Mobilizing Enterprise Data with LoopBack Models

How to use LoopBack, the open source backend-as-a-service built-on Node.js, to connect devices and browsers to enterprise data

Table of Contents

Introduction

Rich mobile applications are driven by data. Data can be produced and consumed by mobile devices, browsers, cloud services, legacy applications, databases, and other backend systems. LoopBack by StrongLoop enables mobile developers to connect devices and browsers to data, whether it's behind the corporate firewall or in the cloud. LoopBack is an open source, backend-as-a-service built-on Node.js.

LoopBack mobilizes data through *models* that represent business data and behavior. LoopBack exposes models to mobile apps through REST APIs and client SDKs. You'll need to interact with the model differently, depending on the location and type of data. In this technical paper, we'll explain some of the most important recipes for working with LoopBack models:

- Part 1: Open models - for free-form data.
- Part 2: Models with schema definitions such as relational databases.
- Part 3: Model discovery with relational databases - consuming existing data from a relational database.
- Part 4: Models by instance introspection - Consuming JSON data from NoSQL databases or REST APIs.
- Part 5: Model synchronization with relational databases - keeping your model synchronized with the database.

The source code for the examples in this paper are available at:

https://github.com/strongloop/loopback-sample-recipes

Prerequisites

If you'd like to follow along in the examples provide, simply install StrongLoop on your platform of choice from our Get Started page:

http://strongloop.com/get-started/

And then follow the appropriate installation instructions for your platform, which can be found in the Getting Started section of our documentation.

http://docs.strongloop.com/#getting-started

Part 1: Open Models

Let's start with the simplest one: open models.

I'm mobile developer. Can LoopBack help me save and load data transparently? I don't need to worry about the backend or define the model up front, because my data is free form.

For free-form data, use an *open model* that allows you to set any properties on model instances.

The following code creates an open model and exposes it as a REST API:

```
   var loopback = require('loopback');
   var app = loopback(); // Create an instance of LoopBack

   // Create an in memory data source
   var ds = loopback.createDataSource('memory');

   // Create a open model that doesn't require predefined
properties
   var FormModel = ds.createModel('form');

   // Expose the model as REST APIs
   app.model(FormModel);
   app.use(loopback.rest());

   // Listen on HTTP requests
   app.listen(3000, function () {
       console.log('The form application is ready at
http://127.0.0.1:3000');
   });
```

Notice the call to `ds.createModel()` with only a name to create an open model.

To try it out, enter the following command:

```
curl -X POST -H "Content-Type:application/json" -d '{"a":
1, "b": "B"}' http://127.0.0.1:3000/forms
```

This command POSTs some simple JSON data to the LoopBack `/forms` URI.

The output that the application returns is a JSON object for the newly created instance.

```
{
 "id": "52389f5f7d365dd52a000005",
 "a": 1,
 "b": "B"
}
```

The id field is a unique identifier you can use to retrieve the instance:

```
curl -X GET http://127.0.0.1:3000/forms/52389f5f7d365dd52a000005
```

Note: Your ID will be different as it is generated by the database. Please copy it from the POST response.

Try submitting a different form:

```
curl -X POST -H "Content-Type:application/json" -d '{"a":
"A", "c": "C", "d": true}' http://localhost:3000/forms
```

Now you see the newly created instance as follows:

```
{
 "id": "5238c1e492f7b69535000001",
 "a": "A",
 "c": "C",
 "d": true
}
```

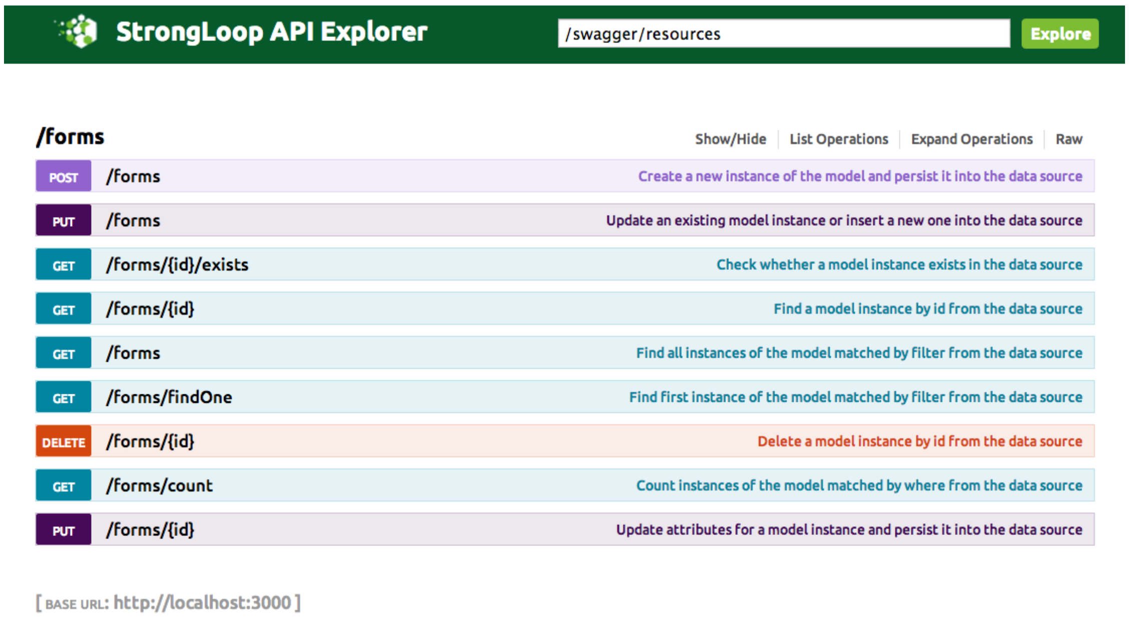
For the complete list of REST APIs that LoopBack scaffolds for a model, please refer to the documentation:

http://docs.strongloop.com/loopback/#rest-api

If you have StrongLoop Suite (http://strongloop.com/strongloop-suite/downloads/) installed, you can build and run this project from scratch in 5 commands:

```
$ slc lb project free-form
$ cd free-form
$ slc lb model form
$ slc install
$ slc run app
```

Now open a browser and point it to http://localhost:3000/explorer.


StrongLoop API Explorer

/swagger/resources

Explore

/forms

Show/Hide | List Operations | Expand Operations | Raw

POST	/forms	Create a new instance of the model and persist it into the data source
PUT	/forms	Update an existing model instance or insert a new one into the data source
GET	/forms/{id}/exists	Check whether a model instance exists in the data source
GET	/forms/{id}	Find a model instance by id from the data source
GET	/forms	Find all instances of the model matched by filter from the data source
GET	/forms/findOne	Find first instance of the model matched by filter from the data source
DELETE	/forms/{id}	Delete a model instance by id from the data source
GET	/forms/count	Count instances of the model matched by where from the data source
PUT	/forms/{id}	Update attributes for a model instance and persist it into the data source

[BASE URL: http://localhost:3000]

You get an explorer to try the forms APIs right away. The UI is pretty straightforward; feel free to play with it. For more information, check out:

http://docs.strongloop.com/loopback/#api-explorer

The open model is simple and flexible. It works well for free-form style data because the model doesn't constrain the properties and their types. But for other scenarios, a predefined model is preferred to validate the data and ensure it can be exchanged among multiple systems.

Next, let’s talk about models with schema definitions.

Part 2: Models with Schema Definitions

In Part 1, we looked at how you can mobilize data through LoopBack with open models, which works well for free-form style data. In this section we’ll look at creating models with schema definitions.

I want to build a mobile application that will interact with some backend data. I would love to see a working REST API and mobile SDK before I implement the server side logic.

In this case, we'll define a model first and use an in-memory data source to mock up the data access. You'll get a fully-fledged REST API without writing a lot of server side code.

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

   var ds = loopback.createDataSource('memory');

   var Customer = ds.createModel('customer', {
       id: {type: Number, id: true},
       name: String,
       emails: [String],
       age: Number},
       {strict: true});
```

The snippet above creates a 'Customer' model with a numeric id, a string name, an array of string emails, and a numeric age. Please also note we set the 'strict' option to be true for the settings object so that LoopBack will enforce the schema and ignore unknown ones.

For more information about the syntax and APIs to define a data model, check out:

http://docs.strongloop.com/loopback-datasource-juggler/#loopback-definition-language-guide

You can now test the CRUD operations on the server side. The following code creates two customers, finds a customer by ID, and then finds customers by name to return up to three customer records.

```
  // Create two instances
   Customer.create({
       name: 'John1',
       emails: ['john@x.com', 'jhon@y.com'],
       age: 30
   }, function (err, customer1) {
       console.log('Customer 1: ', customer1.toObject());
       Customer.create({
           name: 'John2',
           emails: ['john@x.com', 'jhon@y.com'],
           age: 30
       }, function (err, customer2) {
           console.log('Customer 2: ',
customer2.toObject());
           Customer.findById(customer2.id, function(err,
customer3) {
               console.log(customer3.toObject());
           });
           Customer.find({where: {name: 'John1'}, limit:
3}, function(err, customers) {
```

```
                customers.forEach(function(c) {
                    console.log(c.toObject());
                });
            });
        });
    });
```

To expose the model as a REST API, use the following:

```
    var app = loopback();
    app.model(Customer);
    app.use(loopback.rest());
    app.listen(3000, function() {
        console.log('The form application is ready at
http://127.0.0.1:3000');
    });
```

Until now the data access has been backed by an in-memory store. To make your data persistent, simply replace it with a MongoDB database by changing the data source configuration:

```
 var ds = loopback.createDataSource('mongodb', {
    "host": "demo.strongloop.com",
    "database": "demo",
    "username": "demo",
    "password": "L00pBack",
    "port": 27017
 });
```

For more information about data sources and connectors, please check out:

http://docs.strongloop.com/loopback-datasource-juggler/#loopback-datasource-and-connector-guide

When defining a model, it may be troublesome to define all the properties from scratch. Fortunately, LoopBack can discover a model definition from existing systems such as relational databases or JSON documents, as we'll see in part 3.

Part 3: Model Discovery with Relational Databases

In Part 2, we looked at schema definitions and defined a model using an in-memory source to mock up the data access. This time around, we are looking at model discovery with existing relational databases or JSON documents.

I have data in an Oracle database. Can LoopBack figure out the models and expose them as APIs to my mobile applications?

LoopBack makes it surprisingly simple to create models from existing data, as illustrated below for an Oracle database. First, the code sets up the Oracle data source. Then the call to `discoverAndBuildModels()` creates models from the database tables. Calling it with `associations: true` makes the discovery follow primary/foreign key relations.

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

   var ds = loopback.createDataSource('oracle', {
                "host": "demo.strongloop.com",
                "port": 1521,
                "database": "XE",
                "username": "demo",
                "password": "L00pBack"
           });

   // Discover and build models from INVENTORY table
   ds.discoverAndBuildModels('INVENTORY', {visited:
{}, associations: true},
       function (err, models) {

       // Now we have a list of models keyed by the
model name
       // Find the first record from the inventory
       models.Inventory.findOne({}, function (err,
inv) {
           if(err) {
               console.error(err);
               return;
           }
           console.log("\nInventory: ", inv);
           // Navigate to the product model
           inv.product(function (err, prod) {
               console.log("\nProduct: ", prod);
               console.log("\n ------------- ");
           });
```

```
        });
    });
```

For more information, please read:

http://docs.strongloop.com/loopback-datasource-juggler/#discovering-model-definitions-from-the-data-source

Discovery from relational databases is a quick way to consume existing data with well-defined schemas. However, some data stores don't have schemas; for example, MongoDB or REST services. LoopBack has another option here. In Part 4 we will take a look at models by instance introspection.

Part 4: Models by Instance Introspection

In Part 3, we looked at using LoopBack with relational databases, allowing you to consume existing data. This time around, we are looking at your options when the data does not have a schema.

I have JSON documents from REST services and NoSQL databases. Can LoopBack get my models from them?

Yes, certainly! Here is an example:

```
var ds = require('../data-sources/db.js')('memory');
```

```
 // Instance JSON document
 var user = {
     name: 'Joe',
     age: 30,
     birthday: new Date(),
     vip: true,
     address: {
        street: '1 Main St',
        city: 'San Jose',
        state: 'CA',
        zipcode: '95131',
        country: 'US'
     },
     friends: ['John', 'Mary'],
     emails: [
        {label: 'work', id: 'x@sample.com'},
        {label: 'home', id: 'x@home.com'}
     ],
     tags: []
```

```
};

// Create a model from the user instance
var User = ds.buildModelFromInstance('User', user, {idInjection: true});

// Use the model for CRUD
var obj = new User(user);

console.log(obj.toObject());

User.create(user, function (err, u1) {
    console.log('Created: ', u1.toObject());
    User.findById(u1.id, function (err, u2) {
        console.log('Found: ', u2.toObject());
    });
});
```

Now we understand that we can define the models from scratch, or discover them from relational databases or JSON documents. How can we make sure that the database models are in sync with LoopBack if some of the database models don't exist or are different? LoopBack has APIs to facilitate the synchronization, as we'll see in the final section.

Part 5: Model Synchronization with Relational Databases

In Part 4 we looked at how to use LoopBack while defining models from scratch. In this last part, we demonstrate how to synchronize your data.

Now I have defined a LoopBack model, can LoopBack create or update the database schemas for me?

LoopBack provides two ways to synchronize model definitions with table schemas:

- Auto-migrate: Automatically create or re-create the table schemas based on the model definitions. WARNING: An existing table will be dropped if its name matches the model name.
- Auto-update: Automatically alter the table schemas based on the model definitions.

Auto-Migration

Let's start with auto-migration of model definition. Here's an example:

```
  var schema_v1 =
   {
        "name": "CustomerTest",
        "options": {
            "idInjection": false,
            "oracle": {
                "schema": "LOOPBACK",
                "table": "CUSTOMER_TEST"
            }
        },
        "properties": {
            "id": {
                "type": "String",
                "length": 20,
                "id": 1
            },
            "name": {
                "type": "String",
                "required": false,
                "length": 40
            },
            "email": {
                "type": "String",
                "required": false,
                "length": 40
            },
            "age": {
                "type": "Number",
                "required": false
            }
        }
   };
```

Assuming the model doesn't have a corresponding table in the Oracle database, you can create the corresponding schema objects to reflect the model definition:

```
  var ds = require('../data-sources/db')('oracle');
   var Customer = require('../models/customer');

   ds.createModel(schema_v1.name, schema_v1.properties,
schema_v1.options);

   ds.automigrate(function () {
       ds.discoverModelProperties('CUSTOMER_TEST', function
(err, props) {
           console.log(props);
       });
   });
```

This creates the following objects in the Oracle database:

- A table CUSTOMER_TEST.
- A sequence CUSTOMER_TEST_ID_SEQUENCE for keeping sequential IDs.
- A trigger CUSTOMER_ID_TRIGGER that sets values for the primary key.

Now we decide to make some changes to the model. Here is the second version:

```
  var schema_v2 =
   {
       "name": "CustomerTest",
       "options": {
           "idInjection": false,
           "oracle": {
               "schema": "LOOPBACK",
               "table": "CUSTOMER_TEST"
           }
       },
       "properties": {
           "id": {
               "type": "String",
               "length": 20,
               "id": 1
           },
           "email": {
               "type": "String",
               "required": false,
               "length": 60,
               "oracle": {
                   "columnName": "EMAIL",
                   "dataType": "VARCHAR",
                   "dataLength": 60,
                   "nullable": "Y"
               }
           },
           "firstName": {
               "type": "String",
               "required": false,
               "length": 40
           },
           "lastName": {
               "type": "String",
               "required": false,
               "length": 40
           }
       }
   }
```

Auto-update

If we run auto-migrate again, the table will be dropped and data will be lost. To avoid this problem use auto-update, as illustrated here:

```
  ds.createModel(schema_v2.name, schema_v2.properties,
schema_v2.options);

   ds.autoupdate(schema_v2.name, function (err,
result) {
       ds.discoverModelProperties('CUSTOMER_TEST',
function (err, props) {
           console.log(props);
       });
   });
```

Instead of dropping tables and recreating them, auto update calculates the difference between the LoopBack model and the database table definition and alters the table accordingly. This way, the column data will be kept as long as the property is not deleted from the model.

Conclusion

This series has walked through a few different use cases and how LoopBack handles each. Take a look at the table below for a round up of what was covered.

Model	Use Case	Strict Mode	Database
Open Model	Taking care of free-form data	false	NoSQL
Plain Model	Defining a model to represent data	true or false	NoSQL or RDBMS
Model from discovery	Consuming existing data from RDB	true	RDBMS
Model from introspection	Consuming JSON data from NoSQL/REST	false	NoSQL
Model synchronization	Making sure models are in sync	true	RDBMS

Next Steps

- Get the code used in this paper
  https://github.com/strongloop/loopback-sample-recipes
- Get LoopBack on your platform of choice
  http://strongloop.com/get-started/
- Installation instructions
  http://docs.strongloop.com/ - getting-started
- LoopBack documentation
  http://docs.strongloop.com/loopback/

References

1. https://github.com/strongloop/loopback-sample-recipes
2. http://docs.strongloop.com/loopback/
3. http://docs.strongloop.com/loopback-connector-oracle/