Axway Flow SDK

The Axway Flow SDK (`axway-flow-sdk`) is a standalone utility that enables the creation of custom flow-nodes for API Builder flows. By offering the Axway Flow SDK as a standalone utility, new flow-nodes can be developed and consumed in API Builder without upgrading the version.

Features

The Axway Flow SDK has the following content:

- CLI tool for starting a new flow-node project
- SDK for building custom modules for API Builder flows

Install

The following command installs the Axway Flow SDK.

```
npm install -g axway-flow-sdk
```

Use the Axway Flow SDK

The following generates a new flow-node starter project in the current directory. You can customize the starter project to meet your requirements.

```
axway-flow -n <node name>
cd <node name>
npm install
npm run build
```

The generated starter project name is prefixed with the required `nodehandler-` prefix.

The starter project is heavily commented to simplify the process of customizing it. It also comes with the `eslint` configuration and the `mocha` unit tests incorporated to help you ensure the quality of your custom flow-node.

Sample encodeURI flow-node

As an example of how to write a flow-node, we will examine creating a flow-node that URI encodes a string.

Create the project

```
axway-flow -n encodeuri -d “URI encoder.”
cd nodehandler-encodeuri
npm install
npm run build
```

Customize the flow-node definition in the index.js file

Expand source

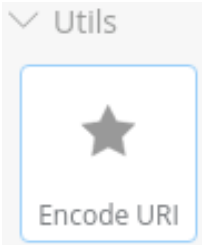
```
const sdk = require('axway-flow-sdk');
const action = require('./action');

const flownodes = sdk.init(module);

flownodes
    .add('encodeuri', {
        name: 'Encode URI',
        icon: 'icon.svg',
        description: 'URI encoder.',
        category: 'utils'
    })
    .method('encode', {
        name: 'Encode URI',
        description: 'Encodes a URI by replacing each instance of certain characters
with UTF-8 encodings.'
    })
    .parameter('uri', {
        description: 'The URI to encode.',
        type: 'string'
    })
    .output('next', {
        name: 'Next',
        description: 'The URI was encoded successfully.',
        context: '$.encodedURI',
        schema: {
            type: 'string'
        }
    })
    .action(action);
exports = module.exports = flownodes;
```

To explain what occurs in the `index.js` file, we will break the file down piece by piece.

1. Describe the flow-node, name, description, category, and icon:

   ```
   .add('encodeuri', {
           name: 'Encode URI',
           icon: 'icon.svg',
           description: 'URI encoder.',
           category: 'utils'
   })
   ```

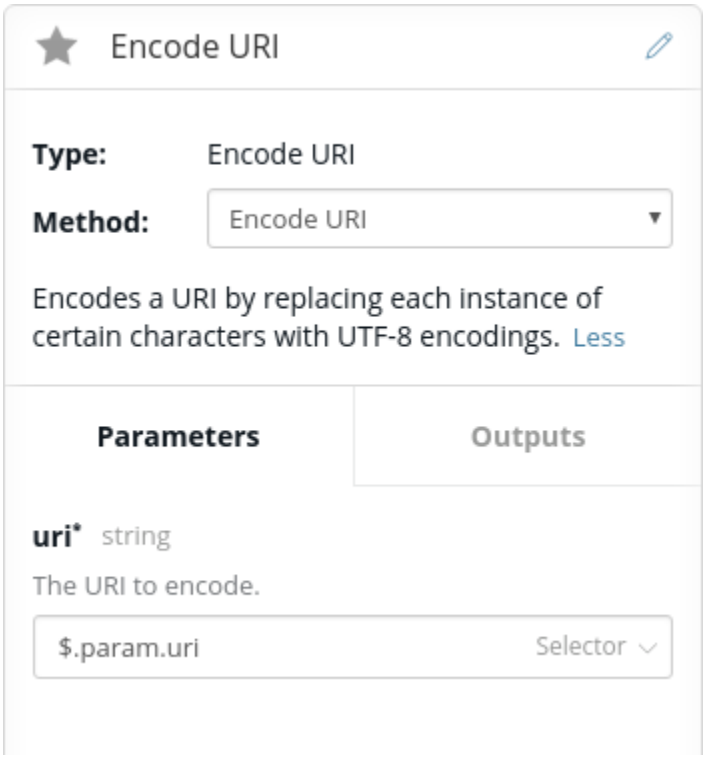
   The `name` is the text that is displayed in the Flow Editor. The default `icon` is a placeholder (a star) that should be replaced with a graphic that represents the action of the flow-node. The icon is displayed at 28 pixels x 28 pixels. The `category` is the section in the Flow Editor tool panel where the flow-node is contained.



2. Add a method to the flow-node and describe its parameters:

```
.method('encode', {
        name: 'Encode URI',
        description: 'Encodes a URI by replacing each instance of certain
characters with UTF-8 encodings.'
    })
    .parameter('uri', {
        description: 'The URI to encode.',
        type: 'string'
})
```

A method called `encode`, that is displayed in the Flow Editor as **Encode URI**, was added. The `encode` method has a single parameter. If there was more than one parameter, we would repeat the `.parameter(name, schema)` block. The second value in the parameter method is a JSON Schema that describes the parameter type.



3. Describe the possible outputs from the method:

```
.output('next', {
    name: 'Next',
    description: 'The URI was encoded successfully.',
    context: '$.encodedURI',
    schema: {
        type: 'string'
    }
})
```

The outputs section defines the possible outcomes of the flow-node. In this simple case there is just one output; however, flow-nodes can have multiple outputs with different return types. For example, this flow-node could have added an **error** output to indicate that encoding failed.

4. Define the implementation:

```
.action(action);
```

The `action()` expects a function that will be passed the request details parameter and a callback object parameter.

Customize the flow-node method implementation

To simplify the management of the code, the starter project puts the implementation of the methods in the `action.js` file. There is not a requirement to follow this pattern, you can structure your project how best suits your needs.

```
exports = module.exports = function (req, cb) {
        const uri = req.params.uri;
        if (!uri) {
                return cb('invalid argument');
        }
        cb.next(null, encodeURI(uri));
};
```

This is a simple scenario, but it highlights the main features. The parameters for the flow-node method are accessed under the `req.params` para meter. In this example, the parameter for the `encode` method is defined as `uri`:

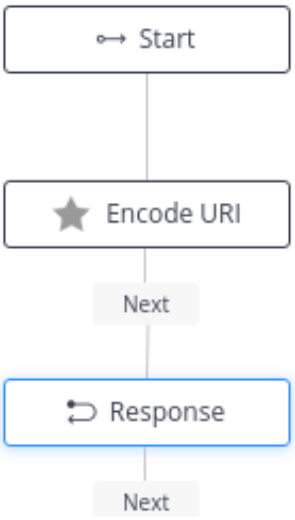
```
.parameter('uri', {
        description: 'The URI to encode.',
        type: 'string'
    })
```

The logic checks that the parameter is set. If `uri` is not set, it fires a generic error callback.

```
return cb('invalid argument');
```

These errors are not handled and will abort the flow execution. In general, avoid doing this for any expected error scenarios. If there are known error situations, it is better to define an output for those scenarios and allow the flow designer the flexibility to specify what to do when an error occurs.

If `uri` is set, the fallback for the next output is fired. The name of this callback will match the name of the output defined in the method. For example, if you defined an output `encoderError`, then there would be a callback `cb.encoderError()`. The encoded string is passed to the callback as the methods output value.



Unit test your flow-node

The starter project includes automatically generated unit tests in the `./test` directory of your project. The tests are for example purposes and will need to be customized to your specific use case. The SDK provides a *`mocknode`* that allows for mock invocations of your flow-node methods.

Use mocknode to test error callback existence - valid argument

This example uses mocha to check that the specification is defined well enough to pass the `uri` argument to the method. It also mocks the callback using the defined output of the specification and ensures that the method invokes the correct callback.

```
it('[TEST-2] should succeed', () => {
        return mocknode(specs).node('encodeuri')
            .invoke('encode', { uri: 'some string' })
            .then((data) => {
                expect(data).to.deep.equal({
                        next: [ null, 'some%20string' ]
                });
            });
    });
```

Using mocknode to test error callback existence - invalid argument

This example is similar to the previous example, except that the method will invoke a `cb('invalid argument')` when given an undefined parameter.

```
it('[TEST-3] should fail to with invalid argument', () => {
        return mocknode(specs).node('encodeuri')
            .invoke('encode', { uri: null })
            .then((data) => {
                expect(data).to.deep.equal(
                [ 'invalid argument' ]
            );
            });
    });
```

Testing that validity of the flow-node specification

The Axway Flow SDK tries to prevent the creation of invalid flow-node specifications, but there are some edge cases where it may be possible to generate a flow-node specification that is invalid at runtime. To detect this, the generated specification should be validated as part of your unit tests.

```
it('[TEST-4] should define valid node specs', () => {
        expect(validate(specs)).to.not.throw;
    });
```

Local API Builder test

While unit testing is important, it is also necessary to be able to install the custom flow-node into your local API Builder application for testing. This can be achieved by packing the module locally:

```
cd nodehandler-encodeuri
npm install
npm run build
npm pack
```

This will create a `tgz` archive (`nodehandler-encodeuri-1.0.0.tgz`) that can then be installed into your Arrow application.

```
cd <app-folder>
npm install <path to flow node project>/nodehandler-encodeuri-1.0.0.tgz
appc run
```

Type references

In Axway API Builder, it is possible to reference other types. For example, types can be loaded from `./schemas`, registered from models, or registered from service connectors. Any registered schema can be referenced whenever a `schema` is required.

```
.parameter('greeting', {
    "$ref": "schema://model/appc.arrowdb/user"
})
.output('next', {
    schema: {
        "$ref": "schema://model/appc.arrowdb/user"
    }
})
```

API Reference

axway-flow-sdk~NodeBuilder

Kind: Inner class of `axway-flow-sdk`

- `NodeBuilder`
    - `.add(key, [options])`
    - `.method(key, [options])`
    - `.parameter(name, schema, [required])`
    - `.output(key)`
    - `.action(handler)`

nodeBuilder.add(key, [options])

Adds a new flow-node specification and prepares the `NodeBuilder` to accept the following specification operations:

- `.method(key, [options])`
- `.output(key, [options])`

The `key` parameter is used to uniquely identify the specification and represents a distinct instance of a flow-node for the flow editor. The `key` will be used as the name unless the `name` option is provided. The new flow-node will appear in the general category by default, or under the provided `category` option.

The `icon` option can be a bmp, jpeg, png, gif, tiff, or svg file. The `.method` option is used to add a method or methods, and the `.output` option is used to define an output. The `.action` option is used to define an action function and finish the specification.

Kind: Instance method of `NodeBuilder`

Returns: Current `NodeBuilder` object

Access: Public

Parameter	Type	Default	Description
`key`	string		A unique key identifier for the flow-node.
`[options]`	object		Options for the flow-node.
`[options.name]`	string		A friendly name for the flow-node as it will appear in the UI.
`[options.icon]`	string		An icon file.
`[options.description]`	string		A description for the flow-node.
`[options.category]`	string	`general`	A category under which the flow-node will appear in the UI.

Example:

```
sdk.init(module).add('encodeURI', { icon: 'encode.svg' });
```

nodeBuilder.method(key, [options])

Adds a new method to the current flow-node specification and prepares the `NodeBuilder` to accept the following method operations:

- `.parameter(name, schema, [required])`
- `.action(handler)`

The `.add(key, [options])` must be called prior to adding a method.

The `key` uniquely identifies the method for the flow-node and will be used as the name unless the `name` option is provided.

Kind: Instance method of `NodeBuilder`

Returns: Current `NodeBuilder` object

Access: Public

Parameter	Type	Description
`key`	string	A unique key identifier for the method.
`[options]`	object	Options for the method.
`[options.name]`	string	A friendly name for the method as it will appear in the UI.

Example:

```
sdk.init(module).add('encodeURI', { icon: 'encode.svg' })
    .method('encode', { name: 'Encode URI' });
```

nodeBuilder.parameter(name, schema, [required])

Adds a new parameter to the current method. Any number of parameters can be added to a method.

The `.method(key, [options])` must be called prior to adding a parameter.

The `name` uniquely identifies the parameter, and the `schema` is a valid JSON Schema definition (both draft-04 and draft-06 are supported).

Kind: Instance method of `NodeBuilder`

Returns: Current `NodeBuilder` object

Access: Public

Parameter	Type	Default	Description
`name`	string		A unique name for the parameter as it will appear in the UI.
`schema`	object		A schema used to validate the parameter.
`[required]`	boolean	`true`	A flag to indicate the parameter is required or optional.

Example:

```
sdk.init(module).add('encodeURI', { icon: 'encode.svg' })
    .method('encode', { name: 'Encode URI' })
    .parameter('uri', { type: 'string' });
```

nodeBuilder.output(key)

Adds a new output to the current method. Any number of outputs can be added to a method, but for usability-sake, you should limit this. The `output` represents one of the possible callback routes for your method. For example, if your method tested prime numbers, then one output might be `prime`, and the other `not-prime`.

The `.method(key, [options])` must be called prior to adding an output.

The `key` uniquely identifies the output route. The `schema` is a valid JSON Schema definition (both draft-04 and draft-06 are supported). If a `schema` is not provided, then the output type is effectively any type.

The `context` is a valid JSON Path and is used as the default by the flow editor. When the output is invoked, the configured context is where the output value will be written.

Kind: Instance method of `NodeBuilder`

Returns: Current `NodeBuilder` object

Access: Public

Parameter	Type	Description
`key`	string	A unique key for the output.
`[options.name]`	string	A friendly name for the output as it will appear in the UI.
`[options.description]`	string	The output description.
`[options.context]`	string	The default context string.
`[options.schema]`	object	The expected JSON schema for the output value.

Example:

```
sdk.init(module).add('encodeURI', { icon: 'encode.svg' })
    .method('encode', { name: 'Encode URI' })
    .parameter('uri', { type: 'string' })
    .output('encoded', { context: '$.encodedURI', schema: { type: 'string' } });
```

nodeBuilder.action(handler)

Assigns an `action` handler to the current method. The method can only have one action handler. Assigning an action will terminate the current method definition.

Kind: Instance method of `NodeBuilder`

Returns: Current `NodeBuilder` object

Access: Public

Parameter	Type	Description
`handler`	handler	The action `handler` function.

Example:

```
sdk.init(module).add('encodeURI', { icon: 'encode.svg' })
    .method('encode', { name: 'Encode URI' })
    .parameter('uri', { type: 'string' })
    .output('encoded', { context: '$.encodedURI', schema: { type: 'string' } })
    .action((req, cb) => cb.encoded(null, encodeURI(req.params.uri));
```

axway-flow-sdk~init(module) - NodeBuilder

Axway API Builder SDK for creating custom flow-nodes to work with flows.

Kind: Inner method of `axway-flow-sdk`

Returns: `NodeBuilder` - A newly constructed `NodeBuilder` object

Parameter	Type	Description
`module`	object	The flow-node module.

Example:

```
const sdk = require('axway-node-sdk');
exports = module.exports = sdk.init(module);
```

axway-flow-sdk~handler: function

A handler function to perform the flow-node method's action. The function will receive all of the provided parameters in `req.params`. If any parameters are not provided or are at the wrong time, or some have defaults, your function will need to handle those situations. On success, your function should invoke the named `output`. On error, your function should invoke a callback with a non-null `err` value.

Kind: Inner type definition of `axway-flow-sdk`

Access: Public

Parameter	Type	Description
`req`	request	The `Request` object.
cb	callback	The output `callback`.

Example:

```
cb.encoded(null, uncodeURI(req.params.uri));
```

Example:

```
cb('error!');
```

axway-flow-sdk~flowCallback: function

A callback function that your method `handler` must invoke.

Kind: Inner type definition of `axway-flow-sdk`

Access: Public

Parameter	Type	Description
`[err]`	*	A non `null` value indicates a terminal error (flow processing will stop).
`[value]`	*	The output value to be written back to the flow processing context.

axway-flow-sdk~Request: object

The request object.

Kind: Inner type definition of `axway-flow-sdk`

Properties:

Name	Type	Description
`env`	object	The application configuration.
`params`	object	The `params` method, as supplied during runtime (see `.parameter`).