API Builder Web

API Builder 3.x is deprecated
Support for API Builder 3.x will cease 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
- Route definition
- Renderer engines
    - Pre-built renderer engines
    - Custom renderer engines
    - Handlebars partials and helpers
        - Helpers
        - Partials
- API Builder APIs from API Builder Web
- Interacting with models

Introduction

This guide covers the basics for creating API Builder Web interfaces. An API Builder Web interface is a custom endpoint that renders HTML content to a client application. An API Builder Web interface is made up of:

- assets (images, CSS, HTML and JavaScript files) located in the `web/public` folder
- templates (EJS, Handelbars, Markdown or ReactJS) located in the `web/views` folder
- API Builder Routes (endpoint definitions) located in the `web/routes` folder

You can create custom logic in your API Builder Routes, which can internally access your service's models and APIs.

Route definition

Place all API Builder route definition files in the service's `web/routes` folder. You can only declare one endpoint definition per file. An API Builder route definition file is a JavaScript file, which:

1. Loads the `arrow` module
2. Calls the module's `Router.extend()` method, passing in an object defining the API endpoint and logic
3. Exports the defined endpoint using the `module.exports` variable

Set the following keys in the object passed to the `Router.extend()` method to define the API endpoint.

Name	Required	Description
`name`	true	Name of the route.
`path`	true	Endpoint/path for the route.
`enabled`	false	Specifies whether the route is enabled. If not, it won't be registered, and won't accept requests.
`sort`	false	An integer that determines the order routes are registered. Routes with a higher sort value are prioritized and registered earlier. For example, say you have `/route/:id` and `/route/foo`. If the route with the wildcard has a higher sort than the static route, the static route runs. So create the first with a lower sort, and `/route/foo` ro utes properly, as does `/api/bar`.
`method`	true	HTTP method (`GET`, `POST`, `PUT`, `DELETE`).
`description`	true	Description of the route.
`action`	true	The function that allows you to interact with API Builder APIs and models and send data to your template engine.

Renderer engines

A renderer engine renders data (or locals in the Express framework) to the view (template file). API Builder provides a few renderer engines and allows you to add your own custom renderer engines.

Pre-built renderer engines

API Builder Web supports the EJS, Handlebars, Markdown, and ReactJS renderer engines. Place all template files with the appropriate extension in the `web/templates` folder.

Renderer Engine	File Extension
EJS	`.ejs`
Handlebars	`.hbs`
Markdown	`.md`
ReactJS	`.jsx`

To use a template in the API Builder Route's logic, reference its filename without the extension. Because the template is referenced using the filename, you cannot have the same filename with multiple extensions.

Custom renderer engines

To create a custom renderer engine you need to create a renderer engine and register it with the API Builder instance's middleware instance.

1. Create an object that implements the `createRenderer()` method and specifies the `extension` property.
2. Pass the object to the Middleware instance's `registerRendererEngine()` method. You can retrieve a Middleware instance by using the `middleware` property of the API Builder instance.

For example, to implement a renderer engine for Jade templates:

```
var jade = require('jade'),
    engine = {};
engine.jade = jade;
engine.createRenderer = function (content, filename, app) {
    return function(filename, opts, callback) {
        if (!content) {
            content = require('fs').readFileSync(filename, 'utf8').toString();
        }
        callback(null, jade.render(content, opts));
    }
};
engine.extension = 'jade';
// server is an Arrow instance
server.middleware.registerRendererEngine(engine);
```

Any view with a `jade` extension will be routed to the Jade renderer engine.

Handlebars partials and helpers

API Builder exposes some APIs to allow you to register Handlebar partials or helpers.

Helpers

Helpers are functions that you can evaluate in your Handlebar templates. To use a helper, register the helper with the Handlebar renderer engine, then in the template, call the helper using the name you gave the helper when registering it.

1. Get a reference to the Handlebar renderer engine using the `Arrow.Middleware.getRendererEngine('hbs')` method.
2. Call either the Handlebar renderer engine's `registerHelper()` to register a helper function. Pass the method the name of the helper and the function to invoke.

```
var Arrow = require('arrow'),
    hbs = Arrow.Middleware.getRendererEngine('hbs');

hbs.registerHelper('doFoo', function(foo) {
    // this.name references the name parameter passed to the template
    // in the render call, that is, res.render('template', {name: 'Joe'});
    if (foo) {
        return this.name + ' is great!';
    } else {
        return this.name + ' is ok.';
    }
});
```

Template example:

```
<div>doFoo(true)</div>
<div>doFoo(false)</div>
```

Partials

Partials are subviews that you can embed in a template. To use a partial, you need to register it with the Handlebar renderer engine, then reference the partial in the template using the `{{> partialName}}` syntax, where `partialName` is the name you gave the partial when you registered it.

1. Get a reference to the Handlebar renderer engine using the `Arrow.Middleware.getRendererEngine('hbs')` method.
2. Call either the Handlebar renderer engine's `registerPartial()` to register a partial file. Pass the method the name of the partial and the template file to use as a partial.

```
var Arrow = require('arrow'),
    hbs = Arrow.Middleware.getRendererEngine('hbs');
hbs.registerPartial('fooView', 'web/views/foo.hbs');
```

Template example:

```
<!-- Partial web/views/foo.hbs -->
<!-- id and name are passed as data to the res.render() method -->
<a href="/people/{{id}}">{{name}}</a>

<!-- Main Template web/views/main.hbs -->
<ul>{{#people}}<li>{{> fooView}}</li>{{/people}}</ul>
```

API Builder APIs from API Builder Web

You can interact with API Builder APIs from your API Builder Web route. The following is an example.

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

var TestRoute = Arrow.Router.extend({
    name: 'car',
    path: '/car',
    method: 'GET',
    description: 'get some cars',
    action: function (req, resp, next) {

        req.server.getAPI('api/car', 'GET').execute({}, function(err, results) {
            if (err) {
                next(err);
            } else {
                req.log.info('got cars ' + JSON.stringify(results));
                resp.render('car', results);
            }
        });
    }
});

module.exports = TestRoute;
```

In the preceding example, the route calls the `car` API. You can retrieve a reference to an API by specifying its path or `nickname` property when specified by the model/API that you are using. For example:

```
req.server.getAPI('api/car');
```

This code returns a reference to the `car` API. Once you have the API, you need to call `execute`:

```
req.server.getAPI('api/car').execute({}, function(err, results){

});
```

This first argument to `execute` is the input required by your API. In this example, none are required since `findAll` is being called. The second argument is a callback function. The first argument in the callback function is an `error` object. The second is data returned from the API call.

The final part of the example is calling the template with the response data from the API call.

```
resp.render('car',results);
```

In this example, `car` references the name of a handlebars template file (`car.hbs`) and `results` contains the API response with the array of cars.

Following is the handlebars template for this example. It iterates through the `cars` array.

```
<html>
<head>
</head>
<body>
    {{#each cars}}
        <div>{{make}} {{model}} {{year}}</div>
    {{/each}}
</body>
</html>
```

Interacting with models

The preceding example shows how to access APIs from a route. You can also directly access models. The following modifies the preceding example to use the `car` model.

```
ar Arrow = require('arrow');

var TestRoute = Arrow.Router.extend({
    name: 'car',
    path: '/car',
    method: 'GET',
    description: 'get some cars',
    action: function (req, resp, next) {
        var model = req.server.getModel('car');
        model.findAll(function(err, results){
            if (err) {
                next(err);
            } else {
                req.log.info('got cars ' + JSON.stringify(results));
                resp.render('car', {cars:results});
            }

        });
    }
});

module.exports = TestRoute;
```

The first line of the `action` function retrieves the `car` model by name:

```
var model = req.server.getModel('car');
```

The next line calls the `findAll` function of the model. It's important to note that calling APIs is different than calling models. Calling an API programmatically on the server is nearly identical to calling it remotely - you supply some input parameters and call execute and it returns the API response. Calling a model programmatically is slightly different. Since it's a model, it does not have a REST interface. Instead, it has the functions that are called underneath the covers when an API is called, so a `GET` call to an API is the same as a `findAll` call on the model. The other difference is in the response data. The model only returns the data results - hence the results are placed in an object property called `car`, so my UI template can render it properly.

As you can see, API Builder Web makes it easy to build responsive desktop and web apps (using your template engine of choice) that seamlessly integrate with API Builder APIs and models.