Software Design Specification

For

Chariot

Submitted by

Medard Azandegbe, Kamalludin Colaire,   
Ryan Hassing, John Ho, Juan Garcia Lopez, Christopher Mak, Enioluwa Segun

|  |  |
| --- | --- |
| **Instructor:** Dr. Gregory Hislop |  |
| **Team Members:** 7 |  |
| **Cycle:** 1 |  |
| **Date Submitted:** |  |

Document template copyright 2005-2019, Gregory W. Hislop. Version 2.3. Use permitted under Creative Commons license CC-BY-NC-SA. See http://creativecommons.org/licenses/by-nc-sa/3.0/.

Grading Rubric - Design Specification

This rubric outlines the grading criteria for this document. Note that the criteria represent a plan for grading. Change is possible, especially given the dynamic nature of this course. Any change will be applied consistently for the entire class.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Achievement** | **Minimal** | **Exemplary** | **Pts** | **Score** |
| **Content** | Section(s) missing, not useful, inconsistent, or wrong. | Provides all relevant information correctly and with appropriate detail |  |  |
| Introduction |  |  | 10 |  |
| Architectural Description |  |  | 10 |  |
| Interface Description |  |  | 15 |  |
| Detailed Design |  |  | 50 |  |
| **Grammar and Spelling** | Many serious mistakes in grammar or spelling | Grammar, punctuation, and spelling all correct | 5 |  |
| **Expression** | Hard to follow or poor word choices | Clear and concise. A pleasure to read | 5 |  |
| **Tone** | Tone not appropriate for technical writing | Tone is consistently professional |  |  |
| **Organization** | Information difficult to locate | All information is easy to find and important points stand out | 5 |  |
| **Layout** | Layout is inconsistent, visually distracting, or hinders use | Layout is attractive, consistent, and helps guide the reader |  |  |
| **Late Submission** |  |  |  |  |
| **Total** |  |  | 100 |  |

Table of Contents

1 Introduction

[1.1 Scope 4](#_Toc23774997)

[1.2 Definitions, Acronyms, and Abbreviations 4](#_Toc23774998)

[1.3 Requirements Traceability Matrix 4](#_Toc23774999)

[2 Architectural Description 7](#_Toc23775000)

[3 Interface Description 7](#_Toc23775001)

[3.1 User Interface 7](#_Toc23775002)

[3.2 Data Interface 8](#_Toc23775003)

[3.3 Programming Interface 8](#_Toc23775004)

[4 Detailed Design 8](#_Toc23775005)

[UI1 – Installation 8](#_Toc23775006)

[UI2 – Interrogator Selector 10](#_Toc23775007)

[UI3 – Load Config Recipe 11](#_Toc23775008)

[UI4 – Name/IP Address 11](#_Toc23775009)

[UI5 – Password 11](#_Toc23775010)

[UI6 – Time 12](#_Toc23775011)

[UI7 – Device 12](#_Toc23775012)

[UI8 – Antenna Number 12](#_Toc23775013)

[UI9 – Tx Power 12](#_Toc23775014)

[UI10 – Rx Sensitivity 12](#_Toc23775015)

[UI11 – Model 12](#_Toc23775016)

[UI12 – Pop 13](#_Toc23775017)

[UI13 – Recipe Confirmation 13](#_Toc23775018)

[UI14 – Add Another Sensor 13](#_Toc23775019)

[UI15 – Run Diagnostics (Visualizer) 13](#_Toc23775020)

[UI16 – Choose Data Output Format 13](#_Toc23775021)

[UI17 – Data Output 13](#_Toc23775022)

[﷟HYPERLINK "bookmark://\_Toc23775023" 14](#_Toc23775023)

Introduction

## Scope

This document describes the design and implementation details of Chariot, an IoT sensor framework that will be used in the development of smart cities, as well as in research involving IoT sensors. The framework is largely based on Dr. Bill Mongan’s IoT sensor framework, with several key improvements to allow for expanded use cases as defined by researchers and technicians. An in-depth look of these requirements can be found in the *Software Requirements Specification*.

## Definitions, Acronyms, and Abbreviations

Shown below is a list of technical terms that you will encounter in this document.

1. Internet of Things (IoT) - the interconnection via the Internet of computing devices embedded in everyday objects, enabling them to send and receive data.
2. IoT device – a hardware device that records data about its environment and has the capability to connect to a network and communicate with other devices.
3. Data Collection Episode – the timeframe during which Chariot is actively collecting and storing data received from IoT devices
4. Received Data – data transmitted, or in transmission from an IoT device that has not yet been stored
5. Collected Data – data received from an IoT device that has been saved to a storage unit
6. Storage Unit – where collected data is saved, be it a database, csv file, or other format or data structure
7. Network - a collection of configured, connected IoT devices
8. Module – an addon to Chariot that adds functionality or compatibility tools to the system
9. UID – User Interface Design, the process of making interfaces for software or computer devices to focus on looks and style. It is a straight forward way to approach User experience improvement.
10. UI – User Interface, a means for a user to interact with a system to interact with one another.

## Requirements Traceability Matrix

This section maps the relationship between requirement statements and detailed design entities. As such it shows how requirements are covered by the design and demonstrates the purpose for which design entity exists.

The values in the cells of the table show which requirements provide the purpose for each entity. The cell values are:

* **Blank** – the design entity does not implement any of that requirement
* **P** for Primary - the design entity implements all or most of the requirement
* **S** for Secondary – the design entity implements a smaller but essential part of the requirement

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **DE1** | **DE2** | **DE3** | **DE4** | **DE5** | **DE6** | **DE7** | **DE8** | **DE9** | **DE10** |
| **FR1** | P |  |  |  |  |  |  |  |  |  |
| **FR2** |  | P |  |  |  |  |  |  |  |  |
| **FR3** |  |  | P |  |  |  |  |  |  |  |
| **FR4** |  |  |  | P |  |  |  |  |  |  |
| **FR5** |  |  |  |  |  |  | P |  |  |  |
| **FR6** |  |  |  |  | P |  |  |  |  |  |
| **FR7** |  |  |  |  |  | P |  |  |  |  |
| **FR8** |  |  |  |  | P |  | P |  |  |  |
| **FR9** |  |  |  |  |  |  |  |  |  |  |
| **FR10** |  |  |  |  |  |  |  |  |  | P |
| **FR11** |  |  |  |  |  |  |  | P |  |  |
| **FR12** |  |  |  |  |  |  |  |  |  |  |
| **FR13** |  |  |  |  |  |  | S |  | P |  |
| **FR14** |  |  |  |  |  |  |  |  |  |  |
| **FR15** |  |  |  |  |  |  |  |  |  |  |
| **FR16** |  |  |  |  |  |  |  |  |  |  |
| **FR17** |  |  |  |  |  |  |  |  |  |  |
| **FR18** |  |  |  |  |  |  |  |  |  |  |
| **FR19** |  |  |  |  |  |  |  |  |  |  |
| **FR20** |  |  |  |  |  |  |  |  |  |  |
| **FR21** |  |  |  |  |  |  |  |  |  |  |
| **FR22** |  |  |  |  |  |  |  |  |  |  |
| **FR23** |  |  |  |  |  |  |  |  |  |  |
| **FR24** |  |  |  |  |  |  |  |  |  |  |
| **FR25** |  |  |  |  |  |  |  |  |  |  |
| **FR26** |  |  |  |  |  |  |  |  |  |  |
| **FR27** |  |  |  |  |  |  |  |  |  |  |
| **FR28** |  |  |  |  |  |  |  |  |  |  |
| **FR29** |  |  |  |  |  |  |  |  |  |  |
| **FR30** |  |  |  |  |  |  |  |  |  |  |
| **FR31** |  |  |  |  |  |  |  |  |  |  |
| **FR32** |  |  |  |  |  |  |  |  |  |  |
| **FR33** |  |  |  |  |  |  |  |  |  |  |
| **FR34** |  |  |  |  |  |  |  |  |  |  |
| **FR35** |  |  |  |  |  |  |  |  |  |  |
| **FR36** |  |  |  |  |  |  |  |  |  |  |

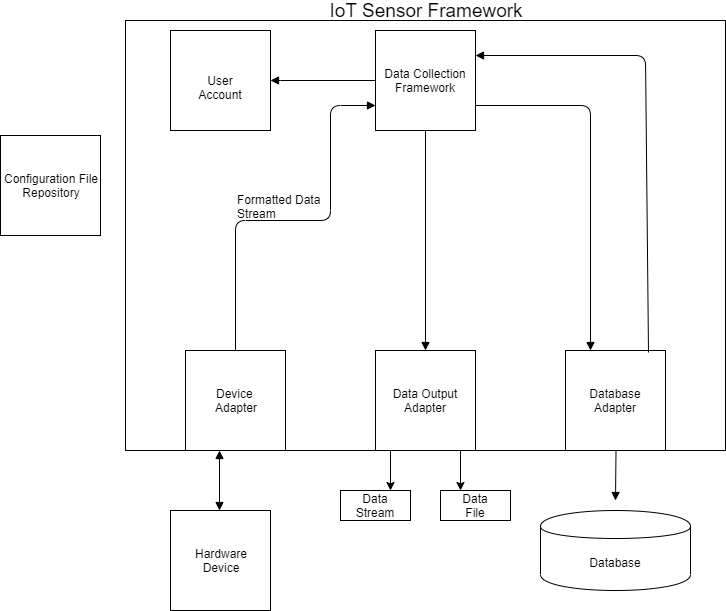
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **DE11** | **DE12** | **DE13** | **DE14** | **DE15** | **DE16** |
| **FR1** |  |  |  |  |  |  |
| **FR2** |  |  |  |  |  |  |
| **FR3** |  |  |  |  |  |  |
| **FR4** |  |  |  |  |  |  |
| **FR5** |  |  |  |  |  |  |
| **FR6** |  |  |  |  |  |  |
| **FR7** |  |  |  |  |  |  |
| **FR8** |  |  |  |  |  |  |
| **FR9** |  |  |  |  |  |  |
| **FR10** |  |  |  | S |  |  |
| **FR11** | P | P |  | P |  |  |
| **FR12** | P | P |  |  |  |  |
| **FR13** |  |  |  |  |  |  |
| **FR14** | P | P |  |  |  |  |
| **FR15** |  |  |  |  |  |  |
| **FR16** |  |  |  |  |  |  |
| **FR17** | P |  |  |  |  |  |
| **FR18** | P | P |  |  |  |  |
| **FR19** | P | P |  |  |  |  |
| **FR20** |  | P |  |  |  |  |
| **FR21** |  | P |  |  |  |  |
| **FR22** |  |  |  |  |  |  |
| **FR23** |  |  |  |  |  |  |
| **FR24** |  |  |  |  | P |  |
| **FR25** |  |  |  |  |  | P |
| **FR26** |  | P |  |  |  |  |
| **FR27** |  | P |  |  |  |  |
| **FR28** |  |  |  |  |  | P |
| **FR29** |  | P |  |  |  |  |
| **FR30** |  |  | P |  |  |  |
| **FR31** |  |  | P |  |  |  |
| **FR32** | S |  | P |  |  |  |
| **FR33** |  |  |  |  |  |  |
| **FR34** |  |  |  |  |  |  |
| **FR35** |  |  |  |  |  |  |
| **FR36** |  |  |  |  |  |  |

Figure 1 – Traceability between Requirements and Design Entities

# Architectural Description

This section aims to provide a high-level overview of the design of the system, in other words, components decomposition.

**CD1 – System Architecture**



**Figure 2.1**: System Architecture of Chariot.

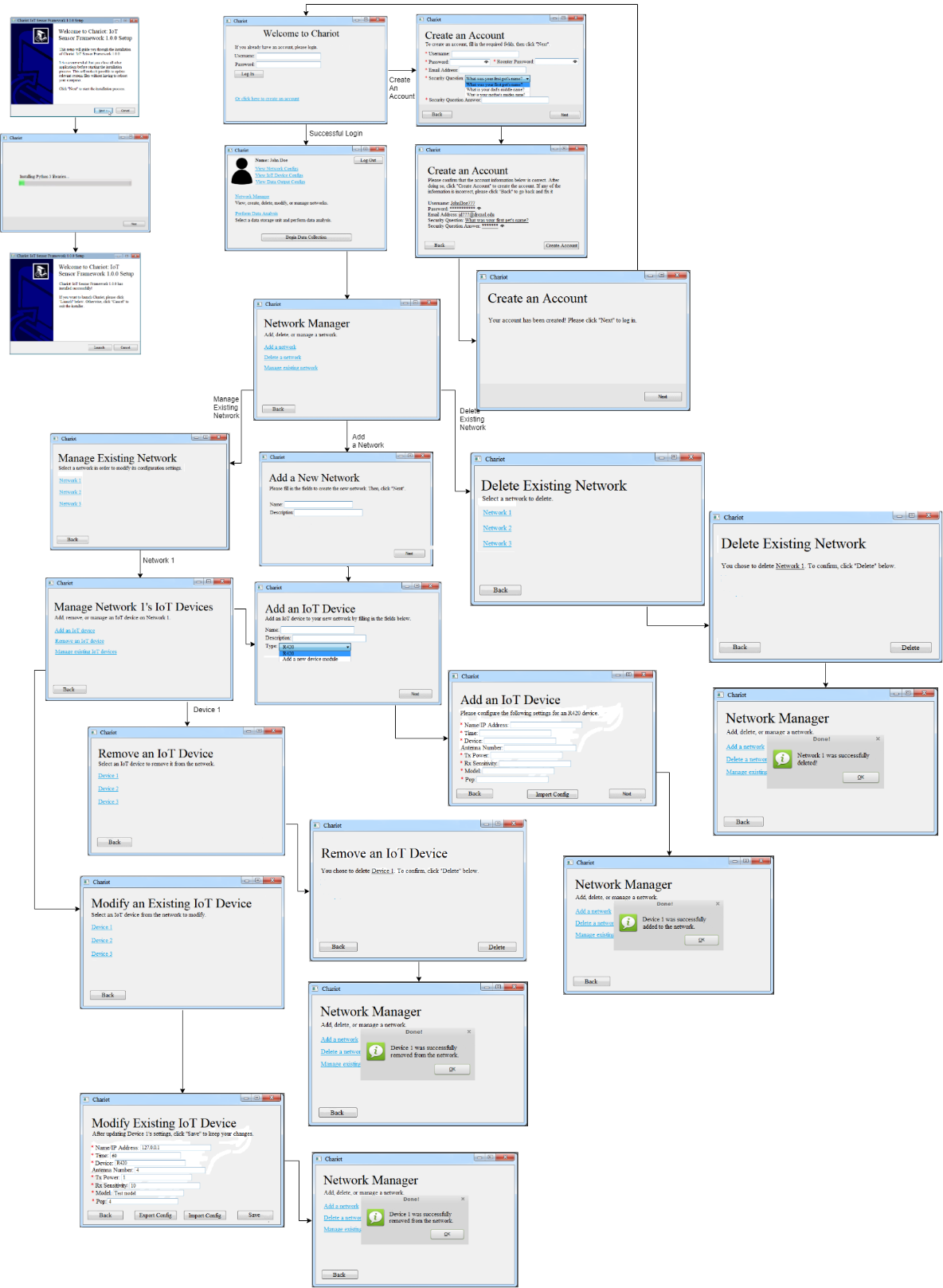
Figure 2.1 displays a high-level architecture of the components that encompass Chariot. Chariot is a platform for IoT device management, data storage, and data analysis. Users can add, remove, and configure a variety of devices on a single network. Given these requirements, it must be that a component of the system is the data collection framework working together with the device adapter to connect a hardware device to Chariot. Once a device is added to a network, a data collection episode can be started. To save the received data to a database of the user’s choosing, the data collection framework component works with the database adapter to write to a database of the user’s choice. For a more detailed look at the overall system class diagram, go to Figure 4.28.

**CD2 – Installer**

The Chariot installer is what is used to install the system onto the end-user's device. This component allows the Chariot installation process to be streamlined, so that a non-technical user can install and use the system on their device. This component will be done using Electron, which allows for desktop GUI applications using HTML, CSS, and JavaScript. See DE1 for more detailed information.

**CD3 – Data Collection Framework GUI**

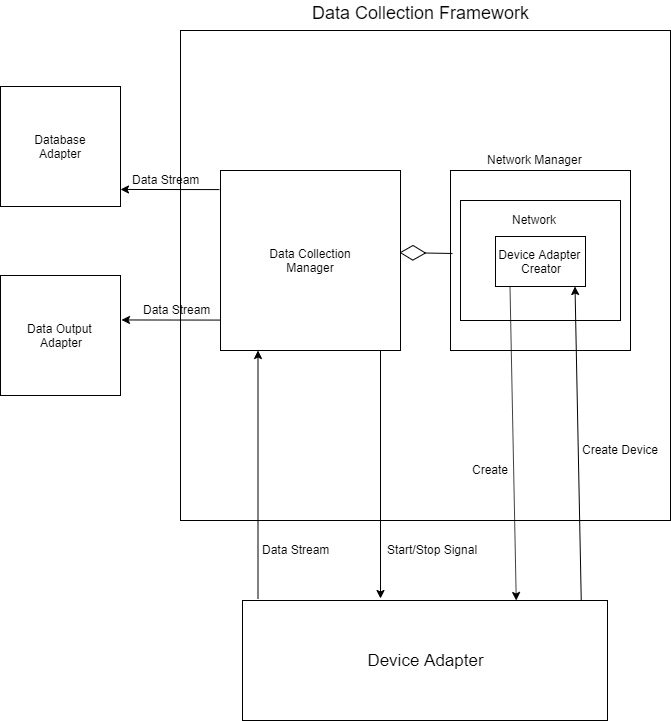
This is the main component in which the end-user will interact with. After installation and launching the software, the user will use the GUI to perform various tasks such as creating and logging into an account, configuring networks and IoT devices, and data collection. More specific information about each individual screen can be found in the Screen Sequence types throughout Section 4.



**CD4 – User Account**

The user account component is used by Chariot to enhance portability of the system. It acts to allow the system to save configurations beyond the local level and onto an account accessible by other machines so that a user does not have to re-configure networks. In particular, the user account is linked with the configuration file repository so that configurations made by the user are stored there and can be retrieved. More detailed information can be found in DE2 and DE3.

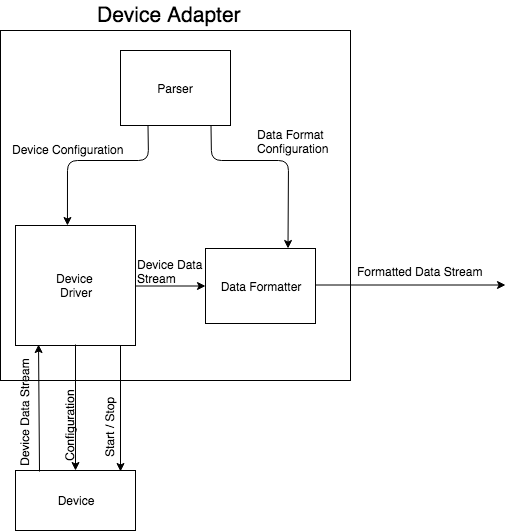
**CD5 – Data Collection Framework**



**Figure 2.2**

The data collection framework is a module that can be further decomposed into the data collection and network management components. These two components work together to establish a data collection episode. That is, the user must establish a network configuration through the network management component so the data collection module can be run. The data collection component acts to receive data from IoT devices and sends the data to the database adapter. The network manager component acts to store device adapter configurations so that the data collection manager can manage those devices during a data collection episode. See DE12 for more detailed information on the data collection process and its interactions.

**CD6 –Device Adapter, Device Adapter Creator**

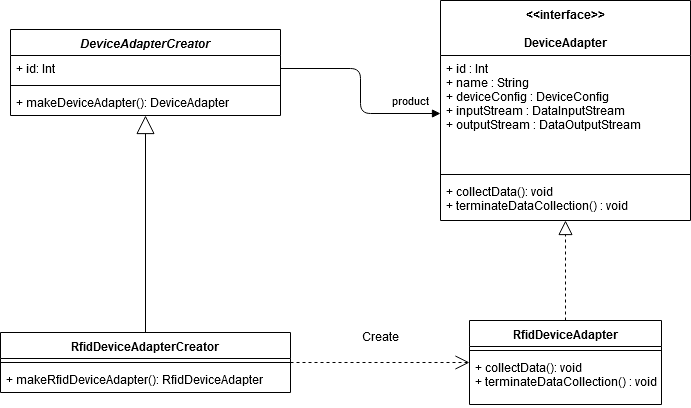


A device adapter manages communication with one external device. According to configuration information, it changes device settings, starts and stops data collection, and processes an incoming data stream from the device.

The device adapter creator is an abstract class. Concrete device adapter creators will inherit from the device adapter creator. Devices are created via the network manager (see CDX), so the device adapter creator is a subcomponent of the network manager component.

Different device types will require different device adapters. Thus, the device adapter will be implemented as an interface, and specific implementations will be created via a factory method design pattern.

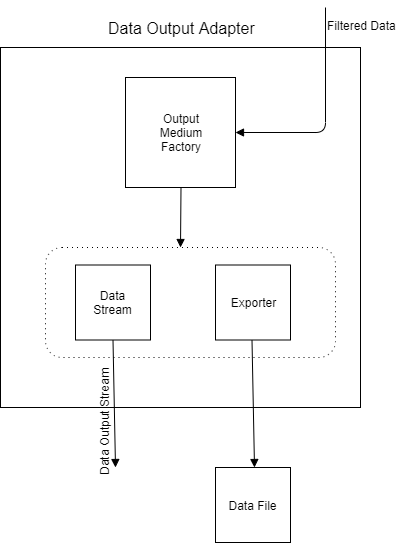
More detailed information about these components can be found in DE14.



**CD8 - Hardware Device**

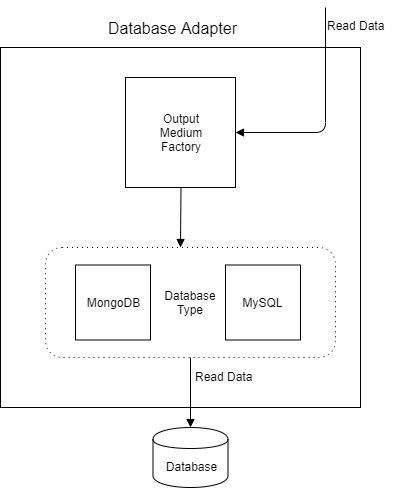
The reason the hardware device abstraction is outside the IoT sensor framework is that not all IoT devices are immediately supported by Chariot. There are two reasons why out of the box compatibility is an issue: 1) driver code needs to be added 2) device configuration varies too much to have a general approach. For Chariot to read data from that device, the adapter must work with the device.

**CD9 – Data Output Adapter**



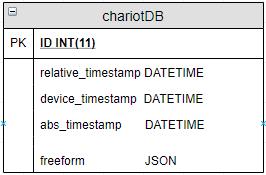
The data output adapter component has the primary purpose of sending device data out in live time or exporting data to a supported file type after data collection has ended. This component makes data extraction easy by using the factory design pattern. That is, the Output Medium Factory acts to be easily extensible so that another file type can be added to the export mediums by inheriting from the factory. Currently, Chariot supports exporting data to a .csv or json file. Apart from exporting data to a file, Chariot also supports live time data output. Again, this is done by making use of the factory. A client will connect to a port that Chariot uses for data output and will then be able to receive data in live time. More detailed information about this component can be found in DE16.

**CD10 – Database Adapter**



Chariot allows the user to store read data to a database of their choosing. In this component, the adapter works to make interfacing with multiple database instances possible. Whether a user wants to store data to MySql database, MongoDB database, or another supported database, this module makes it possible. Chariot writes data to a database periodically during the data collection period. This adapter does not alter the structure or content of data received from the data collection framework – it only acts to ensure that the right commands are used to write to a specific database. By using the factory design pattern (seen in OutputMediumFactory), adding implementations for multiple database types will possible and easy. See DE15 for more details on the DatabaseAdapter - and the DatabaseAdapterFactory for more details.

**CD11 – Database**



The database for this implementation will be based on the structure on implementation of Dr. Mongan’s IoT Sensor Framework. All collected data will be saved to a single table for fast storage. The table will have five fields; ID, relative\_timestamp, device\_timestamp, and abs\_timestamp. All devices can reliably provide an identification number, “id”, to serve as the primary key, and three timestamps, the “device\_timestamp” to mark when the device collected the data and sent it to Chariot, an abs\_timestamp to mark when the system saved the received data, and a relative\_timestamp to mark the difference between the receiving and final storage of data. Due to the wide assortment of data a single sensor can collect, and how there is no way to ensure that even two sensors the test for the same environmental factors will send the same kinds of data we will make use of a “freeform” field that contains an entire JSON object containing these data inputs.

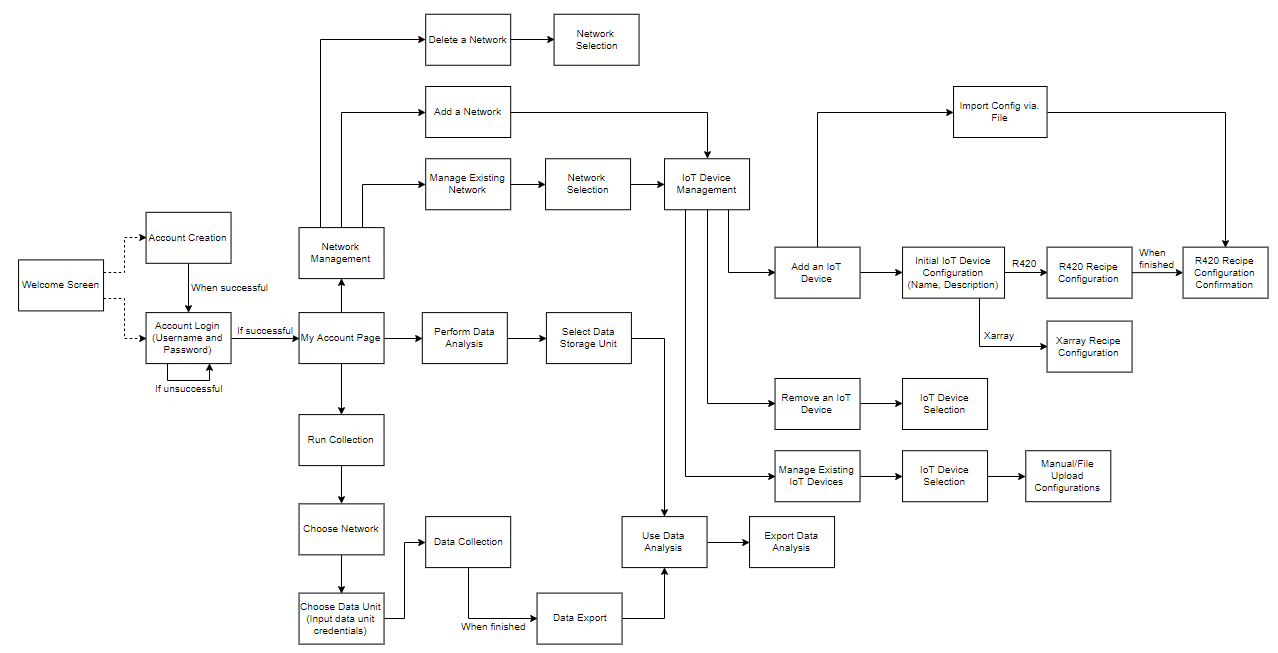
# Interface Description

## User Interface

This section aims to provide a high-level overview of the user interface design. As mentioned before, the UI will be developed in HTML, CSS, and JavaScript using the Electron framework. The UI can be separated into two components, the installer and the data collection framework UI. Each component was described at a high-level and can be referred to individually in CD2 and CD3, respectively. A more detailed explanation of both components can be found in DE1 and DE2, respectively.

The first UI element that the user will see is when they run the executable installer that takes them through the process of installing the software. After the installation is complete, the system will be installed onto the user’s device, and they can run it from there by clicking on the shortcut named, “Chariot”.

After launching the system, the user will be presented with the “Welcome Screen,” which is shown in Figure 3.1. From there, the user can choose to either create an account or login to an existing one. Figure 3.1 shows the rest of the flow throughout the User Interface. In Figure 3.1, screens, which are represent by rectangles, with multiple arrows coming out of them denote a choice the user must make on the screen.

**Figure 3.1**: Screen flow throughout Chariot

## Data Interface

This section aims to define the data transactions used throughout the system.

-Received Data – Data stream from an IoT device to be stored by Chariot.

* Collected Data – Data from an IoT device that has been stored into a data storage unit as a record
* IoT Device Configuration – configuration used by the IoT adapter to define how an IoT device collects data
* Network Configuration – a configuration of a network including its name, description, and an IoT Device list
* IoT Device list – a collection of IoT devices part of a network, includes the devices associated configuration, name, and description
* Network List – a list of networks saved by Chariot and their associated configurations

|  |  |  |
| --- | --- | --- |
| **Name** | **Type** | **Description** |
| Device data stream | Stream | Incoming data from a device connected to Chariot |
| Device configuration | Config file | Configuration settings for a device connected to Chariot. Includes instructions for parsing device data stream. |
| Device list | List | List of devices on a network. |
| Network configuration | Config file | Configuration settings for a Chariot network. Defines device configurations and data collection episode configurations available on the network. |
| Collected data | Data storage unit | During a data collection episode, Chariot parses device data streams and imports into a data storage unit. |
| Exported data | File | User may export collected data into a file. |
| Imported data | Data storage unit | User may import data in same format as exported data into a Chariot data storage unit. |
| Data collection episode configuration | Config file | Configuration settings for a data collection episode. Defines devices and device configurations to be used, storage unit to be used, and duration of data collection episode. |

## Programming Interface

This section aims to define the APIs used in the system. The following are used in the current implementation to allow for IoT devices to interact with the software.

* Low Level Reader Protocol (LLRP) - a software module used for fine control over read operations from RFID devices
* Impinj – RFID device driver
* Impinj Speedway R420 – RFID device driver
* Impinj Speedway R1000 – RFID device driver
* MongoDB Adapter – MongoDB database driver
* MySQL Adapter – MySQL database driver

This system does require an API for the purpose of integrating heterogeneous sensors that vary with each supported device. This is known as driver code and one of the examples is the Imping package. Furthermore, a requirement of this project is to be able to live stream data to clients.

# Detailed Design

## DE1 – Installation

**Type**: Screen Sequence

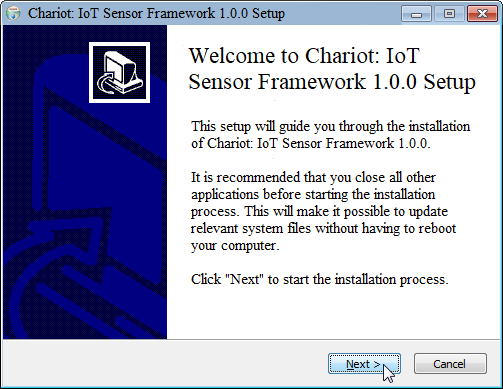
**Description**:

The name of this sequence of screens is “Chariot: IoT Sensor Framework x.x.x Setup,” where ‘x.x.x’ is a placeholder for the current version of the system in production. This screen can be seen after the end-user launches the executable file that will run the installation process of the system on their computer. The initial screen that the user will see in this process is shown in Figure 4.1. At the very least, it introduces the user to the installation process and tells them what they need to do before proceeding with the installation. Clicking on the “Install” button should lead to the screen displayed in Figure 4.2. Clicking “Cancel” just exits the installer.

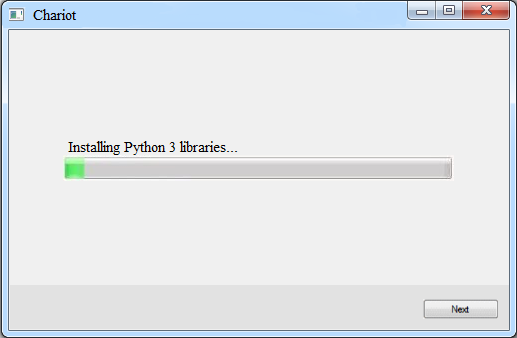
The screen shown in Figure 4.2 is a still-shot of the installation progress of the system onto the user’s device. There is nothing to interact with on this screen. The user must wait for the installation to finish before being able to click “Next.” A progress bar on the screen lets the user know where the installation currently is. Once the progress bar is filled, the status message will let the user know that they can proceed and click “Next” to go to the screen shown in Figure 4.3.

Figure 4.3 shows what the user will see once the installation is finished. Text letting the user know the installation has finished should be on the screen. The user can either click “Launch” and the actual program will be launched or click on “Cancel” to close the installation window.

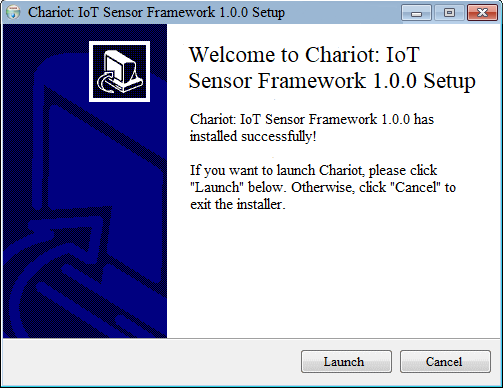
A high-level diagram of the flow in this sequence of screens can be seen in Figure 4.3.



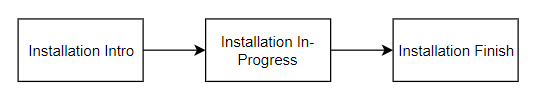
**Figure 4.1**: Chariot’s initial installation screen



**Figure 4.2**: Chariot’s installation-in-progress screen



**Figure 4.3**: Chariot’s successful installation screen



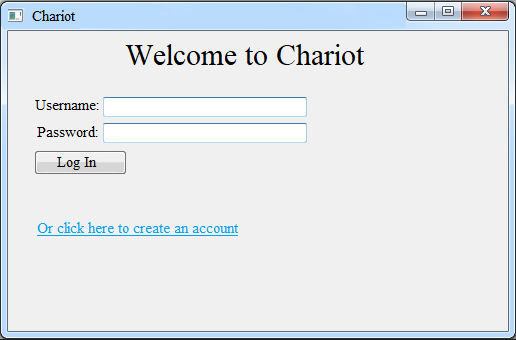
**Figure 4.4**: Chariot’s installation screen flow chart

## DE2 – Welcome Screen

**Type**: Screen

**Description**:

This is the first page that the user will see when they launch Chariot. This page will prompt the user to login to their account or create a new account. A successful login will lead the user to the screen in Figure 4.9.



**Figure 4.5**: Chariot’s welcome screen

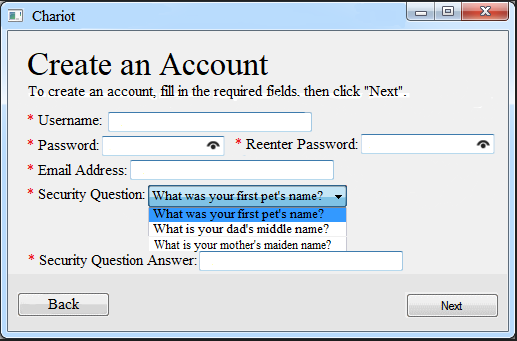
## DE3 – Account Creation Screens

**Type**: Screen Sequence

**Description**:

This sequence of screens allows the user to create an account with Chariot. Fields for a username, password, security question and answer, and email address are required. Without filling out the required fields, the user cannot move forward in the process. After filling out the fields in Figure 4.6, the user will then confirm their input in Figure 4.7. Once confirmed, the account will be created, and they can login with their newly set credentials. Clicking “Next” in Figure 4.8 takes the user back to the welcome screen in Figure 4.5 so they can log in.

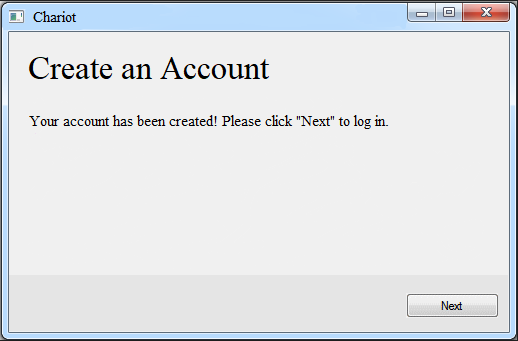
Usernames and email addresses must be unique, otherwise, an error message shall appear to let the user know that they most use a different username and/or email address.



**Figure 4.6**: Chariot’s account creation screen



**Figure 4.7**: Account credential confirmation screen

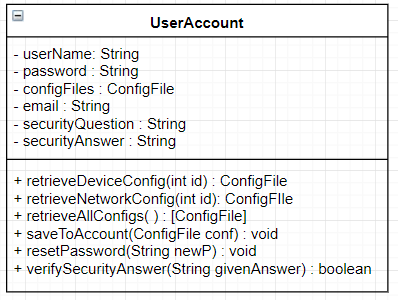


**Figure 4.8:** Account creation success screen

## DE3.1 – Account Creation Class

**Type**: Code Entity

**Description**:



The UserAccount class is meant to represent a user – via a username and password - and can store configuration files that the user specifies.

|  |  |  |
| --- | --- | --- |
| **Attributes** | **Type** | **Description** |
| userName | String | Unique username used for identification |
| password | String | Attribute needed for authentication |
| configFiles | ConfigFile | Entity used to abstract different types of configuration files such as device configuration and network configuration. |
| email | String | Attribute used to store email linked with account |
| securityQuestion | String | Attribute used to store account’s security question |
| securityAnswer | String | Attribute used to store answer to security question |

|  |  |
| --- | --- |
| **Method** | saveToAccount |
| **Input:** | ConfigFile |
| **Output:** | void |
| **Description:** | Saves a configuration file to this account |

|  |  |
| --- | --- |
| **Method** | resetPassword |
| **Input:** | String newP |
| **Output:** | void |
| **Description:** | When the user decides to change their password, use this method. |

|  |  |
| --- | --- |
| **Method** | verifySecurityAnswer |
| **Input:** | String givenAnswer |
| **Output:** | boolean |
| **Description:** | When a user forgets their password, verify that their answer to the security question matches what they said when they created their account |

## DE4 – My Account Screen

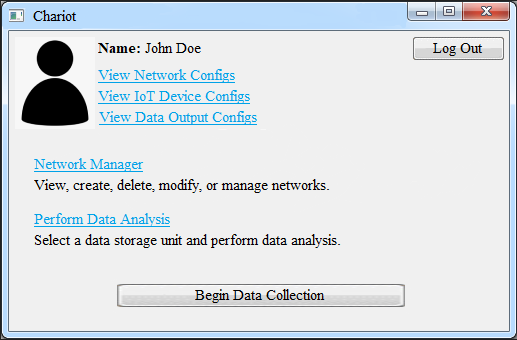
**Type**: Screen

**Description**:

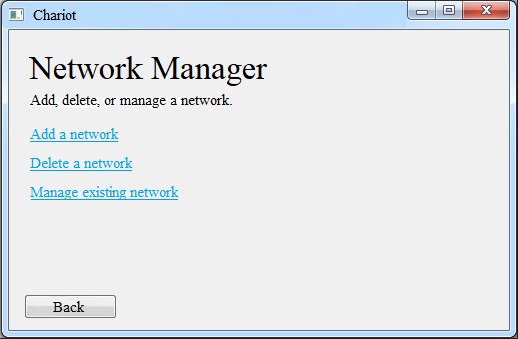
This screen appears after the user has successfully logged into their account. It shows the user their account details, as well as provides them with options on what they want to do next. They can either view their config files, manage their network and its corresponding IoT devices, perform data analysis, or run a data collection episode.

The screen displayed in Figure 4.10 is what the user will see when they choose the “Network Manager” option. From here, the user can either add, delete, or manage a network. Specifics for each of these screens and functionality are explained in more detail in DE5, DE6, and DE7, respectively.

The “Perform Data Analysis” and “Begin Data Collection” screens are still works-in-progress because the requirements for each given from the stakeholders are still unclear. Further details of these screens will be included in future versions of this document as their requirements from the stakeholders become clearer.



**Figure 4.9:** The user’s account page



**Figure 4.10:** Network Manager screen, which appears when the user clicks on “Network Manager” on the Account Page (Figure 4.9)

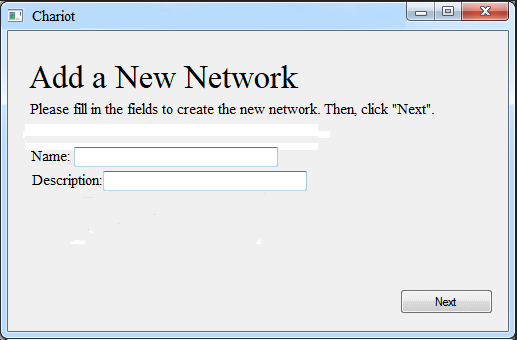
## DE5 – Create Network Screens

**Type**: Screen Sequence

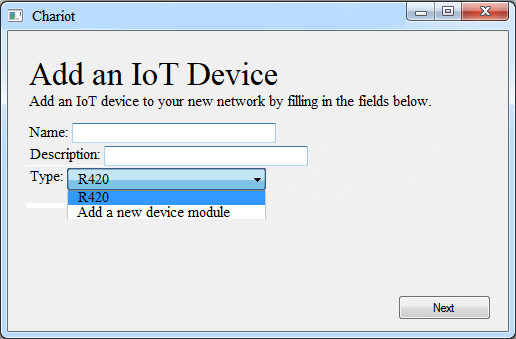
**Description**:

This collection of screens is available to the user when they choose to create a new network. This sequence of screens will take the user through the process of creating a new network to their list of networks.

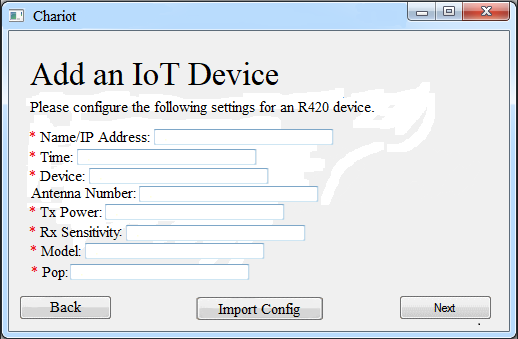
After clicking “Next” in Figure 4.11, the user will be able to add an IoT device to the network in the next screen. The screen depicting how a user would add a device is shown in Figure 4.12 and Figure 4.13. After adding the IoT device, the network will have been created and the user will go back to their Account page (Figure 4.9) with a notification that the network was created successfully.



**Figure 4.11:** Screen used to add a new network to the user’s account



**Figure 4.12:** First screen used to add an IoT device to a network



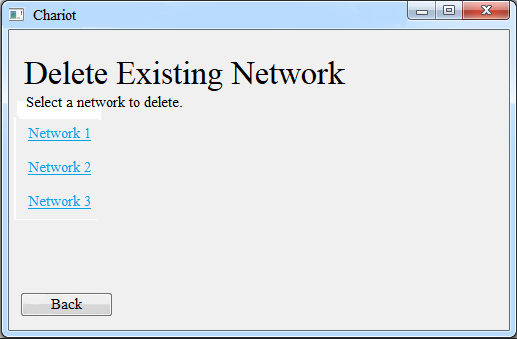
**Figure 4.13:** Sample screen allowing for configuration of IoT device being added to a network

## DE6 – Delete Network Screens

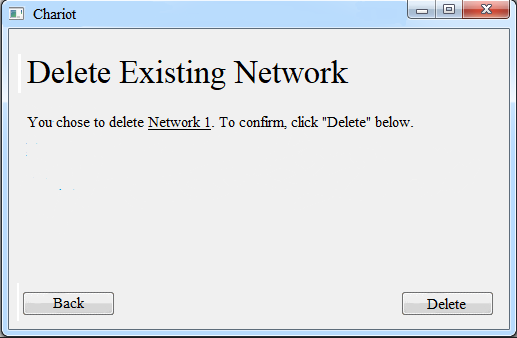
**Type**: Screen Sequence

**Description**:

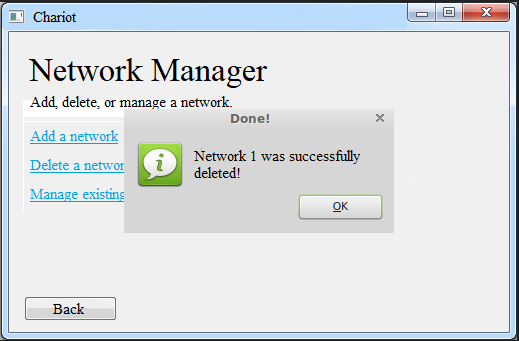
This collection of screens is available to the user when they choose to delete an existing network. The screens will guide the user through the process of deleting a network.



**Figure 4.14:** Screen used to delete an existing network



**Figure 4.15:** Confirmation page for deleting a network



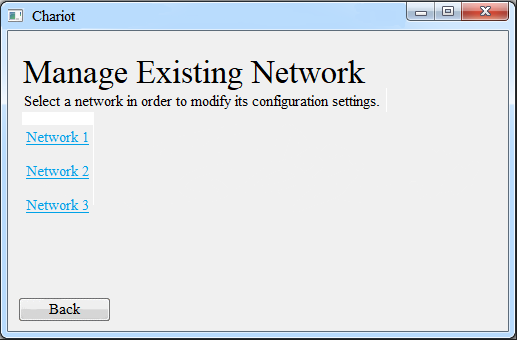
**Figure 4.16:** Notification letting user know that chosen network was successfully deleted

## DE7 – Manage Existing Networks Screens

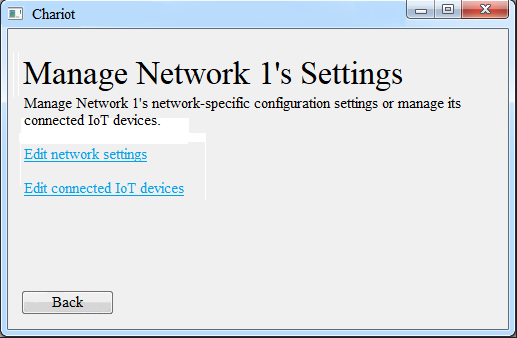
**Type**: Screen Sequence

**Description**:

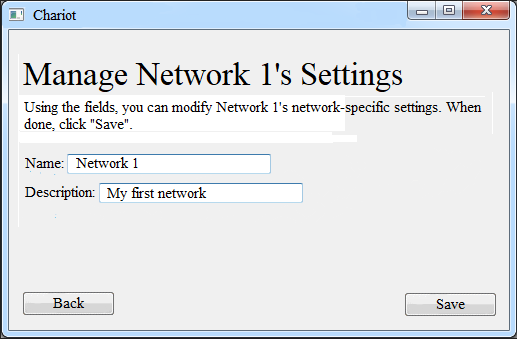
This collection of screens is available to the user when they choose to manage existing networks. These set of screens will allow the user to manage network specific settings such as name, description, connected IoT devices, etc.



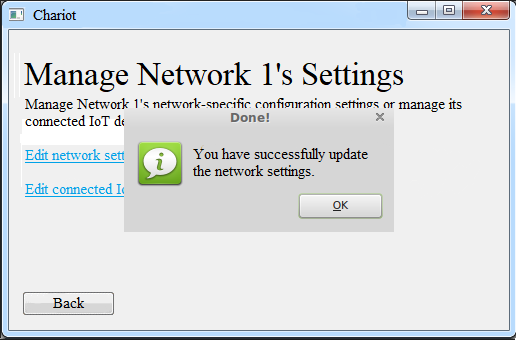
**Figure 4.17:** Screen where user chooses a network to modify its settings



**Figure 4.18:** Screen after user chooses a network to modify, and is now presented with a choice to select what to modify about the network’s settings



**Figure 4.19:** Modify network-specific settings screen



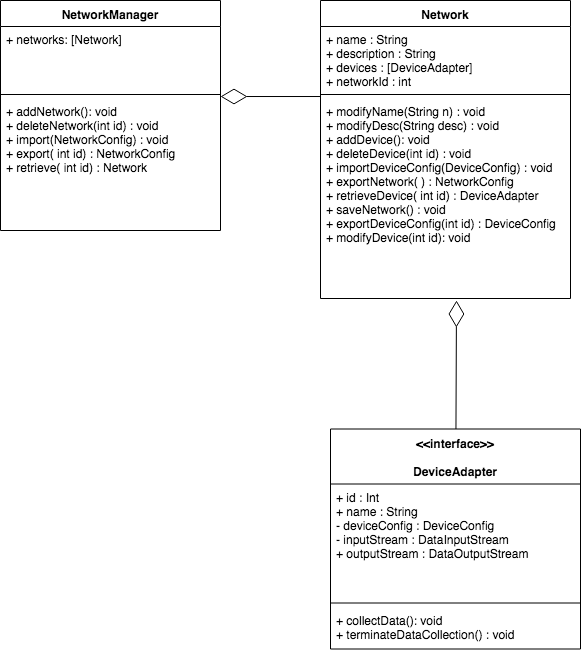
**Figure 4.20:** Successful modification of network-specific settings



**Figure 4.21:** Modifying network’s IoT devices screen

## DE7.1 – Manage Existing Networks

**Type**: Code Entity



**Description**:

The NetworkManager and Network classes act to make managing networks, adding networks, and deleting networks possible. The concept of a network is an array of devices that a network knows about. A network should be able to add, delete, or modify a device on its collection.

**Network Manager**

**Type:** Code Entity

**Description:**

The manager is able to handle storing multiple networks and managing them by deleting, modifying, or adding to its collection.

|  |  |
| --- | --- |
| **Method** | import |
| **Input:** | NetworkConfig |
| **Output:** | void |
| **Description:** | This method acts to create a network object based on the configuration file |

|  |  |
| --- | --- |
| **Method** | export |
| **Input:** | void |
| **Output:** | NetworkConfig |
| **Description:** | This method acts to create a NetworkConfig file that the user has chosen |

|  |  |
| --- | --- |
| **Method** | retrieve |
| **Input:** | int id |
| **Output:** | Network |
| **Description:** | Gets Network by id. |

## DE7.2 – Network

**Type**: Code Entity

**Description:**

Stores DeviceAdapters that a user has added to a network. Allows for import and export of their configuration at the network level.

|  |  |
| --- | --- |
| **Method** | exportDeviceConfig |
| **Input:** | DeviceConfig |
| **Output:** | void |
| **Description:** | This method acts to create a DeviceConfig file of a device that is currently on this network |

|  |  |
| --- | --- |
| **Method** | exportNetworkConfig |
| **Input:** | void |
| **Output:** | NetworkConfig |
| **Description:** | This method acts to create a NetworkConfig file that can be used by the NetworkManager to create a network according to the configuration file |

|  |  |
| --- | --- |
| **Method** | importDeviceConfig |
| **Input:** | DeviceConfig |
| **Output:** | void |
| **Description:** | This method acts to add a DeviceAdapter to the network via importing of a configuration file |

|  |  |
| --- | --- |
| **Method** | saveNetwork |
| **Input:** | void |
| **Output:** | void |
| **Description:** | This method acts to save the currently saved network configuration to the user account list of configurations. |

## DE7.3 – Network Configuration

**Type**: Code Entity

**Description:**

Stores network configuration information in a JsonObject.

|  |  |  |
| --- | --- | --- |
| Attributes | Type | Description |
| id | Int | Identification number |
| name | String | User defined name |
| config | JsonObject | Stores network configuration info as JSON |

|  |  |
| --- | --- |
| Method | exportToFile |
| Input: | String path |
| Output: | void |
| Description: | Saves configuration info as JSON file. |

## DE8 – Add IoT Device Screens

**Type**: Screen Sequence

**Description**:

This collection of screens is available to the user when they choose to add a new IoT device to an existing network, or when they’re creating a new network. To get here, the user must first choose *Manage Existing Network* and then *Add an IoT Device,* or *Add a Network*, respectively. These screens were previously shown in Figure 4.12 and 4.13.

## DE9 – Delete IoT Device Screens

**Type**: Screen Sequence

**Description**:

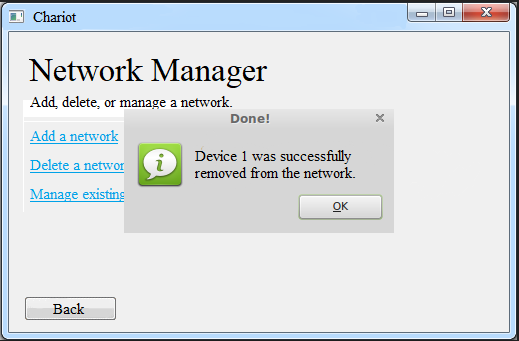
This collection of screens is available to the user when they choose to delete an existing IoT device from an existing network. Like the *Add an IoT Device* screen, the user must first choose to *Manage Existing Network* to see this collection of screens. These screens will let the user delete an IoT device from an existing network.



**Figure 4.22:** Screen where user chooses an IoT device to remove from the network



**Figure 4.23:** User confirms the device they intend to delete on this screen



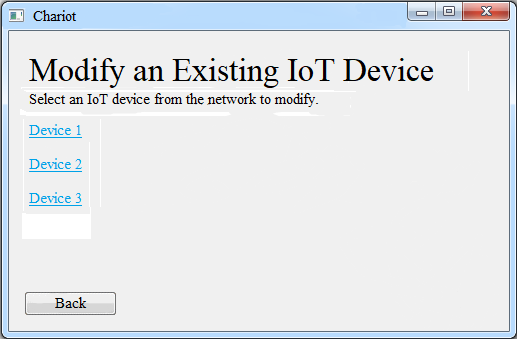
**Figure 4.24:** Successful deletion of network’s connected IoT device

## DE10 – Manage Existing IoT Devices Screens

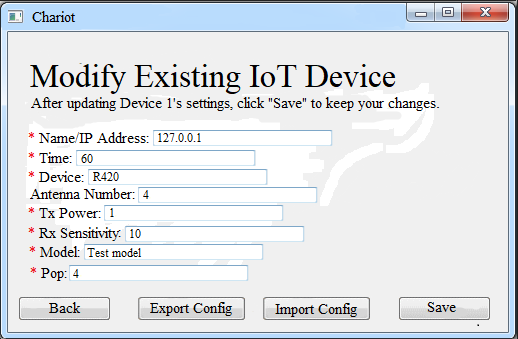
**Type**: Screen Sequence

**Description**:

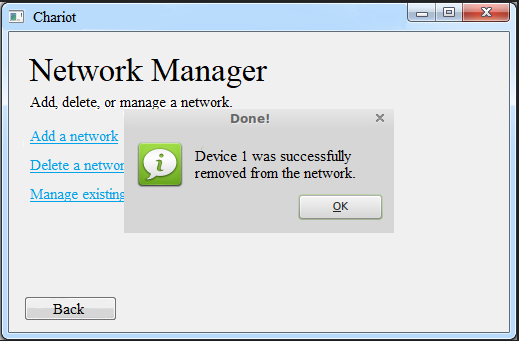
This collection of screens is available to the user when they choose to manage existing IoT devices on an existing network. These screens allow the user to modify IoT device-specific settings. From here, the user can modify device specific configuration settings via manual input or configuration file import or export the device’s current configuration settings to a file.



**Figure 4.25:** Screen where user chooses an IoT device to modify



**Figure 4.26:** Modifying an existing IoT device screen



**Figure 4.27:** Notification after an IoT device’s configuration settings were successfully modified

## DE11 – Data Collection Screens

**Type**: Screen Sequence

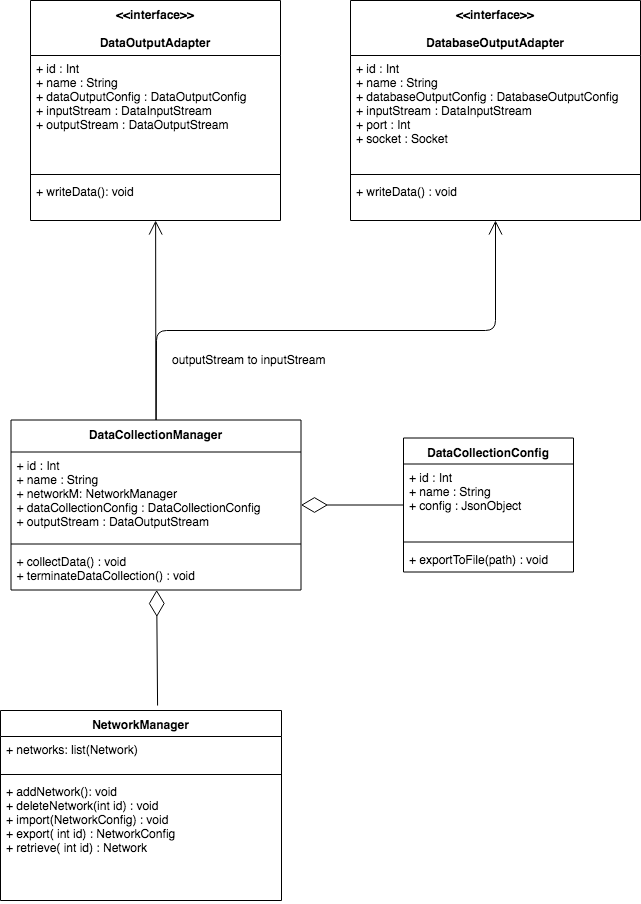
**Description**:

Before data is collected, the user must first choose the network from which the data is coming from. After that, they must choose the data unit that is going to be used to store the data. Then, the button to start the data collection is made available to click.

While collecting data, the user has the option to enable the real-time data visualizer. They also have a button that will allow them to stop the data collection episode. When the user chooses to terminate the data collection episode, they will be presented with the option to export the collected data and/or perform data analysis.

## DE12 – Data Collection Manger

The data collection manager handles data collection from a network. It translates all of the output streams from devices on the network to one unified output stream.



## DE12.1 – Data Collection Manager

**Type**: Code Entity

**Description:**

Merges data streams from devices on a network into one collection and then sends to appropriate place based on user input.

|  |  |  |
| --- | --- | --- |
| **Attributes** | **Type** | **Description** |
| id | Int | Identification number |
| name | String | User specified name |
| network | Network | Network to collect data from |
| dataCollectionConfig | DataCollectionConfig | Configuration object, see below |
| outputStream | DataOutputStream | Outgoing data stream |

|  |  |
| --- | --- |
| **Method** | collectData |
| **Input:** | void |
| **Output:** | void |
| **Description:** | Initializes data collection from all devices on network. |

|  |  |
| --- | --- |
| **Method** | terminateDataCollection |
| **Input:** | void |
| **Output:** | void |
| **Description:** | Terminates data collection on all devices on network. |

|  |  |
| --- | --- |
| **Method** | toOutputMediums |
| **Input:** | void |
| **Output:** | void |
| **Description:** | Sends read data from hardware to the data output adapter |

|  |  |
| --- | --- |
| **Method** | toDatabase |
| **Input:** | void |
| **Output:** | void |
| **Description:** | Sends read data from hardware to the database adapter periodically |

## DE12.2 – Data Collection Configuration

**Type**: Code Entity

**Description:**

Stores data collection configuration information in a JsonObject.

|  |  |  |
| --- | --- | --- |
| **Attributes** | **Type** | **Description** |
| id | Int | Identification number |
| name | String | User defined name |
| config | JsonObject | Stores data collection configuration info as JSON |

|  |  |
| --- | --- |
| **Method** | exportToFile |
| **Input:** | String path |
| **Output:** | void |
| **Description:** | Saves configuration info as JSON file. |

## DE13 – Data Analysis Screens

**Type**: Screen Sequence

**Description**:

This collection of screens allows the user to interact and begin the refining of raw data into information based on the data analysis module. This series of screens have to have their requirements defined by the stakeholders before a visual representation for them can be created.

## DE14 – Device Adapter and Device Adapter Creator

A close up of a map

Description automatically generated

## DE14.1 – Device Adapter Creator

**Type**: Code Entity

**Description:**

Creates Device Adapter objects. Concrete classes that derive from this class will create adapters to a specific kind of device. For example, an RFID Device Adapter Creator will create RFID Device Adapter objects, which implement the Device Adapter interface. Follows Factory Method design pattern.

|  |  |  |
| --- | --- | --- |
| **Attributes** | **Type** | **Description** |
| id | Int | Identification number |
| deviceConfig | DeviceConfig | Packages a JSON object which contains configuration info for a specific kind of device |

|  |  |
| --- | --- |
| **Method** | makeDeviceAdapter |
| **Input:** | void |
| **Output:** | DeviceAdapter |
| **Description:** | Abstract implementation: creates a new DeviceAdapter.  Concrete implementations will actually interface with the device. |

## DE14.2 – Device Adapter

**Type**: Interface

**Description:**

A device adapter manages communication with one external hardware device. According to configuration information, it changes device settings, starts and stops data collection, and processes an incoming data stream from the device.

The device adapter creator is an abstract class. Concrete device adapter creators will inherit from the device adapter creator.

Different device types will require different device adapters. Thus, the device adapter will be implemented as an interface, and specific implementations will be created via a factory method design pattern.

|  |  |  |
| --- | --- | --- |
| **Attributes** | **Type** | **Description** |
| id | Int | Identification number |
| name | String | User may define name to make device recognizable |
| deviceConfig | DeviceConfig | A DeviceConfig object |
| inputStream | DataInputStream | An incoming stream |
| outputStream | DataOutputStream | Device sends processed data out here |

|  |  |
| --- | --- |
| **Method** | collectData |
| **Input:** | void |
| **Output:** | void |
| **Description:** | Reads input stream, processes it, and sends to output stream according to rules established in deviceConfig. |

|  |  |
| --- | --- |
| **Method** | terminateDataCollection |
| **Input:** | void |
| **Output:** | void |
| **Description:** | If device is collecting data, immediately terminates. |

## DE14.3 – Device Config

**Type**: Code Entity

**Description:**

Packages a JSON object which contains configuration info for a specific kind of device.

|  |  |  |
| --- | --- | --- |
| **Attributes** | **Type** | **Description** |
| id | Int | Identification number |
| name | String | User defined name |
| config | JsonObject | Device configuration settings packaged in JSON object |

|  |  |
| --- | --- |
| **Method** | exportToFile |
| **Input:** | String path |
| **Output:** | void |
| **Description:** | Saves configuration information to a JSON file. |

## DE15 – Database Output Adapter

A database output adapter translates an incoming data stream from the Data Collection Manager to a database. Will be implemented using factory method design pattern.

A close up of a map

Description automatically generated

## DE15.1 – Database Output Adapter

**Type**: Code Entity

**Description:**

Outputs data to a database.

|  |  |  |
| --- | --- | --- |
| Attributes | Type | Description |
| id | Int | Identification number |
| name | String | User specified name |
| databaseOutputConfig | DatabaseOutputConfig | Configuration object, see below |
| inputStream | DataInputStream | Incoming data stream, output of Data Collection Manager |
| port | Int | Port of database server |
| socket | Socket | Socket of database server |

|  |  |
| --- | --- |
| Method | writeData |
| Input: | void |
| Output: | void |
| Description: | Writes inputStream to database. Implemented differently depending on type of database. |

## DE15.2 – Database Output Adapter Creator

**Type**: Code Entity

**Description:**

Creates DatabaseOutputAdapter objects. Concrete classes will create implementations of the DatabaseOutputAdapter interface.

|  |  |  |
| --- | --- | --- |
| Attributes | Type | Description |
| id | Int | Identification number |
| databaseOutputConfig | DatabaseOutputConfig | Configuration object, see below |

|  |  |
| --- | --- |
| Method | makeDatabaseOutputAdapter |
| Input: | void |
| Output: | DatabaseOutputAdapter |
| Description: | Creates new DatabaseOutputAdapter object. |

## DE15.3 – Database Output Adapter Config

**Type**: Code Entity

**Description:**

Contains configuration information for a DatabaseOutputAdapter. Configuration information stored as JsonObject.

|  |  |  |
| --- | --- | --- |
| Attributes | Type | Description |
| id | int | Identification number |
| name | String | User specified name |
| config | JsonObject | Contains key-value pairs specifying configuration information for a DatabaseOutputAdapter |

|  |  |
| --- | --- |
| Method | exportToFile |
| Input: | String path |
| Output: | void |
| Description: | Saves configuration information to a JSON file. |

## DE16 – Data Output Adapter

A data output adapter translates an incoming data stream from the Data Collection Manager into a specific type of output file or stream. The Data Output Adapter will be implemented as an interface using the factory method design pattern.

A screenshot of a map

Description automatically generated

## DE16.1 – Data Output Adapter

**Type**: Code Entity

**Description:**

Outputs data as a specific file type or stream.

|  |  |  |
| --- | --- | --- |
| Attributes | Type | Description |
| id | Int | Identification number |
| name | String | User specified name |
| dataOutputConfig | DataOutputConfig | Configuration object, see below |
| inputStream | DataInputStream | Incoming data stream, output of Data Collection Manager |
| outputStream | DataOutputStream | Outgoing data stream |

|  |  |
| --- | --- |
| Method | writeData |
| Input: | void |
| Output: | void |
| Description: | Writes inputStream to outputStream. Implemented differently depending on type of data output. |

## DE16.2 – Data Output Adapter Creator

**Type**: Code Entity

**Description:**

Creates DataOutputAdapter objects. Concrete classes will create implementations of the DataOutputAdapter interface.

|  |  |  |
| --- | --- | --- |
| Attributes | Type | Description |
| id | Int | Identification number |
| dataOutputConfig | DataOutputConfig | Configuration object, see below |

|  |  |
| --- | --- |
| Method | makeDataOutputAdapter |
| Input: | void |
| Output: | DataOutputAdapter |
| Description: | Creates new DataOutputAdapter object. |

## DE16.3 – Data Output Config

**Type**: Class

**Description:**

Contains configuration information for a DataOutputAdapter. Configuration information stored as JsonObject.

|  |  |  |
| --- | --- | --- |
| Attributes | Type | Description |
| id | int | Identification number |
| name | String | User specified name |
| config | JsonObject | Contains key-value pairs specifying configuration information for a DataOutputAdapter |

|  |  |
| --- | --- |
| Method | exportToFile |
| Input: | String path |
| Output: | void |
| Description: | Saves configuration information to a JSON file. |

**Appendix**

A) System Class Diagram

## 

Figure 4.28 System Class Diagram

**References**

<https://platform.impinj.com/site/developer/itemsense/apidocs/index.gsp>

<https://github.com/drexelwireless/iot-sensor-framework/blob/master/README.md>