API Builder Connector Project

API Builder 3.x is deprecated
Support for API Builder 3.x ceased on 30 April 2020. Use the v3 to v4 upgrade guide to migrate all your applications to API Builder 4.x.

Contact support@axway.com if you require migration assistance.

Introduction

A Connector project is similar in structure to a project. This guide covers how to manage your Connector project.

To add a connector to your project, see Add a Connector.

Project structure

A Connector project is made up of several components. To simplify development, API Builder primarily uses a strict directory structure and naming convention to organize the application rather than configuration files.

The following is a list of directories and files that can be found in a Connector project:

File/Folder Name	Description
`app.js`	The entry point to the connector for testing, which launches a server instance.
`appc.json`	Project configuration file. Do not modify this file.
`conf`	Contains configuration files in JSON format for the connector. The file `default.js` is used for testing the connector. You can create an example configuration file, which is copied to the project when it is installed. See the Connector Configuration File section.
`index.js`	The entry point to the connector.
`lib`	Contains the logic for your connector. Requires a `index.js` file. See the Connector Logic section.
`logs`	Contains generated log files when running your project locally. If you test the connector, the generated log files will get packaged with the application. You may want to disable logging by setting the `transactionLogEnabled` property to `false` in the `conf/default.js` file.
`models`	Contains Model JavaScript files, used to declare the schema for your data and generate API endpoints for the connector. See Models and API Endpoints below.
`node_modules`	Contains project dependencies. API Builder automatically installs any project dependencies declared in the `package.json` file.
`package.json`	NPM configuration file to declare project dependencies and other build or runtime configurations.

CLI tasks

Use the Appcelerator CLI to create, test and deploy your Connector project.

Create a connector

To create a new connector, from your workspace directory, execute the `appc generate` command. When prompted:

- Select **Component** for the type of component
- Select **Connector** for the component
- Select **Empty Connector Project** to use a boilerplate project.
- Enter a name and directory name for your project.

Expand source

```
$ appc generate
Appcelerator Command-Line Interface, version 0.2.230
Copyright (c) 2014-2015, Appcelerator, Inc.  All Rights Reserved.

? What type of component would you like to generate? Arrow Component
? What Arrow component would you like to generate? Arrow Connector
? Which Connector would you like to generate? Empty Connector Project
? What is the connector name? sample.connector
? Which directory to generate into? sample.connector
```

Test the connector

Like an API Builder project, you can locally run the Connector project and make APIs calls to it. From the project directory, execute:

```
appc run
```

Once the server starts, you can make cURL or other requests to the server. Open the admin console, then go to the **API Docs** tab to retrieve the cURL commands for the methods. Copy and paste a command in a terminal to test it.

Publish the connector

To publish the connector, execute the following command from the project directory:

```
appc publish
```

By default, the access level for the connector is set to private, so only the creator can access the connector. To share the connector with other people or publicly, specify a different access level with the `appc access` command and add people or organizations to your component using the `appc user`and commands`appc org`.

Connector logic

Place all the connector logic in the `lib` folder. There must be an `index.js` file in your `lib` folder, which is the first file loaded by the connector.

The boilerplate `index.js` file exposes a `create()` method, which is passed the API Builder class. The method must return a Connector instance, which is created using the `Arrow.Connector.extend()` method. Uncomment each Capability constant in the `capabilities` field to have API Builder generate boilerplate logic for each capability you want the Connector to support. Each method will have its own JavaScript file. You may also pass the `extend()` method an object, which implements the methods of the Connector class and a few methods from the Model class.

To start developing your connector, run the project in one console window, then edit the files with the connector logic in another console or editor. As you save your files, API Builder will automatically update your connector and restart the server instance, allowing you to work on and test the connector incrementally.

lib/index.js Expand source

```
/*
 Welcome to your new connector!
 TODO: First things first, look at the "capabilities" array TODOs down below.
 */
var _ = require('lodash');

/**
 * Creates your connector for Arrow.
 */
exports.create = function (Arrow) {
        var Connector = Arrow.Connector,
                Capabilities = Connector.Capabilities;

        return Connector.extend({
                filename: module.filename,
                capabilities: [
                        // TODO: Get started by uncommenting the next line and running
`appc run`.
                        //Capabilities.ConnectsToADataSource,


                        // TODO: Each of these capabilities is optional; add the ones
you want, and delete the rest.
                        // (Hint: I've found it to be easiest to add these one at a
time, running `appc run` for guidance.)
                        //Capabilities.ValidatesConfiguration,
                        //Capabilities.ContainsModels,
                        //Capabilities.GeneratesModels,
                        //Capabilities.CanCreate,
                        //Capabilities.CanRetrieve,
                        //Capabilities.CanUpdate,
                        //Capabilities.CanDelete,
                        //Capabilities.AuthenticatesThroughConnector
                ]
        });
};
```

Capabilities

API Builder is based on Arrow and starting with API Builder 1.2.48 or Appcelerator CLI 4.1.3, the connector configuration object passed to the `Connector.extend()` method supports a `capabilities` property. The first time you run a project after adding a Capability constant to the `capabilities` property, API Builder will generate boilerplate logic, which you can modify. The table below explains what each capability constant exposes and creates. If a certain connector is not exposed through a capability, you can implement the method in the object passed to the `Connector.extend()` method.

Capability Constant	Location	Exposed Connector Methods
`ConnectToADataSource`	`./lib/lifecycle`	• `connect` • `disconnect`
`ValidatesConfiguration`	`./lib/metadata`	• `fetchMetadata`
`AddsCustomTypes`	`./lib/metadata`	• `coerceCustomType` • `getCustomType`

`GenerateModels`	`./lib/schema`	• `createModelsFromSchema` • `fetchSchema`
`ContainsModels`	`./models`	Creates a boilerplate model in the `models` folder.
`CanCreate`	`./lib/methods`	• `create`
`CanRetrieve`	`./lib/methods`	• `distinct` • `findAll` • `findById` • `query`
`CanUpdate`	`./lib/methods`	• `findAndModify` • `save` • `upsert`
`CanDelete`	`./lib/methods`	• `delete` • `deleteAll`
`AuthenticatesThroughConnector`	`./lib/lifecycle`	• `login` • `loginRequired`

Initialization

If you need to add some custom initialization logic when creating the connector, implement the following methods. The connector instance is the value passed to `this` in the functions. The functions do not take any arguments or return any values:

Method Signature	Description
`constructor(void)`	Use to execute some custom logic when the connector is created.
`postCreate(void)`	Use to execute some custom logic after the connector instance is created but before it is returned.

Connection lifecycle

When the connector is loaded, the following methods are executed (in order and if defined). You do not need to implement any of the methods. Each method is passed a callback. After completing the operation, invoke the callback function and pass it an Error object (or null if successful) as the first parameter, and the results of the operation as the second parameter. None of the methods have a return value.

Method Signature	Capability	Boiler Plate File	Description	Result to Pass to the Callback
`fetchMetadata(callback)`	ValidatesConfiguration	`./lib/metadata/fetchMetadata.js`	Retrieves the metadata of the data source. The metadata is used to validate the configuration object.	Metadata object. Set the `fields` key to an array of Metadata objects to verify the keys in the configuration object.
`fetchConfig(callback)`	-	-	Retrieves the configuration of the data source.	Configuration object. Key-value pairs describing the configuration of the connector.
`connect(callback)`	ConnectToADataSource	`./lib/lifecycle/connect.js`	Connects to the data source.	None.
`fetchSchema(callback)`	GenerateModels	`./lib/schema/fetchSchema.js`	Retrieves the model schema of the data source.	Schema object.
`disconnect(callback)`	ConnectToADataSource	`./lib/lifecycle/connect.js`	Disconnect from the data source.	None.

CRUD methods

To access data from the Connector, you need to implement the following methods. Each method is passed the Model class as its first parameter and a callback as its last parameter. After completing the operation, invoke the callback function and pass it an Error object (or null if successful) as the first parameter, and the results of the operation as the second parameter. None of the methods have a return value.

Method Signature	Capability	Boiler Plate File	Description	Result to Pass to the Callback
`create(Model, values, callback)`	`CanCreate`	`./lib/methods/create.js`	Creates a new model using the passed values.	New model
`delete(Model, instance, callback)`	`CanDelete`	`./lib/methods/delete.js`	Deletes the model instance.	Deleted model
`deleteAll(Model, callback)`	`CanDelete`	`./lib/methods/deleteAll.js`	Deletes all the models.	An array of deleted models
`findAll(Model, callback)`	`CanRetrieve`	`./lib/methods/findAll.js`	Retrieves all the models.	An array of models
`findById(Model, id, callback)`	`CanRetrieve`	`./lib/methods/findById.js`	Retrieves one model with the specified ID (`id` parameter).	A model
`query(Model, options, callback)`	`CanRetrieve`	`./lib/methods/query.js`	Retrieves all models based on the query options.	An array of found models
`distinct(Model, field, options, callback)`	`CanRetrieve`	`./lib/methods/ditinct.js`	Retrieves a distinct set of models based on the field(s) and query options.	An array of distinct models
`save(Model, instance, callback)`	`CanUpdate`	`./lib/methods/save.js`	Updates the model instance.	Updated model
`findAndModify(Model, options, doc, args, callback)`	`CanUpdate`	`./lib/methods/findAndModify.js`	Finds one model instance and modifies it.	Modified model
`upsert(Model, id, doc, callback)`	`CanUpdate`	`./lib/methods/upsert.js`	Updates the model instance if found or creates a new model instance.	Updated or new model

Request lifecycle

If a request requires a login or if you want to intercept the request before or after it completes, you can implement the following methods:

Method Signature	Capability	Boiler Plate File	Description
`startRequest(methodName, args, request, next)`	-	-	Request interceptor invoked before the request is initiated and the login method is invoked. The method is passed the name of the method that started the request and the arguments passed to the method. Invoke the next function when the operation completes.
`loginRequired(request, callback)`	`AuthenticatesThroughConnector`	`./lib/lifecycle/loginRequired.js`	Determines if the request required a login. Pass an error message (or null if successful) as the first parameter to callback and a boolean value indicating if the login method needs to be executed (true) or not (false) as the second parameter.
`login(request, next)`	`AuthenticatesThroughConnector`	`./lib/lifecycle/login.js`	Logs into the data source to make a request. Invoke the next function when the operation completes.
`endRequest(methodName, args, request, next)`	-	-	Request interceptor invoked after the request completes . The method is passed the name of the method that started the request and the arguments passed to the method. Invoke the next function when the operation completes.

Connector configuration file

To have the Appcelerator CLI create a default configuration file for your connector when it is installed, create a sample configuration file that exports a JSON object in the `conf` directory, then reference the file in the connector logic with the `defaultConfig` property in the implementation object passed to the `Arrow.Connector.extend()` method.

For example, create a file called `example.config.js` in the `conf` directory and add the following content to it:

conf/example.config.js

```
module.exports = {
    connectors: {
        'connector.name': {
            setting1: 'foo',
            setting2: 'bar',
            setting3: 'baz'
        }
    }
};
```

Then reference the file in the connector logic using the `defaultConfig` property:

lib/index.js

```
exports.create = function(Arrow) {
    var Connector = Arrow.Connector;
    return Connector.extend({
        defaultConfig: require('fs').readFileSync(__dirname +
'/../conf/example.config.js', 'utf8'),
        ...
    });
}
```

When the Appcelerator CLI installs the connector, it will copy and rename the file to the project's `conf`.

Models and API endpoints

To allow an application to interact with the connector, you need to create a model definition file to define the schema of the data from the connector for the project to access. For more details about creating a model, see Models.

By default, when you install a connector, it will add its API endpoints to the application. If you do not want to generate these API endpoints, set the `modelAutogen` key to `false` in the connector's configuration file in the project.

conf/myconnector.default.js

```
module.exports = {
    connectors: {
        'connector.name': {
            setting1: 'foo',
            setting2: 'bar',
            setting3: 'baz',
            modelAutogen: false
        }
    }
};
```

You may specify specific models to generate from a connector. Set the `generateModels` key to an array of model names you want to include.

```
module.exports = {
    connectors: {
        'connector.name': {
            generateModels: [
                'foo',
                'bar',
                'baz'
            ]
        }
    }
}
```

Declare dependencies

The application can import any third-party modules that are supported by standard Node.js applications. Before publishing the app to the cloud, make sure all dependencies are listed in the `dependencies` field in the application's `package.json` file. For example, to add support for MongoDB 1.2.0 or greater:

package.json

```
{
    "dependencies":{ "mongodb": ">1.2.0" }
}
```