Quick Start

Installation#

Flume can be installed from npm like so:

```
npm install flume
```

or alternatively

```
yarn add flume
```

Basic Usage#

To get started, import and render the node editor like so:

```
import React from 'react'
import { NodeEditor } from "flume";
const App = () => {
return (
<div style={{width: 800, height: 600}}>
<NodeEditor />
</div>
)
}
```

As you may guess, this alone will render only a blank editor grid with no nodes. This obviously isn't very useful, so lets get started configuring our first node editor.

note

We've also wrapped our node editor in a div with a width and height. The node editor is designed to take up all the available space of its container. If you place it in a container with a variable width, it will respond to any changes in the width.

Basic Configuration

For our node editor to be useful, we need to define some nodes. For this example we're going to pretend that we're building a node editor to control the properties of a page on our website. Flume can be used for a lot more interesting use-cases than this, but this example will suffice for now.

Create a config file#

Start by creating a new blank Javascript file in your project called config.js. Then, import FlumeConfig from flume like so:

```
import { FlumeConfig } from 'flume'
```

Next, create a new instance of FlumeConfig like this.

```
import { FlumeConfig } from 'flume'
const config = new FlumeConfig()
export default config;
```

And that's it! Now we're ready to start defining our ports and nodes.

Create your first port#

The FlumeConfig class exposes some helpful methods for configuring ports and nodes. We're going to start by creating a port for text strings:

```
import { FlumeConfig, Colors, Controls } from 'flume'
const config = new FlumeConfig()
config
.addPortType({
type: "string",
name: "string",
label: "Text",
color: Colors.green,
controls: [
Controls.text({
name: "string",
label: "Text"
```

```
})
]
})
```

Let's break down what's going on here. First we decide on a type for this port. Because this port is going to output and accept strings, we'll just call it string. We also need to give it a name that we can use to identify it later. For simplicity, we'll also name it string.

note

Keep in mind, there's nothing magic about using the word string as the type name. This name is entirely arbitrary. We could give this port any unique type name and it would behave exactly the same.

Now "string" may be a confusing word for non-programmers so we'll give it a label of "Text". This is the label that will be shown to users when the port renders. Next we'll pick a color for the port. By default, ports are a neutral gray color, but to make it easier to recognize this port, we'll color it green by importing Colors from flume, and setting color to Colors.green.

Last, we need to define the controls available when this port is an input. In most cases, ports only have one control, but you may have any number of them. In this case, we want users to be able to type in text, so we'll import Controls from flume, and then call Controls.text(). We'll give the text input a name, which again, for simplicity, we'll just name it string, and we'll give it a user-readable label of Text.

So far we've only defined a text port, but not a text node. In order to use our port we need to create a node for it. Let's do that now:

Create your first node#

We want our users to be able to use our text port, so let's create a node for it. Using the FlumeConfig class again, lets add our first node:

```
import { FlumeConfig, Colors, Controls } from 'flume'
const config = new FlumeConfig()
config
.addPortType({
type: "string",
name: "string",
label: "Text",
color: Colors.green,
controls: [
Controls.text({
name: "string",
label: "Text"
```

```
})
]
})
.addNodeType({
type: "string",
label: "Text",
description: "Outputs a string of text",
inputs: ports => [
ports.string()
],
outputs: ports => [
ports.string()
]
})
```

Let's break this down. Like when we defined our port, we start by deciding on a type for our node. Because this node will only be responsible for inputting and outputting text strings, we'll call the type string. And as before, we also pick a user-friendly label, which we'll call Text. We can also provide an optional description, which can help users understand how to use a node.

Next, we can define the inputs and outputs for this node. Because we've already defined the ports for this node, FlumeConfig will provide them for us in our inputs function. The inputs and outputs functions must return an array of previously-defined ports, so we'll return an array with a call to ports.string().

Rendering the editor#

Now that we have our first node, let's return to where we've rendered our node editor and import our config.

```
import React from 'react'
import { NodeEditor } from 'flume'
import config from './config'
const App = () => {
return (
<div style={{width: 800, height: 600}}>
<NodeEditor
portTypes={config.portTypes}
nodeTypes={config.nodeTypes}
/>
</div>
```

```
)
}
```

As you can see, we've provided our portTypes and nodeTypes to props of the same name. Right click anywhere on the node editor and click on the Text option. If everything is hooked up correctly, you should now have a node editor that looks like this:

Try creating a few Text nodes and connecting them together. This still isn't very useful, but let's head back to config.js and create a few more ports and nodes.

Adding more ports & nodes#

Let's add two more ports, one for booleans, and one for numbers. We'll start with the boolean port & node.

```
import { FlumeConfig, Colors, Controls } from 'flume'
const config = new FlumeConfig()
config
.addPortType({
type: "string",
name: "string",
label: "Text",
color: Colors.green,
controls: [
Controls.text({
name: "string",
label: "Text"
})
]
})
.addPortType({
type: "boolean",
name: "boolean",
label: "True/False",
color: Colors.blue,
```

```
controls: [
Controls.checkbox({
name: "boolean",
label: "True/False"
})
]
})
.addNodeType({
type: "string",
label: "Text",
description: "Outputs a string of text",
inputs: ports => [
ports.string()
],
outputs: ports => [
ports.string()
]
})
.addNodeType({
type: "boolean",
label: "True/False",
description: "Outputs a true/false value",
initialWidth: 140,
inputs: ports => [
ports.boolean()
],
outputs: ports => [
ports.boolean()
]
})
```

As you can see, the process for adding this port is very similar. Instead of the text control we'll use the checkbox, and we'll set the color to blue. Then when we create the node we'll use the boolean port for the input and output. By default, nodes are 200px wide, but our boolean node can be a little more compact, so we'll give it an initialWidth of 140.

Next let's add the number port and node:

```
import { FlumeConfig, Colors, Controls } from 'flume'
const config = new FlumeConfig()
```

```
config
.addPortType({
type: "string",
name: "string",
label: "Text",
color: Colors.green,
controls: [
Controls.text({
name: "string",
label: "Text"
})
]
})
.addPortType({
type: "boolean",
name: "boolean",
label: "True/False",
color: Colors.blue,
controls: [
Controls.checkbox({
name: "boolean",
label: "True/False"
})
]
})
.addPortType({
type: "number",
name: "number",
label: "Number",
color: Colors.red,
controls: [
Controls.number({
name: "number",
label: "Number"
})
]
})
.addNodeType({
type: "string",
```

```
label: "Text",
description: "Outputs a string of text",
inputs: ports => [
ports.string()
],
outputs: ports => [
ports.string()
]
})
.addNodeType({
type: "boolean",
label: "True/False",
description: "Outputs a true/false value",
initialWidth: 140,
inputs: ports => [
ports.boolean()
],
outputs: ports => [
ports.boolean()
]
})
.addNodeType({
type: "number",
label: "Number",
description: "Outputs a numeric value",
initialWidth: 160,
inputs: ports => [
ports.number()
],
outputs: ports => [
ports.number()
]
})
```

If you come back to your node editor and right click anywhere, you should have 2 new nodes available. If everything is hooked up correctly, your node editor should look like this:

While this is all fine and well, it's still not very useful yet. What we need next is a "root" node to connect our other nodes to. This will serve as the output node for our graph.

The Root Node

The root node is a special node that serves as the final output for our graph. If you've been following along from the previous examples, we're configuring a node editor to control the properties of a page on our website. The node that accepts these final properties is our root node.

Adding a root node#

Let's get started adding our root node. We want this node to represent the properties we'll use in our website. Let's head back to our config.js file.

```
import { FlumeConfig, Colors, Controls } from 'flume'
const config = new FlumeConfig()
```

```
config
/* ... */
.addRootNodeType({
type: "homepage",
label: "Homepage",
initialWidth: 170,
inputs: ports => [
ports.string({
name: "title",
label: "Title"
}),
ports.string({
name: "description",
label: "Description"
}),
ports.boolean({
name: "showSignup",
label: "Show Signup"
}),
ports.number({
name: "copyrightYear",
label: "Copyright Year"
})
]
})
```

You may notice a few things about the above example. First, instead of addNodeType we're using the addRootNodeType function, but the properties you pass to this function are the same. Under the hood, this node type is marked as the "root" node, and by default it can't be manually added or deleted in the node editor.

Default nodes#

Let's return to the node editor. You should have what appears to be the same editor as before. If you right-click to add a node, the homepage node doesn't appear. Because this node is the root node, there should be exactly one of this node in the editor at all times. We don't want users to be able to add or delete this node, so we'll provide it to the editor as a default node.

In the file where we're rendering the node editor, we'll add a new prop called defaultNodes.

```
import React from 'react'
```

```
import { NodeEditor } from 'flume'
import config from './config'
const App = () => {
return (
<div style={{width: 800, height: 600}}>
<NodeEditor
portTypes={config.portTypes}
nodeTypes={config.nodeTypes}
defaultNodes={[
{
type: "homepage",
x: 190,
y: -150
}
]}
/>
</div>
)
}
```

defaultNodes is an array of objects with a type, and an x and y position, relative to the center of the editor. If everything has gone according to plan, you should have an editor that looks like this:

Try adding your other nodes and connecting them together. Our node editor is finally starting to take shape, but we need to create some more useful nodes if we really want to create something powerful. Let's head back to our config file and add a few more nodes.

Logic Nodes

So far we've created some basic nodes. We've created text, number, and boolean nodes, and a root node for the final output. While this is fine and well, let's take a minute and think about some nodes we could create that could make our editor more powerful. Here's a few ideas:

- A user node, that outputs attributes of the current user
- A time node that outputs the current time
- A text join node, that combines two strings of text
- Nodes representing mathmatical operations (+, -, *, ÷)
- A find and replace node, for replacing strings of text
- Boolean operation nodes (and, or)
- A window node to expose attributes of the window
- Logical comparison nodes (greater than, less than, equals)

The kinds of nodes your present to your users are entirely up to you. When making this determination, ask yourself questions like,

- What kind of data inputs will my users need access to?
- What kind of logic will be intuitive for my users to use?
- What capabilities make sense to expose to the end user?

Let's get back to our example app.

Adding logic nodes#

Let's pick a few ideas from the list above. We'll first create a node that represents the current user, a node that concatenates strings, and a node that reverses a boolean. We already have all the ports we need, so let's add a new user node in config.js:

```
import { FlumeConfig, Colors, Controls } from 'flume'
const config = new FlumeConfig()
config
/* ... */
.addNodeType({
type: "user",
label: "User",
description: "Outputs attributes of the current user",
initialWidth: 130,
outputs: ports => [
ports.string({
name: "firstName",
label: "First Name"
}),
ports.string({
name: "lastName",
label: "Last Name"
```

```
}),
ports.boolean({
name: "isLoggedIn",
label: "Is Logged-In"
}),
ports.boolean({
name: "isAdmin",
label: "Is Admin"
})
]
})
```

Nothing new to see here, we've just added a node type called user, with 4 outputs and no inputs. In a later section we'll walk through how to actually hook up the data for this node, but for now let's save and return to our node editor. You should now have a new node available like this:

Let's head back to config.js and add our "Join Text" node.

```
import { FlumeConfig, Colors, Controls } from 'flume'
const config = new FlumeConfig()
config
/* ... */
.addNodeType({
type: "joinText",
label: "Join Text",
description: "Combines two strings of text into one string",
initialWidth: 160,
inputs: ports => [
ports.string({
name: "string1",
label: "First text"
}),
ports.string({
```

```
name: "string2",
label: "Second text"
})
],
outputs: ports => [
ports.string({
name: "joinedText",
label: "Joined Text"
}),
]
})
```

As you can see, this node will take in two strings as inputs, and outputs a single combined string. You should now have a new node available in your editor like this:

Now let's add our last node that reverses a boolean.

```
import { FlumeConfig, Colors, Controls } from 'flume'
const config = new FlumeConfig()
config
/* ... */
.addNodeType({
type: "reverseBoolean",
label: "Reverse True/False",
description: "Reverses a true/false value",
initialWidth: 140,
inputs: ports => [
ports.boolean(),
],
outputs: ports => [
ports.boolean(),
]
})
```

This node just takes in a boolean, and outputs its inverse. Heading back to the node editor this new node should be available:

Summary#

So far we've created a useful node editor, but we don't have a way to get our logic graph in and out of the editor. In the next section we'll look at how to save the nodes in our editor so we can use them later.

Saving Nodes

The node editor internally manages its own state, but we can get our nodes in and out in two different ways.

onChange#

The easiest way to get your nodes out of the node editor is to hook it up to the onChange handler. This is similar to how you might use a text input in React.

```
import React from 'react'
import { NodeEditor } from 'flume'
import config from './config'
const App = () => {
const [nodes, setNodes] = React.useState({})
React.useCallback((nodes) => {
// Do whatever you want with the nodes
setNodes(nodes)
}, [])
return (
<div style={{width: 800, height: 600}}>
<NodeEditor
portTypes={config.portTypes}
nodeTypes={config.nodeTypes}
nodes={nodes}
onChange={setNodes}
/>
```

```
</div>
)
}
```

The onChange handler will be called any time any of the nodes change? This includes any time their position or control values change.

warning

It's critical to note that the onChange handler **MUST** be memoized. If you don't memoize the handler you will get an infinite loop.

getNodes#

Because the onChange handler may be called quite often, there are times you may want to only get out the nodes when the user is done editing. For these times the node editor provides an imperative handler for extracting the nodes.

```
import React from 'react'
import { NodeEditor } from 'flume'
import config from './config'
const App = () => {
const nodeEditor = React.useRef()
const saveNodes = () => {
const nodes = nodeEditor.current.getNodes()
// Do whatever you want with the nodes
}
return (
<div>
<button onClick={saveNodes}>Save Logic</button>
<div style={{width: 800, height: 600}}>
<NodeEditor
ref={nodeEditor}
portTypes={config.portTypes}
nodeTypes={config.nodeTypes}
/>
</div>
</div>
)
}
```

Summary#

Now we know how to get our nodes in and out of the node editor, but we still need a way to actually run our logic. In the next section we'll demonstrate how to setup the "Root Engine" to run our logic.

The Root Engine

Now that we've created a simple node editor with some useful nodes, we need a way to actually run our logic. While you are free to parse your logic graphs any way you see fit, Flume ships with a pre-built engine for running "root-style" logic graphs. The root engine is responsible for taking in a root-style graph generated with the Flume node editor, and returning the resolved properties of the root node. Let's take a look at how we can get the root engine up and running.

Setting up the engine#

To get started, let's create a new file called engine.js where we'll import RootEngine from flume, and our config file from config.js

```
import { RootEngine } from 'flume'
import config from './config'
const engine = new RootEngine(config)
export default engine
```

Like before, we create a new instance of the RootEngine and set it to a variable that we'll export below. We also imported our config file and provided it to the root engine as the first parameter. In order for the engine to work though, we need to provide it with 2 helper functions: resolvePorts, and resolveNodes.

Resolving ports#

The first thing we need to do is tell the root engine how to handle the controls for each of our ports. To keep things organized, let's create a new function above our root engine.

```
import { RootEngine } from 'flume'
import config from './config'
const resolvePorts = (portType, data) => {
switch (portType) {
case 'string':
return data.string
case 'boolean':
return data.boolean
case 'number':
return data.number
default:
return data
}
}
const engine = new RootEngine(config, resolvePorts)
```

Let's break this down. resolvePorts is a function that takes in the type of the port currently being processed, and all of the data from its controls. Then we open a switch statement with the port type, and return the port data for each port type. Because we only have 3 port types, we only have 3 entries in our switch statement. In our case, each of our ports only has one control, and we gave each control the same name as the port, so we can fill this function our pretty easily. In advanced use-cases, ports may have any number of controls, so this function will need to resolve each control, but this work for now.

note

This might seem a little unintuitive or redundant at first, but keep in mind that Flume doesn't make many assumptions about how you build or run your logic, so these functions are how you define that behavior.

Resolving nodes#

The last function we need is resolveNodes. This function tells each node how it should transform its inputs into its outputs. Let's create this function above the engine.

```
import { RootEngine } from 'flume'
import config from './config'
```

```
/* ... */
const resolveNodes = (node, inputValues, nodeType, context) => {
switch (node.type) {
case 'string':
return { string: inputValues.string }
case 'boolean':
return { boolean: inputValues.boolean }
case 'number':
return { number: inputValues.number }
case 'user':
return context.user
case 'joinText':
return { joinedText: inputValues.string1 + inputValues.string2 }
case "reverseBoolean":
return { boolean: !inputValues.boolean }
default:
return inputValues
}
}
const engine = new RootEngine(config, resolvePorts, resolveNodes)
```

This function looks similar to resolvePorts but has some key differences. resolveNodes takes in the current node, the resolved input values, the nodeType (which we're not using in this example), and an object called context. We create a switch statement using the node type, and return an object representing all of the outputs of that node. You may also notice that we didn't create a function for the homepage node. Because we marked this node as the "root" node, the root engine will take care of resolving and returning the values of its inputs for us.

Remember, each node can conceptually be thought of as a single function that transforms inputs into outputs. Are you starting to see how easy it can be to expose some powerful programming functions in a visual way?

note
We'll explain the context object in more detail in the next section, but in short, the context object is how you can pass live data into your engine at runtime. In this case we're using it to get the current user.

Summary#

In this section we setup a new RootEngine, and gave it instructions for resolving our ports and nodes. In the next section we'll show how to hook up the root engine to React.

Using With React

While React is required to render the Flume node editor, you are free to run your logic graphs with or without React. For this example we'll show you how to easily hook up your React components to the root engine we just created. Let's start by creating an example component.

Setting up an example component#

Let's create a file called called homepage.js. Remember for this example we've created a node editor for editing the attributes of a homepage.

note

Again, this is a bit of a contrived example, but it demonstrates all of the main features of Flume without much setup. In reality, creating logic graphs with Flume can be applied to a wide range of applications.

```
import React from 'react'
const fakeUser = {
firstName: "Bustopher",
lastName: "Jones",
isLoggedIn: true,
isAdmin: false
}
const nullUser = {
firstName: "",
lastName: "",
isLoggedIn: false,
isAdmin: false
}
const Homepage = ({ nodes }) => {
const [user, setUser] = React.useState(fakeUser)
const login = () => setUser(fakeUser)
const logout = () => setUser(nullUser)
return (
<div className="homepage">
<h1 className="title"></h1>
<p className="description"></p>
{
user.isLoggedIn ?
<button onClick={logout}>Logout</button>
:
<button onClick={login}>Login</button>
}
<form className="signup">
<input type="email" />
<button>Signup!</button>
</form>
```

```
<footer>© flume </footer>
</div>
)
}
export default Homepage
```

So far our homepage isn't much to look at. We've created a basic page structure, and put a fake user in the component's state. Now let's hook this component up to our logic graph.

useRootEngine#

Flume provides a simple hook for using the root engine with React. Let's import useRootEngine from flume, and our engine from engine.js.

```
import React from 'react'
import { useRootEngine } from 'flume'
import engine from './engine'
const fakeUser = {
firstName: "Bustopher",
lastName: "Jones",
isLoggedIn: true,
isAdmin: false
}
const Homepage = ({ nodes }) => {
const [user, setUser] = React.useState(fakeUser)
const {
title,
description,
showSignup,
copyrightYear
} = useRootEngine(nodes, engine, {user})
const login = () => setUser(fakeUser)
const logout = () => setUser(nullUser)
return (
<div className="homepage">
<h1 className="title">{title}</h1>
<p className="description">{description}</p>
{
user.isLoggedIn ?
<button onClick={logout}>Logout</button>
```

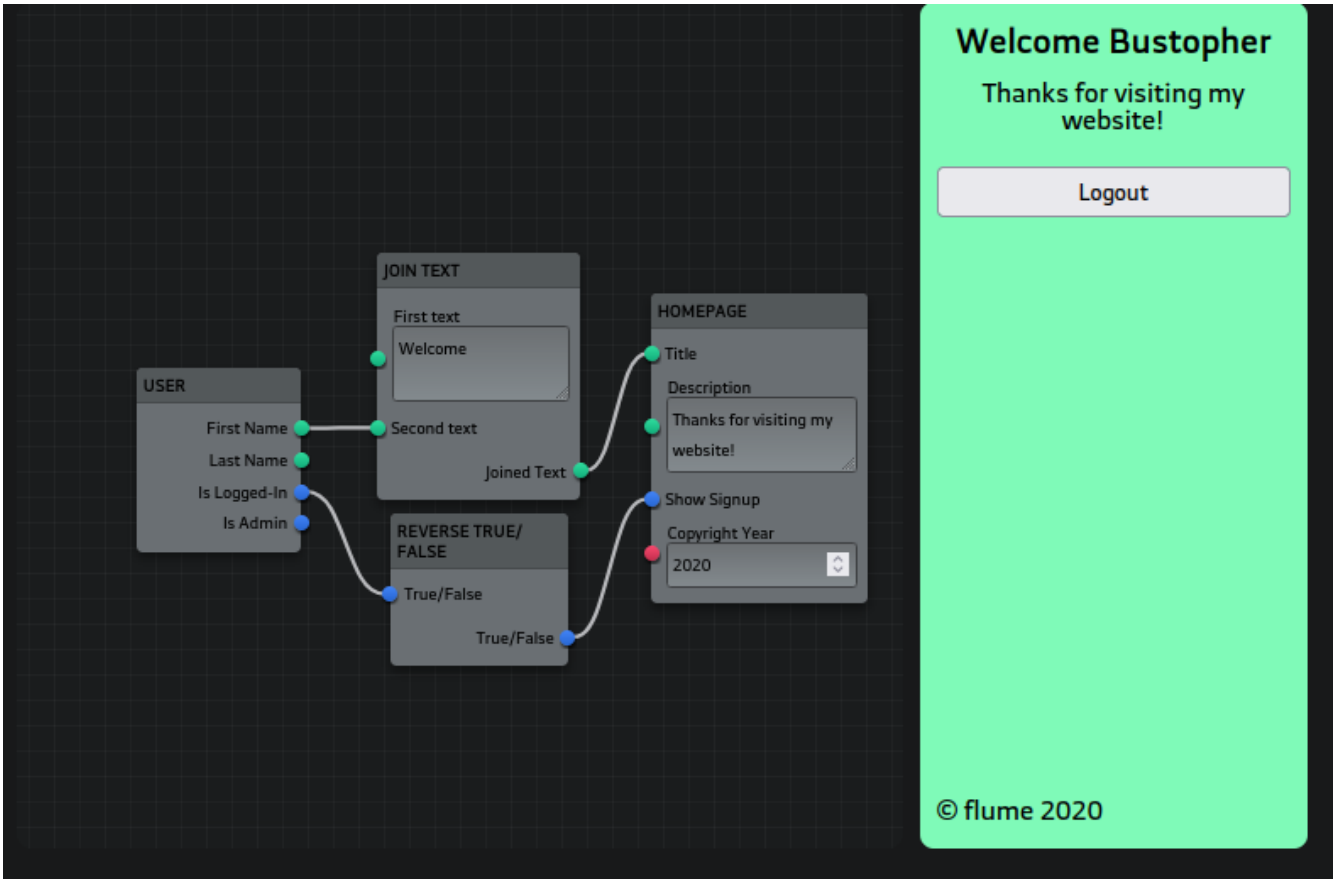
```
:
<button onClick={login}>Login</button>
}
{
showSignup &&
<form className="signup">
<input type="email" />
<button>Signup!</button>
</form>
}
<footer>© flume {copyrightYear}</footer>
</div>
)
}
export default Homepage
```

Boom! That's all we need to hook up our engine to the homepage component. Every time this component re-renders, the useRootEngine hook will run the logic defined in the node editor, and return the attributes we defined on the root node.

note

Remember our context object from the previous section? The third parameter of useRootEngine is that context object, and it's how you can pass data into the root engine. In this case we're just passing it our user. Because our component will re-render any time the user changes, our root engine will always have the latest user object.

Play with the example below to see it live. If you need more space to add nodes, you can use your scroll wheel on a mouse, or pinch gestures on a trackpad, to zoom in and out. To pan around, just click and drag anywhere inside the editor.



Summary#

Are you starting to catch the vision? What we've done is abstracted all of the "business logic" for our app into a logic graph. Now we can visually program this homepage, and make changes any time we want without needing to update any of our code. Can you think of some other useful nodes you could build?

Hello {Name}! You are lucky number {Number}!!
Alice

If you wanted to use the built-in RootEngine to run this node then you'd add something like this to resolveNodes:

```
case "compose":
const { template, ...inputs } = inputValues
const re = /\{(.*?)\}/g
const message = template.replace(re, (_, key) => inputs[key])
return { message }
```

Dynamically defining outputs#

Now let's do the same thing but for outputs. We're going to re-create our JoinText node from earlier but with one important twist: We'll only expose the output if both inputs are valid.

```
import { FlumeConfig, Colors, Controls } from 'flume'
const config = new FlumeConfig()
config
/* ... */
.addNodeType({
type: "joinText",
label: "Required Join Text",
description: "Combines two strings of text into one string",
initialWidth: 160,
inputs: ports => [
ports.string({
name: "string1",
label: "First text"
}),
ports.string({
name: "string2",
label: "Second text"
})
```

```
],
outputs: ports => (data, connections) => {
if (!data.string1.string && !connections.inputs.string1) return []
if (!data.string2.string && !connections.inputs.string2) return []
return [ports.string({ label: "Joined Text" })]
}
})
```

This node takes in two strings and joins them. We check each input to make sure that the user has typed something in or connected a node to it. If the input doesn't have a value then we return an empty array to signify that no outputs should be rendered. Try it out!

Note: Be very careful when restricting what outputs are available like this. It can potentially have a chain reaction effect that can frustrate users. Try out the following example. Delete the text that says "delete me". What happened? Invalidating the first node removed its output, which invalidated the second node, removing it's output, etc.

delete me
test

Summary#

Here we've created some pretty powerful nodes that can adapt to their inputs, connections, and context. Take a minute and think of the different nodes you might build that could benefit from this feature while solving your use cases.

Thanks for visiting my website!
Welcome

Saving comments#

Comments have no effect on the function of your logic graphs, so they aren't stored with the nodes. Like with nodes, there are two ways to get comemnts in and out of the node editor.

onCommentsChange#

First, you can pass an onCommentsChange handler to the node editor. This works the same as the normal onChange handler, but will only be called any time any of the comments change.

```
import React from 'react'
import { NodeEditor } from 'flume'
import config from './config'
const App = () => {
const [comments, setComments] = React.useState({})
return (
<div style={{width: 800, height: 600}}>
<NodeEditor
comments={comments}
onCommentsChange={setComments}
portTypes={config.portTypes}
nodeTypes={config.nodeTypes}
/>
</div>
)
}
```

getComments#

As with the onChange handler, the onCommentsChange handler may be called quite often. There may be cases where you may not need to save the comments until the user is done editing. For these cases, the node editor exposes an imperative getComments function.

```
import React from 'react'
import { NodeEditor } from 'flume'
import config from './config'
const App = () => {
const nodeEditor = React.useRef()
const saveComments = () => {
const comments = nodeEditor.current.getComments()
// Do whatever you want with the comments
}
return (
<div>
<button onClick={saveComments}>Save</button>
<div style={{width: 800, height: 600}}>
<NodeEditor
ref={nodeEditor}
portTypes={config.portTypes}
nodeTypes={config.nodeTypes}
/>
</div>
</div>
)
}
```

Theming

All required styles are automatically included with the NodeEditor, but each Flume component has a data attribute that you can target using normal css.

Example#

For example, you can change the background color of the nodes with the following CSS:

```
[data-flume-component="node"]{
background: #ffffff;
}
```

Reference#

The following is a comprehensive list of all available data attributes. Each component name is prefixed with data-flume-component.

- stage
- node
- node-header
- connection-svg
- connection-path
- ports
- ports-inputs

- ports-outputs-
- port-input
- port-output
- port-label
- port-handle
- comment
- comment-textarea
- comment-text
- comment-resize-handle
- control
- control-label
- select
- select-label
- select-desc
- text-input
- text-input-number
- text-input-textarea
- checkbox
- checkbox-label
- color-picker
- color-button
- toast
- toast-title
- toast-message
- toast-close
- ctx-menu
- ctx-menu-header
- ctx-menu-title
- ctx-menu-input
- ctx-menu-list
- ctx-menu-empty
- ctx-menu-option

Theming based on node type#

In some cases you may want to style some nodes based on their type. There are a few dynamic data attributes you can use for this.

- data-flume-node-type=[nodeType]
- data-flume-component-is-root=[boolean]

- data-port-color=[color]
- data-port-name=[portName]
- `data-port-type=[portType]

For example, to target a node with type addNumbers you could add this to your CSS:

```
[data-flume-node-type="addNumbers"] {
background: linear-gradient(to top, #4e2020 0%, #20204e 100%);
border: 1px solid rgba(255,255,255,.5);
color: #fff;
}
[data-flume-node-type="addNumbers"] [data-flume-component="node-header"] {
background: none;
}
```

API:

NodeEditor

NodeEditor is a React component for rendering the node editor. It requires portTypes and nodeTypes, and can be rendered with or without existing nodes.

Zooming & Panning#

By default, the node editor can be zoomed in and out using the scroll wheel, and panned by dragging anywhere on the background. In this release these are the only methods of zooming and panning, but an upcoming release will likely add an optional UI for using these features.

Importing#

```
import { NodeEditor } from "flume";
```

Props#

portTypes : object#

Required. portTypes are generated using the FlumeConfig class.

nodeTypes : object#

Required. nodeTypes are generated using the FlumeConfig class.

nodes : object#

Optional. An object of nodes previously generated through the node editor.

comments : object#

Optional. An object of comments previously generated through the node editor.

defaultNodes : array<object>#

Optional. For many logic graphs, it's helpful to have the editor start with a set of default nodes. To define these nodes, pass an array of objects with the type of the node, and an x and y position relative to the center of the editor.

Key	Type	Default	Required	Description
type	string	""	Required	The type key of the node you want to add as a default. This type must have been previously defined in the nodeTypes object.
x	int | float	0	Optional	The **horizontal** starting position of the node, relative to the center of the node editor.
y	int | float	0	Optional	The **vertical** starting position of the node, relative to the center of the node editor.

onChange : function#

Optional. This function will be called any time any of the nodes in the node editor change. This includes their position, connections, and the values of their controls.

note

For performance reasons, this function will **not** be called while the user is dragging a node. As soon as the user releases a dragged node, this function will be called with the updated position.

onCommentsChange : function#

Optional. This function will be called any time any of the comments in the node editor change. This includes their position, size, color, and text.

note

For performance reasons, this function will **not** be called while the user is dragging or resizing a comment. As soon as the user releases a dragged comment, this function will be called with the updated comment.

context : object#

Optional. Context is used to pass data to the node editor if your nodes require live data. For example, you may have a node with a dropdown control for selecting a color, but the available colors depend on the state of your app. To allow this kind of calculation, the context object is provided to your node at runtime to aid in these kinds of calculations.

initialScale : int | float#

Optional. The initial level of zoom for the node editor. This must be a number between 0.1 and 7. If a number is provided outside of this range it will be clamped to the nearest valid number. Defaults to 1.

disableZoom : boolean#

Optional. Prevents the user from zooming in and out of the node editor. This can be helpful if your node editor is on a scrollable page, and you don't want to capture scroll events as the user's mouse passes the editor. Defaults to false.

disablePan : boolean#

Optional. Prevents the user from panning inside of the node editor. Defaults to false.

spaceToPan : boolean#

Optional. If true, the spacebar must be depressed in order to pan the node editor. The cursor will also be updated to show a draggable indicator when the spacbar is depressed.

hideComments : boolean#

Optional. If the current node editor contains comment blocks, they will not be rendered. This can be helpful for providing a UI for toggling comments on and off if desired. Defaults to false.

disableComments : boolean#

Optional. Prevents the user from adding additional comments. Defaults to false.

disableFocusCapture : boolean#

Optional. Prevents the node editor from capturing focus when mousing over the editor. This can help prevent focus bugs in complex applications. Defaults to false.

circularBehavior : string#

Optional. Can be one of 3 possible values:

- prevent: Default. When a user attempts to connect nodes in a way that would create a circular graph, the connection is rejected and a warning is displayed.
- warn: Creating circular connections is allowed, but a warning is displayed.
- allow: Creating circular connections is allowed without a warning.

renderNodeHeader : function#

Optional. Allows you to override the rendering of the node header. Common use-cases might include overriding the label of a node, or adding custom UI elements like a button to delete the node, or open the node context menu. This prop accepts a function which is expected to return jsx.

Function parameters#

The renderNodeHeader function provides three parameters to help you render a custom header. Together they make up the following function signature:

```
(Wrapper, nodeType, actions) => jsx;
```

Wrapper#

Wrapper is a React component in which you can render your custom UI. Usage of the wrapper is not required, but is recommended for accessibility and consistency with the other components of the UI.

nodeType#

nodeType represents the current type definition of the node being rendered. For documentation for the properties available on nodeType see FlumeConfig.

actions#

Actions is an object of functions which operate in the context of the current node. Currently the following actions are available:

- openMenu: Opens the context menu for the node.
- closeMenu: Closes the context menu for the node.
- deleteNode: Deletes the current node.

FlumeConfig

The FlumeConfig class is a helper class for configuring the nodeTypes and portTypes that the NodeEditor and RootEngine need. It uses a chainable API for adding types.

Importing#

```
import { FlumeConfig } from 'flume'
```

Creating a new config#

```
const config = new FlumeConfig()
```

Editing an existing config#

In some cases you may have JSON representing an existing config that you'd like to edit. In these cases, FlumeConfig will accept an existing config as the first parameter of the constructor like this:

```
const config = new FlumeConfig(existingConfig)
```

addPortType : method(portType)#

This method allows you to add a new port type to your config.

Example#

```
config
.addPortType({
type: "string",
name: "string",
label: "Text",
color: Colors.green,
controls: [
Controls.text({
name: "string",
label: "Text"
})
]
})
```

type : string#

Required. The type key for the new port. It's recommended to use camel-case for type keys as you will later call this type as a function when building node types. This type must be unique between port types in the same config.

name: string#

Required. The default name for the new port. This is the name that will be used to identify this port on each node. For simplicity, it's a good practice to have the name be the same as the type, but this is not required. When adding a port to a node type, this name may be overridden.

label : string#

Required. A human-readable label for this port. This label will be shown on nodes when it is used as an output, when another node is connected to this port, or when this port has no controls. When adding a port to a node type, this label may be overridden.

color : color#

Optional. By default, all ports are a neutral gray color. To change this color, use the Colors object imported from flume to select an available color. For a list of available colors see Colors

acceptTypes : array<port.type>#

Optional. By default, each port will only accept connections from ports of the same type. In some cases you may have ports of different types which are logically compatible with each other. For these case you may pass an array of port types that this port may be connected to.

For example, let's say I have a port type of "string", and a port type of "color". In this case both of these ports return a Javascript string. If I would like the string port to also accept connections from color ports I would pass an array to the acceptTypes key like this:

```
.addPortType({
type: "color",
name: "color",
label: "Color",
acceptTypes: ["string", "color"],
controls: [...]
})
```

hidePort : boolean#

Optional. By default each input displays a port that other nodes can connect to. In some cases you may have an input that shouldn't accept any connections. Passing true to this key will hide this port but not hide the port label or controls.

controls : array<Control>#

Optional. An array of controls that will be displayed to the user when this port is used as an input, and is not connected. For a list of available controls see Controls

note

While this key is optional, it is rare that you will need to create a port without any controls.

addNodeType : method(nodeType)#

This method allows you to add a new node type to your config.

Example#

```
config
.addNodeType({
type: "string",
label: "Text",
description: "Outputs a string of text",
initialWidth: 160,
inputs: ports => [
ports.string()
],
outputs: ports => [
ports.string()
]
})
```

type : string#

Required. The type key for the new node. This type must be unique between node types in the same config.

label : label#

Required. A human-readable label for this node. This label will be displayed above each node of this type, and in the context menu for adding nodes.

description : string#

Optional. A brief description for this node. This description will be shown below the label in the context menu for adding nodes.

initialWidth : int | float#

Optional. Defaults to 200. The initial width of the node in the editor.

sortIndex : int | float#

Optional. By default, the available nodes in the "Add Node" context menu are sorted alphabetically by their label. This property lets you manually sort your nodes. If this property is present, nodes will be sorted in order by their index before being sorted alphabetically.

addable : boolean#

Optional. Denotes if users are allowed to add this node to the node editor. If this property is set to false, you may still add instances of this node using the defaultNodes key. Defaults to true.

deletable : boolean#

Optional. Denotes if users are allowed to delete this node in the node editor. Defaults to true.

root : boolean#

Optional. Normally you should add root nodes using the addRootNodeType method, however, you may mark any node as a root node by passing root: true.

inputs : function#

Optional. A function which must return either an array of inputs or another function that itself returns an array of inputs for this node. These inputs must be made up of port types which have previously been added to the config. The function you provide to this key will be injected with an object containing all of the available ports as the first parameter. Properties of each port may be overridden inline, which is useful for changing things like labels and names. The name of each input **must** be unique, even if they are different types.

Example#

Let's say you have already added 2 port types, a boolean port type, and a number port type. We're adding a new node type for calculating the total price of a cart on an e-commerce website. We would create the inputs like this:

```
config
/*...*/
.addNodeType({
type: "calculateTotal",
label: "Calculate Total",
description: "Calculates the total price of the checkout",
inputs: ports => [
ports.number({ name: "subtotal", label: "Subtotal"}),
ports.number({ name: "taxRate", label: "Tax Rate"}),
ports.boolean({ name: "isTaxExempt", label: "Is Tax-Exempt"})
],
outputs: ports => [
ports.number({ name: "total", label: "Total"})
]
```

```
})
```

In this example we've added 3 inputs for the subtotal, the tax rate, and whether or not the tax should be applied. Because we've already defined the number and boolean ports, the inputs function will provide those port types for us to use. We call each port type as a function and optionally give it a name and a label. In this case, this is required to make sure that each port has a unique name, and to add a user-friendly label to each port.

noControls : boolean#

In some cases you may want an input to hide its controls without being connected. Passing true to this key will hide the controls for this instance of the port. Defaults to false.

Advanced Inputs Usage#

Flume supports defining inputs dynamically at runtime. To do so you need to define your inputs with a signature like this: inputs: ports => (inputData, connections, context) => []. Your function will be invoked any time one of its arguments changes and its return value will be used as the new set of inputs. The first two arguments (inputData and connections) refer to the information that is filled out using Controls on your node and the list of connections entering/leaving your node respectively. The last argument, context, is the same context you pass into the NodeEditor or RootEngine. For more details check out the Dynamic Nodes guide.

outputs : function#

Optional. A function which must return either an array of outputs or another function that itself returns an array of outputs for this node. These outputs must be made up of port types which have previously been added to the config. The function you provide to this key will be injected with an object containing all of the available ports as the first parameter. Properties of each port may be overridden inline, which is useful for changing things like labels and names. The name of each output **must** be unique, even if they are different types.

Example#

Let's say you have already added 2 port types, a string port type, and a number port type. We're adding a new node type for inputting and outputting text and the character and word counts:

```
config
/*...*/
.addNodeType({
type: "string",
label: "Text",
description: "Outputs a string of text",
inputs: ports => [
```

```
ports.string()
],
outputs: ports => [
ports.string(),
ports.number({ name: "charCount", label: "Character Count"}),
ports.number({ name: "wordCount", label: "Word Count"}),
]
})
```

In this example we've created 3 outputs, one with the text of the input, one with the character count, and one with the word count. Because we've already defined the string and number ports, the outputs function will provide these port types for us to use. We call each port type as a function and optionally give it a name and a label. In this case, this is required to make sure that each port has a unique name, and to add a user-friendly label to each port.

Advanced Inputs Usage#

Flume supports defining outputs dynamically at runtime. To do so you need to define your outputs with a signature like this: outputs: ports => (inputData, connections, context) => []. Your function will be invoked any time one of its arguments changes and its return value will be used as the new set of outputs. The first two arguments (inputData and connections) refer to the information that is filled out using Controls on your node and the list of connections entering/leaving your node respectively. The last argument, context, is the same context you pass into the NodeEditor or RootEngine. For more details check out the Dynamic Nodes guide.

addRootNodeType : method(nodeType)#

The addRootNodeType method takes all the same parameters as the addNodeType method. This method adds a node type to the config and automatically marks it as a root node. It also prevents this node type from being added or deleted by users. For node types added through this method to be useable in the node editor, they must be added as a default node, or provided to the nodes key of the node editor.

RootEngine

RootEngine is a class for creating a new engine for executing your logic graphs. For more information see Running Logic.

Importing#

```
import { RootEngine } from 'flume'
```

Creating a new root engine#

```
const engine = new RootEngine(config, resolvePorts, resolveNodes)
```

resolvePorts : function#

resolvePorts is the second parameter of the RootEngine constructor. This function is responsible for returning the resolved control values of a port every time it's called.

Example#

Typically this function is a switch statement that might look something like this:

```
const resolvePorts = (portType, data, context) => {
switch (portType) {
case 'string':
return data.string
case 'boolean':
return data.boolean
case 'number':
return data.number
default:
return data
}
}
```

Returns : any#

This function must return the control values for the port currently being processed. In **most** cases, ports have only one control, so this function simply returns the value of that control out of the data object. If the port being processed has more than one control, this function *typically* returns an object with each key being the name of each control, and the value being its value.

portType : string#

This is the first parameter injected into the resolvePorts function. It refers to the string type of the port currently being resolved.

data : object#

An object representing all of the control values of the current port.

context : object#

An object representing the current execution context of the root engine. This context object is used to pass live data into the root engine at runtime.

resolveNodes : function#

resolveNodes is the third parameter of the RootEngine constructor. This function is responsible for transforming any given inputs to the current node, into an object representing all the outputs of the node.

Example#

Typically this function is a switch statement that might look something like this:

```
const resolveNodes = (node, inputValues, nodeType, context) => {
switch (node.type) {
case 'string':
return { string: inputValues.string }
case 'boolean':
return { boolean: inputValues.boolean }
case 'number':
return { number: inputValues.number }
case 'user':
return context.user
case 'joinText':
return { joinedText: inputValues.string1 + inputValues.string2 }
case "reverseBoolean":
```

```
return { boolean: !inputValues.boolean }
default:
return inputValues
}
}
```

Returns : object#

This function must return an object representing every output available on the current port. Regardless of whether or not they are currently connected to a port on another node. Each key of this object should correspond to the name of each output port.

node : Node#

An object representing the current state of the node being processed.

inputValues : object#

An object representing the resolved control values returned by the the resolvePorts function.

nodeType : NodeType#

The type definition of the node being processed.

context : object#

An object representing the current execution context of the root engine. This context object is used to pass live data into the root engine at runtime.

resolveRootNode : method#

A function which accepts a set of nodes created through the node editor, and an optional options object; and returns an object representing all of the resolved values of the inputs for the root node.

note

This method must be called with nodes having exactly one root node. If multiple root nodes are found, the first root node found will be used. In most cases this is not a desired behavior. If you need to support multiple root nodes you should manually keep track of their node ids, and resolve them seperately by passing their ids to the rootNodeId key of the options object, as described below.

options : object#

A configuration object representing settings for the requested resolution of the root node.

context : object#

Optional. An object which will be passed to both the resolvePorts and resolveNodes functions. This context object is used to pass live data into the root engine at runtime.

onlyResolveConnected : boolean#

Optional. If this property is set to true, the object returned by the resolveRootNode function will only contain port values for inputs which are connected to other nodes. Defaults to false.

rootNodeId : string#

Optional. In some cases you may wish to override which node is considered the root node. Passing the id to this property, of a node in the set of nodes passed into the resolveRootNode function, will cause the corresponding node to be used as the root node.

maxLoops : int#

Optional. By default, the root engine will only allow up to 1000 recursive loops through nodes before it will bail out of processing the current root node input and log an error. This prevents the call stack from being overwhelmed by a circular node setup. Passing an integer to this property will change the number of max loops before this occurs. Passing a -1 will disable this check and allow infinite loops.

Multiselect#

The multiselect control renders a dropdown control for selecting multiples from a set of options. The selected option values are stored in an array.

Example#

```
.addPortType({
type: "colors",
name: "colors",
label: "Colors",
color: Colors.purple,
controls: [
Controls.multiselect({
name: "colors",
label: "Colors",
options: [
{value: "red", label: "Red"},
{value: "blue", label: "Blue"},
{value: "yellow", label: "Yellow"},
{value: "green", label: "Green"},
{value: "orange", label: "Orange"},
{value: "purple", label: "Purple"},
]
})
```

]

options : array<option>#

Required. An array of objects representing the available options.

Option: value : string#

Required. The value of the option. When this option is selected, this will be the value stored as the value for this control.

Option: label : string#

Required. A human-readable label for the option.

Option: description : string#

Optional. A brief description for the option. This description will be displayed in the dropdown menu when the user is selecting an option.

defaultValue : array#

Optional. The initial value for the select. Defaults to an empty array.

Custom#

While the built-in controls cover the majority of use-cases, you may occasionally want to render a custom control. This is a powerful option for providing custom user interfaces for your nodes.

Example#

```
.addPortType({
type: "randomPhoto",
name: "randomPhoto",
label: "Random Photo",
color: Colors.pink,
controls: [
Controls.custom({
name: "randomPhoto",
```

```
label: "Random Photo",
defaultValue: "D6Tqla-tWRY", // Random Unsplash photo id
render: (data, onChange) => (
<RandomPhotoComponent
url={data}
onChange={onChange}
/>
)
})
]
})
```

In this example, we've created a new port with a custom control. In our custom control we'ved added a render function which returns a React component with our custom UI. To save the value of our control, we're provided an onChange function we can call whenever the value of our custom control changes.

render : function#

Required. The render key accepts a function that returns a React component. This component will be rendered in place of the control for this port. The render function takes in several parameters described below in order.

```
(
data,
onChange,
context,
redraw,
portProps,
inputData
) => (
<ReactComponent />
)
```

render : data : any#

The current value for the control. On the intial render this will be equal to the default value of the control.

render : onChange : function#

An updater function to set the new value of the control anytime it changes.

render : context : object#

The optional context object provided to the node editor.

render : redraw : function#

For built-in controls, the node editor stage is redrawn whenever a control changes in such a way that the dimensions of its parent node changes. By default, the node editor is also redrawn any time the onChange function provided in this render function is called. On some occasions you may build a control which changes the dimensions of the node without calling the onChange function. For these times you are responsible for notifying the node editor that the stage needs to be redrawn by calling the redraw function.

note

For performance reasons it's recommended that you limit how often this function is called.

render : portProps : object#

The portProps provides a handful of properties that can be useful in especially advanced use-cases. The properties of this object are described below:

portProps : label : string#

The label of the control.

portProps : inputLabel : string#

The label of this control's parent port.

portProps : name : string#

The name of the control.

portProps : portName : string#

The name of this control's parent port.

portProps : defaultValue : any#

The default value of this control.

render : inputData : object#

An object representing all of the data for the inputs of the current node. This can be useful for rendering controls which rely on the current values of controls in other inputs.

Colors

Flume exports a set of colors to make it easy to change the color of your ports. An upcoming release will provide the ability to define your own colors, but until then, the built-in colors have been hand-picked to be visually distinct and coordinate perfectly with each other.

Importing#

```
import { Colors } from 'flume'
```

Example#

Port colors are chosen when adding port types to your flume config. Let's say you want to create a blue port representing an employee, you could do so like this:

```
.addPortType({
type: "employee",
name: "employee",
label: "Employee",
color: Colors.blue,
controls: [
/* ... */
]
})
```