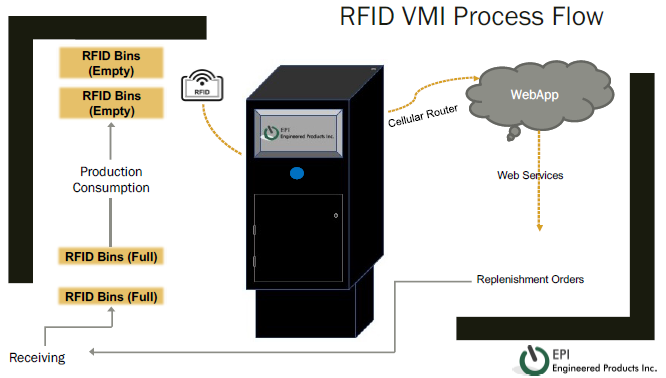
RFID Bin Capture Application

Design Specification

V1.2

20-April-2018

# System Description



## RFID Reader Application

Inside the box there shall be an Impinj Speedway reader (Speedway R120). A custom application on the reader will scan for RFID tags continuously. The RFID tag data that is read and a corresponding date/time stamp will be cached on the reader in non-volatile memory. Periodically, the reader will connect to a webservice via its ethernet connection and upload a list of tags and timestamps. In the event of a connection or other failure to upload data to the webservice, the reader will retry at its normal periodic interval indefinitely. The reader will only cache a limited number of scan records (not to exceed 256) and when the cache is full, the reader will overwrite the oldest scans without uploading them to the webservice. If the cache is still empty, the reader will still connect to the API and pass a payload with no tag reads which will act as a ‘heartbeat’.

The application will also toggle the reader’s GPIO lines to turn on an indicator light for 1 second when the a tag is read.

## Webservice

Hosted in IIS, there will be a webservice that exposes a RESTful API. This API will handle incoming connections from RFID readers. The API will expose only one interface which will be a POST operation where the client will pass a JSON object with an array of RFID scan data, and the reader’s unique ID (likely its MAC address, serial number, or some other indicator) as the payload. The webservice will validate the JSON, add the scan data to a SQL database, and respond to the caller with a status for each element in the array (success or failure). If an EPC is received that has been received previously in the last 36 hours, the API will not add another record to the database.

## Webapp

In addition to the webservice, there shall be a webapp also hosted in a local instance of IIS. Functionally, the webapp allows a user view tabular data from the SQL database. A user will be able to select a reader or location (where location is just an alias for a reader) and view a table of all tag reads, sorted by timestamp descending. A user will be able to also view a table of all readers and their ‘last seen’ time stamp. This is the last time that the reader had connected to the webservice. Finally, there will be a section where the user can create the readerId -> location alias for each reader.

## Database

The database will be an MS SQL database. All operations with the database should be non-blocking to allow other applications to have read-only access the data.

# Technology Stack Summary

## Reader Application

The reader application will be built as one or more Linux applications or scripts using C++ and bash. This application and its dependencies will be compiled as a customer application for Impinj Speedway readers.

## Webservice

The webservice will expose a RESTful API and will be hosted in IIS. The application will be written in C#.

## Database

The database will be an Azure-hosted SQL database.

## Webapp

The webapp will also be hosted in Azure. The application will be a reactive AngularJS application.

# Reader->Webservice Communication

The RFID reader will perform an HTTP POST to the webservice’s RESTful API. The payload will include the reader’s unique ID (string) which is provided by the reader (and not the backend), and 0-n tag reads. A typical payload might look like this:

|  |
| --- |
| {  "readerId": "00:0c:29:e0:40:56",  "tagCount": 3,  "tags": [{  "readId": 1234567890,  "epc": "ABCDEF1234567890",  "timestamp": 12345687890,  "antenna": 0  },  {  "readId": 1234567891,  "epc": "ABCDEF1234567890",  "timestamp": 12345687890,  "antenna": 0  },  {  "readId": 1234567892,  "epc": "ABCDEF1234567890",  "timestamp": 12345687890,  "antenna": 0  }  ]  } |

|  |  |  |  |
| --- | --- | --- | --- |
| **Variable** | **Data Type** | **Required?** | **Description** |
| readerId | String | Required | Unique identifier for the client device |
| tagCount | Int | Required | Number of tag reads in this payload (0 – 32767) |
| tags | Array | Optional | Array of tag read data |
| readId | Int | Required | Unique (in this payload) identifier for each tag read |
| epc | String | Required | EPC value from tag read |
| timestamp | Int | Required | Time stamp of tag read (UNIX time) |
| antenna | Int | Required | Antenna port of the read (0-32767) |

The webservice will receive this payload and perform the following actions:

1. Add record to the Readers table (readerId, current datetime (UNIX time), tagCount)
2. For each tag in tags[]
   1. Create a record in the Scans table (readerId, epc, timestamp, antenna)
3. Send response to caller

The webservice JSON response payload will be similar to the following:

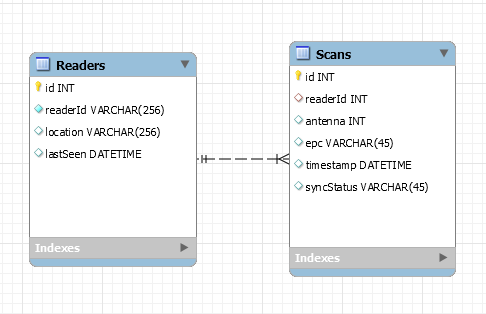
|  |
| --- |
| {  "status": "success",  "message": "",  "tagCount": 3,  "tags": [{  "readId": 1234567890,  "status": "success",  "message": ""  },  {  "readId": 1234567890,  "status": "success",  "message": ""  },  {  "readId": 1234567890,  "status": "success",  "message": ""  }]  } |

|  |  |  |  |
| --- | --- | --- | --- |
| **Variable** | **Data Type** | **Required?** | **Description** |
| status | String | Required | Status of the total operation (success | error) |
| message | String | Required | If there was an error, this should contain a useful error message |
| tagCount | Int | Required | Number of tag reads processed in this payload (0 – 32767) |
| tags | Array | Optional | Array of tag update statuses |
| readId | Int | Required | Unique (in this payload) identifier for each tag read |
| status | String | Required | Status of this tag’s processing (success | error | skipped) |
| message | String | Required | If status is error, this should be a useful error message. If status is skipped, this should be an explanation for why the tag was skipped (duplicate tag, etc.) |

In addition to the creation of the webservice, a test script (such as for use with Postman) should be created to simulate a reader in order to test the API webservice without using a reader.

# Database Schema

The database will have 2 tables as shown below.



## Readers Table

|  |  |  |  |
| --- | --- | --- | --- |
| **Column Name** | **Data Type** | **Options** | **Description** |
| id | Unsigned int | Primary Key, Not Null, Unique, Unsigned, Auto-increment | System-wide unique identifier |
| readerId | Varchar(256) | Not Null, Unique | Reader’s unique ID, as provided by the reader |
| location | Varchar(256) |  | User-defined alias for this reader |
| lastSeen | Datetime |  | The last time this reader has communicated with the webservice |

## Scans Table

|  |  |  |  |
| --- | --- | --- | --- |
| **Column Name** | **Data Type** | **Options** | **Description** |
| id | Unsigned int | Primary Key, Not Null, Unique, Unsigned, Auto-increment | System-wide unique identifier |
| readerId | Int | Not Null, Unsigned, Unique | Foreign key link to the Id column in the Readers table of the reader that submitted this tag scan |
| antenna | Int | Unsigned | Antenna value associated with this tag scan |
| epc | Varchar(45) |  | EPC value associated with this tag scan |
| timestamp | Datetime |  | Time stamp value associated with this tag scan |
| syncStatus | Varchar(45) |  | Used by another system. This column should be set to NULL when the record is created. |

## Stored Procedures

A number of stored procedures are included that facilitate the application. Additionally, there is a stored procedure available to make reading information from the database easier. The signature of the procedure is as follows:

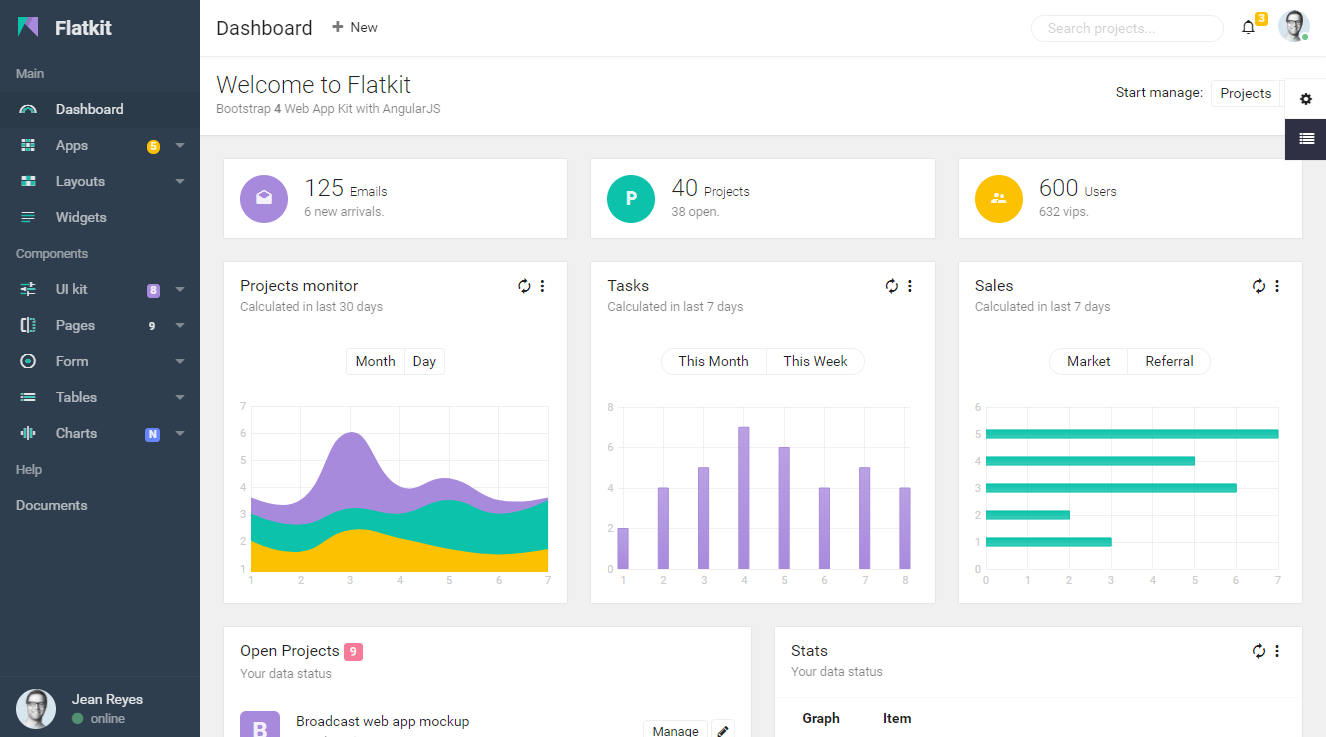
GetTagScans (@location, @fromtimestamp, @totimestamp)

|  |  |  |  |
| --- | --- | --- | --- |
| **Parameter Name** | **Data Type** | **Required?** | **Description** |
| location | String | Required | The location column in the Readers table to filter on. |
| fromtimestamp | Datetime | Optional | If not specified, the procedure will return all data since the last time it was called. If specified, the procedure will return all data since the date/time specified. |
| totimestamp | Datetime | Optional | If specified, fromtimestamp is required. The procedure will return all data between the fromtimestamp and totimestamp date/time range. |

# Web Application

A user will access a URL and a website hosted in Azure will be presented. The website will be built from a basic “administrative panel” template such as this one: <http://flatfull.com/themes/flatkit/angular/#/app/dashboard>

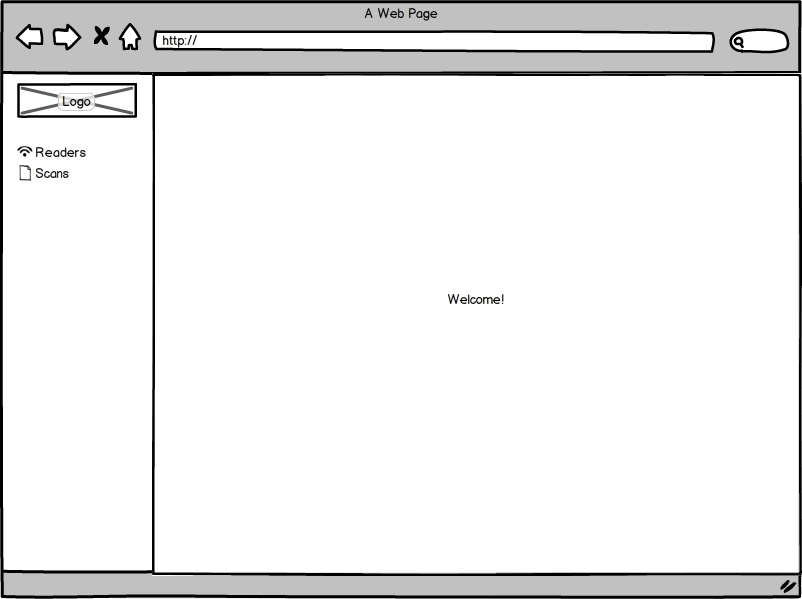
(from Themeforest: <https://themeforest.net/item/flatkit-app-ui-kit/13231484?s_rank=1>)



## Main Page

A user will be asked to provide a username and password to access the website. At this time, this basic security will be a static username and password that are stored (and therefore modifiable) in the application.config file.

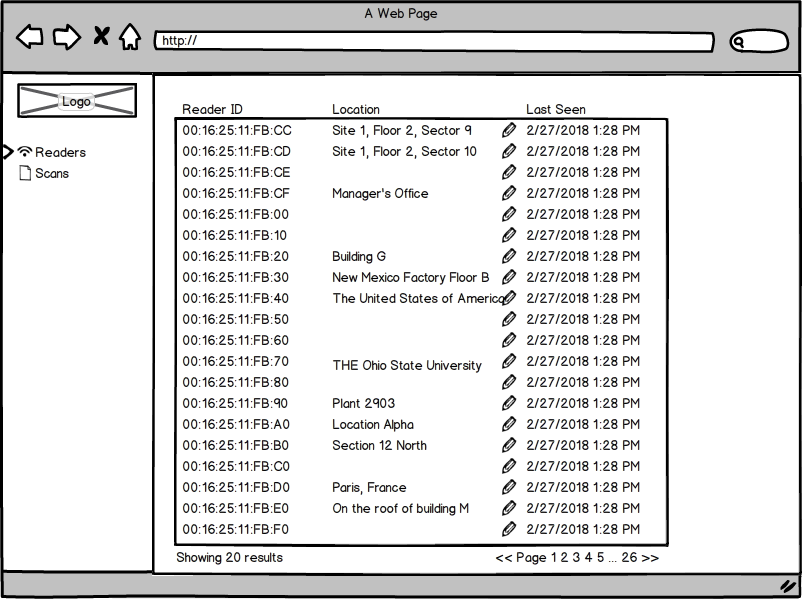
After logging in, the user will be shown a basic welcome screen with the company’s logo in the top-left of the screen, and the message “Welcome.” on the screen:



From this screen, the user can visit one of the following pages:

1. Readers
2. Scans

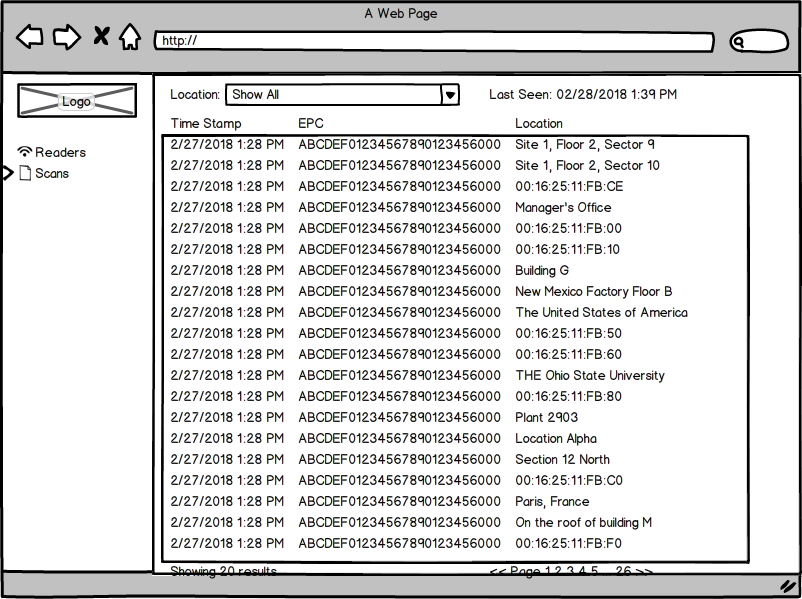
## Readers Page



On this page, the user can:

1. View a table which shows the contents of the Readers table, sorted by lastSeen, descending (
   1. only columns readerId, location, and lastSeen are shown in the table
   2. The lastSeen value should be shown to the user in Eastern Standard (or Daylight, depending on the date) Time and should be formatted as mm/dd/yyyy hh:MM AM/PM
   3. The table will show 20 entries and will support pagination
   4. If there is no value for lastSeen, show the user “Never” (it should not be possible for this to be the case, but it should be handled correctly if it happens for an unknown reason)
   5. If there is no value for Location, do not show any value in the table
2. Edit the location name for a reader
   1. click the pencil icon next to the location value for any record
   2. a textbox and save and cancel buttons appear. The current value is in the text box
   3. Edit the value, and click a Save button to persist the change to the DB, otherwise click cancel to cancel the operation.
   4. The page should not refresh for this operation, but the new value will be shown to the user if they save it

## Scans Page



On this page, the user can:

1. View a table which shows the contents of the Scans table, sorted by timestamp, descending
   1. only the columns readerId.location, EPC, and timestamp are shown. If readerId.location is NULL, then the value of readerId.readerId should be shown instead
   2. The table will show 20 entries and will support pagination
   3. The timestamp value should be shown to the user in Eastern Standard (or Daylight, depending on the date) Time and should be formatted as mm/dd/yyyy hh:MM AM/PM
2. There will be a dropdown box with a list of all Readers.location | Readers.readerId values. The user can select a value from this dropdown list and filter the table to only show Scans where the selected Readers record matches the Scans record
   1. If Readers.location is NULL for any entry, the Readers.readerId value should be added to the dropdown list.
   2. When a user selects a specific reader, the reader’s “Last Seen” value should also be shown. This value should be shown to the user in Eastern Standard (or Daylight, depending on the date) Time and should be formatted as mm/dd/yyyy hh:MM AM/PM
   3. The list should have a value of “Show All” which will remove the filter. This will be the default value. If this value is selected, do not show the Last Seen text and time/date.

# Reader Application

The reader application will be a custom application that is loaded on to the reader through the normal processes. The application will initiate tag scans continuously and add the scanned EPCs, scanning antenna, and current time to local non-volatile storage.

Another application on the reader will connect to the webservice on a periodic basis and will upload a tag read payload. Upon successful transmission, the uploaded tag read data will be removed from the non-volatile cache. This application will connect to the API every 10 seconds and send a payload with no tag data (if no reads were made since the last check in) to serve as a heartbeat function.

The application will also control an external indicator light by toggling a GPIO to engage the indicator light for 1 second when a tag read was successful.

The reader application also supports a number of configuration parameters that can be specified, as detailed below. This file is accessible by making an FTP connection to the reader, and modifying the config.json file. The file must be a properly formatted JSON file.

|  |  |  |
| --- | --- | --- |
| **Parameter Name** | **Data Type** | **Description** |
| TxPower | Unsigned int | Transmit power of the RF Antenna(s) in dBm. This must be a whole number with no decimal |
| RxSensativity | Unsigned int | Receiver sensitivity of the reader in dBm. This must be a whole number with no decimal |
| ApiUrl | String | The URL for the API that the reader should connect to |
| HeartbeatRate | Unsigned int | The amount of time the reader will wait between heartbeat checkins, in milliseconds |
| Antennas | Unsigned int | The number of antennas that the reader should use. This number should be between 1 and 4, depending on the reader. |