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:** 3 |  |
| **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 6](#_Toc38726623)

[1.2 Definitions, Acronyms, and Abbreviations 6](#_Toc38726624)

[1.3 Requirements Traceability Matrix 6](#_Toc38726625)

[2 Architectural Description 10](#_Toc38726626)

[3 Interface Description 17](#_Toc38726627)

[3.1 User Interface 17](#_Toc38726628)

[3.2 Data Interface 18](#_Toc38726629)

[3.2.1 Device data stream 19](#_Toc38726630)

[3.2.2 Database Writer 20](#_Toc38726631)

[3.3 Programming Interface 20](#_Toc38726632)

[4 Detailed Design 20](#_Toc38726633)

[DE1 – Installation 20](#_Toc38726634)

[DE2 – Login Screen 21](#_Toc38726635)

[DE3 – Account Creation Screens 22](#_Toc38726636)

[DE3.1 – User Account 24](#_Toc38726637)

[DE4 – Welcome Screen 25](#_Toc38726638)

[DE5 – Create Network Screens 26](#_Toc38726639)

[DE6 – Delete Network Screens 29](#_Toc38726640)

[DE7 – Manage Existing Networks Screens 30](#_Toc38726641)

[DE7.1 – Manage Existing Networks 33](#_Toc38726642)

[DE7.2 – Network 34](#_Toc38726643)

[DE8 – Add IoT Device Screens 35](#_Toc38726644)

[DE9 – Delete IoT Device Screens 35](#_Toc38726645)

[DE10 – Manage Existing IoT Devices Screens 37](#_Toc38726646)

[DE11 – Data Collection Screens 39](#_Toc38726647)

[DE12 – Data Collection 39](#_Toc38726648)

[DE12.1 – Data Collection Manager 40](#_Toc38726649)

[DE12.2 - Data Collector 41](#_Toc38726650)

[DE12.3 – Data Collection Worker 43](#_Toc38726651)

[DE13 – Data Analysis Screens 46](#_Toc38726652)

[DE14 – Device Adapters and Device Adapter Factories 46](#_Toc38726653)

[DE14.1 – Device Adapter Factory 46](#_Toc38726654)

[DE14.2 – Device Adapter 47](#_Toc38726655)

[DE14.3 – Device Config 48](#_Toc38726656)

[DE15.1 – MongoDatabaseWriter 50](#_Toc38726657)

[DE15.2 – MySQLDatabaseWriter 50](#_Toc38726658)

[DE15.3 – DatabaseWriterFactory 50](#_Toc38726659)

[DE15.4 – DatabaseWriterConfiguration 50](#_Toc38726660)

[DE16 – Data Output Adapter 51](#_Toc38726661)

[DE16.1 – Data Output Adapter 52](#_Toc38726662)

[DE16.2 – Data Output Adapter Creator 53](#_Toc38726663)

[DE16.3 – Data Output Config 53](#_Toc38726664)

[DE18 – API Endpoints 54](#_Toc38726665)

[DE18.1 – Getting a network name and its description 55](#_Toc38726666)

[DE18.2 – Retrieving all Networks and their devices 55](#_Toc38726667)

[DE18.3 – Creating a Network 55](#_Toc38726668)

[DE18.4 – Modifying a Network 56](#_Toc38726669)

[DE18.5 – Deleting a Network 56](#_Toc38726670)

[DE18.6 – Get a Network’s details 56](#_Toc38726671)

[DE18.7 – Get all the information about supported devices 57](#_Toc38726672)

[DE18.8 – Get all the information about a specific device 57](#_Toc38726673)

[DE18.9 – Get information about a configured device on a network 58](#_Toc38726674)

[DE18.10 – Create a device on a specified network 58](#_Toc38726675)

[DE18.11 – Modifying a device on a specified network 59](#_Toc38726676)

[DE18.12 – Deleting a device on a specified network 59](#_Toc38726677)

[DE18.13 – Get all the information about supported databases 59](#_Toc38726678)

[DE18.14 – Get information about a specific database configuration 60](#_Toc38726679)

[DE18.15 – Get information about a configured database 60](#_Toc38726680)

[DE18.16 – Create a database configuration 61](#_Toc38726681)

[DE18.17 – Modifying a database configuration 61](#_Toc38726682)

[DE18.12 – Deleting a database configuration 62](#_Toc38726683)

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 can be installed onto a user’s machine by running specific commands on the command line. To use Chariot, the user must first go through this installation process. More details can be seen in DE1.

**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, IoT devices, and database configurations, and starting a data collection episode. More specific details about each individual screen and feature can be found in the Screen Sequence types throughout Section 4.

**CD4 – User Account**

Chariot manages user information via user accounts. User-specific data and settings such as network and device configurations are saved to the user’s account. Account creation is described in DE2 and DE3.

**CD5 – Data Collection Framework**

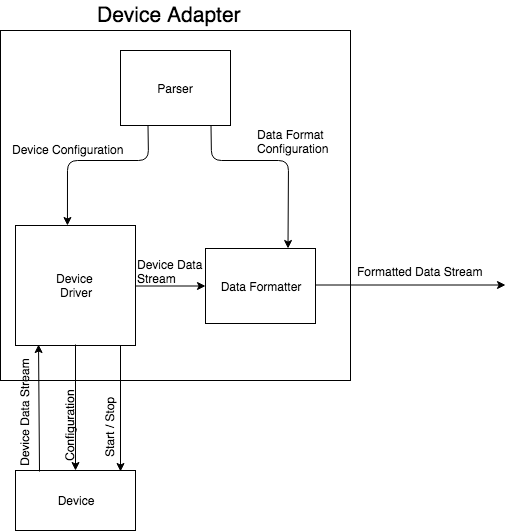
A close up of a piece of paper

Description automatically generated

**Figure 2.2**: Data Collection Framework diagram.

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**



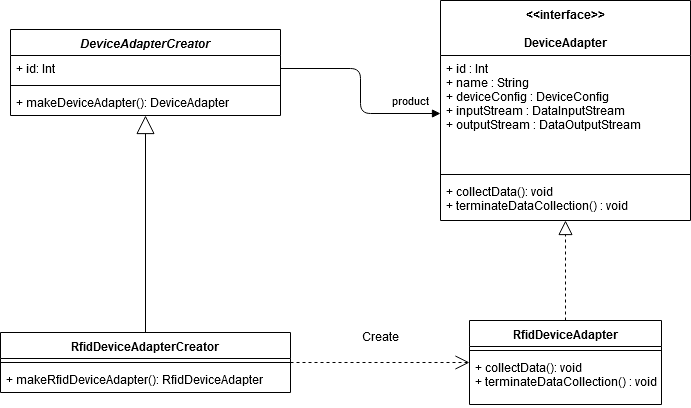
**Figure 2.3**: Device Adapter component diagram.

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 DE7), 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. See also figure 2.4.

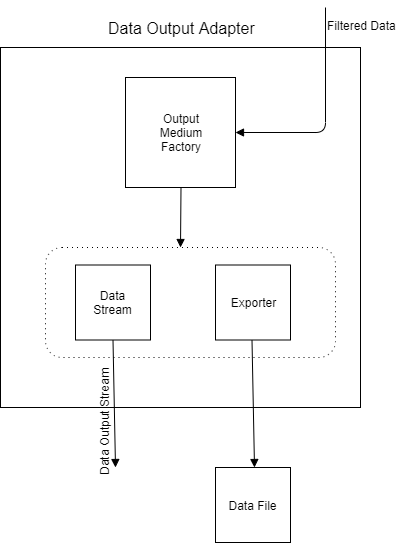


**Figure 2.4** Device Adapter, Device Adapter Creator class diagram.

**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**



**Figure 2.5** Data Output Adapter component diagram.

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**

A screenshot of a social media post

Description automatically generated

**Figure 2.6:** Database Writer component diagram.

Chariot writes data to a MongoDB or MySQL database. The system can be extended to support other database types. See DE15 for more detail on Database Writer implementation. The Database Writer reads data from a queue and writes to the database, looping while a data collection episode is running. See section 3.2 for more detail about the data collection process.

**CD11 – Database**

The database is relatively simple, consisting of only one table. This table contains five fields:

|  |  |  |
| --- | --- | --- |
| **Variable name** | **Type** | **Description** |
| ID | Int | The primary key, simply increments with every record. |
| device\_name | String | As data from multiple devices may be inserted into the database, Chariot must track which device each record is associated with. |
| insertion\_time | Int | The time the record is inserted into the database, recorded as milliseconds since epoch. |
| production\_time | Int | The time the data was produced by the device, recorded as milliseconds since epoch. |
| freeform | JSON | The data produced by the device. Stored as a JSON array to allow for flexibility in the format of the data. See section 3.2.1 for discussion on device data. |

Data production and insertion into the database occur asynchronously, as these operations are performed by different threads. Chariot records both timestamps so that a user can evaluate the time difference between data production and insertion.

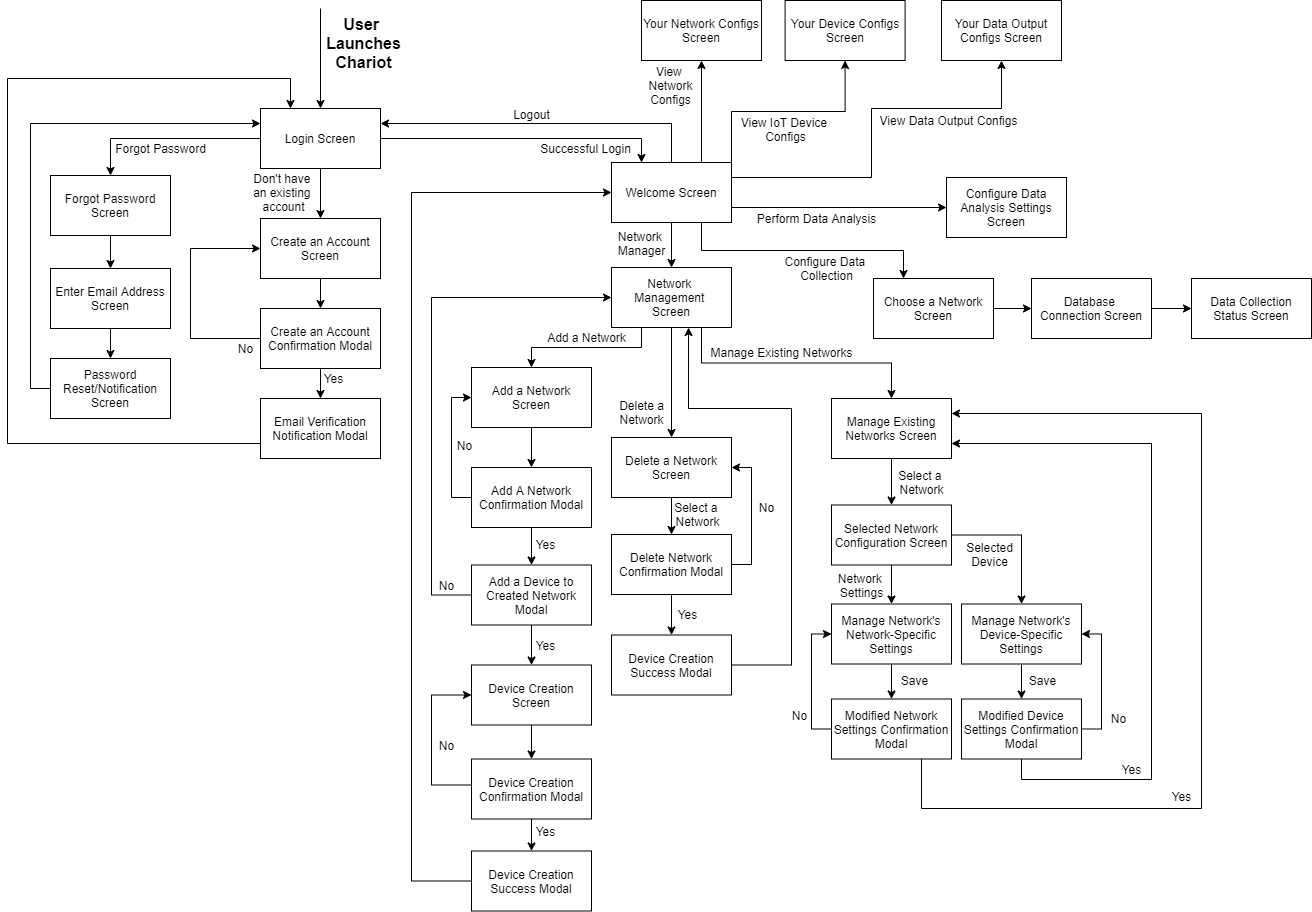
# Interface Description

## 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 user must first install Chariot via the command line. See installation details in DE1. After installation, the user may launch the system.

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.

Chariot aims to collect and store data from multiple devices. Thus, Chariot must utilize concurrency to collect and store data in a timely manner. This is an instance of the producer-consumer problem: one class of threads, producers, adds data to a shared collection object, while another class of threads, consumers, retrieves and writes the data. The producers should stop adding data if the collection is filled, and consumers should only retrieve data when the collection is not empty.

A system like Chariot can face problems if the producer-consumer problem is not addressed carefully. In a simple solution, each IoT device adapter acts as a producer and appends data to a shared collection object, implemented as a queue. A single reader acts as the consumer and is tasked with popping data from the queue and writing it to the database. Though this solution is attractive in its simplicity, it is inefficient, as threads would constantly block each other.

A close up of a map

Description automatically generated

**Figure 3.2:** A naïve approach to Chariot’s producer-consumer problem

Another solution is to create a queue for every producer thread while still using a single consumer thread. Each device has its own polling rate and fills the queue at a different rate. The consumer thread collects data from each queue, collecting data and writing it to the database. If one queue is empty, it moves on to the next queue, looping until the data collection episode terminates. The issue with this approach is that the consumer thread could be overwhelmed by the amount of incoming data from the producers. This could cause significant delay depending on the device polling rates and the size of the network.

A close up of a map

Description automatically generated

**Figure 3.3:** Another approach to Chariot’s producer-consumer problem

The actual implementation Chariot uses to solve the producer-consumer problem is a combination of the above approaches. As in the second solution (Figure 3.3), every producer thread puts data into its own queue. The consumer thread polls each of the device queues, collecting all the data in the queue and appending it to a singular queue only accessed by the consumer. This allows producers to keep adding to their respective queues and allows the consumer to add records to the database in a timely manner. It is the best solution as it requires the least amount of interaction between the threads, protecting them from starving or deadlocking one another. There is still a risk with this solution as the consumer can build a list of writable data that is so large, that it takes a noticeable amount of time for it to write to the database.

A close up of a map

Description automatically generated

**Figure 3.4:** The actual solution used in 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.

### Database Writer

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 more details.

# Detailed Design

## DE1 – Installation

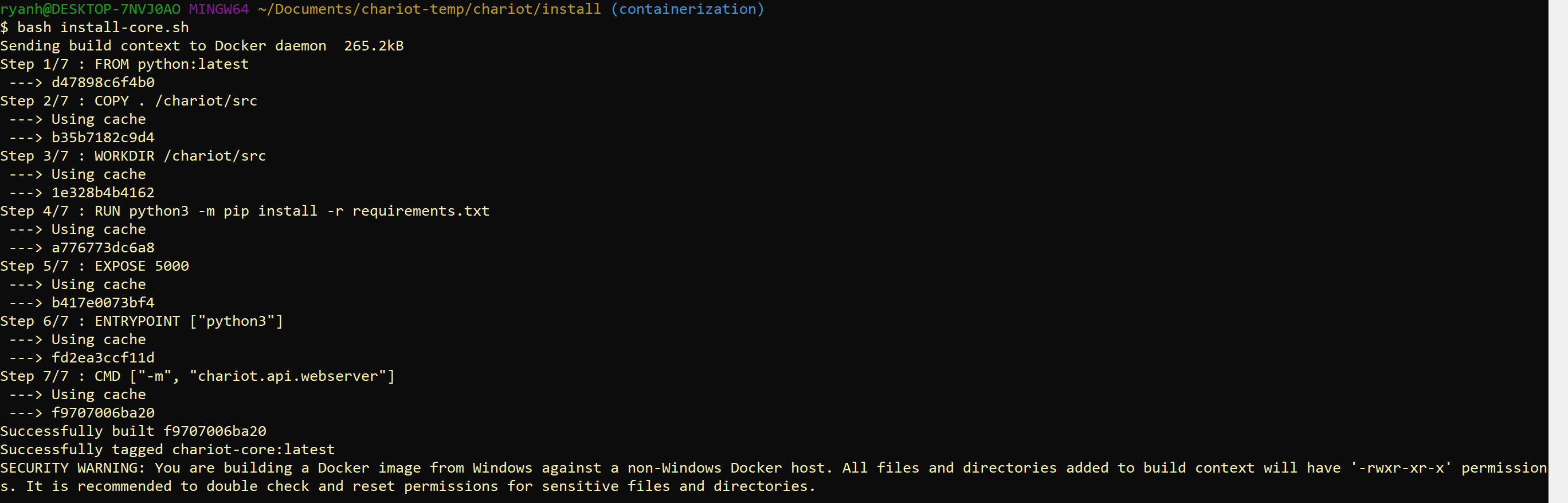
**Type:** Shell scripts

**Description**:

Chariot is installed via the command line. Upon downloading the repository from Github, the user navigates to the ‘install’ directory in the shell. Then, they may run the command

$ bash install-core.sh

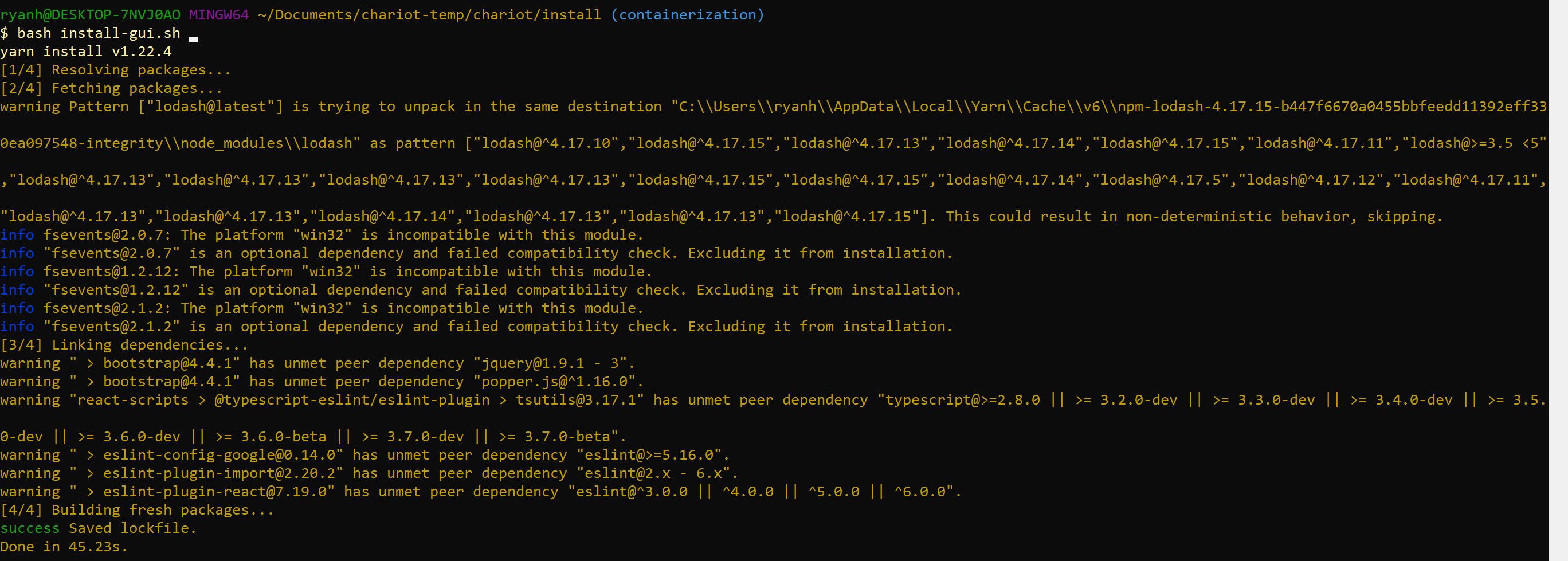
to install the Chariot core.



**Figure 4.1:** Successful install of the Chariot core.

If the user wishes to use Chariot via the official GUI, they may install it via the command

$ bash install-gui.sh



**Figure 4.2:** Successful install of the Chariot GUI. Note that warnings may vary.

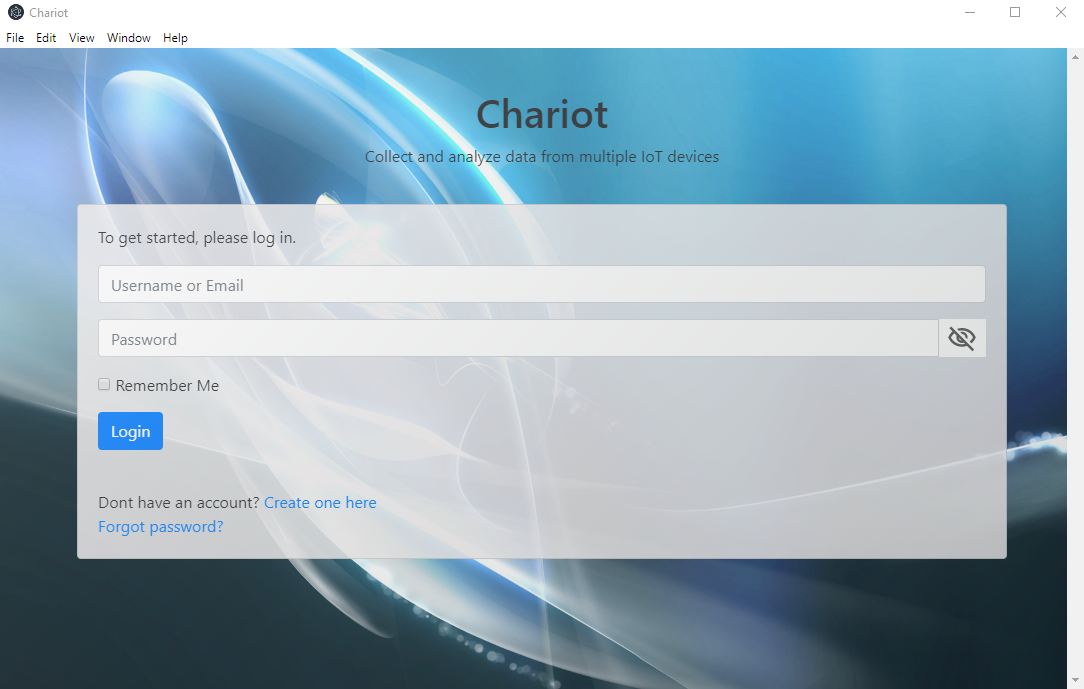
Full details on installation may be found on the Chariot Github.

## 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

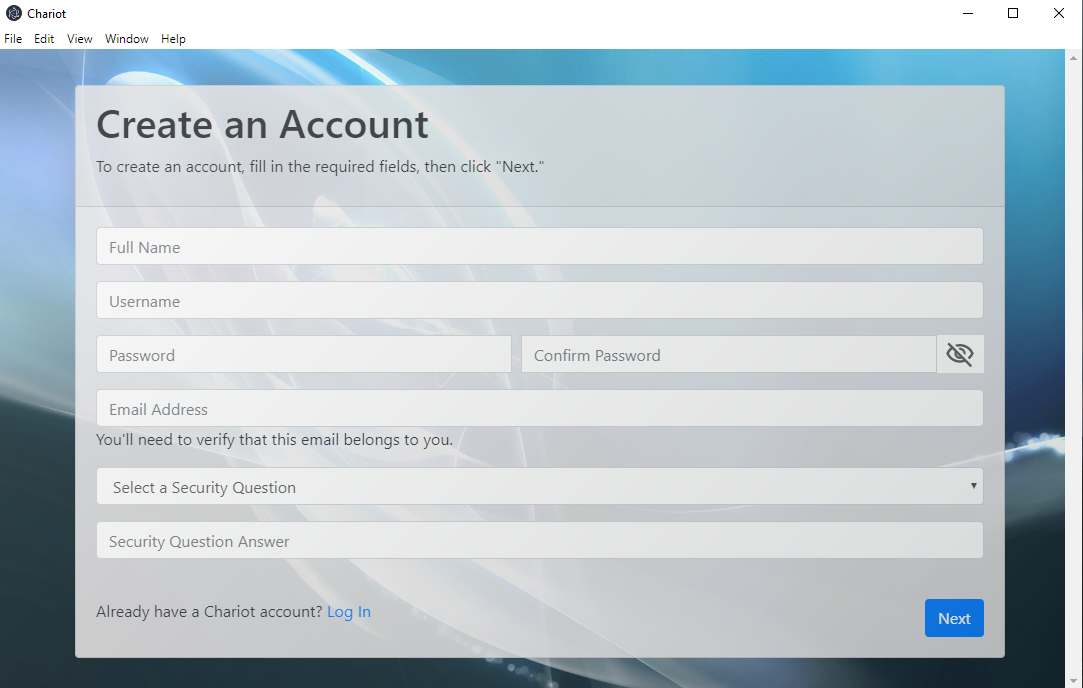
## DE3 – Account Creation Screens

**Type**: Screen Sequence

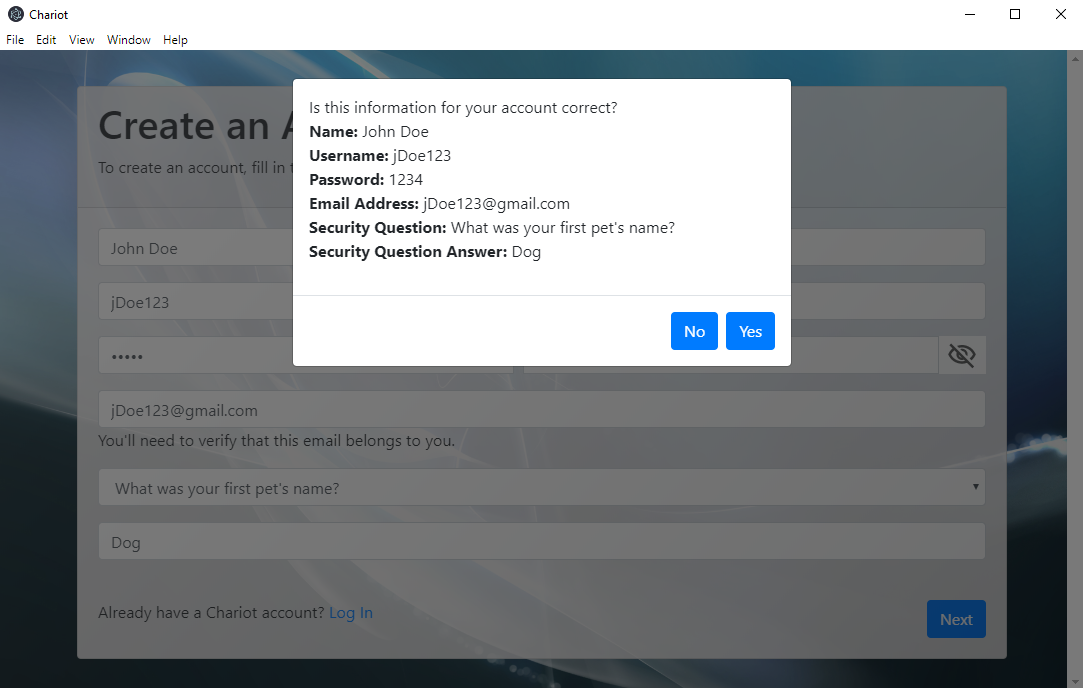
**Description**:

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 fields, 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.

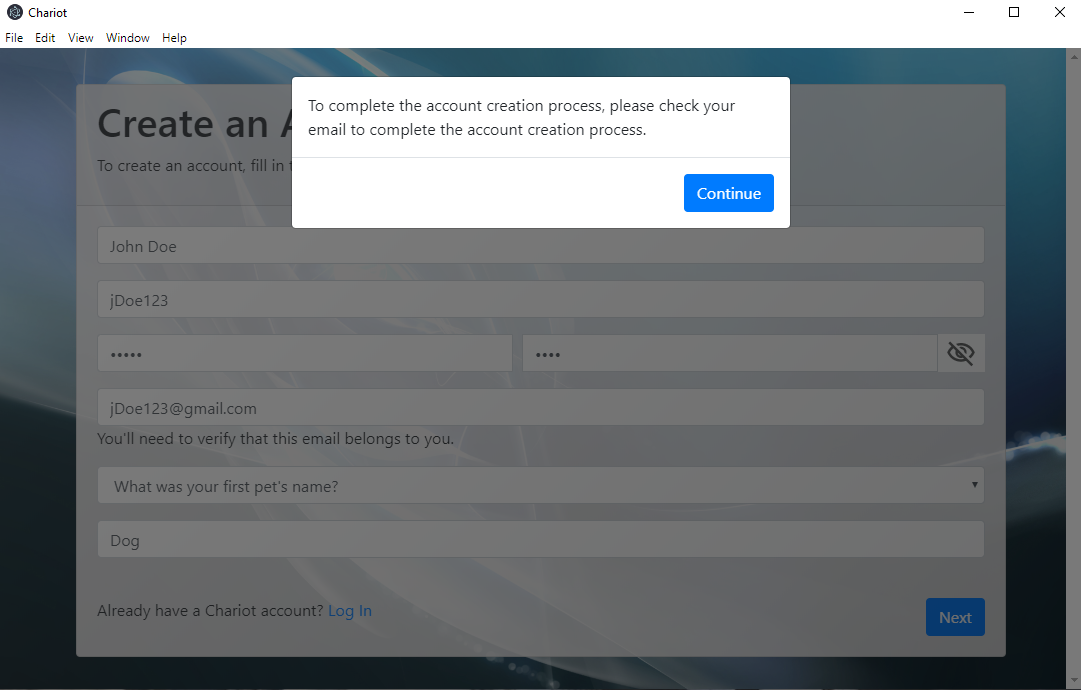
Usernames and email addresses must be unique between users, otherwise, an error message shall appear to let the user know that they most use a different username and/or email address.



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



**Figure 4.7**: Account credential confirmation screen

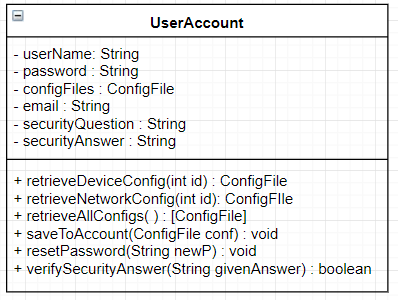


**Figure 4.8:** Account creation success screen

## 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

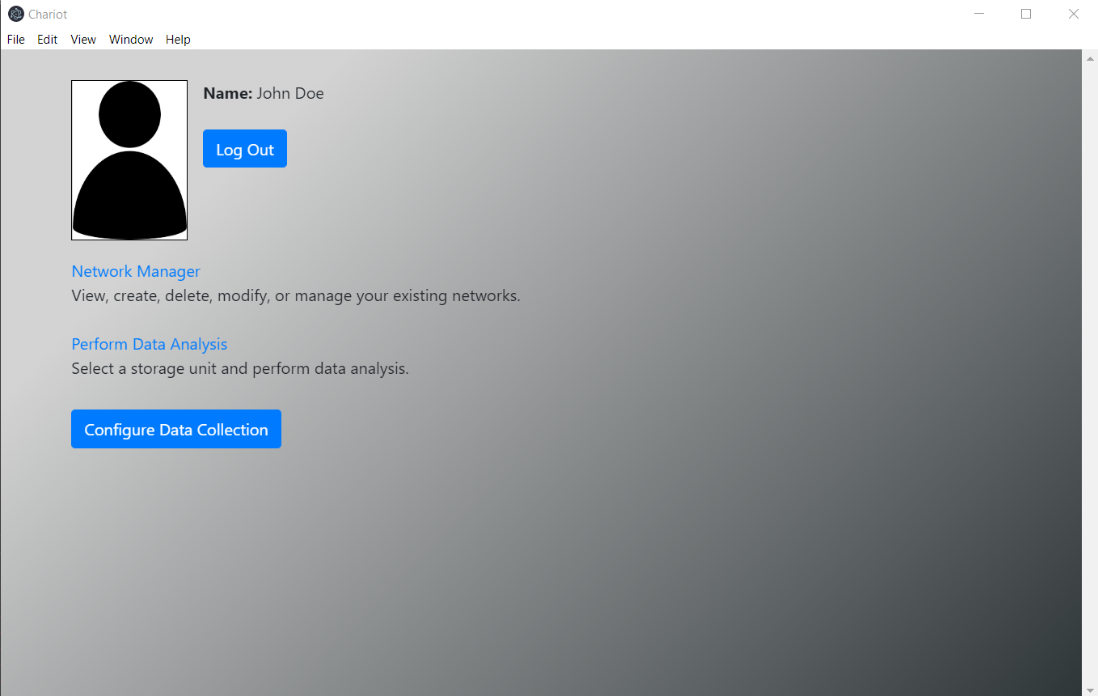
**Type**: Screen

**Description**:

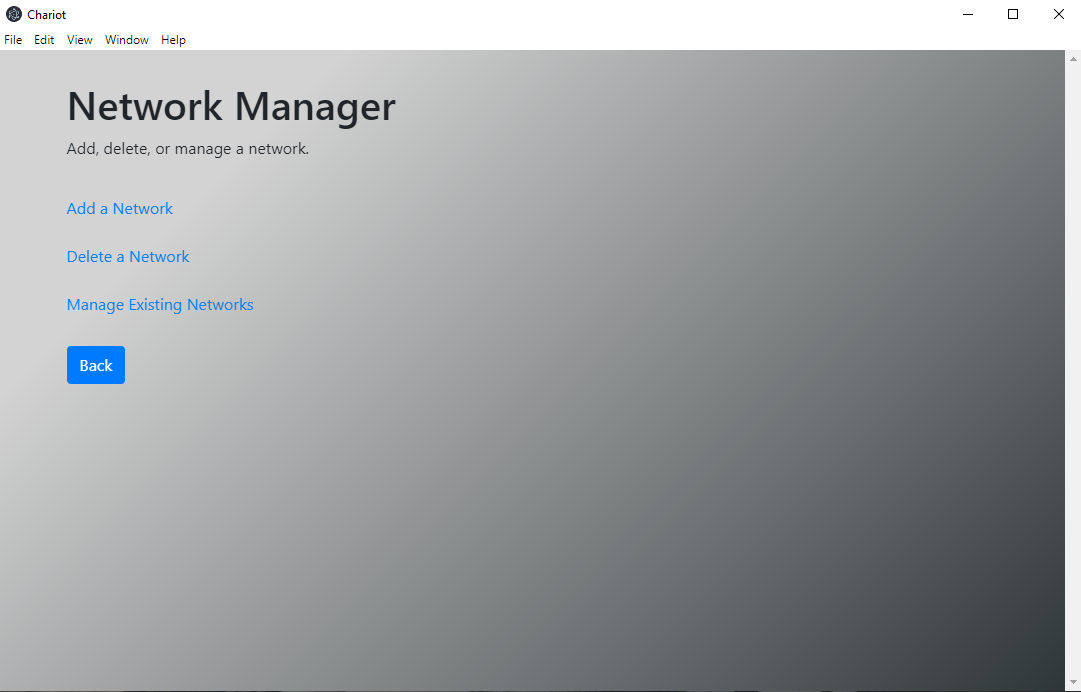
This screen appears after the user has successfully logged into their account. What it currently looks like is shown in Figure 4.9.

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 Welcome Page (Figure 4.9)

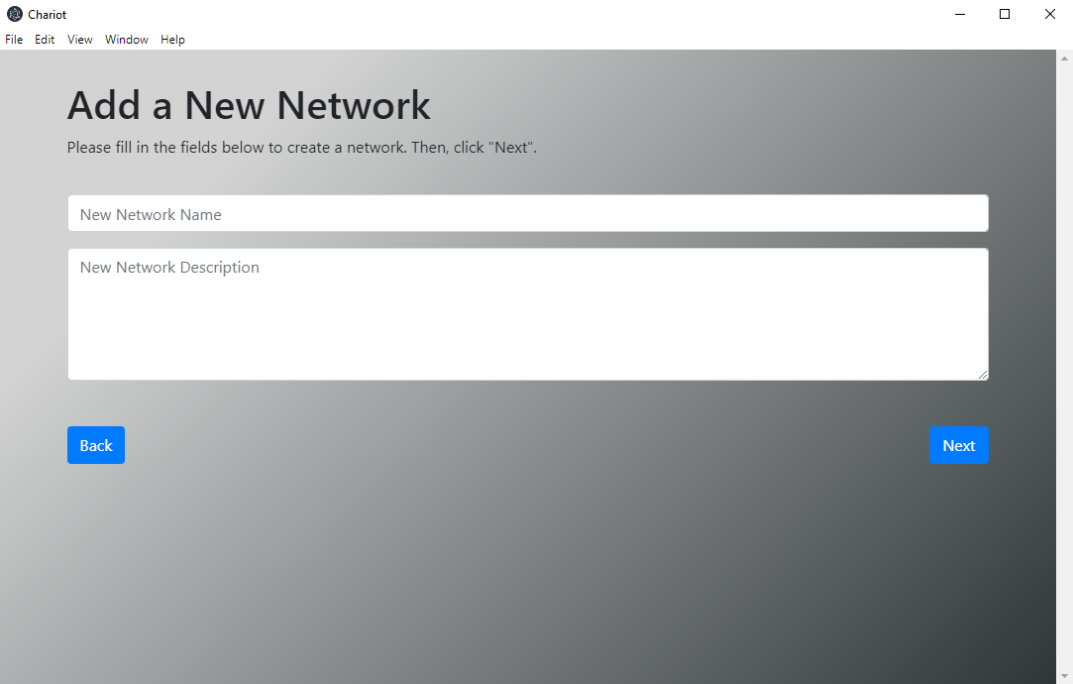
## DE5 – Create Network Screens

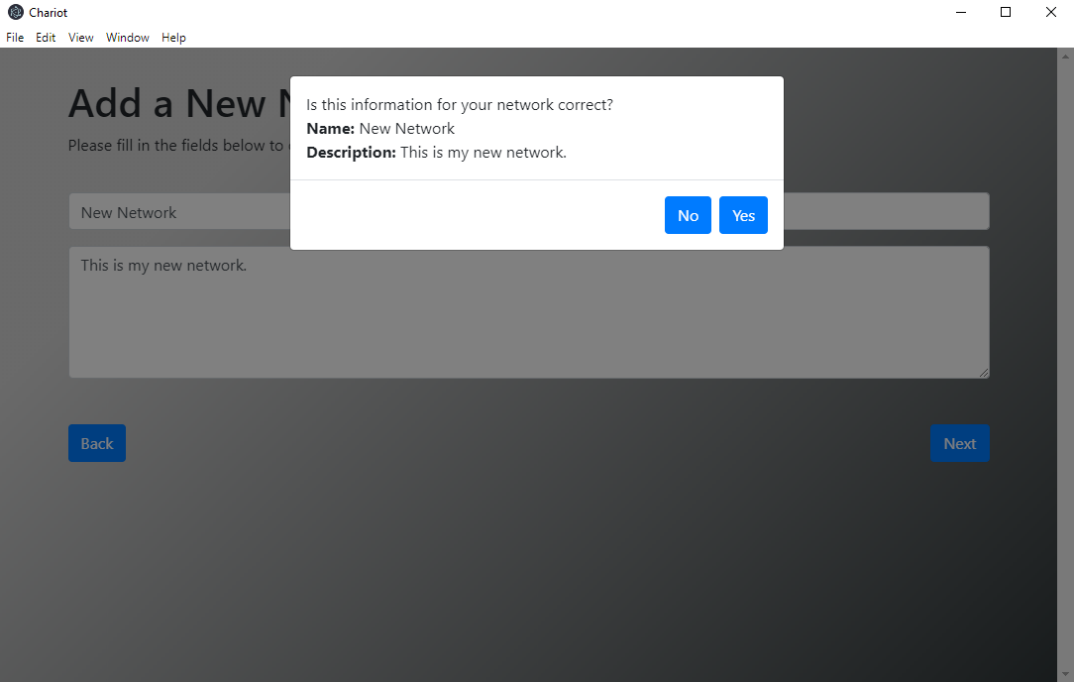
**Type**: Screen Sequence

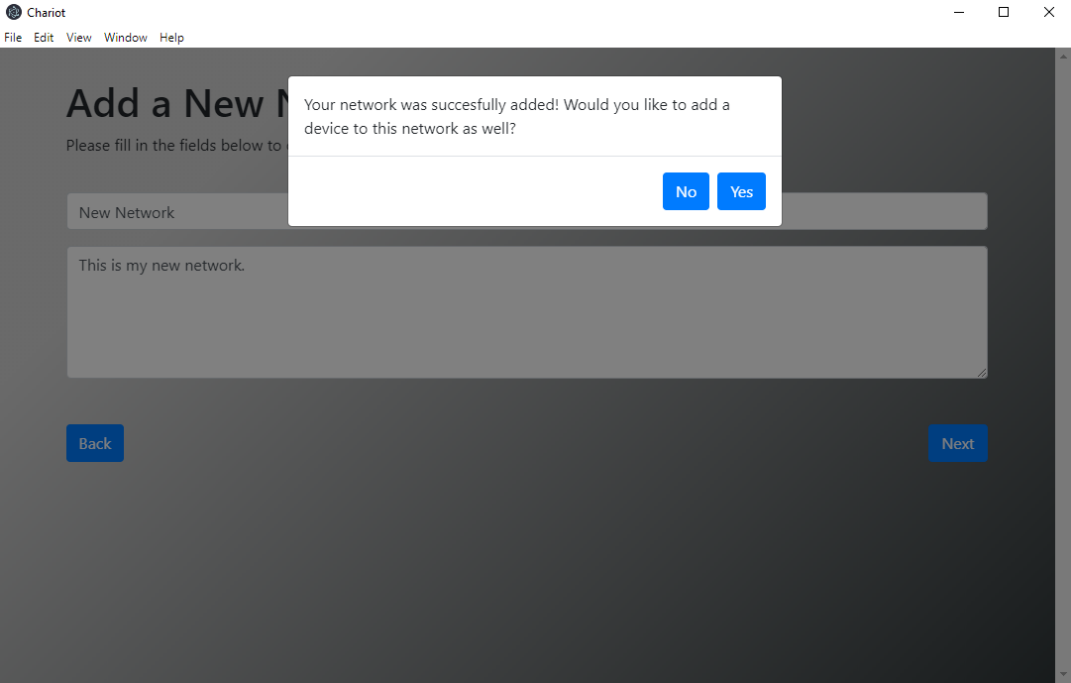
**Description**:

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.



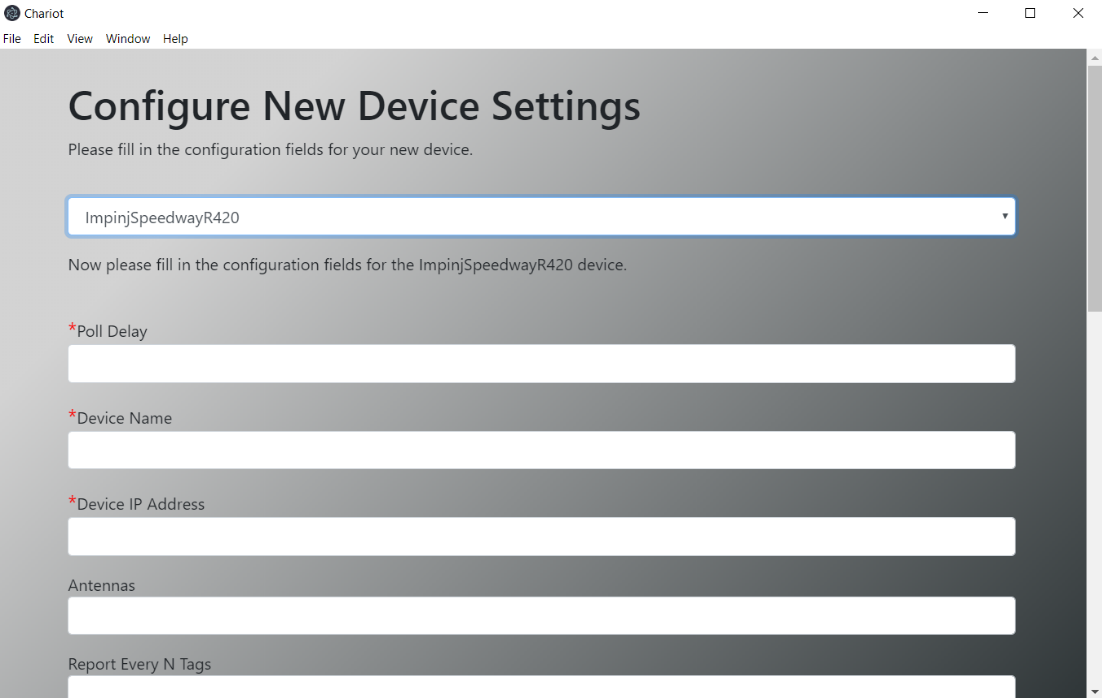




**Figure 4.11:** Screens used to add a new network to the user’s account



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



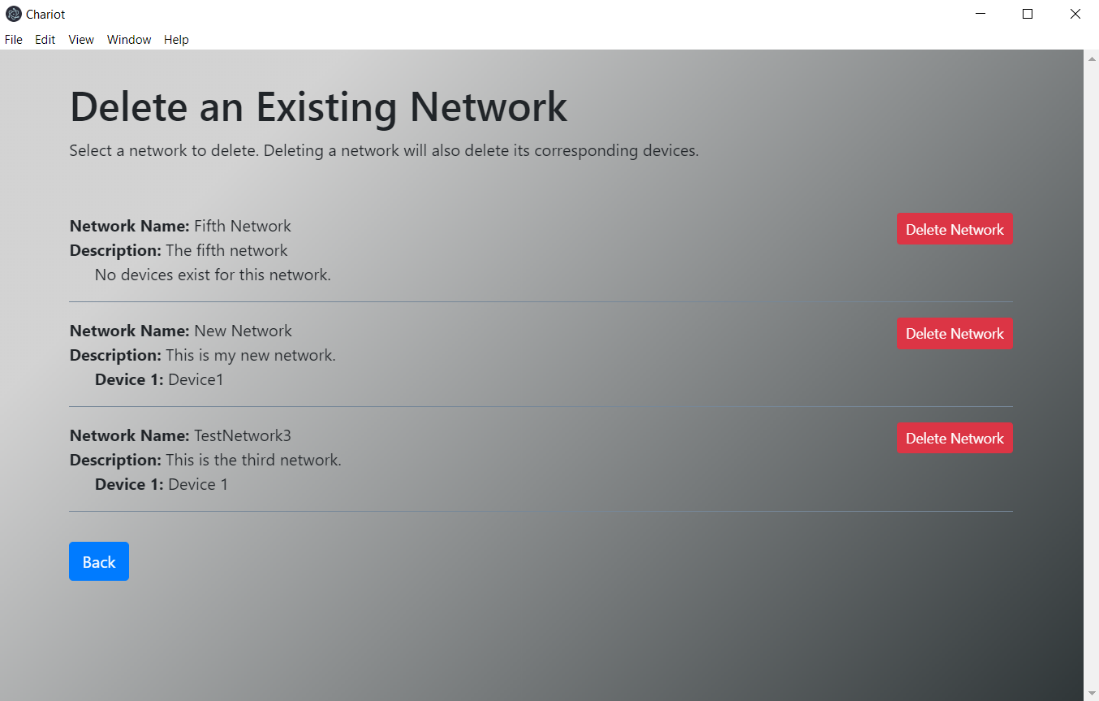
**Figure 4.13:** Screen allow the configuration of IoT device-specific parameters

## DE6 – Delete Network Screens

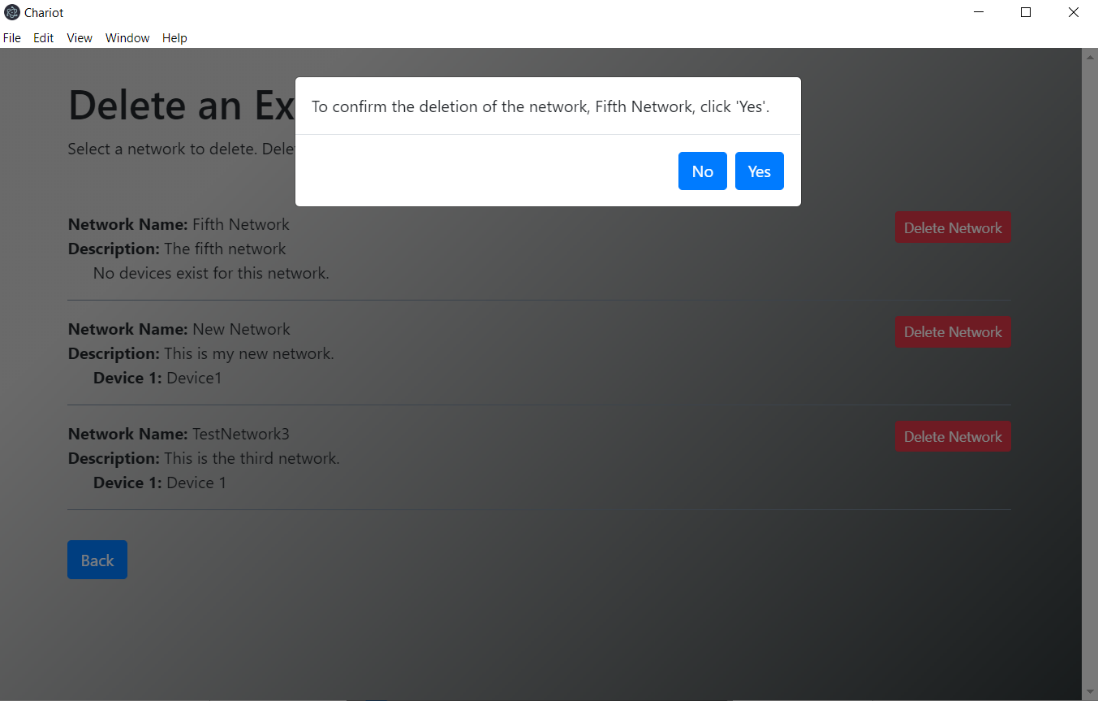
**Type**: Screen Sequence

**Description**:

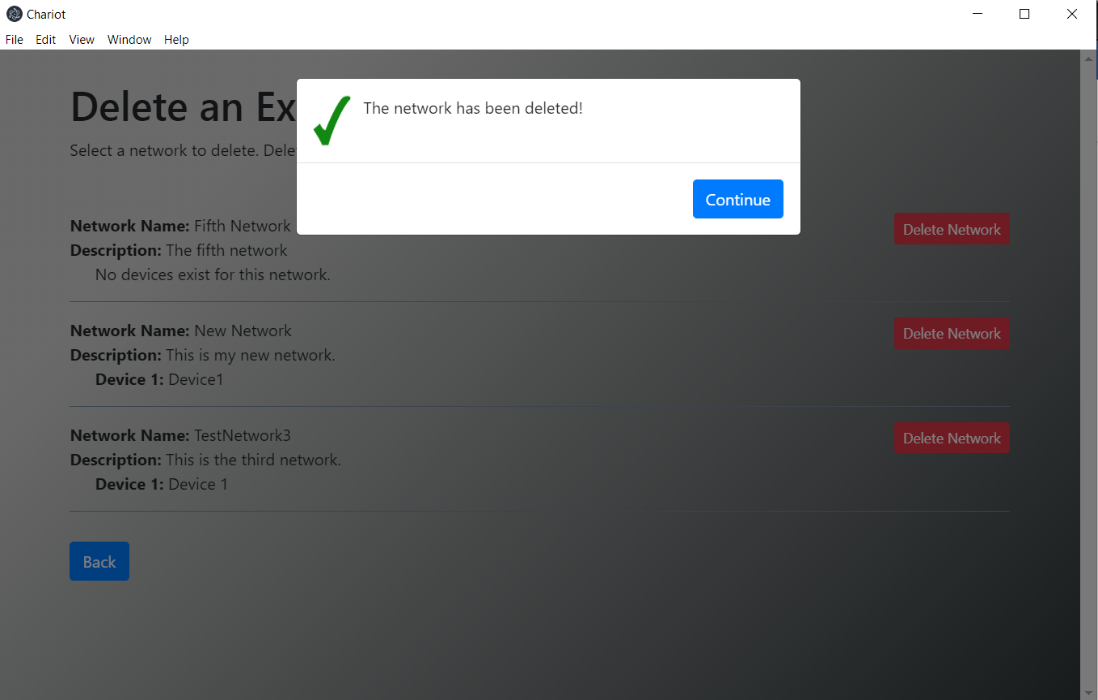
This collection of screens is available to the user when they choose to delete an existing network. The screens will guide the user through the process of deleting a network.



**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

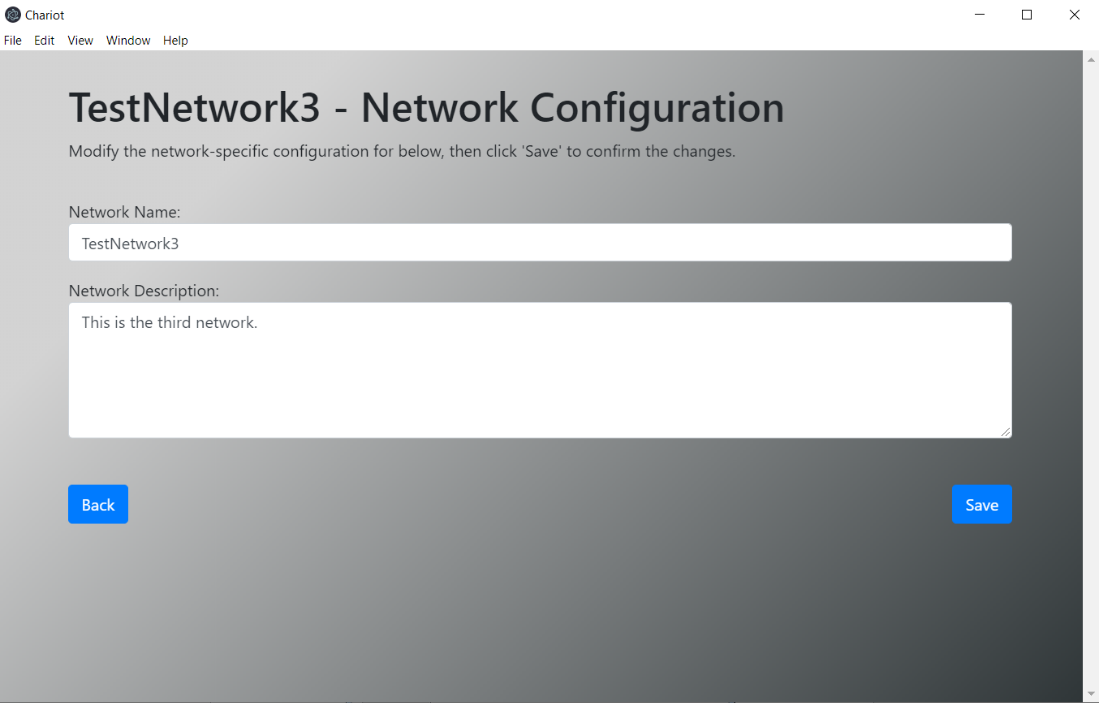
**Type**: Screen Sequence

**Description**:

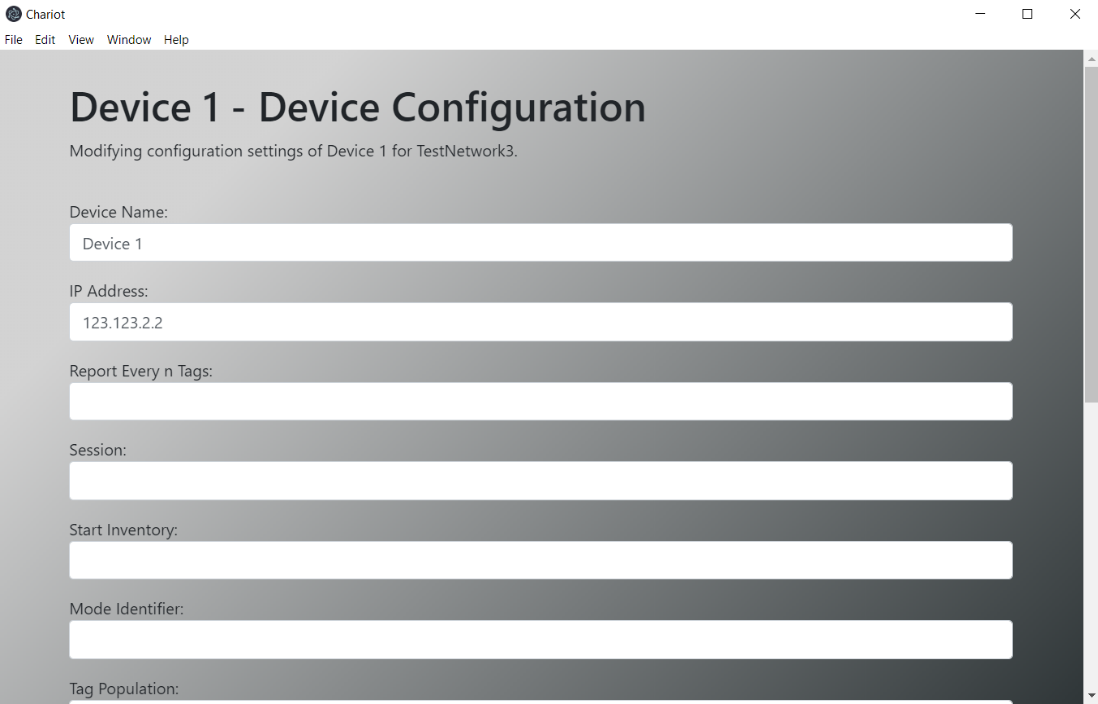
This collection of screens is available to the user when they choose to manage existing networks. By clicking on a network name, the user will go through a set of screens that will allow them to manage network specific settings such as name and description. By clicking on a device below a stated network, the user can edit that device’s device-specific settings or delete the device from the network. By clicking a network’s corresponding “Add Device” button, a user can add a new device to that network.



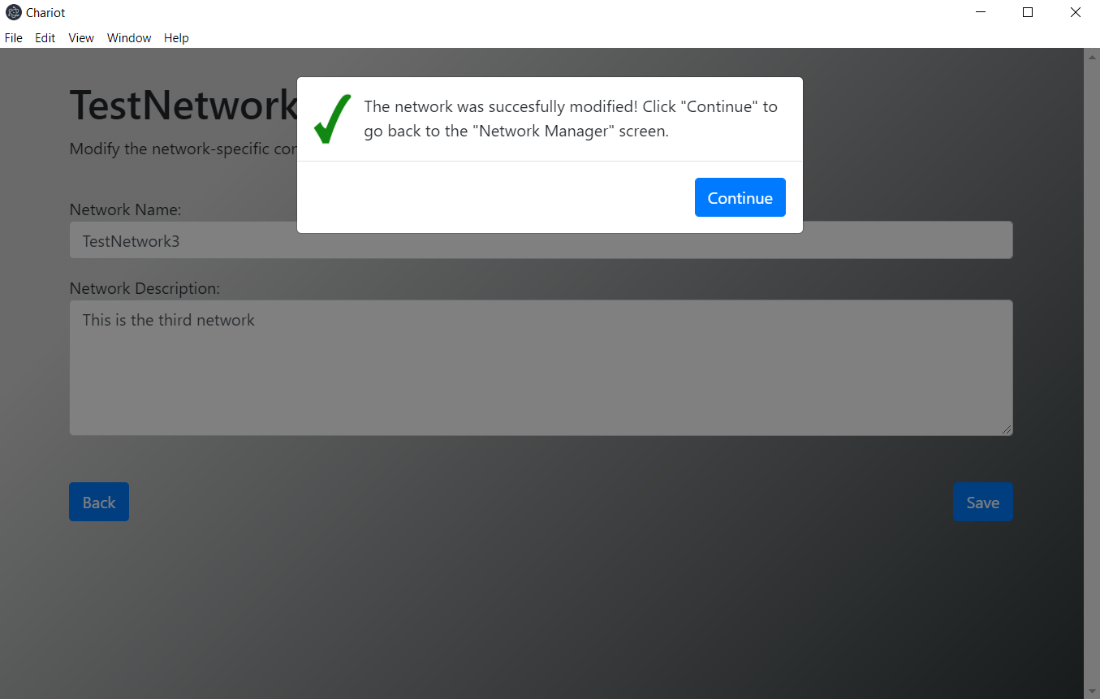
**Figure 4.17:** Screen where user chooses a network to modify its settings



**Figure 4.18:** Screen after user clicks on a network to modify, and is now presented with the network’s network-specific settings (in this case, TestNetwork3)



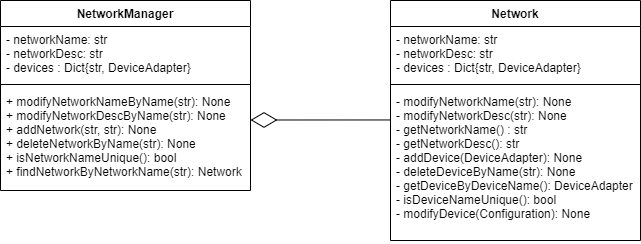
**Figure 4.19:** Screen after user clicks on a network’s device to modify (in this case, Device 1)



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

## 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. Both of these classes make use of the base abstract class Manager, in order to support code reuse since both Network and NetworkManager are responsible for managing collections.

**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. Note: this NetworkManager class inherits from the Manager class. In other words, the method addNetwork() makes use of the Manager’s \_addToCollection() method. Furthermore, the \_deleteFromCollection is called by deleteNetwork(). The reason for adding an additional method in the NetworkManager class (to add, delete, modify a key, and get from the collection) is to support a more conventional name such as addNetwork().

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

|  |  |
| --- | --- |
| **Method** | getAllNetworks |
| **Input:** | None |
| **Output:** | Array |
| **Description:** | Returns all the networks, with each index having a dictionary with the value being the network description. |

## 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. Note: this Network class inherits from the Manager class. In other words, the method addDevice() makes use of the Manager’s \_addToCollection() method. Furthermore, the \_deleteFromCollection is called by deleteDevice. The reason for adding an additional method in the Network class (to add, delete, modify a key, and get from the collection) is to support a more conventional name such as addDevice().

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

|  |  |
| --- | --- |
| **Method** | deleteDevice |
| **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 |

## DE8 – Add IoT Device Screens

**Type**: Screen Sequence

**Description**:

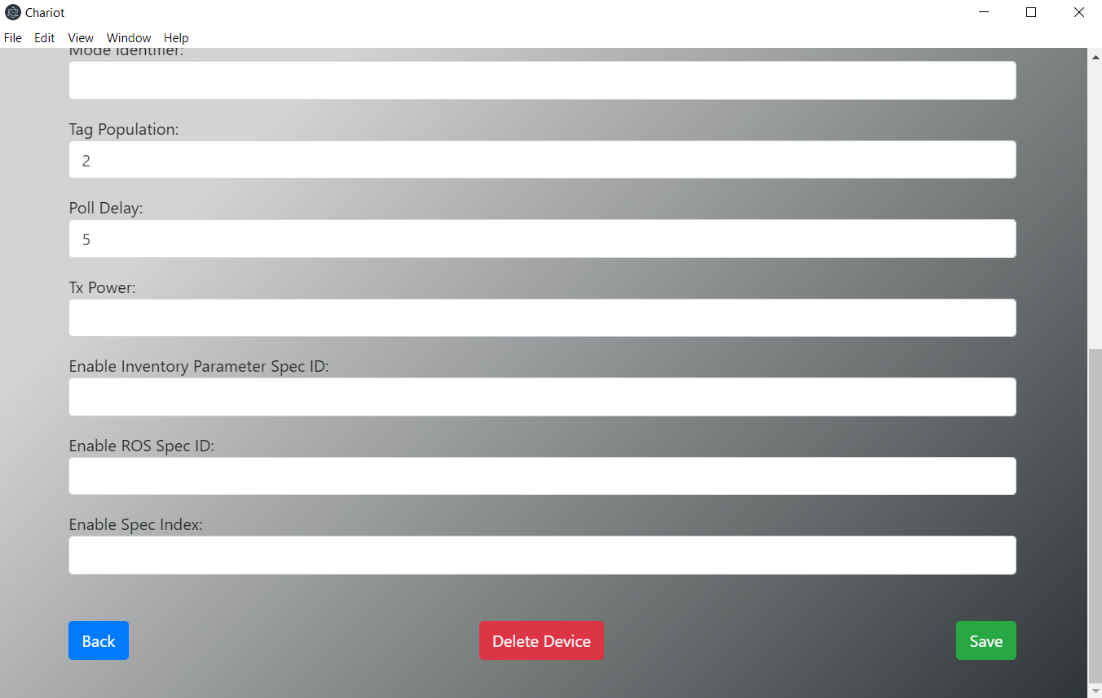
This collection of screens is available to the user when they choose to add a new IoT device to an existing network, or when they’re creating a new network. To get here, the user must first choose *Manage Existing Network* and then *Add an IoT Device,* or *Add a Network*, respectively. These screens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

## DE9 – Delete IoT Device Screens

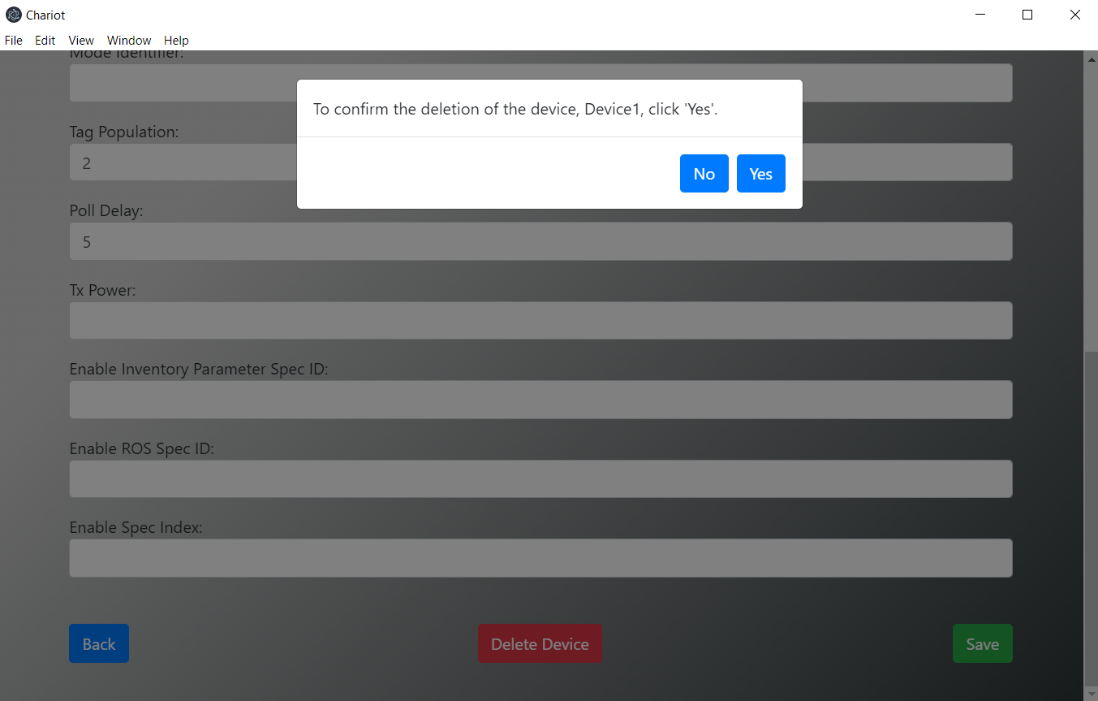
**Type**: Screen Sequence

**Description**:

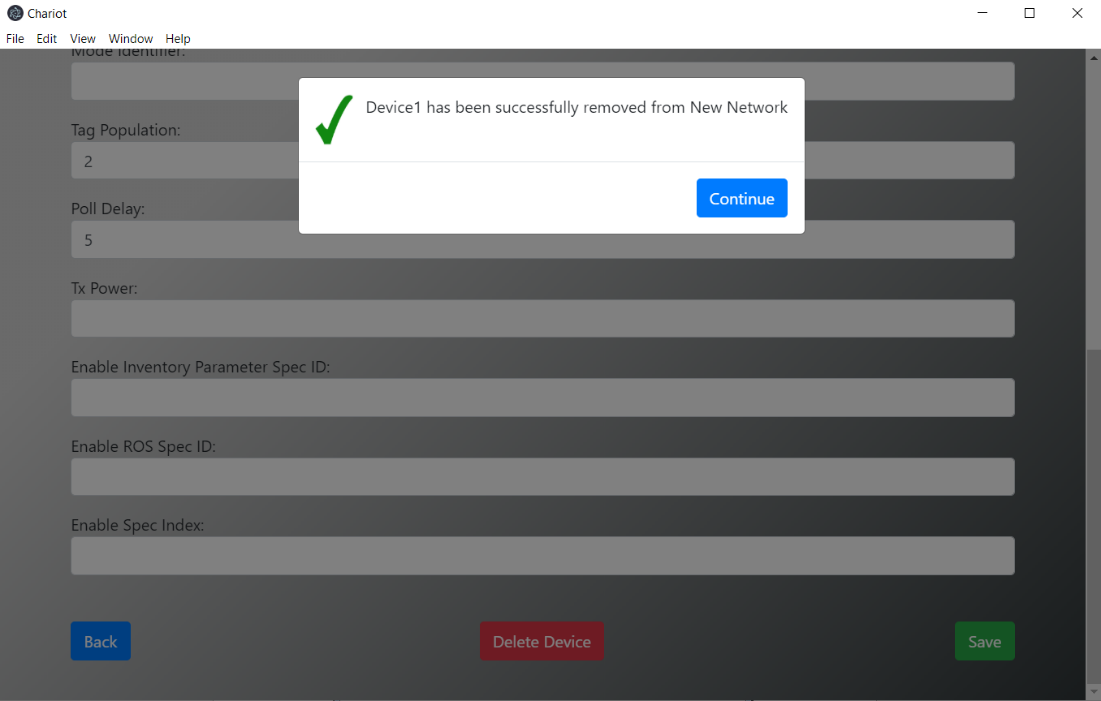
This collection of screens is available to the user when they want to delete an existing IoT device from an existing network. As mentioned in DE7, the user must first choose to *Manage Existing Network* and click a device name to see this collection of screens. These screens will let the user delete an IoT device from an existing network.



**Figure 4.21:** At the bottom of the Device Management screen will be a button that allows the user to delete a device from the corresponding network



**Figure 4.22:** User confirms the device they intend to delete through this modal



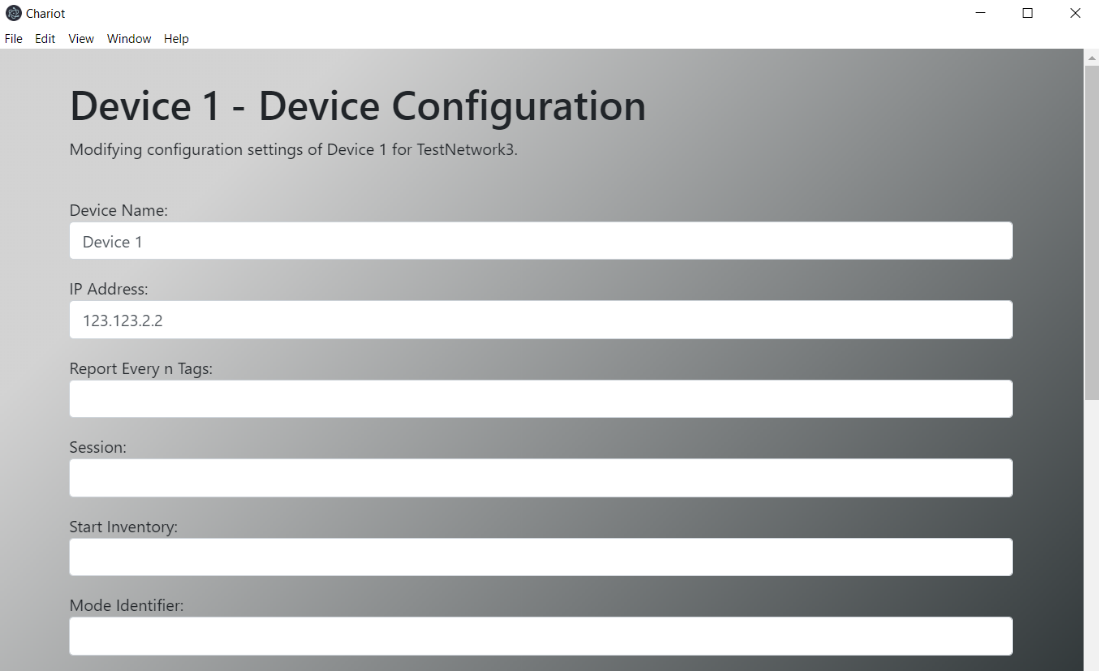
**Figure 4.23:** Successful deletion of network’s connected IoT device

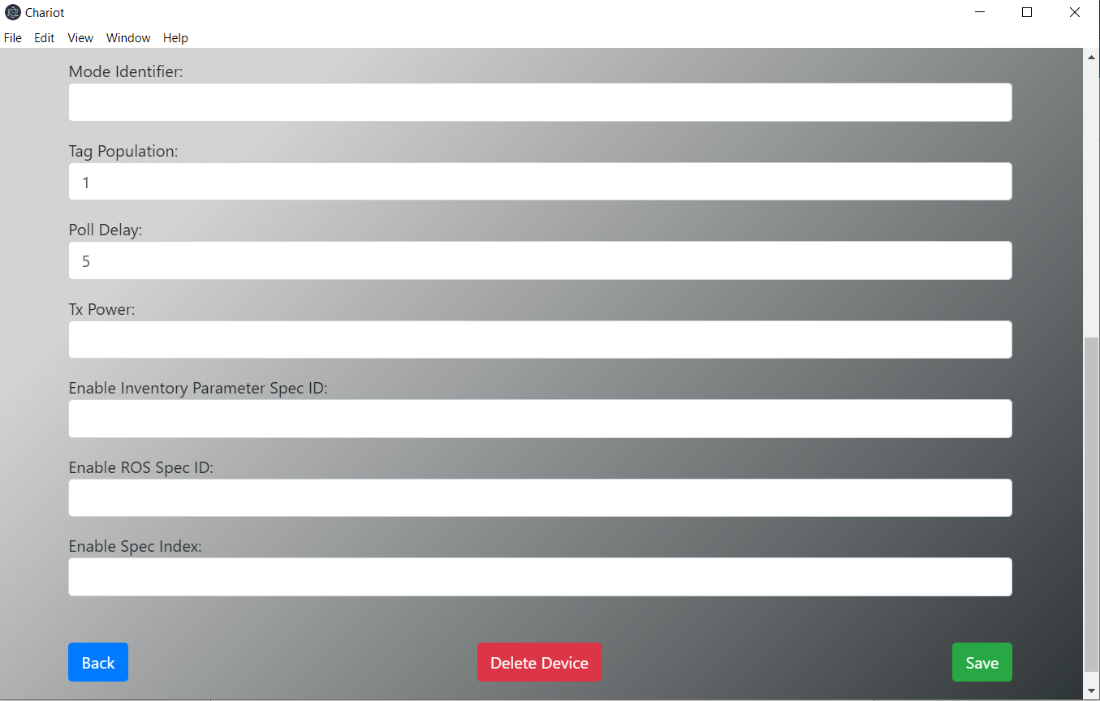
## DE10 – Manage Existing IoT Devices Screens

**Type**: Screen Sequence

**Description**:

This collection of screens is available to the user when they want to modify the settings of an existing IoT device on an existing network. As mentioned in DE7, the user must first choose to *Manage Existing Network* and click a device name to see this collection of screens. These screens allow the user to modify the IoT device’s device-specific settings. The user can modify device specific configuration settings via manual input or configuration file import.





**Figure 4.24:** Modifying an existing IoT device screen

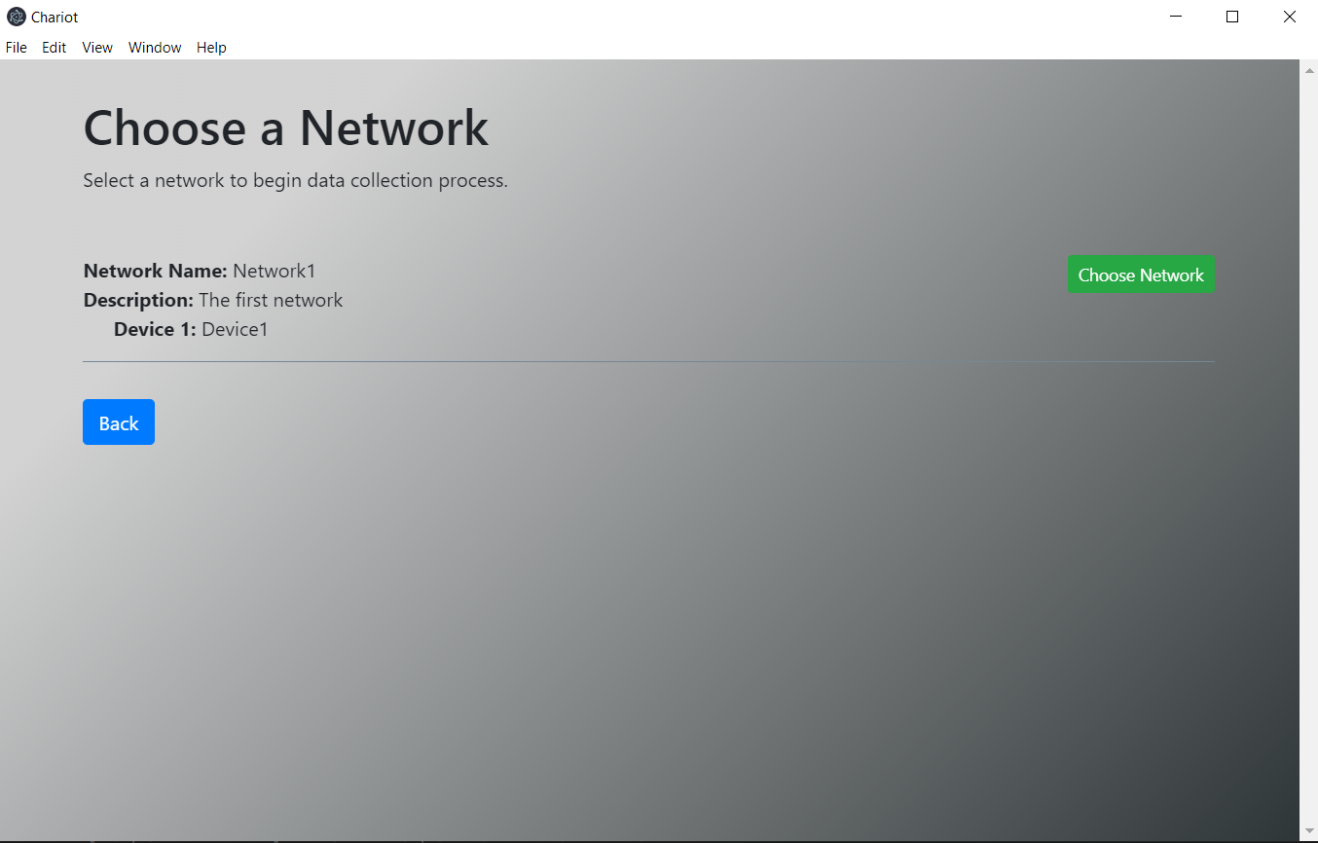
## DE11 – Data Collection Screens

**Type**: Screen Sequence

**Description**:

Before data is collected, the user must first choose the network. This provides the user with the devices where data will be collected from. After that, they must select the database to write to. Another screen will then appear where the user confirms that the data collection episode configuration is correct, and then the user can begin the data collection episode.

While collecting data, the user will have real-time visualizer so that they see the data that is being collected. 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.

**Figure 4.25:** Screen user will use to choose the network being used in the data collection episode



**Figure 4.26:** Data Collection Episode screen

## DE12 – Data Collection

The data collection is the main goal of Chariot and for maximum efficiency requires the use multi-threaded and multi-processor techniques to evenly distribute the potential load. Due to the primarily I/O bound nature of many of these procedures there can be a large amount of processes and threads used by Chariot without greatly affecting the overall system performance.

A screenshot of a cell phone

Description automatically generated

## DE12.1 – Data Collection Manager

**Type** : Code Entity

**Description:**

A manager concrete class that handles all available DataCollector objects in use by the system. It works in tandem with the NetworkManager that provides a network of devices for the DataCollector to use. This manager provides an interface from which users can store and use multiple DataCollector objects. All of the methods make use of private methods from the manager abstract class.

|  |  |  |
| --- | --- | --- |
| **Attributes** | **Type** | **Description** |
| collection | Dict[str, DataCollector] | Collection of all DataCollectors |

|  |  |
| --- | --- |
| **Method** | addCollector |
| **Input:** | collector: DataCollector |
| **Output:** | None |
| **Description:** | Adds a DataCollector to the collection |

|  |  |
| --- | --- |
| **Method** | deleteCollector |
| **Input:** | collectorName: str |
| **Output:** | None |
| **Description:** | Deletes a DataCollector from the collection |

|  |  |
| --- | --- |
| **Method** | getCollector |
| **Input:** | collectorName: str |
| **Output:** | DataCollector |
| **Description:** | Retrieves a DataCollector from the collection using it’s name |

|  |  |
| --- | --- |
| **Method** | getCollectors |
| **Input:** | json: bool |
| **Output:** | Dict[str, str] |
| **Description:** | Retrieves multiple DataCollectors at once, if json is set as True, it will return a JSONObject instead. |

|  |  |
| --- | --- |
| **Method** | replaceCollector |
| **Input:** | collectorName: str, newName: str |
| **Output:** | None |
| **Description:** | Replaces the key name for a DataCollector in the collection dictionary. |

## DE12.2 - Data Collector

**Type:** Code Entity

**Description**:

A Data Collector is in charge of performing a Data Collection Episode (DCE) for a network of devices. Using the provided DataCollectionConfiguration object, and optional Callable objects. It performs the task of creating child processes to perform the data collection and the cleanup and error handling for its child processes.

|  |  |  |
| --- | --- | --- |
| **Attributes** | **Type** | **Description** |
| \_config | DataCollectionConfiguration | Configuration for the Data Collector |
| \_devices | List[DeviceAdapter] | Collection of device adapters from a network |
| \_errorHandler | HandledThread | A thread that handles all Collection errors using the \_handleErrors method |
| \_errorQueue | Queue | A queue shared by all processes and threads of a data collection episode, used to store errors for use by the \_errorHandler thread |
| \_runLock | Lock | Lock used by the collector |
| \_onEnd | Optional[Callable] | An optional service that can be called by the Data Collector once the collection has ended. |
| \_onError | Optional[Callable] | On optional service that allows errors to be displayed for user intervention |
| \_running | bool | Used for checking when in a data collection episode |
| \_stopEvent | Event | Used to cause all worker processes to stop and terminate gracefully when the collection is over |
| \_stopTimer | Optional[Timer] | If a timer is set, the collection will end when the timer ends. |
| \_workers | List[DataCollectionWorker] | Used to test whether there is an ongoing DCE or not |
| \_workerProcesses | List[HandledProcess] | Used for timing how long the DCE has gone for |
| \_minPollDelay | float | Used in determining the minimum poll delay that can exist for any device on the network |

|  |  |
| --- | --- |
| **Method** | \_\_init\_\_ |
| **Input:** | configuration: DataCollectionConfiguration, onEnd: Optional[Callable], onError: Optional[Callable] |
| **Output:** | DataCollector |
| **Description:** | Initializes a Data Collector object. Many of the attributes are left in a basic state, such as empty lists, and will be formally used later by other methods. |

|  |  |
| --- | --- |
| **Method** | \_handleerrors |
| **Input:** | None |
| **Output:** | None |
| **Description:** | This is the error handler for the entire Data Collection Episode. It will read the error queues of all workers, which are populated by errors generated by the different worker devices, threads, and the workers themselves. It will resolve possible errors while allowing the rest of the data collection continue unimpeded. |

|  |  |
| --- | --- |
| **Method** | startCollection |
| **Input:** | None |
| **Output:** | None |
| **Description:** | Begins the data collection episode by evenly dividing available network devices to new DataCollectionWorker objects, and then creating HandledProcess objects for each DataCollectionWorker object. It will then lock the network and database to prevent tampering while the collection process happens, starting all necessary processes and threads. |

|  |  |
| --- | --- |
| **Method** | \_stopCollection |
| **Input:** | calledFromTimer: bool |
| **Output:** | None |
| **Description:** | Stops the data collection episode by setting the \_stopEvent object and then waiting for all processes and threads to join. It then disconnects from the database, clears any list objects and releases \_runLock. |

|  |  |
| --- | --- |
| **Method** | stopCollection |
| **Input:** | args: Tuple |
| **Output:** | None |
| **Description:** | Public method allowing a user to stop the data collection episode at any time. Calls the \_stopCollection method to actually stop collection. |

## DE12.3 – Data Collection Worker

**Type:** Code Entity

**Description:**

A Data Collection Worker is in charge of communicating with multiple device adapter producer threads, and collecting their data for storage and other uses. It will utilize producer, consumer, output, and stop threads to perform different tasks asynchronously.

|  |  |  |
| --- | --- | --- |
| **Attributes** | **Type** | **Description** |
| \_devices | List[DeviceAdapter] | Collection of device adapters from the network |
| \_dataQueue | Queue | Queue in which data from all devices collects to. |
| \_errorQueue | Queue | Queue for the collection and handling of errors received from producer threads |
| \_consumerThreads | List[HandledThread] | Collection of consumer threads that receives data from an adapter’s dataQueue |
| \_minPollDelay | float | Used in determining the minimum poll delay that can exist for any devices |
| \_outputDelay | float | Delay between outputting data to prevent the \_outputThread thread from starving consumer Threads from using the \_dataQueue object |
| \_consumeTimeout | float | Limit for how long the consumer Threads will wait to get data from device data queues. |
| \_running | Bool | Used to test whether there is an ongoing DCE or not |
| producerThreads | Dict[str, HandledThread] | Collection of producer threads the put data in an adapter’s dataQueue |
| \_outputHooks | Set[Callable] | Services that are obtaining the data retrieved from devices |
| \_outputThread | Optional[HandledThread] | A thread that collects data from different devices |
| \_stopThread | Optional[HandledThread] | A thread that exists to stop all other threads when the stop event is set from the parent DataCollector process |

|  |  |
| --- | --- |
| **Method** | \_\_init\_\_ |
| **Input:** | devices: List[DeviceAdapter], minPollDelay: float |
| **Output:** | DataCollectionWorker |
| **Description:** | Initializes a data collection worker to be used in a data collection episode. |

|  |  |
| --- | --- |
| **Method** | \_consumeDataFromDevice |
| **Input:** | deviceIdx: int |
| **Output:** | None |
| **Description:** | Collects data from a single device, and adds the collected data to the \_dataQueue. |

|  |  |
| --- | --- |
| **Method** | \_consumeDataFromDevices |
| **Input:** | startIdx: int, numDevices: int |
| **Output:** | None |
| **Description:** | Collects data from multiple devices, adding their data to the \_dataQueue. If only one device is present, this method calls \_consumeDataFromDevice. Used by all consumer threads. |

|  |  |
| --- | --- |
| **Method** | outputData |
| **Input:** | None |
| **Output:** | None |
| **Description:** | Takes data from the \_dataQueue and sends it to the database and other connected output hooks. |

|  |  |
| --- | --- |
| **Method** | start |
| **Input:** | stopEvent: Event |
| **Output:** | None |
| **Description:** | Begins a data collection episode for the DataCollectionWorker. All devices are assigned to a single producer HandledThread to collect data from an IoT device. All available devices are evenly distributed among consumer HandledThreads to be collected. An outputThread that performs the \_outputData method, and a \_stopThread\_that performs the \_waitForStopEvent are finally created before all threads are started. |

|  |  |
| --- | --- |
| **Method** | addDeviceDuringDCE |
| **Input:** | device: DeviceAdapter, |
| **Output:** | None |
| **Description:** | Adds a device to the DataCollectionWorker while it is in a DCE. Creates a producer HandledThread for the device, and if it is not a device reconnecting, it will create a consumer HandledThread to collect from the device. |

|  |  |
| --- | --- |
| **Method** | \_waitForStopEvent |
| **Input:** | event: Event |
| **Output:** | None |
| **Description:** | Used by the \_stopThread to obtain the stop event from the DataCollector parent process, signalling the DataCollectionWorker to stop the collection. |

|  |  |
| --- | --- |
| **Method** | stop |
| **Input:** | \*args: Tuple |
| **Output:** | None |
| **Description:** | Ends the data collection by using the stopDataCollection method on all devices. The DataCollectionWorker then joins with all threads waiting for them to end before doing final cleanup. |

## 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 Adapters and Device Adapter Factories

A screenshot of a social media post

Description automatically generated

## DE14.1 – Device Adapter Factory

**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.

|  |  |
| --- | --- |
| **Method** | getInstance |
| **Input:** | config: DeviceConfiguration |
| **Output:** | DeviceAdapter |
| **Description:** | Creates a DeviceAdapter object based on the specifications provided by the config. |

## 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.

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** |
| \_config | DeviceConfiguration | A DeviceConfiguration object that has all the settings needed for connecting and interacting with a specific IoT device |
| connected | bool | A boolean to test if connected to the associated device. |
| \_dataQueue | Queue | Where all data collected from a device is stored for collection by other objects. |
| \_inCollectionEpisode | bool | A Boolean to test if the adapter is currently part of a DCE |

|  |  |
| --- | --- |
| **Method** | startDataCollection |
| **Input:** | void |
| **Output:** | void |
| **Description:** | Begins a DCE with the device by connecting to the device, and then using the device adapter’s specific \_startDataCollection method. |

|  |  |
| --- | --- |
| **Method** | stopDataCollection |
| **Input:** | void |
| **Output:** | void |
| **Description:** | Gracefully ends a DCE with the device |

|  |  |
| --- | --- |
| **Method** | connect |
| **Input:** | reconnect: bool |
| **Output:** | None |
| **Description:** | Performs the necessary procedures to capture data from a device. This only happens if the device is not already connected, or if it is a reconnect. This prevents and issues that may occur from connecting multiple times. |

|  |  |
| --- | --- |
| **Method** | disconnect |
| **Input:** | None |
| **Output:** | None |
| **Description:** | Gracefully closes the connection to the device. |

|  |  |
| --- | --- |
| **Method** | getDataQueue |
| **Input:** | None |
| **Output:** | Queue |
| **Description:** | Returns \_dataQueue so other objects may use the collected data. |

|  |  |
| --- | --- |
| **Method** | reportData |
| **Input:** | data: Dict[str, JSONObject] |
| **Output:** | None |
| **Description:** | Formats the raw collected data from a device to a format usable within Chariot, and then puts the data into \_dataQueue. |

## 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 databases. 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 – Manager**

The manager class is an abstract class that defines common behavior necessary to manage a collection. This base abstract class is extended by the NetworkManager, Network, DatabaseManager, and DataCollectionManager. The reason for adding this base abstract class is to follow the coding principle of code-reuse to enable easier modification. This class uses generics in order to be able to support multiple objects.

|  |  |
| --- | --- |
| **Method** | \_addToCollection |
| **Input:** | item: T |
| **Output:** | None |
| **Description:** | Adds an object to the dictionary, in which the key is the item’s id and the value depends on the object added to the collection. |

|  |  |
| --- | --- |
| **Method** | \_deleteFromCollection |
| **Input:** | itemId: str |
| **Output:** | None |
| **Description:** | Deletes an object on the dictionary, which is the key value in the dictionary. |

|  |  |
| --- | --- |
| **Method** | \_retrieveFromCollection |
| **Input:** | itemId: str |
| **Output:** | T (type generic) |
| **Description:** | Returns, if found, the object in the collection which is defined by the itemId parameter. If the itemId is not found in the collection, a custom NameNotFoundError is thrown. |

|  |  |
| --- | --- |
| **Method** | \_modifyNameInCollection |
| **Input:** | toFind: str, newName: str |
| **Output:** | None |
| **Description:** | When modifying a key in a dictionary collection, one must also delete the old key to ensure data accuracy. This method handles changing the key value and deleting the old item. |

## DE18 – 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

## DE18.1 – 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. |

## DE18.2 – 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 |

## DE18.3 – 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. |

## DE18.4 – 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. |

## DE18.5 – 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. |

## DE18.6 – 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 retrieving information about a network and the devices on that network. Done correctly, a 200-success code is returned. |

## DE18.7 – 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. |

## DE18.8 – 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. |

## DE18.9 – 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. |

## DE18.10 – 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. |

## DE18.11 – 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. |

## DE18.12 – 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/database/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. |

## DE18.13 – Get all the information about supported databases

**Type**: API

**Description:**

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

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

## DE18.14 – Get information about a specific database configuration

**Type**: API

**Description:**

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

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

## DE18.15 – Get information about a configured database

**Type**: API

**Description:**

This endpoint is responsible for retrieving the configuration information of a specified database configuration. This GET request expects that “dbId” be in the URL. This is so that the configuration information of a created database configuration is returned . The returned content is JSON content that includes the database configuration id and the fields specific to that database 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 database that has already been configured.

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

## DE18.16 – Create a database configuration

**Type**: API

**Description:**

This endpoint is responsible for creating a database configuration. This POST request at minimum expects that “dbId” be in the payload. Additional information will be dependent on the database configuration being created. Devices Ids of a database configuration must be unique so there are possibilities for 400 errors

|  |  |
| --- | --- |
| **Method** | createDatabaseConfiguration |
| **Input:** | Endpoint: Chariot/api/v1.0/database  Payload specifying at least“dbId”, additional information is dependant upon what database configuration is to be created. |
| **Output:** | 200 success |
| **Description:** | This POST request is responsible for creating a supported database configuration. |

## DE18.17 – Modifying a database configuration

**Type**: API

**Description:**

This endpoint is responsible for modifying a database configuration. This PUT request at minimum expects that “dbId” be in the payload. Additional information will be dependent on the database configuration being modified. DbIds in a collection must be unique so there are possibilities for 400 errors

|  |  |
| --- | --- |
| **Method** | modifyDatabaseConfiguration |
| **Input:** | Endpoint: Chariot/api/v1.0/database  Payload specifying at least “dbId”, additional information is dependant upon what database configuration is to be created. |
| **Output:** | 200 success |
| **Description:** | This POST request is responsible for creating a supported database configuration. |

## DE18.12 – Deleting a database configuration

**Type**: API

**Description:**

This endpoint is responsible for deleting a database configuration. This DELETE request at expects “DbId” be in the URL. If the specified database configuration does not exist, then a 400 error is returned.

|  |  |
| --- | --- |
| **Method** | deleteDatabaseConfiguration |
| **Input:** | Endpoint: Chariot/api/v1.0/database?dbId=  URL specifying a“dbId”. This is used to delete a specific database configuration. |
| **Output:** | 200 success |
| **Description:** | This DELETE request is responsible for deleting a database configuration. |

**Appendix**

A) System Class Diagram

## A close up of a map Description automatically generated

**Figure A:** 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>