Software Design Specification

For

Chariot

Submitted by

Chariot Dev

|  |  |
| --- | --- |
| **Instructor:** Dr. Gregory Hislop |  |
| **Team Members:** Medard Azandegbe, Kamalludin Colaire, Ryan Hassing, John Ho, Juan Garcia Lopez, Christopher Mak, Enioluwa Segun |  |
| **Cycle:** 2 |  |
| **Date Submitted:** 2/16/2020 |  |

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 5](#_Toc32784946)

[1.2 Definitions, Acronyms, and Abbreviations 5](#_Toc32784947)

[1.3 Requirements Traceability Matrix 5](#_Toc32784948)

[2 Architectural Description 9](#_Toc32784949)

[3 Interface Description 16](#_Toc32784950)

[3.1 User Interface 16](#_Toc32784951)

[3.2 Data Interface 17](#_Toc32784952)

[3.2.1 Data flow diagram 17](#_Toc32784953)

[3.2.2 Raw data stream 17](#_Toc32784954)

[3.2.3 Device data stream 17](#_Toc32784955)

[3.2.4 Network data stream 18](#_Toc32784956)

[3.2.5 Database Write 19](#_Toc32784957)

[3.3 Programming Interface 19](#_Toc32784958)

[4 Detailed Design 20](#_Toc32784959)

[DE1 – Installation 20](#_Toc32784960)

[DE2 – Login Screen 22](#_Toc32784961)

[DE3 – Account Creation Screens 22](#_Toc32784962)

[DE3.1 – User Account 24](#_Toc32784963)

[DE4 – Welcome Screen 25](#_Toc32784964)

[DE5 – Create Network Screens 26](#_Toc32784965)

[DE6 – Delete Network Screens 29](#_Toc32784966)

[DE7 – Manage Existing Networks Screens 30](#_Toc32784967)

[DE7.1 – Manage Existing Networks 33](#_Toc32784968)

[4.1 DE7.2 – Network 34](#_Toc32784969)

[DE8 – Add IoT Device Screens 35](#_Toc32784970)

[DE9 – Delete IoT Device Screens 35](#_Toc32784971)

[DE10 – Manage Existing IoT Devices Screens 37](#_Toc32784972)

[DE11 – Data Collection Screens 39](#_Toc32784973)

[DE12 – Data Collection Manger 39](#_Toc32784974)

[DE12.1 – Data Collection Manager 40](#_Toc32784975)

[DE13 – Data Analysis Screens 42](#_Toc32784976)

[DE14 – Device Adapter and Device Adapter Creator 43](#_Toc32784977)

[DE14.1 – Device Adapter Creator 43](#_Toc32784978)

[DE14.2 – Device Adapter 44](#_Toc32784979)

[DE14.3 – Device Config 44](#_Toc32784980)

[DE15.1 – MongoDatabaseWriter 46](#_Toc32784981)

[DE15.2 – MySQLDatabaseWriter 46](#_Toc32784982)

[DE15.3 – DatabaseWriterFactory 46](#_Toc32784983)

[DE15.4 – DatabaseWriterConfiguration 47](#_Toc32784984)

[DE16 – Data Output Adapter 48](#_Toc32784985)

[DE16.1 – Data Output Adapter 48](#_Toc32784986)

[DE16.2 – Data Output Adapter Creator 49](#_Toc32784987)

[DE16.3 – Data Output Config 49](#_Toc32784988)

[DE17 – API Endpoints 50](#_Toc32784989)

[DE17.1 – Getting a network name and its description 50](#_Toc32784990)

[DE17.2 – Retrieving all Networks and their devices 50](#_Toc32784991)

[DE17.3 – Creating a Network 50](#_Toc32784992)

[DE17.4 – Modifying a Network 51](#_Toc32784993)

[DE17.5 – Deleting a Network 51](#_Toc32784994)

[DE17.6 – Get a Network’s details 52](#_Toc32784995)

[DE17.7 – Get all the information about supported devices 52](#_Toc32784996)

[DE17.8 – Get all the information about a specific device 52](#_Toc32784997)

[DE17.9 – Get information about a configured device on a network 53](#_Toc32784998)

[DE17.10 – Create a device on a specified network 53](#_Toc32784999)

[DE17.11 – Modifying a device on a specified network 54](#_Toc32785000)

[DE17.12 – Deleting a device on a specified network 54](#_Toc32785001)

Appendix [56](#_Toc32785002)

Introduction

## Scope

This document describes the design and implementation details of Chariot, an IoT device framework. The framework is largely based on Dr. Bill Mongan’s IoT sensor framework, with several key improvements to allow for more general use cases. The design of Chariot is meant to satisfy the requirements outlined 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 (DCE) – 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. UI – User Interface. Manages interactions between the user and the system.

## 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 | P | P |  |  |  |  |  |  |
| **FR3** |  | P | P | P |  |  |  |  |  |  |
| **FR4** |  |  |  |  | P |  |  |  |  |  |
| **FR5** |  |  |  |  | S |  |  |  |  |  |
| **FR6** |  |  |  |  | P |  |  |  |  |  |
| **FR7** |  |  |  |  |  | P |  |  |  |  |
| **FR8** |  |  |  |  | S |  | P |  |  |  |
| **FR9** |  |  |  |  |  |  |  | S |  |  |
| **FR10** |  |  |  |  |  |  |  | S |  |  |
| **FR11** |  |  |  |  |  |  |  | P |  |  |
| **FR12** |  |  |  |  |  |  |  | P |  |  |
| **FR13** |  |  |  |  |  |  |  |  | P |  |
| **FR14** |  |  |  |  |  |  |  |  | P |  |
| **FR15** |  |  |  |  |  |  |  |  | P |  |
| **FR16** |  |  |  |  |  |  |  |  |  | S |
| **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** |  |  |  |  |  |  |
| **FR11** |  |  |  |  |  |  |
| **FR12** |  |  |  |  |  |  |
| **FR13** |  |  |  |  |  |  |
| **FR14** | S |  |  |  |  |  |
| **FR15** | S |  |  |  |  |  |
| **FR16** | P |  |  |  |  |  |
| **FR17** | P | P |  |  |  |  |
| **FR18** |  | P |  |  |  |  |
| **FR19** |  | S |  |  |  |  |
| **FR20** |  | P |  |  |  |  |
| **FR21** |  | S |  |  | P |  |
| **FR22** |  |  |  |  | P |  |
| **FR23** |  |  |  |  | P |  |
| **FR24** |  |  |  |  | P |  |
| **FR25** |  | S |  |  | S | P |
| **FR26** |  | S |  |  | S | P |
| **FR27** |  |  | P |  | S |  |

Figure 1 – Traceability between Requirements and Design Entities

# Architectural Description

This section aims to provide a high-level overview of the system architecture.

**CD1 – System Architecture**

A screenshot of a cell phone

Description automatically generated

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

Figure 2.1 presents an overview of Chariot. The core interfaces with IoT devices, controlling data collection, network management, and output to a database. The web server exposes API endpoints to interface with the core via HTTP requests. The GUI uses the web server to provide access to functions of the system core via an easy-to-use interface.

The core can be further decomposed. The internal data manager interfaces with an internal database to store and access data for use by other modules in the core. For example, it stores user information, database connection information, and device information. The core uses device adapters to interface with IoT devices. There is a one-to-one relation between an IoT device and its corresponding device adapter. The network manager is responsible for grouping device adapters into a network. The data collection manager controls data collection from a network. The database writer manages output of data from the data collection manager to an external database.

For a more detailed illustration of how the core functions, please see the core class diagram in the appendix.

**CD2 – Installer**

Chariot will be installed via an installer. To use Chariot, the user will first have to go through this installation process. See the installer Screen Sequence items in section 4 for more detail.

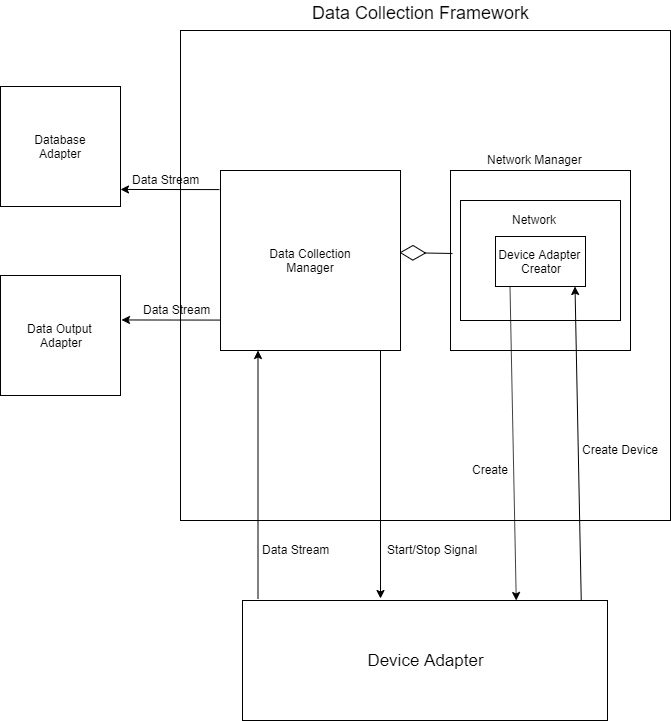
**CD3 – Data Collection Framework GUI**

This is the main component that the users 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. While navigating through the Design Interfaces pertaining to the GUI, please keep in mind that some mock-ups may be out-of-date and will be updated once they are implemented, more specifically, the device management screens. For now, they are included so that an idea of what the final product should resemble.

**CD4 – User Account**

The user account component is used by Chariot to enhance portability of the system. It allows the system to save user-specific data and settings beyond the local level and onto an account accessible by other machines so that a user does not everything when using a different machine. In particular, the user account is linked with various configurations such as those for networks and devices. How an account is created is described in DE2 and DE3.

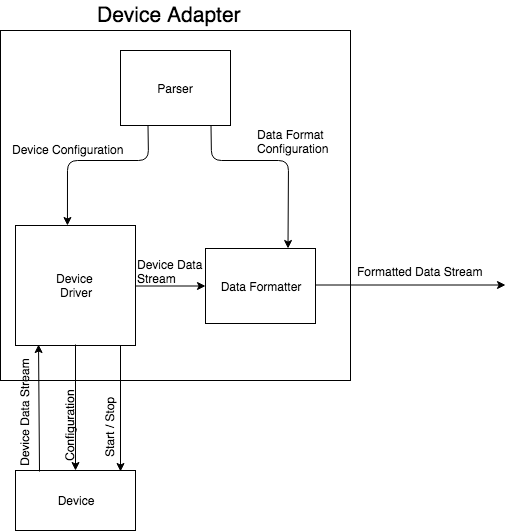
**CD5 – Data Collection Framework**



**Figure 2.2**: Data Collection Framework of Chariot.

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 (see **DE 5**) 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-specific 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**



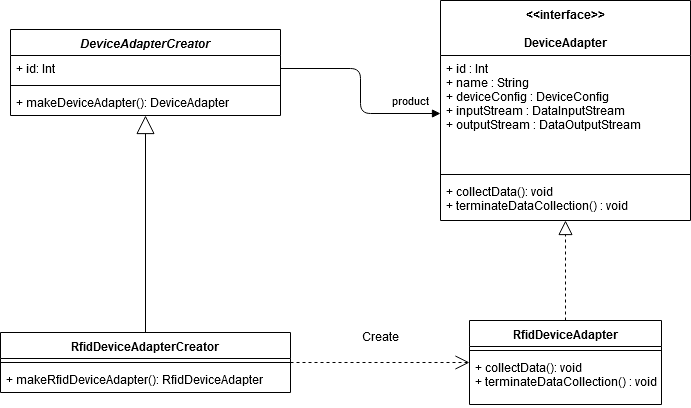
**Figure 2.3**: Device Adapter component of Chariot.

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.

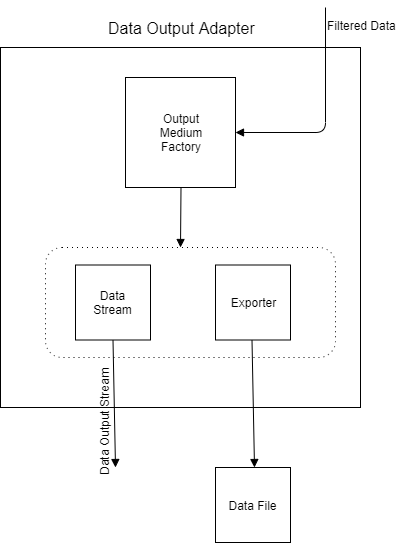
More detailed information about these components can be found in DE14.



**CD8 - Hardware Device**

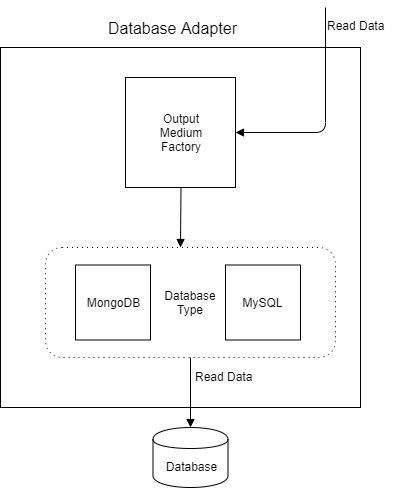
The reason the hardware device abstraction is outside the IoT sensor framework is that not all IoT devices are immediately supported by Chariot. There are two reasons why out of the box compatibility is an issue: 1) driver code needs to be added 2) device configuration varies too much to have a general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

**CD9 – Data Output Adapter**



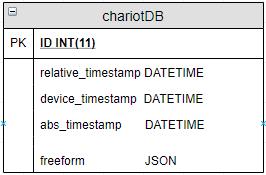
The data output adapter component has the primary purpose of sending device data out in live time or exporting data to a supported file type after data collection has ended. This component makes data extraction easy by using the factory design pattern. That is, the Output Medium Factory acts to be easily 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 Writer**



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 simplified. 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” (an incremental integer), 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 that 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.

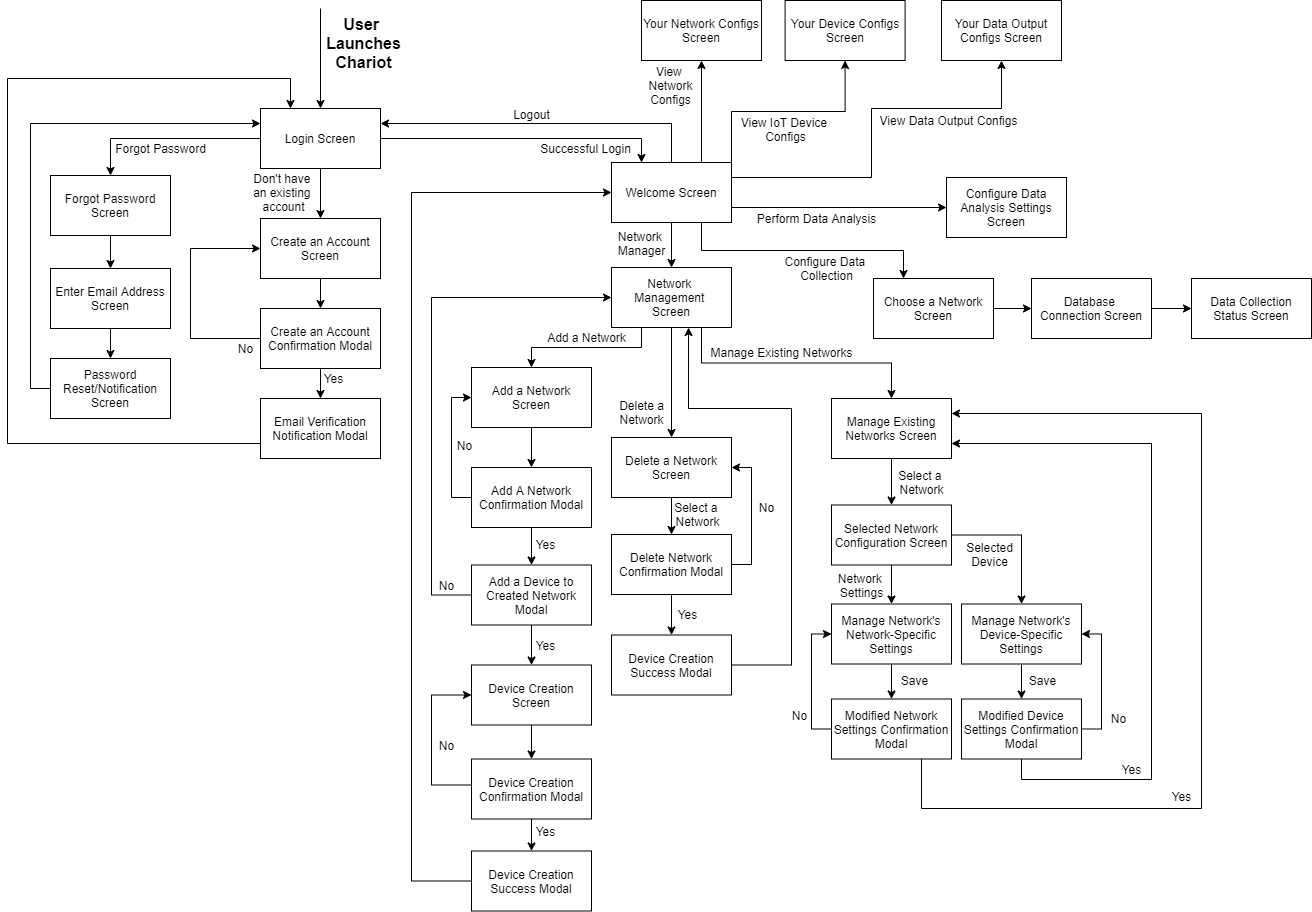
# Interface Description

## 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.



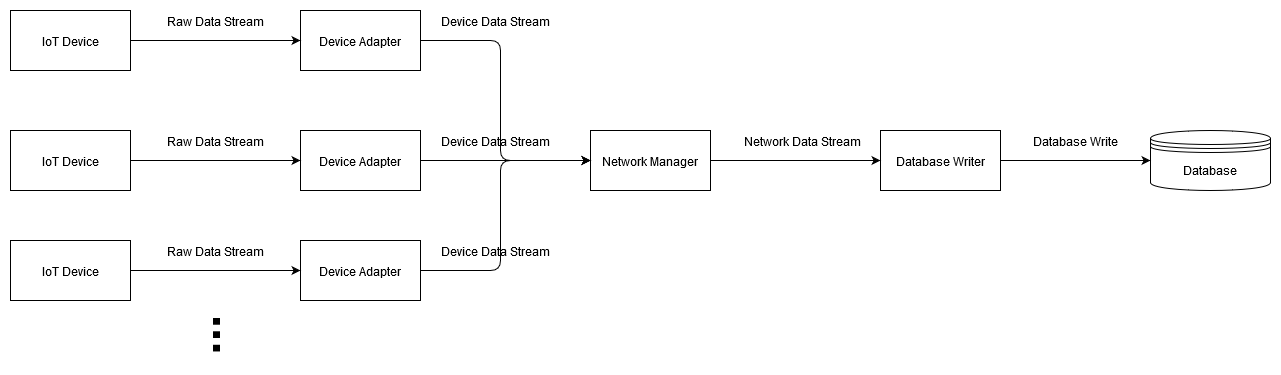
**Figure 3.1**: Screen flow throughout Chariot

## Data Interface

This section defines the data transactions used between Chariot components, and with external systems.

### Data flow diagram

Below is a data flow diagram illustrating the data transactions Chariot makes to collect and store IoT device data.



The data at each step is discussed below.

### Raw data stream

Type: Stream

Description:

A raw data stream is a data stream from an external IoT device that has not been processed by Chariot. Raw data streams from different devices may vary in appearance, but all raw data streams must include a timestamp for each data point. If not, the raw data stream is incompatible with Chariot.

### Device data stream

Type: Stream

Description:

A device data stream is a unified model for data from external IoT devices. For each IoT device, a device adapter is responsible for translating the raw data stream from the device to a device data stream. The data format of a device data stream is described below.

A single data point is represented by a JSON object with three fields: deviceID, time, and freeform. freeform is itself a JSON object containing key-value pairs with data labels and values. For example, a single data point from a temperature-humidity sensor would be represented as follows:

{

“deviceID”: “1”,

“time”: “14.05”,

“freeform”: {

“temp”: “24.5”,

“humidity”: “.34”

}

}

A data stream, then, is simply a sequence of data points with the same deviceID and ascending time values.

### Network data stream

Type: Stream

Description:

A network data stream associates data from multiple devices by time. Consider a network composed of the temperature-humidity sensor described above, as well as a pH sensor with the following representation:

{

“deviceID”: “2”,

“time”: “14.05”,

“freeform”: {

“pH”: “8.00”

}

}

Then the network data stream would associate these data points in the following format:

{

“time”: “14.05”,

“data”: {

“1”: {

“temp”: “24.5”,

“humidity”: “.34”

},

“2”: {

“pH”: “8.00”

}

}

}

A network data stream is a sequence of such objects with ascending time values.

### Database Write

Type: Struct

Description:

The database writer reads a network data stream and writes data points into a database. It will continue writing as long as the incoming network data stream is active.

## Programming Interface

Chariot will provide a programming interface via a web server module that exposes API endpoints. These will allow interaction with the core via HTTP requests. Through the web server, a user or program may retrieve details on connected devices, networks, or databases. See DE17 for details.

# Detailed Design

## DE1 – Installation

**Type**: Screen Sequence

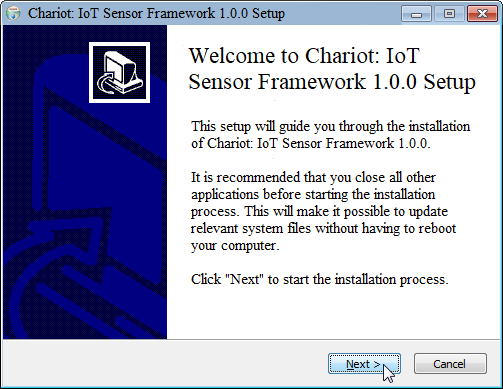
**Description**:

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.

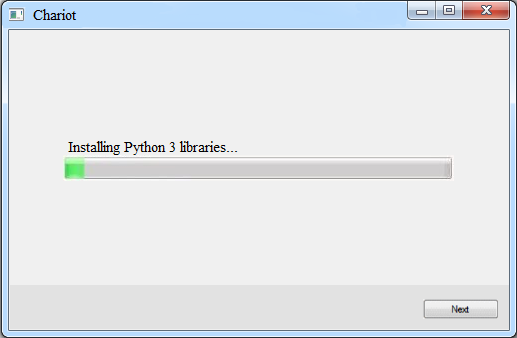
The screen shown in Figure 4.2 is a still-shot of the installation progress of the system onto the user’s device. There is nothing to interact with on this screen. The user must wait for the installation to finish before being able to click “Next.” A progress bar on the screen lets the user know where 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 “Close” to close the installation window.

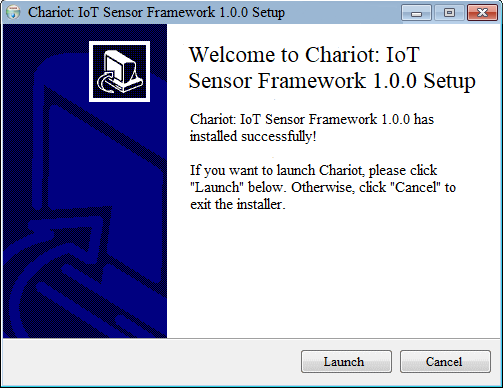
A high-level diagram of the flow in this sequence of screens can be seen in Figure 4.3.



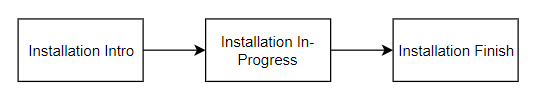
**Figure 4.1**: Chariot’s initial installation screen



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



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



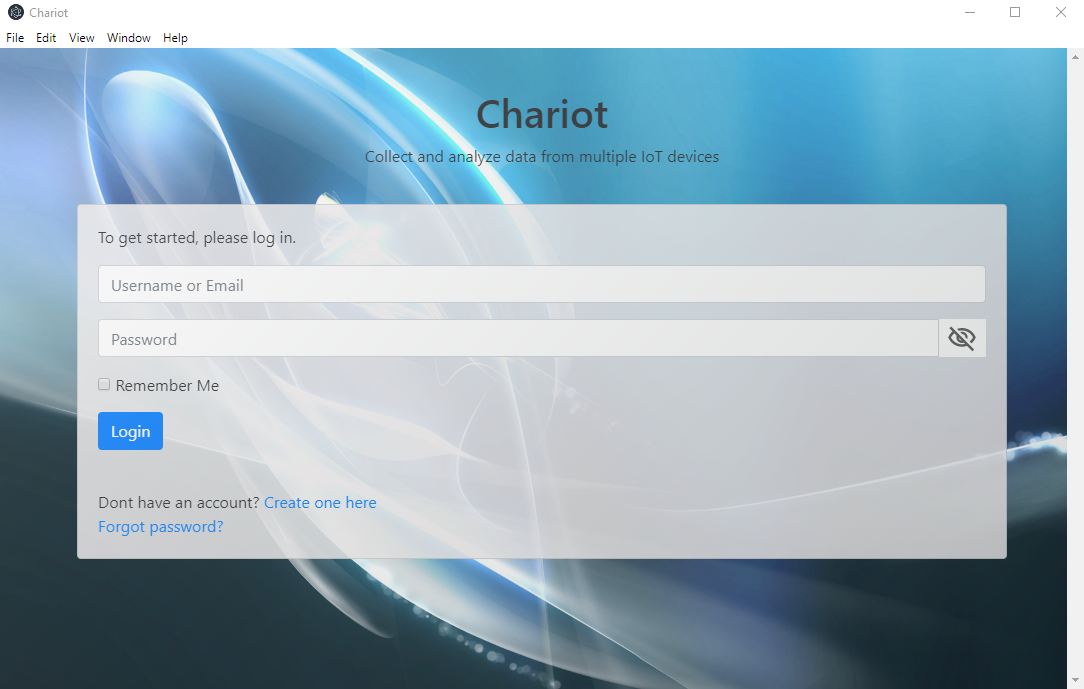
**Figure 4.4**: Chariot’s installation screen flow chart

## DE2 – Login 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 either login to their account, create a new account, or recover their password. A successful login will lead the user to the screen in Figure 4.9.



**Figure 4.5**: Chariot’s Login Screen

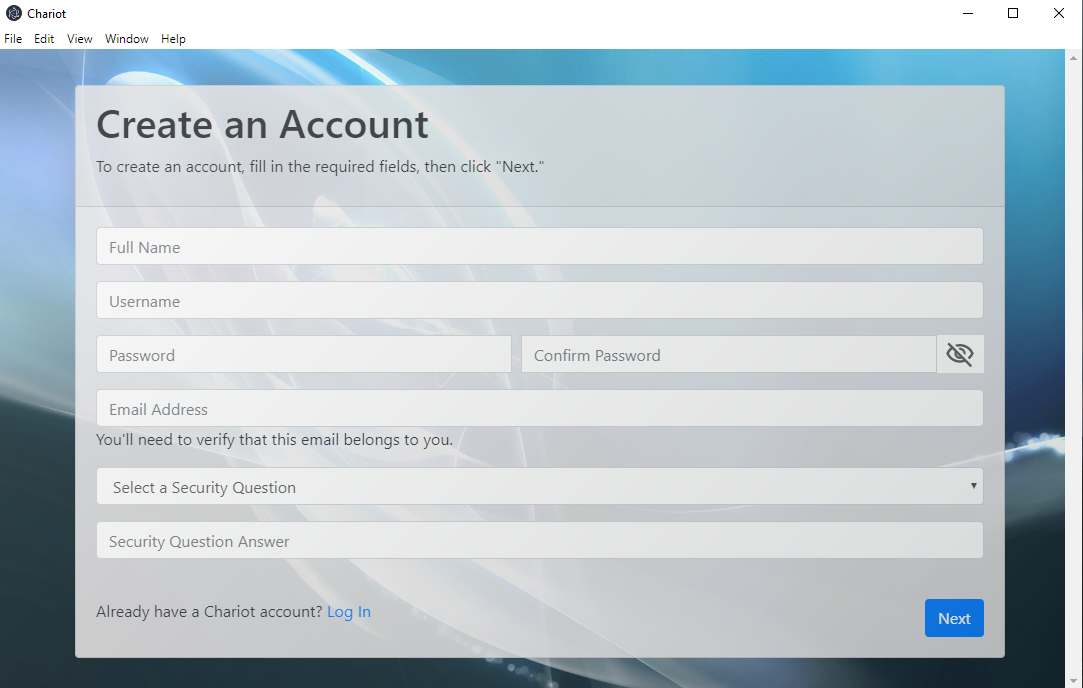
## DE3 – Account Creation Screens

**Type**: Screen Sequence

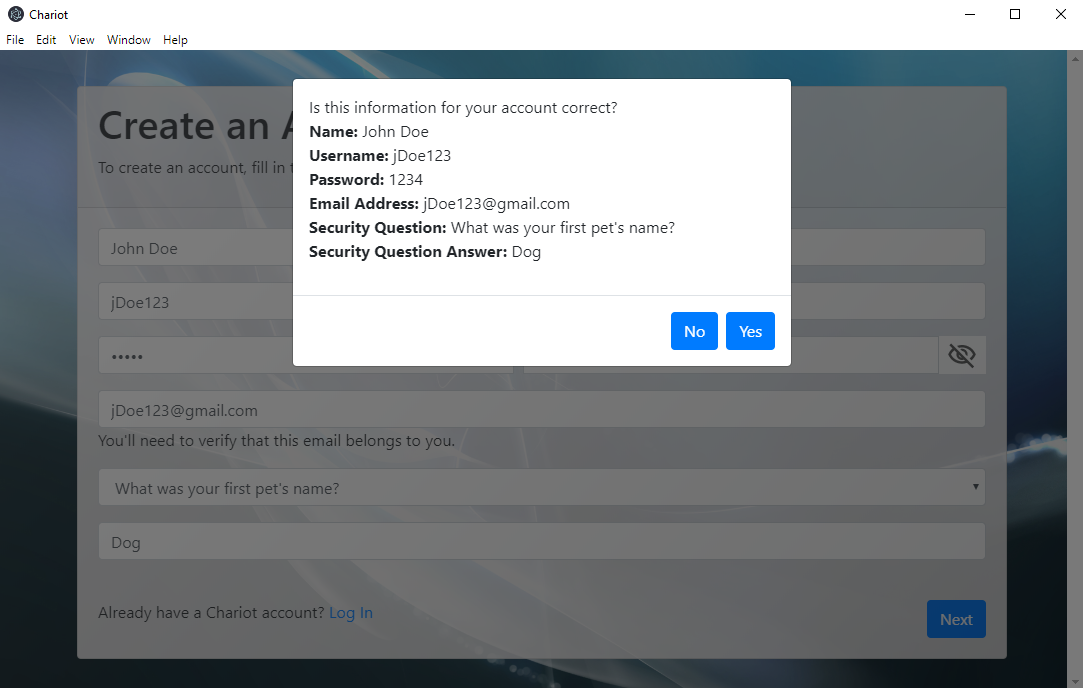
**Description**:

This sequence of screens allows the user to create an account with Chariot. Fields for a name, username, password, security question and answer, and email address are required. Without filling out these required fields, the user cannot move forward in the registration process. After filling out the aforementioned fields, which is also depicted in Figure 4.6, the user will then confirm their information in Figure 4.7. Once confirmed, the account will have to be validated, and then they can login with their newly set credentials. Clicking “Continue” in Figure 4.8 takes the user back to the login screen so they can log in.

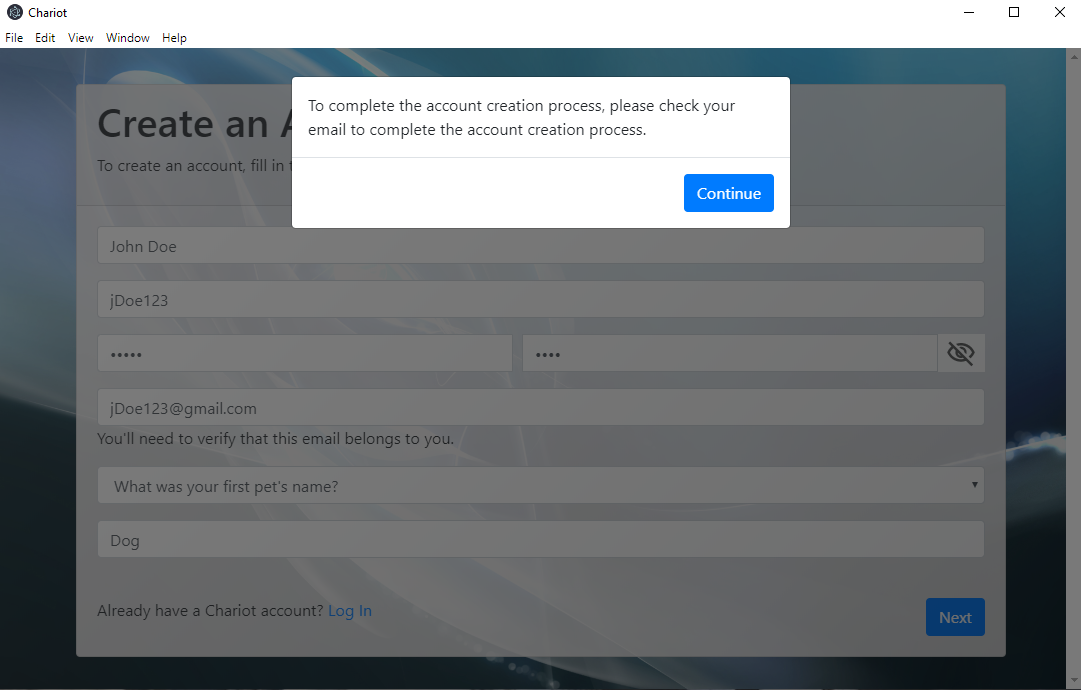
Usernames and email addresses must be unique between users, otherwise, an error message shall appear to let the user know that they most use a different username and/or email address.



**Figure 4.6**: Chariot’s Account Creation Screen



**Figure 4.7**: Account credential confirmation screen

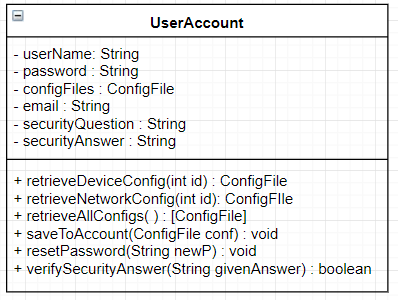


**Figure 4.8:** Account creation success screen

## DE3.1 – User Account

**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 – Welcome Screen

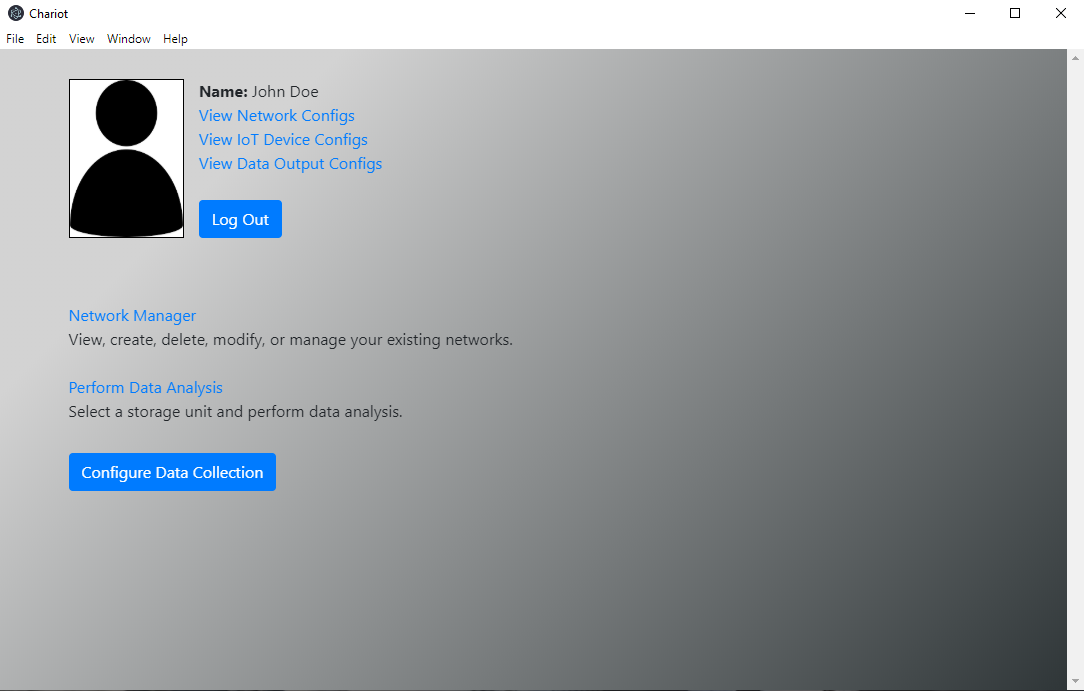
**Type**: Screen

**Description**:

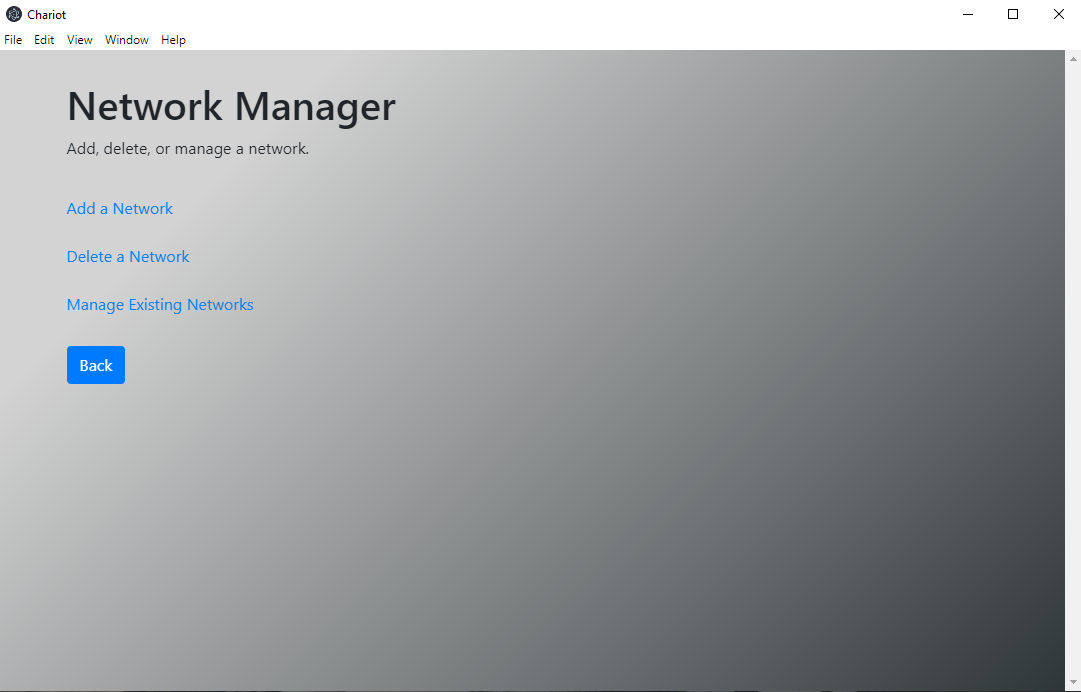
This screen appears after the user has successfully logged into their account. It shows the user their account details, as well as provides them with options on what they want to do next. They can either view their config files, manage a network and its corresponding IoT devices, perform data analysis, 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. They can also reconfigure devices connected to the 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. Further details of these screens will be included in future versions of this document as their requirements become clearer.



**Figure 4.9:** The user’s account (Welcome) page



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

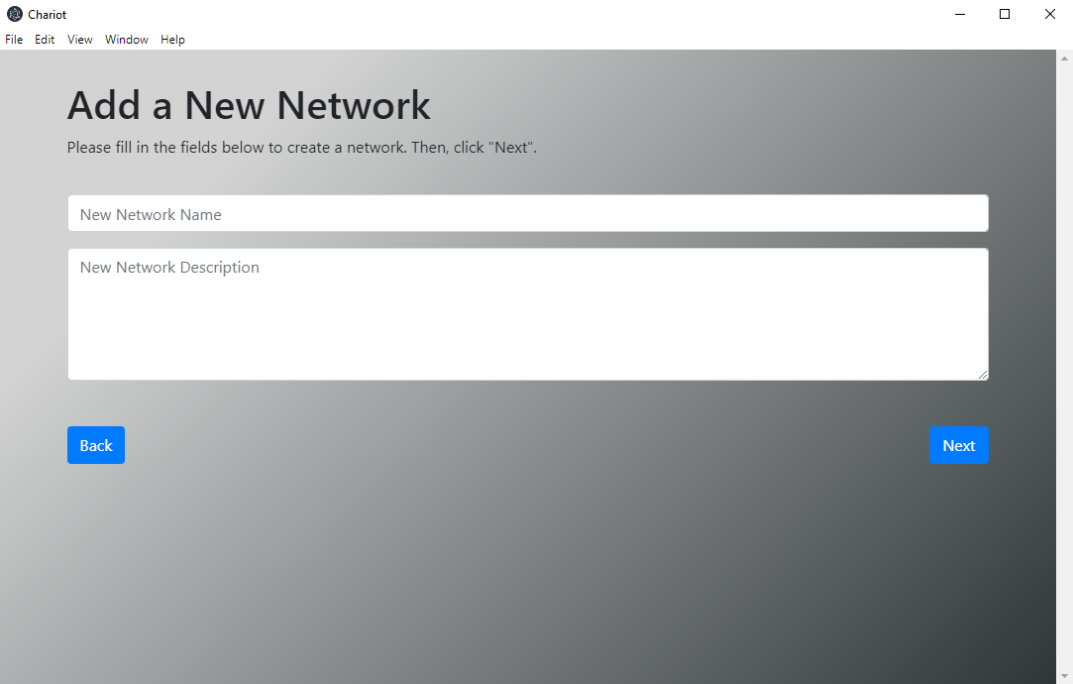
## DE5 – Create Network Screens

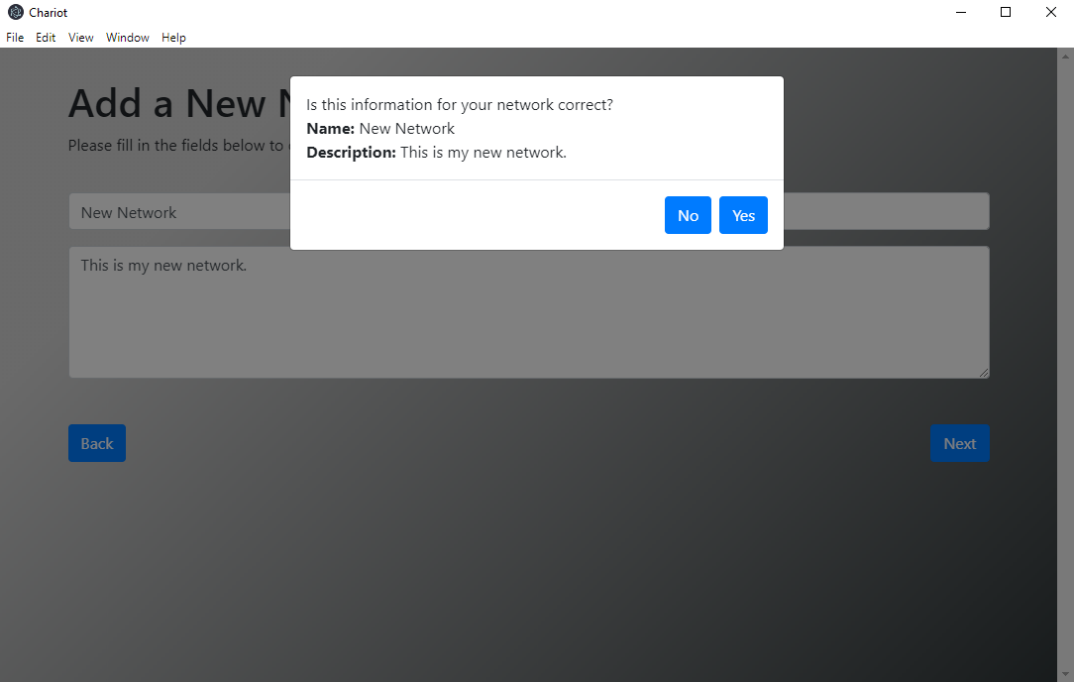
**Type**: Screen Sequence

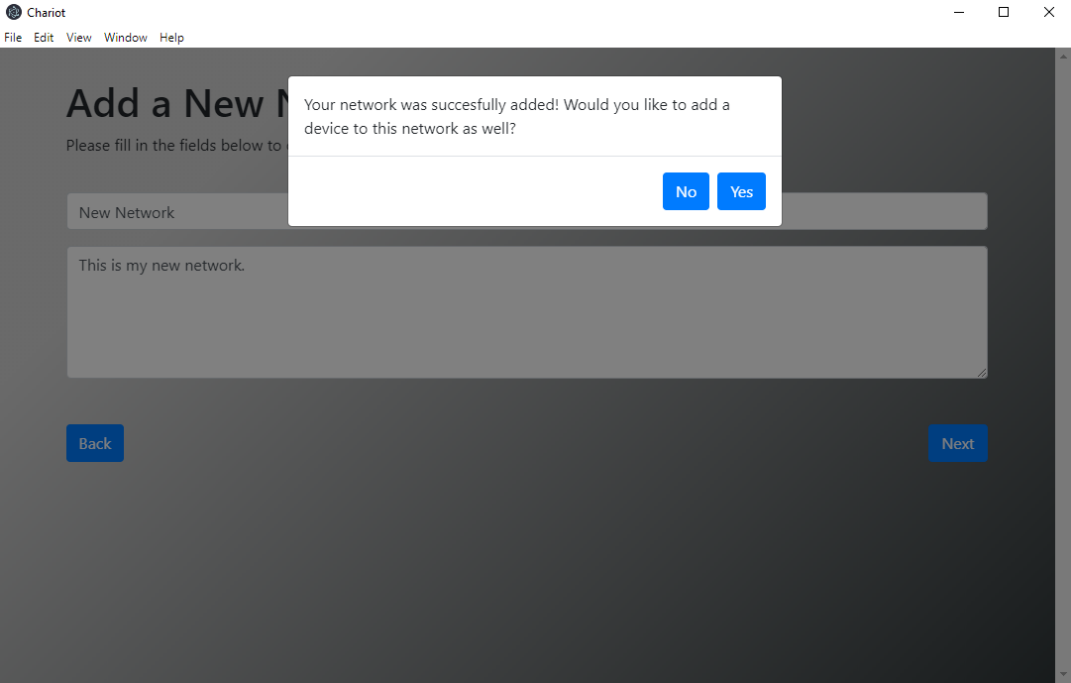
**Description**:

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 add to their existing list of networks.

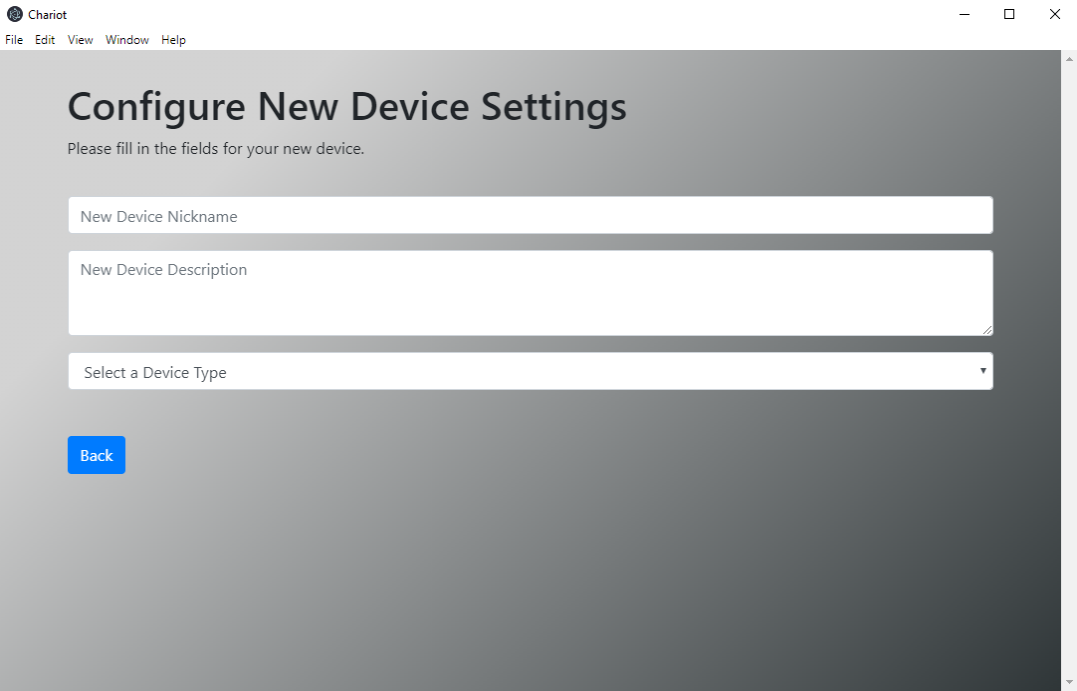
After creating a network, the user will have the option to add an IoT device to the newly created network. The screen depicting how a user would add a device is shown in Figure 4.12 and Figure 4.13. Choosing a device in Figure 4.12 expands the required fields, which is shown in Figure 4.13. After adding the IoT device, the network will have been created and the user will go back to their Welcome page (Figure 4.9) with a notification that the network/device was created successfully.



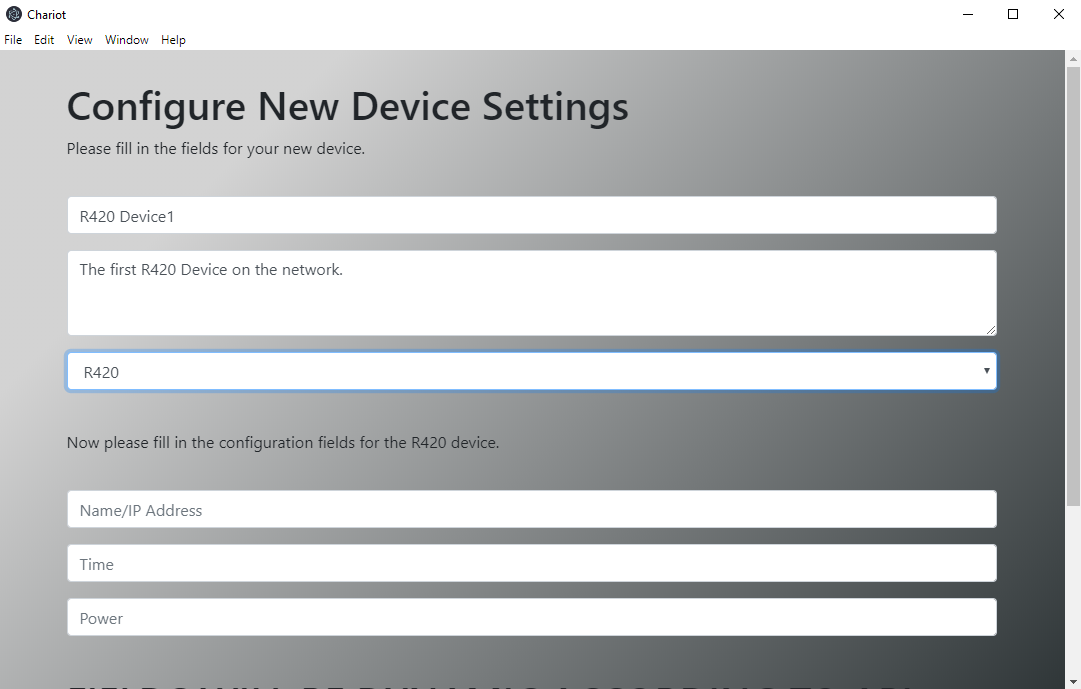




**Figure 4.11:** Screens used to add a new network to the user’s account



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



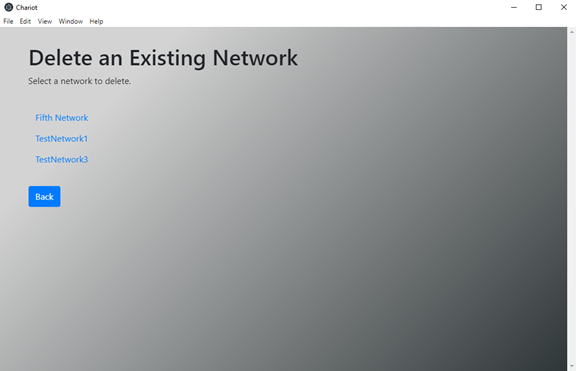
**Figure 4.13:** Screen allow the configuration of IoT device-specific parameters

## DE6 – Delete Network Screens

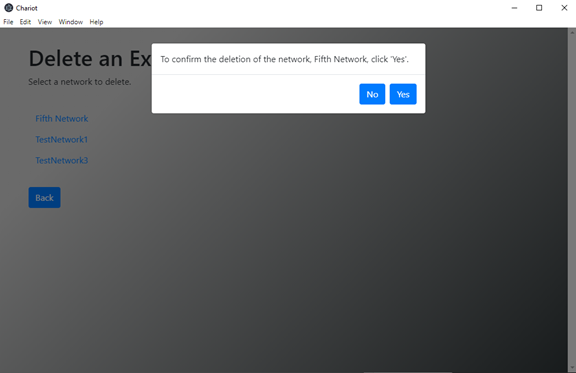
**Type**: Screen Sequence

**Description**:

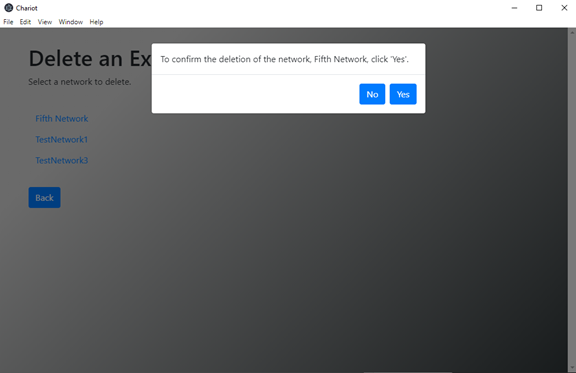
This collection of screens is available to the user when they choose to delete an existing network. The screens will guide the user through the process of deleting a network.



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



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



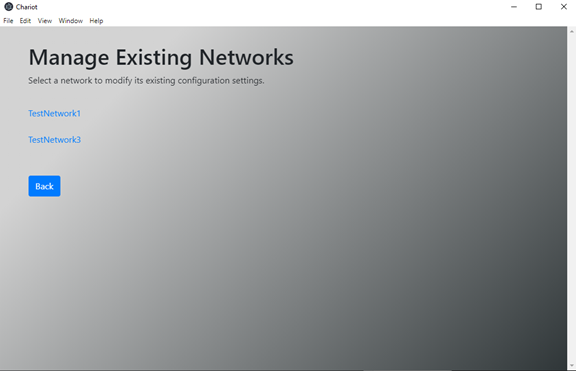
**Figure 4.16:** Notification letting user know that chosen network was successfully deleted

## DE7 – Manage Existing Networks Screens

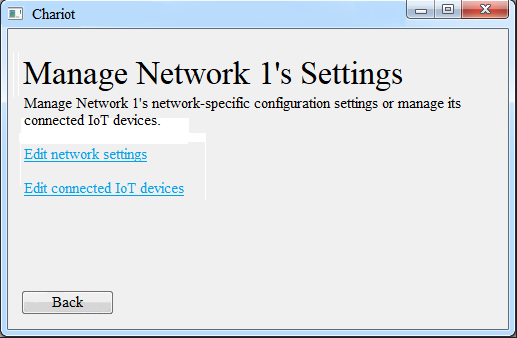
**Type**: Screen Sequence

**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.



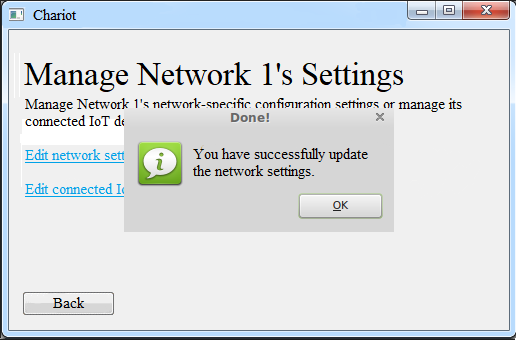
**Figure 4.17:** Screen where user chooses a network to modify its settings



**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



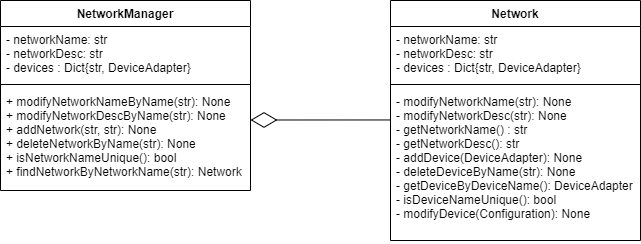
**Figure 4.20:** Successful modification of network-specific settings



**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** | findNetworkByNetworkName |
| **Input:** | String |
| **Output:** | Network |
| **Description:** | This method acts to return a Network instance if the input to the method matches the name of a network in the collection |

|  |  |
| --- | --- |
| **Method** | addNetwork |
| **Input:** | String, String |
| **Output:** | void |
| **Description:** | This method acts to create a Network and add it to the collection |

|  |  |
| --- | --- |
| **Method** | deleteNetworkByNetworkName |
| **Input:** | String |
| **Output:** | void |
| **Description:** | Given a network name, delete that name if it is found in the collection |

|  |  |
| --- | --- |
| **Method** | isNetworkNameUnique |
| **Input:** | String |
| **Output:** | Bool |
| **Description:** | Given a network name, check if the Network by that name already exists. |

## 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** | addDevice |
| **Input:** | DeviceAdapter |
| **Output:** | void |
| **Description:** | This method acts to create a Device that inherits from DeviceAdapter |

|  |  |
| --- | --- |
| **Method** | deleteDeviceByName |
| **Input:** | String |
| **Output:** | void |
| **Description:** | This method acts to create delete a device in the collection if the input string matches a device name in the collection |

|  |  |
| --- | --- |
| **Method** | getDeviceByDeviceName |
| **Input:** | String |
| **Output:** | DeviceAdapter |
| **Description:** | This method acts to retrieve a Device if the string inputted matches a device in the collection |

|  |  |
| --- | --- |
| **Method** | isDeviceNameUnique |
| **Input:** | String |
| **Output:** | Boolean |
| **Description:** | This method acts to return a boolean depending on if the input string matches a name in the collection |

## 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**:

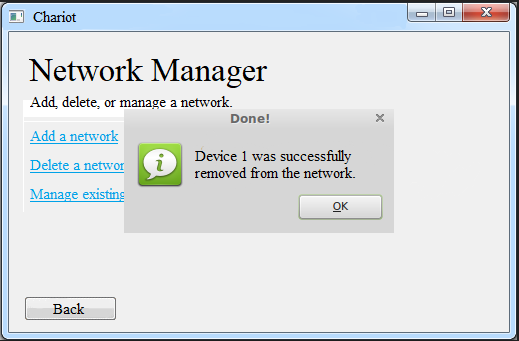
This collection of screens is available to the user when they choose to delete an existing IoT device from an existing network. Like the *Add an IoT Device* screen, the user must first choose to *Manage Existing Network* to see this collection of screens. These screens will let the user delete an IoT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



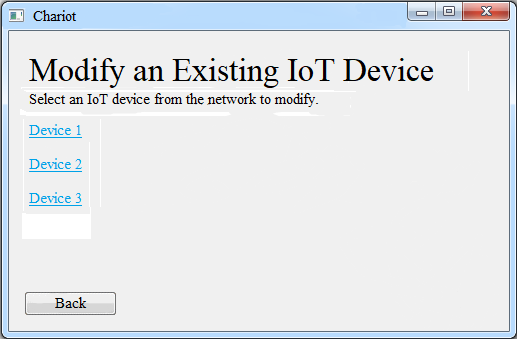
**Figure 4.24:** Successful deletion of network’s connected IoT device

## DE10 – Manage Existing IoT Devices Screens

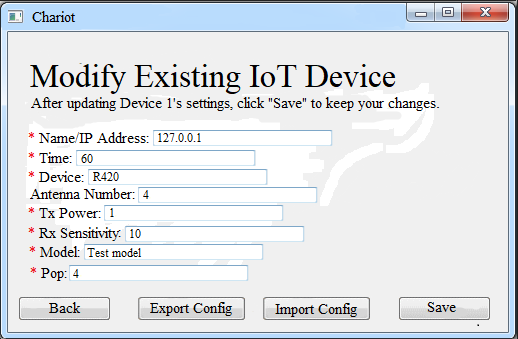
**Type**: Screen Sequence

**Description**:

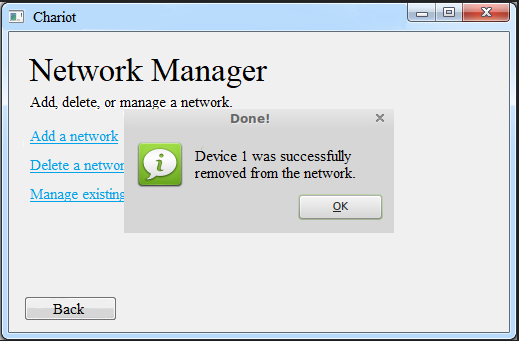
This collection of screens is available to the user when they choose to manage existing IoT devices on an existing network. These screens allow the user to modify IoT device-specific settings. From here, the user can modify device specific configuration settings via manual input or configuration file 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. This provides the devices where data will be collected 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.

A screenshot of a social media post

Description automatically generated

## DE12.1 – Data Collection Manager

**Type** : Code Entity

**Description:**

Manages the Data Collection Episode (DCE) by initializing threads and queues used for the collection and storage of device data. The Data Collection Manger also manages resolving of thread errors that occur during a DCE.

|  |  |  |
| --- | --- | --- |
| **Attributes** | **Type** | **Description** |
| activeNetwork | Union[Network, None] | Network to collect data from |
| devices | List[DeviceAdapter] | Collection of device adapters from the network |
| consumerThreads | List[ConsumerThread] | Collection of consumer threads that receives data from an adapter’s dataQueue |
| producerThreads | OrderedDict[str, ProducerThreads] | Collection of producer threads the put data in an adapter’s dataQueue |
| outputThread | ConsumerThread | A thread that collects data from different devices |
| errorQueue | ThreadQueue | Queue for the collection and handling of errors received from producer threads |
| dataQueue | ThreadQueue | Queue in which data from all devices collects to. |
| databaseWriter | DatabaseWriter | Configuration object, see below |
| outputStream | DataOutputStream | Outgoing data stream |
| \_inCollectionEpisode | Bool | Used to test whether there is an ongoing DCE or not |
| \_episodeStartTime | float | Used for timing how long the DCE has gone for |

|  |  |
| --- | --- |
| **Method** | \_consumeDataFromDevice |
| **Input:** | int: deviceIdx |
| **Output:** | None |
| **Description:** | Special case for when the network has only one connected device. |

|  |  |
| --- | --- |
| **Method** | \_consumeDataFromDevices |
| **Input:** | int: startIdx, int: numDevices |
| **Output:** | None |
| **Description:** | Consumer thread collects data from a subset of devices queues and then stores them to the dataQueue. |

|  |  |
| --- | --- |
| **Method** | \_handleErrorsInQueue |
| **Input:** | None |
| **Output:** | None |
| **Description:** | Goes through the errorQueue and resolves errors raised by the different producer threads in a loop. |

|  |  |
| --- | --- |
| **Method** | \_outputData |
| **Input:** | None |
| **Output:** | None |
| **Description:** | The outputThread read data from the dataQueue and using the databaseWrite stores it in a database and passes it to the outputStream) |

|  |  |
| --- | --- |
| **Method** | inCollectionEpisode |
| **Input:** | None |
| **Output:** | Bool |
| **Description:** | Returns the value of the \_inCollectionEpisode boolean |

|  |  |
| --- | --- |
| **Method** | setActiveNetwork |
| **Input:** | Network: Union[Network,None] |
| **Output:** | None |
| **Description:** | Sets the active network, this method cannot be performed during a DCE. |

|  |  |
| --- | --- |
| **Method** | getActiveNetwork |
| **Input:** | None |
| **Output:** | Network |
| **Description:** | Returns the current active network |

|  |  |
| --- | --- |
| **Method** | beginDataCollection |
| **Input:** | None |
| **Output:** | None |
| **Description:** | Begins a DCE, initializing producer, consumer and ouput threads. The final step would be to start \_handleErrorsInQueue. |

|  |  |
| --- | --- |
| **Method** | stopDataCollection |
| **Input:** | void |
| **Output:** | void |
| **Description:** | Ends the DCE, signaling all threads to stop. The Data Collection Manager will wait for all threads to stop before completing this method. |

## 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 must 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 – DatabaseWriter**

**Type**: Code Entity

**Description:**

Defines methods to write to a remote database. Implemented as an abstract class. MongoDatabaseWriter and MySQLDatabaseWriter, described below, implement this class to allow connection to MongoDB or MySQL database. The implementation of DatabaseWriter methods varies depending on the concrete class. The DatabaseWriterFactory class is responsible for creating DatabaseWriter instances.

A screenshot of a cell phone

Description automatically generated

|  |  |  |
| --- | --- | --- |
| Attributes | Type | Description |
| conn | Database connection | A connection link to the remote database. |
| cursor | Database cursor | A cursor capable of executing actions on the database. |

|  |  |
| --- | --- |
| Method | connect |
| Input: | void |
| Output: | void |
| Description: | Establishes a connection to the database. |

|  |  |
| --- | --- |
| Method | disconnect |
| Input: | void |
| Output: | void |
| Description: | End connection to the database gracefully. |

|  |  |
| --- | --- |
| Method | initializeTable |
| Input: | void |
| Output: | void |
| Description: | Create a table in the database to receive incoming IoT data. |

|  |  |
| --- | --- |
| Method | checkDataPoint |
| Input: | dataPoint: dict |
| Output: | void |
| Raises: | ValueError: if dataPoint is not valid |
| Description: | Ensures a dataPoint is method. A dataPoint should be a dictionary with exactly two fields: 1. relative\_time, and 2. freeform. If this is not the case, this method raises a ValueError. |

|  |  |
| --- | --- |
| Method | insertOne |
| Input: | dataPoint: dict |
| Output: | void |
| Description: | Check for validity of the dataPoint (using the checkDataPoint method described below), then insert into the table. |

|  |  |
| --- | --- |
| Method | insertMany |
| Input: | dataPoints: List[dict] |
| Output: | void |
| Description: | When possible, it is more efficient to insert multiple items in the database at once. This method allows for that. It checks for validity of each dataPoint, then inserts into the table. |

## DE15.1 – MongoDatabaseWriter

**Type**: Code Entity

**Description:**

A concrete class that implements the DatabaseWriter abstract class for use with MongoDB databases.

## DE15.2 – MySQLDatabaseWriter

**Type**: Code Entity

**Description:**

A concrete class that implements the DatabaseWriter abstract class for use with MySQL databases.

## DE15.3 – DatabaseWriterFactory

**Type**: Code Entity

**Description:**

This class is responsible for creating DatabaseWriter instances. It has only one method, getInstance. This method initializes construction of either a MongoDatabaseWriter or a MySQLDatabaseWriter depending on the DatabaseConfig input. These objects follow the factory method pattern to provide a consistent DatabaseWriter interface, despite different construction between the two classes. The factory is extensible: if a user creates another DatabaseWriter subclass to interface with an additional database type, they can also adapt the getInstance method to allow for construction of new objects.

|  |  |
| --- | --- |
| Method | getInstance |
| Input: | databaseConfiguration: DatabaseConfiguration |
| Output: | DatabaseWriter |
| Description: | Creates and returns a DatabaseWriter of the type specified by databaseConfiguration. |

## DE15.4 – DatabaseWriterConfiguration

**Type**: Code Entity

**Description:**

A DatabaseWriterConfiguration object contains all configuration variables a DatabaseWriter object needs to connect to a database. It is capable of outputting these variables to JSON for easy export.

|  |  |  |
| --- | --- | --- |
| Attributes | Type | Description |
| requiredFields | Dict[str, type] | Lists fields required by all DatabaseWriters and specifies their type. |
| optionalFields | Dict[str, type] | Lists optional fields. Empty by default. |

|  |  |
| --- | --- |
| Method | modifyConfig |
| Input: | newConfig: JSONDict |
| Output: | void |
| Description: | Modifies attributes the configuration represents. |

|  |  |
| --- | --- |
| Method | toJSON |
| Input: | void |
| Output: | String |
| Description: | Outputs attributes and their types as a JSON object |

MySQLDatabaseConfiguration and MongoDatabaseConfiguration both inherit from this class. The subclasses change the requiredFields and optionalFields dictionaries, depending on the variables needed by the corresponding DatabaseWriter.

## 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. |

## DE17 – API Endpoints

To interact with Chariot, there are a set of endpoints that can be used. The file that contains these endpoints and error handlers is called webserver.py. In order to manage interaction, just run the class and use the endpoints as necessary. As good practice for endpoints, this first version will have the following URL common to all: chariot/api/v1.0

## DE17.1 – Getting a network name and its description

**Type**: API

**Description:**

This endpoint is responsible for getting all network names and descriptions for a user. This GET request will retrieve network names with the name as key and description as value for that key.

|  |  |
| --- | --- |
| **Method** | retrieveAllNetworkNames |
| **Input:** | URL: chariot/api/v1.0/networks/names |
| **Output:** | Dictionary {String, String} |
| **Description:** | This GET request will retrieve network names with the name as key and description as value for that key. |

## DE17.2 – Retrieving all Networks and their devices

**Type**: API

**Description:**

This endpoint is responsible for getting all network names, their descriptions and the devices that are contained in the network.

|  |  |
| --- | --- |
| **Method** | retrieveAllNetworkDetails |
| **Input:** | URL: chariot/api/v1.0/networks/all |
| **Output:** | Dictionary {String, String} |
| **Description:** | This GET request is responsible for getting all network names, their descriptions and the devices that are contained in the network |

## DE17.3 – Creating a Network

**Type**: API

**Description:**

This endpoint is responsible for the creation of a Network. This POST request expects that “NetworkName” be in the payload. Furthermore, a description is optional for networks so if “Description” is also in the payload, then the value will be registered with the new network. Network names must be unique, failure to add a unique name will result in an error.

|  |  |
| --- | --- |
| **Method** | createNetwork |
| **Input:** | Endpoint: Chariot/api/v1.0/network  Payload specifying using “NetworkName” and value as key-value pair and the optional “Description” |
| **Output:** | 200 success or 400 error |
| **Description:** | This POST request is responsible for creating a new network. Done correctly, a 200-success code is returned. |

## DE17.4 – Modifying a Network

**Type**: API

**Description:**

This endpoint is responsible for the modification of a Network. This PUT request expects that “NewName” and “NetworkName” be in the payload. Furthermore, a description is optional for networks so if “Description” is also in the payload, then the value will be registered and overwrite the old network description. Network names must be unique, failure to add a unique name will result in an error.

|  |  |
| --- | --- |
| **Method** | modifyNetwork |
| **Input:** | Endpoint: Chariot/api/v1.0/network  Payload specifying using “NetworkName” and value as key-value pair and “NewName” for the new name of the modified network |
| **Output:** | 200 success or 400 error |
| **Description:** | This PUT request is responsible for modifying a network. Done correctly, a 200-success code is returned. |

## DE17.5 – Deleting a Network

**Type**: API

**Description:**

This endpoint is responsible for the deletion of a Network. This DELETE request expects that “NetworkName” be in the payload. The value of the key NetworkName will be deleted from Chariot along with devices configured to that network. If a name that does not exist in the collection is specified in the payload, then an error is returned.

|  |  |
| --- | --- |
| **Method** | deleteNetwork |
| **Input:** | Endpoint: Chariot/api/v1.0/network?NetworkName=  URL specifying “?NetworkName=” and the name of a network |
| **Output:** | 200 success or 400 error |
| **Description:** | This DELETE request is responsible for deleting a network and its contents. Done correctly, a 200-success code is returned. |

## DE17.6 – Get a Network’s details

**Type**: API

**Description:**

This endpoint is responsible for retrieving a Network’s contents. This GET request expects that “NetworkName” be in the URL. The value of the key NetworkName in the url will then be used to retrieve information from a specified network such as. If a name that does not exist in the collection is specified in the url, then an error is returned.

|  |  |
| --- | --- |
| **Method** | getNetworkDetails |
| **Input:** | Endpoint: Chariot/api/v1.0/network?NetworkName=  URL specifying “?NetworkName=” and the name of a network |
| **Output:** | 200 success or 400 error |
| **Description:** | This GET request is responsible for deleting a network and its contents. Done correctly, a 200-success code is returned. |

## DE17.7 – Get all the information about supported devices

**Type**: API

**Description:**

This endpoint is responsible for retrieving all of the supported devices known to Chariot. This GET request expects that no additional information to be in the URL. The returned content is that of the file supportedDevices.json which has supported device names as well as required and optional fields specific to that device.

|  |  |
| --- | --- |
| **Method** | GetSupportedDevices() |
| **Input:** | Endpoint: Chariot/api/v1.0/network/device/supportedDevices  No additional information needed |
| **Output:** | 200 success |
| **Description:** | This GET request is responsible for retrieving supported devices as well as the configuration fields from each device that are required and optional. |

## DE17.8 – Get all the information about a specific device

**Type**: API

**Description:**

This endpoint is responsible for retrieving all of the information specific to a device known to Chariot. This GET request expects that a “DeviceName” be in the URL. The returned content is JSON content that includes the device name and the fields specific to that device. For example, specifying ImpinjXArray in the GET request would return the contents of ImpinjXArray.json which has supported device name as well as required and optional fields specific to that device. That is, this endpoint returns a template to be filled out with configuration values.

|  |  |
| --- | --- |
| **Method** | getSupportedDeviceConfig |
| **Input:** | Endpoint: Chariot/api/v1.0/network/device/config?DeviceName=  URL specifying “?DeviceName=” for which to receive configuration information |
| **Output:** | 200 success |
| **Description:** | This GET request is responsible for retrieving a supported device name as well as required and optional fields specific to that device. |

## DE17.9 – Get information about a configured device on a network

**Type**: API

**Description:**

This endpoint is responsible for retrieving the configuration information of a specified device in a specified network. This GET request expects that both “NetworkName” and “DeviceName” be in the URL. This is so that the configuration information of a device from a specific network is retrieved. The returned content is JSON content that includes the device name and the fields specific to that device as well as the values currently filled in for those fields. The difference between this and the last endpoint is that the previous endpoint returns a template to be filled out whereas this endpoint retrieves information about a device that has already been configured.

|  |  |
| --- | --- |
| **Method** | getDeviceDetails |
| **Input:** | Endpoint: Chariot/api/v1.0/network/device?DeviceName=  URL specifying “?DeviceName=” for which to receive configuration information |
| **Output:** | 200 success |
| **Description:** | This GET request is responsible for retrieving a supported device’s configuration as well as currently registered values from a specified network. |

## DE17.10 – Create a device on a specified network

**Type**: API

**Description:**

This endpoint is responsible for creating a device on a specified network. This POST request at minimum expects that “NetworkName”, “DeviceId”, and “DeviceType” be in the payload. Additional information will be dependent on the device being created. Devices Ids in a single network must be unique so there are possibilities for 400 errors

|  |  |
| --- | --- |
| **Method** | createDevice |
| **Input:** | Endpoint: Chariot/api/v1.0/network/device  Payload specifying at least“NetworkName”, “DeviceId”, and “DeviceType” as key value pairs, additional information is dependant upon what deviceType is to be created. |
| **Output:** | 200 success |
| **Description:** | This POST request is responsible for creating a supported device on a specified network. |

## DE17.11 – Modifying a device on a specified network

**Type**: API

**Description:**

This endpoint is responsible for modifying a device on a specified network. This PUT request at minimum expects that “NetworkName” and “DeviceId” be in the payload. Additional information will be dependent on the device being modified. Devices Ids in a single network must be unique so there are possibilities for 400 errors

|  |  |
| --- | --- |
| **Method** | modifyDevice |
| **Input:** | Endpoint: Chariot/api/v1.0/network/device  Payload specifying at least“NetworkName”, “DeviceId”, and “DeviceType” as key value pairs, additional information is dependant upon what deviceType is to be created. |
| **Output:** | 200 success |
| **Description:** | This POST request is responsible for creating a supported device on a specified network. |

## DE17.12 – Deleting a device on a specified network

**Type**: API

**Description:**

This endpoint is responsible for deleting a device on a specified network. This DELETE request at minimum expects that “NetworkName” and “DeviceId” be in the URL. If the specified device does not exist, then a 400 error is returned.

|  |  |
| --- | --- |
| **Method** | deleteDevice |
| **Input:** | Endpoint: Chariot/api/v1.0/network/device?NetworkName=xx?DeviceName=  URL specifying at “NetworkName” and “DeviceName”. This is used to delete a specific device from a network. |
| **Output:** | 200 success |
| **Description:** | This DELETE request is responsible for deleting a device on a specified network. |

**Appendix**

A) System Class Diagram

## A close up of a map Description automatically generated

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>