LoopBack 2.0

LoopBack is a highly-extensible, open-source Node.js framework that enables you to:

- Create dynamic end-to-end REST APIs with little or no coding.
- Access data from major relational databases, MongoDB, SOAP and REST APIs.
- Incorporate model relationships and access controls for complex APIs.
- Use built-in push, geolocation, and file services for mobile apps.
- Easily create client apps using Android, iOS, and JavaScript SDKs.
- Run your application on-premises or in the cloud.

Follow Getting Started with LoopBack to run through some of LoopBack's key features:

- Create a simple API
- Use API Explorer
- Connect an API to a datasource
- Extend your API
- Add a client app
- Learn more

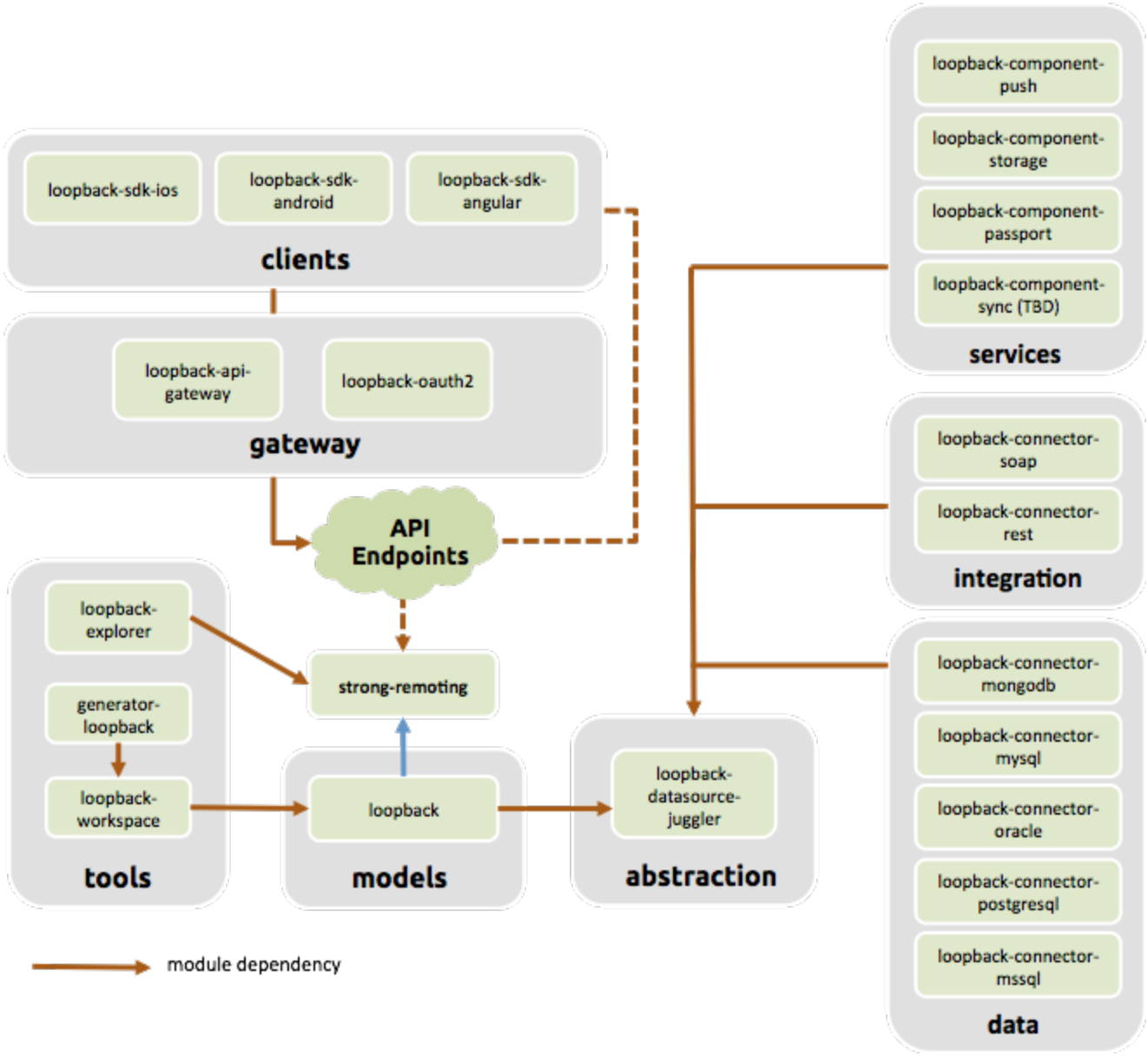
See also: LoopBack API documentation

The LoopBack framework is a set of Node.js modules that you can use independently or together.

An application interacts with data sources through the LoopBack model API, available locally within Node.js, remotely over REST, and via native client APIs for iOS, Android, and HTML5. Using the API, apps can query databases, store data, upload files, send emails, create push notifications, register users, and perform other actions provided by data sources and services.

Clients can call LoopBack APIs directly using Strong Remoting, a pluggable transport layer that enables you to provide backend APIs over REST, WebSockets, and other transports.

The following diagram illustrates key LoopBack modules, how they are related, and their dependencies.



LoopBack framework modules

Category	Description	Use to...	Modules
models	Model and API server	Quickly and dynamically mock up models and expose them as APIs without worrying about persisting.	loopback
abstraction	Model data abstraction to physical persistence	Connect to multiple data sources or services and get back an abstracted model with CRUD capabilities independent on how it is physically stored.	loopback-datasource-juggler
data	RDBMS and noSQL physical data sources	Enable connections to RDBMS, noSQL data sources and get back an abstracted model.	loopback-connector-mongodb loopback-connector-mysql loopback-connector-postgresql loopback-connector-msssql loopback-connector-oracle

integration	General system connectors	Connect to an existing system that expose APIs through common enterprise and web interfaces	loopback-connector-rest loopback-connector-soap
services	Prebuilt services	Integrate with prebuilt services for common use cases to be utilized within LoopBack applications packaged into components.	loopback-component-push loopback-component-storage loopback-component-passport loopback-component-sync
gateway (in dev)	API gateway	Secure your APIs and inject quality of service aspects to the invocation and response workflow.	loopback-api-gateway loopback-oauth2
clients	Client SDKs	Develop your client app using native platform objects (iOS, Android, AngularJS) that interact with LoopBack APIs via REST.	loopback-sdk-ios loopback-sdk-android loopback-sdk-angular

Getting Started with LoopBack

Prerequisite: Install Node.js, if you have not already done so.

- **Windows and OSX**: Download the native installers from nodejs.org. **NOTE**: LoopBack does not support Cygwin on Windows.
- **Linux**: See Installing Node.js via package manager.

- Install StrongLoop software
- Start using LoopBack

If you previously installed StrongLoop software prior to 6 August 2014, see Updating to the latest version.

Install StrongLoop software

Enter:

```
$ npm install -g strongloop
```

Depending on your how your file and directory privileges are configured, you may need to use:

```
$ sudo npm install -g strongloop
```

This will install

- StrongLoop Controller (slc) that enables you to run and manage Node applications including LoopBack applications.
- Yeoman and the LoopBack generators to create and scaffold LoopBack applications.
- Grunt, the JavaScript task runner.
- LoopBack Angular command line tools (`lb-ng` and `lb-ng-doc`). See AngularJS JavaScript SDK for details.

If you run into any problems, see Installation troubleshooting.

Start using LoopBack

You can run through these sections in order to get a sense for all the things LoopBack can do, or just skip to the one that interests you.

- Create a simple API
- Use API Explorer
- Connect an API to a datasource
- Extend your API
- Add a client app
- Learn more

Create a simple API

Prerequisite: You must have installed StrongLoop software as described in Getting Started with LoopBack.

LoopBack uses Yeoman to create and *scaffold* applications. Scaffolding simply means generating the basic code for your application. You can then extend and modify the code as desired for your specific needs.

- Create new application
- Create models
- Check out the project structure
- Run the application

Create new application

To create a new application, run the Yeoman LoopBack generator:

```
$ slc loopback
```

The LoopBack Yeoman generator will greet you with some friendly ASCII art and prompt you for the name of the directory in which to create your application. The default is the directory where you ran the command. Enter `myproject`:

```
_-----_
    |       |    .--------------------------.
    |--(o)--|    |  Let's create a LoopBack |
   `---------´   |       application!       |
    ( _´U`_ )    '--------------------------'
    /___A___\
     |  ~  |
   __'.___.'__
 ´   `  |° ´ Y `
[?] Enter a directory name where to create the project: (.) myproject
```

Next, the generator will prompt you for the name of the application. This defaults to the directory name you entered previously, `myproject`. Accept this default by pressing **Enter**.

```
create myproject/
     info change the working directory to myproject
[?] What's the name of your application? (myproject)
```

The generator will scaffold the application including:

1. Initializing the project folder structure
2. Creating default JSON files
3. Creating deafult JavaScript files
4. Downloading and installing dependent Node modules (as if you had manually done `npm install`)

When finished, the generator will suggest creating a model as the next step

```
...
Next steps:
  Create a model in your app
    $ slc loopback:model
  Optional: Enable StrongOps monitoring
    $ slc strongops
  Run the app
    $ slc run .
```

Create models

Now that you've scaffolded the initial project, create a *Person* model that will automatically configure a REST API endpoint for a Person resource.

First, go into your new application directory, then run the generator for LoopBack models:

```
$ cd myproject
$ slc loopback:model
```

The generator will ask for a model name. Enter `Person` as the name of the model:

```
$ slc loopback:model
[?] Enter the model name: Person
```

It'll ask if you want to attach the model to any data-sources that have already been defined. The default in-memory datasource will be the only one available. Hit **Enter** to select the only choice, `db (memory)`:

```
...
[?] Select the data-source to attach Person to: (Use arrow keys)
 db (memory)
```

One of the powerful advantages of LoopBack is that it automatically generates a REST API for your model. The generator will ask whether you want to expose this REST API.

Hit **Enter** again to accept the default and expose the Person model via REST:

```
...
[?] Expose Person via the REST API? (Y/n) Y
```

LoopBack automatically creates a REST route associated with your model using the plural of the model name. By default, it pluralizes the name for you (by adding "s"), but you can specify a custom plural form. See Exposing models over a REST API for all the details.

So, here, enter `People` as the plural form of the model to be used in the REST URL:

```
...
[?] Custom plural form (used to build REST URL): People
```

Every model has properties and you're going to define four properties for the Person model. For each property you'll enter a name, choose a data type, and choose whether its required.

Start with `FirstName` and select `string` type (press **Enter**, since string is the default choice):

```
Let's add some Person properties now.
Enter an empty property name when done.
[?] Property name: FirstName
   invoke   loopback:property
[?] Property type: (Use arrow keys)
 string
  number
  boolean
  object
  array
  date
  buffer
  geopoint
  (other)
```

Each property can be optional or required for the model. Enter `y` to make FirstName required:

```
...
[?] Required? (y/N)
```

Repeat the process above and create three more properties:

- LastName (string), required,
- Age (number), required.
- DOB (date), required.

End the model creation process by simply pressing **Enter** when prompted for a new property.

Check out the project structure

The following describes the application structure as created by the `slc loopback` command. LoopBack does not require that you follow this structure, but if you don't, then you can't use `slc loopback` commands to modify or extend your application.

LoopBack project files and directories are in the *application root directory*. Within this directory the standard LoopBack project structure has three sub-directories:

- `server` - Node application scripts and configuration files.
- `client` - Client JavaScript, HTML, and CSS files.
- `common` - Files common to client and server. The `/models` sub-directory contains all model JSON and JavaScript files.

Put all your model JSON and JavaScript files in the `/common/models` directory.

/server directory - Node application files		
File or directory	**Description**	**How to access in code**
`server.js`	Main application program file.	N/A
`config.json`	Application settings. See config.json.	`app.get('option-name')`
`datasources.json`	Data source configuration file. See datasources.json.	`app.datasources['datasource-name']`
`model-config.json`	Model configuration file. See model-config.json.	N/A
`/boot` directory	Add scripts to perform initialization and setup. See Boot scripts.	Scripts are automatically executed in alphabetical order.
/client directory - Client application files		

README.md	LoopBack generators create empty README file in markdown format.	N/A
Other	Add your HTML, CSS, client JavaScript files.	
/common directory - shared application files		
`/models` directory	Custom model files: • Model definitions, by convention named *`modelName`*`.json`; for example `customer.json`. • Custom model scripts by convention named *`modelName`*`.js`; for example, `customer.js`. See Model definition JSON file.	Node: `myModel = app.models.myModelName`
Top-level application directory		
package.json	Standard npm package specification. See package.json	N/A

Additionally, the top-level directory contains the stub `README.md` file, and `node_modules` directory (for Node dependencies).

Run the application

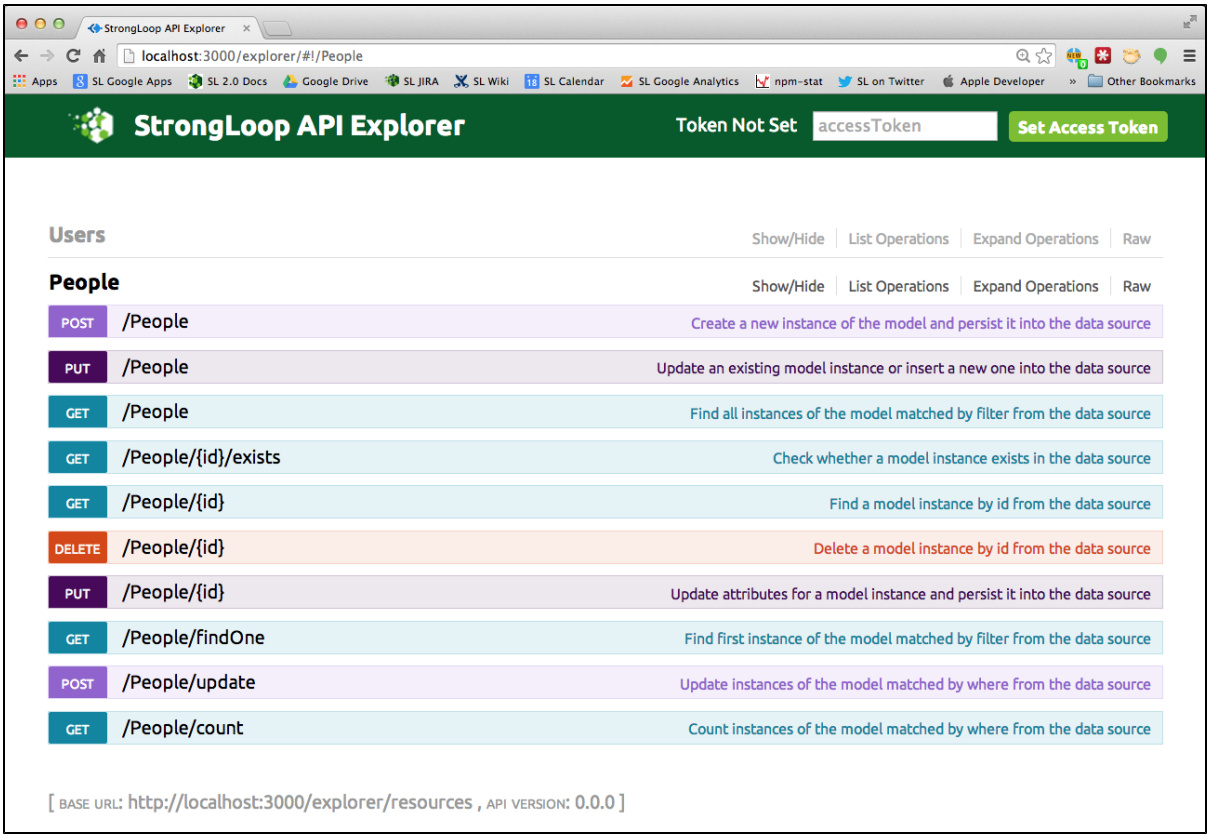
Start the application:

```
$ slc run
Browse your REST API at http://localhost:3000/explorer
Web server listening at: http://localhost:3000/
```

If you want to run the server in a multiprocess cluster, use this command:

```
$ slc run --cluster cpus
```

Now open your browser to http://localhost:3000/explorer. You'll see the StrongLoop API Explorer where you can view the REST endpoint for the Person model you created:



Through a set of simple steps using LoopBack, you've created a Person model, specified its properties and then exposed it through REST. To learn more about LoopBack modeling, see Working with models.

> **Next**: In Use API Explorer, you'll explore the REST API you just created in more depth and exercise some of its operations.

Use API Explorer

> ⚠ **Prerequisite**: You must have installed StrongLoop software as described in Getting Started with LoopBack.

You're probably not the only one who'll use the API you just created. That means you'll need to document your API. Fortunately, LoopBack generates a developer portal / API Explorer for you.

If you followed Create a simple API, keep that app running!

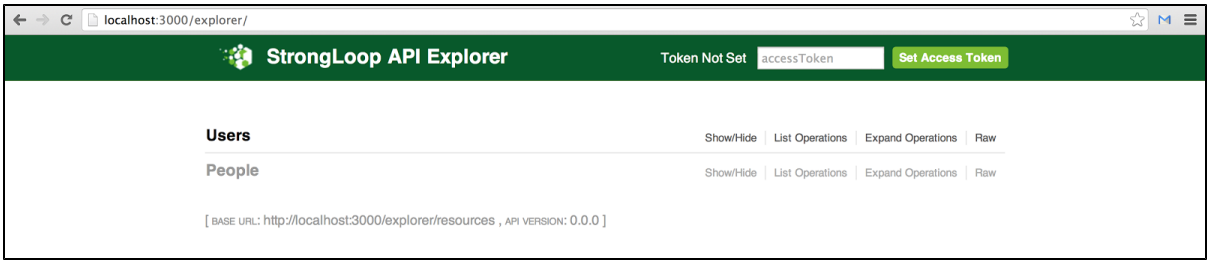
If you're just jumping in just, get the app from GitHub and use `npm install` to install all the app's dependencies:

```
$ git clone https://github.com/strongloop/loopback-example-mySimpleAPI.git
$ npm install
```

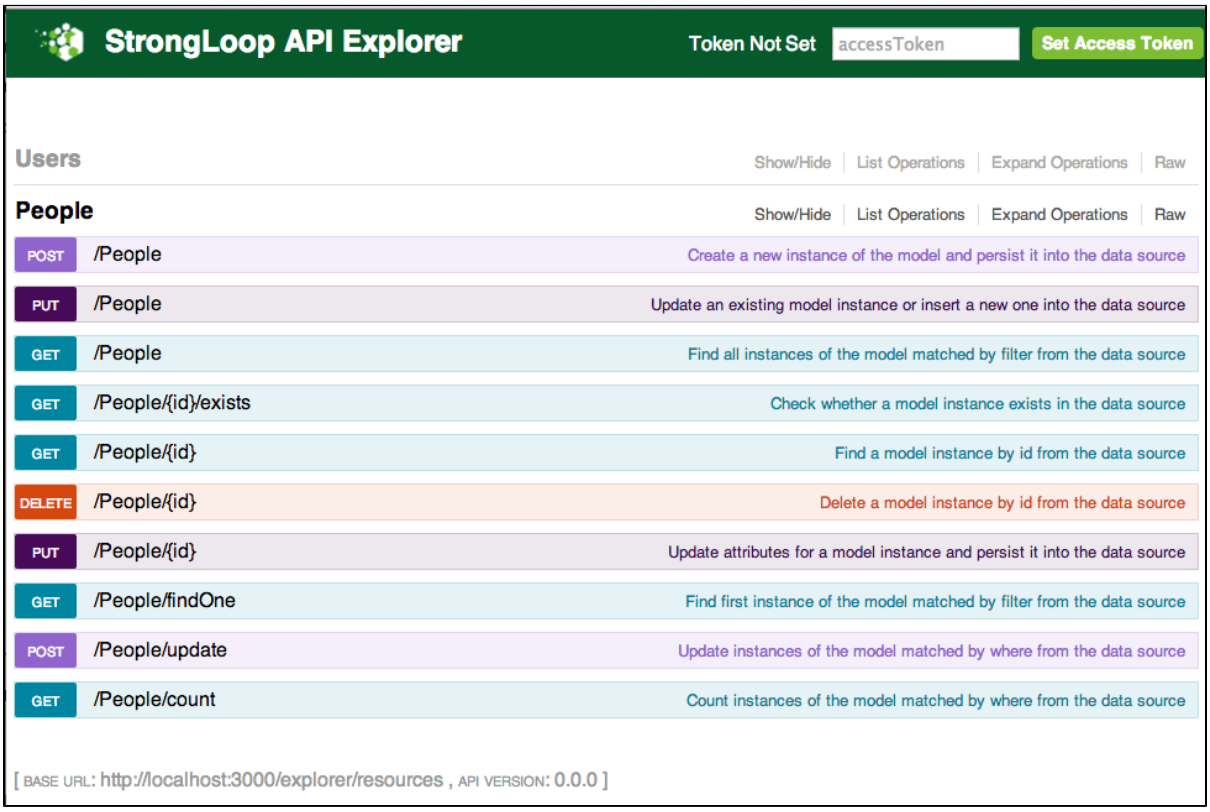
Then run it:

```
$ cd loopback-example-mySimpleAPI
$ slc run
```

Now go to http://localhost:3000/explorer. You'll see the StrongLoop API Explorer showing the two models this application has: **Users** and **People**. In addition to the People model that you defined in the previous section, by default Loopback generates the User model and its endpoints for every application.



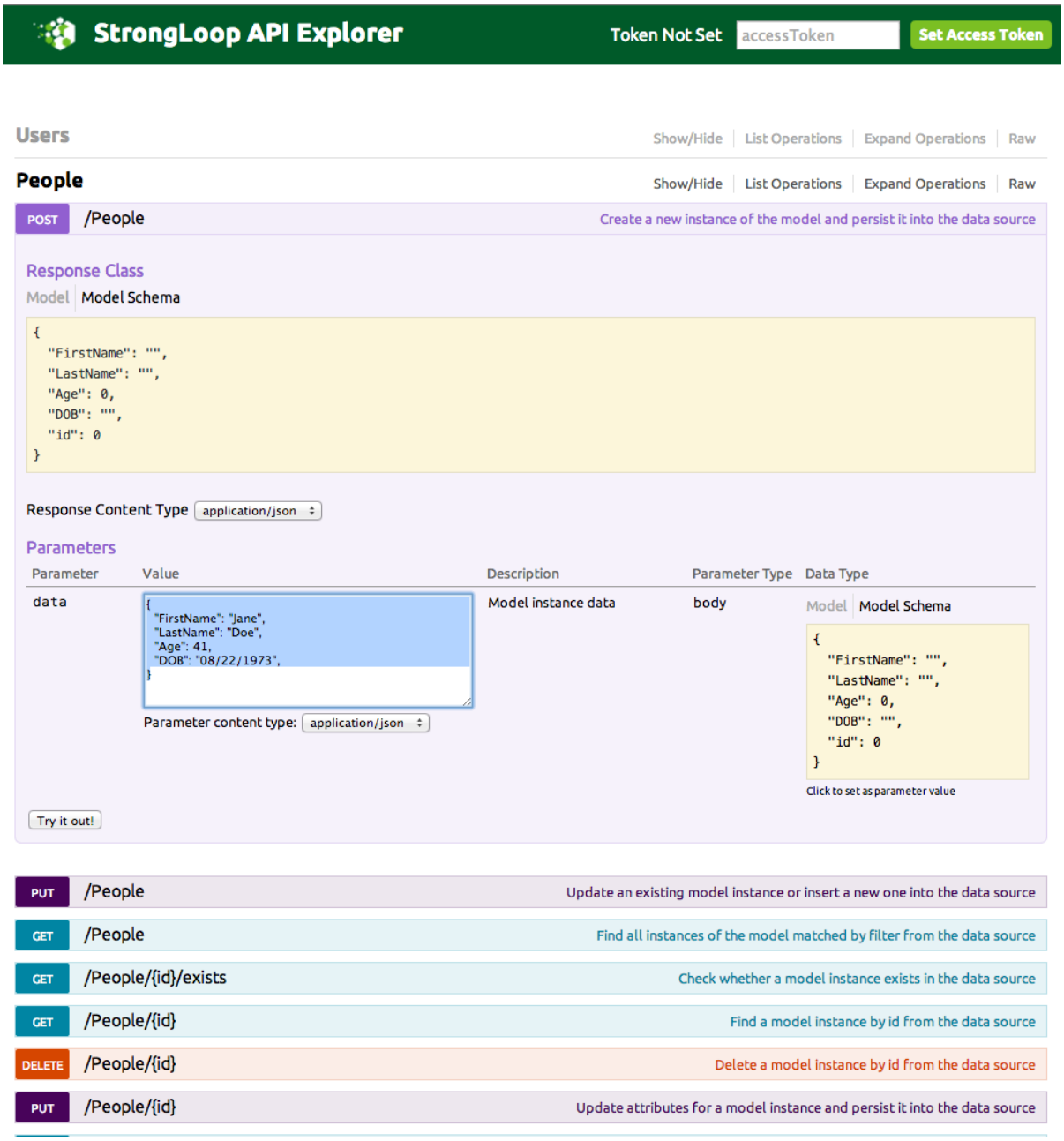
Click on **People** to show all the API endpoints for the People model:



Click on the first row, **POST / People Create a new instance of the model and persist it into the data source** and enter in the JSON string as shown below:

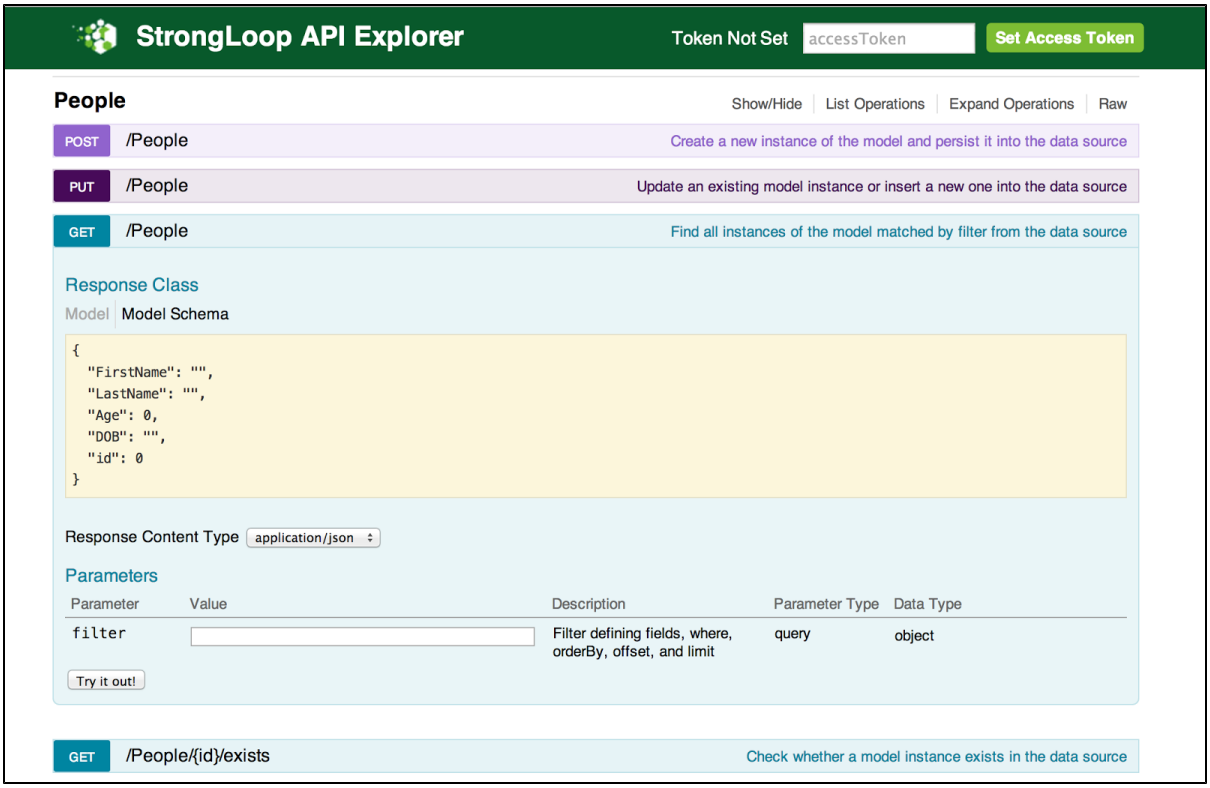
```
{
  "FirstName": "Jane",
  "LastName": "Doe",
  "Age": 41,
  "DOB": "08/22/1973"
}
```

After entering in the JSON string, click the **Try it out!** button.



The Response Body field will show the data that you just entered, returned as confirmation that it was added to the data source.

Click on the third row, **GET /People Find all instances of the model matched by filter from the data source** to expand that endpoint:



Clicking **Try it out!** will retrieve the data associated with the person model. You should see the Jane Doe record you created using the POST API.

Right now you're probably thinking about security. Don't worry, LoopBack provides a full-featured solution for authentication and authorization. See Controlling data access for more information.

> **Next**: In Connect an API to a datasource, you'll learn how to persist your data model to a database such as MongoDB.

Connect an API to a datasource

> ⚠ **Prerequisite**: You must have installed StrongLoop software as described in Getting Started with LoopBack.

LoopBack enables you to easily persist your data model to a variety of data sources without having to write code.

- Add a data source
- Add models for an account in the account DB
- Install MySQL connector
- Configure data source
- Add some test data and view it
- Discover models from the database

You're going to take the app from the previous section and connect it to MySQL. If you're just jumping in just, get the app from GitHub and use `npm install` to install all the app's dependencies:

```
$ git clone https://github.com/strongloop/loopback-example-mySimpleAPI.git
$ cd loopback-example-mySimpleAPI
$ npm install
```

Add a data source

Now you're going to define a data source using the LoopBack datasource generator:

```
$ slc loopback:datasource
```

The generator will prompt you to name the data source:

```
[?] Enter the data-source name:
```

Enter **accountDB** and hit **Enter**.

Next, the generator will prompt you for the type of data source:

```
[?] Select the connector for accountDB: (Use arrow keys)
  other
  In-memory db (supported by StrongLoop)
  MySQL (supported by StrongLoop)
  PostgreSQL (supported by StrongLoop)
  Oracle (supported by StrongLoop)
  Microsoft SQL (supported by StrongLoop)
  MongoDB (supported by StrongLoop)
(Move up and down to reveal more choices)
```

Press the down-arrow key to highlight **MySQL**, then hit **Enter**.

The tool adds the data source definition to the `server/datasources.json` file, which will now look as shown below. Notice the "accountDB" data source you just added, as well as in-memory data source named "db," which is there by default.

datasources.json

```
{
  "db": {
    "name": "db",
    "connector": "memory"
  },
  "accountDB": {
    "name": "accountDB",
    "connector": "mysql"
  }
}
```

Add models for an account in the account DB

Now you're going to run the LoopBack model generator to create a new model. If you followed the previous section Create a simple API, then this will be familiar to you:

```
$ slc loopback:model
```

The generator will prompt you for the model name. Enter **account**.

```
[?] Enter the model name: account
```

The generator will then prompt you for the data source to use. Use the down-arrow key to select **accountDB (mysql)**:

```
[?] Select the data-source to attach account to: accountDB (mysql)
```

Next, the generator will ask if you want to expose the account model over REST and prompt you for a custom plural form. Press **Enter** twice to accept the defaults in both cases: (Yes and "accounts" for the plural):

```
[?] Expose account via the REST API? (Y/n) Yes
[?] Custom plural form (used to build REST URL):
```

Finally, the generator will ask you to define add properties to the model:

```
Let's add some account properties now.
Enter an empty property name when done.
[?] Property name:
```

Since you've already gone through this process in the previous section, Create a simple API, you know the drill. Follow the prompts to add the following properties to the account model:

Property name	Type	Required
email	string	Yes
level	number	Yes
created	date	Yes
modified	date	Yes

Hit **Enter** at the prompt to exit the generator after you define these four properties.

The generator adds the model definition to the `common/models/account.json` file, which will now look as shown below.

account.json

```
{
  "name": "account",
  "properties": {
    "email": {
      "type": "string",
      "required": true
    },
    "level": {
      "type": "number",
      "required": true
    },
    "created": {
      "type": "date",
      "required": true
    },
    "modified": {
      "type": "date",
      "required": true
    }
  },
  "validations": [],
  "relations": {},
  "acls": [],
  "methods": []
}
```

Install MySQL connector

Now add the loopback-connector-mysql module and install the dependencies:

```
$ npm install loopback-connector-mysql --save
```

Configure data source

Next, you need configure the data source to use the StrongLoop MySQL server running on demo.strongloop.com.

Edit `/server/datasources.json` and after the line

`"connector": "mysql"`

add host, port, database, username, and password properties to the accountDB data source, as shown here:

```
{
  "db": {
    "name": "db",
    "connector": "memory"
  },
  "accountDB": {
    "name": "accountDB",
    "connector": "mysql",
    "host": "demo.strongloop.com",
    "port": 3306,
    "database": "demo",
    "username": "demo",
    "password": "L00pBack"
  }
}
```

Add some test data and view it

Now you have an account model in LoopBack, do you need to run some SQL statements to create the corresponding table in MySQL database? You could...but even simpler, LoopBack provides Node APIs to do it for you automatically.

The `loopback-example-mySimpleAPI` module contains the `create-test-data.js` utility script to demonstrate adding data using the Node API. If you've been following along from the beginning (and didn't just clone this module), then you'll need to grab it from GitHub: https://github.com/strongloop/loopback-example-mySimpleAPI/blob/master/server/create-test-data.js. Put it in the application's `/server` directory.

```
$ node server/create-test-data.js
```

This will save some test data to the data source:

```
Record created: { email: 'foo@bar.com',
  level: 10,
  created: Sat Jul 19 2014 17:07:30 GMT-0700 (PDT),
  modified: Sat Jul 19 2014 17:07:30 GMT-0700 (PDT),
  id: 1 }
Record created: { email: 'bar@bar.com',
  level: 20,
  created: Sat Jul 19 2014 17:07:30 GMT-0700 (PDT),
  modified: Sat Jul 19 2014 17:07:30 GMT-0700 (PDT),
  id: 2 }
done
```

Now run the application:

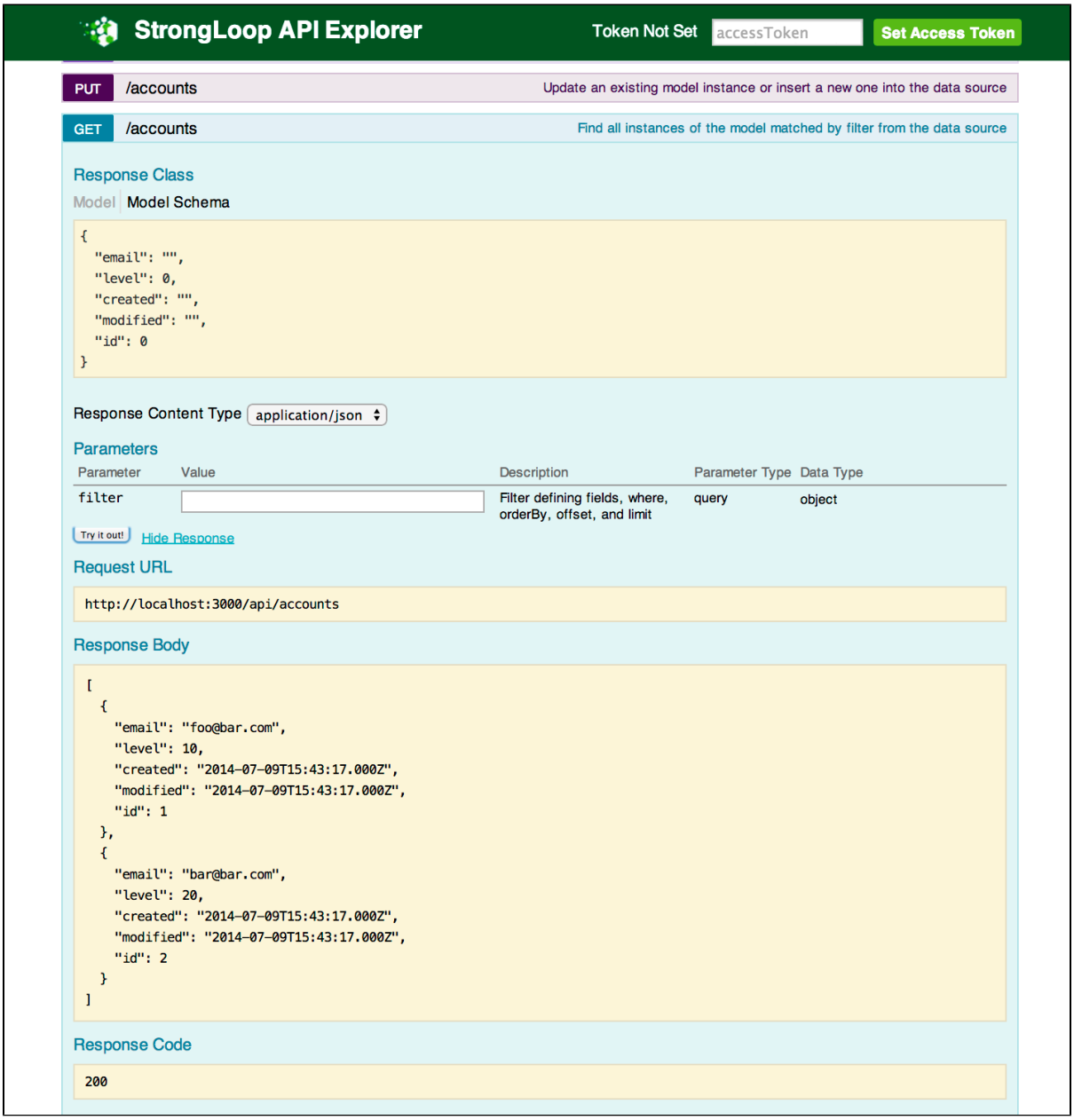
```
$ slc run
```

Load this URL to view the test data you just added: http://localhost:3000/api/accounts.

You'll see the data in JSON format:

```
[{"email":"foo@bar.com","level":10,"created":"2014-07-21T21:05:44.000Z","modified":"20
14-07-21T21:05:44.000Z","id":1},{"email":"bar@bar.com","level":20,"created":"2014-07-2
1T21:05:44.000Z","modified":"2014-07-21T21:05:44.000Z","id":2}]
```


Or, you can use the API Explorer: click **Find all instance of the model matched by filter...** then click **Try it out!**



Discover models from the database

Now you have the account table in MySQL, you can create LoopBack models using the database schema. This is called *model discovery*. See Discovering models from databases for more information.

> The `loopback-example-mySimpleAPI` module contains the `discover.js` utility script to demonstrate model discovery. If you've been following along from the beginning (and didn't just clone this module), then you'll need to grab it from GitHub: https://github.com/strongloop/loopback-example-mySimpleAPI/blob/master/server/discover.js. Put it in the application's `/server` directory.

discover.js Expand source

```
var app = require('./server');
var dataSource = app.dataSources.accountDB;
dataSource.discoverSchema('account', {owner: 'demo'}, function (err, schema) {
    console.log(JSON.stringify(schema, null, '  '));
});
dataSource.discoverAndBuildModels('account', {owner: 'demo'}, function (err, models) {
    models.Account.find(function (err, act) {
        if (err) {
            console.error(err);
        } else {
            console.log(act);
        }
        dataSource.disconnect();
    });
});
```

Run the script:

```
$ node server/discover.js
```

First, you'll see the model definition for account in JSON format displayed in the console:

```
{
  "name": "Account",
  "options": {
    "idInjection": false,
    "mysql": {
      "schema": "demo",
      "table": "account"
    }
  },
...
```

The output above is truncated for brevity.

This example discover script doesn't actually modify the application the model JSON file, but in theory if you wanted to, you could copy and paste the JSON from the console into `account.json` to make the model definition permanent.

Then, the script displays to the console all the account data from MySQL:

```
[ { id: 1,
    email: 'foo@bar.com',
    level: 10,
    created: Mon Jul 21 2014 14:05:44 GMT-0700 (PDT),
    modified: Mon Jul 21 2014 14:05:44 GMT-0700 (PDT) },
  { id: 2,
    email: 'bar@bar.com',
    level: 20,
    created: Mon Jul 21 2014 14:05:44 GMT-0700 (PDT),
    modified: Mon Jul 21 2014 14:05:44 GMT-0700 (PDT) } ]
```

Next: In Extend your API, you'll learn how to add a custom method to your model.

Extend your API

Prerequisite: You must have installed StrongLoop software as described in Getting Started with LoopBack.

If you've been following the tutorial, use your application from the previous section.

If you're just jumping in, get the app from GitHub and then use `npm install` to install all the app's dependencies:

```
$ git clone https://github.com/strongloop/loopback-example-datasourceAPI.git
$ cd loopback-example-datasourceAPI
$ npm install
```

In this section you're going to add a custom remote method to your API. Follow these steps:

1. Create a new JavaScript file in your application's `/common/models` directory, and call it `person.js`.
2. Edit this file and add the following code:

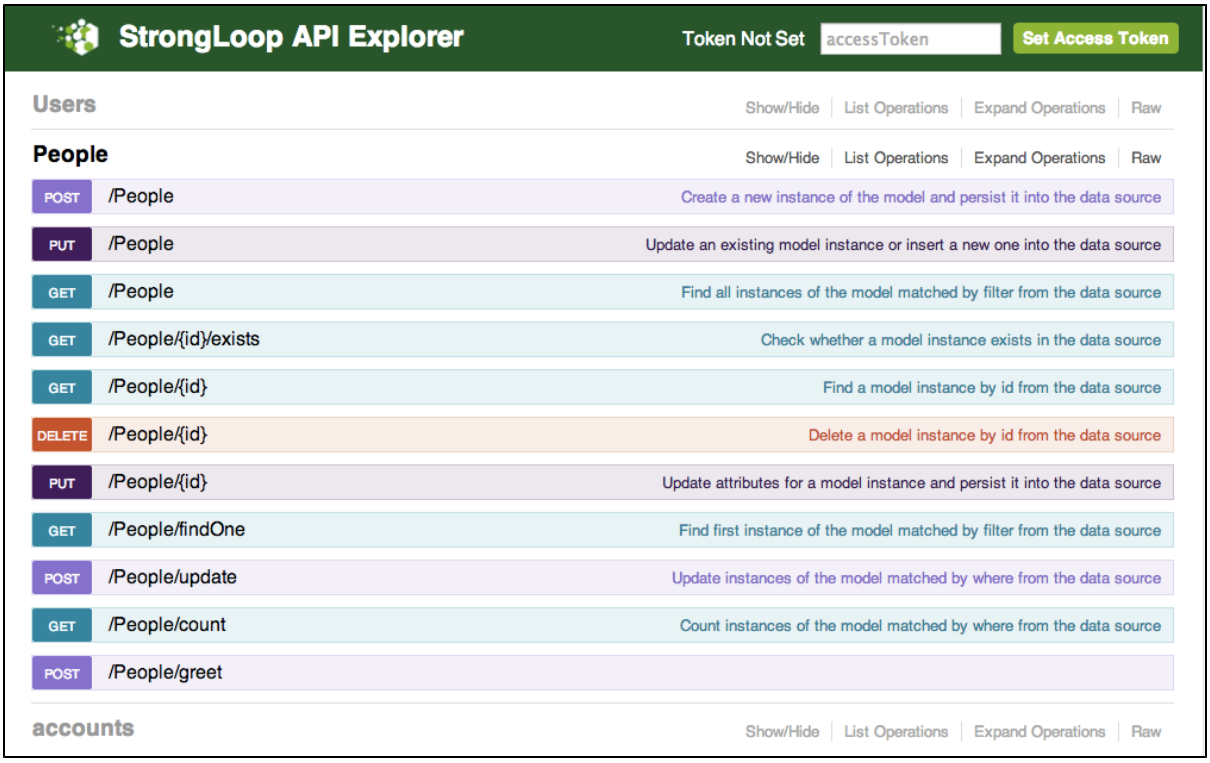
```
module.exports = function(Person){

    Person.greet = function(msg, cb) {
      cb(null, 'Greetings... ' + msg);
    }

    Person.remoteMethod(
        'greet',
        {
          accepts: {arg: 'msg', type: 'string'},
          returns: {arg: 'greeting', type: 'string'}
        }
    );
};
```

   This defines a simple remote method called "greet" that accepts a single string argument, and then just returns JSON containing the message "Greetings... (argument)", for example: `{ greeting: "Greetings ... Fred" }`.

   Of course, in practice you can do much more interesting and complex things with remote methods such as manipulating input data before persisting it to a database. You can also change the route where you call the remote method, and define complex arguments and return values. See Defining remote methods for all the details.
3. Save the file.
4. Now, back in the application root directory, run the app:

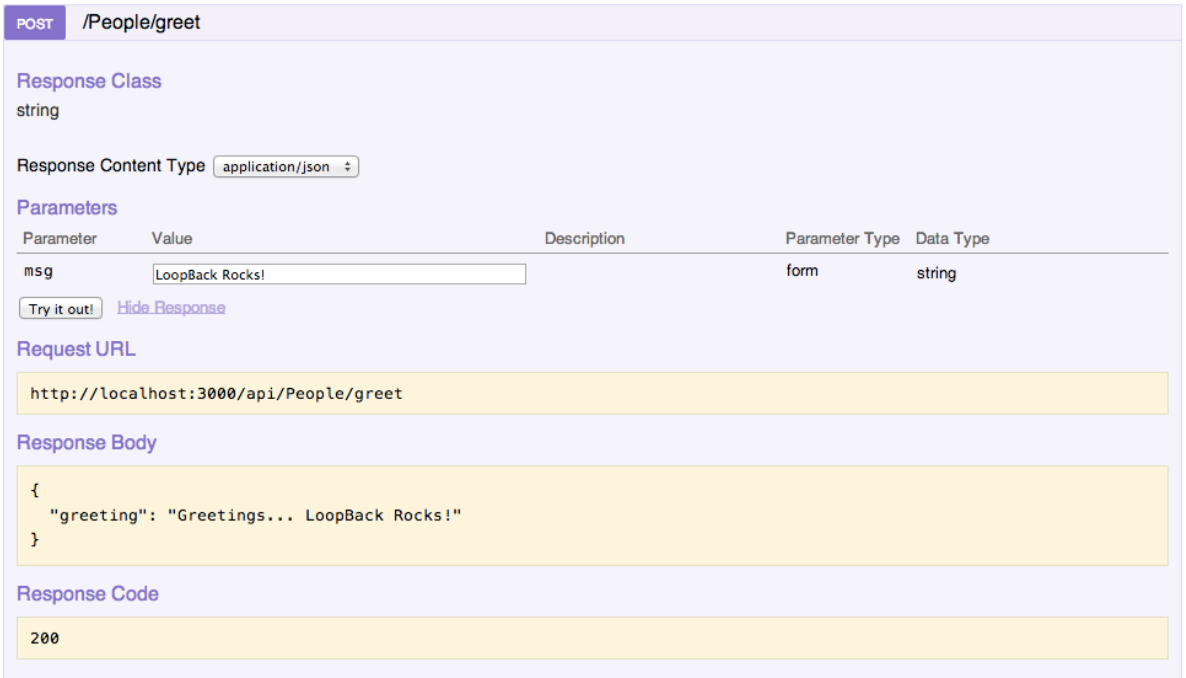
```
$ slc run
```

5. Go to http://localhost:3000/explorer to see the API Explorer. Then click on People to see all the REST API endpoints for the People model:



Check it out! There is a new endpoint, **/People/greet** (at the bottom of the screenshot above) that exposes your remote method.

6. Click on **/People/greet** to view details for the endpoint.
7. Enter a short message in the msg field.
8. Click **Try it Out!**
   You'll see the result of calling your remote method with the string you entered as the argument:



That's how easy it is to add remote methods with LoopBack!

For more information, see Defining remote methods.

Next: In Add a client app, you'll add a client application that connects to the LoopBack server application.

Add a client app

Prerequisite: You must have installed StrongLoop software as described in Getting Started with LoopBack.

LoopBack comes with client libraries for AngularJS, iOS, and Android that can consume the LoopBack REST API. The general principle is that LoopBack's client SDKs reconstruct the server-side models on the client, which enables you to use the LoopBack REST API directly from client JavaScript. So you can just work with your models directly on the client.

In this section, you're going to add a very simple AngularJS client to your app.

If you're just jumping in, get the app from GitHub and then use `npm install` to install all the app's dependencies:

```
$ git clone https://github.com/strongloop/loopback-example-extendedAPI.git
$ cd loopback-example-extendedAPI
$ npm install
```

Create LoopBack Angular services

Now, create a new `/client/js` directory in your application for client JavaScript:

```
$ cd client
$ mkdir js
```

Now, you're going to use the LoopBack AngularJS SDK command-line tool to create AngularJS services that provide client-side representation of the models and remote methods defined in the LoopBack server application.

The `lb-ng` command is the LoopBack Angular SDK command-line tool. You're going to use it to generate the client facet of your application. Enter this command:

```
$ lb-ng ../server/server.js js/lb-services.js
```

This command generates a file called `lb-services.js` that contains the AngularJS interface to the LoopBack models used in `server.js`. In AngularJS, this interface is called the lbServices module.

The SDK also provides a Grunt plugin if you prefer; see AngularJS JavaScript SDK for more information.

View the Angular API

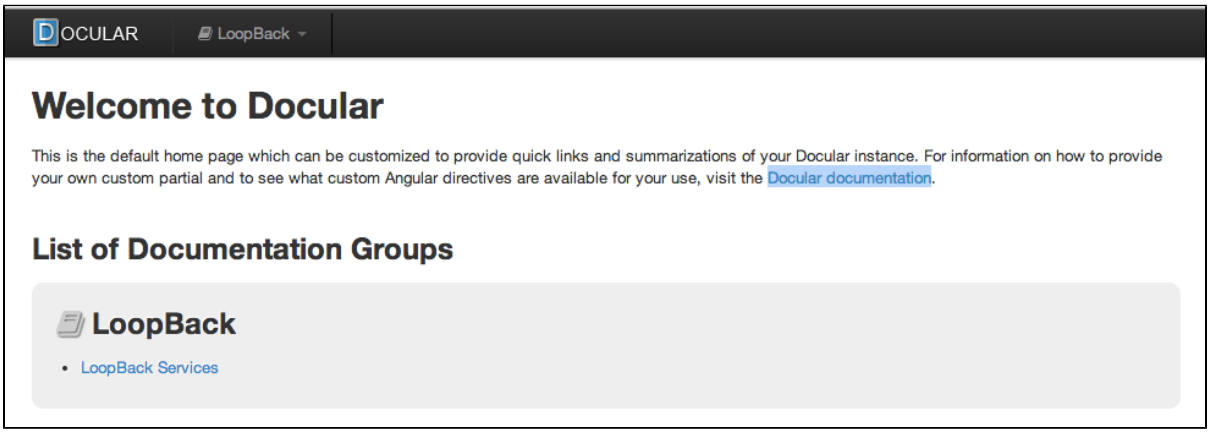
Now you can use another tool (based on Docular) to view documentation for your client-side API. Enter this command:

```
$ lb-ng-doc client/js/lb-services.js
```

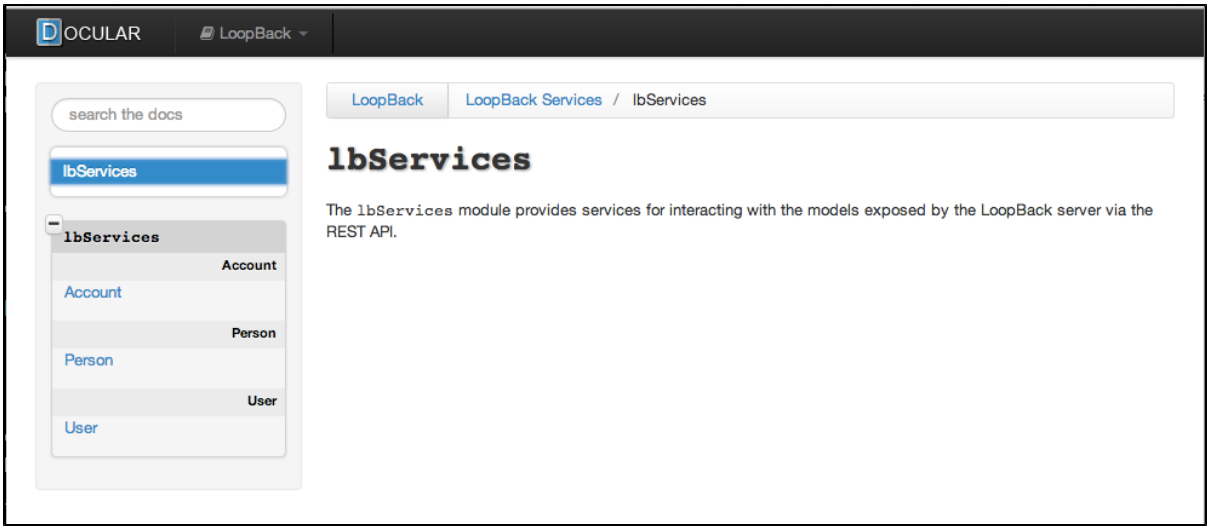
You'll see a bunch of output in your terminal window, ending with:

```
Browse the documentation at http://localhost:3030/
```

Load this UR in your browser: http://localhost:3030/. You'll see the main docular page:



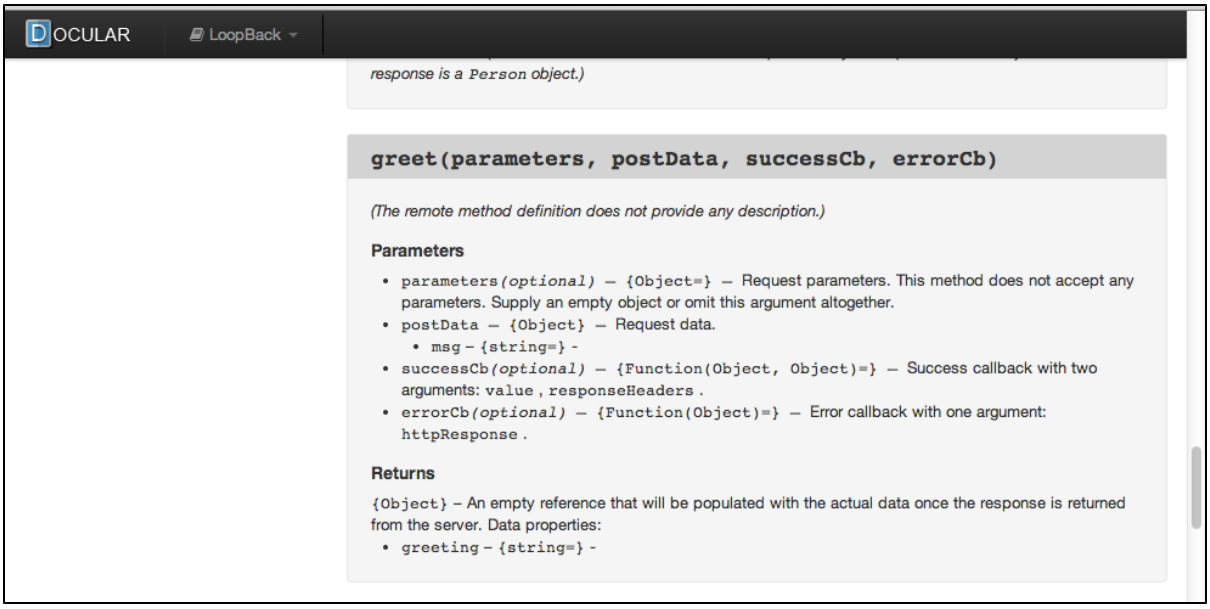
Click on the **LoopBack Services** link. You'll see this page:



> ⚠ In some browsers, there is a bug that causes this page to initially appear empty. If this happens, start typing in the search field, then **lbServices** will appear. Click on that link to display the documentation, or just load http://localhost:3030/documentation/loopback/lbServices/index.

Notice there are links for **Account**, **Person**, and **User**, the three models that the LoopBack application has.

Now, click on **Person** and scroll down to see all the methods defined for this model, for example `count()`, `create()`, `deleteById()`, and so on. These are client-side versions of the standard Model REST API. Scroll down until you see the **greet** method:



This is the server-side remote method you defined in the previous section! But it's here in the client API!

Create your client app

> ⚠ If you're familiar with AngularJS, you may want to skip this section and just refer to the full AngularJS JavaScript SDK documentation. This section is intended for beginning AngularJS developers; however, it may be instructive for more experienced Angular developers to see a very simple example of how LoopBack works with Angular.

AngularJS has copious documentation, and there are lots of different ways to get started. For example, you could use the Yeoman AngularJS generator or ng-boilerplate, but these approaches use tools and frameworks like Bower, Gulp, Less, and Twitter Bootstrap. Although these are great tools and frameworks, this tutorial will demonstrate how to use the AngularJS SDK on its own, so you can pick and choose the tools and client frameworks that are right for you. So, you're going to use the AngularJS seed app, an application skeleton for a typical AngularJS web app. It also uses Bower, but for purposes of this tutorial, you won't be using it.

Since you won't be integrating Bower to manage dependencies or Gulp for builds, you'll have to do things manually. For "real" app development, you'd want to use these tools, but this section will focus just on getting an AngularJS client to work with a LoopBack backend.

Get the Angular seed app

First, clone the Angular seed app:

```
$ git clone https://github.com/angular/angular-seed.git
```

Copy selected files

Now you're going to copy some files from this application into your LoopBack application. All the files will go into the application's existing `/client` directory.

First, create the following sub-directories in `/client`, adding them to the `/js` directory you created previously:

- `/css` for CSS files
- `/partials` for Angular partial templates (fragments of templates)

Now copy the following files from the Angular seed app:

Files in Angular seed app	Copy to LoopBack app
/app/index.html	/client/index.html

/app/js/app.js /app/js/controllers.js /app/js/directives.js /app/js/filters.js /app/js/services.js	/client/js/app.js /client /js/controllers.js /client/js/directives.js /client /js/filters.js /client/js/services.js
/app/partials/partial1.html /app/partials/partial2.html	/client/partials/partial1.html /client/partials/partial2.html
/app/css/app.css	/client/css/app.css

If you cloned the Angular seed app to a directory parallel to your LoopBack application's root directory, you can copy all the files as follows:

```
$ cd angular-seed/app
$ cp index.html ../../loopback-example-extendedAPI/client
$ cp -R js ../../loopback-example-extendedAPI/client
$ cp -R partials ../../loopback-example-extendedAPI/client
$ cp -R css ../../loopback-example-extendedAPI/client
```

If your application root directory is different than `loopback-example-extendedAPI`, then of course use your directory name instead.

Modify index.html

Because you'll be using a simplified app in this tutorial and not using Bower, you're going to comment out a few lines in the Angular seed app index.html file. You can also just delete them if you prefer. Comment out lines as shown in the code snippet below:

index.html

```
...
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <!--
  <link rel="stylesheet" href="bower_components/html5-boilerplate/css/normalize.css">
  <link rel="stylesheet" href="bower_components/html5-boilerplate/css/main.css">
  -->
  <link rel="stylesheet" href="css/app.css"/>
  <!--
  <script
src="bower_components/html5-boilerplate/js/vendor/modernizr-2.6.2.min.js"></script>
  -->
</head>
<body>
  <ul class="menu">
    <li><a href="#/view1">view1</a></li>
    <li><a href="#/view2">view2</a></li>
  </ul>
  <!--[if lt IE 7] Remove this
      <p class="browsehappy">You are using an <strong>outdated</strong> browser.
Please <a href="http://browsehappy.com/">upgrade your browser</a> to improve your
experience.</p>
  <![endif]
  -->
  ...
  <!--
  <script src="bower_components/angular/angular.js"></script>
  <script src="bower_components/angular-route/angular-route.js"></script>
  --->
  ...
```

Modify route handling

Add static route handling for client files

You need to make a small modification to the default scaffolded code to make the application serve the static files (HTML, CSS, and client JavaScript) in the `/client` directory. By default, the `server.js` file that `slc loopback` contains the following code commented out; remove the comments to serve the static files in the `/client` directory:

/server/server.js

```
// -- Mount static files here--
// All static middleware should be registered at the end, as all requests
// passing the static middleware are hitting the file system
// Example:
  var path = require('path');  /* REMOVE COMMENTS ON THESE LINES */
  app.use(loopback.static(path.resolve(__dirname, '../client')));
```

⚠ An earlier version of the LoopBack generator had a mistake in the second line of code, and was missing the `require()` call. Make sure your code matches the above two lines to avoid errors.

Remove LoopBack root route handling

By default, the scaffolded application has the following in `/server/boot/root.js` to handle requests to the "/" route:

/server/boot/root.js

```
module.exports = function(server) {
  // Install a `/` route that returns server status
  var router = server.loopback.Router();
  router.get('/', server.loopback.status());
  server.use(router);
};
```

This code makes the app return JSON that shows the time the application was started and how long it's been running. However, you want to use the root route, so comment out (or delete) the function body, so this file looks like this:

Modified /server/boot/root.js

```
module.exports = function(server) {
  // Install a `/` route that returns server status
  /*
  var router = server.loopback.Router();
  router.get('/', server.loopback.status());
  server.use(router);
  */
};
```

Register the LoopBack services module

Now edit `/client/js/app.js`. You need to register the LoopBack services module (lbServices) that you created previously.

/client/js/app.js

```
angular.module('myApp', [
  'ngRoute',
  'myApp.filters',
  'myApp.services',
  'lbServices',  // ADD THIS LINE
  'myApp.directives',
  'myApp.controllers'
]).
config(['$routeProvider', function($routeProvider) {
  $routeProvider.when('/view1', {templateUrl: 'partials/partial1.html', controller:
'MyCtrl1'});
  $routeProvider.when('/view2', {templateUrl: 'partials/partial2.html', controller:
'MyCtrl2'});
  $routeProvider.otherwise({redirectTo: '/view1'});
}]);
```

Add the line shown above on line 5 to register the LoopBack services in the call to `angular.module()`.

Notice the lines at the bottom that configure routes for the partial templates using `$routeProvider.when()`, specifically that the partial templates `partial1.html` and `partial2.html` are bound to controllers MyCtrl1 and MyCtrl2 and routes /view1 and /view2, respectively.

Modify controller and template

Open `/client/js/controllers.js`, which looks like this:

```
...
angular.module('myApp.controllers', [])
  .controller('MyCtrl1', ['$scope', function($scope) {
  }])
  .controller('MyCtrl2', ['$scope', function($scope) {
  }]);
```

Modify it to add the lbServices Account model to controller MyCtrl1 as follows:

```
angular.module('myApp.controllers', [])
  .controller('MyCtrl1', ['$scope', 'Account', function($scope, Account) {
     $scope.account = Account.count();  // Add LoopBack model
  }])
  .controller('MyCtrl2', ['$scope', function($scope) {
  }]);
```

Now open `/client/partials/partial1.html` and add this line:

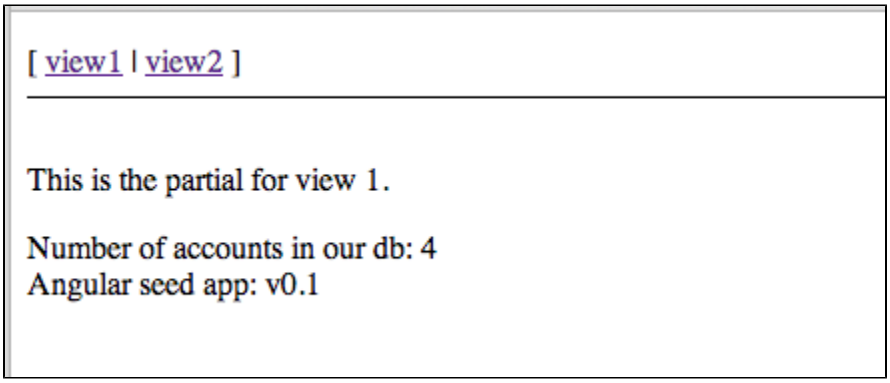
```
<div>Number of accounts in our db: <span>{{account.count}}</span></div>
```

Run it!

Finally the app is ready to go. Run it from the application root directory:

```
$ slc run
```

Then load http://localhost:3000/ to view the app:



It shows the total number of accounts in the Account model. You can test this by opening the API Explorer at http://localhost:3000/explorer/ and adding or deleting accounts, then reload the view1 page to see that it reflects the change.

You've seen how to connect a basic AngularJS client with your LoopBack application. For more information, see AngularJS JavaScript SDK.

Learn more

There's so much more to learn about LoopBack! We've got lots of Tutorials and examples both here in the documentation, in the StrongLoop blog , and on GitHub.

Here are some more links to documentation to get you started:

- Creating models

Migrating existing apps to version 2.0

This guide describes steps to upgrade your LoopBack 1.x project to LoopBack 2.0. There are two steps:

1. Upgrade the runtime to LoopBack 2.0.
2. Upgrade the project structure (layout) to the new convention.

The second step is optional. You can use the LoopBack 2.0 runtime while keeping the old 1.x project structure.

However, if you want to be able to use the `slc loopback` command, you need to move to the new project structure.

Runtime

Follow the procedures in this section to run your application with the LoopBack 2.0 framework.

Update package.json

The first step is to change the module versions in your `package.json`:

package.json

```
{
  "dependencies": {
    "loopback": "^2.0.0",
    "loopback-datasource-juggler": "^2.0.0"
  }
}
```

You must also update the names of modules that were renamed if they are in your `package.json`:

Version 1	Version 2
loopback-passport	loopback-component-passport
loopback-storage-service	loopback-component-storage
loopback-push-notification	loopback-component-push

If your application calls `app.boot()` (true for all projects scaffolded using `slc lb project`), add `loopback-boot@1.x` to your dependencies:

```
$ npm install --save-dev loopback-boot@1.x
```

⚠ Use loopback-boot version 1.x, unless you plan to change your application to use the 2.0 project layout. The loopback-boot 2.x module is for applications using the new layout.

Use Express 4.x

LoopBack 2.0 uses ExpressJS v4.x, which introduces some major backwards-incompatible changes. For more information, see ExpressJS 4.0: New Features and Upgrading from 3.0.

Middleware

First of all, Express 4.x no longer bundles middleware: applications must install middleware. LoopBack 2.0 makes the transition easier by providing wrapper methods exposing the Express 3.x API and printing a helpful error message when the middleware module is not installed.

To determine all middleware you need to install, (repeatedly) run your application and follow the error messages; for example:

An example error message

```
Error: The middleware loopback.errorHandler is not installed.
Please run `npm install --save errorhandler` to fix the problem.
```

Routing

The second important change is the removal of `app.router`.

app.js (loopback 1.x)

```
// ...
app.use(app.router);
// ...
```

If your application adds custom Express routes, move all your route definitions out of the main app file and use `loopback.Router` to provide a middleware that can be mounted instead of `app.router`.

app.js (loopback 2.x)

```
// ...
app.use(require('./routes'));
// ...
```

routes/index.js (loopback 2.x)

```
var loopback = require('loopback');
var router = module.exports = loopback.Router();

// define your custom routes on the router object

router.get('/custom', function(req, res, next) {
  // ...
});
```

If your application does not define any custom routes, you can remove the `app.router` code from your `app.js` file instead.

Model definitions

In LoopBack 1.x, all models were descendants of `loopback.Model`.

In LoopBack 2.x, all models with CRUD operations have to descend from `PersistedModel`. When the model options do not specify a base model, `PersistedModel` is used by default. This is different from LoopBack 1.x, where models extend `Model` by default.

Projects started on LoopBack 1.x are affected in two ways:

1. Models attached to database datasources (for example, MySQL, Oracle, MongoDB) must extend `PersistedModel`. If your `models.js` on file specifies `Model` as the base class, you should either remove this option or change the value to `PersistedModel`.

   model.json (database-backed model)

   ```
   {
     "options": {
       "base": "PersistedModel"
     },
     "properties": {
     },
     "dataSource": "db"
   }
   ```

2. Models attached to non-database-like datasources like REST or SOAP must explicitly specify `Model` as the base class.

   models.json (model attached to a REST-connector datasource)

   ```
   {
     "options": {
       "base": "Model"
     },
     "properties": {
     },
     "dataSource": "rest"
   }
   ```

Remote methods

You define remote methods differently in LoopBack 2.0. Instead of calling `loopback.remoteMethod()`, you call `remoteMethod()` on the model object itself. For details, see:

- LoopBack 2.0 release notes
- Defining remote methods

Not exposing methods over REST

In LoopBack 1.x, to not expose (or "hide") a method via REST, you did, for example:

```
var Location = app.models.Location;
Location.deleteById.shared = false;
```

In version 2.0, you do, for example:

```
var isStatic = true;
MyModel.sharedClass.find('deleteById', isStatic).shared = false;
```

For more information, see Exposing models over a REST API

Troubleshooting

The REST API created for a database-backed model does not expose CRUD operations.

Your model is extending `Model` instead of `PersistedModel`. See the section "Model definitions" above.

The REST API created for a REST-connector based model incorrectly includes CRUD operations.

Your model is extending `PersistedModel` instead of `Model`. See the section "Model definitions" above.

Project layout

The first step is to update the loopback-boot dependency in project's `package.json` file.

package.json

```
{
 "dependencies": {
   "loopback-boot": "^2.0.0"
 }
}
```

App settings

The files with applications settings were renamed from `app.*` to `config.*`. and moved to `server` sub-directory. Rename and move the following files to upgrade a 1.x project for loopback-boot 2.x.

LoopBack 1.x	LoopBack 2.0
app.json	server/config.json
app.local.json	server/config.local.json
app.local.js	server/config.local.js

Data sources

Data source configuration is the same in both 1.x and 2.x versions, but the configuration file was moved to `server` sub-directory.

LoopBack 1.x	LoopBack 2.0
datasources.json	server/datasources.json
datasources.local.json	server/datasources.local.json
datasources.local.js	server/datasources.local.js

Models

Model definitions are now in JSON files in the `/common/models` directory.

The file `models.json` that contained both model definition and model configuration is no longer used. It is replaced by a new file, `server/model-config.json` that describes the models in the application. Models referenced in this file must be either built-in models (for example, `User`) or be custom models defined by a JSON file in the `common/models/` folder.

The folder `models/` has different semantics in 2.x than in 1.x. Instead of extending models already defined by `app.boot` and `models.json`, it provides a set of model definitions that do not depend on any application that may use them.

Perform the following steps to update a 1.x project for loopback-boot 2.x. All code samples are referring to the sample project described above.

1. Move all model definition metadata from `models.json` to new per-model JSON files in `common/models/` directory.

   models/car.json

   ```
   {
     "name": "car",
     "properties": { "color": "string", }
   }
   ```

   Keep only the data source configuration in `models.json` (which you will rename to `/server/model-config.json`).

   model-config.json

   ```
   { "car": { "dataSource": "db" } }
   ```

2. Change per-model JavaScript files to export a function that adds custom methods to the model class.

   models/car.js

   ```
   module.exports = function(Car) {
     // Please note that other models might NOT be available using app.models yet
     Car.prototype.honk = function(duration, cb) {
       // make some noise for `duration` seconds
       cb();
     };
   };
   ```

3. Move `models.json` to `server/model-config.json`.
4. Add an entry to `server/model-config.json` to specify the paths where to look for model definitions.

models.json

```
{
  "_meta": { "sources": ["../common/models", "./models"] },
  "Car": { "dataSource": "db" }
}
```

Accessing the application object and other models

The LoopBack application object is not available at the time when the function exported from the custom model file is executed. However, it is available when custom model methods are executed.

common/models/my-model.js

```
module.exports = function(MyModel) {
  MyModel.custom = function(cb) {
    // Get the main app object created using
    //   var app = loopback();
    var app = MyModel.app;
    // Access another model configured in models-config.json
    var User = app.models.User;
    // etc.
  };
};
```

If your model's setup requires the application object (or other models), you have to defer it until the model was added to the application.

common/models/my-model.js

```
 module.exports = function(MyModel) {
  MyModel.on('attached', function() {
    var app = MyModel.app;
    // Execute the setup steps that require the app object
  });
};
```

Attaching built-in models

Models provided by LoopBack, such as `User` or `Role`, are no longer automatically attached to default data sources. The data source configuration entry `defaultForType` is silently ignored.

You have to explicitly configure the built-in models that your application uses in the `server/model-config.json` file.

```
{
  "Role": { "dataSource": "db" }
}
```

Boot scripts

The boot scripts from `boot` folder must be moved to `server/boot` .

In loopback 1.x, the recommended way of accessing the main application object was to use `require('../app')`. While this option is still supported by loopback-boot 2.0, it is recommended to export a function accepting the application object as the first argument.

server/boot/authentication.js

```
module.exports = function enableAuthentication(server) {
  // enable authentication
  server.enableAuth();
};
```

LoopBack 2.0 release notes

Yeoman generators

For LoopBack 2.0, the Command-line tool, `slc loopback`, now uses Yeoman generators to create and scaffold applications. The `slc lb` and `slc example` commands are now deprecated.

ExpressJS 4.x

LoopBack 2.0 uses ExpressJS v4.x, which introduces some major backwards-incompatible changes. For more information, see ExpressJS 4.0: New Features and Upgrading from 3.0. For instructions on migrating your existing app to LoopBack 2.0, see Migrating existing apps to version 2.0.

New project structure

LoopBack 2.0 has a new canonical project structure. Although you're not required to use this structure, when you create an app with slc loopback , it will have the new structure. And if your app does have the new structure, you can use the Yeoman generators to add new models, properties, data sources, and access control settings.

Support for external configuration

Applications now have greater flexibility in managing application configuration for multiple environments (for example: development, staging, and production). For example, you can have a `config.json` with development settings, `config.staging.json` with settings for the staging environment, and `config.production.json` with settings for production.

New object PersistedModel

The base `Model` class is now specifically for defining data structures. `PersistedModel` is the default class for models defined with `app.model(name, ...);` and `models.json`.

Note: loopback 1.x comes with `DataModel`, which is a predecessor of `PersistedModel`. The `DataModel` is no longer available in loopback 2.0.

Bootstrapper was moved to its own module

The implementation of `app.boot` was moved to the module `loopback-boot`. The 1.x versions are backwards compatible with `app.boot`, the current 2.x versions are using the new project layout.

Email connector uses nodemailer 1.x

See http://www.nodemailer.com/ for details.

The method loopback.getModel throws for unknown model names

In loopback 1.x, `loopback.getModel('unknown-name')` returns `undefined`. In loopback 2.x, the method throws an error instead. Use `loopback.findModel` if you need the old semantics.

Remote methods

The function `loopback.remoteMethod()` is deprecated along with attaching remote configuration directly to model objects. Instead you should use the actual `SharedMethod` object.

Now, define a remote method by calling the `remoteMethod()` method on the model, for example, as follows:

```
// static
MyModel.greet = function(msg, cb) {
  cb(null, 'greetings... ' + msg);
}

MyModel.remoteMethod(
  'greet',
  {
    accepts: [{arg: 'msg', type: 'string'}],
    returns: {arg: 'greeting', type: 'string'}
  }
);
```

⚠ Remote instance method support is also now deprecated. Use static methods instead. If you absolutely need it, you can set `options.isStatic = false`.
Remote instance methods will likely not be supported at all in future releases.

New loopback-component modules

Several modules used by LoopBack have been renamed.

Version 1	Version 2
loopback-passport	loopback-component-passport
loopback-storage-service	loopback-component-storage
loopback-push-notification	loopback-component-push

Experimental features

⛔ The following features have not been fully tested and may still be in a state of flux. They are not officially supported, and the APIs may change in future releases.

Executing native SQL

If you want to execute SQL directly against your data-connected model, use the following:

```
dataSource.connector.query(sql, params, cb);
```

Where:

- *sql* - The SQL string.
- *params* - parameters to the SQL statement.
- *cb* - callback function

⚠ The actual method signature depends on the specific connector being used. See connector source code. Use caution and be advised that the API may change in the future.

Model-level hooks

See https://github.com/strongloop/loopback-datasource-juggler/blob/master/test/hooks.test.js

Multi-module bundles

Previously,you had to install and declare dependencies for individual modules; for example:

```
npm install -g strong-cli
npm install loopback --save
npm install loopback-connector-mongodb --save
```

This is flexible, but not very fun when you have to deal with a lot of modules – especially keeping straight what versions work together.

The StrongLoop multi-module bundles:

- Simplify the dependency management, as now we have the aggregate modules serving as groups
- Ensure consistency: as the modules under a given parfait are tested to work together
- Get what you want without any more anything less
- Allow you to install more as you evolve your application.

When you do `npm install -g strongloop`, you may run into peerDependency conflicts for modules that are already installed globally either through `npm install -g` or `npm link`.

If this occurs, you may have conflicting versions of modules installed globally. Inspect your global Node module directory (typically `/usr/local/lib/node_modules`) and evaluate whether to remove the conflicting modules manually. Be aware that `slc update` will update them automatically.

Behind the scenes, each bundle module has a `package.json` that lists child modules as peer dependencies, as detailed below.

Module	Purpose	Peer Deps
strongloop	Installs all command-line utilities and tools for StrongLoop products that need to be installed globally. • All StrongLoop Controller (slc) commands available • Yeoman generators to scaffold LoopBack applications through `slc loopback` (or, equivalently, `yo loopback`) • Angular SDK generator to automatically bind LoopBack models to native Angular models.	• "strong-cli": "2.x", • "generator-loopback": "0.x", • "loopback-sdk-angular-cli": "1.x"
loopback-base	Add models, the in-memory connector and db to your existing Node applications and expose as REST endpoints. • Build Express-based web applications using models that are server -ide rendered. • Can be added to any Node application to get model and REST capabilities without persistence methods. • Can be used to develop mock REST endpoints. • Build Angular based web applications pre-scaffolded through the LoopBack Angular SDK and CLI. • Build any single page application (SPA) web appls integrated to your choice of JavaScript client MVC (Backbone, Ember, Knockout, etc.) through REST.	• "loopback": "2.x", • "loopback-boot": "2.x", • "loopback-datasource-juggler": "2.x", • "loopback-explorer": "1.x", • "strong-remoting": "2.x"

loopback-mobile	In addition to loopback-base, adds pre-built mobile models and services to your application and also provides SDKs. • Adds LoopBack component that adds pre-built mobile functionality like push notifications • Provides all client SDKs for mobile platforms including iOS, Android and Angular for hybrid or mobile web apps • Provides all open source LoopBack connectors to back end data sources and services	• "loopback-framework": "2.x", • "loopback-sdk-ios": "1.x", • "loopback-sdk-android": "1.x", • "loopback-connectors": "2.x", • "loopback-component-push": "1.x", • "loopback-component-storage": "1.x"
loopback-connectors	Provides persistent methods to models so that you can do CRUD against data sources along with LoopBack open source connectors. • Can be installed on top of loopback-base or loopback-mobile • Comes with all open source LoopBack connectors • Adds CRUD methods to models like .find and save with extensive filter support	• "loopback-datasource-juggler": "2.x", • "loopback-connector": "1.x", • "loopback-connector-mongodb": "1.x", • "loopback-connector-mysql": "1.x", • "loopback-connector-postgresql": "1.x", • "loopback-connector-rest": "1.x"
strongloop-connectors	Provides commercial connectors to LoopBack based applications to access enterprise data sources and integrate with enterprise systems.: • Can be installed in addition to loopback-base or loopback-mobile • Can be used in conjunction with loopback-connectors	• "loopback-datasource-juggler": "2.x", • "loopback-connector": "1.x", • "loopback-connector-mssql": "1.x", • "loopback-connector-oracle": "1.x", • "loopback-connector-soap": "1.x"
strongloop-studio	Studio - Product not yet released.	• "strong-studio": "2.x"
strongloop-agent	Monitoring / apm	• "strong-agent": "0.x", • "strong-agent-statsd": "0.x"

Command-line reference (slc loopback)

Use the `slc loopback` command to create and *scaffold* applications. Scaffolding simply means generating the basic code for your application. You can then extend and modify the code as desired for your specific needs.

Prerequisite

To use the LoopBack generators, install StrongLoop software following the instructions in Getting Started with LoopBack.

Under the hood, `slc loopback` uses uses Yeoman. If you are already using Yeoman and are comfortable with it, you can install the LoopBack generator directly with

```
npm install -g generator-loopback.
```

Then everywhere the documentation says to use `slc loopback` just use `yo loopback` instead. For example, to create a new model, use `yo loopback:model`.

Generators

The `slc loopback` command provides an LoopBack application generator to create a new LoopBack application and a number of sub-generators to scaffold an application:

- LoopBack ACL generator
- LoopBack application generator
- LoopBack datasource generator
- LoopBack example generator
- LoopBack model generator
- LoopBack property generator
- LoopBack relation generator

LoopBack ACL generator

The LoopBack ACL generator adds a new access control list (ACL) entry to the LoopBack application.

```
$ cd <loopback-app-dir>
$ slc loopback:acl
```

The tool will prompt you to determine ACL options and then add an entry to the Model definition JSON file.

LoopBack application generator

The LoopBack application generator creates a new LoopBack application:

```
$ slc loopback
```

You will be greeted by the friendly Yeoman ASCII art (under the hood `slc` uses Yeoman) and prompted for:

- Name of the directory in which to create your application. Press **Enter** to create the application in the current directory.
- Name of the application, that defaults to the directory name you previously entered.

The tool creates the standard LoopBack application structure. See Project layout reference for details.

> By default, a generated application exposes only the User model over REST. To expose other built-in models, edit `/server/model-config.json` and change the model's "public" property to "true". See model-config.json for more information.

LoopBack datasource generator

The LoopBack datasource generator adds a new data source definition to an existing application.

```
$ cd <loopback-app-dir>
$ slc loopback:datasource [name]
```

The tool will prompt you:

- To enter the name of the new data source. If you supplied a name on the command-line, just hit **Enter** to use it.
- To select the connector to use for the data source.

For example:

```
$ slc loopback:datasource
[?] Enter the data-source name: corp2
[?] Select the connector for corp2: (Use arrow keys)
  other
 In-memory db (supported by StrongLoop)
  MySQL (supported by StrongLoop)
  PostgreSQL (supported by StrongLoop)
  Oracle (supported by StrongLoop)
  Microsoft SQL (supported by StrongLoop)
  MongoDB (supported by StrongLoop)
  SOAP webservices (supported by StrongLoop)
  REST services (supported by StrongLoop)
  Neo4j (provided by community)
  Kafka (provided by community)
  other
```

By default, not all the choices are shown initially. Move the cursor down to display additional choices.

LoopBack example generator

The LoopBack example generator (technically, sub-generator) creates a LoopBack example application.

```
$ slc loopback:example
```

Use the `-l` option to list the operations performed.

To connect to a relational or NoSQL database (not just the in-memory connector), you need a correctly configured build environment to install native Node modules. For more information, see Installing compiler tools.

About the sample app

The StrongLoop sample is a mobile app for i-Cars, an (imaginary) car rental dealer with outlets in major cities around the world.

The application enables customers to find the closest available cars using the i-Car app on a smartphone. The app shows a map of nearby rental locations and lists available cars in the area shown on the map. In addition, the customer can filter the list of cars by make, model, class, year and color. The customer can then select the desired car and reserve it via the app. If not logged in the app prompts the customer to login or register. The app indicates if the desired car is available and if so, confirms the reservation.

Note that the sample app is the backend functionality only; that is, the app has a REST API, but no client app or UI to consume the interface.

For more details on the sample app, see StrongLoop sls-sample-app in GitHub.

Connecting to other data sources

By default, the LoopBack sample app connects to the in-memory data source. To connect to other data sources, use the following command to run the application:

```
$ DB=datasource node app
```

where *datasource* is either "mongodb", "mysql", or "oracle". The sample app will connect to database servers running on demo.strongloop.com.

LoopBack model generator

The LoopBack model generator creates a new model in an existing LoopBack application.

```
$ cd <loopback-app-dir>
$ slc loopback:model [model-name]
```

where *`model-name`* is the name of the model you want to create (optional on command line).

The tool will prompt you for:

- The name of the model. If you supplied a name on the command-line, just hit **Enter** to use it.
- The data source to which to attach the model. By default, only the Memory connector data source exists. You can add additional data sources using the LoopBack datasource generator.
- Whether you want to expose the model over a REST API. If the model is exposed over REST, then all the standard create, read, update, and delete (CRUD) operations are available via REST endpoints; see Model REST API for more information. You can also add your own custom remote methods that can be called via REST operations; see Defining remote methods.
- If you choose to expose the model over REST, the custom plural form of the model. By default, the LoopBack uses the standard English plural of the word. The plural form is used in the REST API; for example `http://localhost:3000/api/locations`.

The tool will create a new file `/common/models/`*`model-name`*`.json` defining the specified model. See Model definition JSON file for details.

Then, the tool will invoke the LoopBack property generator and prompt you to enter model properties; for example:

```
$ slc loopback:model
[?] Enter the model name: inventory
[?] Select the data-source to attach inventory to: db (memory)
[?] Expose inventory via the REST API? Yes
Let's add some inventory properties now.
...
```

LoopBack property generator

The LoopBack property generator adds a property to an existing model in a LoopBack application.

```
$ cd <loopback-app-dir>
$ slc loopback:property
```

The tool will then prompt you to:

- Select from the models in the application, to which it will add the new property.
- Enter the name of the property to add.
- Select the data type of the property.
- Whether the property is required.

For example:

```
$ slc loopback:property
[?] Select the model: inventory
[?] Enter the property name: price
[?] Property type: (Use arrow keys)
  string
 number
  boolean
  object
  array
  date
  buffer
  geopoint
  (other)
```

LoopBack relation generator

The LoopBack relation generator creates a new model relation in the LoopBack application.

```
$ cd <loopback-app-dir>
$ slc loopback:relation
```

Documentation for this new feature is incomplete. Check back soon for more information or use the `--help` option in the meantime.

Project layout reference

The following describes the application structure as created by the `slc loopback` command. LoopBack does not require that you follow this structure, but if you don't, then you can't use `slc loopback` commands to modify or extend your application.

LoopBack project files and directories are in the *application root directory*. Within this directory the standard LoopBack project structure has three sub-directories:

- `server` - Node application scripts and configuration files.
- `client` - Client JavaScript, HTML, and CSS files.
- `common` - Files common to client and server. The `/models` sub-directory contains all model JSON and JavaScript files.

Put all your model JSON and JavaScript files in the `/common/models` directory.

/server directory - Node application files		
File or directory	**Description**	**How to access in code**
`server.js`	Main application program file.	N/A
`config.json`	Application settings. See config.json.	`app.get('option-name')`
`datasources.json`	Data source configuration file. See datasources.json.	`app.datasources['datasource-name']`
`model-config.json`	Model configuration file. See model-config.json.	N/A
`/boot` directory	Add scripts to perform initialization and setup. See Boot scripts.	Scripts are automatically executed in alphabetical order.
/client directory - Client application files		
README.md	LoopBack generators create empty README file in markdown format.	N/A
Other	Add your HTML, CSS, client JavaScript files.	
/common directory - shared application files		
`/models` directory	Custom model files: • Model definitions, by convention named *`modelName`*`.json`; for example `customer.json`. • Custom model scripts by convention named *`modelName`*`.js`; for example, `customer.js`. See Model definition JSON file.	Node: `myModel = app.models.myModelName`
Top-level application directory		
package.json	Standard npm package specification. See package.json	N/A

Additionally, the top-level directory contains the stub `README.md` file, and `node_modules` directory (for Node dependencies).

package.json

The `package.json` file is the standard file for npm package management. Use it to set up package dependencies, among other things. This file must be in the application root directory.

For more information, see the package.json documentation.

Your application `package.json` must have an "name" property to monitor the application with StrongOps.

For example:

```
{
  "name": "loopback-example-app",
  "version": "0.0.0",
  "main": "server/server.js",
  "dependencies": {
    "compression": "^1.0.3",
    "errorhandler": "^1.1.1",
    "loopback": "~2.0.0-beta5",
    "loopback-boot": "2.0.0-beta2",
    "loopback-datasource-juggler": "~2.0.0-beta2",
    "function-rate-limit": "~0.0.1",
    "async": "~0.9.0",
    "loopback-connector-rest": "^1.1.4"
  },
  "optionalDependencies": {
    "loopback-explorer": "^1.1.0",
    "loopback-connector-mysql": "^1.2.1",
    "loopback-connector-mongodb": "^1.2.5",
    "loopback-connector-oracle": "^1.2.1"
  },
  "devDependencies": {
    "mocha": "^1.20.1",
    "supertest": "^0.13.0"
  },
  "scripts": {
    "test": "mocha -R spec server/test",
    "pretest": "jshint ."
  }
}
```

server directory

The /server directory contains files that apply only to the backend LoopBack (Node) application.

The standard JSON configuration files in this directory are:

- config.json
- model-config.json
- datasources.json

config.json

Overview

Define application-wide settings in `/server/config.json`. For example:

config.json

```
{
  "host": "localhost",
  "port": 3000,
  "restApiRoot": "/api"
}
```

The following table describes the properties you can configure.

Property	Description	Default
cookieSecret	Unique string to use to identify browser cookies for the application	N/A
host	Host name or IP address to use.	localhost
port	TCP port to use.	3000
restApiRoot	Root URI of REST API	`/api`

To access the settings in application code, use `app.get('option-name')`.

Environment-specific settings

You can override values in `config.json` in:

- `config.local.js` or `config.local.json`
- `config.`*`env`*`.js` or `config.`*`env`*`.json`, where *`env`* is the value of `NODE_ENV` (typically `development` or `production`); so, for example `config.production.json`.

> ⚠ The additional files can override the top-level keys with value-types (strings, numbers) only. Nested objects and arrays are not supported at the moment.

For example:

config.production.js

```
module.exports = {
  host: process.env.CUSTOM_HOST,
  port: process.env.CUSTOM_PORT
};
```

model-config.json

Overview

The file `/server/model-config.json` configures LoopBack models, for example it binds models to data sources and specifies whether a model is exposed over REST. The models referenced in this file must be either a built-in models or custom models defined by a JSON file in the `common/models/` folder.

> ⓘ You can also use a `/client/model-config.json` for client-specific (browser) model configuration.

For example, here is the default `model-config.json` that lists all the built-in models:

model-config.json

```
{
  "_meta": {
    "sources": [
      "../common/models",
      "./models"
    ]
  },
  "User": {
    "dataSource": "db"
  },
  "AccessToken": {
    "dataSource": "db",
    "public": false
  },
  "ACL": {
    "dataSource": "db",
    "public": false
  },
  "RoleMapping": {
    "dataSource": "db",
    "public": false
  },
  "Role": {
    "dataSource": "db",
    "public": false
  }
}
```

Model configuration properties

Property	Type	Description
_meta.sources	Array	Array of relative paths to custom model definitions.
model-name.datasource	String	Name of data source to which the model is connected
model-name.public	Boolean	If true, then the model is exposed over REST. Does not affect accessibility of Node API.

_meta.sources

By default, LoopBack applications load models from `/common/models` subdirectory. To specify a different location (or even multiple locations) use the `_meta.sources` property, whose value is an array of directory paths.

datasources.json

Overview

Configure data sources in `/server/datasources.json`. You can set up as many data sources as you want in this file.

For example:

```
{
  "db": {
    "name": "db",
    "connector": "memory"
  },
  "myDB": {
    "name": "myDB",
    "connector": "mysql",
    "host": "demo.strongloop.com",
    "port": 3306,
    "database": "demo",
    "username": "demo",
    "password": "L00pBack"
  }
}
```

To access data sources in application code, use `app.datasources.`*`datasourceName`*.

Standard properties

All data sources support a few standard properties. Beyond that, specific properties and defaults depend on the connector being used.

Property	Description
connector	LoopBack connector to use; one of: • memory • loopback-connector-oracle or just "oracle" • loopback-connector-mongodb or just "mongodb" • loopback-connector-mysql or just "mysql" • loopback-connector-postgresql or just "postgresql" • loopback-connector-soap or just "soap" • loopback-connector-mssql or just "mssql" • loopback-connector-rest or just "rest" • loopback-storage-service
name	Name of the data source being defined.

Properties for database connectors

Data source properties depend on the specific data source being used. However, data sources for database connectors (Oracle, MySQL, PostgreSQL, MongoDB, and so on) share a common set of properties, as described in the following table.

Property	Description
database	Database name
debug	Boolean. If true, turn on verbose mode to debug database queries and lifecycle.
host	Database host name
password	Password to connect to database
port	Database TCP port
username	Username to connect to database

Environment-specific configuration

You can override values set in `datasources.json` in the following files:

- `datasources.local.js` or `datasources.local.json`
- `datasources.`*`env`*`.js` or `datasources.`*`env`*`.json`, where *env* is the value of `NODE_ENV` environment variable (typically `development` or `production`); for example, `datasources.production.json`.

The additional files can override the top-level data-source options with string and number values only. You cannot use objects or array values.

Example data sources:

datasources.json

```
{
  // the key is the datasource name
  // the value is the config object to pass to
  // app.dataSource(name, config).
  db: {
    connector: 'memory'
  }
}
```

datasources.production.json

```
{
  db: {
    connector: 'mongodb',
    database: 'myapp',
    user: 'myapp',
    password: 'secret'
  }
}
```

common directory

The /common directory contains files shared by the server and client parts of the application. By default, slc loopback creates a `/models` sub-dir ectory with one JSON file per model in the application. See Model definition JSON file for a description of the format of this file.

Put all your model JSON and JavaScript files in the `/common/models` directory.

Model definition JSON file

Related articles

- Creating models
- Extending models
- Creating model relations
- Querying models
- Model definition JSON file
- Model REST API

Overview

Model JSON files are in `/common/models` directory and are named *`model-name`*`.json`, where *`model-name`* is the model name of each model; for example, `customer.json`. The model JSON file defines models, relations between models, and access to models.

Below is an excerpt from an example model definition file for a customer model that would be in `/common/models/customer.json`:

customer.json

```
{
  "name": "Customer",  // See Top-level properties below
  "base": "User",
  "idInjection": false,
  "strict": true,
  "options": { ... }, // See Options below - can also declare as top-level properties
  "properties": { ... }, // See Properties below
  "validations": [...],  // See Validations below
  "relations": {...},  // See Relations below
  "acls": [...],  // See ACLs below
  "scopes": {...},  // See Scopes below
  "indexes" : { ...}, // See Indexes below
  "methods": [...]  // See Methods below - New for LB2.0 - Remoting metadata
}
```

Top-level properties

Properties are required unless otherwise designated.

Property	Type	Description
name	String	Name of the model.
plural	String	Plural form of the model name. **Optional**: Defaults to plural of name property using standard English conventions.
base	String	Name of another model that this model extends. The model will "inherit" properties and methods of the base model.
idInjection	Boolean	Whether to automatically add an id property to the model: • `true`: `id` property is added to the model automatically. This is the default. • `false`: `id` property is not added to the model See Id for more information. **Optional**; default is `true`.
strict	Boolean	Specifies whether the model accepts only predefined properties or not. One of: • `true`: Only properties defined in the model are accepted. Use this mode if you want to make sure only predefined properties are accepted. • `false`: The model will be an open model. All properties are accepted, including the ones that not predefined with the model. This mode is useful if the mobile application just wants to store free form JSON data to a schema-less database such as MongoDB. • Undefined: Default to false unless the data source is backed by a relational database such as Oracle or MySQL.
options	Object	JSON object that specifies model options. See Options below.
properties	Object	JSON object that specifies the properties in the model. See Properties below.
relations	Object	Object containing relation names and relation definitions. See Relations below.
acls	Array	Set of ACL specifications that describes access control for the model. See Acls below.

Options

The `options` key specifies data source-specific options. When a model is attached a data source of certain type such as Oracle or MySQL, you can specify the name of the database schema and table as properties under the key with the name of the connector type.

```
...
  "options": {
    "mysql": {
      "table": "location"
    },
    "mongodb": {
      "collection": "location"
    },
    "oracle": {
      "schema": "BLACKPOOL",
      "table": "LOCATION"
    }
  },
  ...
```

Properties

The properties key defines one or more properties, each of which is an object that has keys described in the following table. Below is an example a basic property definition:

```
...
"properties": {
  "firstName": {
    "type": "String",
    "required": "true"
  },
  "id": {
    "type": "Number",
    "id": true,
    "doc": "User ID"
},
...
```

Key	Required?	Type	Description
doc	No	String	Documentation for the property.
id	No	Boolean	Whether the property is a unique identifier. Default is false. See Id property below.
required	No	Boolean	Whether a value for the property is required. Default is false.
type	Yes	String	Property type. Can be any type described in LoopBack types.
*	No	Various	See below.

ID properties

A model representing data to be persisted in a database usually has one or more *ID properties* that uniquely identify the model instance. For example, the `user` model might have user IDs.

By default, if no ID properties are defined and the `idInjection` property is `true` (or is not set, since `true` is the default), LoopBack automatically adds an `id` property to the model as follows:

```
id: {type: Number, generated: true, id: true}
```

The `generated` property indicates the ID will be automatically generated by the database. If true, the connector decides what type to use for the auto-generated key. For relational databases, such as Oracle or MySQL, it defaults to `number`. If the ID is generated on the client side, set it to false.

To explicitly specify a property as ID, set the `id` property of the option to `true`. The `id` property value must be one of:

- `true`: the property is an ID.
- `false` (or any value that converts to false): the property is not an ID (default).
- Positive number, such as 1 or 2: the property is the index of a composite ID.

In database terms, key column(s) are ID properties. Such properties are defined with the 'id' attribute set to true or a number as the position for a composite key. For example,

```
{
  "myId": {
    "type": "string",
    "id": true
   }
}
```

Then:

1. If a model doesn't have explicitly-defined ID properties, LoopBack automatically injects a property named "id" unless the `idInjection` option is set to false.
2. If an ID property has `generated` set to true, the connector decides what type to use for the auto-generated key. For example for SQL Server, it defaults to `number`.
3. LoopBack CRUD methods expect the model to have an "id" property if the model is backed by a database.
4. A model without any "id" properties can only be used without attaching to a database.

Composite IDs

LoopBack supports the definition of a composite ID that has more than one property. For example:

```
var InventoryDefinition = {
  productId: {type: String, id: 1},
  locationId: {type: String, id: 2},
  qty: Number
}
```

The composite ID is (productId, locationId) for an inventory model.

⚠ Composite IDs are not currently supported as query parameters in REST APIs.

Data mapping properties

When using a relational database data source, you can specify the following properties that describe the columns in the database.

Property	Type	Description
columnName	String	Column name
dataType	String	Data type as defined in the database
dataLength	Number	Data length
dataPrecision	Number	Numeric data precision

dataScale	Number	Numeric data scale
nullable	Boolean	If true, data can be null

For example, to map a property to a column in an Oracle database table, use the following:

```
...
"name": {
      "type": "String",
      "required": false,
      "length": 40,
      "oracle": {
        "columnName": "NAME",
        "dataType": "VARCHAR2",
        "dataLength": 40,
        "nullable": "Y"
      }
    }
...
```

Conversion and formatting properties

Format conversions are declared in properties, as described in the following table:

Key	Type	Description
trim	Boolean	Whether to trim the string
lowercase	Boolean	Whether to convert a string to lowercase
uppercase	Boolean	Whether to convert a string to uppercase
format	Regular expression	Format for a date property.

Validations

> This is not yet implemented. You must currently validate in code; see Validating model data.

Specify constraints on data with `validations` properties. See also Validations class.

Key	Type	Description
default	Any	Default value of the property.
required	Boolean	Whether the property is required.
pattern	String	Regular expression pattern that a string should match
max	Number	Maximum length for string types.
min	Number	Minimum length for string types.
length	Number	Maximum size of a specific type, for example for CHAR types.

For example:

```
"username": {
  "type": "string",
  "doc": “User account name,
  "min": 6,
  "max": 24
}
```

Relations

The `relations` key defines relationships between models through a JSON object. Each key in this object is the name of a related model, and the value is a JSON object as described in the table below. For example:

```
...
      "relations": {
        "accessTokens": {
          "model": "accessToken",
          "type": "hasMany",
          "foreignKey": "userId"
        },
        "account": {
          "model": "account",
          "type": "belongsTo"
        },
        "transactions": {
          "model": "transaction",
          "type": "hasMany"
        }
      },
...
```

Key	Type	Description
model	String	Name of the related model. Required.
type	String	Relation type. Required. See Creating model relations for more information. One of: • hasMany • belongsTo • hasAndBelongsToMany For hasMany, you can also specify a hasManyThrough relation by adding a "through" key: `{through: 'modelName'}` See example below.
foreignKey	String	Optional foreign key used to find related model instances.
through	String	Name of model creating hasManyThrough relation. See example below.

Example of hasManyThrough:

```
"patient": {
    "model": "physician",
    "type": "hasMany",
    "through" : "appointment"
}
```

ACLs

The value of the `acls` key is an array of objects that describes the access controls for the model. Each object has the keys described in the table below.

```
"acls": [
    {
      "permission": "ALLOW",
      "principalType": "ROLE",
      "principalId": "$everyone",
      "property": "myMethod"
    },
    ...
]
```

Key	Type	Description
accessType	String	The type of access to apply. One of: • READ • WRITE • EXECUTE • ALL (default)
permission	String	Type of permission granted. Required. One of: • **ALARM** - Generate an alarm, in a system dependent way, the access specified in the permissions component of the ACL entry. • **ALLOW** - Explicitly grants access to the resource. • **AUDIT** - Log, in a system dependent way, the access specified in the permissions component of the ACL entry. • **DENY** - Explicitly denies access to the resource.

principalId	String	Principal identifier. Required.
		The value must be one of:
		• A user ID (String|number|any)
		• One of the following predefined dynamic roles:
		  ◦ `$everyone` - Everyone
		  ◦ `$owner` - Owner of the object
		  ◦ `$related` - Any user with a relationship to the object
		  ◦ `$authenticated` - Authenticated user
		  ◦ `$unauthenticated` - Unauthenticated user
		• A static role name

principalType	String	Type of the principal. Required. One of: • Application • User • Role
property		Specifies a property/method/relation on a given model. It further constrains where the ACL applies.

Scopes

Scopes enable you to specify commonly-used queries that you can reference as method calls on a model.

The `scopes` key defines one or more scopes (named queries) for models. A scope maps a name to a predefined filter object to be used by the model's `find()` method; for example:

```
"scopes": {
 "vips": {"where": {"vip": true}},
 "top5": {"limit": 5, "order": "age"}
}
```

The snippet above defines two named queries for the model:

- vips: Find all model instances with vip flag set to true
- top5: Find top five model instances ordered by age

Within the scopes object, the keys are the names, and each value defines a filter object for Model.find().

You can also define a scope programmatically using a model's `scope()` method, for example:

```
User.scope('vips', {where: {vip: true}});
User.scope('top5': {limit: 5, order: 'age'});
```

Now you can call the methods defined by the scopes; for example:

```
User.vips(function(err, vips) {
...
});
```

Methods

You can declare remote methods here. Until this feature is implemented, you must declare remote methods in code; see Defining remote methods.

> This feature is not yet implemented.

Indexes

Indexes can be declared for a model as the 'indexes' option:

```
"indexes": {
  "name_age_index": {
    "keys": {"name": 1, "age": -1}
  },
  "age_index": {"age": -1}
}
```

The snippet above creates two indexes for the declaring model:

- A composite index named 'name_age_index' with two keys: 'name' in ascending order and 'age' in descending order
- A simple index named 'age_index' with one key: 'age' in descending order

The full syntax for an index within the 'indexes' is:

```
"<indexName>": {
  "keys": {
     "<key1>": 1,
     "<key2>": -1
   },
   "options": {
     "unique": true
   }
}
```

Note that '1' specifies 'ascending' while '-1' specifies 'descending'. If no 'options' are needed, you can use a shortened form:

```
"<indexName>": {
  "<key1>": 1,
  "<key2>": -1
}
```

Indexes can be marked at model property level too, for example:

```
{
  "name": { "type": "String", "index": true },
  "email": { "type": "String", "index": {"unique": true} },
  "age": "Number"
}
```

Two indexes will be created, one for the 'name' key and another one for the 'email' key. The 'email' index is unique.

Extending a model

```
var properties = {
  firstName: {type: String, required: true}
};

var options = {
  relations: {
    accessTokens: {
      model: accessToken,
      type: hasMany,
      foreignKey: userId
    },
    account: {
      model: account,
      type: belongsTo
    },
    transactions: {
      model: transaction,
      type: hasMany
    }
  },
  acls: [
    {
      permission: ALLOW,
      principalType: ROLE,
      principalId: $everyone,
      property: myMethod
    }
  ]
};

var user = loopback.Model.extend('user', properties, options);
```

client directory

The `/client` directory is where you put client JavaScript, HTML, and CSS files.

Currently, slc loopback does not generate any files in this directory, except for a stub `README.md`.

For information on creating a client LoopBack application, see LoopBack in the client.

Creating a LoopBack application

- Creating a new application
  - Run empty application
  - Create models, properties, and data sources

⚠ Prerequisite: You must install StrongLoop software before you start. See Getting Started with LoopBack for instructions.

Creating a new application

Enter this command to create a new blank template LoopBack application:

```
$ slc loopback
```

The tool will prompt you for the name of your app: enter "myapp". This command creates a new directory called `myapp` (or the name you used)

that contains all the files and directories to provide "scaffolding" for a LoopBack application.

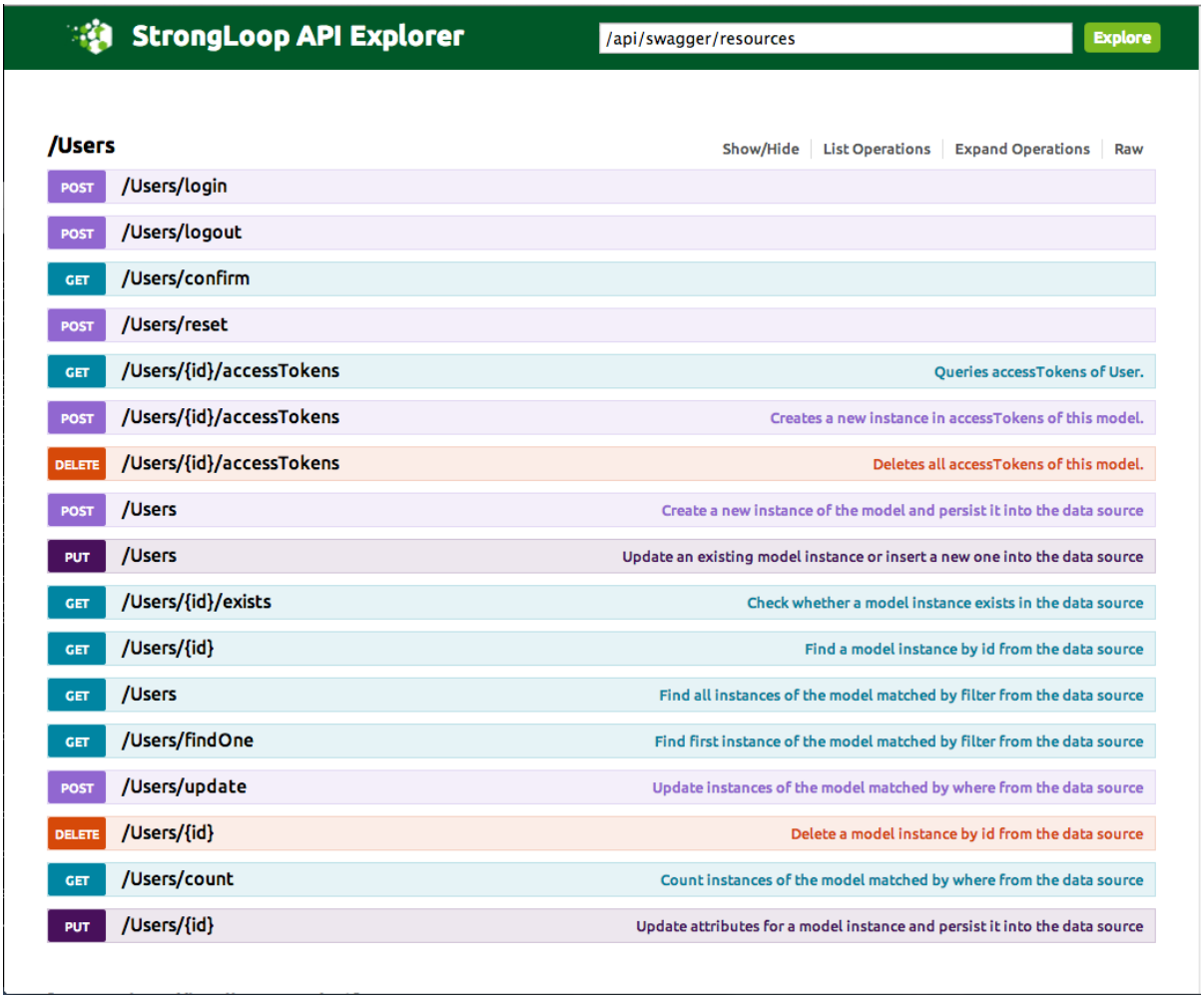
Run empty application

At this point, you have an "empty" LoopBack application. It won't actually do much, but you can run it to get a sense of what LoopBack scaffolding provides. Enter these commands:

```
$ cd myapp
$ slc run
```

Of course, substitute your application name for "myapp."

Now load this URL in your browser to view the API explorer: http://0.0.0.0:3000/explorer. Click on the **/Users** link and the API explorer will show you the REST APIs of the built-in user model.



Create models, properties, and data sources

Use the `slc loopback` command-line tool to scaffold your application. Follow the Getting Started with LoopBack tutorial or refer to the LoopBack generators documentation.

Application initialization

This document describes the configuration for LoopBack `2.x`.

See Migrating existing apps to version 2.0 for information on upgrading existing projects.

Version `1.x` of loopback-boot is backwards compatible with `app.boot` provided by LoopBack 1.x versions and the project layout scaffolded by `slc lb project` up to `slc` version 2.5.

Overview

Application initialization (also called *bootstrapping*):

- Configures data-sources.
- Defines custom models
- Configures models and attaches models to data-sources.
- Configures application settings
- Runs additional boot scripts, so you can put custom setup code in multiple small files instead of in the main application file.

The loopback-boot module performs initialization/bootstrapping.

Project files and directories

⚠ The following describes the application structure as created by the `slc loopback` command. LoopBack does not require that you follow this structure, but if you don't, then you can't use `slc loopback` commands to modify or extend your application.

LoopBack project files and directories are in the *application root directory*. Within this directory the standard LoopBack project structure has three sub-directories:

- `server` - Node application scripts and configuration files.
- `client` - Client JavaScript, HTML, and CSS files.
- `common` - Files common to client and server. The `/models` sub-directory contains all model JSON and JavaScript files.

ⓘ Put all your model JSON and JavaScript files in the `/common/models` directory.

/server directory - Node application files		
File or directory	**Description**	**How to access in code**
`server.js`	Main application program file.	N/A
`config.json`	Application settings. See config.json.	`app.get('option-name')`
`datasources.json`	Data source configuration file. See datasources.json.	`app.datasources['datasource-name']`
`model-config.json`	Model configuration file. See model-config.json.	N/A
`/boot` directory	Add scripts to perform initialization and setup. See Boot scripts.	Scripts are automatically executed in alphabetical order.
/client directory - Client application files		
README.md	LoopBack generators create empty README file in markdown format.	N/A
Other	Add your HTML, CSS, client JavaScript files.	
/common directory - shared application files		
`/models` directory	Custom model files: • Model definitions, by convention named *`modelName`*`.json`; for example `customer.json`. • Custom model scripts by convention named *`modelName`*`.js`; for example, `customer.js`. See Model definition JSON file.	Node: `myModel = app.models.myModelName`
Top-level application directory		
package.json	Standard npm package specification. See package.json	N/A

Additionally, the top-level directory contains the stub `README.md` file, and `node_modules` directory (for Node dependencies).

Bootstrapping an application

To invoke the LoopBack bootstrap module:

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

var app = loopback();
boot(app, __dirname);

app.use(loopback.rest());
app.listen();
```

See API docs for details.

Boot scripts

When data sources and models are configured, LoopBack invokes all scripts in the `/boot` folder. The scripts are sorted alphabetically ignoring case. Use boot scripts to perform other initialization.

Example boot script

boot/authentication.js

```
module.exports = function(app) {
  app.enableAuth();
};
```

Environment-specific configuration

Overview

You can set up configurations for specific environments (for example, development, staging, and production) using config.json and datasources.json (and their corresponding JavaScript files).

Example

For an example, see https://github.com/strongloop/loopback-example-full-stack/tree/master/server.

For application configuration:

- config.json
- config.local.js

For data source configuration:

- datasources.json
- datasources.production.js
- datasources.staging.js

Application-wide configuration

You can override values in `config.json` in:

- `config.local.js` or `config.local.json`
- `config.*env*.js` or `config.*env*.json`, where *env* is the value of `NODE_ENV` (typically `development` or `production`); so, for example `config.production.json`.

> ⚠ The additional files can override the top-level keys with value-types (strings, numbers) only. Nested objects and arrays are not supported at the moment.

For example:

config.production.js

```
module.exports = {
  host: process.env.CUSTOM_HOST,
  port: process.env.CUSTOM_PORT
};
```

Data source configuration

You can override values set in `datasources.json` in the following files:

- `datasources.local.js` or `datasources.local.json`
- `datasources.*env*.js` or `datasources.*env*.json`, where *env* is the value of `NODE_ENV` environment variable (typically `development` or `production`); for example, `datasources.production.json`.

> ⚠ The additional files can override the top-level data-source options with string and number values only. You cannot use objects or array values.

Example data sources:

datasources.json

```
{
  // the key is the datasource name
  // the value is the config object to pass to
  // app.dataSource(name, config).
  db: {
    connector: 'memory'
  }
}
```

datasources.production.json

```
{
  db: {
    connector: 'mongodb',
    database: 'myapp',
    user: 'myapp',
    password: 'secret'
  }
}
```

Getting values from environment variables

You can easily set an environment variable when you run an application like this:

```
$ MY_CUSTOM_VAR="some value" slc run
```

or

```
$ export MY_CUSTOM_VAR="some value"
$ slc run
```

Then this variable is available to your application as `process.env.MY_CUSTOM_VAR`.

Debugging LoopBack apps

Using debug strings

The LoopBack framework has a number of built-in debug strings to help with debugging. Specify a string on the command-line via an environment variable as follows:

$ DEBUG=<*pattern*>[,<*pattern*>...] slc run

where <*pattern*> is a string-matching pattern specifying debug strings to match. You can specify as many matching patterns as you wish.

For example:

```
$ DEBUG=loopback:datasource slc run
```

You'll see output such as (truncated for brevity):

```
loopback:datasource Settings: {"name":"db","debug":true} +0ms
loopback:datasource Settings: {"name":"geo","connector":"rest",...
```

You can use an asterisk (*) in the pattern to match any string. For example the following would match any debug string containing "oracle":

```
$ DEBUG=*oracle slc run
```

You can also exclude specific debuggers by prefixing them with a "-" character. For example, `DEBUG=*,-strong-remoting:*` would include all debuggers except those starting with "strong-remoting:".

Debug string format

These strings have the format

module[:*area*]:*fileName*

Where

- *module* is the name of the module, for example `loopback` or `loopback-connector-rest`.

- *area* is an optional identifier such as `security` or `connector` to identify the purpose of the module
- *fileName* is the name of the JavaScript source file, such as `oracle.js`.

For example

`loopback:security:access-context`

identifies the source file `access-context.js` in the `loopback` module (used for security features).

Debug strings reference

Module / Source file	String
loopback	
loopback/lib/connectors/base-connector.js	connector
loopback/lib/connectors/mail.js	loopback:connector:mail
loopback/lib/connectors/memory.js	memory
loopback/lib/models/access-context.js	loopback:security:access-context
loopback/lib/models/acl.js	loopback:security:acl
loopback/lib/models/change.js	loopback:change
loopback/lib/models/role.js	loopback:security:role
loopback/lib/models/user.js	loopback:user
loopback-datasource-juggler	
loopback-datasource-juggler/lib/datasource.js	loopback:datasource
loopback-boot	
loopback-boot/lib/compiler.js	loopback:boot:compiler
loopback-boot/lib/executor.js	loopback:boot:executor
Components	
loopback-component-push/lib/providers/apns.js	loopback:component:push:provider:apns
loopback-component-push/lib/providers/gcm.js	loopback:component:push:provider:gcm
loopback-component-push/lib/push-manager.js	loopback:component:push:push-manager
Connectors	
loopback-connector-mongodb/lib/mongodb.js	loopback:connector:mongodb
loopback-connector-mssql/example/datagraph.js	datagraph
loopback-connector-mssql/lib/mssql.js	loopback:connector:mssql
loopback-connector-mysql/lib/mysql.js	loopback:connector:mysql
loopback-connector-oracle/lib/oracle.js	loopback:connector:oracle
loopback-connector-postgresql/lib/postgresql.js	loopback:connector:postgresql
loopback-connector-rest/lib/rest-builder.js	loopback:connector:rest
loopback-connector-rest/lib/rest-connector.js	loopback:connector:rest
loopback-connector-rest/lib/rest-model.js	loopback:connector:rest
loopback-connector-rest/lib/swagger-client.js	loopback:connector:rest:swagger
loopback-connector-soap/lib/soap-connector.js	loopback:connector:soap
strong-remoting	

strong-remoting/lib/dynamic.js	strong-remoting:dynamic
strong-remoting/lib/exports-helper.js	strong-remoting:exports-helper
strong-remoting/lib/http-context.js	strong-remoting:http-context
strong-remoting/lib/http-invocation.js	strong-remoting:http-invocation
strong-remoting/lib/jsonrpc-adapter.js	strong-remoting:jsonrpc-adapter
strong-remoting/lib/remote-objects.js	strong-remoting:remotes
strong-remoting/lib/rest-adapter.js	strong-remoting:rest-adapter
strong-remoting/lib/shared-class.js	strong-remoting:shared-class
strong-remoting/lib/shared-method.js	strong-remoting:shared-method
strong-remoting/lib/socket-io-adapter.js	strong-remoting:socket-io-adapter
strong-remoting/lib/socket-io-context.js	strong-remoting:socket-io-context
loopback-explorer	
loopback-explorer/lib/route-helper.js	loopback:explorer:routeHelpers
loopback-workspace	
loopback-workspace/connector.js	workspace:connector
loopback-workspace/connector.js	workspace:connector:save-sync
loopback-workspace/models/config-file.js	workspace:config-file
loopback-workspace/models/definition.js	workspace:definition
loopback-workspace/models/facet.js	workspace:facet
loopback-workspace/models/facet.js:	var workspace:facet:load: + facetName
loopback-workspace/models/facet.js:	var workspace:facet:save: + facetName
loopback-workspace/models/workspace.js	workspace

Tutorials and examples

If you're new to LoopBack, the best place to start is Getting Started with LoopBack.

The documentation provides a number of tutorials:


Tutorial: push notifications - Android client


Tutorial: push notifications - iOS client


Access control tutorial


Tutorial: Push notifications


Tutorial: push notifications - LoopBack app


REST example - creating the back-end


Tutorial: push notifications - putting it all together


Creating a LoopBack iOS app: part two


Creating a LoopBack iOS app: part one

See also:

- LoopBack example app
- LoopBack examples in GitHub
- LoopBack blog posts

LoopBack example app

This section will get you up and running with LoopBack and the LoopBack example app in just a few minutes.

Prerequisites

You must first install StrongLoop software as described in Getting Started with LoopBack.

If you don't have a C compiler installed, you will see error messages when you create or run the example application. You'll still be able to run it, but you won't be able to connect to Oracle or view certain strong agent metrics. See Installing compiler tools for more information.

Creating the example app

You can create the LoopBack example app:

- By cloning it from GitHub:

```
$ git clone loopback-example-app
```

- By using `slc loopback:example`, as described below. This is the LoopBack example generator that uses Yeoman under the hood.

Creating the example app using slc

Follow these steps:

1. Create the example app with this command:

```
$ slc loopback:example
```

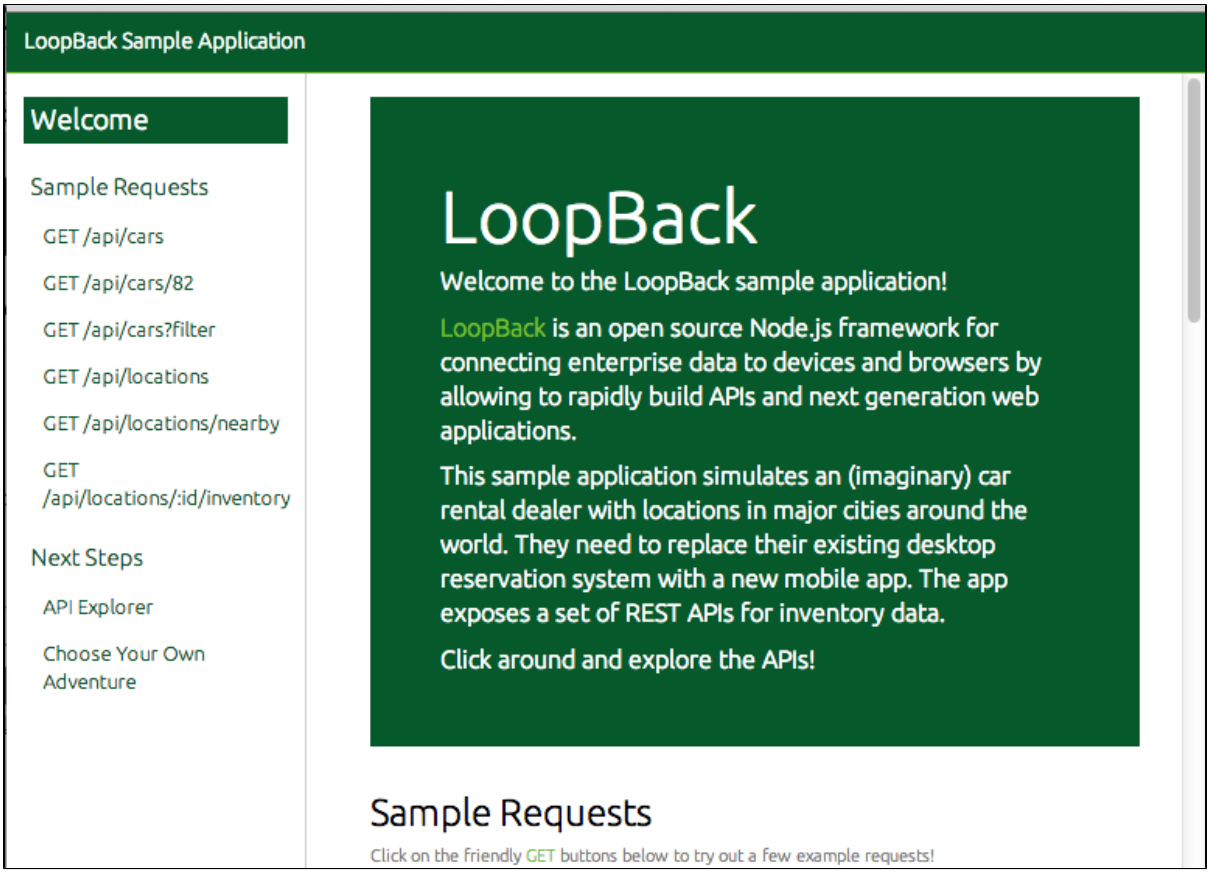
   You'll be prompted for the name of the directory in which to create the application. The default is the current directory.

```
[?] Enter a directory name where to create the project: (.) my-lb-example
```

   For example, suppose you create the app in `my-lb-example`.
2. Run the example application by entering these commands:

```
$ cd my-lb-example
$ slc run
```

3. To view the application in a browser, load http://localhost:3000. The homepage (illustrated below) lists sample requests you can make against the LoopBack REST API. Click the GET buttons to see the JSON data returned. You can also see the API explorer at http://localhost:3000/explorer.



About the example app

The StrongLoop example is a mobile app for i-Cars, an (imaginary) car rental dealer with outlets in major cities around the world.

The application enables customers to find the closest available cars using the i-Car app on a smartphone. The app shows a map of nearby rental locations and lists available cars in the area shown on the map. In addition, the customer can filter the list of cars by make, model, class, year and color. The customer can then select the desired car and reserve it via the app. If not logged in the app prompts the customer to login or register. The app indicates if the desired car is available and if so, confirms the reservation.

Note that the example app is the backend functionality only; that is, the app has a REST API, but no client app or UI to consume the interface.

For more details on the example app, see loopback-example-app in GitHub.

Connecting to other data sources

By default, the LoopBack example app connects to the in-memory data source. To connect to other data sources, use the following command to run the application:

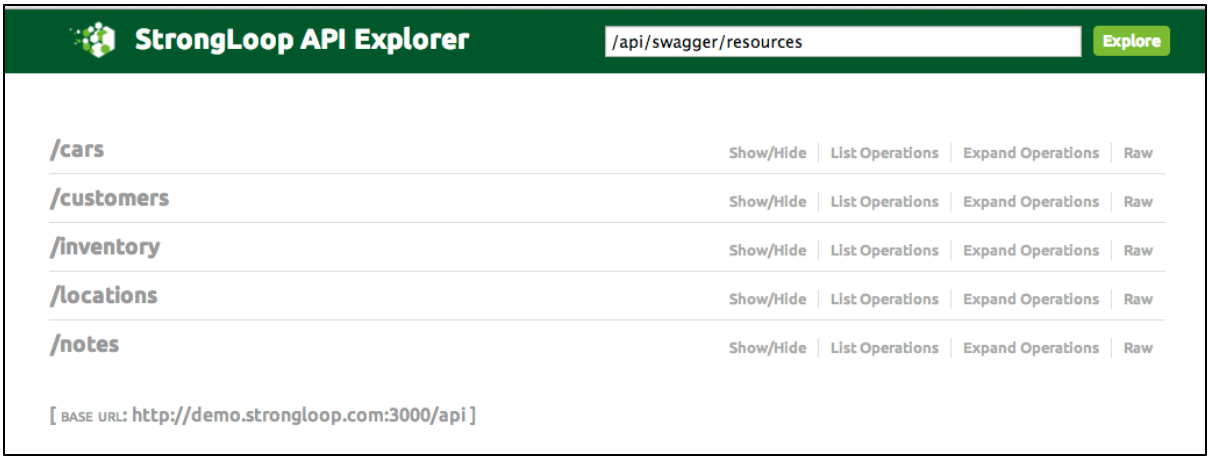
```
$ DB=<datasource> slc run
```

where `<datasource>` is either "mongodb", "mysql", or "oracle". The example app will connect to the specified database running on demo.strongloop.com.

Using the API Explorer

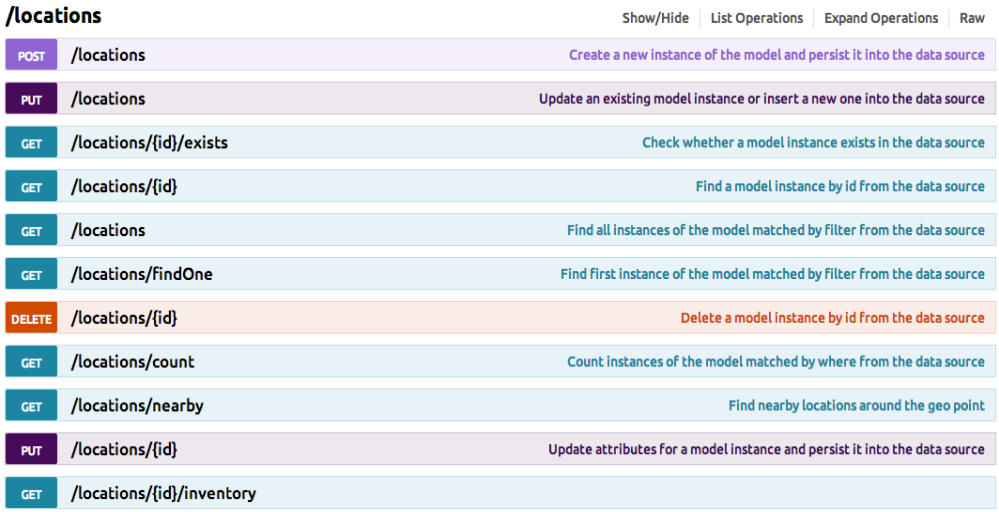
Follow these steps to explore the example app's REST API:

1. Open your browser to http://localhost:3000/explorer. You'll see a list of REST API endpoints as illustrated below.

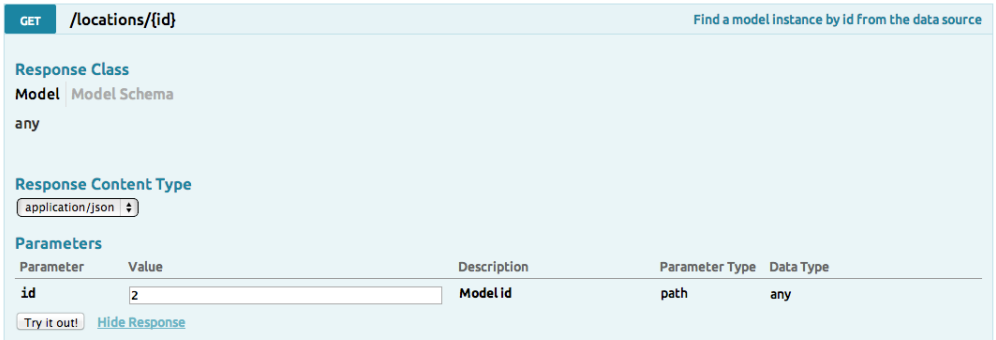


The endpoints are grouped by the model names. Each endpoint consists of a list of operations for the model.

2. Click on one of the endpoint paths (such as /locations) to see available operations for a given model. You'll see the CRUD operations mapped to HTTP verbs and paths.



3. Click on a given operation to see the signature; for example, as shown below for `GET /locations/{id}`. Notice that each operation has the HTTP verb, path, description, response model, and a list of request parameters.



4. Now, invoke the GET locations operation: click the **Try it out!** button. Just leave the filter field blank, to get all the locations. You'll see something like this:

You can see the request URL, the JSON in the response body, and the HTTP response code and headers.

Next steps

If you haven't gone through Getting Started with LoopBack, that's a good place to start.

To gain a deeper understanding of LoopBack and how it works, read Working with models and Data sources and connectors.

LoopBack examples in GitHub

⚠ Some of these examples are LoopBack 1.x applications. We're in the process of migrating them to 2.0.

The iCars application

The iCars application is a comprehensive LoopBack example app.

- iCars client - client example for iCars.
- iCars server - LoopBack backend for iCars

Examples using LoopBack connectors

- Database example application - Examples to demonstrate LoopBack database connectors for MySQL, MongoDB, Oracle, and PostgreSQL.

Examples using LoopBack client SDKs

- Android getting started app - Example using Android SDK
- iOS example apps - Examples using iOS SDK
- iOS books example app - Used in Creating a LoopBack iOS app: part one and Creating a LoopBack iOS app: part two
- loopback-mobile-getting-started - Basic example app with both iOs and Android front-ends.

Other examples

- loopback-example-full-stack - An example running LoopBack in the browser and server
- loopback-example-office-supplies - An introductory LoopBack example app
- example-generators - Generator examples
- loopback-example-access-control - Example demonstrating LoopBack's Access Control features
- loopback-example-todo - Basic todo example using LoopBack and the AngularJS SDK
- loopback-example-proxy - Example demonstrating how to use LoopBack as a gateway / proxy
- loopback-example-remote-methods - LoopBack example demonstrating remote methods.
- loopback-example-passport - Example of using third-party login with loopback-component-passportmodule.
- loopback-example-ssl - demonstrates how to set up SSL for LoopBack applications so that the REST APIs can be called using HTTPS.

Community examples

- https://github.com/strongloop-community - contains many other example applications

LoopBack blog posts

The StrongLoop blog contains lots of helpful posts relevant to LoopBack. We try to capture most of the information in the documentation as well, but you may find some helpful tips and examples in the blog.

Modeling

- Defining and Mapping Data Relations with LoopBack Connected Models
- Recipes for LoopBack Models Part 1: Open Models
- Recipes for LoopBack Models Part 2: Models with Schema Definitions
- Recipes for LoopBack Models Part 3: Model Discovery with Relational Databases
- Recipes for LoopBack Models Part 4: Models by Instance Introspection
- Recipes for LoopBack Models Part 5: Model Synchronization with Relational Databases

Connectors

- Turn SOAP into REST APIs with LoopBack
- Managing Objects in LoopBack with the Storage Provider of Your Choice
- Getting Started with the MySQL Connector for LoopBack
- Getting Started with the PostgreSQL Connector for LoopBack

Client SDKs

- Announcing the AngularJS SDK for LoopBack
- Getting Started with iOS Push Notifications in LoopBack
- Getting Started with Android Push Notifications in LoopBack
- How-to Build CRUD Enabled iOS Apps with the LoopBack API Server – Part 1 and Part 2
- Announcing LoopBack Android SDK

Cloud providers

- Install Node.js monitoring and a mobile backend on Nodejitsu
- Installing Node.js Monitoring and an API Server on Elastic Beanstalk
- Install Node.js, LoopBack and StrongOps on Digital Ocean
- Deploying LoopBack mBaaS on Rackspace
- Getting Started with StrongLoop on Cloud9 IDE

Working with models

A LoopBack model represents data in backend systems such as databases, and by default has both Node and REST APIs.

LoopBack supports both "dynamic" schema-less models for data sources such as SOAP and REST services, and "static", schema-driven models, for database-backed data sources. For data sources backed by a relational database, a model typically

corresponds to a table. Additionally, a model is where you add functionality such as validation rules and business logic.

Related articles

- Creating models
- Extending models
- Creating model relations
- Querying models
- Model definition JSON file
- Model REST API

Creating models

Related articles

- Creating models
- Extending models
- Creating model relations
- Querying models
- Model definition JSON file
- Model REST API

Overview

You can create LoopBack models in various ways, depending on what kind of data source the model is based on. You can create:

- *Dynamic models*, for free-form data.
  - You can also create dynamic models by instance introspection based on JSON data from NoSQL databases or REST APIs.
- *Static models*, for models with schema definitions, such as those based on relational databases.
  - You can create static models using LoopBack's discovery API by consuming existing data from a relational database.
  - Then you can keep your model synchronized with the database using LoopBack's schema / model synchronization API.

Using the LoopBack model generator

The easiest way to create a model is with the LoopBack model generator.

Follow the example below to create an example model called "books" that represents a book database, or create your own model if you prefer, using fields appropriate for your application.

Enter the following command to create a new model using the LoopBack model generator.

```
$ slc loopback:model book
```

You'll be prompted to choose the data source that the model will connect to. By default, there will be only the in-memory data source (named "db"). When you create additional data sources, they will be listed as options.

```
[?] Select the data-source to attach book to: (Use arrow keys)
 db
```

Simply press RETURN here to use the in-memory data source.

Next, the generator will ask whether to expose the you'll be asked

```
[?] Expose book via the REST API? (Y/n)
```

Adding properties

Add properties to a model with the LoopBack property generator.

```
$ slc loopback:property
```

Following the example, you can press RETURN to accept the default (string) for all the properties except `totalPages`; for that, choose `number`.

The final prompt is:

```
Done defining model book (books).
  - title (string)
  - author (string)
  - description (string)
  - totalPages (number)
  - genre (string)
Create this model? (yes):
```

Press RETURN again to create the model.

Discovering models from databases

Overview

LoopBack makes it surprisingly simple to create models from an existing database. This process is called *discovery*.

In addition to the asynchronous APIs, `DataSource` also provides the synchronous ones. For more information, see Database discovery API.

Example discovery

For example, consider 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 the `associations: true` option 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 ------------- ");
    });
  });
});
```

Additional discovery functions

Some connectors provide discovery capability so that we can use DataSource to discover model definitions from existing database schema. The following APIs enable UI or code to discover database schema definitions that can be used to build LoopBack models.

```
// List database tables and/or views
ds.discoverModelDefinitions({views: true, limit: 20}, cb);

// List database columns for a given table/view
ds.discoverModelProperties('PRODUCT', cb);
ds.discoverModelProperties('INVENTORY_VIEW', {owner: 'STRONGLOOP'}, cb);

// List primary keys for a given table
ds.discoverPrimaryKeys('INVENTORY',  cb);

// List foreign keys for a given table
ds.discoverForeignKeys('INVENTORY',  cb);

// List foreign keys that reference the primary key of the given table
ds.discoverExportedForeignKeys('PRODUCT',  cb);

// Create a model definition by discovering the given table
ds.discoverSchema(table, {owner: 'STRONGLOOP'}, cb);
```

Database discovery API

Overview

LoopBack provides a unified API to discover model definition information from relational databases. The same discovery API is available when using any of these connectors:

- **Oracle**: loopback-connector-oracle
- **MySQL**: loopback-connector-mysql
- **PostgreSQL**: loopback-connector-postgresql
- **SQL Server**: loopback-connector-mssql

Methods

The methods described below are asynchronous. For Oracle, there are also corresponding synchronous methods that accomplish the same things and return the same results:

- `discoverModelDefinitionsSync(options)`
- `discoverModelPropertiesSync(table, options)`
- `discoverPrimaryKeysSync(table, options)`
- `discoverForeignKeysSync(table, options)`
- `discoverExportedForeignKeysSync(table, options)`

Note there are performance implications in using synchronous methods.

discoverModelDefinitions

Call `discoverModelDefinitions()` to discover model definitions (table or collection names), based on tables or collections in a data source.

```
```

```
discoverModelDefinitions(options, cb)
```

Parameter	Description
options	Object with properties described below.
cb	Get a list of table/view names; see example below.

Properties of options parameter:

Property	Type	Description
all	Boolean	If true, include tables/views from all schemas/owners
owner/schema	String	Schema/owner name
views	Boolean	If true, include views.

Example of callback function return value:

```
{type: 'table', name: 'INVENTORY', owner: 'STRONGLOOP' }
{type: 'table', name: 'LOCATION', owner: 'STRONGLOOP' }
{type: 'view', name: 'INVENTORY_VIEW', owner: 'STRONGLOOP' }
```

Example

For example:

```
datasource.discoverModelDefinitions(function (err, models) {
  models.forEach(function (def) {
    // def.name ~ the model name
    datasource.discoverSchema(null, def.name, function (err, schema) {
      console.log(schema);
    });
  });
});
```

discoverModelProperties

Call `discoverModelProperties()` to discover metadata on columns (properties) of a database table.

```
discoverModelProperties(table, options, cb)
```

Parameter	Description
table	The name of a table or view
options	Options object that can have only the "owner/schema" property to specify the owner or schema name.
cb	Callback function to return a list of model property definitions; see example below.

Example return value of callback function:

```
{ owner: 'STRONGLOOP',
        tableName: 'PRODUCT',
        columnName: 'ID',
        dataType: 'VARCHAR2',
        dataLength: 20,
        nullable: 'N',
        type: 'String' }
      { owner: 'STRONGLOOP',
        tableName: 'PRODUCT',
        columnName: 'NAME',
        dataType: 'VARCHAR2',
        dataLength: 64,
        nullable: 'Y',
        type: 'String' }
```

discoverPrimaryKeys

Call `discoverPrimaryKeys()` to discover primary key definitions in a database.

```
discoverPrimaryKeys(table, options, cb)
```

Parameter	Description
table	The name of a table or view
options	Options object that can have only the "owner/schema" property to specify the owner or schema name.
cb	Callback function to return a list of model property definitions; see example below.

Example return value of callback function:

```
{
  { owner: 'STRONGLOOP',
    tableName: 'INVENTORY',
    columnName: 'PRODUCT_ID',
    keySeq: 1,
    pkName: 'ID_PK' },
  { owner: 'STRONGLOOP',
    tableName: 'INVENTORY',
    columnName: 'LOCATION_ID',
    keySeq: 2,
    pkName: 'ID_PK' },
...
}
```

discoverForeignKeys

Call `discoverForeignKeys()` to discover foreign key definitions from a database.

```
discoverForeignKeys(table, options, cb)
```

Parameter	Description

table	The name of a table or view
options	Options object that can have only the "owner/schema" property to specify the owner or schema name.
cb	Callback function to return a list of model property definitions; see example below.

Example return value of callback function:

```
{ fkOwner: 'STRONGLOOP',
      fkName: 'PRODUCT_FK',
      fkTableName: 'INVENTORY',
      fkColumnName: 'PRODUCT_ID',
      keySeq: 1,
      pkOwner: 'STRONGLOOP',
      pkName: 'PRODUCT_PK',
      pkTableName: 'PRODUCT',
      pkColumnName: 'ID' }
```

discoverExportedForeignKeys

Call `discoverExportedForeignKeys()` to discover foreign key definitions that are exported from a database.

```
discoverExportedForeignKeys(table, options, cb)
```

Parameter	Description
table	The name of a table or view
options	Options object that can have only the "owner/schema" property to specify the owner or schema name.
cb	Callback function to return a list of model property definitions; see example below.

Example return value of callback function:

```
{ fkName: 'PRODUCT_FK',
      fkOwner: 'STRONGLOOP',
      fkTableName: 'INVENTORY',
      fkColumnName: 'PRODUCT_ID',
      keySeq: 1,
      pkName: 'PRODUCT_PK',
      pkOwner: 'STRONGLOOP',
      pkTableName: 'PRODUCT',
      pkColumnName: 'ID' }
```

discoverSchema

Use `discoverSchema` to discover LDL models from a database.

```
discoverSchema(modelName, options, cb)
```

Properties of options parameter:

Property	Type	Description

modelName	String	Name of model to define
options	Object	
cb	Function	Callback function

Example

```
dataSource.discoverSchema('INVENTORY', {owner: 'STRONGLOOP'}, function (err, schema) {
    ...
}
```

The result is shown below.

The result below is an example for MySQL that contains MySQL-specific properties in addition to the regular LDL model options and properties. The 'mysql' objects contain the MySQL-specific mappings. For other databases, the key 'mysql' would be replaced by the database type, for example 'oracle', and the data type mappings would be different.


```
{
  "name":"Inventory",
  "options":{
    "idInjection":false,
    "mysql":{
      "schema":"STRONGLOOP",
      "table":"INVENTORY"
    }
  },
  "properties":{
    "productId":{
      "type":"String",
      "required":false,
      "length":60,
      "precision":null,
      "scale":null,
      "id":1,
      "mysql":{
        "columnName":"PRODUCT_ID",
        "dataType":"varchar",
        "dataLength":60,
        "dataPrecision":null,
        "dataScale":null,
        "nullable":"NO"
      }
    },
    "locationId":{
      "type":"String",
      "required":false,
      "length":60,
      "precision":null,
      "scale":null,
      "id":2,
      "mysql":{
        "columnName":"LOCATION_ID",
        "dataType":"varchar",
        "dataLength":60,
        "dataPrecision":null,
        "dataScale":null,
```

```
      "nullable":"NO"
    }
  },
  "available":{
    "type":"Number",
    "required":false,
    "length":null,
    "precision":10,
    "scale":0,
    "mysql":{
      "columnName":"AVAILABLE",
      "dataType":"int",
      "dataLength":null,
      "dataPrecision":10,
      "dataScale":0,
      "nullable":"YES"
    }
  },
  "total":{
    "type":"Number",
    "required":false,
    "length":null,
    "precision":10,
    "scale":0,
    "mysql":{
      "columnName":"TOTAL",
      "dataType":"int",
      "dataLength":null,
      "dataPrecision":10,
      "dataScale":0,
      "nullable":"YES"
    }
```

```
    }
  }
}
```

Example of building models via discovery

The following example uses `discoverAndBuildModels` to discover, build and try the models.

Note that the string arguments to this function are **case-sensitive**; specifically the table name (in the example below, 'account') and the owner (schema) name (in the example below, 'demo').

```
dataSource.discoverAndBuildModels('account', {owner: 'demo'}, function (err, models) {
    models.Account.find(function (err, act) {
        if (err) {
            console.error(err);
        } else {
            console.log(act);
        }
        dataSource.disconnect();
    });
});
```

Schema / model synchronization

Once you have defined a model, LoopBack can create or update (synchronize) the database schemas accordingly, if you need to adjust the database to match the models. 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

Here’s an example of auto-migration. Consider this model definition:

```
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 using `autoMigrate()`:

```
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 suppose you 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 you run auto-migrate again, the table will be dropped and data will be lost. To avoid this problem use `auto-update()`. Instead of dropping tables and recreating them, `autoupdate()` 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.

For example:

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

To check if any database changes are required, use the `isActual()` method. It accepts and a `callback` argument, which receives a Boolean value depending on database state:

- False if the database structure outdated
- True when dataSource and database is in sync

```
dataSource.isActual(models, function(err, actual) {
    if (!actual) {
        dataSource.autoupdate(models, function(err, result) {
            ...
        });
    }
});
```

Creating models by instance introspection

When the data does not have a schema, you can create models using JSON documents from REST services and NoSQL databases; for 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());
  });
});
```

Extending models

You can make a model extend or "inherit from" an existing model, either one of the built-in models such as User, or a custom model you've defined in your application.

- Extending a model with JSON
- Extending a model programmatically

Extending a model with JSON

If you create your model using slc loopback, then extending a model is easy:

1. You must first use the LoopBack model generator (`slc loopback:model` command) to create a new model in your application. This will create a JSON file in the `/common/models` directory; for example for a customer model, `customer.json`.
2. Edit this file and set the "base" property in the JSON file to the name of the model you want to extend: either one of the built-in models, or one of the custom models you've defined in the application.

For example, here is an excerpt from the `customer.json` file from loopback-example-app that extends the built-in User model to define a new Customer model:

customer.json

```
{

  "name": "Customer",
  "base": "User",
  "idInjection": false,
...
```

In general, you can extend any model this way, not just the built-in models.

> ⚠ Currently you cannot modify a built-in model's required properties. If you need to do this, then create your own custom model as a replacement instead.

You can create custom models that extend from a single base csutom model. For example, to define a model called `MyModel` that extends from a custom model you defined called `mMyBaseModel`, create MyModel using `slc loopback:model`, then edit the JSON file `common/models/MyModel.json` as follows:

/common/models/MyModel.json

```
{
  "name": "Example",
  "base": "MyBaseModel",
}
```

Extending a model programmatically

> ⚠ To extend a model programmatically, create a JavaScript file with **exactly the same base name** as the model JSON file. For example, if your customer model is defined in `customer.json`, then create `customer.js`. Both of these files must be in the `/common/models` directory. See Project layout reference for more information.

Extend a model programmatically with the `extend()` method. The method's signature is:

newModel = *modeName*.extend('*modelName*', *properties* [, *settings*]);

Where:

- *newModel* is the name of the new model you're defining.
- *modelName* is the name of the model you're extending.
- *properties* is a JSON object defining the properties to add to the model.
- *settings* is an optional JSON object defining other options such as relations, ACLs, and so on.

Here's an example of extending the User model to create a new Customer model:

```
var Customer = User.extend('customer', {
  accountId: String,
  vip: Boolean
});
```

Attaching models to data sources

Overview

A data source enables a model to access and modify data in backend system such as a relational database. Data sources encapsulate business logic to exchange data between models and various back-end systems such as relational databases, REST APIs, SOAP web services, storage services, and so on. Data sources generally provide create, retrieve, update, and delete (CRUD) functions.

Models access data sources through *connectors* that are extensible and customizable. In general, application code does not use a connector directly. Rather, the `DataSource` class provides an API to configure the underlying connector.

By default, `slc` creates and uses the memory connector, which is suitable for development. To use a different datasource:

1. Use `slc loopaback:datasource` to create the new data source and add it to the application's `datasources.json`.
2. Edit `datasources.json` to add the appropriate credentials for the datasource.
3. Create a model to connect to the datasource or modify an existing model definition to use the connector.

Add a data source

To add a new data source, use the LoopBack datasource generator:

```
$ slc loopback:datasource
```

It will prompt you for the name of the new data source and the connector to use; for example, MySQL, Oracle, REST, and so on. The tool will then add an entry such as the following to `datasources.json`:

```
...
  "corp1": {
    "name": "corp1",
    "connector": "mysql"
  }
...
```

This example creates a MySQL data source called "corp1". The identifier determines the name by which you refer to the data source and can be any string.

Add data source credentials

Edit `datasources.json` to add the necessary authentication credentials for the data source; typically hostname, username, password, and database name. For example:

```
"corp1": {
    "name": "corp1",
    "connector": "mysql",
    "host": "your-mysql-server.foo.com",
    "user": "db-username",
    "password": "db-password",
    "database": "your-db-name"
  }
```

Make the model use the data source

Edit the application Model definition JSON file to set the data source used by a model:

```
"model-name": {
 "properties" : {
```

...
}
"dataSource": "*datasource-name*",
...
}

For example, using the previous example where the data source was named "mysql," if you followed the procedure in Creating a LoopBack application to create the example "books" data source, edit it to change the "dataSource" property from "db" to "corp1" to use the corresponding MySQL database:

```
"book": {
    "properties": {
      ...
    },
    "public": true,
    "dataSource": "corp1",
    "plural": "books"
  }
```

Then the books model would use the "corp1" data source that uses the MySQL connector instead of the "db" data source that uses the memory connector.

Exposing models over a REST API

Overview

It's easy to expose a LoopBack model over REST. LoopBack automatically binds a model to a list of HTTP endpoints that provide REST APIs for model instance data manipulations (CRUD) and other remote operations.

By default, the REST APIs are mounted to the plural of the model name; specifically:

- `Model.settings.plural`, if defined in `models.json`; see Project layout reference for more information.
- Automatically-pluralized model name (the default).

For example, if you have a location model, it is mounted to `/locations`.

Using the REST Router

By default, scaffolded applications expose models over REST using the `loopback.rest` router:

```
var app = loopback();
app.use(loopback.rest());

// Expose the `Product` model
app.model(Product);
```

After this, you'll have the `Product` model with create, read, update, and delete (CRUD) functions working remotely from mobile clients. At this point, the model is schema-less and the data are not checked.

You can then view generated REST documentation at http://localhost:3000/explorer.

LoopBack provides a number of Built-in models that have REST APIs. See Built-in models REST API for more information.

Predefined remote methods

By default, for a model backed by a data source that supports it, LoopBack exposes a REST API that provides all the standard create, read, update, and delete (CRUD) operations.

As an example, consider a simple model called `Location` (that provides business locations) to illustrate the REST API exposed by LoopBack. LoopBack automatically creates the following endpoints:

Model API	HTTP Method	Example Path
create()	POST	/locations
upsert()	PUT	/locations
exists()	GET	/locations/:id/exists
findById()	GET	/locations/:id
find()	GET	/locations
findOne()	GET	/locations/findOne
deleteById()	DELETE	/locations/:id
count()	GET	/locations/count
prototype.updateAttributes()	PUT	/locations/:id

The above API follows the standard LoopBack model REST API that most built-in models extend. See Model REST API for more details.

Exposing models

To expose a model over REST, set the public property to true in `/server/model-config.json`:

```
...
  "Role": {
    "dataSource": "db",
    "public": false
  },
...
```

Hiding methods

If you don't want to expose certain CRUD operations, you can easily hide them by setting the model's `shared` property to `false`. For example, following the previous example, by convention custom model code would go in the file `server/location.js`. You would add the following lines to "hide" one of the predefined remote methods:

```
var isStatic = true;
MyModel.sharedClass.find('deleteById', isStatic).shared = false;
```

Now the `deleteById()` operation and the corresponding REST endpoint will not be publicly available.

For a method on the prototype object, such as `updateAttributes()`:

```
var isStatic = false;
MyModel.sharedClass.find('updateAttributes', isStatic).shared = false;
```

Defining remote methods

In addition to the standard set of REST API endpoints that a LoopBack model exposes (see Model REST API), you can expose a model's static methods to clients over REST: these are called *remote methods*. Additionally, you can define remote hooks that are functions called when a remote method is executed (typically, before or after the remote method).

A remote method must accept a callback with the conventional `fn(err, result, ...)` signature.

Overview

Expose a remote method in a model's custom script `/common/models/`*`modelName`*`.js` file as follows:

1. Export a function that takes the model as an argument: `module.exports = function(`*`modelName`*`)`. For example:

   ```
   module.exports = function(Person){ ...
   ```

2. In this function, define the function as a static method of the model: *`modelName.functionName`* `= function(`*`args`*`) { ...}`. For example:

   ```
   Person.greet = function(msg, cb){ ... }
   ```

3. Also in this function, call `remoteMethod()` as described below to expose the function as a remote method. The first argument to `remoteMethod()` must be a string that exactly matches the static method defined in step 2. For example:

   ```
   Person.remoteMethod('greet',
                       { accepts: {arg: 'msg', type: 'string'},
                         returns: {arg: 'greeting', type: 'string'} }
   ```

   See Calling remoteMethod() below for more information.

Here is a complete example, using a Person model. You would add the following code in `/common/models/person.js`:

```
module.exports = function(Person){
    Person.greet = function(msg, cb) {
      cb(null, 'Greetings... ' + msg);
    }

    Person.remoteMethod(
        'greet',
        {
          accepts: [{arg: 'msg', type: 'string'}],
          returns: {arg: 'greeting', type: 'string'}
        }
    );
}
```

Running this in a local app and sending a POST request with the argument "LoopBack Developer" to the default URL

`http://localhost:3000/api/people/greet`

will then return:

```
{
  "greeting": "Greetings... LoopBack Developer"
}
```

Calling remoteMethod()

As summarized above, to expose a function as a remote method on a model, call the model's `remoteMethod()` function. The function's signature is:

modelName.remoteMethod(*function*, [*options*])

where:

- *modelName* is the name of the model.
- *function* is the remote method being defined
- *options* is the optional JSON object with metadata about the function (see Options below).

Options

The options argument is a JSON object, described in the following table.

Option	Required?	Description
accepts	No	Describes the remote method's arguments; See Argument descriptions. The `callback` argument is assumed; do not specify.
returns	No	Describes the remote method's callback arguments; See Argument descriptions. The `err` argument is assumed; do not specify.
http.path	No	HTTP path (relative to the model) at which the method is exposed.
http.verb	No	HTTP method (verb) at which the method is available. One of: • get • post (default) • put • del • all
description	No	A text description of the method. This is used by API documentation generators like Swagger.

Argument descriptions

The `accepts` and `returns` properties define either a single argument as an object or an ordered set of arguments as an array. Each individual argument has properties for:

Property (key)	Type	Description
arg	String	Argument name
type	String	Argument datatype; must be a Loopback type.
required	Boolean	True if argument is required; false otherwise.
root	Boolean	For callback arguments: set this property to `true` if your function has a single callback argument to use as the root object returned to remote caller. Otherwise the root object returned is a map (argument-name to argument-value).
http	String	For input arguments: a function or an object describing mapping from HTTP request to the argument value. See HTTP mapping of input arguments below.

For example, a single argument, specified as an object:

```
{arg: 'myArg', type: 'number'}
```

Multiple arguments, specified as an array:

```
[
  {arg: 'arg1', type: 'number', required: true},
  {arg: 'arg2', type: 'array'}
]
```

HTTP mapping of input arguments

There are two ways to specify HTTP mapping for input parameters (what the method accepts):

- Provide an object with a `source` property
- Specify a custom mapping function

Using an object with a source property

To use the first way to specify HTTP mapping for input parameters, provide an object with a `source` property that has one of the values shown in the following table.

Value of source property	Description
body	The whole request body is used as the value.
form query path	The value is looked up using `req.param`, which searches route arguments, the request body and the query string. Note that query and path are aliases for form.
req	The whole HTTP request object is used as the value.

For example, an argument getting the whole request body as the value:

```
{ arg: 'data', type: 'object', http: { source: 'body' } }
```

Using a custom mapping function

The second way to specify HTTP mapping for input parameters is to specify a custom mapping function; for example:

```
{
  arg: 'custom',
  type: 'number',
  http: function(ctx) {
    // ctx is LoopBack Context object

    // 1. Get the HTTP request object as provided by Express
    var req = ctx.req;

    // 2. Get 'a' and 'b' from query string or form data
    // and return their sum as the value
    return +req.param('a') + req.param('b');
  }
}
```

If you don't specify a mapping, LoopBack will determine the value as follows (assuming `name` as the name of the input parameter to resolve):

1. If there is a HTTP request parameter `args` with a JSON content, then the value of `args['name']` is used if it is defined.
2. Otherwise `req.param('name')` is returned.

Setting a remote method route

By default, a remote method is exposed at

> POST http://*apiRoot*/*modelName*/*methodName*

Where

- *apiRoot* is the application API root path.
- *modelName* is the plural name of the model .
- *methodName* is the function name.

Following the above example, if you run the application locally, then by default the remote method is exposed at:

```
POST http://localhost:3000/api/people/greet
```

To change the route, use the `http.path` and `http.verb` properties of the options argument to `remoteMethod()`, for example:

```
Person.remoteMethod(
    'greet',
    {
       accepts: {arg: 'msg', type: 'string'},
       returns: {arg: 'greeting', type: 'string'},
       http: {path: '/sayhi', verb: 'get'}
     }
);
```

This call changes the default route to

```
GET http://localhost:3000/api/people/sayhi
```

So the GET request `http://localhost:3000/api/people/sayhi?msg=LoopBack%20developer` returns:

```
{"greeting":"Greetings... LoopBack developer"}
```

Defining remote hooks

Overview

A *remote hook* enables you to execute a function before or after a remote method is called by a client. The `beforeRemote()` function runs before the remote method and `afterRemote()` runs after.

For example:

```
var request = require('request');
Users.afterRemote('count', function(ctx, unused, next) {
  request.post({
    url: 'http://another.server.com/',
    method: 'POST',
    json: ctx.result
  }, function(err, response) {
    if (err) console.error(err);
    next();
  });
});
```

Another example using wildcards in the remote function name:

```
User.beforeRemote('*.save', function(ctx, user, next) {
  if(ctx.req.accessToken) {
    next();
  } else {
    next(new Error('must be logged in to update'))
  }
});

User.afterRemote('*.save', function(ctx, user, next) {
  console.log('user has been saved', user);
  next();
});
```


The second argument to the hook (`user` in the above example) is the `ctx.result` which is not always available.

More examples of remote hooks with wildcards to run a function before any remote method is called.

```
// ** will match both prototype.* and *.*
User.beforeRemote('**', function(ctx, user, next) {
  console.log(ctx.methodString, 'was invoked remotely'); // users.prototype.save was
invoked remotely
  next();
});

Other wildcard examples
// run before any static method eg. User.find
User.beforeRemote('*', ...);

// run before any instance method eg. User.prototype.save
User.beforeRemote('prototype.*', ...);

// prevent password hashes from being sent to clients
User.afterRemote('**', function (ctx, user, next) {
  if(ctx.result) {
    if(Array.isArray(ctx.result)) {
      ctx.result.forEach(function (result) {
        result.password = undefined;
      });
    } else {
      ctx.result.password = undefined;
    }
  }

  next();
});
```

Context object

Remote hooks are provided with a Context `ctx` object that contains transport-specific data (for HTTP: `req` and `res`). The `ctx` object also has a set of consistent APIs across transports.

Applications that use loopback.rest() middleware provide the following additional `ctx` properties:

- `ctx.req`: Express Request object.
- `ctx.res`: Express Response object.

ctx.req.accessToken

The `accessToken` of the user calling the remote method.

⚠ `ctx.req.accessToken` is undefined if the remote method is not invoked by a logged in user (or other principal).

ctx.result

During `afterRemote` hooks, `ctx.result` will contain the data about to be sent to a client. Modify this object to transform data before it is sent.

⚠ The value of `ctx.result` may not be available at all times.

Additional hooks

In addition to beforeRemote and afterRemote, you can define the following hooks:

- afterInitialize
- beforeValidate / afterValidate
- beforeSave / afterSave

- beforeCreate / afterCreate
- beforeUpdate / afterUpdate
- beforeDestroy / afterDestroy

Validating model data

A *schema* defines a static model that is backed by a database. A model can validate data before passing it on to a data store such as a database to ensure that it conforms to the backend schema.

Adding a schema to a model

For example, the following code defines a schema and assigns it to the product model. The schema defines two fields (columns): name, a string, and price, a number. The field name is a required value.

```
var productSchema = {
  "name": { "type": "string", "required": true },
  "price": "number"
};
var Product = Model.extend('product', productSchema);
```

A schema imposes restrictions on the model. If a remote client tries to save a product with extra properties (for example, `description`), those properties are removed before the app saves the data in the model. Also, since `name` is a required value, the model will *only* be saved if the product contains a value for the `name` property.

See also Validations class.

Localizing validation messages

Rather than modifying the error responses returned by the server, you can localize the error message on the client. The validation error response contains error codes in `error.details.codes`, which enables clients to map errors to localized messages.

Here is an example error response:

```
{
  "name": "ValidationError",
  "status": 422,
  "message": "The Model instance is not valid. \
 See `details` property of the error object for more info.",
  "statusCode": 422,
  "details": {
    "context": "user",
    "codes": {
      "password": [
        "presence"
      ],
      "email": [
        "uniqueness"
      ]
   },
    "messages": {
      "password": [
       "can't be blank"
     ],
      "email": [
        "Email already exists"
      ]
    }
  }
}
```

Creating model relations

Related articles

- Creating models
- Extending models
- Creating model relations
- Querying models
- Model definition JSON file
- Model REST API

Overview of model relations

Individual models are easy to understand and work with. But in reality, models are often connected or related. When you build a real-world application with multiple models, you'll typically need to define relations between models. For example:

- A customer has many orders and each order is owned by a customer.
- A user can be assigned to one or more roles and a role can have zero or more users.
- A physician takes care of many patients through appointments. A patient can see many physicians too.

With connected models, LoopBack exposes as a set of APIs to interact with each of the model instances and query and filter the information based on the client's needs.

You can define the following relations between models:

- BelongsTo relations
- HasMany relations
- HasManyThrough relations
- HasAndBelongsToMany relations
- Polymorphic relations

You can define models relations in JSON in the Model definition JSON file file or in JavaScript code. The end result is the same.

When you define a relation for a model, LoopBack adds a set of methods to the model, as detailed in the article on each type of relation.

Relation options

There are three options for most relation types:

- Scope
- Properties
- Custom scope methods

Scope

The scope property can be an object or function, and applies to all filtering/conditions on the related scope.

The object or returned object (in case of the function call) can have all the usual filter options: `where`, `order`, `include`, `limit`, `offset`, ...

These options are merged into the default filter, which means that the `where` part will be `AND`-ed. The other options usually override the defaults (standard mergeQuery behavior).

When scope is a function, it will receive the current instance, as well as the default filter object.

For example:

```
// only allow products of type: 'shoe', always include products
Category.hasMany(Product,
                 { as: 'shoes',
                   scope: { where: { type: 'shoe' },
                   include: 'products'
});
Product.hasMany(Image,
                { scope: function(inst, filter) {
                           return { type: inst.type };
                         }
});  // inst is a category - match category type with product type.
```

Properties

You can specify the `properties` option in two ways:

- As an object: the keys refer to the instance, the value will be the attribute key on the related model (mapping)
- As a function: the resulting object (key/values) are merged into the related model directly.

For example, the following relation transfers the type to the product, and de-normalizes the category name into `categoryName` on creation:

```
Category.hasMany(Product,
                 { as: 'shoes',
                 properties: { type: 'type', category: 'categoryName' });
```

To accomplish the same thing with a callback function:

```
Product.hasMany(Image,
                { properties: function(inst) { // inst is a category
                                return { type: inst.type, categoryName: inst.name };
                              }
});
```

Custom scope methods

Finally, you can add custom scope methods using the `scopeMethods` property. Again, the option can be either an `object` or a `function` (adv anced use).

⚠ By default custom scope methods are not exposed as remote methods; You must set `functionName.shared = true`.

For example:

```
var reorderFn = function(ids, cb) {
  // `this` refers to the RelationDefinition
  console.log(this.name); // `images` (relation name)
  // do some reordering here & save
  cb(null, [3, 2, 1]);
};

// manually declare remoting params
reorderFn.shared = true;
reorderFn.accepts = { arg: 'ids', type: 'array', http: { source: 'body' } };
reorderFn.returns = { arg: 'ids', type: 'array', root: true };
reorderFn.http = { verb: 'put', path: '/images/reorder' };

Product.hasMany(Image, { scopeMethods: { reorder: reorderFn } });
```

Exposing REST APIs for related models

The following example demonstrates how to access connected models via REST APIs.

```
var db = loopback.createDataSource({connector: 'memory'});
  Customer = db.createModel('customer', {
    name: String,
    age: Number
  });
  Review = db.createModel('review', {
    product: String,
    star: Number
  });
  Order = db.createModel('order', {
    description: String,
    total: Number
  });

  Customer.scope("youngFolks", {where: {age: {lte: 22}}});
  Review.belongsTo(Customer, {foreignKey: 'authorId', as: 'author'});
  Customer.hasMany(Review, {foreignKey: 'authorId', as: 'reviews'});
  Customer.hasMany(Order, {foreignKey: 'customerId', as: 'orders'});
  Order.belongsTo(Customer, {foreignKey: 'customerId'});
```

The code is available at https://github.com/strongloop-community/loopback-example-datagraph.

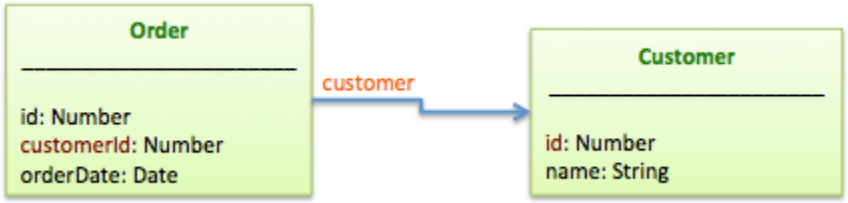
If you run the example application, the REST API is available at http://localhost:3000/api. The home page at http://0.0.0.0:3000/ contains the links shown below. Here are the example endpoints and queries:

Endpoint	Description
/api/customers	List all customers
/api/customers?filter[fields][0]=name	List all customers with the name property only
/api/customers/1	Return customer record for id of 1
/api/customers/youngFolks	List a predefined scope ‘youngFolks’
/api/customers/1/reviews	List all reviews posted by a given customer
/api/customers/1/orders	List all orders placed by a given customer
/api/customers?filter[include]=reviews	List all customers including their reviews
/api/customers?filter[include][reviews]=author	List all customers including their reviews which also includes the author
/api/customers?filter[include][reviews]=author&filter[where][age]=21	List all customers whose age is 21, including their reviews which also includes the author
/api/customers?filter[include][reviews]=author&filter[limit]=2	List first two customers including their reviews which also includes the author
/api/customers?filter[include]=reviews&filter[include]=orders	List all customers including their reviews and orders

BelongsTo relations

Overview

A `belongsTo` relation sets up a one-to-one connection with another model, such that each instance of the declaring model "belongs to" one instance of the other model. For example, if your application includes customers and orders, and each order can be placed by exactly one customer.



The declaring model (Order) has a foreign key property that references the primary key property of the target model (Customer). If a primary key is not present, LoopBack will automatically add one.

Defining a belongsTo relation

You can configure the following properties for a belongsTo relation:

- **Target model**, such as Customer.
- **Relation name**, such as 'customer'. This becomes a method on the declaring model's prototype.
- **Foreign key** property on the declaring model, such as 'customerId' that references the target model's primary key (Customer.id).

You can declare relations in the "options" property in `models.json`, for example:

Defining belongsTo relation in models.json

```
...
"Order": {
   "properties": {
       "customerId": "number",
       "orderDate": "date"
   },
   "options": {
     "relations": {
       "customer": {
         "type": "belongsTo",
         "model": "Customer",
         "foreignKey": "customerId"
       }
     }
   }
 }
...
```

Alternatively, you can define a "belongsTo" relation in code:

Defining belongsTo relation in code

```
 var Order = ds.createModel('Order', {
    customerId: Number,
    orderDate: Date
});

var Customer = ds.createModel('Customer', {
    name: String
});

Order.belongsTo(Customer);
```

The code above specifies that **Order** has a reference called `customer` to **User** using the `customerId` property of **Order** as the foreign key.

Here is a more concise way to specify the same relation:

```
Order.belongsTo(Customer, {as: 'customer', foreignKey: 'customerId'});
```

If the declaring model doesn't have a foreign key property, LoopBack will add a property with the same name. The type of the property will be the same as the type of the target model's **id** property.

If you don't specify them, then LoopBack derives the relation name and foreign key as follows:

- Relation name: Camel case of the model name, for example, for the "Customer" model the relation is "customer".
- Foreign key: The relation name appended with 'Id', for example, for relation name "customer" the default foreign key is "customerId".

Methods added to the model

Once you define the belongsTo relation, LoopBack automatically adds a method with the relation name to the declaring model class's prototype, for example: `Order.prototype.customer(...)`.

Depending on the arguments, the method can be used to get or set the owning model instance.

Example method	Description

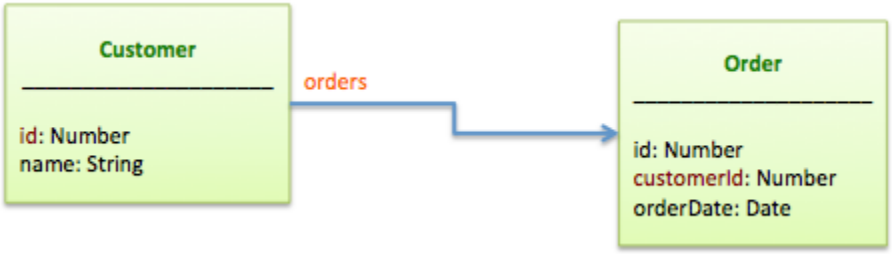
`order.customer(function(err, customer) {`
`...`
`});`
	Get the customer for the order asynchronously

`var customer = order.customer();`	Get the customer for the order synchronously
`order.customer(customer);`	Set the customer for the order

HasMany relations

Overview

A `hasMany` relation builds a one-to-many connection with another model. You'll often find this relation on the "other side" of a `belongsTo` relation . This relation indicates that each instance of the model has zero or more instances of another model. For example, in an application with customers and orders, a customer can have many orders, as illustrated in the diagram below.



The target model, **Order**, has a property, **customerId**, as the foreign key to reference the declaring model (Customer) primary key **id**.

Defining a hasMany relation

You can configure the following properties for a hasMany relation:

- **Target model**, such as Order
- **Relation name**, such as 'orders'. This becomes a method on the declaring model's prototype
- **Foreign key** property on the target model, such as 'customerId', that references the declaring model's primary key (Customer.id)

You can declare relations in the options property in `models.json`, for example:

Defining a hasMany relation in models.json

```
"Customer": {    "properties": {
        "id": {"type": "number", "id": true, "generated": true},
        "name": "string"
    },
    "options": {
      "relations": {
        "orders": {
          "type": "hasMany",
          "model": "Order",
          "foreignKey": "customerId"
        }
      }
    }
  }
```

Alternatively, you can define the relation in code:

Defining a hasMany relation in code

```
var Order = ds.createModel('Order', {
    customerId: Number,
    orderDate: Date
});

var Customer = ds.createModel('Customer', {
    name: String
});
```

Here is a more concise way to the define the same relation:

```
Customer.hasMany(Order, {as: 'orders', foreignKey: 'customerId'});
```

If not specified, LoopBack derives the relation name and foreign key as follows:

- **Relation name**: The plural form of the camel case of the model name; for example, for model name "Order" the relation name is "orders".
- **Foreign key**: The camel case of the declaring model name appended with ‘Id’, for example, for model name "Customer" the foreign key is "customerId".

Methods added to the model

Once you define a "hasMany" relation, LoopBack adds a method with the relation name to the declaring model class’s prototype automatically, for example: `Customer.prototype.orders(...)`.

Example method	Description
`customer.orders(filter, function(err, orders) { ... });`	Find orders for the customer by the filter

`var order = customer.orders.build(data);`
 	
Or equivalently:
 	
`var order = new Order({customerId: customer.id, ...});`
	Build a new order for the customer with the customerId to be set to the id of the customer. No persistence is involved.

`customer.orders.create(data, function(err, order) {`
`...`
`});`
 	
Or, equivalently:
 	
`Order.create({customerId: customer.id, ...}, function(err, order) {`
`...`
`});`
	Create a new order for the customer.

`customer.orders.destroyAll(function(err) {`
`...`
`});`
	Remove all orders for the customer.

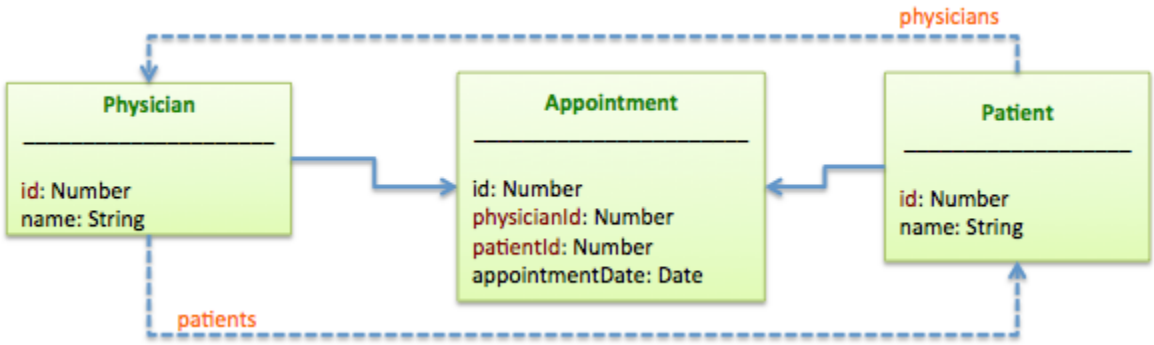
`customer.orders.findById(orderId, function(err, order) {`
`...`
`});`
	Find an order by ID.

`customer.orders.destroy(orderId, function(err) {`
`...`
`});`
	Delete an order by ID.

HasManyThrough relations

Overview

A `hasMany through` relation is often used to set up a many-to-many connection with another model. This relation indicates that the declaring model can be matched with zero or more instances of another model by proceeding through a third model. For example, in an application for a medical practice where patients make appointments to see physicians, the relevant relation declarations might be:



The “through” model, **Appointment**, has two foreign key properties, **physicianId** and **patientId**, that reference the primary keys in the declaring model, **Physician**, and the target model, **Patient**.

Defining a hasManyThrough relation

Define a "hasManyThrough" relation in code, for example:

Defining a hasManyThrough relation in code

```
var Physician = ds.createModel('Physician', {name: String});
var Patient = ds.createModel('Patient', {name: String});
var Appointment = ds.createModel('Appointment', {
    physicianId: Number,
    patientId: Number,
    appointmentDate: Date
});

Appointment.belongsTo(Patient);
Appointment.belongsTo(Physician);

Physician.hasMany(Patient, {through: Appointment});
Patient.hasMany(Physician, {through: Appointment});
// Now the Physician model has a virtual property called patients:
physician.patients(filter, callback); // Find patients for the physician
physician.patients.build(data); // Build a new patient
physician.patients.create(data, callback); // Create a new patient for the physician
physician.patients.destroyAll(callback); // Remove all patients for the physician
physician.patients.add(patient, callback); // Add an patient to the physician
physician.patients.remove(patient, callback); // Remove an patient from the physician
physician.patients.findById(patientId, callback); // Find an patient by id
```

Methods added to the model

Once you define a "hasManyThrough" relation, LoopBack adds methods with the relation name to the declaring model class's prototype automatically, for example: `physician.patients.create(...)`.

Example method	Description
`physician.patients(filter, function(err, patients) { ... });`	Find patients for the physician.
`var patient = physician.patients.build(data);`	Create a new patient.
`physician.patients.create(data, function(err, patient) { ... });`	Create a new patient for the physician.
`physician.patients.destroyAll(function(err) { ... });`	Remove all patients for the physician
`physician.patients.add(patient, function(err, patient) { ... });`	Add a patient to the physician.
`physician.patients.remove(patient, function(err) { ... });`	Remove a patient from the physician.
`physician.patients.findById(patientId, function(err, patient) { ... });`	Find an patient by ID.

These relation methods provide an API for working with the related object (patient in the example above). However, they do not allow you to access both the related object (Patient) and the "through" record (Appointment) in a single call.

For example, if you want to add a new patient and create an appointment at a certain date, you have to make two calls (REST requests):

1. Create the patient via `Patient.create`

```
POST /patients
{
  "name": "Jane Smith"
}
```

2. Create the appointment via `Appointment.create`, setting the `patientId` property to the `id` returned by `Patient.create`.

```
POST /appointments
{
  "patientId": 1,
  "physicianId": 1,
  "appointmentDate": "2014-06-01"
}
```

The following query can be used to list all patients of a given physician, including their appointment date:

```
GET /appointments?filter={"include":["patient"],"where":{"physicianId":2}}
```

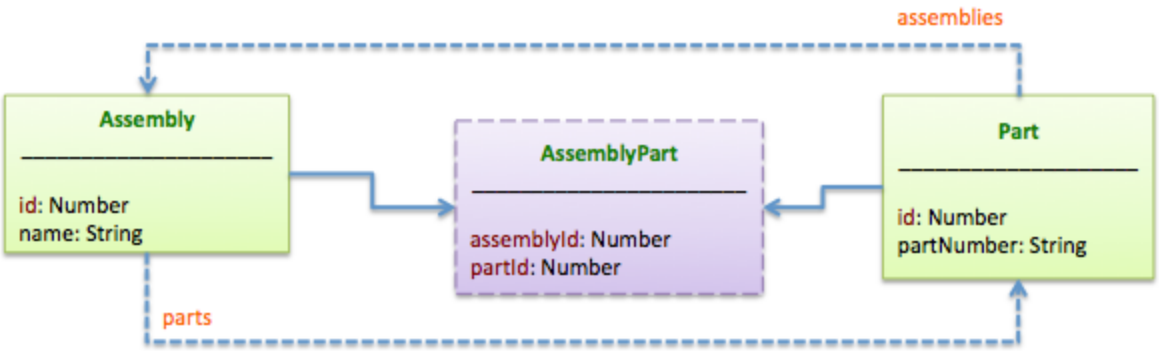
Sample response:

```
[
  {
    "appointmentDate": "2014-06-01",
    "id": 1,
    "patientId": 1,
    "physicianId": 1,
    "patient": {
      "name": "Jane Smith",
      "id": 1
    }
  }
]
```

HasAndBelongsToMany relations

Overview

A hasAndBelongsToMany relation creates a direct many-to-many connection with another model, with no intervening model. For example, in an application with assemblies and parts, where each assembly has many parts and each part appears in many assemblies, you could declare the models this way:



Defining a hasAndBelongsToMany relation

Define a hasAndBelongsToMany relation in code; for example:

```
var Assembly = ds.createModel('Assembly', {name: String});
var Part = ds.createModel('Part', {partNumber: String});
Assembly.hasAndBelongsToMany(Part);
Part.hasAndBelongsToMany(Assembly);
```

Methods added to the model

Once you define a "hasAndBelongsToMany" relation, LoopBack adds methods with the relation name to the declaring model class's prototype automatically, for example: `assembly.parts.create(...)`.

Example method	Description
`assembly.parts(filter, function(err, parts) { ... });`	Find parts for the assembly.
`var part = assembly.parts.build(data);`	Build a new part.
`assembly.parts.create(data, function(err, part) { ... });`	Create a new part for the assembly.
`assembly.parts.add(part, function(err) { ... });`	Add a part to the assembly.
`assembly.parts.remove(part, function(err) { ... });`	Remove a part from the assembly.
`assembly.parts.findById(partId, function(err, part) { ... });`	Find a part by ID.
`assembly.parts.destroy(partId, function(err) { ... });`	Delete a part by ID.

Polymorphic relations

> This documentation is still a work in progress.

LoopBack supports polymorphic relations that are conceptually similar to polymorphic relations in Ruby on Rails.

- HasMany
- BelongsTo
- HasAndBelongsToMany
- HasOne

The examples below use three example models: Picture, Author, and Reader.

HasMany

The usual options apply, for example: `as: 'photos'` to specify a different relation name/accessor.

```
Author.hasMany(Picture, { polymorphic: 'imageable' });
Reader.hasMany(Picture, { polymorphic: { // alternative syntax
  as: 'imageable', // if not set, default to: reference
  foreignKey: 'imageableId', // defaults to 'as + Id'
  discriminator: 'imageableType' // defaults to 'as + Type'
} });
```

BelongsTo

Because the related model is dynamically defined, it cannot be declared upfront.

So instead of passing in the related model(name), the name of the polymorphic relation is specified.

```
Picture.belongsTo('imageable', { polymorphic: true });

// Alternatively, use an object for setup:

Picture.belongsTo('imageable', { polymorphic: {
  foreignKey: 'imageableId',
  discriminator: 'imageableType'
} });
```

HasAndBelongsToMany

This requires an explicit 'through' model, in this case: PictureLink

The relations `Picture.belongsTo(PictureLink)` and `Picture.belongsTo('imageable', { polymorphic: true });` will be setup automatically.

The same is true for the needed properties on PictureLink.

```
Author.hasAndBelongsToMany(Picture, { through: PictureLink, polymorphic: 'imageable'
});
Reader.hasAndBelongsToMany(Picture, { through: PictureLink, polymorphic: 'imageable'
});

// Optionally, define inverse hasMany relations (invert: true):

Picture.hasMany(Author, { through: PictureLink, polymorphic: 'imageable', invert: true
});
Picture.hasMany(Reader, { through: PictureLink, polymorphic: 'imageable', invert: true
});
```

HasOne

As can be seen here, you can specify `as: 'avatar'` to explicitly set the name of the relation. If not set, it will default to the polymorphic name.

```
Picture.belongsTo('imageable', { polymorphic: true });

Author.hasOne(Picture, { as: 'avatar', polymorphic: 'imageable' });
Reader.hasOne(Picture, { polymorphic: { as: 'imageable' } });
```

Querying related models

Overview

A relation defines the connection between two models by connecting a foreign key property to a primary key property. For each relation type, LoopBack automatically mixes in helper methods to the model class to help navigate and associate the model instances to load or build a data graph.

Often, client applications want to select relevant data from the graph, for example to get user information and recently-placed orders. LoopBack provides a few ways to express such requirements in queries.

The LoopBack Datagraph example provides examples. For general information on queries, see Querying models.

Inclusion

To include related models in the response for a query, use the 'include' property of the query object or use the `include()` method on the model class. The 'include' can be a string, an array, or an object. For more information, see Include filter.

The following examples illustrate valid formats.

Load all user posts with only one additional request:

```
User.find({include: 'posts'}, function() {
  ...
});
```

Or, equivalently:

```
User.find({include: ['posts']}, function() {
  ...
});
```

Load all user posts and orders with two additional requests:

```
User.find({include: ['posts', 'orders']}, function() {
  ...
});
```

Load all post owners (users), and all orders of each owner:

```
Post.find({include: {owner: 'orders'}}, function() {
  ...
});
```

Load all post owners (users), and all friends and orders of each owner:

```
Post.find({include: {owner: [‘friends’, ‘orders’]}}, function() {
  ...
});
```

Load all post owners (users), all posts (including images), and orders of each owner:

```
Post.find({include: {owner: [{posts: ‘images’} , ‘orders’]}}, function() {
  ...
});
```

The model class also has an `include()` method. For example, the code snippet below will populate the list of user instances with posts:

```
User.include(users, 'posts', function() {
  ...
});
```

Scope

Scoping enables you to define a query as a method to the target model class or prototype. For example,

```
User.scope(‘top10Vips’, {where: {vip: true}, limit: 10});

User.top10Vips(function(err, vips) {
});
```

You can create the same function using a custom method too:

```
User.top10Vips = function(cb) {
  User.find({where: {vip: true}, limit: 10}, cb);
}
```

Embedded models and relations

> This documentation is still a work in progress.

Embedded relations are conceptually similar to sub-documents in MongooseJS.

Embedded models are usually not persisted separately, however, you can do so to de-normalize or "freeze" data.

Example models: TodoList, TodoItem

```
// the accessor name defaults to: 'singular + List' (todoItemList)
TodoList.embedsMany(TodoItem, { as: 'items' });
TodoList.create({ name: 'Work' }, function(err, list) {
  list.items.build({ content: 'Do this', priority: 5 });
  list.items.build({ content: 'Do that', priority: 1 });
  list.save(function(err, list) {
    console.log(err, list);
  });
});
TodoList.findOne(function(err, list) {
  console.log(list.todoItems[0].content); // `Do this`
  console.log(list.items.at(1).content); // `Do that`
  list.items.find({ where: { priority: 1 } }, function(err, item) {
     console.log(item.content); // `Do that`
  });
});
```

Advanced example: embed with belongsTo

Embedded models can have relations, just like any other model. This is a powerful technique that can simplify certain setups, and enables de-normalization of data. An added benefit is that the internal array explicitly defines the order of items.

The following example creates a relation similar to a HasManyThrough relation. However, it uses an embedded model called `Link` instead of a separately-persisted through-model. The models involved are Category, Product, and Link.

The new relation options `scope` and `properties` come in to play here as well.

```
var Category = db.define('Category', { name: String });
  var Product = db.define('Product', { name: String });
  var Link = db.define('Link', { notes: String });

  Category.embedsMany(Link, {
    as: 'items', // rename (default: productList)
    scope: { include: 'product' }
  });

  Link.belongsTo(Product, {
    foreignKey: 'id', // re-use the actual product id
    properties: { id: 'id', name: 'name' }, // denormalize, transfer id
    // For info on invertProperties see
https://github.com/strongloop/loopback-datasource-juggler/pull/219
    options: { invertProperties: true }
  });

  Category.create({ name: 'Category B' }, function(err, cat) {
    var category = cat;
    var link = cat.items.build({ notes: 'Some notes...' });
    link.product.create({ name: 'Product 1' }, function(err, p) {
      //cat.links[0].id.should.eql(p.id);
      console.log(cat.links[0].id); // 1
      console.log(cat.links[0].name);  // Product 1
      console.log(cat.links[0].notes); // Some notes...
      console.log(cat.items.at(0));// Should equal(cat.links[0]);

      Category.findById(category.id, function(err, cat) {
        console.log(cat.name); // Should equal('Category B');
        console.log(cat.links); // Should equal "{id: 1, name: 'Product 1', notes:
'Some notes...'}"
        console.log(cat.items.at(0)); // Should Equal(cat.links[0]);
        cat.items(function(err, items) { // alternative access
          console.log(items); /* Should equal: [{ notes: 'Some notes...', id: 1, name:
'Product 1',
                                                  product: { name: 'Product 1', id: 1
}}] */
          items[0].product(function(err, p) {
            console.log(p.name); // Should equal ('Product 1');
          });
        });
      });
    })
  });
```

Another example

> ⚠ This example requires should.js and Mocha.

The following example creates a relation similar to a hasManyThrough relation. However, it uses an embedded model called `Link` instead of a separately persisted "through" model. The example uses models named Category, Product, and Link

The example also uses relation options `scope` and `properties` as well.

```
var Category = db.define('Category', { name: String });
var Product = db.define('Product', { name: String });
var Link = db.define('Link', { notes: String });
Category.embedsMany(Link, {
  as: 'items', // rename (default: productList)
  scope: { include: 'product' }
});
Link.belongsTo(Product, {
  foreignKey: 'id', // re-use the actual product id
  properties: { id: 'id', name: 'name' }, // denormalize, transfer id
  options: { invertProperties: true }
});

Category.create({ name: 'Category B' }, function(err, cat) {
  category = cat;
  var link = cat.items.build({ notes: 'Some notes...' });
  link.product.create({ name: 'Product 1' }, function(err, p) {
    cat.links[0].id.should.eql(p.id);
    cat.links[0].name.should.equal('Product 1'); // denormalized
    cat.links[0].notes.should.equal('Some notes...');
    cat.items.at(0).should.equal(cat.links[0]);
    done();
  })
});

Category.findById(category.id, function(err, cat) {
  cat.name.should.equal('Category B');
  cat.links.toObject().should.eql([
    {id: 5, name: 'Product 1', notes: 'Some notes...'}
  ]);
  cat.items.at(0).should.equal(cat.links[0]);
  cat.items(function(err, items) { // alternative access
    items.should.be.an.array;
    items.should.have.length(1);
    items[0].product(function(err, p) {
      p.name.should.equal('Product 1'); // actual value
      done();
    });
  });
});
```

Defining in model JSON (LDL)

```
{
  "name": "Person",
  "plural": "people",
  "base": "PersistedModel",
  "properties": {
    "firstName": {
      "type": "string",
      "required": true
    },
    "lastName": {
      "type": "string"
    }
  },
  "validations": [],
  "relations": {
    "addresses": {
      "type": "embedsMany",
      "model": "Location",
      "options": {
        "validate": true,
        "autoId": false
      }
    },
    "pictures": {
      "type": "hasMany",
      "model": "Picture",
      "polymorphic": "imageable"
    }
  },
  "acls": [],
  "methods": [],
  "mixins": {
    "ObjectId": true
  }
}
```

LoopBack Definition Language

Use LoopBack Definition Language (LDL) to define LoopBack data models in JavaScript or plain JSON. With LoopBack, you typically start with a model definition that describes the structure and types of data. The model establishes LoopBack's common specification of data.

Describing a simple model

You can use `slc loopback:model` command to define a model. For more information, see LoopBack model generator.

The simplest form of a property definition in JSON has a `propertyName: type` element for each property. The key is the name of the property and the value is the type of the property. For example:

```
{
    "id": "number",
    "firstName": "string",
    "lastName": "string"
}
```

This example defines a `user` model with three properties:

- id - The user id, a number.
- firstName - The first name, a string.
- lastName - The last name, a string.

Each key in the JSON object defines a property in the model that is cast to its associated type. More advanced forms are covered later in this article.

LDL supports a list of built-in types, including the basic types from JSON:

- String
- Number
- Boolean
- Array
- Object

The type name is case-insensitive; so for example you can use either "Number" or "number." See LoopBack types for more information.

You can also describe the same model in JavaScript code:

```
 var UserDefinition = {
    id: Number,
    firstName: String,
    lastName: String
}
```

The JavaScript version is less verbose, since it doesn't require quotes for property names. The types are described using JavaScript constructors, for example, `Number` for `"Number"`. String literals are also supported.

To use the model in code is easy, because LoopBack builds a JavaScript constructor (or class) for you.

Advanced LDL features

For detailed reference information on defining a model in LDL, see Model definition JSON file.

Extending a model

You can extend an existing model to create a new model. For example, you could extend the `User` model to create the `Customer` model as illustrated in the code below. The `Customer` model will then inherit properties and methods from the `User` model.

```
 var Customer = User.extend('customer', {
    accountId: String,
    vip: Boolean
});
```

Mixing in model definitions

Some models share the common set of properties and logic around. LDL allows a model to mix in one or more other models. For example:

```
var TimeStamp = modelBuilder.define('TimeStamp', {created: Date, modified: Date});
var Group = modelBuilder.define('Group', {groups: [String]});
User.mixin(Group, TimeStamp);
```

LoopBack types

Summary

Various LoopBack methods accept type descriptions, for example remote methods and dataSource.createModel(). The following is a list of supported types.

Type	Description	Example
null	JSON null	`null`
Boolean	JSON Boolean	`true`
Number	JSON number	`3.1415`
String	JSON string	`"StrongLoop"`
Object	JSON object or any type See Object types below.	`{ "firstName": "John", "lastName": "Smith", "age": 25 }`
Array	JSON array See Array types below.	`[ "one", 2, true ]`
Date	JavaScript Date object	`new Date("December 17, 2003 03:24:00");`
Buffer	Node.js Buffer object	`new Buffer(42);`
GeoPoint	LoopBack GeoPoint object	`new GeoPoint({lat: 10.32424, lng: 5.84978});`

⚠ Use the Object type when you need to be able to accept values of different types, for example a string or an array.

Array types

LDL supports array types as follows:

- `{emails: [String]}`
- `{"emails": ["String"]}`
- `{emails: [{type: String, length: 64}]}`

Object types

A model often has properties that consist of other properties. For example, the user model can have an `address` property that in turn has properties such as `street`, `city`, `state`, and `zipCode`.

LDL allows inline declaration of such properties, for example:

```
 var UserModel = {
    firstName: String,
    lastName: String,
    address: {
        street: String,
        city: String,
        state: String,
        zipCode: String
    },
    ...
}
```

The value of the address is the definition of the `address` type, which can be also considered as an anonymous model.

If you intend to reuse the address model, we can define it independently and reference it in the user model. For example:

```
 var AddressModel = {
    street: String,
    city: String,
    state: String,
    zipCode: String
};

var Address = ds.define('Address', AddressModel);

var UserModel = {
        firstName: String,
        lastName: String,
        address: 'Address',  // or address: Address
        ...
}

var User = ds.define('User', UserModel);
```

⚠ The user model has to reference the Address constructor or the model name - `'Address'`.

Using built-in models

Overview

The built-in models all extend the base Model object.

Loopback provides useful built-in models for common use cases:

- **Application model** - contains metadata for a client application that has its own identity and associated configuration with the LoopBack

server.
- **User model** - register and authenticate users of your app locally or against third-party services.
- **Access control models** - ACL, AccessToken, Scope, Role, and RoleMapping models for controlling access to applications, resources, and methods.
- **Email model** - send emails to your app users using SMTP or third-party services.

Application model

Use the Application model to manage client applications and organize their users.

User model

The user model represents an application's users. Use it to register and authenticate users of your app locally or against third-party services.

Creating a new user

Create a user by adding a model instance, in the same way as for any other model; username and password are not required.

```
User.create({email: 'foo@bar.com', password: 'bar'}, function(err, user) {
  console.log(user);
});
```

Logging in a user

Create a access token for a user using the local authentication and authorization strategy.

```
 User.login({username: 'foo', password: 'bar'}, function(err, accessToken) {
  console.log(accessToken);
});
```

Logging out a user

```
 // login a user and logout
User.login({"email": "foo@bar.com", "password": "bar"}, function(err, accessToken) {
  User.logout(accessToken.id, function(err) {
    // user logged out
  });
});

// logout a user (server side only)
User.findOne({email: 'foo@bar.com'}, function(err, user) {
  user.logout();
});
```

Verifying email addresses

Require a user to verify their email address before being able to login. This will send an email to the user containing a link to verify their address. Once the user follows the link they will be redirected to web root (`"/"`) and will be able to login normally.

This example creates a remote hook on the User model executed after the `create()` method is called.

```
 // first setup the mail datasource
var mail = loopback.createDataSource({
  connector: loopback.Mail,
  transports: [{
    type: 'smtp',
    host: 'smtp.gmail.com',
    secure: true,   // was secureConnection
    port: 465,
    auth: {
      user: 'you@gmail.com',
      pass: 'your-password'
    }
  }]
});

User.email.attachTo(mail);
User.afterRemote('create', function(ctx, user, next) {
  var options = {
    type: 'email',
    to: user.email,
    from: 'noreply@myapp.com',
    subject: 'Thanks for Registering at FooBar',
    text: 'Please verify your email address!'
    template: 'verify.ejs',
    redirect: '/'
  };

  user.verify(options, next);
});
```

Reset password

Use the `User.resetPassword` method to reset a user's password. Request a password reset access token.

```
User.resetPassword({
  email: 'foo@bar.com'
}, function () {
  console.log('ready to change password');
});
```

Access control models

Use access control models to control access to applications, resources, and methods. These models include:

- ACL
- AccessToken
- Scope
- Role
- RoleMapping

ACL model

An ACL model connects principals to protected resources. The system grants permissions to principals (users or applications, that can be grouped into roles) .

- Protected resources: the model data and operations (model/property/method/relation)
- Is a given client application or user allowed to access (read, write, or execute) the protected resource?

Creating a new ACL instance.

```
ACL.create( {principalType: ACL.USER,
             principalId: 'u001',
             model: 'User',
             property: ACL.ALL,
          accessType: ACL.ALL,
             permission: ACL.ALLOW},
             function (err, acl) { ACL.create( {principalType: ACL.USER,
                                                principalId: 'u001',
                                                model: 'User',
                                                property: ACL.ALL,
                                                accessType: ACL.READ,
                                                permission: ACL.DENY},
                                                function (err, acl) { }
                                                })
                                 }
             })
```

Email model

Send email messages

The following example illustrates how to send emails from an app. Add the following code to a file in the `/models` directory:

```
var loopback = require('loopback');
var MyEmail = loopback.Email.extend('my-email');

// create a mail data source
var mail = loopback.createDataSource({
  connector: loopback.Mail,
  transports: [{
    type: 'smtp',
    host: 'smtp.gmail.com',
    secure: true,
    port: 465,
    auth: {
      user: 'you@gmail.com',
      pass: 'your-password'
    }
  }]
});

// attach the model
MyEmail.attachTo(mail);

// send an email
MyEmail.send({
  to: 'foo@bar.com',
  from: 'you@gmail.com',
  subject: 'my subject',
  text: 'my text',
  html: 'my <em>html</em>'
}, function(err, mail) {
  console.log('email sent!');
});
```

⚠ The mail connector uses nodemailer. See the nodemailer docs for more information.

Confirming email address

See Verifying email addresses.

Model property reference

⚠ This reference information is being moved to the API documentation. Until that is complete, it is provided here.

- Application model properties
  - Basic properties
  - Authorization properties
  - Security properties
  - Push notification properties
  - Authentication scheme properties
- ACL model
  - Properties
- Role model
- Scope model
- RoleMapping

Application model properties

The application model represents the metadata for a client application that has its own identity and associated configuration with the LoopBack server.

Basic properties

Each application has the basic properties described in the following table.

Property	Type	Required?	Description
id	String		Automatically generated ID
name	String	Yes	Name of the application
description	String	No	Description of the application
icon	String		URL of the icon
status	String		Status of the application, such as production/sandbox/disabled
created	Date		Timestamp of the record being created
modified	Date		Timestamp of the record being modified
owner	String		User ID of the developer who registers the application
collaborators	[String]		Array of user IDs who have permissions to work on this application. NOTE: Currently this property does not do anything; the feature is not yet implemented.

Authorization properties

The following table describes application oAuth 2.0 settings.

> LoopBack does not yet support oAuth2.0.

Property	Type	Description
url	String	The application URL
callbackUrls	[String]	An array of pre-registered callback URLs for oAuth 2.0
permissions		Array of oAuth 2.0 scopes that can be requested by the application

Security properties

The following keys are automatically generated by the application creation process. They can be reset upon request.

Property	Type	Description
clientKey		Secret for mobile clients
javaScriptKey		Secret for JavaScript clients
restApiKey		Secret for REST APIs
windowsKey		Secret for Windows applications
masterKey		Secret for REST APIS. It bypasses model level permissions

Push notification properties

The application can be configured to support multiple methods of push notifications. All push properties are defined within the "pushSettings" key. Within this:

- `apns` key contains properties for Apple Push Notification Service for iOS devices
- `gcm` key contains properties for Google Cloud Messaging for Android devices.

Properties for apns:

Property	Type	Description
certData		

keyData		
production	Boolean	
pushOptions	Object	• port: Number (default is 2195)
feedbackOptions	Object	• batchFeedback: Boolean • interval: Number • port: Number (default is 2196)

Properties for gcm:

Property	Type	Description
serverApiKey		

For example, from the loopback-push-notification-example application:

```
$scope.create = function () {
    $http.post('/api/applications',
      {
        name: $scope.name,
        description: $scope.description,
        pushSettings: {
          apns: {
            certData: $scope.certData,
            keyData: $scope.keyData
          },
          gcm: {
            serverApiKey: $scope.gcmKey
          }
        }
      })
      .success(function (data, status, headers) {
        // console.log(data, status);
        $scope.id = 'Application Id: ' + data.id;
        $scope.restApiKey = 'Application Key: ' + data.restApiKey;
      });
  };
```

Authentication scheme properties

Property	Type	Description
authenticationEnabled		
anonymousAllowed		
authenticationSchemes		
scheme	String	Name of the authentication scheme, such as local, facebook, google, twitter, linkedin, github
credential		Scheme-specific credentials

ACL model

Properties

The following table describes the properties of the ACL object:

Property	Type	Description

model	String	Name of the model
property	String	name of the property, method, scope, or relation
accessType	String	Type of access: READ, WRITE, or EXECUTE
permission	String	Type of permission: ALLOW, or DENY
principalType	String	Type of principal, for example application, user, role
principalId	String	

Role model

The following table describes the properties of the role model:

Property	Type	Description
id	String	Role ID
name	String	Role name
description	String	Description of the role
created	Date	Timestamp of creation date
modified	Date	Timestamp of modification date

LoopBack defines some special roles:

Identifier	Name	Description
Role.OWNER	$owner	Owner of the object
Role.RELATED	$related	Any user with a relationship to the object
Role.AUTHENTICATED	$authenticated	Authenticated user
Role.UNAUTHENTICATED	$unauthenticated	Unauthenticated user
Role.EVERYONE	$everyone	Everyone

Scope model

The following table describes the properties of the Scope model:

Property	Type	Description
name	String	Scope name; required
description	String	Description of the scope

RoleMapping

A RoleMapping entry maps one or more principals to one role. A RoleMapping entry belongs to one role, based on the roleId property.

The following table describes the properties of the roleMapping model:

Property	Type	Description
id	String	ID
roleId	String	Role ID
principalType	String	Principal type, such as user, application, or role
principalId	String	Principal ID

Extending built-in models

See Extending models for general information on how to create a model that extends (or "inherits from") another model.

Querying models

Overview

LoopBack models provide both via Node API functions and REST conventions to query data using *filters*, as outlined in the following table. The capabilities and options are the same–the only difference is the syntax used in HTTP requests versus Node function calls. In both cases, LoopBack models return JSON values.

Query	Model API (Node)	REST API
Find all model instances using specified filters.	`find(filter, callback)` Where filter is a JSON object containing the query filters. See Filters below.	`GET /modelName?filter...` See Model REST API - Find matching instances. See Filters below.
Find first model instance using specified filters.	`findOne(filter, callback)` Where filter is a JSON object containing the query filters. See Filters below.	`GET /modelName/findOne?filter...` See Model REST API - Find first instance. See Filters below.
Find instance by ID.	`findById()`	`GET /modelName/modelID` See Model REST API - Find instance by ID.

⚠ A REST query must include the literal string "filter" in the URL query string. The Node API call does not include the literal string "filter" in the JSON.

Examples

See additional examples of each kind of filter in the individual articles on filters (for example Where filter).

An example of using the `find()` method with both a *where* and a *limit* filter:

```
Account.find({where: {name: 'John'}, limit: 3}, function(err, accounts) { ... });
```

Equivalent using REST:

```
/accounts?filter[where][name]=John&filter[limit]=[3]
```

Filters

In both REST and Node API, you can use any number of filters to define a query. The following table describes the filter types:

Filter type	Type	Description
fields	Object, Array, or String	Specify fields to include in or exclude from the response. See Fields filter.

include	String, Object, or Array	Include results from related models, for relations such as *belongsTo* and *hasMany*. See Include filter.
limit	Number	Limit the number of instances to return. See Limit filter.
order	String	Specify sort order: ascending or descending. See Order filter.
skip (offset)	Number	Skip the specified number of instances. See Skip filter.
where	Object	Specify search criteria; similar to a WHERE clause in SQL. See Where filter.

REST syntax

Specify filters in the HTTP query string:

```
?filter=filterType=spec&filterType=spec....
```

The number of filters that you can apply to a single request is limited only by the maximum URL length, which generally depends on the client used.

Node syntax

Specify filters as the first argument to `find()` and `findOne()`:

```
{ filterType: spec, filterType: spec, ... }
```

There is no theoretical limit on the number of filters you can apply.

Where:

- *filterType* is the filter: where, include, order, limit, skip, or fields.
- *spec* is the specification of the filter: for example for a *where* filter, this is a logical condition that the results must match; for an *include* filter it specifies the related fields to include.

Using stringified JSON in REST queries

Instead of the standard REST syntax described above, you can also use "stringified JSON" in REST queries. To do this, simply use the JSON specified for the Node syntax, as follows:

```
?{ filter: Node-API-query-JSON }
```

where *Node-API-query-JSON* is the JSON used in the Node sntax. Essentially, you must wrap the JSON used in the Node query within `{ filter: ... }`.

For example:

```
GET /api/activities/findOne?{ filter: { where: { id: '53825529' } } }
```

However, the query string must be URL-encoded, so the actual request is:

```
GET
/api/activities/findOne?%3F%7B%20filter%3A%7Bwhere%3A%7Bid%3A%2753825529%27%7D%7D%7D
```

Space characters were stripped from the above URL for brevity.

Fields filter

A *fields* filter specifies properties (fields) to include or exclude from the results.

REST

```
```

filter[fields][*propertyName*]=<true|false>&filter[fields][propertyName]=<true|false>...

Note that to include more than one field in REST, use multiple filters.

You can also use stringified JSON format in a REST query.

Node API

{ fields: {*propertyName*: <true|false>, *propertyName*: <true|false>, ... } }

Where:

- *propertyName* is the name of the property (field) to include or exclude.
- <true|false> signifies either `true` or `false` Boolean literal. Use `true` to include the property or `false` to exclude it from results. You can also use 1 for true and 0 for false.

By default, queries return all model properties in results. However, if you specify at least one fields filter with a value of `true`, then by default the query will include **only** those you specifically include with filters.

Examples

Return only `id`, `make`, and `model` properties:

REST

```
?filter[fields][id]=true&filter[fields][make]=true&filter[fields][model]=true
```

Node API

```
{ fields: {id: true, make: true, model: true} }
```

Returns:

```
[
  {
    "id": "1",
    "make": "Nissan",
    "model": "Titan"
  },
  {
    "id": "2",
    "make": "Nissan",
    "model": "Avalon"
  },
  ...
]
```

Exclude the `vin` property:

REST

```
?filter[fields][vin]=false
```

Node API

```
{ fields: {vin: false} }
```

Include filter

An *include* filter enables you to include results from related models in a query, for example models that have belongsTo or hasMany relations, to optimize the number of requests. See Creating model relations for more information.

The value of the include filter can be a string, an array, or an object.

REST

filter[include][*relatedModel*]=*propertyName*

You can also use stringified JSON format in a REST query.

Node API

{include: '*relatedModel*'}
{include: ['*relatedModel1*', '*relatedModel2*', ...]}
{include: {*relatedModel1*: [{*relatedModel2*: '*propertyName*'} , '*relatedModel*']}}

Where:

- *relatedModel*, *relatedModel1*, and *relatedModel2* are the names (pluralized) of related models.
- *propertyName* is the name of a property in the related model.

Examples

```
User.find({include: 'posts'}, function() { ... });
```

Return all user posts and orders with two additional requests:

```
User.find({include: ['posts', 'orders']}, function() { ... });
```

Return all post owners (users), and all orders of each owner:

```
Post.find({include: {owner: 'orders'}}, function() { ... });
```

Return all post owners (users), and all friends and orders of each owner:

```
Post.find({include: {owner: [‘friends’, ‘orders’]}}, function() { ... });
```

Return all post owners (users), and all posts and orders of each owner. The posts also include images.

```
Post.find({include: {owner: [{posts: ‘images’} , ‘orders’]}}, function() { ... });
```

REST examples

These examples assume a customer model with a hasMany relationship to a reviews model.

Return all customers including their reviews:

```
/customers?filter[include]=reviews
```

Return all customers including their reviews which also includes the author:

```
/customers?filter[include][reviews]=author
```

Return all customers whose age is 21, including their reviews which also includes the author:

```
/customers?filter[include][reviews]=author&filter[where][age]=21
```

Return first two customers including their reviews which also includes the author

```
/customers?filter[include][reviews]=author&filter[limit]=2
```

Return all customers including their reviews and orders

```
/customers?filter[include]=reviews&filter[include]=orders
```

Limit filter

A *limit* filter limits the number of records returned to the specified number (or less).

REST

filter[limit]=*n*

Node

{limit: *n*}

You can also use stringified JSON format in a REST query.

Where *n* is the maximum number of results (records) to return.

Examples

Return only the first five query results:

REST

```
/cars?filter[limit]=5
```

Node API

```
Cars.find( {limit: 5},  function() { ... } )
```

Order filter

An *order* filter specifies how to sort the results: ascending (ASC) or descending (DESC) based on the specified property.

REST

Order by one property:

```
filter[order]=propertyName <ASC|DESC>
```

Order by two or more properties:

```
filter[order][0]=propertyName <ASC|DESC>&filter[order][1]propertyName]=<ASC|DESC>...
```

You can also use stringified JSON format in a REST query.

Node

Order by one property:

```
{ order: 'propertyName <ASC|DESC>' }
```

Order by two or more properties:

```
{ order: ['propertyName <ASC|DESC>', 'propertyName <ASC|DESC>',...] }
```

Where:

- *propertyName* is the name of the property (field) to sort by.
- <ASC | DESC> signifies either ASC for ascending order or DESC for descending order.

Examples

Return the three loudest three weapons, sorted by the `audibleRange` property:

REST

```
/weapons?filter[order]=audibleRange%20DESC&filter[limit]=3
```

Node API

```
weapons.find({
 order: 'price DESC',
 limit: 3 });
```

Skip filter

A skip filter omits the specified number of returned records. This is useful, for example, to paginate responses.

Use `offset` as an alias for `skip`.

REST:

?filter=[skip]=*n*

Node:

{ skip: *n* }

You can also use stringified JSON format in a REST query.

Where *n* is the number of records to skip.

Examples

The following REST request skips the first 20 records returned and limits the number of returned records to 10; essentially it returns the 21st through 31st records:

```
/cars?filter[limit]=10&filter[skip]=20
```

Where filter

A *where* filter specifies a set of logical conditions to match, similar to a WHERE clause in a SQL query.

In the first form, the condition is equivalence, that is, it tests whether *property* equals *value*. The second form is for all other conditions.

REST

filter[where][*property*]=*value*

filter[where][*property*][*op*]=*value*

You can also use stringified JSON format in a REST query.

Node API

{where: {*property*: *value*}}

{where: {*property*: {*op*: *value*}}}

Where:

- *property* is the name of a property (field) in the model being queried.
- *value* is a literal value.
- *op* is one of the operators listed below.

For example:

```
/api/cars?filter[where][carClass]=fullsize
```

Or equivalently:

```
Cars.find({ where: {carClass:'fullsize'} });
```

See Examples for many more examples.

AND and OR operators

Use the AND and OR operators to create compound logical filters based on simple where filter conditions.

REST

[where][<and|or>][0]condition1&[where][<and|or>]*condition2*...

Node API

{where: {<and|or>: [*condition1*, *condition2*, ...]}}

Where *condition1* and *condition2* are a filter conditions.

Operators

This table describes the operators available in "where" filters. See Examples below.

Operator	Description
and	Logical AND operator
or	Logical OR operator
gt, gte	Numerical greater than (>); greater than or equal (>=). Valid only for numerical and date values.
lt, lte	Numerical less than (<); less than or equal (<=). Valid only for numerical and date values.
between	True if the value is between the two specified values: greater than or equal to first value and less than or equal to second value.
inq, nin	In / not in an array of values.
near	For geolocations, return the closest points, sorted in order of distance. Use with `limit` to return the n closest points.
neq	Not equal (!=)
like, nlike	LIKE / NOT LIKE operators for use with regular expressions. The regular expression format depends on the backend datasource.

Examples

- Equivalence
- gt and lt
- and / or
- between
- near
- like and nlike
- inq

Equivalence

Weapons with name M1911:

```
/weapons?filter[where][name]=M1911
```

gt and lt

```
transaction.find({
      where: {
        userId: user.id,
        time: {gt: Date.now() - ONE_MONTH}
      }
    }
```

For example, the following query returns all instances of the employee model using a *where* filter that specifies a date property after (greater than) the specified date:

```
/employees?filter[where][date][gt]=2014-04-01T18:30:00.000Z
```

The same query using the Node API:

```
Employees.find({
  where: {
    date: {gt: Date('2014-04-01T18:30:00.000Z')}
  }
});
```

The top three weapons with a range over 900 meters:

```
/weapons?filter[where][effectiveRange][gt]=900&filter[limit]=3
```

Weapons with audibleRange < 10:

```
/weapons?filter[where][audibleRange][lt]=10
```

and / or

The following code is an example of using the "and" operator to find posts that have title = 'My Post' and content = 'Hello'.

```
Post.find({where: {and: [{title: 'My Post'}, {content: 'Hello'}]}},
          function (err, posts) {
            ...
});
```

Equivalent in REST:

```
?filter[where][and][0][title]=My%20Post&filter[where][and][1][content]=Hello
```

Example using the "or" operator to finds posts that either have title = 'My Post' or content = 'Hello'

```
Post.find({where: {or: [{title: 'My Post'}, {content: 'Hello'}]}},
          function (err, posts) {
            ...
});
```

More complex example. The following expresses

`(field1= foo and field2=bar) or field1=morefoo`:

```
{
  "or": {
 "and": [ {"field1": "foo"}, {"field2": "bar"}  ],
  "field1": "morefoo"
   }
}
```

between

Example of between operator:

```
filter[where][price][between][0]=0&filter[where][price][between][1]=7
```

In Node API:

```
Shirts.find({where: {size: {between: [0,7]}}}, function (err, posts) { ... } )
```

near

Example using the **near** operator that returns the three closest locations to a given geo point:

```
/locations?filter[where][geo][near]=153.536,-28.1&filter[limit]=3
```

like and nlike

Example of like operator:

```
Post.find({where: {title: {like: 'M.+st'}}}, function (err, posts) { ... });
```

Example of nlike operator:

```
Post.find({where: {title: {nlike: 'M.+XY'}}}, function (err, posts) {
```

When using the memory connector:

```
User.find({where: {name: {like: '%St%'}}}, function (err, posts) { ... });
User.find({where: {name: {nlike: 'M%XY'}}}, function (err, posts) { ... });
```

inq

Example of inq operator:

```
Posts.find({where: {id: {inq: [123, 234]}}},
  function (err, p){... });
```

REST:

```
/medias?filter[where][keywords][inq]=foo&filter[where][keywords][inq]=bar
```

Or

```
?filter={"where": {"keywords": {"inq": ["foo", "bar"]}}}
```

Built-in models REST API

For more information, see:

- Exposing models over a REST API
- REST connector

Overview

LoopBack provides a number of Using built-in models that have REST APIs. Many of them inherit endpoints from the generic Model REST API.

Configuration

By default, LoopBack uses `/api` as the URI root for the application REST API. To change it, set the `apiPath` variable in the application `app.js` file.

Request format

For POST and PUT requests, the request body must be JSON, with the Content-Type header set to `application/json`.

JSON query string encoding

LoopBack uses the syntax from node-querystring to encode JSON objects or arrays as query strings. For example,

Query string	JSON

`user[name][first]=John`
`&user[email]=callback@strongloop.com`
	`{ user:`
	`{ name: { first: 'John' },`
	`email: 'callback@strongloop.com' } }`

`user[names][]=John`
`&user[names][]=Mary`
`&user[email]=callback@strongloop.com`
	`{ user:`
	`{ names: ['John', 'Mary'],`
	`email: 'callback@strongloop.com' } }`

`items=a` `&items=b`	`{ items: ['a', 'b'] }`

For more examples, see node-querystring

Response format

The response format for all requests is a JSON object, generally with the following properties:

- message: String error message.
- stack: String stack trace.
- statusCode: Integer HTTP status code.

Some responses have an empty body.

The HTTP status code indicates whether a request succeeded:

- Status code 2xx indicates success
- Status code 4xx indicates request related issues.
- Status code 5xx indicates server-side problems.

The response for an error is in the following format:

```
{
"error": {
    "message": "could not find a model with id 1",
    "stack": "Error: could not find a model with id 1\n ...",
    "statusCode": 404
    }
}
```

Access token REST API

All of the endpoints in the access token REST API are inherited from the generic model API. The reference is provided here for convenience.

Quick reference

URI Pattern	HTTP Verb	Default Permission	Description	Arguments
`/accessTokens`	POST	Allow	Add access token instance and persist to data source. Inherited from generic model API.	JSON object (in request body)
`/accessTokens`	GET	Deny	Find all instances of accessTokens that match specified filter. Inherited from generic model API .	One or more filters in query parameters: • where • include • order • limit • skip / offset • fields
`/accessTokens`	PUT	Deny	Update / insert access token instance and persist to data source. Inherited from generic model API .	JSON object (in request body)
`/accessTokens/`*`id`*	GET	Deny	Find access token by ID: Return data for the specified access token instance ID. Inherited from generic model API .	*id*, the access token instance ID (in URI path)

`/accessTokens/id`	PUT	Deny	Update attributes for specified access token ID and persist. Inherited from generic model API .	Query parameters: • data - An object containing property name/value pairs • *id* - The model id
`/accessTokens/id`	DELETE	Deny	Delete access token with specified instance ID. Inherited from generic model API .	*id*, access token ID (in URI path)
`/accessTokens/id/exists`	GET	Deny	Check instance existence: Return true if specified access token ID exists. Inherited from generic model API .	URI path: • *id* - Model instance ID
`/accessTokens/count`	GET	Deny	Return the number of access token instances that matches specified where clause. Inherited from generic model API.	Where filter specified in query parameter
`/accessTokens/findOne`	GET	Deny	Find first access token instance that matches specified filter. Inherited from generic model API .	Same as Find matching instances.

ACL REST API

All of the endpoints in the ACL REST API are inherited from the generic model API. The reference is provided here for convenience.

By default, the ACL REST API is not exposed. To expose it, add the following to models.json:

```
"acl": {
    "public": true,
    "options": {
      "base": "ACL"
    },
    "dataSource": "db"
  },
```

Quick reference

URI Pattern	HTTP Verb	Default Permission	Description	Arguments
`/acls`	POST	Allow	Add ACL instance and persist to data source. Inherited from generic model API.	JSON object (in request body)
`/acls`	GET	Deny	Find all instances of ACLs that match specified filter. Inherited from generic model API .	One or more filters in query parameters: • where • include • order • limit • skip / offset • fields
`/acls`	PUT	Deny	Update / insert ACL instance and persist to data source. Inherited from generic model API .	JSON object (in request body)
`/acls/id`	GET	Deny	Find ACLs by ID: Return data for the specified acls instance ID. Inherited from generic model API .	*id*, the ACL instance ID (in URI path)
`/acls/id`	PUT	Deny	Update attributes for specified acls ID and persist. Inherited from generic model API .	Query parameters: • data - An object containing property name/value pairs • *id* - The model id
`/acls/id`	DELETE	Deny	Delete ACLs with specified instance ID. Inherited from generic model API .	*id*, acls ID (in URI path)

`/acls/id/exists`	GET	Deny	Check instance existence: Return true if specified acls ID exists. Inherited from generic model API .	URI path: • *id* - Model instance ID
`/acls/count`	GET	Deny	Return the number of ACL instances that matches specified where clause. Inherited from generic model API .	Where filter specified in query parameter
`/acls/findOne`	GET	Deny	Find first ACL instance that matches specified filter. Inherited from generic model API .	Same as Find matching instances.

Application REST API

All of the endpoints in the Application REST API are inherited from the generic model API. The reference is provided here for convenience.

Quick reference

URI Pattern	HTTP Verb	Default Permission	Description	Arguments
`/applications`	POST	Allow	Add application instance and persist to data source. Inherited from generic model API.	JSON object (in request body)
`/applications`	GET	Deny	Find all instances of applications that match specified filter. Inherited from generic model API .	One or more filters in query parameters: • where • include • order • limit • skip / offset • fields
`/applications`	PUT	Deny	Update / insert application instance and persist to data source. Inherited from generic model API .	JSON object (in request body)
`/applications/id`	GET	Deny	Find applications by ID: Return data for the specified applications instance ID. Inherited from generic model API .	*id*, the application instance ID (in URI path)
`/applications/id`	PUT	Deny	Update attributes for specified applications ID and persist. Inherited from generic model API .	Query parameters: • data - An object containing property name/value pairs • *id* - The model id
`/applications/id`	DELETE	Deny	Delete application with specified instance ID. Inherited from generic model API .	*id*, application ID (in URI path)
`/applications/id/exists`	GET	Deny	Check instance existence: Return true if specified application ID exists. Inherited from generic model API .	URI path: • *id* - Model instance ID
`/applications/count`	GET	Deny	Return the number of application instances that matches specified where clause. Inherited from generic model API.	Where filter specified in query parameter
`/applications/findOne`	GET	Deny	Find first application instance that matches specified filter. Inherited from generic model API .	Same as Find matching instances.

Email REST API

- Operation name

Quick reference

URI Pattern	HTTP Verb	Default Permission	Action	Arguments

`/foo/bar/baz`	One of: GET, POST, PUT, DELETE	Allow / Deny	Description plus link to section with full reference. NOTE: Rand will add links to sections.	List arguments in POST body, query params, or path.

Operation name

Brief description goes here.

```
POST /modelName
```

Arguments

- List of all arguments in POST data or query string

Example

Request:

```
curl -X POST -H "Content-Type:application/json"
-d '{... JSON ... }'
http://localhost:3000/foo
```

Response:

```
// Response JSON
```

Errors

List error codes and return JSON format if applicable.

Installation REST API

All of the endpoints in the table below are inherited from Model REST API, except for the following:

- Find installations by app ID
- Find installations by user ID

Quick reference

URI Pattern	HTTP Verb	Default Permission	Description	Arguments
`/Installations/byApp`	GET		Find installations by app ID	Query parameters: • deviceType • appID • appVersion
`/Installations/byUser`	GET		Find installations by user ID	Query parameters: • deviceType • userId

`/Installations`	POST		Add installation instance and persist to data source. Inherited from generic model API.	JSON object (in request body)
`/Installations`	GET		Find all instances of installations that match specified filter. Inherited from generic model API .	One or more filters in query parameters: • where • include • order • limit • skip / offset • fields
`/Installations`	PUT		Update / insert installation instance and persist to data source. Inherited from generic model API .	JSON object (in request body)
`/Installations/id`	GET		Find installation by ID: Return data for the specified instance ID. Inherited from generic model API .	*id*, the installation ID (in URI path)
`/Installations/id`	PUT		Update installation attributes for specified installation I D and persist. Inherited from generic model API .	Query parameters: • data An object containing property name/value pairs • *id* - The installation ID
`/Installations/id`	DELETE		Delete installation with specified instance ID. Inherited from generic model API .	*id*, installation ID (in URI path)
`/Installations/count`	GET		Return number of installation instances that match specified where clause. Inherited from generic model API.	Query parameter: "where" filter.
`/Installations/id/exists`	GET		Check instance existence: Return true if specified user ID exists. Inherited from generic model API .	URI path: *id* installation ID
`/Installations/findOne`	GET		Find first installation instance that matches specified filter. Inherited from generic model API .	Same as Find matching instances.

Find installations by app ID

Return JSON array of installations of specified app ID that also match the additional specified arguments (if any).

```
GET /Installations/byApp
```

Arguments

All arguments are in query string:

- deviceType
- appId
- appVersion

Example

Request:

```
curl -X GET
http://localhost:3000/Installation/byApp?appId=KrushedKandy&deviceType=ios
```

Response:

```
[
  {
    "id": "1",
    "appId": "KrushedKandy",
    "userId": "raymond",
    "deviceType": "ios",
    "deviceToken": "756244503c9f95b49d7ff82120dc193ca1e3a7cb56f60c2ef2a19241e8f33305",
    "subscriptions": [],
    "created": "2014-01-09T23:18:57.194Z",
    "modified": "2014-01-09T23:18:57.194Z"
  },
...
]
```

Errors

Find installations by user ID

Return JSON array of installations by specified user ID that also match the additional specified argument (if provided).

```
GET /Installations/byUser
```

Arguments

Argumens are in query string:

- deviceType
- userId

Example

Request:

```
curl -X GET
http://localhost:3000/Installations/byUser?userId=raymond
```

Response:

```
[
  {
    "id": "1",
    "appId": "MyLoopBackApp",
    "userId": "raymond",
    "deviceType": "ios",
    "deviceToken": "756244503c9f95b49d7ff82120dc193ca1e3a7cb56f60c2ef2a19241e8f33305",
    "subscriptions": [],
    "created": "2014-01-09T23:18:57.194Z",
    "modified": "2014-01-09T23:18:57.194Z"
  }
]
```

Errors

Model REST API

The model names in the REST API are always the plural form of the model name. By default this is simply the name with an "s" appended; for example, if the model is "car" then "cars" is the plural form. You can customize the plural form in the model definition; see Model definition JSON file.

Related articles

- Creating models
- Extending models
- Creating model relations
- Querying models
- Model definition JSON file
- Model REST API

By default, LoopBack uses `/api` as the URI root for the REST API. You can change this by changing the `restApiRoot` property in the application `/server/config.json` file. See config.json for more information.

Add model instance

Create a new instance of the model and persist it to the data source.

POST /*modelName*

Arguments

- Form data - Model instance data

Example

Request:

```
curl -X POST -H "Content-Type:application/json" \
-d '{"name": "L1", "street": "107 S B St", "city": "San Mateo", "zipcode": "94401"}' \
http://localhost:3000/api/locations
```

Response:

```
{
  "id": "96",
  "street": "107 S B St",
  "city": "San Mateo",
  "zipcode": 94401,
  "name": "L1",
  "geo": {
    "lat": 37.5670042,
    "lng": -122.3240212
  }
}
```

Errors

None

Update / insert instance

Update an existing model instance or insert a new one into the data source. The update will override any specified attributes in the request data object. It won't remove existing ones unless the value is set to null.

PUT /*modelName*

Arguments

- Form data - model instance data in JSON format.

Examples

Request - insert:

```
curl -X PUT -H "Content-Type:application/json" \
-d '{"name": "L1", "street": "107 S B St", "city": "San Mateo", "zipcode": "94401"}' \
http://localhost:3000/api/locations
```

Response:

```
{
  "id": "98",
  "street": "107 S B St",
  "city": "San Mateo",
  "zipcode": 94401,
  "name": "L1",
  "geo": {
    "lat": 37.5670042,
    "lng": -122.3240212
  }
}
```

Request - update:

```
curl -X PUT -H "Content-Type:applicatin/json" \
-d '{"id": "98", "name": "L4", "street": "107 S B St", "city": "San Mateo", \
"zipcode": "94401"}' http://localhost:3000/api/locations
```

Response:

```
{
  "id": "98",
  "street": "107 S B St",
  "city": "San Mateo",
  "zipcode": 94401,
  "name": "L4"
}
```

Errors

None

Check instance existence

Check whether a model instance exists by ID in the data source.

```
GET /modelName/modelID/exists
```

Arguments

- *modelID* - model ID

Example

Request:

```
curl http://localhost:3000/api/locations/88/exists
```

Response:

```
{
    "exists": true
}
```

Errors

None

Find instance by ID

Find a model instance by ID from the data source.

```
GET /modelName/modelID
```

Arguments

- **modelID** - Model ID

Example

Request:

```
curl http://localhost:3000/api/locations/88
```

Response:

```
{
    "id": "88",
    "street": "390 Lang Road",
    "city": "Burlingame",
    "zipcode": 94010,
    "name": "Bay Area Firearms",
    "geo": {
        "lat": 37.5874391,
        "lng": -122.3381437
    }
}
```

Errors

None

Find matching instances

Find all instances of the model matched by filter from the data source.

GET /*modelName*?filter=[*filterType1*]=<*val1*>&filter[*filterType2*]=<*val2*>...

Arguments

Pass the arguments as the value of the `find` HTTP query parameters, where:

- *filterType1*, *filterType2*, and so on, are the filter types.
- *val1*, *val2* are the corresponding values.

See Querying models for an explanation of filter syntax.

Example

Request without filter:

```
curl http://localhost:3000/api/locations
```

Request with a filter to limit response to two records:

```
/locations?filter[limit]=2
```

With curl, you must URL-encode `[` as `%5B`, and `]` as `%5D`, so the actual curl command would be:

```
curl http://localhost:3000/api/locations?filter%5Blimit%5D=2
```

Response:

```
[
  {
    "id": "87",
    "street": "7153 East Thomas Road",
    "city": "Scottsdale",
    "zipcode": 85251,
    "name": "Phoenix Equipment Rentals",
    "geo": {
      "lat": 33.48034450000001,
      "lng": -111.9271738
    }
  },
  {
    "id": "88",
    "street": "390 Lang Road",
    "city": "Burlingame",
    "zipcode": 94010,
    "name": "Bay Area Firearms",
    "geo": {
      "lat": 37.5874391,
      "lng": -122.3381437
    }
  }
]
```

Errors

None

Find first instance

Find first instance of the model matched by filter from the data source.

GET /modelName/findOne?filter=[*filterType1*]=<*val1*>&filter[*filterType2*]=<*val2*>...

Arguments

Query parameters:

- **filter** - Filter that defines where, order, fields, skip, and limit. It's same as find's filter argument. See Querying models details.

Example

Request:

```
/locations/findOne?filter[where][city]=Scottsdale
```

With curl, you must URL-encode `[` as `%5B`, and `]` as `%5D`, so the actual curl command would be:

```
curl
http://localhost:3000/api/locations/findOne?filter%5Bwhere%5D%5Bcity%5D=Scottsdale
```

Response:

```
{
  "id": "87",
  "street": "7153 East Thomas Road",
  "city": "Scottsdale",
  "zipcode": 85251,
  "name": "Phoenix Equipment Rentals",
  "geo": {
    "lat": 33.48034450000001,
    "lng": -111.9271738
  }
}
```

Errors

None

Delete model instance

Delete a model instance by ID from the data source

```
DELETE /modelName/modelID
```

Arguments

- **modelID** - model instance ID

Example

Request:

```
curl -X DELETE http://localhost:3000/api/locations/88
```

Response:

Example TBD.

Errors

None

Get instance count

Count instances of the model from the data source matched by where clause.

```
GET /modelName/count?where[property]=value
```

Arguments

- **where** - criteria to match model instances

Example

Request - count without "where" filter

```
curl http://localhost:3000/api/locations/count
```

Request - count with a "where" filter

```
curl http://localhost:3000/api/locations/count?where%5bcity%5d=Burlingame
```

Response:

```
{
    count: 6
}
```

Errors

None

Update model instance attributes

Update attributes of a model instance and persist into the data source.

PUT /*model*/*modelID*

Arguments

- data - An object containing property name/value pairs
- *modelID* - The model ID

Example

Request:

```
curl -X PUT -H "Content-Type:application/json" -d '{"name": "L2"}'
http://localhost:3000/api/locations/88
```

Response:

```
{
  "id": "88",
  "street": "390 Lang Road",
  "city": "Burlingame",
  "zipcode": 94010,
  "name": "L2",
  "geo": {
    "lat": 37.5874391,
    "lng": -122.3381437
  },
  "state": "CA"
}
```

Errors

- 404 - No instance found for the given ID.

Model relations REST API

⚠ These endpoints are part of the Model REST API, but are presented in a separate page for ease of reference.

Get related model instances

Follow the relations from one model to another one to get instances of the associated model.

```
GET /model1-name/instanceID/model2-name
```

Arguments

- *instanceID* - ID of instance in model1.
- *model1-name* - name of first model.
- *model2-name* - name of second related model.

Example

Request:

```
GET http://localhost:3000/locations/88/inventory
```

Response:

```
[
  {
    "productId": "2",
    "locationId": "88",
    "available": 10,
    "total": 10
  },
  {
    "productId": "3",
    "locationId": "88",
    "available": 1,
    "total": 1
  }
]
```

Get hasMany related model instances

List related model instances for specified *model-name* identified by the *instance-ID*, for hasMany relationship.

```
GET /model-name/instanceID/hasManyRelationName
```

Create hasMany related model instance

Create a related model instance for specified *model-name* identified by *instance-ID*, for hasMany relationship.

POST */model1-name/instanceID/hasManyRelationName*

Delete hasMany related model instances

Delete related model instances for specified *model-name* identified by *instance-ID*, for hasMany relationship.

DELETE */model1-name/instance-ID/hasMany-relation-name*

List belongsTo related model instances

List the related model instances for the given model identified by *instance-ID*, for hasMany relationship.

GET /*model-name*/*instance-ID*/*belongsToRelationName*

Aggregate models following relations

It's often desirable to include related model instances in the response to a query so that the client doesn't have to make multiple calls.

GET /*model1-name?filter[include]=...*

Arguments

- *include* - The object that describes a hierarchy of relations to be included

Example

Retrieve all members including the posts with the following request:

```
GET /api/members?filter[include]=posts
```

The API returns the following JSON:

```
[
  {
    "name": "Member A",
    "age": 21,
    "id": 1,
    "posts": [
      {
        "title": "Post A",
        "id": 1,
        "memberId": 1
      },
      {
        "title": "Post B",
        "id": 2,
        "memberId": 1
      },
      {
        "title": "Post C",
        "id": 3,
        "memberId": 1
      }
    ]
  },
  {
    "name": "Member B",
    "age": 22,
    "id": 2,
    "posts": [
      {
        "title": "Post D",
        "id": 4,
        "memberId": 2
      }
    ]
  },
... ]
```


The following request retrieves all members, including the posts, which further includes the author:

`GET /api/members?filter[include][posts]=author`

The API returns the following JSON:

```
[
  {
    "name": "Member A",
    "age": 21,
    "id": 1,
    "posts": [
      {
        "title": "Post A",
        "id": 1,
        "memberId": 1,
        "author": {
          "name": "Member A",
          "age": 21,
          "id": 1
        }
      },
      {
        "title": "Post B",
        "id": 2,
        "memberId": 1,
        "author": {
          "name": "Member A",
          "age": 21,
          "id": 1
        }
      },
      {
        "title": "Post C",
        "id": 3,
        "memberId": 1,
        "author": {
          "name": "Member A",
          "age": 21,
          "id": 1
        }
      }
    ]
  },
  {
    "name": "Member B",
    "age": 22,
    "id": 2,
    "posts": [
      {
        "title": "Post D",
        "id": 4,
        "memberId": 2,
        "author": {
          "name": "Member B",
          "age": 22,
          "id": 2
        }
      }
    ]
  }, ... ]
```

The following request retrieves all members who are 21 years old, including the posts, which further includes the author:

```
GET /api/members?filter[include][posts]=author&filter[where][age]=21
```

The API returns the following JSON:


```
[
  {
    "name": "Member A",
    "age": 21,
    "id": 1,
    "posts": [
      {
        "title": "Post A",
        "id": 1,
        "memberId": 1,
        "author": {
          "name": "Member A",
          "age": 21,
          "id": 1
        }
      },
      {
        "title": "Post B",
        "id": 2,
        "memberId": 1,
        "author": {
          "name": "Member A",
          "age": 21,
          "id": 1
        }
      },
      {
        "title": "Post C",
        "id": 3,
        "memberId": 1,
        "author": {
          "name": "Member A",
          "age": 21,
          "id": 1
        }
      }
    ]
  }
]
```


The following request retrieves two members, including the posts, which further includes the author:

```
GET /api/members?filter[include][posts]=author&filter[limit]=2
```

The API returns the following JSON:

```
[
  {
    "name": "Member A",
    "age": 21,
    "id": 1,
    "posts": [
      {
        "title": "Post A",
        "id": 1,
        "memberId": 1,
        "author": {
          "name": "Member A",
          "age": 21,
          "id": 1
        }
      },
      {
        "title": "Post B",
        "id": 2,
        "memberId": 1,
        "author": {
          "name": "Member A",
          "age": 21,
          "id": 1
        }
      },
      {
        "title": "Post C",
        "id": 3,
        "memberId": 1,
        "author": {
          "name": "Member A",
          "age": 21,
          "id": 1
        }
      }
    ]
  },
  {
    "name": "Member B",
    "age": 22,
    "id": 2,
    "posts": [
      {
        "title": "Post D",
        "id": 4,
        "memberId": 2,
        "author": {
          "name": "Member B",
          "age": 22,
          "id": 2
        }
      }
    ]
  }
]
```

The following request retrieves all members, including the posts and passports.

`GET /api/members?filter[include]=posts&filter[include]=passports`

The API returns the following JSON:


```
[
  {
    "name": "Member A",
    "age": 21,
    "id": 1,
    "posts": [
      {
        "title": "Post A",
        "id": 1,
        "memberId": 1
      },
      {
        "title": "Post B",
        "id": 2,
        "memberId": 1
      },
      {
        "title": "Post C",
        "id": 3,
        "memberId": 1
      }
    ],
    "passports": [
      {
        "number": "1",
        "id": 1,
        "ownerId": 1
      }
    ]
  },
  {
    "name": "Member B",
    "age": 22,
    "id": 2,
    "posts": [
      {
        "title": "Post D",
        "id": 4,
        "memberId": 2
      }
    ],
    "passports": [
      {
        "number": "2",
        "id": 2,
        "ownerId": 2
      }
    ]
  }, ... ]
```


Errors

None

Push Notification REST API

All of the endpoints in the table below are inherited from Model REST API, except for Send push notification.

URI Pattern	HTTP Verb	Default Permission	Description	Arguments
`/push`	POST	Allow / Deny	Send a push notification by installation query	Query parameters: deviceQuery Request body: notification

Send push notification

Send a pus notification by installation query.

```
POST /push
```

Arguments

- deviceQuery - Object; query parameter.
- notification - Object; request body.

Example

Request:

```
curl -X POST -H "Content-Type:application/json"
-d '{"badge" : 5, "sound": "ping.aiff", "alert": "Hello", "messageFrom": "Ray"}'
http://localhost:3000/push?deviceQuery[userId]=1
```

Response code: 200

Response body:

```
{}
```

Errors

Role REST API

All of the endpoints in the Role REST API are inherited from the generic model API. The reference is provided here for convenience.

Quick reference

URI Pattern	HTTP Verb	Default Permission	Description	Arguments
`/roles`	POST	Allow	Add role instance and persist to data source. Inherited from generic model API.	JSON object (in request body)

`/roles`	GET	Deny	Find all instances of role that match specified filter. Inherited from generic model API .	One or more filters in query parameters: • where • include • order • limit • skip / offset • fields
`/roles`	PUT	Deny	Update / insert role instance and persist to data source. Inherited from generic model API .	JSON object (in request body)
`/roles/id`	GET	Deny	Find role by ID: Return data for the specified role instance ID. Inherited from generic model API .	*id*, the role instance ID (in URI path)
`/roles/id`	PUT	Deny	Update attributes for specified role ID and persist. Inherited from generic model API .	Query parameters: • data - An object containing property name/value pairs • *id* - The model id
`/roles/id`	DELETE	Deny	Delete role with specified instance ID. Inherited from generic model API .	*id*, role ID (in URI path)
`/roles/id/exists`	GET	Deny	Check instance existence: Return true if specified role ID exists. Inherited from generic model API .	URI path: • *id* - Model instance ID
`/roles/count`	GET	Deny	Return the number of role instances that matches specified where clause. Inherited from generic model API.	Where filter specified in query parameter
`/roles/findOne`	GET	Deny	Find first role instance that matches specified filter. Inherited from generic model API .	Same as Find matching instances.

User REST API

All of the endpoints in the table below are inherited from Model REST API, except for the following:

- Log in user
- Log out user
- Confirm email address
- Reset password

Quick reference

URI Pattern	HTTP Verb	Default Permission	Description	Arguments
`/users`	POST	Allow	Add user instance and persist to data source. Inherited from generic model API.	JSON object (in request body)
`/users`	GET	Deny	Find all instances of users that match specified filter. Inh erited from generic model API .	One or more filters in query parameters: • where • include • order • limit • skip / offset • fields
`/users`	PUT	Deny	Update / insert user instance and persist to data source. Inherited from generic model API .	JSON object (in request body)
`/users/id`	GET	Deny	Find user by ID: Return data for the specified user ID. In herited from generic model API .	*id*, the user ID (in URI path)

`/users/id`	PUT	Deny	Update user attributes for specified user ID and persist. Inherited from generic model API .	Query parameters: • data An object containing property name/value pairs • *id* The model id
`/users/id`	DELETE	Deny	Delete user with specified instance ID. Inherited from generic model API .	*id*, user ID (in URI path)
`/users/accessToken`	POST	Deny		
`/users/id/accessTokens`	GET	Deny	Returns access token for specified user ID.	• *id*, user ID, in URI path • where in query parameters
`/users/id/accessTokens`	POST	Deny	Create access token for specified user ID.	*id*, user ID, in URI path
`/users/id/accessTokens`	DELETE	Deny	Delete access token for specified user ID.	*id*, user ID, in URI path
`/users/confirm`	GET	Deny	Confirm email address for specified user.	Query parameters: • uid • token • redirect
`/users/count`	GET	Deny	Return number of user instances that match specified where clause. Inherited from generic model API.	Where filter specified in query parameter
`/users/id/exists`	GET	Deny	Check instance existence: Return true if specified user ID exists. Inherited from generic model API .	URI path: • *users* - Model name • *id* - Model instance ID
`/users/findOne`	GET	Deny	Find first user instance that matches specified filter. Inherited from generic model API .	Same as Find matching instances.
`/users/login`	POST	Allow	Log in the specified user.	Username and password in POST body.
`/users/logout`	POST	Allow	Log out the specified user.	Access token in POST body.
`/users/reset`	POST		Reset password for the specified user.	In POST body

Log in user

```
POST /users/login
```

You must provide a username and password over REST. To ensure these values are encrypted, include these as part of the body and make sure you are serving your app over HTTPS (through a proxy or using the HTTPS node server).

Parameters

POST payload:

```
{
  "email": "foo@bar.com",
  "password": "bar"
}
```

Return value

```
200 OK
  {
    "sid": "1234abcdefg",
    "uid": "123"
  }
```

Log out user

```
POST /users/logout
```

Parameters

POST payload:

```
{
  "sid": "<accessToken id from user login>"
}
```

Confirm email address

Require a user to verify their email address before being able to login. This will send an email to the user containing a link to verify their address. Once the user follows the link they will be redirected to web root (`"/"`) and will be able to login normally.

Reset password

```
POST  /users/reset
```

Parameters

POST payload:

```
{
    "email": "foo@bar.com"
  }
  ...
```

Return value

```
200 OK
```

You must the handle the `'resetPasswordRequest'` event this on the server to send a reset email containing an access token to the correct user. The example below shows a basic setup for sending the reset email.

```
User.on('resetPasswordRequest', function (info) {
  console.log(info.email); // the email of the requested user
  console.log(info.accessToken.id); // the temp access token to allow password reset

  // requires AccessToken.belongsTo(User)
  info.accessToken.user(function (err, user) {
    console.log(user); // the actual user
    var emailData = {
      user: user,
      accessToken: accessToken
    };

    // this email should include a link to a page with a form to
    // change the password using the access token in the email
    Email.send({
      to: user.email,
      subject: 'Reset Your Password',
      text: loopback.template('reset-template.txt.ejs')(emailData),
      html: loopback.template('reset-template.html.ejs')(emailData)
    });
  });
});
```

Advanced topics: models

Creating models manually

> ⚠ In general, best practice is to use the LoopBack model generator to create models. The techniques described here are suitable for advanced users.

Instead of using the LoopBack model generator, you can create a model programmatically either in JavaScript. In general, the recommended way to implement or extend a model is to create a file with the same name as the model (for example, `products.js`) in the `/common/models` directory to contain custom logic and model extensions.

Creating a static model

The PersistedModel object corresponds to a database-backed model, and acts in some sense like an "interface" that connectors implement.

For persisted models backed by database, to extend an existing model:

```
var PersistedModel = require('loopback').PersistedModel;
var Product = PersistedModel.extend('product');
```

Creating a dynamic model

For dynamic models that don't have a schema (such as REST or SOAP services):

```
var Model = require('loopback').Model;
var Product = Model.extend('product');
```

Consider an e-commerce app with `Product` and `Inventory` models. A mobile client could use the `Product` model API to search through all of the products in a database. A client could join the `Product` and `Inventory` data to determine what products are in stock, or the `Product` mod el could provide a server-side function (or remote method) that aggregates this information.

The following code creates product and inventory models:

```
var Model = require('loopback').Model;
var Product = Model.extend('product');
var Inventory = Model.extend('customer');
```

The above code creates two dynamic models, appropriate when data is "free form." However, some data sources, such as relational databases, require schemas. Schemas are also valuable to enable data exchange and validate or sanitize data from clients; see Sanitizing and validating Models.

Getting a reference to a model

Once you've created your models, you will need to get a reference to them in your code. There are two ways to do this. To get a reference to a model created in JSON:

```
var loopback = require('loopback');
var app = module.exports = loopback();
 ...
myModel = app.models.myModelName;
```

Then *`myModel`* will contain a reference to the model called *`myModelName`* .

For example, if thre is a model named "products" then you can get a reference to it as follows:

```
productModel = app.models.products;
```

To get all models, including those created programmatically, instead use:

```
myModel = loopback.getModel('myModelName');
```

Creating dynamic models

Dynamic or open models don't have a predefined schema. 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');

app.model(FormModel);
app.use(loopback.rest());

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 app 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
```


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'll see the newly created instance:

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

For the complete list of REST APIs that LoopBack scaffolds for a model, see Built-in models REST API.

To build and run this project:

```
$ slc loopback free-form
$ cd free-form
$ slc loopback:model form
$ npm install
$ slc run
```

Now open a browser and load http://localhost:3000/explorer.

You'll see the LoopBack API explorer. The interface is pretty straightforward: feel free to play with it.

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.

Creating static models

Static models have schema definitions, for example models based on relational databases.

- Mocking up a model
- Trying out CRUD operations
- Exposing the model over REST
- Connecting to a database

Mocking up a model

When working with models that have schema definitions, you might want to create a working REST API before implementing the server-side logic. To do this, define a model first and use an in-memory data source to mock up the data access. You'll get a working REST API without writing a lot of 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, see LoopBack Definition Language.

Trying out CRUD operations

You can now test create, read, update, and delete (CRUD) operations on the model. For example, 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());
            });
        });
    });
});
```

Exposing the model over REST

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');
});
```

For more information, see Exposing models over a REST API.

Connecting to a database

Until now the data access has been backed by an in-memory store. To make your data persistent, simply replace it (for example, 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, see Data sources and connectors.

Adding logic to a model

Overview

Models describe the shape of data. To leverage the data, you must add logic to the model such as:

- Performing create, read, update, and delete (CRUD) operations.
- Adding behavior around a model instance.
- Adding service operations using the model as the context.

There are a few ways to add methods to a model constructor:

Creating a model constructor from a data source

A LoopBack data source injects methods on the model.

```
 var DataSource = require('loopback-datasource-juggler').DataSource;
var ds = new DataSource('memory');

// Compile the user model definition into a JavaScript constructor
var User = ds.define('User', UserDefinition);

// Create a new instance of User
User.create({id: 1, firstName: 'John', lastName: 'Smith'}, function(err, user) {
    console.log(user); // The newly created user instance
    User.findById(1, function(err, user) {
        console.log(user); // The user instance for id 1
        user.firstName = 'John1'; // Change the property
        user.save(function(err, user) {
            console.log(user); // The modified user instance for id 1
        });
    };
});
```

Attaching the model to a data source

A plain model constructor created from `ModelBuilder` can be attached a `DataSource`.

```
var DataSource = require('loopback-datasource-juggler').DataSource;
var ds = new DataSource('memory');

User.attachTo(ds); // The CRUD methods will be mixed into the User constructor
```

Manually adding methods to the model constructor

Static methods can be added by declaring a function as a member of the model constructor. Within a class method, other class methods can be called using the model as usual.

```
 // Define a static method
User.findByLastName = function(lastName, cb) {
    User.find({where: {lastName: lastName}}, cb);
};

User.findByLastName('Smith', function(err, users) {
    console.log(users); // Print an array of user instances
});
```

Add instance methods to the prototype. Within instance methods, refer to the model instance itself using `this`.

```
 // Define a prototype method
User.prototype.getFullName = function () {
    return this.firstName + ' ' + this.lastName;
};

var user = new User({id: 1, firstName: 'John', lastName: 'Smith'});
console.log(user.getFullName()); // 'John Smith'
```

Creating data sources and attaching models

See also the Datasource Juggler API reference.

`DataSource` is a factory for model classes. Use a connector to connect a DataSource to a specific database or other backend system.

All model classes within single datasource shares same connector type and one database connection. But it's possible to use more than one datasource to connect with different databases.

Creating data sources

`DataSource` constructor available on `loopback-datasource-juggler` module:

```
var DataSource = require('loopback-datasource-juggler').DataSource;
```

The DataSource constructor accepts two arguments:

- The first argument is connector. It could be connector name or connector package.
- The second argument is optional settings.

For example, with no settings argument:

```
var dataSourceByConnectorName = new DataSource('memory');
```

The settings object format and defaults depends on specific connector, but common fields are:

- `host`: Database host
- `port`: Database port
- `username`: Username to connect to database
- `password`: Password to connect to database
- `database`: Database name
- `debug`: Turn on verbose mode to debug db queries and lifecycle

Creating connected models

To define models connected to a DataSource in code, use the `dataSource.define()` method. It accepts three arguments:

- **model name**: String name in camel-case with first upper-case letter. This name will be used later to access model.
- **properties**: Object with property type definitions. Key is property name, value is type definition that is one of:
  - Type of property (String, Number, Date, Boolean), for example: `email: String`
  - Object with {type: String|Number|..., index: true|false} format; for example `email: { type: String, limit: 150, index: true }`
- **settings**: Object with model-wide settings such as `tableName`.

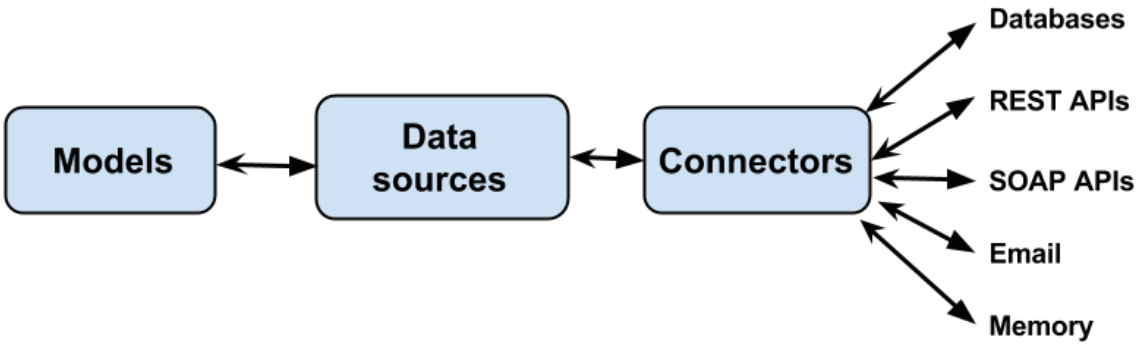
For more information, see LoopBack Definition Language.

Data sources and connectors

Overview

LoopBack is centered around *models* that represent data and behaviors. Models connect to backend systems such as databases via data sources that typically provide create, retrieve, update, and delete (CRUD) functions. LoopBack also generalizes other backend services, such as REST APIs, SOAP web services, and storage services, as data sources.

Data sources are backed by connectors that implement the data exchange logic using database drivers or other client APIs. In general, applications don't use connectors directly, rather they go through data sources using the DataSource and Model / PersistedModel APIs.



Connectors

The following LoopBack connectors are available:

Connector	Module	Installation
Memory connector	Built in to LoopBack	Not required
Email	Built in to LoopBack	Not required
MongoDB	loopback-connector-mongodb	`npm install --save loopback-connector-mongodb`
MySQL	loopback-connector-mysql	`npm install --save loopback-connector-mysql`
Oracle	loopback-connector-oracle	`npm install --save loopback-connector-oracle`
PostgreSQL	loopback-connector-postgresql	`npm install --save loopback-connector-postgersql`
REST	loopback-connector-rest	`npm install --save loopback-connector-rest`
SOAP	loopback-connector-soap	`npm install --save loopback-connector-soap`
SQL Server	loopback-connector-mssql	`npm install --save loopback-connector-mssql`

Installing a connector

Run `npm install --save` for the connector module to add the dependency to `package.json`; for example:

```
...
  "dependencies": {
    "loopback-connector-oracle": "latest"
  }
...
```

Creating a data source for a connector

In general, use the LoopBack datasource generator to create a new data source. You can also create a data source programmatically. Data source properties depend on the specific data source being used. However, data sources for database connectors (Oracle, MySQL, PostgreSQL, MongoDB, and so on) share a common set of properties, as described in the following table.

Property	Description
database	Database name
debug	Boolean. If true, turn on verbose mode to debug database queries and lifecycle.
host	Database host name
password	Password to connect to database
port	Database TCP port
username	Username to connect to database

Memory connector

Overview

LoopBack's built-in memory connector enables you to test your application without connecting to an actual persistent data source such as a database. Although the memory connector is very well tested it is not suitable for production.

The memory connector supports:

- Standard query and create, read, update, and delete (CRUD) operations, so you can test models against an in-memory data source.
- Geo-filtering when using the `find()` operation with an attached model. See GeoPoint class for more information on geo-filtering.

Limitations
The memory connector is designed for development and testing of a single-process application without setting up a database. It cannot be used in a cluster as the worker processes will have their own isolated data not shared in the cluster. The file option doesn't help since the worker processes will race to overwrite each other's data.

Creating a data source

By default, an application created with the LoopBack application generator has a memory data source defined; for example:

```
"db": {
    "name": "db",
    "connector": "memory"
}
```

You can also use the LoopBack datasource generator to add a new memory data source to your application.

Data persistance

By default, data in the memory connector are transient. When an application using the memory connector exits, all model instances are lost. To maintain data across application restarts, specify a JSON file in which to store the data with the `file` property when creating the data source. For example:

```
var memory = loopback.createDataSource({
  connector: loopback.Memory,
  file: "mydata.json"
});
```

When the application exits, the memory connector will then store data in the `mydata.json` file, and when it restarts will load the saved data from

that file.

MongoDB connector

Installation

In your application root directory, enter:

```
$ npm install loopback-connector-mongodb --save
```

This will install the module from npm and add it as a dependency to the application's package.json file.

Using the MongoDB connector

Use the LoopBack datasource generator to add a MongoDB data source to your application.

LoopBack currently does not support property mapping for MongoDB at the moment; you can customize only collection names.

Configuring a MongoDB data source

Configure a MongoDB data source in an application's `/server/datasources.json` file.

The "connector" property can be either "mongodb" or "loopback-connector-mongodb".

For example:

Example datasources.json file

```
{
  "mongodb_dev": {
    "name": "mongodb_dev",
    "connector": "mongodb",
    "host": "127.0.0.1",
    "database": "devDB",
    "username": "devUser",
    "password": "devPassword",
    "port": 27017
  },
  "mongodb_staging": {
    "name": "mongodb_staging",
    "connector": "mongodb",
    "host": "127.0.0.1",
    "database": "stagingDB",
    "username": "stagingUser",
    "password": "stagingPassword",
    "port": 27017
  }
}
```

⚠ Username and password are only required if the MongoDB server has authentication enabled.

Replica set configuration

The LoopBack MongoDB connector supports the replica set configuration using the MongoDB connection string URI format. For example, here is a snippet for the data source configuration:

```
{
 “connector": “mongodb”,
 “url”: "mongodb://example1.com,example2.com,example3.com/?readPreference=secondary"
}
```

About MongoDB _id field

MongoDB uses a specific ID field with BSON `ObjectID` type, named `_id`

The MongoDB connector does not expose the MongoDB `_id` field, to be consistent with other connectors. Instead, it is transparently mapped to the `id` field, which is declared by default in the model if you do not define any `id`.

To access the `_id` property, you must define it explicitly as your model ID, along with its type; For example:

```
var ds = app.dataSources.db;
MyModel = ds.createModel('mymodel', {
    _id: { type: ds.ObjectID, id: true }
});
```

Example with a Number _id :

```
MyModel = ds.createModel('mymodel', {
    _id: { type: Number, id: true }
});
```

Query with logical operators (since v1.2.3)

MongoDB supports query with logical operators such as $and, $or, and $nor. See http://docs.mongodb.org/manual/reference/operator/query-logical/ for more information.

To use the logical operators with LoopBack's query filter, you can use where clause as follows:

```
// Find posts that have title = 'My Post' and content = 'Hello'
Post.find({where: {and: [{title: 'My Post'}, {content: 'Hello'}]}}, function (err,
posts) {
...
});

// Find posts that either have title = 'My Post' or content = 'Hello'
Post.find({where: {or: [{title: 'My Post'}, {content: 'Hello1'}]}}, function (err,
posts) {
...
});

// Find posts that neither have title = 'My Post1' nor content = 'Hello1'
Post.find({where: {nor: [{title: 'My Post1'}, {content: 'Hello1'}]}}, function (err,
posts) {
...
});
```

Using MongoLab

If you are using MongoLab to host your MongoDB database, use the LoopBack `url` property to configure your data source, since the connection string is dynamically generated. For example, the entry in `datasources.json` might look like this:

```
"mongodb": {
   "defaultForType": "mongodb",
   "connector": "loopback-connector-mongodb",
   "url": "mongodb://localhost:27017/mydb"
}
```

For information on how to get your connection URI, see the MongoLab documentation.

MySQL connector

See also:

- Example application with MySQL connector
- Discovering models from databases
- Database discovery API

Installation

```
$ npm install loopback-connector-mysql --save
```

This will install the module from npm and add it as a dependency to the application's package.json file.

Creating a data source

Use the LoopBack datasource generator to add a MySQL data source to your application. The entry in the application's `/server/datasources.json` will look like this:

```
"mydb": {
    "name": "mydb",
    "connector": "mysql",
    ...
}
```

The "connector" property can be either "loopback-connector-mysql" or "mysql".

In addition to the standard set of persistent data source properties, you can pass additional parameters supported by `node-mysql`, for example `password` and `collation`. `Collation` currently defaults to `utf8_general_ci`. The `collation` value will also be used to derive the connection charset.

Type mappings

See LoopBack types for details on LoopBack's data types.

LoopBack to MySQL types

LoopBack Type	MySQL Type
String/JSON	VARCHAR
Text	TEXT
Number	INT
Date	DATETIME
Boolean	TINYINT(1)
GeoPoint object	POINT
Enum	ENUM

MySQL to LoopBack types

MySQL Type	LoopBack Type
CHAR	String
CHAR(1)	Boolean
VARCHAR TINYTEXT MEDIUMTEXT LONGTEXT TEXT ENUM SET	String
TINYBLOB MEDIUMBLOB LONGBLOB BLOB BINARY VARBINARY BIT	Node.js Buffer object

TINYINT SMALLINT INT MEDIUMINT YEAR FLOAT DOUBLE NUMERIC DECIMAL	Number
DATE TIMESTAMP DATETIME	Date

Using the datatype field/column option with MySQL

`loopback-connector-mysql` allows mapping of LoopBack model properties to MySQL columns using the 'mysql' property of the property definition. For example:

```
"locationId":{
    "type":"String",
    "required":true,
    "length":20,
    "mysql":
    {
        "columnName":"LOCATION_ID",
        "dataType":"VARCHAR2",
        "dataLength":20,
        "nullable":"N"
    }
}
```

You can also use the dataType column/property attribute to specify what MySQL column type to use for many loopback-datasource-juggler types. The following type-dataType combinations are supported:

- Number
- integer
- tinyint
- smallint
- mediumint
- int
- bigint

Use the `limit` option to alter the display width. Example:

```
`{ count : { type: Number, dataType: 'smallInt' }}`
```

Floating-point types

For Float and Double data types, use the `precision` and `scale` options to specify custom precision. Default is (16,8). For example:

```
{ average : { type: Number, dataType: 'float', precision: 20, scale: 4 }}
```

Fixed-point exact value types

For Decimal and Numeric types, use the `precision` and `scale` options to specify custom precision. Default is (9,2). These aren't likely to function as true fixed-point.

Example:

```
{ stdDev : { type: Number, dataType: 'decimal', precision: 12, scale: 8 }}
```

Other types

Convert String / DataSource.Text / DataSource.JSON to the following MySQL types:

- varchar
- char
- text
- mediumtext
- tinytext
- longtext

Example:

```
{ userName : { type: String, dataType: 'char', limit: 24 }}
```

Example:

```
{ biography : { type: String, dataType: 'longtext' }}
```

Convert JSON Date types to datetime or timestamp

Example:

```
{ startTime : { type: Date, dataType: 'timestamp' }}
```

Enum

Enums are special. Create an Enum using Enum factory:

```
var MOOD = dataSource.EnumFactory('glad', 'sad', 'mad');
MOOD.SAD; // 'sad'
MOOD(2); // 'sad'
MOOD('SAD'); // 'sad'
MOOD('sad'); // 'sad'
{ mood: { type: MOOD }}
{ choice: { type: dataSource.EnumFactory('yes', 'no', 'maybe'), null: false }}
```

Discovery methods

LoopBack provides a unified API to create models based on schema and tables in relational databases. The same discovery API is available when using connectors for Oracle, MySQL, PostgreSQL, and SQL Server. For more information, see Discovering models from databases and Da tabase discovery API.

Oracle connector

See also:

- Example application
- Database discovery API

Prerequisites

See Installing the Oracle connector for installation instructions.

Connector settings

The connector settings depend on naming methods you use for the Oracle database. LoopBack supports three naming methods:

1. Easy connect: host/port/database.
2. Local naming (TNS): alias to a full connection string, which can specify all the attributes that Oracle supports.
3. Directory naming (LDAP): directory for looking up the full connection string, which can specify all the attributes that Oracle supports.

Easy Connect

Easy Connect is the simplest form that provides out-of-the-box TCP/IP connectivity to databases. The connector can be configured using the following settings from the data source.

- host or hostname (defaults to 'localhost'): The host name or ip address of the Oracle DB server
- port (defaults to 1521): The port number of the Oracle DB server
- username or user: The user name to connect to the Oracle DB
- password: The password
- database (defaults to 'XE'): The Oracle DB listener name

For example, you can configure a data source named 'demoDB' in `datasources.json` as follows:

```
{
  "demoDB": {
    "connector": "oracle",
    "host": "demo.strongloop.com",
    "port": 1521,
    "database": "XE",
    "username": "demo",
    "password": "L00pBack"
  }
}
```

Local and Directory Naming

Both Local and Directory naming require configuration files being placed in a TNS admin directory, such as /oracle/admin.

sqlnet.ora (specifying the supported naming methods)

```
NAMES.DIRECTORY_PATH=(LDAP,TNSNAMES,EZCONNECT)
```

tnsnames.ora (mapping aliases to connection strings)

```
demo1=(DESCRIPTION=(CONNECT_DATA=(SERVICE_NAME=))(ADDRESS=(PROTOCOL=TCP)(HOST=demo.str
ongloop.com)(PORT=1521)))
```

ldap.ora (configuring the LDAP server)

```
DIRECTORY_SERVERS=(localhost:1389)
DEFAULT_ADMIN_CONTEXT="dc=strongloop,dc=com"
DIRECTORY_SERVER_TYPE=OID
```

Set up TNS_ADMIN environment variable

For the oracle connector to pick up the configurations, an environment variable named 'TNS_ADMIN' has to be set up to point to the directory containing the .ora files.

```
export TNS_ADMIN=<the directory containing .ora files>
```

Now you can use either the TNS alias or LDAP service name to configure a data source:

```
var ds = loopback.createDataSource({
"tns": "demo", // The tns property can be a tns name or LDAP service name
"username": "demo",
"password": "L00pBack"
});
```

Here is an example for `datasources.json`:

```
{
  "demoDB": {
    "connector": "oracle",
    "tns": "demo",
    "username": "demo",
    "password": "L00pBack"
  }
}
```

Connection pooling options

Property name	Description	Default value
minConn	The maximum number of connections in the connection pool	1
maxConn	The minimum number of connections in the connection pool	10
incrConn	The incremental number of connections for the connection pool.	1

timeout	The time-out period in seconds for a connection in the connection pool. The oracle connector will terminate any connections related to this connection pool that have been idle for longer than the time-out period specified.	10

For example,

```
{
  "demoDB": {
    "connector": "oracle",
    "minConn":1,
    "maxConn":5,
    "incrConn":1,
    "timeout": 10,
    ...
  }
}
```

Model definition for Oracle

The model definition consists of the following properties:

- name: Name of the model, by default, it's the camel case of the table
- options: Model level operations and mapping to Oracle schema/table
- properties: Property definitions, including mapping to Oracle column

```
{
    "name":"Inventory",
    "options":{
      "idInjection":false,
      "oracle":{
        "schema":"STRONGLOOP",
        "table":"INVENTORY"
      }
    },
    "properties":{
      "productId":{
        "type":"String",
        "required":true,
        "length":20,
        "id":1,
        "oracle":{
          "columnName":"PRODUCT_ID",
          "dataType":"VARCHAR2",
          "dataLength":20,
          "nullable":"N"
        }
      },
      "locationId":{
        "type":"String",
        "required":true,
        "length":20,
        "id":2,
        "oracle":{
          "columnName":"LOCATION_ID",
          "dataType":"VARCHAR2",
          "dataLength":20,
          "nullable":"N"
        }
      },
      "available":{
        "type":"Number",
        "required":false,
        "length":22,
        "oracle":{
          "columnName":"AVAILABLE",
          "dataType":"NUMBER",
          "dataLength":22,
          "nullable":"Y"
        }
      },
      "total":{
        "type":"Number",
        "required":false,
        "length":22,
        "oracle":{
          "columnName":"TOTAL",
          "dataType":"NUMBER",
          "dataLength":22,
          "nullable":"Y"
        }
      }
    }
  }
```

Type mapping

See LoopBack types for details on LoopBack's data types.

JSON to Oracle Types

LoopBack Type	Oracle Type
String JSON Text default	VARCHAR2 Default length is 1024
Number	NUMBER
Date	DATE
Timestamp	TIMESTAMP(3)
Boolean	CHAR(1)

Oracle Types to JSON

Oracle Type	LoopBack Type
CHAR(1)	Boolean
CHAR(n) VARCHAR VARCHAR2, LONG VARCHAR NCHAR NVARCHAR2	String
LONG, BLOB, CLOB, NCLOB	Node.js Buffer object
NUMBER INTEGER DECIMAL DOUBLE FLOAT BIGINT SMALLINT REAL NUMERIC BINARY_FLOAT BINARY_DOUBLE UROWID ROWID	Number
DATE TIMESTAMP	Date

Destroying models

Destroying models may result in errors due to foreign key integrity. Make sure to delete any related models first before calling delete on model's with relationships.

Auto-migrate / Auto-update

After making changes to your model properties you must call `Model.automigrate()` or `Model.autoupdate()`. Call `Model.automigrate()` only on new models since it will drop existing tables.

LoopBack Oracle connector creates the following schema objects for a given model:

- A table, for example, PRODUCT
- A sequence for the primary key, for example, PRODUCT_ID_SEQUENCE
- A trigger to generate the primary key from the sequnce, for example, PRODUCT_ID_TRIGGER

Discovery methods

LoopBack provides a unified API to create models based on schema and tables in relational databases. The same discovery API is available when using connectors for Oracle, MySQL, PostgreSQL, and SQL Server. For more information, see Database discovery API.

Installing the Oracle connector

Overview

Use the LoopBack Oracle installer to simplify installation of node-oracle and Oracle instant clients, when you `npm install` the loopback-connector-oracle module. The LoopBack Oracle installer downloads and extracts the prebuilt LoopBack Oracle binary dependencies into the parent module's `node_modules` directory and sets up the environment for the Oracle Database Instant Client. This makes it easier to build the node-oracle module and install and configure the Oracle Database Instant Client.

If you need to use the Oracle driver directly, see https://github.com/strongloop/strong-oracle

Use

Standalone / local use

1. Ensure that you run `make/build.sh` from the `loopback-oracle-builder` directory and the package (gzipped tarball) is built correctly and uploaded to a publicly available site.
2. Make sure the loopback-oracle-builder and the loopback-oracle-installer projects are siblings within the same directory hierarchy. This is only required if you want to test with the locally-built gzipped tarball.
3. To run it locally enter these commands:

```
cd loopback-oracle-installer
npm install
```

Production use

For production release, set the environment variable LOOPBACK_ORACLE_URL appropriately. For example:

```
export LOOPBACK_ORACLE_URL=http://7e9918db41dd01dbf98e-ec15952f71452bc0809d79c86f5751b6.r22.cf1.rackcdn.com/
```

or

```
export LOOPBACK_ORACLE_URL=/Users/rfeng/Projects/loopback/loopback-oracle-builder/build/MacOSX
```

and then run:

```
$ npm install loopback-oracle-installer
```

Changes made to your environment

To configure Oracle Database Instant Client for Node.js modules, the installer sets up the environment variable depending on the target platform.

MacOSX

The change is made in `$HOME/.bash_profile` or `$HOME/.profile`.

```
# __loopback-oracle-installer__: Fri Aug 30 15:11:11 PDT 2013
export DYLD_LIBRARY_PATH="$DYLD_LIBRARY_PATH:/Users/<user>/<myapp>/node_modules/loopback-connector-oracle/node_modules/instantcl
```

You need to open a terminal window to make the change take effect.

Linux

The change is made in `$HOME/.bash_profile` or `$HOME/.profile`.

```
# __loopback-oracle-installer__: Fri Aug 30 15:11:11 PDT 2013
export LD_LIBRARY_PATH="$LD_LIBRARY_PATH:/Users/<user>/<myapp>/node_modules/loopback-connector-oracle/node_modules/instantclient"
```

You must open a new terminal window to make the change take effect.

Windows

The change is made to the PATH environment variable for the logged in user. Please note the PATH setting will NOT be effective immediately. You have to activate it using one of the methods below:

1. Log off the current user session and log in.
2. Open Control Panel --> System --> Advanced System Settings --> Environment Variables. Examine the Path under User variables, and click OK to activate it. You need to open a new Command Prompt. Please run 'path' command to verify.

Installation from behind a proxy server

⚠ This feature is supported by loopback-oracle-installer vesion 1.1.3 or later.

If your system is behind a corporate HTTP/HTTPS proxy to access the internet, you'll need to set the proxy for npm before running 'npm install'. For example,

```
$ npm config set proxy http://proxy.mycompany.com:8080
$ npm config set https-proxy http://https-proxy.mycompany.com:8080
```

If the proxy url requires username/password, you can use the following syntax:

```
$ npm config set proxy http://youruser:yourpass@proxy.mycompany.com:8080
$ npm config set https-proxy http://youruser:yourpass@https-proxy.mycompany.com:8080
```

The proxy can also be set as part of the npm command as follows:

```
$ npm --proxy=http://proxy.mycompany.com:8080 install
$ npm --https-proxy=http://https-proxy.mycompany.com:8080 install
```

Please note that npm's default value for proxy is from the HTTP_PROXY or http_proxy environment variable. And the default value for https-proxy is from the HTTPS_PROXY, https_proxy, HTTP_PROXY, or http_proxy environment variable. So you can configure the proxy using environment variables too.

Linux or Mac:

```
HTTP_PROXY=http://proxy.mycompany.com:8080 npm install
```

Windows:

```
set HTTP_PROXY=http://proxy.mycompany.com:8080
npm install
```

PostgreSQL connector

See also:

- Example application - with PostgreSQL connector
- Database discovery API

The LoopBack PostgreSQL connector enables LoopBack applications to connect to PostgreSQL databases.

Installation

```
$ npm install loopback-connector-postgresql --save
```

This will install the module from npm and add it as a dependency to the application's package.json file.

Creating a data source

Use the LoopBack datasource generator to add a PostgreSQL data source to your application.

Configure the PostgreSQL connector with the following settings:

Property	Default	Description
database	N/A	PostgreSQL database name
debug	false	Display debugging information
host or hostname	localhost	Host name or IP address of the PostgreSQL database server
password	N/A	Database password
port	5432	Port number of the PostgreSQL database server.
URL	N/A	URL to the database, such as 'postgres://test:mypassword@localhost:5432/dev' Use this instead of host, port, user, password, and database properties.
username or user	N/A	User name to connect to the PostgreSQL database.

⚠ By default, the 'public' schema is used for all tables.

Defining models

The model definition consists of the following properties.

Property	Default	Description
name	Camel-case of the database table name	Name of the model.
options	N/A	Model level operations and mapping to PostgreSQL schema/table

properties	N/A	Property definitions, including mapping to PostgreSQL column

For example:


```
{"name": "Inventory",
   "options": {
      "idInjection": false,
      "postgresql": {
        "schema": "strongloop",
        "table": "inventory"
      }
   },
   "properties": {
      "id": {
        "type": "String",
        "required": false,
        "length": 64,
        "precision": null,
        "scale": null,
        "postgresql": {
          "columnName": "id",
          "dataType": "character varying",
          "dataLength": 64,
          "dataPrecision": null,
          "dataScale": null,
          "nullable": "NO"
        }
      },
      "productId": {
        "type": "String",
        "required": false,
        "length": 20,
        "precision": null,
        "scale": null,
        "id": 1,
        "postgresql": {
          "columnName": "product_id",
          "dataType": "character varying",
          "dataLength": 20,
          "dataPrecision": null,
          "dataScale": null,
          "nullable": "YES"
        }
      },
      "locationId": {
        "type": "String",
        "required": false,
        "length": 20,
        "precision": null,
        "scale": null,
        "id": 1,
        "postgresql": {
          "columnName": "location_id",
          "dataType": "character varying",
          "dataLength": 20,
          "dataPrecision": null,
          "dataScale": null,
          "nullable": "YES"
```

```
    }
  },
  "available": {
    "type": "Number",
    "required": false,
    "length": null,
    "precision": 32,
    "scale": 0,
    "postgresql": {
      "columnName": "available",
      "dataType": "integer",
      "dataLength": null,
      "dataPrecision": 32,
      "dataScale": 0,
      "nullable": "YES"
    }
  },
  "total": {
    "type": "Number",
    "required": false,
    "length": null,
    "precision": 32,
    "scale": 0,
    "postgresql": {
      "columnName": "total",
      "dataType": "integer",
      "dataLength": null,
      "dataPrecision": 32,
      "dataScale": 0,
      "nullable": "YES"
```

```
        }
      }
    }}
```

Destroying models

If you destroy models, you may get errors due to foreign key integrity. Make sure to delete any related models first before calling `delete()` on models that have relationships.

Auto-migrate and auto-update

After making changes to your model properties, you must call `Model.automigrate()` or `Model.autoupdate()`. Call `Model.automigrate()` only on new models since it will drop existing tables. These methods will

- Define a primary key for the properties whose `id` property is true (or a positive number).
- Create a column with 'SERIAL' type if the `generated` property of the `id` property is true.

Type mapping

See LoopBack types for details on LoopBack's data types.

LoopBack to PostgreSQL types

LoopBack Type	PostgreSQL Type
String JSON Text Default	VARCHAR2 Default length is 1024
Number	INTEGER
Date	TIMESTAMP WITH TIME ZONE
Boolean	BOOLEAN

PostgreSQL types to LoopBack

PostgreSQL Type	LoopBack Type
BOOLEAN	Boolean
VARCHAR CHARACTER VARYING CHARACTER CHAR TEXT	String
BYTEA	Node.js Buffer object
SMALLINT INTEGER BIGINT DECIMAL NUMERIC REAL DOUBLE SERIAL BIGSERIAL	Number
DATE TIMESTAMP TIME	Date
POINT	GeoPoint

Discovery methods

LoopBack provides a unified API to create models based on schema and tables in relational databases. The same discovery API is available when using connectors for Oracle, MySQL, PostgreSQL, and SQL Server. For more information, see Database discovery API.

REST connector

See also
- REST connector API
- REST resource API
- Request builder API

Overview

The LoopBack REST connector enables applications to interact with other REST APIs using a template-driven approach. It supports two different styles of API invocations:

- Resource CRUD
- Defining a custom method using a template

Installation

In your application root directory, enter:

```
$ npm install loopback-connector-rest --save
```

This will install the module from npm and add it as a dependency to the application's package.json file.

Creating a data source

Use the LoopBack datasource generator to add a REST data source to your application.

Configuring a REST data source

Configure the REST connector in `datasources.json` as follows (for example using the Google Maps API):

```
...
    "geoRest": {
        "connector": "rest",
        "debug": "false",
        "operations": [{
            "template": {
                "method": "GET",
                "url": "http://maps.googleapis.com/maps/api/geocode/{format=json}",
                "headers": {
                    "accepts": "application/json",
                    "content-type": "application/json"
                },
                "query": {
                    "address": "{street},{city},{zipcode}",
                    "sensor": "{sensor=false}"
                },
                "responsePath": "$.results[0].geometry.location"
            },
            "functions": {
                "geocode": ["street", "city", "zipcode"]
            }
        }]
    }
...
```

Resource CRUD

If the REST API supports create, read, update, and delete (CRUD) operations for resources, such as users or orders, you can simply bind the model to a REST endpoint that follows REST conventions.

The following methods are mixed into your model class:

- create: POST /users
- findById: GET /users/:id
- delete: DELETE /users/:id
- update: PUT /users/:id
- find: GET /users?limit=5&username=ray&order=email

For example:

```
var ds = loopback.createDataSource({
    connector: require("loopback-connector-rest"),
    debug: false,
    baseURL: 'http://localhost:3000'
});

var User = ds.createModel('user', {
    name: String,
    bio: String,
    approved: Boolean,
    joinedAt: Date,
    age: Number
});

User.create(new User({name: 'Mary'}), function (err, user) {
    console.log(user);
});

User.find(function (err, user) {
    console.log(user);
});

User.findById(1, function (err, user) {
    console.log(err, user);
});

User.update(new User({id: 1, name: 'Raymond'}), function (err, user) {
    console.log(err, user);
});
```

Defining a custom method using a template

Imagine that you use a web browser or REST client to test drive a REST API; you will specify the following HTTP request properties:

- method: HTTP method
- url: The URL of the request
- headers: HTTP headers
- query: Query strings
- responsePath: JSONPath applied to the HTTP body

LoopBack REST connector allows you to define the API invocation as a JSON template. For example:

```
template: {
            "method": "GET",
            "url": "http://maps.googleapis.com/maps/api/geocode/{format=json}",
            "headers": {
                "accepts": "application/json",
                "content-type": "application/json"
            },
            "query": {
                "address": "{street},{city},{zipcode}",
                "sensor": "{sensor=false}"
            },
            "responsePath": "$.results[0].geometry.location"
        }
```

The template variable syntax is:

```
{name=defaultValue:type}
```

The variable is required if the name has a prefix of ! or ^

For example:

```
'{x=100:number}'
'{x:number}'
'{x}'
'{x=100}ABC{y}123'
'{!x}'
'{x=100}ABC{^y}123'
```

To use custom methods, you can configure the REST connector with the `operations` property, which is an array of objects that contain `template` and `functions`. The `template` property defines the API structure while the `functions` property defines JavaScript methods that takes the list of parameter names.

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

var ds = loopback.createDataSource({
    connector: require("loopback-connector-rest"),
    debug: false,
    operations: [
    {
        template: {
            "method": "GET",
            "url": "http://maps.googleapis.com/maps/api/geocode/{format=json}",
            "headers": {
                "accepts": "application/json",
                "content-type": "application/json"
            },
            "query": {
                "address": "{street},{city},{zipcode}",
                "sensor": "{sensor=false}"
            },
            "responsePath": "$.results[0].geometry.location"
        },
        functions: {
           "geocode": ["street", "city", "zipcode"]
        }
    }
]});
```

Now you can invoke the geocode API as follows:

```
Model.geocode('107 S B St', 'San Mateo', '94401', processResponse);
```

By default, LoopBack REST connector also provides an 'invoke' method to call the REST API with an object of parameters, for example:

```
 Model.invoke({street: '107 S B St', city: 'San Mateo', zipcode: '94401'},
processResponse);
```

REST connector API

Module: loopback-connector-rest

exports.initialize(dataSource, [callback])

Export the initialize method to loopback-datasource-juggler

Arguments

Name	Type	Description
dataSource	`DataSource`	The loopback data source instance
[callback]	`function`	The callback function

RestConnector(baseURL)

The RestConnector constructor

Arguments

Name	Type	Description
baseURL	`string`	The base URL

restConnector.define(definition)

Hook for defining a model by the data source

Arguments

Name	Type	Description
definition	`object`	The model description

restConnector.installPostProcessor(definition)

Install the post processor

Arguments

Name	Type	Description

definition	`object`	The model description

restConnector.preProcess(data)

Pre-process the request data

Arguments

Name	Type	Description
data	`Any`	The request data

Returns

Name	Type	Description
result		

restConnector.postProcess(model, data, many)

Post-process the response data

Arguments

Name	Type	Description
model	`string`	The model name
data	`Any`	The response data
many	`boolean`	Is it an array

restConnector.getResource(model)

Get a REST resource client for the given model

Arguments

Name	Type	Description
model	`string`	The model name

Returns

Name	Type	Description
result	`Any`	

restConnector.create(model, data, [callback])

Create an instance of the model with the given data

Arguments

Name	Type	Description
model	`string`	The model name
data	`object`	The model instance data
[callback]	`function`	The callback function

restConnector.updateOrCreate(model, data, [callback])

Update or create an instance of the model

Arguments

Name	Type	Description
model	`string`	The model name
data	`object`	The model instance data
[callback]	`function`	The callback function

restConnector.responseHandler(model, [callback])

A factory to build callback function for a response

Arguments

Name	Type	Description
model	`string`	The model name
[callback]	`function`	The callback function

Returns

Name	Type	Description
result	`function`	

restConnector.save(model, data, [callback])

Save an instance of a given model

Arguments

Name	Type	Description
model	`string`	The model name
data	`object`	The model instance data
[callback]	`function`	The callback function

restConnector.exists(model, id, [callback])

Check the existence of a given model/id

Arguments

Name	Type	Description
model	`string`	The model name
id	`Any`	The id value
[callback]	`function`	The callback function

restConnector.find(model, id, [callback])

Find an instance of a given model/id

Arguments

Name	Type	Description
model	`string`	The model name
id	`Any`	The id value
[callback]	`function`	The callback function

restConnector.destroy(model, id, [callback])

Delete an instance for a given model/id

Arguments

Name	Type	Description
model	`string`	The model name
id	`Any`	The id value
[callback]	`function`	The callback function

restConnector.all(model, filter, [callback])

Query all instances for a given model based on the filter

Arguments

Name	Type	Description
model	`string`	The model name
filter	`object`	The filter object
[callback]	`function`	The callback function

restConnector.destroyAll(model, [callback])

Delete all instances for a given model

Arguments

Name	Type	Description
model	`string`	The model name
[callback]	`function`	The callback function

restConnector.count(model, [callback], where)

Count cannot not be supported efficiently.

Arguments

Name	Type	Description
model	`string`	The model name
[callback]	`function`	The callback function
where	`object`	The where object

restConnector.updateAttributes(model, id, data, [callback])

Update attributes for a given model/id

Arguments

Name	Type	Description
model	`string`	The model name
id	`Any`	The id value
data	`object`	The model instance data
[callback]	`function`	The callback function

REST resource API

Module: loopback-connector-rest

RestResource(modelCtor, baseUrl)

Build a REST resource client for CRUD operations

Arguments

Name	Type	Description
modelCtor	`function`	The model constructor
baseUrl	`string`	The base URL

Returns

Name	Type	Description
result	`RestResource`	

wrap(cb)

Wrap the callback so that it takes (err, result, response)

Arguments

Name	Type	Description
cb	`function`	The callback function

Returns

Name	Type	Description
result	Any	

restResource.create(obj, [cb])

Map the create operation to HTTP POST /{model}

Arguments

Name	Type	Description
obj	`object`	The HTTP body
[cb]	`function`	The callback function

restResource.update(id, obj, [cb])

Map the update operation to POST /{model}/{id}

Arguments

Name	Type	Description
id	Any	The id value
obj	`object`	The HTTP body

[cb]	`function`	The callback function

restResource.delete(id, [cb])

Map the delete operation to POST /{model}/{id}

Arguments

Name	Type	Description
id	`Any`	The id value
[cb]	`function`	The callback function

restResource.deleteAll(id, [cb])

Map the delete operation to POST /{model}

Arguments

Name	Type	Description
id	`Any`	The id value
[cb]	`function`	The callback function

restResource.find(id, [cb])

Map the find operation to GET /{model}/{id}

Arguments

Name	Type	Description
id	`Any`	The id value
[cb]	`function`	The callback function

restResource.all

Map the all/query operation to GET /{model}

Arguments

Name	Type	Description
q	`object`	query string
[cb]	`function`	callback with (err, results)

Request builder API

Module: loopback-connector-rest

- requestBuilder.responsePath
- requestBuilder.parse
- requestBuilder.auth
- requestBuilder.toJSON
- RequestBuilder.compile
- requestBuilder.build
- requestBuilder.operation
- requestBuilder.invoke
- RequestBuilder.resource
- RequestBuilder.request

debug

REST request spec builder

RequestBuilder([method], url)

RequestBuilder constructor

Arguments

Name	Type	Description
[method]	`string`	HTTP method
url	`string or object`	The HTTP URL or an object for the options including method, url properties

Returns

Name	Type	Description
result	`RequestBuilder`	

isObject(obj)

Check if `obj` is an object.

Arguments

Name	Type	Description
obj	`object`	

requestBuilder.attach(field, file, filename)

Queue the given `file` as an attachment with optional `filename`.

Arguments

Name	Type	Description
field	`string`	
file	`string`	
filename	`string`	

requestBuilder.redirects(n)

Set the max redirects to `n`.

Arguments

Name	Type	Description
n	`number`	

requestBuilder.url(url)

Configure the url

Arguments

Name	Type	Description
url	`string`	The HTTP URL

Returns

Name	Type	Description
result	`REquestBuilder`	

requestBuilder.method(method)

Configure the HTTP method

Arguments

Name	Type	Description
method	`string`	The HTTP method

Returns

Name	Type	Description
result	`RequestBuilder`	

requestBuilder.timeout(timeout)

Configure the timeout

Arguments

Name	Type	Description
timeout	`number`	The timeout value in ms

Returns

Name	Type	Description
result	`RequestBuilder`	

requestBuilder.header(field, val)

Set header `field` to `val`, or multiple fields with one object.

Examples:

```
req.get('/')
   .header('Accept', 'application/json')
   .header('X-API-Key', 'foobar');

 req.get('/')
   .header({ Accept: 'application/json', 'X-API-Key': 'foobar' });
```

Arguments

Name	Type	Description
field	`string or object`	
val	`string`	

requestBuilder.type(type)

Set *Content-Type* response header passed through `mime.lookup()`.

Examples:

```
request.post('/')
   .type('xml')
   .body(xmlstring);

 request.post('/')
   .type('json')
   .body(jsonstring);

 request.post('/')
   .type('application/json')
   .body(jsonstring);
```

Arguments

Name	Type	Description
type	`string`	

requestBuilder.query(val)

Add query-string `val`.

Examples:

request.get('/shoes') .query('size=10') .query({ color: 'blue' })

Arguments

Name	Type	Description
val	`object or string`	

requestBuilder.body(body)

Send `body`, defaulting the `.type()` to "json" when an object is given.

Examples:

```
// manual json
  request.post('/user')
    .type('json')
    .body('{"name":"tj"}');

  // auto json
  request.post('/user')
    .body({ name: 'tj' });

  // manual x-www-form-urlencoded
  request.post('/user')
    .type('form')
    .body('name=tj');

  // auto x-www-form-urlencoded
  request.post('/user')
    .type('form')
    .body({ name: 'tj' });

  // string defaults to x-www-form-urlencoded
  request.post('/user')
    .body('name=tj')
    .body('foo=bar')
    .body('bar=baz');
```

Arguments

Name	Type	Description
body	`string or object`	

requestBuilder.buffer()

Enable / disable buffering.

requestBuilder.timeout(ms)

Set timeout to `ms`.

Arguments

Name	Type	Description
ms	`Number`	

requestBuilder.responsePath(responsePath)

Set the response json path

Arguments

Name	Type	Description

responsePath	`string`	The JSONPath to be applied against the HTTP body

Returns

Name	Type	Description
result	`RequestBuilder`	

requestBuilder.parse(fn)

Define the parser to be used for this response.

Arguments

Name	Type	Description
fn	`function`	The parser function

requestBuilder.auth(user, pass)

Set Authorization field value with `user` and `pass`.

Arguments

Name	Type	Description
user	`string`	The user name
pass	`string`	The password

requestBuilder.toJSON()

Serialize the RequestBuilder to a JSON object

Returns

Name	Type	Description
result	`method:`	*, url:* }}

RequestBuilder.compile

Load the REST request from a JSON object

Arguments

Name	Type	Description
req	`object`	The request json object

Returns

Name	Type	Description
result	`RequestBuilder`	

requestBuilder.build(options)

Build the request by expanding the templatized properties with the named values from options

Arguments

Name	Type	Description
options	`object`	

Returns

Name	Type	Description
result	`Any`	

requestBuilder.operation(parameterNames)

Map the request builder to a function

Arguments

Name	Type	Description
parameterNames	`[string]`	The parameter names that define the order of args. It be an array of strings or multiple string arguments

Returns

Name	Type	Description
result	`function`	A function to invoke the REST operation

requestBuilder.invoke(parameters, cb)

Invoke a REST API with the provided parameter values in the parameters object

Arguments

Name	Type	Description
parameters	`object`	An object that provide {name: value} for parameters
cb	`function`	The callback function

RequestBuilder.resource(modelCtor, baseUrl)

Attach a model to the REST connector

Arguments

Name	Type	Description
modelCtor	`function`	The model constructor
baseUrl	`string`	The base URL

Returns

Name	Type	Description
result	`RestResource`	

RequestBuilder.request(uri, options, cb)

Delegation to request Please note the cb takes (err, body, response)

Arguments

Name	Type	Description
uri	`string`	The HTTP URI
options	`options`	The options
cb	`function`	The callback function

REST example with SharePoint

Imagine that you need to get document details from a Microsoft Sharepoint server repository published as a lightweight JSON API that can be consumed by various client apps. This tutorial walks you through how to do this with LoopBack.

REST example - creating the back-end

Setup

Start with the following app from GitHub and use `npm install` to install all the app's dependencies:

```
$ git clone https://github.com/strongloop/loopback-example-datasourceAPI.git
$ cd loopback-example-datasourceAPI
$ npm install
```

If you're impatient or just lazy, you can download the completed application from GitHub:

```
$ git clone https://github.com/strongloop/loopback-example-customAPI.git
```

Create REST data source with custom methods

First, create a custom data source using the LoopBack REST connector.

```
$ slc loopback:datasource
[?] Enter the data-source name: Sharepoint
[?] Select the connector for Sharepoint: REST services (supported by StrongLoop)
```

Now open the `datasources.json` file and add custom logic and methods. Copy the code below starting with the line after

`"connector": "rest",`

and paste it into `datasources.json`, so it looks as shown below when you're done.

The API endpoint http://sharepoint.global.strongloop.com/... is just an example; it doesn't actually exist. Below you'll use the `documentsList.json` file to mock up data that would come from the SharePoint REST API.

```
{
  "Sharepoint": {
    "name": "Sharepoint",
    "connector": "rest",
    "debug": "true",
    "operations": [{
        "template": {
           "method": "GET",
           "url":
"http://sharepoint.global.strongloop.com/Corporate/_api/web/lists/getByTitle('StrongLo
opCorporate')/items",
           "headers": {
               "accept": "application/json; odata=verbose",
               "content-type": "application/json"
           },
           "query": {
               "$orderby": "{orderby}",
                "$top": "{top}"
              },
           "responsePath": "$.results[0]"
         },
         "functions": {
             "getFileList": ["orderby", "top"]
         }
     } ,
         {
          "template": {
             "method": "GET",
             "url":
"http://sharepoint.global.strongloop.com/Corporate/_api/web/lists/GetByTitle('StrongLo
opCorporate')/items({fileID})/File",
             "headers": {
                 "accept": "application/json; odata=verbose",
                 "content-type": "application/json"
              },
             "responsePath": "$.results[0]"
            },
            "functions": {
               "getFileAttributes": ["fileID"]
            }
     }]
  },
  "db": {
    "name": "db",
    "connector": "memory"
  },
  "accountDB": {
    "host": "demo.strongloop.com",
    "port": 3306,
    "database": "demo",
    "username": "demo",
    "password": "L00pBack",
    "name": "accountDB",
    "connector": "mysql"
  }
}
```

The LoopBack REST connector enables Node.js applications to interact with HTTP REST APIs using a template-driven approach. It supports two different styles of API invocations:

- Resource create, read, update, and delete (CRUD)
- Defining a custom method using REST template

In the code above, you can see that you added template-based modeling logic to the REST connector. You are making two independent API calls to Sharepoint: one to get a list of files/documents and another to return the attributes of each file within the list.

Add model for CRUD operations

If you were only interested in CRUD operations, as usual with the generators-based methodology, you can create a Document model using Yeoman and attach it to the Sharepoint REST connector.

```
$ slc loopback:model
[?] Enter the model name: Document
[?] Select the data-source to attach Document to:
 Sharepoint (rest)
  accountDB (mysql)
  db (memory)
[?] Expose Document via the REST API? (Y/n) :Y
[?] Custom plural form (used to build REST URL):
```

When the generator prompts you to add properties, follow the prompts to add these properties:

Property	Type	Required?
name	string	Yes
type	string	Yes
size	string	Yes
date_created	date	Yes
last_modified	date	Yes

You can see the corresponding changes made to `/common/models/document.json`.

Add model with custom logic

If you're not interested only in CRUD operations, but want to implement custom logic, skip the Yeoman steps and add custom logic for post-processing the data returned by the Sharepoint API call.

- Copy and paste the code below.
- Save it to `/server/boot/document.js`. That's where you put custom models and other code you want executed on app startup.

document.js › Expand source

```
var loopback = require("loopback");
var app = require('../server');
var fs = require("fs");

var ds = app.dataSources.Sharepoint;
var fileListModel = ds.createModel ('Documents',{},{base:loopback.Model});
console.log(fileListModel.super_.modelName);

module.exports=fileListModel;
var queryParam = {
    arg1: 'Modified desc',
    arg2: '5'
};
```

```
var fileIDParam = {
        fileID: '8'
};

fileListModel.getFileList = function(orderBy, top, cb) {
        var file = __dirname + '/documentList.json';
        fs.readFile(file, 'utf8', function (err, data) {
        if (err) {
          console.log('Error: ' + err);
          cb(err);
        } else {
          data = JSON.parse(data);
          cb(null, data);
        }
   });
};

fileListModel.getFileList.shared = true;
fileListModel.getFileList.accepts = [{arg: 'orderBy', type: 'string', http: {source:
'query'}},
{arg: 'top', type: 'number', http: {source: 'query'}}];
fileListModel.getFileList.returns = [{arg: 'data', type: 'array', root: true} ];
fileListModel.getFileList.http = {verb: 'get', path: '/'};

fileListModel.getFileAttributes = function(id, cb) {
        console.log("findbyid", id);
        var file = __dirname + '/documentList.json';
        fs.readFile(file, 'utf8', function (err, data) {
        if (err) {
          console.log('Error: ' + err);
          cb(err);
        } else {
          data = JSON.parse(data);
          console.log(data[id-1]);
          cb(null, data[id-1]);
        }
});
};

fileListModel.getFileAttributes.shared = true;
fileListModel.getFileAttributes.accepts = [{arg: 'id', type: 'number', http: {source:
'path'}}];
fileListModel.getFileAttributes.returns = [{arg: 'data', type: 'object', root: true}
];
```

```
fileListModel.getFileAttributes.http = {verb: 'get', path: '/:id'};

app.model(fileListModel);
```

This file does the following:

- Overrides the default persistence model created by the generators and adds custom model to extend the base model.
- Extends the defined custom methods in the datasource setup and converts them into API endpoints.
- Provides dummy data (`documentsList.json`) to represent data that in practice would come from the Sharepoint Server REST interface.
- Exposes local server-side methods that other Node functions can invoke directly without having to interact with the JSON API endpoint.

The dummy data in `documentList.json` is shown below. This test data is the same format that the backend Sharepoint call would return.

documentList.json Expand source

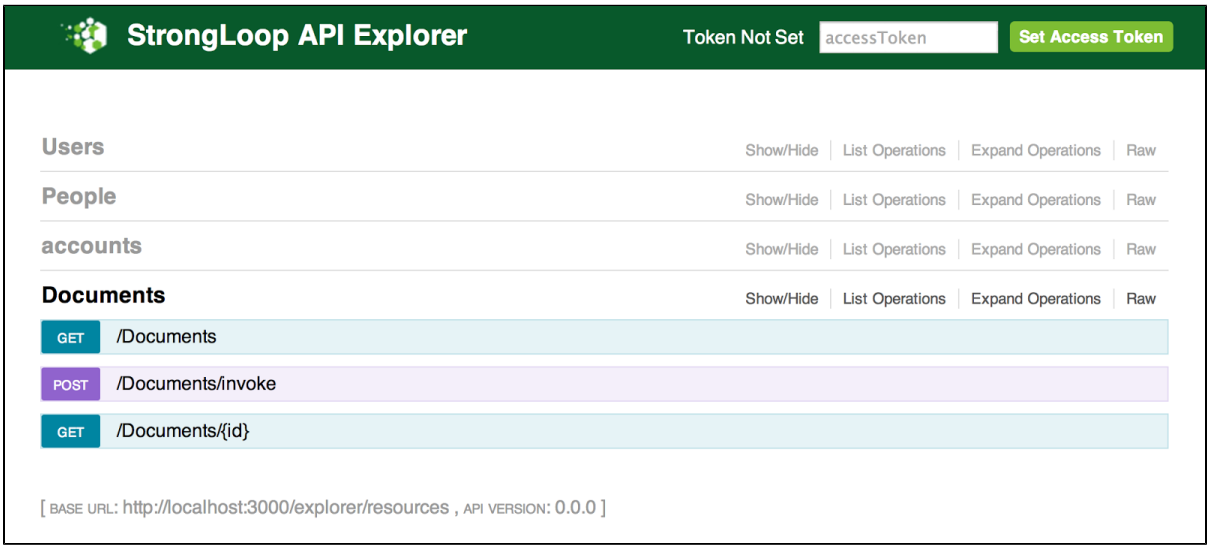
```
[
  {
    "name": "PriceList",
    "created_by": "Michelle Williams",
    "type": "xls",
    "size": "20Kb",
    "date_created": "2011-06-23T18:25:43.511Z",
    "last_modified": "2014-03-23T18:25:43.511Z",
    "id": 1
  },
  {
    "name": "CustomerList",
    "created_by": "Gorge Clooney",
    "type": "xls",
    "size": "10Kb",
    "date_created": "2010-05-23T18:25:43.511Z",
    "last_modified": "2012-09-23T18:25:43.511Z",
    "id": 2
  },
  {
    "name": "SalesPipeline",
    "created_by": "Brad Pitt",
    "type": "pdf",
    "size": "2Kb",
    "date_created": "2011-02-25T18:25:43.511Z",
    "last_modified": "2012-04-23T18:25:43.511Z",
    "id": 3
  },
 {
    "name": "Forecast",
    "created_by": "Olivia Wilde",
    "type": "xls",
    "size": "2MB",
    "date_created": "2012-01-23T18:25:43.511Z",
    "last_modified": "2012-04-23T18:25:43.511Z",
    "id": 4
  },
  {
    "name": "ProductRoadMap",
    "created_by": "Ryan Gosling",
    "type": "pdf",
    "size": "200Kb",
    "date_created": "2012-07-23T18:25:43.511Z",
    "last_modified": "2014-04-23T18:25:43.511Z",
    "id": 5
  }
]
```

Run the application

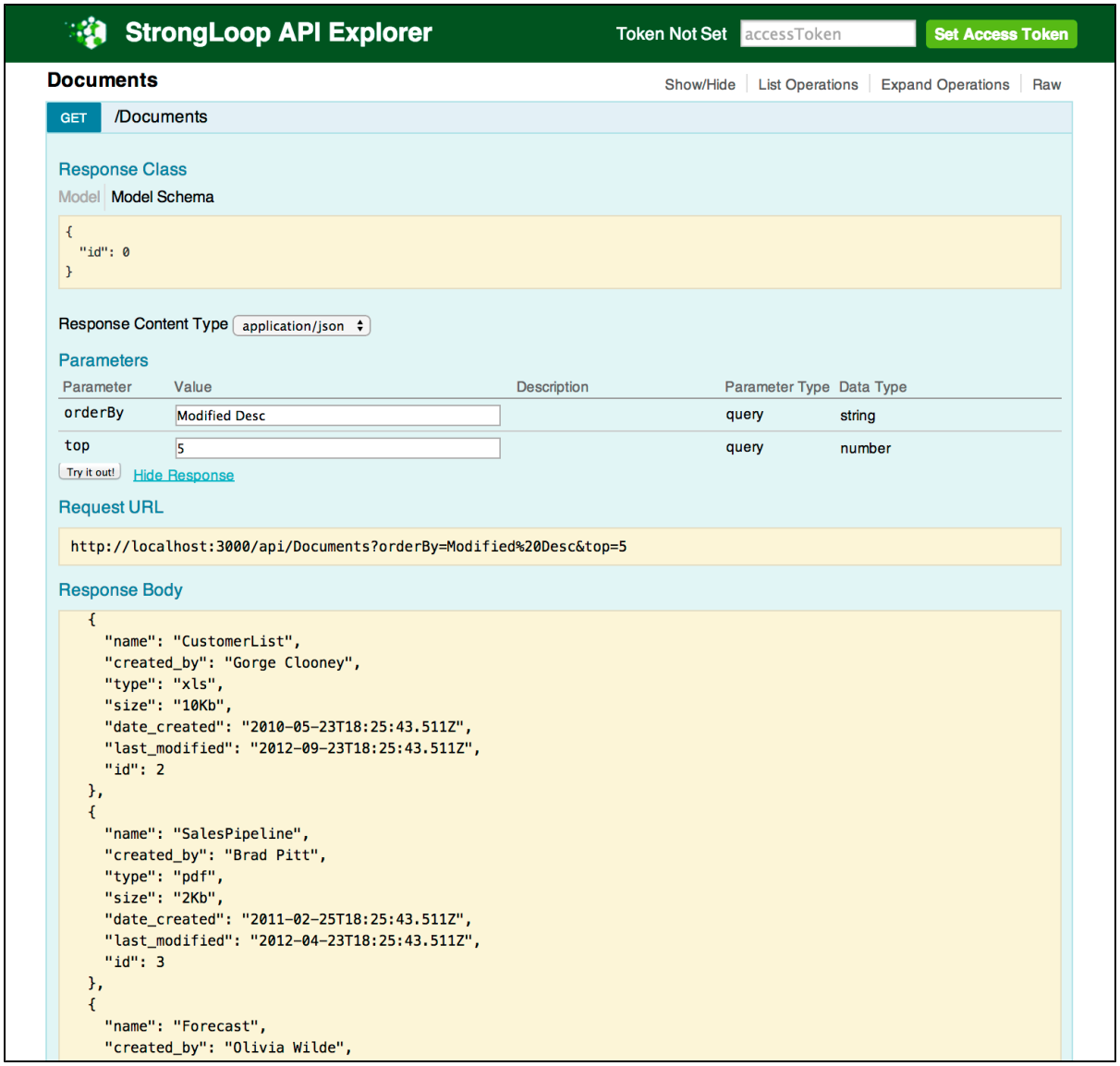
Now run the application:

```
$ slc run
```

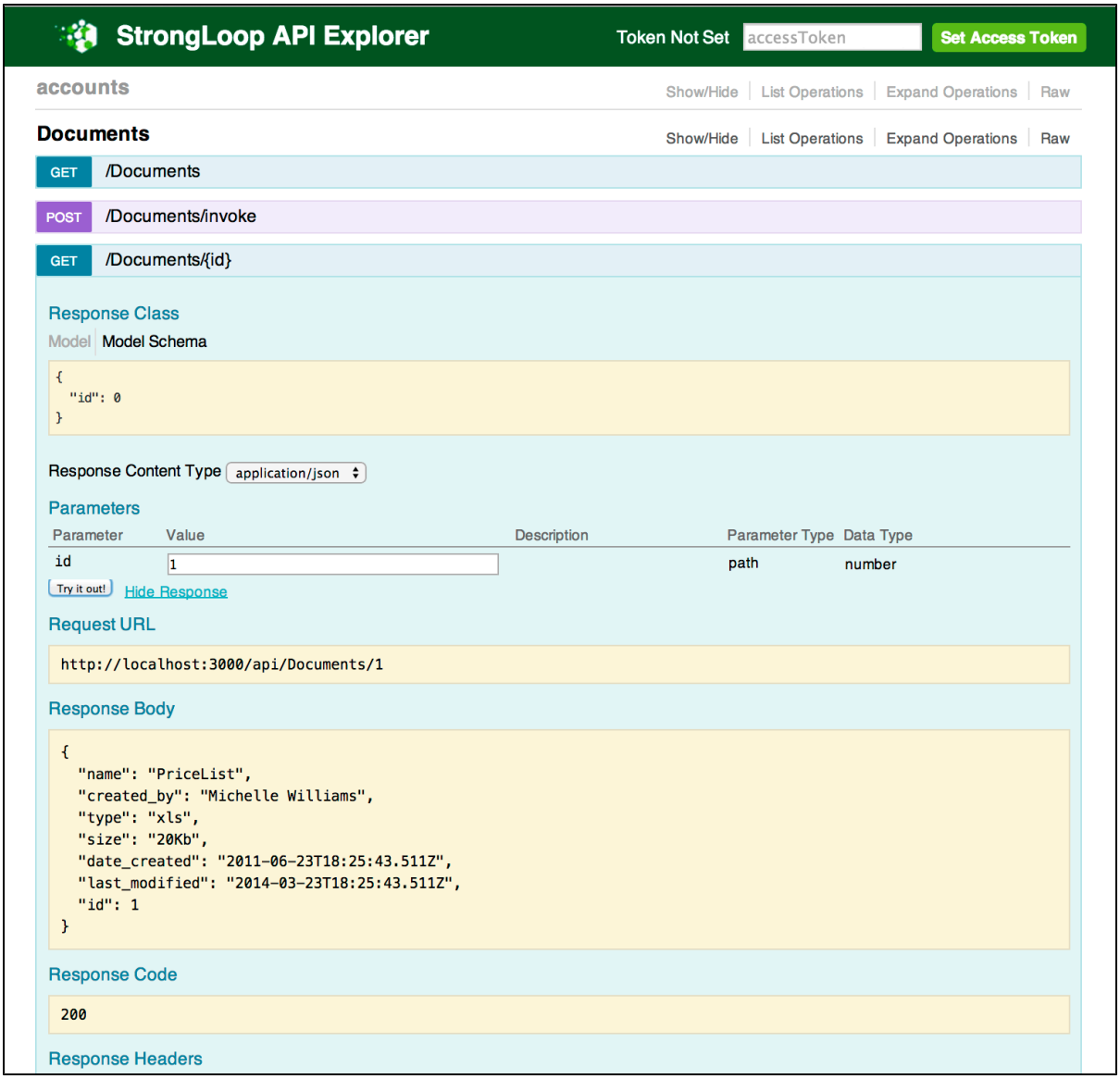
Browsing the API explorer, you can see both the API endpoints as well as individual query results.



Get list of all documents from Sharepoint



Get individual documents from Sharepoint filtered by ID



> **Next**: In Old tutorial - Add a client app, you'll add a client application that connects to the LoopBack server application.

REST example - adding a client app

In this section explains how to add an iOS client app to connect to the custom API that connects to SharePoint you created previously.

Create the iOS app

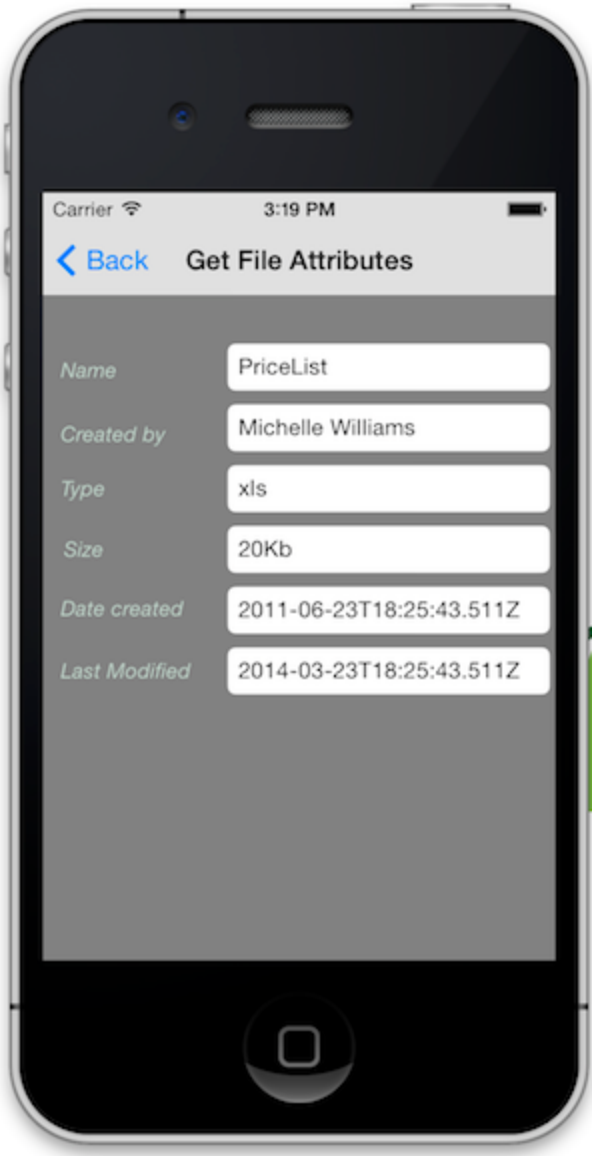
Follow the steps listed in this example of a sample Books collection application

- Creating a LoopBack iOS app: part one
- Creating a LoopBack iOS app: part two

Our app is going to be a replica copy of this app, just that we will make some visual updates to reflect a “Sharepoint Library” instead of a “Books collection”

See some screenshots of our iOS app. In case you are unable to stand up your own iOS app following the instructions in the guides above, you can also clone the following github project

```
$ git clone https://github.com/strongloop/loopback-example-APIClientApp.git
```



Key aspects of using the loopback backend API in an IOS App are:

- Import the Loopback framework (iOS SDK) as a library into the application
- Import Looback models as Prototypes
- Import the LoopBack.h header into your application just as you would Foundation/Foundation.h. Type this line:

```
#import <LoopBack/LoopBack.h>
```

- You need an Adapter to tell the SDK where to find the server. Enter this code:

```
LBRESTAdapter *adapter =
  [LBRESTAdapter adapterWithURL:[NSURL URLWithString:@"http://example.com"]];
```

- This LBRESTAdapter provides the starting point for all our interactions with the running and anxiously waiting server.
- Once we have access to adapter (for the sake of example, we'll assume the Adapter is available through our AppDelegate), we can create basic LBModel and LBModelRepository objects. Assuming we've previously created a model named "product":

```
LBRESTAdapter *adapter = [[UIApplication sharedApplication] delegate].adapter;
LBModelRepository *productReposiory = [adapter
repositoryWithModelName:@"products"];
LBModel *pen = [Product modelWithDictionary:@{ "name": "Awesome Pen" }];
```

Once you have an adapter, you can create a repository instance.

```
WidgetRepository *repository = (WidgetRepository *)[adapter
repositoryWithModelClass:[WidgetRepository class]];
```

Now that you have a WidgetRepository instance, you can create, save, find, and delete widgets, as illustrated below.

- Create a Widget:

```
Widget *pencil = (Widget *)[repository modelWithDictionary:@{ @"name": @"Pencil",
@"price": @1.50 }];
```

Save a Widget:

```
[pencil saveWithSuccess:^{
                    // Pencil now exists on the server!
                }
                failure:^(NSError *error) {
                    NSLog("An error occurred: %@", error);
                }];
```

Find another Widget:

```
[repository findWithId:@2
              success:^(LBModel *model) {
                  Widget *pen = (Widget *)model;
              }
              failure:^(NSError *error) {
                  NSLog("An error occurred: %@", error);
              }];
```

Remove a Widget:

```
[pencil destroyWithSuccess:^{
                       // No more pencil. Long live Pen!
                   }
```

SOAP connector

The SOAP connector enables LoopBack applications to interact with SOAP-based web services described using WSDL.

Installation

In your application root directory, enter:

```
$ npm install loopback-connector-soap --save
```

This will install the module from npm and add it as a dependency to the application's package.json file.

Creating a data source

Use the LoopBack datasource generator to add a REST data source to your application.

SOAP data source properties

The following table describes the SOAP data source properties you can set in `datasources.json`.

Property	Type	Description
url	String	URL to the SOAP web service endpoint. If not present, defaults to the `location` attribute of the SOAP address for the service/port from the WSDL document; for example: `<wsdl:service name="Weather"> <wsdl:port name="WeatherSoap" binding="tns:WeatherSoap"> <soap:address location="http://wsf.cdyne.com/WeatherWS/Weather.asmx" /> </wsdl:port> ... </wsdl:service>`
wsdl	String	HTTP URL or local file system path to the WSDL file, if not present, defaults to `?wsdl`.
remotingEnabled	Boolean	Indicates whether the operations are exposed as REST APIs. To expose or hide a specific method, you can override this with: `<Model>.<method>.shared = true / false;`
operations	Object	Maps WSDL binding operations to Node.js methods. Each key in the JSON object becomes the name of a method on the model. See Operations property below.

Operations property

The operations property value is a JSON object that has a property (key) for each method being defined for the model. The corresponding value is an object with the following properties:

Property	Type	Description
service	String	WSDL service name
port	String	WSDL port name
operation	String	WSDL operation name

Here is an example operations property for the stock quote service:

```
operations: {
  // The key is the method name
  stockQuote: {
    service: 'StockQuote', // The WSDL service name
    port: 'StockQuoteSoap', // The WSDL port name
    operation: 'GetQuote' // The WSDL operation name
  },
  stockQuote12: {
    service: 'StockQuote',
    port: 'StockQuoteSoap12',
    operation: 'GetQuote'
  }
}
```

Creating a model from a SOAP data source

The SOAP connector loads WSDL documents asynchronously. As a result, the data source won't be ready to create models until it's connected. The recommended way is to use an event handler for the 'connected' event; for example:

```
ds.once('connected', function () {
   // Create the model
   var WeatherService = ds.createModel('WeatherService', {});
   ...
}
```

Extending a model to wrap and mediate SOAP operations

Once you define the model, you can extend it to wrap or mediate SOAP operations and define new methods. The following example simplifies the `GetCityForecastByZIP` operation to a method that takes `zip` and returns an array of forecasts.

```
// Refine the methods
WeatherService.forecast = function (zip, cb) {
    WeatherService.GetCityForecastByZIP({ZIP: zip || '94555'}, function (err,
response) {
        console.log('Forecast: %j', response);
        var result = (!err && response.GetCityForecastByZIPResult.Success) ?
        response.GetCityForecastByZIPResult.ForecastResult.Forecast : [];
        cb(err, result);
    });
};
```

The custom method on the model can be exposed as REST APIs. It uses the `loopback.remoteMethod` to define the mappings.

```
// Map to REST/HTTP
loopback.remoteMethod(
    WeatherService.forecast, {
       accepts: [
         {arg: 'zip', type: 'string', required: true, http: {source: 'query'}}
       ],
       returns: {arg: 'result', type: 'object', root: true},
       http: {verb: 'get', path: '/forecast'}
    }
);
```

Examples

The loopback-connector-soap repository provides several examples:

Get stock quotes by symbols: stock-ws.js. Run with the command:

```
$ node example/stock-ws
```

Get weather and forecast information for a given zip code: weather-ws.js. Run with the command:

```
$ node example/weather-ws
```

Expose REST APIs to proxy the SOAP web services: weather-rest.js. Run with the command:

```
$ node example/weather-rest
```

View the results at http://localhost:3000/explorer.

SQL Server connector

The LoopBack SQL Server connector enables LoopBack applications to connect to Microsoft SQL Server databases.

See also:

- Example application (GitHub)
- Database discovery API

Installation

In your application root directory, enter:

```
$ npm install loopback-connector-mssql --save
```

This will install the module from npm and add it as a dependency to the application's package.json file.

Creating a data source

If your application has the standard Project layout, the easiest way to create a SQL Server data source is with the LoopBack datasource generator, for example:

```
$ slc loopback:datasource
[?] Enter the data-source name: sqlserverdb
[?] Select the connector for sqlserverdb: Microsoft SQL (supported by StrongLoop)
```

This adds the following to the application's `server/datasources.json` file:

```
"sqlserverdb": {
    "name": "sqlserverdb",
    "connector": "mssql"
}
```

Note that "sqlserverdb" is just an arbitrary identifier for the connection.

Connector settings

To configure the data source to use your MS SQL Server database, edit `datasources.json` and add the following settings as appropriate. The MSSQL connector uses node-mssql as the driver. For more information about configuration parameters, see node-mssql documentation.

Property	Default	Description
host or hostname	localhost	Host name or IP address of the Microsoft SQL Server
port	1433	Port number of the Microsoft SQL Server database server.
username or user	N/A	User name to connect to the Microsoft SQL Server
password	N/A	Password for specified user name
database	N/A	Microsoft SQL Server database name
schema	dbo	The database schema

For example:

datasources.json

```
...
"accountDB": {
    "connector": "mssql",
    "host": "demo.strongloop.com",
    "port": 3306,
    "database": "demo",
    "username": "demo",
    "password": "L00pBack"
  }
...
```

Alternatively you can use a single 'url' property that combines all the database configuration settings, for example:

```
"accountDB": {
    "url": "mssql://test:mypassword@localhost:1433/demo?schema=dbo"
}
```

The application will automatically load the datasource when it starts. You can then refer to it in code, for example:

```
var app = require('./app');
var dataSource = app.dataSources.accountDB;
```

Alternatively, you can create the data source in application code; for example:

```
var DataSource = require('loopback-datasource-juggler').DataSource;
var dataSource = new DataSource('mssql', config);
config = { ... };  // JSON object as specified above in "Connector settings"
```

Defining models

The model definition consists of the following properties:

- name: Name of the model, by default, it's the camel case of the table
- options: Model level operations and mapping to Microsoft SQL Server schema/table
- properties: Property definitions, including mapping to Microsoft SQL Server columns

For example:

› Expand source

```
{"name": "Inventory",
     "options": {
       "idInjection": false,
       "mssql": {
         "schema": "strongloop",
         "table": "inventory"
       }
     }, "properties": {
      "id": {
        "type": "String",
        "required": false,
        "length": 64,
        "precision": null,
        "scale": null,
        "mssql": {
          "columnName": "id",
          "dataType": "varchar",
          "dataLength": 64,
          "dataPrecision": null,
          "dataScale": null,
          "nullable": "NO"
        }
      },
      "productId": {
        "type": "String",
        "required": false,
        "length": 64,
        "precision": null,
        "scale": null,
        "id": 1,
        "mssql": {
          "columnName": "product_id",
          "dataType": "varchar",
          "dataLength": 64,
          "dataPrecision": null,
```

```
      "dataScale": null,
      "nullable": "YES"
    }
  },
  "locationId": {
    "type": "String",
    "required": false,
    "length": 64,
    "precision": null,
    "scale": null,
    "id": 1,
    "mssql": {
      "columnName": "location_id",
      "dataType": "varchar",
      "dataLength": 64,
      "dataPrecision": null,
      "dataScale": null,
      "nullable": "YES"
    }
  },
  "available": {
    "type": "Number",
    "required": false,
    "length": null,
    "precision": 10,
    "scale": 0,
    "mssql": {
      "columnName": "available",
      "dataType": "int",
      "dataLength": null,
      "dataPrecision": 10,
      "dataScale": 0,
      "nullable": "YES"
    }
  },
  "total": {
    "type": "Number",
    "required": false,
    "length": null,
    "precision": 10,
    "scale": 0,
    "mssql": {
      "columnName": "total",
      "dataType": "int",
      "dataLength": null,
      "dataPrecision": 10,
      "dataScale": 0,
      "nullable": "YES"
```

```
        }
      }
    }}
```

Auto migrating and auto-updating

After making changes to model properties you must call `Model.automigrate()` or `Model.autoupdate()`. Call `Model.automigrate()` only on a new model, since it will drop existing tables.

For each model, the LoopBack SQL Server connector creates a table in the 'dbo' schema in the database.

Destroying models

Destroying models may result in errors due to foreign key integrity. First delete any related models first calling delete on models with relationships.

Type mapping

See LoopBack types for details on LoopBack's data types.

LoopBack to SQL Server types

LoopBack Type	SQL Server Type
Boolean	BIT
Date	DATETIME
GeoPoint	FLOAT
Number	INT
String JSON	NVARCHAR

SQL Server to LoopBack types

SQL Server Type	LoopBack Type
BIT	Boolean
BINARY VARBINARY IMAGE	Node.js Buffer object
DATE DATETIMEOFFSET DATETIME2 SMALLDATETIME DATETIME TIME	Date
POINT	GeoPoint
BIGINT NUMERIC SMALLINT DECIMAL SMALLMONEY INT TINYINT MONEY FLOAT REAL	Number

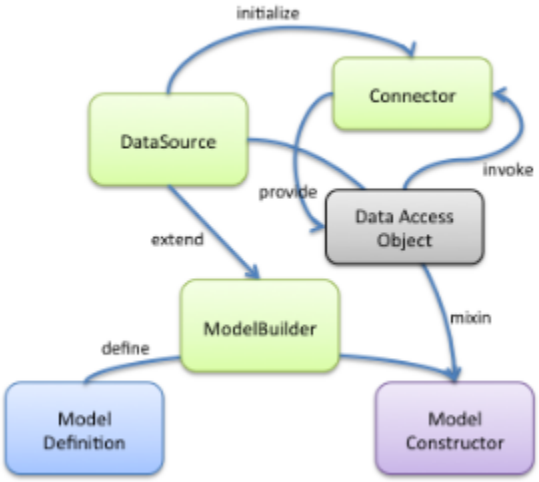
CHAR VARCHAR TEXT NCHAR NVARCHAR NTEXT CHARACTER VARYING CHARACTER	String

Discovery methods

LoopBack provides a unified API to create models based on schema and tables in relational databases. The same discovery API is available when using connectors for Oracle, MySQL, PostgreSQL, and SQL Server. For more information, see Database discovery API.

Advanced topics: data sources

Overview



The diagram illustrates the relationship between LoopBack `Model`, `DataSource`, and `Connector`.

1. Define the model using LoopBack Definition Language (LDL). This provides a model definition in JSON or as a JavaScript object.
2. Create an instance of `ModelBuilder` or `DataSource`. `DataSource` extends from `ModelBuilder`. `ModelBuilder` is responsible for compiling model definitions to JavaScript constructors representing model classes. `DataSource` inherits that function from `ModelBuilder`.
3. Use `ModelBuilder` or `DataSource` to build a JavaScript constructor (i.e. the model class) from the model definition. Model classes built from `ModelBuilder` can be later attached to a DataSource to receive the mixin of data access functions.
4. As part of step 2, `DataSource` initializes the underlying `Connector` with a settings object which provides configurations to the connector instance. `Connector` collaborates with `DataSource` to define the functions as `DataAccessObject` to be mixed into the model class.
   The `DataAccessObject` consists of a list of static and prototype methods. It can be CRUD operations or other specific functions depending on the connector's capabilities.

The `DataSource` object is the unified interface for LoopBack applications to integrate with backend systems. It's a factory for data access logic around model classes. With the ability to plug in various connectors, `DataSource` provides the necessary abstraction to interact with databases or services to decouple the business logic from plumbing technologies.

Creating a DataSource programmatically

The `DataSource` constructor is in `loopback-datasource-juggler`. The DataSource constructor accepts two arguments:

- connector: The name or instance of the connector module.
- settings: An object of properties to configure the connector.

For example:

```
var DataSource = require('loopback-datasource-juggler').DataSource;

var dataSource = new DataSource({
    connector: require('loopback-connector-mongodb'),
    host: 'localhost',
    port: 27017,
    database: 'mydb'
});
```

The `connector` argument passed the `DataSource` constructor can be one of the following:

- The connector module from `require(connectorName)`
- The full name of the connector module, such as 'loopback-connector-oracle'
- The short name of the connector module, such as 'oracle', which will be converted to 'loopback-connector-'
- A local module under ./connectors/ folder

```
var ds1 = new DataSource('memory');
var ds2 = new DataSource('loopback-connector-mongodb'));
var ds3 = new DataSource(require('loopback-connector-oracle'));
```

LoopBack provides the built-in memory connector that uses in-memory store for CRUD operations.

The `settings` argument configures the connector. Settings object format and defaults depends on specific connector, but common fields are:

- `host`: Database host
- `port`: Database port
- `username`: Username to connect to database
- `password`: Password to connect to database
- `database`: Database name
- `debug`: Turn on verbose mode to debug db queries and lifecycle

For connector-specific settings, see the connector's documentation.

Creating a model from a data source

`DataSource` extends from `ModelBuilder`, which is a factory for plain model classes that only have properties. `DataSource` connects to databases and other backend systems using `Connector`.

```
var DataSource = require('loopback-datasource-juggler').DataSource;
var ds = new DataSource('memory');

var User = ds.define('User', {
    name: String,
    bio: String,
    approved: Boolean,
    joinedAt: Date,
    age: Number
});
```

All model classes within single data source share the same connector type and one database connection or connection pool. But it's possible to use more than one data source to connect to different databases.
Alternatively, you can attach a plain model constructor created from `ModelBuilder` to a `DataSource`.

```
var ModelBuilder = require('loopback-datasource-juggler').ModelBuilder;
var builder = new ModelBuilder();

var User = builder.define('User', {
        name: String,
        bio: String,
        approved: Boolean,
        joinedAt: Date,
        age: Number
});

var DataSource = require('loopback-datasource-juggler').DataSource;
var ds = new DataSource('memory');

User.attachTo(ds); // The CRUD methods will be mixed into the User constructor
```

Creating a data source for a connector

Application code does not directly use a connector. Rather, you create a DataSource to interact with the connector.

The simplest example is for the in-memory connector:

```
var memory = loopback.createDataSource({
  connector: loopback.Memory
});
```

Here is another example, this time for the Oracle connector:

```
var DataSource = require('loopback-datasource-juggler').DataSource;
var oracleConnector = require('loopback-connector-oracle');

var ds = new DataSource(oracleConnector, {
    host : 'localhost',
    database : 'XE',
    username : 'username',
    password : 'password',
    debug : true
});
```

The connector argument passed the DataSource constructor can be one of the following:

- The connector module from `require('connectorName')`
- The full name of the connector module, such as `'loopback-connector-oracle'`.
- The short name of the connector module, such as `'oracle'`, that LoopBack converts to `'loopback-connector-oracle'` (for example).
- A local module in the `/connectors` folder

Initializing a connector

The connector module can export an `initialize` function to be called by the owning DataSource instance.

```
exports.initialize = function (dataSource, postInit) {

    var settings = dataSource.settings || {}; // The settings is passed in from the
dataSource

    var connector = new MyConnector(settings); // Construct the connector instance
    dataSource.connector = connector; // Attach connector to dataSource
    connector.dataSource = dataSource; // Hold a reference to dataSource
    ...
};
```

The DataSource calls the `initialize` method with itself and an optional `postInit` callback function. The connector receives the settings from the `dataSource` argument and use it to configure connections to backend systems.

Please note connector and dataSource set up a reference to each other.

Upon initialization, the connector might connect to database automatically. Once connection established dataSource object emit 'connected' event, and set `connected` flag to true, but it is not necessary to wait for 'connected' event because all queries cached and executed when dataSource emit 'connected' event.

To disconnect from database server call `dataSource.disconnect` method. This call is forwarded to the connector if the connector have ability to connect/disconnect.

Building a connector

Overview

This article is for developers who want to create a new connector type to connect to a data source not currently supported.

Interface

A connector module can implement the following methods to interact with the data source.

```
exports.initialize = function (dataSource, postInit) {

    var settings = dataSource.settings || {}; // The settings is passed in from the
dataSource

    var connector = new MyConnector(settings); // Construct the connector instance
    dataSource.connector = connector; // Attach connector to dataSource
    connector.dataSource = dataSource; // Hold a reference to dataSource

    /**
     * Connector instance can have an optional property named as DataAccessObject that
provides
     * static and prototype methods to be mixed into the model constructor. The
property can be defined
     * on the prototype.
     */
    connector.DataAccessObject = function {};

    /**
     * Connector instance can have an optional function to be called to handle data
model definitions.
     * The function can be defined on the prototype too.
     * @param model The name of the model
     * @param properties An object for property definitions keyed by propery names
     * @param settings An object for the model settings
     */
    connector.define = function(model, properties, settings) {
        ...
    };

    connector.connect(..., postInit); // Run some async code for initialization
    // process.nextTick(postInit);
}
```

Another way is to directly export the connection function which takes a settings object.

```
module.exports = function(settings) {
    ...
}
```

Implementing a CRUD connector

To support CRUD operations for a model class that is attached to the dataSource/connector, the connector needs to provide the following functions:

```
/**
 * Create a new model instance
 */
CRUDConnector.prototype.create = function (model, data, callback) {
};

/**
 * Save a model instance
 */
CRUDConnector.prototype.save = function (model, data, callback) {
```

```
};

/**
 * Check if a model instance exists by id
 */
CRUDConnector.prototype.exists = function (model, id, callback) {
};

/**
 * Find a model instance by id
 */
CRUDConnector.prototype.find = function find(model, id, callback) {
};

/**
 * Update a model instance or create a new model instance if it doesn't exist
 */
CRUDConnector.prototype.updateOrCreate = function updateOrCreate(model, data,
callback) {
};

/**
 * Delete a model instance by id
 */
CRUDConnector.prototype.destroy = function destroy(model, id, callback) {
};

/**
 * Query model instances by the filter
 */
CRUDConnector.prototype.all = function all(model, filter, callback) {
};

/**
 * Delete all model instances
 */
CRUDConnector.prototype.destroyAll = function destroyAll(model, callback) {
};

/**
 * Count the model instances by the where criteria
 */
CRUDConnector.prototype.count = function count(model, callback, where) {
};

/**
 * Update the attributes for a model instance by id
 */
```

```
CRUDConnector.prototype.updateAttributes = function updateAttrs(model, id, data,
callback) {
};
```

Authentication and authorization

Most applications need to control who (or what) can access data. Typically, this involves requiring users to login to access protected data, or requiring authorization tokens for other applications to access protected data.

For a simple example of implementing LoopBack access control, see the GitHub loopback-example-access-control repository.

See also:

- Third-party login (Facebook, Google, etc.)
- Access control models

⚠ LoopBack does not currently support application-scoped access tokens. Application identity will be provided with oAuth2 support, which is an upcoming feature.

LoopBack apps access data through models (see Working with models), so controlling access to data means putting restrictions on models; that is, specifying who or what can read, write, or change the data in the models.

The general process to implement access control for an application is:

1. **Specify user roles**. Define the user roles that your application requires. For example, you might create roles for anonymous users, authorized users, and administrators.
2. **Define access for each role and model method**. For example, you might enable anonymous users to read a list of banks, but not allow them to do anything else.
   LoopBack models have a set of built-in methods, and each method maps to either the READ or WRITE access type. In essence, this step amounts to specifying whether access is allowed for each role and each Model + access type, as illustrated in the example below.
3. **Implement authentication**: in the application, add code to create (register) new users, login users (get and use authentication tokens), and logout users.

Controlling data access

Specifying user roles

The first step in specifying user roles is to determine what roles your application needs. Most applications will have un-authenticated or anonymous users (those who have not logged in) and authenticated users (those who have logged in). Additionally, many applications will have an administrative role that provides broad access rights. And applications can have any number of additional user roles as appropriate.

User access types

LoopBack provides a built-in User model with a corresponding REST API that inherits all the "CRUD" (create, reade, update, and delete) methods of the Data access object. Each data access method on the LoopBack User model maps to either the READ or WRITE access type, as follows:

READ:

- exists - Boolean method that determines whether a user exists
- findById - Find a user by ID
- find - Find all users that match specified conditions.
- findOne - Finds a single user instance that matches specified conditions.
- count - Returns the number of users that match the specified conditions.

WRITE:

- create - create a new user
- upsert (equivalent to updateOrCreate) - update or insert a new user record.
- deleteById (equivalent to removeById or destroyById) - delete the user with the specified ID.

For other methods, the default access type is EXECUTE; for example, a custom method maps to the EXECUTE access type.

Defining access control

Use the LoopBack ACL generator to set up access control for an application. Before you do that, though, you must have a clear idea of how you're going to configure access control for your application.

The table below is an example of the access control specification for the LoopBack access control example application. It specifies ALLOW or DENY for any combination of role and access type (READ or WRITE for a specific model).

Model and access type	Anonymous user role	Authenticated user role	Teller (admin) role
READ Bank	ALLOW	ALLOW	ALLOW
WRITE Bank	DENY	DENY	ALLOW
READ Account	DENY	ALLOW	ALLOW
WRITE Account	DENY	DENY	ALLOW
READ Transaction	DENY	ALLOW	ALLOW
WRITE Transaction	DENY	DENY	DENY

No one has WRITE access to the Transaction model, because it is only updated by the system, when a user performs a transaction; you never directly update the model.

Once you've created this kind of specification, you can easily construct `slc loopback:acl` commands to set up access control, as illustrated below.

Using the Yeoman ACL generator to define access control

The easiest way to define access control for an app is with the Yeoman ACL sub-generator. This enables you to create a static definition before runtime. The general syntax is:

```
slc loopback:acl

...
```

For more information, see LoopBack ACL generator.

Creating and authenticating users

The basic process for an application to create and authenticate users is:

1. Register a new user with the `User.create()` method, inherited from the generic Model object. See Creating users for details.
2. Request an access token from the client application on behalf of the user by calling `User.login()`. You can create your own access tokens to customize authentication. `User.login()` is a good example of doing that. You need to do verification on the server though to prevent someone from creating tokens even though they don't belong to them.
3. Invoke an API using the access token. Provide the access token in the HTTP header or as a query parameter to the REST API call, as shown in Making authenticated requests with access tokens.

Creating users

Create (register) a new user with the `User.create` method.

REST

```
curl -X POST -H "Content-Type:application/json" \
-d '{"email": "me@domain.com", "password": "secret"}' \
http://localhost:3000/api/users
```

Node.js

```
User.create({
  email: 'me@domain.com',   // required by default
  password: 'secret'        // required by default
}, function (err, user) {
  console.log(user.id);     // => the user id (default type: db specific | number)
  console.log(user.email);  // => the user's email
});
```

Logging in and authenticating users

Authenticate a user by calling the `User.login()` method and providing a credentials object.

By default, you must provide a `password` and either a `username` or `email`. You may also specify how long you would like the access token to be valid for by providing a `ttl` (time to live) in **seconds**. See the access token section below.

REST

```
curl -X POST -H "Content-Type:application/json" \
-d '{"email": "me@domain.com", "password": "secret", "ttl": 1209600000}' \
http://localhost:3000/api/users/login
```

This example returns:

```
{
  "id": "GOkZRwgZ61q0XXVxvxlB8TS1D6lrG7Vb9V8YwRDfy3YGAN7TM7EnxWHqdbIZfheZ",
  "ttl": 1209600,
  "created": "2013-12-20T21:10:20.377Z",
  "userId": 1
}
```

Node.js

```
var TWO_WEEKS = 1000 * 60 * 60 * 24 * 7 * 2;
User.login({
  email: 'me@domain.com',           // must provide email or "username"
  password: 'secret',               // required by default
  ttl: TWO_WEEKS                    // keep the AccessToken alive for at least two
weeks
}, function (err, accessToken) {
  console.log(accessToken.id);      // => GOkZRwg... the access token
  console.log(accessToken.ttl);     // => 1209600 time to live
  console.log(accessToken.created); // => 2013-12-20T21:10:20.377Z
  console.log(accessToken.userId);  // => 1
});
```

Making authenticated requests with access tokens

If a login attempt is successful a new AccessToken is created that points to the user. This token is required when making subsequent REST requests for the access control system to validate that the user can invoke methods on a given `Model`.

REST

```
ACCESS_TOKEN=6Nb2ti5QEXIoDBS5FQGWIz4poRFiBCMMYJbYXSGHWuulOuy0GTEuGx2VCEVvbpBK

# Authorization Header
curl -X GET -H "Authorization: $ACCESS_TOKEN" \
http://localhost:3000/api/widgets

# Query Parameter
curl -X GET http://localhost:3000/api/widgets?access_token=$ACCESS_TOKEN
```

Logging out users and deleting access tokens

A user will be effectively logged out by deleting the access token they were issued at login. This affects only the specified access token; other tokens attached to the user will still be valid.

To destroy access tokens over REST API, use the `/logout` endpoint.

REST

```
ACCESS_TOKEN=6Nb2ti5QEXIoDBS5FQGWIz4poRFiBCMMYJbYXSGHWuulOuy0GTEuGx2VCEVvbpBK
VERB=POST # any verb is allowed

# Authorization Header
curl -X VERB -H "Authorization: $ACCESS_TOKEN" \
http://localhost:3000/api/users/logout

# Query Parameter
curl -X VERB http://localhost:3000/api/users/logout?access_token=$ACCESS_TOKEN
```

Node.js

```
var USER_ID = 1;
var ACCESS_TOKEN = '6Nb2ti5QEXIoDBS5FQGWIz4poRFiBCMMYJbYXSGHWuulOuy0GTEuGx2VCEVvbpBK';
// remove just the token
var token = new AccessToken({id: ACCESS_TOKEN});
token.destroy();
// remove all user tokens
AccessToken.destroyAll({
  where: {userId: USER_ID}
});
```

Defining roles

LoopBack enables you to define both static and dynamic roles. Static roles are stored in a data source and are mapped to users. In contrast, dynamic roles aren't assigned to users and are determined during access.

Here is an example defining a new static role and assigning a user to that role.

```
// Create a new user
User.create({name: 'John', email: 'x@y.com', password: 'foobar'}, function (err, user)
{
 // Create the static admin Role
 Role.create({name: 'admin'}, function (err, role) {
   // Make John an admin
   role.principals.create({principalType: RoleMapping.USER, principalId: user.id});
 });
});
```

Now you can use the role defined above in the access controls. For example, add the following to `models.json` to enable users in the "admin" role to call all REST APIs.

```
{
 "accessType": "*",
 "permission": "ALLOW",
 "principalType": "ROLE",
 "principalId": "admin"
}
```

Sometimes static roles aren't flexible enough. In the first example, we used the "$owner" dynamic role to allow access to the owner of the requested todo model. Below is an example defining our own dynamic role.

```
Role.registerResolver('$friend', function(role, ctx, callback) {
 var targetUserId = ctx.modelId;
 // Below has a callback signature: callback(err, isFriend)
 MyUser.isFriend(targetUserId, ctx.getUserId(), callback);
});
```

Using the dynamic role defined above, we can restrict access of user info to users that are friends.

```
{
 "accessType": "READ",
 "permission": "ALLOW",
 "principalType": "ROLE",
 "principalId": "$friend"
}
```

Advanced topics

Manually enabling access control

If you created your app with `slc loopback`, then you don't need to do anything to enable access control.

Otherwise, if you're adding access control manually, you must call the LoopBack `enableAuth()` method, for example:

```
var loopback = require('loopback');
var app = loopback();
app.enableAuth();
```

Defining access control at runtime

In some applications, you may need to make changes to ACL definitions at runtime. There are two ways to do this:

- Call the DataSource method `createModel()`, providing an ACL specification (in LDL) as an argument.
- The `ACL.create()` method. You can apply this at run-time.

Using DataSource createModel() method

You can also control access to a model by passing an LDL specification when creating the model with the data source createModel() method.

```
var Customer = loopback.createModel('Customer', {
      name: {
        type: String,
        // Property level ACLs
        acls: [
          {principalType: ACL.USER, principalId: 'u001', accessType: ACL.WRITE,
permission: ACL.DENY},
          {principalType: ACL.USER, principalId: 'u001', accessType: ACL.ALL,
permission: ACL.ALLOW}
        ]
      }
    }, {
      // By default, access will be denied if no matching ACL entry is found
      defaultPermission: ACL.DENY,
      // Model level ACLs
      acls: [
        {principalType: ACL.USER, principalId: 'u001', accessType: ACL.ALL,
permission: ACL.ALLOW}
      ]
    });
```

For more information on LDL, see LoopBack Definition Language.

Using the ACL create() method

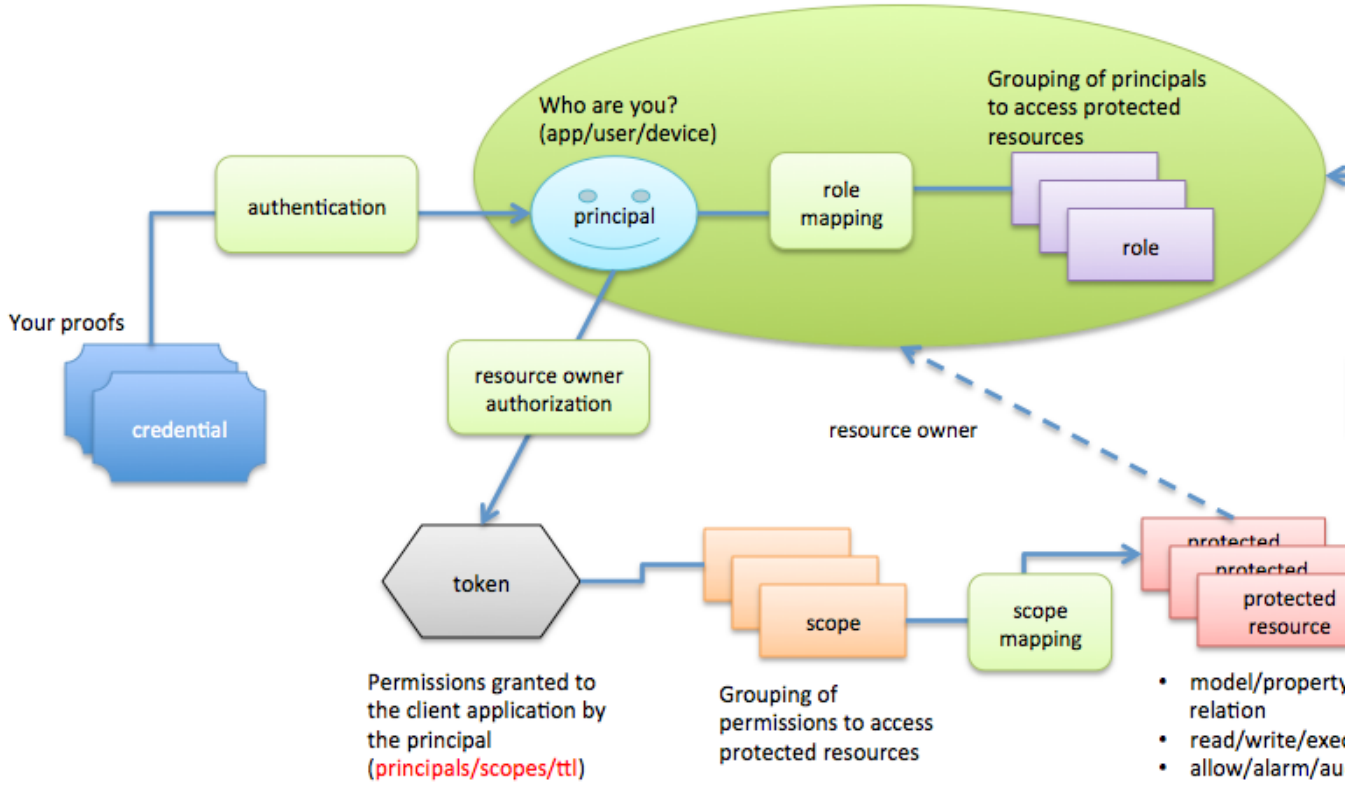
ACLs defined as part of the model creation are hard-coded into your application. LoopBack also allows you dynamically defines ACLs through code or a dashboard. The ACLs can be saved to and loaded from a database.

```
ACL.create({principalType: ACL.USER, principalId: 'u001', model: 'User', property:
ACL.ALL,
    accessType: ACL.ALL, permission: ACL.ALLOW}, function (err, acl) {...});

ACL.create({principalType: ACL.USER, principalId: 'u001', model: 'User', property:
ACL.ALL,
 accessType: ACL.READ, permission: ACL.DENY}, function (err, acl) {...});
```

See Using built-in models for more information.

Architecture

The following diagram illustrates the architecture of the LoopBack access control system.



Security considerations

CORS

By default LoopBack enables Cross-origin resource sharing (CORS).

If you are using a JavaScript client, you must also enable CORS on the client side. For example, one way to enable it with AngularJS is:

```
var myApp = angular.module('myApp', [
    'myAppApiService']);

myApp.config(['$httpProvider', function($httpProvider) {
        $httpProvider.defaults.useXDomain = true;
        delete $httpProvider.defaults.headers.common['X-Requested-With'];
    }
]);
```

Mitigating XSS exploits

LoopBack stores the user's access token in a JavaScript object, which may make it susceptible to a cross-site scripting (XSS) security exploit. As a best practice to mitigate such threats, use appropriate Express middleware, for example:

- Lusca
- Helmet

See also Express 3.x csrf() function.

LoopBack components

LoopBack applications are composed of *components*. Components are bundles of functionality that you can add to your LoopBack application. Your main LoopBack application itself is nothing more than a grouping of components with a few added elements to tie them all together like a main `app.js`.

Component Name	**Component Description**	NPM Module
Push Notifications	Adds push notification capabilities to your LoopBack application as a mobile back end service.	loopback-component-push
Storage service	Adds an interface to abstract storage providers like S3, filesystem into general containers and files.	loopback-component-storage
Synchronization	Adds replication capability between LoopBack running in a browser or between LoopBack back-end instances to enable offline synchronization and server-to-server data synchronization.	Built into LoopBack; will be refactored into loopback-component-sync
LoopBack Passport	Adds third party login capabilities to your LoopBack application like Facebook, GitHub etc.	loopback-component-passport

Push notifications

See also:

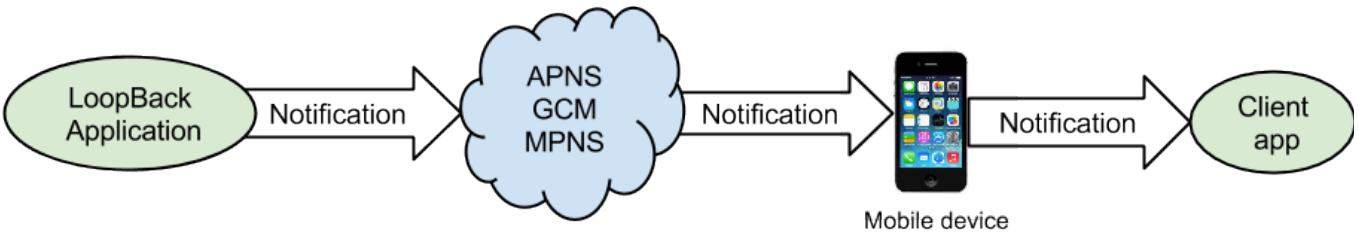
- API reference
- Example server application
- Example iOS app
- Example Android app

Overview

Push notifications enable server applications (known as *providers* in push parlance) to send information to mobile apps even when the app isn't in use. The device displays the information using a "badge," alert, or pop up message. A push notification uses the service provided by the device's operating system:

- **iOS** - Apple Push Notification service (APNS)
- **Android** - Google Cloud Messaging (GCM)

The following diagram illustrates how it works.



The components involved on the server are:

- Device model and APIs to manage devices with applications and users.
- Application model to provide push settings for device types such as iOS and Android.
- Notification model to capture notification messages and persist scheduled notifications.
- Optional job to take scheduled notification requests.

- Push connector that interacts with device registration records and push providers APNS for iOS apps and GCM for Android apps.
- Push model to provide high-level APIs for device-independent push notifications.

To send push notifications, you'll need to create a LoopBack server application, then configure your iOS and Android client apps accordingly. You can either use the example LoopBack push application or create your own. Using the example application is a good way to learn about LoopBack push notifications.

Use the LoopBack push notification sample application

If you are creating your own LoopBack server application, skip this section and go to Set up your LoopBack application to send push notifications.

First, download the sample app:

```
$ git clone https://github.com/strongloop/loopback-component-push.git
```

Set up messaging credentials for Android apps

First, if you haven't already done so, get your Google Cloud Messaging (GCM) credentials for Android apps. After following the instructions, you will have a GCM API key. Then edit `example/server/config.js` and look for the line:

```
exports.gcmServerApiKey = 'Your-server-api-key';
```

Replace `Your-server-api-key` with your GCM server API key. For example:

```
exports.gcmServerApiKey = 'AIzaSyDEPWYN9Dxf3xDOqbQluCwuHsGfK4aJehc';
```

Set up messaging credentials for iOS apps

If you have not already done so, create your APNS certificates for iOS apps. After following the instructions, you will have APNS certificates on your system. Then edit `example/server/config.js` and look for these lines:

```
exports.apnsCertData = readCredentialsFile('apns_cert_dev.pem');
exports.apnsKeyData = readCredentialsFile('apns_key_dev.pem');
```

Replace the file names with the names of the files containing your APNS certificates. By deafult, `readCredentialsFile()` looks in the `/credentials` sub-directory for your APNS certificates.

If you don't have a client app yet, leave the default appName in `config.js` for now. Once you have created your client app, update the appName.

Now follow the instructions in:

- Push notifications for Android apps to set up Android client apps.
- Push notifications for iOS apps to set up iOS client apps

Run the sample server application

First install all dependencies, then run the Node application as follows:

```
$ cd example/server
$ npm install
...
$ node app
```

You will see the message:

```
The server is running at http://127.0.0.1:3010
```

Set up your LoopBack application to send push notifications

Follow the directions in this section to configure your own LoopBack application to send push notifications. It may be helpful to refer to the example LoopBack application.

Install Loopback push notification module

```
$ npm install loopback-component-push
```

Create a push model

To send push notifications, you must create a push model. The code below illustrates how to do this with a database as the data source. The database is used to load and store the corresponding application/user/installation models.

```
var loopback = require('loopback');
var app = loopback();
var db = require('./data-sources/db');
// Load & configure loopback-component-push
var PushModel = require('loopback-component-push')(app, { dataSource: db });
var Application = PushModel.Application;
var Installation = PushModel.Installation;
var Notification = PushModel.Notification;
```

Configure the application with push settings

Register a mobile application

The mobile application needs to register with LoopBack so it can have an identity for the application and corresponding settings for push services. Use the Application model's `register()` function for sign-up.

For information on getting API keys, see:

- Get your Google Cloud Messaging credentials for Android.
- Set up iOS clients for iOS.

```
Application.register('put your developer id here',
    'put your unique application name here',
    {
      description: 'LoopBack Push Notification Demo Application',
      pushSettings: {
        apns: {
          certData: readCredentialsFile('apns_cert_dev.pem'),
          keyData: readCredentialsFile('apns_key_dev.pem'),

          pushOptions: {
          },
          feedbackOptions: {
            batchFeedback: true,
            interval: 300
          }
        },
        gcm: {
          serverApiKey: 'your GCM server API Key'
        }
      }
    },
    function(err, app) {
      if (err) return cb(err);
      return cb(null, app);
    }
  );

function readCredentialsFile(name) {
 return fs.readFileSync(
   path.resolve(__dirname, 'credentials', name),
   'UTF-8'
 );
}
```

Register a mobile device

The mobile device also needs to register itself with the backend using the Installation model and APIs. To register a device from the server side, call the `Installation.create()` function, as shown in the following example:

```
Installation.create({
    appId: 'MyLoopBackApp',
    userId: 'raymond',
    deviceToken: '756244503c9f95b49d7ff82120dc193ca1e3a7cb56f60c2ef2a19241e8f33305',
    deviceType: 'ios',
    created: new Date(),
    modified: new Date(),
    status: 'Active'
}, function (err, result) {
    console.log('Registration record is created: ', result);
});
```

Most likely, the mobile application registers the device with LoopBack using REST APIs or SDKs from the client side, for example:

```
POST http://localhost:3010/api/installations
    {
        "appId": "MyLoopBackApp",
        "userId": "raymond",
        "deviceToken":
"756244503c9f95b49d7ff82120dc193ca1e3a7cb56f60c2ef2a19241e8f33305",
        "deviceType": "ios"
    }
```

Send push notifications

Send out the push notification immediately

LoopBack provides two Node.js methods to select devices and send notifications to them:

- `notifyById()`: Select a device by registration ID and send a notification to it.
- `notifyByQuery()`: Get a list of devices using a query (same as the **where** property for `Installation.find()`) and send a notification to all of them.

For example, the code below creates a custom endpoint to send out a dummy notification for the selected device:

```
var badge = 1;
app.post('/notify/:id', function (req, res, next) {
 var note = new Notification({
   expirationInterval: 3600, // Expires 1 hour from now.
   badge: badge++,
   sound: 'ping.aiff',
   alert: '\uD83D\uDCE7 \u2709 ' + 'Hello',
   messageFrom: 'Ray'
 });

 PushModel.notifyById(req.params.id, note, function(err) {
   if (err) {
     // let the default error handling middleware
     // report the error in an appropriate way
     return next(err);
   }
   console.log('pushing notification to %j', req.params.id);
   res.send(200, 'OK');
 });
});
```

To select a list of devices by query, use the `PushModel.notifyByQuery()`, for example:

```
PushModel.notifyByQuery({userId: {inq: selectedUserIds}}, note, function(err) {
   console.log('pushing notification to %j', selectedUserIds);
 });
```

Schedule the push notification request

This feature is not yet available. When you are ready to deploy your app, contact StrongLoop for more information.

Error handling

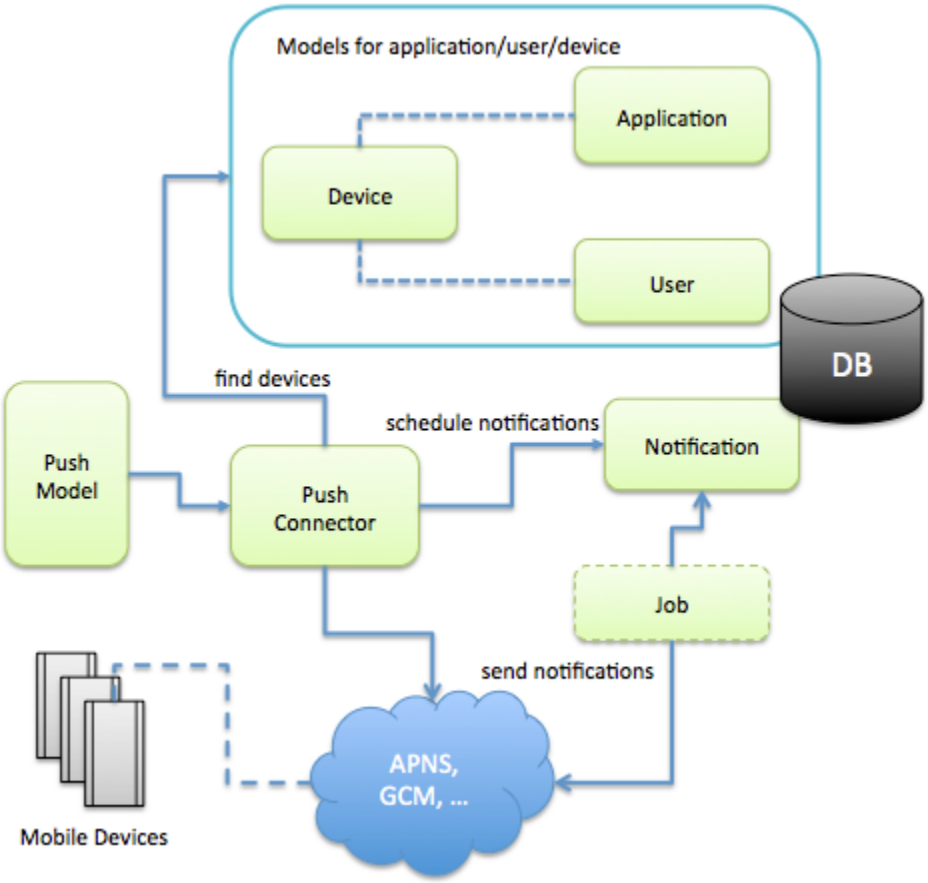
LoopBack has two mechanisms for reporting push notification errors:

- Most configuration-related errors are reported to the callback argument of notification functions. These errors should be reported back to the caller (HTTP client) as can be seen in the `notifyById()` code example above.
- Transport-related errors are reported via "error" events on the push connector. The application should listen for these events and report errors in the same way as other errors are reported (typically via console.error, bunyan, and so forth.).

```
PushModel.on('error', function(err) {
  console.error('Push Notification error: ', err.stack);
});
```

Architecture

The following diagram illustrates the LoopBack push notification system architecture.



Push notifications for iOS apps

For a complete working example iOS app, see LoopBack push notification iOS sample app.

Overview

This article provides information on creating iOS apps that can get push notifications from a LoopBack application. See Push notifications for information on creating the corresponding LoopBack server application.

The basic steps to set up push notifications for iOS clients are:

1. Provision an application with Apple and configure it to enable push notifications.
2. Provide a hook to receive the device token when the application launches and register it with the LoopBack server using the `LBInstallation` class.
3. Provide code to receive notifications, under three different application modes: foreground, background, and offline.
4. Process notifications.

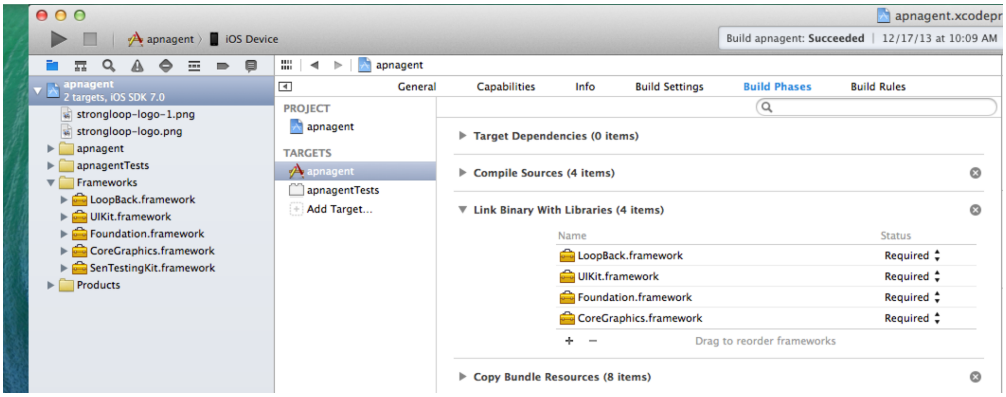
For general information on the Apple push notifications, see Apple iOS Local and Push Notification Programming Guide. For additional useful information, see Delivering iOS Push Notifications with Node.js.

Configure APN push settings in your server application

Please see Register a mobile application.

Add LoopBack iOS SDK as a framework

Open your XCode project, select targets, under build phases unfold **Link Binary with Libraries**, and click on '+' to add LoopBack framework.



The LoopBack iOS SDK provides two classes to simplify push notification programming:

- LBInstallation - enables the iOS application to register mobile devices with LoopBack.
- LBPushNotification - provides a set of helper methods to handle common tasks for push notifications.

Initialize LBRESTAdapter

The following code instantiates the shared `LBRESTAdapter`. In most circumstances, you do this only once; putting the reference in a singleton is recommended for the sake of simplicity. However, some applications will need to talk to more than one server; in this case, create as many adapters as you need.

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.settings = [self loadSettings];
    self.adapter = [LBRESTAdapter adapterWithURL:[NSURL
URLWithString:self.settings[@"RootPath"]]];

    // Reference to Push notifs List VC
    self.pnListVC = (NotificationListVC *)[[(UINavigationController
*)self.window.rootViewController viewControllers]
                                           objectAtIndex:0];

    LBPushNotification* notification = [LBPushNotification application:application
                                         didFinishLaunchingWithOptions:launchOptions];

    // Handle APN on Terminated state, app launched because of APN
    if (notification) {
        NSLog(@"Payload from notification: %@", notification.userInfo);
        [self.pnListVC addPushNotification:notification];
    }

    return YES;
}
```

Register the device

```
- (void)application:(UIApplication*)application
didRegisterForRemoteNotificationsWithDeviceToken:(NSData*)deviceToken
{
    __unsafe_unretained typeof(self) weakSelf = self;

    // Register the device token with the LoopBack push notification service
    [LBPushNotification application:application
didRegisterForRemoteNotificationsWithDeviceToken:deviceToken
                            adapter:self.adapter
                             userId:@"anonymous"
                      subscriptions:@[@"all"]
                            success:^(id model) {
                                LBInstallation *device = (LBInstallation *)model;
                                weakSelf.registrationId = device._id;
                            }
                            failure:^(NSError *err) {
                                NSLog(@"Failed to register device, error: %@", err);
                            }
     ];
...
}

- (void)application:(UIApplication*)application
didFailToRegisterForRemoteNotificationsWithError:(NSError*)error {
    // Handle errors if it fails to receive the device token
        [LBPushNotification application:application
didFailToRegisterForRemoteNotificationsWithError:error];
}
```

Handle received notifications

```
- (void)application:(UIApplication *)application
didReceiveRemoteNotification:(NSDictionary *)userInfo {
    // Receive push notifications
    LBPushNotification* notification = [LBPushNotification application:application
                                        didReceiveRemoteNotification:userInfo];
    [self.pnListVC addPushNotification:notification];
}
```

Push notifications for Android apps

For a complete working example Android app, see LoopBack push notification Android sample app..

Overview

This article provides information on creating Android apps that can get push notifications from a LoopBack application. See Push notifications for information on creating the corresponding LoopBack server application.

To enable an Android app to receive LoopBack push notifications:

1. Setup your app to use Google Play Services.
2. On app startup, register with GCM servers to obtain a device registration ID (device token) and register the device with the LoopBack server application.
3. Configure your LoopBack application to receive incoming messages from GCM.
4. Process the notifications received.

Prerequisites

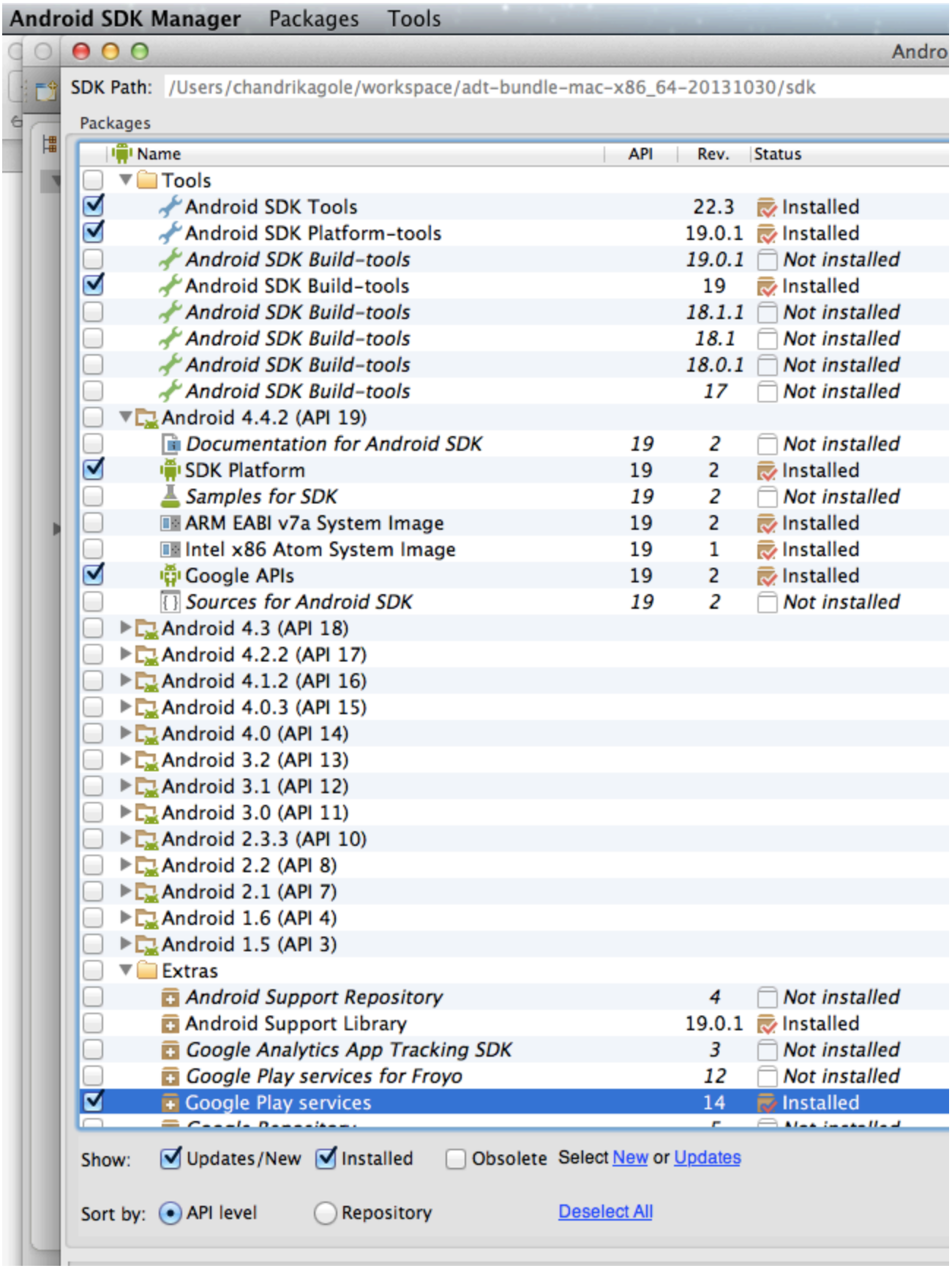
Before you start developing your application make sure you've performed all the prerequisite steps outlined in this section.

- **Download the LoopBack Android SDK**
- **Install Eclipse development tools (ADT)**

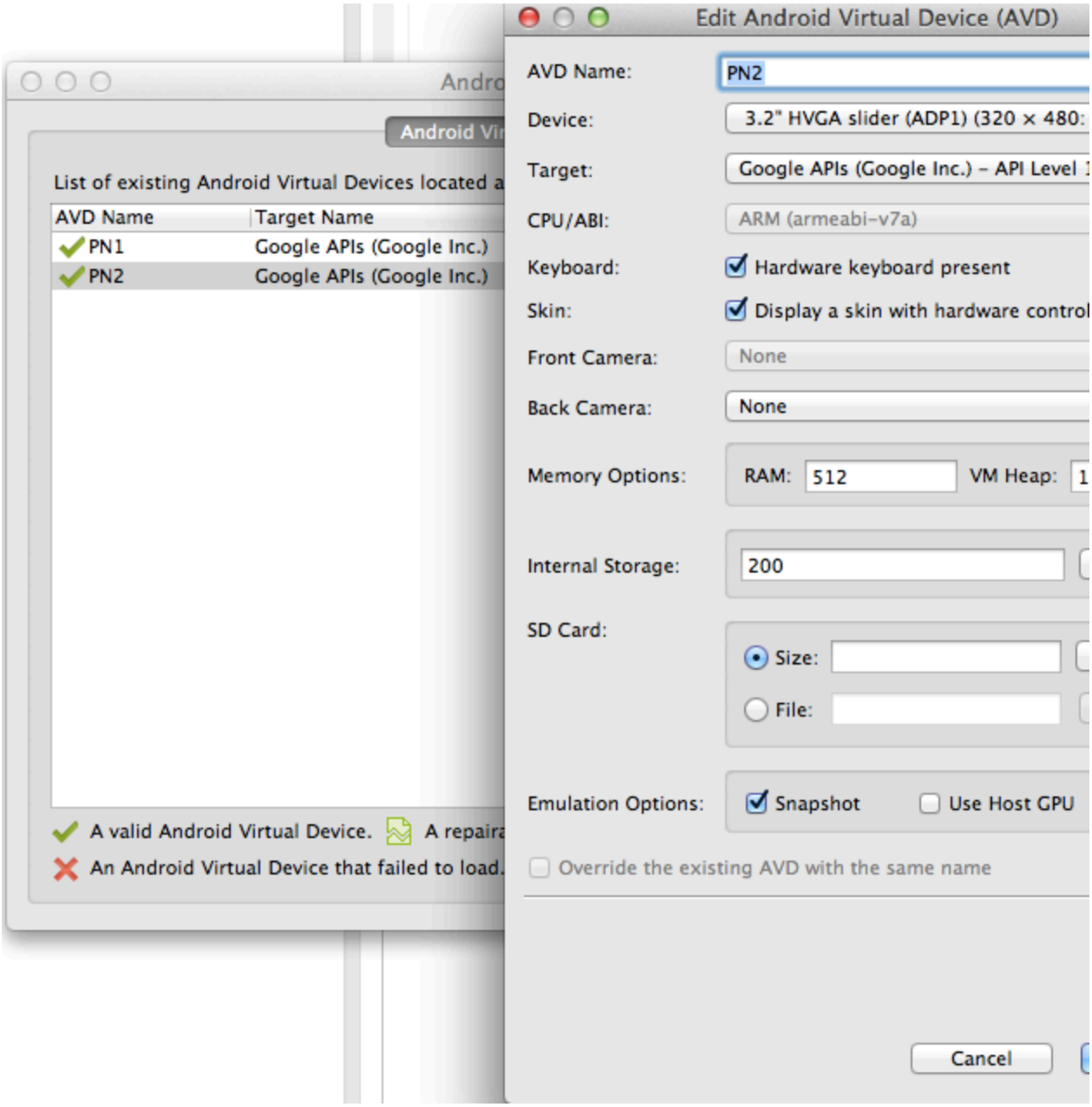
Configure Android Development Tools

Now configure Eclipse ADT as follows:

1. Open Eclipse from the downloaded ADT bundle.
2. In ADT, choose **Window > Android SDK Manager**.
3. Install the following if they are not already installed:
    - Tools:
        - Android SDK Platform-tools 18 or newer
        - Android SDK Build-tools 18 or newer
    - Android 4.3 (API 18):
        - SDK Platform.
        - Google APIs
    - Extras:
        - Google Play Services
        - Intel x86 Emulator Accelerator (HAXM)



4. Before you start, make sure you have set up at least one Android virtual device: Choose **Window > Android Virtual Device Manager** .
5. Configure the target virtual device as shown in the screenshot below. See AVD Manager for more information.



⚠ If you are using the virtual device suggested above, you must also install the ARM EABI v7a System Image SDK.

Get your Google Cloud Messaging credentials

To send push notifications to your Android app, you need to setup a Google API project and enable the Google Cloud Messaging (GCM) service.
Open the Android Developer's Guide

Follow the instructions to get your GCM credentials:

1. Follow steps to create a Google API project and enable the GCM service.
2. Create an Android API key
    a. In the sidebar on the left, select **APIs & auth > Credentials**.
    b. Click **Create new key**.
    c. Select **Android key**.
    d. Enter the SHA-1 fingerprint followed by the package name, for example

    45:B5:E4:6F:36:AD:0A:98:94:B4:02:66:2B:12:17:F2:56:26:A0:E0;com.example
    NOTE: Leave the package name as "com.example" for the time being.

3. You also have to create a new server API key that will be used by the LoopBack server:

a. Click **Create new key**.
b. Select **Server key**.
c. Leave the list of allowed IP addresses empty for now.
d. Click **Create**.
e. Copy down the API key. Later you will use this when you configure the LoopBack server application.

Install and run LoopBack Push Notification app

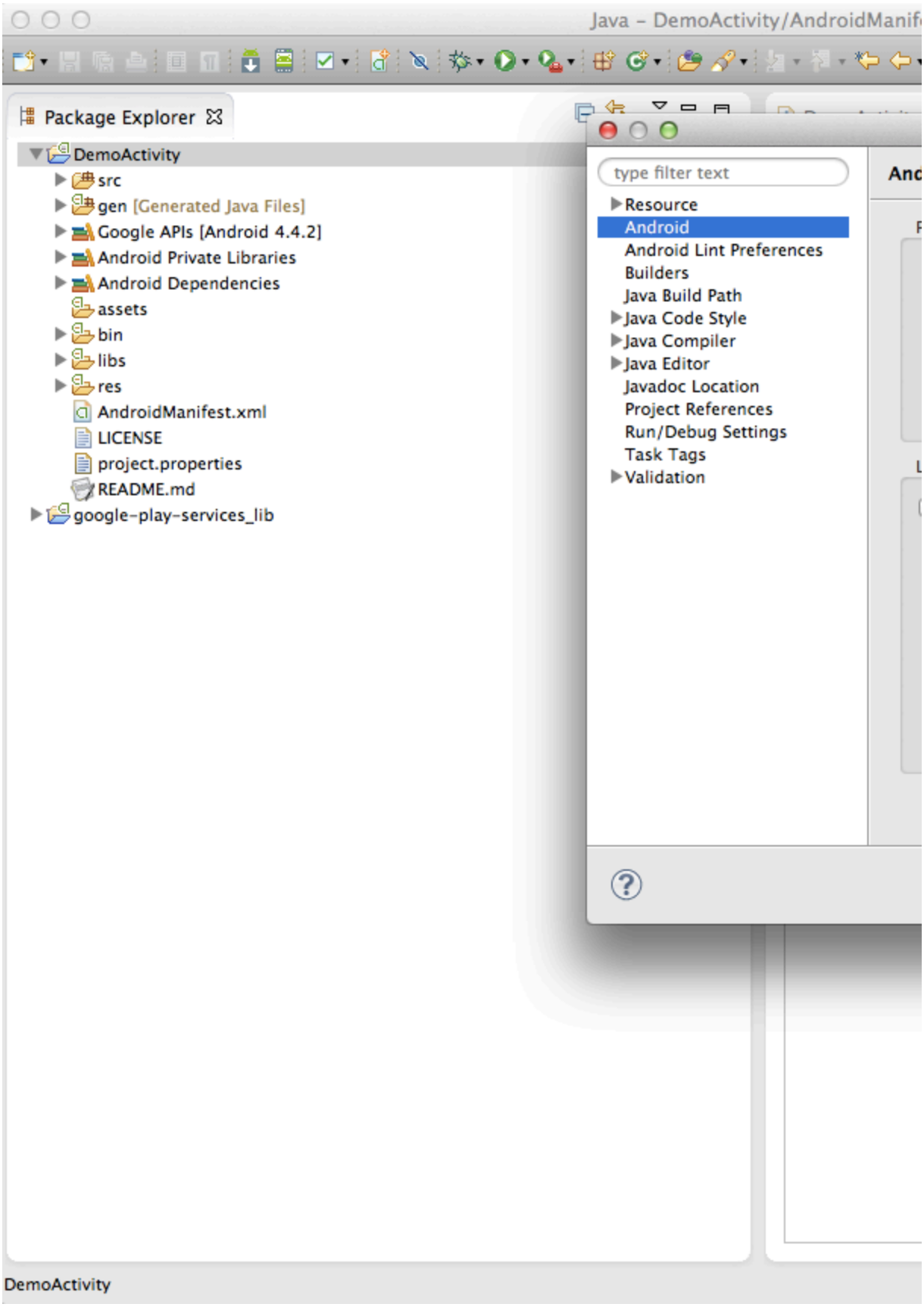
If you want to use the sample Android client app, download the Push Notification Example Android app . Then follow these steps to run the app:

1. Open ADT Eclipse.
2. Import the push notification application to your workspace:
    a. Choose **File > Import**.
    b. Choose **Android > Existing Android Code into Workspace**.
    c. Click **Next**.
    d. Browse to the example Android app you just downloaded.
    e. Click **Finish**.

    > ⚠ ADT does not take long to import the guide app. Don't be misguided by the progress bar at the bottom of the IDE window: it indicates memory use, not loading status.

3. Import Google Play Services library project into your workspace. The project is located inside the directory where you have installed the Android SDK.
    a. Choose **File > Import**.
    b. Choose **Android > Existing Android Code into Workspace**.
    c. Click **Next**.
    d. Browse to the `<android-sdk>/extras/google/google_play_services/libproject/google-play-services_lib` directory.
    e. Check **Copy projects into workspace**
    f. Click **Finish**.

    See Google Play Services SDK for more details.
4. Add the imported google-play-services_lib as an Android build dependency of the push notification application.
    a. In the Package Explorer frame in Eclipse, select the push notification application.
    b. Choose **File > Properties**.
    c. Select **Android**.
    d. In the Library frame, click on **Add...** and select `google-play-services_lib`.
    e. Also under Project Build Target, set the target as Google APIs.



5. Edit `src/com/google/android/gcm/demo/app/DemoActivity.java`.
    - Set SENDER_ID to the project number from the Google Developers Console you created earlier in Get your Google Cloud Messaging credentials.
6. Go back to the https://cloud.google.com/console/project and edit the Android Key to reflect your unique application ID. Set the value of **A**

ndroid applications to something like this:

Android applications	XX:XX:XX:XX:XX:XX:XX:XX:XX:X LOOPBACK_APP_ID X:XX:XX:XX:XX:XX:XX:XX:XX;com.google.android.gcm.demo.app.DemoApplication

7. Run the LoopBack server application you set up earlier. If you didn't set the appName in the server application's `config.js` earlier, do it now.
   Set it to "**com.google.android.gcm.demo.app.DemoActivity**".
8. Click the green **Run** button in the toolbar to run the application. Run it as an Android application. You will be prompted to select the target on which to run the application. Select the AVD you created earlier.

It may take several minutes to launch your application and the Android virtual device the first time.

Due to a known issue with Google Play Services, you must download and import an older version of Google Play services.

1. Download https://dl-ssl.google.com/android/repository/google_play_services_3225130_r10.zip
2. Extract the zip file.
3. In Eclipse ADT, right-click on your project and choose **Import**...
4. Choose **Existing Android Code into Workspace** then click Next.
5. Click **Browse**...
6. Browse to the `google-play-services/libproject/google-play-services_lib/` directory created when you extracted the zip file and select it in the dialog box.
7. Click **Finish**.

You must also update `AndroidManifest.xml` as follows:

1. In Eclipse ADT, browse to DemoActivity/AndroidManifest.xml.
2. Change the line
   `<meta-data android:name="com.google.android.gms.version" android:value="@integer/google_play_services_version"/>`
   to
   `<meta-data android:name="com.google.android.gms.version" android:value="4030500”/>`
3. Save the file.

Configure GCM push settings in your server application

Add the following key and value to the push settings of your application:

```
{
  gcm: {
    serverApiKey: "server-api-key"
  }
}
```

Replace `server-api-key` with the API key you obtained in Get your Google Cloud Messaging credentials.

Prepare your own Android project

Follow the instructions in Android SDK documentation to add LoopBack Android SDK to your Android project.

Follow the instructions in Google's Implementing GCM Client guide for setting up Google Play Services in your project.

To use push notifications, you must install a compatible version of the Google APIs platform. To test your app on the emulator, expand the directory for Android 4.2.2 (API 17) or a higher version, select **Google APIs**, and install it. Then create a new AVD with Google APIs as the platform target. You must install the package from the SDK manager. For more information, see Set Up Google Play Services.

Check for Google Play Services APK

Applications that rely on the Google Play Services SDK should always check the device for a compatible Google Play services APK before using

Google Cloud Messaging.

For example, the following code checks the device for Google Play Services APK by calling `checkPlayServices()` if this method returns true, it proceeds with GCM registration. The `checkPlayServices()` method checks whether the device has the Google Play Services APK. If it doesn't, it displays a dialog that allows users to download the APK from the Google Play Store or enables it in the device's system settings.

```
@Override
public void onCreate(final Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    if (checkPlayServices()) {
        updateRegistration();
    } else {
        Log.i(TAG, "No valid Google Play Services APK found.");
    }
}
private boolean checkPlayServices() {
    int resultCode = GooglePlayServicesUtil.isGooglePlayServicesAvailable(this);
    if (resultCode != ConnectionResult.SUCCESS) {
        if (GooglePlayServicesUtil.isUserRecoverableError(resultCode)) {
            GooglePlayServicesUtil.getErrorDialog(resultCode, this,
                    PLAY_SERVICES_RESOLUTION_REQUEST).show();
        } else {
            Log.i(TAG, "This device is not supported.");
            finish();
        }
        return false;
    }
    return true;
}
```

Create LocalInstallation

Once you have ensured the device provides Google Play Services, the app can register with GCM and LoopBack (for example, by calling a method such as `updateRegistration()` as shown below). Rather than register with GCM every time the app starts, simply store and retrieve the registration ID (device token). The `LocalInstallation` class in the LoopBack SDK handles these details for you.

The example `updateRegistration()` method does the following:

- Lines 3 - 4: get a reference to the shared `RestAdapter` instance.
- Line 5: Create an instance of `LocalInstallation`.
- Line 13: Subscribe to topics.
- Lines 15-19: Check if there is a valid GCM registration ID. If so, then save the installation to the server; if not, get one from GCM and then save the installation.

```
private void updateRegistration() {

    final DemoApplication app = (DemoApplication) getApplication();
    final RestAdapter adapter = app.getLoopBackAdapter();
    final LocalInstallation installation = new LocalInstallation(context, adapter);

    // Substitute the real ID of the LoopBack application as created by the server
    installation.setAppId("loopback-app-id");

    // Substitute a real ID of the user logged in to the application
    installation.setUserId("loopback-android");

    installation.setSubscriptions(new String[] { "all" });

    if (installation.getDeviceToken() != null) {
        saveInstallation(installation);
    } else {
        registerInBackground(installation);
    }
}
```

Register with GCM if needed

In the following code, the application obtains a new registration ID from GCM. Because the `register()` method is blocking, you must call it on a background thread.

```
private void registerInBackground(final LocalInstallation installation) {
    new AsyncTask<Void, Void, Exception>() {
        @Override
        protected Exception doInBackground(final Void... params) {
            try {
                GoogleCloudMessaging gcm = GoogleCloudMessaging.getInstance(this);
                // substitute 12345 with the real Google API Project number
    final String regid = gcm.register("12345");
                installation.setDeviceToken(regid);
                return null;
            } catch (final IOException ex) {
                return ex;
                // If there is an error, don't just keep trying to
                // register.
                // Require the user to click a button again, or perform
                // exponential back-off.
            }
        }
        @Override
        protected void onPostExecute(final Exception error) {
            if (err != null) {
                Log.e(TAG, "GCM Registration failed.", error);
            } else {
                saveInstallation(installation);
            }
        }
    }.execute(null, null, null);
}
```

Register with LoopBack server

Once you have all Installation properties set, you can register with the LoopBack server. The first run of the application should create a new Installation record, subsequent runs should update this existing record. The LoopBack Android SDK handles the details. Your code just needs to call `save()`.

```
void saveInstallation(final LocalInstallation installation) {
    installation.save(new Model.Callback() {
        @Override
        public void onSuccess() {
            // Installation was saved.
            // You can access the id assigned by the server via
            //   installation.getId();
        }
        @Override
        public void onError(final Throwable t) {
            Log.e(TAG, "Cannot save Installation", t);
        }
    });
}
```

Handle received notifications

Android apps handle incoming notifications in the standard way; LoopBack does not require any special changes. For more information, see the section "Receive a message" of Google's Implementing GCM Client guide.

Troubleshooting

When running your app in the Eclipse device emulator, you may encounter the following error:

```
Google Play services, which some of your applications rely on, is not supported by your device. Please contact the manufacturer for assistance.
```

To resolve this, install a compatible version of the Google APIs platform. See Prepare your Android project for more information.

Push notification API

Installation API

Module: loopback-component-push

Installation

Installation Model connects a mobile application to the device, the user and other information for the server side to locate devices using application id/version, user id, device type, and subscriptions.

Installation.findByApp(deviceType, appId, [appVersion], cb)

Find installations by application id/version

Arguments

Name	Type	Description
deviceType	`String`	The device type
appId	`String`	The application id
[appVersion]	`String`	The application version

cb	`Function`	The callback function

cb

Name	Type	Description
err	`Error or String`	The error object
installations	`Installation[]`	The selected installations

Installation.findByUser(userId, deviceType, cb, cb)

Find installations by user id

Arguments

Name	Type	Description
userId	`String`	The user id
deviceType	`String`	The device type
cb	`Function`	The callback function
cb	`Function`	The callback function

cb

Name	Type	Description
err	`Error or String`	The error object
installations	`Installation[]`	The selected installations

Installation.findBySubscriptions(subscriptions, deviceType, cb)

Find installations by subscriptions

Arguments

Name	Type	Description
subscriptions	`String or String[]`	A list of subscriptions
deviceType	`String`	The device type
cb	`Function`	The callback function

cb

Name	Type	Description
err	`Error or String`	The error object
installations	`Installation[]`	The selected installations

Notification API

- Notification

Module: loopback-component-push

Notification

Notification Model

See the official documentation for more details on provider-specific properties.

Android - GCM

iOS - APN

PushManager API

Module: loopback-component-push

PushManager(settings)

@class

Arguments

Name	Type	Description
settings	`Object`	The push settings

Returns

Name	Type	Description
result	`PushManager`	

PushManager.providers

Registry of providers Key: device type, e.g. 'ios' or 'android' Value: constructor function, e.g providers.ApnsProvider

pushManager.configureProvider(deviceType, pushSettings)

Configure push notification for a given device type. Return null when no provider is registered for the device type.

Arguments

Name	Type	Description
deviceType	`String`	The device type
pushSettings	`Object`	The push settings

Returns

Name	Type	Description
result	`Any`	

pushManager.configureApplication(appId)

Lookup or set up push notification service for the given appId

Arguments

Name	Type	Description
appId	`String`	The application id

Returns

Name	Type	Description
result	`Any`	

pushManager.notifyById(installationId, notification, cb)

Push a notification to the device with the given registration id.

Arguments

Name	Type	Description
installationId	`Object`	Registration id created by call to Installation.create().
notification	`Notification`	The notification to send.
cb	`function(Error=)`	

pushManager.notifyByQuery(installationQuery, notification, cb)

Push a notification to all installations matching the given query.

Arguments

Name	Type	Description
installationQuery	`Object`	Installation query, e.g. `{ appId: 'iCarsAppId', userId: 'jane.smith.id' }`
notification	`Notification`	The notification to send.
cb	`function(Error=)`	

pushManager.notify(installation, notification, cb)

Push a notification to the given installation. This is a low-level function used by the other higher-level APIs like `notifyById` and `notifyByQuery`.

Arguments

Name	Type	Description
installation	`Installation`	Installation instance - the recipient.
notification	`Notification`	The notification to send.
cb	`function(Error=)`	

pushManager.notifyMany(application, type, device, notification, cb)

Push notification to installations for given devices tokens, device type and app.

Arguments

Name	Type	Description
application	`appId`	id
type	`deviceType`	of device (android, ios)
device	`deviceTokens`	tokens of recipients.
notification	`Notification`	The notification to send.
cb	`function(Error=)`	

Tutorial: Push notifications

This four-part tutorial explains how to create a mobile application on Amazon EC2 that can send push notification to iOS and Android client applications:

- Part one shows how to use the `slc` command-line tool to create a LoopBack application running on Amazon EC2 to send push notifications.
- Part two explains how to setup and create an Android app to receive push notifications.
- Part three explains how to setup and create an iOS app to receive push notifications.
- Part four explains how to use LoopBack's swagger REST api and send/receive push notifications on your Android and iOS devices.

You can use the StrongLoop AMI as a starting point for building a backend node application using the `slc` command line tool. Using this app, mobile developers can dynamically manage ‘mobile object’ schemas directly from the LoopBack SDK as they are building their mobile app. The pre-built StrongLoop AMI makes it easy for Node developers who’ve chosen Amazon as their infrastructure provider to quickly get up and running.

Now dive in and start on part one!

Tutorial: push notifications - LoopBack app

This is part one of a four-part tutorial on setting up a mobile application on Amazon EC2 that can send push notification to iOS and Android client applications

- Part one (this article) shows how to use the `slc` command-line tool to create a LoopBack application running on Amazon EC2 to send push notifications.

Overview

Push notifications enable server applications (known as *providers* in push parlance) to send information to mobile apps even when the app isn't in use. The device displays the information using a "badge," alert, or pop up message. A push notification uses the service provided by the device's operating system:

- **iOS** - Apple Push Notification service (APNS)
- **Android** - Google Cloud Messaging (GCM)

Prerequisites

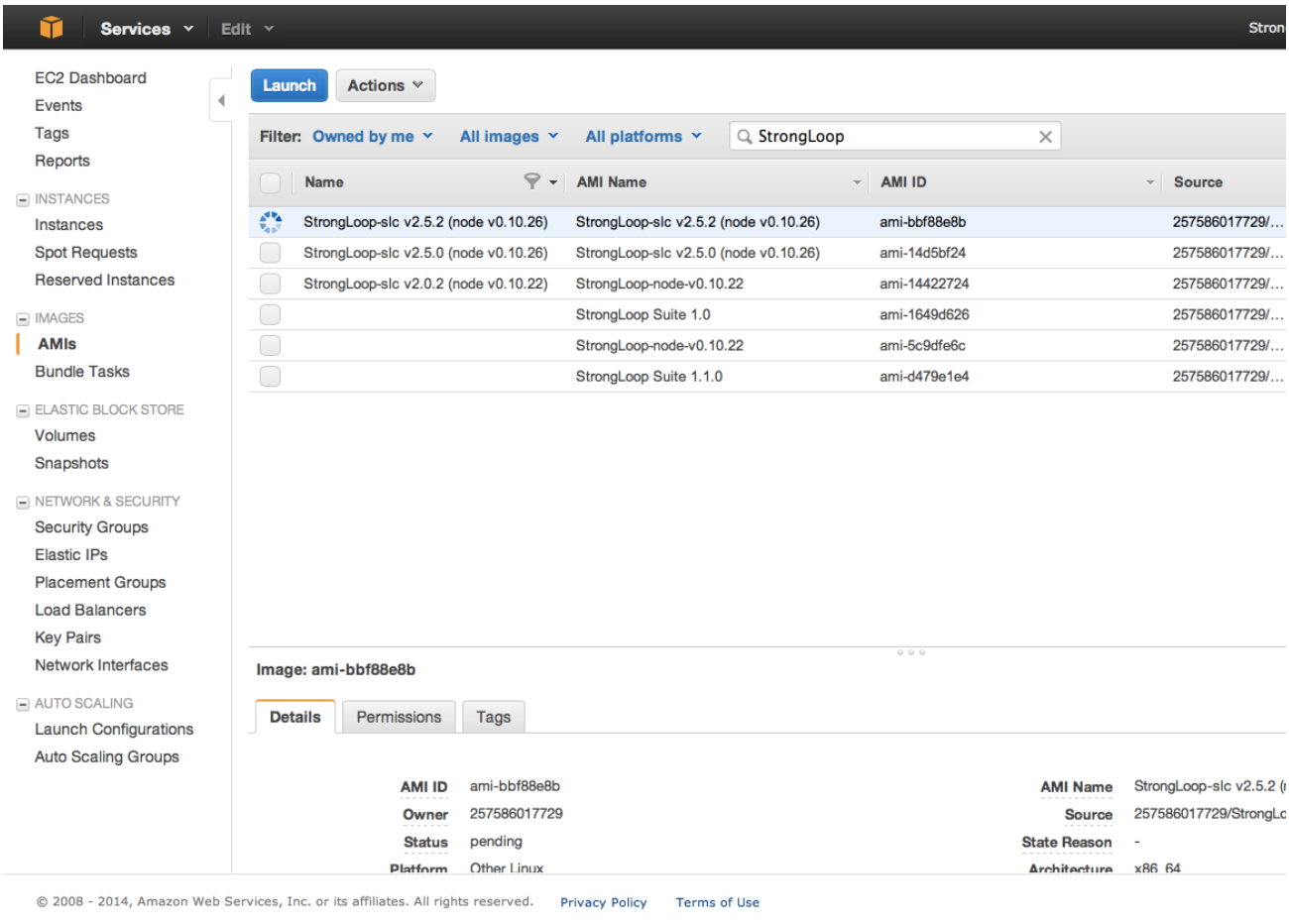
Before starting this tutorial:

- Make sure you have an Amazon AWS account.
- To set up push notifications for an iOS app:
    - **Download the LoopBack iOS SDK**
    - **Install Xcode**
- To set up push notifications for an Android app:
    - **Download the LoopBack Android SDK**
    - **Install Eclipse development tools (ADT)**

Set up mobile backend server on Amazon

Launch the instance

1. To follow along you will need to have an Amazon AWS account.
2. To find the StrongLoop AMI, simply log into the AWS Console and select "EC2." Browse images by selecting "AMIs" under the "Images" drop down. Make sure the filtering shows "Public Images" , "All Images" and "All Platforms". From here you can simply search "StrongLoop" and select the latest version. Current - StrongLoop-slc v2.5.2 (node v0.10.26)



3. Launch a new instance from the console using this AMI. Once the instance is up and running, you can remote ssh into your newly created server instance using the same ec2-keypair and the machine instance ID.

```
$ssh -i ec2-keypair ec2-user@ec2-54-222-22-59.us-west-1.compute.amazonaws.com
```

Create the application

> ⚠ StrongLoop Controller (slc) and MongoDB are pre-installed in the StrongLoop AMI. To run MongoDB, use `~/mongodb/bin/mongod &`. You may need to use `sudo`.

1. Use the slc loopback command to create a LoopBack application:

```
$ slc loopback
[?] Enter a directory name where to create the project: push
[?] What's the name of your application? push
```

2. Add the loopback-component-push and loopback-connector-mongodb modules as dependencies in `package.json`:

```
"dependencies": {
    "loopback": "~1.7.0",
    "loopback-component-push": "~1.2.0",
    "loopback-connector-mongodb": "~1.2.0"
  },
```

3. Install dependencies:

```
npm install
```

4. Add the mongo connection string in `/server/datasources.json`:

```
"db": {
    "defaultForType": "db",
    "connector": "mongodb",
    "url": "mongodb://localhost/demo"
  },
```

Try it with an example
To try out an example, you can replace the server.js file in your application with this. You will also need the model-config.js file to save your configurations. Alternatively, to enable push notifications for your own application using the loopback-component-push module, follow the steps below (5-8)

5. Create a push model
    a. To send push notifications, you must create a push model. The code below illustrates how to do this with a database as the data source. The database is used to load and store the corresponding application/user/installation models.

```
var loopback = require('loopback');
var app = loopback();
var db = require('./data-sources/db');
// Load & configure loopback-component-push
var PushModel = require('loopback-component-push')(app, { dataSource: db });
var Application = PushModel.Application;
var Installation = PushModel.Installation;
var Notification = PushModel.Notification;
```

6. Register a mobile(client) application
    a. The mobile application needs to register with LoopBack so it can have an identity for the application and corresponding settings for push services. Use the Application model's `register()` function for sign-up.

For information on getting API keys, see:

- Get your Google Cloud Messaging credentials for Android.
- Set up iOS clients for iOS.

```
Application.register('put your developer id here',
    'put your unique application name here',
    {
      description: 'LoopBack Push Notification Demo Application',
      pushSettings: {
        apns: {
          certData: readCredentialsFile('apns_cert_dev.pem'),
          keyData: readCredentialsFile('apns_key_dev.pem'),

          pushOptions: {
          },
          feedbackOptions: {
            batchFeedback: true,
            interval: 300
          }
        },
        gcm: {
          serverApiKey: 'your GCM server API Key'
        }
      }
    },
    function(err, app) {
      if (err) return cb(err);
      return cb(null, app);
    }
  );

function readCredentialsFile(name) {
 return fs.readFileSync(
   path.resolve(__dirname, 'credentials', name),
   'UTF-8'
 );
}
```

7. Register a mobile device
    a. The mobile device also needs to register itself with the backend using the Installation model and APIs. To register a device from the server side, call the`Installation.create()` function, as shown in the following example:

```
Installation.create({
    appId: 'MyLoopBackApp',
    userId: 'raymond',
    deviceToken:
'756244503c9f95b49d7ff82120dc193ca1e3a7cb56f60c2ef2a19241e8f33305',
    deviceType: 'ios',
    created: new Date(),
    modified: new Date(),
    status: 'Active'
}, function (err, result) {
    console.log('Registration record is created: ', result);
});
```

Most likely, the mobile application registers the device with LoopBack using REST APIs or SDKs from the client side, for example

```
POST http://localhost:3010/api/installations
    {
        "appId": "MyLoopBackApp",
        "userId": "raymond",
        "deviceToken":
"756244503c9f95b49d7ff82120dc193ca1e3a7cb56f60c2ef2a19241e8f33305",
        "deviceType": "ios"
    }
```

8. Send push notifications
    a. Send out the push notification

LoopBack provides two Node.js methods to select devices and send notifications to them:

- `notifyById()`: Select a device by registration ID and send a notification to it.
- `notifyByQuery()`: Get a list of devices using a query (same as the **where** property for `Installation.find()`) and send a notification to all of them.

For example, the code below creates a custom endpoint to send out a dummy notification for the selected device:

```
var badge = 1;app.post('/notify/:id', function (req, res, next) {
 var note = new Notification({
   expirationInterval: 3600, // Expires 1 hour from now.
   badge: badge++,
   sound: 'ping.aiff',
   alert: '\uD83D\uDCE7 \u2709 ' + 'Hello',
   messageFrom: 'Ray'
 });

 PushModel.notifyById(req.params.id, note, function(err) {
   if (err) {
     // let the default error handling middleware
     // report the error in an appropriate way
     return next(err);
   }
   console.log('pushing notification to %j', req.params.id);
   res.send(200, 'OK');
 });
});
```

To select a list of devices by query, use the `PushModel.notifyByQuery()`, for example:

```
PushModel.notifyByQuery({userId: {inq: selectedUserIds}}, note,
function(err) {   console.log('pushing notification to %j',
selectedUserIds);
 });
```

Next steps

- To setup and create an Android app to receive push notifications go to Part two
- To setup and create an iOS app to receive push notifications go to Part three
- To use LoopBack's swagger REST API and send/receive push notifications on your Android and iOS devices got to Part four

Tutorial: push notifications - Android client

This is the second of a four-part tutorial on setting up a mobile backend as a service on Amazon and setting up iOS and Android client applications to enable push notification. If you want to setup Push Notifications for an iOS app refer to Part 3.

Overview

This article provides information on creating Android apps that can get push notifications from a LoopBack application.

To enable an Android app to receive LoopBack push notifications:

1. Setup your android client app to use Google Play Services.
2. On app startup, register with GCM servers to obtain a device registration ID (device token) and register the device with the LoopBack server application.
3. Configure your LoopBack application to receive incoming messages from GCM.
4. Process the notifications received.

Prerequisites

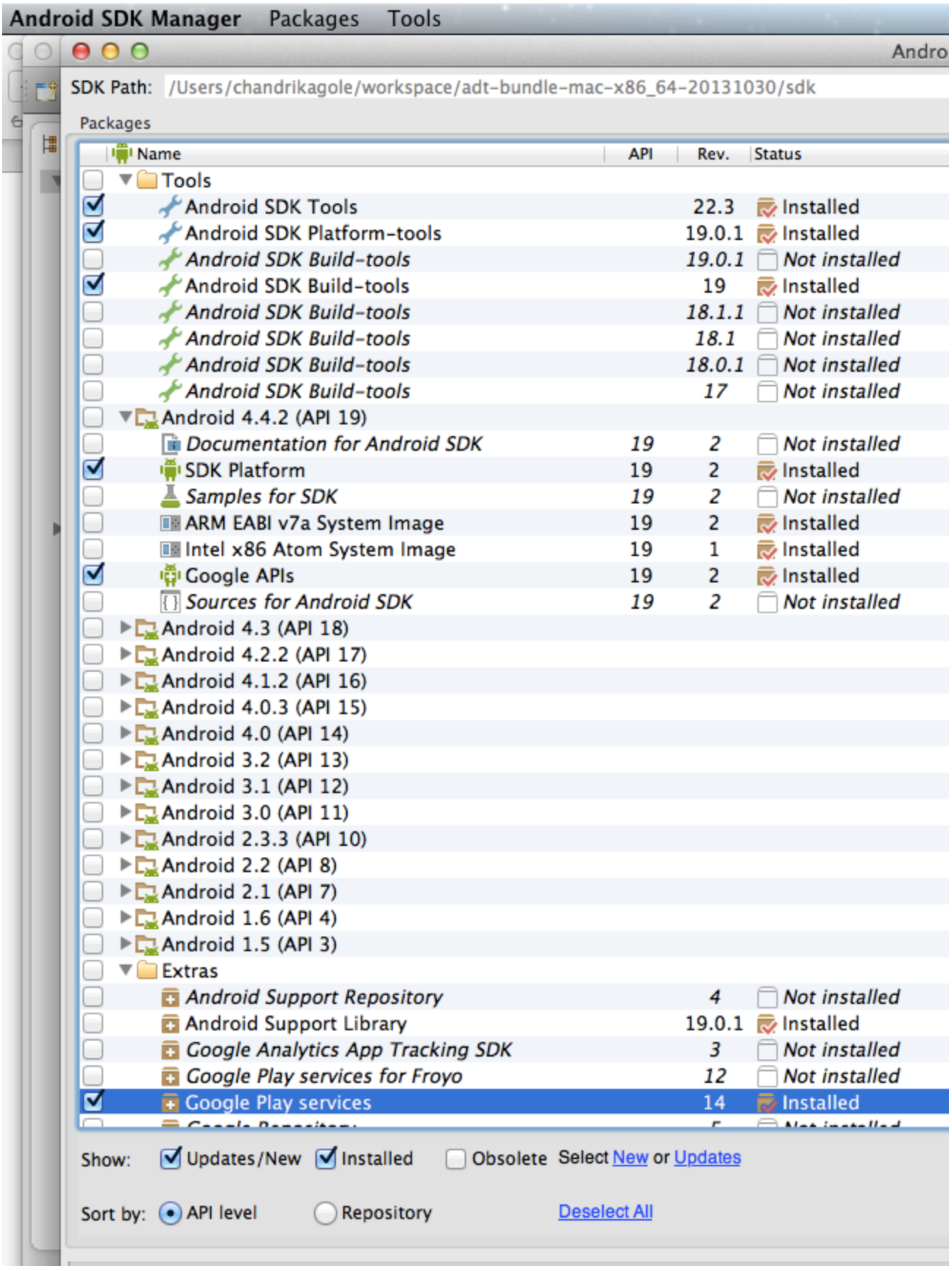
Before you start developing your application make sure you've performed all the prerequisite steps outlined in this section.

- **Download the LoopBack Android SDK**
- **Install Eclipse development tools (ADT)**

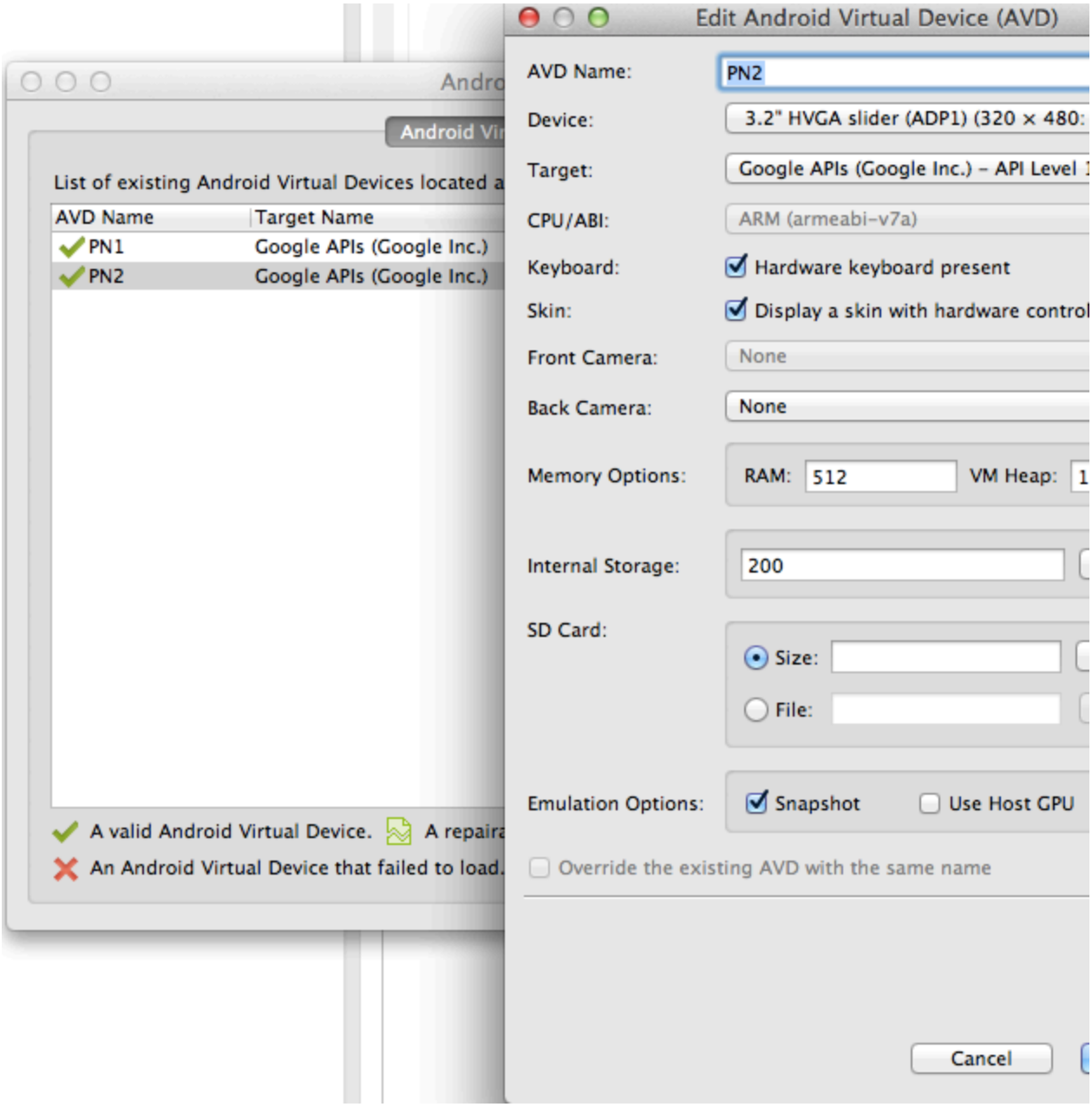
Configure Android Development Tools

Now configure Eclipse ADT as follows:

1. Open Eclipse from the downloaded ADT bundle.
2. In ADT, choose **Window > Android SDK Manager**.
3. Install the following if they are not already installed:
    - Tools:
        - Android SDK Platform-tools 18 or newer
        - Android SDK Build-tools 18 or newer
    - Android 4.3 (API 18):
        - SDK Platform.
        - Google APIs
    - Extras:
        - Google Play Services
        - Intel x86 Emulator Accelerator (HAXM)



4. Before you start, make sure you have set up at least one Android virtual device: Choose **Window > Android Virtual Device Manager** .
5. Configure the target virtual device as shown in the screenshot below. See AVD Manager for more information.



⚠ If you are using the virtual device suggested above, you must also install the ARM EABI v7a System Image SDK.

Get your Google Cloud Messaging credentials

To send push notifications to your Android app, you need to setup a Google API project and enable the Google Cloud Messaging (GCM) service.

- Open the Android Developer's Guide

Follow the instructions to get your GCM credentials:

1. Follow steps to create a Google API project and enable the GCM service.
2. Create an Android API key
    a. In the sidebar on the left, select **APIs & auth > Credentials**.
    b. Click **Create new key**.
    c. Select **Android key**.
    d. Enter the SHA-1 fingerprint followed by the package name, for example

    45:B5:E4:6F:36:AD:0A:98:94:B4:02:66:2B:12:17:F2:56:26:A0:E0;com.example
    NOTE: Leave the package name as "com.example" for the time being.

3. You also have to create a new server API key that will be used by the LoopBack server:

a. Click **Create new key**.
b. Select **Server key**.
c. Leave the list of allowed IP addresses empty for now.
d. Click **Create**.
e. Copy down the API key. Later you will use this when you configure the LoopBack server application.

Configure GCM push settings in your server application

Add the following key and value to the push settings of your application in the config.js file -

```
{
  gcm: {
    serverApiKey: "server-api-key"
  }
}
```

Replace `server-api-key` with the API key you obtained in the section Get your Google Cloud Messaging credentials.

If you want to try a sample client application follow steps in "Install and run LoopBack Push Notification app" OR if you want to enable push notifications for your own android application using the LoopBack SDK follow steps in "Prepare your own Android project"

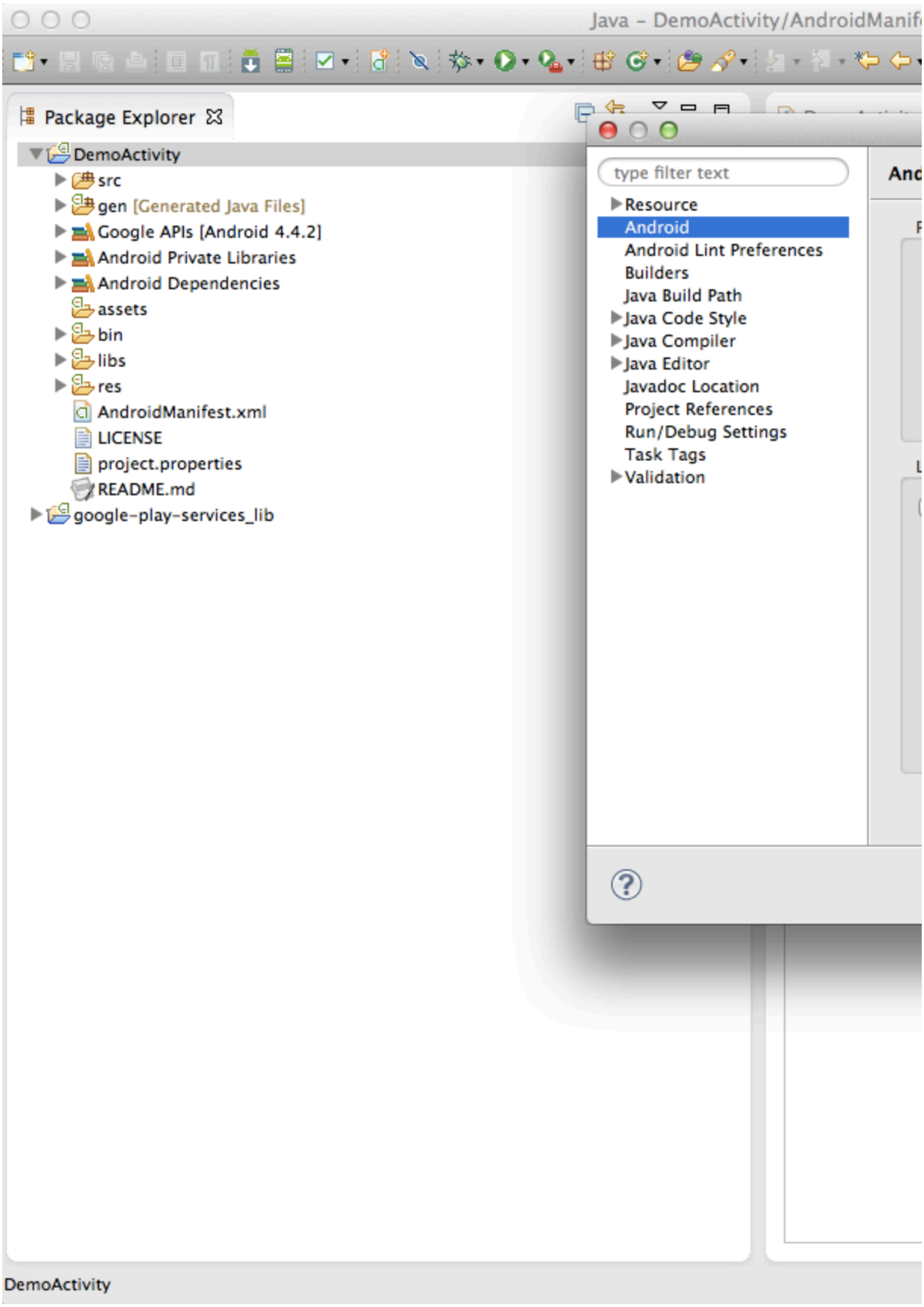
Install and run LoopBack Push Notification app

If you want to use the sample Android client app, download the Push Notification Example Android app . Then follow these steps to run the app:

1. Open ADT Eclipse.
2. Import the push notification application to your workspace:
   a. Choose **File > Import**.
   b. Choose **Android > Existing Android Code into Workspace**.
   c. Click **Next**.
   d. Browse to the example Android app you just downloaded.
   e. Click **Finish**.

   ADT does not take long to import the guide app. Don't be misguided by the progress bar at the bottom of the IDE window: it indicates memory use, not loading status.
3. Import Google Play Services library project into your workspace. The project is located inside the directory where you have installed the Android SDK.
   a. Choose **File > Import**.
   b. Choose **Android > Existing Android Code into Workspace**.
   c. Click **Next**.
   d. Browse to the `<android-sdk>/extras/google/google_play_services/libproject/google-play-services_lib` directory.
   e. Check **Copy projects into workspace**
   f. Click **Finish**.

   See Google Play Services SDK for more details.
4. Add the imported google-play-services_lib as an Android build dependency of the push notification application.
   a. In the Package Explorer frame in Eclipse, select the push notification application.
   b. Choose **File > Properties**.
   c. Select **Android**.
   d. In the Library frame, click on **Add...** and select `google-play-services_lib`.
   e. Also under Project Build Target, set the target as Google APIs.



5. Edit `src/com/google/android/gcm/demo/app/DemoActivity.java`.
    - Set SENDER_ID to the project number from the Google Developers Console you created earlier in Get your Google Cloud Messaging credentials.
6. Go back to the https://cloud.google.com/console/project and edit the Android Key to reflect your unique application ID. Set the value of **A**

ndroid applications to something like this:

Android applications	XX:XX:XX:XX:XX:XX:XX:XX:XX:X LOOPBACK_APP_ID X:XX:XX:XX:XX:XX:XX:XX:XX;com.google.android.gcm.demo.app.DemoApplication

7. Set the appName in the server application's `config.js` to "**com.google.android.gcm.demo.app.DemoActivity**".
8. Edit src/com/google/android/gcm/demo/app/DemoApplication.java
   - Set adaptor to our server ip. In my case it is

DemoApplication.java

```
adapter = new RestAdapter(
                    getApplicationContext(),

"http://ec2-54-184-36-164.us-west-2.compute.amazonaws.com:3000/api/");
        }
```

9. Click the green **Run** button in the toolbar to run the application. Run it as an Android application. You will be prompted to select the target on which to run the application. Select the AVD you created earlier.

It may take several minutes to launch your application and the Android virtual device the first time.

Due to a known issue with Google Play Services, you must download and import an older version of Google Play services.

1. Download https://dl-ssl.google.com/android/repository/google_play_services_3225130_r10.zip
2. Extract the zip file.
3. In Eclipse ADT, right-click on your project and choose **Import**...
4. Choose **Existing Android Code into Workspace** then click Next.
5. Click **Browse**...
6. Browse to the `google-play-services/libproject/google-play-services_lib/` directory created when you extracted the zip file and select it in the dialog box.
7. Click **Finish**.

You must also update `AndroidManifest.xml` as follows:

1. In Eclipse ADT, browse to DemoActivity/AndroidManifest.xml.
2. Change the line
   `<meta-data android:name="com.google.android.gms.version" android:value="@integer/google_play_services_version"/>`
   to
   `<meta-data android:name="com.google.android.gms.version" android:value="4030500”/>`
3. Save the file.

Prepare your own Android project

Follow the instructions in Android SDK documentation to add LoopBack Android SDK to your Android project.

Follow the instructions in Google's Implementing GCM Client guide for setting up Google Play Services in your project.

To use push notifications, you must install a compatible version of the Google APIs platform. To test your app on the emulator, expand the directory for Android 4.2.2 (API 17) or a higher version, select **Google APIs**, and install it. Then create a new AVD with Google APIs as the platform target. You must install the package from the SDK manager. For more information, see Set Up Google Play Services.

Check for Google Play Services APK

Applications that rely on the Google Play Services SDK should always check the device for a compatible Google Play services APK before using Google Cloud Messaging.

For example, the following code checks the device for Google Play Services APK by calling `checkPlayServices()` if this method returns true, it proceeds with GCM registration. The `checkPlayServices()` method checks whether the device has the Google Play Services APK. If it doesn't, it displays a dialog that allows users to download the APK from the Google Play Store or enables it in the device's system settings.

```
@Override
public void onCreate(final Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    if (checkPlayServices()) {
        updateRegistration();
    } else {
        Log.i(TAG, "No valid Google Play Services APK found.");
    }
}
private boolean checkPlayServices() {
    int resultCode = GooglePlayServicesUtil.isGooglePlayServicesAvailable(this);
    if (resultCode != ConnectionResult.SUCCESS) {
        if (GooglePlayServicesUtil.isUserRecoverableError(resultCode)) {
            GooglePlayServicesUtil.getErrorDialog(resultCode, this,
                    PLAY_SERVICES_RESOLUTION_REQUEST).show();
        } else {
            Log.i(TAG, "This device is not supported.");
            finish();
        }
        return false;
    }
    return true;
}
```

Create LocalInstallation

Once you have ensured the device provides Google Play Services, the app can register with GCM and LoopBack (for example, by calling a method such as `updateRegistration()` as shown below). Rather than register with GCM every time the app starts, simply store and retrieve the registration ID (device token). The `LocalInstallation` class in the LoopBack SDK handles these details for you.

The example `updateRegistration()` method does the following:

- Lines 3 - 4: get a reference to the shared `RestAdapter` instance.
- Line 5: Create an instance of `LocalInstallation`.
- Line 13: Subscribe to topics.
- Lines 15-19: Check if there is a valid GCM registration ID. If so, then save the installation to the server; if not, get one from GCM and then save the installation.

```
private void updateRegistration() {

    final DemoApplication app = (DemoApplication) getApplication();
    final RestAdapter adapter = app.getLoopBackAdapter();
    final LocalInstallation installation = new LocalInstallation(context, adapter);

    // Substitute the real ID of the LoopBack application as created by the server
    installation.setAppId("loopback-app-id");

    // Substitute a real ID of the user logged in to the application
    installation.setUserId("loopback-android");

    installation.setSubscriptions(new String[] { "all" });

    if (installation.getDeviceToken() != null) {
        saveInstallation(installation);
    } else {
        registerInBackground(installation);
    }
}
```

Register with GCM if needed

In the following code, the application obtains a new registration ID from GCM. Because the `register()` method is blocking, you must call it on a background thread.

```
private void registerInBackground(final LocalInstallation installation) {
    new AsyncTask<Void, Void, Exception>() {
        @Override
        protected Exception doInBackground(final Void... params) {
            try {
                GoogleCloudMessaging gcm = GoogleCloudMessaging.getInstance(this);
                // substitute 12345 with the real Google API Project number
    final String regid = gcm.register("12345");
                installation.setDeviceToken(regid);
                return null;
            } catch (final IOException ex) {
                return ex;
                // If there is an error, don't just keep trying to
                // register.
                // Require the user to click a button again, or perform
                // exponential back-off.
            }
        }
        @Override
        protected void onPostExecute(final Exception error) {
            if (err != null) {
                Log.e(TAG, "GCM Registration failed.", error);
            } else {
                saveInstallation(installation);
            }
        }
    }.execute(null, null, null);
}
```

Register with LoopBack server

Once you have all Installation properties set, you can register with the LoopBack server. The first run of the application should create a new Installation record, subsequent runs should update this existing record. The LoopBack Android SDK handles the details. Your code just needs to call `save()`.

```
void saveInstallation(final LocalInstallation installation) {
    installation.save(new Model.Callback() {
        @Override
        public void onSuccess() {
            // Installation was saved.
            // You can access the id assigned by the server via
            //   installation.getId();
        }
        @Override
        public void onError(final Throwable t) {
            Log.e(TAG, "Cannot save Installation", t);
        }
    });
}
```

Handle received notifications

Android apps handle incoming notifications in the standard way; LoopBack does not require any special changes. For more information, see the section "Receive a message" of Google's Implementing GCM Client guide.

Troubleshooting

When running your app in the Eclipse device emulator, you may encounter the following error:

```
Google Play services, which some of your applications rely on, is not supported by your device. Please contact the manufacturer for assistance.
```

To resolve this, install a compatible version of the Google APIs platform.

Tutorial: push notifications - iOS client

This is the third of a four-part tutorial on setting up a mobile backend as a service on Amazon and setting up iOS and Android client applications to enable push notification. See Tutorial: push notifications - LoopBack app for information on setting up the server application.

Overview

This article provides information on creating iOS apps that can get push notifications from a LoopBack application. See Push notifications for information on creating the corresponding LoopBack server application.

The basic steps to set up push notifications for iOS clients are:

1. Provision an application with Apple and configure it to enable push notifications.
2. Provide a hook to receive the device token when the application launches and register it with the LoopBack server using the `LBInstallation` class.
3. Provide code to receive notifications, under three different application modes: foreground, background, and offline.
4. Process notifications.

For general information on the Apple push notifications, see Apple iOS Local and Push Notification Programming Guide. For additional useful information, see Delivering iOS Push Notifications with Node.js.

Prerequisites

Before you start developing your application make sure you've performed all the prerequisite steps outlined in this section.

- **Download the LoopBack iOS SDK**
- **Install Xcode**

Configure APN push settings in your server application

Set up messaging credentials for iOS apps

If you have not already done so, create your APNS certificates for iOS apps. After following the instructions, you will have APNS certificates on your system. Then edit `config.js` in your backend application and look for these lines:

```
exports.apnsCertData = readCredentialsFile('apns_cert_dev.pem');
exports.apnsKeyData = readCredentialsFile('apns_key_dev.pem');
```

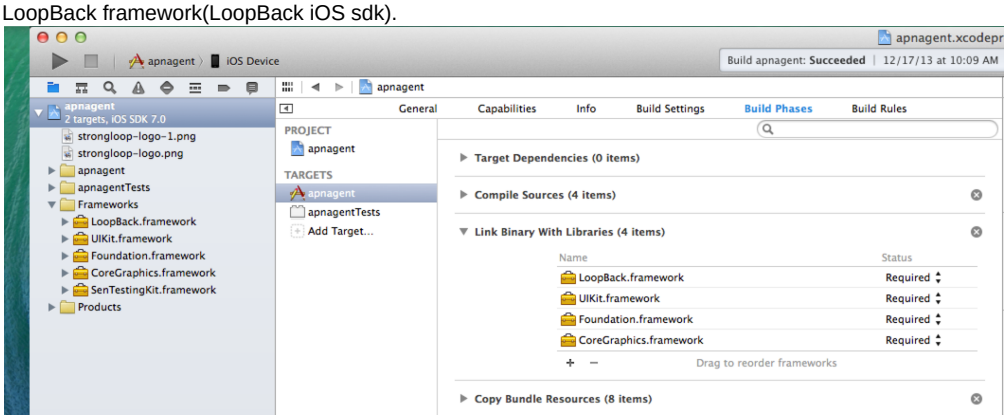
Replace the file names with the names of the files containing your APNS certificates. By default, `readCredentialsFile()` looks in the `/credentials` sub-directory for your APNS certificates. You can create a credentials directory and copy your pem files into that directory.

> **To try a sample client application, follow Install and run LoopBack Push Notification app.**
>
> **To enable push notifications for your own Android app see Add LoopBack iOS SDK as a framework to your own iOS Project.**

Install and run LoopBack Push Notification app

If you want to use the sample iOS client app, download the Push Notification Example iOS app . Then follow these steps to run the app

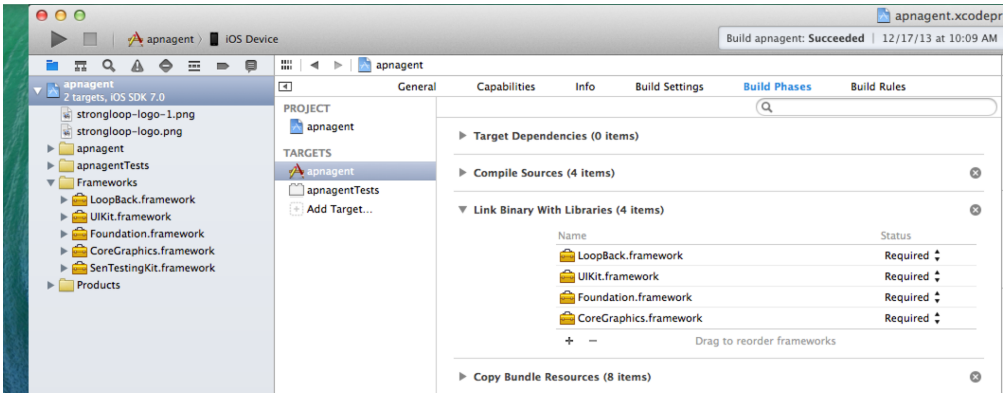
1. Open the apnagent.xcodeproj in Xcode, select targets, under build phases unfold **Link Binary with Libraries**, and click on '+' to add LoopBack framework(LoopBack iOS sdk).



2. Edit Settings.plist and update the RootPath to your instance ip. In my case it is - http://ec2-54-184-36-164.us-west-2.compute.amazonaws.com:3000/api

Add LoopBack iOS SDK as a framework to your own iOS Project

Open your XCode project, select targets, under build phases unfold **Link Binary with Libraries**, and click on '+' to add LoopBack framework.



The LoopBack iOS SDK provides two classes to simplify push notification programming:

- LBInstallation - enables the iOS application to register mobile devices with LoopBack.
- LBPushNotification - provides a set of helper methods to handle common tasks for push notifications.

Initialize LBRESTAdapter

The following code instantiates the shared `LBRESTAdapter`. In most circumstances, you do this only once; putting the reference in a singleton is recommended for the sake of simplicity. However, some applications will need to talk to more than one server; in this case, create as many adapters as you need.

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.settings = [self loadSettings];
    self.adapter = [LBRESTAdapter adapterWithURL:[NSURL
URLWithString:self.settings[@"RootPath"]]];

    // Reference to Push notifs List VC
    self.pnListVC = (NotificationListVC *)[[(UINavigationController
*)self.window.rootViewController viewControllers]
                                           objectAtIndex:0];

    LBPushNotification* notification = [LBPushNotification application:application
                                         didFinishLaunchingWithOptions:launchOptions];

    // Handle APN on Terminated state, app launched because of APN
    if (notification) {
        NSLog(@"Payload from notification: %@", notification.userInfo);
        [self.pnListVC addPushNotification:notification];
    }

    return YES;
}
```

Register the device

```
- (void)application:(UIApplication*)application
didRegisterForRemoteNotificationsWithDeviceToken:(NSData*)deviceToken
{
    __unsafe_unretained typeof(self) weakSelf = self;

    // Register the device token with the LoopBack push notification service
    [LBPushNotification application:application
didRegisterForRemoteNotificationsWithDeviceToken:deviceToken
                            adapter:self.adapter
                             userId:@"anonymous"
                      subscriptions:@[@"all"]
                            success:^(id model) {
                                LBInstallation *device = (LBInstallation *)model;
                                weakSelf.registrationId = device._id;
                            }
                            failure:^(NSError *err) {
                                NSLog(@"Failed to register device, error: %@", err);
                            }
     ];
...
}

- (void)application:(UIApplication*)application
didFailToRegisterForRemoteNotificationsWithError:(NSError*)error {
    // Handle errors if it fails to receive the device token
        [LBPushNotification application:application
didFailToRegisterForRemoteNotificationsWithError:error];
}
```

Handle received notifications

```
- (void)application:(UIApplication *)application
didReceiveRemoteNotification:(NSDictionary *)userInfo {
    // Receive push notifications
    LBPushNotification* notification = [LBPushNotification application:application
                                          didReceiveRemoteNotification:userInfo];
    [self.pnListVC addPushNotification:notification];
}
```

Tutorial: push notifications - putting it all together

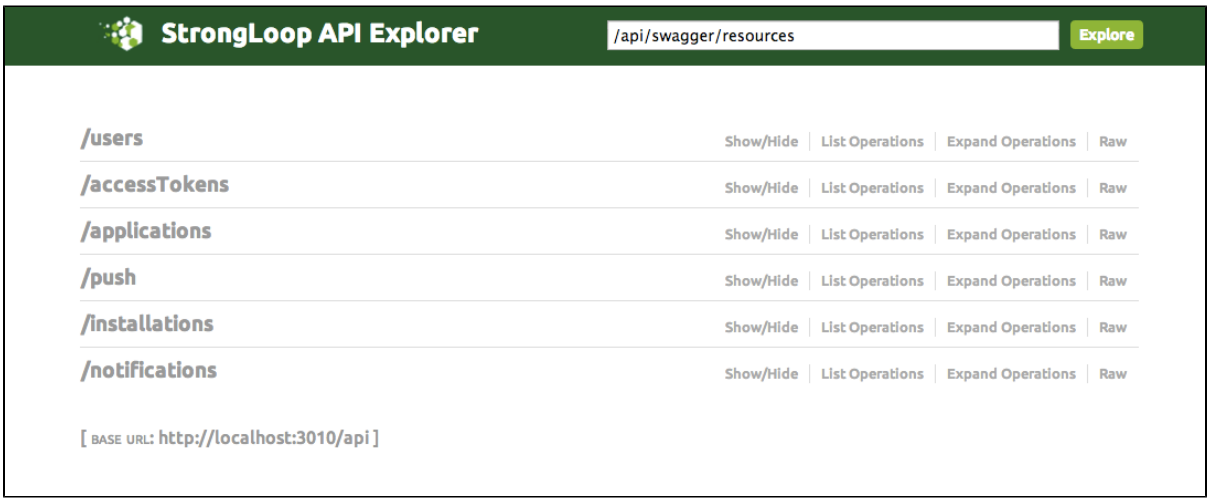
This is the final part of a four-part tutorial on setting up a mobile backend as a service on Amazon and setting up iOS and Android client applications to enable push notifications.

Run the backend sever

1. Make sure you have followed the steps in part 1 to create your backend application and you have setup the credentials and api keys for your iOS and android applications.
2. Run the application as node app.js. If the app runs successfully, you should see this -

```
[ec2-user@ip-10-252-201-3 push]$ node app.js
connect.multipart() will be removed in connect 3.0
visit https://github.com/senchalabs/connect/wiki/Connect-3.0 for alternatives
connect.limit() will be removed in connect 3.0
Browse your REST API at http://0.0.0.0:3000/explorer
LoopBack server listening @ http://0.0.0.0:3000/
Registering a new Application...
Application id: "loopback-push-notification-app"
```

3. You can access the swagger REST API explorer which comes with the loopback app at `http://<server_ip>:3000/explorer`:



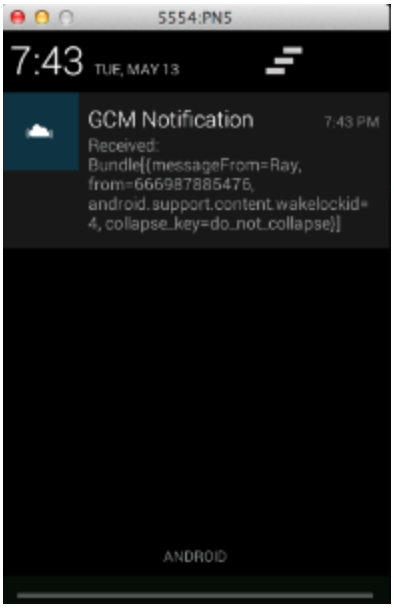
Receive push notifications for Android client application

1. Make sure you have set the rest adaptor in `src/com/google/android/gcm/demo/app/DemoApplication.java` to your server's IP address.
2. Click the green **Run** button in the toolbar to run the application. Run it as an Android application. You will be prompted to select the target on which to run the application. Select the AVD you created earlier in Part 2
3. Register your application with Loopback:
    a. Since you have set your gcmServerApiKey and appName in config.js, your application will be registered with loopback when you run the server.
    b. You can verify this using this GET request - /api/applications
    c. You can registered any application using this POST request - /api/applications
4. Register your device:
    a. If the AVD launches and the app gets installed successfully, the device will be registered with the backend and you should be able to verify the installation using GET request /api /installations.
    b. You should also be able to see the GCM application on your android emulator. You will need the "id" from the to send the push notification to the device. You can also get the id from the /api/installations

5. To send a push notification:

```
curl -X POST
http://ec2-54-184-36-164.us-west-2.compute.amazonaws.com:3000/notify/<installatio
n id>
OK
```

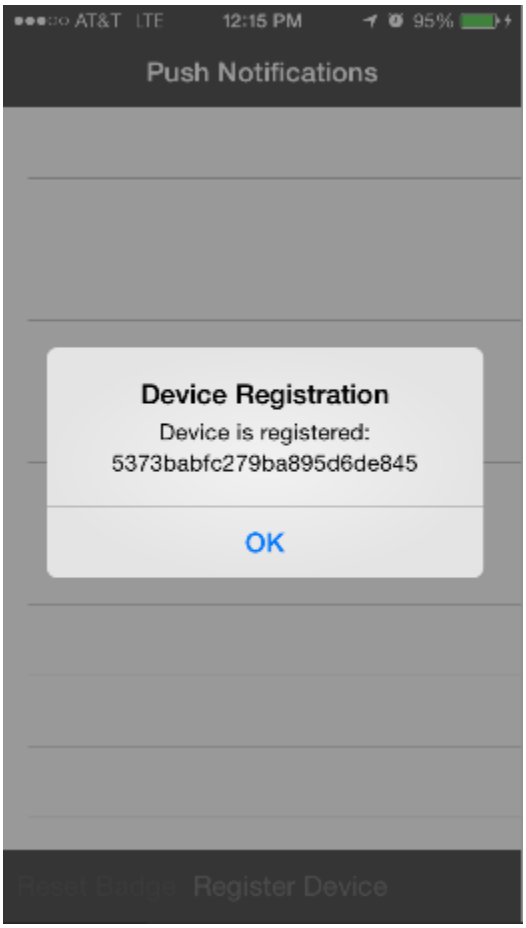
6. The notification received on your emulator should look like this:



Receive push notifications for iOS client application

1. Make sure you have set the RootPath in apnagent/Settings.plist set to your server's ip address.
2. Your phone should be connected to your laptop.
3. Register your application with loopback -
    a. Register your application using this POST request - /api/applications.
    b. You can verify that the app is registered using this GET request - /api/applications
    c. You need the "id" from the response to identify the client application. In your client app, edit Settings.plist and set AppId to the id from the previous step.
4. Run your XCode application and remember to connect your registered device. Select your device while running the app.
5. Register your device -

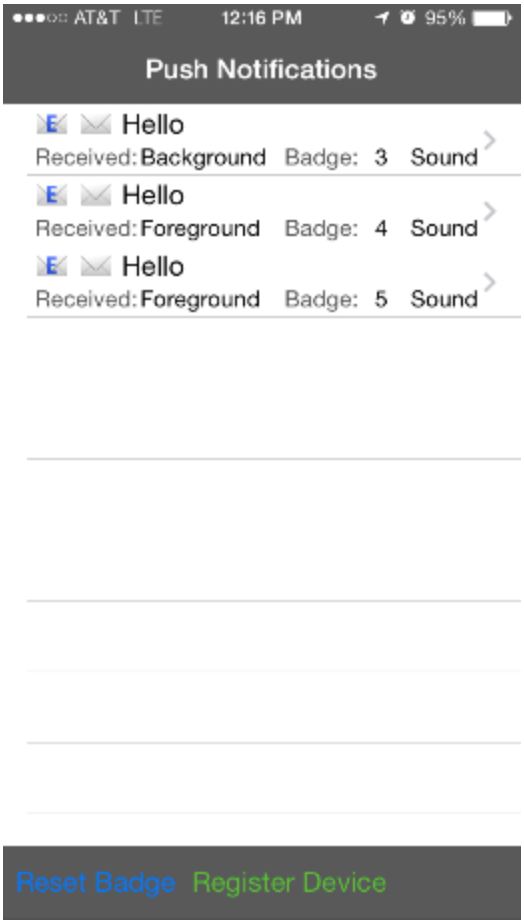
a. If the app runs successfully, the iOS device will be registered with the backend and you should be able to verify the installation using GET request /api /installations.
b. You should also be able to see the apnagent application on your phone. Click on Register Device and you should get an alert with the registration number.



6. To send a push notification using REST API:

```
curl -X POST
http://ec2-54-184-36-164.us-west-2.compute.amazonaws.com:3000/notify/<device_regi
stration>
OK
```

7. You should receive a notification on your device and it should look like this:



Storage service

Overview

LoopBack storage service makes it easy to upload and download files to cloud storage providers. It has Node.js and REST APIs for managing binary content in cloud providers, including:

- Amazon
- Rackspace
- Openstack
- Azure

You use the storage service like any other LoopBack datasource such as a database. Like other datasources, it supports create, read, update, and delete (CRUD) operations with exactly the same LoopBack and REST APIs.

Containers and files

The storage service organizes content as *containers* and *files*. A container holds a collection of files, and each file belongs to one container.

- **Container** groups files, similar to a directory or folder. A container defines the namespace for objects and is uniquely identified by its name, typically within a user account. NOTE: A container cannot have child containers.
- **File** stores the data, such as a document or image. A file is always in one (and only one) container. Within a container, each file has a unique name. Files in different containers can have the same name.

Creating a storage service data source

You create a storage service data source the same way you create any other data source, with the `loopback.createDataSource()` method.

For example, using local file system storage:

```
var ds = loopback.createDataSource({
    connector: require('loopback-component-storage'),
    provider: 'filesystem',
    root: path.join(__dirname, 'storage')
});

var container = ds.createModel('container');
```

Here's another example, this time for Amazon:

```
var ds = loopback.createDataSource({
  connector: require('loopback-component-storage'),
  provider: 'amazon',
  key: 'your amazon key',
  keyId: 'your amazon key id'
});
var container = ds.createModel('container');
app.model(container);
```

Provider credentials

Each cloud storage provider requires different credentials to authenticate. Provide these credentials as properties of the JSON object argument to `createDataSource()`, in addition to the `connector` property, as shown in the following table.

Provider	Property	Description	Example
Amazon	provider: 'amazon'		`{ provider: 'amazon', key: '...', keyId: '...' }`
	key	Amazon key	
	keyId	Amazon key ID	
Rackspace	provider: 'rackspace'		`{ provider: 'rackspace', username: '...', apiKey: '...' }`
	username	Your username	
	apiKey	Your API key	
Azure	provider: 'azure'		`{ provider: 'azure', storageAccount: "test-storage-account", storageAccessKey: "test-storage-access-key" }`
	storageAccount	Name of your storage account	
	storageAccessKey	Access key for storage account	
OpenStack	provider: 'openstack'		`{ provider: 'openstack', username: 'your-user-name', password: 'your-password', authUrl: 'https://your-identity-service' }`
	username	Your username	
	password	Your password	
	authUrl	Your identity service	
Local File System	provider: 'filesystem'		`{ provider: 'filesystem', root: '/tmp/storage' }`
	root	File path to storage root directory.	

API

Once you create a container, it will provide both a REST and Node API, as described in the following table. For details, see the complete API documentation.

Description	Container Model Method	REST URI
List all containers.	getContainers(cb)	GET /api/containers
Get information about specified container.	getContainer(container, cb)	GET /api/containers/:container
Create a new container.	createContainer(options, cb)	POST /api/containers
Delete specified container.	destroyContainer(container, cb)	DELETE /api/containers/:container
List all files within specified container.	getFiles(container, download, cb)	GET /api/containers/:container/files
Get information for specified file within specified container.	getFile(container, file, cb)	GET /api/containers/:container/files/:file
Delete a file within a given container by name.	removeFile(container, file, cb)	DELETE /api/containers/:container/files/:file
Upload one or more files into the specified container. The request body must use multipart/form-data which the file input type for HTML uses.	upload(req, res, cb)	POST /api/containers/:container/upload
Download a file within specified container.	download(container, file, res, cb)	GET /api/containers/:container/download/:file
Get a stream for uploading.	uploadStream(container, file, options, cb)	
Get a stream for downloading.	downloadStream(container, file, options, cb)	

Example

For an example of using the storage service, see the LoopBack Storage Service GitHub repository.

Follow these steps to run the example:

```
$ git clone http://github.com/strongloop/loopback-component-storage.git
$ cd loopback-component-storage
$ npm install
$ node example/app
```

Then load http://localhost:3000 in your browser.

Storage service API

Module: loopback-component-storage

- Class: StorageService
  - storageService.getContainers
  - storageService.createContainer
  - storageService.destroyContainer
  - storageService.getContainer
  - storageService.uploadStream
  - storageService.downloadStream
  - storageService.getFiles
  - storageService.getFile
  - storageService.removeFile
  - storageService.upload
  - storageService.download

storageService = new StorageService(options)

Storage service constructor. Properties of options object depend on the storage service provider.

Arguments

Name	Type	Description
options	`Object`	Options to create a provider; see below.

options

Name	Type	Description
provider	`String`	Storage service provider. Must be one of: • 'filesystem' - local file system. • 'amazon' • 'rackspace' • 'azure' • 'openstack' Other supported values depend on the provider. See the documentation for more information.

storageService.getContainers(callback)

List all storage service containers.

Arguments

Name	Type	Description
callback	`Function`	Callback function

callback

Name	Type	Description
err	`Object or String`	Error string or object
containers	`Object[]`	An array of container metadata objects

storageService.createContainer(options, cb)

Create a new storage service container.

Arguments

Name	Type	Description
options	`Object`	Options to create a container. Option properties depend on the provider.
cb	`Function`	Callback function

options

Name	Type	Description
name	`String`	Container name

cb

Name	Type	Description
err	`Object or String`	Error string or object
container	`Object`	Container metadata object

storageService.destroyContainer(container, callback)

Destroy an existing storage service container.

Arguments

Name	Type	Description
container	`String`	Container name.
callback	`Function`	Callback function.

callback

Name	Type	Description
err	`Object or String`	Error string or object

storageService.getContainer(container, callback)

Look up a container metadata object by name.

Arguments

Name	Type	Description
container	`String`	Container name.
callback	`Function`	Callback function.

callback

Name	Type	Description
err	`Object or String`	Error string or object
container	`Object`	Container metadata object

storageService.uploadStream(container, file, [options], Callback)

Get the stream for uploading

Arguments

Name	Type	Description
container	`String`	Container name
file	`String`	File name
[options]	`Object`	Options for uploading
Callback	`callback`	function

Callback

Name	Type	Description
err	`String or Object`	Error string or object

Returns

Name	Type	Description
result	`Stream`	Stream for uploading

storageService.downloadStream(container, file, options, callback)

Get the stream for downloading.

Arguments

Name	Type	Description

container	`String`	Container name.
file	`String`	File name.
options	`Object`	Options for downloading
callback	`Function`	Callback function

callback

Name	Type	Description
err	`String or Object`	Error string or object

Returns

Name	Type	Description
result	`Stream`	Stream for downloading

storageService.getFiles(container, [options], cb)

List all files within the given container.

Arguments

Name	Type	Description
container	`String`	Container name.
[options]	`Object`	Options for download
cb	`Function`	Callback function

cb

Name	Type	Description
err	`Object or String`	Error string or object
files	`Object[]`	An array of file metadata objects

storageService.getFile(container, file, cb)

Look up the metadata object for a file by name

Arguments

Name	Type	Description
container	`String`	Container name
file	`String`	File name
cb	`Function`	Callback function

cb

Name	Type	Description
err	`Object or String`	Error string or object
file	`Object`	File metadata object

storageService.removeFile(container, file, cb)

Remove an existing file

Arguments

Name	Type	Description

container	`String`	Container name
file	`String`	File name
cb	`Function`	Callback function

cb

Name	Type	Description
err	`Object or String`	Error string or object

storageService.upload(req, res, cb)

Upload middleware for the HTTP request/response

Arguments

Name	Type	Description
req	`Request`	Request object
res	`Response`	Response object
cb	`Function`	Callback function

storageService.download(container, file, res, cb)

Download middleware

Arguments

Name	Type	Description
container	`String`	Container name
file	`String`	File name
res	`Response`	HTTP response
cb	`Function`	Callback function

Storage service REST API

List containers

List all containers for the current storage provider.

```
GET /api/containers
```

Arguments

None.

Get container information

Get information about specified container.

```
GET /api/containers/container-name
```

Arguments

- *container-name* - name of container for which to return information.

Create container

Create a new container with the current storage provider.

```
POST /api/containers
```

Arguments

Container specification in POST body.

Delete container

Delete the specified container.

```
DELETE /api/containers/container-name
```

Arguments

- *container-name* - name of container to delete.

List files in container

List all files within a given container by name.

```
GET /api/containers/container-name/files
```

Arguments

- *container-name* - name of container for which to list files.

Get file information

Get information for a file within a given container by name

```
GET /api/containers/container-name/files/file-name
```

Arguments

- *container-name* - name of container.
- file-name - name of file for which to get information.

Delete file

Delete specified file within specified container.

```
DELETE /api/containers/container-name/files/file-name
```

Arguments

- *container-name* - name of container.

- file-name - name of file to delete.

Upload files

Upload one or more files into the given container by name. The request body should use multipart/form-data which the file input type for HTML uses.

```
POST /api/containers/container-name/upload
```

Arguments

- *container-name* - name of container to which to upload files.

Download file

Download specified file within specified container.

```
GET /api/containers/container-name/download/file-name
```

Arguments

- *container-name* - name of container from which to download file.
- file-name - name of file to download.

Third-party login

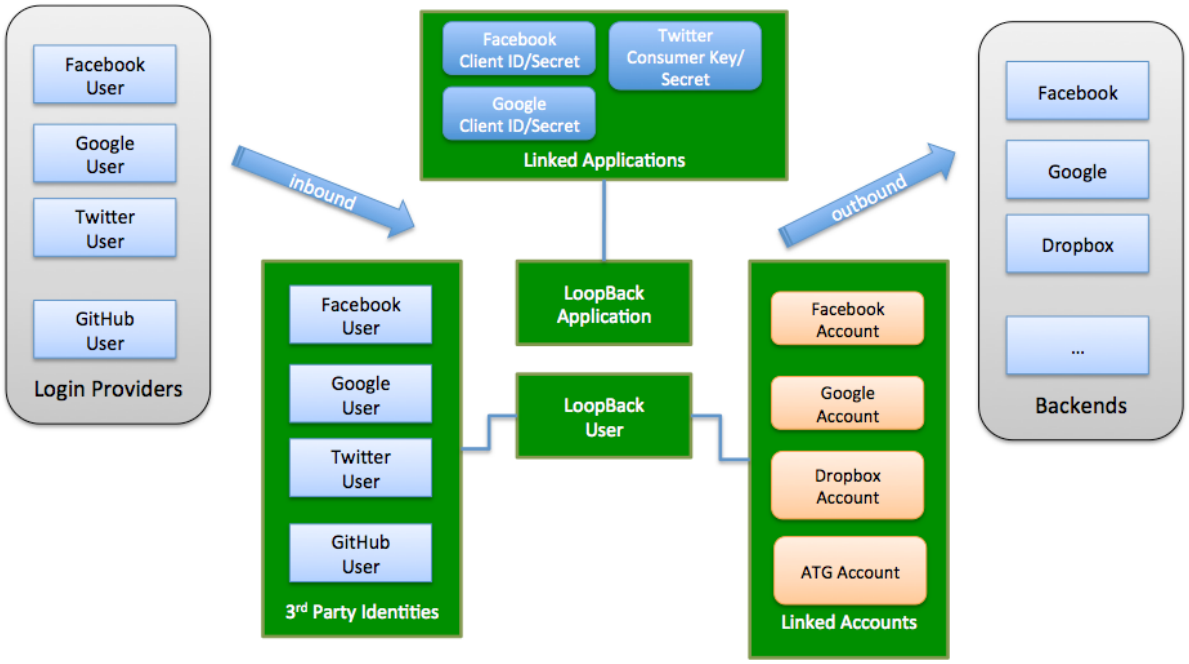
See also:

- Example application
- Passport documentation

Overview

The loopback-component-passport module integrates Passport and supports:

- **Third-party login**, so LoopBack apps can allow users to login using existing accounts on Facebook, Google, Twitter, or Github, and others.
- **Integration with enterprise security services** so that users can login with them instead of username and password-based authentication.
- **Linking an account** authorized through one of the above services with a LoopBack application user record. This enables you to manage and control your own user account records while still integrating with third-party services.



This module has the following key components:

- UserIdentity model - keeps track of third-party login profiles
- UserCredential model - stores credentials from a third-party provider to represent users' permissions and authorizations.
- ApplicationCredential model - stores credentials associated with a client application
- PassportConfigurator - the bridge between LoopBack and Passport

The example application demonstrates how to implement third-party login with a LoopBack application. The code is available at: https://github.com/strongloop-community/loopback-example-passport.

Models

UserIdentity model

The UserIdentity model keeps track of third-party login profiles. Each user identity is uniquely identified by provider and externalId. The UserIdentity model has a 'belongsTo' relation to the User model.

The following table describes the properties of the UserIdentity model.

Property	Type	Description
provider	String	Auth provider name; for example facebook, google, twitter, or linkedin.
authScheme	String	auth scheme, such as oAuth, oAuth 2.0, OpenID, OpenID Connect
externalId	String	Provider specific user ID.
profile	Object	User profile, see http://passportjs.org/guide/profile

credentials	Object	• User credentials
		  • oAuth: token, tokenSecret
		  • oAuth 2.0: accessToken, refreshToken
		  • OpenID: openId
		  • OpenID Connect: accessToken, refreshToken, profile

userId	Any	LoopBack user ID
created	Date	Created date
modified	Date	Last modified date

UserCredential model

UserCredential has the same set of properties as UserIdentity. It's used to store the credentials from a third party authentication/authorization provider to represent the permissions and authorizations of a user in the third-party system.

ApplicationCredential model

Interacting with third-party systems often requires client application level credentials. For example, you need oAuth 2.0 client ID and client secret to call Facebook APIs. Such credentials can be supplied from a configuration file to your server globally. But if your server accepts API requests from multiple client applications, each client application should have its own credentials. To support the multi-tenancy, this module provides the ApplicationCredential model to store credentials associated with a client application.

The ApplicationCredential model has a 'belongsTo' relation to the Application model.

The following table describes the properties of ApplicationCredential model

Property	Type	Description
provider	String	Auth provider name, such as facebook, google, twitter, linkedin
authScheme	String	Auth scheme, such as oAuth, oAuth 2.0, OpenID, OpenID Connect

credentials	Object	• Provider-specific credentials
		  • openId: {returnURL: String, realm: String}
		  • oAuth2: {clientID: String, clientSecret: String, callbackURL: String}
		  • oAuth: {consumerKey: String, consumerSecret: String, callbackURL: String}

created	Date	Created date
modified	Date	Last modified date

PassportConfigurator

PassportConfigurator is the bridge between LoopBack and Passport. It:

- Sets up models with LoopBack
- Initializes passport
- Creates Passport strategies from provider configurations
- Sets up routes for authorization and callback

Login and account linking

Third party login

The following steps use Facebook oAuth 2.0 login as an example. The basic procedure is:

1. A visitor requests to log in using Facebook by clicking on a link or button backed by LoopBack to initiate oAuth 2.0 authorization.
2. LoopBack redirects the browser to Facebook's authorization endpoint so the user can log into Facebook and grant permissions to LoopBack
3. Facebook redirects the browser to a callback URL hosted by LoopBack with the oAuth 2.0 authorization code
4. LoopBack makes a request to the Facebook token endpoint to get an access token using the authorization code
5. LoopBack uses the access token to retrieve the user's Facebook profile
6. LoopBack searches the UserIdentity model by (provider, externalId) to see there is an existing LoopBack user for the given Facebook id
7. If yes, set the LoopBack user to the current context
8. If not, create a LoopBack user from the profile and create a corresponding record in UserIdentity to track the 3rd party login. Set the newly created user to the current context.

Third-party account linking

The following steps use Facebook oAuth 2.0 login as an example. The basic procedure is:

1. The user log into LoopBack first directly or through third party login
2. The user clicks on a link or button by LoopBack to kick off oAuth 2.0 authorization code flow so that the user can grant permissions to LoopBack
3. Perform the same steps 2-5 as third party login
4. LoopBack searches the UserCredential model by (provider, externalId) to see there is an existing LoopBack user for the given Facebook id
5. Link the Facebook account to the current user by creating a record in the UserCredential model to store the Facebook credentials, such as access token
6. Now the LoopBack user wants to get a list of pictures from the linked Facebook account(s). LoopBack can look up the Facebook

credentials associated with the current user and use them to call Facebook APIs to retrieve the pictures.

Configure third-party providers

The following example illustrates using two providers: facebook-login for login with facebook and google-link for linking Google accounts with the current LoopBack user.

```
{
 "facebook-login": {
   "provider": "facebook",
   "module": "passport-facebook",
   "clientID": "{facebook-client-id-1}",
   "clientSecret": "{facebook-client-secret-1}",
   "callbackURL": "http://localhost:3000/auth/facebook/callback",
   "authPath": "/auth/facebook",
   "callbackPath": "/auth/facebook/callback",
   "successRedirect": "/auth/account",
   "scope": ["email"]
 },
 ...
 "google-link": {
   "provider": "google",
   "module": "passport-google-oauth",
   "strategy": "OAuth2Strategy",
   "clientID": "{google-client-id-2}",
   "clientSecret": "{google-client-secret-2}",
   "callbackURL": "http://localhost:3000/link/google/callback",
   "authPath": "/link/google",
   "callbackPath": "/link/google/callback",
   "successRedirect": "/link/account",
   "scope": ["email", "profile"],
   "link": true
 }
}
```


You must register with Facebook and Google to get your own client ID and client secret.

- Facebook: https://developers.facebook.com/apps
- Google: https://console.developers.google.com/project

Add the following code snippet to `app.js`:

```
var loopback = require('loopback');
var path = require('path');
var app = module.exports = loopback();
// Create an instance of PassportConfigurator with the app instance
var PassportConfigurator =
require('loopback-component-passport').PassportConfigurator;
var passportConfigurator = new PassportConfigurator(app);

app.boot(__dirname);
...
// Enable http session
app.use(loopback.session({ secret: 'keyboard cat' }));

// Load the provider configurations
var config = {};
try {
 config = require('./providers.json');
} catch(err) {
 console.error('Please configure your passport strategy in `providers.json`.');
 console.error('Copy `providers.json.template` to `providers.json` and replace the
clientID/clientSecret values with your own.');
 process.exit(1);
}
// Initialize passport
passportConfigurator.init();

// Set up related models
passportConfigurator.setupModels({
 userModel: app.models.user,
 userIdentityModel: app.models.userIdentity,
 userCredentialModel: app.models.userCredential
});
// Configure passport strategies for third party auth providers
for(var s in config) {
 var c = config[s];
 c.session = c.session !== false;
 passportConfigurator.configureProvider(s, c);
}
```

Synchronization

This feature is in beta. Documentation is still a work in progress.

Overview

In general, mobile applications need to be able to operate without constant network connectivity. This means the client app must synchronize data with the server application after a disconnected period. To do this:

- The client (browser) app replicates changes made in the server application.
- The server application replicates changes made in the client (browser) app.

This process is called *synchronization* (abbreviated as *sync*). Sync replicates data from the *source* to the *target*, and the target calls the LoopBack replication API.

The LoopBack replication API is a JavaScript API, and thus (currently, at least) works only with a JavaScript client.

Replication means intelligently copying data from one location to another. LoopBack copies data that has changed from source to target, but does not overwrite data that was modified on the target since the last replication. So, sync is just bi-directional replication.

In general there may be conflicts when performing replication. So, for example, while disconnected, a user may make changes on the client that conflict with changes made on the server; what happens when an object or field is modified both locally and remotely? LoopBack handles conflict resolution for you, and enables you to easily present a user interface to allow the end user to make informed decisions to resolve conflicts when they occur. See Resolving conflicts below.

LoopBack in the browser

LoopBack implements synchronization using the LoopBack browser API, that provides the same client JavaScript API as for Node. Thus, LoopBack in the browser is sometimes referred to as *isomorphic*, because you can call exactly the same APIs on client and server.

LoopBack in the browser uses Browserify to handle dependencies and gulp to generate the client API based on the back-end models and REST API.

Additional uses

In addition to basic client-server replication, LoopBack also supports replicating data:

- From a LoopBack server application to another LoopBack server application.
- From one database to another database.

Synchronization as described above to handle offline operation is called *offline sync* . LoopBack also provides the ability to consolidate (or "batch") data changes the user makes on the device and send them to the server in a single HTTP request. This is called *online sync*.

Resolving conflicts

Conflict resolution can be complex. Fortunately, LoopBack handles the complexity for you, and provides an API to resolve conflicts intelligently.

Terminology

In addition to standard terminology, conflict resolution uses a number of specific terms.

Change list

A list of the current and previous revisions of all models. Each data source has a unique change list.

Checkpoint

An order-able identifier for tracking the last time a source completed replication. Used for filtering the change list during replication.

Checkpoint list

An ordered list of replication checkpoints used by clients to filter out old changes.

Conflict

When replicating a change made to a source model, a conflict occurs when the source's previous revision differs from the target's current revision.

Rebasing

Conflicts can only be resolved by changing the revision they are based on. Once a source model is "rebased" on the current target version of a model, it is no longer a conflict and can be replicated normally.

Revision

A string that uniquely identifies the state of a model.

General process

A conflict occurs when a change to a model is not based on the previous revision of the model.

The callback of `Model.replicate()` takes `err` and `conflict[]`. Each `conflict` represents a change that was not replicated and must be manually resolved. You can fetch the current versions of the local and remote models by calling `conflict.models()`. You can manaully merge the conflict by modifying both models.

Calling `conflict.resolve()` will set the source change's previous revision to the current revision of the (conflicting) target change. Since the changes are no longer conflicting and appear as if the source change was based on the target, they will be replicated normally as part of the next `replicate()` call.

Sync methods

The LoopBack Model object provides a number of methods to support sync, mixed in via the DataModel object:

- **diff** - Get a set of deltas and conflicts since the given checkpoint.
- **changes** - Get the changes to a model since a given checkpoint. Provide a filter object to reduce the number of results returned.
- **checkpoint** - Create a checkpoint.
- **currentCheckpoint** - Get the current checkpoint ID.
- **replicate** - Replicate changes since the given checkpoint to the given target model.
- **createUpdates** - Create an update list for `Model.bulkUpdate()` from a delta list from `Change.diff()`.
- **bulkUpdate** - Apply an update list.
- **getChangeModel** - Get the `Change` model.
- **getSourceId** - Get the source identifier for this model / dataSource.
- **enableChangeTracking** - Start tracking changes made to the model.
- **handleChangeError** - Handle a change error. Override this method in a subclassing model to customize change error handling.
- **rectifyChange** - Tell LoopBack that a change to the model with the given ID has occurred.

Frequently asked questions

Does LoopBack support continuous replication?

Yes: with continuous replication, the client immediately triggers a replication when local data changes and the server pushes changes when then occur.

Here is a basic example that relies on a `socket.io` style `EventEmitter`.

```
// psuedo-server.js
MyModel.on('changed', function(obj) {
  socket.emit('changed');
});

// psuedo-client.js
socket.on('changed', function(obj) {
  LocalModel.replicate(RemoteModel);
});
```

How do you trigger immediate replication?

Call `Model.replicate()` to trigger immediate replication.

Sync example app

> This feature is in beta. Documentation is still a work in progress.

This example application is a simple "To Do" list that illustrates LoopBack synchronization using isomorphic LoopBack. It is based on the well-known TodoMVC example.

Prerequisites

You must install the following:

- Node.js. See Getting Started with LoopBack.

- Git
- MongoDB - if you want to be able to preserve state between application restarts.
- The application uses Grunt to generate the isomorphic client-side JavaScript API.
  If you have not already done so, install Grunt:

```
$ npm install grunt-cli -g
```

Run the application

Follow these steps to build and run the example application.

1. Clone the repository and change your current directory:

```
$ git clone https://github.com/strongloop/loopback-example-full-stack.git
$ cd loopback-example-full-stack
```

2. Install the root package dependencies:

```
$ npm install
```

3. Run Bower to install the client scripts:

```
$ bower install
```

4. **Optional**: If you want to run MongoDB, then:

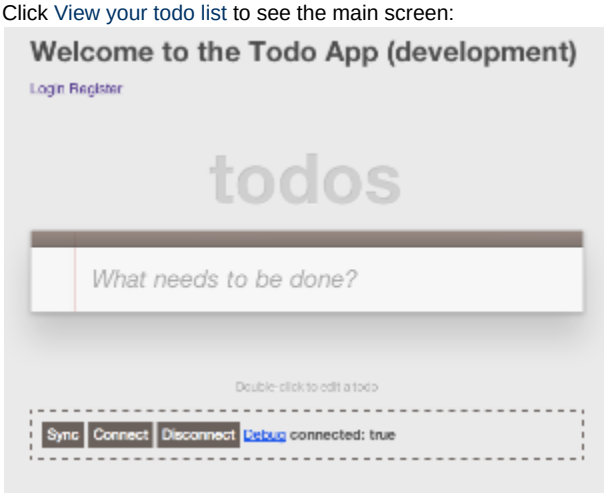
```
$ mongod
```

> ⚠ Make sure you have set the environment variable `NODE_ENV=production`.

5. Build and run the entire project in development mode

```
$ grunt serve
```

6. Open `http://localhost:3000` in your browser to view the running app.
7. Click View your todo list to see the main screen:



Try it out

First, add a couple of "To Do" items: click View your todo list then in the "What needs to be done" field enter a short text item and hit RETURN.

Then you can use the API Explorer to check out the LoopBack API: http://localhost:3000/explorer/.

Follow these steps to see the synchronization API in action.

1. Create a new todo item.
2. Find the ID of the todo: In the API explorer, go to http://localhost:3000/explorer/#!/Todos/Todo_findOne_get_5, then click **Try it out!** You'll see a JSON response that looks something like this:

```
{
  "id": "t-2562",
  "title": "Spackle the den",
  "completed": false,
  "created": 1404245971346
}
```

3. Update the To Do item: In the API explorer, go to http://localhost:3000/explorer/#!/Todos/Todo_upsert_put_1.
4. In the data field, enter JSON such as the following, but change the value of the id field to the value returned in step 2.

```
{"id": "t-2562", "title": "zarflag"}
```

   Where the id is the actual ID of the To Do item.
5. Make a change in the browser (against the stale local data / without syncing).
6. The conflict GUI pops up and you can choose how to resolve the conflict.

Advanced topics - sync

> This feature is in beta. Documentation is still a work in progress.

Replication algorithm

1. Assign an unique identifier to the source. Most of the time it will be a GUID.
2. Save this identifier on the target. This will track a source's replication checkpoint.
3. Get a subset of the source change list since the last known replication checkpoint.
4. Send the subset change list, containing model ids, current revisions, and previous revisions to the target.
5. The target will compare this change list with its own and send back a further subset change list containing revisions that differ and conflict.
6. The source then sends a set of bulk updates based on the revisions that differ. Conflicting changes must be resolved by the user before being sent.
7. Once the bulk update is complete, the source should save the new checkpoint, provided as the result of the bulk update.
8. The target adds a new checkpoint for the source identifier from the first step.

Conflict resolution

To resolve a conflict, a model must be rebased on the target's current revision. There are two basic rebase approaches that will resolve a conflict:

1. Fetch the target model's revision
   a. Merge the source and target models
   b. Rectify the local change list, effectively rebasing the source model on the current revision of the target model
   c. The source model may now be replicated
2. Forcibly rebase the source model on the current target model's revision

a. Rectify the local change list by modifying the previous version for the source model's entry
b. The source model may now be replicated

Change and revision semantics

Changes are determined using the following logic:

- **Created** - an entry in the change list without a previous revision
  - ...or if an entry does not exist and the given model **does exist**
- **Modified** - an entry with a current and previous revision
- **Deleted** - an entry with only a previous revision
  - ...or if an entry does not exist and the given model also **does not exist**

Existing data

The change list and replication can be used against existing data sources with large sets of data. This is possible using the revision semantics above.

Note: changes made without the loopback API will be treated as "created" or "deleted" change list entries.

Revision tracking

Changes made to a data source that supports replication will be tracked as revisions in that data source's change list.

Change list / change model

A data source's change list is stored like any other model data. Each Model has its own "Change" model. The "Change" model may be attached to any data source. This allows you to store your change lists in any data source.

Basic performance optimizations

Filtering

During replication you may supply a filter. The less data you are trying to replicate, the faster and more memory efficient the replication will be.

Parallel replication

If you are manually implementing the replication algorithm, you should run as many steps in the replication process in parallel as possible. This is usually the "bulk update" step(s).

LoopBack in the client

LoopBack in the client is sometimes referred to as *isomorphic LoopBack*, because it provides the exact same API as the LoopBack server framework.

We first got LoopBack running in a browser using browserify. You could create models and use the memory adapter and have your LoopBack app fully running in the browser. To further extend on this, now you can seamlessly connect models using the Remote Connector to connect LoopBack to LoopBack. For example, browser to server or server to server.

We did this by enhancing strong-remoting to be able to accept and return ModelTypes in addition to the JSON and its JSON primitives. Because your models don’t need to know where they are being run other than in LoopBack, you can share the same API regardless of where your code is running, thus making the API isomorphic.

Models are referenced through one way - “local” versus “remote” as seen in our offline sync capabilities and example in loopback-example-full-stack. This is foundation for how we built the replication API for data synchronization between browser and server.

Running LoopBack in the browser

In the browser, the main application file must call the function exported by the `loopback-boot` module to setup the LoopBack application by executing the instructions contained in the browser bundle:

browser-app.js

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

var app = module.exports = loopback();
boot(app);
```

The app object created above can be accessed via `require('loopback-app')`, where `loopback-app` is the identifier used for the main app file in the Browserify build shown above.

Here is a simple example demonstrating the concept:

index.html

```
<script src="app.bundle.js"> </script>
<script>
  var app = require('loopback-app');
  var User = app.models.User;
  User.login({
    email: 'test@example.com',
    password: '12345'
  }, function(err, res) {
    if (err) {
      console.error('Login failed: ', err);
    } else {
      console.log('Logged in.');
    }
  });
</script>
```

Using Browserify

Use Browserify to create a LoopBack API in the client.

The build step loads all configuration files, merges values from additional config files like `app.local.js` and produces a set of instructions that can be used to boot the application.

These instructions must be included in the browser bundle together with all configuration scripts from `models/` and `boot/`.

Don't worry, you don't have to understand these details. Just call `boot.compileToBrowserify()`, and it will take care of everything for you.

Build file (Gruntfile.js, gulpfile.js)

```
var browserify = require('browserify');
var boot = require('loopback-boot');

var b = browserify({
  basedir: appDir,
});

// add the main application file
b.require('./browser-app.js', { expose: 'loopback-app' });

// add boot instructions
boot.compileToBrowserify(appDir, b);

// create the bundle
var out = fs.createWriteStream('browser-bundle.js');
b.bundle().pipe(out);
// handle out.on('error') and out.on('close')
```