**j-EDI**

**Juniper Event Driven Infrastructure**

**Demo Walkthrough and Code Base Documentation**

Developers: Sudhishna Sendhilvelan and Kelcy Newton (Summer ’16 Interns- Partner and Channel SEs)

**Table of Contents**

Introduction……………………………………………………………………………………………………………………………3

* Who can use this?
* How to run the demo
* Value added to the business

Demo and Walkthrough……………………………………………………………………………………….…………………4

Slipstream Jargon, Keywords, and Other Important Concepts………………………………………………10

* Widgets
* Basic widget configuration
* Views
* Customization of widgets using rowData
* Using widgets in other views and files

Code Structure for Python API (Backend)………………………………………………………….………………….15

File and Code Structure for Slipstream Framework (Frontend).……………………………………….……16

Integration with JEANIE, Blue-J, and other ZTP Processes……………………………………………………..17

Appendix A (more detail on Slipstream)…………………………………………………………………..………..19

**Introduction**

j-EDI is an event driven infrastructure based on Salt that allows for the maintenance and configuration of mass quantities of Juniper devices in parallel. It relies on the 0MQ message bus where events are fired, allowing listeners to react accordingly, creating a fully automated system. To create the basic platform for the EDI that can be built on later, we created 5 workflows that can be seen in j-EDI’s UI.

* Device Listing – See a full listing of all devices in the EDI with details about each
* Add a Device – Add a new provisioned device to the EDI
* Run CLI Commands – Run a CLI command across multiple devices
* Configure Devices – Configure multiple Devices
* View Device Statistics – See statistics about all devices in the EDI and the system as a whole

In order to ensure all widgets, CSS and HTML files, and other elements integral to the look and feel of Slipstream were not compromised, I chose to build the front-end of j-EDI by editing the full Slipstream library I obtained from Juniper Git Labs. Consequently, there are several widgets and nested files within those widgets that are unused. These widgets could of course be integrated into the management platform for j-EDI if a developer chose to do so. Below is a comprehensive overview of all used widgets and the files necessary to render the widgets used in the j-EDI demo. Further detail can be found in the javaDocs for all main functions and in the comments of the code in the Juniper/J-EDI repository on the public Github server. All front-end code is in the ‘Slipstream’ Folder in the repo. Please email [knewton@juniper.net](mailto:knewton@juniper.net) or [kelcynewton@gmail.com](mailto:kelcynewton@gmail.com) for any questions, comments, or concerns.

**Who can use this?**

The slipstream repository is an internal front-end framework utilized by Juniper Network to create clean, industry ready web interfaces. Based on backbone.js, it is a mixture of javascript, html and css. Anyone with proficiency in these languages can understand and develop applications using slipstream. All that is required is the ability to learn.

**How to Run (once slipstream is installed)\*\***

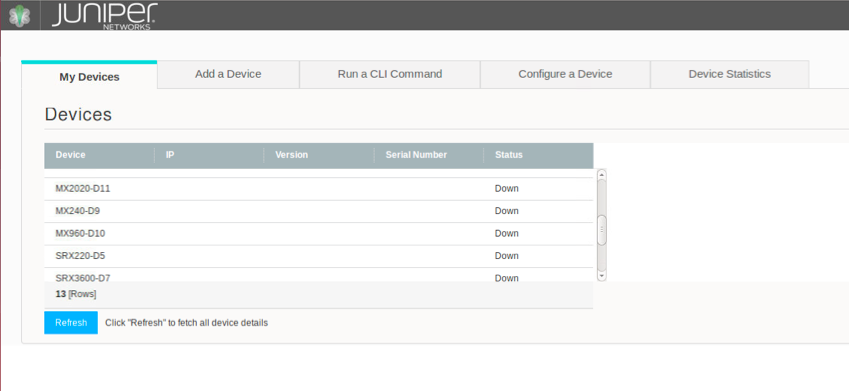
* Navigate to the Slipstream directory in your local environment
* Start the Redis Server with “redis-server&”
* Start the local host server (for development and testing purposes) with “node app.js”
* Navigate to <http://localhost:3000/assets/js/widgets/tabContainer/tests/testTabContainer.html>

**\*\*** For details on how to set up the slipstream environment, please see j-EDI Front End Environment Setup.docx

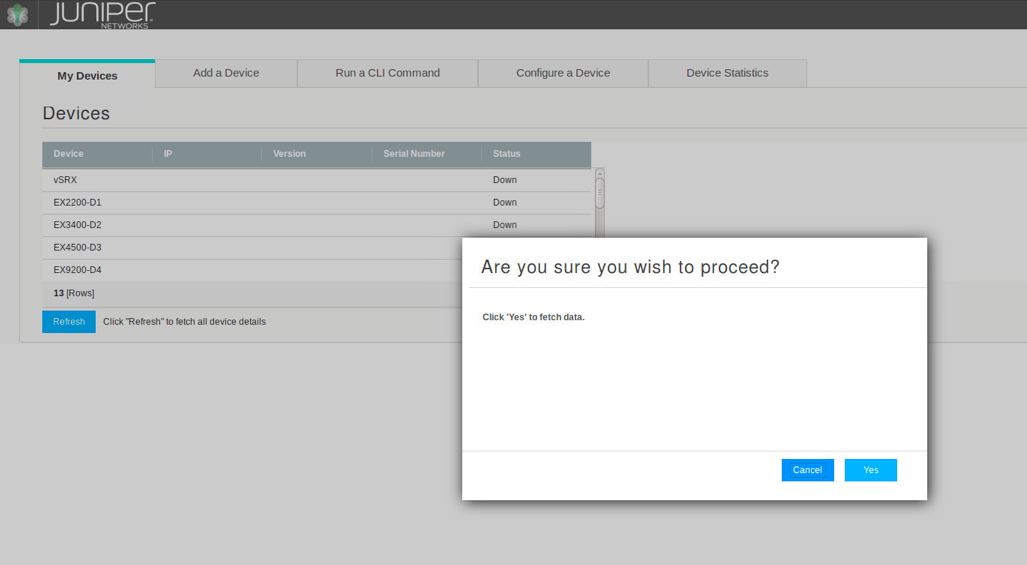
**Why is this valuable to Juniper and how does it help the business?**

At the end of the day, Juniper is a business and the overall goal is to increase both revenue and profit. One of the best ways to do this is to decrease Operational Expenditure (opex). J-EDI directly lowers opex by creating an automated system that requires little to absolutely no human maintenance or interaction to do tasks that once were done manually. Additionally, Juniper currently has no technology like j-EDI and its core, the Event Driven Infrastructure. Many of the IT automation and orchestration tools used in practice today such as Ansible, Puppet or Chef, are simply not powerful enough, scalable enough, or fast enough to keep up with the high paced networking field we find ourselves in today, especially at this precipice of industry transformation towards software driven networking. Consequently, we need a platform like j-EDI, which runs on Salt, to provide us with the ability to automate menial daily tasks as well as help in third party areas like Junos Space or OS Ticket, where manual effort is still required to bring a device up in “Discovery” or to open a ticket for newly configured device.

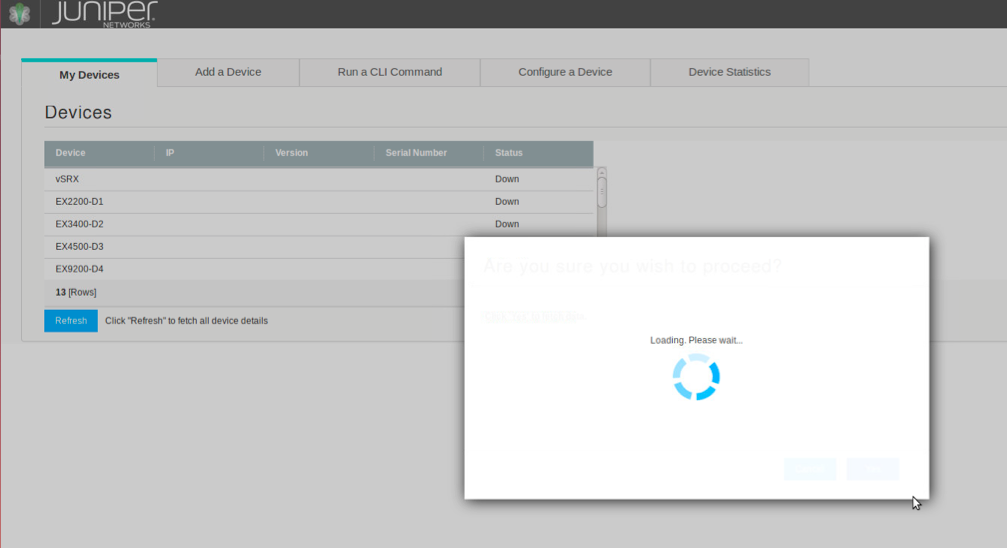
**Demo Walkthrough**



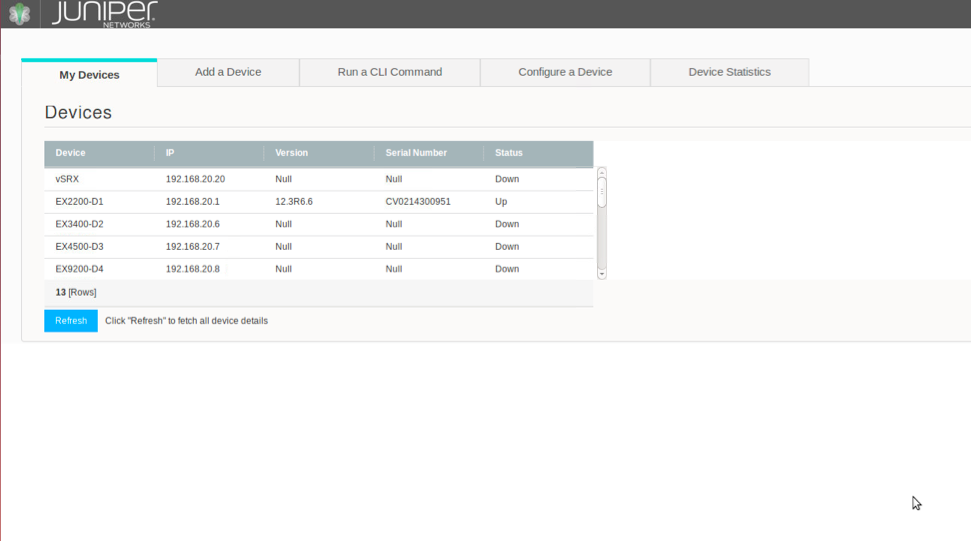
This page shows on initial loading of the application. It gives the user a text-based overview of all devices currently in the Event Driven Infrastructure with a listing of device hostnames and device statuses.



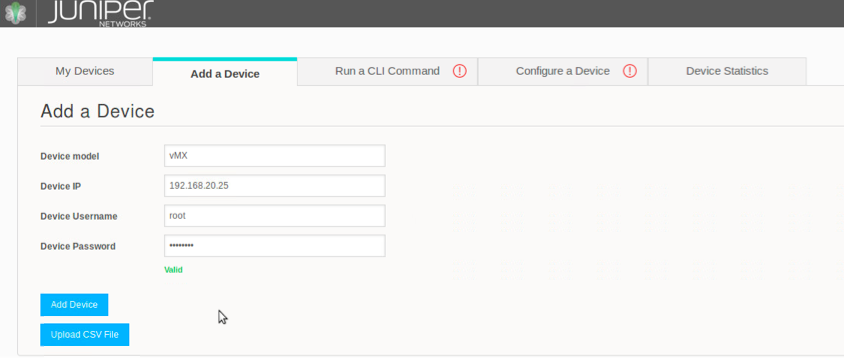
On any button click, data is fetched, and while that call to the backend API is occurring, the user is presented with a confirmation box ensuring they want to proceed with the action. Clicking ‘Yes’ will fetch the data, clicking ‘cancel’ will cancel the action and destroy the confirmation overlay.



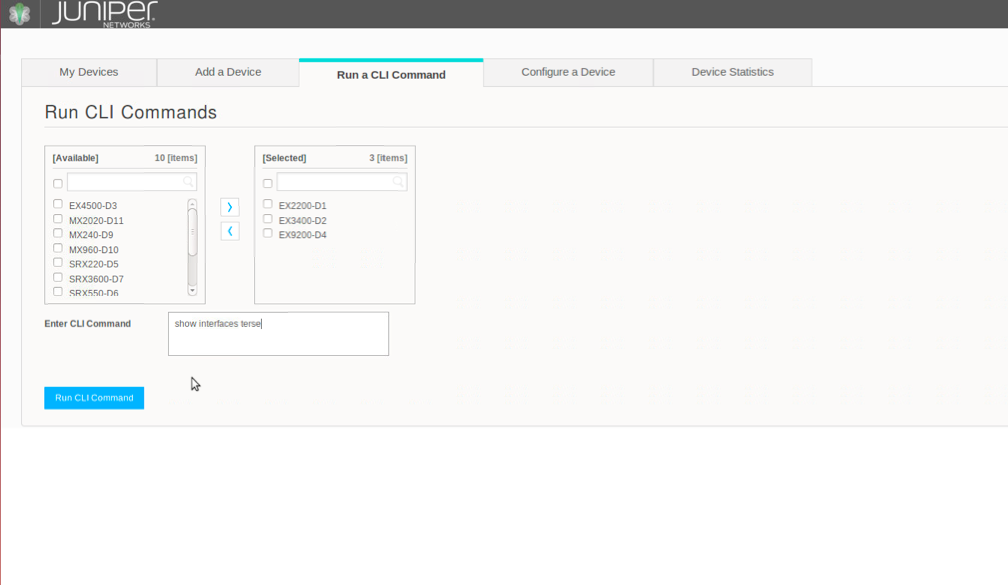
After confirming that you’d like to continue, a loading indicator opens, locking the screen until all data is fetched and the overlay is destroyed.



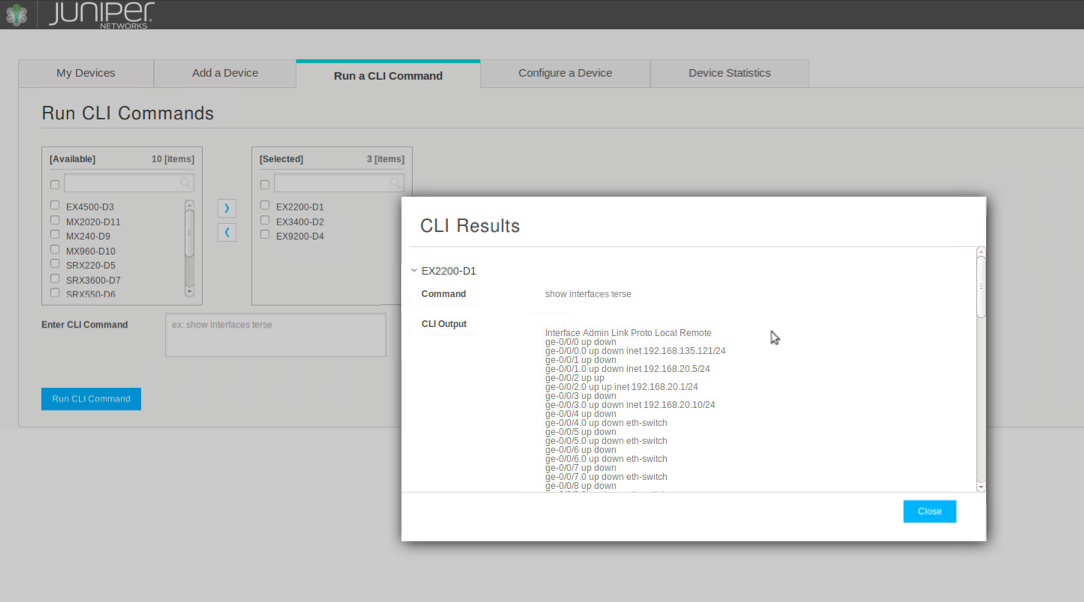
After data is fetched, device details like IP, Serial Number and Version are brought into the view. Each device also gets a real-time status value of whether it is ‘up’ (powered on) or ‘down’ (powered off). Clicking ‘Refresh’ after this first, initial time, will update the grid details.

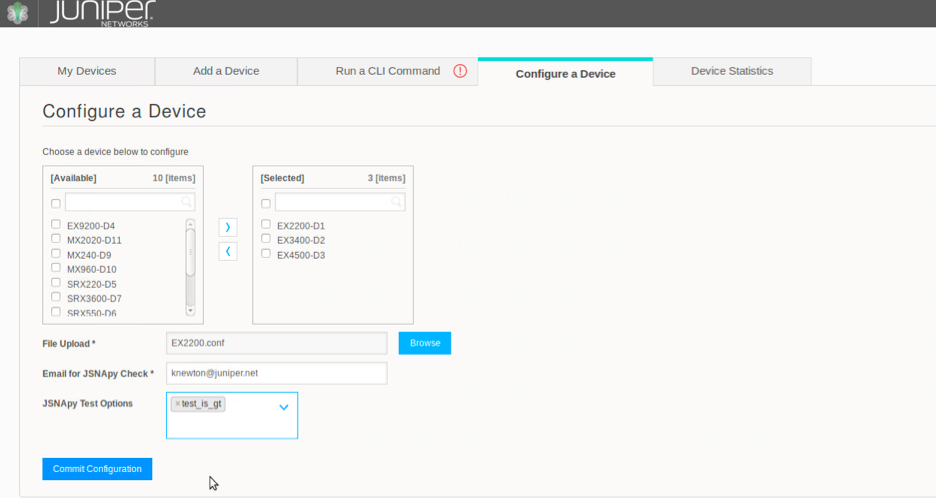


Devices are added to the EDI in two ways: through a ZTP process like JEANIE using JEAP or Blue-J, or added through the j-EDI GUI. After entering model/hostname, ip, and user credentials, the device is added to the EDI and will appear in the device listing in tab 1. J-EDI also has the potential to add multiple devices at a time via a CSV file. This page checks for valid IPs as well as strong passwords.

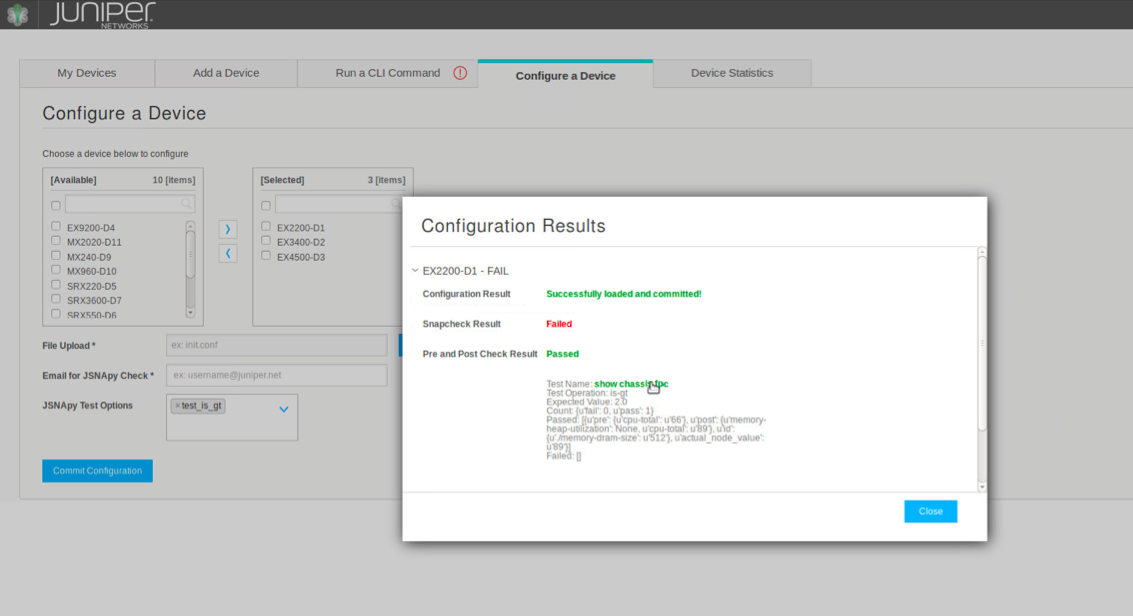


Tab 3 allows the user to run a CLI command simultaneously across any number of devices. Select one or multiple devices from the available column and click the right arrow to “select” them, and enter a CLI Command to run.

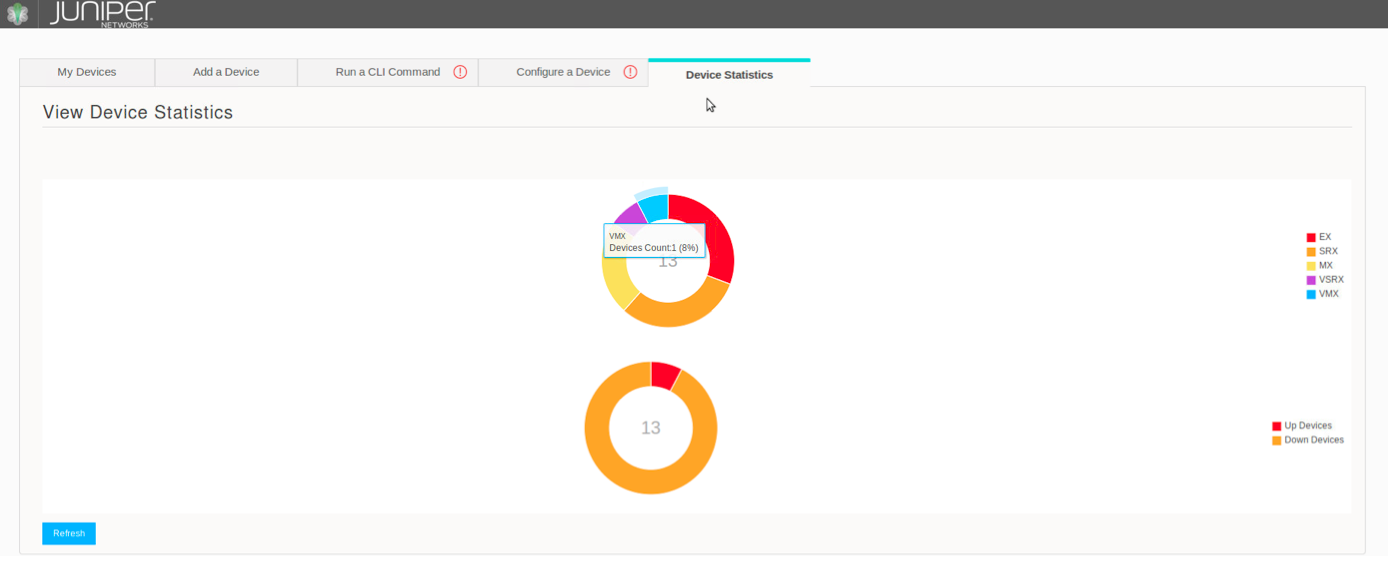
After receiving the results of the CLI command from all devices, they are displayed in an overlay. All devices are listed with the hostname, and if the device is not connected, a “device not connected” tag is attached to the hostname. This allows the user to only dive deeper into the devices that are up and will return results. Clicking the arrow to the left of the hostname expands the menu and reveals the results of the CLI command. If you expand a device that was not connected, the results return null.



The 4th tab allows the user to configure multiple devices in parallel. Select your devices, browse for a configuration file, enter your email, and select jSNApy tests (optional). The entering of a user email is purely for demo purposes. It serves to compare the pushed configuration to the Golden Config to ensure the system doesn’t break in the future. In the case of a failed configuration due to the Golden Config, the user receives an email with details to revert the system. In reality, the system administrator in charge of j-EDI will receive the email to his or user credentials. The user also has the option to select various tests for a pre and post check of the system, where a pre snapshot and post snapshot are compared to check functions like how many CPU cores are running before and after the configuration was committed.



Upon completion of the configuration commit, the results are displayed in an overlay similar to the CLI results. Again initially we see a list of all devices, both connected and not connected. For connected devices, each will have a “PASS” or “FAIL” label attached for an instant notification to the user. Upon expansion of any device, the results are revealed. To give the user the clearest indication of what happened during the commit, good results are displayed in green, and bad results are displayed in red. If pre and post tests are selected, the details from those tests are listed as well. If a device is not connected, all result fields will be displayed as “null”.



The 5th and final tab, once again, allows a user to analyze the entire EDI from a high level perspective. Here, 2 different graphs are available to give the user real-time statistics about the system. First, there is a donut chart displaying all types of devices in the system such as EX, MX, vSRX, MX, etc. Scrolling over one part of the chart reveals more details about that particular type as it relates to the system in terms of percentage. Every chart can have a legend indicating the colors and parts of the donut chart. The second chart displays the statuses of all devices, alerting the user how many devices are up and how many are down. Many more graphs can be created depending on the need of the user or desires of the developer. Two more zoomed in looks at these graphs are seen below.



Donut Chart of device types in the EDI



Donut Chart of device statuses in the EDI

**Slipstream Jargon, Keywords, and Other Important Concepts**

Below are some key concepts and phrases that are vital to this project. In order to understand the code, you must understand the keywords used in slipstream to create its interface. These conventions are used often throughout the code base. Please use this section as a reference for widget instantiation and management.

* **Widget** – The items that populate any slipstream page. Can be both interactive and static. Here I’ve provided a few code examples that create these widgets in the j-EDI demo.
  + [Confirmation Dialog](https://ssd-git.juniper.net/spog/slipstream/tree/master/public/assets/js/widgets/confirmationDialog/confirmatonDialogWidget)
    - A “pop-up box” with some kind of information for the user
    - Can implement callback functions on button clicks (“okay”, “yes”, “no”, “cancel”) if necessary
    - In j-EDI, the Confirmation Dialog widget is used to alert the user of a successful or failed device addition.
    - A Confirmation Dialog can be created with user specifications from any view:

**new** ConfirmationBox({

'rowData': {"title": "title", "question:"question"}

            }).render();

* + - Then, in a different js file where the Confirmation Dialog Widget is built, we get the ‘rowData’ variable and create the configuration and build the widget from it.

**var** rowData = **this**.options.rowData;

**var** title = rowData["title"]

**var** question = rowData["question"]

//Create the widget configuration

**var** conf = {

title: title,

question: question,

yesButtonLabel: 'okay',

//cancelLinkLabel: 'press esc to close',

yesButtonCallback: yesButtonCallback,

//cancelLinkCallback: cancelLinkCallback,

xIcon: true

};

//create the widget and build it

**this**.confirmationDialogWidget =

**new** ConfirmationDialogWidget(conf);

**this**.confirmationDialogWidget.build();

* + [Donut Chart](http://ssd-git.juniper.net/spog/slipstream/blob/master/public/assets/js/widgets/donutChart/donutChart.md)
    - A beautiful, visually appealing way to display real time or static data
    - Can override default colors and choose your own
    - In j-EDI, the Donut Chart widget is used to display statistics about the devices
    - In j-EDI, I create 2 donut charts and add them to a single template where each is looking for a specific div element.
  + [Grid](https://ssd-git.juniper.net/spog/slipstream/blob/master/public/assets/js/widgets/grid/grid.md)
    - Helpful when you want to display *a lot* of data in many rows and many columns or pull data from multiple APIs.
    - In j-EDI, the Grid widget is used to display the list of devices and it’s details in Tab 1.
  + [Form](https://ssd-git.juniper.net/spog/slipstream/blob/master/public/assets/js/widgets/form/form.md)
    - Potentially the most used widget, the Form widget allows the user to enter information via text box, check box, file upload, button click, and more.
    - Sometimes, a view will be rendered as a form, and the form can contain more widgets. For example, in the CLI and Configuration tabs, the List Builder widget is sitting inside a Form widget. In the form configuration for each of those files, there is an *text\_element* with the ID ‘application’. When the list builder widget is rendered, it looks for that ID to know where to build. See the code under ‘elements’ below:

formConfiguration.constructCLICommand = {

"title": "Run CLI Commands",

"form\_id": "cli\_form",

"form\_name": "cli\_form",

"err\_div\_id": "errorDiv",

"err\_div\_message": "One or more fields have errors. Update the fields highlighted below. For detailed information on possible values see",

"err\_div\_link":"http://www.juniper.net/techpubs/en\_US/junos12.1x46/topics/task/configuration/j-web-basic-settings.html",

"err\_div\_link\_text":"Configuring Basic Settings",

"err\_timeout": "1000",

"valid\_timeout": "5000",

"sections": [

{

"elements": [

{

"element\_text": true,

"id": "application",

"class": "list-builder",

"name": "application",

"placeholder": "Fetching Devices..."

},

* + - In j-EDI, the form Widget is used to create the basis for all 5 tabs, as well as the forms shown in the overlay elements.
  + [List Builder](https://ssd-git.juniper.net/spog/slipstream/blob/master/public/assets/js/widgets/listBuilder/listBuilder.md)
    - The List Builder element is helpful in listing a lot of devices in a confined space. It’s also helpful when wanting to select one or more devices fairly quickly, more so than a drop-down type menu.
    - In j-EDI, the List Builder is used to show a list of devices available in CLI and Config tabs. A user can search for a particular device or scroll through. There’s also the option to select all devices at once.
    - Rather than a single list with checkboxes, List Builder uses two parallel lists, one for Available and one for Selected. This functionality allows the user to ensure they have only selected what they want.
    - In j-EDI, the List Builder widget is used to display available and selected devices in the CLI and Configuration tabs.
    - In this application, the list builder is actually created as an element of a form widget as mentioned above in the “form” section.
  + [Overlay](https://ssd-git.juniper.net/spog/slipstream/blob/master/public/assets/js/widgets/overlay/overlay.md)
    - Similar to Confirmation Dialog, the Overlay widget pops up on top of the main page with some information for the user. Other widgets can sit inside this overlay such as forms, charts or grids.
    - In j-EDI, the Overlay widget is used to display the CLI and Configuration results
    - The creation of an overlay widget has two main parts: the overlay element and the quickview
    - The overlay is called in both CLI and Configuration using “QuickView”. What makes it a little more complicated is that the QuickView is comprised of multiple elements layered on top of one another. Here, “rowData”, is a jSON object being passed to the QuickView class. This instantiation calls cliQuickView.js to create the overlay view with ‘rowData’.

**new** QuickView({

'rowData': rowData

}).render();

* + - In cliQuickView.js, the overlay widget is rendered like most any other widget, and the cli results form is layered inside the overlay using “rowData”. See 'widgets/grid/tests/view/cliQuickView' for more details and to customize the data added to the overlay.

**this**.overlay = **new** OverlayWidget({

view: **this**,

xIconEl: true,

showScrollbar: true,

type: 'medium'

});

**this**.overlay.build();

* + [Spinner](http://ssd-git.juniper.net/spog/slipstream/blob/master/public/assets/js/widgets/spinner/spinner.md)
    - Also called a progress indicator, the Spinner widget is used to alert the user of the progress of some process. The spinner can be determinate (i.e., give the user a definite answer of where the progress is) or indeterminate (i.e., just a simple spinner with no numerical value)
    - The Spinner is helpful for 2 main reasons. 1) It alerts the user of what is happening in the backend where they cannot see the processes and likely prevents them from clicking a button multiple times. 2) It locks the display on that spinner until the data is fetched and loaded.
    - The spinner is most easily rendered inside an Overlay widget
    - In j-EDI, the spinner is used anytime a user fetches data in any tab.
  + [Tab Container](https://ssd-git.juniper.net/spog/slipstream/blob/master/public/assets/js/widgets/tabContainer/tabContainer.md) 
    - The TabContainer Widget creates a tabbed application. Based on the configurations, the tabs can be horizontal, vertical, aligned right or left, etc.
    - In j-EDI, the Tab Container widget is used to create the main view that houses all of j-EDI’s functionality.
* **Basic Widget Configurations – how each widget is created**
  + In this code base, widgets are configured 1 of 2 ways:

**this**.myWidget = **new** ExampleWidget({

"elements": elementsForWidget,

"container": **this**.el

});

**this**.myWidget.build();

or

**var** conf = {

container: **this**.el,

options: options

};

**var** myWidgetj = **new** fakeWidget(conf);

myWidget.build();

* + In this first scenario, the configuration for the widget is built inside the widget itself and in the second scenario, the configuration is declared and then passed as a parameter to the new instantiation of the widget
  + “Elements” is the data that makes up the element. Check each widgets documentation to see how the widget expects the data it receives to be formatted. “Container” is where the element is to be held and typically its “this.el” meaning this same element the view is instantiated in. But when you start nesting widgets, the container could change.
  + I’ve shown a few particular examples in the widgets section above. Reference the Slipstream Repo for more information.
* **What’s a view?** 
  + Views are rendered inside .js files and can be used to populate different slipstream pages. Typically there is one main view that holds some widget or maybe multiple widgets. And then each of those widgets can also hold a view. And that view inside that main widget can hold one or more widgets as well.
  + In the case of j-EDI, the main view is TestTabContainerView that returns a testTabContainerView object, and inside TestTabContainerView a tabContainerWidget is rendered. Within this tabContainerWidget, each tab renders its own view (5 in total). Inside each of those views are many widgets that make up the elements of that page the user sees. For example, the Grid to display all current devices in Tab 1, the form to add a new device in Tab 2, the form and List Builder element in Tab 3 and 4, and the Donut Charts to display device statistics in Tab 5.
* **Customization of widgets using rowData**
  + If a widget is instantiated in one file and configured and built in another, It is possible to pass data from the file where the widget is called to the file that actually builds the widget, using RowData. This allows for the custom use of a single widget across many different scenarios.
  + In the file where the widget is instantiated you do the following:

**new** myWidget({

'rowData': {"key1": "value1", "key2: "value1"}

            });

* + Then in the file where the widget is rendered, obtain that data with

**var** dataFromOtherFile = **this.**options.rowData;

* + This is especially useful when you want to repeatedly use a widget with different text that might be specific to a certain view. You only need one widget configuration for any view you need.
* **Widgets are passed as parameters to different views and then rendered inside the view by that parameter name. All the file needs to use that widget is the directory it lives in.**
  + At the top of most any file in this directory, (and specifically the 5 main views used to create the 5 tabs) you will see this format:

define([

'backbone',

'widgets/form/formWidget',

'../conf/formConfiguration',

'widgets/listBuilder/listBuilderWidget',

'widgets/grid/tests/view/cliQuickView',

'widgets/confirmationDialog/tests/confirmationBox',

'widgets/spinner/tests/appOverlaySpinner'

], **function**(Backbone, FormWidget, formConfiguration, ListBuilderWidget,QuickView, ConfirmationBox, SpinnerView){

**var** myView = Backbone.View.extend({

//code here

});

**return** myView;

});

This is the initial setup for a view.js file. We define any widgets we intend to use in that view by specifiying the path to that widget configuration, and then we pass in our own variable name corresponding to that file as parameters. The pairs must be sequential and matching or a compilation error will occur. For example, the 1st path ‘/widgets/form/formWidget’ corresponds to the 1st parameter FormWidget, the 2nd path ‘/…/conf/formConfiguration’ corresponds to the 2nd parameter formConfiguration and so on. Now when we want to create an instance of the SpinnerView for example, we can declare var mySpinner = new SpinnerView() and the corresponding file will be called.

* **Combining Multiple Widgets in One View**
  + Since widgets are standalone, they do not stack very neatly. A way around this, allowing a developer to use multiple widgets in the same view is to create an html template with div elements, where each div is the placeholder for a different widget. Create a separate view for each widget where you configure and build it. In the main view that will hold all the widgets, you simply add the content of your widget-views to that view.
  + The example from Donut View is below. The example code would be written inside the main view js file where you wish to render multiple widgets in parallel.

**var** StatsView = {};

**var** addContent = **function**($container, template) {

$container.append((render\_template(template)));

};

StatsView.view1 = Backbone.View.extend({

render: **function** () {

addContent(**this**.$el, donutChartTemplate);

//render a new donut chart view in the specified div

**new** DonutChartView1({

el: **this**.$el.find('#device\_status')

});

//render a new donut chart view in the specified div

**new** DonutChartView2({

el: **this**.$el.find('#device\_type')

});

**return** **this**;

}

});

**return** StatsView;

* + And separately, you have donutChartTemplate.html, where each donut chart is rendered based on the div id it is looking for.

<**div** id="form"></**div**>

<**div** id="device\_type"></**div**>

<**div** id="device\_status"></**div**>

**Code Structure for Python API (Backend)**

For reference, the main Python API files that might be referenced in this document are as follows:

* DeviceListing.py
  + Used in Tabs 1, 3, 4 for lists of all EDI devices
  + Returns a list of all devices in the infrastructure
* GetDeviceDetails.py
  + Used in Tab 1 to list device details
  + Returns device details like IP, Serial Number, and Version
* AddNewDevice.py
  + Used in Tab 2 to add a new device to the EDI
  + Takes new device parameters and adds a new device to the infrastructure
* ExecuteCli.py
  + Used in Tab 3 to execute a CLI command across parallel devices
  + Returns the results from the CLI command called on selected devices
* ConfigDevice.py
  + Used in Tab 4 to configure multiple devices
  + Returns the configuration results called on selected devices
* StatusStatistics.py
  + Used in Tab 5 to display the statuses of devices
  + Returns the types of all devices present in the infrastructure by number and percentage
* TypeStatistics.py
  + Used in Tab 5 to display the device types
  + Returns the status of all devices present in the infrastructure by number and percentage

**File and Code Structure for Slipstream Framework (Frontend)**

Main Directory – j-EDI/slipstream/public/assets/js/widgets

The following shows the file structure used in the front end creation of j-EDI. *All of these widgets are found in j-EDI/slipstream/public/assets/js/widgets. For function details, see javadocs inside files.*

**/confirmationDialog**

* tests/confirmationBox.js
  + creates a ConfirmationDialog widget used to interact with the user and determine navigation actions
  + Alerts the user of success or failure when adding a device

**/donutChart**

* /tests/getDeviceStatus.js
  + creates the donut chart with data about the status (up or down) of all j-EDI devices
* /tests/getDeviceTypes.js
  + creates the donut chart with data about the types of devices (EX, MX, vSRX, etc.) and how many of each are present in j-EDI

**/grid**

* /conf/configurationSample.js\*\*
  + holds the form configuration used to render the device listing in Tab 1-Devices
* /tests/appSmallGrid.js
  + creates the view that iterates through and displays all present devices with details such as IP, Serial Number, Version, and Status
* /tests/view/configQuickView.js\*\*
  + Creates the overlay view that renders with the results from running a CLI command on one or more devices
* /tests/view/cliQuickView.js\*\*
  + Creates the overlay view that renders with the results from configuring one or more devices

**/spinner**

* /tests/appOverlaySpinner.js
  + Opens the overlay when a user clicks to request data. This overlay allows them to proceed or cancel if they clicked by mistake
* /tests/views/activityIndicatorView.js
  + Renders the progress indicator (spinner view) inside the overlay. It locks the display until the requested data has been fetched and displayed in the corresponding display.

**/tabContainer**

* /tests/main.js
  + the main javascript with all the dependencies required to create all widgets and views properly. main.js also renders the TestTabContainerView with standardTabContainerWidget.js as the configuration.
* /tests/testTabContainer.html
  + j-EDI’s main html file. All views are rendered within this file
  + The main tabContainer view is rendered in #main\_content ( see assets/public/widgets/tabContainer/tests/main.js for more information)
* /tests/view/StandardTabContainerWidget.js
  + initializes, renders, and builds a tabContainerView with all 5 tabs used in j-EDI. Indicates which view is to be rendered in each tab. Can handle action events if needed.
* /tests/conf/formConfiguration.js\*\*
  + holds the form configurations used for each view in Tabs 1-5. These form configurations dictate what elements (textboxes, buttons, checkboxes, other widgets) are rendered in each view. Each view has its own form, but views can share forms also.
* /tests/models/devicesModel.js
  + Creates the model used to feed data into the widgets that use the Device Listing API (deviceListing.py)
* /tests/view/addDeviceView.js
  + creates the view with a specific form configuration that allows the user to add a new device to the EDI
* /tests/view/cliView.js
  + creates the view that allows a user to run CLI commands across multiple devices in parallel
* /tests/view/configView.js
  + creates the view that allows a user to configure multiple devices in parallel
* /tests/view/statsView.js
  + creates the view that renders the various charts used to describe the system at a high level such as device types and device statuses
* /tests/view/donutChartTemplate.html
  + basic html file that each donut chart references to render correctly.

**\*\*** The form and overlay widgets were used also, but they are nested within the files listed above.

**Integration with JEANIE, Blue-J and other ZTP Processes**

In order to create a full workflow, I wrote a python script that is called from JEAP and fires the device details to the Event Driven Infrastructure. Once the device is in the EDI, the Salt Master has full control of the device via proxy minion. Ideally, as long as a ZTP process like JEANIE or Blue-J through the use of JEAP can provision a device, it can be automatically added to the EDI via a Salt fire to the message bus.

**JEAP**

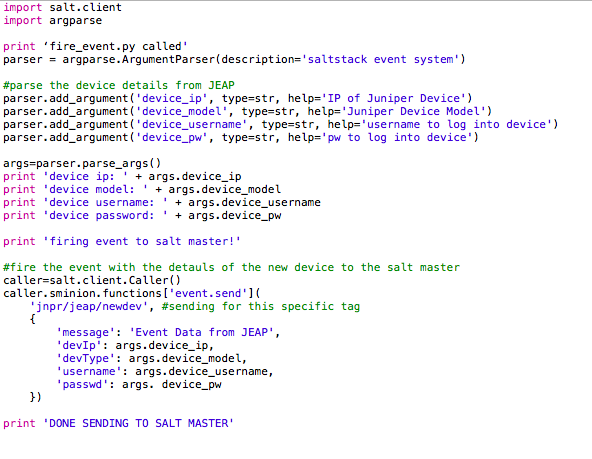
From JEAP, a script called fire\_event.py is called using a subprocess command seen here

command = “python fire\_event.py” + sample[“ip”] + sample[“hostname”] + “username” + “password”

subprocess.call(cmd, shell = True)

Upon firing the device details to the message bus, a proxy minion is spun up for the device and it is under the complete control of the salt-master.

**fire\_event.py**



The code above, parses the arguments given from JEAP (a subprocess call is fired from JEAP to this script) and then those arguments are given as arguments to an event and fired to the Salt Master via the message bus. Any listener looking for the tag ‘jnpr/jeap/newdev’ will be able to grab these details and the device can be added to the EDI. There is currently not a subscriber for JEAP, but one can be easily created.

**APPENDIX A**

**Check out the link below to see the entire j-EDI repository featuring this documentation and all code.**

[j-EDI Git Repository](https://github.com/Juniper/j-EDI)

**Check out the links below for more specifications about how to create and build these widgets.**

* [Full Widget Listing](https://ssd-git.juniper.net/spog/slipstream/tree/master/public/assets/js/widgets)
* [Confirmation Dialog Widget](https://ssd-git.juniper.net/spog/slipstream/blob/master/public/assets/js/widgets/confirmationDialog/confirmationDialog.md)
* [Donut Chart](http://ssd-git.juniper.net/spog/slipstream/blob/master/public/assets/js/widgets/donutChart/donutChart.md)
* [Form Widget](https://ssd-git.juniper.net/spog/slipstream/blob/master/public/assets/js/widgets/form/form.md)
* [Grid Widget](https://ssd-git.juniper.net/spog/slipstream/blob/master/public/assets/js/widgets/grid/grid.md)
* [List Builder Widget](https://ssd-git.juniper.net/spog/slipstream/blob/master/public/assets/js/widgets/listBuilder/listBuilder.md)
* [Spinner Widget](http://ssd-git.juniper.net/spog/slipstream/blob/master/public/assets/js/widgets/spinner/spinner.md)
* [Tab Container Widget](https://ssd-git.juniper.net/spog/slipstream/blob/master/public/assets/js/widgets/tabContainer/tabContainer.md)