low latency.

VI.2.1.2 Routing

Second, Flask provides the routing functionality behind navigating to webpages. Using lines called decorators, Flask allows for beautiful URLs that are bound to a Python function. In this project, these functions usually render templates, but they may also perform other processes as well. The current routing setup utilizes dynamic URLs, allowing Flask to pattern match and serve the corresponding template. For example, upon entering “https://worrds-dev/battery/<id>” in the browser address bar, Flask routes to the battery page for the testbed with that id. Flask also has capabilities to generate URLs, a feature that may be used in the future to save developers time defining decorators for every template.

VI.2.1.3 Web API

Third, Flask provides a Web API for client applications, such as the Map Editor, to retrieve data needed to accomplish their tasks.

VI.2.1.4 Real-time Event Logging

Finally, Flask logs the real-time events received in WORRDSdb to use as historical data later.

VI.2.2 Nginx Server Functionality

Nginx is a lightweight, high-performance HTTP web proxy ideal for serving static webpages, redirecting requests, and load balancing across multiple application. In this prototype, the Nginx server performs two main functions: proxying client requests and serving static webpages.
VI.2.2.1 Proxying Client Requests

The Nginx server is configured to forward client requests for webpages to the Flask server. It redirects any requests coming through port 80 to port 443 for secure communication over HTTPS.

VI.2.2.2 Serving Static Webpages

If the client requests a static webpage, Nginx is configured to serve it directly rather than consult Flask unnecessarily. This results in excellent performance for loading static webpages. In the future, WORRDS may utilize Nginx caching capabilities for even better performance.

VI.3 WORRDS Database

There are two databases that relate to the WORRDS project. CASASDb, hosted on PostgreSQL, contains the master table of event data. This table, called all_events, contains the columns by, category, channel, event_id, message, package_type, sensor_type, serial, target, tbname, stamp, timezone, and uuid. A second table, experiment_all_data, stores behavioral information derived from specific events. Reporting and analytics software used by CASAS relies on the aforementioned database.

WORRDDb, hosted on MySQL, contains a cache of all_events and experiment_all_data. The cache consists of events and experiments from CASASDb that occurred no more than $X$ days ago. $X$ is a value specified by the user in the DbLoadSave configuration file. In addition to the cache, the WORRDS database will contain tables relating to user management and testbed map information. Flask retrieves records from WORRDDb using the GetData service and uses those records to populate real-time charts and tables on dynamic web pages.

Team Wildlings has developed several utilities that interact with WORRDDb. GetData retrieves data from a SQL database and displays it on a webpage. This utility is made up of multiple pieces. The PHP component running on the server connects to the database and executes a query. The Javascript component running on the client fetches data from the PHP component via AJAX.

GetData makes use of PHP Data Objects, enabling the utility to be modular. If, in the future, CASAS wanted GetData to fetch results from a different DBMS, GetData could do so with the modification of one line of code.

DbLoadSave is a Java applet that migrates data from CASASDb to WORRDDb. It works by querying CASASDb and storing the results into an in-memory array. Each record in that array is then inserted into WORRDDb.

SQLGenerator is a C++ application that generates SQL INSERT statements from a character-delimited file and stores them into an output file. The table name, column
names, input filename, output filename, and the character used as the delimiter are passed in as arguments when SQLGenerator is run.

VI.4 Miscellaneous Functions and Interfaces Implemented

VI.4.1 Security

Flask is a lightweight Python microframework. Since Flask is more lightweight, most of the features had to be implemented manually. The login feature is implemented via a Python library called Flask-Login. This keeps track of the user that is logged in, as well as other metadata about them, e.g. last login time. WORRDS also incorporates Fail2Ban, which blocks an IP address after five incorrect login attempts for five minutes.
VI. Conclusions and Future Work

VI.1. Limitations and Recommendations

- Limitation of the server and the number of database connections it can accept at any given time causes an occasional lag
  - The machine the server runs on may be lacking enough RAM
  - Increasing the CPU speed and RAM of the machine, or allocating a physical machine for WORRDS (ideal), may solve this issue
  - Utilizing more efficient queries so that database overhead decreases might also work.
- Images stored in the database cannot be extracted and displayed on the map editor
  - Consulting someone with more experience in web development may solve this issue.
  - Alternatively, images could instead be uploaded to a separate sharing site to then provide the MapEditor page with the image’s url for display - as is currently being done.

VI.2. Future Work

All functional requirements for the WORRDS project were completed including additional work to fix the events upload to the RabbitMQ server which was assigned mid-project. Additional ways this project could be extended is as follows:

- Create a Testbed Map Placement table that relates to a set of map dragables and map packages so that on the map editor, a sensor layout and map image can be chosen and shown on the editor independently.
- Implement group-based permission inheritance
- Implement real-time resident activity visualization for live monitoring
- Write an algorithm to predict failing batteries with machine learning
- Write an algorithm for anomaly detection in system and behavioral events since WORRDS cannot automatically detect anomalies in SHiB Sensor Data.
- WORRDS contains a large data set that is not completely being utilized in the data visualization graphs.
VII. Acknowledgements

Special thanks to Dr. Aaron Crandall and Dr. Sakire Arslan Ay for their guidance and support, and to Biswaranjan Das, Brian Thomas, and Tyler Walker for their assistance.
VIII. References

IX. Appendix A – Team Information

Team Wildlings consists of Joanne Magtibay, Daniel Pepka, Pat McKee, Benjamin Cheung, Bryan Barrows, and Trey Ottaway.
X. Appendix B - Project Management

The team met once a week with the instructor and mentor. At these meetings, the team would communicate each individual’s progress on their tasks, ask for advice/assistance with roadblocks, and welcome critique from the instructor and mentor about implementation or design. The meetings were designed to hold each member responsible for their assigned component and ensure the team delivered on all the project requirements before the deadline.

The team would regularly meet independently when in-person collaboration was necessary. On average, the team would have such meetings about twice a week for 1-3 hours at a time. In the last month before the deadline, the team would meet more frequently and longer, for an average of 3 times per week and 2-6 hours at a time. During the meetings, the team would work on software development, address design issues, or do a final review of documents before submission. These sessions were probably the most beneficial for the team, since it forced individuals to focus on work and seize the opportunity to receive assistance from teammates. These meetings also served as a great team-building experience.

The team also kept in constant contact over group instant messaging and collaborated on documentation digitally using Google Drive. Being able to ask each other questions and coordinate with technology at any time allowed for more efficient meetings.