API Builder Project

Overview

An API Builder application is a standard Node.js application that runs in the AMPLIFY Runtime Services environment. This guide covers how to manage your project.

You may also use Appcelerator Studio to manage your API Builder (formerly known as Arrow Builder) projects. For details, see API Builder Development.

Project structure

A 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 project:

File/Folder Name	Description
`apis`	Contains API JavaScript files, used to create custom entry points for the application. For details, see API Builder APIs .
`app.js`	The entry point to the application if it is used as a server. You can monitor the startup and shutdown sequence.
`appc.json`	Contains component dependencies and AMPLIFY Runtime Services deployment settings. For details, see API Runtime Configuration.
`blocks`	Contains Block JavaScript files, used to create pre- and post-processing filters. For details, see API Builder Blocks.
`codeblocks`	Contains Codeblock, JSON, and Javascript files, used for defining custom functions for use in Flows.
`conf`	Contains configuration files in JSON format for the project and required connectors. For details, see Console Configuration.
`docs`	Contains generated docs for your project's APIs.
`endpoints`	Contains Endpoint JSON files, these are OpenAPI 2.0 (Swagger) documents used to create custom entry points for the application, with execution logic defined by linked Flows
`flows`	Contains Flow JSON files, used for defining business logic for Endpoints.
`index.js`	The entry point to the application if it is used as a module.

`logs`	Contains generated log files when running your project locally.
`models`	Contains Model JavaScript files, used to define the schema for your data. For details, see Models.
`node_modules`	Contains project dependencies. API Builder automatically installs any project dependencies declared in the `package.json` file.
`nodes`	Contains custom flow-nodes for use in Flows. Flow-nodes must be Node.js packages in their own folders named `nodehandler-*`.
`package.json`	NPM configuration file to declare project dependencies and other build or runtime configurations. For details, see NodeJS Configuration.
`package-lock.json`	NPM configuration file that is generated when NPM modifies either the `node_modules` tree or the `package.json` file. It describes the exact tree that was generated so that subsequent installs are able to generate identical trees, regardless of intermediate dependency updates. The `npm shrinkwrap` command repurposes the `package-lock.json` file into a publishable `npm-shrinkwrap.json` file. For additional information on shrink wrapping a project, refer to `npm shrinkwrap` .
`serviceconnectors`	Contains installed service connectors used within Flows for connecting to and interacting with external services.
`web`	Contains Web files, used to create endpoints that render UI.
`web/public`	Contains static assets, such as CSS, HTML, image, or JavaScript files, for your Web interface.
`web/routes`	Contains Route JavaScript files, used to define the API endpoint and logic for your Web interface.
`web/views`	Contains template files for your Web interface. Files must have one of the following extensions: `ejs`, `hbs`, `md`, or `jsx`.

Initializer file

The `app.js` file contains code that initializes the server instance. You can hook into the lifecycle events of the server as well as make additional setup or middleware calls to the server or Express app instance.

For example, the following `server.app.use` call forces the client to use a secure connection by redirecting any unsecured connections to the HTTPS URL.

app.js

```
var Arrow = require('arrow');
var server = new Arrow();

server.on('starting', function () {
    server.logger.debug('server is starting!');

    // server.app reference the Express app instance
    server.app.use(function (req, res, next) {
        if (!req.secure) {
            return res.redirect('https://' + req.get('Host') + req.originalUrl);
        } else {
            next();
        }
    });
});
// start the server
server.start();
```

Deploy the application

To deploy an application to AMPLIFY Runtime Services, run the `appc publish` command from the AMPLIFY Runtime Services project directory. Your project source code will be uploaded to the AMPLIFY Runtime Services where `npm install` will be executed against the project, which downloads and installs any NPM module dependencies. When the installation completes, the application is executed using `node`.

If the application is already deployed, you need to either increment the `version` field in the `package.json` file to publish a new version of the application or pass the `--force` flag to the publish command to republish the application. Before republishing the application, AMPLIFY Runtime Services sends the `SIGTERM` signal to the currently deployed application to let it shutdown gracefully. If the app does not shut down in time, AMPLIFY Runtime Services will kill the application.

Log output

AMPLIFY Runtime Services can capture two kinds of log output from applications:

- access logs – HTTP requests to the application
- application logs – explicit log calls made in the application

To capture access logs, use the `appc-logger` module, and to capture application logs, you can either use the standard JavaScript `console.log()` and `console.error()` methods, or you can use the `appc-logger` module.

By default, when an application is initialized, it loads and creates an `appc-logger` instance. The `appc-logger` instance is bound to the server instance, which will automatically capture access logs for the application.

To make log calls, use the `logger` property from either the API Builder instance, request object or response object to invoke `appc-logger` APIs.

```
var Arrow = require('arrow');
var TestAPI = Arrow.API.extend({
    group: 'ping',
    path: '/healthCheck',
    method: 'GET',
    description: 'Health-check endpoint',
    action: function (req, res, next) {
        req.logger.info('ping');
        res.status(200).send('OK');
        res.logger.info('pong');
        next();
    }
});

module.exports = TestAPI;
```

View log files

An application typically runs in the cloud, so being able to see what is happening in the application is very important. Any log output written to `stdout` or `stderr` in the application's root process is captured and stored by AMPLIFY Runtime Services, and can be viewed using the Appcelerator command-line tool or in the **Logs** tab of the Dashboard.

Notes:

- Only output written by the application's root process is included in the log file; output written by child processes forked by the application's root process will not be caught.
- Errors such as syntax errors, application crashes, and system level failures are logged automatically.

Logging utilities

The Appcelerator CLI provides three commands for viewing logs for a published application: **accesslog**, **logcat**, and **loglist**.

- The `appc cloud accesslog` command lists all requests processed by the Appcelerator Cloud in a specified time period. By default, a maximum of 100 log messages is returned at a time.
- The `appc cloud loglist` command lists your published application's log for a specific period. By default, a maximum of 100 log messages is returned at a time.
- The `appc cloud logcat` command lists your published application's log continuously from Appcelerator Cloud.

About logged execution times

The execution time reported by the `loglist` and `accesslog` commands report slightly different values. The `accesslog` command reports the time required by Appcelerator Cloud to handle the initial user request, pass it to your Node.js application for processing, and deliver the response. In contrast, `loglist` only reports the execution time for your application itself, not including the time required to process the request and response. Consequently, the `accesslog` execution times are a slightly longer than those of the corresponding `loglist` log item. For instance, below is `accesslog` output:

```
[12/11/2013 08:43:55.790]   127.0.0.1   /   10074ms
[12/11/2013 08:43:23.712]   127.0.0.1   /   10073ms
[12/11/2013 08:43:07.237]   127.0.0.1   /   10073ms
[12/11/2013 08:42:48.365]   127.0.0.1   /   10075ms
```

And below is the corresponding `loglist` output.

```
12/11/2013 16:42:028.139 [INFO] [43210] App started
[PERF]  GET / 10069 ms
[PERF]  GET / 10072 ms
[PERF]  GET / 10072 ms
[PERF]  GET / 10066 ms
```

Node.js version

Prior to AMPLIFY Runtime Services (formerly known as Arrow Cloud) 1.2.0, the only Node.js versions you could use were 0.8.26 and 0.10.22.

Starting with AMPLIFY Runtime Services 1.2.0, you may specify any version of Node.js. Node.js 0.8.26, 0.10.22 and 0.12.4 are built in, but other versions will be downloaded from https://nodejs.org/ when the application is built prior to running `npm install`.

To specify a Node.js version, in the `package.json` file, set the `engines.node` key to the version of Node.js you want to use. DO NOT SPECIFY A RANGE. If you do not specify a Node.js version, the application will use 4.4.7 by default (as of SDK 6.0.0).

package.json

```
{
  "engines": {
    "node": "0.12.4"
  }
}
```

Port binding

Starting with AMPLIFY Runtime Services 1.2.0, you may explicitly set the port the application listens on. Set the `cloud.environment.PORT` key in the `appc.json` file. or use the `appc cloud config --set "PORT=<PORT_NUMBER>"` command to set the special environment variable PORT. If you do not set PORT explicitly before publishing your application, AMPLIFY Runtime Services sets it to 80 by default.

appc.json

```
{
  cloud: {
    environment: {
      PORT: 8080
    }
  }
}
```

Verify that the application listens on PORT, otherwise your app cannot be connected. Use `process.env.PORT` in the application to verify the application is connected to a port.

Install custom binaries

AMPLIFY Runtime Services allows you to install additional binaries before your application is built.

Starting with AMPLIFY Runtime Services 1.3.0, to install additional third-party tools, create a script called `install.sh` in the project's root folder, which installs the required packages.

ImageMagick and PhantomJS
Both ImageMagick and PhantomJS are pre-installed on the containers.

Below is a sample script located in the `./install.sh` folder in the application's directory.

install.sh

```
#!/bin/bash

echo "---"
echo "Start installing some tools..."
echo "(This bash script is run at npm preinstall.)"
echo "---"

apt-get install -qq tar bzip2 libaio1

apt-get install -y wget

echo "---"
echo "done installing additional tools!"
echo "---"
```

Prior to Release 1.3.0, you needed to create a script in your project folder (no name restrictions) and add the script to the `package.json` file. In the `package.json` file, set the `scripts.preinstall` or `scripts.postinstall` field to the path to the script:

package.json

```
"scripts": {
    "preinstall": "myscript.sh"
  }
```

Note that from AMPLIFY Runtime Services 1.3.0 and later, you can still use the above method but do not name the script `install.sh` or it will run twice, and only install the binaries in the project directory. Prior to AMPLIFY Runtime Services 1.3.0, binaries could be installed outside the project directory.

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" }
}
```

Define environment variables

To set environment variables, add them to the `cloud.environment` object in the project's `appc.json` file.

appc.json

```
{
  cloud: {
    environment: {
      foo: 'abc'
    }
  }
}
```

You can also use the Appcelerator CLI to manage the environment variables.

To set environment variables, use the `appc cloud config --set <key>=<value>` command. To set more than one variable at a time, comma-separate the key-value pairs.

You can access the environment variables from the application using the `process.env` namespace. For example, if you set a variable called `fo o`, use `process.env.foo` to access it in the application.

To unset an environment variable, use the `appc cloud config --unset <key>` command.

To check the current environment variables, use the `appc cloud config --env` command.

Blacklist Variable Names
Prior to AMPLIFY Runtime Services 1.2.0, you could not use the following names for environment variables: "appid", "basedir", "bodyParser", "customConfig", "dirname", "framework", "fullpath", "name", "serverId", "port", "version", "NODE", "NODE_PATH", "PATH", "PWD", "PORT", "TMPDIR", and "USER".

After changing an environment variable, you will be prompted to restart the application.

The following example sets the `foo` environment variables, lists all set environment variables, then unsets the `foo` variable:

```
appc cloud config --set foo=abc
appc cloud config --env
appc cloud config --unset foo
```

Scale the application

To customize the number of cloud servers the application can enable the auto-scaling feature to automatically scale up and down the number of cloud servers based on the number of queued requests. You specify the maximum number of queued requests that should occur before the application is scaled up. AMPLIFY Runtime Services will increase the number of containers if the queue is too high for at least one minute. You can also specify the minimum and the maximum number of servers that should be used.

The following example enables autoscaling, using a maximum of five servers when there are at least 20 queued requests. The application is also configured to automatically scale down the number of servers when the number of queued requests drops below 20.

appc.json

```
{
  cloud: {
    maximum: 5,
    minimum: 1,
    maxqueuedrequests: 20
  }
}
```

You can also use the `appc cloud config` command to configure autoscaling:

```
appc cloud config --maxsize 5 MyFirstApp
appc cloud config --maxqueuedrequests 20 MyFirstApp
appc cloud config --autoscaleup true MyFirstApp
appc cloud config --autoscaledown true MyFirstApp
```

Access a specific container

Starting with AMPLIFY Runtime Services 1.3.1, if you have scaled your application to use more than one server container, you can make a request to a specific container that your application is running on. To make a request to a specific server container, pass the `_serverid` paramet er with the request and set it to the ID of the server container. To retrieve the server container ID, run the `appc cloud accesslog --show_serverid` command.

Example:

```
https://<APP_GUID>.cloudapp-enterprise.appcelerator.com?_serverid=<SERVER_ID>
```

Custom domains

Set a custom domain and path

You can bind a custom domain to your application that points to either a CNAME record or A record as well as assign it a path. When setting the domain, do not specify the protocol, that is, do not prefix the URL with `http://` or `https://`.

The alias to set should be a valid domain name that has been already configured with either a CNAME or A record pointing to the published application's URL. Appcelerator Cloud validates the domain record before binding it to the application.

To set a custom domain, set the `cloud.domain` field in the `appc.json` file. You can optionally set the `cloud.domainPath` field to assign a path to the application. The following example sets a domain and path on the application that can be accessed from `www.foo.com/v2`:

appc.json

```
{
  cloud: {
    domain: 'www.foo.com',
    domainPath: 'v2'
  }
}
```

You can also use the Appcelerator CLI to manage the domain and path.

To bind a domain to an application, use the `appc cloud config --set <domain_name>` command to bind a domain to the application. You can bind multiple domains to an application. Use the `--set <domain_name>` parameter to bind up to five additional domains to the application.

If you need to remove a domain, use the `--remove` parameter. For applications with multiple domains, you will be prompted to select which domain to remove. You may optionally pass the domain name to remove with the `--remove` parameter.

To route, an application based on a path with the domain name, use the `appc cloud config --path <path_name>` command to set a path for the application after setting a domain. For example, if you want to bind two applications to the same domain, specify a path for each to route a client to the correct application.

The following example allows the application to be accessed from `www.foo.com` :

```
appc cloud domain --set www.foo.com
```

The following example allows the LegacyApp application to be accessed from `www.foo.com/v1` and the BrandNewApp application to be accessed from `www.foo.com/v2` :

```
appc cloud domain --set www.foo.com LegacyApp
appc cloud domain --path v1 LegacyApp
appc cloud domain --set www.foo.com BrandNewApp
appc cloud domain --path v2 BrandNewApp
```

Wildcard subdomains

The pre-defined `cloudapp.appcelerator.com` URL that Appcelerator Cloud uses to publish your application supports a wildcard subdomain. You may append a token to the beginning of the URL that can be parsed by the application. This does not apply to custom domain names using the `acs domain` command to bind the application to a CNAME or A record.

For example, if your published URL is `https://1234567890.cloudapp.appcelerator.com/`, you can navigate to your application using the same domain and add a custom token as the subdomain, for example, `https://deadbeef.1234567890.cloudapp.appcelerator.com/`, where `deadbeef` is the wildcard subdomain. Then, the application can retrieve the host and subdomain:

```
var http = require('http'),
    url = require('url');
http.createServer(function (req, res) {
    var host = req.headers['host'],
        subdomain = host && url.parse('http://'+host).hostname.split('.')[0];
    res.writeHead(200, {'Content-Type': 'text/plain'});
    res.end('Your host is: ' + host + '. Subdomain is: ' + subdomain);
}).listen(process.env.PORT || 8080);
```

Add a custom SSL certificate

To use a custom SSL certificate to access your application using HTTPS, you need to create a PEM file, then add the PEM file to the application.

To create a PEM file, you will need the following three files provided by your SSL certificate provider:

- Certificate file (`customapp.com.crt`, for example)
- An intermediate certificate authority (`gd_bundle.crt`, for example)
- Key used to generate the certificate (`customapp.com.key`, for example)

You need to combine the contents of the three files into a single text file, called a PEM file, which you will add to your application. The PEM file must have the following structure:

```
-----BEGIN CERTIFICATE-----
<SSL certificate file contents>
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
<Intermediate certificate file contents>
-----END CERTIFICATE-----
-----BEGIN RSA PRIVATE KEY-----
<Private key file contents>
-----END RSA PRIVATE KEY----
```

Use a text editor or the `cat` command to merge the files together:

```
cat customapp.com.crt gd_bundle.crt customapp.com.key > customapp.com.pem
```

Once you have created the PEM file, set the `cloud.certificate` field to the path of the certificate file.

appc.json

```
{
  cloud: {
    certificate: 'customapp.com.pem'
  }
}
```

You can also add it to your application by executing the following command:

```
appc cloud crt --add customapp.com.pem
```

Create child processes

A single instance of Node.js runs in a single thread. To take advantage of multi-core systems, you can launch a cluster of Node.js processes to handle the load. Applications can use the core Node Cluster module to easily create child processes that all share server ports.

To use this feature, your application uses cluster.setupMaster() to set a path to a custom file to use for each child process. A cluster should listen on port 9000 or greater to avoid port conflicts. If the port your application is trying to listen on is in use, an `EADDRINUSE` error will result. Your application must also have privileges to listen on the specified port, otherwise, an `EACCES` error will result.

Example:

app.js

```
var cluster = require('cluster');
var http = require('http');
var numCPUs = require('os').cpus().length;

cluster.setupMaster({exec: __dirname + '/child.js'});

// Fork workers
for (var i = 0; i < numCPUs; i++) {
    cluster.fork();
}
```

child.js

```
var http = require('http');

console.log('Running in child process of a cluster.')

http.createServer(function(req, res) {
    res.writeHead(200);
    res.end("hello world!!!\n");
}).listen(9000);
```

Application limitations

Disk space

Each application can use 1.8 GB of disk space. The application can only write files to the project's root directory and to the `/tmp` folder.

Server containers

Each application runs in a specific container size with different resources (memory and number of CPUs). By default, when the application is published, it will run in one Medium container.

- To specify a bigger container for the application, set the `cloud.container` field in the `appc.json` file or use the `appc cloud server` command.
- To use more than one container for your application, see Scale the Application.

You can specify one of the following container sizes depending on your AMPLIFY Appcelerator Services subscription:

Name	Point Cost	Memory	CPU Shares	Archive Behavior
Dev	1	512MB	1000	After an hour of inactivity
Small	2	256MB	1000	After a week of inactivity
Medium	4	512MB	2000	After a week of inactivity
Large	8	1024MB	4000	After a week of inactivity
XLarge	16	2048MB	8000	After a week of inactivity

It is important to choose Dev (512MB) and Small (256MB) containers wisely. The minimum recommended container sizeis "Medium". Though you may be able to deploy to a "Dev" or "Small" container, for better memory usage and performance it is highly recommended that you use medium or bigger size containers.

If your application is archived, to reactivate the application, make a request to it. The first request will be slow, but subsequent requests will respond with normal speed.

Note that each container your application runs on costs a certain number of points. To see how many points you have used and are allocated, or to see which containers your application is using, execute the `appc cloud list` command.

```
$ appc cloud list MyArrowApp
ACS: Appcelerator Cloud Services Command-Line Interface, version 1.0.23
Copyright (c) 2012-2015, Appcelerator, Inc.  All Rights Reserved.
Admin Hostname: https://admin.cloudapp-enterprise-preprod.appctest.com

============
Points:
 -- Quota: 17
 -- Used: 16

App name: MyArrowApp
 -- Created by: joeuser@appcelerator.com
 -- URL: https://myapp.cloudapp-enterprise-preprod.appctest.com
 -- Created at: Mon Mar 23 2015 21:58:36 GMT-0700 (PDT)
 -- Node Version: 0.10.22
 -- Server Size: XLarge
 -- Maximum allowed number of servers: 1
 -- Desired minimum number of servers: 1
 -- Active version: 1.0.0
 -- Published at: Thu Mar 26 2015 13:54:07 GMT-0700 (PDT)
 -- Status: Deployed
```

Server Ports

Currently, AMPLIFY Runtime Services only supports applications opening one server listening port. There cannot be more than one TCP/HTTP server started in one application.