Table of Contents

nio Documentation

Welcome to nio. This documentation strives to help you design complex systems with nio—a distributed orchestration platform.

This is a work in progress and niolabs welcomes your feedback on the documentation. If you have questions, comments, or suggestions about this document, please email editor@n.io.

Getting Started

nio is a highly-scalable stream processing engine that allows you to asynchronously process streaming data signals and act on them in real time. nio is the brain of your real-time application, particularly systems that require distributed processing.

Here are a few sample use cases for nio:

- You want to know what people are posting about your company on Twitter. You can instantaneously receive notifications when tweets mention your business, automate responses, and identify trends are received. You can even alert your support team when the number of "help me" tweets passes your acceptable threshold.
- You run a factory and want to measure equipment performance and maintenance factors. To optimize the business processes and reduce downtime, nio can read the signal from any sensor and act on the data in real time.

In this getting started tutorial you will learn how to get nio up and running before you proceed to the tutorials.

Running nio in the Cloud

The easiest way to run nio is by running it in the cloud.

Below you can walk through the foundational steps of building a project including creating a system, cloud instance, and service, followed by adding blocks.

Create a System

1. Open the **System Designer** in a new tab.

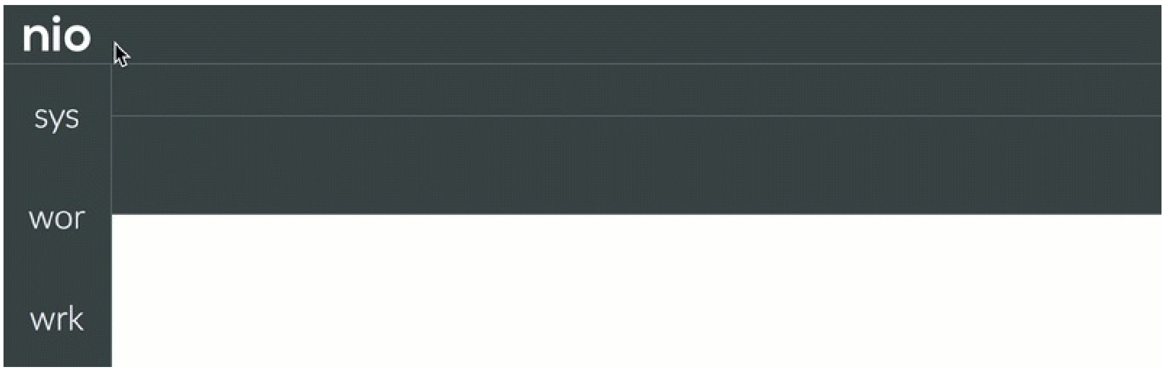
   https://designer.n.io/

2. In the lower-left corner, click the `+` button to build.

3. Complete the **Create a new system** window:
   - In the **System name** box, enter your system name.
   - Select **Auto** for the Pubkeeper configuration. This is the default.
4. Click **Accept**.

Create a Cloud Instance

1. Select the name of your system in the left navigation panel or on the breadcrumb above the contextual toolbar.



2. Click **Create a cloud instance**.

3. Complete the **Create a new cloud instance** window:
   - In the **Instance name** box, enter your instance name.
   - Leave the **Instance type** as **n.io Cloud**.
4. Click **Accept**.

Create a Service

1. Select the name of your instance under the system name in the left navigation panel or on the breadcrumb above the contextual toolbar.
2. Click **Add new service**.
3. Complete the **Create new service** window:
   - In the **Service name** box, enter a service name.
   - Leave the **Service type** as **Service**.
4. Click **Accept**.

5. Click **Save** in the toolbar.
6. Click **Edit** in the toolbar.
7. Click **Auto-Start Off** in the toolbar.

Add Blocks

You now have an empty canvas, and now you are ready for blocks.

nio blocks are installed to instances using the Block Library. The collection of blocks created by nio is also stored in the Block Library at blocks.n.io. You can search for blocks in the System Designer or in the Block Library.

1. Click the service name under the instance name in the left navigation panel or on the breadcrumb above the contextual toolbar.
2. In the **Block library** search box, enter a search term.
3. If the block is not displayed, click **Available**, **Installed**, and **Configured** to search for the block.
4. If the block is not already pre-installed, click the **Install Block** button which resembles a cloud with a down arrow.
5. Drag the block type to the canvas.
6. Name the block.
7. Click **Accept**.

Once you have nio running, you can create many different projects. To guide you through the process, view the tutorials at https://workshops.n.io/.

Running nio Locally

The cloud is an easy way to get nio up and running, but doesn't fully encapsulate the distributed power of the nio platform. Instead, you should run nio on a local or edge node. In this guide, you will create a local instance that uses a cloud-based Pubkeeper server to handle communications.

Running the nio platform requires Python version 3.4.5 or 3.5.2. Other versions of Python 3.4.x may work, but Python 3.5.3 and later do not work. When running the nio binary, the `Bad magic number` error is most likely caused by an incompatible version of Python.

Requirements

- Download Python 3.4.5 or 3.5.2 from https://www.python.org/downloads/.
- Obtain the nio binary Python wheel file (`.whl`) with your license agreement.

Download nio

Download nio from the following URL:

https://app.n.io/binaries/download

Install nio binary and CLI

To install nio, enter the following command:

```
pip3 install your_wheel_file.whl
```

To install the nio Command Line Interface (CLI), enter the following command:

```
pip3 install nio-cli
```

Create a Project

Now that you have the nio binary to run, you need a nio project to run it against. Obtain a nio project template by cloning the Project Template repository or use the nio CLI.

Download using the nio CLI

To clone the project template using CLI, enter the following command:

`nio new first_project`

The `first_project` directory is created in your working directory containing the nio project.

Download using git

To clone the project template using git, clone the template, and initialize the submodules which contain the blocks.

```
git clone https://github.com/niolabs/project_template.git first_project
cd first_project
git submodule update --init --recursive
```

Set up nio environment

After adding the new project, change to the project directory

```
cd first_project
```

Create a System

1. Open the **System Designer** in a browser.

   https://designer.n.io/

2. In the lower-left corner, click the `+` button to build.

3. Complete the **Create a new system** window:
   - In the **System name** box, enter your system name.
   - Select **Auto** for the Pubkeeper configuration. This is the default.
4. Click **Accept**.
5. Click **Edit** in the contextual toolbar to open the system's configuration.
   > Make note of the values for **host** and **token** which will be used in the following steps.

6. From your terminal, in your first_project directory, open the `nio.env` file.
7. Update the following four lines in the `# Pubkeeper Client` section

   ```
   PK_TOKEN: [your copied token]
   PK_HOST: [your copied host]
   PK_PORT: 443
   PK_SECURE: True
   ```

8. Update the following three lines in the `# Websocket Brew Variables` section

   ```
   WS_HOST: [your copied host, replacing `pubkeeper` with `websocket`]
   WS_PORT: 443
   WS_SECURE: True
   ```

9. To run nio, enter the following command:

   ```
   nio_run
   ```

   > If that command is not available, make sure your Python binary installation directory is on your PATH.

   - If you need help setting your PATH in Windows, click here.
   - If you need help setting your PATH in MacOS, click here.
10. The log messages display, similar to the following output, but there should be no errors.

   ```
   [2016-03-04 23:49:41.035] NIO [INFO] [main.WebServer] Server configured on 0.0.0.0 : 8181
   [2016-03-04 23:49:41.035] NIO [INFO] [main.WebServer] Server configured on 0.0.0.0:8181
   [2016-03-04 23:49:41.042] NIO [INFO] [main.ServiceManager] Component: ServiceManager status changed from:
   to: created
   [2016-03-04 23:49:41.055] NIO [INFO] [main.ServiceManager] Component: ServiceManager status changed from: c
   reated to: configuring
   [2016-03-04 23:49:41.109] NIO [INFO] [main.ServiceManager] Component: ServiceManager status changed from: c
   onfiguring to: configured
   [2016-03-04 23:49:41.123] NIO [INFO] [main.WebServer] Starting server on 0.0.0.0:8181
   [2016-03-04 23:49:41.226] NIO [INFO] [main.WebServer] Server 0.0.0.0:8181 started on 0.0.0.0:8181
   [2016-03-04 23:49:41.226] NIO [INFO] [main.ServiceManager] Component: ServiceManager status changed from: c
   onfigured to: starting
   ```

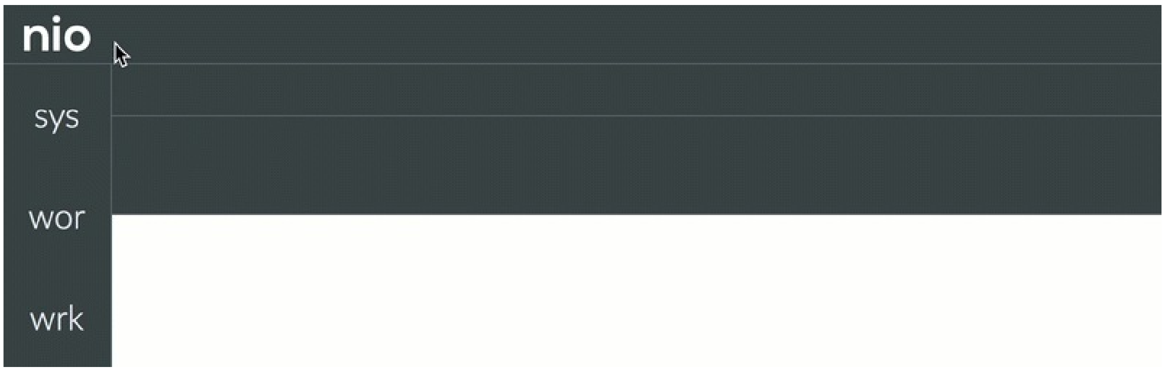
```
[2016-03-04 23:49:41.227] NIO [INFO] [main.ServiceManager] Component: ServiceManager status changed from: s
tarting to: started
```

If you see those logs, nio is up and running. Congratulations!

Add a Local Instance

Once you have a local instance running, you can edit it using the System Designer. Based on the log messages, your nio instance is available at `http://localhost:8181` and you need basic authentication to communicate with the instance.

1. Select the name of your system in the left navigation panel or on the breadcrumb above the contextual toolbar.



2. Click **Add new instance**.
3. Complete the **Create a new cloud instance** window:
   - In the **Instance name** box, enter your instance name.
   - In the **Host name** box, enter **localhost**.
   - In the **Port** box, enter **8181**.
   - Leave the **Access mode** as **basic**.
4. Click **Accept**.
5. Wait for the instance to spin-up and note the name of the new instance on the left.

Note: When you connect to a nio instance, you are communicating with that instance directly from your browser via an XHR request. Hostnames like `localhost` and other internal IP addresses will work. You must have access to the localhost or other IP address from your machine to use the System Designer.

You may see an issue regarding HTTPS and HTTP instances. Since you launched your instance and presumably didn't load any SSL certificates, the instance is accessible only by HTTP. However, if you are logged into the System Designer via HTTPS, then an XHR request going over HTTP is not permitted due to a browser restriction. Instead, log into the designer via HTTP. All of your instances and systems will be the same, except the nio commands to edit these instances won't happen over HTTPS.

Once your instance is loaded and available, you can add services and blocks in the same manner as the cloud instance. Any errors or activity are available in the logs in your terminal that is running nio. When you need to debug a system, a local instance is a useful tool.

Create a Service

1. Select the name of your instance under the system name in the left navigation panel or on the breadcrumb above the contextual toolbar.
2. Click **Add new service**.
3. Complete the **Create new service** window:
   - In the **Service name** box, enter a service name.

   - Leave the **Service type** as **Service**.
4. Click **Accept**.
5. Click **Save** in the toolbar.
6. Click **Edit** in the toolbar.
7. Click **Auto-Start Off** in the toolbar.

Add Blocks

Before you move on, you're going to want to add some blocks to your project. Blocks can be added in the System Designer or from the command line.

nio blocks are installed to instances using the Block Library. The collection of blocks created by nio is also stored in the Block Library at blocks.n.io. You can search for blocks in the System Designer or in the Block Library.

To add a block in the System Designer:

1. Click the service name under the instance name in the left navigation panel or on the breadcrumb above the contextual toolbar.
2. In the **Block library** search box, enter a search term.
3. If the block is not displayed, click **Available**, **Installed**, and **Configured** to search for the block.
4. If the block is not already pre-installed, click the **Install Block** button which resembles a cloud with a down arrow.
5. Drag the block type to the canvas.
6. Name the block.
7. Click **Accept**.

To add blocks manually:

1. From the project root directory, add the relevant block repository into the `blocks/` folder as a submodule. For example, the logger block:

   ```
   git submodule add https://github.com/nio-blocks/logger.git blocks/logger
   ```

   This can also be done with the nio-cli with `nio add logger`.
2. Restart the System Designer.
3. Click the **Block Library** in the upper-right corner.
4. In the Search box, enter the name of the block. As you type, the list is filtered.
5. Drag the your block onto the canvas.
6. Type the name of the block and click **Accept**.

Once you have nio running, you can create many different projects. To guide you through the process, view the tutorials at https://workshops.n.io/.

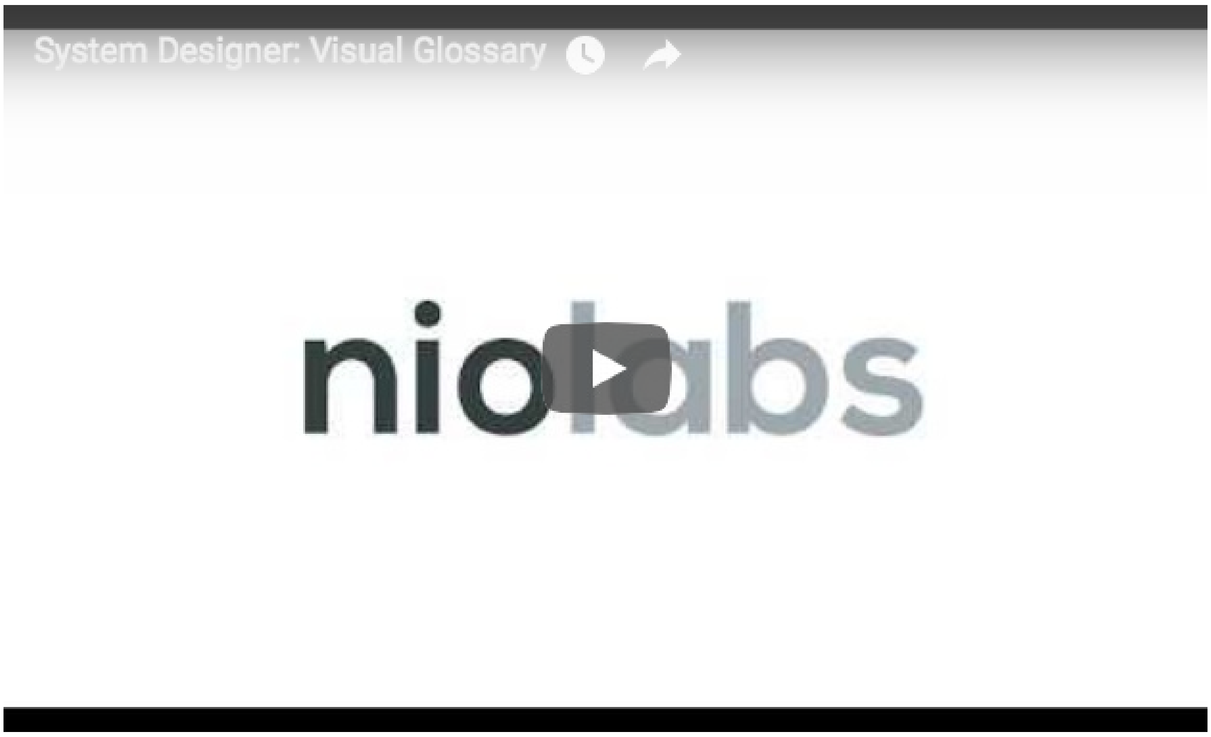
Workshops

Ready to jump in and start creating the next new thing? Head over to the tutorial site at https://workshops.n.io/ and start the Hello nio! tutorial.

System Designer Overview

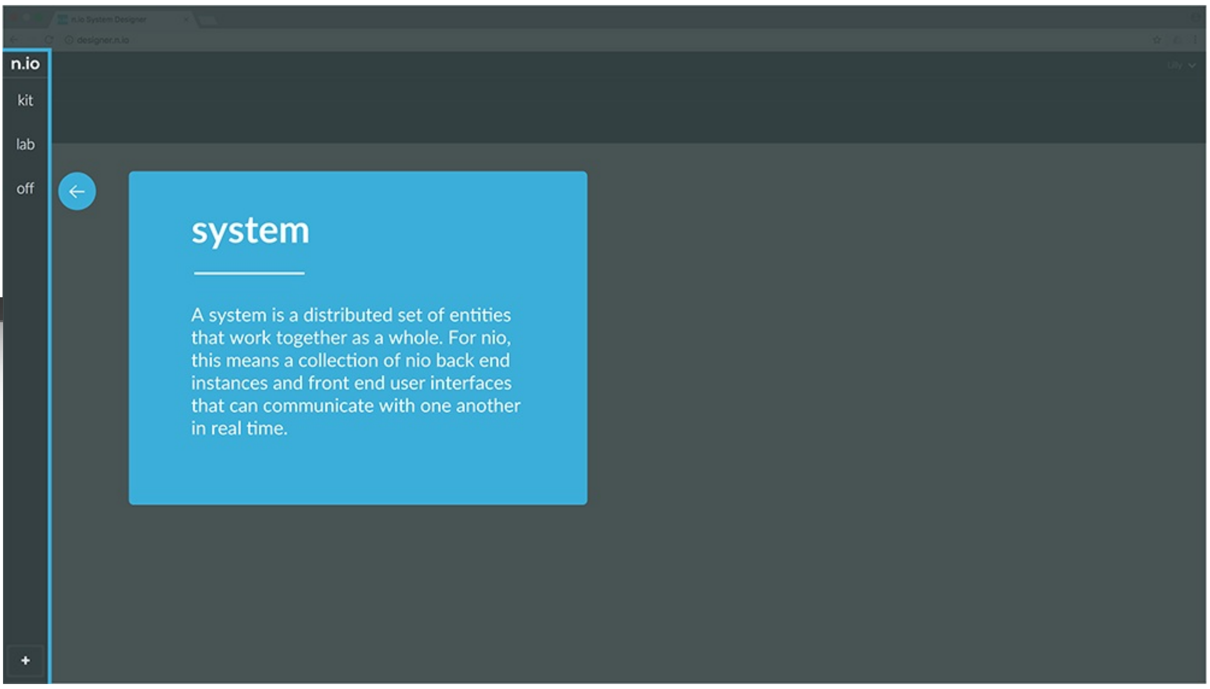
The System Designer is the graphical user interface used to build your nio system. This overview provides you with a quick tour of the user interface including the navigation, work areas, and toolbars.

Note that your screen may look slightly different than the images in the document and video.



System

A list of systems is displayed in a column on the left side of the screen. The designer automatically creates an abbreviation for each system name.



system

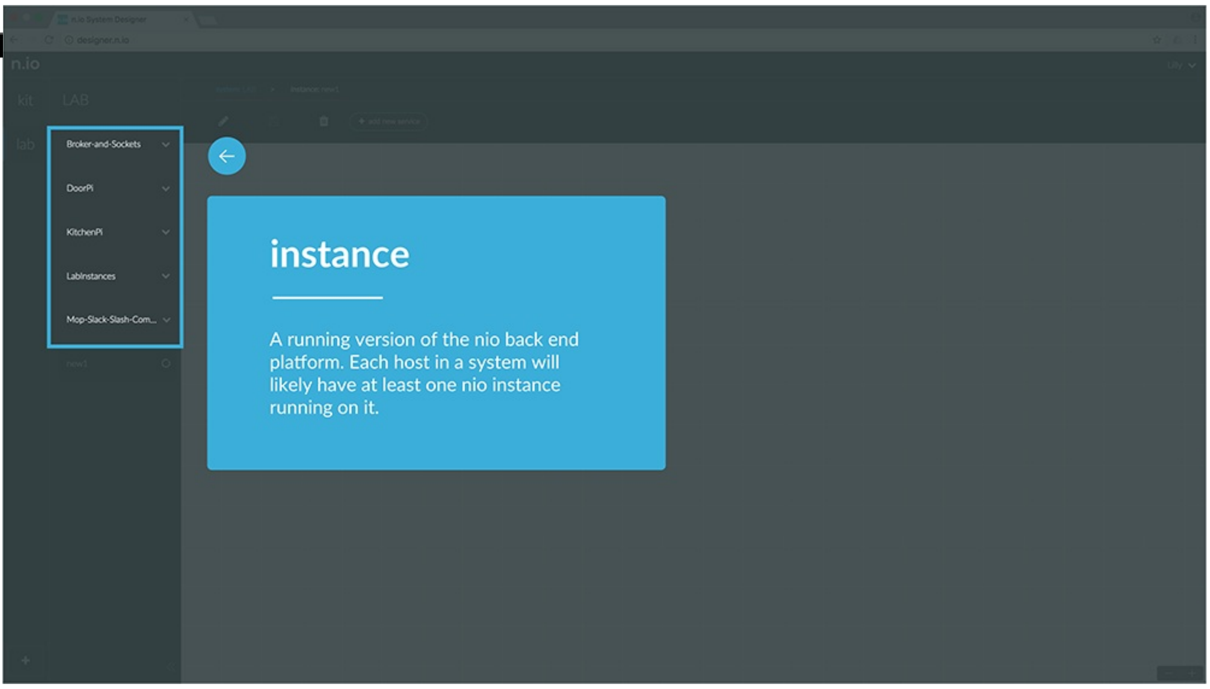
A system is a distributed set of entities that work together as a whole. For nio, this means a collection of nio back end instances and front end user interfaces that can communicate with one another in real time.

To view the system, click the system name.

To create a new system, click the **+** button in the lower-left corner.

Instance

A list of instances is displayed in a column next to the list of systems.



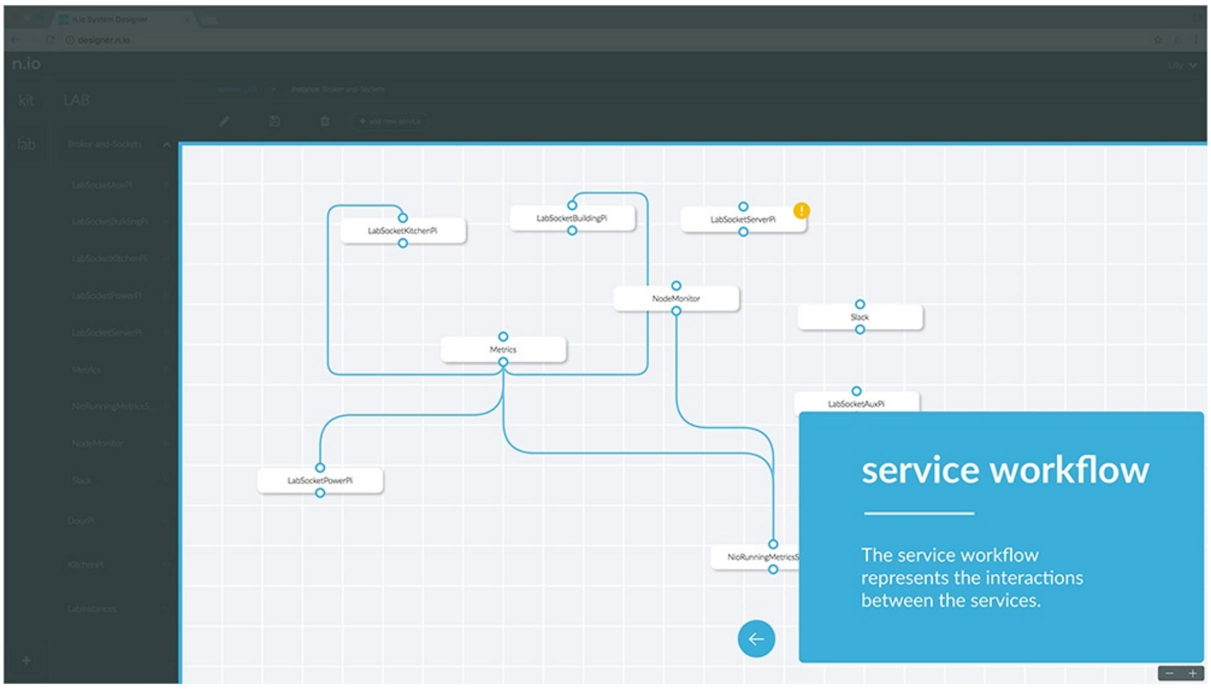
instance

A running version of the nio back end platform. Each host in a system will likely have at least one nio instance running on it.

To create a new instance, click **Create a Cloud Instance** or **Add a New Instance**.

Service Workflow

The service workflow represents the interactions between the services.



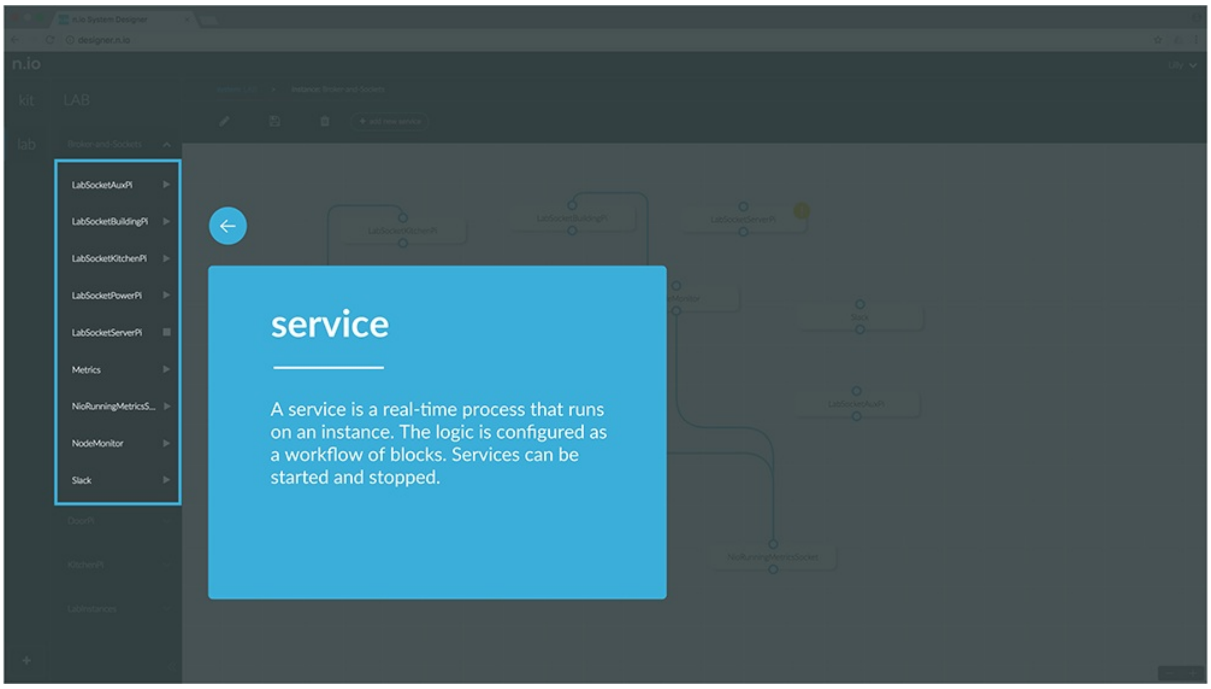
service workflow

The service workflow represents the interactions between the services.

To view the service workflow within the instance, click the arrow next to the name of the instance. The list of services displays below the instance name.

Service

A list of services is displayed below the instance name. Toggle the arrow to hide and display the list.



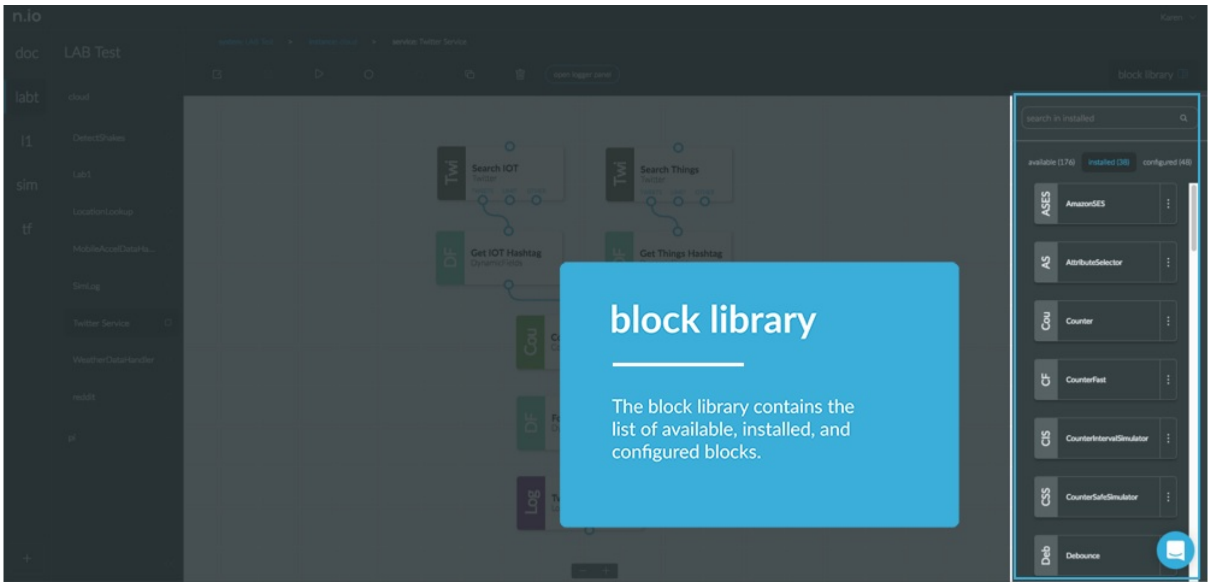
service

A service is a real-time process that runs on an instance. The logic is configured as a workflow of blocks. Services can be started and stopped.

To create a new service, click **Add New Service**.

Block Library

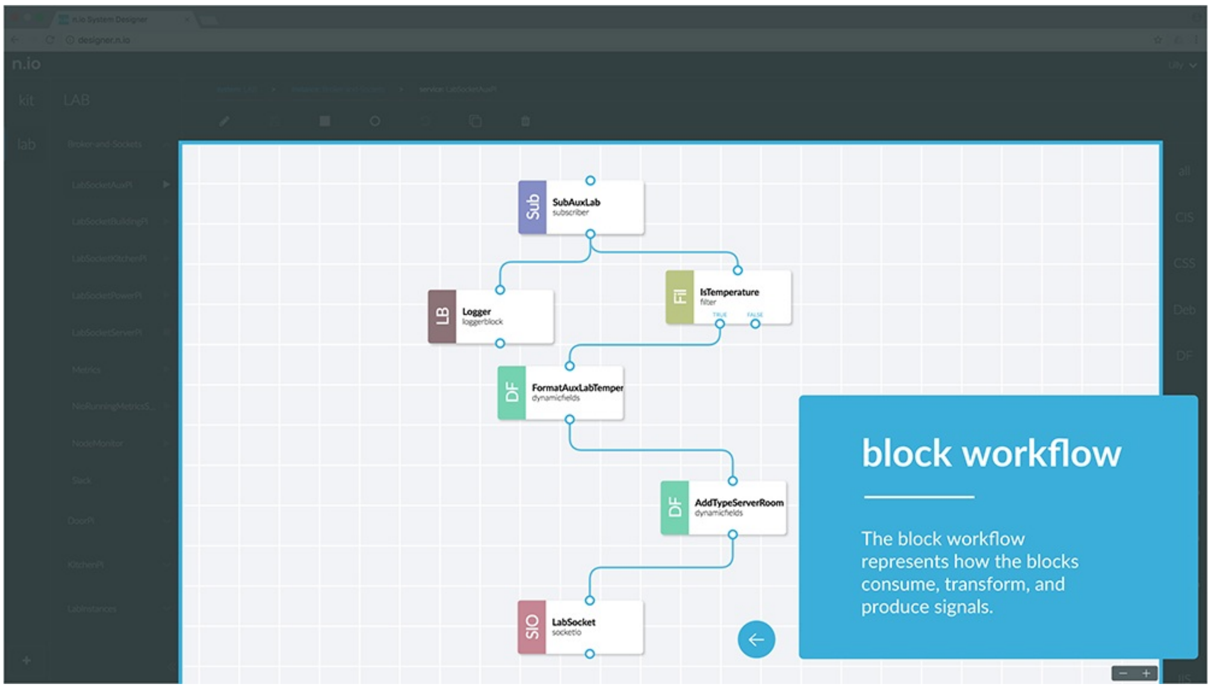
The block library contains a list of blocks.



In the Search box, enter the name of a block. As you type, the list is filtered.

Block Workflow

The block workflow represents the the inputs, outputs, and logic for the workflow.

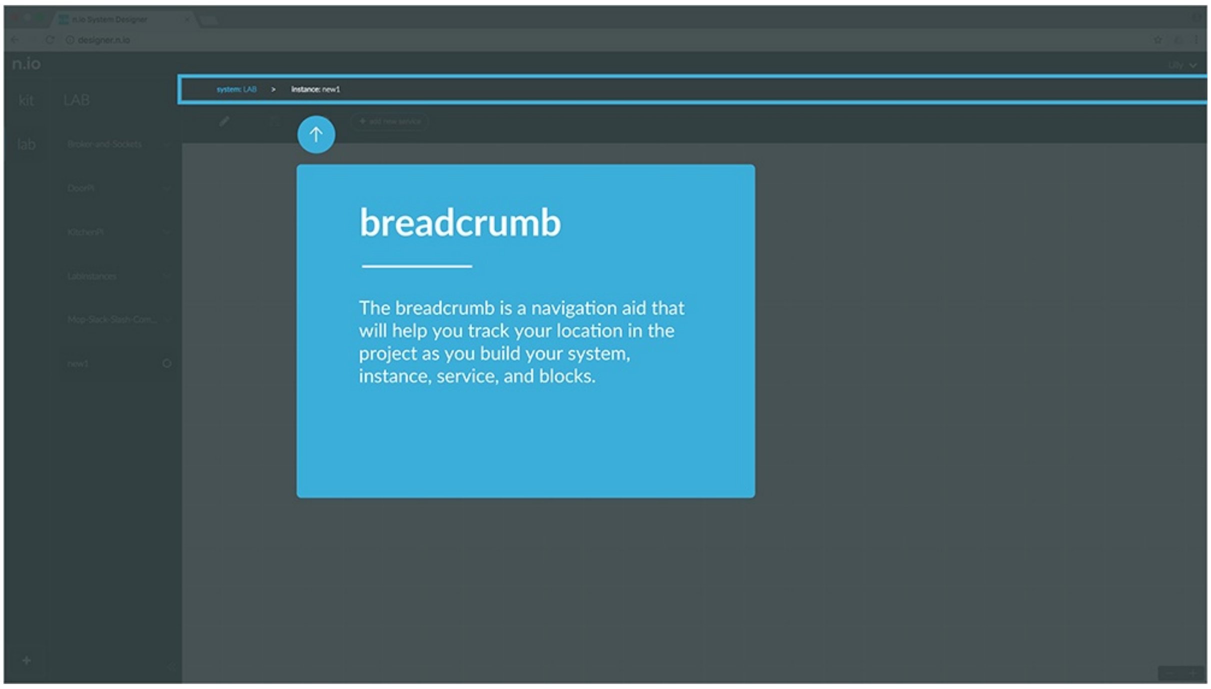


To view the block workflow, click the service name.

To add a block to the service, install and then drag and drop a block from the block library to the canvas.

Breadcrumb

As you travel up and down the levels of your system, note that the breadcrumb changes allowing you to easily identify your location in the system.

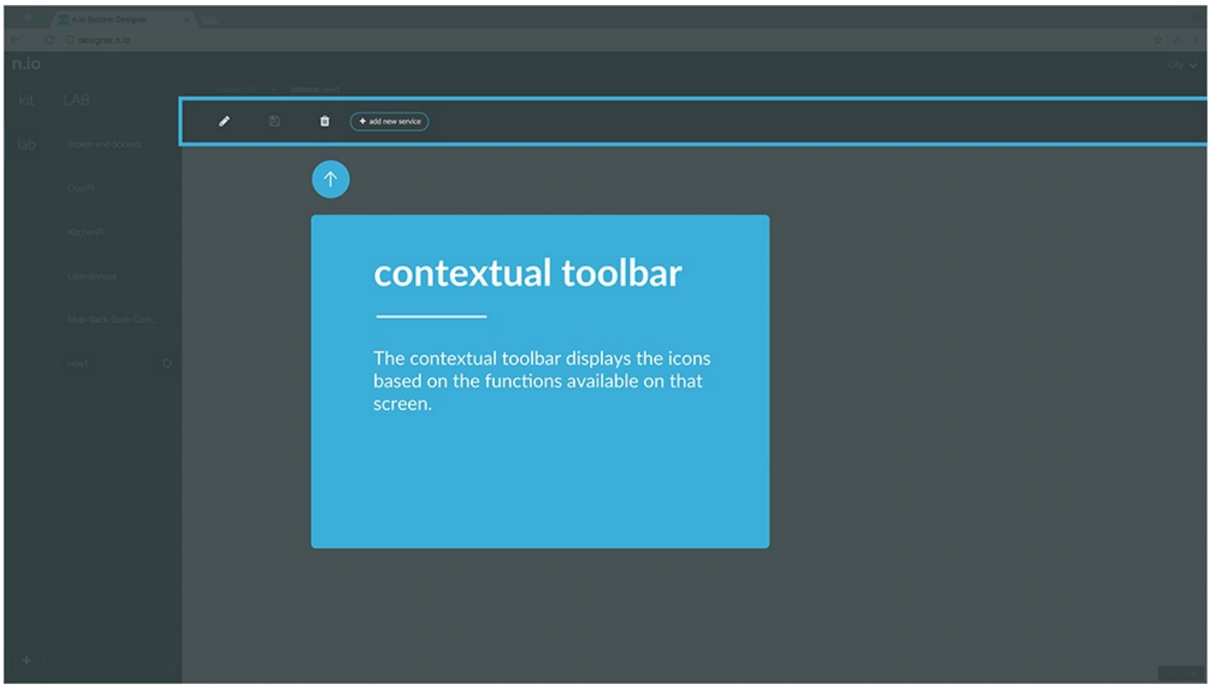


breadcrumb

The breadcrumb is a navigation aid that will help you track your location in the project as you build your system, instance, service, and blocks.

Contextual Toolbar

The contextual toolbar changes based on your location in the project to represent the available functions. Hover over the icon to reveal the name of the icon. Some icons have dual functions where you click to toggle an action, such as starting and stopping a service.



contextual toolbar

The contextual toolbar displays the icons based on the functions available on that screen.

Designer Tasks

System Designer Tasks

The System Designer is the graphical user interface used to build your nio system. This overview provides you with a quick tour of the user interface including the navigation, work areas, and toolbars.

h1 Heading

h2 Heading

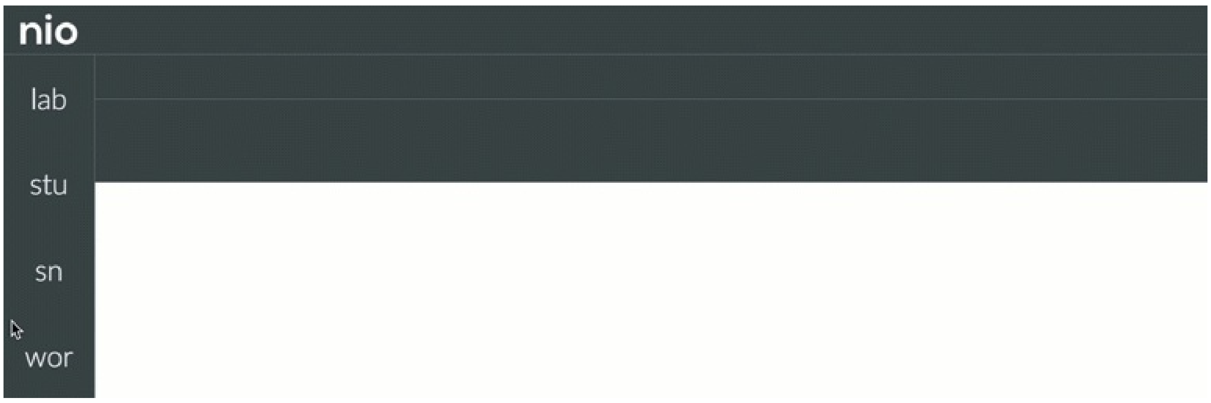
h3 Heading

h4 Heading

h5 Heading

h6 Heading

Project Hierarchy

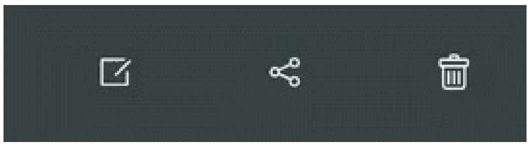


Contextual Toolbar

Icon	Description
|	Auto-Start On/Off
⧉	Clone
</>	Command
🗑	Delete
✎	Edit

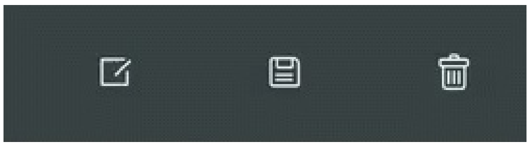
↺	Revert
💾	Save
⋖	Share
▢	Start/Stop

System Toolbar



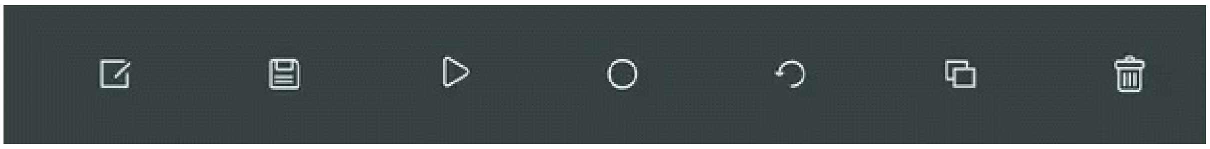
Icon	Label	Description
✎	Edit	Edit system. View Pubkeeper configuration information.
⋖	Share	Share system.
🗑	Delete	Delete system. Instances must be deleted first.

Instance Toolbar



Icon	Label	Description
✎	Edit	Edit instance.
💾	Save	Save instance.
🗑	Delete	Delete instance.

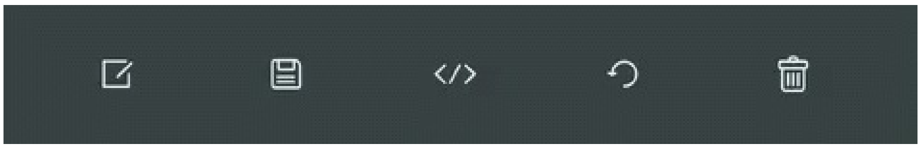
Service Toolbar



Icon	Label	Description
✎	Edit	Edit service. Colorize

💾	Save	Save service.
▢	Start/Stop	Toggle to start and stop service.
|	Auto-Start On/Off	Toggle Auto-Start on and off.
↺	Revert	Revert changes to service.
⧉	Clone	Clone service.
🗑	Delete	Delete service.

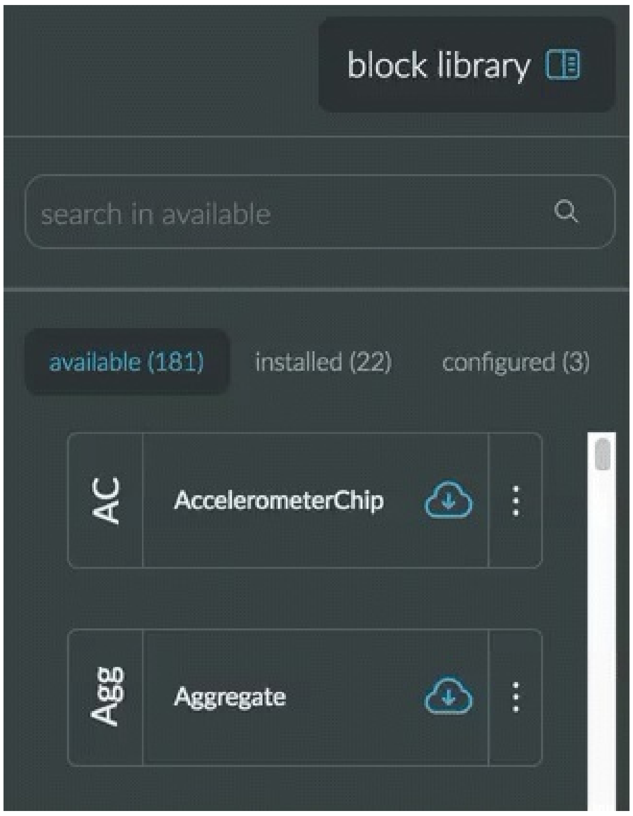
Block Toolbar



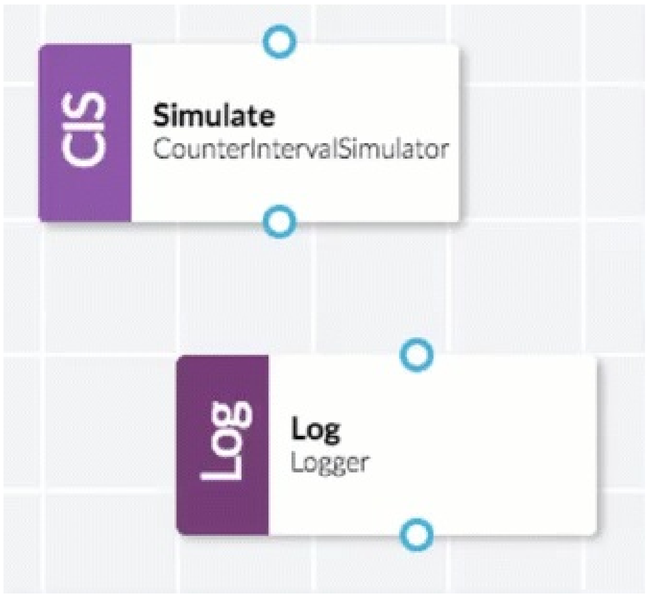
Icon	Label	Description
✎	Edit	Edit block configuration.
💾	Save	Save block.
</>	Command	Command block.
↺	Revert	Revert changes to block including deletion.
🗑	Delete	Delete block.

Blocks

Block Library



Connect and Disconnect blocks



Systems

Create a System

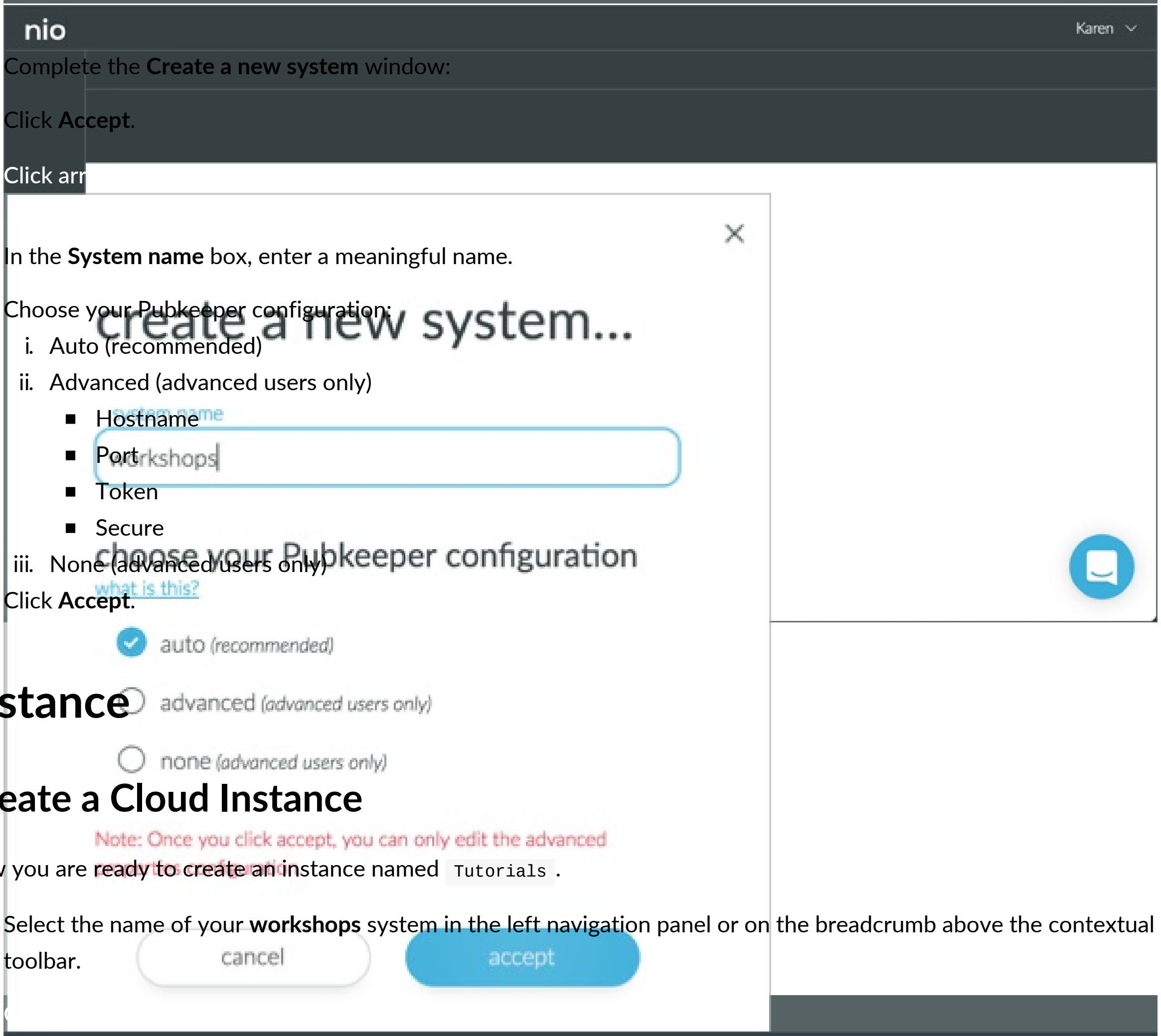
The very first time you open the nio platform, you are literally starting with a blank canvas. This is where start building a system containing instances and blocks.

1. Open the **System Designer** in a new tab.

https://designer.n.io/

2. In the lower-left corner, click the `+` button to build.

Click arrow to expand image


nio
Karen
create a new system...
system name
workshops
choose your Pubkeeper configuration
what is this?
auto (recommended)
advanced (advanced users only)
none (advanced users only)
Note: Once you click accept, you can only edit the advanced
cancel
accept

3. Complete the **Create a new system** window:
4. Click **Accept**.

Click arr

5. In the **System name** box, enter a meaningful name.
6. Choose your Pubkeeper configuration:
   i. Auto (recommended)
   ii. Advanced (advanced users only)
      - Hostname
      - Port
      - Token
      - Secure
   iii. None (advanced users only)
7. Click **Accept**.

Instance

Create a Cloud Instance

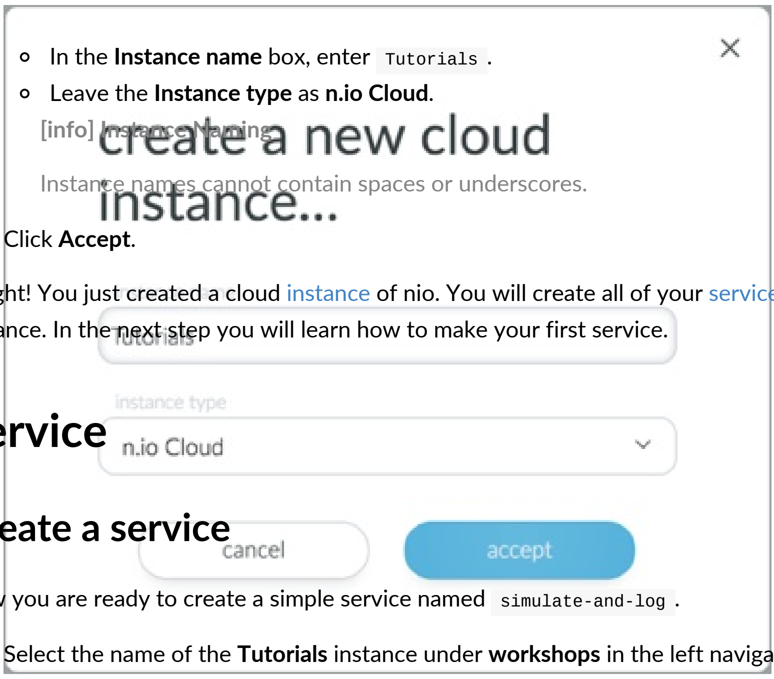
Now you are ready to create an instance named `Tutorials`.

1. Select the name of your **workshops** system in the left navigation panel or on the breadcrumb above the contextual toolbar.

> Click on a hyperlink in the breadcrumb above the toolbar to navigate between the levels of your system. The breadcrumb allows you to visualize your location in the project.

2. Click **Create a cloud instance**.
3. Complete the **Create a new cloud instance** window:

Click arrow to expand image



- In the **Instance name** box, enter `Tutorials`.
- Leave the **Instance type** as **n.io Cloud**.

  [info] Instance Naming

  Instance names cannot contain spaces or underscores.

4. Click **Accept**.

Alright! You just created a cloud instance of nio. You will create all of your services for the tutorials within this cloud instance. In the next step you will learn how to make your first service.

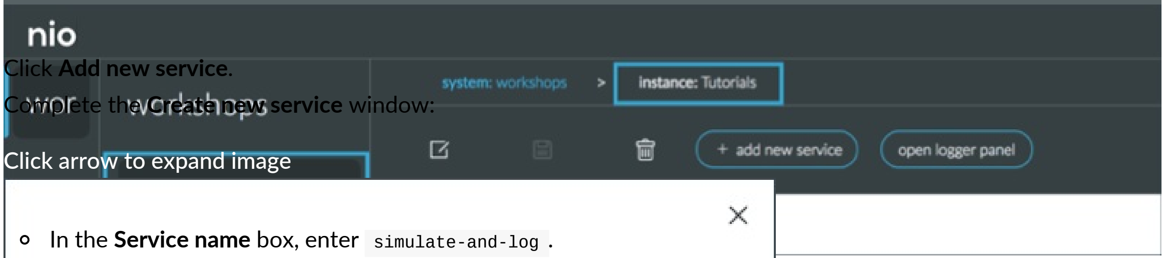
Service

Create a service

Now you are ready to create a simple service named `simulate-and-log`.

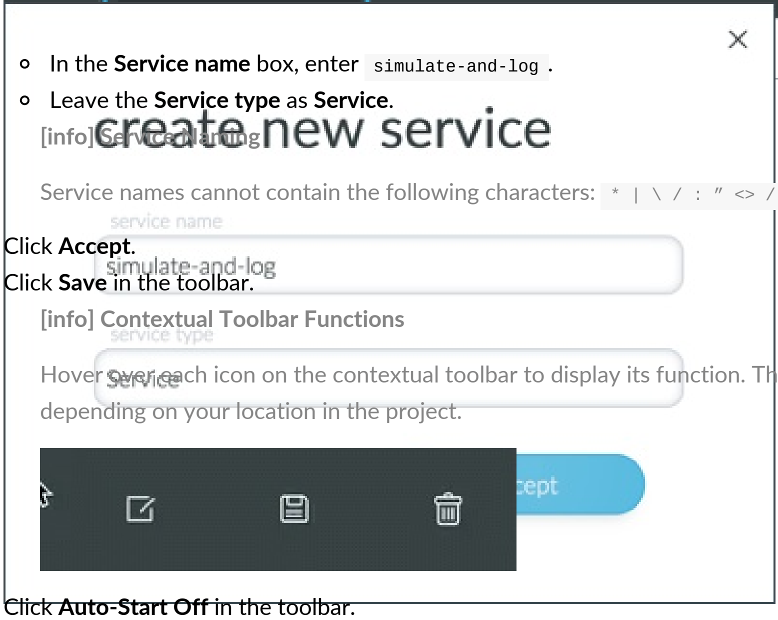
1. Select the name of the **Tutorials** instance under **workshops** in the left navigation panel or on the breadcrumb above the contextual toolbar.

   Click arrow to expand image



2. Click **Add new service**.
3. Complete the **Create new service** window:

   Click arrow to expand image



- In the **Service name** box, enter `simulate-and-log`.
- Leave the **Service type** as **Service**.

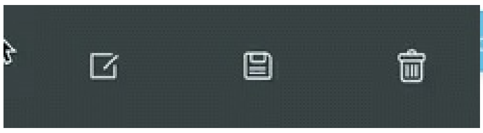
  [info] Service Naming

  Service names cannot contain the following characters: `* | \ / : " <> / ?`

4. Click **Accept**.
5. Click **Save** in the toolbar.

   [info] Contextual Toolbar Functions

   Hover over each icon on the contextual toolbar to display its function. The available functions change depending on your location in the project.



6. Click **Auto-Start Off** in the toolbar.

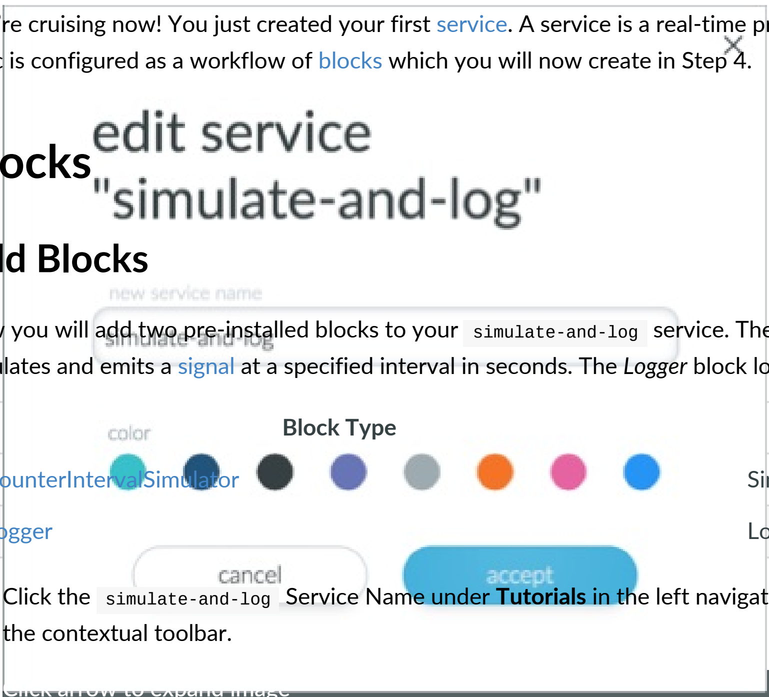
   > By default, services are set to auto-start when a nio instance is started.

7. Click **Edit** in the toolbar.
8. Optional. Select one of the colored circles to represent this service and click **Accept**.

   > By default, the color of a service is white. Color coding your services can help you keep your instances organized when working on larger projects with multiple instances.

   Click arrow to expand image

You're cruising now! You just created your first service. A service is a real-time process that runs on an instance. The logic is configured as a workflow of blocks which you will now create in Step 4.



Blocks

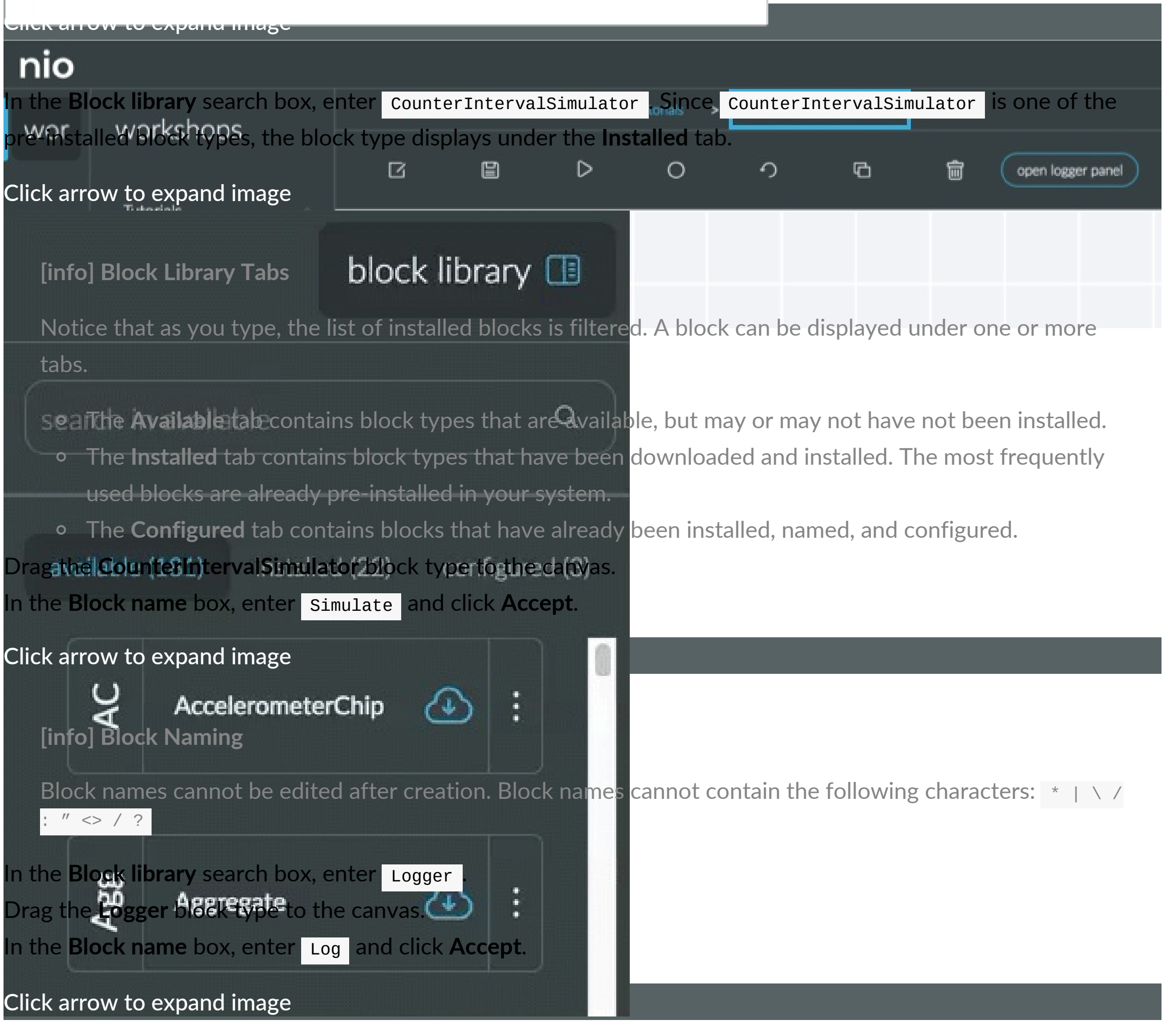
Add Blocks

Now you will add two pre-installed blocks to your `simulate-and-log` service. The *CounterIntervalSimulator* block simulates and emits a signal at a specified interval in seconds. The *Logger* block logs each incoming signal.

Block Type	Block Name
CounterIntervalSimulator	Simulate
Logger	Log

1. Click the `simulate-and-log` Service Name under **Tutorials** in the left navigation panel or on the breadcrumb above the contextual toolbar.

Click arrow to expand image



2. In the **Block library** search box, enter `CounterIntervalSimulator`. Since `CounterIntervalSimulator` is one of the pre-installed block types, the block type displays under the **Installed** tab.

Click arrow to expand image

[info] Block Library Tabs

Notice that as you type, the list of installed blocks is filtered. A block can be displayed under one or more tabs.

- The **Available** tab contains block types that are available, but may or may not have not been installed.
- The **Installed** tab contains block types that have been downloaded and installed. The most frequently used blocks are already pre-installed in your system.
- The **Configured** tab contains blocks that have already been installed, named, and configured.

3. Drag the **CounterIntervalSimulator** block type to the canvas.
4. In the **Block name** box, enter `Simulate` and click **Accept**.

Click arrow to expand image

[info] Block Naming

Block names cannot be edited after creation. Block names cannot contain the following characters: `* | \ / : " <> / ?`

5. In the **Block library** search box, enter `Logger`.
6. Drag the **Logger** block type to the canvas.
7. In the **Block name** box, enter `Log` and click **Accept**.

Click arrow to expand image

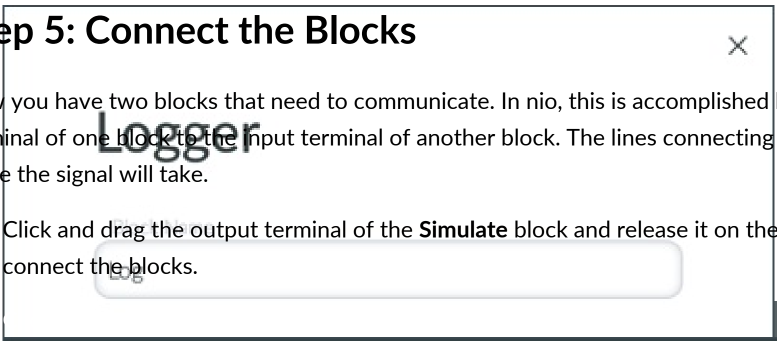
To learn about all the blocks nio has to offer, visit https://blocks.n.io.

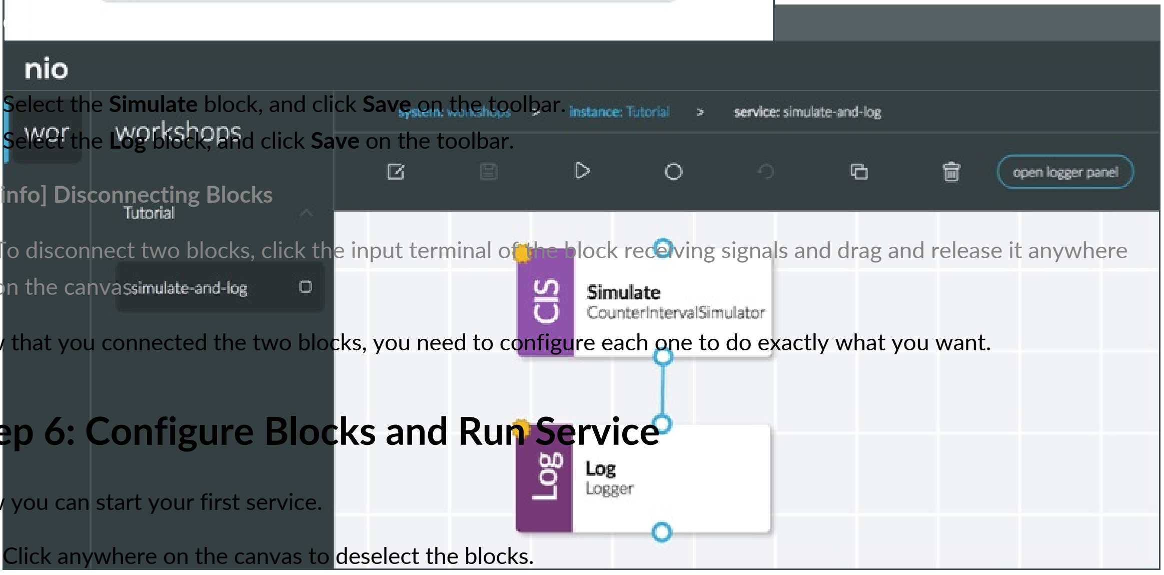
Way to go! You are now familiar with how to add, name, and save the most basic building "blocks" of a nio system - no pun intended! The first block will simulate data and the second will log that data. Let's make that happen!

Step 5: Connect the Blocks

Now you have two blocks that need to communicate. In nio, this is accomplished by simply connecting the output terminal of one block to the input terminal of another block. The lines connecting the blocks together configure the route the signal will take.

1. Click and drag the output terminal of the **Simulate** block and release it on the input terminal of the **Log** block to connect the blocks.





2. Select the **Simulate** block, and click **Save** on the toolbar.
3. Select the **Log** block, and click **Save** on the toolbar.

> [info] Disconnecting Blocks
>
> To disconnect two blocks, click the input terminal of the block receiving signals and drag and release it anywhere on the canvas.

Now that you connected the two blocks, you need to configure each one to do exactly what you want.

Step 6: Configure Blocks and Run Service

Now you can start your first service.

1. Click anywhere on the canvas to deselect the blocks.
2. Click **Start** on the toolbar.
3. Click **Open logger panel** to view the logs. The logs are displayed with the default settings.

Click arrow to expand image

Icon	Description
☁	Download block
◧	Block Library

Some title here

Any content here

> Use this for help for beginners--informational messages that add to the learning experience. Do not place steps in this message.

INVALID CHARACTERS

Info styling

> [info] For beginners/tips
>
> Use this for help for beginners--informational messages that add to the learning experience. Do not place steps in this message.

Warning styling

> [warning] For warning

> Use this for warning messages.

Danger styling

> **[danger] For danger**
>
> Use this for danger messages. Use sparingly.

Success styling

> **[success] For success**
>
> Use this for success messages. Also, for items needed in tutorials before you begin.

Common C++ Naming Conventions

Types start with upper case: MyClass. Functions and variables start with lower case: myMethod. Constants are all upper case: const double PI=3.14159265358979323;. C++ Standard Library (and other well-known C++ libraries like Boost) use these guidelines: Macro names use upper case with underscores: INT_MAX. Template parameter names use camel case: InputIterator. All other names use snake case: unordered_map.

1. Do this (1)
2. Do this (1)
    i. Do this (1)
    ii. Do this (1)

Use this for help for beginners--informational messages that add to the learning experience. Do not place steps in this message.

- Level one (**)
- Level one (**)
    - Level two (**)
        - Level three (**)
            - Level four (**)

Use this for help for beginners--informational messages that add to the learning experience. Do not place steps in this message.

1. Level one (1)
2. Level one (1)
    i. Level two (1)
        i. Level three (1)
        ii. Level four (1)

Use this for help for beginners--informational messages that add to the learning experience. Do not place steps in this message.

1. Level one (1)
2. Level one (1)
    - Level two (**)
        i. Level three (1)
            - Level four (**)

Use this for help for beginners--informational messages that add to the learning experience. Do not place steps in this message.

- Level one (-)
- Level one (-)
    - Level two (-)

    - Level three (-)
      - Level four (-)

Use this for help for beginners--informational messages that add to the learning experience. Do not place steps in this message.

1. Level one (1)
2. Level one (1)
   - Level two (-)
     1. Level three (1)
        - Level four (-)

> Use this for help for beginners--informational messages that add to the learning experience. Do not place steps in this message.

- [ ] Open
- [x] Closed

Browser Support

To have the best experience with the System Designer on your computer, use a supported, up-to-date browser. You will have access to the latest features along with improved security and performance.

Level of Browser Support

Browser	Level of Support
Chrome	Recommended. Supports all System Designer features and functionality.
Firefox Safari Microsoft Edge	Supports all System Designer features and functionality. There may be minor visual imperfections.
Internet Explorer	Supports all System Designer features and functionality. There may be moderate visual imperfections.

Supported Versions

The current, or most recent, version of all supported browsers is supported.

Required Browser Settings

To use the System Designer, enable cookies and JavaScript® in your browser.

Using a Virtual Browser

Browsers in virtual environments, such as Citrix® and VMware®, might not support all the System Designer functionality. For the best experience, use a supported browser on your local computer.

Using an Unsupported Browser

If you use the System Designer on an older or unsupported browser, some features might not work.

Mobile

The System Designer is not supported on mobile devices.

Blocks

Blocks are the units of nio that send, receive, and process signals. What makes a block a block?

A block has a life cycle and at least one input terminal or one output terminal. Optionally, you can configure properties within the block. A block also has the potential to accept commands from the block REST API.

In summary, a nio block is:

- A unit of functionality that runs in a service
- A process that goes through the six stages of the block life cycle: configuring, configured, starting, started, stopping, stopped, (and hopefully not an error)
- A functional piece of code that receives streaming input signals and/or emits streaming output signals
- A commandable through the block REST API
- A Python class that inherits from `nio.block.base.Block`

Properties

Properties are the configuration fields of a block.

In the System Designer, you modify the properties in the configuration panel. Properties may include rules governing how the block should process signals.

Inputs/Outputs

A block receives or sends signals through the input and output terminals to allow interblock interaction. In the System Designer, you draw connectors between the terminals to show the signal flow.

Signals are implemented as a collection of key-value pairs. Signals are always passed between blocks inside of a list. See Understanding Signals.

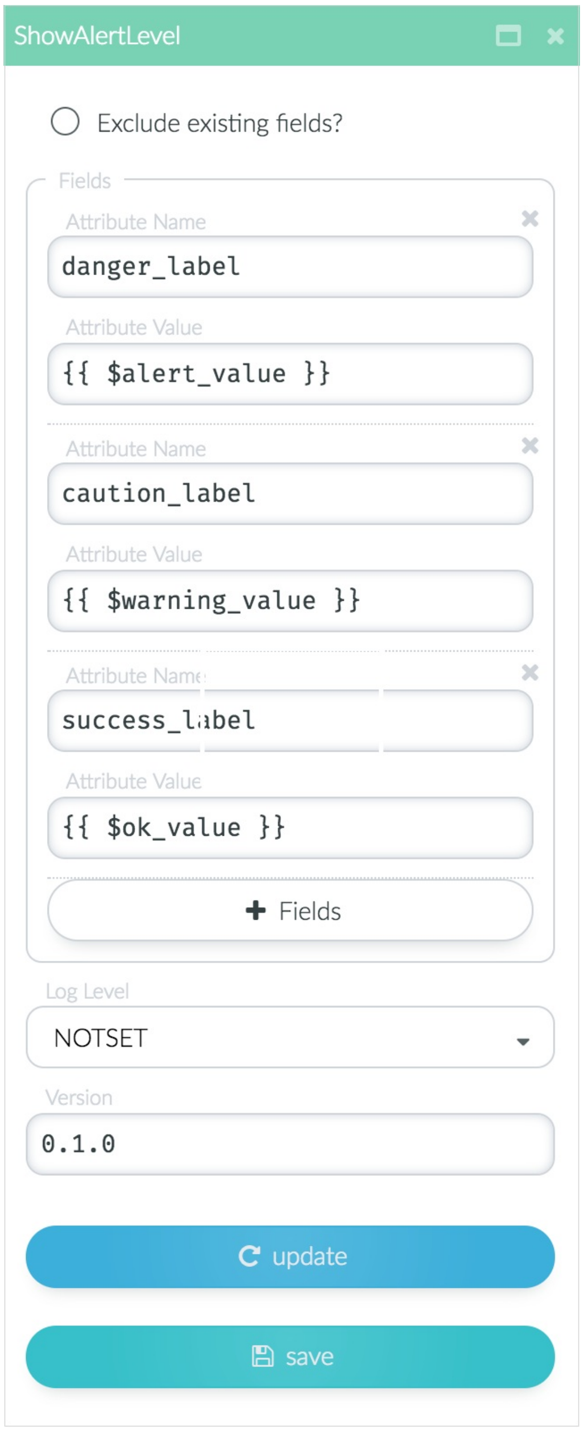
Commands

Commands allow a user to interact with a running instance of a block. You can use a command to inspect the internal state of a block. When you develop a block type, you can expose certain methods in your block class as commands. This allows you to run functions inside the instance of your block from the System Designer.

Block Properties

Block properties are the configuration fields of a block.

In the System Designer, when you double-click a block, the configuration panel displays the block's properties. The configuration panel for a *Modifier* block is shown below. It displays four properties: **Exclude Existing Fields**, **Fields**, **Log Level**, and **Version**.



Attribute Name: danger_label
Attribute Value: {{ $alert_value }}
Attribute Name: caution_label
Attribute Value: {{ $warning_value }}
Attribute Name: success_label
Attribute Value: {{ $ok_value }}
Log Level: NOTSET
Version: 0.1.0

The input fields are block properties that can be edited, updated, and saved to create a configured block.

The property configuration of the block makes it unique. A **block type name**—*Filter*, *Modifier*, *Counter Interval Simulator* —defines the block type and the configurable fields. (In the System Designer, this name is abbreviated in a vertical bar on the left side of the block.) The unique **block name**, which you define when dragging a new instance of the block type onto the canvas of the System Designer, saves a specific configuration of the block and defines how the block acts on signals. All blocks with the same **block name** share a configuration. For example, if you change the configuration of your "isWatering" *Filter* block, the configuration of every "isWatering" block will be be updated to match. The unique **block name** is the link to a block's property configuration. For this reason, it is good to give your blocks descriptive names that reflect their configured properties.

Property Types

In the example above, Exclude Existing Fields is a checkbox, Fields is a list of input pairs, Log Level is a drop-down menu, and Version is a series of three numbers separated by periods (0.1.0).

The different types of inputs in the configuration panel represent different property types.

In the above example, Exclude Existing Fields is a "BoolProperty" that can be true or false and is represented by a checkbox. Log Level is a "SelectProperty" that enumerates available options and is represented by a drop-down menu. Fields is a "ListProperty" and is represented by a list of items that can be defined by the user with `+ Fields` to add items and `x` to remove items. In this example, there are three key-value pairs defined in the list. Each item in a list can also have a type, in this case, a dictionary defined by a "PropertyHolder" property made up of two properties represented by the Attribute Name and Attribute Value input areas. And finally, Version is a special "VersionProperty" that takes three integers separated by periods.

Some properties (the base Property property) can take any type of input while others require specific types of input. The IntProperty can only take integers while the SelectProperty requires a selection from a specific list. If your property input is filled with the incorrect type of content, your block will not configure and your service will not run.

In the *Modifier* block shown above, the `ShowAlertLevel` block is configured so that it will enrich the signal with three new fields, one each for danger, caution, and success. The structure of the fields added to the signal would look similar to the following. (For illustration purposes, the expressions in curly braces have not been evaluated.)

```
[
  {
    "attribute_name": "danger_label",
    "attribute_value": {{ $alert_value }}
  },
  {
    "attribute_name": "caution_label",
    "attribute_value": {{ $warning_value }}
  },
  {
    "attribute_name": "success_label",
    "attribute_value": {{ $ok_value }}
  },
]
```

Expressions

Expressions are used to dynamically define properties. Many property types can accept expressions. Expressions are found inside the double curly braces. In the *Dynamic Fields* block configuration shown above, you can see example expressions in the Attribute Value inputs. Learn more about how to create expressions in nio Expressions.

Environment Variables

Another syntax that nio uses in property configuration is the double-square-bracket notation for an environment variable.

```
[[YOUR_ENV_VAR]]
```

Environment variables are used when you want your instance to function in different environments or you do not want to expose an access token or other secret.

Your environment variable can be used alone, or you can embed it, with the double-square-bracket notation, inside of strings and expressions.

To embed you environment variable inside of single brackets, you need to include a space between the single bracket and the double brackets of the environment variable

```
[ [[MY_ACCESS_TOKEN]] ]
```

nio Expressions

An expression is an element of code that needs be evaluated to return a result. You can use nio expressions to dynamically define the value of a property when the signal enters a block.

nio expressions are modeled on string interpolation in other dynamic languages. The piece of code between the double curly braces is evaluated as Python code.

```
{{ <code goes here> }}
```

Once the code is evaluated, the value will be replaced by the curly braces and its contents.

```
"{{1 + 5}} dogs went to the park"
-> "6 dogs went to the park"
"{{1 + 5}} dogs went to the {{'p'+'ark'}}"
-> "6 dogs went to the park"
```

Inner dictionaries need spaces before and after the curly braces.

```
'{{ {"a": 1} }}' -> {"a": 1}
```

Signal Property Syntax `$`

Use the `$` syntax to access signal properties.

```
# given a signal s where s.v1 == 6
"{{$v1}} dogs went to the park"
-> "6 dogs went to the park"
```

You can also combine the two approaches.

```
"My favorite Integer is {{$v1 if isinstance($v1, int) else 'not an Integer…'}}"

# given a signal s where s.v1 == 6
-> "My favorite Integer is 6"

# given a signal s where s.v1 == 'A'
-> "My favorite Integer is not an Integer…"
```

Raw Signal

You can access the raw signal itself, rather than just the attributes, with a lone `$`. As long as the character following the `$` is not a valid Python identifier, the `$` will evaluate to the incoming signal.

Escape Characters

To include a $ character in a string literal inside a piece of code, use the backslash (`\`) character to escape it. Similarly, escaping the `}}` or `{{` with a `\` causes the braces to be treated as strings rather than as delimiters.

```
"Code snippets are delimited by \{{ and \}}"
```

```
-> "Code snippets are delimited by {{ and }}"
```

Conditionals

Conditional expressions can be formatted in Python in one line.

```
# given a signal s == { v1: 23, v2: "zabow!", v3: "sad trombone…"}

"When there are {{$v1}} things, we say {{$v2 if $v1 == 23 else $v3}}"
-> "When there are 23 things, we say zabow!"
"When there are {{$v1}} things, we say {{$v2 if $v1 == 42 else $v3}}"
-> "When there are 23 things, we say sad trombone…"
```

Incorporating Libraries

The following libraries are imported by default and can be used in expressions:

- datetime
- json
- math
- random
- re

You can import other libraries from your Python installation.

```
{{ __import__('module_name').method_name() }}
{{ __import__('numpy').amax([2, 1, 4]) }}
```

Examples

Bracket notation can be used.

```
# given a signal s where s.v1 == {'who': 'Baron Samedi'}
"{{$v1['who']}} and the Jets"
-> "Baron Samedi and the Jets"
```

Methods on signals can be called.

```
# given a signal s where s.get_val() == 'foobar'
"Opened it with a {{$get_val()}}"
-> "Opened it with a foobar"
```

```
# given a signal s where s has the attribute v1 but not v2
"{{hasattr($, 'v1') and hasattr($, 'v2')}}"
-> False
```

Check that a signal attribute exists.

```
# given a signal s where s.v1 raises AttributeError, s.v2 == 'Cogito' and a default value of None
"{{ ($v2 + ' ') if (hasattr($, 'v1') or hasattr($, 'v2')) else ''}}ergo sum"
-> "Cogito ergo sum"
"{{ ($v2 + ' ') if (hasattr($, 'v1') and hasattr($, 'v2')) else '' }}ergo sum"
-> "ergo sum"
```

Default libraries can be used in expressions.

```
# math operations are allowed by default (as are regex, datetime, random, and json)
"{{ math.sin(math.radians(90)) }}"
-> 1.0
```

Syntax error

```
"If you don't close the brackets {{1 + 5"
-> "If you don't close the brackets {{1 + 5"
```

Raw signal with method

```
# given a signal s == { v1: 23, v2: "zabow!", v3: "sad trombone…"}
{{ $.to_dict() }}
-> {v1: 23, v2: "zabow!", v3: "sad trombone…"}
```

Block Development

To connect to a new framework, library, or custom piece of hardware not currently included in the Block Library at blocks.n.io, you can develop your own custom block.

You can adopt one of two philosophies when creating a block:

1. Blocks are simple, generic, and reusable. Examples of generic blocks can be found at both the Block Library at blocks.n.io and the nio-blocks repository on GitHub. Blocks developed with this approach have a single function and you can chain blocks together to develop complex behaviors.

2. Blocks can be complex or built for a single use. You would not likely use this block for more than the service it was defined for. For example, a block might contain a script to run a series of calculations for a model. Instead of a complicated chain of blocks, a single block may encompass the entire model. You could also design a single block to parse a Tweet in a unique way that would not be reused.

If the function you need does not exist in Block Library at blocks.n.io, you can easily create a custom block.

Begin development of your block with the block template.

Block Template

The Block Template is a base template used to create a new block type with the recommended files and file structure.

How to Use

If you need to first create a project directory, the `project_template` repository at https://github.com/nioinnovation/project_template provides a good starting point.

Clone the Block Template

1. From the command line, navigate to the `/blocks` folder in your project.

   ```
   cd nio/projects/<project_name>/blocks
   ```

2. Clone the block template. Use a meaningful name to describe the block's function.

   ```
   git clone --depth=1 https://github.com/nio-blocks/block_template.git <new_block_name>
   ```

3. Navigate to the new block folder.

   ```
   cd <new_block_name>
   ```

Rename the Appropriate Files

1. Rename `example_block.py` to the name of your new block. Note: The filename should end in `_block.py` or `_base.py`. Use `_block.py` if it is intended to be a discoverable block and `_base.py` if it is meant to have common, reusable base functionality but not be discoverable.

   ```
   mv example_block.py new_name_block.py
   ```

2. Edit this file and rename `class Example(Block)` to `class New_Name(Block)`. Note: You do not need to include `Block` in the class name since this is implied in the block name.

3. Rename `BLOCK_README.md` to `README.md` and update the contents of this file.

   ```
   mv BLOCK_README.md README.md
   ```

4. In the `/tests` folder, rename `test_example_block.py` to match the class name of your new block.

   ```
   mv test_example_block.py new_name_block.py
   ```

5. Edit this file and rename `class TestExample(NIOBlockTestCase)` to `class TestNew_Name(NIOBlockTestCase)`.

Track Your New Block on GitHub

1. Create a new GitHub repository. For convenience, use the same name as `<new_block_name>`.
2. Copy the unique URL for your new repository to your clipboard.
3. Remove the tracking link to the original template repository.

   ```
   git remote remove origin
   ```

4. Stage the new files.

   ```
   git add -A
   ```

5. Reset ownership to yourself.

   ```
   git commit --amend --reset-author -m "Initial Commit"
   ```

6. Add tracking to the new remote repository using the URL you copied.

   ```
   git remote add origin <new_repo_url>
   ```

7. Push to a branch (usually `master`).

   ```
   git push --set-upstream origin master
   ```

File Reference

example_block.py
This is the block code. Additional Python classes and files are definitely welcome. If the file contains a block class, make sure the filename ends with `_block.py`. If the file represents a base block (a block type that is not intended to be discoverable by itself), rename the filename to end with `_base.py`.

requirements.txt
Lists required Python dependencies. The file is installed by pip when the block is installed. To install the dependencies, enter `pip install -r requirements.txt`.

release.json
Contains release data for one or more blocks.

spec.json
Defines the specification for a block type. This is the metadata which is used for block discovery.

tests/test_example_block.py
The `tests` folder contains a sample test file. Be sure to submit accompanying unit tests with your blocks.

Developing Your Block

After cloning the block template repository, you are ready to develop your block.

A good resource for new block development is the nio base block along with other blocks that have functionality similar to the one you would like to develop.

The basic elements of a block that you will define are its properties, commands, and inputs and outputs.

Any required Python dependencies for your block can be added to your `requirements.txt` file and will be automatically installed when your block is installed.

Other elements of a block to keep in mind include block patterns, the nio framework (including discoverability), and mixins.

Once you have developed your block, you will want to test and document your block.

Base Block Class

All nio blocks inherit from the base block class. The first import in the block template's `example_block.py` is `nio.block.base`. If you explore the code inside `nio.block.base`, you'll find explanatory docstrings for each method—including methods to override in your custom block—along with higher-level context.

An important principle to remember when developing your block is that signals are passed as lists.

Methods to Override

The following methods from the base block are designed to be overridden:

life cycle management

- `configure`: at the end of the `configure` method, the block is ready to receive signals. If an exception is raised during configure, the service will not start.
- `start`: during start, a block begins to send out signals. This method needs to eventually return so that the block status can change to “started”. For this reason, anything that runs continuously, should be run in a new thread.
- `stop`: after stop, the block stops sending out signals and cancels jobs.

signaling

- `process_signals(<list of signals>, input_id)`: receives input signals.
- `notify_signals(<list of signals>, output_id)`: emits signals from the block. This method isn't intended to be overridden, but should be called by the block to send out signals. For example, you will usually call `notify_signals` at the end of your `process_signals` method.

Current nio Blocks

An additional resource for developing your custom block is the nio-blocks library. Search the nio-blocks library for a block that has similar functionality to the block you need. Explore its properties, methods, commands, inputs, outputs, mixins, and any modules imported from the framework.

Properties

Block properties are declared as class attributes and have a property type. For example, to declare a configurable speed property as an `IntProperty` type

```
speed = IntProperty(title='Speed', default=30)
```

And in the *File Reader* block type a `file` property is declared with a `FileProperty` type.

```
file = FileProperty(title='File', default='/tmp/file.txt')
```

To obtain the value of a block property, call the property with a function invocation. For example, you can get the value of the speed property defined above with

```
 self.setSpeed(self.speed())
```

If you want to use the `$` syntax to refer to your signal, pass in `signal` when you call the property. The *File Reader* block accesses the value of the `file` property with

```
 file = self.file(signal).value
```

Property Types

Block properties can include the following types:

- **BoolProperty**
  One of two mutually exclusive options.
- **StringProperty**
  A valid Python string.
- **IntProperty**
  A Python integer.
- **FloatProperty**
  A Python float.
- **FileProperty**
  A file stream.
- **ListProperty**
  List properties hold a list of object types which contain properties themselves or can be nio types such as IntType.
- **ObjectProperty**
  Object types contain properties themselves.
- **SelectProperty**
  Select properties enumerate a list of options.
- **TimeDeltaProperty**
  A duration property expressing the difference between two date, time, or datetime instances.
- **VersionProperty**
  A version property with the format `(major.minor.build)`.
- **Property**
  A property that can assume any type.

Commands

Commands are declared as decorators.

```
from nio.block.base import Block
from nio.command import command

@command("emit")
class MyBlock(Block):
  # properties and block methods here…

  # emit method that will be invoked when the "emit" command is received
  def emit(self, foo=None):
        self._emit_job(foo) # where you have set up your own _emit_job method…
```

Inputs and Outputs

Inputs and outputs are declared as decorators.

```
from nio.block.base import Block
from nio.block import output

@output('false', label='False')
@output('true', label='True')
class MyBlock(Block):
  # properties and block methods here…

  def process_signals(self, signals):
    true_result, false_result = self._filter_signals(signals) # where you have set up your own _filter_signal
s method…

    # add any other functionality to process_signals

    self.notify_signals(true_result, 'true')
    self.notify_signals(false_result, 'false')
```

Testing Your Block

nio blocks are not meant to be run as stand-alone Python modules, so testing can be a challenging process. nio provides a couple of tools and offers best practices to make your testing easier.

NIOBlockTestCase

When building your tests, use the `NIOBlockTestCase`. The test case extends the `unittest.TestCase` and uses the same infrastructure as the base class. However, `NIOBlockTestCase` also provides a number of helpful methods for testing blocks. Here's a real-world example from our internal block repositories:

```
from nio.testing.block_test_case import NIOBlockTestCase
from nio.signal.base import Signal
from ..your_block import YourBlock

class TestYourBlock(NIOBlockTestCase):

     def test_some_feature(self):
         blk = YourBlock()
         self.configure_block(blk, {
             'int_prop': 23,
             'obj_prop': {
                 'p1': 'foo',
                 'p2': 'bar'
             }
         })
         blk.start()
         blk.process_signals([Signal()])
         blk.stop()
         # your assertions here…
```

setUp/tearDown

Just like the `unittest.TestCase`, nio supports the setUp/tearDown pattern. This is a great place to do any initialization and/or cleanup that will be required across every test. Do not repeat yourself! Be aware, though, that the block provides crucial initialization and finalization in `NIOBlockTestCase.setUp/tearDown`. If you override either of these methods, you need to call the method in the parent class at the top of your method.

Helper Methods

- **configure_block(block, block_properties)** - The process of configuring and initializing blocks manually is somewhat nuanced (and not something we want you to worry about). We provide this method to configure your block instance semi-automatically. Just pass the block object itself and a dictionary containing any block properties you want to configure (and the associated values).
- **assert_num_signals_notified(num, block=None)** - This method provides access to the total number of signals notified over the course of the current test. If `block` is not `None`, then you will receive the number of signals notified by that block over its lifetime.
- **last_signal_notified(output_id)** - This method returns the last signal that was notified from a particular output. If an output_id is not specified, it will return the last signal notified from any output on the block.

Overridable Methods

- **get_test_modules()** - By default, `NIOBlockTestCase` automatically initializes the logging, threading, scheduler, and security modules. However, you can customize this by overriding this method and returning a list of strings corresponding to the particular modules you want to initialize.
    - logging
    - threading
    - scheduler
    - security
    - communication
    - persistence
    - web
- **signals_notified(signals, output_id)** - This method gets called every time signals are notified in your tests. If you'd like to record something in the test case, trigger an event, or perform some aggregation when that happens, override this method. One common use is to add `self.signals = defaultdict(list)` to `setUp` and

```
def signals_notified(self, signals, output_id):
    self.signals[output_id].extend(signals)
```

Events

Blocks are not required to behave synchronously, and sometimes you may want to wait for an event (after instantiating or configuring a block) before proceeding with the tests. Rather than sleeping for a prescribed amount of time (asynchronous processes can be fickle and unpredictable from machine to machine). You can extend the block and add one of Python's Event objects to signify readiness.

```
class EventBlock(YourBlock):
    def __init__(self, event):
        super().__init__()
        self.e = event

    def configure(self, context):
        super().configure(context)
        self.e.set()
```

Now, instead of instantiating `YourBlock` in the tests, instantiate `EventBlock` passing an instance of `Event` to its constructor.

```
from threading import Event
e = Event()
blk = EventBlock(e)
self.configure_block(blk, {'p1': 23})
e.wait(2)
```

Using the `EventBlock`, your test will wait until `YourBlock.configure` returns control to the method on the child class. Your test will never proceed until `EventBlock.e` is set.

Mocking

Patching and mocking are extremely useful concepts in software verification; this is especially relevant when the modules in question interact with external resources (such as APIs and OS services). We won't go into too much details of mocking right now, but the Python documentation contains great material on the subject. We recommend using these concepts liberally; in fact, in many cases you won't have much choice.

As you progress, one thing you may notice is that `unittest.mock.patch` doesn't play nice with relative module paths. This makes patching a method at the class or module level difficult. One solution is to directly import the object using `unittest.mock.patch.object`

```
from unittest.mock import patch, ANY
from ..queue_block import Queue

@patch.object(Queue, '_load')
def test_it(self, load_patch):
    ...
    load_patch.assert_called_once_with(ANY)
```

Again, you don't necessarily have to construct your tests in this manner; however, we've found this practice to be more convenient and less prone to user error than others.

Mocking Persistence Module

To mock `load` the persistence module:

```
class TestPersistenceBlock(NIOBlockTestCase):

    def test_persist_load(self):
        blk = Block()
        with path('nio.modules.persistence.default.Persistence.load') as load:
            load.return_value = 'i was persisted'
            self.configure_block(blk, {})
```

To mock `save` :

```
from unittest.mock import  MagicMock

class TestPersistenceBlock(NIOBlockTestCase):

    def test_persist_save(self):
         blk = Block()
         self.configure_block(blk, {})
         blk.persistence.save = MagicMock()
```

nio Modules

`NIOBlockTestCase` configures the following nio modules by default: `['logging', 'scheduler', 'security', 'threading']` . If your block test case needs to use any other nio modules, you must specify by implementing the `get_test_modules` method.

If your test case uses persistence, enter the following code:

```
class TestBlock(NIOBlockTestCase):

    def get_test_modules(self):
        return super().get_test_modules() + ['persistence']
```

You can override the default configuration of modules by implementing `get_module_config_*` . This ensures that the test case uses the `default` implementation of the persistence module.

```
class TestBlock(NIOBlockTestCase):

    def get_module_config_persistence(self):
        return {'persistence': 'default'}
```

Documenting Your Block

There is a standard way to document a nio block. Much of this documentation can be generated automatically using the `nio-cli`.

Once you have built your block, navigate to the root of your project directory and type

```
nio buildrelease <block repo name>
```

This will create a `release.json` file with meta information about your block.

To create your `spec.json` file, type

```
nio buildspec <block repo name>
```

Navigate into your `spec.json` file and manually add a text string description to any key labeled "description". You will need to add a description to the block and its properties, commands, inputs, and outputs.

```
"description": "List of attribute names and corresponding values to add to the incoming signals."
```

Once your `spec.json` is complete, you are ready to build your block's README. Navigate into the block folder and type

```
nio buildreadme
```

This will populate the README. Manually add any dependencies in this format

```
Dependencies
------------
- face_recognition
- numpy
- opencv-python
```

You can also add example code and other helpful information at the bottom.

Finally, remain in your block directory and run

```
nio blockcheck
```

to check and lint your block. It will check PEP8 styles, confirm you have added descriptions to your `spec.json`, and check your `README.md`, `release.json`, and class and file names.

Base Block Pattern

Implementing a base block pattern is useful in complex configurable blocks. In this pattern, configure the options that are not block-specific in a non-discoverable base block. The blocks that do the work and process signals will inherit from the base block.

For example, in a block that accesses an external API, you can use a base block to set up the base URL, and then create discoverable blocks to access each specific endpoint in the API. A base block pattern is easier to maintain and reinforces the "one function per block" philosophy.

The nio Framework

When you develop nio blocks, you use the nio framework. The framework holds all the classes required to create your block as well as the functionality to tie everything together. Think of the framework as a toolshed of useful tools for working with nio.

Block Context

In the configure method, blocks are passed context about themselves and the environment in which they run. Block developers should refer to the following information:

- **block_router** (BlockRouter): The router in which the block runs. The router must be able to handle signals notified by its blocks.
- **properties** (dict): The block properties (metadata) that will be deserialized and loaded.
- **hooks** (Hooks): Hooks are used by the service internally to subscribe to certain overall nio instance lifecycle events. It is not advised to use or rely on these hooks in your blocks.
- **service_name** (str): The name of the service the block belongs to.
- **command_url** (str): The URL at which this block can be commanded. This URL will not have host or port information, as that may be different based on the public/private IP. For example, "/services/ServiceName/BlockAlias/".
- **mgmt_signal_handler** (method): The method used to publish management signals.

Discoverability

By default, a class is marked as `discoverable` to permit the system to identify and register the class. Marking a class as `not_discoverable` produces the opposite effect.

To mark a class as not_discoverable, use the parameter-less decorator `@not_discoverable`.

```
from nio import not_discoverable, Block
@not_discoverable
class MyBlock(Block):
    pass
```

Base blocks do not need to be discoverable. If a block does not consume or emit signals, include the `@not_discoverable` decorator.

Mixins

Mixins are not blocks. Instead, mixins add commonly used functionality to existing blocks. You can add functions such as **persistence**, **group-by**, or **retry** to blocks. Mixins are shared on GitHub in the mixin repository.

Mixins follow the Python mixin model, thus any block mixins need to be extended prior to extending the base block class. You can see an example of using the persistence and group-by mixins in the Buffer block.

```
from nio.block.base import Block
from nio.block.mixins import Persistence, GroupBy

class Buffer(Persistence, GroupBy, Block):
```

Docstrings inside each mixin provide more information on each mixin's functionality and arguments.

Deployment

Designing a stable nio system means having reliable mechanisms for deployment and maintenance. The deployment of nio systems is very flexible, but there are a few best practices to follow.

Git for Version Control

nio projects can be defined as files in a file system. Git permits you to use a standard version control system to record changes to a project. The project template repository provides a good starting point to model how you store a project in a git repository. With git, you can use different branches for development and production.

Different Environments

You should use staging and other test environments to run nio projects outside of production. Using nio environment variable files is a good way to achieve this. For example, your project could have two environment variable files `stage.env` and `prod.env` that define how nio should run in the respective environment. You can run nio using a different environment variable by using the `-e` flag of `nio_run`.

```
nio_run -e stage.env
```

Sometimes, it doesn't make sense to run an individual service in a non-production environment. To achieve this, create an environment variable such as `SHOULD_RUN_SERVICE` and set it to `yes` or `no` depending on the environment. Then, you can set the service's auto start value in the service configuration to the result of the variable.

```
{
  "name": "ServiceName",
  "auto_start": "[[ SHOULD_RUN_SERVICE ]]"
}
```

Service Unit Testing

Testing is a necessary part of designing a robust system. To ensure your block or service configurations are performing correctly, you need to set up service unit tests.

Use the the Service Unit Test Framework to define unit tests for a service's behavior. Service unit tests are written in Python code and make use of the Python unittest module.

If you installed the project template from the repository, then you are ready. If not, complete the following steps:

1. Install the `jsonschema` Python package. This allows you to validate publishers and subscribers in services.
2. Clone the nio Service Units Test repository into your project directory as a submodule.
3. Create a `tests` directory to store your own custom service unit tests.
4. Set up a test class based on the Service Unit Test readme file.
5. Execute the service tests using a Python test runner.

   ```
   py.test tests
   ```

Service Design Patterns

You define the logic of your entire system by designing services and configuring and connecting blocks. The nio service design language is very flexible and allows you to design services in many different ways. However, it is often helpful to follow the suggested patterns in this section of the documentation.

Understanding Signals

Signals are the pieces of information that are passed from block to block throughout your system. They are the fundamental messages type that nio understands. Structurally, signals are key-value objects.

```
{
  "name": "John Doe",
  "age": 34
}
```

That signal has two attributes, a `name` and an `age`.

Lists of Signals

Signals are passed from block to block. You define your systems by designing how signals should move from block to block throughout the system. However, what is actually happening is that **lists** of signals are being passed between blocks. If a block notifies only one signal, then the next block will receive a list of signals that contains that signal.

Lists of signals are an important concept to understand. Blocks will typically operate independently on an entire list of incoming signals. This means that blocks act on the entire list of incoming signals rather than the individual signals themselves. When a block only receives one list of signals as input, you can safely assume that you will only get one list of signals as output.

For example, let's imagine a filter block that filters out a stream of odd numbers:

```
{
  "name": "Only Odds",
  "type": "Filter",
  "condition": "{{ $num % 2 == 1}}"
}
```

We can call `process_signals` on that block with a list of the following signals:

```
[
  { "num": 1 },
  { "num": 2 },
  { "num": 3 },
  { "num": 4 },
  { "num": 5 }
]
```

then the block would output the following list:

```
[
  { "num": 1 },
  { "num": 3 },
  { "num": 5 }
]
```

In other words, the block will only call `notify_signals` once for the incoming list, even though three signals would be notified. It would not call `notify_signals` three times--once for each signal out.

Paring Down Lists of Signals

If you have a list of signals that you want to condense into fewer signals or even one signal, there are a few blocks that will help. The join block is commonly used.

The `Join` block goes through every signal in an incoming list and consolidates them into a single signal based on the specified criteria. For example, if your have your with the numbers 1 through 5 from the previous example, you can shrink the list of five signals into one signal containing two attributes--one for odds and one for evens.

```
{
  "name": "Split Odds and Evens",
  "type": "Join",
  "key": "{{ $num % 2 }}",
  "value": "{{ $num }}"
}
```

This configuration (based on the configuration of `key` \ will create a new signal with two attributes.

```
{
  0: [2, 4],
  1: [1, 3, 5]
}
```

Group By

By default, a block operates over an entire list of signals. However, sometimes you need to perform the block's operation multiple times over a subset of the incoming list. For example, you may want to pass around a list of all employees at a company and want to reduce the list down to a signal that contains all of their names.

```
[
  {
    "name": "Jim Halpert",
    "department": "sales"
  },
  {
    "name": "Angela Martin",
    "department": "accounting"
  },
  {
    "name": "Stanley Hudson",
    "department": "sales"
  }
]
```

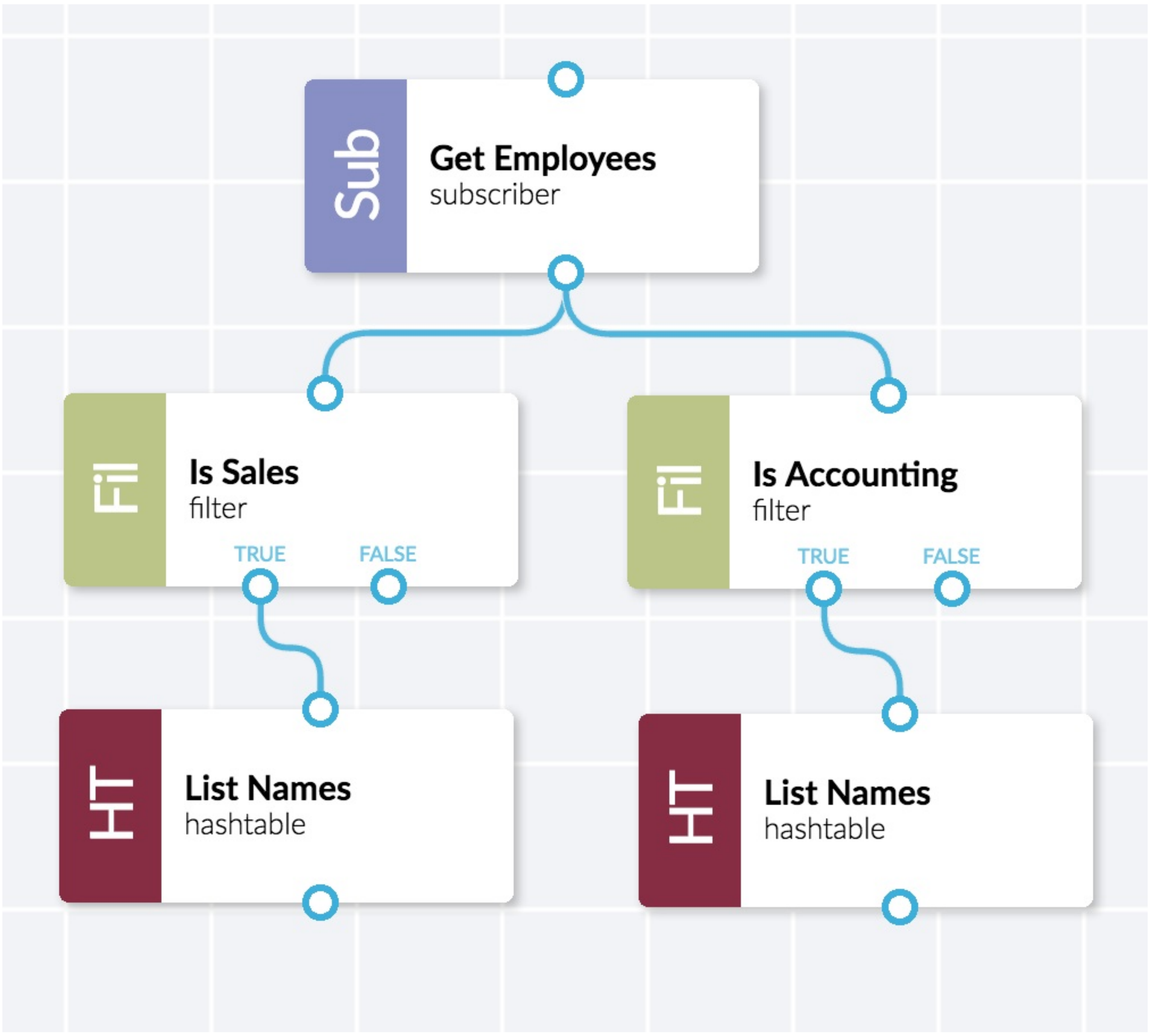
You can use a `Join` block to create a signal where the key is the department name and the value of that key is a list of names in the department.

```
{
  "name": "List Names",
  "type": "Join",
  "key": "{{ $department }}",
  "value": "{{ $name }}"
}
```

The block configuration results in a single output signal.

```
{
  "sales": ["Jim Halpert", "Stanley Hudson"],
  "accounting": ["Angela Martin"]
}
```

That looks great, but what if you want to have a separate signal for each department, but still have a list of names. One inefficient option would be to filter the stream based on the department, and then put the individual streams into copies of the `Join` block from before. The following image displays this example. Note that this is **not** the advised way to do this.



There are several downsides to this approach.

- It is not dynamic based on department. If we add another department, we have to add a new `Filter` block and replicate the `HashTable` again.
- You lose your original list. When the chain started you had a list of all employees, and now you have different lists floating around. You would have to merge the streams back together to get a list of all the employee names.
- It is very tedious and repetitive. We have the same blocks used over and over again.

The Group By mixin groups the signals into smaller lists first before performing the same action. This is very similar to the SQL `GROUP BY` operator. Fortunately, the `HashTable` block uses the Group By mixin, so you can eliminate the `Filter` blocks and rely on only one `HashTable` block. The block will group the hash table functionality by the department of the employee. Since you don't need the department name as the signal attribute key anymore, you can hard code that to be a string `"names"` instead.

```
{
  "name": "List Names",
  "type": "HashTable",
  "key": "names",
  "value": "{{ $name }}",
  "group_by": "{{ $department }}"
}
```

You added an attribute to your block configuration indentifying the value to group by. The output of the block is a list of two signals, one for each department. Again, remember that since you only have one list of inputs, you can only have one list of outputs. Even though there are two signals being notified, they will be notified together in the same list.

```
  [
    {
      "group": "accounting",
      "names": ["Angela Martin"]
    },
    {
      "group": "sales",
      "names": ["Jim Halpert", "Stanley Hudson"]
    }
  ]
```

Understanding how Group By works and when to use it can save a lot of headaches and repetition in nio services allowing you to build very powerful services without many blocks. In general, if you are frequently repeating the same blocks when designing in nio, there is probably a more efficient way to perform the task.

Publishing and Subscribing to Data

Designing systems with nio includes designing how information, or signals, moves through the system. nio recommends using the Pub/Sub mechanisms to move data from one service to another. The following communication blocks are located at blocks.n.io:

- Publisher
- Subscriber

Topic Tree

When using the `Publisher` and `Subscriber` blocks, you need to decide which topic you wish to publish to or subscribe to. The information organization of an entire nio system can be viewed as a hierarchical topic tree.

Publishers

Publishers publish data to a specific topic, denoted by a dot-separated string. The following is an example of publisher topics and the resulting topic tree that would be generated in the system.

Publisher Topics:

1. California.San Francisco.Temperature
2. California.San Francisco.Population
3. California.Los Angeles.Temperature
4. California.Population
5. Colorado.Broomfield.Temperature

These topics can be visualized as the following tree:

```
                                        ┌────────┐
                                        │  Root  │
                                        └────────┘
                                             │
                            ┌────────────────┴──────────────────┐
                            ▼                                   ▼
                     ┌────────────┐                      ┌────────────┐
                     │ California │                      │  Colorado  │
                     └────────────┘                      └────────────┘
                            │                                   │
              ┌─────────────┴──────┬─────────────┐              │
              ▼                    ▼             ▼              ▼
       ┌────────────┐       ┌────────────┐┌────────────┐ ┌────────────┐
       │    San     │       │    Los     ││    (4)     │ │ Broomfield │
       │ Francisco  │       │  Angeles   ││ Population │ │            │
       └────────────┘       └────────────┘└────────────┘ └────────────┘
              │                    │                            │
              │                    │                            │
       ┌──────┴──────┐             │                            │
       ▼             ▼             ▼                            ▼
┌─────────────┬────────────┐┌─────────────┐              ┌─────────────┐
│     (1)     │    (2)     ││     (3)     │              │     (5)     │
│ Temperature │ Population ││ Temperature │              │ Temperature │
└─────────────┴────────────┘└─────────────┘              └─────────────┘
```

Subscribers

Subscribers can subscribe to a single topic or also portions of the topic tree through a similar dot-separated string that publishers use. The only difference is that subscribers can supply wildcards represented as `*` and `**`. The single asterisk will match one and only one level of the topic tree. The double asterisk will match zero or more levels of the topic tree. Below are some examples of subscribers and the corresponding publishers they would receive data from according to the publisher examples above.

- Subscribe: `California.*.Temperature` - Publishers: 1, 3
- Subscribe: `California.*.Population` - Publishers: 2 - Publisher 4 is not matched since the single asterisk is used
- Subscribe: `California.**.Population` - Publishers: 2, 4 - Publisher 4 is matched this time because we supplied a double asterisk which can match zero levels of the tree.
- Subscribe: `**.Temperature` - Publishers: 1, 3, 5
- Subscribe: `California.*.*` - Publishers: 1, 2, 3
- Subscribe: `California.*.**` - Publishers: 1, 2, 3 - Publisher 4 is still not matched since the single asterisk forces us down one more level
- Subscribe: `California.**` - Publishers: 1, 2, 3, 4

Hints in the System Designer

One of the benefits of using the nio pub/sub mechanism is that the System Designer understands the topic tree as well. If your service has a `Publisher` or `Subscriber` block with a topic, the designer can show you services that either subscribe to or publish to that topic, respectively. It also permits the System Designer to draw the graph relationship between services and instances.

Simulators

Most of the time the first service you build with nio involves simulating signals using a `Simulator` block and then logging that data using a `Logger` block. That is the "Hello, World" service of nio. However, simulators are much more powerful and do much more than generate random data. This document attempts to define and explain other use cases for simulators.

Understanding Simulators

Before we get into too many use cases, you must understand how simulators work and the naming conventions. There are many types of simulator blocks, but once you understand the naming you'll be a pro at using them. The basic idea is that a simulator block consists of a trigger and a generator. Each simulator block will be named `<Generator><Trigger>Simulator`. For example, if you want to use the `Counter` generator and the `Interval` trigger, then you can use the `CounterIntervalSimulator` simulator block.

Triggers

Triggers define when a simulator should notify signals. Triggers are only concerned with the timing of simulators.

- `Interval` - Notify signals at a fixed time interval (for example, every 5 minutes)
- `Cron` - Notify signals at certain times of the day, similar to how you would configure a cron job

Generators

Generators define the shape and contents of the signals that are notified.

- `Counter` - Each new signal generated will have an attribute that is incremented by some value
- `File` - Each new signal generated is pulled from a file of signals
- `Identity` - Each signal generated is an empty signal `{}`

Driving Another Block

The general design of nio is that **signals** represent data and information and **blocks** represent functions and transformations. However, blocks need to know when and how to perform these functions. Some blocks may need to do an action periodically while others may need to wait for some external action, such as a button press. Rather than trying to define that logic in the block's configuration, you can use incoming signals to trigger the action instead. This also allows the block to "tap in to" the incoming signals and perhaps change its behavior based on the content of the signals. For example, rather than making a fixed HTTP request every interval, a block could send the contents of the incoming signal as part of the HTTP payload instead.

In some cases, you will not care about the contents of the incoming signal, but want to perform an action. For example, if you wanted to poll the Twitter API every five minutes for new tweets, you would need an incoming signal to the Twitter block when you wanted to poll. Since you don't care about the contents of the signal, any signal, even an empty signal, would suffice. You could use the `IdentityIntervalSimulator` to create such a signal and drive the polling of the Twitter block. The `Identity` generator could be used since we only need an empty signal. The `Interval` trigger could be used to poll the API periodically.

Because of the way we use simulators to kick off the actions of a service, almost every service will either begin with a `Subscriber` block (to subscribe to signals from another service) or with a `Simulator` block (to generate the driving signals on its own).

Testing and Custom Simulators

To test a service, you may want data that looks like an actual signal, but do not want to connect to a real data source. To do this, you can create a custom generator for your data format, and then use a simulator block with that generator. The process of creating a custom generator requires writing python code and is documented in more detail in the [signal generator / simulator readme files] https://blocks.n.io/?category=Signal%20Generator).

You can create custom simulated data without writing any code, but this is not as flexible. By pairing a simulator with the `Identity` trigger with a `Modifier` block, a service builder can generate empty signals on a schedule and then route the empty signal to a `Modifier` block where it fills in the information that the signal should consist of. A service building expert would make use of the `random` module inside of a nio expression to generate random data. For example, to generate a signal with a random number from 1 to 10 each time, connect the output of an `IdentityIntervalSimulator` block to a `Modifier` block.

```
{
  "name": "Random Number",
  "type": "Modifier",
  "fields": [{
    "title": "num",
    "formula": "{{ random.randint(1,10) }}"
  }]
}
```

Environment Variables

nio allows you to define and use variables when configuring your instance, services, or blocks. These variables are called "environment variables" and are specified using the square bracket syntax: `[[ ENV_VAR_NAME ]]` . There are several different use cases for environment variables which are detailed below.

Access Tokens and Other Secrets

Often a block will need some sort of access token or password in its configuration. Rather than store those in the block config directly, where they are visible in plain text, we recommend using an environment variable for that. Add an entry to your environment variable such as `nio.env`

```
MY_SECRET: p@$$w0rd
```

Then, in your block config you can use that secret token by using the environment variable syntax: `[[ MY_SECRET ]]` . The block will receive the proper value when the service is started , but the block's configuration will always contain the unreplaced environment variable value.

Different Environments

A large-scale nio system will generally run in multiple environments. Here are several examples:

1. A multi-tenant setup where the same project runs in many different locations. For example, a smart retail store system where the project runs in every store.
2. A staging environment used for validation and testing.
3. A local test environment used for building blocks or writing service unit tests.

For any of these setups, the use of environment variables can help you. If you find that a value may differ from one environment to the next (e.g., an IP address, a database password, whether a service should run or not), then replace that value with an environment variable and create a different env file for the respective environment. For example, your production env file (`prod.env`) may look something like this:

```
DB_HOST: real-data-host.com
DB_PASS: a357v34bs434vb34
```

but your local testing environment file (`test.env`) may look like this:

```
DB_HOST: localhost
DB_PASS: password
```

In your blocks and services, you can use `[[ DB_HOST ]]` as the database host and be assured that when you are running nio locally you aren't talking to the production database.

See the Deployment section for more information.

Setting Environment Variables

nio environment variables are slightly different than operating system environment variables. nio environment variables can also be set by operation system environment variables. This can be useful when running nio using Docker, systemd, or some other process manager where you can pass environment variables to the process.

In general, environment variables can be sourced from two places.

1. The operating system environment variables.

   ```
   $ export DB_HOST=localhost
   $ nio_run
   ```

   ```
   $ docker run -e DB_HOST=localhost nio_binary_image
   ```

2. The `.env` files in your project directory. Assuming you have a `prod.env` file in your project directory, you can source from that using the `-e` flag of `nio_run`.

   ```
   $ nio_run -e prod.env
   ```

In the event that an environment variable is set both at the system level and the local `.env` file level, the system environment variable will take precedence.

nio API

The nio API is the mechanism for interacting with a running nio instance and its blocks and services. It follows a simple, RESTful design.

API Conventions

You can interact with most nio binaries through a REST API. The API supports the standard HTTP request types of `GET` , `POST` , `PUT` , and `DELETE` . Data responses are returned in JSON format.

The REST API is available in nio binaries that include the nio REST Component. Read more about nio core components in components.

Authentication and Authorization

Authentication is the action to verify the identity of a user or process.

Authorization is the process of giving someone permission to perform an action.

You will need to include an authorization header with your nio API requests. For examples, see Basic Auth and JWT. For examples of adding headers to cURL requests, see Headers.

User names and passwords are specified in the `users.json` file in the `etc` directory.

Authorization for your project's users is specified in the `permissions.json` file in the `etc` directory of your project.

If you want to secure your project and its API, change these user names and passwords and specify the permissions.

You can change the location of your users and permissions files in the `[security]` section of `nio.conf` , a YAML file.

```
[security]

# json configuration that defines users and their passwords - can point to a conf file
#
#users=etc/users.json

# json configuration that maps users to permissions - can point to a conf file
#
#permissions=etc/permissions.json
```

The default configuration is shown. You can uncomment the default configuration and it will not change. To find your users and permissions in files other than `etc/users.json` and `etc/permissions.json` , uncomment the configuration and edit it.

You can use different types of authentication, such as basic authentication and JSON Web Tokens, with nio.

Basic Authentication (Basic Auth)

When using basic authentication, the REST API requires a username and password.

To use basic auth, include a base-64 encoded version of your `username:password` in the Authorization header of your request. The default `username:password` from the project template is `Admin:Admin` . An example of basic authorization in your request header follows:

`{ "authorization": "Basic QWRtaW46QWRtaW4=" }`

For examples of adding headers to a curl request, see Headers.

JSON Web Tokens (JWT)

When using JSON Web Tokens (JWT), first obtain an access, and then include it in the header.

`{ "authorization": "Bearer <your token>" }`

For examples of adding headers to a curl request, see Headers.

Testing with curl

To test nio API requests and responses, you can use a tool such as Postman or a curl command from your terminal. curl commands start with `curl`, and then include the request type, a URL, and possibly a request header and a request body. For the nio API, an authorization header is required.

Request Type

Your request type will be one of `GET`, `POST`, `PUT`, or `DELETE`.

With the curl command, you can specify with following options:

```
   -XGET
   -XPOST
   -XPUT
   -XDELETE
```

URL

Your request type will be followed by a URL.

Base URL

The base of the URL for your API request is the address where your nio project is running. Obtain the base URL from terminal in your nio logs.

For a local project

```
http://localhost:8181/
```

For a remote project

```
https:// <IP address:port> /
```

Endpoint

The endpoint is added to the base URL

```
http://localhost:8181/blocks_types
```

You can also add query parameters to the URL, but for the most part, the nio API does not use query parameters.

Headers

In your request, you need to include a header with your Basic Auth and JWT authentication. You can add auth to your cURL requests with the `-H` header flag or the `--user` flag followed by a string. The following formats will work:

```
   -H 'authorization: Basic <base 64 encoded username:password>'

   -H 'authorization: Bearer <your token here>'
```

```
  --user 'Admin:Admin'
```

To send data with your request, you need to include the data's Content-Type in your header.

```
  -H 'Content-Type: application/json'
```

Body

In the request body, if you need to include data, add body data to a curl request with the `-d` or `--data` flag.

```
-d '{"type": "LoggerBlock", "name": "Log"}'

--data '{"type": "LoggerBlock", "name": "Log"}'
```

Putting It All Together

A typical curl request to the nio API incorporating all these parts follows:

```
curl -XPOST 'http://localhost:8181/blocks' --user 'Admin:Admin' --data '{"type": "LoggerBlock", "name": "Log"}'
 -H 'Content-Type: application/json'
```

Core APIs

nio core has a few crucial API endpoints: `/nio` and `/shutdown` .

nio

The `/nio` endpoint is used to find information, such as versioning, about your running nio instances

```
curl -XGET 'localhost:8181/nio'
```

The following JSON body is returned:

```
{
    "components": {
        "ServiceMonitor": {
            "version": "0.1.0"
        },
        "ConfigManager": {
            "version": "0.1.0"
        },
        "ProjectManager": {
            "version": "2.0.0"
        },
        "LogManager": {
            "version": "0.1.0"
        },
        "BlockManager": {
            "version": "0.1.0"
        },
        "ManagementPublisher": {
            "version": "0.1.1"
        },
        "RESTManager": {
            "version": "0.1.0"
        },
        "ServiceManager": {
            "version": "0.1.0"
        }
    },
    "modules": {
        "security": {
            "BasicSecurityModule": {
                "version": "0.1.0"
            }
        },
        "settings": {
            "SettingsIniModule": {
                "version": "0.1.0"
            }
        },
        "scheduler": {
            "CustomSchedulerModule": {
                "version": "0.1.0"
            }
        },
        "communication": {
            "PubkeeperCommunicationModule": {
                "version": "0.1.0"
            }
        },
        "persistence": {
            "FilePersistenceModule": {
                "version": "0.1.0"
            }
        },
        "web": {
            "CherryPyWebModule": {
                "version": "0.1.0"
            }
        }
    },
    "nio": {
        "instance_id": "agriculture_master",
        "version": "2.1.0",
        "binary": "nio_full",
        "build": "20170509"
    },
    "start_time": "2017-08-02 11:24:19.427352"
}
```

components
The core components running in the binary.

modules
The modules that interface between the core and blocks, such as Communication and Persistence.

nio

- **binary**
  The name of the executable binary that was run against the project.
- **build**
  The build version of the binary.
- **version**
  The version of the nio framework (not the version of the **binary**).

start_time
A datetime string that indicates when nio was started.

Shutdown

The `/shutdown` endpoint is used to shutdown a running nio instance.

```
curl -XGET 'http://localhost:8181/shutdown'
```

The following HTML is returned after a successful shutdown request:

```
Shutdown complete
```

Blocks Types API

nio block types are the functional pieces of code that generate or transform signals. Earlier we saw examples of block types, such as *CounterIntervalSimulator* and *Logger*. Information about and interaction with the blocks of a running nio instance are available through the `/blocks_types` API.

Get API

A `GET` request sent to the `/blocks_types/<block type name>` endpoint will return a JSON body with information about that block type.

The following example gets the information for the *Logger* block type

```
curl -XGET 'http://localhost:8181/blocks_types/Logger'
```

The result of the previous request follows:

```
{
  "name": "Logger",
  "version": "0.0.0",
  "namespace": "blocks.logger.logger_block.Logger",
  "properties": {
    "type": {
      "allow_none": false,
      "title": "Type",
      "default": null,
      "type": "StringType",
      "visible": false,
      "readonly": true
    },
    "version": {
      "type": "StringType",
      "allow_none": false,
      "visible": true,
      "title": "Version",
      "default": "0.0.0"
    },
    "log_level": {
      "allow_none": false,
      "options": {
        "ERROR": 40,
        "DEBUG": 10,
        "INFO": 20,
        "WARNING": 30,
        "NOTSET": 0,
        "CRITICAL": 50
      },
      "title": "Log Level",
      "enum": "LogLevel",
      "type": "SelectType",
      "default": "INFO",
      "visible": true
    },
    "log_at": {
      "allow_none": false,
      "options": {
        "ERROR": 40,
        "DEBUG": 10,
        "INFO": 20,
        "WARNING": 30,
        "NOTSET": 0,
        "CRITICAL": 50
      },
      "title": "Log At",
      "enum": "LogLevel",
      "type": "SelectType",
      "default": "INFO",
      "visible": true
    },
    "name": {
      "type": "StringType",
      "allow_none": false,
      "visible": false,
      "title": "Name",
      "default": null
    }
  },
  "commands": {
    "properties": {
      "title": "properties",
      "params": {}
    },
    "log": {
      "title": "log",
      "params": {
        "phrase": {
          "allow_none": false,
          "title": "phrase",
          "default": "Default phrase"
        }
      }
    }
  },
  "attributes": {
    "output": [
      {
        "id": "__default_terminal_value",
        "label": "default",
        "order": 0,
        "default": true,
        "type": "output",
        "description": "",
        "visible": true
      }
    ],
    "input": [
      {
        "id": "__default_terminal_value",
        "label": "default",
        "order": 0,
        "default": true,
        "type": "input",
        "description": "",
        "visible": true
      }
    ]
  },
}
```

name
The name of the block type.

version
The version of the block type that is being used by the running nio instance.

namespace
The class the block type is imported from.

properties
The configurable attributes and information of the block type including:

- **type**
  The property type.
- **name**
  The block name.
- **default**
  The default value.
- **title**
  The title that will display in the block configuration panel.
- **allow_none**
  True if the property configuration is optional; false, if required.
- **visible**
  True if the property displays in the configuration panel; false, if does not appear.

commands
Executable commands on running blocks and all information about them including:

- **title**
  The name of the command.
- **params**
  Any parameters the command method takes.

attributes
Any additional block attributes including:

- **input**
  The input terminal(s) for incoming signals.
- **output**
  The output terminal(s) for outgoing signals.

Get All API

In addition to getting the details of one block type, specified by name, you can get the details of all the block types in your project with one request.

```
curl -XGET 'http://localhost:8181/blocks_types'
```

Add API

When nio starts, it discovers and adds all the block types in the project. If you add a new block type to a running instance of nio, it will not be discovered until nio restarts. However, you can add the new block type to a running instance with the Add API. After the block code is added to the project directory, send a PUT request to the new block type name to load the new block type into the running nio instance

```
curl -XPUT 'http://localhost:8181/blocks_types/<NewBlockTypeName>'
```

Update API

When nio starts, it discovers and adds all the block types in the project. When code in an existing block type is changed in a project (for example, when upgrading to a newer version of the block type) in a running nio instance, you need to tell nio to use the updated code. You can do that with the Update API. Once the new block code is updated in the project directory, send a PUT request to the updated block type to load it into the running nio instance

```
curl -XPUT 'http://localhost:8181/blocks_types/Logger'
```

Blocks API

A configured instance of a nio block type is referred to simply as a block. While block types are classes and have code, blocks are specific configurations of a block type's properties. Earlier we saw an example of a *Logger* block instance created with the name `Log`. Information about and interaction with individual block configurations in a running nio instance are available through the `/blocks` API.

Get API

The Get API returns a JSON body with information about a block based on its name. The following example gets the information for the configured *Logger* block with the name `Log`

```
curl -XGET 'http://localhost:8181/blocks/Log'
```

The result of the previous request is

```
{
  "type": "Logger",
  "version": "1.0.0",
  "name": "Log",
  "log_level": "INFO",
  "log_at": "INFO"
}
```

The following four properties are common to all block types:

type
The block type of the block configuration.

version
The version of the block when the block configuration was created.

name
The name of the block configuration. This must be unique among all blocks, regardless of type.

log_level
The number of log messages displayed, from `NOTSET` on the bottom, which includes all messages logged, to `CRITICAL` on the top, which contains only the most critical messages. Messages are shown for the specified `log_level` and all levels above.

Other properties are block-type specific:

log_at
`log_at` is a property specific to the *Logger* block type and is detailed in the *Logger* block type specifications.

Other block types will show their own properties in the response. For example, the *AttributeSelector* block type has an `attributes` property instead of 'log_at'.

```
  "attributes": []
```

Get All API

In addition to getting the details of one block configuration, specified by name, you can get the details of all block configurations in one request

```
curl -XGET 'http://localhost:8181/blocks'
```

Create API

If you're working with a nio project from scratch, you need to create and configure blocks. You can create a new configuration of a block with the Create API by sending a POST request to `/blocks` with the applicable JSON data. When creating a new block configuration, optionally you can include the configured values of the block type's properties. At a minimum, you must specify the block `type`, the `name` of the new configuration, and values for any required properties that do not have a default value. For example, to create a *Logger* block type with the name `Log`

```
curl -XPOST 'http://localhost:8181/blocks' --data '{"type": "LoggerBlock", "name": "Log"}' -H 'Content-Type: ap
plication/json'
```

Update API

When you want to update the configuration of a block, send a PUT request to `/blocks` with the block-name endpoint and include the new JSON data. You only need to include the properties that you are updating in your PUT request, in this case `log_level`.

```
curl -XPUT 'http://localhost:8181/blocks/Log' --data '{"log_level": "DEBUG"}' -H 'Content-Type: application/jso
n'
```

Delete API

To delete a named block, send a DELETE request to `/blocks` with the name of the block you want to delete as the endpoint.

```
curl -XDELETE 'http://localhost:8181/blocks/Log'
```

Services API

Services are the real-time processes that run on an instance in nio where you configure the logic of the workflow of blocks to make interesting things happen. Earlier we created a service called `SimulateAndLog` where we connected a *CounterIntervalSimulator* to a *Logger*. Information about and interaction with the services of a running nio instance are available through the `/services` API.

Get API

The Get API returns a JSON body with information about a service based on its name. The following example gets the information for the service `SimulateAndLog`.

```
curl -XGET 'http://localhost:8181/services/SimulateAndLog'
```

The result of the previous request is

```
{
  "type": "Service",
  "version": "1.0.0",
  "name": "SimulateAndLog",
  "log_level": "NOTSET",
  "auto_start": false,
  "status": "stopped",
  "execution": [
    {
      "name": "Simulate",
      "receivers": {
        "__default_terminal_value": [
          {
            "name": "Log",
            "input": "__default_terminal_value"
          }
        ]
      }
    },
    {
      "name": "Log",
      "receivers": []
    }
  ],
  "mappings": [],
  "sys_metadata": "{\"Simulate\":{\"locX\":248,\"locY\":102},\"Log\":{\"locX\":202,\"locY\":210}}"
}
```

type
The service type of the service configuration. Usually, the type is `Service`.

version
The version of the service type when the service configuration was created.

name
The name of the service configuration. This must be unique among all services in the instance.

log_level
The number of log messages displayed, from `NOTSET` on the bottom, which includes all messages logged, to `CRITICAL` on the top, which contains only the most critical messages. Messages are shown for the specified `log_level` and all

levels above.

auto_start
A boolean that indicates if the service will start when nio starts up.

status
Current status of the service: configuring, configured, starting, started, stopping, stopped, and error.

execution
The configuration of inter-block connections in the service. Block connections are defined as a list of dictionaries that specify a block `name` and the blocks it sends signals to, called `receivers`. The `receivers` specify the output terminal in the source block that emits signals and the input terminal in the receiver block that receives signals.

mappings
The local ID of a block configuration. If a single block configuration is used multiple times in a service, a new local name is created for each additional use of that block, and those new names are specified in mappings.

sys_metadata
Data that is used by the System Designer to store block locations on the canvas.

Get All API

In addition to getting the details of one service configuration, specified by name, you can get the details of all service configurations in one request

```
curl -XGET 'http://localhost:8181/services'
```

Create API

If you're working with a nio project from scratch, you need to create and configure services. Create a new service with the Create API by sending a POST request with the applicable JSON data. When creating a new service, you can optionally include the configured values of the service type properties. At a minimum, you must specify the service `type` and `name` and any configuration values for required service properties that do not have a default value. For example, to create the `SimulateAndLog` service of the basic type `Service`

```
curl -XPOST 'http://localhost:8181/services' --data '{"type": "Service", "name": "SimulateAndLog"}' -H 'Content
-Type: applcation/json'
```

Update API

When you want to update the configuration of a service, send a PUT request to `/services` with the service name as the endpoint and include any new JSON data. You only need to include the properties that you are updating in your PUT request, in this case, `log_level`

```
curl -XPUT 'http://localhost:8181/services/SimulateAndLog' --data '{"log_level": "DEBUG"}' -H 'Content-Type: ap
plcation/json'
```

Delete API

To delete a service, send a DELETE request to `/services` with the name of the service you want to delete as the endpoint.

```
curl -XDELETE 'http://localhost:8181/services/SimulateAndLog'
```

Services Types API

Service types contain metadata for services and can be accessed with `/services_types` API.

Get API

The Get API returns a JSON body with information about the service type with a particular name. For example, send a GET request to `/services_types` with the name `Service` as its endpoint

```
curl -XGET 'http://localhost:8181/services_types/Service'
```

and the result will be

```
{
  "Service": {
    "name": "Service",
    "properties": {
      "name": {
        "default": null,
        "allow_none": false,
        "type": "StringType",
        "title": "Name",
        "visible": true
      },
      "mappings": {
        "obj_type": "BlockMapping",
        "list_obj_type": "ObjectType",
        "visible": true,
        "default": [],
        "allow_none": false,
        "type": "ListType",
        "title": "Mappings",
        "template": {
          "name": {
            "default": null,
            "allow_none": false,
            "type": "StringType",
            "title": "Name",
            "visible": true
          },
          "mapping": {
            "default": null,
            "allow_none": false,
            "type": "StringType",
            "title": "Mapping",
            "visible": true
          }
        }
      },
      "auto_start": {
          "default": true,
          "allow_none": false,
          "type": "BoolType",
          "title": "Auto Start",
          "visible": true
      },
      "sys_metadata": {
          "default": "",
          "allow_none": false,
          "type": "StringType",
          "title": "Metadata",
            "visible": true
        },
        "log_level": {
          "options": {
            "CRITICAL": 50,
            "DEBUG": 10,
            "INFO": 20,
            "WARNING": 30,
            "NOTSET": 0,
            "ERROR": 40
          },
          "visible": true,
          "default": "NOTSET",
          "allow_none": false,
          "type": "SelectType",
          "title": "Log Level",
          "enum": "LogLevel"
        },
        "execution": {
          "obj_type": "BlockExecution",
          "list_obj_type": "ObjectType",
          "visible": true,
          "default": [],
          "allow_none": false,
          "type": "ListType",
          "title": "Execution",
          "template": {
            "name": {
              "default": null,
              "allow_none": false,
              "type": "StringType",
              "title": "Name",
              "visible": true
            },
            "receivers": {
              "default": null,
              "allow_none": false,
              "type": "Type",
              "title": "Receivers",
              "visible": true
            }
          }
        },
        "type": {
          "visible": false,
          "default": null,
          "allow_none": false,
          "type": "StringType",
          "title": "Type",
          "readonly": true
        },
        "version": {
          "default": "0.1.0",
          "allow_none": false,
          "type": "StringType",
          "title": "Version",
          "visible": true
        }
      },
      "namespace": "nio.service.base.Service",
      "commands": {
        "status": {
          "params": {},
          "title": "status"
        },
        "runproperties": {
          "params": {},
          "title": "runproperties"
        },
        "heartbeat": {
          "params": {},
          "title": "heartbeat"
        },
        "stop": {
          "params": {},
          "title": "stop"
        },
        "start": {
          "params": {},
          "title": "start"
        }
      }
    }
  }
```

name
The name of the service type, in this case, `Service`.

properties
The properties of the service. In this case, there is only one service type, `Service`, and all services contain these properties.

Get All API

You can get the details of all the service types configurations in one request with a request to the `/services_types` endpoint.

```
curl -XGET 'http://localhost:8181/services_types'
```

Additional APIs

nio has additional APIs that you may find useful as you explore what nio has to offer.

Configuration Refresh API

The `/config/refresh` endpoint, provided by the `config` component, is used to update a running nio instance when its block or service configuration files were modified through some means other than the `/blocks` and `/services` APIs. For example, if you use a text editor to modify a block configuration file, you need to tell the running nio instance to use those changes. You can do this with a GET request to the `/config/refresh` endpoint of your running instance.

```
curl -XGET 'localhost:8181/config/refresh' --user 'Admin:Admin'
```

Core Components

Part of the nio binary consists of core components. These are pieces of functionality that run alongside the nio core process. Unlike modules which run in every nio service, a core component will only be run once and will run in the core/main process.

Examples of core components include:

- A project manager component that exposes an API to interact with your project contents
- A REST component that provides an interface for interacting with nio services and blocks
- An SNMP agent component that exposes runtime information in SNMP form for easy ingestion from monitoring tools
- A logging component that exposes an API to retrieve an instance's log messages and to change log levels for currently running services, blocks and components
- A service monitor component that monitors service statuses able to restart a service if it becomes unresponsive

Disabling Core Components

You cannot add core components to your binary, but you can disable existing ones if you do not want or need them. To disable a core component, in the `nio.conf` file under the `component` section, add or uncomment the disable line. For example, to disable the `SNMPAgent` component, your `nio.conf` would look like the following example:

```
[components]
disable=SNMPAgent
```

SNMP Agent Component

The SNMP Agent core component allows you to monitor the health of a nio system through the standard SNMP protocol.

Behavior

The SNMP component can be configured to work as an agent that can issue traps asynchronously or respond to requests synchronously.

- Asynchronous mode sends traps whenever a service changes its status, and the new status is configured as part of the `status_traps` setting.
- Synchronous mode allows retrieving the system uptime and service statuses.

Configuration

You can configure the behavior of this component in your `nio.conf` file under the `[snmp]` section.

The following configuration options are available:

- `host` —The host address for the SNMP Agent (default= `127.0.0.1`)
- `port` —The port for the SNMP Agent (default= `161`)
- `community_index` —The name of the index to use in the SNMP community (default= `agent`)
- `community_name` —The name of the community (default= `public`)
- `trap_host` —The host address for the SNMP trap server (default= `127.0.0.1`)
- `trap_port` —The port for the SNMP trap server (default= `162`)
- `trap_community_index` —The name of the index to use in the SNMP trap community (default= `agent`)
- `status_traps` —The comma separated list of services to register status change of traps (default= `error`)

Glossary of Terms

Binary

A binary is the executable program that you run to run nio. Depending on your circumstances, you will run one of many nio binaries. Binaries range in complexity and are often tuned for particular hardware. If you're tinkering around on a Raspberry Pi, you don't need to run the same binary as a network of a dozen supercomputers built to analyze hundreds of thousands signals per second.

Block

A block is a class that is built off of the Block Development Framework that defines methods to operate on streams of data. A block receives, emits, and/or transforms signals. Blocks are written in Python and tend to focus on one particular function. Blocks are reusable in any other services so it is beneficial to keep them modular.

Framework

The Block Development Framework is the layered structure that defines how to develop blocks. This nio framework consists of the agreed upon interfaces that blocks and binaries understand. Install the framework with pip: `pip install nio` .

Instance

A running version of the nio back end platform. Each host in a system will likely have at least one nio instance running on it.

Project

A nio project is stored in a directory on your computer that contains the configuration that defines your nio implementation. Projects can run on their own or can be a part of a larger nio network where many instances communicate with one another.

Pubkeeper

Pubkeeper™ is a communications broker that enables real-time data transfer between a system of disparate devices through standard publish-subscribe patterns. With Pubkeeper, two devices can interact with each other even if they do not share the same communication protocol.

Service

A service is a real-time process that runs on an instance. The logic is configured as a workflow of blocks. Services can be started and stopped.

Signal

A signal is a unit of data that is passed from block to block within a service or between services. You can think of it as a little box of data, containing any data from a temperature reading to a tweet. It is internally represented as a key-value store.

System

A system is a distributed set of entities that work together as a whole. For nio, this means a collection of nio back end instances and front end user interfaces that can communicate with one another in real time.

Support

We understand that designing distributed systems isn't always easy. That's why niolabs is here to help!

Choose one of the options below to reach out to the nio support community:

- Live chat from the System Designer - Click the blue conversation bubble in the System Designer to be connected to a live niolabs support representative. If no one is online, we will still see your message and contact you as soon as we can.
- Support email - Email support@n.io any time with any questions or requests.
- Community support forums - Coming soon!