API Builder Models

Introduction

This guide covers the basics for creating Models. Models are data stored in either server memory or a backend service, such as Mobile Backend Services or a MySQL database, using an API Builder Connector. Models are accessed like standard REST objects using predefined endpoints that API Builder automatically generates by default. You can either create a model by defining your own schema, use an existing model defined by a connector, modify an existing model by either extending or reducing it, or create a composite model by joining two or more models together.

To programmatically create Models, see the API Builder.Model API reference.

Model definition

Place all Model files in the `models` folder. You can only declare one model per file. A Model file is a JavaScript file, which:

1. Loads the `arrow` module
2. Calls the module's `createModel('name', schema)` method (or another `Model` method), passing in the name of the model as the first parameter and an object defining the model schema as the second parameter
3. Exports the defined endpoint using the `module.exports` variable

Set the following keys in the object passed to the `createModel()` method to define the model:

Name	Required	Description
`fields`	true	An object that represents the model's schema defined as key-value pairs. The key is the name of the field and the value is the `fields` object. See the next table for details.
`connector`	true	Connector to which the model is bound (string). Each model can only have **one** connector. Connectors are responsible for reading and writing data from/to their data source.
`documented`	false	**Since Release 5.0.0.** Determines whether to generate API documentation (`true`) or not (`false`). The default value is `true`.
`metadata`	false	Used to provide connector specific configuration (for example, mapping the model to a specific database table for the MySQL connector or defining the join properties).
`autogen`	false	Used to determine whether to generate API endpoints directly from the model. The default value is `true`. If the endpoint is auto-generated, you do not need to create an API endpoint definition.
`actions`	false	An array of data operations supported by the model. The valid values are: `create`, `read`, `update`, and `delete`. By default, all are supported by the model.

`plural`	false	A string used as the property name when your API endpoint returns an array. By default, the plural value is the plural of the model name. For example, if your model is named **car**, the default plural would be **cars**. ⚠ This value can be set on an API or a model.
`singular`	false	A string used as the property name when your API endpoint returns a single record. By default, the singular value is the name of the model. ⚠ This value can be set on an API or a model.
`before`	false	One or more blocks to be executed before the request. Blocks are referenced by their `name` property. If you want to execute multiple blocks, you should specify them as an array of block names. If multiple blocks are specified, they are executed in the order specified.
`after`	false	One or more blocks to be executed after the request. Blocks are referenced by their `name` property. If you want to execute multiple blocks, you should specify them as an array of block names. If multiple blocks are specified, they are executed in the order specified.

Field definition

The property`fields` (mentioned above) supports a number of sub-properties as well. The table below outlines these properties.

Name	Required	Description
`type`	true	The field primitive type plus others (for example, string, number, boolean, object, array, date). Type can be any valid JavaScript primitive type. Type can be specified as a string (for example, string) or by the type class (for example, String).
`required`	false	Specifies whether the field is required. The default value is `false`.
`validator`	false	A function or regular expression that validates the value of the field. The function is passed the data to validate and should return either `null` or `undefined` if the validation succeeds. Any other return value means the validation failed, and the return value will be used in the exception message. If a regular expression is used, it should evaluate to either true or false.
`name`	false	Used if the model field name is different than the field name in the connector's model or the underlying data source for the field name. For example, if my model field is `first_name` and the column in a MySQL database is `fname`, the value of the `name` property should be `fname`.
`default`	false	The default value for the field.
`description`	false	The description of the field (used for API documentation).
`readonly`	false	Either `true` or `false`. If `true` the field will be read-only and any attempt to write the field value will fail.
`maxlength`	false	The max length of the field (specified as an integer)
`get`	false	A function used to set the value of a property that will be sent to the client. This property is useful if you want to define a custom field where the value is derived.
`set`	false	A function used to set the value of a property that will be sent to the connector.
custom	false	This property should be specified and set to `true` if you are defining a custom field. A custom field is one that does not exist in the underlying data source for the connector you specified.
model	false	Model name of the field property. This is either the logical name of a custom model or a connector model name in the form **connector/model_name** (e.g., appc.mysql/employee)

Model schema example

The example below creates the `car` model with the specified schema. The car models will be stored in Mobile Backend Services as CustomObjects. Since the `autogen` property was not set to `false`, API Builder automatically generates the pre-defined endpoints for the client to access the car models using the `<SEVER_ADDRESS>/ api/car` endpoints.

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

var car = Arrow.createModel('car', {
    fields: {
        make:{type:String, description:'the make of a car '},
        model:{type:String, description:'the model of the car', required:true},
        year: {type:Number, description:'year the car was made', required:true},
        bluebook: {type:Number, description:'kelly bluebook value of the car',
required:true},
        mileage: {type:Number, description:'current mileage of the car',
required:true}
    },
    connector: 'appc.arrowdb'
});

module.exports = car;
```

Modify an existing model

Besides creating a fully defined model, you can modify an existing model either by reducing or extending it.

Reduce a model

A reduced model is an existing model where you only use specific fields from it. To create a reduced model, follow the same procedure when creating a regular model, except invoke the module's `Model.reduce()` method instead of the `createModel()` method. Pass the model you want to reduce as the first parameter, the name of the new model as the second parameter, and the new model schema as the last parameter.

Example

The Model file below extracts three fields from the `employee` table of the `appc.mysql` connector, indicated by the `appc.mysql/employee` par ameter, and renames the fields for the `baseEmp` model, for example, `email_address` in the MySQL employee table maps to `email` in the new model.

models/baseemp.js

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

var baseEmp = Arrow.Model.reduce('appc.mysql/employee','baseEmp',{
    fields: {
        fname: {type:String, description:'First name', required:true,
name:'first_name'},
        lname: {type:String, description:'Last name', required:true,
name:'last_name'},
        email: {type:String, description:'Email address', required:true,
name:'email_address'}
    }
});

module.exports = baseEmp;
```

Extend a model

An extended model is an existing model where you modify the fields or add more fields. To create an extended model, follow the same procedure when creating a regular model, except invoke the module's `Model.extend()` method instead of the `createModel()` method. Pass the model you want to extend as the first parameter, the name of the new model as the second parameter, and the new model schema as the last

parameter.

Example

The Model below extends the `employee` model by adding the `headquarters` field to it.

models/fullemp.js

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

var fullEmp = Arrow.Model.extend('employee','fullEmp',{
    fields: {
        headquarters:{type:Boolean, custom:true,
            get:function(val,key,model){
                return model.get('state') === 'CA';
            }
        }
    }
});

module.exports = fullEmp;
```

Create a composite model

Composite models allow you to create a single model that is composed of one or more models based on the same or different connectors. Composite models can be joined together via a common set of properties, such as primary keys or foreign keys, or they can have no properties in common at all. The power of composite models is that you can represent multiple data sources and entities as a single API endpoint, which is ideal for many mobile use cases.

To create a composite model, follow the same procedure when creating a regular model except the `connector` property must be set to `appc.composite`, each field in the definition object must specify the `model` property to indicate which model the field originates from, and the `metadata` property must define the join operation to combine the models or leave it undefined to perform no join operations.

The following terms are used to refer to models:

- Model definition: The composite model which is being created
- Main model: The main source of data for the composite model. This is the left table in SQL terminology. It is implicitly defined.
- Secondary model: Any model other than the main model. This will be the right table in SQL terminology.

The composite connector can either perform a left join or inner join:

- left join: all records from the main model are returned regardless if it found a match in the secondary models
- inner join: only records that match both models are returned

The composite connector can also perform either a one-to-one join or one-to-many join:

- one-to-one: only one record from the secondary model matches a record in the primary model
- one-to-many: multiple records from the secondary model can match a record in the main model

There are different ways that a one-to-one join and a one-to-many join can work when merging (mapping) data from the main model into the primary model:

- Merge as object: This is a one-to-one relationship where the whole secondary model record will be mapped to a field in the main model.
- Merge as an array: This is a one-to-many relation where multiple records from the secondary model will be mapped to an array field in the model definition.
- Merge as the field: This is a field which comes directly from a joined model. The field in the model definition ***must*** have a `name` property which refers to the field being joined from the secondary model. By default, this is a one-to-one relationship where the field will contain a single match. In the Join-Object Definition, `multiple` may be set to `true` for all of the matches to be mapped to the field. Since this returns multiple values, the field type must be Array if `multiple` is set to `true`.

The composite connector can be used to perform reduce functionality on a single model. This only requires the main model and does not require any joins. The API Builder Console offers its functionality using this method. Without any joins, a one-to-one merge as a field is the only functionality available.

To define the join operation, set the `metadata` property to the `left_join` key or `inner_join` key, either of which takes an array of objects defining the join. Each object in the `left_join` or `inner_join` property defines the model to join (`model` property), the key to join (`join_prop`

erties property) and, optionally, if the join is a multiple property.

Join object definition

Key	Type	Value
model	String	Name of the model. For left joins, this is the secondary model you want to join with the main model.
join_properties	Object	Collection of key-value pairs that determine the keys in each model to perform the join operation. The key is the property of the model defined in this object and the value is the property to join in another model (or the main model for left joins).
multiple	Boolean	Determines whether the match is one-to-one (false) or one-to-many (true). The default value is false. If true, the field being joined on must be of type Array and have a name property referring to the field from the secondary model to be used.

Left join example

The example below combines the employee and managers models to create the employee_manager model. The models are joined based on a match between the managers model's employee_id and the employee model's auto-generated id.

models/employee_manager.js

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

var employee_manager = Arrow.createModel('employee_manager',{
    fields: {
        fname: {type:String, description:'First name', name:'fname',
model:'employee'},
        manager: {type:String, description:'manager of employee', name: 'manager',
model:'managers'}
    },
    connector: 'appc.composite',
    metadata: {
        left_join: {
            model: 'managers',
            join_properties: {
                employee_id: 'id'
            }
        }
    }
});

module.exports = employee_manager;
```

models/employee.js

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

var employee = Arrow.Model.reduce('appc.mysql/employee','employee',{
    fields: {
        fname: {type:String, description:'First name', name:'first_name'}
    },
    connector: 'appc.mysql'
});

module.exports = employee;
```

models/managers.js

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

var managers = Arrow.Model.reduce('appc.mysql/employee_manager','managers',{
    fields: {
        employee_id: { type: Number, description: 'Employee ID' },
        manager: {type:String, name:'manager_name', description:'manager name'}
    },
    connector: 'appc.mysql'
});

module.exports = managers;
```

Inner join example

The example below performs an inner join on the `employee`, `employee_manager` and `employee_habit` models. Both the `employee_manager` and `employee_habit` `employee_id` properties will try to match the employee `id` property. The description of every habit which matches the employee ID will be listed in the `habit` property.

Expand source

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

// create a model from a mysql table
var employee_composite = Arrow.createModel('employee_composite',{
    fields: {
        fname: {type: String, description: 'First name', name: 'fname', model:
'employee'},
        manager: {type: String, description: 'Manager of employee', name: 'manager',
model: 'employee_manager'},
        habits: {type: Array, description: 'Habits of employee', name: 'description',
model: 'employee_habit'}
    },
    connector: 'appc.composite',
    metadata: {
        inner_join: [
            {
                model: 'employee_manager',
                join_properties: {
                    employee_id: 'id'
                }
            },
            {
                model:'employee_habit',
                multiple: true,
                join_properties:{
                    employee_id:'id'
                }
            }
        ]
    }
});

module.exports = employee_composite;
```

Field name mappings

You often want the ability to use a field property name in your model that is different from its name in an existing model. The following example shows how you can use the `name` sub-property of a field to map a model property name to a specific property name of an existing custom model or connector generated model. For example, the **employee** model has a property called **first_name**, but the new model wants that property to be called **fname**. The API Builder framework ensures this mapping occurs bidirectionally.

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

var emp = Arrow.Model.reduce('appc.mysql/employee','emp',{
    fields: {
        fname: { type:String, description:'First name', name:'first_name',
required:true},
        lname: { type:String, description:'Last name', required:true,
name:'last_name'},
        email: { type:String, description:'Email address', readonly:true,
name:'email_address'}
    },
    connector: 'appc.mysql'
});

module.exports = emp;
```

Field input validation

You might need to perform validation on a field when creating or updating a record. Each property in your model definition can specify a validation function using the `validator` field property. This function is called before sending data to your model's connector. The `validator` function is passed the value of the property. If the value is valid, the function should return `null` or `undefined`. If not valid, the function should return a message indicating why the validation failed. The following is an example of a validator function on a field.

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

var emp = Arrow.Model.reduce('appc.mysql/employee','emp',{
    fields: {
        fname: {
            type:String, description:'First name', name:'first_name', required:true,
            validator:function(val) {
                if (val.length < 5) {
                    return 'First name must be greater than 5 characters'
                }
            }
        },
        lname: { type:String, description:'Last name', required:true,
name:'last_name'},
        email: { type:String, description:'Email address', readonly:true,
name:'email_address'}
    },
    connector: 'appc.mysql'
});

module.exports = emp;
```

Model input validation

You might need to perform validation on a whole model. Specify in your model definition a validation function using the `validator` model property. This function is called before sending data to your model's connector. The `validator` function is passed the instance of the model. If the value is valid, the function should return `null` or `undefined`. If not valid, the function should return a message indicating why the validation failed or throw an exception. The following is an example of a validator function on a model.

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

var emp = Arrow.Model.reduce('appc.mysql/employee','emp',{
    fields: {
        fame: { type:String },
        lname: { type:String }
    },
    validator: function (instance) {
        var errors = [];
        if (instance.get('fame') === "Rick") {
            errors.push('Sorry, Rick is not allowed to play here.');
        }
        if (instance.get('lname').length < 5) {
            errors.push('The lname must be at least 5 characters long.');
        }
        if (errors.length) {
            return errors.join('\n');
        }
    },
    connector: 'appc.mysql'
});

module.exports = emp;
```

Customizing generated model APIs

You can customize the generated APIs for your models. For example, by default, the create API only returns a status 201 with a header Location pointing to the newly created instance. No content is returned in the body. If you want to directly receive the newly created instance in the body of the request, add the `includeResponseBody: true` metadata to your model.

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

var emp = Arrow.Model.reduce('appc.mysql/employee','emp',{
    fields: {
        fname: {
            type:String, description:'First name', name:'first_name', required:true
        },
        lname: { type:String, description:'Last name', required:true,
name:'last_name'},
        email: { type:String, description:'Email address', readonly:true,
name:'email_address'}
    },
    connector: 'appc.mysql',
    metadata: {
        includeResponseBody: true
    }
});

module.exports = emp;
```

Programmatic CRUD interface

All models inherit the CRUD interfaces supported by their underlying connector. As a result, you can programmatically call these interfaces. The main use case for using a model's CRUD interface is when you want more control of an API's functionality. You can place logic in your API

endpoint's `action` function to handle custom business functionality and control execution of data access.

The following are the main interfaces most connectors support.

```
 // delete all records for a model
Model.deleteAll(callback);

// query a model.
Model.query(options, callback);

// find all records for a model
Model.findAll(callback);

// find a record by id for a model
Model.findByID(id, callback);

// delete a record for a model
Model.delete(instance, callback);

// update a record
Model.update(instance, callback);

// create a record
Model.create(object, callback);
```

The following model has example uses.

```
// example model
Model = Arrow.Model.extend(testTableName, {
    fields: {
        title: { type: String },
        content: { type: String }
    },
    connector: 'appc.mssql'
});
```

Delete all records

Use the `deleteAll` function on a model to delete all of its records.

```
Model.deleteAll(function(err) {
    if (err) {
        return next(err);
    }
    next();
});
```

Create, update, delete a record

The following is an example of creating a record and then updating and deleting it. It's not necessarily a practical example but demonstrates how to use some additional interfaces available on a model.

Expand source

```
// setup record object
var title = 'Test',
    content = 'Hello world',
    object = {
        title: title,
        content: content
    };

// create record then update then delete
Model.create(object, function(err, instance) {
    if (err) {
        // do something
    }

    // update instance
    instance.set('content', 'foo');

    // save instance
    instance.update(function(err, result){
        // logic here
    });

    // delete instance
    instance.delete(function(err,result){
        // logic here
    });
});
```

Run a query

The following is a simple example of performing a query against a model.

```
// setup query options
var options = {
    where: { content: { $like: 'Hello%' } },
    sel: { content: 1 },
    order: { title: -1, content: 1 },
    limit: 3,
    skip: 0
};
// execute query
Model.query(options, function(err, coll) {
    // process results
});
```

If none of these values are present in `options`, the `options` object is treated as a `where` statement.

```
// setup query options
var options = {
    content: { $like: 'Hello%' }
};
// execute query
Model.query(options, function(err, coll) {
    // process results
});
```

Restricting CRUD endpoints

By default, models support the basic CRUD methods (CREATE, READ, UPDATE, and DELETE). You can limit the methods supported by a model by using the `actions` property.

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

var emp = Arrow.Model.reduce('appc.mysql/employee','emp',{
    fields: {
        fname: { type:String, description:'First name', name:'first_name',
required:true},
        lname: { type:String, description:'Last name', required:true,
name:'last_name'},
        email: { type:String, description:'Email address', required:true,
name:'email_address'}
    },
    actions:['create','read'],
    connector: 'appc.mysql'
});

module.exports = emp;
```

In this example, the model only allows `create` (POST) or `read` (GET). DELETE and PUT are not allowed and would fail.

The valid values for the `action` property are: `create`, `read`, `update`, `delete`, and `deleteAll`.

Predefined or custom endpoints

By default, API Builder generates the following API endpoints for models:

- `GET /api/<model_name>` : Return all objects (the first 1000 records).
- `GET /api/<model_name>/query` : Return all objects that satisfy a query.
- `GET /api/<model_name>/:id` : Return a specific object by id
- `GET /api/<model_name>/distinct` : Find distinct objects
- `GET /api/<model_name>/count` : Count objects
- `PUT /api/<model_name>/:id` : Update a specific user by id
- `PUT /api/<model_name>/findAndModify` : Find and modify an object
- `POST /api/<model_name>` : Create a new object
- `POST /api/<model_name>/upsert` : Create or update an object
- `DELETE /api/<model_name>/:id` : Delete a specific object by id
- `DELETE /api/<model_name>` : Delete all objects

To disable API Builder from generating these endpoints, set the Model's `autogen` property to `false` when defining the model. You will need to create API Builder API objects to access the model.

Example

The following model disabled generating pre-defined endpoints. An API endpoint needs to be defined to access the model data as shown below.

models/employee.js

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

var employee = createModel('employee', {
    fields: {
        first_name: {type:String, description:'First name', required:true},
        last_name: {type:String, description:'Last name', required:true},
        email_address: {type:String, description:'Email address', required:true}
    },
    connector: 'memory',
    autogen: false
});

module.exports = employee;
```

The example below implements the `GET /api/<employee>/:id` endpoint that would normally be generated by API Builder.

apis/employeefindById.js

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

var findEmployeeById = Arrow.API.extend({
    group: 'employeeAPIs',
    path: '/api/employee/:id',
    method: 'GET',
    description: 'This API finds one employee record',
    model: 'employee',
    parameters: {
        id: {description: 'the employee id'}
    },
    action: function (req, resp, next) {
        resp.stream(req.model.find, req.params.id, next);
    }
});

module.exports = findEmployeeById;
```